Technology
Arts Sciences
TH Köln

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

Bachelorarbeit Sommersemester 2019

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

Betreuer (Zweidenker GmbH):
Anton Borries *(anton.borries@2denker.de)*

Betreuer (TH Köln):
Prof. Christian Kohls

1. Juli 2019

Contents

1 Einleitung

Die Verbindungskonfiguration von IoT Geräten wurde bereits auf Basis von Wi-Fi Direct untersucht[1]. 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 PharotThings eingebunden werden, um auch hier Probleme aufzudecken und zu einer optimalen Lösung des Problems zu gelangen.

Die vorhergehende Lösung [2] besteht aus Service Discovery und einer p2p Verbindung auf Basis von Wi-Fi Direct. Ob Service Discovery in seiner aktuellen Form beibehalten werden soll, ist ebenfalls zu evaluieren, da sich eine erweiterte Lösung auch nicht auf nur eine Technologie beschränken muss. Es ist daher zu überprüfen, ob eine Hybridlösung in Verbindung mit Wi-Fi Direct oder Bluetooth LE sinnvoll ist, sodass lediglich die p2p Verbindung über eine andere Technologie stattfindet.

Softwarequalität muss definiert werden.[3]

1.1 Evaluationsziel

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 und es 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 Erfolgs 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, in wie weit 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

[1] Kretzschmar: *Verbindungskonfiguration von PharoThings auf Raspberry Pi durch Android App* TH Köln Praxisprojekt Sommersemester (2019).

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

[3] TODO: Liggesmeyer

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, in wie weit 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 willen, benötigt er eine relativ latenzfreie und hochvolumige 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 (LE), 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 [4] definiert. Hierbei soll auf die folgenden Punkte eingegangen werden und in wie weit diese extern gemessen werden können, da nicht einsehbar ist, wie ausgelieferte Implementierungsartefakte intern getestet werden und dokumentiert sind.

1. *Zuverlässigkeit:* Um die Robustheit der Technologien feststellen und vergleichen zu können, muss festgestellt werden können, wie viele Fehler in der genutzten Implementierung existieren. 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. 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 aussagekräftiges 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 der erwarteten Funktionalität kann nur sehr eingeschränkt überprüft werden, da aus einer externen Sicht die Menge der Funktionalitäten sich auf den Verbindungsaufbau und -abbau, sowie ein Senden und Empfangen von verbindungslosen Daten beschränken und alle Technologien diese Funktionalitäten anbieten. Es muss somit darauf geachtet werden in wie weit die internen Zustände dokumentiert sind und erreicht werden können.

4. *Effizienz:* Die Effizienz der Technologien kann durch eine Vergleich von Datendurchsatzraten und Antwortzeiten festgestellt werden. Diese beiden Kennzahlen sind für eine Verbindungskonfiguration nicht kritisch, jedoch können

[4] Quelle

mittels eines externen HTTP-Wrappers auch Drittprogramme eine solche p2p Verbindung nutzen.

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 Referenzwerte und Messbarkeit

1.4 Einschränkungen

Um den Aufwand der Arbeit bewältigbar 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 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 und im Support bedeutet. [5] 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 sind.

2 Warum p2p?

- Warum p2p
  - Präzisierung IoT - Internet of Things

2.1 Services

- Definition eines Services
  - Anbieten eines Services als Client/Server-Modell

[5] Nachteil Qualitätsmodell: Bezug zu Kosten und Nutzen [Lange Qualitätsmodelle]

2.2 Servicestabilität (Todo: entfällt eventuell)

- API Versionierung
  - Absichern von Änderungen durch Contract-Driven-Development
  - Integrationstests

2.3 Netzwerktypen

- LAN, WAN, etc.
  - ad hoc, Stern-basiert, Bus-basiert

3 Grundlagen der Verbindungstechnologien

3.1 Bluetooth

3.2 NFC

3.3 USB

4 p2p Verbindung

Im Folgenden wird die Implementierung einer p2p Verbindung über Bluetooth, NFC und USB beschrieben. Jede dieser Technologien soll zunächst separat betrachtet werden, um Details und Probleme bei der Implementierung aufzuzeigen.

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.

4.1 HTTP Kapselung

Um eine p2p Verbindung unabhängig von der genutzten Verbindungstechnologie nutzen 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. Auf der Serverseite stellt dies 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.

Listing 1: Instanziierung eines Sockets (Server: 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));
```

[6] [7]Der kapselnde Server baut 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. Dies erfüllt sonst die Anfangskriterien eines slow-louis-Angriffes auf einer einzelnen Verbindung. [8]

Listing 2: Datenweiterleitung durch Sockets (Server: C)

```
void* hookSockets(void* data) {
    SocketInfo sockets = *((SocketInfo*) data);
    char buffer[BUF_SIZE] = { 0 };
    while(pipeData(sockets.readingSocket, sockets.writingSocket, buffer)
    return NULL;
}

int pipeData(int sourceSocket, int sinkSocket, char* buffer) {
    int bytes_read, bytes_sent;

    bytes_read = read(sourceSocket, buffer, BUF_SIZE);
    if(bytes_read < 0) return -1;
    if(bytes_read == 0) return -20; // indicates a EOF

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

    return 0;
}
```

[9]

Die bestehende Socket-Verbindung zum Zielserver wird genutzt, um die ankommenden Daten und deren Antworten voll duplex weiterzuleiten. Dies ist realisiert, indem die Method *hookSockets* als zwei Threads mit invertiertem *SocketInfo* gestartet werden. 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.

[6]TODO:RAHMEN UM CODE
[7]TODO: REFERENZ ZUM CODE BEISPIEL
[8]Quelle?
[9]TODO:RAHMEN UM CODE

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[10] als HTTP-Client genutzt. Um zu verstehen, welche Änderungen am Client nötig sind, sollte zuerst die interne Struktur der okhttp Bibliothek erläutert werden.

Listing 3: Interner Aufbau von okhttp (Client: Java)

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

[11][12] Aufrufe werden 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 *HTTPCodec* genutzt, um die Anfrage auf den Socket zu schreiben und die Antwort zu lesen.

Listing 4: Änderungen an okhttp (Client: Kotlin)

```
val interceptors = ArrayList<Interceptor>()
...
interceptors.add(BridgeInterceptor(wrapper.cookieJar()))
```

[10] Referenz

[11] Todo: Rahmen um Code

[12] Quelle zu Datei (github: https://github.com/square/okhttp/blob/okhttp_3.11.x/okhttp/src/main/java/okhttp3/R
Zeile 185-194)

```
interceptors.add(CacheInterceptor(wrapper.internalCache()))
...
interceptors.add(SimpleServerInterceptor(wrapper.httpCodec()))
```

[13] Im Gegensatz zur okhttp Implementierung muss die p2p Verbindung so verwaltet werden, wie es von der Implementierung der Technologie vorgegeben wird. Dazu wird der *RetryAndFollowUpInterceptor* sowie der *ConnectInterceptor* weggelassen und der *CallServerInterceptor* im *SimpleServerInterceptor* soweit vereinfacht wird, sodass 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, genutzt werden.

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.

4.2 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 hierbei 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 Socket Verbindungen ein automatisches Pairing mit Schlüsselaustausch durchführen, ist keinerlei Eingriff oder Bestätigung des Nutzers nötig, um Daten übertragen zu können.

Listing 5: Verbindungsaufbau mit Bluetooth (Client: Kotlin)

```
val bluetoothDevice = bluetoothAdapter.bondedDevices?.firstOrNull { bluet
    bluetoothDevice.address == device.bluetoothDetails.mac
} ?: bluetoothAdapter.getRemoteDevice(device.bluetoothDetails.mac)

val method = bluetoothDevice::class.java.getMethod("createInsecureRfcommS
val socket = method.invoke(bluetoothDevice, device.bluetoothDetails.port)
socket.connect()
```

[14] Für den Verbindungsaufbau muss die Bluetooth MAC-Addresse sowie der Bluetooth Port angegeben werden. Der *BluetoothAdapter* aus dem Android Framework

[13] Todo: Rahmen um Code
[14] Todo: Rahmen um Code

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 *BluetoothDevice::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 festlegen der Portnummern zu vermeiden, da lediglich 30 RFCOMM Ports zur Verfügung stehen.[15] 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. Um diese exemplarische Implementierung 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 auffindbar für den Client sein.

Listing 6: Verbindungsaufbau mit Bluetooth (Server: 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);
```

[16] Auf Seite des Servers kann die Implementierung der HTTP Kapselung fast vollständig übernommen werden,jedoch müssen Verbindungen auf einem weiteren Socket mit dem Bluetooth RFCOMM Protokoll akzeptiert werden.

Mit diesen beiden Anbindungen von Bluetooth an die generische HTTP Kapselung lässt sich diese simple Lösung bereits nutzen. Um jedoch die geheimen Methoden unter Android nicht nutzen zu müssen, 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.[17] 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.

Listing 7: Veraltete Nutzung von SDP (Server: C)

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

[15] Quelle z.B. https://people.csail.mit.edu/albert/bluez-intro/x148.html

[16] Todo: Rahmen um Code

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

[18] In älteren Versionen von BlueZ war es möglich einen SDP Eintrag mit der Methode *sdp_record_register* anzulegen. 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.

Listing 8: DBus Nutzung von SDP (Server: C)

```
static DBusHandlerResult wrapper_messages(DBusConnection* connection, DB
    const char* interface_name = dbus_message_get_interface(message);
    const char* member_name = dbus_message_get_member(message);
    if (0==strcmp("org.bluez.Profile1", interface_name)) {
        if(0==strcmp("Release", member_name)) {
            profileRelease();
            return DBUS_HANDLER_RESULT_HANDLED;
        } else if(0==strcmp("NewConnection", member_name)) {
            profileNewConnection(message);
            return DBUS_HANDLER_RESULT_HANDLED;
        } else if(0==strcmp("RequestDisconnection", member_name)) {
            profileRequestDisconnection();
            return DBUS_HANDLER_RESULT_HANDLED;
        }
    }
    return DBUS_HANDLER_RESULT_NOT_YET_HANDLED;
}

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, NUI

DBusMessage* msg = dbus_message_new_method_call("org.bluez", "/org/bluez"
...
dbus_connection_send_with_reply_and_block(conn, msg, -1, &err);
```

[19][20] Um die SDP Funktionalität dennoch nutzen zu können, ist es nötig zunächst ein Callback-Objekt zu registrieren. 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 Vergöeoch zu einer Lösung ohne SDP stark erhöht wird, da nicht nur die DBus Nachrichten auf einem separaten Thread gehandhabt werden müssen, sondern auch eine Auflösung zwis-

[18] Todo: Rahmen um Code
[19] Todo: Rahmen um Code
[20] TODO: Code Beispiel istzu lang.

chen 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.

4.3 NFC

4.4 USB

5 Evaluation (Todo: Besserer Name)

5.1 Zuverlässigkeit

5.2 Wartbarkeit

5.3 Funktionalität

5.4 Effizienz

5.5 Bedienbarkeit

5.6 Zusammenfassung

6 Ausblick