Technology
Arts Sciences
TH Köln

Evaluation von Bluetooth, NFC und USB im Rahmen von PharoThings und Android als p2p Verbindung

Bachelorarbeit Informatik

vorgelegt von **Jan Phillip Kretzschmar**

Erstprüfer: **Prof. Dr. Christian Kohls**

Zweitprüfer: **Anton Borries**
(Zweidenker GmbH)

August 2019

ADRESSEN

Jan Phillip Kretzschmar
Halzenberg 43
42929 Wermelskirchen

Prof. Dr. Christian Kohls
Campus Gummersbach
Steinmüllerallee 1
51643 Gummersbach

Anton Borries
Charlottenstraße 47
40210 Düsseldorf

Zweidenker GmbH
Luxemburger Straße 72
50674 Köln

Gliederung

Liste der Algorithmen

Liste der Diagramme

1 Einleitung

In der Firma *Zweidenker GmbH* soll ein Internet of Things System auf Grundlage von PharoThings [PharoThings] geschaffen werden, um Sensoren und Aktoren in einem Netzwerk oder zu einem Server verfügbar zu machen. Im Internet of Things (IoT) werden Geräte miteinander vernetzt, die nicht zwingend eine Internetanbindung benötigen würden, jedoch erweiterte Funktionalitäten der Geräte dem Nutzer so zur Verfügung gestellt werden. Um solchen Geräten eine Internetanbindung geben zu können, ist es nötig, eine kabelgebundene Verbindung zu schaffen oder die Geräte in ein kabelloses Wi-Fi Netzwerk aufzunehmen. Die Verbindungskonfiguration von IoT Geräten für Wi-Fi Netzwerke wurde bereits auf Basis von Wi-Fi Direct untersucht [Kretzschmar]. Da die Implementierung dieser Technologie einige Unzulänglichkeiten besitzt, soll nun ein Vergleich von weiteren p2p Technologien vorgenommen werden. Hierunter fallen Bluetooth, Near Field Communication (NFC) und Universal Serial Bus (USB), da diese alle von Android Smartphones nativ unterstützt werden. Da die Vor- und Nachteile sowie Funktionsweisen dieser p2p Schnittstellen bereits bekannt sind, sollen sie im Rahmen dieser Arbeit in die Verbindungskonfiguration mit PharoThings eingebunden werden, um auch hier Probleme aufzudecken und zu einer optimalen Lösung des Problems zu gelangen.

Die existierende Lösung besteht aus einer Service Discovery und einer p2p Verbindung auf Basis von Wi-Fi Direct. Die Service Discovery wird als bestehend beibehalten, um die Evaluation auf die Nutzbarkeit der p2p Verbindung zu konzentrieren. Ziel soll sein, eine Alternative p2p Technologie zu Wi-Fi Direct zu finden.

1.1 Zielsetzung

Im vorausgegangenen Praxisprojekt wurde deutlich, dass Wi-Fi Direct einige gravierende Mängel im Hinblick auf die Funktionalität und Robustheit der bereitgestellten Bibliotheken aufweist. So ist die Dokumentation der meisten Nachrichten und Events unvollständig oder fehlt, zudem ist unklar wie eine Verbindung aufgebaut werden muss. Gleichzeitig sind Fehlermeldungen entweder nicht aussagekräftig oder führen dazu, dass der laufende Daemon abstürzt. Daraus lassen sich für diese Evaluation zwei Kriterien im Bereich der Robustheit ableiten. Zunächst kann getestet werden wie aussagekräftig Fehlercodes oder Fehlermeldungen sind. Das eine Ende des Spektrums bildet hierbei die Darstellung des Erfolges eines Aufrufs durch eine boolesche Variable, auf der anderen Seite wird ein genauer Fehlercode in Verbindung mit einer Fehlermeldung ausgegeben. Die Fehlercodes sind dabei für jede genutzte Methode dokumentiert und bieten Aufschluss auf die Gründe der erhaltenen Fehlschläge. Gleichzeitig soll evaluiert werden, inwieweit sich von aufgetretenen Fehlern erholt werden kann, um dennoch eine erfolgreiche Verbindung aufbauen zu können. Hierbei wird der messbare Bereich auf der einen Seite durch den Absturz des verwendeten Moduls beschränkt. Es besteht somit keine Möglichkeit zur Erholung von Fehlern und es muss darüber hinaus sogar das verwendete Modul neu gestartet werden. Dem gegenüber steht die gute Dokumentation von Fehlercodes, welche es erlaubt, auf nicht kritische Fehler reagieren zu können. Ebenso können die Fehler bereits

einen Codeblock beinhalten, der es erlaubt, den Fehler, so fern dieser unerwünscht gewesen ist, zu beseitigen. Ebenso sollte die Anbindung einer eingesetzten Technologie testbar sein. Die Testbarkeit der Software ergibt sich zum einen aus einem hohen Abstraktionsgrad, so dass sich die genutzten Technologien mit wenig Aufwand durch Mocks und Stubs auswechseln lassen. Ebenso ist eine Testbarkeit erst dann gegeben, wenn die genutzt Technologie eine hohe Stabilität im Hinblick auf Wiederholbarkeit bietet.

Die Funktionalität von Wi-Fi Direct wird aktuell durch mehrere Faktoren eingeschränkt. Zunächst mangelt es an einem internen Zustandsdiagramm des genutzten Moduls. Dadurch ist es nicht möglich, festzustellen, welche Methoden in einer bestimmten Reihenfolge aufgerufen werden müssen, um den erwünschten Zustand einer Verbindung zu erreichen. Zudem sind hierbei interne Zustandsübergänge durch ankommende Events oder weitere unbekannte Gründe nicht dokumentiert. Dies führt dazu, dass nicht klar ist, wann Methoden aufgerufen werden müssen um den aktuellen Zustand der p2p Schnittstelle beizubehalten. Als Evaluationskriterium kann hierbei wieder die Dokumentation im Hinblick auf die Funktionalität dienen, denn es kann festgestellt werden, wie gut der Verbindungszustand und Modulzustand dokumentiert sind. Es besteht zum Einen die Möglichkeit, dass keine Dokumentation vorliegt und das genutzte Modul lediglich als Blackbox genutzt werden kann. Als Optimum sollte jedoch die Dokumentation soweit vorhanden sein, dass interne sowie externe Events dokumentiert sind und gemeinsam mit einem Zustandsdiagramm sowohl der Verbindung als auch der Software dazu genutzt werden können, genau nachzuvollziehen, wann sich der aktuelle Zustand ändert und wie dieser wiederhergestellt werden kann.

Die Qualität der eingesetzten Technologie kann zudem auf Effizienz überprüft werden. Da die genutzten Module in das Gerät fest integriert sind, lässt sich Energieverbrauch nicht messen und ist außerdem von dem im Gerät konkret verbauten Chip und dessen Treibern abhängig. Es kann jedoch eine Aussage über Übertragungsraten getroffen werden, welche eine numerische Skala abbilden. Hierdurch kann festgestellt werden, inwieweit manche Anwendungsfälle mit der entsprechenden Technologie möglich sind, oder die Nutzbarkeit dadurch eingeschränkt werden. Die Konfiguration über eine REST-Schnittstelle benötigt keine hohen Datenraten zur Kommunikation, falls der Nutzer jedoch beispielsweise eine Remote-Verbindung durch Telepharo über die p2p Schnittstelle zwischen seinem persönlichen Computer und dem eingesetzten IoT-Gerät aufbauen will, benötigt er eine relativ latenzfreie und großvolumige Datenübertragung.

Letztlich kann überprüft werden, wie leicht sich die Technologie implementieren lässt, sodass sie für das bestehende Projekt genutzt werden kann. Da die Schwierigkeit von Aufgaben im Rahmen von Softwareentwicklung immer eine Kombination aus Zeitaufwand und Komplexität sind, lässt sich hierbei immer nur eine Schätzung oder persönliche Meinung auf Basis der eigenen Präferenz angeben.

1.2 Evaluationsmethodik

Um die genannten Evaluationsziele überprüfen zu können, soll jede der Technologien Bluetooth, NFC und USB in einem vergleichbaren Maße zu Wi-Fi Direct im Hinblick auf die Vollständigkeit ihrer Dokumentation und Nutzbarkeit ihrer Implementierung überprüft werden. Unter dem Aspekt der Softwarequalität lassen sich die Technologien gegenüberstellen. Dabei soll auf Grundlage einer beispielhaften Implementierung im Rahmen des bestehenden Projektes die Vor- und Nachteile verdeutlicht werden und zudem eine Evaluation der Komplexität und des Aufwandes einer Implementierung vorgenommen werden. Metriken, um die Qualität der Software zu messen, sind als ISO-Standard 9126 definiert [Liggesmeyer, S.6]. Hierbei soll auf die folgenden Punkte eingegangen werden und wie diese extern gemessen werden können, da nicht einsehbar ist, wie ausgelieferte Implementierungsartefakte intern getestet werden und dokumentiert sind. Es soll so auf die Punkte Zuverlässigkeit, Wartbarkeit (Instandhaltbarkeit), Funktionalität, Effizienz und Benutzbarkeit als Softwarequalitätsmerkmale eingegangen werden.

1. *Zuverlässigkeit:* Um die Robustheit und Zuverlässigkeit der Technologien feststellen und vergleichen zu können, muss festgestellt werden können, wie viele Fehler in der genutzten Implementierung existieren [Liggesmeyer, S.9]. Da dieser Punkt jedoch unter anderem stark von der genutzten Hardware abhängt soll sich zunächst auf das Android Gerät Samsung Galaxy S7 und einen Raspberry Pi 3 B+ beschränkt werden. Die Anzahl von Fehlern in einer Technologie lässt sich ebenfalls nicht messbar festsetzen, da zwar die Zahl von gemeldeten Fehlern in einer Anwendung über öffentliche Bugtracker der eingesetzten Bibliothek nachvollziehbar ist, jedoch so nicht alle Fehler abgefangen werden können, da zum einen auch Fehler in Treibern und Hardwaredesign bestehen können und ebenso ein Bugtracker nicht zwingend die vollständige Menge aller Fehler enthalten kann. Gleichzeitig sollte hierbei zur Evaluation die Qualität der Dokumentation bewertet werden. Dazu zählt die Dokumentation und Verfügbarkeit des Sourcecodes im Hinblick auf die Schicht, welche als Schnittstelle bereit gestellt wird.

2. *Wartbarkeit:* Um Fehler bei der Benutzung einer der Technologien beheben zu können, ist es nötig, dass die Technologie auf der einen Seite durch Dokumentation und Quellcode leicht zu verstehen ist und gleichzeitig es erlaubt, reproduzierbar Fehler zu testen. Um hierbei eine aussagekräftige Bewertung vornehmen zu können, sollte keine Metrik wie die Häufigkeit von abweichenden Ergebnissen bei gleichen Eingabeparametern genutzt werden, da dies von nicht überschaubaren Faktoren abhängt. Es kann daher nur auf den Abstraktionsgrad der angebotenen Schnittstellen eingegangen werden, da diese für eigene Tests ersetzt werden müssen. Jedoch kann dieser Punkt auch in Verbindung mit der Zuverlässigkeit gesehen werden, da Fehler bei einer sauberen Dokumentation kein großes Hindernis für Wiederholbarkeit und damit Testbarkeit darstellen.

3. *Funktionalität:* Die Abdeckung der Implementierungen nach der Definition einer Vollständigkeit der erwarteten Funktionalität [Liggesmeyer, S.8] kann nur sehr eingeschränkt überprüft werden, da aus einer externen Sicht die Menge der Funktionalitäten sich auf den Verbindungsaufbau und Verbindungsabbau, sowie ein Senden und Empfangen von verbindungslosen Daten beschränkt und alle Technologien diese Funktionalitäten anbieten. Es muss somit darauf geachtet werden inwieweit die internen Zustände dokumentiert sind und erreicht werden können.

4. *Effizienz:* Die Effizienz der Technologien kann durch einen Vergleich von Datendurchsatzraten und Antwortzeiten festgestellt werden. Diese beiden Kennzahlen sind für eine Verbindungskonfiguration nicht kritisch, jedoch können mittels eines externen HTTP-Wrappers auch Drittprogramme eine solche p2p Verbindung nutzen und in Anwendungsfällen wie Streaming von Audio und Video werden solche Raten dann interessant. Eine andere Seite der Effizienz im Hinblick auf Energieverbrauch der Technologien wird vor allem im Betrieb eines Gerätes mit einer Batterie wichtig. Ein Stromverbrauch lässt sich jedoch ebenfalls kaum bewerten, da dies stark von der verbauten Hardware abhängt. Es ist auch für eine Verbindungskonfiguration zunächst unerheblich, somit wird auf eine Bewertung der Energieaufnahme verzichtet.

5. *Benutzbarkeit:* Die Benutzbarkeit der eingesetzten p2p Lösung, ergibt sich zum Einen daraus, wie stark der Nutzer in die verwendete Technologie eingebunden werden muss und zum Anderen daraus, wie komplex und zeitaufwändig eine Anbindung der Technologie ist. Der Nutzer ist zum größten Teil dazu angehalten, einen Verbindungsaufbau als erwünscht zu bestätigen. Dieser Vorgang sollte so leicht wie möglich gestaltet sein und kann durch eine Gegenüberstellung der einzelnen Schritte, die der Nutzer ausführen muss, bis eine erfolgreiche Verbindung besteht, verglichen werden. Die Komplexität und der Aufwand einer Implementierung kann anhand der Menge an Methoden, die implementiert werden müssen, besonders in einer C-Bibliothek, die eine eigene Schnittstelle zu pharo abbildet, festgestellt werden. Außerdem ist ein Vergleich im Hinblick auf die Komplexität der Logik, die bei der Anbindung gehandhabt werden muss, nötig.

1.3 Messbarkeit

Da Softwarequalität ebenso wie die benannten Metriken größtenteils nicht quantitative messbare Kriterien sind, können auch keine Referenzwerte für die einzelnen Kriterien zu Hilfe gezogen werden. Eine quantitative Aussage wie eine Anzahl an existierenden Tests ist ebenfalls nur bedingt aussagekräftig, da nicht bewertet werden kann, wie robust ein Test geschrieben wurde und welche Ergebnisse getestet werden. Der letztere Punkt lässt sich teilweise über eine Analyse der Quellcodeabdeckung der Tests erreichen, jedoch entspricht eine Implementierung einer Technologie nicht zwingend ihrer Spezifikation und erlaubt so dennoch Abweichungen in den getesteten Ergebnissen und somit auch Inkompatibilitäten zu anderen Implementierungen der selben Technologie.

Eine Bewertung kann somit primär nur auf Grundlage der gesammelten Erfahrungswerte und der vorgenommenen Umsetzung mit den zu überprüfenden Technologien getroffen werden.

1.4 Einschränkungen

Um den Aufwand der Arbeit überschaubar zu halten, gibt es ein paar weitere Rahmenbedingungen, die im Vorfeld definiert werden sollen. Hierbei ist darauf zu achten, wenn bei einer der Technologien ein deutlich größerer Aufwand abzusehen sein sollte, muss die Implementierung dieser Technologie ausgeklammert werden, da es auch einer Nutzbarkeit widerspricht, eine solche Technologie trotz einem hohen Aufwand in der Implementierung zu nutzen, da dies gleichzeitig auch einen hohen Aufwand in der Wartung bedeutet.

Ein Bezug zwischen Kosten und Nutzen lässt sich über ein Qualitätsmodell ebenfalls nur schlecht aufstellen, da alle Technologien frei zugängliche Software besitzen und Kostenunterschiede lediglich durch die genutzte Hardware entstehen. Bei NFC entstehen als Einziges Mehrkosten, da es nicht bereits im Raspberry Pi integriert ist. Es wird somit darauf verzichtet, die Nutzbarkeit der einzelnen Technologien anhand ihrer Kosten gegenüberzustellen.

Weiterhin kann nicht auf Dokumentationen der entsprechenden Standards und Implementierungen der Technologien eingegangen werden, die nicht öffentlich zugänglich sind, da ebenso Änderungen und Erweiterungen an diesen Spezifikationen dann nicht nachvollziehbar sein werden und ein Implementierungsaufwand einer eigenen Anbindung somit unvergleichbar hoch ist.

2 Warum p2p?

Das vorausgegangene Projekt einer Verbindungskonfiguration von IoT-Geräten [Kretzschmar] hat gezeigt, dass eine initiale p2p Verbindung aufgebaut werden muss, um eine Wi-Fi Verbindung konfigurieren zu können. In Verbindung mit einer ad-hoc Verbindung muss auch gleichzeitig eine Service Discovery betrachtet werden, um den initialen verbindungslosen Informationsaustausch über mögliche Verbindungspartner zu schaffen. Diese findet aktuell ebenfalls über Wi-Fi Direct in Form von *DNS Service Discovery* statt. Dies hat den Vorteil, dass sich zusätzliche Daten mit einem einzigartigen Schlüssel dem Eintrag eines Service hinzufügen lassen. Eine p2p Technologie muss somit nicht zwingend verbindungslosen Beacon Signale mit Nutzdaten zur Verfügung stellen, da auch eine Hybridlösung genutzt werden kann. Das Erkennen von verfügbaren Services funktioniert bereits stabil über Wi-Fi Direct, somit wird im Weiteren lediglich der Aufbau einer p2p Verbindung thematisiert.

Eine direkte Verbindung zwischen zwei Endgeräten ist nötig, wenn Daten zwischen Anwendungen ausgetauscht werden sollen und keine Verbindung über ein gemeinsames Netzwerk oder einen Server stattfinden kann oder soll. Mögliche Anwendungsfälle einer solchen p2p Verbindung sind oftmals die kabellose Nutzung von Peripheriegeräten oder der Austausch von Dateien ähnlich zu *Apple Air Drop*. Verbindungen zwischen Endgeräten ohne die Nutzung einer bestehenden Netzwerkinfrastruktur resultiert in einer von der genutzten Technologie abhängigen, stark reduzierten Reichweite, welche zwischen den Geräten überwunden werden kann.

Das Zustandsmodell Fig. 1 eines p2p Verbindungsaufbaus lässt sich aus Sicht der beteiligten Geräte soweit vereinfachen, dass lediglich fünf von außen beobachtbare Zustände für eine erfolgreiche Nutzung der Verbindung relevant sind. Damit der Nutzer beim Verbindungsaufbau lediglich in den Prozess mit seinem Smartphone interagieren muss, sollte der p2p Verbindungsaufbau seine Zustände so vereinfacht darstellen können, um einen konsistenten Ablauf automatisiert gewährleisten zu können.

Zunächst soll über den Zustand *INACTIVE* erkannt werden können, wenn die zu verwendende Schnittstelle ausgeschaltet ist, um sie im Ablauf des Programms einschalten zu können. Sobald der Zustand *IDLE* erreicht ist, kann die Schnittstelle genutzt werden, um Verbindungen über *ACCEPTING* zu akzeptieren oder eine neue Verbindung zu einem anderen Gerät über *CONNECTING* aufzubauen. Sobald ein Gerät Verbindungen annimmt und ein weiteres Gerät versucht, sich zu diesem zu verbinden, wechseln beide Geräte im Erfolgsfall in den Zustand *CONNECTED* und besitzen einen gemeinsamen Kommunikationskanal. Sobald die Verbindung geschlossen wird, befinden sich beide Geräte wieder im Grundzustand *IDLE*. Falls eine Technologie mehr als eine parallele Verbindung erlaubt, teilt sich diese beim Akzeptieren einer neuen Verbindung auf und verbleibt weiterhin gleichzeitig im Zustand *ACCEPTING*.

Ein besonderer Nutzungspunkt für p2p Verbindungen stellen IoT-Geräte dar, da so zum einen Kommunikation ohne Konfiguration des Nutzers zwischen den IoT-Geräten stattfinden kann und eine Konfiguration der IoT-Geräte durch den Nutzer stattfinden kann ohne Peripherie an das Gerät anschließen zu müssen. In-

ternet of Things ist ein Sammelbegriff für Gebrauchsgegenstände wie Heizungsthermostate oder Maschinen wie CNC-Fräsen, die in einer moderneren Interpretation ihrer Funktionalität über ein Netzwerk untereinander und mit Servern kommunizieren. Anwendungsgebiete können die Hausautomation, Sensornetzwerke und Betriebsüberwachung von Maschinen sein. In einem weiter gefassten Blickfeld wird unter diesem Begriff auch die Vernetzung von ganzen Städten und deren Infrastruktur untersucht [ITU Study Group].

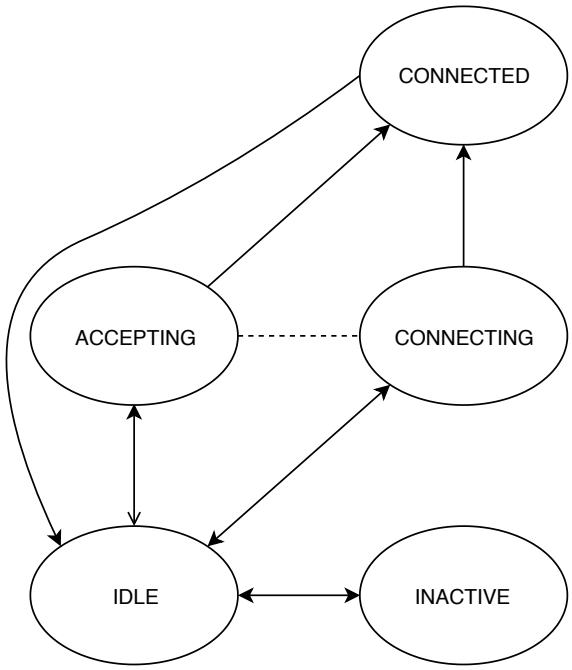

Fig. 1: Das Zustandsmodell eines p2p Verbindungsaufbaus zeigt die von außen beobachtbaren Zustände für eine erfolgreiche Nutzung einer p2p Verbindung.

2.1 Services

Für solche IoT-Geräte muss eine Erstkonfiguration initial vorgenommen werden, um eine Kommunikation mit anderen Geräte zu ermöglichen. Da nicht jedes Gerät einen Bildschirm oder komplexe Eingabemöglichkeiten besitzt, ist es sinnvoll, diese Einrichtung auf ein Smartphone auszulagern, wie es bereits im vorausgegangenen Projekt geschehen ist. Da die Konfiguration durch einen REST-Server angeboten wird, ist es so auch möglich, weitere Konfigurationsmöglichkeiten für Sensoren und Aktuatoren als weiteren Service anzubieten.

Ein Service definiert sich als ein Dienst, welcher Anderen eine Leistung oder Aufgabe bereitstellt. Im Sinne der Informatik ist ein solcher Dienst eine Einheit von Programmen, welche durch eine Schnittstelle eine Funktionalität verwaltet und extern zur Verfügung stellt. Eine solche Schnittstelle kann auf verschiedene Arten bereitgestellt werden, jedoch sollten diese Schnittstellen über Verträge abgesichert werden, um ungewollte Änderungen im Betrieb zu vermeiden und eine reibungslose Anbindung von neuen Funktionalitäten zu gewährleisten [Finger et al., S.11].

Im Rahmen einer Anwendungsarchitektur als Sammlung von *Microservices* treten Services als *Messagehandler* auf und definieren ihre Schnittstellen durch Nachrichten oder *RPC* Aufrufe. Microservices können dabei wiederum als Services betrachtet werden. Vorteile bietet eine solche Architektur in einer leichten Skalierbarkeit der Anwendung oder einzelner Teile und einer sauberen Kapselung von Anwendungsteilen. Zwischen *Messagehandlern* wird oftmals auf Authentifizierung verzichtet, wenn die teilnehmenden Dienste bekannt sind, da auf diese Weise ein höherer Durchsatz erzielt werden kann [Microservices].

Ein *Daemon*, welcher eine Hardwarekomponente verwaltet, kann ebenfalls als Service, welcher abstraktere Funktionalität auf dieser Hardware bereitstellt, definiert werden. Auf diese Kategorie von Services trifft die Einschränkung der Einzigartigkeit zu, da immer eine Systemresource nur von einem Programm genutzt werden kann.

Im simpelsten Fall kann ein Service auch als Server definiert werden, um seine Funktionalität Clients zur Verfügung zu stellen. Dies bietet sich besonders dann an, wenn verschiedene Architekturen von Clients in einer hohen Anzahl den selben Service benutzen sollen oder Clients zunächst eine Authentifizierung vornehmen sollen. Ein klassisches Client/Server-Modell sieht hierbei vor, dass auszuführende Berechnungen so aufgeteilt werden, dass ein Server Daten konsistent nutzen kann und in der Lage ist, die verwalteten Daten mehreren Clients auszuliefern [Abts, S.8].

2.2 Netzwerktopologie

Der Begriff *peer-to-peer* beschreibt eine Art von infrastrukturlosen Netzwerk zwischen zwei oder mehr lokalen Geräten. Um die Vor- und Nachteile einer solchen Verbindung darstellen zu können, werden zunächst die unterschiedlichen Netzwerktypen gegenüber gestellt. Ebenso ist es sinnvoll die zu evaluierenden Technologien in verschieden Topologien einzuordnen, um sie im Hinblick auf Reichweiten und Eigenschaften vergleichen zu können.

Ad Hoc Netzwerktechnologien erlauben es zwei oder mehr Geräten ein dezentrales Netzwerk aufzubauen. Um die Kontrolle über die gemeinsame Schnittstelle zu erlangen, agiert hierbei entweder eines der Geräte als Verwalter der Verbindung oder es werden spezielle Sequenzen zum Frequenzwechsel in Verbindung mit einer Kollisionsauflösung durch zufällig langes Warten definiert, um Kommunikationskonflikte zwischen Teilnehmern zu vermeiden.

Im Gegensatz zum dezentralisierten *ad hoc* Netzwerk nutzt ein Stern-basiertes Netzwerk Knotenpunkte, um die Kommunikation zu Endgeräten zu übernehmen. Wi-Fi bildet ein solches Netzwerk mit *Access Points* ab, da diese ein gemeinsames Übertragungsmedium aller verbundenen Endgeräte verwalten und keine Endgeräte direkt miteinander kommunizieren. Ein solches Netzwerk ist dann sinnvoll, wenn der Übergang zu einem anderen Netzwerk in einer Singularität verwaltet werden soll und Endgeräte generell wenig untereinander kommunizieren, da eine Datenübertragung so die doppelte Bandbreite benötigt, um die selbe Übertragungsgeschwindigkeit zu erzielen, da Pakete zunächst zum Knotenpunkt übertragen werden und dieser die Pakete über das gemeinsame Medium erneut an das Zielgerät übermittelt.

Alle zu überprüfenden Technologien bilden ein *ad hoc*, da das Ziel ist, das IoT-

Gerät erst in ein Wi-Fi Netzwerk zu bringen und somit noch keine gemeinsame Netzwerkverbindung besteht. Relevant ist auch, dass lokale Geräte in einem Raum angesprochen werden können, ohne große Distanzen überbrücken zu müssen. Die genutzten Technologien gehören unterschiedlichen Topologien an, wodurch sich ihre Einsatzgebiete auch unterscheiden:

- *Wi-Fi Direct* ist ein Wireless Local Area Network (WLAN) und kann in dieser Kategorie über theoretische Distanzen von bis zu 100 Metern Daten übertragen. Ähnlich zu Wi-Fi ist die Signalstärke von Wi-Fi Direct ausreichend, um in einem Gebäude eine Verbindung zu anderen Geräten aufbauen zu können. Da Signale stark durch Objekte und Wände gedämpft werden, reduziert sich jedoch die Reichweite innerhalb von Gebäuden drastisch, wodurch eine Einordnung als Personal Area Network (PAN) zutreffender ist.

- *Bluetooth* ordnet sich ebenfalls als PAN ein und erlaubt den Datenaustausch bis zu einer theoretischen Distanz von 10 Metern. Dies ist ausreichend, um Geräte im Bereich um die eigene Person anzusteuern, was eine durchschnittliche Raumgröße abdeckt. Bluetooth unterstützt eine Vielzahl von Protokollen, insbesondere zur Audioübertragung. Typische Anwendungsfälle sind hierbei drahtlose Peripheriegeräte und der lokale Dateienaustausch.

- *NFC* kann lediglich Distanzen von unter 10 Zentimetern überbrücken, und kommt so größtenteils in Verbindung mit mobilen Zahlungsmitteln zum Einsatz. NFC ist zum einen der Name der eingesetzten Technologie als Weiterführung von RFID, gleichzeitig beschreibt es jedoch auch eine Netzwerktopologie, die es Geräten erlaubt auf einem engen Raum miteinander zu kommunizieren.

- *USB* ist ein kabelgebundenes Übertragungsmedium und erfordert so einen direkten Zugang zum Raspberry Pi. Da in der Spezifikation zu USB 2.0 eine maximale Kabellänge von 5 Metern definiert ist [USB Spezifikation], ordnet sich dieser Standard ebenfalls als PAN ein.

3 Grundlagen der Verbindungstechnologien

Da die Nutzbarkeit der Übertragungstechnologien für die Verbindungskonfiguration bereits in [Kretzschmar] überprüft wurden, soll noch einmal jede Technologie in ihrer Funktionsweise kurz vorgestellt werden und der Ablauf eines Verbindungsaufbaus beschrieben werden. Im Gegensatz zu Wi-Fi Direct werden die Zustände beim Verbindungsaufbau intern gehandhabt. Es ist dennoch hilfreich zu verstehen, wie eine Technologie funktioniert, um Fehlerursachen bei einem Verbindungsaufbau leicht identifizieren zu können. Die genutzten Bibliothek dieser Technologien werden in Abschnitt 4 beschrieben. Zwischen der theoretischen Funktionsweise und ihrer Umsetzung besteht immer eine gewisse Abweichung. Diese Abweichungen sollten jedoch bei der Anbindung der Bibliotheken nicht mehr extern bemerkbar sein. Eine solche Abweichung ist nicht kritisch für die Nutzung der Technologie, jedoch reduziert sie die Interoperabilität zwischen verschiedenen Anbindungen der Technologie.

3.1 Bluetooth

Um Peripheriegeräte untereinander zu verbinden, kann Bluetooth als drahtloses Übertragungsmedium auf Basis von Piconetzwerken genutzt werden. Bluetooth besteht aus mehreren Protokollschichten, die für einen Verbindungsaufbau durchlaufen werden. Zunächst wird ein *Asynchronous Conneciton-Less* (ACL) Kanal zum gewünschten Gerät geöffnet, über den alle folgenden Kommunikationen stattfinden werden. Dieser Kanal übernimmt die Aufgabe, die Kommunikation mit dem Bluetooth Chip zu steuern und erhaltene Nachrichten an die nächste Schicht zurückzugeben [Sauter, S.400]. Das *Host Controller interface* (HCI) ist nicht Teil des Bluetooth Standards, wird jedoch in den meisten Implementierungen des Bluetoothstacks genutzt, um niedrigere Schichten der Implementierung von höheren Schichten zu trennen. Wie der Name bereits suggeriert, bildet diese Schnittstelle den Übergang zwischen Bluetooth Modul und Treiber [Miller et al., S.65]. Auf Basis dieser Schnittstelle erhalten die höheren Protokollschichten nur für das Gerät relevante Datenpakete aus dem ACL Kanal.

Das Logical Link Control and Adaption Protocol (L2CAP) wird als erstes Element der höheren Protokollschichten genutzt, um mehrere gleichzeitige Verbindungen unterschiedlicher Anwendungen über einen einzigen Kanal übertragen zu können [Sauter, S.395]. Das Service Discovery Protocol (SDP) schließt an dieser Stelle an, um entfernten Geräten die bestehenden Services des Gerätes bekannt machen zu können, sodass dieser eine Auswahl des zu verbindenden Services tätigen kann [Morrow, S.395]. RFCOMM erlaubt es eine serielle Schnittstelle über Bluetooth abzubilden. Da es ebenfalls ein fixer Teil des Bluetooth Protokollstacks ist, kann dies von L2DAP auch ohne SDP genutzt werden. Hierbei ist jedoch zu beachten, dass RFCOMM lediglich 30 Ports unterstützt, was 30 gleichzeitigen Anwendungsverbindungen zwischen zwei Bluetooth Geräten entspricht [Sauter, S.398]. SDP ist nicht in der Lage, diese Anzahl zu erhöhen, da jedoch nicht alle angebotenen Bluetooth Services immer genutzt werden, macht es Sinn, den eigenen Service im SDP zu

definieren und so sich von der RFCOMM-Schicht erst einen Port zuweisen zu lassen, wenn die Verbindung tatsächlich genutzt werden soll. Damit eine Anwendung sich so im SDP-Protokoll registrieren kann, muss sie ein entsprechendes Bluetooth Profil, im Falle von RFCOMM das Serial Port Profile, anbieten um die Interoperabilität mit anderen Geräten gewährleisten zu können [Sauter, S.411].

3.2 NFC

Near Field Communciation (NFC) ist eine auf RFID basierende Technologie, die sich dahingehend von RFID unterscheidet, dass keine Trennung zwischen lesenden Geräten und passiven Transpondern besteht. Durch diese Aufhebung können NFC Geräte sowohl lesendes Gerät als auch passiver Verbindungsteilnehmer sein und somit eine beidseitige Kompatibilität zu RFID bieten [Langer et al., S.6]. NFC ist auch prinzipiell eine Erweiterung des bestehenden Standards, welche in der Lage ist, sich bestehende Infrastruktur zunutze zu machen. RFID Lesegeräte erzeugen ein Magnetfeld, um durch Induktion die passiven Chips mit Strom versorgen zu können. Eine Datenübertragung findet dabei immer durch die Antwort der passiven Komponenten auf eine Anfrage der aktiven Komponente der Verbindung statt. Solange das Trägersignal besteht, können beliebig viele dieser Anfragen gestellt werden [Langer et al., S.89]. Da NFC-Geräte sowohl aktive als auch passive Komponente dieser Kommunikation repräsentieren können, unterstützt die NFC Architektur drei Funktionsmodi.

1. *p2p Modus:* Wenn zwei NFC Geräte miteinander kommunizieren, können diese direkt Daten zueinander übertragen, ohne die anderen beiden Modi nutzen zu müssen. Diese Kommunikationsart teilt sich wiederum in aktiven und passiven Modus auf [Langer et al., S.91]. Diese beiden Modi unterscheiden sich dahingehend, dass im aktiven Modus jedes Gerät sein eigenes Trägersignal generieren muss und so ein geringerer Energieverbrauch auf dem initiierendem Gerät besteht [Langer et al., S.94].

2. *Lesender Modus:* Um mit passiven RFID Chips kommunizieren zu können, muss das NFC Gerät als Lesegerät agieren [Langer et al., S.99]. Hierbei erzeugt das NFC Gerät wieder, ähnlich zum passiven p2p Modus, das Trägersignal für den Kommunikationskanal. Der Datenaustausch findet dann wieder über die Abfrage von Nachrichten statt.

3. *Modus der Karten-Emulation:* Damit ein NFC Gerät auch mit RFID-Lesegeräten kompatibel sein kann, muss es in der Lage sein, eine passive Komponente abzubilden. Dies wird erreicht, indem kein aktives Trägersignal aufgebaut wird und lediglich auf ankommende Anfragen im RFID-Protokoll geantwortet wird [Langer et al., S.100].

Wenn der p2p Modus nicht direkt unterstützt wird, können auch NFC Geräte durch den lesenden Modus in Verbindung mit einer Emulation einer RFID Chipkarte untereinander kommunizieren. Hierzu muss jedoch im Vorausgang festgelegt werden,

welche Seite als aktiver Teilnehmer und welche Seite als passiver Teilnehmer der Kommunikation auftritt [Langer et al., S.101].

3.3 USB

Die Schnittstelle Universal Serial Bus (USB) ist ein kabelgebundenes Bussystem bestehend aus einem Host und vielen Endgeräten mit optionalen Hubs [Eberhardt et al., S.23f]. Ein Hub ist in der Lage über eine USB Schnittstelle mehrere andere USB-Geräte anzusteuern und mit Strom zu versorgen. Er erhöht so die physikalische Kapazität für USB-Geräte eines Hosts [Eberhardt et al., S.40]. USB erlaubt es, den Großteil der Verbindungslogik im Host zu behandeln, was dazu führt, dass Peripheriegeräte leicht implementiert werden können [Axelson, S.37]. Im Kontext von USB werden diese Geräte als *Functions* bezeichnet. Functions können nicht direkt untereinander kommunizieren, dies kann lediglich über Interaktionen mit dem Host vollzogen werden. Da USB-Geräte, welche hinter Hubs angebunden sind, nicht eindeutig vom Host adressiert werden können, spricht ein Host Functions über Broadcastnachrichten auf dem Bus an [Eberhardt et al., S.41].

Bevor jedoch eine solche Kommunikation stattfinden kann, muss das Gerät auf dem Bus registriert werden. Dazu ist USB in der Lage, Hot Plugging zu erkennen und versucht dann eine Kommunikation mit der neuen Function aufzubauen [Eberhardt et al., S.45]. Wenn die Function auf die Kommunikationsanfrage reagiert, wird ein Reset der USB Function durchgeführt, sie kann nun eine höhere Stromstärke aus dem USB anfragen und ihr wird eine Adresse zugewiesen [Eberhardt et al., S.94f]. Daraufhin fragt der Host eine Beschreibung des USB-Gerätes ab und wählt anhand dieser Beschreibung einen passenden Treiber aus [Eberhardt et al., S.95]. USB unterstützt mehrere Transfermodi, um verschieden Anwendungsfälle abzudecken.

- Der Standardmodus ist ein *Control-Transfer*, welcher es erlaubt, geringe Datenmengen als Blöcke zu Senden und zu Empfangen. Dieser Modus ist dazu gedacht, Endgeräte zu konfigurieren und zu steuern. Jedes USB-Gerät muss diesen Modus unterstützen, da zum Verbindungsaufbau Befehle in darüber gesendet werden [Axelson, S.79].

- Um große Datenmengen mit einem möglichst geringen Overhead üertragen zu können, wird der *Bulk-Transfer* genutzt. Im Gegenzug zu einem Datendurchsatz wird eine solche Übertragung im Bus niedrig priorisiert, um zeitkritischeren Geräten den Vorrang zu geben [Axelson, S.85]. Dieser Modus wird besonders von Massenspeichergeräten um Dateien vom Speichermedium über den Bus zu übertragen oder auf das Speichermedium aus dem Bus zu schreiben.

- Ein *Interrupt-Transfer* kann genutzt werden um geringe Datenmengen zeitkritisch übertragen zu können. Der Name suggeriert, dass Functions Hardware-Interrupts auslösen können, jedoch stellt eine solche Übertragung lediglich eine Priorisierung der anstehenden Daten dar [Axelson, S.88]. Interrupt-Transfer kommt vor Allem bei Eingabegeräten des Nutzers zum Einsatz, damit keine spürbare Latenzzeit entstehen kann.

- Wenn tatsächlich Daten in Echtzeit übertragen werden sollen, kommt ein *Isochrone-Transfer*in Frage. Diese Art von Übertragung erlaubt es, mehr Daten als ein Interrupt-Transfer über den Bus zu senden, jedoch können Aufgrund der zeitkritschen Natur einer solchen Übertragung keine fehlerhaften Daten erneut übertragen werden [Axelson, S.85].

Abhängig vom Anwendungsfall muss aus diesen Modi ein passender Transfermodus gewählt werden. Im Fall von Android Open Accessory werden die Geräte über einen Bulk-Transfer angesprochen, um eine höchstmögliche Datenrate zu bieten.

3.3.1 Android Open Accessory

Um ein Android Gerät über USB ansteuern zu können, ist es nötig, das Android Open Accessory (AOA) Protokoll in der Anwendung zu implementieren. Hierbei wird das Androidgerät auf Hostseite im Bulk-Transfer geöffnet und es werden zwei Befehle über USB ausgetauscht. Zunächst muss der Host die unterstützte Accessory Protokoll Version vom Android Gerät abfragen und seine Kommunikation darauf auslegen. Im nächsten Schritt gibt der Host Details über sich selbst dem Android Gerät bekannt. Dies geschieht durch eine Menge von Strings, die dem Nutzer gezeigt werden und einem URI-Link, über welchen passende Anwendungen auf dem Smartphone ausgewählt werden können. Sobald der Host alle diese Daten gesendet hat, wird der Befehl, dass ein Accessory nun von Android zu nutzen ist, über den Control-Transfer ausgespielt [Android Open Accessory].

4 Eigene Umsetzung

Das Projekt einer Verbindungskonfiguration [Kretzschmar] wurde über eine Wi-Fi Direct Verbindung implementiert. Um Pharo Installationen aufzufinden, wird eine Service Discovery Anfrage über Wi-Fi Direct vorgenommen. Alg. 1 beschreibt die Definition und das Bereitstellen dieser Definition über Wi-Fi Direct. Zu sehen ist, dass DNS Service Discovery genutzt wird, da ein Name sowie ein Identifikationspfad angegeben werden müssen. Die *serviceValues* beschreiben eine optionales Dictionary von Werten, welche über die Service Discovery ausgespielt werden. In diesen zusätzlichen Daten muss für Wi-Fi Direct jedoch noch der Port des Konfigurationsservers angegeben werden, um eine Verbindung auf Anwendungsebene zu ermöglichen.

Für NFC und USB muss keine Service Discovery stattfinden, da diese lediglich über eine Hardwareschnittstelle mit einem Zielgerät kommunizieren. Es ist jedoch sinnvoll, die Service Discovery auch bei solchen Technologien beizubehalten, um eine Statusüberwachung aller nahegelegenen Geräte zu ermöglichen. Die Zuordnung eines gefundenen Services zur Adresse eines physikalischen Gerätes fällt dann jedoch in die Verantwortung des Nutzers.

Alg. 1: Service Discovery Definition (Servercode in pharo)

```
serviceValues := (SmallDictionary new: 3)
  at: #identifier put: 'Pharo Device';
  at: #connection put: 'up';
  at: #port put: (8889 asString);
  yourself.
serviceDescription := LWpaBonjourService new:
  'ConfigurationService' as: 'connectivity.pharo._tcp'
  with: serviceValues.
p2pDevice := LWpaInterface onAnyP2P.
p2pDevice configureDynamic: [
  p2pDevice p2pServiceAdd: serviceDescription;
  p2pServiceUpdate. ].
```

4.1 REST Schnittstelle

Eine REST Schnittstelle wird über OpenAPI bereitgestellt, um die eigentliche Konfiguration der Wi-Fi Verbindung vornehmen zu können. Jedes Übertragungsmedium muss in der Lage sein, diese auf HTTP basierende Schnittstelle anzusprechen. Der REST-Client dieser Schnittstelle wird asynchron bereitgestellt, um einen Verbindungsaufbau abhängig von der genutzten Technologie zu ermöglichen. In Alg. 2 ist zu sehen, wie ein REST-Client über Wi-Fi Direct bereitgestellt wird. Da keine Anpassungen am HTTP-Client nötig sind, muss die Instanziierung des gesamten REST-Client einen erfolgreichen Verbindungsaufbau über den *ipReceiver* abwarten, was sich leicht durch die Verkettung der Observables realisieren lässt.

Alg. 2: Asynchrones Warten für Wi-Fi Direct Verbindung (Clientcode in Kotlin)

```
Observable.unsafeCreate<> { subscriber ->
    wifiManager.connect(wifiChannel, device.asConfig(),
object : WifiP2pManager.ActionListener {
      override fun onSuccess() {
        subscriber.onNext(Unit)
        subscriber.onCompleted()
      }
      ...
    })
}.zipWith<String, DeviceConfigurationProvider>(ipReceiver) {
_, hostAddress ->
  subscriber.onNext(DeviceConfigurationProvider.getInstance(
getHttpClient(), device, hostAddress))
  subscriber.onCompleted()
}
```

Alg. 3 zeigt, wie der eigentliche REST-Client von den speziellen Änderung einzelner Technologien unberührt bleibt. Für dessen Instanziierung wird lediglich der zugrundeliegende HTTP-Client als Factory explizit hinzugegeben, damit dieser frei konfiguriert werden kann.

Alg. 3: REST-Client Instanziierung (Clientcode in Kotlin)

```
fun getInstance(callFactory: Factory, device: Device,
deviceHost: String): DeviceConfigurationProvider {
  val httpUrl = HttpUrl.Builder()
    .host(deviceHost)
    .port(device.wifiDetails.port)
    .scheme("http")
    .build()
  return Retrofit.Builder()
    .callFactory(callFactory)
...
    .baseUrl(httpUrl)
    .build()
    .create(DeviceConfigurationProvider::class.java)
}
```

Auf Basis dieser Anwendung kann nun die Implementierung einer p2p Verbindung über Bluetooth, NFC und USB beschrieben vorgenommen werden. Jede dieser Technologien soll separat betrachtet werden, um Details und Probleme bei ihrer Implementierung aufzuzeigen. Jede Technologie wird clientseitig als *DeviceConfigurationProvider* implementiert, um eine Austauschbarkeit mit geringem Aufwand ausschließlich durch das Auswechseln einer *Dependency Injection Defintion* zu ermöglichen. Dies ist in Alg. 4 gelöst, indem in der laufenden Anwendung lediglich der *scope* ausgewechselt wird (Alg. 5) und ein Wechsel zwischen den einzelnen Technolo-

gien dann auch über eine Einstellungsseite möglich ist. Alternativ wäre denkbar, dass auch eine Service Discovery eine Feature Matrix ausliefert, die angibt, welche Technologien auf dem Gerät unterstützt werden und so ein kompatibler Provider selbstständig ausgewählt wird.

Alg. 4: Auswechselbarkeit durch Dependency Injection (Clientcode in Kotlin)

```
scope(wifiDirectScope) { factory<DeviceConnectionProvider> {
  WiFiConnectionProvider(get())
} }
scope(bluetoothScope) { factory<DeviceConnectionProvider> {
  BluetoothConnectionProvider()
} }
scope(nfcScope) { factory<DeviceConnectionProvider> {
  NFCConnectionProvider()
} }
scope(usbScope) { factory<DeviceConnectionProvider> {
  USBConnectionProvider(get())
} }
```

Alg. 5: Auswechselbarkeit durch Dependency Injection (Clientcode in Kotlin)

```
fun switchToTechnology(scopeId: String) {
  currentScope?.close()
  currentScope = getKoin().getOrCreateScope(scopeId)
}
```

4.2 HTTP-Kapselung

Zunächst wird jedoch eine HTTP-Kapselung betrachtet, da diese für alle p2p Verbindungen nötig ist und daher auch generisch definiert werden soll. Um eine solche Kapselung zu ermöglichen, müssen HTTP Anfragen über ein anderes Protokoll als TCP/IP gesendet werden können.

Bei der Kapselung von HTTP Anfragen muss bedacht werden, wie eine Technologie Verbindungen zur Verfügung stellt, da so zwischen kurzlebigen Verbindungen ähnlich zu HTTP Anfragen und langlebigen Verbindungen wie einer Datenübertragung per Kabel unterschieden werden muss. Letztere weisen dabei das Problem auf, HTTP Anfragen, die mit einem *EOF* beendet werden, über eine langlebige Verbindung zu senden, da ein *EOF* immer auch das Ende der Verbindung aufzeigt. Weiterhin können Fehler eine langlebige Verbindung beenden, jedoch ist es bei solchen Verbindungen nur begrenzt möglich, einen erneuten Verbindungsaufbau zu versuchen, da der Zustand einer langlebigen Verbindung bei NFC und USB außerhalb der Kontrolle der Anwendung liegt.

Um eine p2p Verbindung unabhängig von der genutzten Verbindungstechnologie verwenden zu können, ist es nötig, eine HTTP-Verbindung zum existierenden REST Server aufbauen zu können. Da das REST-Prinzip eng mit HTTP verbunden ist,

sind auch die Implementierungen von REST meistens fest mit einem HTTP-Server oder Client verbunden. Dies stellt jedoch keine tiefgreifenden Probleme dar, da ein Eingreifen in eine HTTP-Implementierung nur nötig wird, wenn die Technologie nicht mit vertretbarem Aufwand als Socketverbindung dargestellt werden kann.

4.2.1 Serverseite

Auf der Serverseite stellt eine enge Bindung zwischen HTTP-Server und REST-Anbindung kein Problem dar, da der HTTP-Server nicht von den genutzten Sockets entkoppelt werden muss. Dazu wird ein zweiter Server vorgeschaltet, welcher lokale Sockets nutzt um mit dem eigentlichen Server zu kommunizieren und den Verbindungsaufbau sowie Verbindungsabbau für die genutzte Technologie und deren mögliche virtuelle Socket-Verbindung zu verwalten.

Wie im Alg. 6 zu sehen, baut der kapselnde Server eine Socket-Verbindung auf, um Daten zum Zielserver zu senden. Diese Verbindung wird erst aufgebaut, sobald das Endnutzergerät kurz davor steht eine Anfrage an den Zielserver zu stellen, um mögliche Zeitüberschreitungen im HTTP-Server zu vermeiden.

Alg. 6: Instanziierung eines Sockets (Servercode in C)

```
int s = socket(AF_INET, SOCK_STREAM , 0);
struct sockaddr_in rem_addr = { 0 };
rem_addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
rem_addr.sin_family = AF_INET;
rem_addr.sin_port = htons(targetPort);
connect(s, (struct sockaddr *)&rem_addr , sizeof(rem_addr));
```

Die bestehende Socket-Verbindung zum Zielserver wird genutzt, um die ankommenden Daten und deren Antworten voll duplex weiterzuleiten. Dies ist realisiert, indem eine Methode *pipeData* in zwei Threads mit invertiertem *sourceSocket* und *sinkSocket* aufgerufen wird, bis ein *EOF* gesendet wird (Alg. 7). Für den kapselnden Server ist es so unerheblich, welche Seite Daten zuerst senden möchte. Die Verbindung zwischen dem Zielserver und dem kapselnden Server kann so auch unabhängig von der Verbindung zum Endgerät verwaltet werden, da beim Lesen der Daten vom Zielserver ein *EOF* das Ende der Verbindung zum Zielserver aufzeigt. Die Verbindung zum Endgerät muss so nicht zwingend geschlossen werden.

Alg. 7: Datenweiterleitung durch Sockets (Servercode in C)

```
int pipeData(int sourceSocket, int sinkSocket, char* buffer){
  int bytes_read = read(sourceSocket, buffer, BUF_SIZE);
  if(bytes_read < 0) return -1;
  if(bytes_read == 0) return -20; // indicates an EOF

  int bytes_sent = send(sinkSocket, buffer, bytes_read, 0);
  if(bytes_sent < 0) return -2;
  return 0;
}
```

4.2.2 Clientseite

Ein Abkapseln der Technologie über einen weiteren Server auch auf Seite des Clients erscheint als vermeidbarer Overhead, da so keine Serverimplementierung auch im Client stattfinden muss. Stattdessen wird hier der HTTP-Client, welcher der REST-Bibliothek zugrunde liegt, so aufgetrennt, dass keine TCP/IP-Verbindungen aufgebaut werden, jedoch die Anfragen aus der Bibliothek als String entnommen werden können und deren Antworten als String eingespeist werden können. Für den REST-Client retrofit wird intern okhttp als HTTP-Client genutzt. Um zu verstehen, welche Änderungen am Client nötig sind, sollte zuerst die interne Struktur der okhttp Bibliothek erläutert werden.

Serveraufrufe werden im okhttp-Client (Alg. 8) durch eine Kette von Interceptors verarbeitet. Jeder *Interceptor* hat dabei die Möglichkeit den Aufruf oder die Kette beliebig zu verändern und in dieser Liste nimmt jeder Interceptor eine andere Rolle ein. Der *RetryAndFollowUpInterceptor* stellt in der Kette eine *StreamAllocation* bereit und übernimmt das Abbrechen von Aufrufen. Der darauf folgende *BridgeInterceptor* verwaltet Cookies aus den Anfragen und Antworten, sowie die Übersetzung von Anwendungsanfragen zu Netzwerkanfragen. Dies beinhaltet ebenfalls die Verwaltung von netzwerkrelevanten Headern wie zum Beispiel den "User-Agent"-Header oder "Content-Encoding"-Header. Wie der Name des *CacheInterceptor* bereits vermuten lässt, wird in diesem das Speichern und Abrufen von Antworten auf wiederkehrende Anfragen ermöglicht. Sowohl Cookies als auch der Cache lassen sich einfach umgehen, indem der Client jeweils kein Objekt ausliefert oder ein Objekt bereitstellt, welche alle Methoden mit leeren Ergebnissen quittiert. Bevor der Aufruf vom *CallServerInterceptor* tatsächlich ausgeführt wird und auf ein Ergebnis gewartet wird, baut der *ConnectInterceptor* noch eine HTTP-Verbindung über die *StreamAllocation* der Kette auf. Für diese Verbindung wird dann ein *HTTP-Codec* genutzt, um die Anfrage auf den Socket zu schreiben und die Antwort zu lesen.

Alg. 8: Interner Aufbau von okhttp (Clientcode in Java) [okhttp RealCall]

```
List<Interceptor> iceptors = new ArrayList<>();
...
iceptors.add(retryAndFollowUpInterceptor);
iceptors.add(new BridgeInterceptor(client.cookieJar()));
iceptors.add(new CacheInterceptor(client.internalCache()));
iceptors.add(new ConnectInterceptor(client));
...
iceptors.add(new CallServerInterceptor(forWebSocket));
```

Im Gegensatz zur okhttp-Implementierung muss die p2p Verbindung so verwaltet werden, wie es von der Implementierung der Technologie vorgegeben wird. Dazu wird im Alg. 9 der *RetryAndFollowUpInterceptor* sowie der *ConnectInterceptor* weggelassen und der *CallServerInterceptor* im *SimpleServerInterceptor* soweit vereinfacht, dass dieser keine Handshakes mehr unterstützt und nicht mit Websockets genutzt werden kann. Um die Kapselung so simpel wie möglich zu halten, wird ebenfalls auf HTTP 2 verzichtet, wodurch der abgewandelte HTTP-Client

nur HTTP1.1 unterstützt. Da im okhttp-Client der *HttpCodec* sowohl die Aufgabe erfüllt, den Request in einen HTTP-String umzuwandeln, als auch den Request über die Verbindung zu schreiben, muss so lediglich eine weitere Klasse angepasst werden. der *Http1Codec* wird dabei minimal angepasst, sodass interne Klassen der okhttp-Implementierung, die nicht im Rahmen dieser HTTP-Kapselung nötig sind, entfallen. Diese generische HTTP-Kapselung lässt sich nun ähnlich der okhttp-Implementierung über eine zentrale Klasse, den *SimpleHttpWrapper* nutzen. Diese Klasse nutzt ebenfalls das Builder-Pattern, um so nah wie möglich an der okhttp Bilbiothek zu bleiben. Über diesen Builder lässt sich nun ein *ConnectionStream* definieren, welcher dann den Inputstream und Outputstream der Verbindung bereitstellt. Die Verwaltung der p2p Verbindung sollte dann auch in diesem *ConnectionStream* gehandhabt werden, um die Inputstreams und Outputstreams im Fehlerfall schließen zu können.

Alg. 9: Änderungen an okhttp (Clientcode in Kotlin)

```
val iceptors = ArrayList<Interceptor>()
...
iceptors.add(BridgeInterceptor(wrapper.cookieJar()))
iceptors.add(CacheInterceptor(wrapper.internalCache()))
...
iceptors.add(SimpleServerInterceptor(wrapper.httpCodec()))
```

Im Alg. 10 ist zu sehen, wie eine *SocketImpl* dazu genutzt wird HTTP über eine p2p Verbindung zu senden, ohne Änderungen an okhttp vornehmen zu müssen. Dies ist dann möglich, wenn sich die genutzte Technologie in eine *SocketImpl* verpacken lässt, sodass eine *WrappingSocketFactory* im okhttp-Client genutzt werden kann. Da okhttp jedoch die Annahme trifft, dass die Sockets in einem IP-basierten Netzwerk Daten senden werden, wird zunächst eine DNS-Abfrage getätigt, um je nach IP-Route gesondertes Verhalten zu nutzen. Dies scheitert bereits, da es keinen zwingend existierenden Hostnamen außer *localhost* für das Android-Gerät existiert.

Alg. 10: HTTP-Kapselung als SocketFactory (Clientcode in Kotlin)

```
class WrappingSocketImpl(device: Device): SocketImpl() {
  // Delegiert o. Ignoriert alle Methoden zur Verbindung
  // Stellt InputStream und OutputStream bereit
}
class WrappingSocket(socketImpl: SocketImpl)
: Socket(socketImpl)

class WrappingSocketFactory(
  private val device: Device
): SocketFactory() {
  override fun createSocket(): Socket {
    return WrappingSocket(WrappingSocketImpl(device))
  }
}
```

Weiterhin tritt das Problem auf, dass für einen existierenden Host eine *Network-SecurityPolicy* des Android Gerätes befragt wird, ob Klartext zum Ziel erlaubt ist. Jenes schlägt für neuere Geräte mit Android 8 oder neuer ebenfalls fehl, auf Grund der Änderung, dass Klartext explizit für Domains freigegeben werden muss. Dies lässt sich zwar mit einer XML zur *network-security-config* (Alg. 11) umgehen, jedoch ist es sinnvoll hierbei eher eine verschlüsselte Datenübertragung in Erwägung zu ziehen. Der Versuch HTTPS zu nutzen, scheitert im weiteren Verlauf daran, dass die Socket-Verbindungen während dem Aufbau einer SSL Verbindung bereits auf Grund von Fehlern geschlossen werden. Okhttp ist jedoch flexibel genug, sodass die Socketverbindungen, welche für SSL erstellt werden, ebenfalls separat über eine *SSLSocketFactory* erstellt werden und so ebenfalls über eine p2p Technologie angebunden werden können.

Alg. 11: Sicherheitskonfiguration für erlaube Klartexthosts (Clientcode in XML)

```
<network-security-config>
  <domain-config cleartextTrafficPermitted="true">
    <domain includeSubdomains="false">localhost</domain>
  </domain-config>
</network-security-config>
```

4.3 Bluetooth

Die Umsetzung einer p2p Verbindung über Bluetooth besteht darin, dass ähnlich zu HTTP Sockets, RFCOMM-Sockets genutzt werden, um mit einem Server kurzweilig zu kommunizieren. Jede dieser RFCOMM-Verbindungen bildet in diesem Projekt eine HTTP Anfrage und HTTP Antwort ab. Wie bereits in der HTTP-Kapselung beschrieben, wird die Verbindung von selbst wieder geschlossen, sobald ein *EOF* gesendet wird. Da RFCOMM-Socketverbindungen ein automatisches Pairing mit Schlüsselaustausch durchführen, ist keinerlei Eingriff oder Bestätigung des Nutzers nötig, um Daten übertragen zu können.

4.3.1 Serverseite

Unter Linux steht die Bibliothek BlueZ in der Version 5 zur Verfügung, um Bluetooth Hardware nutzen zu können [BlueZ 5 Porting]. Diese Bibliothek bietet die Möglichkeit, RFCOMM-Sockets wie HTTP-Sockets in C zu nutzen, als auch über eine C und DBus Schnittstelle andere Funktionen wie das Service Discovery Protokoll zu verwenden.

Die Implementierung der HTTP-Kapselung kann im Server fast vollständig übernommen werden, jedoch müssen Verbindungen auf einem weiteren Socket mit dem Bluetooth RFCOMM-Protokoll akzeptiert werden. Mit dieser simplen Anbindungen im Alg. 12 von Bluetooth an die generische HTTP-Kapselung lässt sich eine einfache Lösung bereits nutzen. Um jedoch die Anbindung unter Android der offiziell unterstützten API anzugleichen, ist es nötig, auf dem Server die Anbindung

im Bluetooth SDP als Service zu hinterlegen. Kürzlich wurde die BlueZ 5 Bibliothek von einer simplen C-Schnittstelle auf eine DBus-Schnittstelle umgewandelt [BlueZ 5 Porting]. Dies hat zur Folge, dass viele der Beispiele und Erklärungen, ebenso wie Bücher nicht mehr aktuell sind und erst auf die neue API hingewiesen wird, wenn nach der expliziten Fehlermeldung gesucht wird.

Alg. 12: Verbindungsaufbau mit Bluetooth (Servercode in C)

```
int s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
struct sockaddr_rc loc_addr = { 0 };
loc_addr.rc_family = AF_BLUETOOTH;
loc_addr.rc_bdaddr = *BDADDR_ANY;
loc_addr.rc_channel = (uint8_t) bluetoothPort;
bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
listen(s, LISTEN_QUEUE_SIZE);
```

In älteren Versionen von BlueZ war es möglich einen SDP Eintrag mit der Methode *sdp_record_register* anzulegen (Alg. 13). Dieser Eintrag wurde von SDP selbstständig verwaltet und entfernt wenn die Sitzung zu SDP beendet wurde. Auf Grund des Wechsels zu DBus kann diese SDP-Schnittstelle nicht mehr genutzt werden, da sich keine Sitzungen zum SDP Daemon aufbauen lassen, da dieser nicht mehr existiert.

Alg. 13: Veraltete Nutzung von SDP (Servercode in C)

```
sdp_set_info_attr(record, serviceName,
  serviceProvider, serviceDescription);
*s= sdp_connect(BDADDR_ANY, BDADDR_LOCAL, SDP_RETRY_IF_BUSY);
sdp_record_register(*s, record, 0);
```

Um die SDP Funktionalität dennoch nutzen zu können, wird im Alg. 14 gezeigt, wie ein Callback-Objekt registriert wird. Dieses Objekt hat die Aufgabe, ankommende Verbindungen zu akzeptieren und zu nutzen. Es ersetzt somit den akzeptierenden Serversocket. Weiterhin muss das Objekt in der Lage sein, bestehende Verbindungen schließen zu können. Dies hat zur Folge, dass die Komplexität im Vergleich zu einer Lösung ohne SDP stark erhöht wird, da nicht nur die DBus Nachrichten auf einem separaten Thread gehändelt werden müssen, sondern auch eine Auflösung zwischen offenen Dateideskriptoren und den Client Kennungen stattfinden muss. Über diese gegebenen Dateideskriptoren konnte jedoch keine erfolgreiche Datenübertragung erzielt werden. Eine Verbindung wurde immer erfolgreich aufgebaut, jedoch schienen keine Daten tatsächlich übertragen zu werden, wodurch die Verbindung nach einem Timeout wieder geschlossen wurde.

Alg. 14: DBus Nutzung von SDP (Servercode in C)

```
static DBusHandlerResult wrapper_messages(
  DBusConnection* connection,
  DBusMessage* message,
  void* user_data);

DBusObjectPathVTable vtable;
vtable.message_function = wrapper_messages;
vtable.unregister_function = NULL;

DBusConnection* conn = dbus_bus_get(DBUS_BUS_SYSTEM, &err);
dbus_connection_try_register_object_path(conn, PROFILE_PATH,
  &vtable, NULL, &err);
DBusMessage* msg = dbus_message_new_method_call("org.bluez",
  "/org/bluez", "org.bluez.ProfileManager1",
  "RegisterProfile");
...
dbus_connection_send_with_reply_and_block(conn, msg,
-1, &err);
```

Ebenso ist es serverseitig nicht sinnvoll, die nötigen Sockets in pharo zu verwalten, wie es beispielhaft in Alg. 15 zu sehen ist. Dem liegt zu Grunde, dass Sockets in der pharo VM über das SocketsPlugin [Pharo Socket Plugin] gekapselt verwaltet werden und dort fest als IP basierte Sockets erstellt werden. Ein weiteres Plugin zu schreiben, welches Bluetooth Sockets erstellt, wäre denkbar, um die Verwaltung der Sockets dennoch im SocketsPlugin halten zu können. Hierbei entsteht jedoch ein sehr hoher Wartungsaufwand, da dieses Plugin sich an fremden Code bindet und dadurch Gefahr läuft, bei Änderungen in der VM nicht mehr zu funktionieren.

Alg. 15: Pharo Bluetooth Sockets als Plugin (Servercode in C)

```
void lib_bluez_bind_socket(SocketPtr s, int port) {
  struct sockaddr_rc loc_addr = { 0 };
  privateSocketStruct *pss= PSP(s);
  if (!socketValid(s)) return;

  loc_addr.rc_family = AF_BLUETOOTH;
  loc_addr.rc_bdaddr = *BDADDR_ANY;
  loc_addr.rc_channel = (uint8_t) port;
  if (bind(SOCKET(s), (struct sockaddr *)&loc_addr,
sizeof(loc_addr)) < 0) {
    pss->sockError= errno;
    success(false);
    return;
  }
}
```

Hierbei tritt eine weitere Schwierigkeit auf, alle nötigen Methoden und Datenstrukturen aus dem VM-Sourcecode richtig zu importieren, da jeder einzelne Eintrag der entsprechenden C-Header dem Compiler als extern angegeben werden muss. Eine Lösung, die Sockets in pharo verwaltet würde auch wieder Mehraufwand gegenüber anderen Lösungen bedeuten, da andere Technologien nicht zwingend ebenfalls Sockets unterstützen und somit ein Mehraufwand durch abweichende Implementierungen entsteht, wenn mehrere Lösungen parallel genutzt werden.

4.3.2 Clientseite

Um unter Android eine Bluetoothverbindung über RFCOMM aufbauen zu können, muss die Bluetooth MAC-Addresse sowie der Bluetooth Port angegeben werden. Diese beiden Parameter lassen sich, wie bereits bei einer Implementierung mit Wi-Fi Direct gezeigt [Kretzschmar], über die implementierte Service Discovery bereitstellen.

Der *BluetoothAdapter* aus dem Android Framework gibt ein *BluetoothDevice* für die angegebene MAC-Addresse zurück. Es ist hierbei noch unerheblich, dass dieses Gerät auch in der Nähe erreichbar ist oder existiert. Über die versteckten Methoden *BluetoothDevice::createInsecureRfcommSocket* und *::createRfcommSocket* kann eine Socket-Verbindung zu einem bestimmten Port des entfernten Gerätes erstellt werden. In Android sind diese Methoden versteckt, um Konflikte zwischen Apps beim festschreiben der Portnummern zu vermeiden, da lediglich 30 RFCOMM-Ports zur Verfügung stehen [BlueZ SDP]. Stattdessen sollen Services eine UUID generieren, welche über Bluetooth Service Discovery Protocol (SDP) von anderen Geräten abgefragt werden kann. Das SDP-Protokoll übernimmt dann die Vergabe von Ports für die registrierten Services [Android Bluetooth].

Um diese exemplarische Implementierung im Alg. 16 jedoch simpel zu halten und volle Kontrolle über das Servergerät besteht, wird hierbei ein vordefinierter Port genutzt. Jegliche Logik zum tatsächlichen Verbindungsaufbau ist in der Methode *BluetoothSocket::connect* gekapselt und muss bei der Umsetzung nicht beachtet werden. Auf beiden Geräten muss lediglich Bluetooth eingeschaltet sein und das Servergerät muss für den Client auffindbar sein.

Alg. 16: Verbindungsaufbau mit Bluetooth (Clientcode in Kotlin)

```
val bluetoothDevice = bluetoothAdapter.bondedDevices
  ?.firstOrNull { bluetoothDevice ->
    bluetoothDevice.address == bluetoothMac
} ?: bluetoothAdapter.getRemoteDevice(bluetoothMac)

val method = bluetoothDevice::class.java
  .getMethod("createInsecureRfcommSocket",
  Int::class.javaPrimitiveType)
val socket = method.invoke(bluetoothDevice, bluetoothPort)
  as BluetoothSocket
socket.connect()
```

4.4 NFC

Eine p2p Verbindung wird mit Near Field Communication so gestaltet, dass beide Seiten abwechselnd Daten über das gemeinsame Trägersignal übertragen können. Aufgrund der genutzten Hardware steht hierbei lediglich NFC-A (ISO 14443-3A) und als NDEF formattierte NFC Tags zur Verfügung.

4.4.1 Serverseite

Um NFC mit einem Raspberry Pi nutzen zu können, wird ein zusätzliches NFC/RFID Modul benötigt, welches an die GPIO-Pins des Einplatinenrechners angeschlossen wird. Für dieses Projekt wurde der Chip *NXP PN532* genutzt, da dieser bereits als vollständiges Breakout-Board angeboten wird und gleichzeitig die nötige ISO 14443-3A Kommunikation als Kartenleser unterstützt [NXP PN532 Datenblatt]. Die Bibliothek *libnfc* bietet eine Anbindung von NFC als C Bibliothek. Ähnlich zu Bluetooth wird hier eine Schnittstelle bereitgestellt, mit der die Kommunikation mit dem NFC Chip gehandhabt wird. Anders als Bluetooth werden jedoch keine Socketverbindungen direkt unterstützt. Stattdessen wird das Senden und Empfangen von Daten als *transceive* Methode zusammengefasst [NFC-Tools libnfc]. Für das Problem, HTTP über NFC zu übertragen, existiert bereits eine Lösung [NFC Sockets Blog], die unverändert übernommen werden kann. Socketverbindungen werden hierbei über NFC übertragen, indem die Methoden, welche auf den Clientsockets aufgerufen werden, als Nachrichten über NFC übertragen werden und ein Service im Server eigene Sockets anhand dieser Nachrichten verwaltet [NFC Sockets Blog]. Dieser Service kann Nachrichten parallel abarbeiten, da jedem Socket eine ID zugeordnet wird.

Pro erhaltenem Befehl wird im Alg. 17 immer ein neuer Thread gestartet, welcher den Befehl auf dem entsprechenden Socket abarbeitet und das Ergebnis der gegebenen Socketdatenstruktur anhängt. Aktuell ist dies für *connect*, *recv*, *send* und *close* möglich, um eine minimale Socketanbindung nutzen zu können.

Alg. 17: Datenempfang über NFC (Servercode in C) [NFC Sockets Server]

```
int handle_socket_recv_message(uint8_t *message,
size_t message_len) {
  int msg_id, fd, len;
  parse_socket_recv_message(message, message_len,
&msg_id, &fd, &len);
  struct socket_info *si = find_socket_info(fd);
  if (si == NULL) return -1;
  ...
  pthread_create(&si->thread_recv, NULL, recv_socket, si);
  pthread_detach(si->thread_recv);
}
```

Der Hauptthread, welcher den Datenempfang von NFC handhabt, kümmert sich auch um das Senden der Ergebnisse von Befehlen zum verbundenen NFC-Client. Die in der Socketdatenstruktur hinterlegten Daten werden im Alg. 18 wieder in

Nachrichten umgewandelt, die dann bei einem NFC *KEEP_ALIVE* Event gesendet werden. Das Senden und Empfangen von Daten passiert hierbei über die *transceive* Methode der *libnfc*, weshalb ausstehende Nachrichten hier zunächst in den statischen Buffer *cmd_apdu* verschoben werden.

Alg. 18: Senden von Daten über NFC (Servercode in C) [NFC Sockets Server]

```
void handle_pending_recv(socket_info *si,
uint8_t *cmd_apdu, size_t *cmd_apdu_len) {
  int res = *(si->pending_recv_res);
  if (res > 0) make_socket_recv_response(cmd_apdu,
cmd_apdu_len, si->pending_recv, res, si->pending_recv_data);
  else make_socket_recv_response(cmd_apdu, cmd_apdu_len,
si->pending_recv, res, NULL);
}
```

4.4.2 Clientseite

Die vorgestellte Lösung zeigt ebenfalls bereits eine Clientimplementierung für Android. Diese nutzt auch *okhttp* und eine *SocketFactory*, um die Socketverbindung über NFC zu übertragen. Der Alg. 19 zeigt als Auszug, dass Nachrichten über einen Eventbus im Socket empfangen werden. Im *InputStream* des Sockets wird dann darauf gewartet, dass eine Nachricht mit einer passenden ID aus dem Eventbus in eine Instanzvariable geschrieben wird.

Alg. 19: Datenempfang über NFC (Clientcode in Java) [NFC Sockets Client]

```
@Subscribe @Synchronized
public void onRecvResponse(RecvResponse recvResponse) {
  if (pendingRecv != null && recvResponse.getInReplyTo() ==
pendingRecv.getRequestId()) {
    this.recvResponse = recvResponse;
    notifyAll();
  }
}
```

Diese starke Nutzung eines Bussystems deutet jedoch an, dass die Architektur der Lösung nicht optimal ist. Der explizite Service zum Senden und Empfangen von Daten kann beibehalten werden, jedoch sollten statt den Identifikatoren in einem Bus asynchrone Datentypen wie *Futures* oder *Observables* an den Service übergeben werden. Dadurch wird es nötig, die Zuweisung von Anfrage und Antwort im Service z.B. als *HashMap* mit schwachen Referenzen durchzuführen, jedoch ist es so nicht mehr nötig, alle Daten über einen Bus senden zu müssen. Für eine Lösung mit rxJava entfällt so die Definition eines Filters der Nachrichten in den Zuhörern des Busses. Eine solche Implementierung ist in Alg. 20 skizziert.

Alg. 20: Datenempfang über Observables (Clientcode in Kotlin)

```
val pending: HashMap<Int, WeakReference<Subscriber>>()

fun send(request: Message): Observable<Message> {
  return Observable.unsafeCreate { subscriber ->
    pending.add(message.requestId(), subscriber)
    ...// Senden der Nachricht wie zuvor
  }
}
fun onReceiveMessage(message: Message) {
  pending.remove(message.inReplyTo())?.get()?.let {
    it.onNext(message)
    it.onCompleted()
  }
}
```

4.5 USB

Da noch jedes Android Smartphone einen USB-Port besitzt, soll dieser für eine kabelgebundene p2p Verbindung betrachtet werden. USB kann unter Linux über das Android Open Accessory (AOA) Protokoll angesprochen werden [Android Open Accessory]. Von der Android Website lassen sich jedoch keine weiteren Informationen mehr bezüglich diesem Protokoll abrufen. Ebenso kann das dort erwähnte Accessory Development Kit *ADK2012* nicht mehr heruntergeladen werden. Einzig ein älterer Klon bietet noch weitere Informationen zu diesem Kit [MIT ADK], welches dazu gedacht ist, eine Beispielimplementierung von USB Geräten für Android bereitzustellen. Da dieses Kit jedoch seit 2012 keine Updates mehr erhalten hat, ist seine zukünftige Lebenszeit zu beobachten. Die USB Technologie scheint auf Grund der mangelnden Unterstützung seitens Google nicht mehr lange weiter zu existieren. In Verbindung mit Android 8.0 wurde die Nutzung von Accessories für Audio-Übertragungen eingestellt [Android Open Accessory].

4.5.1 Serverseite

Wie bei *NFC* kann auch Rfür USB eine bereits bestehende Lösung als weiterer Service in den Server integriert werden [AOA Proxy Server]. Dieser Service spricht jedes Androidgerät über das AOA Protokoll an und erlaubt diesen Geräten eine Socketverbindung zu einem lokalen Port des Servers aufzubauen. Dadurch ist ebenfalls eine Kapselung außerhalb des HTTP-Servers gegeben. Intern verwendet dieser Service wieder eine Bibliothek [libusb], um die Kommunikation über USB vorzunehmen. Diese Bibliothek erlaubt das Senden und Empfangen von Daten in den bereits vorgestellten Modi von USB. Um AOA nutzen zu können, wird Bulk-Transfer mit der vorausgegangenen Nutzung des Control-Transfers zur Identifikation genutzt.

Alg. 21: Verbindungsaufbau über USB (Servercode in C) [AOA Proxy Server]

```
libusb_open(usbDevice, &device->usbHandle);
libusb_claim_interface(device->usbHandle, device->bulkIface);
libusb_fill_bulk_transfer(device->xfr, device->usbHandle,
device->endpointAddr, buffer, sizeof(buffer), NULL, NULL, 0);
```

Jedes angeschlossene Android Gerät wird zunächst wie eine Datei über Methoden der Bibliothek geöffnet und eine Datenübertragung durchgeführt (Alg. 21). Das Senden und Empfangen von Daten ist dabei davon abhängig, welcher Befehl der Methode *libusb_fill_bulk_transfer* übergeben wird. Je nach Methode werden Daten vom Buffer geladen oder in den gegebenen Buffer geschrieben.

Das Anschließen von Geräten wird durch Hotplugging-Events erkannt. Sobald eine neue Function an die USB Schnittstelle angeschlossen wird, kann eine Anwendung oder Treiber über einen Callback *libusb_hotplug_register_callback* über neue Geräte informiert und so eine Anwendungsverbindung zu diesen beginnen. Um AOA dann nutzen zu können, wird das USB-Gerät über einen URI-String, welcher über Control-Transfer gesendet wird, über die Art von Gerät informiert.

4.5.2 Clientseite

Da USB ein System ist, welches lediglich einen einzelnen Host der Verbindung erlaubt, muss das Smartphone als Function dem Bus beitreten. Die App muss in der Lage sein, die Kontrolle über die Kommunikation mit verbundenen USB Geräte zu übernehmen, um Daten zu einem der Geräte übertragen zu können. Da der Raspberry Pi als Host der USB-Verbindung agiert, werden unter Android *UsbAccessory* genutzt und keine *UsbDevice*. Unter Android wird USB Funktionalität über den Systemservice *UsbManager* bereitgestellt. Dieser verwaltet, wie in Alg. 22 zu sehen, eine Liste von verbundenen USB-Geräten. Um AOA nutzen zu können, agiert Android als Client der USB-Verbindung, wodurch das erste USB-Gerät dieser Liste genutzt werden kann. Dieses Gerät wird dann ähnlich zur Serverseite als Datei geöffnet und kann über einen InputStream und OutputStream eine bidirektionale Datenübertragung vornehmen.

Alg. 22: Verbindungsaufbau über USB (Clientcode in Kotlin)

```
val usbMan = context.getSystemService(Context.USB_SERVICE)
val fd = usbMan.openAccessory(usbMan.accessoryList.first())
val inputStream = FileInputStream(fd.fileDescriptor)
val outputStream = FileOutputStream(fd.fileDescriptor)
```

Um langlebige Verbindungen wie USB abbilden zu können, muss, wie in Alg. 23 beschrieben, ein Stream definiert werden, welcher lediglich eine Fassade eines anderen Streams darstellt und sich selbst schließt ohne den verpackten Stream zu beeinflussen, sodass dieser nicht von der Anwendungsseite aus geschlossen werden kann.

Alg. 23: Socketwrapper für USB (Clientcode in Kotlin)

```
class DetachableInputStream(
  private var attachedInputStream: InputStream?
) : InputStream() {
  override fun close() {
    attachedInputStream = null
    super.close()
  }
  override fun read(): Int {
    return attachedInputStream?.read()
?: throw IOException("closed")
  }
}
```

Für die vorgestellte Lösung über AOA konnte keine erfolgreiche Datenübertragung vollzogen werden. Zwar wurde eine Verbindung korrekt aufgebaut, jedoch ließen sich keine Daten über den gegebenen Socket empfangen oder senden. Alle Daten, die clientseitig dem USB-Gerät gesendet wurden, konnten serverseitig nicht empfangen werden, wodurch die Verbindung immer mit einer Zeitüberschreitung geendet ist. Dies lässt sich zum einen dadurch erklären, dass AOA zuletzt im Jahr 2012 aktiv entwickelt wurde und somit die Technologie inzwischen ungetestete Abweichungen zwischen der nutzbaren Schnittstelle und der Hardware bestehen oder weiterer undokumentierter Konfigurationsaufwand auf Seiten von Android vorgenommen werden muss. Beide Probleme lassen sich nicht ohne funktionstüchtige Beispiele beheben oder umgehen, wodurch auf eine weitere Implementierung über USB verzichtet wurde.

5 Ergebnisse der Evaluation

Durch die Betrachtung der Umsetzungsmöglichkeiten der einzelnen p2p Technologien kann nun eine Evaluation auf Grundlage der definierten Ziele vorgenommen werden. Zum Vergleich wird Wi-Fi Direct als bereits implementierte Technologie genutzt, um einen Bezug zur bestehenden Lösung aufbauen zu können.

5.1 Zuverlässigkeit

Die Robustheit der einzelnen Technologien bei Fehlern und die Reproduzierbarkeit von Verbindungen bilden zusammen das Kriterium der Zuverlässigkeit. Wie bei Wi-Fi Direct bereits deutlich wurde, sind diese beiden Punkte nicht zwingend für jede Technologie ausreichend gegeben.

5.1.1 Wi-Fi Direct

Eine hohe Fehleranfälligkeit in Kombination mit einer schlechten Fehlererholung durch Abstürze des laufenden Wi-Fi Daemon führen zu einer unzureichenden Zuverlässigkeit von Wi-Fi Direct. Ebenso konnte beobachtet werden, dass die internen Zustände der genutzten Bibliothek weder dokumentiert sind, noch waren diese Zustände sauber voneinander gekapselt, wodurch der Erfolg eines Verbindungsaufbaus von vorhergehenden Verbindungen abhängt. Die internen Zustandswechsel konnten zwar über Events der Bibliothek beobachtet werden, jedoch wurden daraus die Menge der möglichen Zustände und deren Übergänge nicht deutlich, da nicht zwingend jedes Event einem neuen Zustand entspricht. Die Dokumentation des *wpa_supplicant* versäumt es ebenfalls zu beschreiben, in welcher Reihenfolge ein erfolgreicher Verbindungsaufbau vollzogen wird, wodurch nicht sichergestellt werden kann, dass die eigene Anbindung der erwarteten Nutzung entspricht. Zwar steht der Sourcecode der *wpa_supplicant* Bibliothek und damit auch die Schnittstelle als Sourcecode zur Verfügung, da die Schnittstelle über Nachrichten statt über Methoden genutzt wird, ist es schwierig zu erkennen, welche Teile des Sourcecodes relevant sind. Es kommt hierbei erschwerend hinzu, dass nicht jede Bibliothek mit den gleichen Parametern gebaut wird, wodurch nicht zwingend der volle Funktions-umfang zur Verfügung steht. Ebenso muss dafür gesorgt werden, dass sich das Wi-Fi Gerät im zuhörenden Modus befindet und somit auf Anfragen reagiert. Sobald ein Befehl auf einer anderen Schnittstelle ausgeführt wurde, verlässt das Wi-Fi Gerät den zuhörenden Modus, wodurch eine weitere Nutzung erst dann möglich ist, wenn das Gerät wieder explizit zum Zuhören auf der Wi-Fi Schnittstelle gezwungen wird.

5.1.2 Bluetooth

Im Gegensatz dazu konnte mit Bluetooth und der genutzten Bibliothek ein Verbindungsaufbau mit reproduzierbaren Ergebnissen erreicht werden. Durch das integrierte Hilfsmittel *bluetoothctl* der BlueZ Bibliothek kann die Menge der bestehenden Zustände gut beobachtet werden. Die Benennung der einzelnen Zustände orientiert sich hierbei an der theoretischen Spezifikation von Bluetooth, wodurch auch

die Zustandsübergänge bereits klar werden. Eine feinere Betrachtung der Zustände ist nicht nötig, da die Bluetooth Bibliothek jeglichen Zustand der Verbindung selbstständig verwaltet und dies der Anwendung als Serversocket zur Verfügung stellt. Fehler werden somit als Schließen des Sockets oder Fehler des Verbindungsaufbaus bekannt gemacht. Daraus lässt sich nicht der exakte Grund ableiten, jedoch besitzt Bluetooth aufgrund des sauber definierten Zustandsmodells und durch Socketverbindungen auch die BlueZ Bibliothek eine solche Fehlerstabilität, dass die Verbindung lediglich neu aufgebaut werden muss [Bluetooth Spezifikation]. Für eine Bluetooth-Verbindung muss jedoch auch sichergestellt werden, dass das Gerät über Bluetooth aufgefunden werden kann.

5.1.3 NFC

Wenn ein bereits bestehender Wrapper genutzt wird, besteht das Problem, dass nicht ersichtlich ist, wann Fehler auftreten können. Die Kapselung auf Socketverbindungen für NFC ermöglicht es jedoch auch, eine Behandlung von Fehlern durch das Schließen bestehender Verbindungen durchzuführen und eine Verbindung erneut aufzubauen. Die internen Zustände eines NFC Verbindungsaufbaus lassen sich nicht direkt über die genutzte Bibliothek *libnfc* beobachten, jedoch kann über das Hilfsmittel *nfc-poll* ein NFC Gerät und die ausgetauschten Befehle ausgelesen werden. Eine Reproduzierbarkeit von Verbindungen ist durch das von NFC genutzte Trägersignal zwar gegeben, jedoch wird in der genutzten Implementierung ein hochfrequentes *Keep-Alive* Signal genutzt, damit die Verbindung aufrecht erhalten wird.

5.1.4 USB

Ähnlich zu NFC schränkt die bestehende Implementierung von USB das Fehlerverhalten der zugrunde liegenden *libusb* Bibliothek ein und bündelt dieses ebenfalls über eine Socketverbindung. Da der Verbindungsaufbau von USB-Geräten bereits vom Kernel übernommen wird, können Zustandsübergänge an neuen Geräten über *udev* beobachtet werden. Eine Fehleranfälligkeit der USB-Verbindung ergibt sich aus den genutzten Treibern. Da jedoch das USB-Gerät ohne Treiber mit *libusb* angesprochen wird, besteht eine sehr geringe Fehleranfälligkeit einer USB-Verbindung, da diese kabelgebunden ist und sich erst eine Fehleranfälligkeit aus den genutzten Treibern oder einer Abweichung von der USB Spezifikation ergibt [USB Spezifikation]. Die Spezifikation sieht eine Fehlererholung so vor, dass der Host der Verbindung laufende Datentransfers steuern kann. Diese Fehler werden erst propagiert, wenn sie nicht erfolgreich abgefangen werden konnten. Diese Fehler tauchen dann lediglich im Log des Kernels auf und resultieren in einem Zurücksetzen der USB-Verbindung über das Hotplugging von USB.

5.2 Wartbarkeit

Fehler in der Nutzung einer Technologie lassen sich nur dann leicht beheben, wenn die Dokumentation so vollständig ist, dass sie den betroffenen Nutzungsfall abdeckt

und somit eine Abweichung der Anbindung von der Dokumentation aufgedeckt werden kann. Sollte die Dokumentation jedoch nicht ausreichend auf diesen Fall eingehen, muss der Quellcode der angebotenen Schnittstelle zur Verfügung stehen und leicht zu verstehen sein, sodass die Abweichung leicht gefunden werden kann. Sollten Tests für die genutzten Bibliotheken im Sourcecode vorhanden sein, verbessert dies die externe Wartbarkeit ebenfalls, da so zum Einen bereits eine Abstraktion der Hardware besteht und diese für eigene Tests genutzt werden kann und zum Anderen die Nutzung der Bibliothek anhand der getesteten Nutzungsfälle exemplarisch gezeigt wird.

5.2.1 Wi-Fi Direct

Die Dokumentation von Wi-Fi Direct ist bei der Wi-Fi Association nicht öffentlich zugänglich, jedoch enthält die Beschreibung der *wpa_supplicant* Implementierung eine Aufstellung einiger Ereignisse, die bei der Nutzung auftreten können. Fehler beim Verbindungsaufbau oder während der Verbindungsnutzung werden zwar als solche Ereignisse der Anbindung mitgeteilt, jedoch ließen sich auch eine mangelnde Fehlerrobustheit durch Abstürze bei einigen dieser Ereignisse feststellen. Tests sind im *wpa_supplicant* für die einzelnen Schritte eines p2p Verbindungsaufbaus als Python-Skripte vorhanden und basieren auf einer simulierten Hardware. Diese simulierte Hardware lässt sich somit auch für eigene Tests nutzen, um die eigene Implementierung mit möglichst geringen Abänderungen testen zu können.

5.2.2 Bluetooth

Eine Anbindung an Bluetooth geschieht über Sockets, welche bei der Erstellung durch das Austauschen einiger Parameter bereits eine testbare Umgebung bereitstellen können. Weiterhin wird die Spezifikation von Bluetooth durch die Bluetooth SIG öffentlich einsehbar bereitgestellt und beschreibt ausführlich interne Zustände einer Bluetooth Implementierung sowie die Testbarkeit eines Bluetoothgerätes [Bluetooth Spezifikation]. Die BlueZ Bibliothek steht ebenfalls als Referenz zur Verfügung und enthält in den Tests ebenfalls eine Beispielanbindung für die Nutzungsfälle der Bibliothek jedoch keine Virtualisierung der Hardware. Dies bedeutet, dass für einen vollständigen Integrationstest mindestens zwei Bluetoothschnittstellen am testenden Gerät vorhanden sein müssen, da ein Loopback ebenfalls nicht möglich ist.

5.2.3 NFC

Die existierende Lösung nutzt ebenfalls Sockets um eine Kommunikation aufbauen zu können, somit kann auch hier die eigene Implementierung getestet werden, indem dieser Serversocket angesprochen wird. Um auch die Kapselung der NFC Bibliothek testen zu können, ist es nötig, die zugrundeliegende Bibliothek *libnfc* zu betrachten. In ihr finden sich sowohl Beispiele für alle unterstützten Nutzungsfälle des genutzten NFC Chips als auch Tests der Bibliothek. Eine Dokumentation der Bibliothek als

solche existiert nur minimal, jedoch können die Beispiele genutzt werden eine Implementierung vorzunehmen. Eine Konfiguration von Geräten, die mit der Bibliothek genutzt werden sollen, ist nötig, da die NFC Bibliothek UART zur Kommunikation nutzt und Geräte somit nicht automatisch als NFC-Modul erkannt werden. Ein solcher Konfigurationsaufwand verschlechtert auch die Wartbarkeit der Anbindung, da diese Anbindung nicht ohne die Konfiguration funktionieren kann und sich nicht testen lässt, ob eine Konfiguration für alle Anwendungsfälle korrekt ist oder nur für einen getesteten Anwendungsfall richtiges Verhalten zeigt, da eine Dokumentation über die Konfigurationsmöglichkeiten nicht gegeben ist.

5.2.4 USB

Eine Spezifikation über USB kann öffentlich eingesehen werden, jedoch umschließt diese nicht nur die Definition eines Übertragungsprotokolls sondern auch die vollständige Definition der Hardwareschnittstelle, wodurch die für das Protkoll relevanten Teile schwer auffindbar sind und die Spezifikation dadurch einen geringeren Nutzen im Vergleich zu den anderen Technologien aufweist [USB Spezifikation]. Socketverbindungen werden auch hierbei von der genutzten Implementierung für die Dauer der USB-Verbindung aufgebaut und im nicht erholbaren Fehlerfall geschlossen. Intern werden Daten über explizit blockierende Methoden gesendet und empfangen, welche auch Fehler der Verbindung als Resultat ausliefern. Jeder dieser Fehler ist in der Dokumentation der *libusb* dokumentiert und gibt Aufschluss darüber, wodurch dieser Fehler aufgetreten ist. Die Bibliothek kann ebenfalls im Quellcode eingesehen werden und enthält Beispiele sowie Tests. Einige Beispiel werden ebenfalls gleichzeitig als Tests genutzt, jedoch ist zu beobachten, dass nur sehr grundlegende Tests der Bibliothek durchgeführt werden und keine Tests existieren, die eine vollständige Datenübertragung über ein virtuelles USB-Gerät überprüfen. Da das AOA Protokoll implementiert wird, ist es auch nicht möglich, einen Test ohne ein Android Gerät zu vollziehen, da nicht geprüft werden kann, dass die im Protokoll gesendeten Daten auch im vorgesehenen Format gesendet wurden. Testbarkeit und damit auch Wartbarkeit ist somit nur bedingt für eine USB-Schnittstelle gegeben.

5.3 Funktionalität

Da sich der genutzte Funktionsumfang nicht zwischen den Technologien unterscheidet, ist es schwierig für den Punkt der Funktionalität ein zu überprüfendes Kriterium zu definieren. Daher wird hier auf die Dokumentation und Erreichbarkeit der internen Zustände der Technologie eingegangen. Diese Zustände können vereinfacht als die Zustände einer p2p Verbindung, wie in Fig. 1 beschrieben, betrachtet werden. Ebenso ist im Rahmen der Funktionalität zu betrachten, wie leicht sich der nötige Funktionsumfang über eine der Technologien bereitstellen lässt.

5.3.1 Wi-Fi Direct

Die internen Zustände von Wi-Fi Direct lassen sich nur bedingt auf Wi-Fi Direct abbilden, da der Verbindungsaufbau zu einem Gerät nicht vollständig automatisch

stattfinden kann und somit die Zustände *ACCEPTING* und *CONNECTING* sich nicht als jeweils ein atomarer Zustand abbilden lassen. Ebenso ist es der Zustand *CONNECTED* unvollständig, da nicht beiden Verbindungsteilnehmern die IP-Adresse des Gegenüber mitgeteilt wird und diese über einen weiteren Kanal übertragen werden muss. Wi-Fi Direct stellt eine p2p Verbindung bereit, indem ein Netzwerkadapter für eine ad-hoc Gruppe angelegt wird. Dieser Netzwerkadapter hat Zugriff auf die Sockets des verbundenen Gerätes und spricht dieser über IP Pakete an. Da die Sockets direkt angesprochen werden können, entfällt für diese Technologie ein HTTP Wrapper, wodurch die Anbindung eher einer Konfiguration als einer Implementierung entspricht. Um diese Konfiguration vornehmen zu können muss jedoch die externe Schnittstelle des *wpa_supplicant* angebunden werden. Da diese auf Nachrichten basiert muss nur eine geringe Menge an Methoden implementiert werden, jedoch muss die Menge der möglichen Nachrichten auch unterstützt werden. Nicht jede Nachricht, die genutzt wird, ist hierbei dokumentiert, wodurch der Aufwand, die nötige Funktionalität anzubieten, sich erhöht. Um eine Verbindung erfolgreich aufbauen zu können, müssen nötige Events abgefragt werden und aktiv im Verbindungsaufbau mitgewirkt werden. Dadurch wird der Verbindungsaufbau unzuverlässig und somit auch aufwändiger umzusetzen.

5.3.2 Bluetooth

Gegenüber Wi-Fi Direct ist die Implementierung von Bluetooth so stark gekapselt, dass ein Verbindungsaufbau in den Zuständen *ACCEPTING* und *CONNECTING* dargestellt werden kann. Nachdem beide Geräte in den Zustand *CONNECTED* gewechselt haben, können beide Geräte über Sockets miteinander kommunizieren. Bluetooth ist so vollständig durch das simple Zustandsmodel von p2p Verbin-dungen abgebildet und kann dessen Funktionsumfang in vollem Umfang bereitstellen. Da Bluetooth Sockets verwendet, um eine p2p Verbindung zu realisieren, können lediglich Standardmethoden eines jeden Linuxsystems genutzt werden, um erfolgreich eine Verbindung herzustellen. Sockets sind darauf ausgelegt, eine simple Implementierung von Interprozesskommunikation zur Verfügung zu stellen, was sich bei dieser Implementierung zunutze gemacht wurde.

5.3.3 NFC

NFC nutzt ein Trägersignal, um Daten zu übertragen. Dieses Trägersignal wird selbstständig auf- und abgebaut, und stellt den Zustand *CONNECTED* dar. Eine Kommunikation auf Seiten von Android findet über einen zuvor definierten Service statt, welcher angesprochen wird, wenn auf einen Befehl reagiert werden muss. Es ist dabei aber nicht ersichtlich, wie der Zustandsübergang zwischen *IDLE* und *CONNECTED* gehandhabt wird. Somit sind die Zustände *ACCEPTING* und *CONNECTING* durch NFC nicht nach außen hin abgebildet. Dadurch wird es komplizierter, asynchrone Daten über diese Schnittstelle zu senden, da die Daten in einer Schlange vorgehalten werden müssen und die Verbindung über leere Datenpakete aufrecht erhalten werden muss. Die verbleibende Implementierung besteht daraus, Datenpakete von der NFC Bibliothek abzufragen und auf diese zu reagieren indem

Sockets verwaltet werden. Da jedoch für diese Schritte eine existierende Lösung genutzt werden konnte, besteht kein Implementierungsaufwand, um die nötige Funktionalität anzubieten.

5.3.4 USB

Da USB als einzige vorgestellte Technologie ein kabelgebundenes Medium ist, entspricht im Zustandsdiagramm einer p2p Verbindung der Zustand *IDLE* gleichzeitig dem Zustand *ACCEPTING* auf einem Hostgerät. Auf dem Clientgerät wird durch das Verbinden mit dem Host über das Kabel eine Spannung übertragen und das Gerät wechselt intern zum Zustand *CONNECTING* um sich beim Host zu registrieren. Sobald das Hotplugging-Event auf dem Hostgerät erhalten wurde, wird es geöffnet und über AOA eine Verbindung hergestellt. Danach befindet der Host sich im Zustand *CONNECTED* und kann vom Client als USB Accessory angesprochen werden. Das USB Accessory besitzt dabei Inputstream und Outputstream somit befindet sich auch der Client im Zustand *CONNECTED*. Um die p2p Funktionalität bereitstellen zu können, ist es nötig, die Bibliothek *libusb* so anzubinden, dass Hotplugging-Events abgefangen werden und eine blockierende Kommunikation über die Bibliothek stattfindet. Um AOA noch zu implementieren, muss der Host zwei USB Befehle implementieren, um eine gemeinsame Protokollversion zu finden und sich dem Smartphone gegenüber zu identifizieren. Es wurde versucht eine bestehenden Lösung zu nutzen, welche eine eigene Implementierung überflüssig gemacht hätte, jedoch konnte kein erfolgreiches Ergebnis mit dieser Lösung erzielt werden, wodurch eine eigene Implementierung doch nötig ist.

5.4 Effizienz

Der Begriff der Effizienz setzt sich aus Datenraten der Verbindung sowie möglichen Differenzen in der Antwortzeit aufgrund der genutzten Technologie zusammen. Da für drahtlose Netzwerkverbindungen mittlere Antwortzeiten jedoch stark von Interferenzen abhängen und sich keine reproduzierbaren Ergebnisse erzielen ließen, kann über diesen Punkte keine Aussage getroffen werden. Da reale Datenraten stark von Interferenzen und anderen Störfaktoren abhängig sind, werden lediglich die theoretischen Datenraten gegenüber gestellt. Gemessen werden die Datenraten in Kilobit pro Sekunde (kbps).

5.4.1 Wi-Fi Direct

Die mögliche Übertragunsrate von Wi-Fi Direct hängt von den genutzten Geräten ab und welcher Teil des Standards *IEEE 802.11* von ihnen unterstützt wird. Theoretisch sind dabei Datenraten bis zu 250.000kbps möglich [Wi-Fi Direct Rates]. Da nicht aktiv in Verbindungen eingegriffen wird, entsteht hierbei auch kein Overhead der Anwendungsseite für die Übertragung.

5.4.2 Bluetooth

Der genutzte Raspberry Pi verwendet Bluetooth in der Version 4.2. Somit kann eine Bluetooth Verbindung basierend auf *IEEE 802.11a* aufgebaut werden. Klassisches Bluetooth leistet jedoch bloß eine theoretische Datenrate von 723kbps [Sauter, S.377].

5.4.3 NFC

Theoretisch unterstützt NFC die Datenraten 106kbps, 212 kpbs und 424kbps [NFC Rates], jedoch können diese Datenraten einer Verbindung über NFC mit einem Android Telefon nicht erreicht werden, da Android keine erweiterten Tag-Pakete unterstützt [NFC Google Bug] und so nur 255 Byte Nutzdaten pro Paket möglich sind. Hierdurch erhöht sich Overhead für Befehle im Vergleich zu Nutzdaten von 0,01% auf 2,68% deutlich und die Nutzdaten einer Socket Verbindung erhalten noch weiteren Overhead, da Befehle zur Steuerung der externen Sockets ebenfalls zur Verfügung stehen müssen.

5.4.4 USB

Im verwendeten Raspberry Pi wird USB 2 unterstützt, wodurch USB eine Datenrate von 12.000kbits im *Full Speed* Modus und 480.000kbits im *High Speed* Modus [Axelson, S.36]. Die Verwendung von AOA schränkt diese Datenraten nicht ein, da über dieses Protokoll lediglich ausgetauscht wird, welches Gerät an das Android Smartphone angeschlossen wurde, damit eine passende Anwendung ausgewählt werden kann.

5.5 Bedienbarkeit

Im Rahmen der Bedienbarkeit soll der Aufwand betrachtet werden, den der Nutzer aufbringen muss, um eine Verbindung über die Technologien erfolgreich aufbauen zu können. Dieser Aufwand lässt sich daran messen, wie viele Aktionen der Nutzer ausführen muss, bevor eine p2p Verbindung besteht.

5.5.1 Wi-Fi Direct

Für eine Verbindung über Wi-Fi Direct muss die Wi-Fi Schnittstelle des Telefons eingeschaltet sein, was sich mit der entsprechenden Berechtigung ohne Nutzerinteraktion erreichen. Im nächsten Schritt wählt der Nutzer, wie bei allen Technologien, das zu verbindende Gerät aus. Um eine Verbindung zu dem gewählten Gerät aufzubauen, muss der Nutzer in einem Dialog des Systems bestätigen, dass der Verbindungsaufbau erwünscht ist. Sobald der Nutzer dies bestätigt hat, wird der Verbindungsaufbau abgeschlossen und beide Geräte sind in einem Netzwerk miteinander verbunden. Es ist also eine einzelne Aktion des Nutzers für eine Wi-Fi Direct Verbindung von Nöten.

5.5.2 Bluetooth

Bevor eine Bluetooth Verbindung aufgebaut werden kann, muss das Android-Gerät Bluetooth aktiviert haben. Wenn der Nutzer der App Administrator Rechte über Bluetooth erlaubt hat, kann dies bereits automatisch geschehen. Bluetooth muss aktiv sein, damit zum einen Service Discovery über Bluetooth LE stattfinden kann und eine p2p Verbindung aufgebaut werden kann. Der eigentliche Aufbau der p2p Verbindung geschieht automatisch zwischen den beiden Bluetooth-Stacks ohne einen Eingriff des Nutzers, da ein temporäres Pairing für die Dauer der Verbindung vorgenommen wird. Der Nutzer muss somit keine Aktionen durchführen, um eine p2p Verbindung zu einem Gerät über Bluetooth aufzubauen.

5.5.3 NFC

Um NFC nutzen zu können, muss auch hier der Nutzer wieder sicherstellen, dass NFC in den Einstellungen des Smartphones eingeschaltet ist, jedoch kann dies nicht ohne Eingriff des Nutzers geändert werden. Ebenso muss der Nutzer für die Dauer der Datenübertragung beide Geräte auf einer Distanz von unter 10 Zentimetern halten, was beim aktuellen Nutzerfluss der Verbindungskonfiguration mindestens zwei Mal geschehen muss, da davon auszugehen ist, dass die Verbindung geschlossen wird, sobald der Nutzer mit der App interagieren muss.

5.5.4 USB

Die Verbindung über USB erfordert zunächst, dass das Android-Gerät über ein maximal 5 Meter langes USB Kabel durch den Nutzer mit dem Raspberry Pi verbunden wird [Eberhardt et al., S.22]. Zusätzlich muss der Nutzer meistens auf neueren Android-Betriebssystemversionen die USB-Verbindung zur Datenübertragung erlauben. Sobald dies geschehen ist, öffnet sich ein weiterer Dialog, in dem der Nutzer darauf hingewiesen wird, dass ein Android Accessory angeschlossen wurde und für dieses eine App ausgewählt werden soll. Beide dieser Nutzerinteraktionen brechen den Nutzerfluss durch die App drastisch, da Systemdialoge die Benutzung der App vollständig unterbrechen.

5.6 Zusammenfassung

Generell lässt sich sagen, dass jede der vorgestellten Technologien im Hinblick auf die überprüften Kriterien einen anderen Schwerpunkt besitzt. Fig. 2 zeigt eine Aufstellung der einzelnen Technologien im Hinblick auf die geprüften Evaluationskriterien. Zwar lässt sich keine qualitative oder quantitative Bewertung für die einzelnen Werte treffen, wodurch die Verhältnisse zwischen den einzelnen Technologien nicht repräsentativ sind, jedoch lassen sich anhand dieses Graphen leicht die Stärken und Schwächen der p2p Übertragungsmedien gegenüberstellen.

Bluetooth weißt einen hohen Nutzen im Hinblick auf Funktionalität, Wartbarkeit und Bedienbarkeit auf. Insgesamt zeigt sich so, dass Bluetooth als p2p Technologie am Besten geeignet ist, da sie auch gute Zuverlässigkeit bietet.

Wenn Bluetooth nicht genutzt werden kann, sollte als Nächstes NFC in Be-tracht gezogen werden. Es besitzt zwar die schlechteste Effizienz aller Technologien, jedoch erfüllt es die Kriterien der Funktionalität und Wartbarkeit immer noch gut. Es besitzt eine höhere Zuverlässigkeit gegenüber den anderen Technologien wird auch die Benutzbarkeit durch die starke Nutzereinbindung in Mitleidenschaft gezogen.

Effizienz ist für den Anwendungsfall der Verbindungskonfiguration, welche eine REST-Schnittstelle darstellt, eher unerheblich, da alle Technologien, sobald eine Verbindung aufgebaut wurde, geringe Datenmengen schnell genug übertragen können. USB stellt für dieses Kriterium die beste Wahl dar, sollte dieser Punkt für ein anderes Evaluationsziel relevant sein.

Wi-Fi Direct kann in keiner der geprüften Kategorien hervorstechen und zeigt lediglich eine gute Effizienz und Bedienbarkeit. Für alle anderen Evaluationspunkte ist davon abzuraten, Wi-Fi Direct als p2p Technologie einzusetzen.

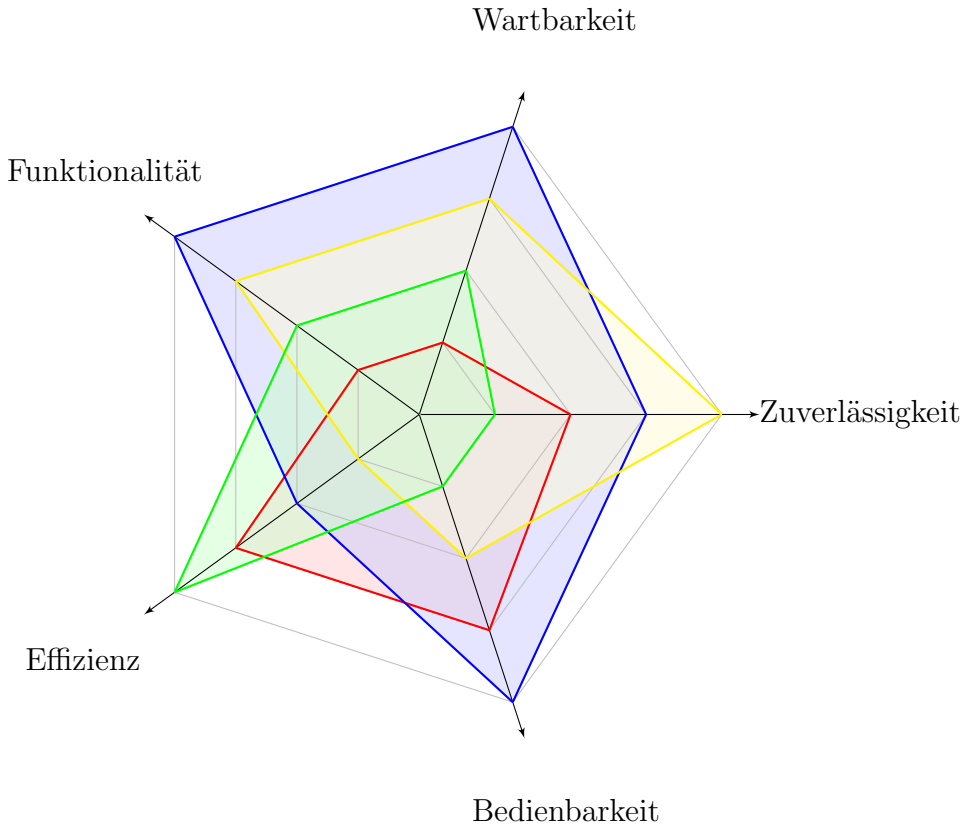

Fig. 2: Dieses Netzdiagramm der p2p Technologien in den Evalutaionskriterien zeigt, wie stark die einzelnen Kriterien von den Technologien abgedeckt werden. Je größer die gesamte Fläche, die eine Technologie dabei bildet, ist, desto besser ist sie als p2p Technologie zur Vebrindungskonfiguration geeignet. **Blau** entspricht hierbei Bluetooth, **Rot** stellt Wi-Fi Direct dar, **Grün** wird von USB genutzt und der **gelbe** Graph repräsentiert NFC.

6 Fazit

Die Überprüfung von Softwarequalität auf einer Kombination aus Technologie und Software stellt die Herausforderung, bewertbare Ergebnisse zu erhalten. Da die Qualitätsmerkmale der Zuverlässigkeit, Wartbarkeit, Funktionalität und Bedienbarkeit lediglich auf Grund von Erfahrungswerten bewertet wurden und ihre Spezifikation wesentlich zu komplex sind, um sie im Rahmen dieser Arbeit vollständig auszuwerten, kann nur eine informale Aussage über die eingesetzten Technologien getroffen werden. Jede der Technologien setzt unterschiedliche Schwerpunkte bei den überprüften Qualitätskriterien, wodurch jede der Technologien für manche Anwendungsfälle in Frage kommt. Der Anwendungsfall einer Verbindungskonfiguration als Client-/Server-Lösung zeichnet sich durch ein geringes Datenvolumen der Anfragen und Antworten in Verbindung mit teilweise hohen Antwortzeiten auf Anwendungsebene aus. Auf Grund dieser Gegebenheiten wird der Evaluationspunkt der Effizienz übergangen und lediglich anhand der verbleibenden Evaluationspunkte ein Ergebnis der Evaluation genutzt.

Obwohl sich die drei überprüften Technologien in ihrer Funktionsweise grundlegend unterscheiden, können sie alle für eine p2p Verbindung zwischen zwei Endnutzergeräten verwendet werden. Die eigentliche Art und Weise der angebotenen Funktionalität rückt bei der Umsetzung deutlich in den Hintergrund, da jede genutzte Bibliothek eine API bereitstellt, über die eine vereinfachte Steuerung der Verbindung zu anderen Geräten ermöglicht wird.

Für jede der genutzten Technologien kommt ein HTTP-Wrapper zum Einsatz, welcher es ermöglicht, die HTTP-Befehle über eine andere Technologie als Wi-Fi zum Server zu übertragen. Dieser Wrapper ist eine simple Lösung, die jegliche Technologie als Sockets abbildet, um eine simple Auswechselbarkeit der Technologien zu ermöglichen. Dies hilft ebenfalls das gesetzte Ziel einer Alternative zu Wi-Fi Direct zu erfüllen, da sich die unterschiedlichen Technologien mit geringem Aufwand auch parallel in der Anwendung nutzen lassen und so dem Nutzer eine Wahl zwischen mehreren Verbindungsarten gelassen werden kann. Ebenso müssen keine Änderungen an den Anbindungen der Technologien vorgenommen werden, wenn die Schnittstelle zur Verbindungskonfiguration erweitert werden soll. Da alle Anbindungen von Technologien die selbe Grundlage nutzen, um sie im Wrapper nutzen zu können, entsteht kein zusätzlicher Wartungsaufwand der existierenden Anbindungen, wenn eine neue Technologie ebenfalls genutzt werden soll.

Wichtig ist auch, dass die definierte REST-Schnittstelle in Verbindung mit dem HTTP-Wrapper unverändert weitergenutzt werden kann. Ein Weiterbestehen der bereits vorhandenen Architektur ist nötig, um eine wartbare Implementierung der verschiedenen Technologien vornehmen zu können, ohne tiefgreifende Änderungen an der Client/Server Architektur vornehmen zu müssen.

Während der Umsetzung aller p2p Technologien wurde deutlich, dass nicht nur Wi-Fi Direct Unzulänglichkeiten in seiner Implementierung besitzt. Alle getesteten Technologien weisen in unterschiedlichen Punkten Probleme auf, die nicht zwingend die Nutzbarkeit für eine Verbindungskonfiguration beeinträchtigen. Die aufgetretenen Probleme verdeutlichen jedoch noch einmal die unterschiedlichen

Ansätze der Technologien.

Bluetooth mangelt es auf Grund des Wechsels der Art von Bibliotheks-API an Dokumentation und allgemeinen Beispielen über die Nutzung der Bibliothek. Gleichzeitig traten Probleme bei der Nutzung von *SDP* auf, wodurch eine vereinfachte Lösung über einen expliziten RFCOMM Port genutzt werden muss. Dies hat den Vorteil, dass die Anbindung von Bluetooth wesentlich simpler ist, da keine DBus-API angesprochen werden muss.

Für eine Nutzung von NFC zeigt sich eine Unzulänglichkeit im Bereich der Benutzbarkeit, da der Nutzer gezwungen ist, das Gerät mehrmals in die direkte Nähe des zu konfigurierenden Gerätes bringen, was je nach Einsatzort keine akzeptable Lösung darstellt. Ebenso basiert NFC auf dem Austausch von kurzen Nachrichten über ein Trägersignal und unterstützt so lediglich eine indirekte Kommunikation beider Teilnehmer durch das Reagieren auf Befehle, was die Funktionalität der Lösung ein wenig dämpft, da so eine bidirektionale Kommunikation zusätzliche Komplexität birgt.

Eine Verwendung von USB scheitert zur Zeit daran, dass eine AOA Verbindung nicht korrekt aufgebaut werden kann. Dem kann zu Grunde liegen, dass AOA seit dem Jahr 2012 von Google nicht mehr aktiv weiter entwickelt wurde und inzwischen Fehler in der Anbindung dieses Protokolls existieren. Andererseits ist es auch möglich, dass zusätzlicher undokumentierter Konfigurationsaufwand unter Android vorgenommen werden muss. Da keine erfolgreiche Datenübertragung über USB vollzogen werden konnte, kann diese Technologie auch nicht im Rahmen einer Verbindungskonfiguration genutzt werden. USB besitzt durch die Bindung an ein Kabel gegenüber den drahtlosen Technologien einen erheblichen Nachteil in der Benutzbarkeit. Dies ist auch der Hauptvorteil im Hinblick auf Effizienz. Dadurch, dass Störsignale lediglich stark reduziert auftreten, können wesentlich höhere Datenraten erzielt werden.

NFC ist gegenüber USB und Wi-Fi Direct ebenfalls eine bessere Alternative, um p2p Verbindungen zu ermöglichen. Zwar liegen die Bedienbarkeit und Funktionalität nicht auf dem Niveau von Bluetooth, da NFC eine für den Nutzer und die Implementierung umständlichere Technologie nutzt, aufgewogen werden diese wieder durch die Wartbarkeit und Zuverlässigkeit des Mediums. Besonders der Punkt der Zuverlässigkeit muss bei dieser Technologie hervorgehoben werden, da hier Reproduzierbarkeit von Ergebnissen durch das Senden von Daten als festgeschriebene Bündel priorisiert wird.

In der Evaluation hat sich so Bluetooth als beste Technologie für eine p2p Verbindung durchgesetzt, weil sie die für eine Verbindungskonfiguration relevanten Kriterien den anderen Technologien gegenüber besser erfüllt. Besonders die Bedienbarkeit durch den Nutzer zusammen mit einer sehr guten Wartbarkeit geben Bluetooth einen großen Vorteil gegenüber den verbleibenden Übertragungsmedien. Bluetooth ist eine stabile und leicht zu nutzende Technologie, die die einfache Benutzbarkeit in den Vordergrund stellt. Die Bedienbarkeit stellt auch das wichtigste Kriterium der geprüften Softwarequalität dar, da die anderen Merkmal nicht zwingend vom Nutzer wahrgenommen werden und eine Software ohne Nutzer auch keinen Nutzen besitzt.

6.1 Ausblick

Im Nachgang kann nun auf Grundlage dieses Prototypen eine Umsetzung der p2p Funktionalität über die einzelnen Technologien stattfinden. Besonders kann hier noch einmal Bluetooth betrachtet werden, da die Bluetooth Lösung über einen festgelegten RFCOMM Port noch mittels des Bluetooth Service Discovery Protocol ersetzt werden sollte. Ebenfalls muss noch Discoverability standardmäßig in der genutzten BlueZ Instanz aktiviert werden.

Da der Fokus dieser Arbeit auf dem Aufbau einer p2p Verbindung lag, kann weiterführend der Fokus auf die Service Discovery geschoben werden. Hier kann besonders Bluetooth Low Energy (LE) betrachtet werden. Es besitzt eine ähnliche Funktionsweise zum bereits implementierten DNS Service Discovery über Wi-Fi Direct. Es erlaubt aber auf eine allgemeinere Art und Weise das verbindungslose Senden von Datenpaketen. Aufgrund der geringen Leistungsaufnahme eignet sich dies auch eher für batteriebetriebene Geräte als Wi-Fi Direct.

Android hat mit der Version Q neue Funktionalität für Wi-Fi Direct und Bluetooth LE bereitgestellt [Android Q Features]. Es kann hierbei überprüft werden, ob dies lediglich die Nutzung vereinfachen soll oder auch die Stabilität der Wi-Fi Direct Implementierung verbessert wurde.

USB sollte aufgrund einer mangelnden Unterstützung seitens Android in Verbindung mit einer umständlichen Nutzung für ähnlich Anwendungsfälle nicht mehr betrachtet werden. Stattdessen kann sich auf fortschrittlichere p2p Verbindungen konzentriert werden, die kein Kabel zur Datenübertragung mehr benötigen.

Nachdem die Verbindungskonfiguration zufriedenstellend abgehandelt ist, lassen sich für die vorgestellten Verbindungen weitere Anwendungsfälle finden, die von einer p2p Verbindung profitieren. Da REST als Schnittstellentechnologie genutzt wurde, lassen sich neue Anwendungsfälle wie ein p2p Remote Debugging für Pharo zukünftig leicht umsetzen.

Literaturverzeichnis

[Abts] Dietmar Abts: *Masterkurs Client/Server-Programmierung mit Java* Springer, Wiesbaden (2007).

[Axelson] Jan Axelson: *USB - Handbuch für Entwickler* MITP, Landsberg (2001).

[Eberhardt et al.] Udo Eberhardt, Hans Joachim Kelm [Hrsg.] et al.: *USB - Universal Serial Bus* Franzis, Poing (1999).

[Finger et al.] Patrick Finger, Prof. Dr. Klaus Zeppenfeld: *SOA und WebServices* Springer, Heidelberg (2009).

[Kretzschmar] Jan Phillip Kretzschmar: *Verbindungskonfiguration von PharoThings auf Raspberry Pi durch Android App* TH Köln, Anhang (2019).

[Langer et al.] Josef Langer und Michael Roland: *Anwendungen und Technik von Near Field Communication (NFC)* Springer, Heidelberg (2010).

[Liggesmeyer] Peter Liggesmeyer: *Software-Qualität - Testen, Analysieren und Verifizieren von Software* Spektrum Akademischer Verlag, Heidelberg (2002).

[Miller et al.] Brent A. Miller und Chatschik Bisdikian: *Bluetooth Revealed* Prentice Hall, NJ-USA (2001).

[Morrow] Robert Morrow: *Bluetooth Operation and Use* McGraw-Hill, NY-USA (2002).

[Sauter] Martin Sauter: *Grundkurs Mobile Kommunikationssysteme: LTE-Advanced, UMTS, HSPA, GSM, GPRS, Wireless LAN und Bluetooth* 6.Auflage Springer Vieweg, Wiesbaden (2015).

Internetquellen

[Android Bluetooth] https://developer.android.com/guide/topics/connectivity/bluetooth#ConnectAsAServer

[Android Open Accessory] https://source.android.com/devices/accessories/custom

[Android Q Features] https://developer.android.com/preview/features

[AOA Proxy Server] https://github.com/timotto/AOA-Proxy/blob/master/src/accessory.c (Zeilen 131&137)

[Bluetooth Spezifikation] https://www.bluetooth.com/specifications/bluetooth-core-specification/

[BlueZ 5 Porting] http://www.bluez.org/bluez-5-api-introduction-and-porting-guide/

[BlueZ SDP] https://people.csail.mit.edu/albert/bluez-intro/x148.html

[okhttp RealCall] https://github.com/square/okhttp/blob/okhttp_3.11.x/okhttp/src/main/java/okhttp3/RealCall.java (Zeilen 185-194)

[ITU Study Group] https://www.itu.int/en/ITU-T/about/groups/Pages/sg20.aspx

[libusb] http://libusb.sourceforge.net/api-1.0/index.html

[Pharo Socket Plugin] https://github.com/pharo-project/pharo-vm/tree/master/mc/VMMaker.oscog.package/SocketPlugin.class

[PharoThings] https://github.com/pharo-iot/PharoThings

[Microservices] https://microservices.io/

[MIT ADK] https://stuff.mit.edu/afs/sipb/project/android/docs/tools/adk/adk2.html

[NFC Google Bug] https://issuetracker.google.com/issues/37005118

[NFC Rates] https://nfc-forum.org/resources/what-are-the-data-transmission-rates/

[NFC Sockets Client] https://github.com/classycodeoss/nfc-sockets/blob/master/android/NFCSockets/app/src/main/java/com/classycode/nfcsockets/sockets/NFCSocket.java (Zeilen 467-475)

[NFC Sockets Server] https://github.com/classycodeoss/nfc-sockets/blob/master/pn532_socket_tunnel/pn532_socket_tunnel.c (Zeilen 113-167)

[NFC Sockets Blog] blog.classycode.com/sockets-over-nfc-on-android-c294b6c58bbf

[NFC-Tools libnfc] https://github.com/nfc-tools/libnfc

[NXP PN532 Datenblatt] https://www.nxp.com/docs/en/nxp/data-sheets/PN532_C1.pdf

[USB Spezifikation] https://usb.org/documents

[Wi-Fi Direct Rates] https://www.wi-fi.org/knowledge-center/faq/how-fast-is-wi-fi-direct

A Anhang: Verbindungskonfiguration von PharoThings auf Raspberry Pi durch Android App

Technology
Arts Sciences
TH Köln

Verbindungskonfiguration von PharoThings auf Raspberry Pi durch Android App

Praxisprojekt Sommersemester 2019

Jan Phillip Kretzschmar *(jan@2denker.de)*

Betreuer (Zweidenker GmbH):
Christian Denker *(christian@2denker.de)*

Betreuer (TH Köln):
Prof. Christian Kohls

30. Juni 2019

Gliederung

A.1 Einleitung

Das Unternehmen Zweidenker GmbH ist ein Softwareunternehmen mit Expertise in den Bereichen der App-, Web- und Backend Entwicklung.[Zweidenker] Für die unterschiedlichen Bereiche kommen, je nach Bedarf, unterschiedlichste Technologien zum Einsatz, vor Allem baut die Backend Entwicklung jedoch auf der virtuellen Laufzeitumgebung und Programmiersprache Pharo auf. Da die Zweidenker GmbH im engen Kontakt zum französischen Computerinstitut INRIA steht und sich stark in die Pharo-Community einbringt, ist es auch leicht bei Problemen der Umgebung Hilfe zu finden. Pharo ist eine auf Smalltalk basierende objektorientierte und dynamisch getypte Programmiersprache, welche gleichzeitig ihre eigene live Entwicklungsumgebung mit mächtigen Debugging-Tools ist.[Pharo] Aktuell wurde die Laufzeitumgebung PharoThings [Pharo Things] vorgestellt, welche es ermöglicht, im Rahmen des Begriffes Internet Of Things (IoT) eine reduzierte Plattform auf Kleinstcomputern wie dem Raspberry Pi zu ermöglichen. Auf Grund dieser reduzierten Laufzeitumgebung kam im Unternehmen die Idee auf, im Bereich IoT Expertise zu sammeln und in Zukunft Komplettlösungen für diesen Bereich anbieten zu können. Als ersten Schritt in diese Richtung ist diese Projektarbeit zu sehen, in der dem Nutzer die Möglichkeit gegeben werden soll, Drahtlosnetzwerkschnittstellen von IoT Geräten per Smartphone-App konfigurieren zu können. Aktuell ist es mit einem Raspberry Pi nötig, diese Konfiguration über das Dateisystem an einem Computer mit SD-Kartenlesegerät vorzunehmen oder einen Monitor und Tastatur an den Raspberry Pi anzuschließen. Für den allgemeinen Benutzer ist es jedoch wünschenswert eine Lösung anbieten zu können, die ohne Hintergrundwissen der eingesetzten Technologien genutzt werden kann. Um die Pharo Things Laufzeitumgebungen als IoT-Geräte simpel nutzen zu können, soll eine Android App den Konfigurationsvorgang übernehmen:

1. Erkennen und Auflisten von IoT-Geräten in der Nähe. Die zu verwendende Kommunikationstechnologie ist dabei zu evaluieren.

2. Aufbau einer Verbindung zu ausgewähltem IoT-Gerät

3. IoT-Gerät erhält WLAN-Konfiguration und weitere Einstellungen über Eingabemaske

4. Eventuelle Verbindungsprobleme der WLAN-Verbindung werden über die bestehende Verbindung zurückgemeldet

5. Pharo Things-Installationen zeigen Verbindungsstatus in der Auflistung an

Um diesen Vorgang umsetzen zu können, werden drei Komponenten implementiert:

1. Ein Protokoll muss definiert werden, welches die Kommunikation zu PharoThings Instanzen zur Konfiguration festlegt. Weiterhin muss festgelegt werden, in welcher Art und Weise ein Beacon-Signal gesendet wird. Um die verfügbare WLAN Bandbreite so wenig wie möglich zu belasten, empfiehlt es sich diese Nachrichten kurz zu halten. Es ist zu überprüfen, ob sich Installationen auch

gegenseitig erkennen können, sodass ein gebündelter Beacon gesendet werden kann.

2. Eine Android App, welche den Nutzer durch den beschriebenen Vorgang leitet, muss implementiert werden. Der Fokus hierbei liegt darin, diesen Vorgang mit möglichst wenig Nutzerinteraktion durchzuführen.

3. Eine Anwendung in PharoThings muss erstellt werden, welche das Protokoll implementiert und basierend darauf sich in einem WLAN einwählen kann und diese Konfiguration speichert.

Hierbei sollte zunächst evaluiert werden, welche Kommunikationstechnologien für dieses Projekt in Frage kommen, um klar zu stellen, welche Paradigmen den einzelnen Technologien zu Grunde liegen und welchen Limitierungen daraus für das Projekt entstehen. [1]

A.2 Mögliche Kommunikationstechnologien

Ein Ad Hoc Netzwerk bietet im Allgemeinen die Möglichkeit *peer to peer* (p2p) Verbindungen zwischen Geräten dezentralisiert aufzubauen. Geräte können hierbei selbstständig eine Netzwerkverbindung untereinander aushandeln. Da solche Verbindungen nur dann sinnvoll sind, wenn es Daten gibt, die nur zwischen den beiden verbundenen Geräten ausgetauscht werden müssen, ergibt ein solches Netzwerk meist nur im Bezug auf eine tatsächliche Anwendung Sinn. Die erweiterte Definition des Ad Hoc Netzwerks bezieht somit alle Netzwerkschichten des OSI-Modells mit ein.[Sikora, S.23] Obwohl das OSI-Modell vor Allem auf Ethernet und WLAN ausgelegt ist, lässt sich die Definition des Ad Hoc Netzwerks dennoch für weitere Kommunikationstechnologie übernehmen, da diese ebenfalls p2p Verbindungen aufbauen können. Optimalerweise sollte es möglich sein die bestehenden WLAN-Verbindungen beider Geräte während einer p2p Verbindung beibehalten zu können. Für Kommunikationsmedien in diesem Projekt fallen einige Beschränkungen an:

1. *Reichweite:* Netzwerke werden oft nach ihrer Reichweite klassifiziert. Dabei gibt es die geläufigen Bezeichnungen Local Area Network (LAN), Metropolitan Area Network, Wide Area Network und Global Area Network, die in ihrer Klassifizierung von Gebäuden zu einer Globalen Reichweite übergehen. Üblicherweise verbinden höhere klassifizierte Netzwerke niedriger klassifizierte Netzwerke miteinander. Im Bereich der Drahtlosnetzwerke gibt es jede dieser Klassen ebenfalls als Drahtlos-Variante: WLAN, WMAN, WWAN und WGAN, es kommen jedoch noch zwei in ihrer Reichweite kleinere Netzwerke hinzu, das Wireless Body Area Network und das Wireless Personal Area Network. Übliche Einsatzgebiet des WBAN sind im medizinischen Bereich zu finden, aber auch Near Feald Communication fällt in diese Kategorie. Unter die Klassifizierung des WPAN fällt unter anderem Bluetooth, für dieses Projekt sind somit Funknetzwerke der untersten drei Kategorien WBAN, WPAN und WLAN oder deren kabelgebundenen Derivate interessant.[Lüders, S.17]

[1] Der Code zu diesem Projekt findet sich in [iot-connectivity].

2. *Unterstützung in Android Smartphones:* Damit eine große Anzahl an potentiellen Nutzern angesprochen werden kann, muss die Verbindungsschnittstelle von Smartphones unterstützt werden. Für dieses Projekt wird dabei nur Android betrachtet. Aktuelle Smartphones bieten im Allgemeinen zur Zeit die vier Schnittstellen **USB, NFC, Bluetooth und Wi-Fi**, über die sich Verbindungen zu Geräten in der näheren Umgebung aufbauen lassen.

3. *Hardware an IoT Geräten:* Als IoT Gerät dient in diesem Projekt ein Raspberry Pi. In den Varianten *Model 3 B, Model 3 B+ and Model Zero W* bietet Dieser USB, Bluetooth und Wi-Fi als mögliche Schnittstellen. Durch die Verwendung der GPIO-Pins ist es außerdem möglich, ein NFC-Modul anzubinden, jedoch würde Dies die später nutzbaren Pins unerwünscht einschränken. Weiterhin bietet der Raspberry Pi ein vollständiges Betriebssystem mit Benutzeroberfläche, jedoch soll eine Internetverbindung ohne Peripherie am Raspberry Pi konfiguriert werden können.

A.2.1 Kommunikation über WLAN

Der IEEE802.11 Standard siedelt sich im OSI-Modell lediglich in der Physical Layer und Data Link Layer an. Ihr eigentlicher Sinn ist es, IP-Pakete der Network Layer im gleichen Maße wie ein LAN übertragen zu können. Die Definition des Wireless LAN unterscheidet sich jedoch vom LAN Standard dahingehend, dass eine vollständig eigene Physical Layer geschaffen wurde, da das Übertragungsmedium andere Restriktionen besitzt. Die Data Link Layer setzt sich für WLAN größtenteils aus drei Teilen zusammen. Die Logic Link Control nach 802.2 und das Bridging nach 802.1 sind mit LAN identisch, um der Network Layer eine einheitliche Schnittstelle unabhängig des Übertragungsmediums zu bieten. In der Data Link Layer unterscheidet sich lediglich der Media Access Control (MAC).[Sauter, S.311] Dieser regelt im Fall von WLAN den Zugriff auf das Übertragungsmedium durch unterschiedliche Wartezeiten zwischen Frames und die Reservierung des Mediums zum Senden von Frames. Da das MAC-Protokoll zudem die Adressierung von Geräten ermöglicht, bietet es ebenfalls bereits die Möglichkeit, Broadcasts zu senden. Um die hohe Fehleranfälligkeit eines Drahtlosnetzwerks für höhere Schichten zu reduzieren, wird jedes Frame vom Empfänger bestätigt.[Sauter, S.325-327]

Ein Netzwerk nach 802.11 kann hierbei entweder im Infrastruktur Modus, in dem alle Geräte ausschließlich mit einem Access Point kommunizieren, oder im Ad Hoc Modus, welcher die direkte Kommunikation zwischen Geräten erlaubt, betrieben werden.[Sikora, S.82] Ein Verbindungsaufbau ist für eine p2p Verbindung entweder durch eine konkrete Implementierung des Ad Hoc Modus oder im MAC-Protokoll zu ersuchen.

Unter dem Markennamen Wi-Fi™ werden 802.11-kompatible Geräte zertifiziert. [Sikora, S.80] Für den Ad-hoc Modus nach 802.11 wurde dabei Wi-Fi Peer-to-Peer (Wi-Fi Direct)®[WiFi Direct] als ein universeller Standard definiert. Durch Wi-Fi Direct ist es jedoch ebenfalls möglich, einen Verbindungsaufbau in der **Application Layer** des OSI-Modells anzusiedeln. Dazu bietet diese Spezifikation neben dem normalen Peer-To-Peer Modus die Möglichkeit, Services anzubieten und zu finden,

bevor eine Verbindung etabliert werden muss. Grundlage für diese Services bilden dabei DNS Service-Discovery (DNS SD) und Universal Plug and Play (UPnP). DNS SD ist die Weiterentwicklung von Apple Bonjour und wird größtenteils genutzt um eine Zero-Configuration Service Discovery von beispielsweise Netzwerkdruckern zu ermöglichen. Da UPnP im Gegensatz zu DNS SD auch die Kontrolle über die Services übernimmt, besitzt es einige Verwundbarkeiten. Man kann es jedoch ähnlich zu DNS SD lediglich dazu benutzen, Services zu ernennen.[Esnaashari et al.]

Nutzung unter Android Im Gegensatz zu Apple AirPlay kann für eine App nicht auf Gerätetreiber-Ebene entwickelt werden. Um MAC in Linux verwenden zu können, ist es nötig, Sockets mit dem Attribut **SOCK_RAW** zu öffnen, um eigene MAC-Pakete senden zu können. Solche Sockets können jedoch nur mit der Berechtigung **CAP_NET_RAW** erstellt werden.[Linux Packet] Unter Android fällt diese Berechtigung mangels Granularität root zu, wodurch diese Lösung unpraktikabel wird, wenn eine möglichst große Nutzergruppe angesprochen werden soll.[Android Permissions] Android bietet jedoch ab API 14 die Möglichkeit, sich über Wi-Fi Direct als möglicher peer anderen Geräten zu präsentieren und p2p Verbindungen aufzubauen, sowie Services bereitzustellen und zu erkennen. Android stellt diese p2p Funktionalität als *WifiP2PManager* bereit. Ein kurzer Test mit zwei Android Geräten hat dabei ergeben, dass dieser bestehende Wi-Fi-Verbindungen während der Service Discovery beibehält.[iot-p2p-test] Ein ähnlicher Ansatz, in dem Wi-Fi Direct genutzt wird, um ein Ad Hoc Netzwerk aufzubauen findet sich in [Aneja et al.].

A.2.2 Kommunikation über Bluetooth

Die grundlegenden Komponenten von Bluetooth sind im Standard IEEE802.15.1 als *Bluetooth Core* definiert. Anwendungen können über diesen Kern oder speziellere Protokolle Bluetooth Verbindungen zu anderen Geräten aufbauen.[Lüders, S.228] Um Daten über Bluetooth senden zu können, muss eine Verbindung zwischen den beiden Geräten aufgebaut werden. Für diese Verbindung ist es nötig, dass sich die entsprechenden Geräte zunächst koppeln. Zum Koppeln verfügbare Geräte werden hierbei durch eine sogenannte Inquiry, welche erreichbare Geräte auffordert, sich zu identifizieren, aufgelistet. Da nun die Adresse des zu koppelnden Gerätes bekannt ist, kann zu diesem eine Koppelung angefragt werden, was als Paging bezeichnet wird. Die Kopplung dient dabei dem Austausch der Frequency Hop Sequenzen, welche festlegt, wann auf welcher Frequenz Pakete gesendet werden, sowie dem Pairing von Geräten, welches sicherstellt, dass das richtige Gerät gekoppelt wird und Schlüssel zur Authentifizierung und Verschlüsselung überträgt.[Sauter, S.402f.]

Sobald eine Verbindung zwischen den Geräten aufgebaut wurde, können Daten bidirektional übertragen werden, indem ein Bluetoothkanal, anders als WLAN, in Zeitslots unterteilt wird. Die alleinige Kontrolle, welches Gerät wann Daten senden darf, hat dabei das Mastergerät des Netzes. Das Gerät, welches den Verbindungsaufbau angefragt hat, agiert dabei als Master und bis zu sieben weitere Geräte können als Slave in einem Piconetz verbunden sein.[Sauter, S.379f.]

Für bestimmte Anwendungsfälle gibt es spezielle Protokolle, welche generische Lösungen dieser Fälle bieten. So ermöglicht das Service Discovery Protocol (SDP) den Informationsaustausch über verfügbare Dienste der Kommunikationspartner, Radio Frequency Communications (RFCOMM) kann serielle Schnittstellen abbilden und das Object Exchange Protocol (OBEX) kann Datenobjekte über RFCOMM übertragen.[Lüders, S.229] SDP erlaubt nicht nur das Auflisten von verfügbaren Diensten des Verbindungspartners, sondern auch das Suchen von einem bestimmten Dienst auf erreichbaren Geräten, es kann dabei jedoch keinen Zugriff zu den Diensten bereitstellen oder die Verfügbarkeit der Dienste auf den erreichbaren Geräten angeben.[Morrow, S.395f]

Bluetooth lehnt sich mit dem *Bluetooth Core* nur lose an das OSI-Referenzmodell an, da nicht jedes Element des Bluetoothstacks sich sauber einer OSI-Schicht zuordnen lässt. Die oben vorgestellten Protokolle wie SDP oder RFCOMM siedeln sich jedoch in der **Presentation Layer** an, da diese lediglich dazu gedacht sind, Daten leichter der Anwendung zur Verfügung stellen zu können.[Sauter, S.382]

Bluetooth Low Energy Als Teil des Bluetooth Standards ist Bluetooth Low Energy (LE) darauf ausgelegt, Daten mit einem möglichst geringen Energieverbrauch über die Bluetooth-Hardware zu übertragen. Dies wird ermöglicht, indem verbindungslose Broadcast, die Nutzerdaten enthalten können definiert werden, sowie aktive Verbindungen zwischen zwei Geräten auf eine kurze Übertragung von 6 Paketen pro Verbindungsevent limitiert werden. Verbindungsevents sind dabei durch ein wählbares Intervall zwischen 7.5ms und 4s von einander getrennt. Weiterhin beschränkt sich Bluetooth LE im Hinblick auf die Reichweite darauf, möglichst wenig Energie zu verbrauchen, da eine hohe Sendeleistung in einem hohen Energieverbrauch resultiert. Da sich die Sendeleistung jedoch pro Gerät konfigurieren lässt, ist die theoretische Reichweite ähnlich zu Bluetooth gegeben.[Townsend et al., S.7f.]

Nutzung unter Android Android stellt sowohl Bluetooth als auch Bluetooth LE als *BluetoothAdapter* bereit. Ähnlich zu Wi-Fi Direct Android bietet ab API 5 beziehungsweise API 18 die Möglichkeit, über Bluetooth als möglicher peer anderen Geräten zu präsentieren und p2p Verbindungen aufzubauen. Über Bluetooth LE können hingegen Services als Broadcasts bereitgestellt werden. Bluetooth in Verbindung mit Bluetooth LE lässt sich somit im Rahmen dieser Arbeit ohne Probleme auf Android benutzen.

A.2.3 Kommunikation über NFC

Die Near Field Communication (NFC)-Technologie basiert auf RFID-Systemen und erlaubt die Übertragung von Daten auf eine Distanz bis zu 10 Zentimeter. Grundlegend ist NFC in den Standards NFCIP-1 (ECMA-340) und NFCIP-2 (ECMA-352) definiert. Wie bei klassischen RFID-Systemen ist ein magnetisches Feld als Trägersignal der Daten definiert. Dieses magnetische Feld dient dazu, passiven Komponenten, die keine eigene Stromversorgung besitzen, auf Anfragen der steuernden

Komponente zu antworten. Anders als RFID hebt NFC jedoch die strikte Trennung zwischen steuernder (Initiator) und gesteuerter (Target) Komponente auf, sodass jedes teilnehmende NFC-Gerät zumindest theoretisch beide Rollen übernehmen kann.[Langer et al., S.89] Rückwärtskompatibilität erreicht NFC, indem sowohl ein Reader-Writer-Modus, der die Kommunikation mit passiven RFID-Transpondern ermöglicht, als auch ein Card-Emulation-Modus, der die Kommunikation mit RFID-Lesegeräten erlaubt, definiert ist.[Langer et al., S.99f.] Als letzter Modus ist der hier relevante Peer-to-Peer-Modus, dazu in der lage, Daten zwischen zwei NFC-Geräten zu übertragen. Ähnlich zu Bluetooth lehnt sich auch NFC nur lose an das OSI-Modell an. Es gibt für NFC drei aufeinander aufbauende Protokollschichten. Die Bitübertragung ist in NFCIP-1 definiert und teilt sich in einen aktiven und passiven Modus. Der aktive Modus hebt sich dabei vom passiven Modus, welcher in seiner Funktionsweise äquivalent zum RFID-System ist, dadurch ab, dass das Trägersignal von beiden Kommunikationspartnern abwechselnd generiert wird. Die darauffolgende Schicht ist ein MAC Protokoll, welches den Zugriff auf das Übertragungsmedium sichert, indem geprüft wird, ob bereits ein Trägersignal existiert, bevor ein Gerät in den Initiator-Modus wechselt, andernfalls verbleibt es im Target-Modus. Die letzte Schicht bildet das Logic Link Control Protocol (LLCP), welches es beiden Kommunikationspartnern erlaubt, eine Datenübertragung zu initiieren. [Langer et al., S91.f, S.97]

Nutzung unter Android NFC wurde unter Android, verteilt über die API Versionen 9, 10 & 14, als *NfcManager* und *NfcAdapter* zur Verfügung gestellt. Im Gegensatz zu Wifi, wo lediglich Verbindungszustände als Broadcasts gemeldet werden, werden bei Nfc die gelesenen Daten als Broadcasts an alle zuhörenden Apps gesendet, wodurch sich eine gesicherte Verbindung nicht gewährleisten lässt.

A.2.4 Kommunikation über USB

Im Gegensatz zu den vorhergehenden Übertragungsmedien ist der Universal Serial Bus (USB) eine kabelgebundene Schnittstelle. Aus der maximalen Kabellänge von 5 Metern ergibt sich somit auch die maximale Reichweite der Technologie. USB ist primär darauf ausgelegt, (Peripherie-)Geräte an einen Host anzuschließen. Der Host stellt hierbei eine minimale Versorgungsspannung auf dem Bus bereit, sodass Geräte sich beim Host registrieren können und eine höhere Stromversorgung anfordern können. Weiterhin übernimmt der Host zwangsläufig auch die Kontrolle über die Verbindung, da Geräte zu jedem Zeitpunkt physisch vom Bus entfernt werden können, sodass Hot-Plug-and-Play möglich wird.[Eberhardt et al., S.21-24] Da USB eine möglichst breite Anzahl an Geräten zu unterstützen versucht, ist es nötig in den meisten Fällen einen Treiber auf dem Host bereitzustellen, sodass Anwendungen mit dem USB-Gerät kommunizieren können.[Eberhardt et al., S.197]

Nutzung unter Android Durch das Android Open Accessory Framework, welches ab API 10 angeboten wird, ist es möglich, mit einem Android-Gerät über USB zu kommunizieren und dabei als Host zu agieren. Es ist hierbei jedoch nötig

einen Treiber auf Host-Seite zu schreiben, um dieses Framework nutzen zu können. [Android Accessory]

A.2.5 Evaluation der Übertragungsmedien

Alle vorgestellten Schnittstellen, außer USB, lassen sich unter Android, wie in [iot-p2p-test] zu sehen, mit ähnlichem Aufwand anbinden. Da für USB ein Treiber auf Hostseite nötig ist, der AOA implementiert, ist der Aufwand wesentlich höher im Vergleich zu den restlichen Schnittstellen. Unter Pharo wurde bisher keine dieser Kommunikationskanäle angebunden, jedoch lassen sich bereits Sockets in Pharo nutzen und damit MAC-Pakete versenden. Da WLAN die größte Reichweite im Vergleich zu den restlichen gezeigten Technologien bietet, stellt sich hierbei ebenfalls Wi-Fi Direct als sinnvollste Schnittstelle heraus. Da nicht jedes Android-Gerät Wi-Fi Direct unterstützt, ist es jedoch sinnvoll, die Implementierung der Konfigurationsschnittstelle so zu kapseln, dass sie auch über andere Kanäle angesprochen werden kann.

Der Austausch über Bluetooth über verfügbare Dienste der Geräte ist erst nach einer vollständigen Kopplung und Verbindung möglich. Dadurch ist ein Filtern der verfügbaren Bluetooth Geräte zur Auswahl durch den Nutzer nur über den Namen oder die Adresse des Gerätes möglich. Im Gegensatz dazu ermöglicht Bluetooth Low Energy eben genau die gleiche Funktionalität wie Wi-Fi Direct, indem Daten ohne eine aktive p2p-Verbindung übertragen werden können. Es sollte somit Bluetooth nur in Kombination mit Bluetooth Low Energy genutzt werden.

Um dem Nutzer möglichst einfach erreichbare Geräte zeigen zu können, ist es wünschenswert, dem Nutzer wenig bis keinen Aufwand bei der Anbindung des Gerätes über die Übertragungsmedien zu geben. Wi-Fi Direct bietet hierbei mit DNS SD eine Lösung, um Dienste ohne Konfiguration auf Seite von Client-Geräten vorzustellen, die keinerlei Aufwand für den Nutzer bedeutet. Bluetooth Low Energy bietet ebenso die Möglichkeit, Beacon-Signale mit Nutzdaten zu versenden, wodurch die IoT-Geräte auf Client-Geräten ohne Nutzereingabe identifiziert werden können. Für NFC und USB sind ein physischer Zugang zu dem IoT-Gerät notwendig, wodurch sie in manchen Szenarien, wie die Montage an einer Decke, einen relativ hohen Aufwand für den Nutzer bedeuten. NFC ermöglicht eine Datenübertragung nur für einen kurzen Zeitraum, wodurch es für dieses Projekt weniger praktisch als eine USB-Verbindung ist. Da das Ergebnis der WLAN-Verbindungsversuche auf dem Smartphone zu sehen sein soll, würde somit ein mehrfacher Verbindungsaufbau über NFC nötig werden.

A.3 Architektur

Unabhängig des verwendeten p2p Übertragungsmediums soll eine Schnittstelle in Form eines auf Requests antwortenden Servers zur Verfügung stehen. Weit verbreitet ist in diesem Rahmen HTTP als Grundlage für REST und SOAP um einen Webservice anbieten zu können, auch über Proxies und Firewalls hinweg. Für einen p2p Schnittstelle sind solche Vorteile natürlich unerheblich, da die Schnittstelle nicht über eine entfernte Verbindung nutzbar ist, jedoch ist es dennoch von Vorteil, sich darauf festzulegen, eine solche Protokolldefinition zu nutzen, um zur Schnittstellenanbindung und -Implementierung weit verbreitete und damit stabile Bibliotheken nutzen zu können. Da sich eine REST Schnittstelle durch die Pakete zinc [Pharo Zinc] und OpenAPI [Pharo OpenAPI] leicht in pharo definieren lässt, bietet es sich an, diese fertigen Serverkomponenten auch zu nutzen. Um REST als Schnittstelle jedoch nutzen zu können muss der genutzte Kommunikationskanal HTTP übertragen können. Da jede der vorgestellten Schnittstellen Bytes oder Strings übertragen kann, stellen die Übertragungsmedien kein Problem dar, jedoch muss der REST-Server im Falle von Bluetooth, NFC und USB die HTTP-String Repräsentationen zur Verfügung stellen können, ohne sie über eine TCP/IP-Netzwerkschnittstelle zu versenden. Ebenso muss der Client die HTTP-Nachrichten korrekt verarbeiten können. Da die meisten Bibliotheken an den TCP/IP-Stack von Android gekoppelt sind, um einen gekapselten HTTP-Client anbieten zu können, ist es nicht immer möglich eine String-Repräsentation des zu tätigenden Aufrufs zu erhalten oder an die Bibliothek zu übergeben. Die OkHttp Bibliothek [Android OkHttp] sollte diese Probleme jedoch umgehen können, da Netzwerk Aufrufe durch eine Kette von Interceptoren gereicht werden und der RealNetworkInterceptor, welcher sich als Letztes in dieser Kette befindet, überschrieben werden kann. Im folgenden wird jedoch nur noch, Wi-Fi Direct und damit eine TCP/IP-Implementierung betrachtet.

Um die Schnittstelle einfach warten und erweitern zu können, wird sie als JSON-Schema über OpenApi dokumentiert und dem Client zur Verfügung gestellt. JSON-Schema ist eine Spezifikation, die es erlaubt JSON menschenlesbar zu beschreiben und ebenso durch Maschienenlesbarkeit zu validieren.[JSON Schema] In Kombination mit OpenAPI ermöglicht es so, eine gesamte Schnittstelle zu beschreiben und als einzelnen Server-Aufruf anzubieten.[OpenAPI] Dadurch wird so die Dokumentation der Schnittstelle an die Implementierung gekoppelt. Um die Dokumentationsarbeit so gering wie möglich zu halten, wird OpenAPI so genutzt, dass die Dokumentation direkt aus der Implementierung generiert wird. Dadurch sind Abweichungen der Dokumentation von der Implementierung ausgeschlossen.

A.3.1 Funktionsanforderungen

Die zu definierende Schnittstelle muss folgende Funktionalitäten anbieten:

- Eine Auflistung der verfügbaren Netzwerkschnittstellen sollte ähnlich zu *ip link show* zur Verfügung stehen, um die ID der gewünschten Schnittstelle für die nächsten Aufrufe herauszufinden.

- Pro Netzwerkschnittstelle soll eine Liste der aktuell erreichbaren Netzwerke ausgegeben werden können.
- Eine Netzwerkschnittstelle muss konfigurierbar sein, um sich mit einem Netzwerk verbinden zu können. Dazu zählt, eine Liste der bestehenden Konfigurationen zu erhalten, eine Konfiguration auszuwählen, eine einzelne Konfiguration abzufragen und neue Konfigurationen anlegen zu können. Für ein WiFi Netzwerk muss eine Konfiguration aus der gesendeten ID des Netzwerks und einem optionalem Netzwerkschlüssel bestehen, da alle verbleibenden Informationen über das Netzwerk vom Broadcast des Access Points des zu verbindenden Netzwerks erhalten werden kann. Passwörter sollten jedoch aus Sicherheitsgründen nicht wieder über die Schnittstelle ausgegeben werden.
- Die Spezifikation der Schnittstelle soll als Aufruf der Schnittstelle zur Verfügung stehen, sodass sie leicht nutzbar ist.

A.3.2 P2P Anbindung

Die REST Schnittstelle muss jedoch über eine Verbindung zur Verfügung gestellt werden, über die unter Umständen nicht direkt die Sockets des Gerätes, auf dem der REST-Server sich befindet, angesprochen werden können. In diesem Fall ist generell ein Eingriff in den IP-Protokollstack nach dem OSI-Referenzmodell mindestens ab der Transportschicht nötig. Im Rahmen der P2P Anbindung muss das Gerät, auf welchem die Konfiguration der Netzwerkschnittstellen möglich sein soll, zusätzlich sich anderen Geräten in der Nähe zu erkennen geben durch ein Beacon-Signal (p2p-Serivce-Discovery), das ähnlich zu DNS Service Discovery ausgesendet wird um eine Zero-Konfiguration Nutzung zu ermöglichen.[Kaiser et al.] Manche p2p Technologien bieten eine sehr limitierte bis keine Möglichkeit Service-Discovery zu betreiben. Da Diese jedoch nicht an die restliche Implementierung gebunden ist, kann hierbei auch eine andere Technologie genutzt werden, als für die tatsächliche p2p Verbindung genutzt wird.

Fig.3 zeigt, welche Elemente existieren müssen, um die Serveranwendung als REST-Server Clientgeräten zur Verfügung zu stellen. Größtenteils stehen diese Elemente bereits durch das Betriebssystem und Bibliotheken bereit, jedoch müssen diese gegebenenfalls mit einem TCP/HTTP-Wrapper verbunden werden. Dieser muss dazu in der Lage sein, den REST-Server mit HTTP-Anfragen ansprechen zu können und auf der anderen Seite diese HTTP-Anfragen und Antworten in den p2p-Channel speisen. Dieser Wrapper ist dann sowohl auf Client- sowie auf Server-Seite nötig. Generell kann dieser Wrapper unabhängig vom Rest der Anwendung agieren, da er lediglich Verbindungen ähnlich zu einem Socket akzeptiert und auf den Port des REST-Servers zugreifen kann. Jedoch ist es durch Limitierungen unter Android eventuell leichter, den Wrapper soweit in Client und Server einzubetten, dass nicht TCP-Pakete übertragen werden, sondern lediglich HTTP-Strings, dann können diese Wrapper zwar noch parallel auf mehreren Schnittstellen agieren, sie sind aber nicht mehr vollständig unabhängig von der verwendeten REST-Client und -Server Implementierung.

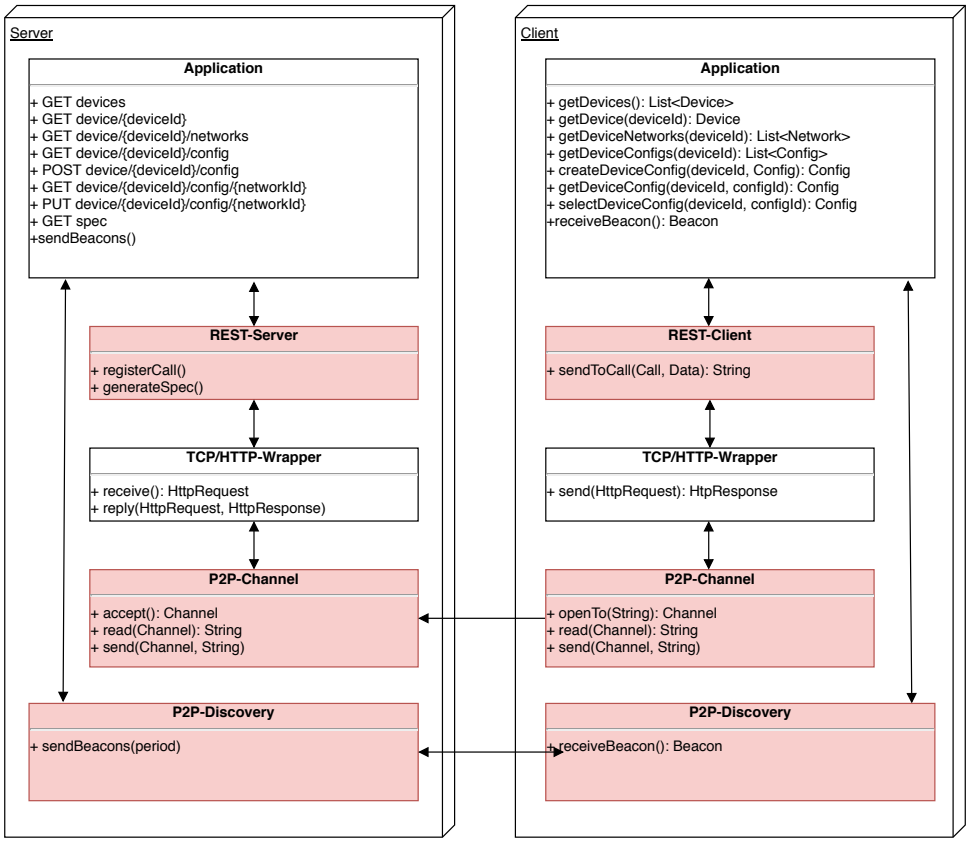

Fig. 3: Zu sehen sind die Schnittstellen, die nötig sind, um die Client-Server Anwendung über p2p nutzen zu können. Hierbei stehen alle rot hinterlegten Schnittstellen bereits zur Verfügung. Wenn die genutzte p2p Schnittstelle es nicht erlaubt, TCP/HTTP Pakete an die Ports einer Netzwerkschnittstelle des Servers zu senden, muss ein TCP/HTTP-Wrapper erstellt werden. Andernfalls besteht die Implementierung aus einem gewöhnlichem REST-Client/Server-Paar. Getrennt davon muss jedoch auch innerhalb der Anwendung die p2p-Service-Discovery angebunden werden.

A.4 Umsetzung

Um eine stabile Lösung schnell und simpel anbieten zu können, wird zur Implementierung der REST-Schnittstelle auf bestehende Server sowie Client Bibliotheken zurückgegriffen. Diese bieten somit die Möglichkeit die Implementierung dieser Schnittstelle deskriptiv vorzunehmen. In Android wird somit die Implementierung zur Laufzeit aus einer Schnittstellenbeschreibung generiert und auf der Serverseite müssen so nur die Implementierungen der Methoden beschrieben werden. Da die Wahl des Übertragungsmediums auf Wi-Fi Direct gefallen ist, können ebenso Teile des Betriebssystems und Standardbibliotheken zum Broadcast und Aufbau der p2p Verbindung genutzt werden. Da unter Linux das Paket wpa_supplicant [WiFi wpa_supplicant] auf allen gängigen Distributionen, so auch unter raspbian auf dem Raspberry Pi, dazu genutzt wird, Wi-Fi Schnittstellen und Netzwerke zu verwalten. Es läuft als daemon im Betriebssystem mit und kann über ein Kom-

mandozeileninterface bedient werden oder mit einer C-Schnittstelle in das eigene Programm eingebunden werden.

A.4.1 Pharo

Auf der Serverseite in Pharo besteht die Implementierung aus der Anbindung der wpa_supplicant Schnittstelle, einem C-Modul für diese Anbindung, den Methoden des REST-Servers und einem Modul, welches die p2p Verbindung und Service Discovery handhabt. Die Anbindung des wpa_supplicant beginnt damit, dass aus der hostap [WiFi HostAP Repository] Bibliothek ein C-Modul kompiliert werden muss, welches es erlaubt, das interne Control Interface (common/wpa_ctrl.c) zu nutzen. Darauf aufbauend muss in pharo eine Klasse geschaffen werden, die in der Lage ist, Handle (LWpaHandle) des Control Interfaces zu verwalten und deren Zustand persistent zu speichern. Persistenz ist nötig, da die pharo-VM zwischen Nachrichten gestoppt und zu einem anderen Zeitpunkt erst wieder fortgesetzt werden kann. Wenn verwendete Netzwerkschnittstellen vom Gerät entfernt werden, muss dies dann auch aus der Bibliothek als Fehler signalisiert werden. Weiterhin müssen Events über Polling an Listener signalisiert werden und außerhalb des wpa_supplicant kann die Liste der verfügbaren Schnittstellen abgefragt werden.

Basierend auf dieser Anbindung wird LWpaInterface als einheitliche Fassade genutzt, die alle genutzten Nachrichten, die an den wpa_supplicant gesendet werden können, als Methoden implementiert. In diesem Interface sind alle Methoden der p2p Service Discovery über p2p Verbindungsaufbau und Netzwerkscans bis hin zur Netzwerkkonfiguration enthalten. Alle Methoden stehen auch auf allen Netzwerkgeräten zur Verfügung, es ist aber zu erwarten, dass bei nicht unterstützten Methoden ein Fehler auftritt. Die p2p Komponente, welche Service Discovery und p2p Verbindungsaufbau ermöglicht, ist ein einzelnes Objekt, welches über start und stop Methoden verfügt, um die Softwarelösung als gebündelte Einheit zu betreiben. Dieses Objekt startet auch gleichzeitig den REST-Server auf einem definierten Port, da dieser Port in der p2p Service Discovery angegeben werden muss, damit Clients den Server auch nutzen können. Weiterhin sind für Wi-Fi Direct Event Handler nötig, die auf Events zum Verbindungsaufbau reagieren und Verbindungen akzeptieren.

Zur Implementierung des REST-Servers müssen die Methoden, die der REST-Server anbieten soll, als Kinder einer abstrakten Klasse (ICCall) definiert werden, sodass die Liste der Kinder dem REST-Server zugeliefert werden kann. Jede Klasse beschreibt dabei einen Pfad des Servers auf dem ein oder mehrere Http-Verben implementiert werden. Da manche Methoden jedoch auf anderen Methoden aufbauen, wird zwischen der wpa_supplicant Bibliothek und der Methodenimplementierung eine Logikschicht (IC...Wrapper) eingebaut, welche es erlaubt, die noch nicht in JSON umgewandelten Resultate (ICData) in mehreren REST-Methoden abzufragen.

A.4.2 Android

Die Android App besteht aus zwei Modulen, einem Frontend, welches die anzuzeigenden Daten über RxJava asynchron abfragt und einem Modul, welches die p2p Broadcasts abfragt sowie die p2p Verbindung aufbaut. Die App besteht aus einer Reihe einfacher Bildschirme, die größtenteils bloß Listen der abgefragten Daten anzeigen. Wie in Fig.4 zu sehen, ist der Nutzerfluss durch die Bildschirme, dass der Nutzer zunächst eine Liste der in der Nähe aktiven p2p Geräte angezeigt bekommt und eines dieser Geräte auswählt. Durch die Auswahl wird eine p2p Verbindung mit dem Gerät aufgebaut, sodass der REST-Server genutzt werden kann. Während der Nutzer einen Ladebildschirm sieht, wird so die Liste der verfügbaren Netzwerkschnittstellen, die nicht exklusiv für p2p vorhanden sind, abgefragt und dem Nutzer angezeigt. Sollte diese Liste nur einen Eintrag enthalten, wird die Auswahl der Netzwerkschnittstelle automatisch vorgenommen und es wird zur nächsten Ansicht vorangeschritten. Der Nutzer sieht hierbei nach wie vor den Ladebildschirm. Sind jedoch mehrere Elemente in der Liste, wird dem Nutzer die Liste angezeigt. Wählt der Nutzer eine der Netzwerkschnittstellen aus, wird ihm eine Liste der aktuell erreichbaren Netzwerke gezeigt, sodass er das Gerät mit Netzwerken verbinden kann, wie es auch in den Einstellungen des Smartphones möglich ist.

Um die Module miteinander zu verbinden, wird Koin [Android Koin] als Dependency Injection in Verbindung mit öffentlich definierten Schnittstellen genutzt. Koin verzichtet im Gegensatz zu Dagger auf generierten Code. Dadurch ist es zwar im Setup langsamer, jedoch macht es für die eigentliche Injection der Abhängigkeiten keinen erheblichen Unterschied mehr, sodass diese Bibliothek auf Grund ihrer leichten Einbindung genutzt wurde [Android Koin Speed]. Als REST-Client wird hierbei Retrofit in Kombination mit Gson genutzt. Retrofit erlaubt es, die REST-Schnittstelle deklarativ zu definieren, sodass jeglicher Code generiert wird. Dies hat den Vorteil, das kein Boilerplate-Code geschrieben werden muss. In Verbindung mit Gson lässt sich so eine saubere REST-Schnittstelle mit einem klar strukturierten Datenmodell umsetzen, da gson es erlaubt, Datenobjekte aus Klassen zu erzeugen, in denen die Felder mit dem gelieferten JSON Objekt korrespondiert.

A.4.3 Stolpersteine

Bei der Umsetzung des Projektes sind einige unvorhergesehene Probleme und Einschränkungen aufgetaucht. Teilweise erforderten diese ein großes Umdenken oder eine lange Suche nach einer alternativen Lösung.

Die Kommandozeilenschnittstelle des wpa_supplicant ist dem C-Interface unterlegen, da Events immer im Standardkanal ausgegeben werden. Dies führt dazu, dass bei der Nutzung von Befehlen nicht zwischen eingehenden Events und dem Ergebnis des Befehls unterschieden werden kann. Die C-Schnittstelle erlaubt es hingegen, Events unabhängig von Befehlen abzufragen und auf Diese zu warten. Foreign Function Calls sind in pharo zwar aktuell möglich, jedoch sind diese noch nicht vollständig eingebunden. So wird aktuell noch die gesamte Ausführung in der VM für die Dauer des externen Aufrufs gestoppt, da nicht überblickt werden kann, inwie weit der externe Aufruf Speicherveränderungen durchführt und somit Inkonsistenzen entstehen

könnten. Außerdem ist es zum aktuellen Zeitpunkt nicht möglich, mit dem Unified Foreign Function Interface (UFFI) in pharo Callbacks zu Implementieren. Dies bedeutet konkret, dass Events lediglich über ein Polling erhalten werden müssen. Weiterhin ist UFFI noch nicht so gut dokumentiert, dass es ohne viele Vermutungen ausprobiert werden muss. Es wurde sich hierbei zwar an bereits bestehendem Code der git Anbindung orientiert, jedoch ist es teilweise undurchsichtig, welche Typen von der VM automatisch übersetzt werden, wie zum Beispiel char* zwar zu String umgewandelt wird, dies jedoch nicht wie ein Stringbuffer funktioniert.

Beim Testen von UFFI besteht das Problem, dass die Aufrufe, die aus pharo getätigt werden, in einem core dump resultieren, falls ein Fehler auftritt und so nicht gespeicherte Änderungen in der VM verloren gehen. Da hierbei viel mit sehr generischem Code gearbeitet wird, der das String Array aus pharo, welches den C-Funktionsaufruf repräsentiert, in einen tatsächlichen Funktionsaufruf umwandelt. Dadurch ist der stacktrace des Codes, der nativ ausgeführt wird, wenig durchsichtig und man bleibt ohne Fehlermeldung dessen, was tatsächlich schief gelaufen ist. Dies ist besonders problematisch im Hinblick darauf, dass die Typauflösung von UFFI nicht ganz deutlich wurde und man somit auf Trial and Error zurückgreifen musste.

Theoretisch blockieren sich die Verwendung von Wi-Fi Direct und gewöhnlichem Wi-Fi mit einem Access Point auf der selben Netzwerkschnittstelle, da diese eine Sende- und Empfangseinheit exklusiv benötigen. Diese Limitierung wird bei Nutzung des wpa_supplicant jedoch nicht deutlich, da gewöhnliche Schnittstelle und p2p Schnittstelle immer als separate Einheiten auf der gleichen Netzwerkkarte aufgelistet werden. Da manche Netzwerkkarten inzwischen mehr als ein Radio enthalten, um höhere Übertragungsraten zu ermöglichen, kann dies jedoch auf manchen Geräten dennoch funktionieren. Im Falle des Raspberry Pi 3 B ist dies jedoch aktuell nicht möglich und eine zweite Netzwerkkarte wird benötigt, um Wi-Fi Direct und Wi-Fi parallel zu nutzen. Diese Limitierung gilt jedoch nicht für die Service Discovery, die über Wi-Fi Direct stattfindet.

Im Allgemeinen ist die Dokumentation und Spezifikation von Wi-Fi Direct und dessen Implementierung unzureichend. Die Spezifikation konnte von der WiFi Association nicht angefragt werden, somit fehlt sie für dieses Projekt. Da so einzig die Dokumentation des wpa_supplicant für die Umsetzung genutzt werden konnte, fielen auch hier Lücken auf. So sind nicht alle relevanten Events dokumentiert und es ist noch fraglich, ob die genutzten Events tatsächlich das widerspiegeln, was ihr Name suggeriert. Ebenso fehlt ein State-Graph der p2p Umsetzung, wodurch unklar ist, in welchem Zustand des Verbindungsaufbaus die Netzwerkpartner sich wie zu verhalten haben. Außerdem fehlen Angaben zu den Datagrammen der Service Discovery in der Dokumentation, da Diese direkt binär codiert angegeben werden müssen. Es wurde hierbei auf die Open Source Android Implementierung zurückgegriffen.[Android Repository]

Unter Android steht Wi-Fi Direct vom Betriebssystem als Sammlung von Java Klassen zur Verfügung. Problematisch ist hierbei auch wieder, dass Android oder der intern verwendete wpa_supplicant einen Cache besitzt, wodurch Daten, die bereits einmal ausgeliefert wurden trotz einer neuen Anfrage nicht noch einmal ausgeliefert werden. Ebenso scheint es eine Blacklist für Geräte zu geben, zu denen ein p2p

Verbindungsaufbau mehrfach fehlgeschlagen ist, wodurch auch kein Service Discovery Austausch mehr mit diesem Gerät möglich ist. Dieses Problem ließ sich lediglich durch einen Neustart des Smartphones beheben.

Weiterhin besitzt Wi-Fi Direct die Einschränkung, dass standardmäßig lediglich die Clients die IP Adresse des Gruppenhosts mitgeteilt bekommen. Dies bedeutet, dass der Anwendungsserver entweder immer die p2p Gruppe hosten muss oder der Anwendungsclient auf einem fix definierten Port oder per Service Discovery die IP Adresse des Anwendungsservers erwarten muss. Dies deutet ein tieferliegendes Problem an, da dem Gruppenhost lediglich mitgeteilt wird, dass Clients der Gruppe beigetreten sind, nicht aber welche IP Adresse ihnen zugewiesen wurde. Dies stellt ein Problem dar, da manche Netzwerkkarten das Hosten einer p2p Gruppe nicht unterstützen und lediglich als p2p Client agieren können. Dieser Fall trifft auch auf den Raspberry Pi 3 B+ zu. Dadurch wird die simple p2p Verbindung erheblich komplexer, da erst ein Verbindungsdatenaustausch auf Anwendungsebene stattfinden muss.

Aus all diesen Faktoren resultiert, dass die Verbindung nicht reproduzierbar stabil zwischen dem Raspberry Pi und einem Android Smartphone aufgebaut werden kann, wodurch eine Nutzbarkeit für Endnutzer hinfällig wird.

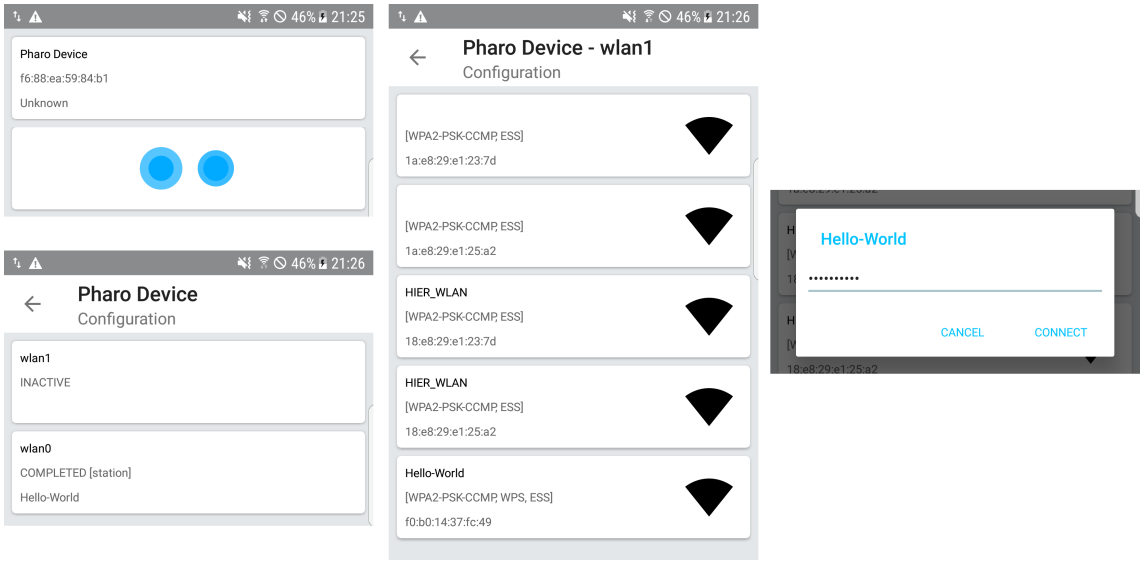

Fig. 4: Der Benutzer sieht zunächst eine Liste an konfigurierbaren Geräten in der Nähe (links oben), durch Auswahl bekommt der Nutzer die Liste der verfügbaren Netzwerkschnittstellen zu sehen (links unten). Sobald auch hier eine Auswahl erfolgt ist, kann der Nutzer eine Liste an verfügbaren und bereits konfigurierten Netzwerken für diesen Netzwerkadapter sehen (mitte). Letztlich kann der Nutzer noch ein Netzwerk durch Eingabe des Passworts konfigurieren (rechts).

A.5 Ausblick

Im Rahmen dieses Projektes wurde Wi-Fi Direct zur Nutzung als p2p Technologie zwischen Android und pharo untersucht. Zwecks mangelnder Dokumentation von Wi-Fi Direct und dessen Implementierung unter Linux war es nicht möglich, eine stabile Verbindung wiederholbar aufzubauen. Da Service Discovery jedoch größtenteils stabil über Wi-Fi Direct funktioniert, ist zu überlegen, ob eine Hybrid Lösung in Verbindung mit Bluetooth oder anderen p2p Technologien sinnvoll ist. Damit wäre es möglich, manche Nachteile von anderen Technologien auszugleichen. In Kombination mit Bluetooth könnte so die Broadcastreichweite spürbar erhöht werden. Generell sollten somit andere Technologien auch auf deren Nutzbarkeit überprüft werden. Für Diese wird es dann auch nötig, einen Wrapper um Socketverbindungen zu bauen. Ebenso ist es denkbar, mehrere p2p Technologien parallel zu betreiben, damit der Nutzer nicht durch eine fehlende Technologie in seinem Smartphone eingeschränkt wird. Hierbei sollte auch darauf geachtet werden, dass der Nutzer nicht von dieser Parallelität behelligt wird, somit die Auswahl automatisch geschieht.

Der implementierte REST-Server verwendet noch den standardmäßigen Error Handler, der im Fehlerfall einen Stacktrace als plain text zurückliefert. Dies sollte abgefangen werden und in ein JSON Objekt mit Fehlercodes und sprechenden Fehlermeldungen umgewandelt werden. Aktuell führt der REST-Server noch alle Aufrufe ohne Authentifizierung durch. Es sollte jedoch wenigstens eine Authentifizierung bei der Erstkonfiguration festgelegt werden, sodass nicht autorisierte Personen keine Änderungen an den Einstellungen vornehmen können. In Verbindung mit dieser Authentifizierung auf Anwendungsebene kann auch das Gerät an ein virtuelles Netzwerk gebunden werden, welches eigene Geräte bündelt. Dieses virtuelle Netzwerk wird dazu genutzt, dass die Geräte einen Server in diesem Netzwerk besitzen, auf den sie sich beziehen können. Dies kann entweder eine lokal laufende pharo Instanz sein oder ein Cloudservice, an dem das virtuelle Netzwerk einem Account entspricht. Damit wird nicht nur der Zugriff lokal sondern auch Remote eingeschränkt. Um diese virtuellen Netzwerke gut verwalten zu können, empfiehlt es sich auch hierbei ein DNS Service Discovery einzusetzen, die Daten jedoch durch verschiedene Maßnahmen zu schützen.[Kaiser, Waldvogel, S.8]

Im Rahmen von Low Energy Netzwerken könnten die Broadcasts der Service Discovery von anderen pharo Instanzen auch erkannt werden und so gebündelt werden. Dies erlaubt es, Energie dadurch einzusparen, dass nur noch ein Gerät aus dem Low Energy Netzwerk den Broadcastanfragen antworten muss. Ebenfalls lässt sich so die Reichweite von p2p Verbindungen erhöhen, da die Anfragen zum entsprechenden Ziel weitergeleitet werden können. Im Hinblick auf Bandbreite stellt dies erst ein Problem dar, wenn mehrere Geräte im selben Netzwerk parallel administriert werden, was jedoch einen Randfall darstellen sollte und somit vorerst ignoriert werden kann. Es ist somit also denkbar die pharo Instanzen als Knoten in einem Netzwerk aus oben genannten Gründen agieren zu lassen. Diese Knoten müssten dann jedoch ebenfalls Routingtabellen verwalten, was dieses dezentrales Netzwerk im Vergleich zum Nutzen, der daraus gewonnen wird, zu kompliziert macht.

Quellen

[Aneja et al.] Nagender Aneja und Sapna Gambhir: "Profile-Based Ad Hoc Social Networking Using Wi-Fi Direct on the Top of Android" *Mobile Information Systems, Volume 2018, Article ID 9469536, 7 pages* (2018).

[Eberhardt et al.] Udo Eberhardt und Hans Joachim Kelm [Hrsg.]: *USB - Universal Serial Bus* Franzis, Poing (1999).

[Esnaashari et al.] Shadi Esnaashari, Ian Welch und Peter Komisarczuk: *Determining home users' vulnerability to Universal Plug and Play (UPnP) attacks* Erschienen in: Proceedings of the 2013 27th International Conference on Advanced Information Networking and Applications Workshops (WAINA). IEEE, 2013. - S. 725-729. - ISBN 9781467362399 .

[Kaiser et al.] Daniel Kaiser, Marcel Waldvogel, Holger Strittmatter und Oliver Haese: *User-Friendly, Versatile, and Efficient Multi-Link DNS Service Discovery* Erschienen in: Proceedings 2016 IEEE 36th International Conference on Distributed Computing Systems Workshops : ICDCSW 2016. - Piscataway, NJ : IEEE, 2016. - S. 146-155. - ISBN 978-1-5090-3686-8.

[Kaiser, Waldvogel] Daniel Kaiser und Marcel Waldvogel: *Adding Privacy to Multicast DNS Service Discovery* Erschienen in: Proceedings - 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) ; Beijing, China, 24 Sep - 26 Sep 2014. - Piscataway : IEEE, 2014. - S. 809-816. - ISBN 978-1-4799-6513-7.

[Langer et al.] Josef Langer und Michael Roland: *Anwendungen und Technik von Near Field Communication (NFC)* Springer, Heidelberg (2010).

[Lüders] Christian Lüders: *Lokale Funknetze: Wireless LANs (IEEE 802.11), Bluetooth, DECT* Vogel, Würzburg (2007).

[Morrow] Robert Morrow: *Bluetooth Operation and Use* McGraw-Hill (2002).

[Sauter] Martin Sauter: *Grundkurs Mobile Kommunikationssysteme: LTE-Advanced, UMTS, HSPA, GSM, GPRS, Wireless LAN und Bluetooth* 6.Auflage Springer Vieweg, Wiesbaden (2015).

[Sikora] Axel Sikora: *Wireless LAN: Protokolle und Anwendungen* Addison-Wesley, München u.A. (2001).

[Townsend et al.] Kevin Townsend, Carles Cuffí, Akiba und Robert Davidson: *Getting Started with Bluetooth Low Energy* O'Reilly (2014).

[Android Koin] https://insert-koin.io/

[Android Koin Speed] https://medium.com/koin-developers/ready-for-koin-2-0-2722ab59cac3

[Android Accessory] https://source.android.com/devices/accessories/custom

[Android OkHttp] https://github.com/square/okhttp

[Android Permissions] https://elinux.org/Android_Security#Paranoid_network-ing

[Android Repository] https://android.googlesource.com/platform/frameworks/base

[iot-connectivity] https://github.com/janphkre/iot-connectivity

[iot-p2p-test] https://github.com/janphkre/iot-p2p-test

[JSON Schema] https://json-schema.org/

[Linux Packet] http://man7.org/linux/man-pages/man7/packet.7.html

[OpenAPI] https://swagger.io/docs/specification/about/

[Pharo] http://pharo.org/

[Pharo OpenAPI] https://github.com/zweidenker/openapi

[Pharo Things] https://github.com/pharo-iot/PharoThings

[Pharo Zinc] https://github.com/zweidenker/zinc/

[WiFi Direct] https://www.wi-fi.org/discover-wi-fi/wi-fi-direct

[WiFi HostAP Repository] git://w1.fi/hostap.git

[WiFi wpa_supplicant] https://w1.fi/wpa_supplicant/

[Zweidenker] https://zweidenker.de

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Wermelskirchen, den 23. August 2019

Jan Phillip Kretzschmar