|  |  |
| --- | --- |
|  | Fakultät Informationstechnik  Studiengang Technische Informatik |
| Bachelorarbeit  Netzunabhängige lokale Peer-To-Peer-Verbindungen auf mobilen Geräten  Alexis Danilo Morgado dos Santos | |

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

|  |  |  |
| --- | --- | --- |
|  | Vorgelegt von  Am  Referent  Korreferent | Alexis Danilo Morgado dos Santos  17.06.2020  Prof. Dr. Peter Barth  Prof. Dr. Mark Hastenteufel |

Schriftliche Versicherung laut Studien- und Prüfungsordnung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine

anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

|  |  |
| --- | --- |
| Mannheim, den 17.06.2020 |  |
|  | Alexis Danilo Morgado dos Santos |

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Peter Barth für die durchgehende Unterstützung, Geduld und dessen Rat in Bezug auf fachspezifischen Problemen, sowie schreibtechnische Angelegenheiten und die Bereitschaft zum mehrmaligen Korrekturlesen trotz aktueller Corona Krise bedanken.

Weiterhin möchte ich mich bei Herr Prof. Dr. Mark Hastenteufel für die Bereitschaft zum Korreferenten für diese Thesis bedanken.

Zu guter Letzt gilt mein besonderer Dank meiner Familie, die mich während meines bisherigen Studiums jederzeit finanziell und moralisch unterstützt hat.

Zusammenfassung

Die vorliegende Thesis beschäftigt sich mit der Konzeption und Realisierung einer Bibliothek, welche den Aufbau und die Kommunikation eines mobilen Netzwerkes für Android basierte Systeme unabhängig von der Internetanbindung gewährleistet. Als zugrunde liegende Technologie wird im offline Fall die Nearby Connections API und im online Fall die MQTT API gewählt. Die APIs werden durch Adapter erweitert, um so eine einheitliche Schnittstelle anbieten zu können.

Dazu wird ein zustandsbehaftetes autonom aufbauendes sternenförmiges Netzwerk mithilfe der Nearby Connections API erstellt, welcher einen MQTT ähnlichen Verbindungsaufbau erlaubt.

Für die Datenübertragung beider APIs werden Kanal, Priorität, Sendepuffer sowie ein mqtt-seitiges Chunking-Mechanismus für große Daten über 30 KByte erstellt.

Im Anschluss werden sowohl die für den Verbindungsaufbau als auch die für das Senden von Daten nötigen Zeiten beider unterliegender Technologien gemessen und evaluiert. Dabei zeigt sich das Nearby Connections beim Verbindungsaufbau verglichen mit MQTT schlecht skaliert und bereits für die Verbindung zwischen zwei Knoten 7.5 bis 11 Sekunden benötigt. Ein Netzwerk mit sechs Knoten braucht etwa eine Minute, während MQTT dasselbe Netzwerk in 454 ms aufbauen kann. Bei der Datenübertragung zwischen zwei Knoten weist Nearby Connections bei aktivem Hotspot drei Mal höhere Datenraten als MQTT auf, während MQTT bei mehr als drei Teilnehmern bessere Datenraten aufweist. Ist der Hotspot hingegen deaktiviert ist die Datenübertragung um Faktor 27 langsamer. Auch zeigt sich die Eignung von Nearby Connections für offline Datenübertragung innerhalb eines Netzwerks bestehend aus bis zu sieben Teilnehmen bei mittleren bis kleinen Datenverkehr.

Inhaltsverzeichnis

1 Einleitung 1

2 Anforderungsanalyse 3

2.1 Szenarien und Features 3

2.2 Funktionale und Nichtfunktionale Anforderungen 6

2.3 Gegenüberstellung von Nearby Connections und MQTT 8

3 Software-Design 13

4 Verbindungsaufbau 15

4.1 Obligatorische Berechtigungen 15

4.2 Sichtbarkeitsbeschränkung 16

4.3 Wahl der Topologie 17

4.4 Erstellung des autonomen Peer-to-Peer Netzwerks 18

5 Übertragung von Daten 22

5.1 Senden mit Nearfly 22

5.2 Realisierung mit NearbyConnections 23

5.3 Realisierung mit MQTT 26

6 Evaluation 29

6.1 Verwendete Testgeräte 29

6.2 Zeiten für den Verbindungsaufbau 30

6.3 Zeiten für die Datenübertragung 35

6.4 Verifikation der Anforderungen 38

6.5 Nutzung der Nearfly API 40

7 Résumé 42

Anhang A - Ausführlichere Diagramme zum Verbindungsaufbau im NeCon-ConnectionMode 43

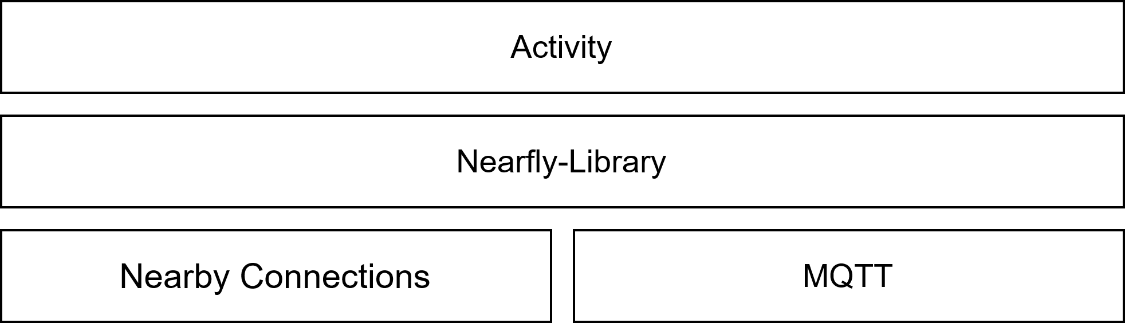
Literaturverzeichnis 44

Online-Quellen 44

# Einleitung

Die Konnektivität ist der Zeitgeist des 21. Jahrhunderts und als Megatrend die treibende Kraft des graduellen gesellschaftlichen Wandels. Gerade Smartphones, welche der Gesellschaft die Möglichkeit des ständigen Informationsabrufs und unentwegte Disponibilität ermöglichen, tragen eine nicht unwesentliche Rolle zum steigenden Bedarf einer ubiquitären Konnektivität bei. Lange erfüllen diese dem Handy entlehnten mobilen Computer nicht mehr den primären Zweck des Telefonierens, sondern werden hauptsächlich zum Nachrichtenaustausch, Browsing, sowie andere Internetgebundene Aktivitäten benutzt [1]. Doch existieren auch Situationen, in denen diese Vorteile wünschenswert sind, eine internetgebundene Vernetzung jedoch unangebracht oder schlichtweg nicht möglich ist. Etwa, wenn Multimedia-Daten aus datenschutzrechtlichen Gründen dezentral transferiert werden sollen, kein Datenvolumen aufgebraucht werden soll oder dringende Wetterwarnungen in Gebieten verbreitet werden sollen, in denen das Mobilnetzt z.B. aufgrund eines Ausfalls oder mangelnder Infrastruktur (zeitweise) nicht verfügbar ist[3]. Für die Realisierung dieser vernetzenden Funktionalitäten wird häufig auf bestehende Technologien, wie etwa Bluetooth, Wifi-Direct oder die im Juli 2017 von Google veröffentlichte Nearby Connections API zurückgegriffen. Während internetgebundene Kommunikation etwa durch MQTT möglich ist. Die Technologien werden in Form von Bibliotheken, welche komplexe Sachverhalte abstrahieren und durch eine API ansprechbar sind, in das Projekt eingebunden.

Im Rahmen dieser Thesis soll, die NearbyConnection- und die MQTT-Technologie analysiert und eine Android Middleware konzipiert und implementiert werden, welche die Vorzüge beider unterliegenden Technologien in Form einer auf beide Technologien aufsetzenden Ebene vereinheitlicht und dem Android-App-Entwickler die Möglichkeit einer generischen Nutzung der vernetzenden unterliegenden Offline-, sowie Online-Lösung (wie in **Abbildung 1**) bietet.



**Abbildung 1:** Abstrahierte drei-Ebenen Darstellung der Nearfly-Bibliothek

Die Anforderungsanalyse (Kapitel 2) definiert ein Messenger, ein rundenbasiertes Spiel und zwei direkt-Interaktive Spiele, aus welchen die Anforderungen für die Nearfly-Bibliothek ermittelt werden. Die Nearfly-Bibliothek soll dabei große multimediale Daten und kleine Nachrichten, in naher Echtzeit und bei möglichst kleiner Latenz übertragen. Durch die Anschließende Gegenüberstellung wird gezeigt, dass das Angleichen beider unterliegender Technologien fordert, dass in Nearby Connections zunächst eine Strategie für den Netzwerkaufbau und das Verteilen von Nachrichten an alle Netzwerkteilnehmen, wie dies in MQTT der Fall ist, gewählt werden muss.

Im Kapitel 3 (Software-Design) wird jeweils eine Mqtt- und NearbyConnections-Adapter-Komponente für das Erweitern der unterliegenden Technologien definiert. Weiterhin wird definiert, dass die Nearfly-Bibliothek durch ein NearflyClient-Objekt benutzt und auf Nachrichten mithilfe des Nearfly-Listener gehört werden kann.

Im Kapitel 4 (Verbindungsaufbau) wird gezeigt, dass die Nearfly-Bibliothek sieben mandatorischen und zwei optionale Berechtigungen fordert, von denen die Berechtigungen für den Standort-, sowie Datenzugriffe zur Laufzeit abgefragt werden müssen. Durch Erweiterung der NearbyConnections API wird ein sternförmiges Netzwerk mithilfe eines Zustandsautomaten realisiert.

In Kapitel 5 (Übertragung von Daten) werden zwei Methoden für das Senden von Daten mithilfe der Nearfly-Bibliothek definiert. Dabei wird unterschieden zwischen Daten unter 32 KByte und großen Binären Daten, welche vor der Übertragung zerlegt werden.

Anschließend zeigt die Evaluation (Kapitel 6 ), dass der Verbindungsaufbau mithilfe von Nearby Connections bereits für den ersten Knoten 7.5 bis 11 Sekunden braucht und nicht gut skaliert, während mit Mqtt maximal zwei Sekunde benötigt. Nearby Connections zeigt sich hingegen bei der Übertragung in kleine Netzwerke bestehend aus zwei Knoten schneller.

Der Résumé zeigt, dass Nearby Connections eher für kleinere Netzwerke bestehend aus zwei bis drei Knoten vorzuziehen ist, während Mqtt für größeren Netzwerken eingesetzt werden sollte. Steht hingegen kein Internet zur Verfügung kann Nearby Connections für bis zu sieben Knoten benutzt werden.

# Anforderungsanalyse

Während die Szenarien Messenger, Share Touchpoint Canvas, Boucing Ball und Score Board Notepad Anwendungsfälle definiert werden, in welche unterschiedliche Datenmengen mit unterschiedlichen Latenzen und unterschiedlichen Datenraten übertragen werden müssen, leiten die Funktionale Anforderungen aus den Szenarien ab, dass die Nearfly Biblitohek kleine bis große priorisierbare und nur im Anwendungskontext sichtbare Nachrichten senden soll. Diese sollen verzögerungsarm und mit nahe Echtzeit Datenraten gesendet werden. Wird die Verbindung abgebrochen müssen die Daten gehalten und die Verbindung erneut aufgebaut werden. Durch die nichtfunktionale Anforderung wird Zielplattform definiert.

Die Gegenüberstellung von Nearby Connections und MQTT macht deutlich, dass ein Assimilieren beider Technologien eine recht große Anpassung beider APIs fordert. So ist etwa eine Verbindung seitens MQTT mit dem Verbinden zum Server getan, während in Nearby Connections für den Netzwerkaufbau zunächst eine Netzwerktopologie gewählt und eine Verbindung zu jedem Knoten einzeln hergestellt werden muss. Auch werden in MQTT Byte-Nachrichten zu allen Netzwerkteilnehmern gesendet werden, während in Nearby Connections zwar Byte, File und Stream Nachrichten gesendet werden können, doch diese zunächst nur an einen Knoten gehen.

## Szenarien und Features

Die Szenarien decken die Datenrate von kleinen bis große Datenmengen und langen Übertragungszeit bis nahe Echtzeit auf. Während beim *Messenger* auch größere binäre Daten übertragen werden, müssen Berührungspunkte beim *Shared Touchpoint Canvas* sehr schnell (nahe Echzeit) versandt werden. Als praxisnahes Extrem-Szenario für eine geringe Latenz dient *Bouncing Ball*, welche eine Übertragungsfrequenz von 30 Paketen pro partizipierenden Nutzer fordert. Das Szenario *Score Board Notepad* muss als rundenbasiertes Spiel ausfallsicher sein, wobei die Geschwindigkeit im Sekundenbereich liegend darf und nur wenige Daten übertragen werden. Während das Feature sichtbarkeitsbeschränkter Nachrichtenaustausch dafür sorgt, dass der Nachrichtenaustausch unterschiedlicher Szenarien nicht interferiert.

### Messenger

*Marius ist der Scrum-Master eines fünf-köpfigen Entwicklungsteams für mobile Anwendungen und damit zuständig für die Zusammenarbeit unter den Entwicklern. Er legt viel Wert auf Datenschutz und möchte am besten keine Daten über das Internet übertragen. Dazu hat er eine Gruppe erstellt und fordert seine Teammitglieder auf seiner Gruppe beizutreten. Danach schreibt er an allen einen Begrüßungstext und frag, ob es derzeit Probleme geben würde. Während Tommy die Nachricht liest läuft dieser zur Toilette, welche durch die dicken Wände ein Funkloch ist und antwortet Marius, dass die Kaffee-Pads leer seien. Kurze darauf sendet Tommy ein Foto von den Kaffee-Pads, mit der Bitte an Marius, neue zu erwerben.*

Das erste Szenario entspricht dem einer klassischen Messenger App, welche es dem Entwicklerteam erlaubt offline zu kommunizieren. So soll der Nutzer die Möglichkeit haben, Text- und Multimedia zu versenden. Denkbar ist jedoch auch das Senden von beliebigen Binärdaten und damit etwa den Datentransfer von ausführbaren Daten zu ermöglichen. Weiterhin soll durch das Erstellen und Betreten von geschlossenen Benutzergruppen (Chatrooms) eine Isolation innerhalb der App geschaffen werden dürfen, welche die Sichtbarkeit dedizierter Nachrichten auf befugte Nutzer reduziert. Gesendete Nachrichten sollen zudem im Falle einer getrennten Verbindung nicht verloren gehen, sondern so lange aufbewahrt werden, bis diese versendet werden können.

**Tabelle 1:** Empirisch ermittelte durchschnittlicher Größenbereich von Multimedia-Daten[[1]](#footnote-1)

|  |  |  |
| --- | --- | --- |
| Datentyp | Durchschnittliche Größe in MByte | |
| Von | Bis |
| Dokumente (PDF) | 0.1 | 1 |
| Fotos (JPG) | 1 (4032x3024) | 5 (4032x3024) |
| Screenshots (PNG) | 0.4 - 3 | 3 |
| Mobile Anwendungen (APKS) | 5 | 62,5 |

Da das Entwicklungsteam hauptsächlich Bilder oder Text über das Smartphone übertragen will, zeigt **Tabelle 1** die durchschnittliche Datengröße, der auf drei verschiedenen Smartphones erfassten Multimedia-Daten. Ein bis wenige Seiten große PDFs weisen dabei eine durchschnittliche Größe von 60-800 KByte auf, während Fotos oder Screenshots, welche von den Smartphones geschossen werden, teilweise stark von Bilddetail und Smartphone-Modell abhängen. Durchschnitt lassen sich diese im Größenbereich 1-4MByte einordnen. Da das Team auch Android Anwendungen entwickelt und dadurch APKs zwischenzeitlich im Rahmen von Systemtests versandt werden müssen, zeigt **Tabelle 1** weiterhin den Größenbereich von Android Anwendungen[[2]](#footnote-2). Die Anwendungen mit 5 Mbyte [5] haben im Play Store durchschnittlich die wenigsten Downloads, während die meisten Menschen größere Anwendungen mit durchschnittlich 62,5 MByte [5] downloaden und besitzen.

### Shared Touchpoint Canvas

*Tom, Beni und Jim sitzen im Büro und zeichnen gemeinsam mithilfe ihrer Smartphones auf einer Berührungspunkte-basierten virtuellen Leinwand. Da jeder Benutzer maximal 10 Berührungspunkte setzen kann, die recht schnell wieder ausgeblendet werden, kommt Tom eine Idee. Er fordert seine Freunde auf, gemeinsam mit ihm ein Auto zu zeichnen. Jeder der Freunde sucht sich dazu ein Bereich des Autos aus, das dieser zeichnen möchte und Beni gibt das Start-Signal damit die Berührungspunkte möglichst zeitgleich eingeblendet werden.*

Das kollaborativen Szenario umfasst eine Leinwand, welche gemeinsam von allen beteiligten Benutzern durch Antippen bemalt werden kann. Berührt ein Benutzer die Leinwand, entsteht ein kolorierter Berührungspunkt (Touchpoint), welcher nach Ablauf einer gewissen Zeit verschwindet. Da alle gemeinsam zeichnen wollen, müssen die die Daten möglichst zeitnah empfange werden. Die Datenpakete sind dabei klein und beinhalten lediglich die Koordinaten, sowie die Farbe der Berührungspunkte. Alle Teilnehmer senden ihre Daten und empfangen alle gesendeten Daten.

### Bouncing Ball

*Mary, Robert und Frederik befinden sich während der großen Pause im Klassenzimmer und versuchen durch Schwenken einer virtuellen Plattform gemeinsam eine Kugel zu balancieren, welche zu Beginn vom Spiel willkürlich angestoßen wurde. Da das Verhalten der Kugel, die Summe der Aktionen aller Spieler ist, müssen alle jeweils die Bewegungen der Mitspieler kompensieren. Nachdem die Kugel erfolgreich zentriert wurde, wird diese erneut einem Impuls ausgesetzt. Dies wiederholt sich einige Male bis die Kugel letzten Endes nach vier Minuten aus der Plattform fällt und die persönliche Bestzeit auf allen drei Geräten angezeigt wird.*

Beim dritten Szenario werden sehr viele Daten innerhalb kürzester Zeit übertragen. Dazu müssen die Neigungsdaten aller Spieler kontinuierlich erfasst und die Kugel entsprechend der Summe aller erfassten Neigungsdaten bewegt werden. Zeitgleich fordert das Spiel eine möglichst geringe Latenz, damit die einzelnen Spieler ohne Reaktionszeit entsprechend balancieren können und dadurch ein durchgehend synchrones Spielerlebnis (gleichend dem gemeinsamen Festhalten eines physischen-Objektes) zu gewährleisten.

### Score Board Notepad

*Steffan, Mark und Ricky spielen gemeinsam in der Mittagspause Papierwerfen, dazu versucht jeder je Runde mit zehn zerknüllten Papierbällen die in der Ecke stehende Mülltonne zu treffen. Derjenige der als letztes dran war, ist zuständig für das Notieren der Punkte desjenigen Spielers, welcher gerade dran ist. Mark beginnt mit der Rolle des Schreibers, während die zwei anderen Mitspieler kontrollieren, ob Mark die Punkte richtig vergibt. Ricky startet und trifft die Mülltonne. Der Ball berührt zuvor jedoch die Wand, sodass Mark Rickys Punktestand nur um eins inkrementiert. Daraufhin trifft Ricky die Mülltonne ohne, dass der Ball etwas berührt und erhält hierfür von Mark zwei Punkte. Acht weitere Versuche später, wird Rickys Runde beendet. Nun ist Steffan dran und sein Vorgänger Ricky notiert Punkte. Nach Ablauf von 20 Runden, in denen jeder jeweils 10-mal werfen durfte wird das Spiel beendet und der Gewinner bestimmt.*

Beim rundenbasierten Spiel müssen Daten zur Steuerung des Spiels innerhalb dynamischer Zeitintervalle versandt werden. Dabei muss sowohl die Neuzuordnung des Schreibers wie auch das Ermitteln des aktiven Spielers und der Zuschauer erfolgen. Weiterhin muss das System eine einer Smartphone-übergreifende Synchronisation der Punktetabelle gewährleisten.

### Sichtbarkeitsbeschränkter Nachrichtenaustausch

*Leonard, Mandy und Kevin befinden sich im Klassenzimmer und spielen in der großen Pause das „Bouncing Ball“-Spiel, während im benachbarten Zimmer, die Lehrer den bevorstehenden Unterricht vorbereiten und sich gegenseitig Bilder und Artikel über die Nearfly Messenger App senden.*

Das Feature zum sichtbarkeitsbeschränkten Nachrichtenaustausch zeigt, dass der Datenaustausch parallellaufender Anwendungen unterschiedliche Szenarien unsichtbar für die jeweils andere Anwendung ist. Dabei sollen sich die Übertragungen unterschiedlicher Geräte, so wenig wie möglich stören.

## Funktionale und Nichtfunktionale Anforderungen

Die aufgestellte Anforderungsmatrix (Tabelle 2) zeigt die Korrelation der für die der o.g. Szenarien ermittelten Anforderungen. Tabelle 2 zeigt, dass das Versenden von textbasierten Nachrichten von allen Szenarien z.B. zum Senden von Chatnachrichten, Status oder Koordinatendaten gefordert wird. Beim Messenger-Szenario wird diese Anforderung, um das Senden von Binärobjekten erweitert wird. Des Weiteren ergibt sich aus dem Messenger-Szenario aufgrund der angeforderten Isolation für Raum und Chat die Anforderung, dass unterschiedlicher Nachrichten innerhalb derselben Anwendung trennbar sein sollen. Im Falle eines ungewollten Verbindungsabbruch sollen wichtige Nachrichten, wie zusendende Bild- oder Textnachrichten beim Messenger oder die Punkte beim Score Board Notepad gehalten werden und bei einem erneuten Verbindungsaufbau gesendet werden. Dies setzt natürlich voraus, dass die Verbindung im Falle eines Verbindungsabbruchs automatisch wiederaufgebaut wird. Die Spiele und Animationsszenarien verlangen zudem eine nahe Echtzeit Latenz, welche eine flüssige Animation (mindestens 25 FPS) und direkt-interaktive Steuerung erlaubt. Zuletzt entsteht für alle Szenarien die Anforderung, dass der Nachrichtenaustausch unterschiedliche Anwendungen, welche die Nearfly-Bibliothek verwenden nicht interferieren darf.

Weiterhin sollte das System sowohl beim Nutzen der Nearby Connections API, als auch beim Nutzen der MQTT API ein ähnliches Verhalten in Bezug auf den Verbindungsaufbau, das Senden und dem Empfangen der Nachrichten aufweisen, sodass beide unterliegenden Technologien für dieselben Szenarien genutzt werden können.

Tabelle 2: Anforderungsmatrix der Nearfly-Bibliothek

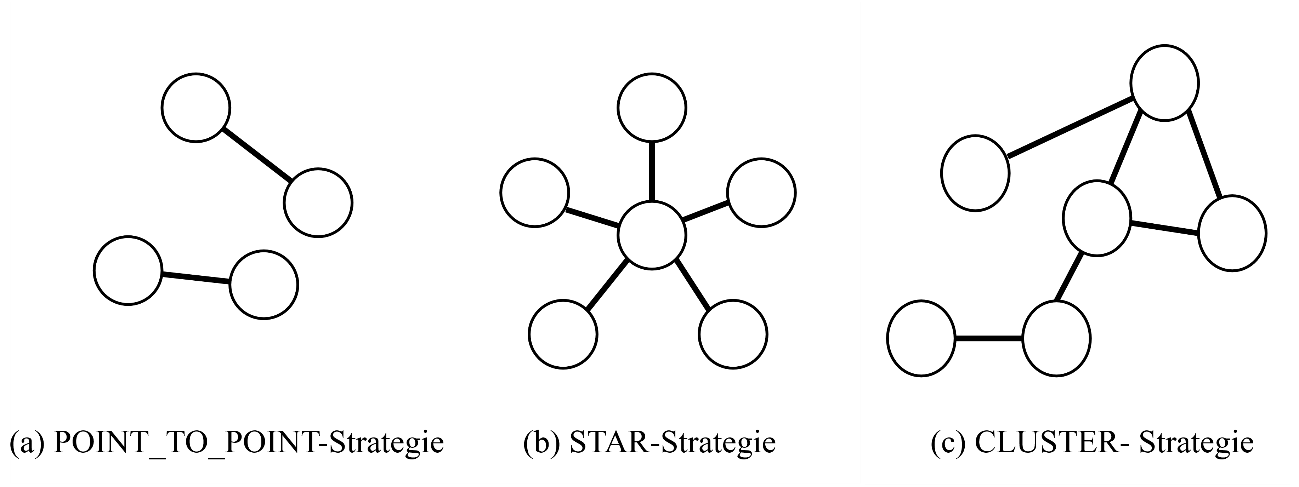
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Funktionale Anforderungen** | Messenger | Shared Touchpoint Canvas | Bouncing Ball | Score Board Notepad |
| Textbasierte Nachrichten senden | x | x | x | x |
| Multimedia-Nachrichten senden | x |  |  |  |
| Priorisierung der Nachrichtentypen | x |  |  |  |
| Trennbarkeit unterschiedlicher Nachrichten innerhalb derselben App | x |  |  |  |
| Nachrichten bei getrennter Verbindung halten | x |  |  |  |
| Automatischer Verbindungsaufbau | x |  |  |  |
| Nachrichten enthalten den Absender | x | x | x | x |
| Verzögerungsarmes Empfangen der Nachrichten |  | x | x |  |
| Nahe Echtzeit für ein flüssiges Multiplayer-Spiel |  | x | x |  |
| Sichtbarkeit der Nachrichten nur innerhalb derselben App | x | x | x | x |
| API der unterliegenden Technologien identisch | x | x | x | x |

Des Weiteren ist als Nichtfunktionale Anforderung vorgegeben, dass die zu entwickelnde Bibliothek auf allen Android Versionen ab dem API Level 24 (Android 7.0) laufen muss.

## Gegenüberstellung von Nearby Connections und MQTT

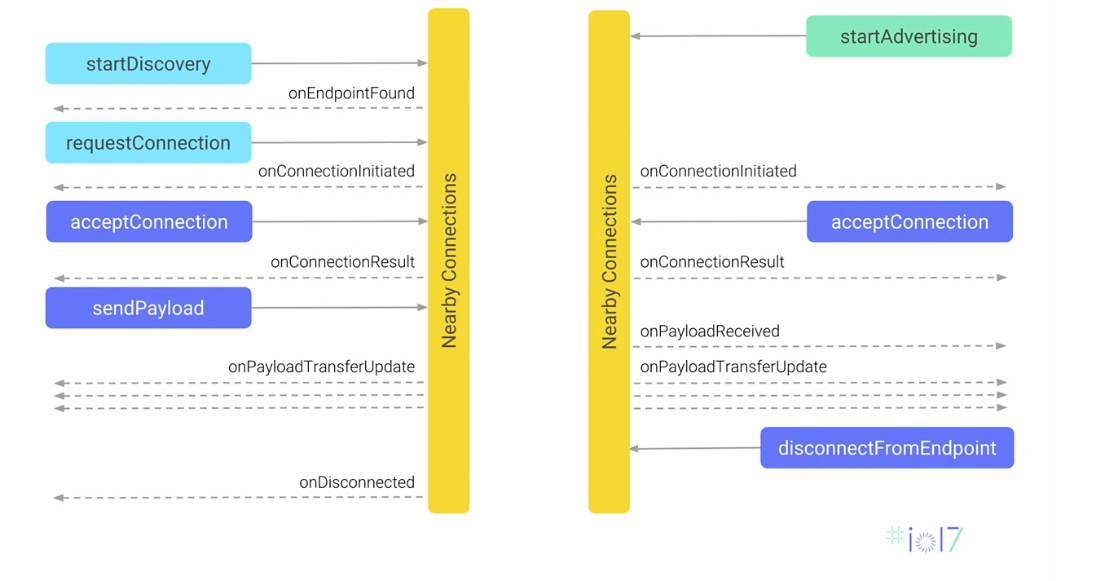
Da die API sowohl mit Nearby Connections als auch mit MQTT laufen soll, müssen beide Technologien bezüglich der Verwendung und das Verhalten gegenübergestellt werden. Nearby Connections [6] wird vom Nearby Connection Team als eine *Peer-To-Peer-API* mit hoher Bandbreite und kleiner Latenz beschrieben, welche durch eine optionale Verschlüsselung einen sicheren Datentransfer zwischen den verbundenen Knoten ermöglicht [6]. Verbindungen werden dabei durch Nutzung von Bluetooth, Bluetooth-Low-Energy (BLE) und automatisch angelegten WiFi Hotspot hergestellt und sorgen damit für eine beinahe vollständig automatische offline Vernetzung. Um dies zu ermöglichen, besitzt die API die Berechtigung sowohl WiFi wie auch Bluetooth einzuschalten und diese beim Beenden wieder in ihren Ausgangszustand zu versetzen.

Vor dem eigentlichen Verbindungsaufbau muss eine Netzwerktopologie gewählt werden, welche sich ebenfalls auf die maximale Übertragungsrate im Netzwerk auswirkt. Nearby Connections bietet drei verschiedene Topologien (genannt Strategien), wie in **Abbildung 2** gezeigt, an. Die Erste nennt sich CLUSTER-Strategie (**Abbildung 2**a) und erlaubt jedem Knoten im Netzwerk beliebige Verbindungen anzunehmen. Die damit entstandene N-M Topologie unterstützt jedoch nur Bluetooth und erlaubt nur eine geringe Bandbreite. Zudem ist zu beachten, dass die realistische maximale Anzahl gleichzeitig verbundener Geräte von Bluetooth auf 3-4 limitiert ist [4]. Durch die STAR-Strategie (**Abbildung 2**b) hingegen wird eine 1-N Topologie ermöglicht, welche zusätzlich zum Bluetooth ein WiFi-Hotspot verwenden kann und damit deutlich mehr Bandbreite ermöglicht. Sobald ein Netzwerk zustande gekommen ist, versucht die API das Netzwerk auf ein WiFi-Hotspot upzugraden [7]. Das Netzwerk kann dann bis zu 7 Knoten aufnehmen. Zuletzt existiert die POINT\_TO\_POINT-Strategie (**Abbildung 2**c), welche im Verhalten der STAR-Strategie entspricht, mit dem Unterschied das jedem Knoten nur noch ein Verbindungsknoten gestattet wird.



**Abbildung 2:** Verfügbare Netzwerktopologien (Strategien) der Nearby Conenctions API

Um das Erkennen der Geräte untereinander zu ermöglichen, verwendet die Nearby Connection API ein Advertising/ Discovery-Verfahren. Das *Advertising* ist ein passiver Zustand, bei dem ein Gerät gefunden werden kann, während das *Discovering* eine aktive Suche nach *Advertisern* initiiert, welche den Batteriekonsum deutlich steigert. Ein Gerät kann dabei sowohl einen, wie auch beide Zustände zeitglich einnehmen [6], was jedoch zu *trashing* führen kann[8]. Sendet ein Gerät im *Discovering*-Zustand eine Verbindungsanfrage, welche vom *Advertiser* bestätigt wird, wird eine symmetrische Verbindung (siehe **Abbildung 3**) zwischen beiden Knoten initialisiert. Eine Authentifizierung ist standardmäßig nicht realisiert, kann jedoch nachträglich vom API Nutzer z.B. als Popup-Anfrage, welche den Endpunkt-Namen des Gerätes nennt und die Verbindung nur nach erfolgreicher Bestätigung instanziiert, realisiert werden. Sind mindestens zwei Geräte verbunden können entsprechend **Abbildung 3**, durch Angabe eines Empfängers in Form eines *EndpointName*, Daten ausgetauscht werden. Der Datenaustausch verläuft dabei stehts verschlüsselt [6] und wird in 3 Nachrichtentypen unterschieden. Diese sind einerseits Byte-Nachrichten, welche auf 32 KByte limitiert sind und andererseits die zwei unlimitierten Datentypen File und Stream [6].



**Abbildung 3:** Nachrichtenverlauf der Nearby Connections API [9]

MQTT [10] hingegen wird in der OASIS Spezifikation [10] als ein leichtgewichtiges, leicht benutzbares Client Server *Pubish/Subscribe* Nachrichten-Protokoll beschrieben, welches wenig Bandbreite benötigt und einen kleinen *Code*-*Footprint* hinterlässt. Im Gegensatz zu Nearby Connections setzt MQTT in der Praxis eine TCP-Verbindung zum Server (Broker) voraus [11], welcher die Verbindungen und Nachrichtenübermittlung verwalteten und die Clients voneinander entkoppelt, sodass diese nie direkt miteinander verbunden sind. Die MQTT Spezifikation erlaubt auch andere Protokolle, welche eine geordnete, verlustfreie, bi-direktionale Verbindung zulassen [10].

Hat ein Client die Verbindung zum Broker hergestellt, kann dieser Nachrichten publishen, sowie Subskriptionen auf gewisse *Topics[[3]](#footnote-3)* (z.B. „Test/topic“) durchführen [11] Ein Topic meint dabei ein UTF-8 Encodierter String der durch einen Schrägstrich (‚/‘ U+002F) hierarchisch, gleichend einer Baumstruktur, untergliedert werden kann z.B. „Baum/Zweig“ inkludiert „Baum/Zweig/Blatt“[10]. Die Nachrichten müssen stehts ein *Topic*, wie auch ein Byte Payload unter 256 MByte [12] haben, sodass der Broker die Nachrichten an Subskribierte Clients weiterleiten kann.

Die MQTT Spezifikation sieht keine Verschlüsselung der Nachrichten vor, um das Protokoll möglichst einfach zu halten [10], jedoch wird dies in vielen Implementierungen, wie auch der zu verwendenden Paho-Client-Bibliothek unterstützt.

Zudem erhalten Nachrichten Angaben wie *Quality-Of-Service* (QoS) und ein *Retain-Flag*, welche dafür sorgt, dass der Server die letzte Nachricht speichert und diese fortan an allen *Subscriber* des zugehörigen Topics beim erfolgreichen Verbindungsaufbau, sendet [10].

**Tabelle 3:** Überblick über Unterschiede zwischen Nearby Connections und MQTT [6]

|  |  |  |
| --- | --- | --- |
| Technologie | MQTT | Nearby Connections |
| Netzwerktopologie | Server/Client | Cluster, Star oder Point-to-Point |
| Verbindung | Persistent zum Server | Persistent zu den einzelnen Peers |
| Verwendetes Protokoll | TCP/ IP | Bluetooth, BLE, WiFi Direct |
| Datenrate  (theoretisch) | WiFi: max. 250 Mbit/s  LTE: max. 300 Mbit/s | Bluetooth 3.0: max. 24 Mbit/s  WiFi Direct: max. 250 Mbit/s |
| Latenz | 114,30 ms | BLE: min. 46 ms  WiFi: 20 – 80 ms |
| Nachrichtenaustausch | Publish (1: N) | Bidirektional One-Way (1: 1) |
| Authentifikation | Namen und Passwort-Feld | Verbindungsanfrage an Nutzer  (Optional implementierbar) |
| Nachrichtenverschlüsselung | TLS möglich | durchgehend |
| Payload-typ | Byte | Byte, Stream oder File |
| Adressat | Topic-Subscriber | EndpointName |
| Max Packet-Größe | Maximal 256MB (Seitens des Brokers einstellbar) | Bytes: 32KB  Files&Stream: unbegrenzt |

Auch ist der Vergleich beider Technologien in Bezug auf die Datenrate sinnvoll, da Nearby Connections im Gegensatz zu MQTT, nicht ständig auf ein Internet-Protokoll zurückgreift, sondern als Hybrid-Technologie zunächst durch Bluetooth kommuniziert und erst bei günstiger Gelegenheit wechselt. Im Grunde schaffen alle Chips ab Bluetooth Version 3.0 durch Kombination von Bluetooth und WLAN-Übertragungstechnik einen zusätzlichen Highspeed-Kanal, welcher eine theoretische maximale Übertragungsgeschwindigkeit von 24Mbit/s, z.B. zum Übertragen von Fotos, Videos oder Musik zulässt [13]. Ab Version 4.0 ist in den Chips zusätzlich Bluetooth Low Energy (BLE) integriert, welcher deutlich weniger Strom konsumiert, jedoch nur maximal 220kBit/s und eine eingeschränkte Reichweite von 10 Metern besitzt [14]. Mit Version 4.2 ist die BLE-Datenrate im Vergleich zur Vorgängerversion, um Faktor 2.5 erhöht und die Übertragung sicherer, während mit Bluetooth 5.0 [15] je nach Anforderung einen verdoppelter Bruttodurchsatz von 2Mbit/s oder eine Reichweite von bis zu 200m durch BLE erzielen werden kann.

Die tatsächliche Datenrate liegt jedoch, wie Evan Fallis und Petros Spachos [16] durch das Testen von vier verschiedenen Smartphones mit Bluetooth 4.0 und 4.1 zeigt, mit 1 Mbit/s weit unter den theoretischen Angaben. Weiterhin zeigen die Messungen dieser beiden, das die WiFi-Übertragung via Smartphone mit einem durchschnittlichen Datenrate von 28.95 Mbit/s und Extrema von 25.7 Mbit/s, sowie 37.8 Mbit/s im clientseitigen Übertragen von TCP-Paketen [16], die theoretische Angabe von bis zu 250 Mbit/s weit verfehlt. Befindet sich ein Smartphone hingegen im LTE Netzt, schafft dieser durch Nutzung der 4G LTE-Advanced Technology durchschnittlich 42 Mbit/s [17] (bei einer theoretischen Angabe von 300 Mbit/s).

Während die theoretische Angabe demnach nur in kontrollierten Laborumfeld erzielbar sind, zeigen die tatsächlichen Datenraten, in welchem Bereich sich die Werte bei der Übertragung mithilfe der unterliegenden Technologien bewegen sollten. Dabei fällt der drastische Unterschied von Faktor 29 zwischen Bluetooth und Wifi auf. Wird eine Übertragung ohne aktiviertem Wifi-Hotspot übertragen sollte diese spürbar langsamer sein.

Bezüglich der Latenzen zeigt eine Latenzevaluation für MQTT und WebSocket Protokolle, dass MQTT eine durchschnittliche Latenz von 114,30 ms [18] bei einem Payload von 5 Bytes aufweist. Während einige Messergebnisse der Andorid App Network Latency Checker zeigen, dass sich die WiFi Latenz bei einem Samsung S9 im Bereich von 20 bis 80 ms bewegt. Die Latenz von Bluetooth hingegen misst Robert Triggs auf vier unterschiedlichen Smartphones auf 200 bis 500 ms [19], wobei diese teilweise stark abhängig vom Gerät ist. Durch Bluetooth Low Energy sind jedoch auch Latenzen von bis zu 46 ms [20] erzielbar.

# Software-Design

Die Nearfly-Bibliothek soll über die Nearfly-API ansprechbar sein, welche durch instanziieren des NearflyClients benutzet werden kann. Um das Reagieren auf bestimmte Events seitens der Nearfly-Bibliothek, wie etwa den Eingang von Nachrichten oder Statusänderungen (z.B. verbunden/ getrennt) zu reagieren, sollen beliebige Observer (genannt NearflyListener) beim NearflyClient an- und abgemeldet werden können.

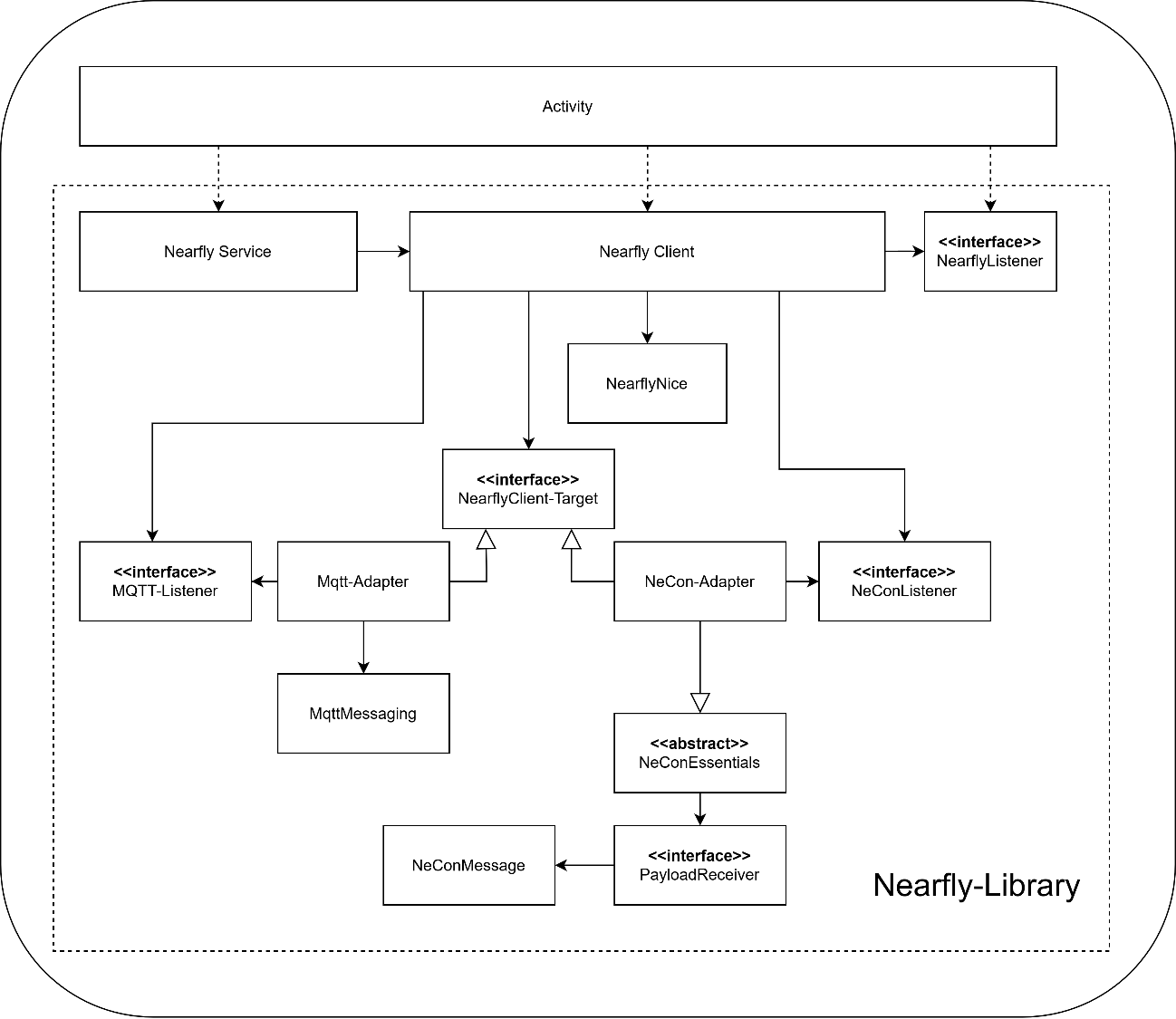
Der NearflyClient (wie in **Abbildung 4**) soll die unterliegenden NearbyConnections- und MQTT-API über dedizierte Adapter, welche die Anfragen per Delegation an die jeweiligen APIs weiterleiten und den NearflyClientTarget (wie in **Abbildung 4**) implementieren, ansprechen. Das NearflyClientTarget-Interface definiert die Methoden, wie das Senden von Nachrichten, welche der NearflyClient benötigt.

Die auf die Nearby Connections (NeCon) API aufbauenden Funktionalitäten sollen in zwei verschiedenen Komponenten, wie in **Abbildung 4** gezeigt, ausgelagert werden. Während die NeConEssentials-Komponente die Grundfunktionalitäten für eine korrekt funktionierende Nearby Connections API, wie etwa das Versetzen der Endpunkte in den *Advertising*-/*Discovering*-Zustand oder das Verwalten von entdeckten, akzeptierten sowie schwebenden (pending) Verbindungen, bietet. Soll die NeConAdapter-Komponente hauptsächlich die Funktionalitäten anbieten, welche speziell für die Nearfly-Bibliothek benötigt werden. Dazu gehört das autonome Aufbauen eines lokalen Peer-to-Peer Netzwerkes aus den nahliegenden Knoten, sowie die für die Datenübertragung notwendigen Funktionalitäten. Zwischen der NeConAdapter- und der NeConEssentials-Komponente soll dabei die Abhängigkeit einer Basis-/Kind-Klasse bestehen.

Um auf eintreffende Nachrichten zu reagieren, soll die NeConEssentials-Komponente weiterhin eine Abhängigkeit zur PayloadReceiver-Komponente, welche eine Erweiterung des von Nearby Connections API angebotenen primitiven PayloadCallback-Listerners ist, besitzen.

Auch sollen die Nutzung der MQTT API über zwei Komponenten verteilt werden. Während die MqttMessaging-Komponente jene ist, die das Funktionieren von MQTT sicherstellt, werden in der MqttAdapter-Komponente die für die Nearfly-Bibliothek benötigten Funktionalitäten, wie etwa das Senden von Multimedia umgesetzt.

Entsprechend der Kommunikation zwischen Nearfly-Bibliothek und Activity, sollen die Adapterklassen durch die Nutzung dedizierte Interface-Komponenten, dem NearflyClient den Eingang von Nachrichten oder Statusänderungen signalisieren.



**Abbildung 4:** System-Architektur der Nearfly API

Zudem soll die Nearfly-API über einen Android-Service (genannt NearflyService), welcher selbst als Wrapper-Klasse (wie in **Abbildung 4**) den NearflyClient innehält, benutzt werden können. Das Ansprechen der Nearfly API über den NearflyService, soll dabei jener Weg sein, welcher in dieser Thesis bevorzugt benutzt wird und hat den Vorteil, dass der Service nach dem einmaligen Starten solange, wie die eigentliche App im Hintergrund läuft. Wird innerhalb derselben Anwendung die *Activity* gewechselt, bleibt die Verbindung zum Netzwerk bestehen und die neue *Activity* kann ebenfalls die NearflyAPI nutzen. Will diese auf Events reagieren, kann diese ebenfalls einen NearflyListener anmelden.

# Verbindungsaufbau



Die Obligatorischen Berechtigungen zeigen, das MQTT nur eine Berechtigung für den Internetzugang benötigten, während Nearby Connections API insgesamt neun Berechtigung für Bluetooth-, Internet-, Standort und Datenzugriffe benötigt. Da Standort- und Datenzugriffs potenziell viel Schaden anrichten können, müssen diese Berechtigungen gesondert zur Laufzeit abgefragt werden.

Durch Erstellen einer Sichtbarkeitsbeschränkung ist sichergestellt, dass ein Netzwerkaufbau zwischen Geräten ungleicher Applikationen, welche die Nearfly API verwenden, stattfindet.

Die Wahl der Topologie zeigt das zwischen Overlay-Queue und Stern-Netzwerk, das Stern-Netzwerk gewählt wird, um die *Wifi-Direct*-Technologie zu verwenden und dadurch mehr Bandbreite zur Datenübertragung innerhalb des Netzwerkes zu haben.

Die Erstellung eines autonomen Peer-to-Peer Netzwerkes zeigt, dass der Netzwerkaufbauprozess zustandsbasiert ist und nach einer Root-Findungsphase das leistungsfähigste Gerät von allen Knoten als Root gesehen, zu welchen sich alle anderen Knoten verbinden.

## Obligatorische Berechtigungen



Apps brauchen in Android statische und dynamische Berechtigung für alle Operationen, welche andere Apps, das Betriebssystem oder den Benutzer nachteilig beeinflussen können [21]. Dazu gehört etwa das Auslesen von sämtlichen Dateninhalten außerhalb der eigenen Anwendungskontext, wie dies für Nearby Connections benötigt wird. Die Berechtigungen werden dabei in drei Risikogruppen (genannt *protection level*) unterteilt. *Normal-*Berechtigungen können in das Manifest eingetragen werden und werden daraufhin einmalig beim Installieren der App abgefragt. *Signature*-Berechtigungen hingegen müssen nicht explizit abgefragt werden, wenn diese bereits einer App gleicher Signatur gewährt wurden. Zuletzt existieren die *Dangerous*-Berechtigungen, wie etwa das Lesen von Benutzerkontakten [21]. Diese müssen ab Android 6.0 (API level 23) jeweils explizit von der Anwendung durch Implementieren einer Abfrage, zur Laufzeit abgefragt werden [21]. Während MQTT nur eine *Normal*-Berechtigung zum Öffnen von Internet Sockets (wie in **Tabelle 4**) braucht, erfordert Nearby Connections inkl. der Möglichkeit zum Senden von binären Daten insgesamt acht verschiedene Berechtigungen. Zu den *Normal*-Berechtigungen zählen zwei Berechtigungen zum *Discovern*, sowie Koppeln von Geräten via Bluetooth (wie in **Tabelle 4**) und zwei Berechtigungen zum Auslesen der WiFi-Informationen, sowie Ändern des Wi-Fi-Konnektivitätsstatus, welche für den Hotspot verwendet werden (wie in **Tabelle 4**). Die restlichen vier obligatorischen Berechtigungen gehören zum *Protection-Level* *Dangerous* und umfassen zwei Berechtigungen zum Abfragen vom ungefähren (FINE\_LOCATION in **Tabelle 4**), wie auch dem genauen Standort (COARSE\_LOCATION in **Tabelle 4**) des Gerätes und zwei Berechtigungen für das Lesen und Schreiben von Daten außerhalb des anwendungsspezifischen Verzeichnisses. Hierbei ist zu beachten, dass die Leseberechtigung automatisch beim Erteilen der Schreibberechtigung gewährt wird. Dies gilt ebenso für das Ersteilen der FINE\_LOCATION-Berechtigung, welche den Zugriff auf den Dienst, welcher durch COARSE\_LOCATION benutzt werden kann, inkludiert. Damit zeigt sich das die Neafly-Bibliothek insgesamt neun Berechtigungen benötigt, von denen vier zur Laufzeit abgefragt werden müssen, jedoch dem Endnutzer aufgrund der genannten Abhängigkeiten nur als zwei Abfragen angezeigt werden.

**Tabelle 4:** Obligatorische Berechtigungen zur Nutzung der Nearfly-Bibliothek

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | BLUETOOTH | BLUETOOTH\_ADMIN | ACCESS\_WIFI\_STATE | CHANGE\_WIFI\_STATE | ACCESS\_COARSE\_LOCATION | ACCESS\_FINE\_LOCATION | INTERNET | READ\_EXTERNAL\_STORAGE | WRITE\_EXTERNAL\_STORAGE | ACCESS\_MEDIA\_LOCATION | ACCESS\_COARSE\_LOCATION |  |
|  |
| Nearby Connections | X | X | X | X | X | X |  | (X) | (X) | X | X |  |
| MQTT |  |  |  |  |  |  | X |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| (X) optional | □ | Normal permission | | | |  | ■ | Dangerous permission | | | | |

## Sichtbarkeitsbeschränkung

Der Entwickler muss sich vor dem Verbindungsaufbau auf die zu benutzende Technologie (durch Setzen des ConnectionMode) festlegen. Hat sich der Entwickler zuvor für die Nutzung des NearflyServices entschlossen, darf die Methode zum initialisieren des Verbindungsaufbaus erst nach dem erfolgreichen Binding des Services aufgerufen werden, da die Service-Instanz erst mit dem *Binding* übergeben wird.

Entsprechend der Anforderung, einer Sichtbarkeitsbeschränkung der Nachrichten außerhalb derselben Anwendung, soll dabei vor dem Verbinden eine für die Anwendung eindeutige textbasierter Identifikator (genannt Room-String) festgelegt werden. Anwendungen, die denselben Room-String verwenden, können dabei miteinander kommunizieren. Technisch wird dies seitens MQTT über das Eintragen des Room-Strings als Top-Level Topic beim *subscriben* und *publishen* realisiert. Nearby Connections bietet hierfür eine sogenannte SERVICE\_ID an. Diese wird beim Starten vom *Advertising* oder *Discovering* angegeben und fortan zur Sichtbarkeitsbeschränkung von Knoten ungleicher SERVICE\_IDs verwendet. Nach Eintragung des Room-Strings kann nun die Verbindung aufgebaut werden.

## Wahl der Topologie

Durch die Server/Client Architektur genügt dabei seitens MQTT das Aufrufen der korrekt parametrisierten Connect-Methode, durch welchen eine indirekte Verbindung zu allen anderen Broker-Clients entsteht (direkt nur zum Server). Anders sieht es bei Nearby Connections aus, welche als Peer-to-Peer API das sukzessive Aufbauen eines Netzwerkes durch Anfragen der zu verbindenden Knoten fordert. Zur Bewältigung diese Probleme wurden zwei Möglichkeiten gefunden. Die erste Möglichkeit, welche bereits intern vom NearbyConnections Team entwickelt wurde und bis zu etwa 30 Geräte [22] verbinden kann, beinhaltet das Erstellen einer Overlay-Queue, in welcher Header und Tail durchgehend zeitgleich *advertisen* und *discovern [22]*. Ein nicht verbundenes Gerät wird als Queue der Länge eins gesehen wird, während Header und Tail sich nicht gegenseitig verbinden dürfen [22]. Jedoch führt dies dazu, dass zwangläufig die CLUSTER Topologie gewählt werden muss, die wie bereits erwähnt auf Bluetooth limitiert ist und damit zu einem vergleichsweise niedrigeren Datentransfer führt. Um *WiFi-Direct* nutzen zu können, muss also ein sternförmiges Netzwerk genutzt werden. Dies hat den Vorteil einer breitbandigeren, stärkere Datenübertragung, die besonders bei niedriger Teilnehmeranzahl (drei bis vier) gut funktioniert. Diese zweite Möglichkeit hat jedoch auch Schwächen. Zum einen wird der zentralen Knoten (Root-Knoten) stärker belastet und die Last nicht mehr verteilt. Zum andren bildet der Root-Knoten als verbindende Einheit aller Netzwerkteilnehmer, einen *Single point of failure*. Fällt der Root-Knoten weg, muss das Netzwerk wiederaufgebaut werden. Auch ist zu beachten, dass das Netzwerk, wie bereits bekannt, trotz WiFi-Hotspot durch Bluetooth auf sieben Teilnehmer beschränkt wird. Um Gebrauch vom der WiFi-Direct-Technologie zu machen, wird eine sternenförmige Netzwerk-Topologie verwendet.

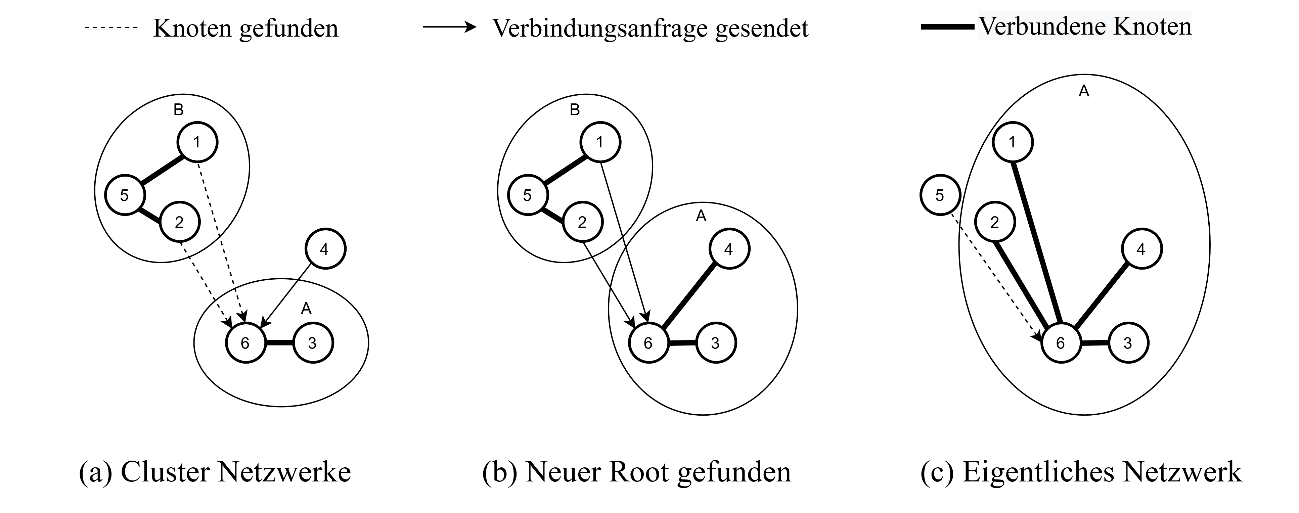
## Erstellung des autonomen Peer-to-Peer Netzwerks

Da jedem Gerät während des *Discovering*s, der Endpoint-Name aller entdeckter *Advertiser* angezeigt wird, muss dieser über den Endpoint-Namen entscheiden, welches Gerät als Root-Knoten dienen kann. Dazu soll der Endpoint-Name entsprechend der Formel 1 aus der maximalen CPU Frequenz in Megahertz () des Gerätes und der Differenz zwischen der absoluten Tageszeit und aktuellen Tageszeit zusammengestellt werden, wobei die Gesamtstunden eines Tages[[4]](#footnote-4) in Millisekunden und die Differenz der Millisekunden ausgehend von 0 Uhr (Mitternacht) meint. Dabei dient der Bewertung des jeweiligen Gerätes. Durch soll sichergestellt werden, dass leistungsfähigere Geräte, welche dem Netzwerk später beitreten, nichtmehr als Root dienen können. ist dabei eine Konstante, welche empirisch ermittelt wurde und festsetzt, wie verspätet ein leistungsfähigeres Gerät dem Netzwerk beitreten kann, ohne auf die Root-Position verzichten zu müssen.

|  |  |  |  |
| --- | --- | --- | --- |
|  | ( | 1 | ) |

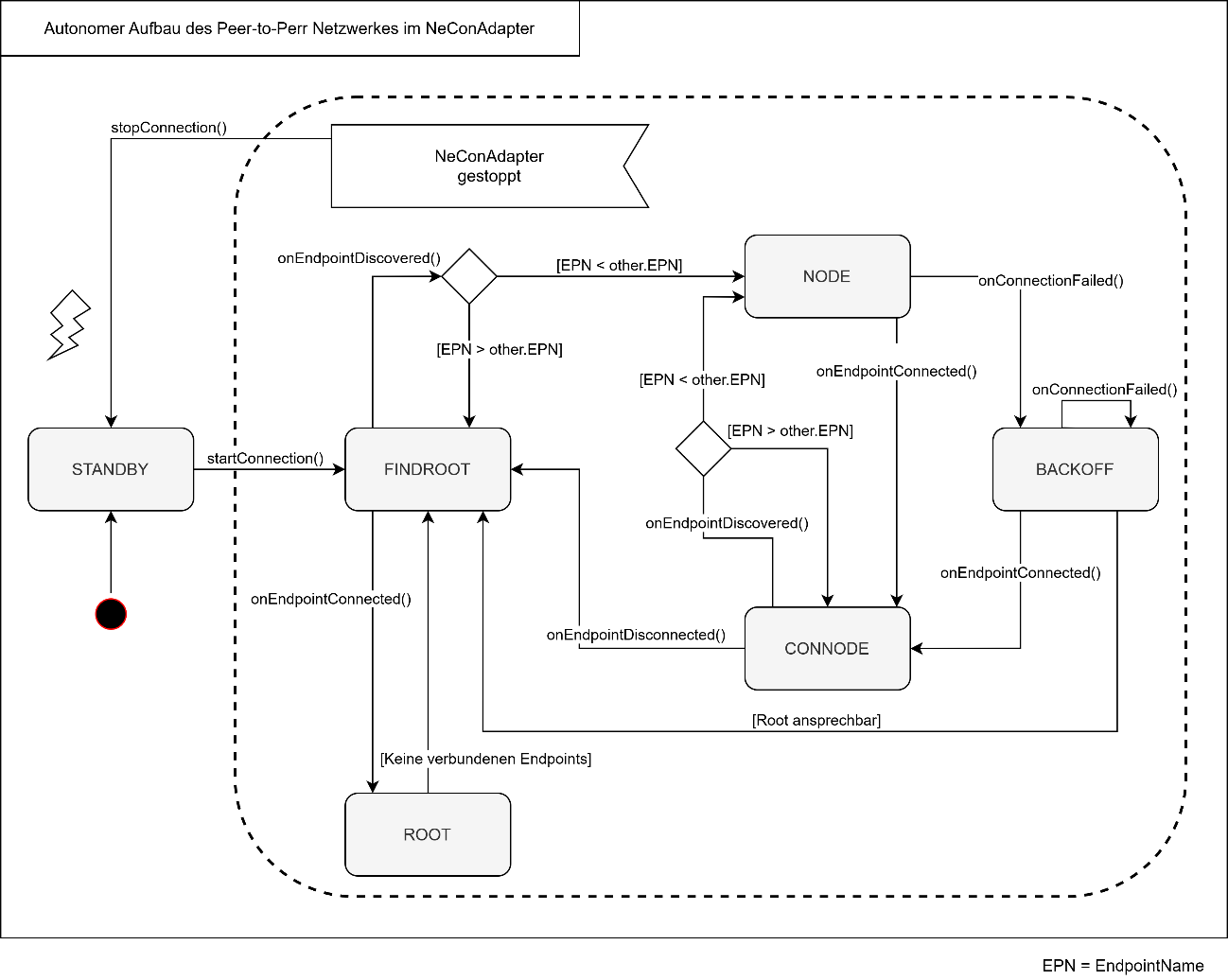
Der in **Abbildung 4** gezeigte NeConAdapter nutzt bei für den Verbindungsaufbau einen Zustandsautomat mit 6 Zustände (siehe **Abbildung 6**). Während der NeConAdapter nicht vom NearflyClient verwendet wird, soll dieser im STANDBY-Zustand verweilen, andernfalls wechselt dieser in den FINDROOT-Zustand. Während des FINDROOT-Zustandes wird sowohl das *Advertising,* wie auch das *Discovering* aktiviert. In dieser Phase prüft jeder Knoten beim Entdecken (onEndpointDiscovered) eines anderen Knotens durch Vergleichen der Endpoint-Namen, ob dieser kleiner ist. Bewahrheitet sich diese Bedingung wechselt dieser in den NODE-Zustand und deaktiviert zeitgleich das Advertising, sodass dieser für andere nicht mehr sichtbar ist.

Daraufhin wird vom NODE-Knoten eine Verbindungsanfrage an den Knoten mit dem größten entdeckten Endpoint-Namen gesendet. Um die Wahrscheinlichkeit von zeitgleichen Anfragen an den Root zu senken, war ursprünglich sowohl eine initiale Wartezeit zum Finden des Roots (INITIAL\_DISCOVERY\_TIME), wie auch eine zufällige Zeit (RANDRANGE\_COLAVOID) vorgesehen, innerhalb der ein NODE-Knoten vor dem Senden der Verbindungsanfrage warten muss.



**Abbildung 5:** Netzwerkaufbau bei Clusterbildungen

Nach einigen empirischen Beobachtungen wurde die INITIAL\_DISCOVERY\_TIME, wie auch die RANDRANGE\_COLAVOID jedoch auf 1 Millisekunden reduziert. Dies hat den Grund, das Festgestellt wurde, das der Verbindungsaufbau ohne jene Wartezeiten etwas schneller verläuft. Anders als vermutet, hat das Entfernen dieser Zeiten, keine negativen Auswirkungen auf die durchschnittlichen Zeiten für den Verbindungsaufbau im Mehrknoten Netzwerk. Durch das Aussetzen dieser beider Zeiten, können nämlich Geräte entweder sich direkt zum Root verbinden (wie Knoten 3 in **Abbildung 5**a) oder zunächst Cluster (wie Cluster B in **Abbildung 5**b ) bilden und nachträglich zum Root (**Abbildung 5**b und **Abbildung 5**c) verbinden. Während dieser Zeit können entsprechend Knoten 4 **Abbildung 5**a anderen Knoten eine Verbindung zum eigentlichen Root aufbauen. Die durchschnittliche Zeit wird dadurch etwas besser.



**Abbildung 6:** Zustandsdiagramm vom Verbindungsaufbau des NeConAdapters

Bemerkt ein Knoten A eine eingehende Verbindunganfrage (onConnectionInstantiated) von Knoten B und befindet sich dieser nicht im Verbindungsaufbau mit einem anderen Knoten C, akzeptiert dieser diesen. Nachdem die Verbindung erfolgreich aufgebaut wurde (onEndpointConnection), wechselt Knoten A in den ROOT-Zustand und deaktiviert das *Discovering*, da dieser (als Root-Knoten) nunmehr gefunden werden will. Zeitgleich wechselt Knoten B vom NODE-Zustand in den CONNODE-Zustand. Befindet sich Knoten A gerade hingegen im Verbindungsaufbau, wird die Verbindungsanfrage von Knoten B abgelehnt (onConnectionFailed). Der anfragende Knoten B (im NODE-Zustand) verfällt daraufhin in den BACKOFF-Zustand und wartet innerhalb eines gegebenen Zeitfensters eine zufällige Zeit (BACKOFF\_TIME), bis dieser daraufhin eine erneute Verbindungsanfrage sendet. Da der Verbindungsaufbau auch dann fehlschlagen kann, wenn der Root-Knoten aufgrund eines Netzwerkaustritts (*peer churn*) nicht mehr erreichbar ist, wechselt Knoten B nach MAX\_FAILATTEMPTS gescheiterten Anfragen in den FINDROOT-Zustand und löscht seinen Zwischenspeicher (u.a. *pending Endpoints*). Haben sich alle Knoten zum Root-Netzwerk verbunden, besteht das Netzwerk.

Will nun ein neues Gerät am bestehenden Netzwerk teilnehmen, sieht dieser nur noch den Root-Knoten, welcher als einziger das *Advertising* aktiviert hat und versucht sich wie o.g. zu diesem zu Verbinden. Will ein Knoten dem Netzwerk beitreten und hat aufgrund der eingestellten einen größeren Endpoint-Namen als der aktuelle Root, verbinden sich alle Netzwerkteilnehmen nach und nach, gefolgt vom alten Root, zu diesen. Je nach Leistungsunterschied zwischen aktuellen Root und neuem Root, beträgt diese Zeit aktuell ca. 10 bis 40 Sekunden. Nach Ablauf von 60 Sekunden deaktivieren alle Knoten im CONNODE-Zustand das *Discovering*, um den Akku zu schonen. Trennt der NearflyClient die Verbindung zum NeConAdapter, kehrt der Knoten zum STANDBY-Zustand. Dabei wird das *Advertising* und *Discovering* deaktiviert, sowie bestehende Verbindungen getrennt und aufgeräumt.

# Übertragung von Daten



Die Datenübertragung des NearflyClients wird durch einen Kanal, eine Priorität, sowie eine Flag zum Puffern der Nachrichten parametrisiert. Daten unter 30 KByte können durch die pubIt-Methode versandt werden, während größere Daten durch die pubFile-Methode versandt werden müssen.

Die Nearfly API ruft entsprechend dem aktuellen ConnectionMode die jeweilige Adapter-Komponente auf, welche eine fragmentierte Datenübertragung, durch Nutzung des NeCon-Nachrichtenprotokoll mit unterschiedlichen Feldern ausführt. Die Felder beinhalten vor allem Dateierweiterung und Identifikator und dienen zur empfängerseitigen Zusammensetzung der Datei im Standard-Download-Verzeichnis.

Der NeConAdapter sendet limitiert durch Nearby Connections die reine Datei und die Informationen für das Zusammensetzten der Datei getrennt, während der MqttAdapter Datei und zugehörige Informationen als eine Datei sendet.

## Senden mit Nearfly

Um der Anforderung, welche die Trennbarkeit unterschiedlicher Nachrichten innerhalb derselben Anwendung fordert nachzukommen, muss vor dem ersten Nachrichtenaustausuch mindestens ein Kanal (*Channel*) abonniert werden. Das Prinzip gleicht dabei dem der MQTT Topics. Wird ein Kanal abonniert, kann fortan nach Übergabe eines NearflyListeners auf alle Nachrichten, welche denselben Kanal adressieren reagiert werden. Umgekehrt werden nicht abonnierte oder deabonnierte Kanäle ignoriert. Der NearflyListener unterscheidet hierbei drei Nachrichtentypen. Zum einen können Status-Events (onLogMessage) der Adapter-Komponenten, wie die Zustandsänderungen seitens des NeCon-Adapters oder der erfolgreiche Verbindungsaufbau unterschieden werden. Zum anderen kann auf Multimedia-Nachrichten (onFile) und Binary-Nachrichten (onMessage) reagiert werden.

Besteht eine Verbindung zum Netzwerk, können Daten mithilfe eines der zwei vom NearflyClient angebotenen Methoden (pubIt und pubFile) übertragen werden. Während kleinere Byte-Daten bis zu 30 KBytes (MAX\_PUBIT\_MESSAGE\_SIZE) durch pubIt versandt werden können, müssen größere multimediale binäre Daten mithilfe der pubFile-Methode übertragen werden. Dabei entsteht die Grenze aus der bereits aus dem Abschnitt 2.3 bekannten Limitierung für Byte-Nachrichten der Nearby Connections API von 32 KBytes und einer großzügigen Toleranz, um größere zukünftige Erweiterungen des NeCon-Protokolls (welcher im folgenden Abschnitt 5.2 besprochen wird) zu kompensieren. Zusätzlich zur zwingend erforderlichen Angabe des Empfangskanals, können die Methoden mit einem Prioritätslevel von -20 (niedrigste Priorität) bis 19 (höchstmögliche Priorisierung), in Anlehnung an die nice-Level im Unix-System [23], sowie ein Retain-Flag parametrisiert werden.

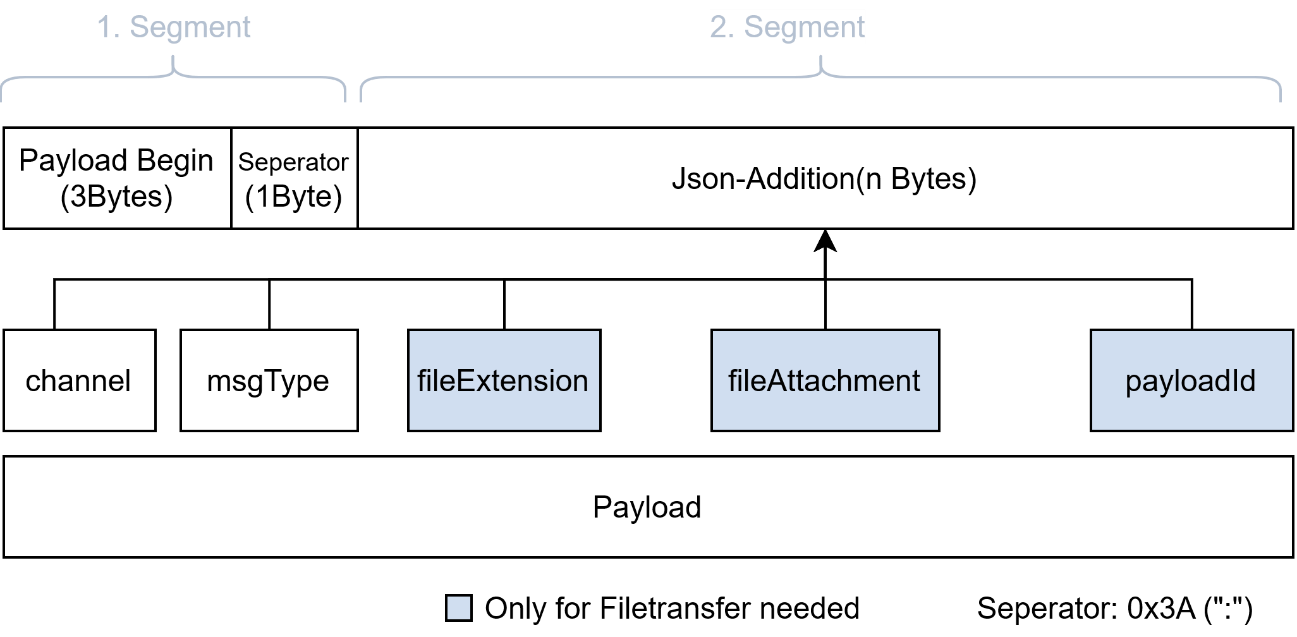
Ein gesetztes Retain-Flag meint dabei, dass Nachrichten selbst im verbindungslosen Zustand „gesendet“ werden können. Dazu gelangen diese zunächst in einen prioritätenbasierten Sendepuffer und werden erst dann gesendet, wenn eine Verbindung zur unterliegenden Technologie besteht und die Verbindung derzeitig nicht gewechselt wird. Ist das Retain-Flag nicht gesetzt, wird der Sendevorgang im verbindungslosen Zustand abgebrochen und dies dem Benutzer per Rückgabewert mitgeteilt.

Da MQTT das Einhalten der Reihenfolge der Nachrichten für Nachrichten gleicher QoS garantiert [10] und die NearbyConnection API dieses Garant für gleicher Payload-Typen (Bytes und Files) gibt [6], reicht die Realisierung einer sendeseitigen Priorisierung.

## Realisierung mit NearbyConnections

Während Bytes-Nachrichten von der Nearby Connections API ad-hoc gesendet und empfangen werden, ist dies bei File-Nachrichten anders. Diese werden automatisch zerlegt (*chunked*) und stückweise übertragen. Die Nearby Connection API stellt dabei die empfangene Datei stückweise in einem selbstangelegten „Nearby“-Ordner im Standard-Downloadverzeichnis her. Dies erklärt die notwendige Dangerous-Berechtigung für das Lesen und Schreiben von Daten außerhalb des Anwendungsspezifischen Verzeichnisses. Denn obwohl die Nearby Connections API durch Nutzung der Google Play Services[[5]](#footnote-5) selbst schon über diese Berechtigungen zum Bearbeiten von externen Daten verfügt [24], braucht die *Activity*, welche die Empfangene Datei letzten Endes öffnen, umbenennen oder löschen will Zugang zu diesen Verzeichnis.

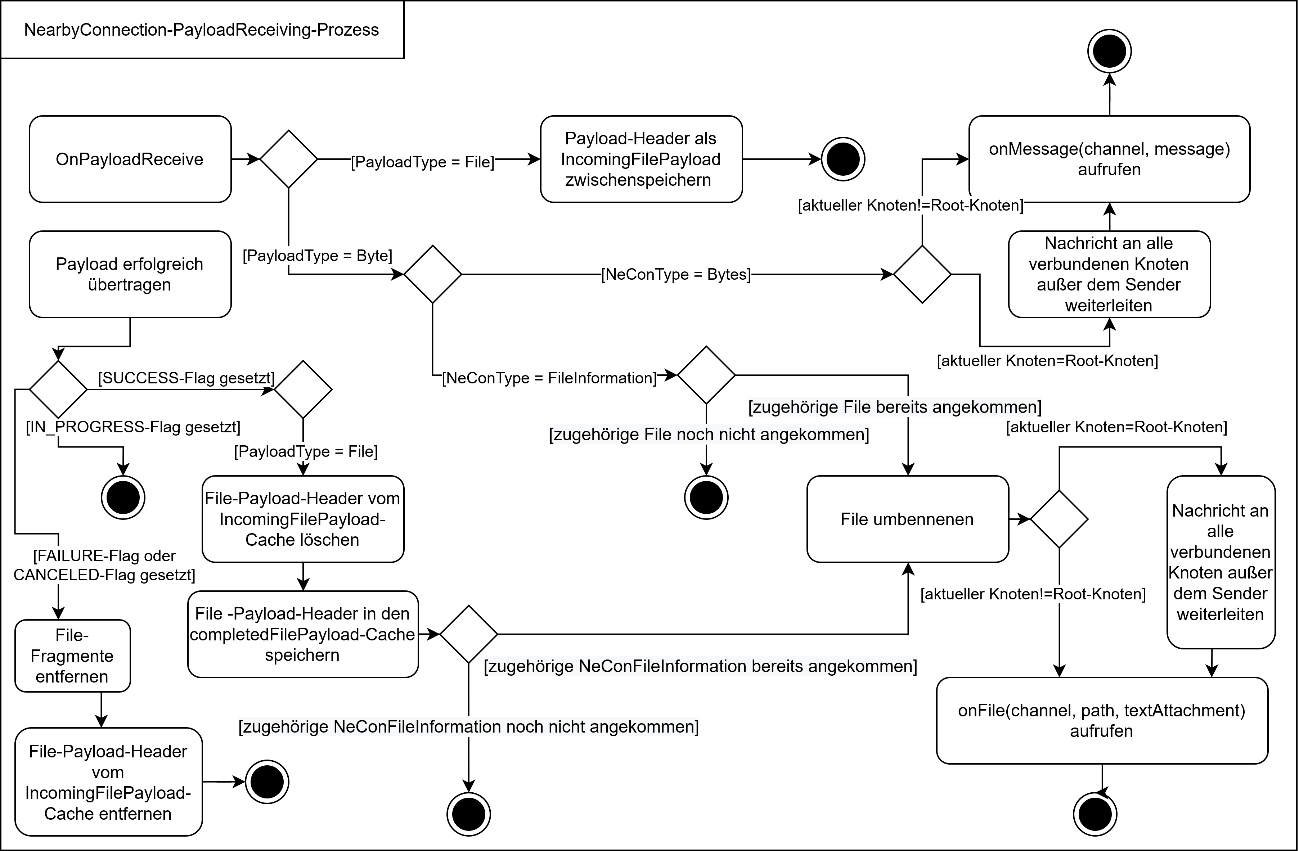
Da die gespeicherten Daten allerdings entsprechend ihres Identifikators benannt (z.B. -7808006164344755168) sind, der Endnutzer diese Datei aufgrund der fehlenden Dateiendung nicht öffnen kann und das Wiederherstellen der Dateiendung anhand des MIME-Typs erfahrungsgemäß nicht immer mit Erfolg gesegnet ist, muss beim Versenden einer File-Nachricht ebenso eine Nachricht mit den Meta-Daten gesendet werden. Diese beinhaltet die Dateiendung und den File-Identifikator, welcher zur späteren Zuordnung der Datei dient und soll im Folgenden als NeConFileInformation bezeichnet werden. Zusätzlich beinhaltet diese einen Textanhang (textAttachment), welcher dem späteren Entwickler die Möglichkeit zum Anhängen eines zusätzlichen Strings (z.B. für weitere Meta-Daten) offenhält.



**Abbildung 7:** NeCon-Nachrichtenprotokoll

Ruft der NearflyClient nun die pubFile-Methode auf, sendet der NeConAdapter die Datei und zusätzlich die NeConFileinformation, während das Aufrufen der pubIt-Methode eine NeConBytesMessage sendet. Dabei wird für beide NeCon-Nachrichten das in **Abbildung 7** gezeigte NeCon-Nachrichtenpotokoll verwendet. Um eine dynamische Headerlänge zu ermöglichen, wird der Header in zwei Segmente unterteilt. Das erste Segment hat eine fixe Bytelänge und wird aus einem UFT-8 encodierten String gewonnen. Es beinhaltet das Feld für den Beginn der Bytes-Nutzlast, welche zunächst auf 3 Bytes begrenzt wird und damit eine maximale Headerlänge von 999 Bytes zulässt. Jedes String-Element braucht dabei genau ein Byte, da dies für UTF-8 enkodierte Zeichen, welche in der ASCII-Tabelle zu finden sind, garantiert wird. Andere Zeichen hingegen können bis zu 4Bytes beanspruchen. Die Größe des zweiten Segments ist dynamisch und beinhaltet ein JSON-Objekt im UTF-8-Stringformat, welcher bei NeConBytes-Nachrichten derzeitig nur den Channel erhält. Im Falle von NeConFileinformation, beinhaltet dieses zusätzlich noch die Dateierweiterung, einen Nutzdaten-Identifikator und den Textanhang, welcher der Datei angehängt werden kann.

Empfängt die NearbyConnection API (entsprechend **Abbildung 8**) einen Payload-Header, prüft der PayloadReceiver, ob diese vom Payload-Typ File oder Bytes ist. Handelt es sich hierbei um eine Bytes-Nachricht (nicht zu verwechseln mit der ähnlich benannten NeConByte-Nachricht), wird der NeCon-Nachrichtentyp identifiziert. Die Nachricht wird dem NeCon-Client (onMessage) übergeben. Dieser prüft, ob der entsprechende Kanal der Nachricht abonniert ist und leitet diese bei Erfolg an den NearflyService weiter. Ist dies nicht der Fall, wird die Nachricht verworfen. Ist der aktuelle Knoten als Root-Knoten deklariert, wird die Nachricht zusätzlich an alle verbundenen Knoten, außer dem Sender der Nachricht, gesendet. Anhand des NearflyListeners kann schlussendlich auf die Nachricht reagiert werden. Eine NeConFileInformation-Nachricht hingegen wird zunächst zwischengespeichert, zeitgleich wird kontrolliert, ob sich der Payload-Header der File-Nachricht mit demselben Identifikator, bereits angekommen ist und sich im CompletedFilePayload-Zwischerspeicher befindet. Ist dies nicht der Fall wird passiv auf den Eingang der Datei gewartet.



**Abbildung 8:** Prozessablauf beim Empfangen einer Nachricht seitens des NeCon-Clients

Kommt nun der Payload-Header einer File-Nachricht an (onPayloadReceive), signalisiert dies den Anfang der Übertragung einer File-Nachricht. Daraufhin wird der Payload-Header (welcher den Identifikator beinhaltet) als IncomingFilePayload zwischengespeichert, um die laufende Datenübertragung zu vermerken. Während der Übertragung wird bei jedem übertragenen Datei-Fragment ein Payload-Header (OnPayloadTransferUpdate) mit gesetztem IN\_PROGRESS-Flag empfangen, durch welchen sich die Anzahl der transferierten Bytes einsehen lassen. Tritt dabei ein Fehler auf, wird dies durch ein gesetztes FAILURE- oder CANCELED-Flag signalisiert. Der initial zwischengespeicherte Payload-Header, samt NeConFileInformation und die fragmentierte-Datei im Download-Verzeichnis müssen daraufhin gelöscht werden. Wurde die Datei vollständig übertragen, enthält der Payoad-Header ein gesetzten SUCCESS-Flag. Der vermerkte Header wird nun aus dem IncomingFilePayload-Zwischenspeicher entfernt und in den CompletedFilePayload-Zwischenspeicher geschoben. Befindet sich nun auch die NeConFileInformation-Nachricht im Zwischenspeicher, wird der Prozess zum Umbenennen der Datei initialisiert.

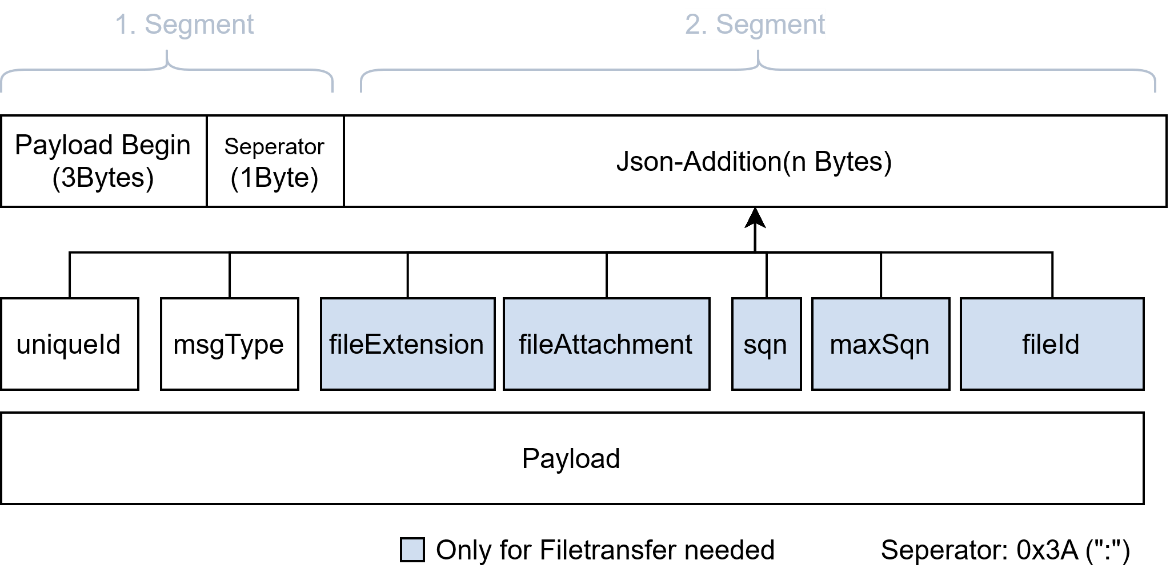
Fungiert der aktuelle Knoten als Root-Knoten, wird die empfangene Datei zusätzlich an alle verbundenen Knoten weitergeleitet. Ist der Empfangsvorgang erfolgreich abgeschlossen werden Payload-Header, wie auch die NeConFileInformation aus deren Zwischenspeicher gelöscht und die onFile-Methode mit dem Kanal, dem Dateipfad und dem textAttachment aufgerufen.

Aufgrund des beschränkten Einflusses, welcher dem Nearby Connection API Nutzer auf den Prozess der Dateiübertragung gegeben wird, kann dabei der Datenübertragungsprozess seitens des Empfängers nicht abgebrochen werden, wenn der jeweilige Kanal nicht abonniert ist.

## Realisierung mit MQTT

Da der Nearfly-Kanal auf dem MQTT-Topic basiert, kann dieser als Topic (room/kanal) dem Room-String angehängt werden. Um das Übertragung von Multimedia-Daten Mithilfe der MQTT API möglichst der Nearby Connections API gleichzustellen, sollen versandte Daten nach Ankunft ebenfalls im Standarddownload-Verzeichnis im „Nearby“-Ordner abgelegt werden. Dazu muss auch hier zuzüglich der zu übermittelnden Datei die Dateiendung, ein Beitext und der Nachrichtentyp mitversandt werden.

Eine unkomplizierte und daher oft gewählte Möglichkeit, um dieses Problem zu lösen, ist das Senden der gesamten Nachricht als String im JSON-Format. Die binäre Multimedia-Datei wird dabei durch das Base64-Kodierungsverfahren in einen String kodiert und beim empfangen erneut dekodiert. Allerdingt sorgt die Base64-Kodierung durch das Kodieren von drei Byte-Oktetts in vier Zeichen-Oktetts und das anschließende Padding der Daten auf ein vielfachen von vier dafür, das für alle Binäre-Daten im kodierten Zustand das 1.37 Fache [25] der ursprünglichen Datengröße benötigt wird. Um diesen Overhead zu vermeiden, soll daher eine andere Möglichkeit gewählt werden. Diese ergibt sich durch den Einsatz des NeCon-Nachrichtenprotokolls (**Abbildung 7**), das durch den dynamischen String Header, welcher durch das JSON-Segment leicht modifiziert werden kann, String von Payload-Bytes trennt.



**Abbildung 9:** Modifizierter NeCon-Nachrichtenprotokoll zur Nachrichtenübertragung seitens MQTT

Da die Nearby Connections API keine maximale Datengröße für Multimedia-Daten angibt und das MQTT-Protokoll eine maximale Payloadgröße von 256 MByte [12] vorsieht, wird der MQTT-Adapter um ein Chunking-Mechanismus[[6]](#footnote-6), das Pakete ab 2 Mbyte (MAC\_CHUNK\_SIZE) zerlegt, erweitert. Der Schwellwert ist dabei so groß gewählt, da ein zu kleiner Schwellwert z.B. 30KByte bei einem 5 MByte großen Bild, den Broker zunächst mit 167 Chunks fluten würde und damit MQTT stark verlangsamen würde. Um die Daten zu senden, wird das JSON-Segment, um eine Sequenznummer (sqn), die maximal zu sendenden Fragmenten (maxSqn) und einen Identifikator (fileId) ergänzt. Die fileId dient der empfängerseitigen Unterscheidung unterschiedlichen Multimedia-Fragmente und ist bei jedem Fragment dergleichen Datei identisch. Die Sequenznummer wird verwendet, um doppelte empfangene Datei-Fragmente, welche durch die eingestellte QoS von eins auftauchen können, erkennen und ignoriert zu können. Damit ergibt sich das in **Abbildung 9** gezeigte JSON-Segment. Soll nun eine Multimedia-Datei größer der MAC\_CHUNK\_SIZE gesendet werden, erzeugt der Sender eine zufällige fünfstellige fileId und sendet die Datei-Fragmente, welche daraufhin vom Empfänger zwischengespeichert werden. Empfängt der Empfänger das Datei-Fragment mit der Sequenznummer (maxSqn-1), wird die Datei rekonstruiert, und die zwischengespeicherten Datei-Fragmente gelöscht. Daraufhin wird die onFile-Methode des NearflyServices getriggert.

Um nun auch die die Bytes-Übertragung zu ermöglichen, erhält das JSON-Segment zusätzlich den Nachrichtentyp (msgType) und eine uniqueId (wie in **Abbildung 9** gezeigt), welche verhindern soll, dass der Sender seine eigenen Nachrichten bekommt und zunächst für das Angleichen der Verhaltensweisen in Hinsicht auf das Empfangen beider ConnectionModes verwendet wird.

# Evaluation

Die Tests zum Messen der Zeiten für den Verbindungsaufbau zeigen, dass der Verbindungsaufbau mithilfe des NeConAdapters zwischen zwei Knoten mit durchschnittlichen 7.5 bis 11 Sekunden und durchschnittlichen 60 Sekunden für das Aufbauen eines Netzwerkes mit sechs Knoten deutlich schlechter als der MqttAdapter skaliert, welches für den Netzwerkaufbau von fünf Netzwerkteilnehmern durchschnittlich nur 454 Millisekunden benötigt.

Die gemessenen Zeiten für die Datenübertragung zeigen hingegen, das zwar die Latenz in beiden Adapter-Klassen mit etwa 150 Millisekunden gleich ist, jedoch der NeConAdapter bei aktiven Wifi-Hotspot Daten etwa drei Mal schneller übertragen kann, während sich Datenübertragung derselben Dateien bei deaktiviertem Wifi-Hotspot etwa um Faktor 27 verlangsamt.

In Beispiel-Apps mit Nearfly werden die Anforderungen mithilfe der implementierten Szenarien verifziert. Es zeigt sich das alle Anforderungen erfüllt wurden, wobei die für die Bouncingball App verlangte nahe Echtzeitfähigkeit nur bei Nutzung des NeConAdapters im Netzwerk mit zwei Knoten erfüllt ist. Weiterhin wird durch den Shared Touchpoint Canvas gezeigt, dass der MqttAdapter auch in Bezug zur Datenübertragung besser als der NeconAdapter skaliert und das sich beide Adapter im Netzwerk mit drei Knoten in etwa gleich verhalten. Anwendungen mit wenig Datenverkehr, wie die Score Board Notepad zeigen sich als gut geeignet für beide Adapter, wenn Vorkehrungen für mögliche zwischenzeitliche Verbindungabstürzte getroffen werden.

Abschließend wird in Abschnitt 6.5 die konkrete Nutzung der Nearfly API durch das sequenzielle Aufrufen der vom NearflyClient angebotenen Methoden gezeigt.

## Verwendete Testgeräte

Für die Datenerhebung werden entsprechend **Tabelle 5** das Motorola G5, G6, sowie das Samsung Galaxy S7 edge, das S9 und das LG G2 mini LTE verwendet. Alle Smartphones unterstützen dieselben WiFi Protokolle, unterscheiden sich jedoch sowohl in den Android-Versionen wie auch teilweise in den verbauten Bluetooth Chips, welche wesentlich für den Verbindungsaufbau im NeCon-ConnectionMode sind. Zu den in **Tabelle 5** genannten Smartphones, musste ein Samsung Galaxy s5 mini, welcher ebenfalls zur Verfügung stand und über Bluetooth 4.0 verfügte, als Testobjekt ausgeschlossen werden, weil dieser des Öfteren Verbindungen abgelehnte und dies zu sehr langen Verbindungszeiten führte.

**Tabelle 5:** Verwendete Smartphones für die Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Smartphone** | **Android Version** | **WiFi 802.11** | **Bluetooth** |
| Motorola Moto G6 | 8.0 | a/b/g/n | 4.2 |
| Motorola Moto G5 | 8.1 | a/b/g/n | 4.2 |
| Samsung Galaxy S9 | 10 | a/b/g/n | 5.0 |
| Samsung Galaxy S7 edge | 7.0 | a/b/g/n | 4.2 |
| LG G2 mini LTE | 7.1.2 | a/b/g/n | 4.0 |
| Samsung S5 mini | 7.1.1 | a/b/g/n | 4.0 |

## Zeiten für den Verbindungsaufbau

Zunächst werden die Zeiten gemessen, welche für den Aufbau eines zwei Knoten Netzwerkes benötigt werden. Dies hat den Vorteil das ein besserer Einblick in die benötigten Zeiten eines einzelnen Knotens gewonnen werden kann. Der Test zum Verbindungsaufbau verläuft automatisch, wobei die Zeit, welche für das Erreichen der Zustände (NODE, CONNODE) benötigt werden vom System protokolliert werden. Wurde eine Verbindung vom letzten Knoten erfolgreich aufgestellt, wechselt dieser nach fünf Sekunden in den STANDBY-Zustand und trennt damit die Verbindung. Durch das Versetzten in den STANDBY-Zustand ist sichergestellt, dass alle zuvor gefundenen Knoten aus dem Cache gelöscht werden und die initiale Root-Findungsphase erneut ausgedauert werden muss. Weitere fünf Sekunden später, welche als Toleranzzeit zum Erkennen der getrennten Verbindung dienen, kehrt der Nicht-Root-Knoten in den FINDROOT-Zustand zurück, um erneut am Netzwerkaufbau zu partizipieren. **Abbildung 10** bis **Abbildung 13** stellen die für den Aufbau dieses kleinsten Netzwerkes benötigte Zeit in 50 Versuchen mit verschiedene Gerätekombinationen dar.

**Abbildung 10:** Benötigte Zeiten für den Verbindungsaufbau zwischen G5 und S9

In **Abbildung 10** findet das G5 den S9 nach durchschnittlichen 4445 Millisekunden, während eine Verbindung letzten Endes nach durchschnittlichen 7548 Millisekunden zu Stande kommt. Die Verbindung kommt minimal nach 2295 Millisekunden zu Stande und braucht maximal 20533 Millisekunden.

**Abbildung 11:** Benötigte Zeiten für den Verbindungsaufbau zwischen s7 edge und S9

Während Werte in **Abbildung 10** weniger schwanken, ist der Verbindungsaufbau zwischen S7 edge und S9 (wie in **Abbildung 11** zu sehen) stärkeren Fluktuationen aufgesetzt, wobei jedoch auch hier wieder der S9 nach durchschnittlichen 3845 Sekunden gefunden wird und eine Verbindung nach durchschnittlichen 7934 Millisekunden zu Stande kommt.

**Abbildung 12:** Benötigte Zeiten für den Verbindungsaufbau zwischen G6 und S9

Zwischen dem Verbindungsaufbau zwischen G6 und S9 (wie in **Abbildung 12**) sind, die durchschnittlichen Zeiten entsprechend den vorherigen Abbildungen ähnlich. Während ein Knoten nach durchschnittlichen 4183 Millisekunden gefunden wird, braucht es durchschnittliche 7376 Millisekunden bis eine Verbindung zu Stande kommt.

**Abbildung 13:** Benötigte Zeiten für den Verbindungsaufbau zwischen LG und S9

Obwohl der LG (wie in **Abbildung 13**) mitunter den Bluetooth Chip mit der niedrigsten Version besitzt, schafft es dieser das S9 nach durchschnittlichen 1336 Millisekunden zu finden und ist damit hinsichtlich des Discovery-Zeiten etwa drei Mal schneller als die vorher getesteten Kombinationen. Beim Verbindungsaufbau hingegen, liegt auch hier wieder die durchschnittliche Verbindungszeit bei 7824 Millisekunden und damit im Bereich der vorherigen Geräte-Kombinationen.

**Abbildung 14:** Benötigte Zeiten für den Verbindungsaufbau zwischen G5 und G6

Die Kombination zwischen G5 und G6, welche beide mit Bluetooth Chips der Version 4.2 ausgestattet sind, braucht mit 3848 Millisekunden etwa dieselbe Zeit, wie die vorherigen Kombinationen. Der Verbindungsaubau dauert jedoch verglichen mit den vorherigen Gerätekombinationen etwa 1.3 Mal länger. Auch zeigt **Abbildung 14** hohe Spitzenwerte im Bereich von 35 Millisekunden.

Als Ergebnis lässt sich festhalten, dass ein Verbindungsaufbau durchschnittlich in etwa 7.5 bis 11 Sekunden benötigt, wobei nicht ausgeschlossen werden kann, dass ein Verbindungsaufbau bis zu 35 Millisekunden braucht. Im besten Fall können Verbindungen jedoch auch in 3 Sekunden aufgebaut werden.

Der Verbindungsaufbau im zwei Knoten Netzwerk erscheint zunächst langsam. Doch gibt der Nearby Connections API Mitentwickler Will Harmon für den Unidirektionalen Verbindungsaufbau (d.h. im Falle eines einzigen *Advertisers* und eines einzigen *Discoverers*) bereits eine durchschnittliche Verbindungslatenz von 2-7 Sekunden [8] an, wobei die Verbindungslatenz bei Geräte, welche sich in einen schlechteren Zustand befinden auf bis zu einer Minute [26] ansteigen kann.

Da der Advertiser (Root) im NeConAdapter jedoch von vornerein nicht festgelegt ist, muss die Root Findungsphase, in welcher sowohl das *Advertising* wie auch das *Discovering* aktiviert wird, durchlaufen werden. Dadurch entsteht *Trashing (link)*. Wichtig ist diese Phase, weil Smartphones, wie das LG G2 aufgrund von zwischenzeitlichen Trennungen der Verbindung und mangelnder Leistung für das *Forwarding* von Nachrichten, wenn möglich nicht als Root gewählt werden sollten. Auch beim Samsung S5 Mini, welcher nicht in die o.g. Testfälle aufgenommen wurde, konnte festgestellt werden, dass dieser als Root kein Netzwerk größer als zwei Knoten aufbauen kann und weitere Knoten nicht akzeptiert.

Mit dem Wissen, das bereits zwischen zwei unterschiedlichen Smartphone-Kombinationen unterschiedliche Latenzen entstehen, können nun die Zeiten für den Aufbau eines drei Knoten-Netzwerkes (wie in **Abbildung 15**) evaluiert werden. Auch dieser Prozess zur Datenerhebung verläuft automatisch, indem dieses Mal seitens der Root-Knoten die Zeiten vom Initialisieren der Verbindung bis zum Knotenbeitritt aller zwei Knoten protokolliert. Wurde das Netzwerk aus drei Knoten aufgebaut, trennt der Root-Knoten durch Wechseln in den STANDBY-Zustand die Verbindung. Damit das Netzwerke von den Knoten nicht wiederaufgebaut wird, wechseln diese ebenfalls in den STANDBY-Zustand. Ist die aktuelle Minute, eine durch zwei teilbare Zahl, wechseln alle Geräte (einmalig in dieser Minute) in den FINDROOT-Zustand und versuchen erneut das Netzwerk aufzubauen.

**Abbildung 15:** Netzwerkaufbau im sechs Knoten Netzwerk

Für den Aufbau des Netzwerkes wurden hierbei alle zur Verfügung stehenden Smartphones (wie in **Tabelle 5**) benutzt. Aus **Abbildung 15** ist ersichtlich, dass die Verbindung zwischen den ersten zwei Knoten nach durchschnittlich 7027 Millisekunden zu standen kommt. Der dritte Knoten tritt dem Netzwerk nach durchschnittlich 15120 Millisekunden bei, während der vierte Knoten nach durchschnittlichen 23278 Millisekunden verbunden ist. Nach durchschnittlichen 35000 Millisekunden ist auch der fünfte Knoten verbunden. Der letzte Knoten, welcher sich durchschnittlich nach 64290 Millisekunden verbindet, ist dabei fast immer der Samsung S5 Mini, welcher in diesen Testfall aufgenommen wird, um zu demonstrieren das eine Verbindung zu sechs Knoten möglich ist. Damit gilt der letzte Wert nicht unbedingt als Richtwert für den erfolgreichen Verbindungsaufbau von sechs Knoten, da ein Netzwerkaufbau teilweise auch in unter 35-40 Sekunden erstellt werden konnte. Dabei ist zu beachten, dass es während des Netzwerkaufbaus zu Verbindungsabbrüchen kommen kann, welche den Verbindungsprozess teilweise stark verlangsamen können. Verbindet sich ein Gerät mit einem falschen Root, hat dies oft den Vorteil, dass die andere Knoten im Netzwerk währenddessen eine Verbindung ohne größeren gegenseitige Störung zum eigentlichen Root aufbauen können, während die zwei anderen zunächst den Root entdecken und sich zu diesen verbinden.

**Abbildung 16:** Zeiten für den Verbindungsaufbau mit MQTT

Der Verbindungsaufbau seitens MQTT ist hingegen spürbar schneller, da der Netzwerkaufbau kaum abhängig der Netzwerkgröße ist. Stattdessen ist dieser von den Kapazitäten des Brokers abhängig. Die Datenerhebung findet auch hier wieder durch einen automatisierten Test statt, dabei wird die Differenzzeit zwischen Verbindungsanfrage und CONNECT-Event der onLogMessage-Methode des NearflyListeners über 160 Verbindungsversuchen festgehalten. Der Verbindungsaufbau findet in einem WLAN Netzwerk, in welchem die Smartphones eine durchschnittliche Downloadgeschwindigkeit 5 bis 90 Mbit/s[[7]](#footnote-7) haben. Der Broker ist etwa 4 Kilometer entfernt. **Abbildung 16** zeigt, dass der LG G2 mit durchschnittlich 1100 Millisekunden fast das Fünffache der Zeit für die Verbindung benötigt, diese verteilt über alle mit durchschnittlichen 454 Millisekunden jedoch deutlich schneller aufgebaut wird. Dabei beeinflussen sich die Knoten aufgrund der serverbasierten Architektur kaum, sodass das Netzwerk aus zwei und drei Knoten nach einer spätestens aber zwei Sekunden aufgestellt ist.

## Zeiten für die Datenübertragung

Um die Datenübertragungen zwischen beiden Connection-Modi zu vergleichen, werden 100 Bytes, welche in etwa dem durchschnittlichen JSON-String-Payload der implementierten Szenarien entspricht, in über 100 Versuchen übertragen. Da auch hier ein automatischer Test benutzt werden soll und sich die Zeiten im Millisekunden-Bereich zwischen den einzelnen Smartphones unterschieden, wird die Zeit gemessen, die eine Nachricht im Round-Trip braucht. Die Round-Trip-Gegenstelle ist ein zweites Smartphone, welcher die Nachricht nach Eingang sofort wieder publisht. Die Zeitdifferenz zwischen Sendezeitpunkt und Empfangszeitpunkt des Senders dividiert durch zwei wird dabei als Latenz einer Nachricht gesendet.

**Tabelle 6:** Vergleich der Latenzen in beiden Nearfly-Connection-Modis

|  |  |  |  |
| --- | --- | --- | --- |
| **Network Participants** | **MQTT (in ms)** | **NeCon (in ms)** | **NeCon Bluetooth only (in ms)** |
| S9 and s7edge | 156 | 139 | 66 |
| G9 and LG G2 | 152 | 151 | - |
| G5 and G6 | 101 | 155 | - |

**Tabelle 6**, welche die gemessenen Durchschnittswerte mehrerer Testversuche enthält, ist ersichtlich, dass die durchschnittlichen Latenzen, welche beim Publishen von Byte-Nachrichten der Nearfly API entstehen, bei der S9-S7edge-und S9-LG-Kombination durchschnittlich niedriger im NeCon-ConnectionMode ist. Hierbei war das WiFi-Direct im NeCon-Modus fast durchgehend aktiviert. Die ersten Übertragungen verliefen mit Bluetooth, da Wifi-Direct zu Beginn noch nicht von der Nearby Connections API aktiviert wurde. Bei der G5-G6-Kombination hingegen weist MQTT, trotz aktiviertem WiFi-Direct die niedrigeren Latenzen auf. Aus den Messungen lässt sich erkennen, dass das Senden von Byte-Nachrichten im Durchschnitt in beiden Connection-Modi annährend gleich ist. Beim Betrachten der Latenzen eines einzelnen Versuchens, wie dies in **Abbildung 17** gezeigt wird, fällt jedoch auf, dass die Übertragung über die Nearby Connections API deutlich mehr Fluktuation (Jitter) ausgesetzt ist und zeitweise Latenzen im Bereich von 40 bis 800 Millisekunden hat.

Durch Setzen der Strategie auf P2P\_CLUSTER konnte zudem (wie in

**Tabelle 6** zu sehen) die durchschnittliche Latenz im NeCon-Modus ohne Zuhilfenahme des WiFi-Hotspots gemessen werden, die bei einer 100 Bytes Übertragung zwischen S9 und S7Edge mit durchschnittlichen 66 Millisekunden verhältnismäßig niedriger ist.

**Abbildung 17:** Latenzvergleich beiden Connection-Modi mit der S9-S7edge-Kombination

Des Weiteren sollen nun die Zeiten für die Übertragung von Multimedia-Daten in beiden Connection-Modis erfasst und verglichen werden. Analog zum vorherigen Test, entsprechen auch hierbei die Zeiten der Round-Trip-Zeit der Nachricht dividiert durch zwei. Beim NeCon-Modus war dieses Mal der Wifi-Hotspot durchgehend aktiviert. Um ein Kontrast zwischen deaktiviertem und aktiviertem Wifi-Direct zu schaffen, wurde auch hier durch das temporäre Nutzen der P2P\_CLUSTER-Strategie die Nearby Connections API auf die Nutzung von Bluetooth und BLE beschränkt.

**Tabelle 7:** Vergleich der Zeiten der pubFile-Methode in beiden Connection-Modi mit der S9-S7edge-Kombination

|  |  |  |  |
| --- | --- | --- | --- |
| Datengröße | MQTT (in ms) | NeCon (in ms) | Nur Bluetooth (in ms) |
| 67,19 KBytes | 286 | 103 | 1953 |
| 1,01 MByte | 1614 | 556 | 13560 |
| 5,26 MByte | 6121 | 1787 | 71308 |
| 55,57 MByte | 43357 | 14077 | - |

**Tabelle 7** zeigt die Stärke der Nearby Connections API bei größeren Datenübertragungen. Im Verhältnis zu MQTT schafft es die Nearby Connections API, dieselben Multimedia-Daten fast, um Faktor drei schneller zu übertragen. Anders sieht dies vor dem Aufbauen des WiFi-Hotspot aus, wo die Datenübertragung durchschnittlich um Faktor 27 verlangsamt mit steigender Tendenz bei größeren Datenmengen wird. Leider konnte die durchschnittliche Zeit für das Übertragen der 55,57 MByte großen APK in der Cluster-Topologie nicht gemessen werden, da die Verbindung innerhalb mehrerer Versuche, welche mehrere Minuten dauerten, immer wieder nach dem übertragenen von einigen MBytes getrennt und die Übertragung abgebrochen wurde. Damit zeigt sich, dass es ratsam ist, vor dem Übertragen größerer Dateien zu warten, bis der Wifi-Hotspot aktiviert ist. Erkennbar ist dies daran, dass alle Knoten außer der Root mit einem Wifi-Hotspot beginnen mit DIRECT (z.B. DIRECT-0C-B28449) verbunden sind.

## Beispiel-Apps mit Nearfly

Der Nachrichtenaustausch von Text- und Multimedia-Daten konnten erfolgreich implementiert werden und wurde mit einer Messenger App getestet, welche durch Nutzung der Nearfly-Kanäle den Nutzern die Möglichkeit zum Betreten verschiedener Chaträume bietet. Auch ist die Nearfly-Bibliothek entsprechend der in **Tabelle 8** genannten Anforderungen in der Lage priorisierbare Nachrichten im Falle eines Verbindungsabbruchs durch Setzen des Retain-Flags zu puffern. Die Verbindung wir daraufhin automatisch wiederaufgebaut und die Nachricht bei bestehender Verbindung dann versendet.

**Tabelle 8:** Anforderungsmatrix

|  |  |
| --- | --- |
| **Funktionale Anforderungen** | Status |
| textbasierte Nachrichten senden | ✓ |
| Multimedia-Nachrichten senden | ✓ |
| Priorisierung der Nachrichtentypen | ✓ |
| Trennbarkeit unterschiedlicher Nachrichten innerhalb derselben App | ✓ |
| Möglichkeit, dass Nachrichten bei getrennter Verbindung gehalten werden | ✓ |
| automatischer Verbindungsaufbau | ✓ |
| Nahe Echtzeitfähigkeit für ein flüssiges Multiplayer-Spiel-Verfalten bei 30 FPS pro Spieler | O |
| Sichtbarkeit der Nachrichten nur innerhalb derselben App | ✓ |
| Wechselt der Nutzer von Online- in den Offline-Modus sollen keine Daten keine Daten verloren gehen | ✓ |

Durch Vergleichen beider ConnectionModes in der Bouncing Ball App kann die App auf ihre nahe Echtzeitfähigkeit hin getestet werden. Es zeigt sich, wie bereits das die Bouncen Ball App die Nearfly API ausreizt. Im MQTT-Netzwerk steigt die Latenz bereits im zwei Knoten immer weiter an, womit nach einiger Zeit ein Spielen nichtmehr möglich ist. Tatsächlich ist eine derartige nahe Echtzeitfähigkeit durch verringern der Quality of Service (QoS) auf null möglich. Dies wird derzeitig jedoch nicht von der Nearfly API angeboten, da im NeCon-Adapter bisher kein Pendant zur QoS implementiert wurde. Im NeCon-Netzwerk sind die Latenzen ohne aktiviertem Wifi-Hotspot deutlich schlechter und ein Spiel nicht möglich. Wird hingegen der Wifi-Hotspot aktiviert treten mit derselben Anzahl an Knoten hingegen zwar zeitweise spürbare Latenzen auf, welche einen kurzen Nachrichtenburst zu Folge haben, doch lässt sich das Spiel flüssig spielen. Ab drei Netzwerkteilnehmer ist jedoch auch das NeCon-Netzwerk überlastet und kommt mit dem Datentransfer nicht hinterher[[8]](#footnote-8). Womit die Anforderung für nahe Echtzeitfähigkeit (10-30 FPS) nur beschränkt für ein NeCon-Netzwerk mit zwei Knoten und aktivierten Wifi-Hotspot erfüllt wird.

Um die Grenze für das Verhältnis zwischen Geschwindigkeit und Skalierbarkeit von Byte-Nachrichten besser einschätzen zu können, kann jedoch ein Systemtest mithilfe der Share Touchpoint Canvas durchgeführt werden. Die Ergebnisse dieses Systemtest sind in **Tabelle 9** festgehalten und zeigen wie viele Berührungspunkte gleichzeitig ohne größere Verzögerungen seitens des Empfängers (etwa unter einer Sekunde) im Durchschnitt versandt werden konnten. Dabei wurden die Berührungspunkte von einem Nicht-Root Knoten versandt und der Versuch etwa jede dritte Sekunde wiederholt. Es zeigt sich, dass die Übertragung im NeCon-ConnectionMode wie bereits aus der Bouncing Ball App bekannt, im zwei Knoten Netzwerk schneller als in MQTT verläuft. Kommt ein dritter Knoten hinzu, entspricht das Verhalten etwa dem von MQTT. Betritt ein vierter Knoten das Netzwerk können im Durschnitt noch zwei Berührungspunkte relativ schnell versandt werden. Letzten Endes entstehen beim fünf und sechs-Knoten-Netzwerk beim Senden von mehr als zwei gleichzeitigen Berührungspunk zeitweise größere Latenzen. Im MQTT-ConnectionMode hingegen macht die Größe des Netzwerkes meist keinen spürbaren Unterschied.

**Tabelle 9:** Übertragbare Berührungspunkte mit Latenzen unter etwa 1 Sekunde

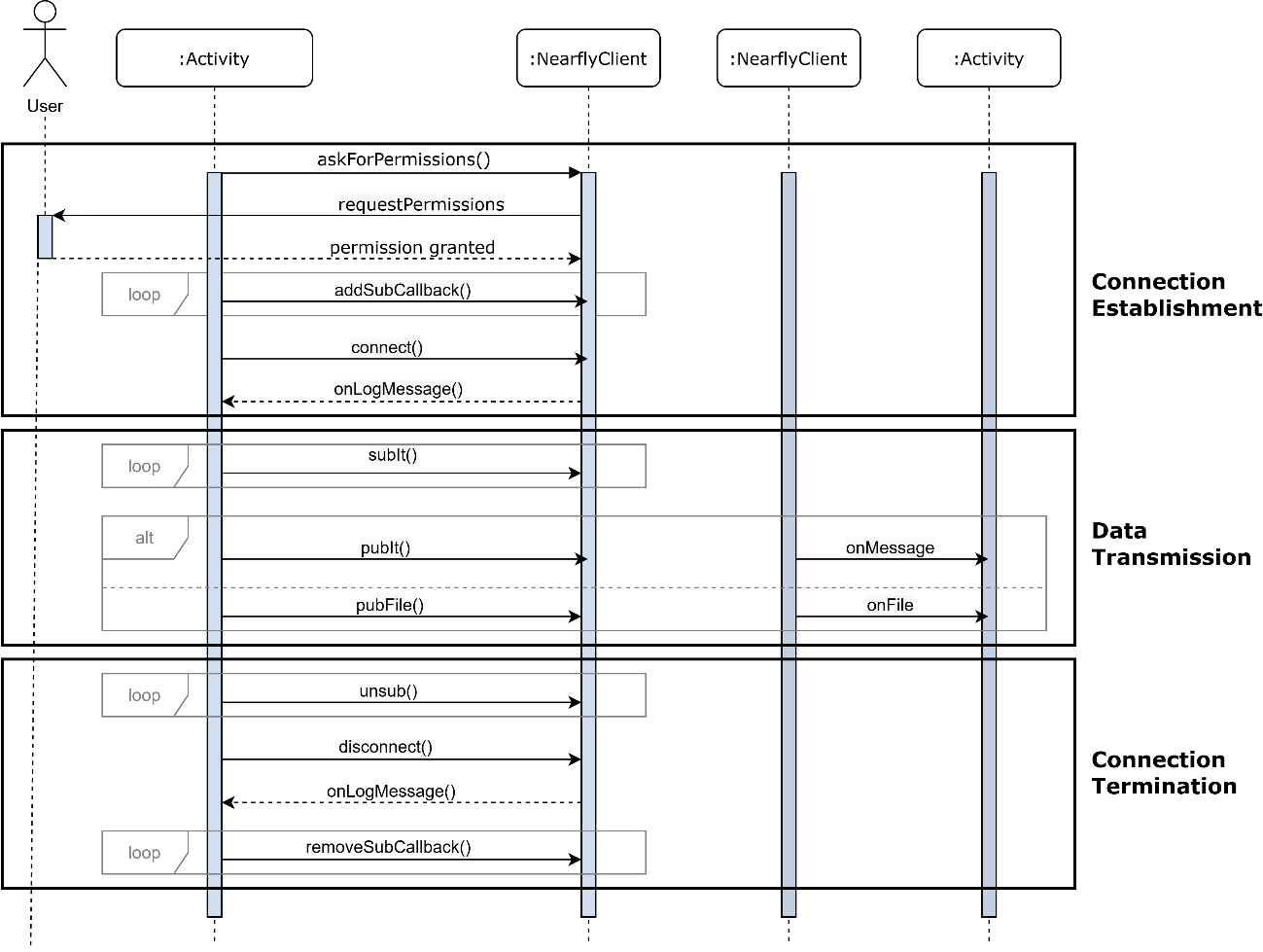
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Knoten | 2 | 3 | 4 | 5 | 6 |
| NeCon | 7 | 4 | 2 | 1-2 | 1 |
| MQTT | 4 | 4 | 4 | 4 | 4 |

Des Weiteren konnte die Anforderung eins dynamischen Wechselns zwischen den ConnectionModes realisiert werden. Die Verbindung zum bestehenden Netzwerk wird dabei erst aufgebaut, wenn eine zum Netzwerk der jeweils anderen Technology besteht. Durch die Einführung des Room-Strings konnte zudem eine Sichtbarkeitsbeschränkung für Apps implementiert werden, welche dieselbe Nearfly API verwenden sich jedoch nicht sehen sollen.

Die Score Board Notepad App ist zudem als Anwendung mit wenigem Datenaufkommen gut durch beide ConnectionModes der Nearfly API bedient. Dabei kann es im NeCon-ConnectionMode besonders bei älteren Geräten vorkommen, dass die Verbindung während einer Spielsession beendet wird. Konkret kommt dies beim LG G2 seltener vor. Auch beim S7 und G5 ist dies bereits vorgekommen. Ganz besonders existiert dieses Problem jedoch beim S5 Mini, welcher deswegen kaum verwendet wurde. Um dem Verbindungsaufbau entgegenzuwirken, werden nur Nachrichten mit absoluten Werten und einem gesetztem Retain-Flag gesendet. Die Wahrscheinlichkeit, dass ein Verbindungsproblem auftaucht, sollte auch bei anderen Anwendungen, welche die Nearfly API im NeCon-ConnectionMode nutzen nicht außer Acht gelassen werden.

## Nutzung der Nearfly API

Nachdem die Bibliothek erfolgreich eingebunden und die obligatorischen Berechtigungen im Manifest eingetragen wurden, kann die Activity gestartet werden. Vor dem Verbinden zum NearflyClient müssen hierfür die mandatorischen Dangerous-Berechtigungen z.B. durch Aufrufen der askForPermissions-Methode[[9]](#footnote-9) abgefragt und vom Anwendungsnutzer (wie in **Abbildung 18**) gewährt werden.



**Abbildung 18:** Ablauf der Nutzung der Nearfly API

Verfügt die App über alle nötigen Berechtigungen, kann ein oder mehr NearflyListener registriert und durch connect unter Angabe vom ConnectionMode die Verbindung zur unterliegenden Technologie hergestellt. Wurde die Verbindung erfolgreich hergestellt wird das onLogMessage-Event vom NearflyListener mit dem CONNECT-Schlüsselwort abgerufen. Dies ist besonders bei Nutzung der Nearby Connections API nützlich, da der Verbindungsaufbau über mehrere Sekunden dauern kann und zuvor kein Datenaustausch möglich ist. Zu beachten ist, dass der NeConAdapter eine Verbindung als aufgebautes Netzwerk mit mindestens zwei Knoten definiert. Werden mehr Knoten erwartet, muss dies durch den API Nutzer implementiert werden.

Nun können beliebige Kanäle durch die subIt-Methode abonnieren und im Anschluss beliebige Nachrichten durch pubIt und pubFile gesendet werden. Der Empfang dieser Nachrichten löst dabei bei den anderen Netzwerkteilnehmern das onMessage bzw. onFile-Event aus.

Soll die Verbindung abgebaut werden, müssen die zum Verbindungsaufbau entsprechenden Gegenoperationen (wie in **Abbildung 18**) in umgekehrter Reihenfolge verwendet werden. Demnach müssen die abonnierten Kanäle durch unsub deabonniert und ein disconnect aufgerufen werden. Durch das Abmelden der registrierten NearflyListener wird schlussendlich der NearflyClient in den Ausgangszustand versetzt.

# Résumé

Die realisierte Wrapper-Bibliothek erlaubt Nearby Connections und MQTT über eine einheitliche API anzusprechen. Dadurch wird dem API Nutzer die Möglichkei geboten, eine Anwendung zu erstellen, welche sowohl im lokalen Internetfreien- wie auch in Umgebungen mit Internetzugang eine Kommunikation zwischen mehreren Smartphones zulässt. Dazu wurde die Nearfly API erstellt, welche durch Nutzung von Adapter-Komponenten die jeweils unterliegenden Technologien erweitern und eine alternative Nutzung ohne große Programmierung ermöglicht.

Es hat sich gezeigt, dass der Netzwerkaufbau mit Nearby Connections derzeitig recht langsam und somit nur einen begrenzten Anwendungsfall besitzt. Die Wahl einer anderen Methodik, wie etwa das manuelle Entscheiden von Advertisers (Root) und Discoverer beschleunigt den Netzwerkaufbau, doch signifikante Verbesserung der Zeiten sind aufgrund der aktuellen Nearby Connections API derzeitig nicht möglich. Wird der Verbindungsaufbau außer Acht gelassen, zeigt Nearby Connections bei aktiviertem Hotspot jedoch gute Datenraten. MQTT hingegen schafft keine es Verbindungen schnell aufzubauen und bei verträglichen Latenzen guter Datenrate zu erzielen.

Besteht die Möglichkeit eines Internetzugangs, sollte beim Erstellen von Netzwerken mit mehr als drei Netzwerkteilnehmern MQTT als ConnectionMode bevorzugt werden. Netzwerke mit zwei bis drei Knoten sind mit dem NeCon-ConnectionMode jedoch besser bedient und können bei aktiviertem Wifi-Hotspot größere Datenmengen deutlich schnellen Austauschen. Besteht ein Netzwerk nur aus zwei Knoten, ist sogar eine nahe Echtzeitkommunikation von bis zu 25 Datenpaketen pro Sekunde bei aktiviertem Hotspot für Spiele möglich. Ist das Internet nicht zugänglich, kann der NeCon-ConnectionMode bei Anwendungen mit geringem Datenaufkommen, wie etwa lokalen Umfrage Apps oder rundenbasierten Spielen für bis zu sieben Teilnehmern verwendet werden. Auch hierbei gilt jedoch die Voraussetzung das spätestens nach dem Aufbauen eines Netzwerkes mit drei Knoten auf das aktivieren des Wifi-Hotspot gewartet werden muss, damit die restlichen Knoten dem Netzwerk beitreten können.

1. - Ausführlichere Diagramme zum Verbindungsaufbau im NeCon-ConnectionMode

**Abbildung 19:** Zeiten für den Verbindungsaufbau zwischen dem LG G2 und dem S9

**Abbildung 20:** Zeiten für den Verbindungsaufbau zwischen dem G5 und dem G6

Literaturverzeichnis

[10] Banks, A, Gupta, R (29.10.2014): MQTT Version 3.1.1 - OASIS Standard.

[16] E. Fallis, P. Spachos (2018): Power Consumption and Throughput of Wireless Communication Technologies for Smartphones. In: , *2018 Global Information Infrastructure and Networking Symposium (GIIS)*.

Online-Quellen

[1] Alexander Kunst (2017): Statista-Umfrage Telekommunikation 2017. https://de.statista.com/statistik/daten/studie/722248/umfrage/umfrage-zur-nutzung-von-smartphone-funktionen-nach-haeufigkeit-in-deutschland/. Abgerufen am 04.06.2020.

[2] Zhang, L (2011): Building Facebook Messenger. https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920. Abgerufen am 13.04.2020.

[3] Perez, S (31.7.2017): Google opens its Nearby Connections tech to Android developers to enable smarter offline apps. https://techcrunch.com/2017/07/31/google-opens-its-nearby-connections-tech-to-android-developers-to-enable-smarter-offline-apps/. Abgerufen am 24.05.2020.

[4] Harmon, W (2018): Google Nearby Connections 2.0 capabilities. https://stackoverflow.com/questions/51976470/google-nearby-connections-2-0-capabilities. Abgerufen am 13.04.2020.

[5] Boshell, B (2017): Average App File Size: Data for Android and iOS Mobile Apps. https://sweetpricing.com/blog/2017/02/average-app-file-size/. Abgerufen am 25.05.2020.

[6] Nearby Connections Team (2018): Nearby Connections API Leitfaden. https://developers.google.com/nearby/connections. Abgerufen am 04.04.2020.

[7] Harmon, W (2019): How performant is Nearby Connections? https://stackoverflow.com/questions/54434616/how-performant-is-nearby-connections/54470958#54470958. Abgerufen am 13.04.2020.

[8] Harmon, W (2018): How can I speed up Nearby Connections API discovery? https://stackoverflow.com/questions/52825617/how-can-i-speed-up-nearby-connections-api-discovery/52882054#52882054. Abgerufen am 13.04.2020.

[9] Google I/O (19.05.2017): How to Enable Contextual App Experiences (Google I/O '17). https://youtu.be/1a0wII96cpE?t=1819. Abgerufen am 04.06.2020.

[11] HiveMQ Team (2019): Client, Broker / Server and Connection Establishment - MQTT Essentials: Part 3. https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/. Abgerufen am 14.04.2020.

[12] Roger Light (27.02.2020): mosquitto.conf man page. https://mosquitto.org/man/mosquitto-8.html. Abgerufen am 04.04.2020.

[13] Ihlenfeld, J (22.4.2009): Bluetooth 3.0 HS mit WLAN-Beschleunigung - Golem.de. https://www.golem.de/0904/66633.html. Abgerufen am 17.05.2020.

[14] Schnabel, P (05/17/2020 22:47:49): Bluetooth Low Energy (4.0 / 4.1 / 4.2). https://www.elektronik-kompendium.de/sites/kom/1805171.htm. Abgerufen am 18.05.2020.

[15] Schnabel, P (05/17/2020 22:48:00): Bluetooth 5. https://www.elektronik-kompendium.de/sites/kom/2107121.htm. Abgerufen am 18.05.2020.

[17] 4G.co.uk (o.J.): How fast is 4G? https://www.4g.co.uk/how-fast-is-4g/. Abgerufen am 04.06.2020.

[18] D. R. C. Silva, G. M. B. Oliveira, I. Silva, P. Ferrari and E. Sisinni (2018): Latency evaluation for MQTT and WebSocket Protocols: an Industry 4.0 perspective. https://ieeexplore.ieee.org/document/8538692. Abgerufen am 04.06.2020.

[19] Triggs, R (2019): Android's Bluetooth latency needs a serious overhaul. https://www.soundguys.com/android-bluetooth-latency-22732/. Abgerufen am 17.06.2020.

[20] Rondón, R, Gidlund, M, Landernäs, K (2017): Evaluating Bluetooth Low Energy Suitability for Time-Critical Industrial IoT Applications. https://doi.org/10.1007/s10776-017-0357-0. Abgerufen am 17.06.2020.

[21] Google (2019): App permissions. https://developer.android.com/guide/topics/permissions/overview. Abgerufen am 02.05.2020.

[22] Harmon, W (2018): Be able to send Messages/Bytes Simultaneous to multiple devices using Nearby Connections. https://stackoverflow.com/questions/52773197/be-able-to-send-messages-bytes-simultaneous-to-multiple-devices-using-nearby-con/52785805#52785805. Abgerufen am 05.06.2020.

[23] MacKenzie, D (2020): nice(1) - Linux man page. https://www.man7.org/linux/man-pages/man1/nice.1.html. Abgerufen am 02.06.2020.

[24] Harmon, W (2019): NearbyConnection: payload.asFile.asJavaFile is null when retrying to saveFile after storage permissions initially not granted. https://stackoverflow.com/questions/55328027/nearbyconnection-payload-asfile-asjavafile-is-null-when-retrying-to-savefil/55345391#55345391. Abgerufen am 05.06.2020.

[25] Siahaan, APU (2017): Base64 Character Encoding and Decoding Modeling. https://10.31227/osf.io/ndzqp. Abgerufen am 04.06.2020.

[26] Harmon, W (2020): Is there a lower bound for the connection time of Nearby Connections API? https://stackoverflow.com/questions/61614967/is-there-a-lower-bound-for-the-connection-time-of-nearby-connections-api/61639915#61639915. Abgerufen am 05.06.2020.

1. Die Daten, welche zur Bildung der in der Tabelle ermittelten Werte dienen aus einem LG G2 mini LTE, ein Samsung Galaxy S7 Edge, sowie ein Samsung Galaxy S9. Ausgenommen sind die Werte für den Größenbereich der APKs. [↑](#footnote-ref-1)
2. Der Durchschnitt aller im Play Store angebotenen Anwendungen beträgt dabei 11,5 MByte [5]. [↑](#footnote-ref-2)
3. Ein Topic ist ein UTF-8 Encodierter String der durch einen Schrägstrich (‚/‘ U+002F) hierarchisch, gleichend einer Baumstruktur, untergliedert werden kann z.B. „Baum/Zweig“ inkludiert „Baum/Zweig/Blatt“ ( [10]). [↑](#footnote-ref-3)
4. Die 24 Stunden in Millisekunden entsprechen dabei . [↑](#footnote-ref-4)
5. Dabei werden die Daten von Nearby Connections in einem externen Verzeichnis gespeichert, da Nearby Connections in einem separaten Prozess läuft und dadurch keinen Zugriff auf den Anwendungsspezifischen Verzeichnis der Anwendung hat, welche die API nutzt. [↑](#footnote-ref-5)
6. Denkbar wäre es auch das Chunking zu benutzten, um eine Priorisierung zu ermöglichen. Um die gleiche Verhaltensweise seitens beider Adapter zu erzielen, müsste jedoch auch beim NeConAdapter eine solche File-Priorisierung durch Chunking implementiert werden. Dies würde den Datenübertragung Prozess verlangsamen und soll daher nicht implementiert wird. [↑](#footnote-ref-6)
7. Der Samsung hatte hierbei zwischenzeitlich eine Downloadgeschwindigkeit von 90 Mbit/s. Während es beim G6 8 Mbit/s und bei G5 31.5 Mbit/s waren. Der LG schaffte 5 Mbit/s und der Samsung s7edge wiederum 35 Mbit/s. [↑](#footnote-ref-7)
8. Hierbei sollte beachtet werden, dass bei zu schnellen Übertragungen das Smartphone selbst zum Flaschenhals werden kann und dies in diesem Fall (90 Nachrichten/s) nicht vollständig ausgeschlossen werden kann. [↑](#footnote-ref-8)
9. Bei Nutzung der Nearfly API über den NearflyService, ist hierbei zu beachten, das Methodenaufrufe des NearflyService erst nach dem erfolgreichen Binding des Services möglich sind. [↑](#footnote-ref-9)