Etablierung von OSI-Layer 2 Punkt-zu-Punkt-Verbindungen mittels Service-Discovery unter Android

Abschlussarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin

Fachbereich 4: Informatik, Kommunikation und Wirtschaft

Studiengang Angewandte Informatik

1. Prüfer: Prof. Dr.-Ing. Thomas Schwotzer

2. Prüfer: Prof. Dr. Alexander Huhn

Eingereicht von Willi Bölke [566754]

13.12.2022

Abstract

Das Ziel dieser Arbeit war es eine Softwarekomponente zu entwickeln, welche eine Service-Discovery für Bluetooth und Wi-Fi Direct Services in der Umgebung ermöglicht, um so weitere Instanzen einer Android-Anwendung oder andere, benötigte Dienste zu finden. Ebenso sollte eine darauf aufbauende Erweiterung entwickelt werden, um automatisiert P2P-Verbindungen zu diesen Services aufzubauen. Diese Komponente soll das bisher in *ASAPAndroid* verwendete Verfahren der Peer-Discovery ersetzen und ebenso für weitere Anwendungen nutzbar sein.

Inhalt

1 Einleitung ... 1
1.1 Hintergrund der Arbeit ... 1
1.2 Problemstellung und Zielsetzung ... 1
1.3 Aufbau der Arbeit ... 2
2 Grundlagen ... 3
2.1 Service ... 3
2.2 Service-Discovery ... 3
2.3 Client und Server ... 3
2.4 Servicebeschreibungen ... 3
2.5 Bluetooth SDP ... 4
2.5.1 Service Records ... 4
2.5.2 Service Attributes ... 5
2.5.3 Serviceklassen ... 5
2.5.4 Service-Discovery ... 5
2.6 Wi-Fi Direct Netzwerktopologie ... 6
2.7 Wi-Fi Direct Service-Discovery ... 7
2.7.1 Bonjour ... 7
2.7.2 Multicast DNS ... 7
2.7.3 DNS-Service-Discovery ... 7
2.7.4 Service-Discovery ... 9
2.8 Android Bluetooth API ... 9
2.8.1 Adapter und Discovery ... 9
2.8.2 Device-Discovery ... 9
2.8.3 Bluetooth Device ... 9
2.8.4 Service Advertisement ... 10
2.8.5 Service-Discovery ... 10
2.8.6 Einschränkungen ... 10
2.9 Android Wi-Fi P2P API ... 11

2.9.1 WifiP2pManager 11
2.9.2 WifiP2pDevice 11
2.9.3 Service Advertisement 11
2.9.4 Service-Discovery 12
2.9.5 Port und TXT-Record 13
2.10 UUID 13
2.10.1 Aufbau einer UUID 13
2.10.2 Versionen 14
2.10.3 Name-Based UUIDs 14
2.11 Singleton Pattern 14
2.12 Observer Pattern 14
3 Zielgruppe und Anforderungen 16
3.1 Zielgruppe 16
3.2 Anforderungen 16
3.2.1 Funktionale Anforderungen 16
3.2.2 Nichtfunktionale Anforderungen 18
4 Entwurf 19
4.1 Servicebeschreibungen 19
4.1.1 Ziele der Servicebeschreibung 19
4.1.2 Wege zum Austausch von Servicebeschreibungen unter Android 19
4.1.3 Lösungsansatz 20
4.2 Aufbau der Komponente 22
4.3 Entwurf der Service-Discovery Teilkomponente 23
4.4 Entwurf der Service-Connection Teilkomponente 24
5 Implementierung Service-Discovery Teilkomponente 26
5.1 ServiceDescription 26
5.2 ServiceDiscoveryEngine 28
5.3 Bluetooth Service-Discovery 28
5.3.1 Broadcast Receiver 28
5.3.2 Varianten der Bluetooth Service-Discovery 29

5.3.3 Optimierungen ... 32
5.3.4 Anbieten von Services ... 33
5.3.5 Little Endian UUIDs ... 34
5.4 Wi-Fi Direct Service-Discovery ... 35
5.4.1 Service-Discovery ... 35
5.4.2 Service Advertisement ... 36
5.4.3 WifiDirectServiceDiscoveryListener ... 36
5.4.4 Zuverlässigkeit ... 37
6 Implementierung der Teilkomponente zum Verbindungausbau ... 38
6.1 Wi-Fi Direct ... 38
6.1.1 Aufbau der Gruppe ... 38
6.1.2 WifiDirectPeer ... 38
6.1.3 WifiConnection ... 39
6.2 Bluetooth ... 39
6.2.1 BluetoothServiceConnectionEngine ... 39
6.2.2 BluetoothConnection ... 40
6.2.3 BluetoothConnectionManager ... 41
6.2.4 Bluetooth Connector Threads ... 41
6.2.5 Listener ... 42
7 Tests ... 42
7.1 Integration-Tests ... 43
7.2 Unit-Tests ... 45
7.3 Manuelle Tests ... 45
7.3.1 … für die Bluetooth Service-Discovery ... 45
7.3.2 … für die mDNS-Service-Discovery ... 46
8 Fazit ... 48
8.1 Zusammenfassung ... 48
8.2 Ergebnisbewertung ... 49
8.2.1 … für die Bluetooth Service-Discovery ... 49
8.2.2 …für die Wi-Fi Direct Service-Discovery ... 49

8.2.3 …für die Teilkomponente zum Aufbau von Verbindungen 50
8.2.4 …für die Servicebeschreibungen 51
8.2.5 Gerätespezifische Probleme 52
8.3 Ausblick 52
Abkürzungsverzeichnis I
A. Appendix II
A.1 Source Code II
A.2 Unit-Tests II
A.2.1 BluetoothConnectionEngine II
A.2.2 BluetoothServiceDiscoveryEngineVTwo II
A.2.3 BluetoothServiceDiscoveryEngineVOne III
A.2.3 WifiDirectServiceConnectionEngine III
A.2.4 WifiDirectServiceDiscoveryEngine III

Abbildungsverzeichnis

Bluetooth SDP Request Response [4, S. 1213] 4
Wi-Fi Direct Netzwerkarchitektur und Limitation [7, S. 2] 6
Vollständiger SRV Record nach [12] 8
Vollständiger PTR Record nach [12] 8
grober Entwurf der funktionsweise einer DiscoveryEngine (Eigene Darstellung) 23
Interaktion zwischen einer Connection Engine und einer Discovery Engine 25
Vererbungsstruktur der Service-Discovery Engins 27
Ablauf einer Service-Discovery (Variante 2) 30
Ablauf einer Service-Discovery (Variante 2) 31
Benachrichtigen über bereits gefundene Services 33
Ablauf einer Service-Discovery mit der WifiDirectServiceDiscoveryEngine 35
Anbieten eines Services und akzeptieren von Client-Verbindungen. 39
Ablauf einer Bluetooth Service-Discovery auf Clientseite 40
Integrationstests für die BluetoothServiceDiscovery 44
Integrationstests für die WifiDirectServiceDiscoveryEngine 44
Integrationstests für die BluetoothServiceConnectionEngine 44

Tabellenverzeichnis

Tabelle 1 Beispiel für Service Attributes [5, S. 1216] 5
Tabelle 2 Aufbau einer UUID [27, S. 7] 13
Tabelle 3 UUID-Versionen nach RFC 4122 14
Tabelle 4 Manuelle Tests der Bluetooth Service-Discovery (Variante 1) 46
Tabelle 5 Manuelle Tests der Bluetooth Service-Discovery (Variante 2) 46
Tabelle 6 Manuelle Tests des Bluetooth Verbindungsaufbaus 46
Tabelle 7 Manuelle Tests der Wi-Fi Direct Service-Discovery 46
Tabelle 8 Manuelle Tests des Wi-Fi Direct-Verbindungsaufbaus 47

Codeverzeichnis

Code 1 Initialisierung einer ServiceDescription 26
Code 2 Registrieren eine mDNS-Service mit ServiceDescription 36
Code 3 WifiDirectPeer Interface 39
Code 4 Offenen eines BluetoothServerSocket mit einer ServiceDescription 41
Code 5 Das BluetoothServiceClient Interface 42
Code 6 Das BluetoothServiceServer Interface 42

1 Einleitung

1.1 Hintergrund der Arbeit

In der vorliegenden Arbeit wird eine Komponente entwickelt, welche eine Service-Discovery für Bluetooth und Wi-Fi Direct für Android-Anwendungen ermöglicht.

Der initiale Anlass für diese Arbeit ist ASAPAndroid. ASAP (Asynchronous Semantic Ad-hoc Protocol) ist ein P2P-Routing-Protokoll für mobile Anwendungen. ASAP wird im Schwerpunkt Mobile-Anwendungen des Studiengangs Angewandte Informatik an der HTW Berlin unter der Leitung von Prof. Dr.-Ing Thomas Schwotzer entwickelt. Bei *ASAPAndroid* handelt es sich um den Android-Plattform-Support für *ASAPJava*, der Java Implementierung von ASAP [1].

Derzeit stellt ASAPAndroid Verbindungen zu anderen *ASAPAndroid*-Anwendungen her, welche durch eine Peer- beziehungsweise Device-Discovery gefunden werden. Da keine Informationen über die auf diesen Peers vorhandenen Anwendungen / Diensten verfügbar sind muss versucht werden, zu jedem der gefundenen Geräte eine P2P-Verbindung aufzubauen. Eine Service-Discovery ermöglicht es vor dem Aufbau einer Verbindung die auf Peers verfügbaren Services zu finden. Es würde somit den Prozess für *ASAPAndroid* dahingehend verbessern, dass nicht mehr zu jedem gefundenen Gerät ein Verbindungsversuch unternommen werden muss.

1.2 Problemstellung und Zielsetzung

Derzeit werden weitere Instanzen von ASAP-Anwendungen ausschließlich mithilfe einer Peer- / Device Discovery gefunden. Dies führt dazu, dass für jedes gefundene Gerät versucht werden muss eine P2P-Verbindung aufzubauen. Ein Verbindungsversuch wird dann von Peers, die an einer Verbindung interessiert sind, akzeptiert. Bei anderen Geräten wird in der Regel kein Verbindungsaufbau akzeptiert beziehungsweise, sollte dieser dennoch akzeptiert werden, kann es nicht zu einem sinnvollen Austausch von Informationen kommen. Es werden somit viele Versuche zum Verbindungsaufbau unternommen, die eigentlich nicht notwendig sind.

Zur Bearbeitung dieses Problems soll in dieser Arbeit eine Softwarekomponente entwickelt werden, die es ermöglicht, eine Service-Discovery für Bluetooth und Wi-Fi Direct durchzuführen. Diese soll weitere Instanzen von Android-Anwendungen in der Umgebung finden. Es ist auch erforderlich, eine geeignete Methode zum Beschreiben von Services zu finden, die kompatibel mit den gegebenen Protokollen/Techniken (Wi-Fi Direct und Bluetooth) ist. Diese Methode soll es erlauben, Services voneinander zu unterscheiden und, wenn möglich, weitere Informationen über den betreffenden Service zu transportieren. Darüber hinaus soll die Komponente prototypisch um Funktionalitäten zum Verbindungsaufbau zwischen Clients und Servern erweitert werden. Diese Erweiterungen werden nicht in *ASAPAndroid* integriert, da eine entsprechende Funktion dort bereits vorhanden ist.

Die Komponente soll im Rahmen dieser Arbeit alleinstehend entwickelt werden und erst zu einem späteren Zeitpunkt in *ASAPAndroid* integriert werden. Auch soll sie für weitere Android-Anwendungen nutzbar sein.

1.3 Aufbau der Arbeit

In Kapitel zwei werden zunächst die theoretischen Grundlagen dieser Arbeit dargelegt. Dabei wird vor allem auf Bluetooth und WiFi-Direct sowie auf die verwendeten Protokolle zur Service-Discovery, also Bluetooth-SDP und Bonjour/Multicast DNS-Service-Discovery, eingegangen. Auch die von Android bereitgestellten APIs für beide Protokolle werden hier erläutert.

Im dritten Kapitel wird eine Zielgruppen- und Anforderungsanalyse durchgeführt. Im vierten Kapitel wird der Entwurf der Komponente vorgestellt und ein Überblick über die Architektur der Software gegeben.

Auf dieser Basis werden in Kapitel fünf und sechs die konkreten Implementierungen, zunächst der Service-Discovery und dann der Erweiterung dieser zum Verbindungsaufbau, ausführlich beschrieben.

Kapitel sieben befasst sich mit den automatisierten und manuell durchgeführten Testreihen. Abschließend fasst Kapitel acht die Arbeit und ihre Ergebnisse zusammen und bietet ein Fazit. Basierend auf den Tests wird das Ergebnis der Arbeit bewertet und es wird auf bestehende Probleme eingegangen. Ein Ausblick auf die zukünftigen Möglichkeiten zur Verbesserung und Weiterentwicklung der Komponente wird ebenfalls gegeben.

2 Grundlagen

2.1 Service

Im Kontext von verteilten Systemen wird ein Service als eine Software, Hardware oder eine Kombination aus beidem verstanden, die anderen, entfernten Systemen eine Ressource bereitstellt [2, S. 1]. Eine Ressource kann hierbei eine Information, eine Dienstleistung oder eine anderweitige Operation sein, die von Clients benötigt wird. Dies kann beispielsweise ein Netzwerkdrucker, Speicher oder auch andere Anwendungen oder Instanzen derselben Anwendung sein.

Im Rahmen dieser Arbeit wird ein Service als eine weitere Instanz einer Anwendung betrachtet, mit der eventuell eine P2P-Verbindung aufgebaut werden soll, um einen Datenaustausch zu ermöglichen.

2.2 Service-Discovery

Als Service-Discovery wird der Prozess bezeichnet, welcher automatisiert Serviceanbieter für einen benötigten Service innerhalb eines Netzwerkes finden kann [3, S. 1]. Ein Service-Discovery-Protokoll muss somit mindestens aus zwei Teilnehmern bestehen: Dem Server, welcher einen (oder mehrere) Services bereitstellt und dem Client, welcher an einem oder mehreren dieser Services interessiert ist [3, S. 1]. Service-Discovery-Protokolle, welche ausschließlich diese beiden Entitäten vorsehen werden als *verzeichnislose Service-Discovery-Protokolle* bezeichnet.
Ist eine weitere Entität vorgesehen, welche es erlaubt Services zu registrieren und zu vermitteln (ein Verzeichnis), so spricht man von *verzeichnisorientierten Service-Discovery-Protokollen.*
Bei Service-Discovery-Protokollen für P2P Ad-Hoc Netzwerken handelt es sich, ob der Dynamik des Netzwerkes und des Fehlens einer zentraler Registrierungsstelle, zumeist um *verzeichnislose Verfahren* [4].

2.3 Client und Server

Im Rahmen dieser Arbeit werden die Begriffe *Client* und *Server* verwendet, um die Rollen von Peers beim Anbieten und Suchen von Services und beim Aufbau einer P2P-Verbindung zu unterscheiden. Ein Client ist der Peer, der nach einem Service sucht. Ein Server hingegen ist der Peer, der einen bestimmten Service anbietet und Verbindungsanfragen anderer Peers akzeptiert.

2.4 Servicebeschreibungen

Service-Discovery-Protokolle basieren in der Regel auf der Verfügbarkeit von Servicebeschreibungen. Ein Service kann dann, basierend auf den Attributen, Charakteristiken und deren Werten gesucht bzw. identifiziert werden. Eines der verbreiteten Formate für Servicebeschreibungen besteht aus einer Liste an Key-Value Paaren [3, S. 5]. Ein solches Format wird beispielsweise in Protokollen wie Bluetooth

SDP (Service Records) und Service Location Protocol [5, S. 10–12] genutzt. DNS-SD benutzt DNS-Records zur Identifizierung und Beschreibung eines Service, wobei es sich bei dem TXT-Record ebenfalls um eine Liste an Key-Value Paaren handelt.

2.5 Bluetooth SDP

Die Bluetooth Special Interest Group (SIG) sieht in der Bluetooth-Spezifikation mit dem *Bluetooth Service-Discovery Protocol* (Bluetooth SDP) eine Möglichkeit zur Service-Discovery vor. Bluetooth SDP bietet Anwendungen die Möglichkeit, verfügbare Services zu finden und deren Eigenschaften (Attribute) auszulesen. In diesem Kapitel soll die Funktionsweise von Bluetooth SDP erläutert werden. Dabei wird sich speziell auf die Bluetooth-Spezifikation [6, S. 1206 ff] bezogen.

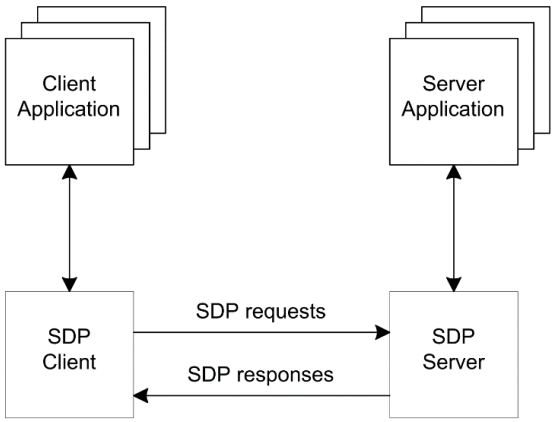

Abbildung 1 Bluetooth SDP Request Response [4, S. 1213]

Die grundlegende Arbeitsweise von *SDP* beseht in einem Request-Response Verfahren (siehe Abbildung 1), bei dem der Client den Server Informationen über verfügbare Services anfragt. Für diesen Vorgang muss der Client um die Existenz des Servers wissen, eine vorangegangen *Device-Discovery* ist also notwendig. Bluetooth SDP beschreibt eine Reihe von Datenstrukturen, um Services zu beschreiben und Informationen über diese auszutauschen.

2.5.1 Service Records

Ein *Service Record* ist eine Liste von *Attributen*, die genau einen Service beschreiben. Jeder SDP-Server hält eine Liste von Service Records für die von ihm angebotenen Services. Auf einem Server wird ein Service Record eindeutig durch seine 32-Bit-Nummer, die *Service Record Handle*, identifiziert. Der Inhalt eines Service Record ist eine Liste von *Service Attributes*. [6, S. 1214–1216]

2.5.2 Service Attributes

Service Attributes sind Key-Value Paare, welche jeweils genau eine Charakteristik eines Services Beschreiben. Der Key wird als Attribute-ID bezeichnet und besteht aus einem 16-bit unsigned Integer, dessen Bedeutung durch die Service-Klasse bestimmt wird. Der Attribut-Wert (Attribute-Value) verfügt über eine variable Länge. Die Bedeutung des Werts ist von seiner Attribute-ID abhängig [6, S. 1216–1217]. Eine beispielhafte Übersicht von Service-Attributen findet sich in Tabelle 1.

ServiceClassIDList	Identifies the type of service represented by a service record. In other words, the list of classes of which the service is an instance
ServiceID	Uniquely identifies a specific instance of a service
ProtocolDescriptorList	The textual name of the individual or organization that provides a service
IconURL	Specifies a URL that refers to an icon image that may be used to represent a service
ServiceName	A text string containing a human-readable name for the service
ServiceDescription	A text string describing the service

Tabelle 1 Beispiel für Service Attributes [6, S. 1216]

2.5.3 Serviceklassen

Jeder Service ist eine Instanz einer *Service Klasse*. Die *Service Klasse* definiert die Bedeutung der Attribute-IDs und wie die Attribute-Values gelesen werden können. Eine Service Klasse kann eine Reihe von Service Attributen vorgeben, welche wiederum von Subklassen erweitert werden kann.
Eine Service Klasse hat darüber hinaus eine zugewiesenen UUID. [6, S. 1217]

2.5.4 Service-Discovery

Bluetooth SDP beschreibt zwei Methoden zur Discovery von Services. Eine *Service Search* und ein *Service Browsing*. Das Finden eines Services führt zum Erhalt der *Service Record Handle*, mit der weitere Service-Attribute vom Client angefragt werden können.

Die *Service Search* ermöglicht es, Services mittels eines *Service Search Patterns* zu suchen. Dabei kann nach einer Reihe von Service-Attributen gesucht werden. Es ist jedoch zu beachten, dass die Suche nach beliebigen Attributen nicht möglich ist, sondern nur nach solchen, die einen UUID-Wert haben. Das bedeutet, dass ein *Service Search Pattern* eine Liste von UUIDs sein muss. Um die Suche erfolgreich durchzuführen, muss das *Service Search Pattern* ein Subset der UUIDs in einem *Service Record* sein. [6, S. 1218–1219]

Das *Service Browsing* ermöglicht es, Services zu finden, ohne deren Charakteristika zu kennen. Für das *Service Browsing* haben alle Services ein Attribut, die BrowseGroupList. Die *BrowseGroupList* enthält eine Liste von weiteren UUIDs, eine für jede *Browse Group*, der ein Service angehört. *Browse Groups* können hierarchisch in einer Baumstruktur angeordnet werden, wobei der Ursprung des Baums die *Root Browse Group* ist. Wenn nur wenige Services auf einem lokalen SDP-Server vorhanden sind, befinden

sie sich normalerweise direkt in der *Root Browse Group*. Wenn ein Client ein *Service Browsing* durchführen möchte, startet er mit einem *Service Search Pattern*, das die UUID der *Root Browse Group* enthält. [6, S. 1219–1220]

2.6 Wi-Fi Direct Netzwerktopologie

Wi-Fi Direct ist ein von er Wi-Fi Alliance entwickelter Standard zur direkten P2P-Verbindung von Wi-Fi fähigen Geräten. Wi-Fi Direct erlaubt es Wi-Fi Netzwerke zu formen, ohne dass ein festgelegter Access-Point benötigt wird. In einem (nicht Wi-Fi Direct) Wi-Fi Netzwerk, verbinden sich Wi-Fi Geräte in der Regel mit WLANs, welche von Access-Points angeboten werden. Die Rollenverteilung (AP, Client) ist hier eindeutig festgelegt. Wi-Fi Direct baut darauf auf. Anstelle der festgelegten Rollen sind diese hier aber dynamisch. Ein Wi-Fi Direct fähiges Gerät muss dafür sowohl die Rolle des AP als auch des Clients implementieren.

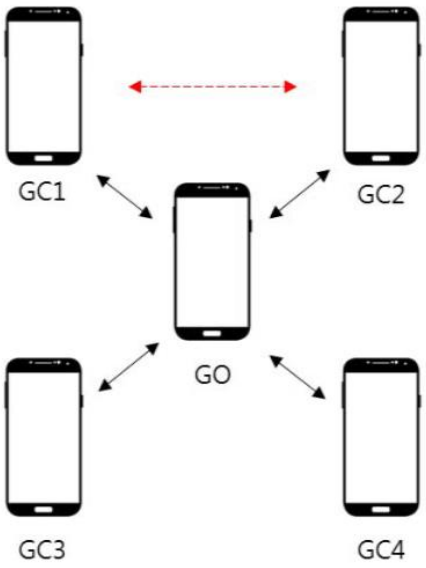

Abbildung 2 Wi-Fi Direct Netzwerkarchitektur und Limitation [7, S. 2]

Wi-Fi Direct-Geräte bilden eine Wi-Fi Direct Group, die dem Aufbau einer Wi-Fi-Infrastruktur ähnelt. Der Peer, der die Access Point-Funktionalität bereitstellt, wird in diesem Fall als *Soft AP* oder *Group Owner* (GO) bezeichnet [7, S. 2]. Eine Wi-Fi Direct Group besteht daher aus einem Group Owner und einer Anzahl von 1-n Clients.

Clients in einer Wi-Fi Direct Group können nicht mit anderen Wi-Fi Direct-Geräten außerhalb der Gruppe kommunizieren. Zudem können sie nicht gleichzeitig Group Owner einer anderen Gruppe sein und innerhalb der aktuellen Gruppe laufen alle Interaktionen über den Group Owner und nicht direkt zwischen den Clients [8, S. 2]. Dies ist in Abbildung 2 dargestellt.

2.7 Wi-Fi Direct Service-Discovery

Wi-Fi Direct spezifiziert kein eigenes Protokoll zur Service-Discovery, bietet aber die Voraussetzung für andere, Higher-Layer-Protokolle / Anwendungen eine solche durchzuführen. In dem Whitepaper *Wi-Fi Certified Wi-Fi Direct* werden dafür *UPnP*, *Bonjour* und Web Service-Discovery genannt [9, S. 8].

2.7.1 Bonjour

Bonjour ist Apples Implementierung einer Reihe von Technologien zum konfigurationslosen Aufbau von Netzwerken und beruht auf der Arbeit der Zeroconf Working Group der IETF. Bonjour erfüllt dabei die drei spezifizierten Kernanforderungen: Addressing, Name-Resolution und Service-Discovery. Dafür kommen *multicast DNS* und *DNS-Service-Discovery* (*DNS-SD*) zum Einsatz. [10]

2.7.2 Multicast DNS

Multicast DNS (mDNS) beschreibt die Möglichkeit DNS-Ressource-Records ohne das Vorhandensein eines konventionellen DNS-Servers zu ermitteln. DNS ähnliche Queries werden dafür auf dem local-link und im Multicast versendet [11, S. 1]. Die empfangenden Geräte agieren, wenn sie eine DNS-Query für den eigenen Namen empfangen, selbständig als DNS-Provider [10]. mDNS reserviert einen Namespace für local-link Hostnamen unter der TDL „.local.“.

2.7.3 DNS-Service-Discovery

DNS-SD [12] ist eine Erweiterung für DNS und Multicast DNS. In dieser Arbeit wird im Zusammenhang mit Wi-Fi Direct ausschließlich von einer Anwendung mit Multicast DNS bzw. Bonjour ausgegangen. Daher bezieht sich der folgende Text ausschließlich auf die Verwendung von DNS-SD im Rahmen von mDNS / Bonjour im Local-Link.

2.7.3.1 Service Publikation

DNS-SD sieht eine Servicebeschreibung durch drei DNS Records vor: Den SRV Record, den TXT Record und den PTR Record. Alle drei Records werden beim Multicast DNS-Responder registriert, um eine Service-Instanz anzubieten [13].

Der SRV Record bildet eine Service-Instanz auf die zur Verbindung mit diesem Service notwendigen Informationen ab. Er enthält den Hostnamen in der Form *<Instance Name>.<Service Type>.<Domain>*, die Portnummer und die Adresse, unter der der Service erreichbar ist. Zusätzlich enthält er weitere Informationen gemäß RFC 2782 [14]. Der SRV-Record bietet die notwendigen Informationen, um eine Verbindung mit einem Service herzustellen.

```
Instance._Service._Proto.Name TTL Class SRV Priority Weight Port Target
```

Abbildung 3 Vollständiger SRV Record nach [13]

Der PTR Record enthält den Namen der Serviceinstanz entsprechend dem des SRV Record. Der Name des PTR Records selbst entspricht dem *Service Type* der Service Instanz welche er beschreibt. Ein DNS PTR verweist von einem DNS-Namen auf einen anderen. In diesem Fall vom *Service Type* auf die *Service Instance* eines Services. Ein DNS-SD PTR Lookup erlaubt es alle verfügbaren Instanzen eines Services zu finden. Dieser Vorgang wird auch *Service Instance Enumeration* genannt. [12, S. 6]

```
_Service._Proto.Name TTL PTR Instance._Service._Proto.Name
```

Abbildung 4 Vollständiger PTR Record nach [13]

Der TXT Record trägt denselben Namen wie der dazugehörige SRV Record, beinhaltet aber eine Anzahl zusätzlicher Informationen in der Form von Key–Value Paaren. Er ist dazu gedacht zusätzliche, aber für den Aufbau einer Verbindung nicht relevante, Informationen zu enthalten. [12, S. 12]

2.7.3.2 Domain Namen

Wie im vorangegangenen Abschnitt gezeigt folgt der Domainname dem Schema <*Instance Name*>.<*Service Type*>.<*Domain*>. Der *Domain*-Anteil wird, wie in 2.7.2 erwähnt für Multicast DNS auf *.local.* gesetzt [11, S. 4–5].

Der *Service Type* setzt sich entsprechend [14] aus *_protocol._transport* zusammen. *transport* meint das verwendete Host-to-Host Transportprotokoll. Hier ist anzumerken das ausschließlich *_tcp* oder *_udp* Anwendung finden. *_tcp* wird nur dann verwendet, wenn es sich um ein TCP-Transport handelt. Für alle anderen Protokolle (beispielsweise UDP, SCRP oder DCCP) wird *_udp* verwendet [12, S. 18]. *protocol* meint das Protokoll des angebotenen Dienstes, hier ist es üblich eines der bereits existierenden Protokolle zu nutzen (beispielsweise _http, _ldap) aber nicht verpflichtend. In einigen Fällen mag dies sogar abkömmlich sein, da ein Services sich zwar dasselbe Nachrichtenformat teilen mag, jedoch nicht notwendigerweise derselben Semantik folgt [12, S. 20]. Letztendlich dient dieser Teil der Identifizierung eines Services durch einen Namen. Er kann beispielsweise ein, den Service beschreibender, Name in englischer Sprache sein oder aber einer anderen Logik folgen. [12, S. 21]

Der *Instance Name* ist der Name der Service-Instanz, der beispielsweise vom Benutzer oder dem Gerät vergeben werden kann, um diesen Service von anderen Instanzen desselben Services zu unterscheiden.

Ein vollständiger Service-Hostname würde also beispielsweise „*Mein Gerät._asap._tcp.local*“ lauten. Domains unter der *.local* TLD können frei gewählt werden und unterliegen keiner zentralen Registrierung. [11, S. 5–7].

2.7.4 Service-Discovery

Die DNS-basierte Service-Discovery nutzt die im vorherigen Kapitel beschriebenen DNS-Records, um Services anzubieten, zu suchen und die für einen Verbindungsaufbau notwendigen Informationen zu erhalten. Dazu wird zunächst eine *Service Instance Enumeration* durchgeführt, um alle Instanzen eines *Service-Typs* zu finden. [12, S. 5–9].

Soll dann einer der gefundenen Services kontaktiert werden, respektive eine Verbindung zu diesem aufgebaut werden wollen, so wird für diese Instanz eine *Service Instance Resolution* durchgeführt, welche eine Abfrage für den SRV und den TXT Records dieser einen Service-Instanz bedeutet. Wobei der SRV Record dann Port und Adresse des Services und der TXT Record weitere Informationen bereitstellt. [12, S. 9]

2.8 Android Bluetooth API

In diesem Abschnitt wird ein Einblick in die Funktionsweise der Android Bluetooth API im Allgemeinen und im Besonderen auf die bereitgestellten Methoden zur Nutzung von Bluetooth SDP gegeben.

2.8.1 Adapter und Discovery

Eine zentrale Klasse der Android Bluetooth API ist die Klasse `BluetoothAdapter`. Sie repräsentiert den Bluetooth-Adapter (Hardware) des jeweiligen Geräts [15].

2.8.2 Device-Discovery

Eine *Device-Discovery* ist ein asynchroner Prozess, welcher nach Geräten in Reichweite scannt. Da Bluetooth SDP eine Service-Discovery im Unicast für einzelne Geräte durchführt, ist eine solche essentiell notwendig. Eine *Device-Discovery* kann durch `BluetoothAdapater.startDiscovery()` gestartet werden und dauert circa 12 Sekunden [15]. Geräte, welche in dieser Zeit nicht in Reichweite oder nicht *Discoverable* waren, werden nicht gefunden. Nachdem eine Discovery gestartet wurde benachrichtigt Android über gefundene Geräte mittels einem Broadcasts mit der Action `BluetoothDevice.ACTION_FOUND` [16].

2.8.3 Bluetooth Device

Die Klasse `BluetoothDevice` in Android repräsentiert Bluetooth fähige Geräte, genauer gesagt handelt es sich dabei um einen Wrapper der Bluetooth Mac-Adresse eines Gerätes [16]. Wird eine Device-Discovery durchgeführt, so werden die dabei gefundenen Peers in Form eines `BluetoothDevice` zur Verfügung gestellt.

2.8.4 Service Advertisement

Services können in Android mit `BluetoothAdapter.listenUsingRfcommWithServiceRecord()` respektive `BluetoothAdapter.listenUsingInsecureRfcommWithServiceRecord()` angeboten werden.
Für beide Methoden wird ein String und eine UUID als Parameter erwartet.

Es wird dadurch ein *Service Record* im lokalen SDP-Server registriert, welcher die gegebene UUID und den String (als Servicenamen) als Attribute enthält [17]. Es gibt an dieser Stelle keine weitere Möglichkeit, die in 2.5 beschriebenen Service-Attribute, Service-Klassen oder Records zu definieren.

2.8.5 Service-Discovery

Eine Service-Discovery wird durch ein Request-Response-Verfahren zwischen zwei Bluetooth Geräten im Unicast realisiert (siehe 2.5). Dabei werden *Service Records* ausgetauscht, welche Informationen zu den verfügbaren Services enthalten. In Android lässt sich dieser Prozess durch `BluetoothDevice.fetchUuidsWithSdp()` anstoßen.

Wie auch bei der Device-Discovery handelt es sich um einen asynchronen Prozess. Es lässt sich ein `BroadcastReceiver` für die Action `BluetoothDevice.ACTION_UUID` registrieren, um die gefundenen Service-UUIDs zu erhalten. Der Broadcast-Intent enthält die Extras `BluetoothDevice.EXTRA_DEVICE` und `BluetoothDevice.EXTRA_UUID` welche das betreffende `BluetoothDevice` respektive ein Array der verfügbaren Service UUIDs aus den *Service Records* enthalten.

Einmal erhaltene UUIDs werden für jedes `BluetoothDevice` gecached und können mit `BluetoothDevice.getUuids()` abgerufen werden [18]. Dieser Cache bleibt erhalten solange der Bluetooth Adapter aktiv[1] bleibt.

An dieser Stelle sei auch zu erwähnen das eine SDP-Request durch ein `fetchUuidsWithSdp()`, nicht (oder nur sehr unzuverlässig) funktioniert, wenn gleichzeitig eine Device-Discovery durchgeführt wird. `BluetoothDevice.EXTRA_UUID` wird dann unter Umständen eine leere Liste oder, sollten bereits UUIDs für das entsprechende Gerät bekannt sein, die gecachten UUIDs enthalten.

2.8.6 Einschränkungen

Wie aus den vorangegangenen Abschnitten ersichtlich wird, wird von Android nur eine sehr schmale API für Bluetooth SDP bereitgestellt. *Service Records*, so wie in der Bluetooth Spezifikation beschreiben (siehe 2.5.1), können nicht genutzt werden, sondern werden intern von Android verwaltet.

---

[1] Meint: Solange Bluetooth auf dem Gerät eingeschaltet bleibt

Es können genau zwei Service Attribute, ein Name und eine UUID, spezifiziert werden.
Auch die Service-Discovery wird von Android durchgeführt, eine *Service Search* mithilfe eines *Service Search Pattern* kann nicht durchgeführt werden. Die Vermutung liegt hierbei nahe das Android automatisch ein *Service Browsing* der *Root Browse Group* des Servers durchführt und alle dort vorhandenen UUIDs listet.

Interessanterweise beinhaltet die Klasse `BluetoothDevice` aber auch eine Methode zur Service Search. Diese ist zwar öffentlich aber mit `@hide` annotiert ist somit aus der Dokumentation sowie dem Compile-SDK entfernt. Sie ist jedoch Teil des Android-Frameworks und kann somit auch im Sourcecode gefunden werden [19]. Dasselbe gilt für die Broadcast Action `ACTION_SDP_RECORD` welche mit `@UnsupportedAppUsage` annotiert ist. Von einer Nutzung dieser soll in Rahmen dieser Arbeit jedoch abgesehen und ausschließlich die öffentlich zugänglichen APIs genutzt werden. Für die Weiterentwicklung der hier beschreiben Komponente könnte dies jedoch interessant sein.

2.9 Android Wi-Fi P2P API

Android stellt eine API zur Nutzung von Wi-Fi Direct bereit. Die APIs zur Service-Discovery unter Wi-Fi Direct befinden sich im Package `android.net.wifi.p2p.nsd` und umfassen die Protokolle *UPnP* und *Bonjour*. Im Folgenden soll näher auf einzelne Klassen und Methoden der API eingegangen werden, wobei der Fokus bei denen in dieser Arbeit verwendeten Bestandteilen liegen soll.

2.9.1 WifiP2pManager

Der `WifiP2pManager` stellt den zentralen Punkt der Wifi-Direct API dar. Zur Nutzung der API muss diese initialisiert werden und läuft fortan asynchron, bis sie durch das Schließen des `WifiP2pManager.Channel` beendet wird. Durch ihre Asynchronität bedingt werden Antworten auf fast alle Methodenaufrufe auf der API durch übergebene Listener beantwortet. Dazu wird hauptsächlich, aber nicht exklusiv, das `WifiP2pManager.ActionListener` Interface genutzt, welches zwei Callback Methoden `onSuccess`() und `onFailure(int reason)` definiert. [20]

2.9.2 WifiP2pDevice

Die Klasse `WifiP2pDevice` repräsentiert Wi-Fi Direct Geräte. Objekte dieser Klasse enthalten Informationen über ein Gerät wie dessen Adresse (MAC-Adresse der Wi-Fi Hardware), den Namen und den Typ des Geräts. [21]

2.9.3 Service Advertisement

Eine Anwendung kann Services durch `Wifip2pManager.addLocalService()` registrieren. Ein Service wird dabei durch ein `WifiP2pServiceInfo` Objekt beschreiben. Die Subklassen von `WifiP2pServiceInfo`,

`WifiP2pDnsSdServiceInfo` und `WifiP2pUpnpServiceInfo` erlauben es entweder einen Bonjour- oder UPnP-Service zu registrieren. [22]

In dieser Arbeit soll Bonjour verwendet werden, weshalb an dieser Stelle etwas näher auf `WifiP2pDnsSdServiceInfo` eingegangen wird. `WifiP2pDnsSdServiceInfo` wird mit den notwendigen Informationen für einen DNS-SD Service initialisiert. Dazu wird hier ein `instanceName` und ein `serviceType` als String sowie ein TXT-Record in Form einer `Map<String, String>`, gleichbedeutend der in 2.7.3.1 beschriebenen Werte benötigt. [23]

Durch `removeLocalService()` kann ein Service wieder beendet beziehungsweise entfernet werden. Alternativ steht `clearLocalServices(WifiP2pManager.Channel,WifiP2pManager.ActionListener)` zum Entfernen aller angebotenen Services zur Verfügung.

2.9.4 Service-Discovery

Analog zum Anbieten von Services können solche durch `WifiP2pManager.addServiceRequest()` in eine Suche aufgenommen werden. Android erlaubt es 3 verschiedene (Bonjour / mDNS) Service Request zu nutzen, welche durch die drei `newInstance()` Methoden der Klasse `WifiP2pDnsSdServiceRequest` [24] initialisiert werden können:

1. Die erste Variante erlaubt es alle verfügbaren Services zu suchen. Dabei werden keine weiteren Informationen benötigt und es wird für jeden der gefundenen Services eine *Service Instance Resolution* durchgeführt.
2. Die zweite Variante ermöglicht eine *Service Instance Enumeration*, also das Suchen nach allen Services mit einem bestimmten *Service Type*, durchzuführen.
3. Die dritte Variante erlaubt es den TXT Record einer bestimmten Service-Instanz anzufordern.

Um die Antworten auf diese Requests zu erhalten, lassen sich zwei Listener auf dem `WifiP2pMananger` registrieren:

Der `DnsSdServiceResponseListener` wird benachrichtigt, wenn eine DNS-Response zu einer der registrierten Service-Request erhalten wurde. Er erhält den *Instance Name*, den *Service Type* und eine Instanz der Klasse `WifiP2pDevice`. [25]

Der `DnsSdTxtRecordListener` kann optional registriert werden, um den TXT-Record eines Services in Form einer einer HashMap zu erhalten [26].

Die Service-Discovery kann durch `WiFiP2pManager.discoverServices()` gestartet werden [20]. Intern wird dadurch ebenfalls eine Wi-Fi Direct Peer-Discovery gestartet, welche Peers in Reichweite sucht, auf denen dann eine mDNS Service-Discovery durchgeführt wird.

2.9.5 Port und TXT-Record

Wie in 2.9.3 ersichtlich wurde lässt sich, entgegen der Bonjour-Dokumentation [13] und RFC 6763, kein Port für den SRV Record übergeben. Dies scheint eine bewusste Entscheidung zu sein, wie anhand der Android-Dokumentation zu erkennen ist, wo für den Port ein Feld im TXT Record genutzt wird [27]. Dies ist nach RFC 6763 ausdrücklich nicht die korrekte Vorgehensweise [12, S. 13].

2.10 UUID

Bei ***U**niversal **U**nique **ID**entifiers* handelt es sich um Labels, welche zur Identifizierung von Ressourcen dienen. UUIDs haben eine fixe Länge von 128 Bits [28]. Die Anzahl der dadurch Verfügung stehenden Kombinationen[2] ermöglicht es, auch ohne eine zentrale Vergabe von UUIDs davon ausgehen zu können, dass es sich um eine eindeutige (Universal Unique) Identifikation handelt.

Als String werden UUIDs üblicherweise in Hexadezimal und in 5, durch Bindestriche getrennte Blöcke unterteil dargestellt:

```
550e8400-e29b-11d4-a716-446655440000
```

Es gibt dabei verschiedenen Versionen[3] und Varianten von UUIDs welche, je nach Version und Variante, verschiedene Informationen beinhalten.

2.10.1 Aufbau einer UUID

Der Aufbau der UUID in verschieden Abschnitte, folgt also einem bestimmten, festgelegten Schema. Zu den in einer UUID codierten Informationen gehören die Version und Variante der UUID. Abhängig von Version und Variante der UUID werden eine Reihe von weiteren Informationen in der UUID codiert [28, S. 8]. Tabelle 2 zeigt den Schematischen Aufbau einer UUID.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td></tr>
<tr><td colspan="32">time_low</td></tr>
<tr><td colspan="16">time_mid</td><td colspan="16">time_hi_and_version</td></tr>
<tr><td colspan="8">clk_seq_hi_res</td><td colspan="8">clk_seq_low</td><td colspan="16">node (0-1)</td></tr>
<tr><td colspan="32">node (2-5)</td></tr>
</table>

Tabelle 2 Aufbau einer UUID [28, S. 7]

[2] 2^{128} = ~340 Sextillion
[3] Damit sind hier eher Typen gemeint und nicht das die Version „2“ neuer ist als Version „1“

2.10.2 Versionen

Version	Beschreibung
1	Zeitbasierte Version
2	DCE Security version, with embedded POSIX UIDs.
3	Name-Based (MD5)
4	Zufällig/ Pseudozufällig generiert
5	Name-Based (SHA-1)

Tabelle 3 UUID-Versionen nach RFC 4122

Es werden 5 UUID-Versionen unterschieden. Die Version einer UUID wird in die betreffende UUID codiert. Sie bestimmt auf welche Weise eine UUID gelesen wird. Die Versionsnummer befindet sich in den 4 Msb`s des *time_hi_and_version* Felds (siehe Tabelle 2), also an der ersten Stelle des 3. Blocks einer String-UUID.

2.10.3 Name-Based UUIDs

Im Rahmen dieser Arbeit wird im speziellen auf die Versionen 3 zurückgegriffen. Bei Version 3 und 5 handelt es sich um sogenannte *Name-Based UUIDs*. Diese werden nicht wie die anderen UUID-Versionen pseudozufällig oder auf Basis von schwer rekonstruierbaren Variablen generiert, sondern deterministisch, durch das Berechnen von Hashwerten. Wie Tabelle 3 zu entnehmen ist, kommt dafür entweder ein MD-5 oder ein SHA-1 Hashalgorithmus zum Einsatz.

2.11 Singleton Pattern

Das Singleton-Pattern beschränkt die Anzahl der Instanzen einer Klasse auf eine einzige. Da sich dieses Pattern mit der Erzeugung von Objekten befasst, fällt es in die Kategorie der *creational patterns*. Die Beschränkung der Anzahl von Instanzen ist dann sinnvoll, wenn es für eine bestimmte Aufgabe eine zentrale, über das ganze System erreichbare Anlaufstelle für die Ausführung von Operationen oder das Abrufen von Informationen geben soll. Die Instanz wird von der Klasse selbst erzeugt und verwaltet. Um weitere Instanziierungen an anderer Stelle zu verhindern, stellt die Klasse keinen öffentlichen Konstruktor zur Verfügung. Stattdessen wird eine Klassenmethode angeboten, die die Instanziierung der Klasse einmalig erlaubt. Die erste erzeugte Instanz wird als Klassenvariable festgehalten und bei späteren Aufrufen der Methode zurückgegeben. In Java ist es Konvention diese Methode `getInstance()` zu nennen. [29, S. 144 ff.]

2.12 Observer Pattern

Das Observer Pattern fällt in der Kategorie der *Behavioral Patterns*. Das Observer Pattern beschreibt eine 1-n Beziehung zwischen Objekten, sodass bei einer Zustandsveränderung eines Objekts, davon abgängige, andere Objekte benachrichtigt werden können. [30, S. 293]

Das Observer Pattern beschreibt zwei grundlegende Akteure:
Den *Observer*, welcher an der Zustandsveränderung eines anderen Objekts interessiert ist und das *Observable, w*elches über den sich verändernden Zustand verfügt.

Ein *Observable* ermöglicht es einem oder mehreren *Observern*, sich zu registrieren. Die registrierten *Observer* werden vom *Observable* gespeichert und können somit über zukünftige Zustandsänderungen informiert werden. Der Steuerungsfluss wird hierbei umgekehrt (Inversion of Control) und ein Polling kann vermieden werden. Um *Observer* zu verallgemeinern kann ein Interface bereitgestellt werden, welches von den Observern erfüllt werden muss. Dieses beschreibt eine oder mehrere Callback-Methoden, die bei Zustandsänderungen zur Benachrichtigung verwendet werden können.

Das Observer Pattern findet an vielen Stellen der Android APIs seine Anwendung. Wobei Observer hier oft als Listener bezeichnet werden. Beispielhaft ist das `WifiP2pManager.ActionListener` Interface zu nennen. Auch mit den `LiveData` und `MutableLiveData` Objekten wird ein Observer Pattern umgesetzt [31].

3 Zielgruppe und Anforderungen

3.1 Zielgruppe

Als Hauptzielgruppe dieser Arbeit sind Anwendungsentwickler zu nennen, welche entweder die hier entwickelte Komponente in ihre Anwendung integrieren wollen oder diese weiterentwickeln. Ebenso gilt dies für die Entwickler, welche entweder an *ASAPAndroid* arbeiten oder dies in ihrer Anwendung nutzen.

Nutzer von Anwendungen, die entweder *ASAPAndroid* oder die Service-Discovery-Komponente verwenden, sind keine direkte Zielgruppe, profitieren aber indirekt von der Geschwindigkeit und Zuverlässigkeit der Service-Discovery.

Auch die Demo-Anwendung erfüllt keinen Nutzen für Endnutzer, sondern richtet sich ebenfalls an Entwickler und dient Test- und Demonstrationszwecken.

3.2 Anforderungen

3.2.1 Funktionale Anforderungen

3.2.1.1 … an die Service-Discovery Komponente

1. Die Service-Discovery Komponente soll es ermöglichen Services zu suchen und anzubieten. Da eine Anwendung mehrere Services nutzen oder bereitstellen kann, soll es ermöglicht werden parallel nach mehreren Services zu suchen oder diese anzubieten. Möglichst sollte dabei nicht für jeden Service getrennt eine Discovery durchgeführt werden müssen.

2. In vielen Anwendungsfällen ist eine feste Rollenverteilung zwischen Services und Servicekonsumenten notwendig oder wünschenswert. Ein Client sucht nach Services und konsumiert diese, während ein Server diese bereitstellt. In Peer-to-Peer Netzwerken ist eine solche, fester Rollenverteilung nicht notwendig oder erwünscht. Die Komponente soll beiden Anwendungsfälle genügen. So soll es möglich sein:
   - Nur nach einem oder mehreren Services zu suchen
   - Nur einen oder mehrere Services anzubieten.
   - Sowohl Services anzubieten als auch, gleichzeitig, nach diesen zu Suchen.

3. Das letztendliche Ziel einer Service-Discovery ist in der Regel ein Verbindungsaufbau zwischen zwei Peers. Daraus ergibt sich, dass es das Ziel der hier implementierten Komponente sein muss, alle Informationen, welche für den Aufbau einer Verbindung zwischen Service und Client notwendig sind, bereitzustellen.

Sollten nach einer Reihe von n (> 1) Services zeitgleich gesucht werden, ist es beispielsweise nicht ausreichend ausschließlich die Geräte (Adresse, Port) zu nennen auf denen einer der Services gefunden wurde. Es muss auch, der auf diesen gefunden, Service bereitgestellt werden. In keinem Fall ist es ausreichen nur den gefundenen Service zu nennen ohne dabei das, den Service anbietende, Gerät, in einer für einen Verbindungsaufbau hinreichenden Weise ebenfalls zu nennen.

3.2.1.2 … an den Verbindungsaufbau

1. Sollten zwei Anwendungen denselben Service anbieten und suchen ist lediglich eine, bidirektionale Verbindung notwendig, um eine Kommunikation zwischen beiden Peers zu erlauben. Aus diesem Grund sollte eine Verbindung zwischen Server-Peer und Client-Peer eindeutig durch den Service und den verbundenen Peer bestimmt werden. Dabei soll kein Unterschied gemacht werden, ob die Verbindung als Client oder Server zustande gekommen ist. Eine zweite Verbindung zwischen den beiden Peers, bei der die Rollen jeweils vertauscht sind sollte ausgeschlossen werden. Eine Verbindung ist also durch das Paar {Peer, Service} eindeutig zu bestimmen.

2. Es soll möglich sein mehrere Verbindungen auf Basis mehrerer, voneinander verschiedener Services zwischen zwei Peers zu unterhalten.

3.2.1.3 … an die Servicebeschreibungen

1. Die Servicebeschreibungen sollen in der Lage sein einen Service eindeutig zu identifizieren und von anderen Services zu unterscheiden.

2. Eine oder mehrere Servicebeschreibungen können durch den Nutzer der Komponente definiert und verwendet werden. Dabei kann es sich sowohl um den Entwickler einer Anwendung als auch um den Nutzer der Anwendung handeln.

3. Es wurde bereits auf die von Android bereitgestellten APIs für Bonjour und Bluetooth SDP eingegangen (siehe 2.9 respektive 2.6). Eine Servicebeschreibung muss mit diesen Kompatibel sein und sollte möglichst sowohl für Bonjour / mDNS-SD als auch Bluetooth SDP genutzt werden können. Sie sollte also sowohl in Form einer UUID als auch als mDNS-SD Records nutzbar sein.

3.2.2 Nichtfunktionale Anforderungen

1. Der Code sollte hinreichend dokumentiert sein, um sowohl weitere Arbeiten an diesem als auch die Nutzung der Komponente zu erleichtern. Da Java als Programmiersprache verwendet wird und das *ASAPAndroid* GitHub Repository eine HTML-Seite für die JavaDoc Dokumentation bereitstellt sollte hier ebenfalls JavaDoc genutzt werden.

2. Die zu Unterstützende Android-SDK Version muss zum jetzigen Stand der von *ASAPAndroid* entsprechen, um Kompatibilitätsproblemen vorzubeugen. Damit wird die Komponente zunächst mit einem minimalen API-level von 23 und einem maximalen Level von 30 entwickelt.

3. Es sollten ausschließlich die, von Android bereitgestellten APIs und Java Standardbibliotheken verwendet werden, um somit Abhängigkeiten an Drittanbieter sowohl für *ASAPAndroid* als auch für die hier entwickelte Komponente zu vermeiden.

4. Die Service-Discovery soll schnell und zuverlässig erfolgen. Die Zuverlässigkeit kann hier anhand des Verhältnisses zwischen (angebotenen und gesuchten aber) nicht gefundenen Services gemessen werden. Hier sollte eine Fehlerquote von 10% nicht überschritten werden. Die Geschwindigkeit bezieht sich hier auf die Zeit, welche vom Starten einer Service-Discovery bis zum Finden aller gesuchten Services benötigt wird. Diese soll maximal 30 Sekunden betragen.

4 Entwurf

In diesem Kapitel wird der grundlegende Entwurf der Software erläutert. Dies soll an dieser Stelle in einer abstrakten Form geschehen, um die Konzeption und Funktionsweise der Komponente sowie die Beweggründe hinter ihrem Aufbau verständlich zu machen. Dabei sollen sowohl die Komponenten zur Service-Discovery und zum Verbindungsaufbau geplant als auch eine Möglichkeit zur Formulierung und Nutzung von Servicebeschreibungen gefunden werden.

4.1 Servicebeschreibungen

Dieser Abschnitt widmet sich den Servicebeschreibungen im Allgemeinen und den Möglichkeiten einer Umsetzung für Bluetooth SDP und Bonjour / mDNS-SD speziell unter Android und für diese Komponente. Ziel soll es hier sein eine Möglichkeit zu finden, Servicebeschreibungen zu formulieren. Diese sollen gleichermaßen für Bluetooth SDP und Bonjour / mDNS-SD nutzbar sein, um so der Komponente eine einheitliche Schnittstelle zu geben. Dabei sollen die in 3.2.1.3 erhobenen Anforderungen erfüllt werden.

4.1.1 Ziele der Servicebeschreibung

Eine Servicebeschreibung sollte in erster Linie dem Ziel dienen einen Service eindeutig zu identifizieren und somit auffindbar zu machen. Servicebeschreibungen können aber ebenfalls Informationen enthalten, welche über die Identifizierung einen Service hinausgehen aber für den Aufbau einer Verbindung relevant sind. Dazu gehören zum Beispiel verwendete Protokolle, Informationen über die Nutzung des Services und Informationen darüber, wie auf den Service zugegriffen werden kann (z.B. die Portnummer). Gesondert zu betrachten sind ebenfalls Informationen, welche weder für die Identifizierung eines Services noch für den Aufbau einer Verbindung zu diesem eine unmittelbare Relevanz haben. Dazu gehören beispielsweise Informationen, welche für den Endnutzer oder die „User-Experience“ zuträglich sind. An dieser Stelle ist auf Tabelle 1.

Tabelle 1 hingewiesen, in welcher eine Icon URL sowie ein lesbarer Servicename genannt werden.

4.1.2 Wege zum Austausch von Servicebeschreibungen unter Android

In diesem Abschnitt soll untersucht werden, welche Möglichkeiten für die Übertagung von Servicebeschreibungen unter Android und durch die gegebenen APIs für Bluetooth SDP und Bonjour geboten werden.

Es wurde bereits auf die von Bluetooth verwendeten Service Records sowie den SRV Record, den PTR Record und den TXT Record, welche von Bonjour genutzt werden (siehe 2.7.1), eingegangen. Auch die

von Android bereitgestellten APIs für Bonjour sowie Bluetooth SDP wurden beschreiben.
Android gewährt Anwendungen durch seine API keinen Zugriff auf die Bluetooth Service Records. Auf Serverseite kann ein Name und eine UUID für einen Service spezifiziert werden. Auf der Clientseite wird ausschließlich die UUID des Services zur Verfügung gestellt. Im Falle von Bonjour können der Name der Service Instanz und der Service Typ sowie der TXT Record bereitgestellt werden.

4.1.2.1 Instance Name und Service Type

Die Nutzung von *Instance Name* und *Service Type* sind in RFC 6763 geregelt und sollen in diese Arbeit entsprechend diesem genutzt werden. Beides soll vom Nutzer (Anwendungsentwickler oder Endnutzer) frei definiert werden können. Zu beachten ist, dass der *Instance Name* eines Services von Peer zu Peer variabel ist und es sich nicht um einen, allen Peers bekannten, festen Wert handelt (siehe 2.7.3.2). Da es sich hier um einen local-link Anwendungsfall handelt unterliegt dieser Teil jedoch keiner zentralen Regulierung, kann also, den Regeln der Formatierung entsprechend, frei gewählt werden (siehe 2.7.2).

4.1.2.2 TXT Records

In Kapitel 2.9 wurde bereits erläutert das ein TXT Record unter Android ebenfalls genutzt wird um verbindungsrelevante Informationen (Portnummern) auszutauschen. Eine optionale Nutzung dieser, soll dem Benutzer also ebenfalls gewährt werden.

4.1.2.3 UUIDs

Eine UUID ist für die Identifizierung zwar zweckmäßig, bring jedoch den Nachteil dass, ohne das Wissen über eine bestimmte UUID und welchem Service diese zugewiesen ist, weder für den Nutzer noch für die Anwendung eine Zuordnung möglich ist. Da eine UUID einem festgelegten Schma folgt und bereits codierte Informationen enthält (siehe 2.10), können nicht ohne weiteres andere, zusätzliche Informationen in diese codiert und auf Clientseite wieder decodiert werden. Dennoch bietet die UUID durch die große Anzahl an Kombinationsmöglichkeiten prinzipiell die Möglichkeit eine beliebig große Menge an Informationen zu identifizieren. Diese Zuweisung von UUID auf Informationen muss aber beiden Parteien (in diesem Fall Client und Server) bekannt sein.

Für diese Arbeit stellt sich damit die Frage, woher eine UUID für einen Service kommen kann und wie ihr weitere Serviceinformationen zugewiesen werden können.

4.1.3 Lösungsansatz

Eine festgelegte UUID innerhalb der hier beschriebenen Komponente würde unmittelbar zu dem folgenden Problem führen: Zwei, voneinander verschiedene, Anwendungen nutzen die Komponente zur

Service-Discovery. Beide stellen einen Service bereit und wollen mit anderen Instanzen derselben Anwendung kommunizieren. Für die Bluetooth Service-Discovery stellen aber beide den gleichen Service bereit, da die UUID des Services von der Service-Discovery-Komponente gegeben wurden.

Eine UUID kann auch vom Nutzer der Komponente (Entwickler einer Anwendung) definiert werden. Diese Möglichkeit besteht definitiv und kann in einigen Fällen auch notwendig sein. UUIDs müssen in diesem Falle jedoch im Vorhinein generiert werden, damit sie allen Anwendungen und allen Instanzen der Anwendung zur Verfügung stehen.

Eine Generierung (beispielsweise durch `java.util.UUID.randomUUID()`) von UUIDs zur Laufzeit auf mehreren Peers ist nicht möglich, da dies zu verschiedenen UUIDs führen würde. Auch für einen Anwendungsfall, in dem der Endnutzer einer Anwendung selbst eine Reihe von Services starten kann, sind UUIDs ein ungeeignetes Mittel.

Eine UUID kann ebenfalls auf Basis von Hashfunktionen / Hashwerten generiert werden, was erlaubt eine UUID für eine bestimmte Menge an Daten deterministisch zu generieren. Dabei handelt es sich um Name-Based UUIDs (siehe 2.10.3). Dies würde es erlauben, dass der Nutzer der Komponente ausschließlich die, für die Anwendung notwendigen Informationen zum Identifizieren des Services angeben muss, auf deren Basis dann eine UUID generiert werden kann. Hierbei ist zu beachten, dass diese Informationen ebenfalls allen Instanzen der Anwendung bekannt sein müssen. Informationen, welche von Instanz zu Instanz verschieden sind, können in der UUID-Generierung nicht berücksichtigt werden.

Dieses Vorgehen würde es ebenfalls erlauben, die für die DNS-SD Records notwendigen Informationen für die Generierung einer solchen UUID zu verwenden und somit eine, nach außen hin einheitliche, Abstraktion der Servicebeschreibung für beide Technologien zu schaffen.

Dabei sollen dann die für die DNS-SD notwendigen Informationen angeben werden. Für DNS-SD werden diese Informationen zwischen Server und Client tatsächlich ausgetauscht. Der Client muss diese nicht kennen. Für Bluetooth wird auf Basis dieser eine UUID generiert, welche auf Clientseite, sollte der Komponente eine entsprechende Beschreibung (mit UUID) bekannt sein, zurück auf diese Abgebildet werden kann. Dieses Vorgehen erlaubt es der Komponente die UUIDs ausschließlich intern nutzt und nach Außen ein einheitliches Interface bietet.

4.1.3.1 Aus welchen Informationen wird die UUID generiert

Die UUID soll aus den, in den DNS Records verwendeten, Informationen generiert werden. Es stellt sich dabei die Frage, welche der darin zur Verfügung stehenden Informationen für die Generierung der UUIDs verwendet werden können.

Der *Instance Name* eines Service ist variabel, kann sich also für denselben Service von Gerät zu Gerät unterscheiden. Es kann also nicht davon ausgegangen werden, dass der *Instance Name* eines Services dem Client bekannt ist. Für Bluetooth und die UUID bedeutet das, dass zwei Geräte verschiedene UUID generieren würden, obwohl es sich um denselben Service handelt.

Weitere Informationen können sich im TXT Record befinden. Ebenfalls kann dieser aber auch variable, nicht allen Peers bekannte Informationen beinhalten [32, Kap. 6.1] [13]. Hierbei kommt vor allem zum Tragen, dass Android den TXT Record auch zum Austausch von verbindungsrelevanten Informationen wie beispielsweise Portnummern vorsieht. Der TXT Record kann also ebenfalls nicht in die Generierung der UUID einbezogen werden.

In DNS-SD (und im weiteren Verlauf dieser Arbeit) werden Services basierend auf ihrem *Service Type* gesucht (*Service Instance Enumeration*), was es erlaubt den *Instance Name* und den TXT Record eines Services zu finden. Davon ausgehend soll die UUID aus dem *Service Type* generiert werden. Also aus dem Paar *<_ServiceName>.<_Transport>*.

4.1.3.2 Zusammenfassung

Es wird eine Datenstruktur vorgeschlagen, welche einen DNS-SD *Service Type* auf eine UUID abbildet. Ebenfalls enthält diese weitere Information für einen DNS-SD Service wie den *Instance Name* und den TXT Record. Wird ein Service durch Bonjour angeboten und gefunden, so erhält der Client alle notwendigen Informationen über den Service. Wird ein Service durch Bluetooth SDP angeboten und gefunden, so wird eine UUID basierend auf dem *Service Type* ausgetauscht. Besitzt der Client eine Servicebeschreibung mit derselben UUID können die Informationen bereitgestellt werden, welche lokal, auf diesem Peer zu diesem *Service Type* bekannt sind.

ASAPAndroid wird beispielsweise einen Service sowohl für Wi-Fi Direct als auch für Bluetooth anbieten. Hier kann mithilfe dieses Ansatzes ein Service „_asap._tcp.local.“ angeboten werden. Der TXT Record kann genutzt werden, um den Port für den Verbindungsaufbau unter Wi-Fi Direct zu übermitteln. Für Bluetooth wird daraus eine UUID generiert (b81b2a0f-dd35-3fef-8f1a-3cfd919d143e) welche von allen anderen *ASAPAndroid* Anwendungen erkannt wird, da diese ebenfalls über eine Service Beschreibung mit dem *Service Type* „_asap._tcp.local.“ verfügen.

4.2 Aufbau der Komponente

In dieser Arbeit soll sowohl eine Service-Discovery als auch ein, darauf basierender, Aufbau von Verbindungen entwickelt werden. Dies soll wie, eingangs erwähnt teilweise in *ASAPAndroid* integriert werden können. Für *ASAPAndroid* ist ausschließlich die Service-Discovery, nicht aber der Aufbau von P2P-

Verbindungen notwendig. Beides sollte also strukturell voneinander getrennt gehalten werden. Demnach wird sich die hier entworfene Komponente in zwei Teilkomponenten, eine zur Service-Discovery (siehe 4.3) und eine zum Aufbau von Verbindungen (siehe 4.4) unterteilenden.

4.3 Entwurf der Service-Discovery Teilkomponente

Die Service-Discovery Teilkomponente soll sowohl die Bluetooth- als auch die Wi-Fi Direct Service-Discovery enthalten. Beide lassen sich ebenfalls als voneinander getrennte Einheiten verstehen, teilen sich jedoch den zugrundeliegenden Entwurf. Dieser soll hier zusammenfassend als *Discovery Engine* bezeichnet werden. Abbildung 5 zeigt den groben Entwurf und die Grundfunktionalität einer *Discovery Engine*. Im Folgenden wird dieser Entwurf (bezugnehmend auf diese Abbildung) erläutert.

Eine *Discovery Engine* soll es ermöglichen eine Anzahl an n Services zu suchen und für den Nutzer

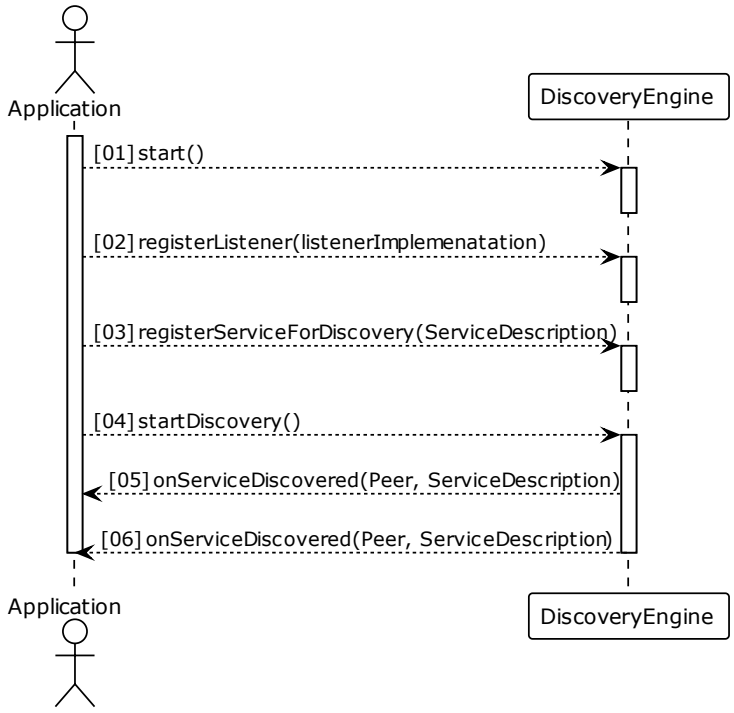

Abbildung 5 grober Entwurf der funktionsweise einer DiscoveryEngine (Eigene Darstellung)

(i.d.R. die Anwendung, welche die Komponente benutzt) alle notwendigen Informationen für einen Aufbau von Verbindungen zu den gefundenen Services bereitstellen. Somit muss es eine Möglichkeit geben einen oder mehrere Services für die Discovery zu registrieren (03) und, wenn nötig, wieder zu entfernen.

Da es möglich sein soll mehrere Services zu suchen und anzubieten, sollen die folgenden Interaktionen als voneinander getrennt betrachtet werden: das Registrieren eines Services (03) und das Starten einer Discovery (04) sowie das Entfernen eines Services aus der *Discovery Engine* und das Beenden der Discovery. Eine Discovery kann auf diese Weise mit 0 – n Services gestartet werden und jederzeit mit diesen oder weiteren Services wiederholt werden.

Es sollte auch festgehalten werden, dass ein Service-Discovery Prozess asynchron ablaufen sollte. Vom Registrieren eines Services über die Discovery selbst bis zum eventuellen Finden eines Services wird unweigerlich Zeit vergehen. Darüber hinaus kann ein Service eventuell über mehrere Discovery Vorgänge hinweg registriert bleiben. Für die Anwendung soll in diesem Fall ein Polling oder ein Blocking-Call auf die „Discovery"-Methode vermieden werden.

Um eine asynchrone Kommunikation zu ermöglichen, soll für die *Discovery Engine* ein Listener-Pattern verwendet werden, bei dem sich mehrere Listener für die Service-Discovery registrieren (und abmelden) können (02).

Eine *Discovery Engine* ist für die Service-Discovery verantwortlich. Es wird an dieser Stelle davon ausgegangen, dass das notwendige Setup der Umgebung vom Nutzer der Komponente durchgeführt, überprüft und in entsprechenden Fällen der Benutzer (der Anwendung) benachrichtigt wird. Dazu gehören *Permissions* (im Sinne von Android [33]), welche für die Nutzung von Bluetooth oder Wi-Fi Direct notwendig sind gleichermaßen, wie das Vorhandensein und die Verfügbargeit der notwendigen Hardware auf dem Gerät. Gleichwohl solle eine *Discovery Engine* keine Fehler produzieren, sollte sie dennoch genutzt werden. Dazu soll ein Mechanismus implementiert werden, bei dem eine *Discovery Engine* vor der Nutzung gestartet werden muss. Die `start()` Methode dient sowohl der Übergabe von notwendigen Abhängigkeiten (wie beispielsweise einem Android `Context`) als auch der Verifizierung des Vorhandenseins notwendiger Hard- und Software. Ist dies nicht der Fall wird die Engine nicht gestartet und kann nicht benutzt werden.

Eine weitere grundlegende Designentscheidung ist, dass es sich bei einer *Discovery Engine* um eine Singleton Klasse handeln muss. Dies liegt darin begründet, dass mehrere Instanzen einer solchen Klasse, welche zeitgleich auf den Bluetooth respektive den Wi-Fi P2P APIs arbeiten, sich gegenseitig behindern würden (beispielsweise durch das Starten der Device-Discovery, während eine andere Instanz eine Service-Discovery durchführt (siehe 2.8)).

4.4 Entwurf der Service-Connection Teilkomponente

Entsprechend der Discovery-Teilkomponente lässt sich die Komponente zum Verbindungsaufbau in zwei Teilkomponenten unterteilen, eine für Bluetooth und eine für Wi-Fi Direkt.

Auch hier lassen sich beide Teilkomponenten im Rahmen dieser Erläuterung zu einer Komponente mit dem Namen *Connection Engine* abstrahieren. Der grundlegende Entwurf für die *Connection Engine* bleibt gleich dem der *Discovery Engine*. Auch hier wird ein Singleton Pattern verwendet, und der Mechanismus zum Starten der Engine wird ebenfalls übernommen.

Eine *Connection Engine* kann als ein Wrapper einer *Discovery Engine* vom gleichen Typ verstanden werden, welcher diese um zusätzliche Funktionen zum Aufbau von Peer-to-Peer Verbindungen erweitert. Dieses Prinzip wird in Abbildung 6 dargestellt. Eine *Connection Engine* verhält sich dabei selbst als eines Listener und Nutzer der *Discovery Engine*.
Das Listener-Pattern kann ebenfalls, analog zu dem der *Discovery Engine*, für eine *Connection Engine* verwendet werden. Listener müssen hier erweitert werden, um auf die zusätzlichen Ereignisse, wie den Aufbau einer Verbindung, anwendbar zu sein.

Die *Discovery Engine*, welche von der *Connection Engine* genutzt wird, soll austauschbar sein, um verschiedene Implementierungen zuzulassen und die Testbarkeit der *Connection Engine* zu verbessern. Dazu wird die zu verwendende *Discovery Engine* ebenso wie andere, notwendige Abhängigkeiten in der `start()` Methode übergeben.

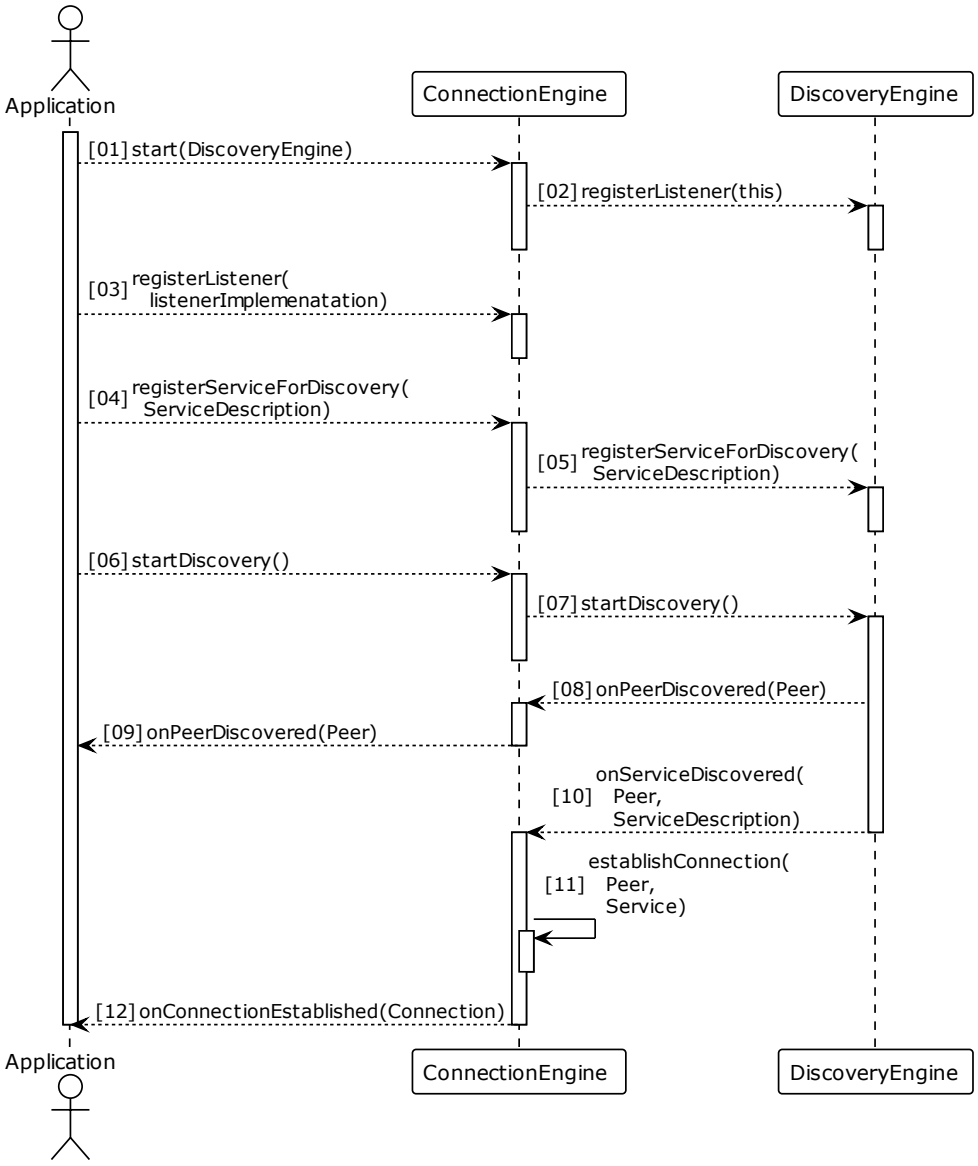

Abbildung 6 Interaktion zwischen einer Connection Engine und einer Discovery Engine

5 Implementierung Service-Discovery Teilkomponente

In den vorangegangenen Kapiteln wurden die technischen Grundlagen erläutert, eine Anforderungsanalyse durchgeführt und ein grober Entwurf für die Komponente dargelegt. Basierend darauf befasst sich dieses Kapitel mit der Implementierung der Komponente zur Service-Discovery. Die Service-Discovery enthält, entsprechend dem Entwurf, zwei Packages `bluetoothServiceDiscovery` und `wifiDirectServiceDiscovery`. Der Hauptbestandteil beider Packages ist sind die Klassen `BluetoothServiceDiscoveryEngine` und `WifiDirectDiscoveryEngine`. Es handelt sich bei beiden um Subklassen der abstrakten Klasse `DiscoveryEngine`, welche die Methoden zum Starten und Stoppen der Engines vorgibt, sowie Methoden zum Registrieren von Services zur Service-Discovery implementiert (siehe Abbildung 7 Vererbungsstruktur der Service-Discovery-Engins). Ebenfalls ist die Implementierung der Servicebeschreibungen in dieser Teilkomponente enthalten.

5.1 ServiceDescription

Die Klasse `ServiceDescription` implementiert die in 4.1 beschriebene Datenstruktur und ihr Verhalten. Eine `ServiceDescription` umfasst die in 4.1 genannten Informationen, wobei `serviceName`, `txtRecord` und der `serviceType` für die Initialisierung gegeben werden müssen.

```
1      HashMap<String, String> txtRecord = new HashMap<>();
2             txtRecord.put("name", "Example Service");
3             txtRecord.put("port", "4242");
4             ServiceDescription example = new ServiceDescription(
5                     "Example Service",
6                     txtRecord,
7                     "_exampleSrv._tcp");
```

Code 1 Initialisierung einer ServiceDescription

Die UUID wird mit `ServiceDescription.getServiceUuid()` einmalig generiert und in der Instanz gespeichert. Sollte es notwendig sein, so kann eine, von der generierten UUID abweichende, UUID angeben werden, welche dann für die Discovery mit Bluetooth SDP genutzt wird. Damit wird es ermöglich auch Services deren UUIDs nicht nach dem hier verwendeten Prinzip generiert werden zu suchen (oder anzubieten).

Beide `DiscoveryEngines` erlauben es ebenfalls alle Services in Reichweite zu suchen. Für Wi-Fi Direct wird dann eine `ServiceDescription` mit den gefundenen Informationen aus den DNS Record initialisiert. Da für Bluetooth in diesem Falle nicht immer eine `ServiceDescription` bekannt ist und somit keine weiteren Informationen verfügbar sind, wird diese mit der gefundenen UUID als Instance-Name initialisiert und die generierte UUID mit dieser überschrieben.

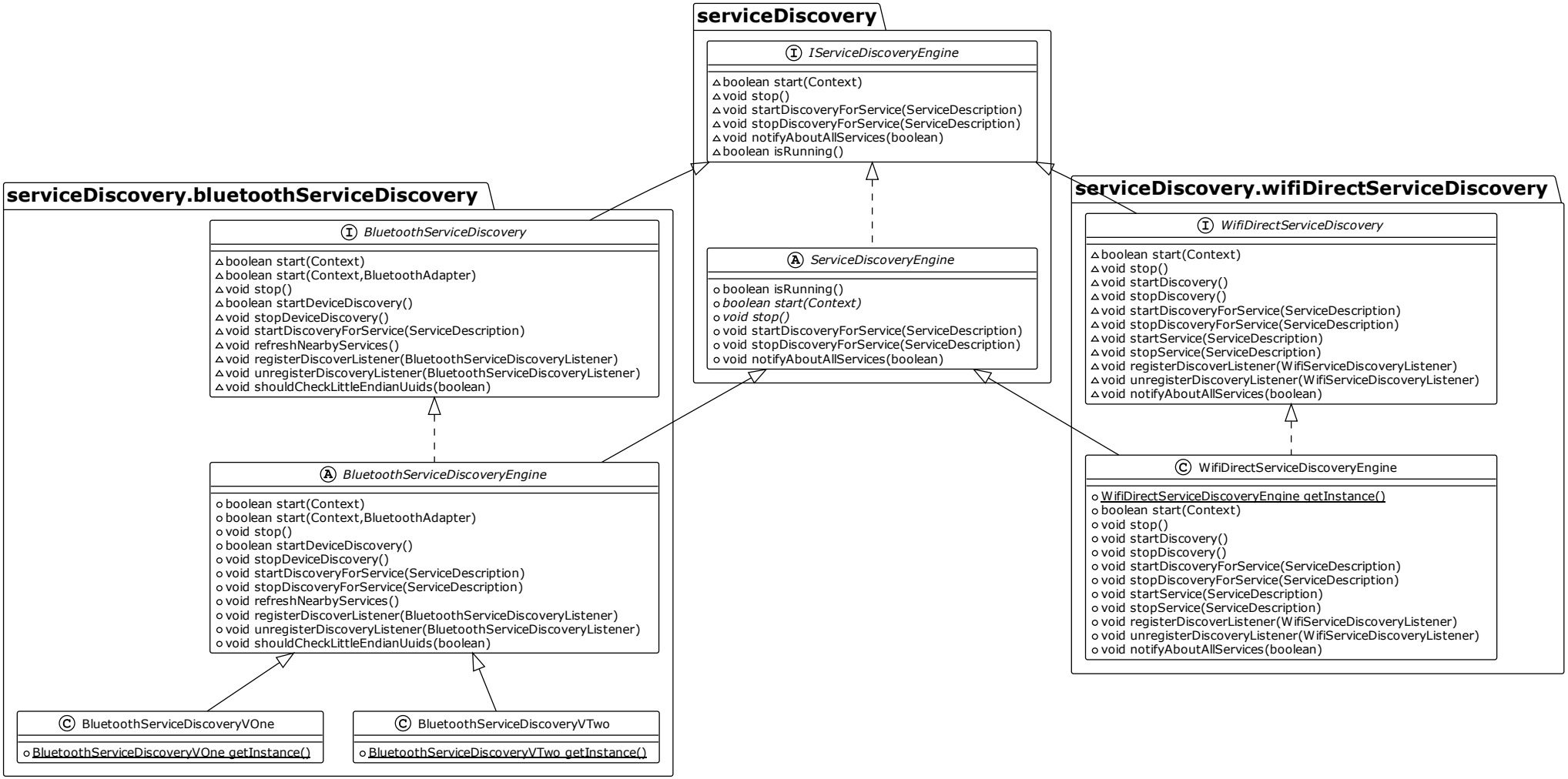

Abbildung 7 Vererbungsstruktur der Service-Discovery-Engins

5.2 ServiceDiscoveryEngine

Die `ServiceDiscoveryEngine` ist die abstrakte Superklasse der beiden Engines für Wi-Fi Direct und Bluetooth. Sie implementiert die Methoden `startDiscoveryForService()` und `stopDiscoveryForService()`, welche es erlauben einen Service, beschrieben durch eine `ServiceDescription`, zur Discovery hinzuzufügen. Grundlegend wird die `ServiceDescription` dafür in einer Liste abgelegt. Wird ein Service gefunden, so kann durch diese Liste überprüft werden, ob er gesucht wird.

5.3 Bluetooth Service-Discovery

Wie Abbildung 8 zu entnehmen ist wird diese Teilkomponente zum Großteil durch die `BluetoothServiceDiscoveryEngine` ausgemacht. Die hier gebotene Schnittstelle der Komponente besteht damit aus den öffentlichen Methoden der `BluetoothServiceDiscoveryEngine`. Wie ebenfalls in Abbildung 8 zu erkennen ist, handelt es sich bei der `BluetoothDiscoveryEngine` um eine abstrakte Klasse. Der Großteil des, für die Service-Discovery notwenigen, Codes ist hier implementiert. Auf die Varianten der `BluetoothServiceDiscoveryEngine` soll in 5.3.2 genauer eingegangen werden.

5.3.1 Broadcast Receiver

Die `BluetoothDiscoveryEngine` nutzt drei `BroadcastReceiver`, um auf eine Reihe von Bluetooth Events zu reagieren. Sie werden beim Start der Engine registriert und benachrichtigen diese, sobald Events eintreten.

Der `DeviceFoundReceiver` ist für den `BluetoothDevice.ACTION_FOUND` Broadcast registriert. Er wird dadurch über (bei einer Device-Discovery) gefundene Geräte informiert. Wird ein neues Gerät gefunden so benachrichtig er die `BluetoothServiceDiscoveryEngine` durch die Methode `onDeviceDiscovered(BluetoothDevice)`.

Der `UUIDFetchedReceiver` ist für den `BluetoothDevice.ACTION_UUID` Broadcast registriert. Der Intent enthält ein Array an UUIDs und das entsprechende `BluetoothDevice`. Er stellt somit die asynchrone Antwort auf `BluetoothDevice.fetchUuidsWithSdp()` dar. Die `BluetoothServiceDiscoveryEngine` wird durch die Methode `onUuidsFetched(BluetoothDevice, Parcelable[])` über den Erhalt informiert.

Der `DeviceDiscoveryStateReciver` ist für die Actions `BluetoothAdapter.ACTION_DISCOVERY_STARTED` und `BluetoothAdapter.ACTION_DISCOVERY_STARTED` registriert. Wird der entsprechende Broadcast empfangen wird die `BluetoothServiceDiscoveryEngine` durch `onDeviceDiscoveryFinished()` darüber informiert.

5.3.2 Varianten der Bluetooth Service-Discovery

Es wurden zwei Varianten der Bluetooth Service-Discovery implementiert, welche von der `BluetoothServiceDiscoveryEngine` erben und somit auch das `BluetoothServiceDiscovery` Interface implementieren (siehe Abbildung 7 Vererbungsstruktur der Service-Discovery-Engins).

Die Discovery für Bluetooth setzt sich aus zwei Teilprozessen zusammen, die Device-Discovery und die Service-Discovery. Wobei die Device-Discovery vor der Service-Discovery durchgeführt werden muss. Ein paralleles Ausführen beider kann zu Fehlern führen und muss vermieden werden (siehe 2.8.5). Dies führt zu zwei möglichen Vorgehensweisen:

1. Die Device-Discovery wird einmal gestartet und dauert 12 Sekunden an. Ist die Device-Discovery abgeschlossen startet der Service-Discovery Prozess, bei dem eine SDP-Request an alle gefundenen Geräte gesendet wird.

2. Die Device-Discovery wird nach jedem (einzelne) gefunden Gerät unterbrochen, eine SDP-Request wird an das gefundene Gerät gesendet und, sobald die UUIDs empfangen wurden, wird die Device-Discovery erneut gestartet.

5.3.2.1 Variante 1

Die Implementierung der erste Variante ist in Abbildung 8 dargestellt. Die Device-Discovery wird hier initial einmal gestartet (05) und läuft für 12 Sekunden. Alle gefunden Geräte werden in einer Liste abgelegt. Das Ende der Device-Discovery wird durch den `DeviceFoundReceiver` mitgeteilt (10). Daraufhin wird mit `fetchUuidsWithSdp()` an jedes gefundenen `BluetoothDevice` eine SDP Requests gesendet. Über gefundene UUIDs wird die `BluetoothServiceDiscoveryEngine` durch den `UUIDFetchedReceiver` informiert. Daraufhin werden die gefunden UUIDs mit denen der registrierten `ServiceDescriptions` verglichen und wenn eine passende UUID gefunden wurde, die Listener benachrichtigt (13). Der Nachteil dieser Variante ist, dass Services nicht vor dem Ende der Device-Discovery, also erst nach minimal 12 Sekunden, gefunden werden können.

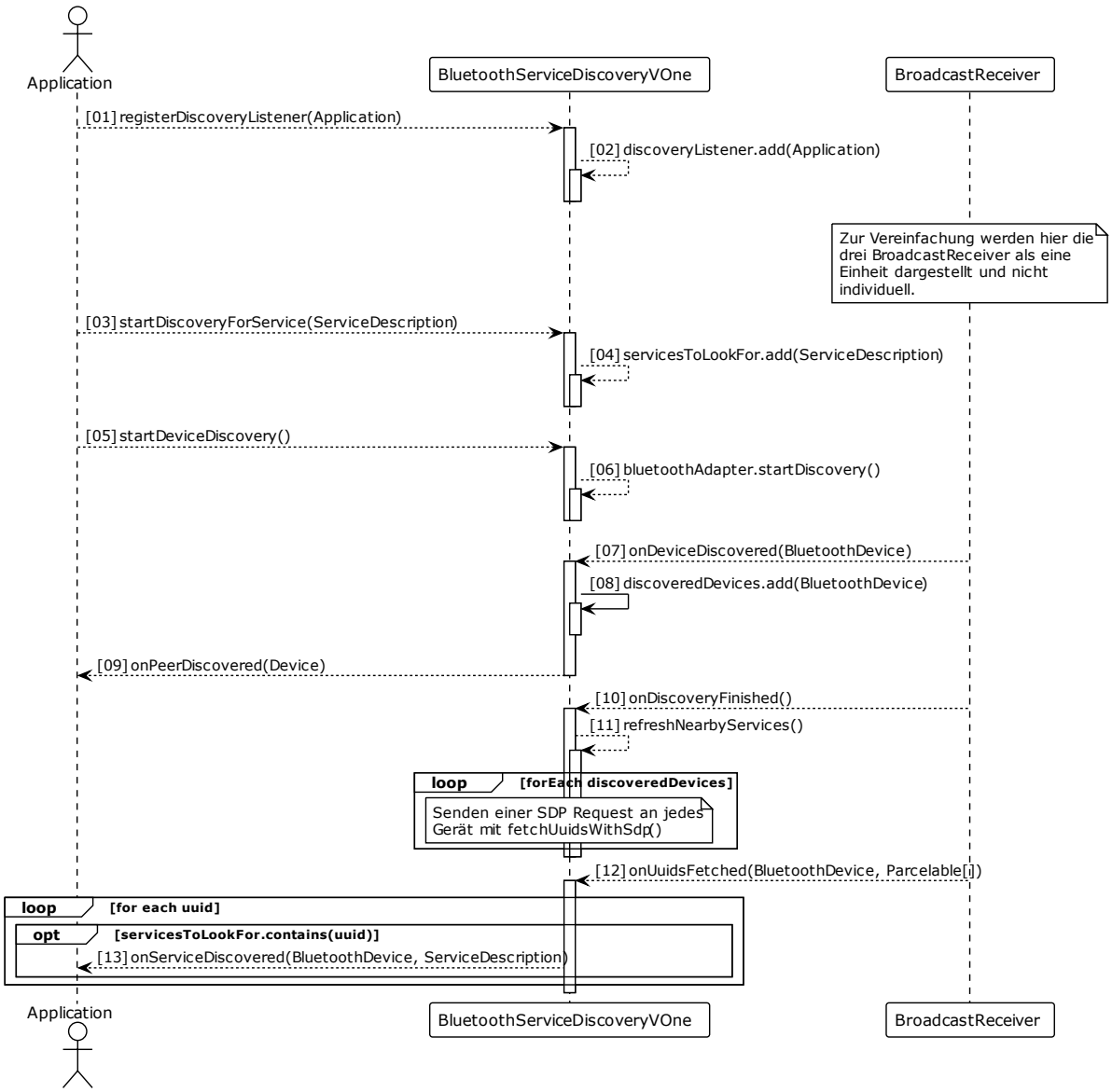

Abbildung 8 Ablauf einer Service-Discovery (Variante 2)

5.3.2.2 Variante 2

Für die zweite Variante wird die Discovery pro gefundenem Device beendet und neugestartet.
Dies sollte es ermöglichen Services schneller zu finden, wenn diese auf einem der ersten gefunden Geräte laufen.

Abbildung 9 zeigt den (vereinfachten) Ablauf einer solchen Discovery. Auf diesen soll im Folgenden genauer eingegangen werden. (01) bis (03) beinhalten das Setup: Ein Listener wird registriert und ein Service (durch eine `ServiceDescription`) wird zur Suche hinzugefügt. Diese Schritte können mehrmals durchlaufen werden.

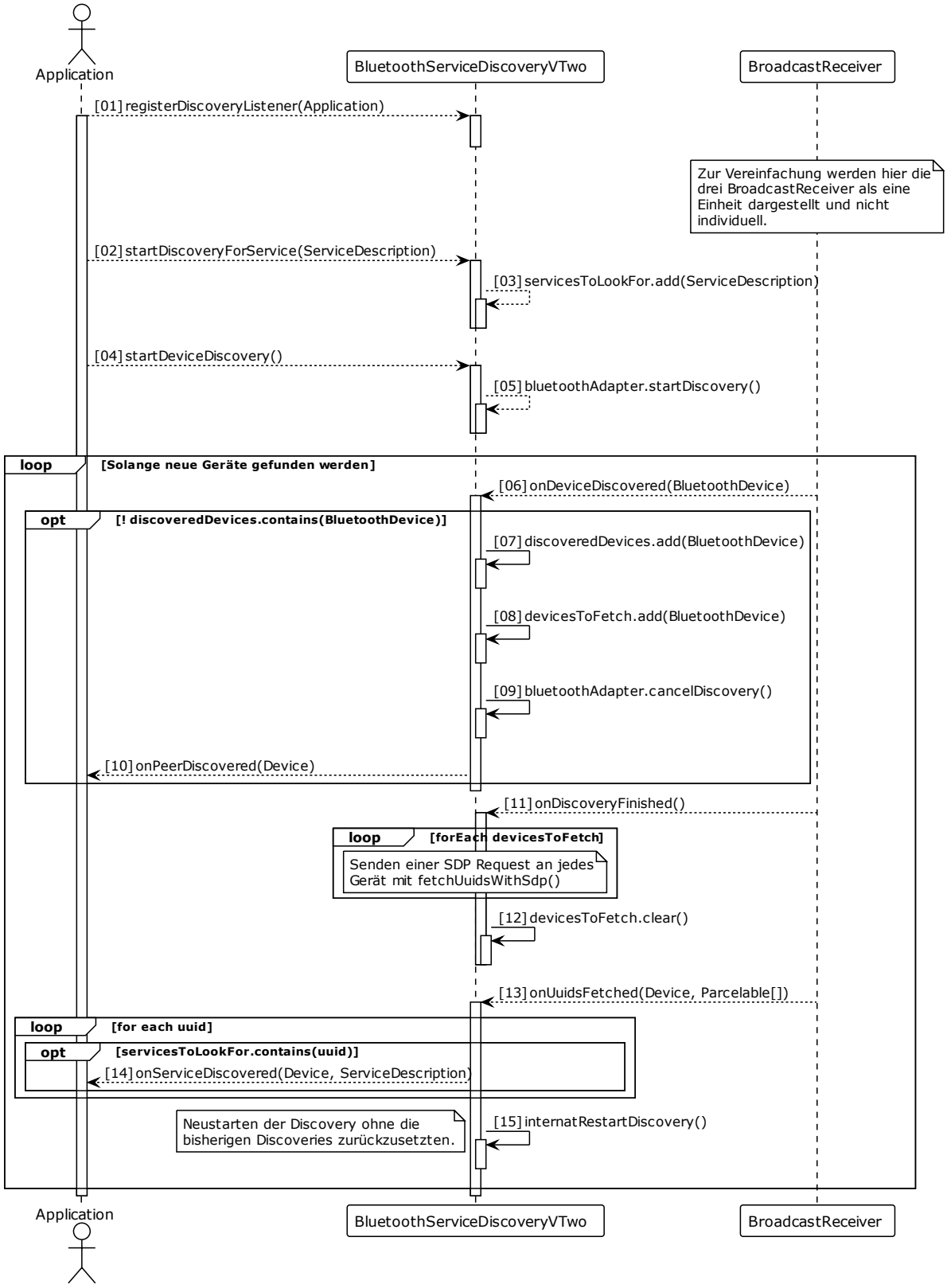

Abbildung 9 Ablauf einer Service-Discovery (Variante 2)

Die Device-Discovery und die Subsequente Service-Discovery werden in (04) gestartet. Wie bereits in Variante 1 wird die Engine durch den `DeviceFoundReceiver` über gefunden Geräte benachrichtigt.

Handelt es sich um ein neues Gerät so wird hier die Device-Discovery beendet (08). Sobald die Discovery von Android beendet wurde (10), wird eine SDP Request an das gefundene Gerät gesendet.
Trotz der schnellen Beendigung der Service-Discovery kam es vor, dass mehrere Geräte gefunden wurden. Daher werden diese in einer Liste zwischengespeichert (07). Es wird dann eine SDP-Request an alle diese Geräte gesendet. Der Zwischenspeicher wird daraufhin geleert (11).

Ist ein gefundenes Gerät bereits bekannt, so werden die Schritte (06)-(08) übersprungen, und die Device-Discovery wird nicht unterbrochen. Dadurch wurde einem der Probleme mit dieser Variante entgegengewirkt, welches durch das Neustarten der Device-Discovery zustande kommt: Wenn die Discovery neu gestartet wird, werden alle Geräte, auch jene, welche in einem vorangegangenen Durchgang bereits gefunden wurden, erneut gefunden. Theoretisch kann also ein Gerät immer wieder gefunden werden und alle anderen Geräte werden ignoriert. Um dem vorzubeugen wird, sobald die Service-Discovery für ein Gerät abgeschlossen ist, dieses Gerät auf eine Blacklist gesetzt. Für Geräte auf der Blacklist wird die Discovery nicht unterbrochen. Die Blacklist bleibt bis zu einem manuellen Neustart der Discovery durch (`startDeviceDiscovery()`) erhalten.

5.3.3 Optimierungen

Egal welche der beiden Varianten (siehe 5.3.2) genutzt wird es dauert eine Weile, bis Geräte in der Nähe und Services auf diesen gefunden werden. Um dies zu verbessern wurden eine Reihe von zusätzlichen Funktionen implementiert.

Dazu gehört in erster Linie das cachen von gefundenen Geräten in einer Device-Discovery. Gefundene Geräte werden dafür in einer Liste vorgehalten. Bekannte Services können durch `BluetoothDevice.getUuids()` abgerufen werden. Da davon ausgegangen werden kann, dass nach einer neuerlichen Discovery neuere Daten verfügbar sind, wird dieser Cache mit jedem Start der Device- / Service-Discovery geleert.

Dieser Zwischenspeicher kann genutzt werden, wenn ein neuer Service zur Suche registriert wird. Ist bereits ein Gerät bekannt, welches den gesuchten Service anbietet, so können die Listener sofort benachrichtigt werden. Dies wird in Abbildung 10 dargestellt.

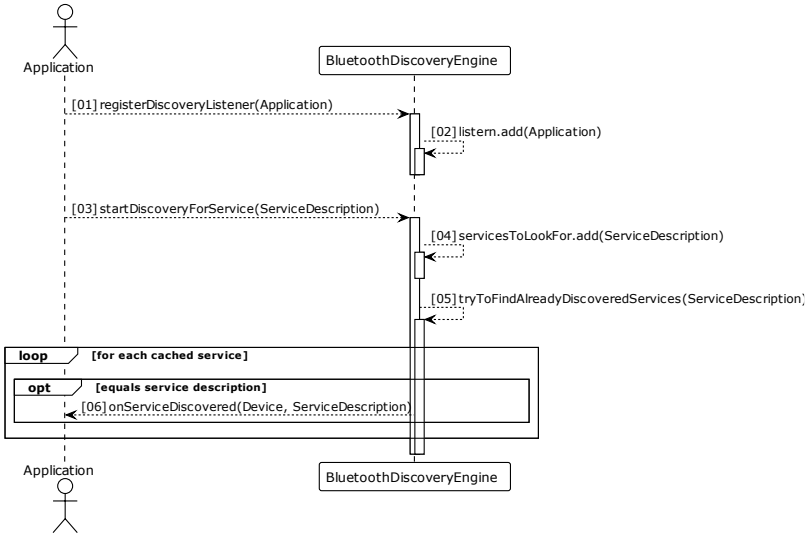

Abbildung 10 Benachrichtigen über bereits gefundene Services

Des Weiteren ist es nicht immer sinnvoll oder notwendig für jede Service-Discovery auch eine Device-Discovery durchzuführen. Durch den schon existierenden Cache können Services für dort gespeicherte Geräte auch ohne eine neuerliche Device-Discovery aktualisiert werden. Dafür bietet die `BluetoothServiceDiscoveryEngine` die Methode `refreshNearbyServices()`, welche eine Service-Discovery auf allen bekannten Geräten durchführt.

Beide Funktionen verbessern die Geschwindigkeit, mit der Services nach einer initialen Device-Discovery / Service-Discovery Kombination, gefunden werden können. Das bringt nicht ausschließlich Vorteile. Sollte ein Service gecached worden sein, sich aber mittlerweile außer Reichweite befinden, so wird über einen veralteten Eintrag benachrichtigt und eine Verbindung kann nicht aufgebaut werden.

5.3.4 Anbieten von Services

Teil der Anforderung war es Services anbieten zu können (siehe 3.2.1.1). Diese Anforderung wird von der `BluetoothServiceDiscoveryEngine` nicht erfüllt. Wie einführend in 2.8.4 erläutert, ist das Anbieten eines Services an das Öffnen eines `BluetoothServerSockets` gebunden. Das Öffen von Sockets und das Akzeptieren von Verbindungen ist nicht Aufgabe der `BluetoothServiceDiscoveryEngine`. Eine Implementierung dessen muss auf Seite der Anwendung geschehen, welche die Verbindungen aufbauen will. Im Rahmen dieser Arbeit wird diese Funktion von der `BluetoothServiceConnectionEngine` implementiert.

5.3.5 Little Endian UUIDs

Im Laufe der Arbeit konnte beobachtet werden, dass auf einem der Testgeräte ein persistentes Problem auftritt, welches darin liegt, dass UUIDs in ihrer Little-Endian Repräsentation erhalten werden. Dies gilt nicht für alle UUIDs, sondern ausschließlich für die, welche bei der Service-Discovery im Feld `BluetoothDevice.EXTRA_UUID` nach eine `fetchUuidsWithSdp()` erhalten werden.

Eine Untersuchung des Problems ergab, dass es nicht auf die verwendeten Android-Version zurückzuführen ist. Das Problem trat auf einem Tablet mit Android 8.1 auf. Auf einem Referenzgerät, ebenfalls mit Android 8.1, konnte dies nicht reproduziert werden. Nach weiterer Recherche konnte ermittelt werden, dass dieses Problem bereits in früheren Android Versionen aufgetreten ist [34]. Neuere Informationen zu diesem Problem konnten nicht gefunden werden.

Es konnte keine Möglichkeit gefunden werden dieses Problem zur Laufzeit und gerätespezifisch zu identifizieren und somit direkt auf betroffenen Geräten zu adressieren. Aus Ermangelung weiterer Geräte, welche dieses Problem aufwiesen, konnte ebenfalls kein etwaiger hardwareseitiger Zusammenhang hergestellt werden. Auch eine Abschätzung darüber, wie weit verbreitet das Problem ist, kann nicht abgegeben werden. Der einzig verbleibende Weg diesem Fehler vorzubeugen war es, sämtliche UUIDs welche empfangen werden, sowohl in ihrer Little- als auch Big-Endian Form zu vergleichen. Diese Funktion kann durch `.shouldCheckLittleEndianUuids(boolean)` deaktiviert bzw. aktiviert werden.

5.4 Wi-Fi Direct Service-Discovery

Die Service-Discovery für Wi-Fi Direct ist in der Klasse `WifiDirectServiceDiscoveryEngine` implementiert. Sie verfolgt den in Kapitel 4.3 beschriebenen Ansatz. Die `WifiDirectServiceDiscoveryEngine` ermöglicht es, eine Service-Discovery auf Basis von mDNS / Bonjour durchzuführen. Als eine Subklasse der `DiscoveryEngine` lassen sich Services durch die Methoden `startDiscoveryForService()` und `stopDiscoveryForService`() für die Discovery anmelden und abmelden.

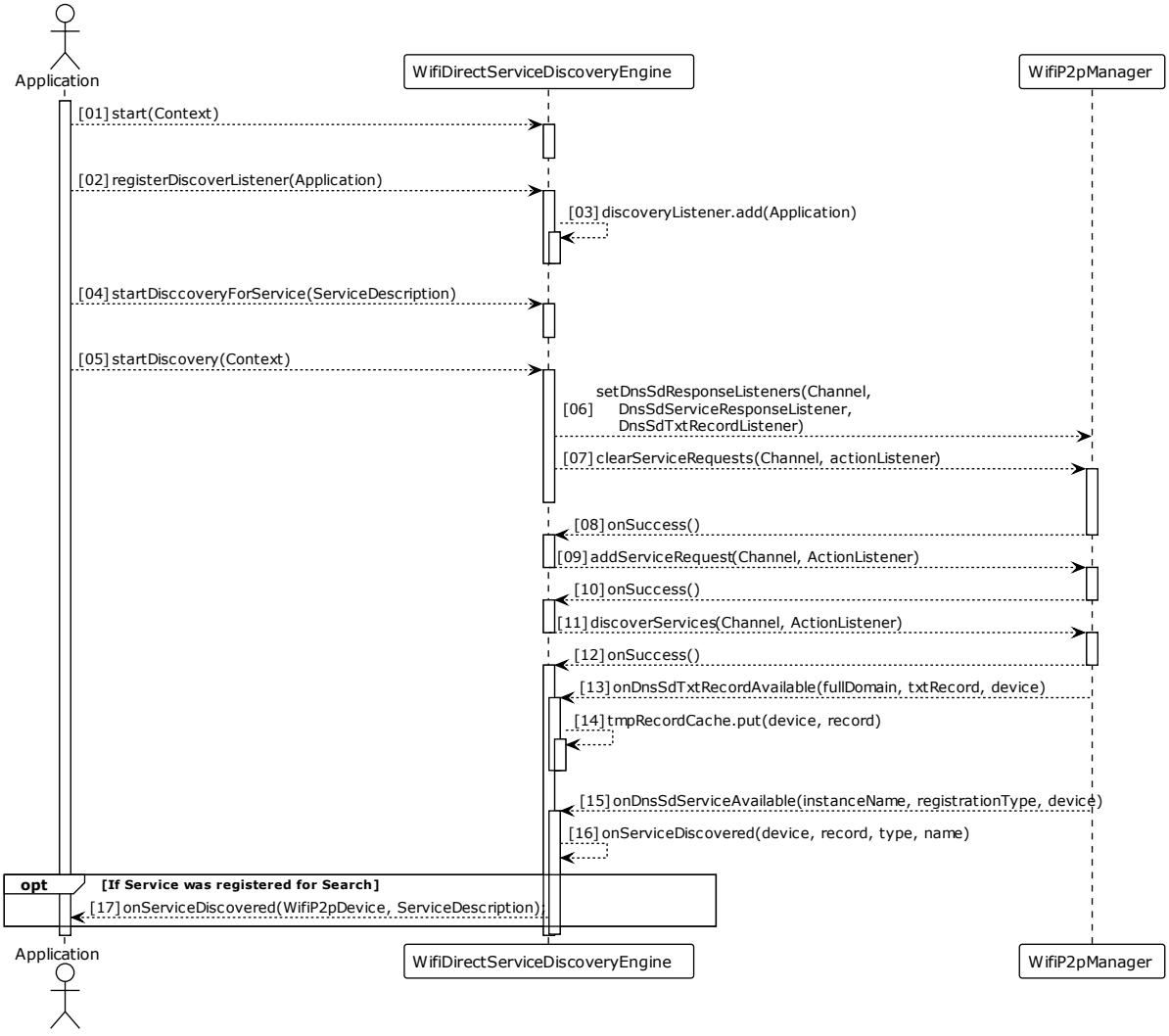

Abbildung 11 Ablauf einer Service-Discovery mit der WifiDirectServiceDiscoveryEngine

5.4.1 Service-Discovery

Für die Service-Discovery wird die erste Variante der in 2.9.4 gelisteten `WifiP2pDnsSdServiceRequest` varianten verwendet. Diese erlaubt es eine mDNS-Service-Discovery für alle Services in der Umgebung durchzuführen ohne das ein *Service Type* oder eine genaue Instanz bekannt sind. Dabei wird auch eine *Service Instance Resolution* durchgeführt und der TXT Record aller gefundenen Instanzen ermittelt.

Die anderen Varianten der (ebenfalls in 2.9.4 gelistet) Requests in Kombination, würden dagegen einige Nachteile mit sich bringen. So kann eine Service-Discovery nur mit mindestens einem bekannten (registrierten) *Service Type* durchgeführt werden. Für jeden weiteren, davon verschiedenen *Service Type*, muss eine weitere Request registrieret werden. Damit wird dann eine *Service Instance Enumeration* durchgeführt. Entsprechend wird dabei kein TXT-Record der gefunden Instanzen ermittelt. Da der TXT Record hier aber auch für weitere, zum Verbindungsaufbau relevante, Informationen genutzt wird müsste dieser dann, mit einer weiteren `WifiP2pDnsSdServiceRequest` (2.9.4, Typ 3) und einem weiteren Start der Discovery, nachträglich ermittelt werden.

Somit ähnelt das hier verwendete Vorgehen dem, welches bereits für die Bluetooth Service-Discovery angewendet wurde. Es werden alle verfügbaren Services gesucht. Benachrichtigt werden Listener aber nur, wenn einer der gefundene Service bei der Engine registriert wurde.

5.4.2 Service Advertisement

Ein Service kann durch die Methode `startService(ServiceDescription)` angeboten werden und bleibt für Clients sichtbar bis dieser durch `stopService(ServiceDescription)` wieder beendet wurde. Dafür wird eine `WifiP2pDnsSdServiceInfo` (siehe 2.9.3) mit den Informationen der `ServiceDescription` initialisiert und registriert (siehe Code 2).

```
1  WifiP2pServiceInfo serviceInfo = WifiP2pDnsSdServiceInfo.newInstance(
2                  description.getInstanceName(),
3                  description.getServiceType(),
4                  description.getTxtRecord());
5          manager.addLocalService(channel, serviceInfo, actionListener);
```

Code 2 Registrieren eine mDNS-Service mit ServiceDescription

Diese kann dann dem `WifiP2pManager` mit `.addLocalService()` übergeben werden. Entsprechend diesem vorgehen kann ein Service auch wieder beendet werden.

5.4.3 WifiDirectServiceDiscoveryListener

Das Interface `WifiDirectServiceDiscoveryListener` gibt die, für einen Listener auf der `WifiDirectServiceDiscoveryEngine` notwendigen, Callbackmethoden vor. Es beschreibt einzig die Methode `onServiceDisoveres(WifiP2pDevice, ServiceDescription)`, welche der Benachrichtigung der Listener über einen gefundenen Service dient.

5.4.4 Zuverlässigkeit

Die mDNS Service-Discovery stellte sich im Laufe der Arbeit als oft unzuverlässig heraus. Es konnten nicht immer alle der angebotenen Services gefunden werden. Phasenweise wurden Services generell oder von bestimmten anderen Geräten nicht gefunden.

Im Zuge der Untersuchung dieses Problems wurden die Android Logs ausgewertet. Es konnte dabei festgestellt werden, dass das Fehlschlagen der Service-Discovery häufig auf Probleme bei der, von Android intern durchgeführten, Wi-Fi Direct Peer-Discovery zurückzuführen ist. In vielen Fällen konnte der Wi-Fi Direct Peer, der den Service anbietet, nicht gefunden werden. Dieses Problem ließ sich in der Regel durch ein Wiederholen der Discovery beheben, konnte aber auch in persistenter Form auftreten. Hier konnte ebenfalls beobachtet werden, dass zeitweise genau einer der Peers über mehrere Discovery-Vorgänge hinweg nicht gefunden wurde während, anderer Peers weiterhin gefunden werden konnten.

Während der Tests wurde auf einem der Geräte ein Verhalten beobachtet, bei dem die Peer- und Service-Discovery ohne Fehlermeldung angehalten wurde. Dieses Verhalten ließ sich über mehrere Testläufe hinweg reproduzieren. Dabei zeigte sich, dass die Discovery kurz nach dem ersten gefundenen Peer abgebrochen wurde. Dabei handelt es sich jedoch um eine Ausnahme die ausschließlich auf einem der genutzten Testgeräte beobachtet werden konnte.

Die häufigste Ursache für nicht gefundene Services konnte somit auf nicht gefunden Peers zurückgeführt werden. Oft konnte dieses Problem durch Wiederholen der Discovery behoben werden, trat aber phasenweise auch über mehrere Discovery-Vorgänge hinweg auf, was dazu führte, dass auf dem betreffenden Gerät eine Service-Discovery unmöglich war.

Es wurden eine Reihe von Versuchen unternommen, um diesem Verhalten entgegenzuwirken.
Dazu gehörte in erster Linie ein Versuch, die Methodenaufrufe auf dem `WifiP2pManger` jeweils einzeln und in korrekter Reihenfolge zu tätigen, wobei jeweils auf den Aufruf von `.onSuccess()` des `ActionListener` gewartet wird (siehe Abbildung 11), um somit eine korrekte Reihenfolge aller Methodenaufrufe zu forcieren. Auch das periodische Neustarten des Discovery-Prozesses durch `discoverServices()` in einem gesonderten Thread, sowie das gleichermaßen periodische Zurücksetzten und Neustarten der gesamten Discovery-Sequenz (siehe Abbildung 7, 07-12), wurde versucht. Ebenfalls wurde es versucht, parallel eine manuelle Wi-Fi Direct Peer-Discovery durchzuführen und periodisch neu zu starten, um so Fehlern in dieser entgegenzuwirken.

Mit keinem der unternommenen Versuche konnte jedoch eine dauerhafte Veränderung dieses Verhaltens erreicht werden. Die hier als final zu betrachtende Implementierung beschränkt sich somit auf eine korrekte Nutzung der bereitgestellten API, ohne die soeben beschriebenen Versuche der Verbesserung dieses Verhaltens zu implementieren.

6 Implementierung der Teilkomponente zum Verbindungausbau

Die *Service Connection* Teilkomponente erlaubt es automatisch Peer-to-Peer Verbindungen zu Services aufzubauen. Sie umfasst dabei, analog zu *Service-Discovery*, die Klassen `BluetoothServiceConnectionEngine` und `WifiDirectServiceConnectionEngine`. Wie bereits im Entwurf dieser Komponente beschrieben, nutzen beide die jeweilige *Discovery Engine* und geben vollen Zugriff auf deren öffentliche Methoden. Sie stellen somit einen Wrapper für die *Service-Discovery* dar, welcher diese um Funktionen zum Aufbau von Verbindungen zwischen Clients und Services erweitert.

6.1 Wi-Fi Direct

Mit der `WiFiDirectServiceConnectionEngine` können automatisiert Wi-Fi Direct Gruppen auf der Grundlage eines Services aufgebaut werden. Sie setzt dabei den in Kapitel 4.4 dargelegten Entwurf um. Aufgrund des Aufbaus eines Wi-Fi Direct Netzwerkes (siehe 2.6) war es für die `WifiDirectServiceConnectionEngine` nicht notwendig mehrere Services anbieten oder suchen zu können. Somit kann genau ein Service angeboten beziehungsweise gesucht werden.

6.1.1 Aufbau der Gruppe

Die Engine versucht sich opportunistisch immer mit dem ersten gefundenen Service/Peer zu verbinden. Sind beide Peers noch nicht Teil einer Gruppe wird eine automatische Group-Owner-Election von Android durchgeführt und einer der beiden Peers wird zum Group Owner.
Der nun entstandene Client Peer lässt ab diesem Zeitpunkt keine weiteren Verbindungsversuche von anderen Peers zu. Beide führen aber die Service-Discovery fort. Der Group Owner kann so neue Peers finden und sich mit diesen verbinden. Der Client wird zwar keine weiteren Peer-To-Peer Verbindungen akzeptieren, kann aber an jeden gefundenen Peer mit demselben Service eine Einladung in die Wi-Fi Direct Gruppe senden. Dies funktioniert in beiden Fällen mit der `WifiP2pManager.connect()` Methode.

6.1.2 WifiDirectPeer

Das `WifiDirectPeer` Interface gibt für die `WifiDirectServiceConnectionEngine` notwendigen Callback-Methoden für die Listener vor. Eine Implementierung dieses Interfaces muss beim Starten einer Discovery oder eines Services übergeben werden, und dient als Schnittstelle zwischen der Implementierung des Services und dem, hier stattfindenden, Verbindungsaufbau. Das Interface ist in Code 3 dargestellt und umfasst sowohl Methoden, um auf die Group Owner Election als auch über aufgebaute TCP-Socket-Verbindungen zu reagieren.

```
1 public interface WifiDirectPeer
2 {
3     void onServiceDiscovered(WifiP2pDevice device, ServiceDescription description);
4
5     void onBecameGroupOwner();
6
7     void onBecameGroupClient();
8
9     void onConnectionEstablished(WifiConnection connection);
10
11    boolean shouldConnectTo(WifiP2pDevice device, ServiceDescription description);
12 }
```

Code 3 WifiDirectPeer Interface

6.1.3 WifiConnection

Die Klasse `WifiConnection` beschreibt eine P2P-Verbindung, welche auf Grundlage eines Services entstanden ist. Sie enthält das verbundene Socket und die `ServiceDescription` des Services, mit dem sie verbunden ist oder auf dessen Basis sie entstanden ist. Sie gibt Zugriff auf den In- bzw. OutputStream des Sockets und ermöglicht so den Nachrichtenaustausch. Auch stellt sie eine Methode bereit, mit der die Streams und das Socket ordnungsgemäß geschlossen werden können. Sie ist das Endprodukt eines erfolgreichen Verbindungsaufbaus und wird dem `WifiDirectPeer` übergeben.

6.2 Bluetooth

6.2.1 BluetoothServiceConnectionEngine

Die `BluetoothServiceConnectionEngine` erlaubt es, mehrere Services anzubieten und sich mit mehreren Services zu verbinden. Die Funktionsweise der `BluetoothServiceConnectionEngine` wird in Abbildung 12 und Abbildung 13 dargestellt.

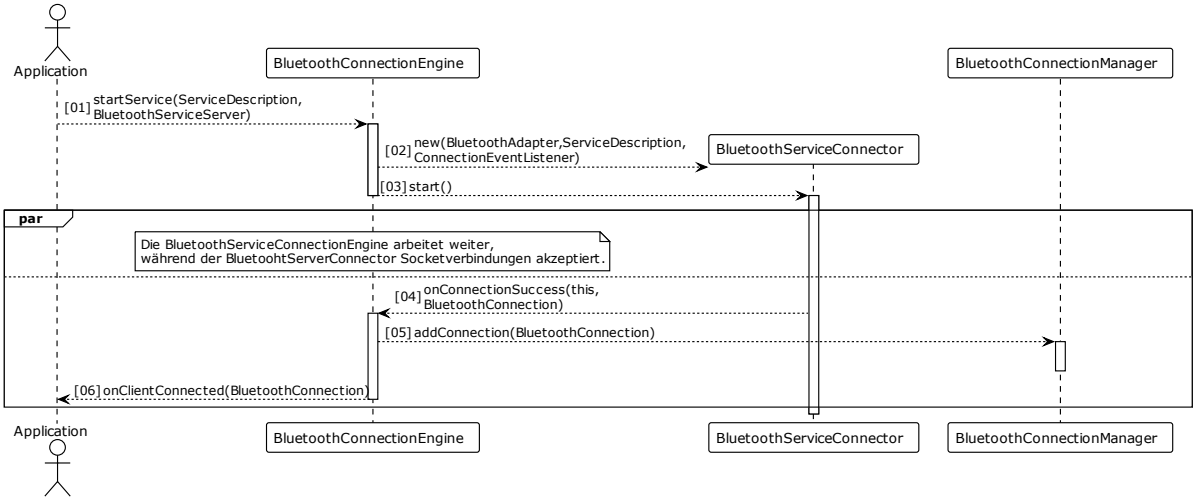

Abbildung 12 Anbieten eines Services und akzeptieren von Client-Verbindungen.

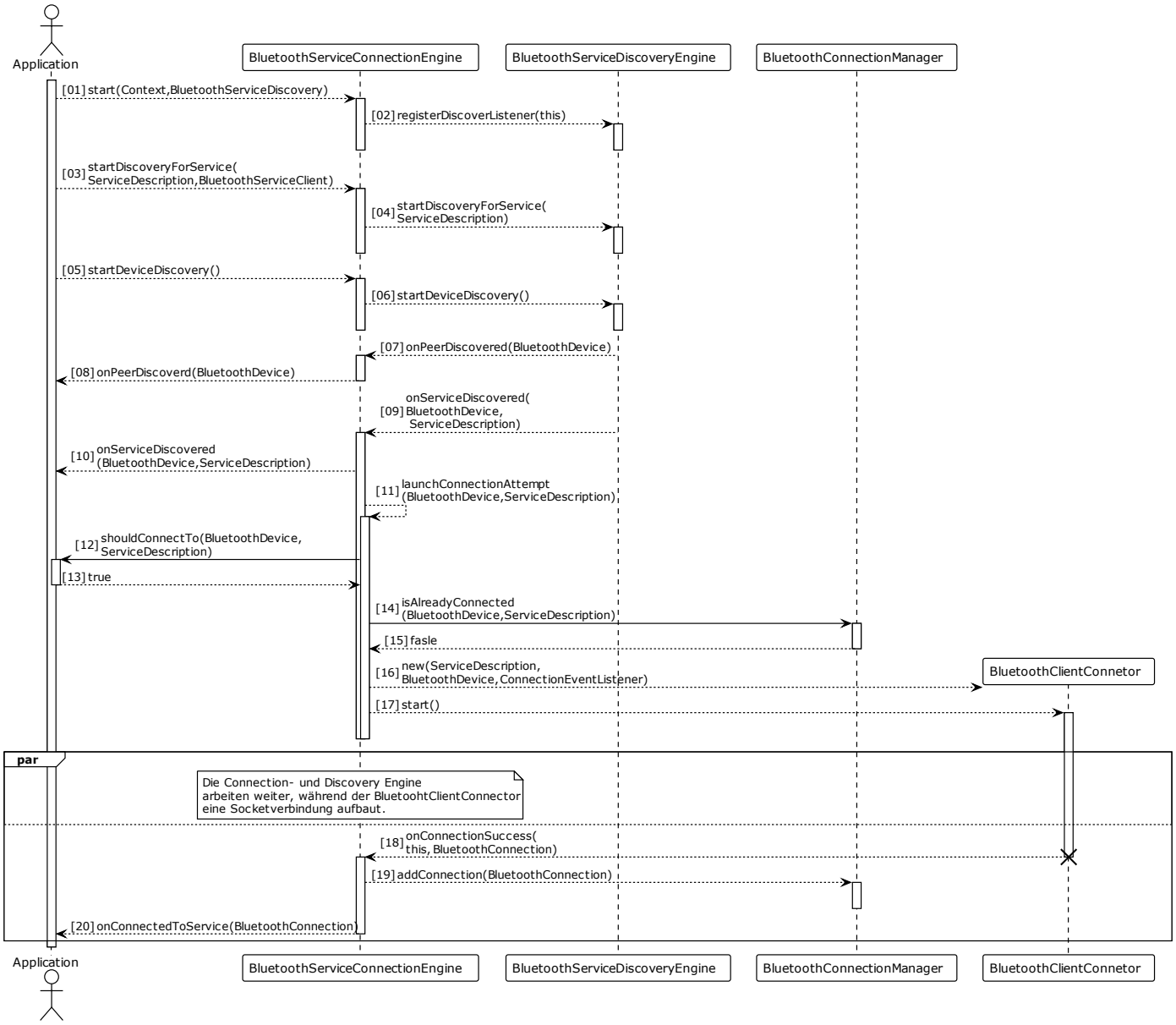

Abbildung 13 Ablauf einer Bluetooth Service Discovery auf Clientseite

6.2.2 BluetoothConnection

Die Klasse `BluetoothConnection` beschreibt eine P2P-Verbindung zweier Bluetooth Geräte, die auf der Grundlage eines Services entstanden ist. Sie enthält das verbundene `BluetoothDevice`, die `ServiceDescription` des Services und das `BluetoothSocket`, welches die Kommunikation mit dem Peer erlaubt. Der Boolean `isServer` gibt Aufschluss, welche Rolle der lokale Peer beim Aufbau der Verbindung hatte. Dies kann für einige Anwendungsfälle wichtig sein. Stellt die `BluetoothServiceConnectionEngine` eine P2P-Verbindung her, so erhält der entsprechende Listener diese in Form eine `BluetoothConnection`.

6.2.3 BluetoothConnectionManager

Es ist Teil der Anforderungen, dass Verbindungen einmalig pro Service und verbundenem Gerät sein sollen. Die `BluetoothConnectionManager` Klasse adressiert dies. Grundlegend verwaltend der `BluetoothConnectionManager` eine Liste aller aufgebauten Verbindungen (`BluetoothConnection`). Wird eine neue Verbindung aufgebaut, so wird diese dem `BluetoothConnectionManager` übergeben. Wird ein neuer Service gefunden, so überprüft die `BluetoothServiceConnectionEngine` mithilfe des `BluetoothConnectionManager`, ob eine entsprechende Verbindung bereits vorhanden ist (siehe Abbildung 13).

6.2.4 Bluetooth Connector Threads

Die Connector-Threads werden von der `BluetoothServiceConnectionEngine` immer dann gestartet, wenn entweder ein Service angeboten oder eine Verbindung zu einem Service aufgebaut werden soll.

6.2.4.1 BluetoothServerConnector

Der `BluetoothServerConnector` ist für das Bereitstellen von Services verantwortlich. Wenn die Engine einen Bluetooth-Service anbietet, wird ein `BluetoothServerConnector` gestartet. Dieser öffnet ein `BluetoothServerSocket`, registriert den Service so im lokalen SDP-Server und macht ihn somit für andere Peers auffindbar. Um ein `BluetoothServerSocket` zu öffnen, müssen ein Name und eine UUID angegeben werden. Dies geschieht mithilfe einer `ServiceDescription` entsprechend Code 4.

```
1    BluetoothServerSocket serverSocket =
2                 mBluetoothAdapter.listenUsingRfcommWithServiceRecord(
3                         description.getInstanceName(),
4                         description.getServiceUuid()
5                 );
```

Code 4 Offenen eines BluetoothServerSocket mit einer ServiceDescription

Jeder `BluetoothServiceConnector` ist mit genau einem Service assoziiert, sollten mehrere Services durch die `BluetoothServerConnector` angeboten werden, so wird für jeden dieser ein `BluetoothServerConnector` gestartet. Ein `BluetoothServerConnector` läuft so lange, bis der Service beendet wird.

6.2.4.2 BluetoothClientConnector

Ein `BluetoothClientConnector` wird immer dann gestartet, wenn ein Service im Rahmen einer Service-Discovery gefunden wurde und eine Verbindung aufgebaut werden soll. Dieser versucht es einmalig, eine Socket-Verbindung zu dem jeweiligen Gerät und Service aufzubauen und benachrichtigt die `BluetoothServiceConnectionEngine` über den Erfolg oder Misserfolg (siehe Abbildung 13).

6.2.5 Listener

Die `BluetoothServiceConnectionEngine` erlaubt es, mehrere Services anzubieten und/oder zu Suchen und Verbindungen zu diesen aufzubauen. Für jede Service-Discovery und für jeden Service, welcher durch die Engine angeboten wird, muss ein Listener angegeben werden. Dieser wird immer nur bezogen auf den von ihm registrierten Service benachrichtigt. Somit wird es erlaubt, auf Anwendungsseite mehrere verschiedene Services zu implementierten. Bei Bedarf kann aber auch eine einzige Implementierung des Interfaces für mehrere der registrieren Services angegeben werden. Die Interfaces, welche die Methoden vorgeben sind, `BluetoothServiceServer` für das Anbieten eines Services, und `BluetoothServiceClient` für die Suche eines Services.

```
1 public interface BluetoothServiceClient
2 {
3     void onServiceDiscovered(BluetoothDevice host, ServiceDescription description);
4
5     void onPeerDiscovered(BluetoothDevice peer);
6
7     void onConnectedToService(BluetoothConnection connection);
8
9     boolean shouldConnectTo(BluetoothDevice host, ServiceDescription description);
10 }
```

Code 5 Das BluetoothServiceClient Interface

```
1 public interface BluetoothServiceServer
2 {
3     void onClientConnected(BluetoothConnection connection);
4 }
```

Code 6 Das BluetoothServiceServer Interface

7 Tests

Für die Service-Discovery Komponente wurden Unit-Tests implementiert, welche das Verhalten einzelner Teilkomponenten mit gemockten APIs testen. Ebenfalls wurden Integration-Tests entwickelt, um die Funktionalität der Teilkomponenten zusammen mit den echten Android-APIs und auf physischen Geräten zu verifizieren. Darüber hinaus wurde eine Reihe von manuellen Tests durchgeführt, um sowohl die Geschwindigkeit als auch die Zuverlässigkeit der Service-Discovery zu untersuchen.

Im Folgenden wird auf die Integration- und Unit-Tests eingegangen und die jeweiligen Testkonzepte erläutert. Es soll dabei nicht auf jeden einzelnen Test im Detail eingegangen werden. Eine Beschreibung der Tests ergibt hier aus den Namen der Testmethoden, genauere Beschreibungen der Tests finden sich im Code. Abbildungen der wichtigen Unit-Testreihen können in A.2 gefunden werden.

7.1 Integration-Tests

Für die `BluetoothServiceDiscoveryEngine` (Variante 1 und 2), die `BluetoothServiceConnectionEngine` sowie für die `WifiDirectServiceDiscoveryEngine` wurden Integration-Tests implementiert.
Diese sollten die Komponente auf mehreren physischen Geräten, automatisiert testen und tatsächliche Service-Discoveries durchführen. Um das Verhalten der Komponente auf diese Weise zu testen, war es notwendig, verschiedene, aufeinander abgestimmte Testfälle parallel auf unterschiedlichen Geräten auszuführen.

Dazu wird jedem Gerät eine Rolle zugewiesen. Da die Geräte zur Laufzeit unabhängig voneinander agieren und keine Informationen über die anderen Geräte haben, muss die Rolle vor dem Ausführen der Tests bestimmt werden. Dadurch bedingt benötigen die Tests eine einmalige Konfiguration, in welcher der Gerätename in der Form `<Hersteller><Gerätemodell>` sowie die Bluetooth- und Wi-Fi Direct Mac-Adressen der verwendeten Geräte festgelegt werden. Zu diesem Zweck wurde die Klasse `DeviceRoleManager` implementiert.

Da die Tests auf verschiedenen Geräten laufen müssen und jeder Testfall von jedem Gerät eine spezifische Rolle verlangt, liegt ein großes Problem der Tests in der Synchronizität. Android Studio erlaubt es Code auf mehreren Geräten gleichzeitig auszuführen bzw. diesen zur selben Zeit zu starten. Es kann dabei jedoch zu Verzögerungen auf den einzelnen Geräten kommen, was unter Umständen zu Problemen führte. Diesem Synchronizitätsproblem konnten durch ein gezieltes Warten (`wait()`) entgegengewirkt werden.

Ein weiteres Problem besteht darin, dass Android eine Benutzerinteraktion benötigt, um ein Gerät für die Bluetooth Device-Discovery sichtbar zu machen. Dies scheint auch für Tests nicht vermeidbar zu sein. Ein 100-prozentig korrektes und reproduzierbares Testergebnis kann jedoch nicht garantiert werden, die Tests haben das Potenzial fehlzuschlagen. Ein Test kann durch falsches Timing oder eine nicht erfolgreiche Discovery aber auch durch einen tatsächlichen Fehler in der Komponente entstehen.

Die Tests können also als ein Indikator für Fehler in der Komponente gesehen werden, sollten dann aber zur Verifizierung dessen mehrfach ausgeführt werden und generell im Zusammenspiel mit den Unit-Tests und manuellen Tests genutzt werden.

Die `BluetoothServiceDiscovery` läuft in ihren beiden Varianten durch dieselbe Testreihe. In dieser übernimmt jeweils eines der drei Geräte die Rolle des Clients und führt eine Service-Discovery durch.

Die verbleibenden Geräte bieten eine variable Anzahl an verschiedenen Services an. Überprüft wird das Ergebnis auf der Seite des Clients.

Tests	Duration	Status	samsung SM-T580	DOOGEE Y8	LENOVO Lenovo TB...
Test Results	6 m 57 s	15/15	5/5	5/5	5/5
IntegrationBluetoothServiceDiscoveryVOne	6 m 57 s	15/15	5/5	5/5	5/5
itShouldFindNearbyDevice	42 s	Pass	✓	✓	✓
itShouldFindOneNearbyAvailableService	1 m 33 s	Pass	✓	✓	✓
itShouldFindTwoDifferentServicesOnOneDevice	1 m 33 s	Pass	✓	✓	✓
itShouldFindTwoNearbyAvailableService	1 m 33 s	Pass	✓	✓	✓
itShouldNotifiedAboutMatchingServicesAlreadyDiscov	1 m 35 s	Pass	✓	✓	✓

Abbildung 14 Integrationstests für die BluetoothServiceDiscovery

Für die `WifiDirectServiceDiscoveryEngine` ist eine weitere Testreihe implementiert worden, welche auf derselben Prämisse beruht wie die der `BluetoothServiceDiscoveryEngine`. Einer der Peers führt die Service-Discovery durch, während die anderen Services anbieten.

Tests	Duration	Status	LENOVO Lenovo TB...	DOOGEE Y8	samsung SM-T580
Test Results	4 m 4 s	12/12	4/4	4/4	4/4
IntegrationWifiDirectServiceDiscovery	4 m 4 s	12/12	4/4	4/4	4/4
itShouldFindOneNearbyService	1 m	Pass	✓	✓	✓
itShouldFindTwoNearbyServiceOnTwoDevices	1 m 3 s	Pass	✓	✓	✓
itShouldFindTwoNearbyServices	58 s	Pass	✓	✓	✓
itShouldNotifyAboutAllServices	1 m 3 s	Pass	✓	✓	✓

Abbildung 15 Integrationstests für die WifiDirectServiceDiscoveryEngine

Auch die `BluetoothServiceConnectionEngine` verfügt über eine Reihe von Integrationstests, welche überprüfen, ob Bluetooth-Verbindungen zwischen Services und Clients aufgebaut werden. Es wird hier jeweils die Client- und die Serverseite betrachtet, also ob ein Client Verbindungen korrekt aufbaut und ob der Server diese annimmt. Auch wird getestet, ob mehrere Verbindungen zur selben Zeit aufgebaut werden können. Die Testreihe erlaubt es, die `BluetoothServiceConnectionEngine` mit beiden Varianten der `BluetoothServiceDiscoveryEngine` zu testen.

Tests	Duration	Status	LENOVO Lenovo TB...	DOOGEE Y8	samsung SM-T580
Test Results	6 m	9/9	3/3	3/3	3/3
IntegrationBluetoothServiceConnectionEngineVTwo	6 m	9/9	3/3	3/3	3/3
itShouldAcceptConnectionsFromSeveralClients	2 m	Pass	✓	✓	✓
itShouldConnectToOneNearbyService	2 m	Pass	✓	✓	✓
itShouldMakeTwoConnectionsToTwoServicesOnTheSa	2 m	Pass	✓	✓	✓

Abbildung 16 Integrationstests für die BluetoothServiceConnectionEngine

7.2 Unit-Tests

Die Integration-Tests testen die grundlegende Funktionalität der Komponenten auf physischer Hardware und mit echten Android APIs. Die Unit-Tests dagegen führen detailliertere Tests der einzelnen Teilkomponenten und Klassen durch, wobei wichtige Android APIs gemockt werden. Dies geschieht mithilfe der Kotlin Mocking-Library *MOCKK*.

Das Konzept ist für alle vier Engines identisch: Notwendige Android-APIs / Klassen wie `Context`, `BluetoothAdapter` und `WiFiP2pManager` können beim Starten der Engines übergeben werden und werden durch Mock-Objekte ersetzt. Events, die von Android in Form von Broadcasts gesendet werden, können durch das Aufrufen der entsprechenden package-privaten Methoden auf den Engines simuliert werden. Um überprüfbare Ergebnisse zu erhalten, können Listener an den Engines registriert werden. Für die Connection Engines kann zusätzlich eine Mock-Implementierung der jeweiligen Discovery Engine beim Starten übergeben werden.

Es kann an dieser Stelle keine komplette Auflistung und Erklärung aller Unit-Tests erfolgen, weshalb hier auf die Abbildungen in A.2 sowie auf die Dokumentation im Source Code A.1 verwiesen werden soll.

7.3 Manuelle Tests

Zusätzlich zu den in den vorangegangenen Abschnitten beschrieben, automatisierten Tests, wurden eine Reihe von manuellem Test durchgeführt. Diese soll sowohl die Dauer als auch die Zuverlässigkeit der Service-Discovery ermitteln. Dazu wurden mithilfe der Demo-Anwendung eine Reihe von Service-Discoveries auf verschiedenen Geräten und mit einer unterschiedlichen Anzahl an (angebotenen und gesuchten) Services durchgeführt. Diese werden in den folgenden Abschnitten tabellarisch dargestellt. Die Tabellen zeigen für jeden durchgeführten Test die benötigte Zeit in Sekunden. Schlägt ein Test fehl, so wird dies mit *//Anzahl* vermerkt, wobei die Anzahl der dennoch gefunden Services genannt wird.

7.3.1 … für die Bluetooth Service-Discovery

Für die Tests wurde jeweils eines der Geräte zum Client, welcher eine Discovery (Device und Service) durchführt. Zwei der Geräte wurden zu Serviceanbietern und boten eine Anzahl von 1-4 Services (maximal 2 pro Gerät) an. Jeder Test wurde auf 3 Geräten durchgeführt. Die gesamte Testreihe wurde 4-mal wiederholt. Es wurde ein Time-out von 30 Sekunden (entsprechend den eingangs erhobenen Anforderungen) definiert. Wird diese Zeit überschritten, so gilt der Test als fehlgeschlagen.

Die Tests wurden für beide Varianten der `BluetoothServiceDiscoveryEngine` durchgeführt.

Wiederholungen	1			2			3			4		
Gerät	1	2	3	1	2	3	1	2	3	1	2	3
1 Service	20	22	23	14	15	19	16	19	16	17	14	16
2 Services	15	15	//0	//0	16	22	16	20	18	17	16	20
3 Services (2 Geräte)	21	20	19	23	17	17	20	23	22	21	20	26
4 Services (2 Geräte)	21	21	22	21	20	20	20	21	23	21	21	//0

Tabelle 4 Manuelle Tests der Bluetooth Service-Discovery (Variante 1)

Wiederholungen	1			2			3			4		
Gerät	1	2	3	1	2	3	1	2	3	1	2	3
1 Service	10	14	5	3	23	3	3	4	//0	3	//0	3
2 Services	3	11	3	4	8	//0	7	8	4	//0	12	//0
3 Services (2 Geräte)	7	11	//0	6	3	3	6	10	5	7	11	4
4 Services (2 Geräte)	9	//0	3	8	//2	3	7	5	8	18	13	3

Tabelle 5 Manuelle Tests der Bluetooth Service-Discovery (Variante 2)

Für die `BluetoothServiceConnectionEngine` wurde die Testreihe ebenfalls durchgeführt. Dabei wurde die Zeit vom Start der Discovery bis zum Aufbau der Bluetooth- Socketverbindung zu allen gesuchten und angebotenen Services gemessen. Es wurde hier die `BluetoothServiceDiscoveryVTwo` als Discovery Engine eingesetzt.

Wiederholungen	1			2			3			4		
Gerät	1	2	3	1	2	3	1	2	3	1	2	3
1 Service	7	5	13	11	8	9	//0	9	8	7	16	6
2 Services	10	//0	11	13	5	8	6	11	8	14	5	10
3 Services (2 Geräte)	7	17	//2	//1	16	14	15	14	20	16	8	10
4 Services (2 Geräte)	19	16	12	15	12	19	//2	13	18	//2	10	//0

Tabelle 6 Manuelle Tests des Bluetooth Verbindungsaufbaus

7.3.2 … für die mDNS-Service-Discovery

Für die Bonjour-Service-Discovery wurde ebenfalls eine Reihe manueller Tests durchgeführt. Es galt hier wieder eine Menge von 1-4 Services auf bis zu zwei anderen Geräten zu finden.

Der Zeitraum für jeden Test musste (entgegen den initialen Anforderungen) auf 1,5 Minuten gesetzt werden, um zu schnellen Time-outs vorzubeugen.

Wiederholungen	1			2			3			4		
Gerät	1	2	3	1	2	3	1	2	3	1	2	3
1 Service	7	4	7	7	4	3	5.2	//0	6	14	//0	8
2 Services	4	5	5	6	7	19	5	//0	6	41	4	4
3 Services (2 Geräte)	//2	//2	36	26	//2	14	8	//2	//1	6	//2	//1
4 Services (2 Geräte)	5	//2	10	//2	//2	//2	//2	//2	8	5	//1	27

Tabelle 7 Manuelle Tests der Wi-Fi Direct Service-Discovery

Es handelt sich bei dieser Auswertung um einen Ausschnitt, welcher jedoch geeignet ist, um einige der Probleme aufzuzeigen, welche bei der Bonjour Service-Discovery auftreten. Die `WifiDirectServiceDiscoveryEngine` hat das Potenzial, dieselbe Testreihe mit deutlich besseren, aber auch deutlich schlechteren Ergebnissen zu durchlaufen.

Da die `WiFiDirectServiceConnectionEngine` nur das Suchen und Anbieten eines einzelnen Services ermöglicht und darauf basierend eine Wi-Fi Direct-Group aufgebaut wird, wurde die Testreihe dahingehend angepasst. Es wurden jeweils 3 Geräte pro Test verwendet, diese sollten eine Wi-Fi Direct Gruppe bilden. Es wird für jeden Durchgang festgehalten, welches der Geräte zum Group Owner wurde und wie lange es für die beiden Clients dauerte, bis eine TCP-Socketverbindung zum Group Owner etabliert war. Fehler (es konnte keine Verbindung zum Group Owner aufgebaut werden) wurden mit einem // markiert.

Versuch	1	2	3	4	5	6	7	8	9	10	11	12
Gerät 1	GO	//	GO	//	GO	GO	GO	GO	22	25	17	18
Gerät 2	20	8	11	18	16	19	16	24	41	GO	GO	//
Gerät 3	//	GO	22	GO	16	21	//	45	GO	50	18	GO

Tabelle 8 Manuelle Tests des Wi-Fi Direct-Verbindungsaufbaus

8 Fazit

8.1 Zusammenfassung

Ziel dieser Arbeit war es, eine Service-Discovery für sowohl Bluetooth als auch Wi-Fi Direct unter Android zu entwickeln. Dies sollte in Form einer Softwarekomponente geschehen, welche von verschiedenen Anwendungen, in erster Linie aber von *ASAPAndroid* genutzt werden können sollte. Die Service-Discovery sollte es erlauben mehrere Services zuverlässig zu suchen und zu finden.

Dies konnte für Bluetooth (SDP) in zwei Varianten umgesetzt werden, welche sich sowohl in Belangen der Geschwindigkeit als auch der Zuverlässigkeit unterscheiden. Ihre Funktionalität konnte sowohl durch eine Reihe automatisierter Tests als auch durch manuelle Tests mit anschließender Auswertung festgehalten werden.

Die Service-Discovery für Wi-Fi Direct wurde mit Bonjour umgesetzt. Sie erlaubt es, mehrere Services anzubieten und zu suchen. Manuelle Tests zeigten, dass die Service-Discovery die initial erhobenen Anforderungen sowohl an die Geschwindigkeit als auch die Zuverlässigkeit nicht durchgehend erfüllen kann. Eine vollumfängliche Lösung für dieses Problem konnte im Rahmen dieser Arbeit und basierend auf den von Android bereitgestellten APIs nicht umgesetzt werden. Wie jedoch durch die weiterführenden Tests zum Verbindungsaufbau gezeigt werden kann, funktioniert sie für diese Zwecke dennoch hinreichend zuverlässig.

Es wurde ebenfalls eine Möglichkeit zur Beschreibung von Services gefunden. Diese wurde mit der Klasse `ServiceDescription` implementiert, welche sowohl mit den DNS-SD Records kompatibel ist als auch von der `BluetoothServiceDiscoveryEngine` genutzt werden kann, um eine UUID zu generieren, welche für Bluetooth SDP benötigt wird.

Darauf aufbauend wurde die Komponente um einen Verbindungsaufbau erweitert, welcher zeigt, dass die Service-Discovery alle für den Aufbau einer P2P-Verbindung notwendigen Informationen bereitstellt sowie als ein Prototyp für die zukünftige Erweiterung der Komponente gesehen werden kann. Diese erlaubt für Bluetooth das gleichzeitige Aufbauen von mehreren Bluetooth-Socketverbindungen zu mehreren anderen Peers. Für Wi-Fi Direct wird es ermöglicht, eine Wi-Fi Direct Group und TCP-Socketverbindungen zwischen den Clients und dem Group Owner aufzubauen. Die Funktionalität dieser konnte ebenfalls durch automatisierte Tests und eine Reihe manueller Tests überprüft werden.

8.2 Ergebnisbewertung

Dieser Abschnitt soll die Ergebnisse einzelner Teilaspekte dieser Arbeit zusammenfassend bewerten. Für die Discovery- und Connection Engines sollen die in 7.3 aufgeführten manuellen Tests als Grundlage der Bewertung dienen.

8.2.1 … für die Bluetooth Service-Discovery

Es kann basierend auf den manuell durchgeführten Tests (Tabelle 4) festgestellt werden, dass die Bluetooth Service-Discovery erfolgreich umgesetzt werden konnte. Services können mit der ersten Variante der `BluetoothServiceDiscoveryEngine` in durchschnittlich circa 19 Sekunden gefunden werden. Dabei wurden 3 Fehlschläge (von insgesamt 48 Versuchen) der Service-Discovery vermerkt, in denen Services nicht in der vorgegebenen Zeit gefunden werden konnten.

Die zweite Variante der Bluetooth Service-Discovery macht es möglich, Services durchschnittlich schneller zu finden, wobei hier ein Wert von circa 7 Sekunden ermittelt werden konnte. Die in Tabelle 5 deutlich zu erkennen Schwankungen in der benötigten Zeit finden ihre Ursache in der Anzahl der, sich in Reichweite befinden Geräte und in welcher Reihenfolge diese gefunden werden. Beispielsweise kann der gesuchte Service bei 5 verfügbaren Geräten schon im ersten Durchgang gefunden werden, was zu einer sehr schnellen Discovery führt. Er kann aber auch erst im fünften Durchlauf gefunden werden, die Discovery muss also 5-mal durchgeführt und angehalten werden, bevor der Service gefunden wurde, was dann zu einer deutlich längeren, benötigten Zeit führt.

Auch scheint das häufige Neustarten der Discovery zu Problemen zu führen, welche bewirken, dass die Device-Discovery nicht gestartet werden kann. In diesen Fällen stoppt dann die gesamte (Service und Device) Discovery und es werden keine weiteren Services gefunden, bis diese manuell neu gestartet wurde. Dies führte in dieser Version zu einer erhöhten Anzahl an Fehlschlägen (8 von 48 und somit über der in den Anforderungen definierten Fehlerquote von 10 %) festgestellt werden.

Die initial erhobenen Anforderungen können von der Bluetooth Service-Discovery somit weitestgehend erfüllt werden. Sowohl die nicht-funktionalen Anforderungen an Zuverlässigkeit und an die benötigte Zeit für die Discovery als auch die funktionalen Anforderungen wie das Suchen mehrerer Services, konnten umgesetzt werden. Das Anbieten eines Services wurde hier bewusst ausgelassen (siehe 5.3.4)

8.2.2 …für die Wi-Fi Direct Service-Discovery

Die Service-Discovery für Wi-Fi Direct kann die Anforderungen weder im Punkte der Zuverlässigkeit noch in Belangen der Geschwindigkeit durchgehend erfüllen. Diese Aussage kann auf Basis der in Tabelle 7 festhaltenden Daten getroffen werden. Services können nicht immer in der vorgegebenen Zeit von maximal 30 Sekunden gefunden werden. Es ergab sich dabei ein Höchstwert von 41 Sekunden.

Auch konnten insgesamt 17 Fehlschläge der Discovery festgehalten werden, womit die ursprünglich auf maximal 10 % festgelegte Fehlerquote mit ~35,42 % deutlich überschritten wird.

Es können hier die folgenden, häufig auftretenden Probleme festgestellt werden. Dabei wird sich auch auf die in 5.4.4 gelisteten Untersuchungen bezogen.

- Es wird nur ein Teil der angebotenen Services gefunden, dabei werden dann meist nur Services von einem der anbietenden Geräte gefunden.
- Zeitweise kann ein Gerät keine Services auf einem der anderen Geräte finden.
  Dies lässt sich in Tabelle 7 anhand von Gerät 2 gut nachvollziehen. In den ersten beiden Wiederholungen konnte es jeweils die Services eines der beiden anderen Geräte finden. In den darauffolgenden 2 Wiederholungen waren die beiden anderen Geräte (Server) in ihren Rollen vertauscht, was dann zu einer Umkehrung der Testergebnisse führte.
- Es kann dazu kommen, dass in einem längeren Zeitraum keine der angebotenen Services gefunden werden können.

Dies war bei näherer Untersuchung (siehe 5.4.4) oft auf eine nicht erfolgreiche Device-Discovery zurückzuführen. Es wurde auch beobachtet, dass die Device- und Service-Discovery frühzeitig beendet wurden.

Eine vollständige Lösung für diese Probleme konnte im Verlauf dieser Arbeit nicht gefunden werden. Vergleicht man dies jedoch mit anderen Anwendungen, die die APIs nutzen, zeigen sich dort ebenfalls diese Probleme. Hier ist insbesondere die von Android bereitgestellte Referenzimplementierung für die Wi-Fi Direct Service-Discovery [35] zu nennen. Basierend darauf kann davon ausgegangen werden, dass die hier festgehaltenen Probleme auf die hier genutzten APIs von Android zurückzuführen sind.

8.2.3 …für die Teilkomponente zum Aufbau von Verbindungen

Eine automatisierter Verbindungsaufbau konnte, sowohl für Bluetooth als auch Wi-Fi Direct prototypisch realisiert werden.

Der Bluetooth-Verbindungsaufbau kann mit beiden Varianten der Service-Discovery umgehen und ist vergleichbar in Zuverlässigkeit und Geschwindigkeit. Exemplarisch konnte dies in

Tabelle 6 gezeigt werden. Hier wurde die zweite Variante der `BluetoothServiceDiscoveryEngine` verwendet, um Peers zu finden. Es wurden durchschnittlich circa 11,5 Sekunden gebraucht, um alle Services in Reichweite zu finden und um sich mit diesen zu verbinden. Was im Vergleich zur reinen Service-Discovery (circa 7 Sekunden) einen zusätzlichen Zeitaufwand von durchschnittlich circa 4,5 Sekunden bedeutet. Der Verbindungsaufbau erfolgt zuverlässig. Es kam insgesamt zu 7 Fehlschlägen von insgesamt 48 Versuchen. Wobei diese hauptsächlich auf die verwendete `BluetoothServiceDiscoveryEngineVTwo` zurückzuführen sind.

Entsprechend der Zuverlässigkeit der Wi-Fi Service-Discovery konnte die `WifiDirectConnectionEngine` nicht den Anforderungen entsprechend umgesetzt werden. Funktioniert aber, sollte die Service-Discovery Geräte finden zuverlässig darin, eine Wi-Fi Direct Group und eine TCP-Socketverbindung zwischen GO und Client zu etablieren. Dies konnte in den in Tabelle 8 festgehaltenen, manuellen Tests erwiesen werden. Dabei zeigte sich, dass, verglichen mit den Ergebnissen der reinen Service-Discovery mit denen der `WifiDirectServiceConnectionEngine` zuverlässigere Ergebnisse prodoziert werden können. Dies ist hauptsächlich darauf zurückzuführen, dass es in vielen Fällen ausreicht, wenn eines der Geräte das andere findet und nicht, wie für die Service-Discovery getestet, jedes Gerät jeden der anderen Services / Geräte.

8.2.4 …für die Servicebeschreibungen

Zum finalen Stand dieser Arbeit erfüllen die Servicebeschreibungen, implementiert durch die Klasse `ServiceDescription`, die gegebenen Anforderungen. Sie erlauben es Services sowohl für Bluetooth als auch für Wi-Fi Direct anzubieten und nach diesen zu suchen.

Der initiale Ansatz der `ServiceDescription` war es, die zur Verfügung stehenden DNS-SD Record und die für Bluetooth verwendete UUID zu vereinheitlichen. So eine, für den Nutzer, einheitliche und somit einfachere Schnittstelle für sowohl die Bluetooth- und die Wi-Fi Direct Service-Discovery geboten werden.

In Retrospektive lässt sich aber sagen, dass, während diese Ziele prinzipiell erfüllt wurden, der diesen zugrundeliegende Ansatz falsch war. Services lassen sich für Bluetooth SDP ausschließlich durch eine UUID beschreiben, ein Transport von Informationen kann nur durch diese stattfinden. Die `ServiceDescriptions` basieren darauf einen Teil ihrer Informationen auf eine UUID abzubilden und sollen somit eine Verknüpfung zwischen UUID und weiteren Informationen ermöglichen. Für Wi-Fi Direct werden jedoch alle Informationen für Clients bereitgestellt. Dieser Unterschied in der Nutzung der durch eine `ServiceDescription` bereitgestellten Informationen, sowohl bei der Service-Discovery selbst als auch bei der übertragenen Informationsmenge, führt zu einem, für den Benutzer der Komponente eventuell unerwarteten Verhalten.

So wird beispielsweise der *Instance Name* bei der Bluetooth Service-Discovery immer der sein, welcher der Engine eingangs, beim Registrieren eines Services für die Discovery in Form der `ServiceDescription` übergeben wurde. Der, vom entfernten Peer verwendete *Instance Name* kann jedoch von diesem verschieden sein.

Ebenso findet eine Service-Discovery immer nur auf Basis des, von der `ServiceDescription` gegebenen, *Service Type* (oder der daraus generierte UUID) statt. Wird also eine `ServiceDescription` für die Suche registriert, so werden sowohl *Instance Name* als auch TXT Record ignoriert. Dies gilt für die `WifiDirect-` ebenso wie für die `BluetoothServiceDiscoveryEngine`.

Die `ServiceDescription` ermöglicht es somit zwar die Schnittstelle für die Service-Discovery zu vereinheitlichen, führen dadurch aber ebenfalls dazu, dass diese für den Nutzer der Komponente, schwerer verständlich ist und ein unerwartetes Verhalten zeigen kann.

Diesen Fehlschluss gilt es, in einer zukünftigen Version der Komponente zu revidieren.

8.2.5 Gerätespezifische Probleme

Im Verlauf dieser Arbeit konnte immer wieder festgestellt werden, dass es zu gerätespezifischen Verhaltensweisen und damit einhergehend zu Fehlern kommen kann. Dies bezieht sich im Speziellen auf die Bluetooth-Service-Discovery. So wurde in Kapitel 5.3.5 bereits auf das Problem mit den Little-Endian UUIDs eingegangen und ein Workaround implementiert. Dieses Problem trat ausschließlich auf einem der hier insgesamt 4 verwendeten Testgeräte auf. Es konnte dabei ein Zusammenhang mit der Android-Version ausgeschlossen werden, da ein weiteres Gerät mit derselben Android-Version dieses Verhalten nicht zeigte.

Auch konnte festgestellt werden, dass zwei der genutzten Geräte (hier A und B) nur eine unvollständige Liste an Service UUIDs austauschten. Sollte B eine Service-Discovery auf A durchführen so erhält B ausnahmslos nur die von Android selbst angebotenen Service-UUIDs. Die durch die Anwendung registrierten UUIDs werden jedoch nicht gefunden. Dabei handelt es sich um ein Unidirektionales Problem, A erhält zuverlässig die komplette Liste an registrierten Services von B. Mit allen anderen verwendeten Geräten funktionieren sowohl Gerät A und Gerät B korrekt.

Diese gerätespezifischen Probleme konnten hier aufgrund der beschränkten Anzahl an Testgeräten nicht final eingeschätzt werden und sollten in Zukunft weiter beobachtet werden.

8.3 Ausblick

Die in dieser Arbeit entwickelte Komponente zeigt, wie eine Service-Discovery und ein darauf basierender Aufbau von P2P-Verbindungen für Android Geräte, mit den von Android bereitgestellten APIs umgesetzt werden kann. In diesem Abschnitt soll auf die, auf der hier geschaffenen Grundlage basierende Weiterentwicklung dieser Komponente eingegangen werden. Dabei soll ebenso auf weitere Funktionen sowie auf Möglichkeiten eingegangen werden, um die im vorangegangenen Kapitel beschriebenen Limitationen der aktuellen Implementierung zu beseitigen.

Eine Integration dieser Komponente in *ASAPAndroid* soll nach Abschluss dieser Arbeit erfolgen. Dabei sollten die beschrieben Probleme der Wi-Fi Direct Service-Discovery beachtet werden und, wenn nötig, eine neue Implementierung dieser in Betracht gezogen werden. Hierfür sollte auf eine Möglichkeit abseits der, von Android bereitgestellten APIs gesetzt werden. Dies gilt im gleichen Maße für die Komponente als eigenständige Einheit. Initiale Recherchen führten hier zu der Java Library jmDNS, welche

mDNS (-SD) in Java implementiert. Es muss hier eine Kompatibilität mit der Android Wi-Fi Direct API untersucht werden.

Weiterhin sollte die `ServiceDescription` mit Hinsicht auf die Ausführungen in 8.2.4 überarbeitet werden. Dafür ist es geplant zwei, voneinander Getrennte, jedoch auf einer Superklasse beruhenden `ServiceDescriptions` zu implementieren. Dies soll dem Trugschluss vorbeugen, ihre Funktionalität wäre für sowohl für die Bluetooth- als auch für Bonjour-Service-Discovery identisch. Der *Service Type* sollte dann ebenfalls als eine eigene, dieser `ServiceDescription` untergeordnete Klasse implementiert werden, welche das Suchen nach Services verwendet wird. Damit sollt ebenfalls potentiellen Verwirrungen über die Bedeutung des *Instance Name* und des TXT-Records in der Servicesuche vermieden werden.

Es ist auch davon auszugehen das sich bei der Nutzung der Komponente in Anwendungsfällen abseits des Testens und der, wenig komplexen, Demo-Anwendung weitere Verbesserungsmöglichkeiten herausstellen werden. Hierbei kann angenommen werden, dass die aktuelle Implementierung zu wenig Feedback im Falle von Fehlern gibt. Scheitert beispielsweise der Versuch eine Verbindung in der `BluetoothServiceConnectionEngine` aufzubauen, so sollte dies dem jeweiligen `BluetoothServiceClient` mitgeteilt werden. Ebenfalls sollten Möglichkeiten geschaffen werden, um auf diese Fehler zu reagieren (beispielsweise ein manuelles Starten eines Verbindungsversuches). Ein funktionaler Ausbau der Komponente in diesen Belangen ist für viele Anwendungsfälle essentiell und sollte mit großer Priorität verfolgt werden.

Quellenverzeichnis

[1] T. Schwotzer, „SharedKnowledge/ASAPJava Wiki“, *Home · SharedKnowledge/ASAPJava Wiki*. https://github.com/SharedKnowledge/ASAPJava/wiki (zugegriffen 1. Juli 2022).

[2] C. Jayapal, S. Jayavel, und V. P. Sumathi, „Enhanced Service-Discovery Protocol for MANET by Effective Cache Management“, *Wirel. Pers. Commun.*, Bd. 103, Nr. 2, S. 1517–1533, Nov. 2018, doi: 10.1007/s11277-018-5866-3.

[3] R. Marin-Perianu, P. Hartel, und H. Scholten, „A Classification of Service-Discovery Protocols“, S. 22.

[4] Karsten Renhak, „Service-Discovery-Protokolle“. 4. Februar 2007.

[5] M. D. Day, C. E. Perkins, J. Veizades, und E. Guttman, „Service Location Protocol, Version 2“, Internet Engineering Task Force, Request for Comments RFC 2608, Juni 1999. doi: 10.17487/RFC2608.

[6] „Bluetooth Core Specification v5.2“. Bluetooth SIG, 31. Dezember 2019.

[7] D. Camps-Mur, A. Garcia-Saavedra, und P. Serrano, „Device-to-device communications with Wi-Fi Direct: overview and experimentation“, *IEEE Wirel. Commun.*, Bd. 20, Nr. 3, S. 96–104, Juni 2013, doi: 10.1109/MWC.2013.6549288.

[8] J. H. Lee, M.-S. Park, und S. C. Shah, „Wi-Fi direct based mobile ad hoc network“, in *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, Juli 2017, S. 116–120. doi: 10.1109/CCOMS.2017.8075279.

[9] „Wi-Fi CERTIFIED Wi-Fi Direct“. Wi-Fi Alliance, 2010. [Online]. Verfügbar unter: https://www.wi-fi.org/system/files/wp_Wi-Fi_Direct_20101025_Industry.pdf

[10] „Bonjour Concepts“. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Articles/about.html#//apple_ref/doc/uid/TP40002458-SW1 (zugegriffen 22. November 2022).

[11] S. Cheshire und M. Krochmal, „Multicast DNS“, Internet Engineering Task Force, Request for Comments RFC 6762, Feb. 2013. doi: 10.17487/RFC6762.

[12] S. Cheshire und M. Krochmal, „DNS-Based Service Discovery“, Internet Engineering Task Force, Request for Comments RFC 6763, Feb. 2013. doi: 10.17487/RFC6763.

[13] „Bonjour Operations“. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Articles/NetServicesArchitecture.html#//apple_ref/doc/uid/20001074-SW1 (zugegriffen 27. November 2022).

[14] A. Gulbrandsen und L. Esibov, „A DNS RR for specifying the location of services (DNS SRV)“, Internet Engineering Task Force, Request for Comments RFC 2782, Feb. 2000. doi: 10.17487/RFC2782.

[15] „BluetoothAdapter“, *Android Developers*. https://developer.android.com/reference/android/bluetooth/BluetoothAdapter (zugegriffen 22. Oktober 2022).

[16] „BluetoothDevice“, *Android Developers*. https://developer.android.com/reference/android/bluetooth/BluetoothDevice (zugegriffen 22. Oktober 2022).

[17] „BluetoothAdapter#listenUsingRfcommWithServiceRecord“, *Android Developers*. https://developer.android.com/reference/android/bluetooth/BluetoothAdapter#listenUsingRfcommWithServiceRecord(java.lang.String,%20java.util.UUID) (zugegriffen 11. Dezember 2022).

[18] „BluetoothDevice#getUuids()“, *Android Developers*. https://developer.android.com/reference/android/bluetooth/BluetoothDevice#getUuids() (zugegriffen 22. Oktober 2022).

[19] „BluetoothDevice.java“, *android.googlesource*. https://android.googlesource.com/platform/prebuilts/fullsdk/sources/android-30/+/refs/heads/master/android/bluetooth/BluetoothDevice.java (zugegriffen 29. August 2022).

[20] „WifiP2pManager“, *Android Developers*. https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager (zugegriffen 8. August 2022).

[21] „WifiP2pDevice“, *Android Developers*. https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pDevice (zugegriffen 12. Dezember 2022).

[22] „WifiP2pServiceInfo“, *Android Developers*. https://developer.android.com/reference/android/net/wifi/p2p/nsd/WifiP2pServiceInfo (zugegriffen 12. Dezember 2022).

[23] „WifiP2pDnsSdServiceInfo | Android Developers“. https://developer.android.com/reference/android/net/wifi/p2p/nsd/WifiP2pDnsSdServiceInfo (zugegriffen 12. Dezember 2022).

[24] „WifiP2pDnsSdServiceRequest“, *Android Developers*. https://developer.android.com/reference/android/net/wifi/p2p/nsd/WifiP2pDnsSdServiceRequest (zugegriffen 12. Dezember 2022).

[25] „WifiP2pManager.DnsSdServiceResponseListener“, *Android Developers*. https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.DnsSdServiceResponseListener (zugegriffen 12. Dezember 2022).

[26] „WifiP2pManager.DnsSdTxtRecordListener“, *Android Developers*. https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.DnsSdTxtRecordListener (zugegriffen 12. Dezember 2022).

[27] „Use Wi-Fi Direct (P2P) for service discovery“, *Android Developers*. https://developer.android.com/training/connect-devices-wirelessly/nsd-wifi-direct (zugegriffen 25. November 2022).

[28] P. J. Leach, R. Salz, und M. H. Mealling, „A Universally Unique IDentifier (UUID) URN Namespace“, Internet Engineering Task Force, Request for Comments RFC 4122, Juli 2005. doi: 10.17487/RFC4122.

[29] J. F. Dooley, *Software Development, Design and Coding*. Berkeley, CA: Apress, 2017. doi: 10.1007/978-1-4842-3153-1.

[30] E. Gemma, R. Helm, R. Johnson, und J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 37. Aufl. Addison-Wesley.

[31] „LiveData overview“, *Android Developers*. https://developer.android.com/topic/libraries/architecture/livedata (zugegriffen 3. September 2022).

[32] S. Cheshire und M. Krochmal, „DNS-Based Service Discovery“, Internet Engineering Task Force, Request for Comments RFC 6763, Feb. 2013. doi: 10.17487/RFC6763.

[33] „Permissions on Android“, *Android Developers*. https://developer.android.com/guide/topics/permissions/overview (zugegriffen 29. November 2022).

[34] „Android 6.0.1 RFCOMM service UUID retrieve error [37075233]“. https://issuetracker.google.com/issues/37075233 (zugegriffen 6. September 2022).

[35] „WiFiDirectServiceDiscovery Sample“, *Git at Google*. https://android.googlesource.com/platform/development/+/refs/heads/master/samples/WiFiDirectServiceDiscovery/src/com/example/android/wifidirect/discovery (zugegriffen 6. Dezember 2022).

Abkürzungsverzeichnis

SDP	Bluetooth Service Discovery Protokoll
mDNS	Multicast DNS
DNS-SD	DNS-Service Discovery
GO	(Wi-Fi Direct) Group Owner
API	Application Programming Interface

A. Appendix

A.1 Source Code

Der im Rahmen dieser Arbeit entstandene Code kann auf GitHub gefunden werden:
WilliBoelke/android-service-discovery

A.2 Unit-Tests

A.2.1 BluetoothConnectionEngine

Tests	Duration	Pixel_4_XL_API_28
Test Results	25 s	16/16
UnitBluetoothServiceConnectionEngine	25 s	16/16
itShouldHandleIoExceptionsWhenTryingToConnect	2 s	✓
itShouldNotifyAboutAServiceDiscovery	1 ms	✓
itShouldNotStartTheSameServiceTwice	5 s	✓
itShouldAcceptConnectionsWhenServiceStarted	5 s	✓
itShouldCloseTheServerSocketWhenEndingTheService	2 s	✓
itShouldBeAbleToRunSeveralServicesAtTheSametime	5 s	✓
itShouldNotifySeveralClientsAboutDiscoveredDevices	1 ms	✓
itShouldStartDiscoveryEngine	1 ms	✓
iShouldNotCrashOnAServerSocketNullPointerException	2 s	✓
itShouldTryToConnectToService	94 ms	✓
itShouldTryToConnectToSeveralServices	1 s	✓
itShouldUnregisterServiceFromDiscovery	0 ms	✓
itShouldRegisterServiceForDiscovery	88 ms	✓
itShouldNotifyWhenAPeerWasFound	0 ms	✓
itShouldNotifyServerOnCreatedConnection	1 s	✓
itShouldOnlyNotifyTheRightClientAboutItsServices	93 ms	✓

A.2.2 BluetoothServiceDiscoveryEngineVTwo

Tests	Duration	Pixel_4_XL_API_28
Test Results	3 s	12/12
UnitBluetoothServiceDiscoveryVTwo	3 s	12/12
itShouldHandleUuidArraysBeingNull	3 s	✓
itFetchesUuidsOfAllDiscoveredDevicesAfterDeviceDiscoveryFinished	70 ms	✓
itShouldCheckLittleEndianUuids	1 ms	✓
itFindsServices	82 ms	✓
itShouldNotCrashIfNotStarted	0 ms	✓
itNotifiesAboutEveryDiscoveredPeer	0 ms	✓
itShouldNotifyAllListener	87 ms	✓
itShouldNotifyAboutServicesThatWhereDiscoveredBefore	93 ms	✓
itShouldBeAbleToSearchForSeveralServicesAtATime	94 ms	✓
itShouldFetchUuidsWhenRefreshStarted	91 ms	✓
theEngineShouldNotCrashIfTheBluetoothAdapterIsNull	0 ms	✓
itShouldPauseTheDiscoveryWhenRefreshingServices	0 ms	✓

A.2.3 BluetoothServiceDiscoveryEngineVOne

Tests	Duration	Pixel_4_XL_API_28
Test Results	2 s	12/12
UnitBluetoothServiceDiscoveryVOne	2 s	12/12
itShouldHandleUuidArraysBeingNull	2 s	✓
itFetchesUuidsOfAllDiscoveredDevicesAfterDeviceDiscoveryFinished	0 ms	✓
itShouldCheckLittleEndianUuids	0 ms	✓
itFindsServices	94 ms	✓
itShouldNotCrashIfNotStarted	0 ms	✓
itNotifiesAboutEveryDiscoveredPeer	0 ms	✓
itShouldNotifyAllListener	94 ms	✓
itShouldNotifyAboutServicesThatWhereDiscoveredBefore	95 ms	✓
itShouldBeAbleToSearchForSeveralServicesAtATime	0 ms	✓
itShouldFetchUuidsWhenRefreshStarted	89 ms	✓
theEngineShouldNotCrashIfTheBluetoothAdapterIsNull	0 ms	✓
itShouldPauseTheDiscoveryWhenRefreshingServices	0 ms	✓

A.2.3 WifiDirectServiceConnectionEngine

Tests	Duration	Pixel_4_XL_API_28
Test Results	4 s	11/11
UnitWifiDirectServiceConnectionEngine	4 s	11/11
itShouldTryToConnect	3 s	✓
itShouldNotifyWhenAConnectionHasBeenCreated	191 ms	✓
itShouldNotifyAboutBecomingAClient	94 ms	✓
itShouldNotifyAboutBecomingGroupOwner	0 ms	✓
itShouldNotifyWhenAServiceWasFound	90 ms	✓
itShouldNotStart	92 ms	✓
itShouldNotifyWhenSeveralServiceWereFound	93 ms	✓
itShouldTryToConnectToSeveralServices	93 ms	✓
itShouldRegisterServiceForDiscovery	0 ms	✓
itShouldNotRegisterASecondService	87 ms	✓
theClientCanDeclineTheConnection	0 ms	✓

A.2.4 WifiDirectServiceDiscoveryEngine

Tests	Duration	Pixel_4_XL_API_28
Test Results	4 s	16/16
UnitWifiDirectServiceDiscovery	4 s	16/16
theServiceRequestsShouldBeCleared	3 s	✓
itShouldStopNotifyingWhenDiscoveryForServiceStopped	0 ms	✓
itShouldNotifyAboutAllServices	86 ms	✓
itShouldNotifyAboutSeveralServiceDiscoveries	94 ms	✓
itShouldNotifyWhenServiceFound	92 ms	✓
itShouldIgnoreIfNotStarted	90 ms	✓
itShouldAllowListenersToUnregister	1 ms	✓
itShouldDiscoverDifferentInstances	87 ms	✓
itShouldNotStart	94 ms	✓
itShouldNotNotifyTwiceAboutAboutTheSameServiceAndHost	0 ms	✓
itShouldNotifyAllListener	79 ms	✓
itShouldSearchForSeveralServices	95 ms	✓
verifyDiscoveryApiUsage	0 ms	✓
itShouldOnlyStopTheCorrectServiceDiscovery	90 ms	✓
itShouldNotNotifyWhenServiceIsNotSearched	0 ms	✓
itShouldNotifyAgainInNewDiscoveryRun	87 ms	✓

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

13.12.2022, Berlin, Willi Bölke