Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Verteilung von Positionsdaten in Wireless-Mesh-Networks

Abschlussarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

an der
Hochschule für Technik und Wirtschaft Berlin
Fachbereich 4 - Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

1. Prüfer:	Prof. Dr. Thomas Schwotzer
2. Prüfer:	Dipl.-Ing. (FH) Stefan Zech
eingereicht von:	Bastian Rosner
Datum:	5. August 2014

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Im Informationszeitalter sind Netzwerk-Infrastrukturen die Basis der digitalen Kommunikation. Drahtlose Netzwerke mit Anbindung an das Internet sind in vielen Haushalten vorhanden, jedoch wird der Öffentlichkeit meist die Mitbenutzung verwehrt.

Freie drahtlose Netzwerke existieren schon seit über 10 Jahren, haben jedoch in der letzten Zeit Aufgrund des technischen Fortschritts im Bereich mobiler Geräte und dem wachsenden Bedarf an Connectivity wieder großes Interesse geweckt. Sie sind heutzutage nicht mehr nur die Spielwiese von Enthusiasten, sondern stellen einen Mehrwert für die Gesellschaft dar.

Die Zukunft wird vom *"Internet der Dinge"* geprägt sein. Sensor-Netzwerke sind schon jetzt in der Heimautomation üblich und bald werden sie ein allgegenwärtiger Bestandteil von Fahrzeugen, Bekleidung und Haushaltsgeräten sein. *Ubiquitous Computing* beschreibt diese drahtlosen und untereinander vernetzen Computer als spontan vermaschte Netzwerke.

1.2. Zielsetzung

Die räumliche Ausbreitung von drahtlosen Mesh-Netzwerken auf topographischen Karten aufzuzeigen stellt ein praktisches Werkzeug für das weitere Wachstum dieser Netzwerke dar:
Bei offenen Netzwerken wird Individuen die Möglichkeit gegeben, durch Betreiben eigener WLAN-Standorte die Verbreitung des Netzwerks zu vergrößern. Mit der geographischen Darstellung von Zugangspunkten wird diesen Personen der eigene Beitrag präsentiert und potentiellen Mitstreitern vorhandene Einstiegspunkte zum Anknüpfen visualisiert.

Diese Arbeit soll einen Einblick in den Teilaspekt der Positionsdaten von WLAN-Knoten in Mesh-Netzwerken liefern. Die darin erarbeiteten Erkenntnisse sollen sowohl den Benutzern solcher Netze beim Finden von Zugangspunkten helfen, als auch den Betreibern von Mesh-Topologien ein Werkzeug zur Darstellung der Infrastruktur liefern.

Mehrere bestehende Systeme zum Austausch von Positionsdaten in Wireless-Mesh-Networks sollen aufgezeigt und im Leistungsumfang verglichen werden. Ein besonderes Augenmerk wird auf die unterschiedlichen Anwendungsszenarien *Darstellung von Zugangspunkten*,

Visualisierung der Mesh-Topologie und *Tracking* gelegt.

Eines dieser Systeme soll im Detail analysiert und das dazugehörige Netzwerk-Protokoll spezifiziert werden. Dies soll die Grundlage zur Entwicklung einer zur Referenz kompatiblen Android-Applikation liefern, welche im praktischen Abschnitt der Arbeit prototypisch implementiert wird.

Die Architektur des Prototypen wird unter Android-spezifischen Gesichtspunkten näher beschrieben. Diese soll bei der Entwicklung zukünftiger Erweiterungen als Dokumentation dienen.

1.3. Aufbau dieser Arbeit

Im nachfolgenden Kapitel sind die thematischen Grundlagen und Definitionen dargestellt. Hierzu werden die verwendeten Technologien beschrieben und Kausalitäten erläutert.

In Kapitel 3 werden bestehende Systeme im Detail vorgestellt und Ihre Vor- und Nachteile gegenübergestellt. Unterschiedliche Anwendungsfälle werden in der Einleitung beschrieben und am Ende als Kriterien für die Bewertung der Technologien verwendet.

Das vierte Kapitel der Arbeit beinhaltet die Analyse einer der zuvor beschriebenen Technologien auf Netzwerk-Ebene. Eine bis dato unvollständige Spezifikation des Netzwerkprotokolls wird anhand der Analyse der Referenz-Implementierung um wichtige Punkte ergänzt. Die Grundlagen zum Definieren einer kompatiblen Netzwerk-Komponente werden gelegt.

Kapitel 5 dient dem Entwurf eines Prototypen für das zuvor detailliert beschriebene System. Zunächst werden die formalen Anforderungen definiert und im Verlauf eine modulare Architektur anhand einzelner Komponenten erarbeitet.

Auf die Implementierung des Entwurfs wird in Kapitel 6 genauer eingegangen. Das Zusammenspiel der Komponenten wird veranschaulicht und Besonderheiten bei der Umsetzung des Netzwerk-Protokolls für Android werden erläutert.

Der fertige Prototyp sowie die Spezifikation des Netzwerk-Protokolls werden im siebten Kapitel durch Tests und Analysen geprüft. Verschiedene Testverfahren und Methoden werden vorgestellt und ein Testkonzept am Prototypen angewendet.

Den Abschluss bildet ein Resümee über die entwickelte Anwendung und dessen Erweiterbarkeit.

2. Grundlagen

2.1. Mobile Ad-Hoc Networks

Viele drahtlose Systeme basieren auf dem Konzept einer direkten *Punkt-zu-Punkt* (PtP) Verbindung. Bei den bekanntesten Technologien – *Group Standard for Mobile communications* (GSM) und *Wireless Local Area Network* (WLAN) – kommunizieren mobile *Knoten* (Nodes) direkt mit einem zentralen Zugriffspunkt. Diese Netzwerke haben eine zentralistische Topologie und werden von einer zentralen Stelle aus konfiguriert und betrieben.

Dazu entgegengesetzt aufgebaut ist das *Multi-Hop* Konzept, bei welchem mobile Nodes spontane (ad hoc) Verbindungen zu anderen Knoten als Teilstrecken zum eigentlichen Ziel verwenden. Dieser Ansatz des Weiterreichens von Daten über mehrere Knoten (*Multi-Hop Relaying*) ist die technologische Grundlage von *Mobile Ad-Hoc Networks* (MANET).

Durch den dezentralen Ansatz kann ein MANET spontan bei Bedarf aufgebaut werden. Die Infrastruktur soll sich automatisch selbst konfigurieren und einzelnen Knoten dürfen mobil sein. Ein MANET baut somit eine dynamische Netzwerk-Topologie auf.

2.1.1. Wireless Ad-Hoc Modus

Zwar sind Ad-Hoc Netzwerke nicht an spezielle Hardware gebunden, aber fast alle Netzwerke dieser Art sind auf Basis von WLAN (IEEE 802.11) aufgebaut. Laut IEEE sieht der 802.11 Standard zwei unterschiedliche Arten der Kommunikation vor, jedoch keiner der beiden bindet das Konzept von Multi-Hop Relays nativ ein:

- **Infrastruktur Modus**: Das drahtlose Netzwerk wird von mindestens einem Access-Point (AP) für eine Anzahl mobiler Knoten aufgespannt. Diese Konfiguration wird im Falle eines einzigen AP *Basic Service Set* (BSS) genannt. Sollten durch mehrere APs mehreren Zellen des selben Netzwerks aufgebaut werden, spricht man von *Extended Service Set* (ESS)
- **Ad-Hoc Modus**: Hierbei handelt es sich um PtP Verbindungen zwischen zwei Knoten. Da dieser Modus unabhängig von einer bestehenden zentralen Infrastruktur ist, spricht man auch von *Independent Basic Service Set* (IBSS)

Durch den Ad-Hoc Modus alleine können Nodes jedoch nicht als Multi-Hop Relay fungieren, da Datenpakete nur an direkt benachbarte Knoten ausgetauscht werden. Für den

Betrieb eines MANET wird deshalb noch ein *Routing-Protokoll* benötigt, welches die dynamischen Pfade im Netzwerk handhabt.

2.1.2. 802.11s

In 2003 wurde durch das "Institute of Electronics and Electrical Engineering" (IEEE) eine Arbeitsgruppe geschaffen, um Mesh-Funktionalität innerhalb des 802.11 Standards zu definieren. 802.11s ist 2006 als Entwurf angenommen worden und wird seither ergänzt. Da seit der letzten Prüfung des Entwurfs im Jahr 2011 noch Diskussions-Punkte offen sind, ist 802.11s bis dato noch nicht als Standard verabschiedet. Kern-Komponenten von 802.11s sind jedoch bereits fertig definiert und in wenigen Hard- und Software-Projekten implementiert. Unter Linux lässt sich inzwischen mit WLAN-Chips der Firma Atheros der derzeitige Entwurf von 802.11s verwenden. Das Projekt open802.11s[1] dient als Basis für die Weiterentwicklung und Tests des Entwurfs. In Laptops des Projekts "One Laptop per Child" (OLPC) wurde 2008 ein früher Entwurf von 802.11s implementiert. Rastogi et al. haben in [RRN09] ein Mesh aus OLPC-Laptops analysiert und stellten fest, dass der bis dahin vorliegende Entwurf nicht den Ansprüchen genügt.

WLAN-Geräte in einem Mesh auf Basis von 802.11s verhalten sich ähnlich wie im IBSS Modus, jedoch werden unterschiedliche Typen von Knoten definiert. Camp und Knightly beschreiben diese:

> "Any node that supports the mesh services of control, management, and operation of the mesh is a mesh point (MP). If the node additionally supports access to client stations (STAs) or non-mesh nodes, it is called a mesh access point (MAP). A mesh portal (MPP) is an MP that has a non-802.11 connection to the Internet and serves as an entry point for MAC service data units (MSDUs) to enter or exit the mesh." [CK08]

802.11s soll auch mehrere Möglichkeiten zum Authentifizieren von Knoten und Verschlüsseln von Datenverkehr bieten. André Egners analysiert in [Egn10] die im Draft vorgeschlagenen Verfahren und kommt dabei zu dem Schluss, dass diese nicht den Bedürfnissen einer flexiblen Mesh-Struktur gerecht werden. Die geplante Authentifizierung auf Basis von Passwörtern lässt sich nur schwer in das Konzept von flexiblen und dynamischen Mesh-Netzwerken integrieren.

Es ist anzunehmen, dass 802.11s in Zukunft eine Rolle in drahtlosen Mesh-Netzwerken spielen wird. Für den Betrieb eines 802.11s Mesh-Networks wird weiterhin ein Routing-Protokoll benötigt. Ob dazu die in MANETs heute üblichen Protokolle Verwendung finden werden, oder neue Ansätze zu erarbeiten sind, bleibt offen.

[1] `http://open80211s.org/open80211s/`

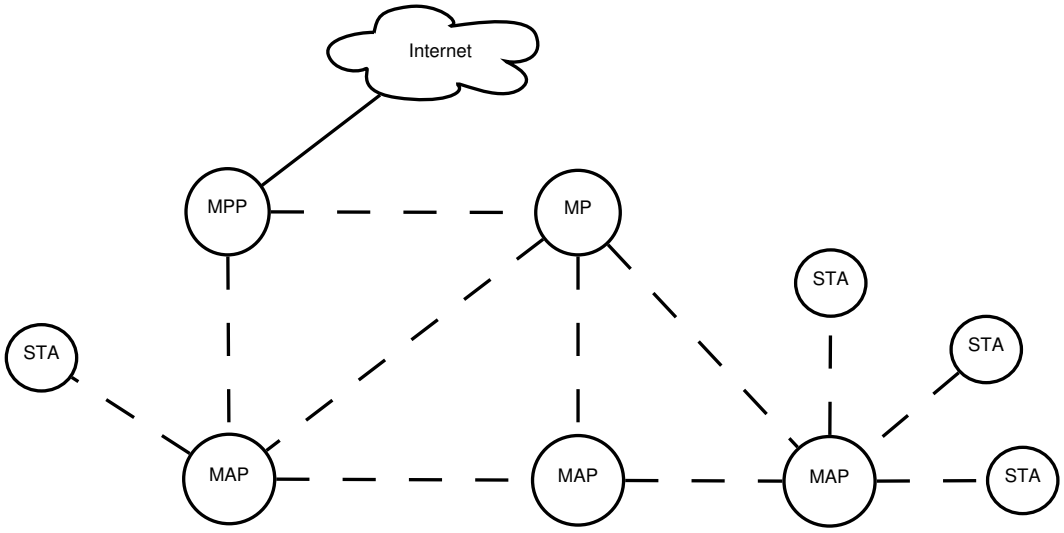

Abbildung 2.1.: 802.11s Node-Types

2.2. Wireless Mesh Networks

Die zuvor beschriebenen Technologien ermöglichen den Aufbau von Wireless Mesh Networks (WMN). Die dabei aufgespannte vermaschte Netzwerktopologie wird häufig als Alternative zu aufwändigen und teuren Verkabelungen über großflächige Areale verwendet. Gleichzeitig dient sie dem Anschluss von mobilen Endgeräten.

Zwar ist die zu erwartende Leistung im Sinne von Bandbreite i.d.R. um ein vielfaches geringer als bei Kabel-Verbindungen, jedoch vereinfacht die dynamische Struktur die Erweiterung um zusätzliche Knoten immens.

WMNs sind nicht auf WLAN nach IEEE 802.11 beschränkt. *Wireless Sensor Networks* arbeiten nach demselben Mesh-Prinzip wie WMNs, jedoch tauschen die drahtlosen Sensor-Module ihre Daten über *ZigBee* (IEEE 802.15.4) aus. Die Anwendungsbereiche sind groß und reichen von Temperatur- und Lichtsensoren in der Heimautomation über Messungen zur Wasser- und Luftqualität bis hin zu implantierten Blutzuckermessgeräten.

Besonders im privaten Gebrauch werden sich WMNs zukünftig stark verbreiten. Die Konzepte von dezentralen ad-hoc Multihop-Relay Netzwerken sind die Grundlage für das "*Internet of Things*": So werden z.B. Fahrzeuge Verkehrs-Informationen in spontanen *Vehicle-to-Vehicle* ([MTKM09]) Mesh-Netzwerken austauschen und Menschen erhalten von Armbändern ausgelesene Gesundheitsdaten auf Smartphones präsentiert.

2.3. Wireless Community Networks

Ein *Wireless Community Networks* (WCN) ist in den meisten Fällen ein *Community Network* (CN) das auch ein WMN mit *bottom-up* Ansatz betreibt. Dabei werden weitere Knoten ohne großen Planungs-Aufwand von Einzelpersonen zum Mesh hinzugefügt. Ein WCN wird von einer lokalen Community als alternative Netzwerk-Infrastruktur gepflegt um Internet-Connectivity zu liefern bzw. zu erhalten, jedoch auch um dezentrale Internet-unabhängige Kommunikationskanäle zu schaffen. Mit heutzutage günstiger WLAN-Hardware für nicht lizenzierte Funkfrequenzen und dem Konzept von Multi-Hop Mesh-Networking lässt sich ein kilometerweit entfernter Internet Breitband-Anschluss über ein großflächiges Areal an viele Endnutzer verteilen.

Tabelle 2.1 soll einen kurzen Überblick über die Verbreitung von WCNs in Europa schaffen.

Name	**Standorte**	**Anzahl Nodes**	**Protokoll**	**Gründungsjahr**
Freifunk.net[2]	Deutschland (>40 Städte)	~ 3500	OLSR batman-adv	2003
Funkfeuer.at[3]	Österreich (Wien, Graz, u.a.)	~ 500	OLSR	2003
guifi.net[4]	Spanien	> 25.000	BGP BMX6	2004
AWMN[5]	Athen	> 3000	OSPF/BGP OLSR	2002

Tabelle 2.1.: Europäische Wireless Community Networks

Einige CNs sind als Internet Service Provider (ISP) registriert und betreiben eigene autonome Systeme (AS) im Internet um sich mit anderen Providern in Rechenzentren zu verbinden und Daten auszutauschen. Das CN *guifi.net* verlegt z.B. seit 2009 eigene Glasfaserleitungen um den Breitbandausbau in ländlichen Regionen zu fördern. Das Projekt wird unter dem Namen "Fibre to the Farms" (FTTF) geführt.

Gleichzeitig bietet ein so organisiertes CN die Möglichkeit, Endnutzer vor rechtlichen Situationen zu schützen. Das Konzept von frei zugänglichen Internet-Anschlüssen ist z.B. in Deutschland nicht mit der derzeitigen Rechtsprechung zur sog. *Störerhaftung* kompatibel. Privatpersonen, die ihren Internet-Zugang öffentlich zugänglich machen, werden für die Unterlassung von Urheberrechtsverstößen, welche durch Dritte verursacht werden, verantwortlich gemacht.[MS14]

CNs funktionieren durch ehrenamtliches Engagement und sind häufig als Vereine bzw NGOs organisiert. Sie handeln nicht im Sinne der wirtschaftlichen Gewinnoptimierung.

[2] `http://freifunk.net`
[3] `http://funkfeuer.at`
[4] `http://guifi.net`
[5] `http://awmn.net`

Damit sind sie ideal als Grundlage für die Entwicklung und Evaluierung von Technologien für Breitband-unterversorgte und wirtschaftlich unrentable Regionen, wie z.B. Entwicklungsländer.

Ein Musterbeispiel ist die NGO *Village Telco*:

> "The Village Telco (VT) is an easy to use, scalable, standards-based, and DIY (Do it Yourself) telephone company toolkit."[AGS11]

VT entwickelt Open-Source Soft- und Hardware für Telefonie auf Basis von WMN. Die dabei entstandene *Mesh Potato*, ein Wireless-Router mit Anschluss für analoge Telefone sowie Voice-over-Internet Protocol (VoIP) Software trägt zur Telefonie-Grundversorgung in Südamerika und Afrika bei.

Die Europäische Union fördert derzeit zwei Forschungsprojekten in Zusammenarbeit mit Community Networks: Bei den Projekten *Community Networks Testbed for the Future Internet* (CONFINE)[6], im Rahmen von *Future Internet Research and Experimentation Initiative* (FIRE) und *Commons4Europe*[7] innerhalb des *Competitiveness and Innovation Framework Programme* wird u.a. in Kooperation mit den CNs *guifi.net* und *Funkfeuer* die Viabilität von CNs als Ergänzung zu den bestehenden Modellen der Internet Infrastruktur erforscht.

2.4. Dynamische Routing Protokolle

Die Mesh-Topologie in WMNs ermöglicht ein hohes Maß an Redundanz. Um diese zu erreichen, sind jedoch Routing-Protokolle nötig, welche die zu Grunde liegende Topologie möglichst effizient ausnutzen. Ein Routing Protokoll muss dynamisch auf den Ausfall von Wireless Routern reagieren und gleichzeitig neue Knoten einbinden können. Dazu werden *dynamische Routing Protokolle* (DRP) eingesetzt. In Wireless-Verbindung zwischen zwei Routern können spontan schwankende Latenzen oder Bandbreiten auftreten, dies ist meist anderen Nutzern des Frequenz-Spektrums geschuldet oder tritt wegen unvorhersehbaren Wettereinflüssen auf. Ein DRP sollte auch auf solche Veränderungen in der Qualität der Verbindungen reagieren und entsprechend bessere Pfade wählen.

An der Entwicklung von DRPs für Mesh-Netzwerke wird hauptsächlich im wissenschaftlichen Umfeld gearbeitet. Raniwala et. al. listen in [RC05] eine Reihe von Architekturen und Algorithmen für WMNs auf. Fast alle Protokolle lassen sich, sofern keine Mischform vorliegt, grob in zwei Klassen unterteilen:

- Proaktives Routing setzt eine ständig aktuelles Abbild der Mesh-Topologie auf jedem einzelnen Teilnehmer des Netzwerks voraus um ständig den idealen Pfad zu jedem anderen Knoten zu kennen. Der Vorteil liegt in der geringen Latenz, da ein Pfad schon vor der ersten Benutzung bekannt ist und nicht erst berechnet werden

[6] `http://confine-project.eu`
[7] `http://commonsforeurope.net`

muss. Der Nachteil ist ein ständiges "Hintergrundrauschen" durch Kontrollpakete zum Aktualisieren der Topologie-Informationen.

- Reaktives Routing verfolgt den On-Demand Ansatz. Ein Pfad wird erst ermittelt wenn dieser wirklich benötigt wird. Dabei werden Kontrollpakete an benachbarte Router verschickt, die diese wiederum weiterleiten bis das eigentliche Ziel erreicht ist. Erst wenn die Hin- und Rückroute bekannt ist, kann eine Datenverbindung aufgebaut werden. Dies wirkt sich negativ auf die initiale Latenz aus, ist jedoch mit weniger Rechenaufwand verbunden und damit stromsparender.

Zudem unterscheidet man bei DRPs auch grob zwischen Link-Status (link-state) und Distanz-Vektor (distance-vector) basierenden Protokollen:

link-state Protokolle berechnen bei der Entscheidung einer Route zuerst den Status jeder Einzelverbindung in einem Pfad und finden diejenige Route mit der besten gesamt-Metrik.
Open Shortest Path First (OSPF) ist ein Beispiel für ein link-state Protokoll, das häufig für das interne Routing eines autonomen Systems (AS) verwendet wird.

distance-vector Protokolle entscheiden Routen auf Basis einer Vektor-Metrik, bestehend aus der Distanz und Richtung zum Ziel.

Das Border Gateway Protocol (BGP) ist ein fast reines distance-vector Protokoll, welches im Internet für das Routing zwischen AS Einsatz findet.

In RFC 2501 [CM99, Seite 6] wird eine Reihe von sinnvollen Qualitäts-Eigenschaften an Routing-Protokolle für MANETs vorgeschlagen. Mohammad Siraj hat in der Arbeit "A Survey on Routing Algorithms and Routing Metrics for Wireless Mesh Networks" [Sir14] u.a. daraus diese Liste für effiziente DRPs in WMNs zusammengefasst:

- **Verteilt**
  Ein Mesh-Protokoll soll nicht zentralistisch aufgebaut sein, da ein gewisses Maß an Mobilität von Knoten möglich sein soll.

- **Anpassung an Topologie-Veränderungen**
  Der verwendete Algorithmus muss das Auftreten von Veränderungen in der Topologie vor allem aufgrund mobiler Knoten in der Berechnung der Routen berücksichtigten.

- **Frei von Schleifen**
  Eine fundamentale Voraussetzung eines jeden DRPs ist die Vermeidung von Schleifen (loop-free) in Pfaden. Loops in Mesh-Netzwerken treten z.B. während der Berechnung von Routen für einen mobilen Knoten auf. Loops verschwenden unnötig Bandbreite.

- **Sicherheit und Integrität**
  Falls keinerlei Sicherheitsmechanismen in einem DRPs eingesetzt werden, sind diverse Angriffsszenarien gegen die Verfügbarkeit des Netzwerks möglich.

- **Skalierbarkeit**
  Mit steigender Anzahl von Knoten sollte die Performance des Netzwerks beibehalten werden. Routing-Protokolle verwenden eigene Kontroll-Pakete um Knoten und Pfade im Netzwerk zu erfassen, weshalb die meisten Routing-Protokolle ab einer gewissen Größe viel Bandbreite nur zum Aufrechthalten der Topologie benötigen.

2.4.1. Metriken

Jedes Routing-Protokoll verwendet eine Metrik zum Bewerten der Güte einer Verbindung, um bei mehreren möglichen Routen zum Ziel eine sinnvolle Entscheidung treffen zu können. Das Maß der Güte kann aufgrund unterschiedlicher Faktoren bestimmt werden. In statischen Netzwerk-Topologien wird meist die Bandbreite oder die Länge eines Pfades verwendet. Andere Faktoren können die Latenz, Verlässlichkeit oder finanziellen Kosten sein. Je nach eingesetztem Faktor werden hohe oder niedrige numerische Werte bevorzugt. Die Metrik $d(n_i, n_j)$ eines Pfades $p = (n_1, n_2, ..., n_m)$ wird durch die Berechnung der Einzelmetriken aller Knoten zwischen den Knoten n_i und n_j entlang dieses Pfades bestimmt. Für solch eine Berechnung gibt es drei Verfahren[BHSW07],[Sir14]:

- **Additiv**
  Latenzen werden pro Hop addiert. Je niedriger die gesamte Latenz, desto besser der Pfad:

$$d(p) = d(n_1, n_2) + d(n_2, n_3) + ... + d(n_{m-1}, n_m)$$

- **Multiplikativ**
  Basiert die Metrik auf prozentualen Werten, wie z.B. Verlässlichkeit bzw. Packet-Loss, werden die Einzelmetriken multipliziert:

$$d(p) = d(n_1, n_2) \cdot d(n_2, n_3) \cdot ... \cdot d(n_{m-1}, n_m)$$

- **Konkav**
  Die maximale Bandbreite eines gesamten Pfades ist immer die kleinste Bandbreite zwischen Knoten:

$$d(p) = min(d(n_1, n_2), d(n_2, n_3), ..., d(n_{m-1}, n_m))$$

Im folgenden werden zwei häufig verwendete Metriken für Mesh-Netzwerke vorgestellt und kurz bewertet:

Expected Transmission Count (ETX) Metrik

ETX wird verwendet um – speziell bei Multi-Hop Wireless-Verbindungen – die Anzahl der benötigten Übertragungsversuche (Transmissions) in einen Wert zu fassen. Zur Erstellung dieser Größe werden in regelmäßigen Abständen Prüfpakete versendet. Die empfangende Seite kann somit aus dem Fehlen von Prüfpaketen einen Paketverlust ermitteln. Die Anzahl der zu erwartenden Übertragungen um ein Datenpaket von einem Knoten zu einem direkt benachbarten zu schicken kann wie folgt berechnet werden:

$$ETX = \sum_{k=1}^{\infty} kp^k(1-p)^{k-1} = \frac{1}{1-p}$$

Da diese Metrik auf der zeitlichen Paketverlust-Rate basiert, ergeben sich mehrere Nachteile: Sie passt sich nicht sofort an Veränderungen an. Erst mit dem Ablauf mehrerer Zeitintervalle zeigt sich eine Qualitätsveränderung auch im ETX-Wert.

Paketverluste werden nicht exakt gemessen. Variierende Verlustraten auf einer Linkstrecke werden nicht berücksichtigt.

ETX berücksichtigt nicht unterschiedliche Einzelmetriken zwischen Knoten. Wenn mehrere Wege zwischen zwei Knoten möglich sind, kann ETX nur die gesamte Verlustrate angeben und nicht unterscheiden, ob einer der Pfade eine bessere Wahrscheinlichkeit zur Übertragung liefert.

Minimal Hop Count Metrik

Eine in vielen Routing-Protokollen verbreitete Metrik ist die sog. *"Minimal Hop Count"* (MHC) Metrik. Die Funktionsweise, immer die kürzeste Route – im Bezug auf die Anzahl der Hops – zu wählen, ist ein simpler Algorithmus. Für nicht-drahtlose Verbindungen ist diese Herangehensweise auch sehr praktikabel, da dadurch Latenzen minimiert und Bandbreiten maximiert werden können. Bei drahtlosen Verbindungen ist jedoch die kürzeste Route nicht zwingend die kürzeste Link-Strecke. WLAN-Verbindungen über große Distanzen arbeiten meist mit niedrigeren Datenraten um eine stabile Verbindung aufrecht zu erhalten. Im von Draves et al. [DPZ04] aufgestellten Vergleich mit ETX liefert MHC in einer Topologie mit sich bewegenden Knoten bessere Werte. Die Neuberechnung der Link-Qualitäten bei ETX – abhängig von den Zeitintervallen der Prüfpakete – reagiert weniger schnell auf Veränderungen als ein simples Zählen von Hops.

2.4.2. Optimized Link-State Protocoll

Das *Optimized Link-State Routing Protocoll* (OLSR) ist ein *pro-aktives table-driven link-state* Routing-Protokoll, optimiert für Wireless Ad-Hoc-Netze und standardisiert nach IETF RFC 3626 [CJ03].

Als Metrik wurde ursprünglich *Hop Count* gewählt, was sich als nicht praktikabel erwies. Bei einer klassischen *minimal hop count metric* wird die Güte der Pfade nicht berücksichtigt. Die kürzeste Route kann wenig Bandbreite liefern oder sogar unter Packet-Loss leiden. Da diese Route aufgrund der Metrik Priorität erhält, wird die Qualität der Verbindung unter Last noch weiter einbrechen. Zusätzlich führt die im RFC vorgeschlagene Hysterese zur Auswahl von *Multipoint Relays* (MPR) nach kurzer Zeit zu oszillierend wechselnden Routen, wodurch die Chance auf Routing-Loops steigt und ein Mesh so instabil wird, dass es als unbenutzbar gilt.[Wag09]

Bei dem im RFC vorgeschlagenen Konzept von MPRs wird versucht, den Broadcast-Traffic optimiert zu verteilen. OLSR-Nachrichten, und somit auch die Information über Topologie-Veränderungen, werden per Broadcast versendet. MPRs sollen Nachrichten mit erhöhter Zuverlässigkeit ausliefern und gleichzeitig Redundanzen vermeiden.

Es werden Nachbar-Knoten mit möglichst vielen weiteren Nachbarn zweiter Ordnung als Multipoint Relays definiert. Für diese speziellen Nodes sind Informationen über Topologie-Änderungen wichtig, da sie entscheiden müssen, zu welchem ihrer Nachbarn geroutet wird.

Die Entscheidung, welcher Knoten als MPR gewählt wird, basiert auf der MHC Metrik und einer Hysterese für Paketverlust. Diese Kombination ist anfällig für Schwankungen. Da mit MPRs die Topologie dazu verwendet wird, effizient u.a. Veränderungen über die Topologie zu übertragen, entsteht eine zirkuläre Abhängigkeit, welche letztendlich zu falschen Routing-Informationen führt.

Für RFC 3626 wurde zwar kein Update eingereicht, jedoch ist OLSR inzwischen soweit verbessert, das es die Basis für eine Vielzahl von WMNs bereit stellt. Die meisten größeren Mesh-Netzwerke auf Basis von OLSR haben jeden Knoten als MPR definiert und verwenden ETX als Metrik. [Mac13]

In einem OLSR-Mesh ist jedem einzelnen Knoten des Netzwerks die gesamte Netztopologie bekannt, womit durch Einsatz des Dijkstra-Algorithmus die jeweils optimale Route zu jedem anderen Knoten berechnet werden kann.

In [Tø04] – der Master-Thesis "Impementing and extending the Optimized Link State Routing Protocol" von Andreas Tønnesen – wird die Eigenschaft *table-driven* von OLSR beschrieben:

> "Being a table-driven protocol, OLSR operation mainly consists of updating and maintaining information in a variety of tables. The data in these tables is based on received control traffic, and control traffic is generated based on

information retrieved from these tables. The route calculation itself is also driven by the tables."

In einer minimalen Konfiguration ohne Plugins verwendet OLSR ein gemeinsames Datenformat:

"OLSR communicates using a unified packet format for all data related to the protocol. The purpose of this is to facilitate extensibility of the protocol without breaking backwards compatibility. This also provides an easy way of piggybacking different "types" of information into a single transmission, and thus for a given implementation to optimize towards utilizing the maximal frame-size, provided by the network." [CJ03][p. 13]

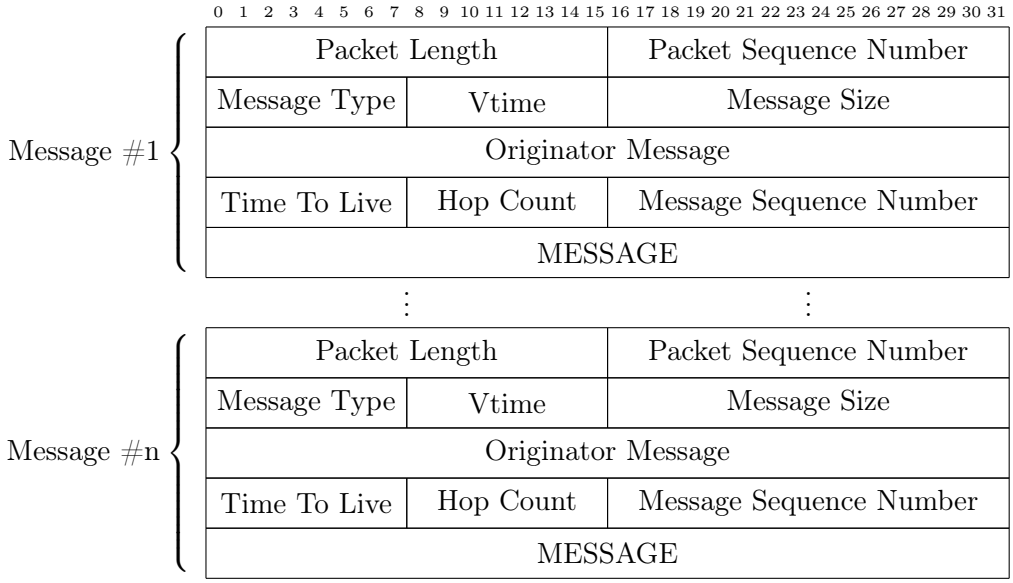

Abbildung 2.2.: OLSR Datagram

Die im folgenden beschriebenen *OLSR-Messages* sind für die Basis-Funktionalität von OLSR zuständig. Sie werden im Feld **MESSAGE** (siehe Abbildung 2.2) eingebettet. Für jeden *Message-Type* können die Intervalle, in denen sie versendet werden, einzeln definiert werden. So werden z.B. *HELLO-Messages* üblicherweise alle fünf Sekunden an die direkten Nachbarn verschickt, während *TC-Messages* nur alle zehn Sekunden in das gesamte Mesh geflutet werden:

- **HELLO** Messages zum Finden von Nachbar-Knoten
  In einer HELLO Messages wird neben der eigenen Absender-Adresse auch die Information über alle direkten Nachbarn (Neighbour-State) und deren Verbindungsqualität (Link-State) verpackt. Durch diese Informationen können neue Knoten schnell und effizient eine Sammlung der Nodes in der näheren Umgebung aufbauen. Über die Anzahl der in einem Zeitintervall empfangenen HELLO Nachrichten kann ein Knoten den Verlust von Paketen eines bestimmten Nachbarn in Empfangsrichtung abschätzen und die Neigbour-Link-Quality (NLQ) berechnen.

- **Topology-Control** (TC) Messages zum Synchronisieren der Topologie:
  Alle Nodes sollen möglichst das selbe Abbild des Netzwerkes halten. Dazu ist ein ständiger Austausch über Veränderungen der Topologie nötig. Damit neue Knoten auch schnell die gesamte Topologie erfahren, wird in regelmäßigen Intervallen das Netzwerk mit der kompletten Topologie via Broadcast geflutet.

- **Multiple Interface Declaration (MID)** Messages enthalten die Adressen aller weiteren Netzwerk-Schnittstellen eines Knotens.

2.4.3. B.A.T.M.A.N

Better Approach To Mobile Adhoc Networking (B.A.T.M.A.N.)[8] ist ein Distanz-Vektor Routing-Protokoll. Einzelne Nodes kennen zwar jeden anderen Knoten, jedoch nicht die gesamte Topologie und somit auch nicht die komplette Route zu einem Ziel. Die Topologie-Information liegt als verteilte Information dezentral im Mesh-Netzwerk über alle Knoten verstreut.

BATMAN wurde ursprünglich von der Berliner Freifunk-Community als Alternative zu OLSR entwickelt. Es sollte die Probleme mit dynamisch wechselnden Gateways lösen, indem ein Knoten mit Internet-Uplink durch den Benutzer statisch ausgewählt werden kann. Zudem sollten die Datenmengen, welche zum Betrieb eines DRP nötig sind, minimiert werden.

Für die erste Version von BATMAN wurde der übliche Ansatz des IP-Routings gewählt. Für die Implementierung des Routing-Algorithmus wurde ein Userspace Daemon `batmand` zum Verwalten der Routing-Tabelle entwickelt.

[8] Zur besseren Lesbarkeit wird in der restlichen Arbeit die Schreibweise BATMAN verwendet

Während der Entwicklung von `batmand` hat sich ein Branch namens *BatMan-eXperimental* (BMX) gebildet. Ursprünglich sollte dieser Branch wieder mit `batmand` zusammen geführt werden, jedoch hat sich die Entwicklung so stark in Richtung natives IPv6-Routing bewegt, das ein Merge unrealistisch wurde. Der Fork *BMX6* ist entstanden und ist bis heute eines der wenigen IPv6-Only DRP für Wireless-Mesh-Networks. BMX6 findet mit über 10000 Knoten starke Verbreitung im spanischen CN *guifi.net*.

Seit Anfang 2010 konzentriert sich die Weiterentwicklung von BATMAN auf Routing im MAC-Layer. Dar dabei entstandene Entwicklungszweig ist eigenständig und wird unter dem Namen *BATMAN-advanced* (batman-adv) geführt. Seit Kernel Version 2.6.38 ist batman-adv als Kernel-Module ein Teil des offiziellen Linux-Kernels. Ein Mesh auf Basis von batman-adv bildet eine große gemeinsame Ethernet Broadcast-Domain. Die dadurch entstehenden Besonderheiten werden von Daniele Furlan in [Fur11, Kapitel 2] aufgezeigt:

- **Transparenz**: Wegen der gemeinsamen Broadcast-Domain sind Layer-3 Protokolle unwissend über die darunterliegende Multi-Hop Mesh-Topologie. Jegliches Layer-3 Protokoll kann verwendet werden. Dies vereinfacht z.B. die Adressvergabe sowohl in IPv4 (DHCP) als auch in IPv6 (NDP) erheblich. In IPv6 Netzwerken können einem Netzwerk-Adapter mehrere IP-Adressen zugewiesen werden. Sogar IPv6-Präfixe von unterschiedlichen Gateways sind vorgesehen, womit eine dezentrale Adress-Vergabe umgesetzt werden kann.

- **Multiple Network-Interfaces**: Knoten mit mehreren Netzwerk-Schnittstellen können über unterschiedliche Pfade kommunizieren. Dadurch lassen sich nicht nur getrennte Netzwerk-Segmente überbrücken, sondern auch im Falle von mehreren unabhängigen Pfaden zwischen Knoten die Übertragungsraten erhöhen. Datenpakete können abwechselnd über verschiedene Interfaces übertragen werden. Falls mehrere Wireless-Interfaces mit getrennten, nicht überlappenden Funk-Kanälen eingesetzt werden, verwendet batman-adv automatisch getrennte Kanäle für eingehende und ausgehende Übertragungen. Somit wird eine für Wireless unübliche Full-Duplex Verbindung möglich.

- **Adressierung**: Anders als bei einem Table-Driven DRP, bei dem sich die Einträge von Routing-Tabellen hierarchisch ordnen und vor allem komprimieren lassen, werden bei batman-adv die Hardware MAC-Adressen einzelner Knoten als Identifikator verwendet. IP-Adressen können z.B. nach geographischen Aspekten vergeben werden und die dadurch entstehenden Subnetze lassen sich in kleinere Einträge zusammenfassen. Eine Ansammlung von Knoten kann somit bei einer Identifikation von IP-Adressen komprimiert abgelegt werden. MAC-Adressen lassen sich höchstens nach Hersteller-Präfixen sortieren und komprimieren. Dies macht jedoch in einer Mesh-Topologie keinen Sinn.

- **Fragmentierung**: Alle Datenpakete in einem batman-adv Netzwerk werden um

einen eigenen *Ethernet Frame Header* ergänzt. Die versendeten Pakete sind mindestens 32 Byte größer als übliche Pakte mit einem *IEEE 802.3* Ethernet Frame. Ein batman-adv Paket kann somit nicht mit der üblichen *Maximum transmission unit* (MTU) von 1500 Byte übertragen werden und muss fragmentiert werden, sofern die MTU auf Netzwerkschnittstellen nicht speziell angepasst wird. Fragmentierung wirkt sich negativ auf Performance aus. Bei vielen Netzwerkadaptern, die in handelsüblichen Wireless-Routern verbaut werden, ist eine Veränderung der MTU nicht möglich.

Die in batman-adv verwendete Metrik wird *Transmit Quality* (TQ) genannt. Sie bezieht sich – ähnlich wie ETX – auf die Wahrscheinlichkeit einer erfolgreichen Datenübertragung. Jeder Knoten sendet spezielle Broadcast-Pakete an alle direkten Nachbarn. Diese werden von einem empfangenden Knoten wieder an alle direkten Nachbarn weiter gereicht.
Für die Qualität der Empfangsrichtung wird die Anzahl der erhaltenen Pakete in einem Zeitfenster berechnet und mit der bei einer verlustfreien Verbindung zu erwartenden Anzahl von Paketen verglichen. Ein Wert für die *Receiving Qualtiy* (RQ) wird erzeugt.

Um die Qualität der Senderichtung zu erfassen, wird ein Wert *Echo Quality* (EQ) benutzt. EQ berechnet sich aus den empfangen *eigenen* Paketen, die von einem Nachbar-Knoten zurück gesendet werden.

> "The transmit quality is the probability of a packet sent from Node A to be received correctly at Node B. Since the EQ is the probability of a packet being delivered correctly both from Node A to Node B and back from Node B to Node A, the TQ can be derived using RQ." [HLP11]

$$EQ = RQ \cdot TQ \Longrightarrow TQ = \frac{EQ}{RQ}$$

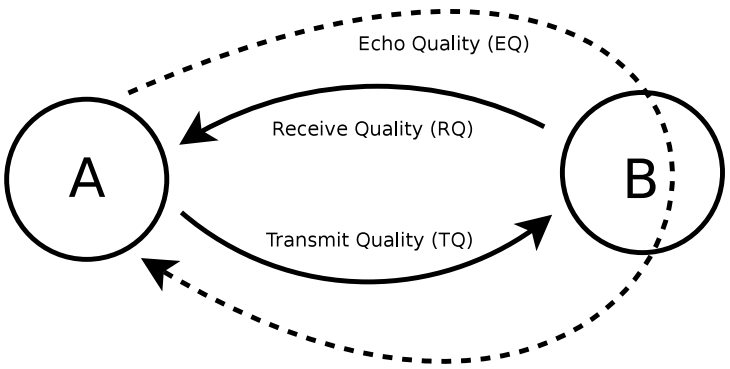

Abbildung 2.3.: Transmit Quality in BATMAN-ADV

Die TQ-Metrik verwendet ganzzahlige Werte im Bereich $0 - 255$. Eine TQ von 255 be-

schreibt einen idealen verlustfreien Link. Für Verbindungen über mehrere Hops wird je Zwischenknoten eine *Hop Penalty* abgezogen. Zusätzlich fließt in TQ auch die Information über unterschiedliche Netzwerk-Schnittstellen ein. Wird ein EQ von einem Interface empfangen, welches das Paket nicht ursprünglich versendet hat, kann batman-adv von unterschiedlichen Pfaden für die Empfangs- und Senderichtung ausgehen. Weil der Verlust von Bandbreite je Hop bei solchen *Full-Duplex* Verbindungen wesentlich geringer ist, wird dabei die Hop Penalty nicht angewendet.

Um batman-adv zu verwenden, müssen im Mesh liegenden Netzwerk-Schnittstellen – meist WLAN-Karten im Ad-Hoc Modus – als *transport interface* definiert werden. Diesen Schnittstellen muss keine Netzwerk-Adresse zugewiesen werden, weil darüber nur Ethernet Frames mit der Erweiterung von BATMAN versendet werden. Schnittstellen auf Knoten ohne batman-adv Kernel-Module können diese nicht interpretieren.

Sobald ein *transport interface* definiert ist, wird automatisch vom Kernel ein neues virtuelles Interface `bat0` angelegt. Diesem Interface lassen sich Netzwerk-Adressen zuweisen. Da das bat0-Interface auch eine MAC-Adresse hat, kann es als Teilnehmer einer virtuellen *Bridge* konfiguriert werden. Bridges werden unter Linux verwendet um physikalisch getrennte LAN-Segmente zu verbinden. Ein weiterer Teilnehmer solch einer Bridge kann z.B. ein LAN-Interface sein, womit die daran angeschlossenen Geräte auch direkt mit dem Mesh verbunden sind.

2.5. Übertragungsarten in Wireless Networks

Ein Funknetzwerk ist ein geteiltes Medium. Es kann zu einem Zeitpunkt immer nur einen Sender geben. Die gängigsten WLAN-Frequenzen liegen im 2.4GHz (IEEE 802.11b/g/n) und 5GHz (IEEE 802.11a/n) Spektrum, den sog. *Industrial, Scientific and Medical* (ISM) Bändern. Bei diese Frequenzbereichen ist keine spezielle Lizensierung für die Nutzung nötig, weshalb diverse andere drahtlose Geräte sich die selben Frequenzen mit WLAN Funknetzwerken teilen. Neben den in vielen Haushalten verbauten privaten Heim-Routern "funken" z.B. auch Mikrowellenherde und drahtlose Garagentor-Öffner. Entsprechend existieren im Medium häufig auch andere Sender, welche durch ihr Rauschen als Störquelle fungieren.

Im 802.11 Standard ist *RTS/CTS* (Request to Send / Clear to Send) – auch bekannt als *"virtual carrier sensing"* – als optionales Verfahren zum Reduzieren von kollidierenden Übertragungen spezifiziert. Für große Mesh-Netzwerke ist dieses Konzept jedoch unbrauchbar, da die Antwort "Clear to Send" auf die Frage "Request to Send" kaskadiert und sehr viel Zeit in Anspruch nehmen kann.

2.5.1. Unicast

Für jede direkte Verbindung zwischen zwei Knoten und unabhängig von Ad-Hoc oder Infrastruktur-Modus, wird Unicast Datenverkehr mit einer möglichst sinnvollen Übertragungsrate (Bitrate) ausgetauscht. Die Bitrate wird durch unterschiedliche Verfahren in Abhängigkeit von der Verbindungsqualität dynamisch festgelegt. Dieses Verfahren des dynamischen Bandbreiten Wechsels wird "Rate Adoption" genannt.

> "High data rates transmit data faster than low data rates, however high data rates are more susceptible to bit errors. This means more packets are lost on low quality wireless channels with high bit error rates (BERs). Low data rates take longer to transmit packets over the link, however they are more resistant to bit errors and are more likely to be successfully transmitted over a poor quality wireless link."[XHF13]

Bei sehr vielen WLAN-Chipsätzen wird der sog. *"Minstrel rate adoption algorithm"* eingesetzt um Lösungen für dieses Optimierungsproblem zu finden. Je nach Betriebssystem und WLAN-Treiber können die Informationen über Verbindungsgeschwindigkeiten auch über definierte Schnittstellen extrahiert werden um diese in die Metrik eines Routing-Protokolls einfließen zu lassen.

In Multi-Hop Netzwerken, in denen jeder Knoten nur mit eine Schnittstelle für Empfang und Versand ausgestattet ist, halbiert sich die Bandbreite mit jedem Hop bestenfalls.

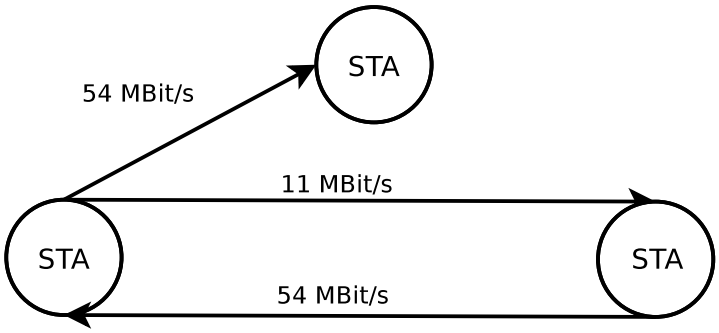

Abbildung 2.4.: Unicast Übertragung

2.5.2. Broadcast

IP-Pakete, die für alle Knoten eines Netzwerks bestimmt sind, werden üblicherweise als Broadcast versendet. Dabei wird als Empfänger-Adresse eine spezielle Broadcast-Adresse verwendet. Ein Broadcast-Paket wird gleichzeitig an alle direkten Nachbarn übermittelt und von diesen als Kopie an alle weiteren Knoten erneut übertragen. Der ursprüngliche Sender erhält von den Empfängern keine Benachrichtigung über den erfolgreichen Erhalt der Nachricht.

Besonders in drahtlosen Netzwerken ist die Verlustrate aufgrund von Packet-Loss hoch. Um sicherzustellen, dass möglichst alle direkten Nachbarn das Broadcast-Paket erhalten, wird als Übertragungsrate die des langsamsten Nachbarn verwendet, oder sogar eine statisch konfigurierte Basis-Rate eingesetzt. Gleichzeitig ermöglicht eine statische Broadcast-Rate auch eine Zugangsbeschränkung für langsame bzw. meist ältere Geräte. 802.11 verwendet *Beacon frames* auf dem MAC-Layer zur Koordination der einzelnen Geräte und u.a. auch für die Übermittlung des Namens des drahtlosen Netzwerks und ggf. kryptographische Vorgaben. Diese Beacons werden als Broadcast versendet. Wenn z.B. als Übertragungsrate für Broadcasts 22 MBit/s konfiguriert wird, können Geräte, die nur die 11MBit/s von 802.11b unterstützen, nicht am Netzwerk teilnehmen.

Während Daten mit einer langsamen Bitrate übertragen werden, müssen schnelle Unicast-Verbindungen aussetzen. Broadcasts kosten deshalb sehr viel Airtime und mindern dementsprechend die gesamte Performance eines Netzwerks. Besonders in Mesh-Topologien mit vielen Knoten und einem Routing-Protokoll, welches intensiv Broadcast-Nachrichten zur Koordination verwendet, leidet der durchschnittliche Daten-Durchsatz während jedem Broadcast.

Durch das wiederholte Versenden kann ein sog. *Broadcast-Storm* auftreten, wenn jeder Knoten jedes Broadcast-Paket blind kopiert und versendet. Abhilfe kann das Setzen eines sinnvollen Wertes für die *"Time to Live"* (TTL) schaffen. Pakete werden dann nach einer gewissen Anzahl von wiederholten Übertragungen bzw. Zahl von Hops nicht mehr weitergeleitet.

Für die möglichst effiziente und zuverlässige Übertragung von Broadcast-Paketen in Wireless-Mesh-Networks gibt es diverse Ansätze die im akademischen Bereich entwickelt werden. Dabei werden meist unterschiedliche Schwerpunkte gesetzt: In den Arbeiten von Stojemnovic und Nayak [SN06] sowie Yang et. al. [YLL09] soll die Zuverlässigkeit von Broadcasts erhöht werden, wohingegen Lee et. al. [LCS05] die verschwendete Airtime und Bandbreite aufgrund von niedrigen Bitraten und Re-Transmitts durch dedizierte Broadcast-Kanäle optimiert.

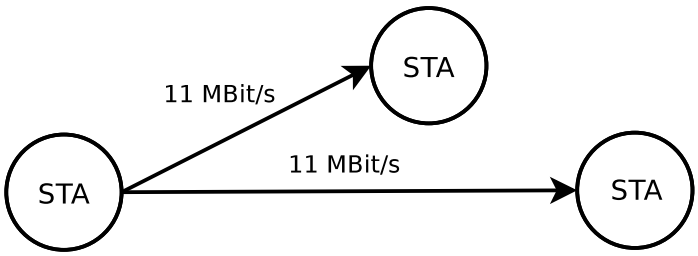

Abbildung 2.5.: Broadcast Übertragung

2.5.3. Multicast

Übertragungen per Multicast verhalten sich ähnlich wie Broadcasts, nur dass auf IP-Ebene eine bestimmte Empfängergruppe (Multicast-Group) definiert werden kann. Im MAC-Layer wird der selbe Frame, für jeden Pfad in dem sich ein Mitglied der Multicast-Gruppe befindet, kopiert und weiter versendet. Ein Multicast-Paket ist ebenso wie ein Broadcast *stateless* und wird vom Empfänger nicht bestätigt.

Tanenbaum beschreibt im Grundlagenwerk "Computer Networks" den Unterschied zwischen Multicast und Broadcast wie folgt:

> "Group addresses allow multiple stations to listen to a single address. When a frame is sent to a group address, all the stations in the group receive it. Sending to a group of stations is called **multicasting**. The special address consisting of all 1 bits is reserved for **broadcasting**. A frame containing all 1s in the destination field is accepted by all stations on the network." [TW11]

In Ethernet-Netzwerken wird spezielle Switching-Hardware benötigt, die erkennt ob und an welche Ports ein Multicast-Paket versendet werden soll. Da solch ein Switch nicht nur auf MAC-Ebene (Layer-2) arbeitet, sondern auch einen Teil des IP-Headers analysieren muss, wird deswegen häufig von Layer-3 Hardware gesprochen.

In drahtlosen Netzwerken wird meist nicht zwischen Broadcast und Multicast unterschieden, da ein Multicast-Paket ohnehin wie ein Broadcast mit einer niedrigen Datenrate versendet wird. Dennoch werden einige neue Ansätze für Multicast in WLAN im wissenschaftlichen Umfeld diskutiert, um *Quality Of Service* und Rate Adoption in sinnvolle Koexistenz zu bringen. [CKA08] [**?**]

Es sei an dieser Stelle angemerkt, dass Broadcasts in IPv6 nicht existieren. Der RFC 4291 zur Spezifikation von speziellen IPv6 Adressen definiert: "There are no broadcast addresses in IPv6, their function being superseded by multicast addresses." [HD06]

Die spezielle Link-Local Multicast-Adresse `FF02::1` definiert *"all nodes"* und ist somit äquivalent zur Broadcast-Adresse `255.255.255.255` womit sich alle Knoten im lokalen Netzwerk erreichen lassen.

3. Positionsdaten in Mesh-Networks

Für die Darstellung der Positionen von Mesh-Nodes können unterschiedliche Gründe vorliegen. Je nach Anwendungsfall bieten sich verschiedene Arten der Übertragung, Darstellung und ggf. auch Kombinationen mit anderen Daten an.

Im einfachsten Fall sendet jeder Knoten die eigene Position in Form von GPS-Koordinaten an einen zentralen Server. Dies wird meist dann praktiziert, wenn der empfangende Server auch gleichzeitig die Positionen in eine Karte einzeichnet und diese fertig gerendert an Benutzer ausliefert. Die Intention liegt dabei in der Darstellung der örtlichen Verteilung von Knoten um z.B. Nutzern die möglichen *Zugangspunkte* des Netzwerkes aufzuzeigen.

Für Betreiber von Mesh-Netzwerken sind jedoch neben der Positionen der Knoten auch die Topologie sowie die Werte der Metrik der einzelnen Verbindung zwischen den Knoten wichtige Informationen. Diese Daten zu erheben ist i.d.R. eine Funktionalität des eingesetzten Routing-Protokolls. Die Darstellung der Kombination aus Topologie, Verbindungsqualität und Positionsdaten auf einer Karte wird zur *Visualisierung* des Mesh-Netzwerkes verwendet. Zusätzlich ist eine visuelle Repräsentation der Anzahl von Nutzern an Standorten für die weitere Planung der Infrastruktur interessant.

Heutzutage sind viele mobile WLAN-Geräten mit GPS-Empfängern ausgestattet. Wenn solche Geräte – z.B. Smartphones – Teilnehmer eines MANET sind, wird durch kontinuierliches Übertragen und Aufzeichnen der Positionen und deren Zeitstempel die Wiedergabe von Bewegungsdaten möglich. Das Verfolgen von bewegten Objekten wird im allgemeinen *Tracking* genannt.

Je nach Anwendungsszenario sind Latenzen beim Übertragen von Positionsdaten mehr oder weniger kritisch. Genauso handhabt es sich mit der Zuverlässigkeit bei der Übermittlung der Nachrichten.
Für *Tracking* werden meist vollständige Echtzeit-Daten bevorzugt, um eine möglichst genau Zeit-Weg Darstellung zu erhalten. Das vereinzelte Ausbleiben von Positionen führt zu Ungenauigkeiten.
Beim *Visualisieren* einer Netztopologie werden zwar meist statische Koordinaten verwendet, jedoch sind die Metriken sehr variabel. Eine hohe Latenz führt zu verfälschten Angaben der Verbindungsqualitäten, was wiederum bei Wartungsarbeiten unpraktisch ist.

Für eine zentralistische Darstellung, z.B. mit einer Karte auf der öffentlichen Website des Netzwerkes, ist es nicht notwendig die Positionsdaten an alle Knoten zu verteilen. Einzelne Übertragungen von den jeweiligen Nodes zum zentralen Server per Unicast

bieten sich an.
Wenn aber alle Knoten gleichermaßen das Wissen über die Positionen aller anderen Nodes teilen sollen, werden Unicast-Übertragungen unrentabel. Jeder Knoten müsste mit jedem anderen Konten die Positionen austauschen, was zu einer sehr hohen Anzahl an Verbindungen führen würde. Solch ein Anwendungsszenario wird typischerweise mittels *Broadcasts-Flooding* gelöst: Jeder Knoten versendet seine eigene Position an alle anderen Nodes mit einem Broadcast-Paket. Die Schwierigkeit dabei ist jedoch sicherzustellen, dass alle Knoten die Information erhalten.

Im folgenden werden für die Routing-Protokolle OLSR und batman-adv die nativen Möglichkeiten zum Verteilen von Positionsdaten aufgezeigt.

3.1. OLSR Plugins

Der von Andreas Tønnesen im Rahmen seiner Master-Thesis[Tø04] entwickelte *olsrd* – die Implementierung von OLSR bzw. RFC 3626 – ermöglicht durch eine modulare Architektur die Verwendung von Plugins. Die dazu bereitgestellten Schnittstellen erlauben die Veränderung aller Kern-Komponenten und Datenpakete von OLSR, aber auch die Ergänzung von neuen Funktionen.

Plugins bieten sich gut als Schnittstelle für andere Anwendungen mittels inter-process communication (IPC) an, da dabei die Kernfunktionalität von OLSR nicht verändert werden muss. Für asynchrone Aufgaben, wie z.B. das Schreiben in Dateien, ist es angebracht ein Plugin zu verwenden um sicherzustellen, dass solch ein Vorgang nicht das restliche Verhalten von OLSR blockiert.

Abbildung 3.1 zeigt, wie Plugins in den Datenfluss von OLSR integriert werden können, um eine Manipulation von OLSR-Nachrichten zu ermöglichen.

Wenn OLSR nach RFC mit zu selektierenden MPR-Knoten betrieben wird und als Metrik nicht ETX sondern Minimal Hop Count zum Einsatz kommt, werden Nachrichten von Plugins effizient im Mesh geflutet.

In den meisten WCNs ist jeder Knoten ein MPR. Damit entfällt der mit ETX inkompatible Algorithmus zum Selektieren von MPR-Knoten. Für die Verwendung von Plugins bedeutet das einen Traffic-Overhead, da die selben Nachrichten unter Umständen mehrfach von einem Knoten empfangen werden. Gleichzeitig wird aber sichergestellt, das jede OLSR-Nachricht, egal ob diese einem Knoten durch ein registriertes Plugin bekannt ist, an alle anderen Knoten weitergereicht wird. Abbildung 3.2 illustriert das Verhalten beim Fluten von OLSR-Nachrichten mit und ohne MPR-Selektion.

Mit dem Registrieren eines Plugins für den jeweiligen Message-Type lässt sich das Verhalten beim Fluten dennoch gezielt steuern:

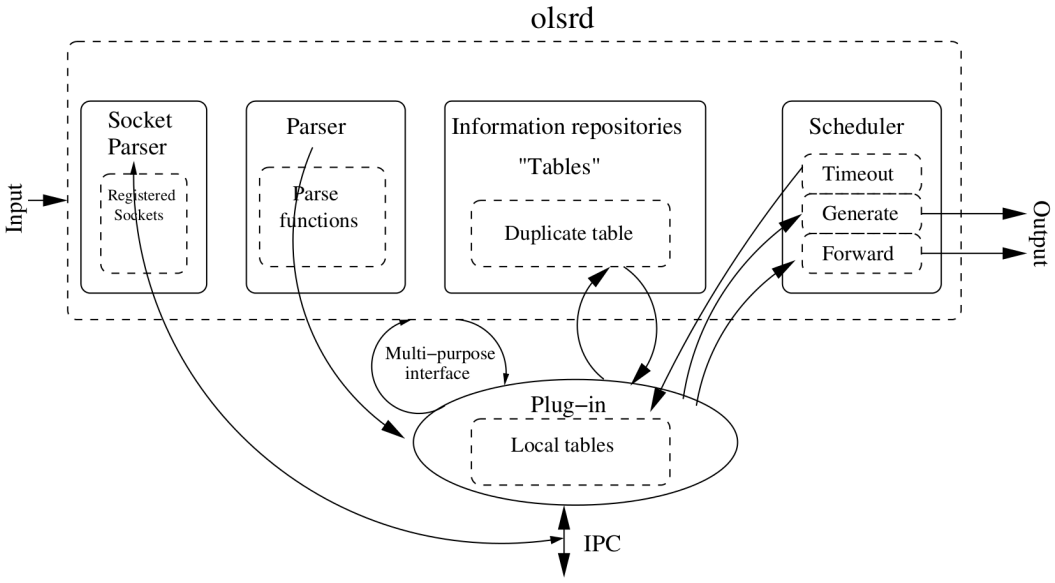

Abbildung 3.1.: OLSR Plugin Architektur (Quelle: [Tø04])

Nachrichten können beim Weiterreichen nicht nur um z.B. eigene Inhalte ergänzt werden, sondern das Konzept vom effizienten Fluten durch MPRs kann im jeweiligen Plugin implementiert werden. Der Aufwand der Entwicklung von Plugins wird dadurch zwar größer, jedoch ist dank der Architektur von OLSR die Möglichkeit gegeben, über Umwege für einen effizienten Datenaustausch unter Plugins zu sorgen.

3.1.1. Position Update Distribution

Das *Position Update Distribution* (PUD) Plugin für OLSR wird von Ferry Huberts und Teco Boot seit 2010 für die niederländischen Streitkräfte entwickelt, um z.B. Positionen von Einsatzfahrzeugen mit einer Leitstelle auszutauschen. Der Sourcecode des Plugins ist über das öffentlich zugängliche GIT-Repository[1] von OLSR frei verfügbar.

PUD arbeitet mit GPS-Daten von einem eingebauten GPS-Empfänger und überträgt deshalb Positionsdaten im NMEA-2000 Format. Der grundlegende Ansatz ist, ständig aktuelle Daten des GPS-Empfängers an alle anderen Knoten zu übermitteln. Somit ist es mit PUD möglich, zu jedem Knoten auch Bewegungsdaten zu erfassen. Die Architektur von PUD geht über die eines einfachen OLSR-Plugins hinaus. Es ist vorgesehen, auch an einen OLSR-Knoten angebundene Geräte in einem separaten und nicht OLSR-betriebenen LAN-Segmente via Multicast mit GPS-Daten zu versorgen. Damit lassen sich z.B. in einem Fahrzeug die Positionsdaten, sowie die Richtung und Geschwindigkeit an andere Geräte ohne OLSR verteilen und gleichzeitig an weitere Fahrzeuge in

[1] `http://olsr.org/git/`

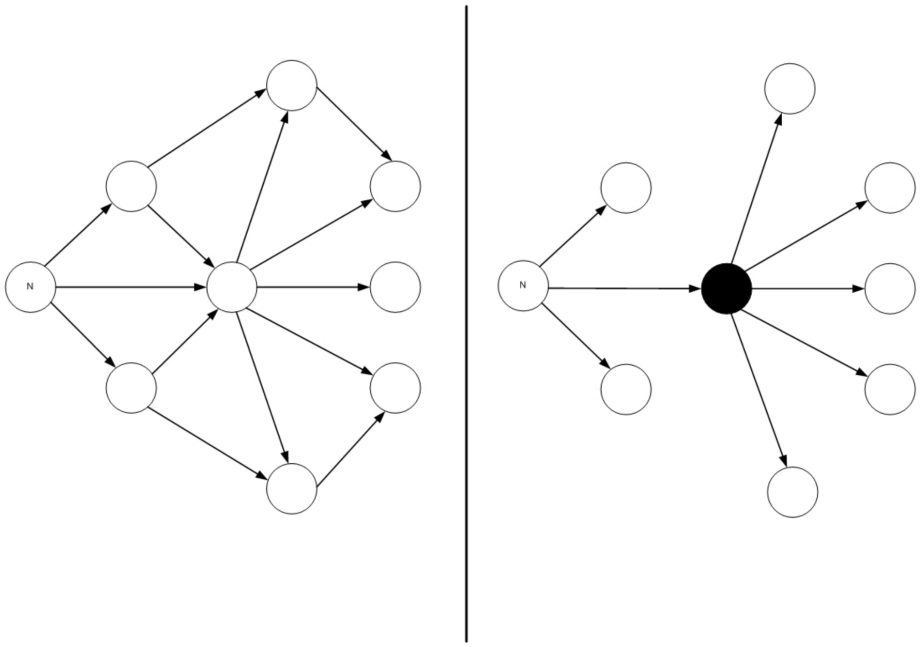

Abbildung 3.2.: Unterschied zwischen Fluten ohne MPR und mit MPR (Quelle: [Spi08])

WLAN-Reichweite per OLSR übertragen.

Des Weiteren können über einen oder mehrere sog. *Relay-Server* allen Knoten die mit PUD generierten GPS-Daten aus einem OLSR-Mesh heraus über anderweitige Übertragungswege (z.B. Internet) in ein anderes OLSR-Mesh hinein übertragen werden. Die einzelnen Nachrichten werden vom Relay-Server gesammelt und komprimiert zu einem entfernten Relay-Server übermittelt. Um den idealen Relay-Server – im Kontext der Topologie – zu wählen, bedient sich PUD einer zusätzlichen OLSR-Erweiterung, dem sog. *Smart-Gatway* Plugin. Knoten mit diesem Plugin verkünden ins Mesh die Möglichkeit, Daten in ein entferntes Netz zu routen. Damit verknüpft wird die maximale Bandbreite in solch ein Netz. Anhand der Informationen über das entfernte Netz, die topologische Position – also Anzahl der dazwischen liegenden Hops – und der verfügbaren Bandbreite, kann ein Knoten mit PUD eine Metrik für den idealen Smart-Gateway ermitteln.

PUD bietet sich für das *Tracking* von Knoten in einem OLSR-Mesh an.

Die Architektur bringt das Übertragen von Positionsdaten über Grenzen von Mesh-Netzwerken hinaus mit, wodurch grundsätzlich auch die Darstellung von *Zugangspunkten* auf einer Karte möglich ist. Für statische Knoten ist jedoch der Kosten-Aufwand von GPS-Empfängern i.d.R. zu hoch.

3.1.2. Nameservice

Das Nameservice Plugin wurde als eine der ersten Erweiterungen für OLSR entwickelt, um eine dezentrale Namensauflösung von Knoten im Mesh bereitzustellen. Das Plugin ist simpel aufgebaut, einfach zu konfigurieren und weit verbreitet. Weitere Funktionalitäten, die über einfache Namensauflösung hinaus gehen, wurden integriert. Es lassen sich inzwischen neben DNS-Einträgen im Format einer `/etc/hosts` Datei auch Dienste in Form von URLs verteilen.

Die zeitliche Gültigkeit der Informationen wird durch das Setzen der Option `timeout` definiert. Es ist jedoch nicht möglich, für unterschiedliche Themen eigene zeitliche Werte zu definieren. Da sich der Hostname eines Knotens i.d.R. nicht ändert, ist dieser Wert per default auf 30 Minuten gesetzt.

Für die Angabe von Positionsdaten sind nur Längen- und Breitengrade vorgesehen. Eine Höheninformation kann mit diesem Plugin derzeit nicht übertragen werden. Ein Hardware GPS-Empfänger lässt sich über einen Umweg integrieren. Wenn die Ausgabe von Längen- und Breitengrad des Empfängers als Komma-getrennte Werte in eine Textdatei geschrieben werden, kann das Nameservice-Plugin mit der Option `"latlon-infile"` dazu veranlasst werden, diese Daten zu verwenden.

Die Dokumentation zur Konfiguration des Plugins bietet keinen Hinweis darauf, in welchem Intervall die aktuellen GPS-Daten eingelesen werden. Der Quellcode deutet jedoch darauf hin, dass direkt vor jedem Senden von Daten über OLSR die `"latlon-infile"` neu geladen wird.

Das Plugin bereitet gesammelte Daten in nach Themen getrennten Dateien auf, welche von externen Anwendungen gelesen werden können. Die Positionsdaten werden automatisch mit Topologie-Daten verknüpft. Eine Darstellung der reinen GPS-Koordinaten aller anderen Knoten liegt nicht vor. Listing 3.1 zeigt eine typische Konfiguration für das Nameservice Plugin.

Listing 3.1: Nameservice Plugin Config

```
1  LoadPlugin "olsrd_nameservice.so.0.3"
2  {
3    PlParam "name" "mesh-node-one"
4    PlParam "lat" "52.507653"
5    PlParam "lon" "13.454175"
6    PlParam "services-file" "/var/etc/services.olsr"
7    PlParam "suffix" ".olsr"
8    PlParam "latlon-file" "/var/run/latlon.js"
9    PlParam "hosts-file" "/tmp/hosts/olsr"
10   PlParam "timeout" "600"
11 }
```

Das Nameservice Plugin ist ein nützliches Werkzeug in der Repräsentation einer OLSR-

Topologie und eignet sich somit sehr gut zur *Visualisierung* eines Mesh. Deshalb kommt es in den meisten OLSR-basierten WCNs zum Einsatz.

Tracking ist zwar grundsätzlich mit der Einbindung eines Hardware GPS-Empfängers möglich, jedoch fehlen Höheninformationen. Das gleichzeitige Übertragen von DNS-Informationen sollte aufgrund des Traffic-Overheads dabei deaktiviert werden.

3.2. BATMAN's ALFRED

Der *Almighty Lightweight Fact Remote Exchange Daemon* (A.L.F.R.E.D)[2] ist ein eigenständiger Dienst zum dezentralen Austausch von Informationen. Marek Lindner, einer der Entwickler von BATMAN und ALFRED, lässt sich zu ALFRED wie folgt zitieren:

> "alfred is a user space daemon to efficiently[tm] flood the network with useless data - like vis, weather data, network notes, etc" [3]

Ursprünglich wurde ALFRED entwickelt, um Teile der Visualisierung einer batman-adv Topologie aus dem Kernel-Modul heraus in eine eigene Anwendung zu kapseln. Der generische Ansatz von ALFRED ermöglicht den Austausch von beliebigen Informationen (Facts), bisher wurden jedoch nur `batadv-vis` zum Visualisieren von batman-adv und `alfred-gpsd` zum Übermitteln von Positionsdaten als eigenständige Applikationen implementiert.

Unter *Fact* wird in ALFRED ein Thema von Informationen verstanden. Zu einem Thema kann es zwar eine Vielzahl von Beiträgen geben, jedoch lässt sich über ALFRED immer nur die Sammlung aller Beiträge zu einem Fact im ganzen erfragen. Es gilt die Konvention, zu einem Thema bzw. Fact immer den selben Identifikator, ein Integer-Wert ≥ 64, zu verwenden. Werte ≤ 64 sind reserviert und lassen sich mit dem `alfred` Binary nicht verwenden. `batadv-vis` und `alfred-gpsd` verwenden z.B. solche Fact-IDs. An jeden Beitrag wird die MAC-Adresse des ursprünglichen Absenders geknüpft. Je MAC-Adresse kann nur ein Beitrag existieren. Die Verwendung dieser Adressen stammt aus dem Kontext von BATMAN, wo diese als Identifikator von Nodes verwendet wird.

Abbildung 3.3 stellt den Zusammenhang von Facts und Beiträgen als Entity-Relationship-Modell dar. Es sei angemerkt, dass eine selektive Anfrage zu einem Beitrag einer bestimmten MAC-Adresse nicht möglich ist.

Bei der Verwendung von ALFRED muss zwischen zwei Modi unterschieden werden:

Ein ALFRED *Master* dient als Datenspeicher beliebiger Facts. Master synchronisieren in einem regelmäßigen 10 Sekunden Intervall ihre Daten mit anderen Mastern. Nach einem

[2] Zur besseren Lesbarkeit wird in der restlichen Arbeit die Schreibweise ALFRED verwendet
[3] Quelle: `http://www.open-mesh.org/projects/open-mesh/wiki/Alfred`

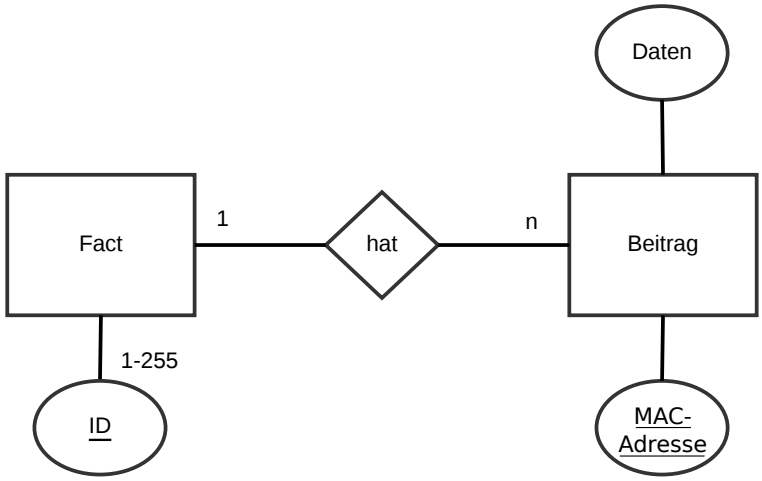

Abbildung 3.3.: ALFRED-Datenstruktur als ER-Diagramm

Zeitintervall von 600 Sekunden werden gespeicherte Daten wieder verworfen. Master dienen als redundante Kommunikationsschnittstelle für mehrere ALFRED Slaves.

Ein ALFRED *Slave* benötigt immer einen Master zum Informationsaustausch. Slaves können Daten zu einem Master schicken oder einen Master nach Daten fragen.

Zusätzlich wird neben *Master* und *Slave* noch zwischen *Server* und *Client* unterschieden:

Ein *Server* ist der Teil der Anwendung, welcher die gesamte Netzwerk-Kommunikation beherbergt. Je Konten muss immer ein Server-Prozess laufen, um lokalen *Clients* über einen UNIX-Socket eine Schnittstelle zur Kommunikation mit anderen ALFRED-Instanzen im Netzwerk zu ermöglichen. Ein Server kann Master oder Slave sein.

Ein *Client* wird i.d.R. im Kontext eines Facts betrieben. Der Client läuft lokal und benutzt den Server um Facts zu ergänzen oder zu erfragen. Je nach Implementierung des Clients ist eine Verarbeitung der Facts direkt möglich. Das `alfred` Binary kann je nach Parameter beim Aufruf als Server oder Client verwendet werden.

Die derzeitige Implementierung von ALFRED ist auf den Betrieb in einem batman-adv Mesh optimiert. Um unnötigen Datentransfer zwischen weit auseinander liegenden Knoten zu vermeiden, greift ALFRED auf die Metrik-Daten von batman-adv zurück, um den nächstgelegenen Master als Kommunikationspartner auszuwählen. Wenn keine Metrik vorliegt, wird ein Master willkürlich gewählt. Dies ist auch der Fall, wenn ein Slave über einen batman-adv Zugangspunkt verbunden ist.

3.2.1. alfred-gpsd

Zum dedizierten Übertragen von GPS-Koordinaten kann der im Sourcecode von `alfred` mitgelieferte `alfred-gpsd` verwendet werden. `alfred-gpsd` selbst ist wiederum in Server und Client im selben Binary aufgeteilt. Im Server-Modus werden regelmäßig die aktuellen GPS-Koordinaten über einen Socket zu einem lokalen `alfred` Prozess als Fact mit der ID 2 an andere ALFRED-Instanzen übertragen. Da im Binary von `alfred` Fact-IDs ≤ 64 reserviert sind und somit vom Benutzer nicht verwendet werden können, soll als Client für diesen Fact zwangsläufig `alfred-gpsd` benutzt werden.

Für die Beschaffung der GPS-Daten wird eine lokaler Prozess von `gpsd` benötigt. `gpsd` ist eine eigenständige Anwendung, die unter Linux gebräuchlich ist um von einem Hardware GPS-Empfänger Daten zu beziehen. Alternativ ist auch die Angabe von statischen Koordinaten möglich.

Wird `alfred-gpsd` im Client-Modus aufgerufen, wird eine Liste aller `alfred-gpsd` Instanzen im Mesh mit den zugehörigen Koordinaten ausgegeben. Für Nodes mit Hardware GPS-Empfänger werden zusätzlich zu den Koordinaten noch weitere Werte übermittelt, wie z.B. den Geräte-Pfad des GPS-Empfängers und Fehlertoleranzen der Daten.

Alle Daten, die über den `gpsd` vom GPS-Empfänger übermittelt werden, sind als *Key-Value* Paar aufgelistet. Es existiert keine Möglichkeit für den Benutzer ohne Eingriff im C-Quellcode das Datenformat zu ändern um z.B. unnötige Einträge zu entfernen.

Listing 3.2 zeigt die Ausgabe eines Node-Eintrags von `alfred-gpsd`. Die mit `ep` beginnenden Schlüsselwörter deuten den Fehler der jeweiligen Daten an. Die Uhrzeit ist die des GPS-Empfängers.

Listing 3.2: Ausgabe von `alfred-gpsd` mit GPS-Empfänger

```
1 { "source" : "f6:00:48:13:d3:1e", "tpv" :
2  {
3   "class":"TPV","tag":"RMC",
4   "device":"/dev/ttyACM0","mode":3,"time":"2013-10-01T10:43:20.000Z",
5   "ept":0.005,"lat":52.575485000,"lon":-1.339716667,"alt":122.500,
6   "epx":10.199,"epy":15.720,"epv":31.050,"track":0.0000,"speed":0.010,
7   "climb":0.000,"eps":31.44
8  }
9 }
```

Falls statische Koordinaten verwendet werden, wird die Datenstruktur um einige Einträge verkürzt. Die Quelle der übertragenen Uhrzeit ist damit auch *nicht* ein sehr genauer GPS-Empfänger, sondern ist je nach verwendetem Gerät ein interner Zeitgeber. Die Genauigkeit der Uhrzeit kann somit stark variieren.

Listing 3.3: Ausgabe von `alfred-gpsd` mit festen Koordinaten

```
{ "source" : "8e:4c:77:b3:65:b4", "tpv" :
 {
  "class":"TPV",
  "device":"command line","time":"2013-10-01T10:43:05.129Z",
  "lat":48.858222,"lon":2.2945,"alt":358.000000,"mode":3
 }
}
```

`alfred-gpsd` wurde von Andrew Lunn im Oktober 2013 zum Quellcode von ALFRED hinzugefügt[4]. Nach eigener Aussage findet `alfred-gpsd` Einsatz im Forschungsbereich Unmanned Ground Vehicle (UGV) als Werkzeug beim *Tracking* von ferngesteuerten und mit BATMAN vernetzten Fahrzeugen.

Um eine Karte von *Zugangspunkten* zu erzeugen bietet es sich bei der Verwendung von BATMAN-Nodes ohne GPS-Empfänger und mit statischen Koordinaten an, mit einem ALFRED-Master die Daten aller `alfred-gpsd` Knoten z.B. an einen zentralen Webserver zu senden.

In Kombination mit `batadv-vis` kann eine BATMAN-Topologie geographisch *visualisiert* werden.

3.2.2. batadv-vis

`batadv-vis` ist der erste eigenständige Client für AFLRED. Es wird dazu verwendet, alle relevanten Daten zu einem BATMAN-Mesh aus dem Kernel-Modul heraus zwischen BATMAN-Knoten auszutauschen. Da die exakten Topologie-Daten in BATMAN einzelnen Knoten nicht vollständig bekannt sind, sondern nur die Topologie der direkten Nachbarn vorliegen, dient `batadv-vis` der *Visualisierung* einer BATMAN-Topologie.

Genau wie bei `alfred-gpsd` wird die lokale Instanz eines AFLRED-Servers benötigt. Die Kommunikation dazu findet über den UNIX-Socket statt. Das Binary kann – ähnlich wie `alfred` und `alfred-gpsd` – über Parameter entweder als Server oder als Client aufgerufen werden:

Im Server-Modus werden in regelmäßigen Abständen die Topologie-Daten als ALFRED-Fact unter der ID 1 aktualisiert. Dieser Prozess sollte als Dienst gestartet werden und dauerhaft im Hintergrund aktiv sein.

Im Client-Modus können die Daten der anderen Knoten als ALFRED-Fact angefragt und ausgegeben werden. Für die Ausgabe sind drei Datenformate möglich:

- **DOT**
  Eine gebräuchliche Beschreibungssprache zum Visualisieren von gerichteten und

[4] `Quelle:https://lists.open-mesh.org/pipermail/b.a.t.m.a.n/2013-October/010630.html`

ungerichteten Graphen. Das unter Unix bekannte Werkzeug *GraphViz* bringt mehrere Anwendungen zum direkten Generieren von Bilddateien aus dem Datenformat **DOT** heraus mit.

- **json**
  Das native Ausgabe-Format von `alfred`. Einzelne Einträge sind in einer Art JSON-Notation dargestellt. Das Format ist jedoch nicht vollständig JSON-konform und kann deshalb nicht direkt mit üblichen JSON-Parsern verarbeitet werden.
- **jsondoc**
  Die gesamte Ausgabe ist ein vollständiges JSON-Dokument.

Als Bezeichner für die Knoten des Graphen werden die MAC-Adressen der BATMAN-Nodes verwendet. Für die Gewichtung der Kanten eines solchen Graphen kommt der Wert der LQ-Metrik zum Einsatz. Die Adressen von WLAN-Clients, die über einen Node angebundenen sind, werden als Blätter eines Knotens abgebildet.

3.2.3. ALFRED Skript

Für *OpenWRT*[5], dem meist genutzten Linux-Derivat für Wireless Router in Mesh Netzwerken, existiert ein angepasstes Software-Paket für ALFRED. Bei der Installation dieses Pakets wird ein *cronjob* angelegt, der im Intervall von 5 Minuten alle Skripte im Ordner `/etc/alfred` ausführt. Damit soll es ermöglicht werden, einfache Facts in ALFRED zu integrieren ohne den Aufwand eines eigenen in C geschriebenen Clients zu tätigen.

Als praktisches Beispiel wird ein LUA-Skript `bat-hosts.lua` mitgeliefert, welches den eigenen Hostname eines Knoten mitsamt aller lokalen MAC-Adressen als Fact mit der ID 64 ins Mesh verteilt. Die gesammelten Hostname/MAC-Adresse Paare werden in der Datei `/etc/bat-hosts` abgelegt. Diese kann von Anwendungen zum Auflösen der MAC-Adressen – ähnlich wie DNS und die Datei `/etc/hosts` – verwendet werden.

Im Kontext von Positionsdaten lassen sich mit einem einfachen kleinen Shell-Script z.B. statische Koordinaten als eigener Fact übertragen. Es kann eine eigene Datenstruktur gewählt werden, um weitere Informationen – wie z.B. die Verfügbarkeit von Internet an einem Standort – mit in die Nachrichten zu kombinieren.

Listing 3.4 zeigt beispielhaft solch ein simples Shell-Script. Die GPS-Koordinaten und die Verfügbarkeit des Internet-Zugangs werden unter dem Fact mit der ID 66 hinterlegt. Das im Skript verwendete Kommando `uci` ist Bestandteil von OpenWRT und dient dort als Konfigurationswerkzeug.

Es biete sich jedoch an, nicht direkt zusammenhängende Variablen in einzelne Facts aufzuteilen um unterschiedliche Zeitintervalle für Updates definieren zu können. So lassen sich z.B. die Positionsdaten alle 10 Minuten in einem Fact übertragen, während der Online-Status alle 60 Sekunden in einem anderen Fact aktualisiert wird.

[5] `https://openwrt.org/`

Listing 3.4: Minimalistes Shell-Script zum Verteilen von GPS-Koordinaten mit `alfred`

```
#!/bin/sh

ALFRED=`which alfred`
if [ -z $ALFRED ] ; then
  /usr/bin/logger -s -t alfred-script "alfred not found"
  exit 0
fi
if `/bin/ping -q -c1 8.8.8.8 >/dev/null 2>&1` ; then
  ONLINE="true"
else
  ONLINE="false"
fi

LON=`uci get system.@system[0].longitude 2>/dev/null` || exit 0
LAT=`uci get system.@system[0].latitude  2>/dev/null` || exit 0
ALT=`uci get system.@system[0].altitude  2>/dev/null` || ALT=0

echo "lat=$LAT lon=$LON alt=$ALT online=$ONLINE" | $ALFRED -s 66
```

Listing 3.5 soll veranschaulichen, wie einfach durch Leerzeichen getrennte Key-Value Paare mit `alfred` ausgetauscht werden können.

Listing 3.5: Ausgabe von `alfred` mit eigener Datenstruktur

```
# alfred -r 66
{ "f8:1a:67:a6:09:f6", "lat=52.5011 lon=13.4658 alt=0 online=true\x0a" },
```

3.3. Systeme im Vergleich

Je nach Anwendungsszenario und Protokoll existieren unterschiedliche Möglichkeiten mit Positionsdaten zu arbeiten. Da Tracking in Community-Netzwerken eine geringe Rolle spielt, kommt dort eher die Visualisierung der Topologie oder die Darstellung von Zugangspunkten zum Einsatz. Beide Systeme zum Tracking kommen aus dem Umfeld der militärischen Forschung und sind nur darauf optimiert.

Es wird auch deutlich, dass für OLSR spezielle Lösungen entwickelt wurden, und für BATMAN-ADV ein generischer Ansatz verfolgt wird. Gleichzeitig wird aber z.B. im Fall vom Nameservice-Plugin die Funktionalität überladen, womit jedoch nur ein Kompromiss erzeugt wird. Die Vermutung liegt nahe, dass die Entwicklung von Plugins für OLSR mit einem hohen Aufwand verbunden ist. Der generische Ansatz von ALFRED bietet eine einfache Möglichkeit, eigene kleine Anwendungen zur Verteilung von Informationen zu entwickeln, und das unabhängig von einer speziellen Programmiersprache.

Da `batadv-vis` schon ein fertiges System zum Visualisieren einer BATMAN-ADV Topologie darstellt, kann es auch als Baustein für eine erweiterte Visualisierung mit Positions-

daten verwendet werden. Als Referenz-Schlüssel dienen die MAC-Adressen der einzelnen BATMAN-Nodes.

Aus den gegebenen Anwendungsszenarien und den analysierten Systemen lässt sich eine Bewertungsmatrix erstellen. Tabelle 3.1 soll diese veranschaulichen, wobei "- -" für die negativste und "++" für die positivste Bewertung steht.

System	Zugangspunkte	Visualisierung	Tracking
PUD	- -	+	++
Nameservice	+	++	-
alfred-gpsd	+	++	++
alfred-script	++	++	- -

Tabelle 3.1.: Bewertungsmatrix

4. Analyse

Aufgrund der Möglichkeit, über ALFRED sowohl Mesh-Nodes als auch an Access-Points angebundene Endgeräte – wie z.B. Smartphones – mit Positionsdaten zu versorgen, wird im späteren Verlauf der Aufbau einer kompatiblen Android-Applikation aufgezeigt.

Android wird als Plattform verwendet, weil die Implementierung des Netzwerkprotokolls auf Basis von Multicast und IPv6 Link-Local durch das Betriebssystem sichergestellt ist.

Für die Entwicklung einer Android-Applikation, welche zur Referenz kompatibel sein soll, muss zunächst das benötigte Netzwerk Protokoll analysiert werden. In der Dokumentation[1] der Architektur von ALFRED wird u.a. die Kommunikation im Netzwerk erklärt. Da dort jedoch nicht konkret auf das reine Netzwerk-Protokoll eingegangen wird und die Spezifikationen fehlen, welche nötig sind, um einen kompatiblen ALFRED-Server zu implementieren, soll im folgenden ein *Vorschlag* für die Spezifikation des ALFRED-Protokolls entwickelt werden.

4.1. System-Analyse

Für die Analyse der Netzwerk-Kommunikation wird die Dokumentation der Architektur, der Source-Code von `alfred` und ein eigen-entwickelter *Dissector* für das Netzwerk-Analyse Werkzeug Wireshark[2] verwendet. Der Dissector ist ein in Lua geschriebenes Modul für Wireshark.

Da die Dokumentation erst auf Nachfrage vom Entwickler-Team hinter ALFRED im Laufe dieser Arbeit angefertigt und bereitgestellt wurde, hat sich die anfängliche Analyse des Netzwerk-Protokolls zunächst auf die Implementierung des Dissectors beschränkt. In der C-Header Datei `packet.h` im Source-Code von ALFRED ist der grundlegende Aufbau für alle von ALFRED verwendeten Datenpakete definiert. Anhand dieser Struktur konnten Regeln für das Sezieren von Datenpaketen in Wireshark definiert werden.

Durch die korrekte graphische Darstellung des Datenaustauschs mehrerer ALFRED-Server in Wireshark liegt ein Werkzeug zum Testen des Netzwerk-Protokolls und dessen Spezifikation vor. Es werden nicht nur die einzelnen Segmente von Datenpaketen sinngemäß aufbereitet und präsentiert, sondern auch der zeitliche Verlauf der Kommunikation in Zusammenhang gesetzt. Anhand der im Dissector definierten Regeln lässt sich das

[1] `http://www.open-mesh.org/projects/batman-adv/wiki/Alfred_architecture`
[2] `http://www.wireshark.org/`

Netzwerk-Protokoll ableiten.

Der Quellcode[3] für den im Rahmen dieser Arbeit entstandenen Dissector steht unter der Open-Source Lizenz GPL und ist in der Architektur-Dokumentation von AFLRED verlinkt.

4.1.1. Referenz-Implementierung

Wie auch in der Dokumentation zur Referenz-Implementierung wird zunächst eine Terminologie für die Teilbereiche von ALFRED aufgestellt:

- Fact
  Ein Beitrag oder eine Sammlung von Beiträgen zu einem Thema. Die eigentlichen Daten.
- Server
  Eine Instanz von ALFRED. Kommuniziert mit anderen ALFRED-Instanzen im Netzwerk. Dient als Schnittstelle für lokale Clients.
- Client
  Eine Anwendung, die lokal über einen ALFRED-Server Facts anfragt oder bereit stellt.
- Master
  Ein ALFRED-Server, der Facts speichert und mit anderen Mastern synchronisiert. Akzeptiert Daten und beantwortet Anfragen von Slaves.
- Slave
  Ein ALFRED-Server, der nur Daten von lokalen Clients speichert. Leitet Daten und Anfragen von lokalen Clients zu einem Master.

4.1.2. ALFRED-Server

Die Referenz-Implementierung von ALFRED verwendet ein einziges Binary für den Betrieb von Clients und Server. Der Unterschied zwischen den Modi Client/Server sowie Master/Slave wird jeweils beim Aufruf der Anwendung durch Parameter beeinflusst. Bei der Netzwerk-Kommunikation unter Servern und beim Datenaustausch zwischen lokalen Clients und einem Server wird deshalb auch das selbe Datenformat verwendet. Unter Server wird in diesem Kontext ein Netzwerk-Dienst (Service) verstanden, welcher folgende Eigenschaften bereitstellt:

- Multicast Empfänger
  Falls im Modus *Slave*, muss ein Master via Multicast-Annoucements an die Gruppen-

[3] `https://github.com/basros/alfred-dissector`

Adresse **FF02::1** gefunden werden

- Multicast Sender
  Falls im Modus *Master*, werden Multicast-Annoucements an **FF02::1** versendet
- Unicast UDP-Socket
  Für die Kommunikation mit anderen Slaves bzw. Masters wird an **Port 16962** ein UDP-Socket gebunden. Hierüber findet der Datenaustausch mit anderen Nodes statt.
- UNIX-Socket
  Für die Kommunikation mit Clients.
- Speichern von *Facts*

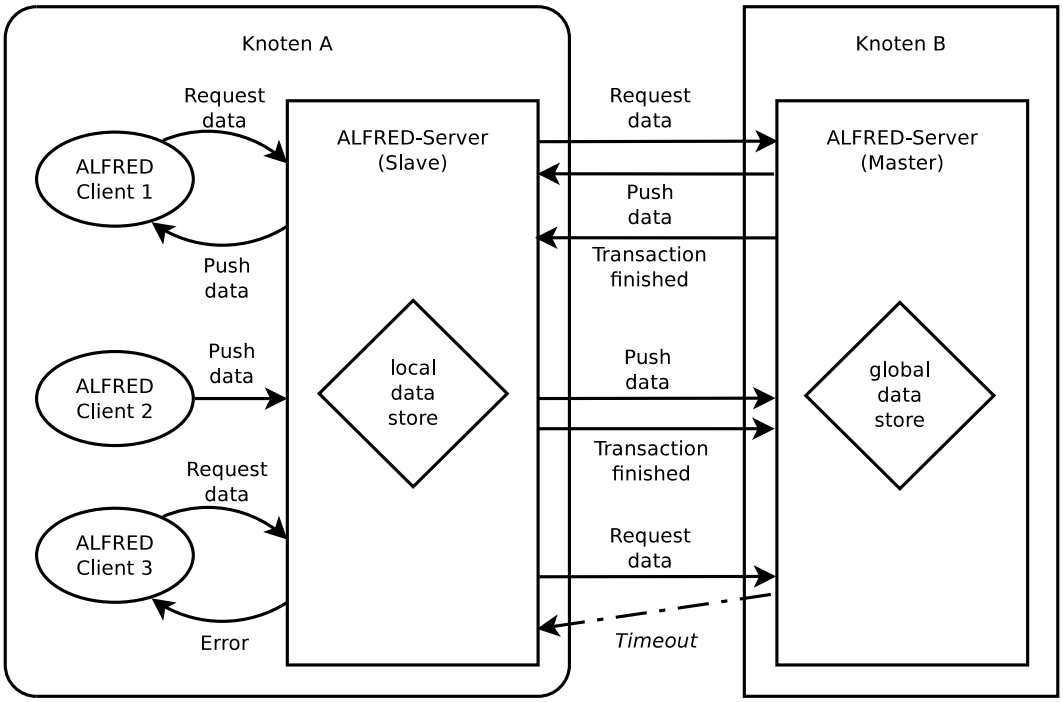

Abbildung 4.1.: Unicast-Traffic zwischen Slave und Master

4.1.3. ALFRED-Client

Die Funktionsweise von ALFRED-Clients wird in 3.2 schon erläutert und wird deshalb in diesem Kapitel nicht im Detail beschrieben. Zudem können ALFRED-Clients nicht eigenständig im Netzwerk interagieren und sind somit für die Definition des Protokolls nicht relevant.

4.1.4. ALFRED Protokoll

Das für ALFRED zugrundeliegende Protokoll arbeitet ausschließlich über **IPv6 Link-Local Adressen**. Dies ist unüblich, da Dienste i.d.R. nicht auf Teilnehmer eines gemeinsamen LAN-Segments beschränkt sind sondern auch über geroutete Netzwerke erreichbar sein sollen. Im Kontext von BATMAN und dem damit aufgespannten virtuellen Layer-2 Switch erweist sich diese Herangehensweise jedoch als praktisch. IPv6 Link-Local Adressen werden durch das *Neighbour Detection Protocol* (NDP) eigenständig vergeben, dadurch benötigt ALFRED keine vorhergehende Konfiguration von IP-Adressen, ist nicht Abhängig von weiteren Diensten im Netzwerk und ist somit als Service sofort verfügbar. ALFRED kann aufgrund des automatische Findens von anderen Instanzen im Netzwerk und wegen des minimalen Konfigurationsaufwands als **Zero-Conf Service** bezeichnet werden.

Als Basis des Netzwerk-Protokolls wurde **UDP** gewählt. Für einen redundant ausgelegten Dienst, welcher meist über drahtlose Verbindungen mit Packet-Loss kommuniziert, bietet es sich mit Hinsicht auf einen geringeren Overhead auch an, ein *stateless* Protokoll zu favorisieren. Alle ALFRED-Datenpakete werden über den **Port 16962**[4] versendet. Source- und Destination-Port sind identisch.

4.1.5. Datenpakete

Alle ALFRED-Datenpakete starten mit einem 4 Byte langem *TLV* (type-length-value) Header. Dieser enthält Informationen über Art (Type, 1 Byte) des Pakets, die Version (1 Byte) und die Länge (Length, 2 Byte) der darauf folgenden Daten. Abbildung 4.2 zeigt einen TLV-Header in Form eines gültigen ALFRED-Pakets.

Die konsequente Verwendung des TLV-Konzepts ermöglicht ein einfaches Aktualisieren der Protokoll-Spezifikationen mit Möglichkeit zur Abwärtskompatibilität, da Pakete mit einem unbekannten Wert im Version-Feld bis zum Offset der im Length-Feld genannten Länge übersprungen bzw. ignoriert werden können. Im Kontext der zum Zeitpunkt dieser Arbeit aktuellen Version (2014.2) von ALFRED gilt die Version 0 im TLV-Header. Für das Feld *Type* sind die Integer-Werte $0-5$ definiert. Die daraus resultierenden Datenpakete werden im folgenden näher beschrieben:

ANNOUNCE_MASTER(1)

Eine Instanz von ALFRED kann entweder im Master- oder Slave-Modus sein. Ein Master verkündet in Intervall von zehn Sekunden die eigene Präsenz mittels eines Pakets. Abbildung 4.2 zeigt, dass solch ein Datenpaket bis auf den Wert 1 im Type-Feld keinen

[4] Port 4242 im für die Programmiersprache C üblichen Hexadezimalsystem

weiteren Inhalt überträgt. Es wird an die Multicast Gruppen-Adresse **FF02::1** gesendet. Diese Adresse wird auch als *All-Nodes* Adresse bezeichnet, weil darüber alle Teilnehmer derselben Link-Local Multicast-Gruppe, also alle im selben Netz-Segment anliegenden Geräte, erreicht werden.

In IPv6 existiert das Konzept der Broadcast-Adressen wie in IPv4 nicht und FF02::1 dient als äquivalenter Ersatz um alle Nodes in der selben Broadcast-Domain zu erreichen.

Ein Client erfährt über die Absender-Adresse des Pakets den jeweiligen Master und nimmt diesen in die Liste der bekannten Master auf. Zwar nicht offiziell dokumentiert, jedoch lässt sich im Quellcode von `alfred` ein definierter Timeout von 60 Sekunden für erlernte Master-Adressen finden.

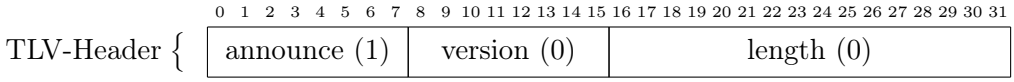

Abbildung 4.2.: ALFRED Master Announcement Packet

Wenn ALFRED auf Nodes mit dem batman-adv Kernel-Module verwendet wird, sieht die Referenz-Implementierung für Slaves eine Selektion des Masters anhand der BATMAN-Metrik vor. Das Kernel-Module verknüpft jede MAC-Adresse eines entfernten Knotens mit dem zugehörigen Wert der LQ-Metrik. In Abbildung 4.3 verkünden zwei Master ihre Präsenz an zwei Slaves via Multicast. Die einzelnen Pfade haben unterschiedliche Verbindungsqualitäten und somit liegen für die Slaves verschiedene LQ-Werte je Master vor.

Im Beispiel wird Slave 1 den Master A wählen, da die Metrik für diesen Master mit $LQ = 150$ besser ist als zu Master B mit $LQ = 45$. Für Slave 2 sind die LQ-Werte beider Master identisch. In diesem Fall wird ein Master zufällig gewählt.

Für die Synchronisation zwischen Masters ist die LQ-Metrik nicht relevant, da immer jeder Master mit jedem anderen Master eine Unicast-Verbindung aufbaut.

REQUEST(2)

Für die Datenübertragung von einem Master zu einem Slave muss zunächst vom Slave eine Anfrage (Request) gestellt werden. Anfragen stehen immer im Kontext eines Facts. Ein Slave versendet ein Request-Packet zu seinem Master um einen Fact anzufragen.

Abbildung 4.4 zeigt im Feld "requested fact" ein Integer-Wert zwischen 0 und 255 als Identifikator für den Fact. In Feld "transaction id" wird je Request eine zufällige Zahl (2 Byte Transaction-ID) zum Verknüpfen von Frage und Antwort verwendet. In der Antwort des Masters muss die selbe Transaction-ID verwendet werden.

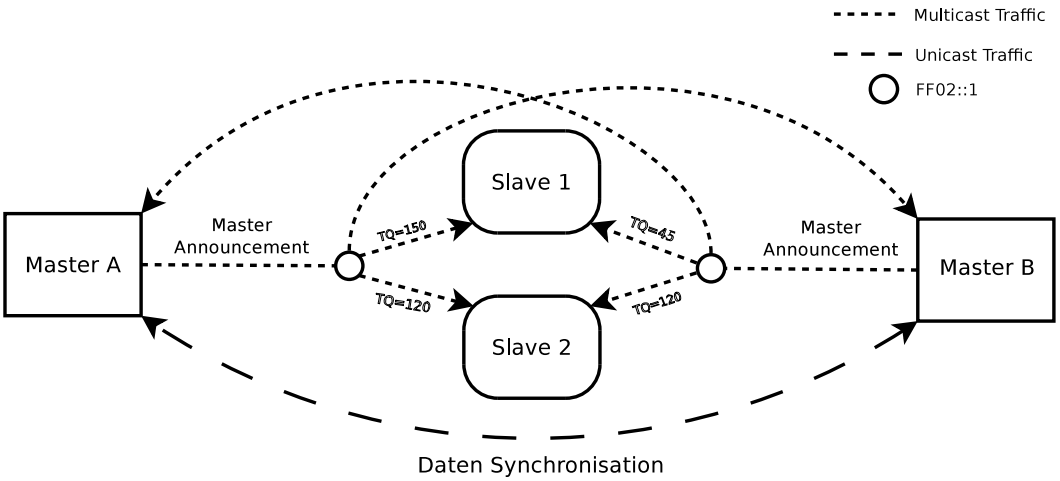

Abbildung 4.3.: Master Annoucements

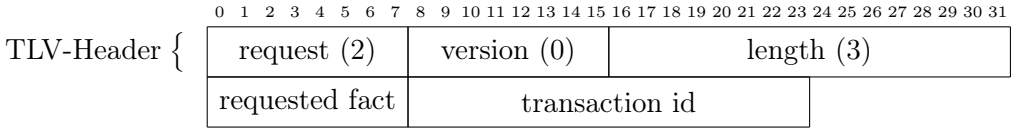

Abbildung 4.4.: ALFRED Fact-Request Packet

PUSH_DATA(0)

Für die Antwort auf einen Request und zum Synchronisieren unter Masters werden *PUSH_DATA* Pakete versendet. Diese Pakete (Abbildung 4.5) enthalten die eigentlichen Daten und können deshalb auch die volle Größe eines UDP-Pakets (65535 Bytes) erreichen. Abzüglich des TLV-Headers (4 Byte) sowie den Feldern für das Transactions-Management (4 Byte) bleiben 65517 Bytes für eine Anreihung von Facts übrig.

Je Fact wird die MAC-Adresse (6 Byte) des ursprünglichen Senders sowie ein zusätzlicher interner TLV-Header verwendet. Type (1 Byte) entspricht hierbei der ID des Facts und Length (2 Byte) der Länge des Data-Feldes.

Das letzte Byte eines Facts muss den Wert [\x0A][5] beinhalten. Dadurch lassen sich die Grenzen zwischen einzelnen Facts im Paket leichter parsen und im Nachhinein aufbereiten.

[5] \n in ASCII, auch bekannt als Linefeed oder New-Line

Sollte beim Synchronisieren unter Masters ein *PUSH_DATA* Paket die maximale UDP-Größe von 65535 Bytes übersteigen, werden weitere Pakete mit einer inkrementierten Sequenz-Nummer versendet.

Ein *PUSH_DATA* Paket muss immer einen Fact enthalten. Die Längen-Angabe im inneren TLV-Header darf nicht 0 sein. Für einzelne Facts, die von einem Client zu einem Master übertragen werden, dürfen nur 65517 Bytes verwendet werden. Außerhalb der Synchronisation werden immer nur einzelne Pakete mit einer Sequenz-Nummer von 0 verwendet. Von einem Slave können entsprechend nur einzelne Facts versendet oder empfangen werden.

Abbildung 4.5 zeigt die schematische Darstellung eines *PUSH_DATA* Pakets mit mehreren Facts.

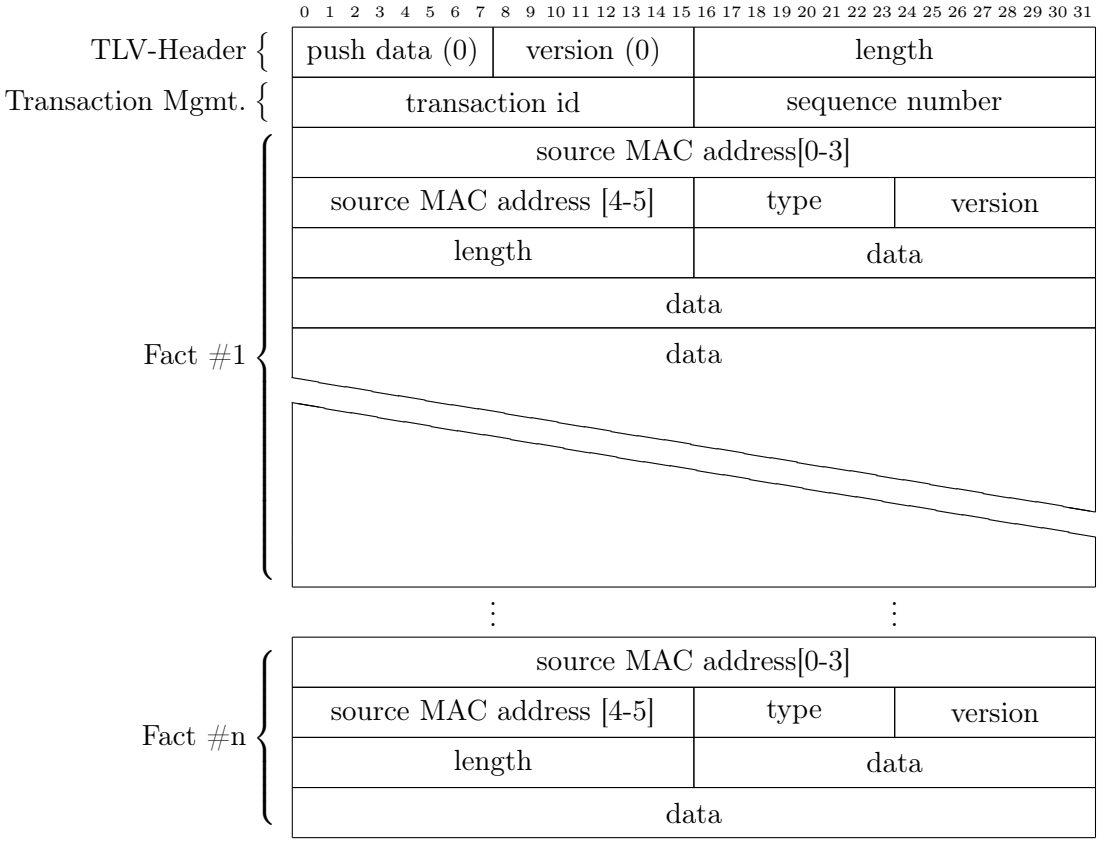

Abbildung 4.5.: ALFRED Push-Data Packet

STATUS_TXEND(3)

Am Ende jeder *PUSH_DATA* Transaktion wird ein spezielles Paket zum Signalisieren einer beendeten Übertragung verschickt. Es enthält die anfangs angegebene "transaction id" sowie die Anzahl der zuvor versendeten Pakete. Nur bei Master-to-Master Synchronisation wird ein Wert größer 1 für die Sequenz-Nummer verwendet, weil dabei mehrere *PUSH_DATA* Pakete in Folge genutzt werden dürfen.

Erst wenn solch ein Paket empfangen wurde, wird der Inhalt von *PUSH_DATA* vom Server an einen Client weiter gereicht.

Sollten zu einem Request keine Daten vorliegen, muss ein Master nicht mit einem leeren *PUSH_DATA* Paket antworten, sondern direkt ein *STATUS_TXEND* Paket mit der Sequenz-Nummer 0 versenden.

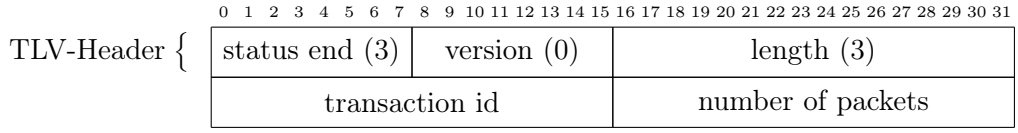

Abbildung 4.6.: ALFRED Finished-Transaction Packet

STATUS_TXERROR(4) und MODESWITCH(5)

Falls ein Slave-Server auf die Anfrage eines Clients vom Master in der Zeiteinheit von 10 Sekunden keine Antwort erhält, wird in der Referenz-Implementierung der Client über den UNIX-Socket mit *STATUS_TXERROR* über einen Fehler benachrichtigt.

Um den Modus (Master/Slave) eines Servers zu wechseln, existiert ein eigener Pakettyp. Ein Client schickt an den lokalen Server ein Paket vom Typ *MODESWITCH(5)*. Auch diese Kommunikation geschieht ausschließlich über den UNIX-Socket.

Für die Spezifikation des Netzwerk-Protokolls sind diese beiden Paket-Typen nicht relevant.

5. Entwurf

5.1. Funktionalität

Die in diesem Kapitel beschriebene Architektur soll gewisse Basis-Funktionalitäten gewährleisten. Diese müssen jedoch zunächst spezifiziert und priorisiert werden. Eine Gewichtung der Anforderungen wird durch die Verwendung der Wörter *"muss"*, *"soll"* und *"kann"* realisiert.

Alle *Muss-Kriterien* stellen die Kompatibilität zur Referenz sicher.

Die *Soll-Kriterien* spiegeln Intentionen der Referenz-Implementierung wieder, welche jedoch nicht in Abhängigkeit zur Kompatibilität stehen.

Kann-Kriterien werden verwendet, um das Verhalten bzgl. Speicher- und Akku-Verbrauch unter Android in einem sinnvollen Rahmen zu halten. Zudem sollen sie einem Benutzer gewisse Freiheiten in der Bedienung ermöglichen.

- Finden eines ALFRED-Master:
  Pakete vom Typ `ANNOUNCE_MASTER` müssen erkannt werden.
  Die IPv6 Link-Local Adresse des Absenders muss extrahiert und gespeichert werden.
  Falls zu einem Master nicht in regelmäßigen Abständen ein `ANNOUNCE_MASTER` Paket empfangen wird, soll die Adresse dieses Masters nach Ablauf von 60 Sekunden entfernt werden.

- Senden von Requests:
  Pakete vom Typ `REQUEST` müssen so generiert und versendet werden, dass die Referenz-Implementierung diese korrekt annimmt und beantwortet.
  Die verwendete Transaction-ID soll solange mit Verweis auf den anfragenden Client gespeichert werden, bis ein passendes `STATUS_TXEND` empfangen wurde.
  Ein bestimmter Master aus einer Liste mehrere Master kann für Anfragen verwendet werden.

- Empfangen von Facts:
  Pakete vom Typ `PUSH_DATA` müssen erkannt werden.
  Der Inhalt muss entsprechend der Datenfelder ausgelesen und als eigene Datentypen zur nachfolgenden Verwendung abgelegt werden.
  Die Struktur eines `PUSH_DATA` Pakets soll validiert werden.
  Die Daten sollen erst nach Empfang eines `STATUS_TXEND` an den anfragenden Client

weitergereicht werden.
Der Inhalt eines `PUSH_DATA` Pakets ohne zugehöriges `STATUS_TXEND` Paket kann gelöscht werden.

- Erkennen abgeschlossener Transaktionen:
  Pakete vom Typ `STATUS_TXEND` müssen erkannt werden.
- Unterstützung multipler Clients:
  Mehrere Clients für spezielle Facts können gleichzeitig verwendet werden.
- Fehlerbehandlung:
  Das Fehlen eines Masters muss erkannt werden.
  Das Ausbleiben von Paketen und daraus resultierende Timeouts sollen erkannt werden.
  Nicht darstellbare ASCII-Codes können entfernt oder ersetzt werden.
- Einstellungsmöglichkeiten:
  Die Anwendung soll in bestimmten WLAN-Umgebungen anhand der ESSID automatisch gestartet werden.
  Die Anwendung kann händisch gestartet und gestoppt werden.

5.2. Android-Architektur

Es biete sich an, für die Implementierung von ALFRED für Android auch eine Trennung von Server und Client zu schaffen. Dadurch wird die Architektur sinnvoll strukturiert und bestimmte Anforderungen an die Funktionalität gewährleistet. Zudem soll durch die dabei entstehende Modularität eine spätere Erweiterung vereinfacht werden.

Gemäß den Developer-Guidelines[1] für Android, sollen die Teile einer Applikation, die beständig arbeiten ohne im Vordergrund sichtbar zu sein, in einem sog. *Background-Service* implementiert werden. Dieser Service soll als Schnittstelle zum Netzwerk dienen, die grundlegenden Funktionalitäten von ALFRED bereitstellen und somit die eigentliche Implementierung des Protokolls beherbergen.

Um mehrere Applikationen für unterschiedlichen *Facts* über einen gemeinsamen lokalen Server mit anderen ALFRED-Instanzen im Netzwerk kommunizieren zu lassen, soll ein *Connector* geschaffen werden. Der Datenaustausch von *Background-Service* zu Applikationen soll mittels *Intents*, die von Android bereitgestellte Variante von *Inter-Process Communication* (IPC), realisiert werden.

Die Unterscheidung zwischen Master und Slave sorgt nur bei Verwendung innerhalb eines BATMAN-Mesh mit Wissen über die Topologie für eine Optimierung des Datenfluss'. ALFRED kann zwar auch in Netzen ohne BATMAN-Infrastruktur verwendet werden, jedoch liegt der Fokus dieser Arbeit bei der Erweiterung für Mesh-Netzwerke. Deswegen soll

[1] https://developer.android.com/guide/components/services.html

der serverseitige Teil der Android-App einen Slave darstellen. Der Background-Service wird im weiteren Verlauf der Arbeit als eigenständige Applikation unter dem Namen *AlfredA* (ALFRED on Android) geführt.

Programmteile, die sowohl vom *Background Service* als auch von jeder Applikationen verwendet werden, sollen in eine gemeinsame Bibliothek ausgegliedert werden. Dies stellt zum einen sicher, dass ein einheitliches Datenformat für die Kommunikation zwischen App und Service mittels Intents verwendet werden kann, zum anderen wird die Entwicklung zukünftiger Applikationen stark vereinfacht, da Entwickler nicht alle Details des Protokolls von ALFRED kennen müssen. Die Kernkomponente der *AlfredaLib* genannten Bibliothek ist ein *Connector* für die Kommunikation zwischen App und Service sowie ein von jeder App zu implementierendes *Interface*.

Eine Android-Applikation, die mit dem Background-Service kommuniziert, soll nur die Erstellung, Verarbeitung und Darstellung der im Kontext des jeweiligen *Facts* vorliegenden Daten ermöglichen. Entsprechend müssen nur die clientseitigen ALFRED-Funktionalitäten implementiert werden. Im Rahmen dieser Arbeit soll lediglich eine einfache Applikation zum Erfragen und Hinzufügen von Facts entworfen werden. Sie dient exemplarisch der Entwicklung von Client-Apps für den AlfredA-Service mittels AlfredaLib.

Abbildung 5.1 zeigt die drei genannten Komponenten im Zusammenhang mit einem Benutzer und der Referenz-Implementierung im Netzwerk.

Die Android-interne Repräsentation der ALFRED-Datenstruktur soll durch einen eigenen komplexen Datentyp realisiert werden. Mit Setter- und Getter-Methoden können Objekte solch einer Klasse aus bzw. zu ALFRED-Datenpaketen konstruiert werden.

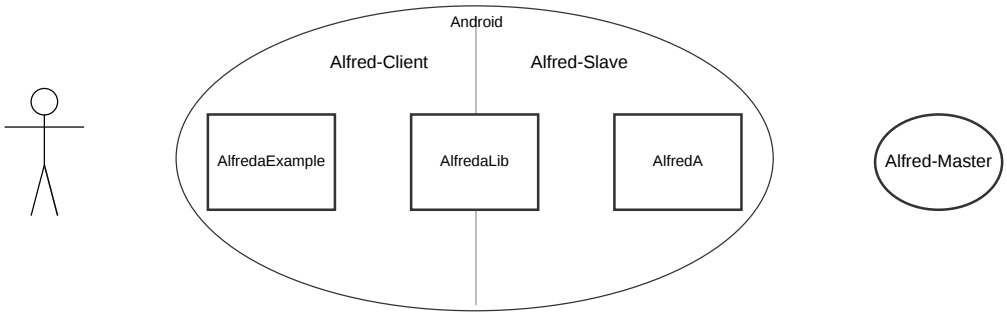

Abbildung 5.1.: AlfredA Komponenten

6. Implementierung

Der zeitliche Ablauf und Zusammenhang der im nachfolgenden beschriebenen Komponenten kann auch als Sequenz-Diagramm repräsentiert werden. Abbildung 6.1 beschränkt sich hierbei auf die Kern-Komponenten und den groben Ablaufplan bei der Anfrage eines Clients nach einem Fact.

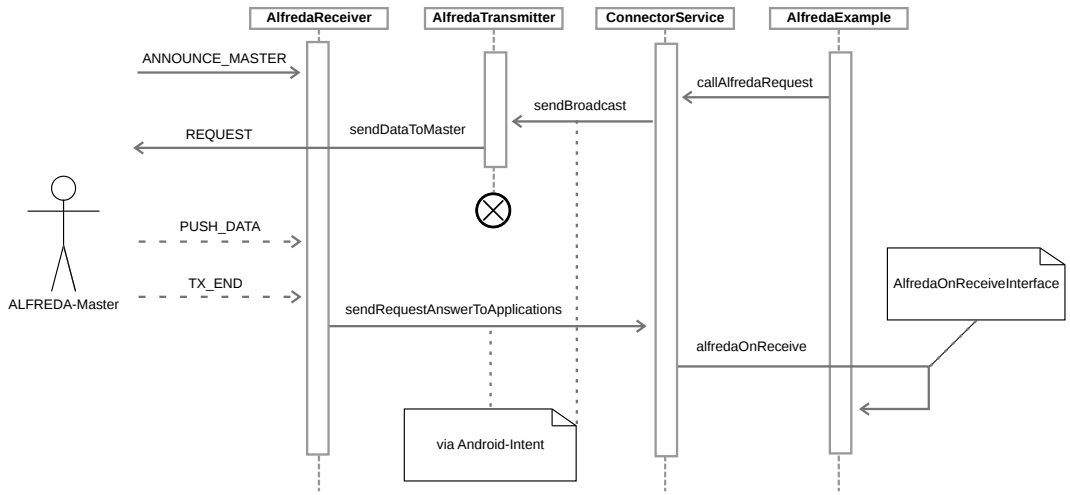

Abbildung 6.1.: AlfredA Komponenten Sequenz-Diagramm

6.1. Alfreda

Der Background-Service ist in zwei Haupt-Komponenten aufgeteilt, die auch zwei Java-Klassen darstellen. Sowohl zum Empfangen als auch zum Senden von ALFRED-Pakten werden dementsprechend dedizierte Klassen verwendet. Da UDP *stateless* ist, kann eine Klasse für dauerhaftes Lauschen auf einem Socket und gleichzeitig eine andere Klasse für das Senden über den selben Port verwendet werden.

Zum Starten und Stoppen des Services, sowie für Einstellungen, wie z.B. das spezifische WLAN, unter welchem der Service automatisch gestartet werden soll, wird eine eigene

Activity namens *AlfredaSettings* verwendet. Die Struktur solcher Konfigurationsparameter ist jedoch bereits für eine Ausgliederung in Client-Apps vorbereitet.

Für alle Komponenten gilt: Methoden und statische Variablen, die von mehreren Klassen verwendet werden, werden in eine dedizierte Klasse `Utils` ausgliedern.

6.1.1. AlfredaReceiver

`AlfredaReceiver` ist für den Empfang aller ALFRED-Datenpakete von einem entfernten Master zuständig. Diese Klasse implementiert einen `Android Service` und darf deshalb dauerhaft als Hintergrund-Prozess verweilen.

Es wird ein *UDP-Socket* auf Port 16926 an das WLAN-Interface gebunden und in einem eigenen Thread alle eingehenden Pakete auf ihren Typ analysiert.

Die durch Master-Announcements gesammelten Adressen werden in Androids persistentem Storage-Mechanismus `SharedPreferences` abgelegt. Dieser Key-Value Speicher kann auch von anderen Klassen im gleichen Package verwendet werden. Die Klasse `AlfredaTransmitter` benötigt diese Adressen zum Senden von Client-Requests an einen Master.

Gleichzeitig wird eine *HashMap* als zusätzliche Datenstruktur für Master-Adressen gepflegt. Neben der eigentlichen Adresse wird der Timestamp des Empfangs gespeichert. Ein Thread entfernt in regelmäßigen Abständen die Adressen von Mastern, die ein Alter von 60 Sekunden überschritten haben. Falls ein Update dieser Map geschieht, wird auch die Master-Liste in den `SharedPreferences` aktualisiert.

Push-Data Pakete werden zunächst in einem zweidimensionalen Array abgelegt. Zugriff auf den Inhalt erfolgt über die Transaction-ID. Erst wenn ein Transaction-Finished Paket mit der selben Transaction-ID und der korrekten Sequenz-Number empfangen wurde, wird der Inhalt des Push-Data Pakets als *Intent* an eine lokale Client-Anwendung weiter gereicht. Diese Datenstruktur wurde gewählt, um in einem fortlaufenden Thread in regelmäßigen Abständen verwaiste Push-Data Pakete zu entsorgen. Gleichzeitig wurde es vorausschauend entworfen, weil für eine zukünftige Entwicklung eines Alfred-Masters unter Android eine Datenstruktur zum Ablegen mehrerer Push-Data Pakete unter der selben Transaction-ID benötigt wird.

Die Klasse hat eine Methode `sendRequestAnswerToApplications()` um den Inhalt eines Facts an den richtige Client-Anwendung zu übermitteln. Die Methode extrahiert aus einem empfangenen `PUSH_DATA` Pakets alle benötigten Datenfelder und versendet diese über ein *Bundle* von *Intents* als Broadcast an das Betriebssystem.

6.1.2. AlfredaTransmitter

`AlfredaTransmitter` versendet alle Alfred-Datenpakete, die von einer lokalen Client-Anwendung erzeugt wurden. Um *Intents* von anderen Klassen annehmen zu können, implementiert die Klasse einen `Android Broadcast-Receiver` und wird dabei mit jedem Aufruf per Intent neu erzeugt. Es werden somit keine Daten persistent gehalten.

Lokale Client-Anwendungen, die Facts anfragen oder Daten zu einem Fact in das Netzwerk senden, starten diese Klasse mit einem Intent und dem zugehörigen Modus. Die dazu benötigte IPv6 Link-Local Adresse eines Alfred-Masters wird aus den `SharedPreferences` ausgelesen. Da Clients nur die Daten als String- oder Byte-Array übertragen, wird das eigentliche UDP-Paket direkt in der Klasse erzeugt. Somit wird für Clients die Komplexität der Alfred-Datenstruktur reduziert, was wiederum zum einfachen Entwickeln von AlfredA Client-Applikationen führen soll.

6.2. AlfredaLib

Das Package `AlfredaLib` dient als Middleware für die Verbindung von Client-Applicatiosn und Background-Service. Alle Komponenten binden diese als *Library* ein. Die zuvor erwähnte Klasse `Utils` befindet sich in diesem Package. Für Clients-Applikationen wird ein zu implementierendes Interface bereit gestellt.

6.2.1. ResponsePacket

In dieser Klasse wird die interne Datenstruktur von AFLRED-Paketen abgebildet. Da innerhalb von Android die Netzwerk-bezogenen Datenfelder von ALFRED-Paketen keine Rolle spielen und nur der eigentliche Inhalt wichtig ist, wird ein Objekt vom Typ `ResponsePacket` für die Kommunikation zwischen dem Background-Service und lokalen Clients verwendet.

Unterschiedliche Konstruktoren werden angeboten, um verschiedene Fälle abzubilden:

Falls ein Fact übertragen wird, unabhängig davon, ob ein `PUSH_DATA` Paket empfangen wurde oder verschickt werden soll, wird der Konstruktor mit allen benötigten Werten als Übergabeparameter aufgerufen.

Wenn ein `STATUS_TXEND` bearbeitet wird, kann nur die Transaktions-ID als Parameter übergeben werden.

Die Klasse enthält zudem für alle internen Datentypen entsprechende Setter- und Getter-Methoden.

6.2.2. ConnectorService

Die Kernkomponente von `AlfredaLib` ist die Klasse `ConnectorService`. Sie implementiert einen BoundService.

> A bound service is the server in a client-server interface. A bound service allows components (such as activities) to bind to the service, send requests, receive responses, and even perform interprocess communication (IPC). A bound service typically lives only while it serves another application component and does not run in the background indefinitely.[1]

Es erscheint zunächst seltsam, neben der Receiver-Klasse in *Alfreda* einen weiteren *Server* zu implementieren, jedoch bietet diese Architektur eine saubere Trennung zwischen *Netzwerk-Server* und *Server für Apps*.

`ConnectorService` implementiert über eine innere Klasse `AlfredaBinder` einen `Android Binder`. Dies wird benötigt, damit sich lokale Clients an den Service "binden" können. Über diese Verbindung können Clients die zwei nachfolgenden Methoden aufrufen:

Die Methode `callAlfredaPush` dient dem Versand von Facts. Die Methode ist *überladen* und ermöglicht somit Client-Applikationen die Daten eines Facts entweder als String-Array oder als Byte-Array zu übermitteln. Die Daten werden in der Methode um eine Transaktions-ID ergänzt und als *Intent-Bundle* per Broadcast an `AlfredaTransmitter` zum letztendlichen Versand ins Netzwerk übermittelt.

Ähnlich verhält es sich mit der Methode `callAlfredaRequest`, die einen anzufragenden Fact von einem Client annimmt aber die Transaktions-ID selbst setzt. Dieses Vorgehen ermöglicht eine Verknüpfung von Anfrage und Antwort zum jeweiligen Client.

Ein Instanz dieser Klasse wird einmalig von einem beliebigen Client erzeugt und kann dann von jedem Client verwendet werden. Erst wenn der letzte Client beendet und damit die letzte Bindung aufgelöst wird, zerstört sich diese Instanz. Entsprechend wird an dieser Stelle das Pattern *Singleton* angewendet.

6.2.3. AlfredaOnReceiveInterface

Damit lokale Clients auch Antworten auf Anfragen erhalten können, müssen diese das Interface `AlfredaOnReceiveInterface` implementieren. Das Interface deklariert zwei Methoden:

`alfredaOnReceive` nimmt ein `ResponsePacket` an. Die Methode wird in der Klasse `ConnectorService` aufgerufen, nachdem von AlfredA per Intent der Inhalt eines `PUSH_DATA` Pakets an den Connector-Service übermittelt wurde.

`alfredaNoMasterFound` dient dem Error-Handling, falls kein Master verfügbar ist. Die

[1] https://developer.android.com/guide/components/services.html

Methode wird ebenso von der Klasse `ConnectorService` aufgerufen. Wie die Fehlermeldung einem Benutzer präsentiert wird, ist den jeweiligen Implementierung der Client-Anwendungen überlassen.

6.3. AlfredaExample

Ein lokaler Client benötigt dank der zuvor beschriebenen Architektur nur sehr geringen Entwicklungsaufwand. Die Applikation `AlfredaExample` dient hierzu als Beispiel.

Die gesamte App besteht aus einer einzigen Java-Klasse und implementiert nur eine *Activity*. Ein Screenshot davon wird in Abbildung A.1 dargestellt:

Dem Benutzer werden zwei Eingabefelder bereit gestellt. Eines für die ID eines Facts, welches nur numerische Werte im Bereich $0 - 255$ annimmt. Das andere Feld nimmt Benutzereingaben in Form von Strings an. Der Benutzer kann über zwei Buttons einen Fact-Request (requestInformation) versenden, oder einen Beitrag zu einem Fact pushen (pushInformation).

Die Verbindung zu `ConnectorService` wird als Member `mService` der Klasse `AlfredaExample` realisiert und wird im folgenden Listing 6.1 dargestellt. Auf dem `mService` Objekt kann nach dem anbinden z.B. die Methode callAlfredaPush aufgerufen werden.

Listing 6.1: Erstellen einer Bound-Service Verbindung

```
1 private ServiceConnection mConnection = new ServiceConnection() {
2
3    @Override
4    public void onServiceConnected(ComponentName className, IBinder
         service) {
5       ConnectorService.AlfredaBinder binder = (ConnectorService.
            AlfredaBinder) service;
6       mService = binder.getService();
7       mBound = true;
8    }
9 }
```

6.4. Besonderheiten unter Android

Um unter Android Multicast-Pakete zu empfangen, ist zwar eigentlich ein Join zur entsprechenden Multicast-Group nötig, für die All-Nodes Adresse FF02::1 werden Pakete jedoch automatisch angenommen. Für einen Join zu anderen Multicast-Gruppen muss eine spezielle *Permission* gesetzt werden.

Der bevorzugte Weg für IPC unter Android sind Intents. Jedoch können komplexe Datentypen nicht ohne weiteres per Intent übertragen werden. Die Datentypen müssen

als *Serializable* oder *Parcelable* definiert und um dafür benötigte Methoden angepasst werden. Alternativ bietet sich die Möglichkeit von einfachen Datentypen als *Bundle* eines Intents an. Im hier vorgestellten Prototypen wird anstelle eines `ResponsePackets` die Repräsentation eines solchen Objekts in seinen Basis-Datentypen als Intent-Bundle übertragen.

Java kennt keinen Datentyp *unsigned byte*. In C werden jedoch besonders für effiziente Implementierungen von Netzwerk-Protokollen unsigned Bytes verwendet. An einigen Stellen muss deshalb durch einen *bitwise shift* der Wertebereich eines Java signed Byte verschoben werden.

7. Tests und Demonstration

Softwareentwicklung hat das Ziel eines fertigen Produkts. Um den Zustand der abgeschlossenen Entwicklung definieren zu können, müssen alle Anforderungen an die Software als erfüllt gelten. Die eingehaltenen Anforderungen können auch in Ihrer Qualität gemessen werden.

Um die Anforderungen und Qualität von Software zu messen, existieren Testverfahren. Erst wenn die Software die Bedingungen erfüllt und gelieferte Werte im Rahmen der Anforderung liegen, sollte Software in den nächsten Schritt der Entwicklung übergehen. Damit kann die Veröffentlichung des Produkts gemeint sein, aber auch der Start eines neuen Entwicklungs-Zyklus, in welchem zusätzliche Features implementiert werden.

Spillner und Linz definieren Mangel und Fehler von Software:

> "Ein Fehler ist [.] die Nichterfüllung einer festgelegten Anforderung, eine Abweichung zwischen dem Istverhalten und dem Sollverhalten. Ein Mangel liegt vor, wenn eine gestellte Anforderung oder eine berechtigte Erwartung nicht angemessen erfüllt wird."[SL05]

7.1. Testverfahren

Um Software sinnvoll zu testen, haben sich zwei Verfahren etabliert.

Beim *Blackbox-Verfahren* wird keine Kenntnis über die innere Funktionsweise von Software angenommen. Nur die Ergebnisse der Testszenarien, welche sich von den Anforderungen und Spezifikationen ableiten lassen, gelten als relevant.

Das *Whitebox-Verfahren* bezieht den Quelltext von Software in Testszenarien aktiv mit ein. Es wird der Ablauf einzelner Software-Komponenten analysiert und nicht der gesamte Zusammenhang der Software einem Test unterzogen. Häufig werden auch speziell für einzelne kleine Tests atomare Bestandteile des Quelltextes angepasst, um Abhängigkeiten zu anderen Komponenten zu vermeiden.

7.2. Testarten

Neben Testverfahren existieren unterschiedliche Arten von Tests, die z.T. auch in unterschiedlichen Stationen der Softwareentwicklung durchgeführt werden:

Funktionale Tests bezieht sich meist auf einzelne Komponenten und deren Ein- und Ausgabeverhalten. Spezifiziert werden sie nach den funktionalen Anforderungen, die beim Entwurf der Software gestellt wurden. Je feiner diese definiert sind, desto genauer lassen sich die Testszenarien beschreiben. Es gilt einzelne Systemfunktionen zu testen. Dies geschieht meist mit einem Whitebox-Verfahren.

Nicht funktionale Tests werden verwendet, um Qualitäts-Eigenschaften von Software zu testen. Sie lassen sich in Kategorien wie z.B. Zuverlässigkeit, Benutzerfreundlichkeit oder Performance klassifizieren. Die Spezifikation ist meist nicht rein subjektiv, da z.B. Benutzerfreundlichkeit eine abstrakte Eigenschaft ist, welche sich nur schwer und ungenau in den Anforderungen spezifizieren lässt. Das Blackbox-Verfahren bietet sich für diese Art von Tests an.

Strukturbezogene Tests behandeln das Zusammenspiel einzelner Komponenten von Software. Sie werden deshalb auch meist in Komponenten- und Integrations-Tests aufgeteilt. Es sollen interne Strukturen von mehreren Komponenten in ihrem Kontrollfluss überprüft werden.

Änderungsbezogene Tests bzw. *Regressionstests* werden angewendet um Veränderungen oder Korrekturen von bereits getesteten Komponenten zu bestätigen. Bei Veränderungen an einer Komponente soll dadurch sichergestellt werden, dass keine negativen Auswirkungen auf andere Komponenten auftreten.

7.3. Testkonzept

In diesem Abschnitt sollen der Android-Service *AlfredA* in Kombination mit dem AlfredA-Client *AlfredaExample* einem Test nach dem Blackbox-Verfahren unterzogen werden. Es gilt festzustellen, ob diese die gestellten Anforderungen erfüllen.
Gleichzeitig wird die Spezifikation des Netzwerk-Protokolls überprüft, indem die netzwerkseitigen Android-Komponenten gegen die Referenz-Implementierung getestet wird.

7.3.1. Testumgebung

Für den Test wird folgende Umgebung verwendet:

Ein mit WPA2 verschlüsseltes WLAN, erzeugt durch einen handelsüblichen Heim-Router.

Ein Huawai Smartphone mit Android 4.0.4. Eingebucht im WLAN. Das WLAN-Interface besitzt die MAC-Adresse `0c:37:dc:f4:86:83`.

Ein Laptop mit Linux, im selben WLAN. Das WLAN-Interface besitzt die MAC-Adresse `78:dd:08:e7:c1:8b`.

Das Laptop agiert als ALFRED-Master. Es wird Version 2014.2 von `alfred` verwendet. Gleichzeitig werden alle ALFRED-Pakete von Wireshark aufgezeichnet und mit dem Alfred-Dissector analysiert.

7.3.2. Testplan

Es soll die Kompatibilität zur Referenz-Implementierung von ALFRED getestet werden.

Testcase #1

Vom AlfredA-Service wird durch den AlfredA-Client *AlfredaExample* ein `PUSH_DATA` Paket an den ALFRED-Master gesendet. Es wird die Fact-ID 66 verwendet, und die übertragene ASCII-String lautet `"test?"`.

Der Test gilt als bestanden, wenn die Referenz-Implementierung den String `"test?"` mit der MAC-Adresse des Smartphones darstellt. Anschließend wird das `PUSH_DATA` Paket in Wireshark analysiert und die einzelnen Felder des Daten-Pakets auf Korrektheit überprüft.

Testcase #2

Vom ALFRED-Master wird ein `PUSH_DATA` Paket mit dem String `"bestanden!"` unter der Fact-ID 66 (siehe Listing 7.1) erzeugt. Die Android-Applikation *AlfredaExample* soll den gesamten Fact 66 mit einem `REQUEST_DATA` Paket beim ALFRED-Master anfragen.

Listing 7.1: Eingabe des Test-Strings unter Fact 66 durch `alfred`

```
1 # echo bestanden! | /usr/local/bin/alfred -s 66
```

Der Test gilt als bestanden, wenn in der Android-Applikation sowohl der eigene Eintrag `"test?"`, als auch der zweite String `"bestanden!"` dargestellt werden. Die MAC-Adressen dieser Beiträge müssen zum jeweiligen Sender passen.

Das `REQUEST_DATA` Paket von AlfredA wird im Wireshark-Dissector analysiert.

7.4. Testergebnisse

Testcase #1

Der unter Android erzeugte und versendete Fact wird von der Referenz-Implementierung erfolgreich angenommen und dargestellt. Die Ausgabe von `alfred` in Listing 7.2 wird mit der korrekten MAC-Adresse dargestellt. Zudem werden keine zusätzlichen bzw. falschen Bestandteile im String übermittelt.

Listing 7.2: Ausgabe des Requests von Fact 66 durch `alfred`

```
1 # /usr/local/bin/alfred -r 66
2 { "0c:37:dc:f4:86:83", "test?\x0a" },
```

Die Analyse durch Wireshark in Abbildung A.2 stellt folgende Anforderungen als erfüllt dar:

- Die Pakete `PUSH_DATA` und `STATUS_TXEND` verwenden die selbe *Transactions-ID* `0x4dde`.
- Die Länge des Facts beträgt 6 Bytes

Testcase #2

In *AlfredaExample* werden beide Beiträge zu Fact 66 korrekt dargestellt. Die MAC-Adressen stimmen mit den Absendern überein. Es werden keine zusätzlichen oder falschen Zeichen dargestellt. Der Screenshot der Applikation in Abbildung A.1 veranschaulicht dies. Die zeitliche Reihenfolge von Beiträgen zu Facts verläuft von oben nach unten, mit dem neusten Beitrag an oberster Stelle.

Da zu diesem Zeitpunkt auf dem Android ALFRED-Slave und dem Linux ALFRED-Master die selben Daten als Fact 66 hinterlegt sein sollten, zeigt Listing 7.3 den Inhalt der Referenz-Implementierung.

Listing 7.3: Ausgabe des 2. Requests von Fact 66 durch `alfred`

```
1 # ./alfred -r 66
2 { "78:dd:08:e7:c1:8b", "bestanden!\x0a" },
3 { "0c:37:dc:f4:86:83", "test?\x0a" },
```

Abbildung A.3 zeigt, wie die Transaction-ID `0x2918` von AlfredA im `REQUEST_DATA` verwendet wird und die Referenz-Implementierung damit bei den Paketen `PUSH_DATA` und `STATUS_TXEND` korrekt antwortet.

8. Auswertung

8.1. Zusammenfassung

Die im Grundlagen-Kapitel beschriebenen Systeme und Konzepte von Mesh-Netzwerken sind nicht nur im akademischen Umfeld vertreten, sondern werden vielerorts produktiv eingesetzt. Durch deren detaillierte Beschreibung werden Kenntnisse zur Funktionsweise von solchen Netzwerken vermittelt, die auch zukünftig im stetig wachsenden Anwendungsfeld "*Internet of Things*" und "*Ubiquitous Computing*" von Nutzen sein werden.

Der Hauptteil dieser Arbeit analysiert mehrere System zum Verteilen von Positionsdaten. Letztendlich wird jedoch festgestellt, dass ALFRED, mit dem allgemeinen Ansatz der "*lightweight facts*", eine sehr effiziente Realisierung für den Austausch jeglicher Art von kleinen Informationseinheiten darstellt. Besonders im Hinblick auf andere Arten von drahtlosen vermaschten Systemen liefert die ausführliche Analyse des Protokolls und der Datenstruktur des Dienstes hilfreiche Anhaltspunkte für die Entwicklung von ähnlichen Systemen.

Die ausgearbeitete Architektur von "*ALFRED on Android*" hat zu einer funktionsfähigen und zur Referenz kompatiblen mobilen Anwendung geführt. Die im Detail beschriebene Implementierung liefert nicht nur einen Einblick in die Netzwerk-Programmierung unter Android, sondern kann auch für eine Portierung von ALFRED auf andere mobile Betriebssysteme herangezogen werden.

8.2. Vergleich von Anforderung und Realisierung

Die Auswertung der beiden Testszenarien in Kapitel 7 zeigt, dass der entwickelte Prototyp eine geeignete Basis zur Verteilung von Positionsdaten darstellt. Die Eingabe und Präsentation der Koordinaten benutzerfreundlich zu gestalten obliegt einer spezialisierten Client-Applikation. Für individuelle, Fact-bezogene Android-Anwendungen wurde jedoch durch die gewählte Architektur eine solide Grundlage geschaffen.

Die Muss-Kriterien der Anforderungs-Spezifikation wurden erfüllt. Die Soll-Bedinungen sind in den Testszenarien zwar nicht in voller Gänze überprüft worden, jedoch liefern die im Detail beschriebenen Komponenten in Kapitel 6 einen Einblick in deren Implementierung.

8.3. Ausblick

Mit *ALFRED on Android* wurde die Basis für eine Vielzahl denkbarer ALFRED-Clients geschaffen. Die für Clients angebotene Datenstruktur ist nicht auf die Übertragung von rein textuellen Inhalten wie im Beispiel *AlfredaExample* beschränkt, sondern auch binäre Daten können ausgetauscht werden. Im folgenden werden Vorschläge für weitere mögliche Anwendungsfälle unterbreitet:

- Service Announcement & Discovery
  In Community Mesh-Networks werden häufig von Teilnehmern eigene lokale Dienste betrieben. Dabei kann es sich z.B. um öffentliche Drucker oder freigegebene File-Server handeln. Diese Dienste an End-Nutzer zu verkünden wird häufig mit zentralen Service-Listen auf Webservern gehandhabt. AlfredA stellt eine native Alternative für die Präsentation unter Android dar.
- Instant remote Webcam
  Mit Apps wie "IP Camera"[1] kann die im Smartphone eingebaut Kamera in eine Webcam verwandelt werden. Die Anwendung erzeugt einen minimalen Webserver und stellt die zugehörige URL bereit. Solch eine URL könnte via AlfredA als eigener Fact übertragen werden ließe sich von anderen AlfredA Benutzern als klickbarer Link für Livestreaming öffnen.
- Mikrofon Stream
  Ähnlich wie die zuvor erwähnte Kamera lässt sich auch das eingebaute Mikrofon an weitere entfernte Smartphones als Audio-Stream verteilen. Die App "LANMic"[2] erstellt ebenso die URL zu einem lokalen Webserver. Diese URL zum Audio-Stream könnte von einem dedizierten AlfredA-Client App in das Netzwerk eingestellt und daraus abgerufen werden.

[1] `https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en`
[2] `https://play.google.com/store/apps/details?id=com.portable.lanmic`

Im Prototypen wurde das zeitliche Verhalten von Facts, welches in der Referenz verwendet wird, bisher nicht berücksichtigt. Es ist vorgesehen, dass Clients die empfangen Daten eines Facts nach 600 Sekunden verwerfen, sofern diese in der Zwischenzeit nicht aktualisiert wurden. Denkbar wäre an dieser Stelle eine Option zum automatischen erneuten Anfragen. Solch ein Thread müsste jedoch von einem Background-Service ausgeführt werden. Activities, wie z.B. AlfredaExample, können Berechnungen nur ausführen, solange sie im Vordergrund sichtbar sind. Ebenso sollen in Activities keine langlebigen Prozesse – wie solch ein Aktualisierungs-Thread – verweilen. Gleichzeitig sollten Clients auch eine Warnung über veraltete Daten an den User präsentieren. Dies könnte z.B. über das zu implementierende Interface `AlfredaOnReceiveInterface` forciert werden.

Das AFLRED-Protokoll spezifiziert, in `PUSH_DATA` Paketen je Fact die MAC-Adresse des Absenders zu nennen. Für Android-User liefern MAC-Adressen jedoch keinen direkten Verwendungszweck. Besonders hinsichtlich der zuvor genannten Beispiele von Client-Apps wäre ein automatisiertes Auflösen der MAC-Adressen in IPv6 Link-Local Adressen sinnvoll. Dies könnte auch bereits im Background-Service geschehen oder von AlfredaLib angeboten werden und müsste somit nicht von jedem Client selbst implementiert werden.

A. Appendix

A.1. Abbildungen

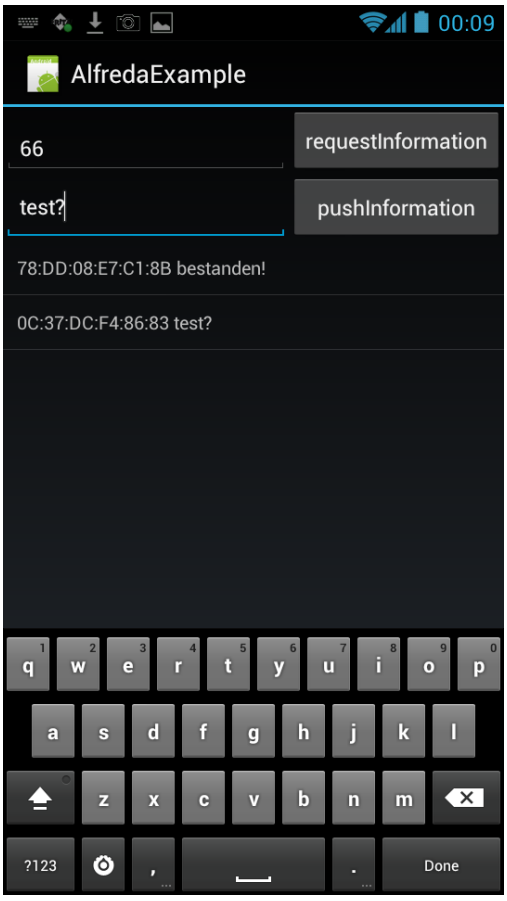

Abbildung A.1.: Screenshot von AlfredaExample

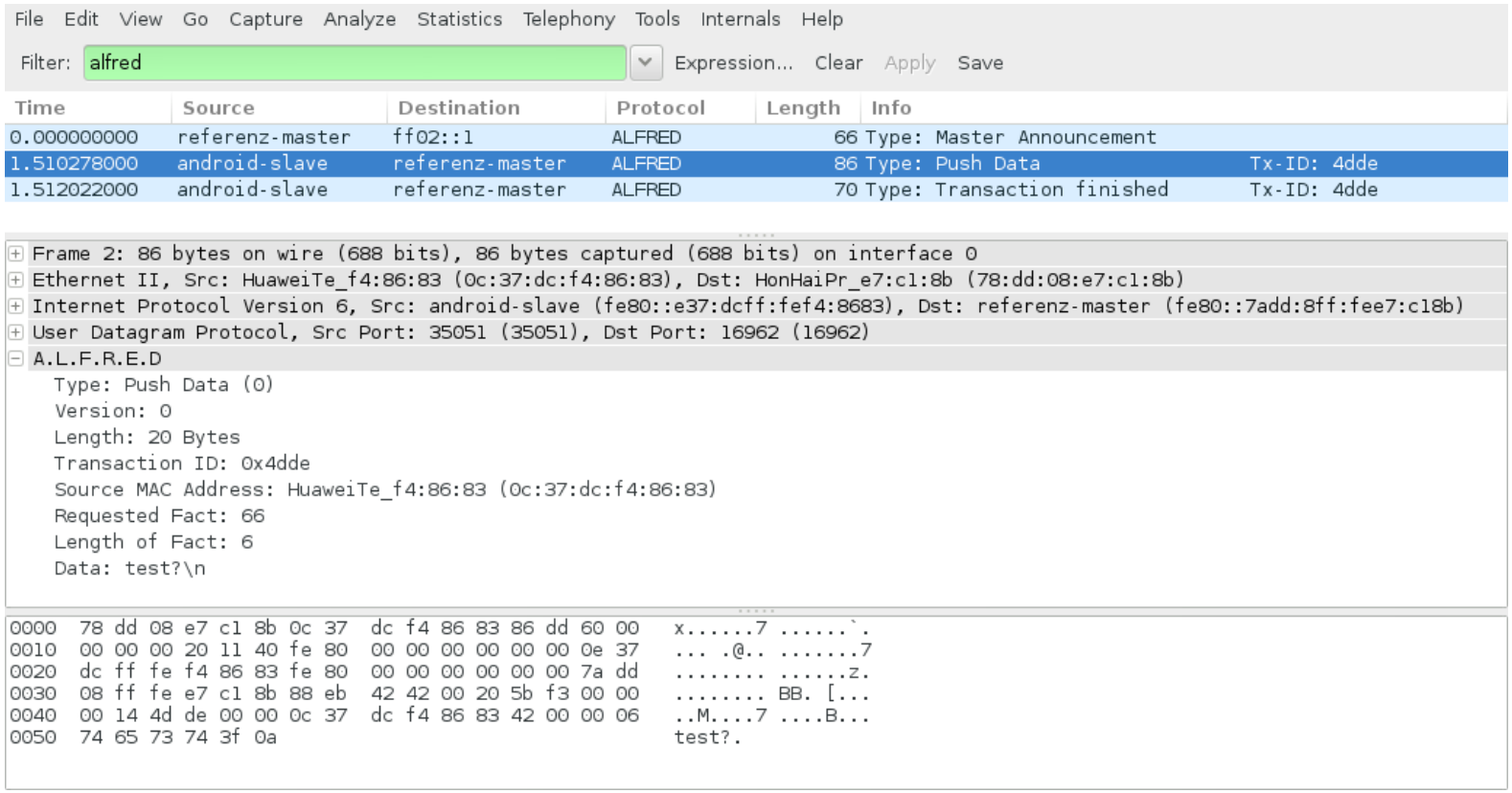

Abbildung A.2.: Screenshot von Wireshark Dissector mit Push-Packet von AlfredA

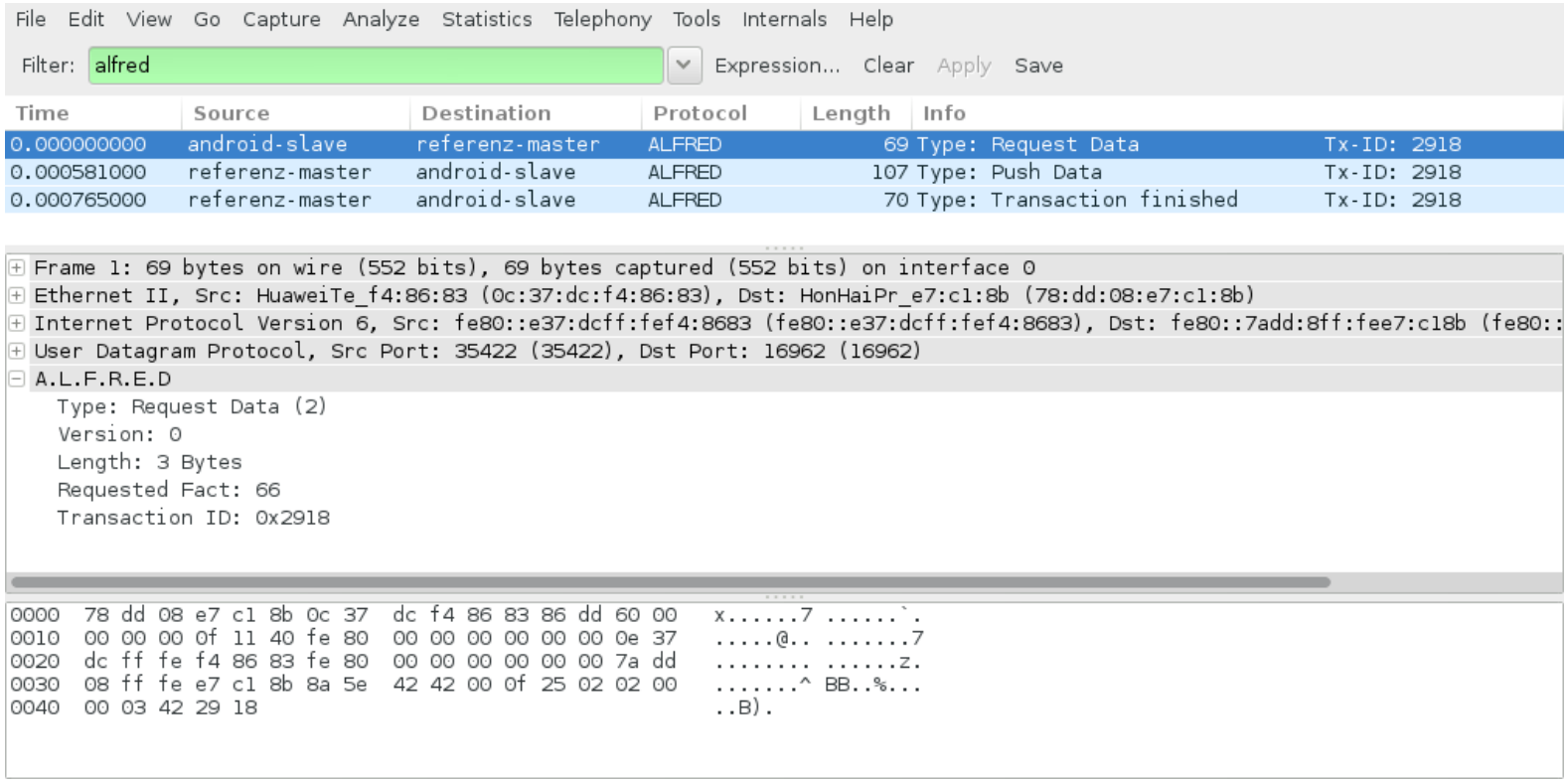

Abbildung A.3.: Screenshot von Wireshark Dissector mit Request-Packet von AlfredA

Abbildungsverzeichnis

Literaturverzeichnis

[AGS11] Michael Adeyeye and Paul Gardner-Stephen. The Village Telco project: a reliable and practical wireless mesh telephony infrastructure. *EURASIP Journal on Wireless Communications and Networking 2011*, 2011.

[BHSW07] Rainer Baumann, Simon Heimlicher, Mario Strasser, and Andreas Weibel. A survey on routing metrics. TIK Report 262, Computer Engineering and Networks Laboratory—ETH-Zentrum, Switzerland, Zürich, February 2007.

[CJ03] T. Clausen and P. Jacquet. Optimized link state routing protocol (olsr). Technical Report 3626, October 2003.

[CK08] J. Camp and E.W. Knightly. The IEEE 802.11s extended service set mesh networking standard. *Communications Magazine, IEEE*, 46(8):120–126, August 2008.

[CKA08] Ian Chakeres, Chivukula Koundinya, and Pankaj Aggarwal. Fast, efficient, and robust multicast in wireless mesh networks. In *Proceedings of the 5th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks*, PE-WASUN '08, pages 19–26, New York, NY, USA, 2008. ACM.

[CM99] S. Corson and J. Macker. Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations. RFC 2501 (Experimental), January 1999.

[DPZ04] Richard Draves, Jitendra Padhye, and Brian Zill. Comparison of routing metrics for static multi-hop wireless networks. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 133–144, New York, NY, USA, 2004. ACM.

[Egn10] Andre Egners. Evaluating ieee 802.11s against security requirements of wireless mesh networks. *Essener Workshop zur Netzsicherheit 2010 (EWNS10)*, 2010.

[Fur11] Daniele Furlan. Improving batman routing stability and performance. Master's thesis, University of Trento, 2011.

[HD06] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052, 7136.

[HDM+10] G.R. Hiertz, D. Denteneer, S. Max, R. Taori, J. Cardona, L. Berlemann, and B. Walke. IEEE 802.11s: The wlan mesh standard. *Wireless Communications, IEEE*, 17(1):104–111, February 2010.

[HLP11] Martin Hundebøll and Jeppe Ledet-Pedersen. Inter-Flow Network Coding for Wireless Mesh Networks. Master's thesis, Aalborg University, 2011.

[HMZ+07] G.R. Hiertz, S. Max, Rui Zhao, D. Denteneer, and L. Berlemann. Principles of IEEE 802.11s. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 1002–1007, Aug 2007.

[LCS05] Dongwook Lee, Gayathri Chandrasekaran, and Prasun Sinha. Optimizing broadcast load in mesh networks using dual-association. In *1st IEEE Workshop on Wireless Mesh Networks WiMESH, Santa Clara, CA, USA*, 2005.

[Mac13] Leonardo Maccari. An analysis of the ninux wireless community network. In *WiMob*, volume 9, pages 1–7. IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, Oktober 2013.

[MS14] Reto Mantz and Thomas Sassenberg. *WLAN und Recht - Aufbau und Betrieb von Internet-Hotspots*. Erich Schmidt Verlag, Mai 2014. ISBN 978 3 503 15660 3.

[MTKM09] AF. Molisch, F. Tufvesson, J. Karedal, and C.F. Mecklenbrauker. A survey on vehicle-to-vehicle propagation channels. *Wireless Communications, IEEE*, 16(6):12–22, December 2009.

[RC05] Ashish Raniwala and Tzi-cker Chiueh. Architecture and algorithms for an ieee 802.11-based multi-channel wireless mesh network. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2223–2234. IEEE, 2005.

[RRN09] V. Rastogi, V.J. Ribeiro, and A.D. Nayar. Measurements in OLPC mesh networks. In *Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks, 2009. WiOPT 2009. 7th International Symposium on*, pages 1–6, June 2009.

[Sir14] Mohammad Siraj. A survey on routing algorithms and routing metrics for wireless mesh networks. *World Applied Sciences Journal*, 30(7):870–886, 2014.

[SL05] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester, Foundation Level nach ISTQB-Standard (3. Aufl.)*. dpunkt.verlag, 2005.

[SN06] M. Stojmenovic and A Nayak. Broadcasting and routing in faulty mesh networks. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[Spi08] Øyvind Spigseth. Introducing name resolution into OLSR. Master's thesis, UniK University of Oslo, 2008.

[TW11] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall, 5th edition, 2011.

[Tø04] Andreas Tønnensen. Implementing and extending the optimized link state routing protocol. Master's thesis, UniK University of Oslo, 2004.

[Wag09] Elektra Wagenrad. The olsr.org story. `http://www.open-mesh.org/projects/open-mesh/wiki/The-olsr-story`, March 2009. Letzter Zugriff: 27.06.2014.

[XHF13] Dong Xia, J. Hart, and Qiang Fu. Evaluation of the Minstrel rate adaptation algorithm in IEEE 802.11g Wlans. In *Communications (ICC), 2013 IEEE International Conference on*, pages 2223–2228, June 2013.

[YLL09] Zhenyu Yang, Ming Li, and Wenjing Lou. A network coding approach to reliable broadcast in wireless mesh networks. In Benyuan Liu, Azer Bestavros, Ding-Zhu Du, and Jie Wang, editors, *Wireless Algorithms, Systems, and Applications*, volume 5682 of *Lecture Notes in Computer Science*, pages 234–243. Springer Berlin Heidelberg, 2009.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum	Unterschrift