SharkNet
Systembeschreibung
Version 0.1.6

Dustin Feurich

19. Januar 2018

Inhaltsverzeichnis

Kapitel 1

Überblick

1.1 Einleitung

Durch die rasante Entwicklung des Internet of Things (IoT) ist das Interesse an einen semantischen Datenaustausch spürbar gestiegen. Wurde in den letzten Jahrzehnten noch fast ausschließlich klassisch über die Zieladresse der Datenpakete geroutet, so werden jetzt auch die Metadaten dieser Datenpakete beim Routing zunehmend beachtet. Das Routing erfolgt hierbei also inhaltsbasiert und ermöglicht ein Routing nach den Interessen der Kommunikationsteilnehmer. Der Datenaustausch zwischen diesen Teilnehmern kann beim inhaltsbasierten Routing sowohl per klassischer Client-Server Architektur, als auch Peer-To-Peer (P2P) erfolgen. In dieser Arbeit wird der Datenaustausch über P2P erfolgen, was mehrere Vorteile bietet:

- Die Verbindungen zwischen Kommunikationsteilnehmern (Peers) können spontan aufgebaut werden, es wird keine Serverinfrastruktur benötigt.
- Die Daten liegen ausschließlich bei den Peers selbst. Da es keine Zwischenstation für die Datenpakete gibt, erhöht dies die Vertraulichkeit der Kommunikation immens.
- Nahezu alle Kommunkationsanwendungen verwenden das Internet um den Datenaustausch zu ermöglichen. Eine Verbindung mit dem Internet ist jedoch nicht zu jeder Zeit und an jedem Ort verfügbar. Weiterhin kann auch hier auf den Zwischenservern die Kommunikation gespeichert und an Dritte weitergegeben werden.

Kapitel 2

Verwandte Veröffentlichungen

Es gibt zahlreiche wissenschaftliche Paper, die Semantisches oder Inhaltsbasiertes Routing zum Thema haben. Viele diese Paper sind jedoch entweder schon mindestens zehn Jahre alt, oder beinhalten nicht exklusiv den Datenaustausch über Peer-To-Peer. Im Folgenden wird jeweils die Grundidee von vier Arbeiten vorgstellt, welche ausschließlich den semantischen Datenaustausch über P2P zum Inhalt haben.
Strassner et. al. präsentieren ein hybrides Routing, bei dem sowohl semantisch als auch traditionell geroutet wird. Die Peers bauen hierbei ein *small world* Netzwerk auf, bei dem jeder Peer viele kurze und nur wenige lange Verbindungen zu anderen Peers hat. Es werden zwei semantische Strukturen definiert - *node profiles* und *object profiles* - welche beide anhand von Metadaten beschrieben werden. Ein Interesse wird mit Hilfe des *node profils* formuliert, das dann an die anderen Peers direkt geschickt wird. Interessiert sind die Peers an die Objekte. Durch eine semantische Ähnlichkietsanalyse wird überprüft, ob ein Peer entweder direkt ein Objekt an den anfragendne Peer liefert, oder ob er das *node profile* an andere Peers weiterleitet. Das *node profile* wird an den Peer weitergeleitet, bei dem die Ähnlichkeitsanalyse zwischen *node profile* und *object profile* am höhsten ist und sich außerdem physisch in Reichweite befindet.
David Faye et. al. stellen in Ihrer Ausarbeitung ein semantisches und abfrageorientiertes (Query) Routing vor. Die neuartige semantische Struktur ist hierbei die *expertise table*, in der mit Metadaten festgehalten ist, welcher Peer über welches Wissen verfügt. Anders als in Sharknet sind die Peers nicht gleichberechtigt, sondern in zwei Kategorien eingeteilt: normale Peers und Super-Peers. Ein Super-Peer verwaltet mehere normale Peers und besitzt dafür eine *expertise table*. Sie reichen die Anfragen entweder an andere Super-Peers weiter oder lassen diese von normalen Peers auswerten. Ein Interesse wird mit Hilfe einer Anfrage gestellt, diese Anfrage wird durch den Routingalgorithmus an das relevante Ziel gesendet. Dies läuft folgendermaßen ab:

- Ein Peer fomuliert sein Iteresse mit einer Query und sendet diese an seinen zuständigen Super-Peer, der im Paper als *Godfather* bezeichnet wird.
- Der *Godfather* wertet nun mit der Query und den *expertise tables* aller verfügbaren anderen Super-Peers aus, an welche Super-Peers er die Query weiterreicht.
- Nachdem ein Super-Peer auf dieser Art eine Query erhalten hat, kann er diese nun entweder abermals an andere Super-Peers weiterleiten oder sie von einen seiner zugeordneten Peers ausführen lassen.
- Das Ergebnis der Ausführung wird nun an den eigentlichen Absender der Query zurückgeleitet.

Einen anderen Ansatz mit komplett gleichberechtigten Peers stellt Antonio Carzaniga et. al. vor, bei dem parallel zwei Protokolle ausgeführt werden. Dies umfasst zum einen das *Broadcast Routing Protocol* und zum anderen das *Content-based Routing Protocol*. Das Broadcast Protokoll ist für das physische Versenden der Nachrichten zwischen den Peers verantwortlich und baut eine Spanning-Tree Topologie auf. Die Nachricht wird zunächst ohne Einschränkung an alle Peers geschickt, die erreichbar sind. Das eigentliche Routing geschieht durch das *Content-based Routing Protocol*. Folgende semantische Strukturen werden benutzt:

- Eine *Message* besteht aus typisierten Attributen
- Ein *predicate* ist eine Disjunktion von Konjunktionen von Bedingungen (constraints), die sich auf einzelne Attribute beziehen
- Die *content-based forwarding table* enthält die von den Peers gesetzten *predicates*

Eine Funktion wertet anhand der *forwarding table* aus, an welche Peers die Nachricht weitergeleitet werden soll. Zusätzlich wird durch das Broadcast Protokoll ermittelt, welche Peers sich physisch in Reichweite befinden. Die Nachricht wird nun alle Peers geschickt, die in beiden Mengen vorkommen. Diese Funktionsweise ähnelt SharkNet, da in der Anwendung die Nachrichten ebenfalls per Broadcast verschickt werden. Die semantische Auswertung erfolgt in SharkNet jedoch durch Profile, die vom Nutzer dynamisch festgelegt werden können und nicht durch eine sich automatisch aufbauende Tabelle.
Luca Mottola et. al. haben eine sich selbst reparierende Baumtopolgie entworfen, mit der inhaltsbasiertes Routing in mobilen Ad Hoc Netzwerken realisiert werden kann. Laut Mottola et. al. benötigt eine Topologie in Form eines Baums bei ad hoc Netzwerken eine stetige Selbstreparatur, die durch das dynamische Entfernen und Hinzufügen von mobilen Geräten notwendig sei. Diese Topologie wird während der Programmausführung auf den

Peers stetig angepasst, um auch bei einem häufigen Peerwechsel weiterhin benutzbar zu sein. Die Baumstruktur ist dabei für das inhaltsbasierte Routing essentiell. Das Routing erfolgt über das publish-subcribe Prinzip, wobei die Peers Nachrichten zu den Themen bekommen, die sie für die sie sich angemeldet (subscribt) haben.
Der wesentliche Unterschied zwischen den vorgestellten Veröffentlichungen und dieser Arbeit sind einerseits die Eingangs- und Ausgangsprofile, mit denen natürliche Personen eingehende und ausgehende Nachrichten semantisch filtern können und andererseits die Präsentation einer konkreten mobilen Applikation, die diese Art des Routings verwirklicht. Außerdem unterscheiden sich die dafür verwendeten semantischen Strukturen deutlich von anderen Veröffentlichungen.
Da diese Arbeit jedoch nicht nur das semantische Routing, sondern mit Sharknet auch ein dezentrales Netzwerk realisiert, soll an dieser Stelle kurz das bereits bekannte dezentrale soziale Netzwerk Diaspora vorgestellt werden.
Jeder Benutzer kann in Diaspora einen eigenen Server benutzen, welche als Pod bezeichnet werden. Diese Pods beinhalten die Benutzerdaten und werden vom Besitzer des Pods verwaltet. Der umfassende Datenschutz ist bei Diaspora jedoch nur dann gegeben, wenn jeder Benutzer auch einen eigenen Webserver benutzt, um damit seinen Pod zu hosten. In der Realität wird häufig aber kein eigener Webserver benutzt, außerdem ist die direkte Kommunikation zwischen den Pods nur eingeschränkt möglich. So lassen sich zum Beispiel keine Kontaktlisten von anderen Pods crawlen, auch wenn diese sie zur Verfügung stellen würden. Dies hat zur Folge, ein Teil der Benutzer sich ausschließlich mit anderen Pods verbinden, die dann zu Sammelpods werden.

Kapitel 3

Grundlagen

3.1 Routing

3.1.1 Traditionelles Routing

Routingalgorithmen werden benötigt, um Pfade für den Datenverkehr innerhalb eines Netzwerks zu finden. Beim traditionellen Routing erstellt jeder Router eine Routingtabelle, die Netzwerkadressen und Netzmasken enthält. Anhand dieser Tabelle, bei der diese Adressen auch Geräte zugeordnet sind, können dann Pakete durch das Netzwerk weitergeleitet bzw. geroutet werden. Bei der Suche nach einer geeigneten Route durch das Netzwerk wird klassischerweise ein Longest Prefix Match durchgeführt. Bei einem Treffer wird das Paket im entsprechenden output port weitergeleitet, ansonsten wird der default link genommen. Die Routingtabellen werden durch eine Analyse der vorhandenen Netzwerktopologie aufgestellt. Sollte sich die Topolgie im Betrieb ändern, muss daher auch die Routingtabelle angepasst werden. Routingschemata werden üblicherweise als Graphen dargestellt, wobei die Knoten die Kommunikationsteilnehmer und die Kanten die Leitungen zwischen den Teilnehmern darstellen. Die Kanten enthalten auch Informationen über die Kosten für die Paketübertragung zwischen Knoten. Die Kosten beziehen sich dabei meistens auf die physikalische Länge der Leitung, je länger die Leitung desto höher sind auch die Kosten. Da es bei der Wegfindung zwischen zwei nicht direkt benachbarter Knoten häufig mehrere Alternativen gibt, werden die nötigen Gesamtkosten die zwischen Anfangs- und Zielknoten auftreten, bei der Wahl des Pfades berücksichtigt. Beim traditionellen Routing wird also vorrangig nach den kostengünstigsten Pfaden zwischen Knoten innerhalb eines Netzwerks gesucht.
Routing-Algorithmen lassen sich in zwei Klassen aufteilen:

- Globaler Routing-Algorithmus: Hierbei ist das komplette Netzwerk bereits vor der

Berechnung der kostengünstigsten Route bekannt. Es können direkt alle möglichen Pfade zwischen Ausgangs- und Zielknoten und deren Gesamtkosten bestimmt werden.

- Dezentraler Routing-Algorithmus: Die Knoten haben hierbei nur Informationen über die Knoten und Kanten, welche sich in der Nähe befinden, das komplette Netzwerk ist nicht bekannt. Das Finden eines geeigneten Pfades kann also nur iterativ von Knoten zu Knoten geschehen und nicht bereits im Voraus.

Da das Ziel dieser Arbeit ein semantischer Broadcast ist, bei dem sich die Kommunikationspartner vorher nicht kennen müssen, wird es sich bei dem Algorithmus um einen dezentralen Routing-Algorithmus handeln.
Eine weitere wichtige Unterscheidung bei Routing-Algorithmen ist die Frage, ob diese statisch oder dynamisch gestaltet sind:

- Statische Routing-Algorithmen werden benutzt, wenn sich die Kanten zwischen den Knoten nur selten ändert. Die Netzwerktopologie bleibt also konstant.

- Bei Dynamischen Routing-Algorithmen werden bei sich häufig ändernden Netzwerktopologien eingesetzt. Sie müssen anders als die statischen Algorithmen den stetigen Wechsel von Knoten, Kanten und Kosten beachten.

Das Ergebnis dieser Arbeit wird in einer mobilen Applikation eingebunden werden und soll von natürlichen Personen per Smartphone bedient werden können. Der Routing-Algorithmus muss also zwingend dynamisch sein, da Menschen mit ihren Smartphones anders als andere Netzwerkgeräte konstant in Bewegung sind. Da bei einem Broadcast unabhängig von den Kosten die Nachricht zunächst an alle Personen geschickt werden wird, handelt es sich zusammengefasst um einen dezentralen, dynamischen und lastinsensitiven Routing-Algorithmus.

3.1.2 Inhaltsbasiertes Routing

Beim inhaltsbasierten Routing wird für die Bestimmung des Pfads nicht die Zieladresse des Pakets ausgewertet, sondern die semantische Beschreibung des Paketinhalts. Jedes Paket verfügt daher über eine Inhaltsbeschreibung, wobei dann alleine diese Beschreibung abhängig ist, an welchen Knoten die Nachricht weitergeleitet wird. Eine wichtige Voraussetzung für diese Art von Routing ist ein Peer-To-Peer (P2P) Netzwerk, da die Pakete stets nur von Punkt zu Punkt gesendet werden. Die Peers müssen ebenfalls ein Interesse formulieren können, anhand dessen bestimmt werden kann, ob ein Paket für sie relevant ist. Dies geschieht meist über themengesteuerte Abonnements, Peers können über diese

Abonnements ihr Interesse an einer bestimmten Gruppe von Paketen bekunden. Es gibt zwei unterschiedliche Arten, diese Abonnements innerhalb eines Netzwerks zu verwalten:

- Die Abonnements werden ausschließlich vom Peer selbst für die Auswertung herangezogen, andere Peers können diese nicht berücksichtigen. Die Pakete müssen nach der Auswertung durch den Peer dann an alle verfügbaren Peers in der Nähe weitergeleitet werden, da deren Abonnements unbekannt sind.

- Die Peers teilen Ihre Abonnements den anderen Teilnehmern im Netzwerk mit. Dadurch können die Pakete nun gezielt nur an die Peers weitergeleitet werden, die sich für das Paket auch interessieren.

Beide Lösungsansätze haben Vor- und Nachteile. So hat der zweite Ansatz den Vorteil, dass nicht bei jedem Peer eine semantische Prüfung der Paketbeschreibung erfolgen muss, da diese Filterung bereits beim sendenden Peer vorgenommen worden ist. Der entscheidende Nachteil dieses Ansatzes ist jedoch, dass sämtliche Modifikationen an bestehenden Abonnements jedem Peer im Netzwerk mitgeteilt werden muss. Bei der App dieser Arbeit ist davon auszugehen, dass Benutzer ihre Abonnements (bzw. Profile) häufig editieren, was zu einer Flut an Benachrichtigungen zu anderen Peers führen könnte. Es ist bei dem Ad-Hoc Broadcast auch nicht realistisch davon auszugehen, dass jeder Peer die Abonnements der anderen Peers kennt, bevor der Benutzer eine Nachricht versendet. Da das Protokoll vorrangig von leistungsstarken Smartphones und nicht von eher leistungsschwachen Kleinstgeräten benutzt werden soll, ist der durch die wiederholte semantische Auswertung der Paketbeschreibung nötige Aufwand vernachlässigbar. In dieser Arbeit wird daher der erstgenannte Lösungsansatz verfolgt.

3.1.3 Broadcast-Routing

Wenn ein Paket an alle interessierten Peers innerhalb eines Netzwerks verschickt werden soll, wird ein Broadcast-Routing benötigt, da das bisher beschriebene Unicast-Routing nur die Wegfindung zwischen zwei Knoten realisiert. Häufig sind für einen Knoten nicht alle anderen Knoten des Netzwerks direkt erreichbar, es werden also Zwischenstationen benötigt welche die Nachricht weiterleiten. Anders als beim Unicast-Routing ist die Anzahl der Pfade also variabel, je nachdem wie viele Knoten das Paket an ihre Nachbarknoten weiterleiten. Wenn ein Knoten eine Nachricht an alle Nachbarknoten schickt und diese sie wiederum an ihre Nachbarknoten schicken, wird dieser Ansatz *flooding* genannt. *Flooding* kann bei unbedachtem Einsatz zu Schleifen führen. Hierbei erhält und versendet ein Knoten wiederholt Nachrichten, die er bereits verwertet hat. Dieser endlose Ping-Pong-Effekt zwischen den Nachbarknoten führt dann zum sogenannten *Broadcast storm*, der

das gesamte Netzwerk unbrauchbar macht. *Flooding* muss also zwingend kontrolliert werden, die Knoten müssen unabhängig von der semantischen Überprüfung der Nachrichten prüfen können, ob sie eine eingehende Nachricht bereits verwertet haben. In der Praxis wird meistens einer der folgenden drei Lösungsansätze für dieses Problem gewählt:

- Beim sequenznummerkontrollierten *Flooding* schickt jeder Knoten seine Adresse und eine Broadcast-Sequenznummer an seine Nachbarknoten. Dadurch kann jeder Knoten eine Tabelle anlegen, die bereits empfangene Pakete den Nachbarknoten zuordnet. Bei eingehenden Paketen wird nun vorher überprüft, ob dieses Paket bereits in der Liste eingetragen ist.

- Das *Reverse Path Forwarding* (RPF) lässt die Pakete nur dann an Nachbarknoten weiterleiten, wenn der das Paket absendende Knoten das Paket über den kürzesten Pfad erhalten hat. Es werden alle Pakete verworfen, die nicht auf dem kürzesten Unicast-Pfad zurück zur Quelle liegen. Die Nachricht wird außerdem nicht an den Nachbarknoten weitergeleitet, der auf diesen kürzesten Pfad liegt.

- Der wohl bekannteste Ansatz ist der Aufbau eines *Spanning Tree*, der alle Knoten genau einmal enthält. Die Knoten leiten die Nachrichten nur an ihre Baumnachbarn weiter. Dadurch können Schleifen vollständig vermieden werden.

In dieser Arbeit wird eine angepasste Variante des sequenznummerierten *flooding* benutzt, um *Broadcast storms* auszuschließen. Dies wird in Unterkapitel x.x genauer erläutert.

3.2 Shark

3.2.1 Shark Framework

Das Protokoll soll auf der Basis des Shark Frameworks entwickelt werden. Das Framework wurde von Prof. Dr.-Ing. Thomas Schwotzer entworfen, um die Entwicklung von semantischen Peer-To-Peer Anwendungen zu erleichtern. Es ist mit seinen semantischen Strukturen und Auswertungsmethoden für dezentrale Anwendungen geeignet. Das Wort Shark steht für Shared Knowledge - Verteiltes Wissen.
Das Framework definiert, dass jeder Benutzer (Peer) über eigene Wissensbasis verfügt, welche mit semantischen Annotationen versehenes Wissen des Benutzers speichert. Jede in der Wissensbasis gespeicherte Information muss daher auch semantisch beschrieben werden, bevor es in der Wissensbasis abgelegt werden kann. Informationen werden semantisch mit Wörtern beschrieben, wofür im Framework die Klasse *Semantic Tag* und von dieser Klasse ableitende Klassen benutzt werden. Es werden *Semantic Tags* statt normale

Zeichenketten benutzt, da die fast jede Sprache semantische nicht eindeutige Wörter wie beispielsweise Homonyme aufweist. Die Tags können innerhalb von Behältern gespeichert werden, wobei es drei Arten von Behälter gibt:

- *Sets* enthalten *Semantic Tags* ohne Beziehungen zwischen den Tags zu speichern.
- *Taxonomies* speichern zusätzlich zu den Tags noch gerichtete Beziehungen. Diese gerichteten Beziehungen zwischen den Tags können entweder den Wert *sup* oder *sub* annehmen und ermöglichen somit eine hierarchische Anordnung der Wörter.
- Das *Semantic Net* verhält sich wie eine *Taxonomy*, die Beziehungen können hierbei aber beliebige Werte (in Form von Zeichenketten) annehmen. Dadurch können beispielsweise Verwandschaftsbeziehungen dargestellt werden.

Die Behälterklassen werden dann dazu benutzt, die Informationen zu beschreiben. Informationen werden mit Hilfe von sieben Dimensionen beschrieben, wobei dafür bis zu sieben Behälter und ein Literal verwendet werden.

Tabelle 3.1: Dimensionen einer Information

Dimension	Definition
Topics	Thematische Beschreibung der Information
Types	Um was für eine Art handelt es sich bei der Information
Approvers	Welche Peers haben diese Nachricht
Receivers	An welche Peers ist die Information gerichtet
Senders	Welche Peers haben diese Nachricht versendet
Locations	An welchen Orten ist diese Information relevant
Times	In welchen Zeiträumen ist diese Information relevant
Direction	Literal welches den Eingang und Ausgang der Information bestimmt

Dieses Unterkapitel ist nur eine rudimentäre Zusammenfassung über das Shark Framework, einen ausführlichen Überblick über das Framework bietet der Shark Developer Guide.

3.2.2 ASIP

Innerhalb der letzten drei Jahre wurde ein grundlegendes Protokoll für Shark entwickelt, welches die zwei zentralen Befehle bezüglich der Kommunikation zwischen Peers vorgibt und den Namen *Ad hoc Semantic Internet Protocol* trägt. Die ebenfalls vom Protokoll für Shark neu eingeführten Strukturen können im entsprechenden Repository auf Github eingesehen werden.[x] In der folgenden Abbildung alle Bestandteile der semantischen Strukturen in ASIP abgebildet.

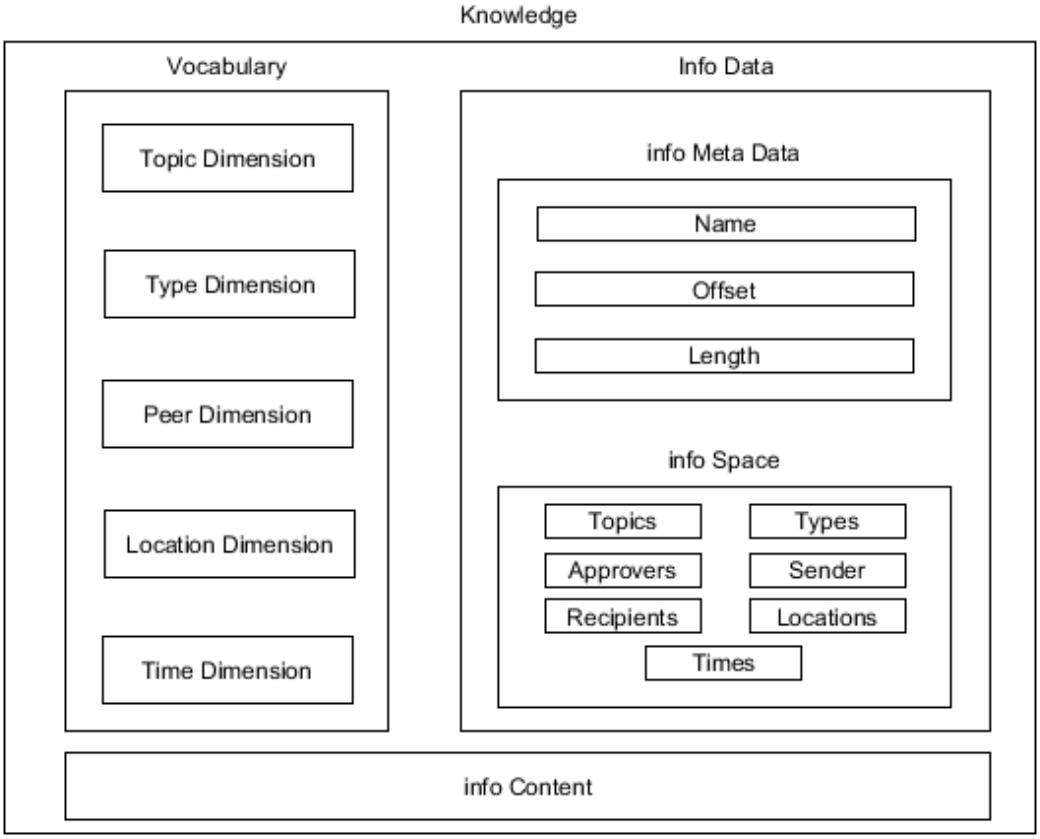

Abbildung 3.1: Die ASIP/Shark Bestandtteile im Überblick

Die beiden wichtigsten Kommandos des Protokolls sind:

- Insert: Dieser Befehl wird dazu benutzt, um neue Informationen (bzw. Wissen) von anderen Peers der eigenen Wissensbasis hinzuzufügen. Dieses Wissen ist folgendermaßen unterteilt:
  - Das Vokabular des Peers welches alle ihm bekannten Wörter enthält. Die Wörter sind wiederum in die fünf Dimensionen Topic, Type, Peer, Location und Time unterteilt.
  - Der eigentliche Informationsinhalt in Form eines byte Streams mit Rohdaten.
  - Technische Metadaten über den Informationsinhalt wie beispielsweise die Anzahl der Bytes
  - Semantische Metadaten über den Informationsinhalt in Form der in Tabelle x.x beschriebenen sieben Dimensionen, technisch umgesetzt mit Behältern von *Semantic Tags*.

- Expose: Neben dem Hinzufügen von neuen Wissen haben Peers auch die Möglichkeit, ihr Interesse an neuem Wissen gegenüber anderen Peers zu bekunden. Dies geschieht über den Befehl *Expose*, wobei auch hier das Interesse in Form der in Tabelle x.x dargestellten sieben Dimensionen formuliert wird.

[...]

3.2.3 SharkNet

SharkNet ist ein dezentrales soziales Netzwerk für Android Geräte und wurde von Michael Schwarz und Prof. Dr.-Ing. Thomas Schwotzer von 2015 bis 2017 entwickelt. Es kann durch die folgenden drei Kernaspekte beschrieben werden:

- Dezentraler Datenaustausch ohne der Verwendung eines Servers
- Eine Public-Key-Infrastruktur, womit die Kommunikationspartner sich gegenseitig authentifizieren können
- Ausschließliche Benutzung von Open-Source Bilbiotheken und Protokollen

Sharknet bildet die Grundlage für diese Arbeit und wurde an diversen Stellen weiterentwickelt, wobei auch einige Probleme im Bereich der Kommunikation zwischen den Peers behoben werden mussten. Die ursprüngliche Zielgruppe von SharkNet sind Schüler der Katholischen Theresienschule Berlin, die als Testpersonen SharkNet anstelle von Facebook oder anderen servergebundenen sozialen Netzwerken nutzen sollten. Über die Webseite `http://sharedknowledge.github.io/` kann bereits ein Prototyp heruntergeladen werden, dieser enthält aber noch nicht die eigentliche Kernfunktionalität, daher keinen Chat bzw. Gruppenchat. Ein wichtiger Bestandteil dieser Arbeit ist es daher, neben dem semantischen Broadcast auch die normale Chatfunktionalität für den Endanwender benutzbar zu machen.
[...]

Kapitel 4

Entwurf

4.1 Aufbau der Datenpakete

4.1.1 Überblick

Es handelt sich bei dem zu entwickelnden Protokoll um ein Nachrichtenprotokoll. Daher wird der aus anderen Protokollen bekannte Begriff Datenpaket hier als eine Nachricht (Message) definiert. Diese Nachricht wird in zwei Hauptteile gegliedert. Den ersten Teil bildet der Nachrichtenheader, während der zweite Teil durch eine Instanz der Klasse *Knowledge* gebildet wird.

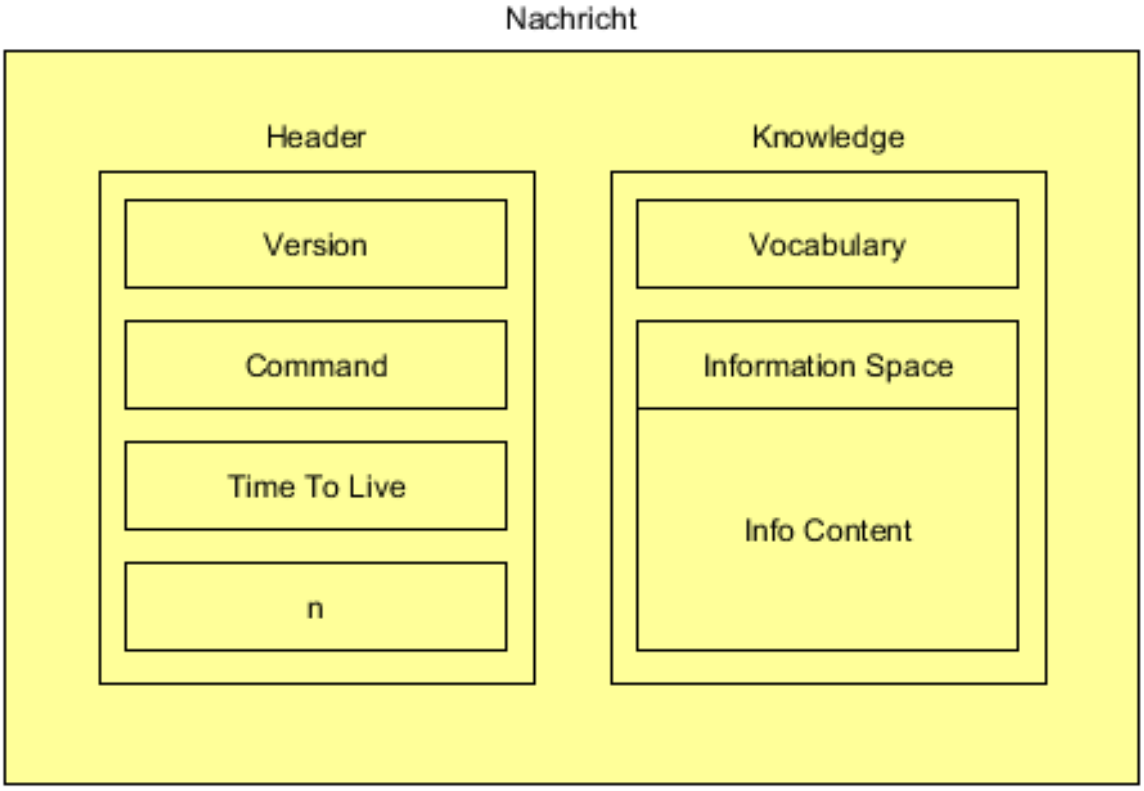

Abbildung 4.1: Die zwei Hauptbestandteile einer Nachricht

4.1.2 Header

Der Nachrichtenheader wird im Gegensatz zu anderen klassischen Routingprotokollen nicht zum Routing eingesetzt, dies ist allein von den semantischen Annotationen innerhalb des *Knowledge* abhängig. Der Header ist dennoch unabdingbar, da er eine Nachricht als SharkNet Broadcast-Nachricht kennzeichnet und die Verarbeitungsart der Nachricht kennzeichnet. Der Header enthält folgende Bestandteile:

Tabelle 4.1: Bestandteile des Headers

Bestandteil	Erläuterung
Version	aktuelle Version des Protokolls
Format	Format der Nachricht, standardmäßig JSON
TTL	Time To Live Wert der Nachricht
Command	Verarbeitungsart der Nachricht, Insert oder Expose
Physical Sender	Absender der Nachricht
Receiver Peer	Zielgerät der Nachricht
Receiver Location	Ort des Absenders beim Versenden der Nachricht
Receiver Time	Zeitpunkt des Absendens der Nachricht
Topic	Sequenznummer, wird nicht vom semantischen Filter ausgewertet
Type	Kennzeichnung der Nachrichtenart

Die obligatorischen Headerbestandteile sind: Version, Format, Command, Receiver Peer und Type. Der Command ist für das Versenden der Nachrichten auf *Insert* gestellt, damit die Empfangsgeräte die neue Nachricht nach erfolgreicher Filterung in ihre Wissensbasis einfügen. Der zweite Command *Expose* wird fast ausschließlich von der WiFi-Komponente zum Austausch von Kontaktinformationen benutzt. Auch wenn es sich um einen Broadcast handelt, muss das Zielgerät stets im Header eingetragen sein, die Nachricht kann sonst nicht zugestellt werden. Wenn sich also beispielsweise fünf Geräte in der Nähe befinden, werden fünf Nachrichten mit angepasstem Header verschickt. Der Type markiert die Nachricht als eine Broadcast-Nachricht, damit diese nicht fälschlicherweise als Chat-Nachricht verarbeitet wird.

4.1.3 Knowledge

Der Hauptteil der Nachricht spaltet sich in drei Bereiche auf.

- Das Vocabulary, welches alle Semantic Tags beinhaltet, die die Nachricht beschreiben
- Einen Informationsraum (InformationSpace), der die semantische Beschreibung des Nachrichteninhalts ist. Er besteht aus den in Kapitel zwei beschriebenen sieben Dimensionen mit den dazugehörigen Semantischen Netzen.
- Der eigentliche Nachrichteninhalt (Info Content)

Der in Abbildung 3.1 dargestellte Bereich *Info Meta Data* befindet daher sich wie oben beschrieben im Header und nicht im Knowledge.

4.2 Nachrichtenaustausch

4.2.1 Ohne semantische Filterung

Der Austausch von Nachrichten mittels des Protokolls erfolgt zunächst ungerichtet an alle Kommunikationsteilnehmer, die sich in Reichweite befinden. Jeder Teilnehmer entscheidet selbst, ob er eine Nachricht seiner Wissensbasis hinzufügt und sie ebenfalls an alle in der Nähe befindlichen Geräte sendet. Wie im Kapitel Grundlagen beschrieben, müssen dabei Loops unterbunden werden, da die Kommunikation sonst durch eine Endlosschleife fehschlagen würde. Das folgende Szenario stellt beispielhaft diese Situation ohne Beachtung einer semantischen Filterung dar.

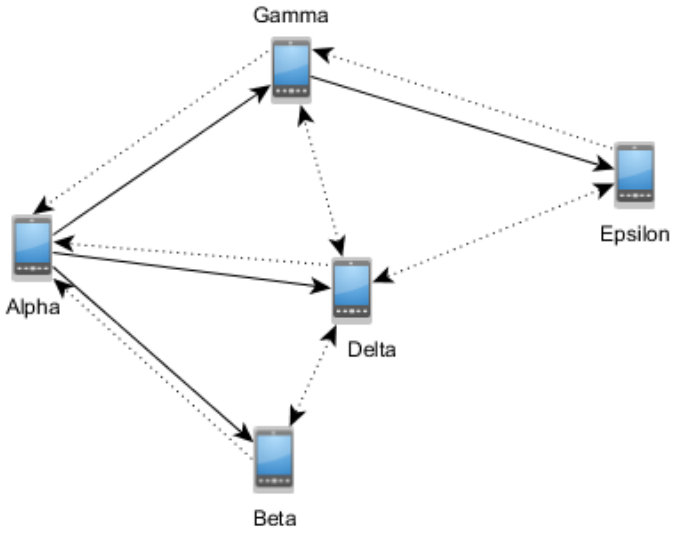

Abbildung 4.2: Kommunikation ohne semantische Filterung

Das Szenario von Abbildung 4.2 beinhaltet die fünf Smartphones Alpha, Beta, Gamma, Delta und Epsilon, welche zusammen ein kabelungebundenes Ad-hoc-Netz bilden. Der Urpsrung der Nachricht ist Alpha, welches eine Nachricht per Broadcast an alle Geräte schickt. Es befinden sich jedoch nur die Geräte Gamma, Delta und Beta in der direkten Reichweite von Alpha. Alpha sendet zunächst an alle Geräte in Reichweite die Nachricht, was jeweils durch die durchgezogene Kante symbolisiert wird. Die Geräte Gamma, Delta und Beta senden nun ihrerseits die empfangene Nachricht an alle Geräte in Reichweite. Dies würde jedoch zu Schleifen innerhalb der Kommunikation führen, falls bereits empfangene oder abgeschickte Nachrichten nicht abgelehnt werden sollten. Dies wird mit gepunkteten Kanten symbolisiert. Eine Ausnahme ist die Nachricht, die von Gamma an Epsilon geschickt wird. Da Epsilon keine Nachricht von Alpha empfangen konnte, wird

die Nachricht von Gamma akzeptiert.

4.2.2 Mit semantischer Filterung

Der Standardfall für die zu entwickelnde Anwendung wird der wiederholte Broadcast mit vorhergehender und nachfolgender semantischer Filterung sein. Jedes Gerät kann durch einen Eingangsfilter und einen Ausgangsfilter festlegen, welche Nachrichten akzeptiert und eventuell zusätzlich an andere Geräte in der Nähe weitergeleitet werden.
Sollte beispielsweise Alpha eine Nachricht versenden, die nicht den Eingangsfilter von Gamma passieren kann, würde sich der eben vorgestellte Kommunikationsablauft nun wie folgt darstellen.

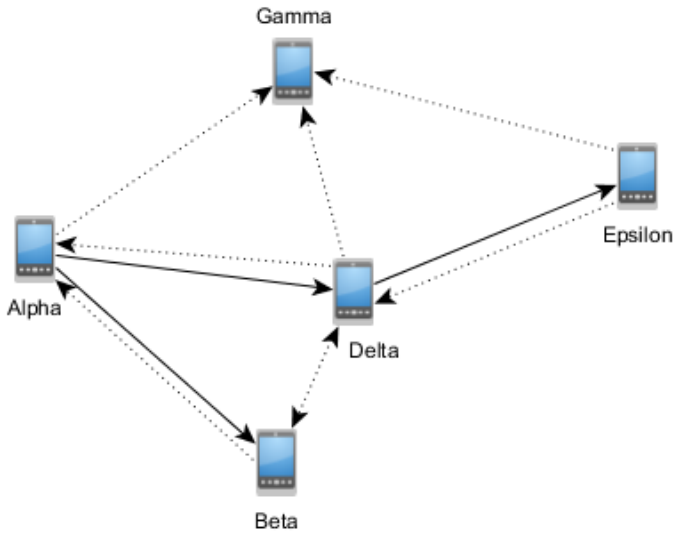

Abbildung 4.3: Kommunikation mit semantischer Filterung

Gamma lehnt die Nachricht aufgrund seines Filters ab, dabei ist es unerheblich von welchem Gerät die Nachricht kommt. Folglich kann Epsilon die Nachricht nicht mehr von Gamma erhalten, stattdessen wird diese nun von Delta empfangen.
Falls mehrere Geräte einen restriktiven Eingangsfilter konfiguriert haben, können Nachrichten teilweise nicht an alle Geräte zugestellt werden, wie das folgende Szenario zeigt.

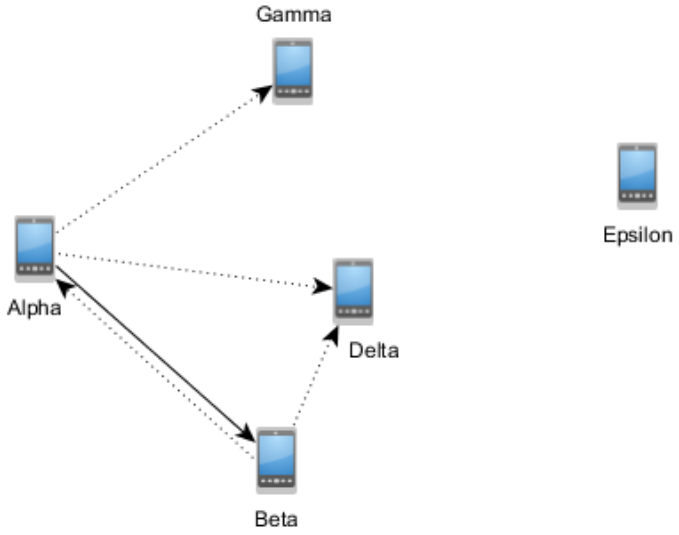

Abbildung 4.4: Kommunikation mit semantischer Filterung

In diesem Szenario lehnen sowohl Gamma als auch Delta die Nachricht ab, nur Beta akzeptiert die Nachricht und versucht diese weiterzuleiten. Da Epsilon sich nicht innerhalb der Sendereichweite von Alpha oder Beta befindet, kann die Nachricht das Gerät nicht erreichen. Die Anzahl der Routen wurde durch die Filterung also deutlich verringert.
Dies mag zunächst problematisch erscheinen, ist aber die Intention des Protokolls. Das hier vorgeschlagene inhaltsbasierte Routing hat nicht das vorrangige Ziel, effiziente Routen zu allen Geräten zu finden, sondern soll allein von den Interessen der Geräte abhängig sein.

4.3 Ablauf der Filterung

Um einen adäquaten Aufbau und Ablauf der Filterung bestimmten zu können, müssen zunächst die Anforderungen an den Filter formuliert werden.

- Es müssen alle Dimensionen (siehe Abbildung 3.1) ausgewertet werden können, die im Shark-Framework/ASIP definiert sind.
- Weiterhin muss dynamisch einstellbar sein, wann welche Dimension durch den Filter ausgewertet wird. Es muss auch nicht zwingend jede Dimension ausgewertet werden, dies ist ebenfalls abhängig vom Benutzer.
- Die Auswertung sollte nur mit Datenstrukturen ausgeführt werden, die aus dem Shark-Framework/ASIP bekannt sind.

- Die Filterung findet nach dem Erhalt der Nachricht und vor der Anzeige auf dem Gerät statt. Der Prozess der Filterung ist daher sehr zeitkritisch und muss möglichst schnell durchgeführt werden können. Sollte ein vom Eingangsfilter abweichender Ausgangsfilter gesetzt sein, muss die Nachricht sogar zweimal gefiltert werden.

- Es muss unbedingt mit Interfaces gearbeitet werden, um die Erweiterbarkeit und Austauschbarkeit des Filterprozesses zu gewährleisten.

Eine Implementierung des Filters muss diese Punkte beachten, in der Komponentenbeschreibung des semantischen Filters finden sich Details zu der im Rahmen dieser Arbeit erstellten Implementierung. [...]
Das folgende Aktivitätsdiagramm skizziert den Ablauf des Nachrichtenempfangs.

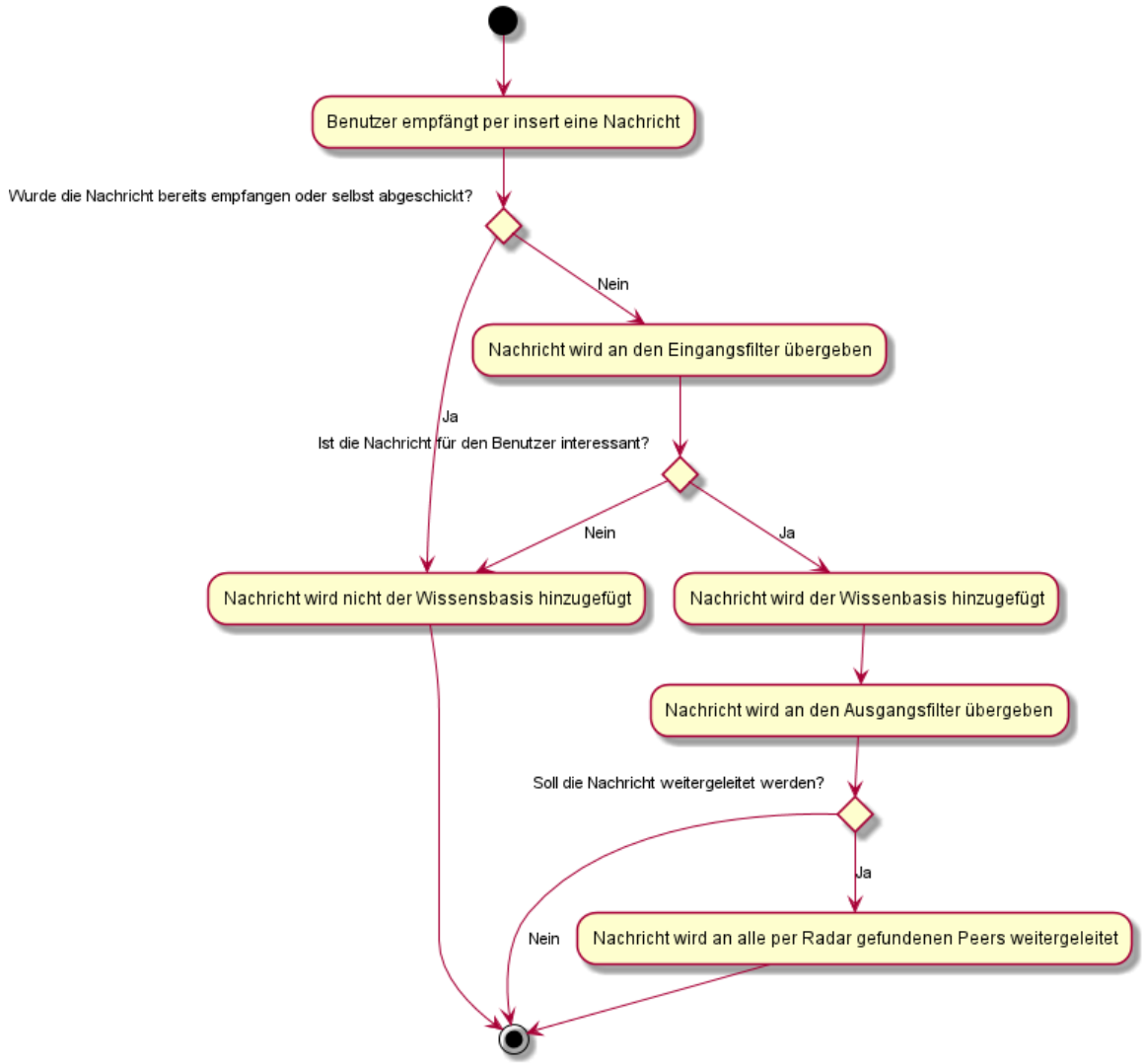

Abbildung 4.5: Filterung nach dem Empfang einer Nachricht

Hierbei wird auch ersichtlich, dass vor der eigentlichen Filterung bereits empfangene oder versandte Nachrichten nicht verwertet werden dürfen, um die Bildung von Schleifen zu vermeiden. Die verschiedenen technischen Umsetzungen zur Schleifenvermeidung wurden im Unterkapitel Broadcast-Routing erläutert.
Für die Implementierung wurde zur Schleifenvermeidung eine Abwandlung des sequenznummerkontrollierten Floodings (SNKF) gewählt (siehe dazu auch die Komponentenbeschreibung des Broadcasts). Es hat den Vorzug vor dem *Reverse Path Forwarding* (RPF) und dem *Spanning Tree* aus den folgenden Gründen erhalten:

- Der Spanning Tree muss nach jeder topologischen Änderung neu aufgebaut werden. Dies ist bei aus Smartphones bestehenden Ad-Hoc-Netzwerken häufig der Fall, da davon auszugehen ist, dass sie von sich stetig bewegenden Menschen getragen werden. Der Aufwand für den wiederholten Aufbau des Spanning Trees ist dementsprechend hoch.

- Vor dem gleichen Problem steht das RPF. Die Bestimmung des kürzesten Pfades gestaltet sich schwierig, wenn die beteiligten geräte nicht stationär sind. Bei Geräten in Bewegung besteht die Gefahr, dass Nachrichten ungewollt abgelehnt werden.

- Im Gegensatz dazu muss beim SNKF nur eine Tabelle pro Gerät pflegen werden, in der die Sequenznummern der Nachrichten gespeichert werden und diese unabhängig von der Topologie ist. Bereits empfangene oder versandte Nachrichten können durch die Sequenznummer erkannt und die Bearbeitung folglich abgelehnt werden.

Die Sequenznummer ist für das Protokoll eindeutig in Form einer Zeichenkette definiert. Diese Zeichenkette enthält zum einen den *Subject Identifier* des Urhebers der Nachricht und zum anderen den genauen Zeitpunkt der Erzeugung der Nachricht. Diese generierte Sequenznummer ist, wie in Tabelle 4.1 dargestellt, Teil des Headers der Nachricht und wird nach Erhalt der Nachricht mit der Tabelle abgeglichen.

4.4 Architektur

Die Android-Anwendung SharkNet bindet das SharkFramework als .jar Datei ein. In beiden Teilbereichen wurden für die praktische Umsetzung des semantischen Broadcasts Klassen hinzugefügt. Der Aufbau der Anwendung folgt dabei dem Model-View-Control Entwicklungsmuster, was mit Hilfe von einander getrennten Activitites, Services, Datenzugriffsobjekten (DAO) und Entitäten umgesetzt worden ist. Das grundlegende Zusammenwirken der zu entwickelnden Klassen zwischen Anwendung und Framework lässt sich

beispielhaft an zwei einfachen Szenarios erklären. Beim ersten Beispiel schreibt der Benutzer eine Broadcast-Nachricht.

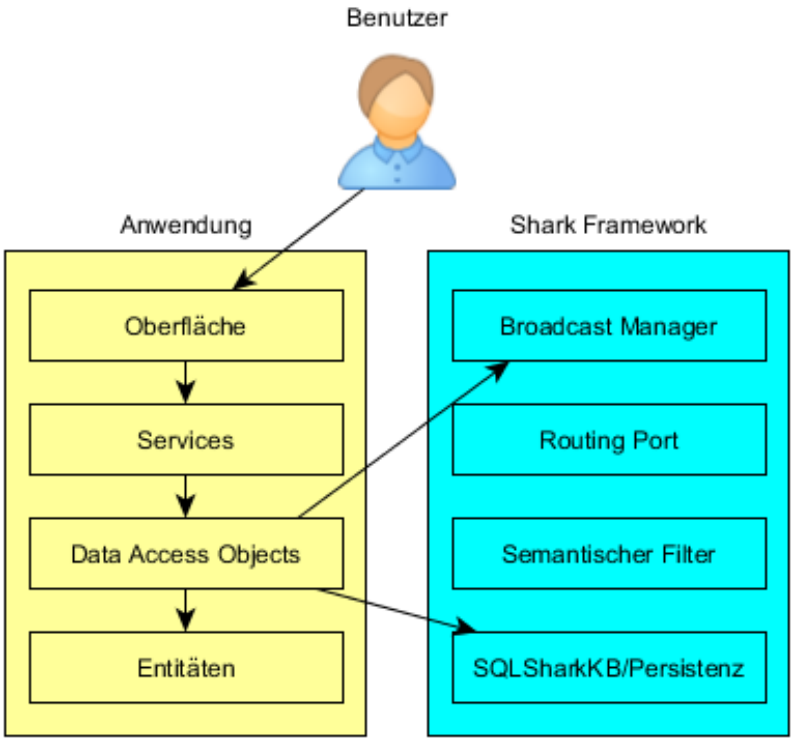

Abbildung 4.6: Anwendung und Framework beim Absenden einer Nachricht

1. Der Benutzer gibt über die Benutzeroberfläche eine Broadcast-Nachricht ein und sendet diese ab.

2. Der zuständige Service nimmmt die Zeichenkette entgegen, führt eine *sanitization* durch und reicht diese an ein DAO weiter

3. Das DAO erhält vom Service eine Message-Entität. Es ist zuständig für das Speichern, Löschen und Editieren von Entitäten. Das DAO nutzt nun die *SQLSharkKB*, um das Objekt der Wissensbasis des Benutzers hinzuzufügen.

4. Die SQLSharkKB speichert nun die Entität innerhalb einer SQLite-Datenbank. Sie wird im Unterkapitel Persistenz näher erläutert.

5. Nachdem die Nachricht gespeichert worden ist, soll sie an alle Geräte in der Umgebung versandt werden. Dies geschieht durch den *Broadcast Manager*, welcher die Nachricht vom DAO entgegen nimmt und sie per WiFi-Direct und Bluetooth versendet.

Der Routing Port und der Semantische Filter sind nur beim Empfangen von Nachrichten relevant. Beim Empfang kehrt sich der Ablauf gewissermaßen um, die Nachricht wandert vom Framework zur Anwendung.

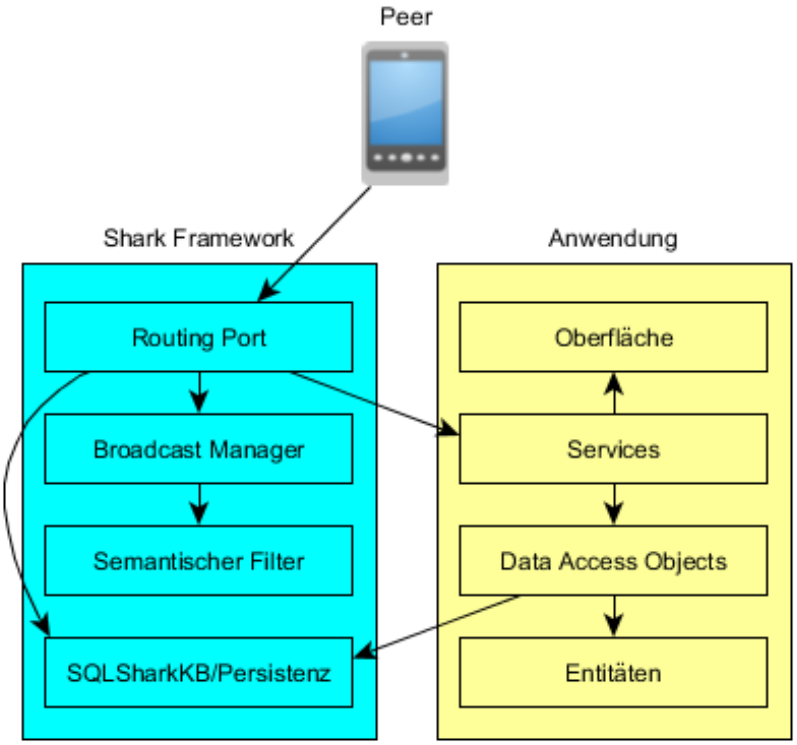

Abbildung 4.7: Anwendung und Framework beim Absenden einer Nachricht

1. Das Gerät empfängt vom Smartphone Beta eine Broadcast-Nachricht. Dies wird zunächst vom *Routing Port* verwertet, welcher die beiden aus dem Kapitel Grundlagen bekannten Methoden *insert()* und *expose()* implementiert. Beim Nachrichteneingang wird nun die Methode *insert()* ausgeführt.

2. Der *Routing Port* lässt vom *Broadcast Manager* prüfen, ob die Nachricht bereits empfangen oder abgeschickt worden ist per Sequenznummernvergleich.

3. Sollte die nachricht noch nicht empfangen worden sein, überprüft nun der Semantische Filter, ob die Nachricht für den Empfänger interessant ist. Die Komponentenbeschreibung Semantischer Filter beschreibt die Umsetzung des Filters innerhalb der Anwendung.

4. Der *Routing Port* fügt nun die Nachricht der Wissensbasis hinzu und lässt dies von der *SQlSharkKB* persistieren, falls die Nachricht für den Benutzer interessant sein sollte.

5. Per Listener wird die Nachricht anschließend von der Framework-Ebene auf die Anwendungsebene übermittelt, indem der Service eine Benachrichtigung über neue Nachrichten innerhalb der Wissensbasis erhält.

6. Der Service lässt vom DAO aus der Wissensbasis die Broadcast-Nachrichten lesen, diese Liste enthält die alten sowie die neuen Nachrichten.

7. Die Broadcast-Nachrichten werden auf der Oberfläche angezeigt.

4.5 Persistenz

Alle Daten der Anwendung werden innerhalb einer Wissensbasis gespeichert. Das Interface der Wissensbasis ist die *SharkKB*, deren Implementierungen die Daten persistieren. Die offiziell von Android unterstützte Datenbanktechnolgie ist SQLite. Es muss daher eine Implementierung der Shark-Wissensbasis mit SQLite entwickelt werden, um die bereits vorgestellten Strukturen dauerhaft auf Android-Smartphones speichern zu können.
Dafür muss zunächst ein Datenbankschema festgelegt werden.

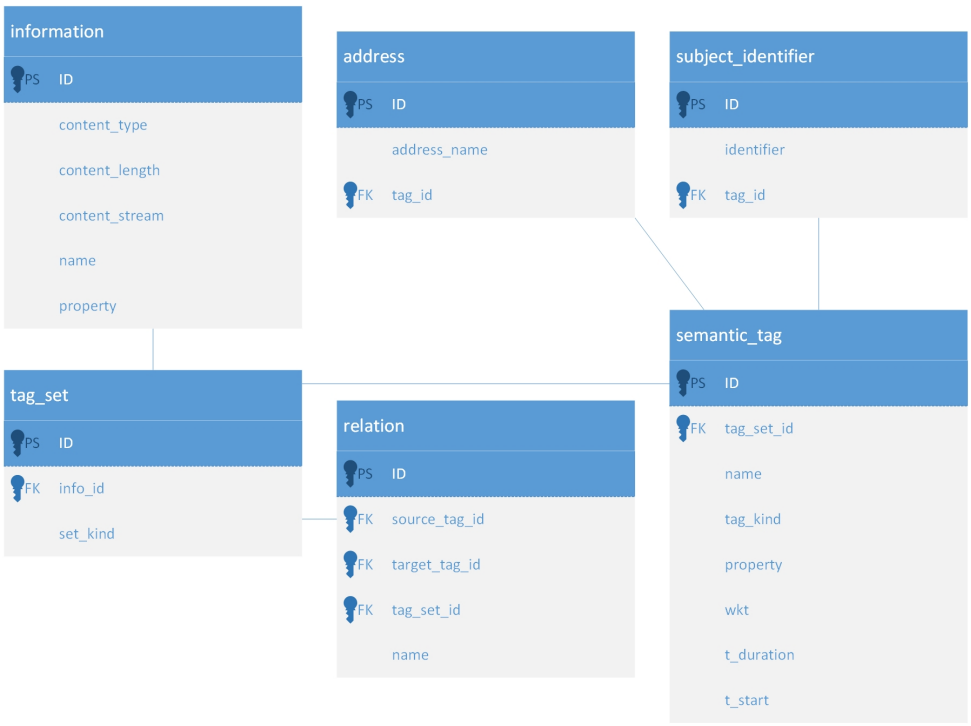

Abbildung 4.8: Auszug aus dem Datenbankschema der SQLite Implementierung der SharkKB

Alle Tabellen verfügen über eine sich für jeden neuen Eintrag automatisch inkrementierende ID, die als Primärschlüssel dient.

Die Tabelle *semantic tag* kann jede Art eines *Semantic Tag* darstellen, diese wird durch das Attribut *tag kind* gekennzeichnet (normal, peer, time, spatial). Entsprechend der Art können zusätzliche Attribute gesetzt sein, *wkt* für *Spatial Tags* und duration sowie start für *Time Tags*. Da ein *Semantic Tag* mehrere *Subject Identifier* haben kann, sind diese in einer extra Tabelle ausgelagert. Das gleiche gilt für die Adressen, falls es sich um ein *Peer Semantic Tag* halten sollte.
Die bereits erläuterten Strukturen *Tag Set* und *Semantic Net* werden durch die Tabelle *tag set* repräsentiert. Ähnlich wie beim *Semantic Tag* wird durch das Attribut *set kind* festgelegt, welche der beiden Strukturen ein Eintrag der Tabelle angehört.
Wenn es sich um ein *Semantic Net* handelt, können jeweils zwei Tags durch benannte Beziehungen miteinander verknüpft werden. Diese Beziehung ist über die Fremdschlüssel dann den beiden Tags und dem Semantischen Netz zugeordnet.
Einem *tag set* kann einer Information zugeordnet werden. Die Tabelle *information* enthält unter anderem die Rohdaten der Nachricht (Attribut *conent stream*) und wird durch die zugeordneten *Tag Sets* semantisch eingeordnet.
[...]

4.6 Benutzeroberfläche

Die grafische Benutzeroberfläche muss es dem Benutzer vor allem bei der Eintragung von semantischen Annotationen so simpel wie möglich machen, mit der Anwendung adäquat zu interagieren. Das Design sollte möglichst modern und schlicht sein, wobei dies jedoch keinen Schwerpunkt dieser Arbeit darstellen soll.
Die dafür teils von der Chatfunktionalität weitergenutzte und teils neu entwickelte Oberfläche wird nun wieder anhand eines kleinen Beispiels beschrieben.

Kapitel 5

Bluetooth

5.1 Aufgabe der Komponente

Die über SharkNet abgeschickten Nachrichten werden über Bluetooth übertragen. Die Komponente ist dabei ausschließlich für die kabellose Übertragung von Daten bzw. Nachrichten verantwortlich, die Ortung von potentiellen Kommunikationspartern erfolgt über die Wifi-Direct Komponente. Auch die Filterung von bereits bekannten oder semantisch uninteressanten Nachrichten wird nicht innerhalb dieser Komponente, sondern innerhalb der Semantischen Routing Komponente vorgenommen.
Da es in SharkNet neben normalen Chats auch Gruppenchats und einen semantischen Broadcast gibt, erfordert der Datenaustausch mit Bluetooth kein Pairing der miteinander kommunizierenden Geräte. Dies trägt maßgeblich zur Benutzerfreundlichkeit bei, da insbesondere beim semantischen Broadcast sonst ständig Anfragen zum Pairing auf dem Gerät erscheinen würden und vom Benutzer zusätzliche Interaktionen erforderlich wären.

5.2 Architektur

5.2.1 Überlick

Im folgenden UML-Klassendiagramm sind alle Bestandteile der Bluetooth Komponente von SharkNet abgebildet.

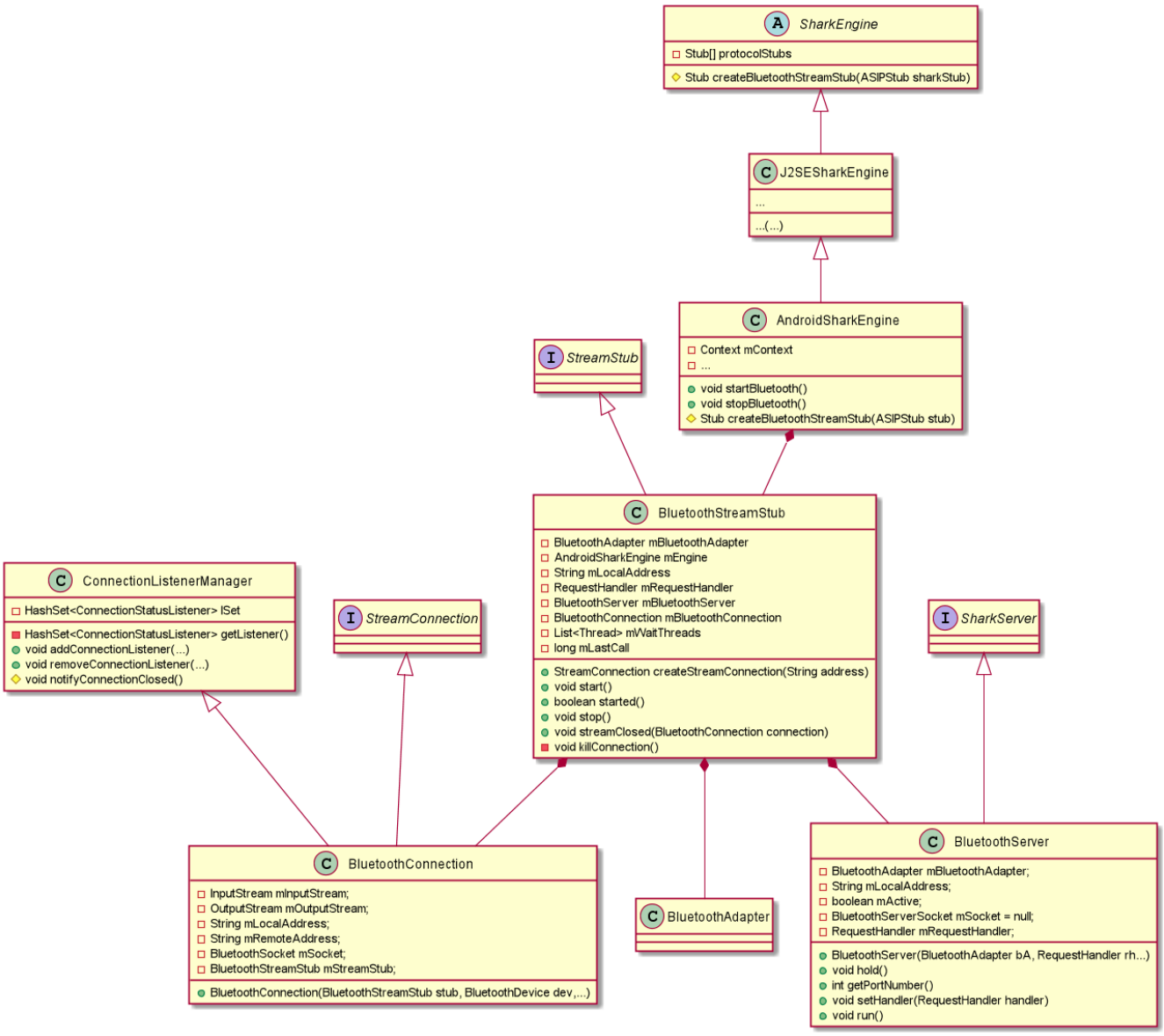

Abbildung 5.1: Die Bluetooth Klassen im Überblick

Im Zentrum dieser Hierarchie steht die Klasse *BluetoothStreamStub*. Eine Instanz dieser Klasse befindet sich als Attribut in der Klasse *AndroidSharkEngine*, von der aus alle Protokolle wie NFC, Wifi-Direct oder Bluetooth gesteuert werden. Sie stellt daher auch Methoden wie *startBluetooth()* oder *stopBluetooth()* bereit.

5.2.2 Schnittstellendefinitionen

Anhand der Klassenhierarchie der Bluetooth-Komponente lässt sich erkennen, dass die folgenden drei Schnittstellen implementiert werden:

- *StreamStub*: Mit Hilfe von Implementierungen dieses Interfaces können streamba-

sierte Ende-zu-Ende Verbindungen zwischen zwei Geräten hergestellt werden. Die Klasse BluetoothStreamStub öffnet und schließt daher die Verbindungen zu anderen Geräten per Bluetooth.

- *StreamConnection*: Das Shark Framework definiert mit dem Interface StreamConnection das Verhalten einer streambasierten Verbindung zweier Geräte. Dieses Interface ist nicht zu verwechseln mit gleichnamigen Interface von Java ME. Klassen wie *BluetoothConnection*, welche dieses Interface implementieren, bauen in ihren jeweiligen Konstruktur die Verbindung mit ihrem jeweiligen Protokoll auf. In der Klasse *BluetoothConnection* erfolgt dies über das Bluetooth-Protokoll RFCOMM.
- *SharkServer*: Eine dieses Interface implementierende Klassen wartet bei der bestehenden Verbindung auf Datenpakete, nimmt diese an und leitet sie an einen *Request Handler* weiter. Die Klasse *BluetoothServer* nimmt daher die Datenpakete an, die per bestehender Bluetoothverbindung eintreffen.

5.3 Nutzung

Der Start der Komponente erfolgt automatisch mit dem Start der Anwendung durch eine Instanzierung der Klasse *AndroidSharkEngine*. Sie kann jedoch in ebenjener Klasse durch die Methoden *startBluetooth()* und *stopBluetooth()* auch manuell gestartet sowie gestoppt werden.

5.3.1 Code

Der Code dieser Komponente kann hier `https://github.com/SharedKnowledge/SharkNet-Api-Android/tree/master/api/src/main/java/net/sharksystem/api/shark/protocols/bluetooth` betrachtet werden. Wie auch die anderen Implementierungen von Übertragungsprotokollen, befindet sich auch die Bluetooth-Implementierung im Projekt *SharkNet-Api-Android* im Package *protocols*.
Es werden nun die wichtigsten Codezeilen der drei im Unterkapitel Schnittstellendefinitionen erwähnten Klassen beschrieben.
Der *BluetoothServer* horcht auf eingehende Verbindungen mit Hilfe der von Android bereitgestellten Klasse *BluetoothSocket*, die folgendermaßen initialisiert wird:

Listing 5.1: Initialisierung des Bluetooth-Server-Sockets

```
1 mSocket = mBluetoothAdapter.
    listenUsingInsecureRfcommWithServiceRecord(BluetoothStreamStub.
    BT_NAME, BluetoothStreamStub.BT_UUID);
```

Hierbei wird mit Hilfe des aktiven *BluetoothAdapter*, welcher auch in der Klasse *BluetoothConnection* benutzt wird, über das Bluetooth-Protokoll *RFCOMM* der Server-Socket erzeugt, zu dem sich andere Geräte verbinden können. Es wird statt der sonst gängigen Methode *listenUsingRfcommWithServiceRecord()* die *insecure* Variante genutzt, um kein Bluetooth-Pairing zwischen den Geräten vorher durchführen zu müssen. Der nächste Auszug zeigt die Annahme von eingehenden Verbindungen auf Serverseite:

Listing 5.2: Serverseitige Annahme der Bluetooth-Verbindungen (Auszug)

```
1 try {
2   while (mActive){
3     BluetoothSocket bluetoothSocket = mSocket.accept();
4     BluetoothConnection con = new BluetoothConnection(bluetoothSocket
          , mLocalAddress);
5     mRequestHandler.handleStream(con);
6   }
7   mSocket.close();
```

Die Annahme der Anfrage geschieht in der dritten Zeile, wobei der daraus resultierende *BluetoothSocket* in der folgenden Zeile für den Aufbau einer Verbindung genutzt wird. Die Verbindung wird anschließend vom Shark-Interface *RequestHandler* verwertet. Dies bedeutet, dass die im Stream enthaltene Nachricht innerhalb der Framework-Ebene nun weiterverarbeitet wird. Dies schließt unter anderem auch die semantische Auswertung der Nachricht mit ein.
Der Aufbau einer Bluetooth-Verbindung erfolgt auf Clientseite ähnlich zum Aufbau auf der Serverseite innerhalb der Klasse *BluetoothConnection*:

Listing 5.3: Clientseitige Initialisierung des Sockets

```
1 mSocket = device.createInsecureRfcommSocketToServiceRecord(
      BluetoothStreamStub.BT_UUID)
```

Auch hier wird die *Insecure* Variante der Methode genommen, um auf ein Pairing verzichten zu können.
Instanzen der beiden bisher vorgestellten Klassen *BluetoothServer* und *BluetoothStreamStub* werden durch die Klasse *BluetoothStreamStub* erzeugt und verwaltet. Neben der Erzeugung dieser Objekte liefert der *BluetoothStreamStub* weiterhin die lokale Bluetooth MAC-Adresse des Geräts: *BluetoothConnection*:

Listing 5.4: Auslesen der Bluetooth MAC-Adresse

```
1 mLocalAddress = android.provider.Settings.Secure.getString(engine.
      getContext().getContentResolver(), "bluetooth_address");
```

```
2 //mLocalAddress = BluetoothAdapter.getDefaultAdapter().getAddress()
    Nur vor Marshmallow nutzbar, liefert nach dieser Version nur eine
    unbrauchbare Konstante!
```

Dies geschieht zwangsweise über eine Reflektion, da die bis Android-Marshmallow dafür vorhergesehene Methode nur eine Konstante liefert, die nicht der eigentlichen Bluetooth MAC-Adresse des Geräts entspricht. Die Bluetooth MAC-Adresse wird jedoch zwingend für *Insecure* Verbindungen benötigt, welche kein Bluetooth-Pairing erfordern, was für die Broadcast-Komponente essentiell ist.
Die restlichen Methoden der Bluetooth-Komponente wie beispielsweise *killConnection()* sind mnemonischer Natur.

5.3.2 Deployment / Runtime

[...]

5.4 Test

5.4.1 Gerätetest

Mit den folgenden Android-Geräten ist die Komponente auf Kompatibilität geprüft worden:

Tabelle 5.1: Kompatibilitätstest der Komponente

Gerät	Android-Version	kompatibel
LG Nexus 5x	8.0	Ja
LG Nexus 5x	8.1	Nein
LG Nexus 5	6.1	Ja
Sony Xperia XZ Premium	8.0	Ja
Sony Xperia Z4 Tablet	7.1.1	Ja
Lenovo B	6.0	Ja
Lenovo A5500-F Tablet	4.4	Nein
Raspberry Pi 3	6.0.1	Ja
Wandboard Quad	5.0.2	Ja

Seit neusten Android-Version 8.1 ist es nicht mehr mit der in Listing 4.4 beschriebenen Refeklektion möglich, die Bluetooth MAC-Adresse programmatisch auszulesen. Dementsprechend kann das Testgerät Nexus 5x seit dem Update des Betriebssystems keine Nachrichten via Bluetooth mehr empfangen, da die anderen Geräte keine valide MAC-Adresse vom Nexus 5x erhalten haben und die Verbindung somit ohne Pairing nicht erfolgreich

aufgebaut werden kann.
Beim Lenovo A5500-F Tablet handelt es sich um eine zu alte Android-Version, wodurch in mehreren Komponenten zu Exceptions kommt, weil benötigte Methoden noch nicht enthalten sind.

5.5 Ausblick

Es ist empfehlenswert, die von Android gestellten Bluetooth Klassen durch die dazu äquivalenten Bluetooth Low Energy (BLE) Klassen entweder zu ersetzen oder zumindest eine Alternative zu dem klassischen Bluetooth Package zu bieten. BLE verbraucht weit weniger Akkuleistung als das klassische Bluetooth, kann dafür aber nur eine geringere Menge an Daten pro Verbindung unterstützen. Da die mit SharkNet verschickten Nachrichten auch trotz der semantischen Annotationen nur wenige Kilobyte benötigen, stellt dies für SharkNet kein Hindernis dar.
Notwendig ist zukünftig außerdem das Ersetzen der in Listing 4.4 dargestellten Reflektion zum Auslesen der Bluetooth MAC-Adresse. Dies funktioniert nur mit Geräten bis einschließlich Android 8.0, ab Android 8.1 ist die ausgelesene MAC-Adresse inkorrekt. Über die WiFi-Komponente an andere Geräte versendete inkorrekte Bluetooth-Adressen führen dann dazu, dass das Gerät keine Nachrichten über Bluetooth empfangen kann.
Lasttests mit mehreren miteinander kommunizierenden Geräten und gleichzeitigem Spammen von Nachrichten haben der Komponente ebenfalls Grenzen der Belastbarkeit aufgezeigt. So können bei zu hoher Last Nachrichten verloren gehen, da das Empfangsgerät die in Listing 4.2 gezeigte *accept()* Methode nicht aufgerufen wird.

Kapitel 6

WiFi

6.1 Aufgabe der Komponente

Über die WiFi-Direct Komponente vermitteln die Peers ihre Kontaktdaten an alle verfügbaren Peers in der Nähe. Dies geschieht über den Expose Befehl des ASIP Protokolls, bei dem ein ASIP-Interesse an die Wissensbasis von anderen Peers gesandt wird. Dies beinhaltet unter anderem die Bluetooth MAC-Adresse, mit der dem Peer dann anschließend Nachrichten per Bluetooth geschickt werden können. Das Verschicken der Bluetooth Mac-Adresse via WiFi-Direct ermöglicht es daher, dass für die darauf folgende Bluetooth-Verbindung kein Pairing benötigt wird.
Die Komponente ist der elementare Bestandteil des Peer-Radars, der alle sich in der Nähe befindlichen Peers anzeigt und die Kommunikation mit diesen erlaubt. Das Radar ist wiederum dafür erforderlich, neue Chats mit Peers anzulegen oder einen semantischen Broadcast ohne Bluetooth-Pairing zu ermöglichen.

6.2 Architektur

6.2.1 Überlick

Im folgenden UML-Klassendiagramm sind alle Bestandteile der WiFi-Direct Komponente von SharkNet abgebildet.

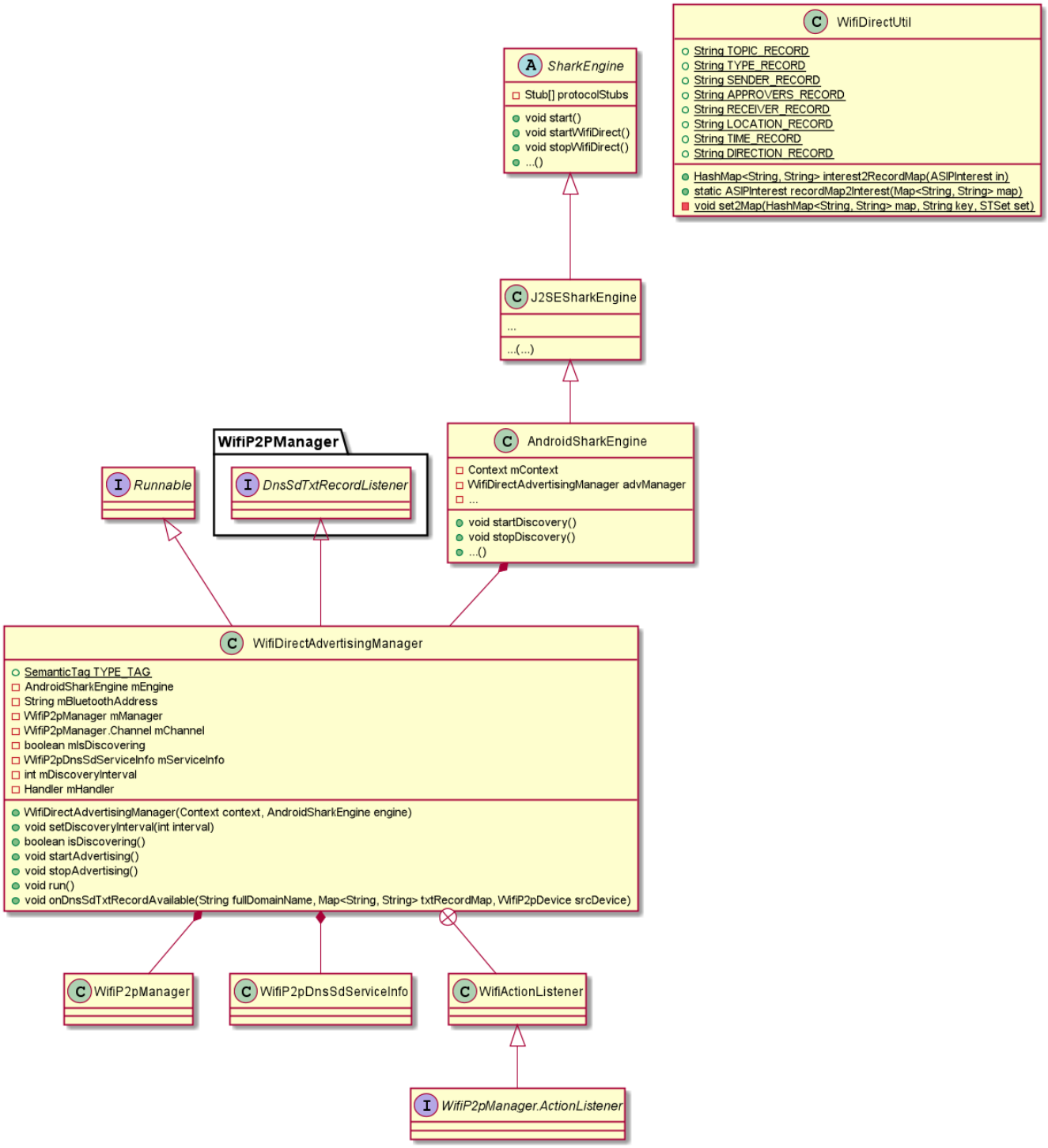

Abbildung 6.1: Die WiFi-Direct Klassen im Überblick

Im Zentrum dieser Hierarchie steht die Klasse *WiFiDirectAdvertisingManager*. Eine Instanz dieser Klasse befindet sich als Attribut in der Klasse *AndroidSharkEngine*, von der aus alle Protokolle wie NFC, Wifi-Direct oder Bluetooth gesteuert werden. Über die Engine kann daher auch das Radar per *startDiscovery()* Methode gestartet oder über

die *stopDiscovery()* Methode beendet werden. Das Starten oder Stoppen der kompletten WiFi-Komponente erfolgt dagegen in der Klasse *AndroidSharkEngine*, die den Ausgangspunkt der Vererbungshierarchie darstellt.
Die Klasse *WifiDirectUtil* bietet statische Methoden an, mit denen ASIP-Interessen in Hashmaps umgewandelt werden können und umgekehrt. Dies ist notwendig, da die von Android gestellte Basisklasse *WifiP2PManager* bei der Anmeldungen von Services keine ASIP-Interessen, sondern Hashmaps als Parameter akzeptiert.

6.2.2 Schnittstellendefinitionen

lore

6.3 Nutzung

Die WiFi-Komponente wird automatisch beim Start der Anwendung gestartet. Manuell kann die Komponente über die Klasse *AndroidSharkEngine* gesteuert werden, welche wie schon im Überblick erwähnt die beiden Methoden *startDiscovery()* und *stopDiscovery()* enthält.

6.3.1 Code

Der Code dieser Komponente kann hier `https://github.com/SharedKnowledge/SharkNet-Api-Android/tree/master/api/src/main/java/net/sharksystem/api/shark/protocols/wifidirect` betrachtet werden. Wie auch die anderen Implementierungen von Übertragungsprotokollen, befindet sich auch die WiFi-Direct-Implementierung im Projekt *SharkNet-Api-Android* im Package *protocols*.
Wie im vorherigen Unterkapitel erläutert liefern die beiden Methoden *startDiscovery()* und *stopDiscovery()* die Funktionalität, um Peers zu finden und andere Peers über das eigene Interesse in Kenntnis zu setzen.
Bei Aufruf der *startDiscovery()* Methode wird innerhalb der Engine ein neuer *WifiDirectAdvertisingManager* angelegt und anschließend dessen *startAdvertising()* Methode aufgerufen. Innerhalb der *startAdvertising()* Methode wird sich nun auf der dritten Schicht des OSI-Modells begeben, wie der folgende Codeausschnitt zeigt:

Listing 6.1: Hinzufügung des Services

```
1 HashMap<String, String> map = WifiDirectUtil.interest2RecordMap(
      interest);
```

```
2 mServiceInfo = WifiP2pDnsSdServiceInfo.newInstance("_sbc", "_presence
    ._tcp", map);
3 mManager.addLocalService(mChannel, mServiceInfo, new
    WifiActionListener("Add_LocalService"));
4 mManager.clearServiceRequests(mChannel, new WifiActionListener("Clear
    _ServiceRequests"));
5 WifiP2pDnsSdServiceRequest wifiP2pDnsSdServiceRequest =
    WifiP2pDnsSdServiceRequest.newInstance();
6 mManager.addServiceRequest(mChannel, wifiP2pDnsSdServiceRequest, new
    WifiActionListener("Add_ServiceRequest"));
```

Nachdem in der erste Zeile eine Hashmap auf dem Interesse erzeugt worden ist, wird diese Hashmap in Zeile zwei als Parameter für die Erzeugung einer Service Information benutzt. Anschließend wird dem *WifiP2PManager* ein neuer lokaler Service hinzugefügt, wobei dieser Service die zuvor erzeugte Service Information enthält. Nachdem etwaige vorherige Service Requests beseitigt worden sind, wird der neue WifiP2P Service Request hinzugefügt. Dadurch wird nun an alle Geräte in der Nähe, die auf WifiP2P Service Requests warten, dieser zur Verfügung gestellt.
Neben dem Hinzufügen von Services, müssen diese aber auch empfangen und ausgewertet werden. Dies ist der Grund, warum der *WifiDirectAdvertisingManager* das Interface *Runnable* implementiert. In der dadurch implementierten Methode *run()* werden die von anderen Geräten gesendeten Service Requests empfangen.

Listing 6.2: Erkennung von Services

```
1 mManager.discoverServices(mChannel, new WifiActionListener("Discover_
    Services"));
2 mHandler.postDelayed(this, mDiscoveryInterval);
```

Sollte ein Service gefunden und erfolgreich eine Peer-To-Peer Verbindung zwischen zwei Geräten aufgebaut werden können, wird nun die aus Listing x.x bekannte Hashmap an das Gerät gesendet, welches den Service gefunden (discovered) hat. Dabei wird automatisch die Methode *onDnsSdTxtRecordAvailable* aufgerufen, welche die empfangene Hashmap in ein ASIP-Interesse umwandelt und dann der Engine weiterreicht.

Listing 6.3: Vewertung des Interesses

```
1 ASIPInterest interest = WifiDirectUtil.recordMap2Interest(
    txtRecordMap);
2 mEngine.handleASIPInterest(interest);
```

6.3.2 Deployment / Runtime

[...]

6.4 Test

6.4.1 Gerätetest

Mit den folgenden Android-Geräten ist die Komponente auf Kompatibilität geprüft worden:

Tabelle 6.1: Kompatibilitätstest der Komponente

Gerät	Android-Version	kompatibel
LG Nexus 5x	8.0	Ja
LG Nexus 5x	8.1	Ja
LG Nexus 5	6.1	Ja
Sony Xperia XZ Premium	8.0	Ja
Sony Xperia Z4 Tablet	7.1.1	Ja
Lenovo B	6.0	Ja
Lenovo A5500-F Tablet	4.4	Nein
Raspberry Pi 3	6.0.1	Nein
Wandboard Quad	5.0.2	Nein

Die beiden Einplatinencomputer Raspberry Pi 3 und Wandboard Quad unterstützen zwar grundsätzlich WLAN, jedoch nicht WiFi-Direct. Beim Raspberry Pi 3 wäre WiFi-Direct zwar technisch möglich, benötigt aber zahlreiche Umkonfigurationen, was dadurch dann nicht mehr eine reine Android-Version darstellt.
Das Lenovo A5500-F Tablet hat mit Android 4.4 eine zu alte Version, die nicht alle von der Komponente benötigten WiFi-Direct Klassen bereitstellt.
Nach dem Update des LG Nexus 5x von Android 8.0 auf die Version 8.1 ist zu beachten, dass das Gerät seine Bluetooth MAC-Adresse nicht mehr programmatisch auslesen kann. Dies betrifft vor allem die Bluetooth-Komponente und wird in der dazugehörigen Komponentenbeschreibung vertieft.

6.5 Ausblick

Die WiFi Komponente wurde SharkNet hinzugefügt, da der wiederholte Austausch von Kontakdaten zwischen den Geräten mit Bluetooth zu viel Zeit in Anspruch genommen hat. Da jedes Gerät standardmäßig alle zehn Sekunden seine Anmeldedaten an Geräte in

der Nähe schickt, musste diese eher ungewöhnliche Aufteilung erfolgen. Wenn zukünftig die Bluetooth-Komponente auf Bluetooth Low Energy umgestellt werden sollte, ist es eventuell möglich, auf die WiFi Komponente zu verzichten und den gesamten Datenaustausch über Bluetooth vorzunehmen.

Kapitel 7

Radar

7.1 Aufgabe der Komponente

Durch das Radar können Geräte, auf welchen ebenfalls die Anwendung Sharknet installiert ist, in räumlicher Nähe ausfindig gemacht und angezeigt werden. Es nutzt dabei die Verhaltensweise der WiFi-Komponente, bei der in regelmäßigen Abständen Geräteinformationen an alle Geräte in der Nähe geschickt werden. Diese Geräteinformationen werden vom Radar empfangen, gebündelt und dem Benutzer dann auf dem Gerät als Liste angezeigt. Der Benutzer kann anschließend anhand dieser Geräteliste einen Chat eröffnen. Neben der Eröffnung von Chats ist diese Geräteliste außerdem wichtig für die Broadcast-Komponente, da die Broadcast Nachricht an alle Geräte geschickt wird, die sich auf dieser Liste befinden.

7.2 Architektur

7.2.1 Überlick

Im folgenden UML-Klassendiagramm sind alle Bestandteile der Radar-Komponente von SharkNet abgebildet.

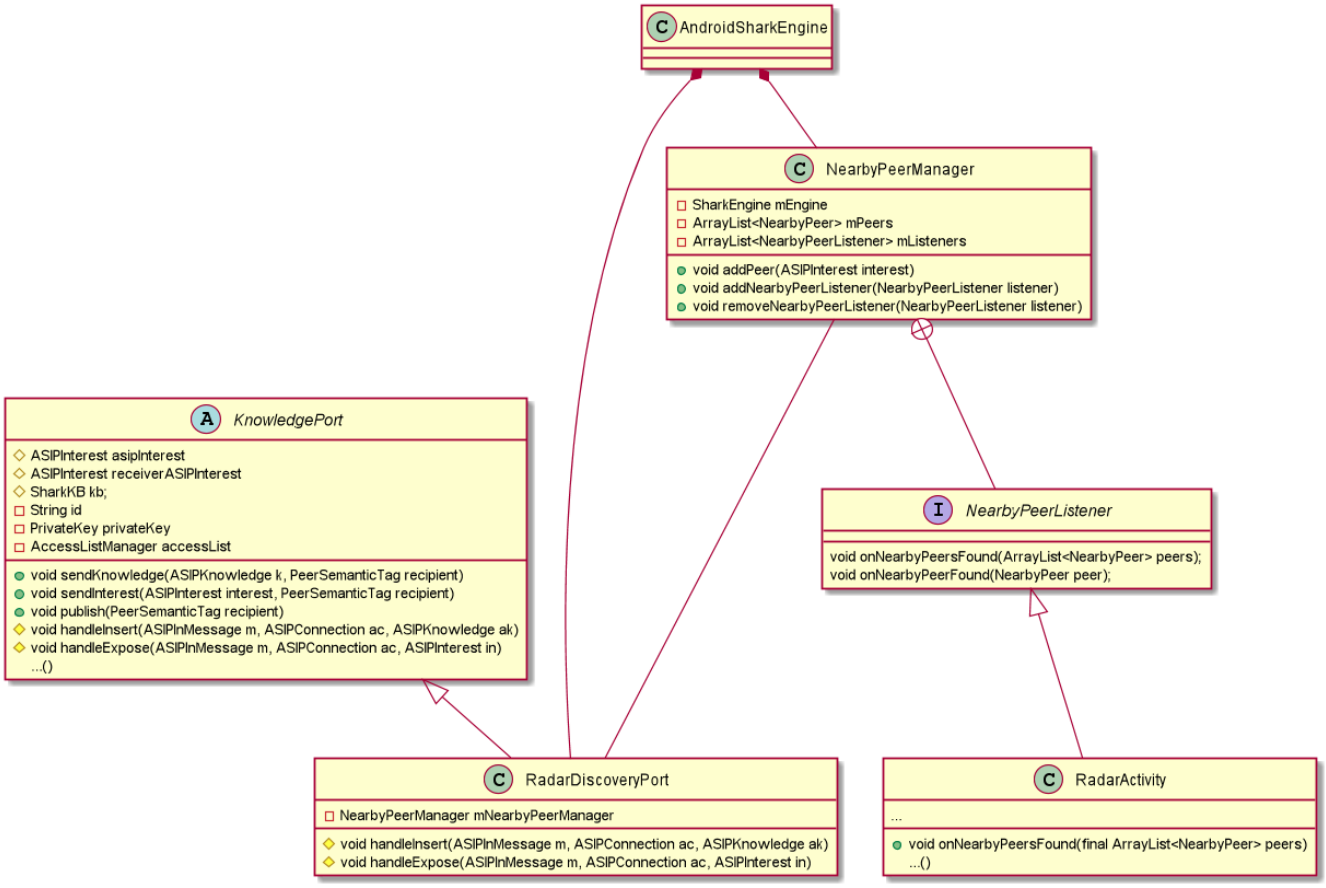

Abbildung 7.1: Die Radar Klassen im Überblick

Die eingangs erwähnte Geräteliste befindet sich als Attribut innerhalb der Klasse *NearbyPeerManager*. Die Klasse enthält das Interface *NearbyPeerListener*, welches für die Benachrichtigungen im Falle von neu gefundenen Geräten zuständig ist. Android Activities wie die *RadarActivity* oder aber auch die *BroadcastActivity* implementieren dieses Interface, um über eine stets über alle Geräte in der Nähe informiert zu sein. Der *RadarDiscoveryPort* ist die Schnittstelle zwischen Anwendung und Shark Framework. Die im Grundlagenkapitel Kapitel erwähnten Knowledge-Port-Methoden *handleInsert()* und *handleExpose()* werden benötigt, weil die Geräte ihre Informationen in Form von ASIP-Interessen versenden. Diese Interessen werden nach dem Empfang dann auf der Ebene des Frameworks durch die Methode *handleInsert()* verarbeitet und anschließend auf der Ebene der Anwendung dem Benutzer dargestellt.

7.2.2 Schnittstellendefinitionen

lore

7.3 Nutzung

Die Komponente kann über die *startDiscovery()* Methode der *AndroidSharkEngine* gestartet werden.

7.3.1 Code

lore

7.3.2 Deployment / Runtime

lore

7.4 Test

lore

7.5 Ausblick

lore

Kapitel 8

Broadcast

8.1 Aufgabe der Komponente

Die Broadcast Komponente ermöglicht es den Benutzern von SharkNet, Nachrichten an andere Benutzer zu schicken. Dabei können auch andere Komponenten, wie etwa der semantische Eingans- und Ausgangsfilter zum Einsatz kommen, was jedoch nicht zwingend erforderlich ist. Falls auf einen Eingangsfilter oder Ausgangsfilter verzichtet werden sollte, werden wie bei einem klassischen Broadcast die Nachrichten an alle sich in der Nähe befindlichen Geräte versendet. Inwiefern der klassische Broadcast vom Benutzer semantisch eingeschränkt werden kann, lässt sich in der Komponentenbeschreibung der Komponente Semantischer Filter in Erfahrung bringen.

8.2 Architektur

8.2.1 Überlick

Die folgenden Komponenten werden von der Komponente Broadcast zwingend benötigt:

- WifI
- Bluetooth
- Persistenz

Optional sind hingegen die Komponenten:

- Semantischer Filter

8.2.2 Überlick

Die Komponente bildet sich vorrangig aus sieben Klassen, wovon drei sich innerhalb des SharkFrameworks und vier sich innerhalb der App befinden. Diese sieben Klassen werden nun ausgehend von der folgenden Abbildung kurz beschrieben.

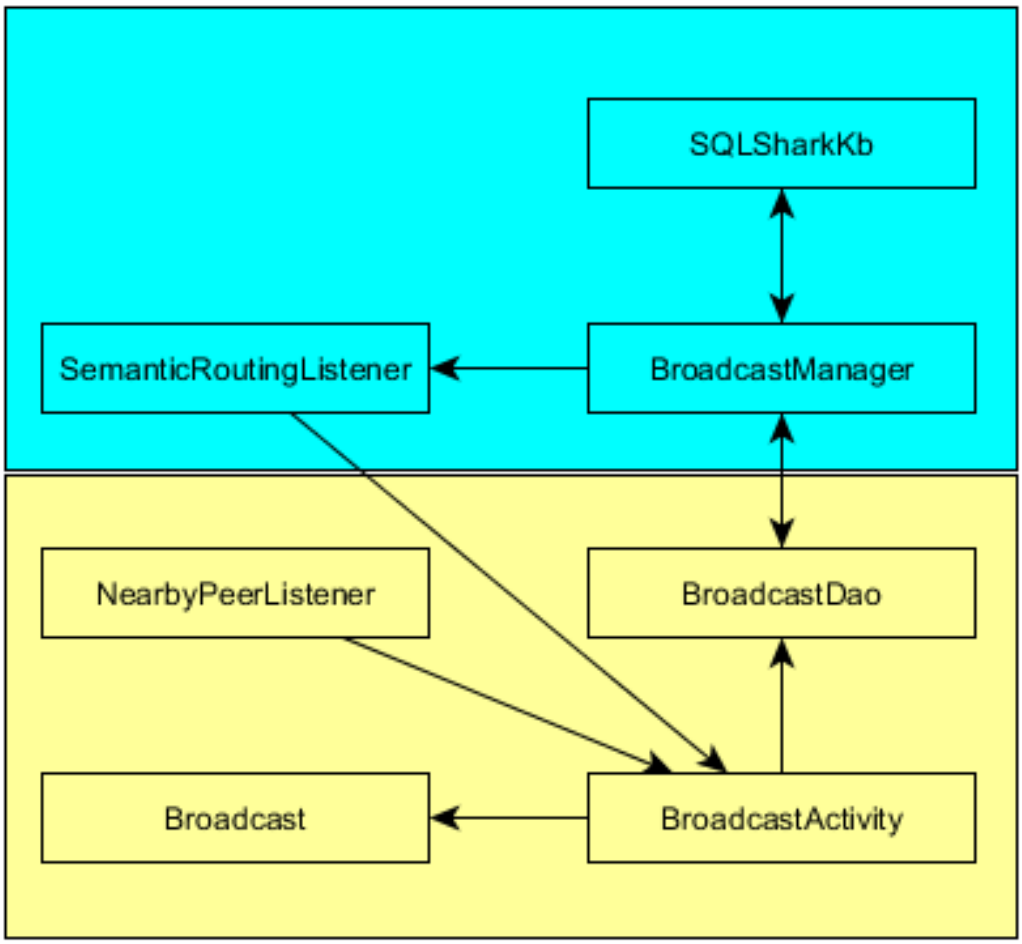

Abbildung 8.1: Die Klassen der Broadcast Komponente

- Die SQLSharkKb ist eine Implementierung der Shark Knowledgebase mit SQLite. Mit ihr werden sämtliche Daten wie bsp. die Nachrichten, semnatische Annotationen oder auch Benutzerprofile gespeichert. Sie nimmt ausschließlich Anfragen von Klassen aus dem SharkFramework entgegen.

- Der BroadcastManager ist die direkte Schnittstelle zwischen dem Framework und der App. Er nimmt Broadcast-Objekte vom BroadcastDao entgegen und lässt diese gegebenfalls von der SQLSharkKb speichern. Sollte eine neue Nachricht den Peer

erreichen, wird vom BroadcastManager die Nachricht auf ihre semantische Relevanz hin überprüft und im Erfolgsfall der Wissenbasis des Peers hinzugefügt.

- Der SemanticRoutingListener liefert neue vom BroadcastManager akzeptierte Nachrichten an die BroadcastActivity

- Der BroadcastDao nimmmt von der BroadcastActivity veränderte Nachrichten in Form eines Objekts vom Typ Broadcast entgegen, baut diese in für das Shark-Framework vewertbare Objekte vom Typ ASIPSpace um und leitet diese an den BroadcastManager weiter.

- Wann immer die Anzahl der sich in Reichweite befindlichen Peers ändert, wird die BroadcastActivity vom NearbyPeerListener mit einer angepassten Liste von Peers versorgt.

- Die BroadcastActivity ist die Schnittstelle zwischen Benutzer und App. Sie nimmt neue Nachrichten vom Benutzer entgegen, wobei das Hinzufügen von semantischen Annotationen optional ist. Sie benutzt die Entitätsklasse Broadcast um die Nachrichten in einer Klasse zu bündeln, welche bei Aktualisierungen an das BroadcastDao weitergereicht wird.

8.2.3 Schnittstellendefinitionen

8.3 Nutzung

Die Komponente ist in der App innerhalb der *BroadcastActivity* eingebunden. Der Endanwender kann über die diese Activity und die dazugehörige XML-Datei die Nachrichten versenden, betrachten und mit semantischen Annotationen versehen, wobei Letzteres auch die Komponente Semantische Filter betrifft.
Die Komponente kann aber auch in eigenen Activities benutzt werden ohne die vorgegebene *BroadcastActivity* benutzen zu müssen. Der Entwickler muss bei seiner eigenen Activity dafür lediglich von der Klasse *BaseActivity* erben. Die Klasse *BaseActivity* stellt das Attribut *mApi* vom Typ *SharkNetApi* bereit, mit dem durch die Methoden *getBroadcast()* und *updateBoradcast(...)* der Broadcast geliefert und verändert werden kann.

8.3.1 Code

Der Code dieser Komponente kann hier `https://github.com/SharedKnowledge/SharkNet/tree/master/app/src/main/java/net/sharksystem/sharknet` betrachtet werden.

8.3.2 Deployment / Runtime

...

8.4 Test

8.5 Ausblick

Der Austausch von Nachrichten mit mehreren Geräten in der Nähe funktioniert grundlegend sicher, aber noch nicht komplett fehlerlos. So kann es bei hoher Last seitens der Benutzer passieren, dass einige Nachrichten nicht empfangen werden können, obwohl sie gemäß dem eingestellten semantischen Filter akzeptiert werden müssten.

Kapitel 9

Semantischer Filter

9.1 Aufgabe der Komponente

Über den Broadcast erhält der Benutzer eine Vielzahl an Nachrichten von anderen Benutzern, von denen einen Großteil für ihn irrelevant sind. Der Semantische Filter ist dafür verantwortlich, dem Benutzer nur die für ihn interessante Nachrichten zu akzeptieren und in seine Wissensbasis einfließen zu lassen. Er ist damit neben dem Broadcast die wichtigste Komponente dieser Arbeit. Neben dem bereits beschriebenen Eingangsfilter gibt es noch einen Ausgangsfilter, der für die etwaige Weiterleitung von Nachrichten an andere Peers verantwortlich ist.
Die Benutzer können ihre Filter über ein Menü innerhalb des Profilbereichs einstellen, wobei dies keine Pflicht ist. Wenn keine Filter gesetzt sind, werden alle Nachrichten akzeptiert und weitergeleitet, sofern diese nicht bereits zuvor empfangen worden sind.

9.2 Architektur

9.2.1 Überlick

Der semantische Filter gliedert sich in verschiedene Teilfilter, diese Trennung richtet sich nach den bereits bekannten Dimensionen des Shark Frameworks. Um den Gesamtfilter mit den kleineren Teilfiltern dynamisch zusammensetzen zu können, wurde das Entwurfsmuster Kompositum gewählt. Mit Hilfe dieses Musters müssen nur jeweils die Teilfilter gesetzt werden, die für den Benutzer auch eine Relevanz haben. Die folgende Klassenhierarchie verdeutlicht dieses Verhältnis:

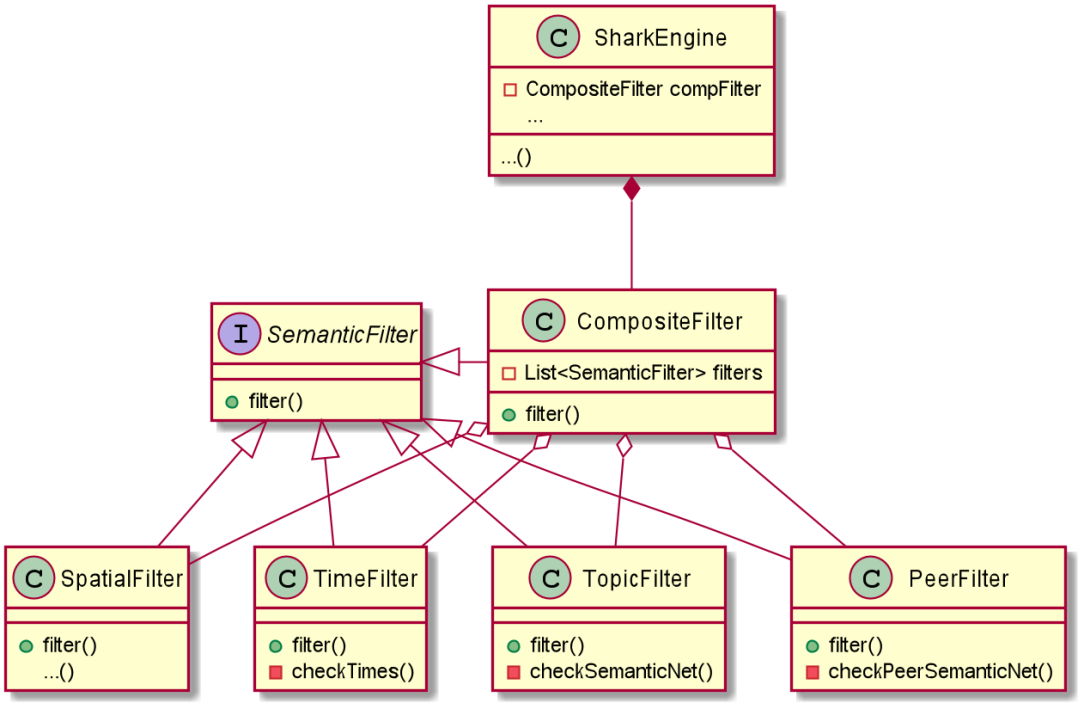

Abbildung 9.1: Klassenhierarchie des semantischen Filters (Auszug)

- Die *SharkEngine* enthält das Kompositum und stellt Methoden zur Erzeugung und Anpassung dafür bereit. Weiterhin ist diese Klasse von der App aus erreichbar, wodurch über die App abhangig von den Eingaben des Benutzers die Filter gesetzt oder entfernt werden können.

- Das Interface *SemanticFilter* wird von allen Teilfilterklassen und der Kompositumsklasse implementiert. Die einzige zu implementierende Methode ist dabei die Filtermethode, die einen booleschen Wert zurückliefert.

- Der *CompositeFilter* besitzt eine Liste aus allen Teilfiltern, ermöglicht durch Polymorphismus. Bei Aufruf der Filtermethode werden sämtliche Teilfilter angewandt, die sich in der Liste befinden. Näheres dazu befindet sich im Unterkapitel 7.3.1 Code.

- Die Relevanz der Themen werden durch den *TopicFilter* geprüft. Der Filter kann für die beiden Dimensionen Topics und Types verwendet werden.

- Die Dimensionen Sender, Approvers und Receivers werden durch den PeerFilter abgedeckt. Da die Dimension Sender in Gegensatz zu Approvers und Receivers nur ein SemanticTag, jedoch kein SemanticNet enthält, findet eine Fallunterscheidung am Anfang der Methode statt.

- Der *TimeFilter* kontrolliert, ob sich mindestens einer der Zeiträume, die sich im semantischen Profil und in der empfangenen Nachricht befinden, überschneiden.

- Die spatiale Auswertung findet im *SpatialFilter* statt, sie wird im Rahmen einer Bachelorarbeit von Maximilian Öhme entwickelt.

9.3 Code

Wie bereits im Überblick angerissen, führt der *CompositeFilter* keine eigene semantische Filterung durch, sondern lässt dies fachgerecht von den Teilfiltern ausführen. Im folgenden Codeausschnitt ist erkennbar, dass der sequentielle Aufruf der Teilfilter sofort abgebrochen wird, wenn ein Teilfilter ein *false* liefert.

Listing 9.1: Filtermethode im Kompositum

```
1  boolean isInteresing = true;
2  int i = 0;
3  while (isInteresing && i < childFilters.size()) {
4  isInteresing =  childFilters.get(i).filter(message, newKnowledge,
       entryProfile);
5  i++; }
6  return isInteresing;
```

Angenommen es handelt sich bei der ersten Iteration der Schleife um eine Instanz der Klasse *TopicFilter*, welche ihre Filtermethode aufruft, dann würde es zunächst zur folgenden Auswertung kommen:

Listing 9.2: Filtermethode des TopicType Filters (Auszug)

```
1  if (activeEntryProfile == null) return true;
2  switch (dimension){
3    case TOPIC:
4      if (activeEntryProfile.getTopics() instanceof SemanticNet) {
5            isInteresting = checkSemanticNet(activeEntryProfile.
                 getTopics(), newKnowledge);
6          }
7          else {
8            isInteresting = checkSemanticTag(newKnowledge,
                 activeEntryProfile);
9      }
10 break;
```

Es wird zunächst wie auch bei allen anderen Teilfiltern überprüft, ob überhaupt ein semantisches Profil vom Benutzer gesetzt worden ist. Falls nicht, wird die Auswertung sofort mit einem *true* als Rückgabewert beendet. Es wird nun wie auch beim *PeerFilter* Überprüft, um welche Dimension es sich bei der Filterauswertung handelt. In Zeile vier von Listing 7.2 wird überprüft, ob es sich nur um ein einzelnes Tag oder um ein gesamtes SemanticNet handelt. Dadurch wird der Besonderheit in Shark Rechnung getragen, dass eine Dimension entweder durch ein einzelnes Tag oder durch ein komplettes Semantisches Netz beschrieben werden kann. Der folgende Auszug zeigt die Auswertung eines Semantischen Netzes:

Listing 9.3: Auswertung des Semantischen Netzes (Auszug)

```
1 SemanticNet resultNet = SharkCSAlgebra.contextualize(inputNet,
    profileSet, fp);
2         if (resultNet == null || resultNet.isEmpty()) {
3                 return false;
4         }
5         else {
6                 return true;
7         }
```

Für die Auswertung wird die vom SharkFramework bereitgestellte Funktionalität der Kontextualisierung von Semantischen Netzen benutzt. Bei der Kontextualisierung soll das gemeinsame Interesse beider Peers bestimmt werden. Das Ergebnis der Kontextualisierung ist ein drittes Semantisches Netz, was als Fragment bezeichnet wird. Sollte dieses Fragment als Ergebnis der Prozedur nicht leer sein, haben beide Benutzer innerhalb dieser Dimension ein gemeinsames Interesse und die Nachricht wird bezüglich dieser Dimension als interessant eingestuft.
Die in Zeile eins dafür aufgerufene Methode benötigt dabei drei Parameter. Diese umfassen das Semantische Netz des Benutzerprofils und das Semantische Netz der Nachricht innerhalb dessen zugeordneten Dimensionen, sowie die Fragmentierungsparameter der Kontextualisierung. Die Fragmentierungsparameter werden ebenfalls vom Benutzer eingetragen und bestehen aus den folgenden drei Teilen:

- Eine Liste aus erlaubten Beziehungen, welche bei der Kontextualisierung berücksichtigt werden können.

- Eine Liste aus nicht erlaubten Beziehungen, welche bei der Kontextualisierung nicht berücksichtigt werden.

- Die Tiefe, die darüber Auskunft gibt, wie viele Beziehungen zwischen den einzelnen SemanticTags berücksichtigt werden.

Nachdem das Fragment bestimmt uns ausgewertet worden ist, liefert die Methode dann dementsprechend ein *true* oder *false* zurück. Damit ist die Auswertung der Nachricht für diese Dimension abgeschlossen.
Die Auswertung für die Peer Dimensionen *Sender*, *Receivers* und *Approvers* läuft fast analog zu dem Auswertungsverfahren der Topic-Dimensionen ab. Hierbei können falls vom Programmierer gewünscht auch die Adressen mit ausgewertet werden.
Die Auswertung der Dimension *Time* ist nicht abhängig von einer Kontextualisierung, sondern von der potentiellen Überschneidung von Zeiträumen.

Listing 9.4: Auswertung der Time-Dimension (Auszug)

```
1  while (messageTimesTags.hasMoreElements()) {
2          currentMessageTag = messageTimesTags.nextElement();
3                  for (TimeSemanticTag currentProfileTag:
                       profileTimesTagsList) {
4                          if (currentMessageTag.getFrom() >
                               currentProfileTag.getFrom() &&
5                          currentMessageTag.getFrom() +
                               currentMessageTag.getDuration() <
6                          currentProfileTag.getFrom() +
                               currentProfileTag.getDuration()) {
7                          return true;
8                  }
9          }
10 }
11 return false;
```

Innerhalb der Schleife werden alle *TimeSemanticTags* des Profils mit denen der eingehenden Nachricht verglichen. Von der vierten bis zur sechsten Zeile werden die Zeiträume von je zwei *Tags* auf Überschneidungen hin überprüft. Falls dies der Fall sein sollte, gilt die nachricht hinsichtlich der Dimension *Time* als interessant, andernfalls werden die restlichen *Tags* ausgewertet. Sollte es nicht mindestens zu einer Überschneidung kommen, gilt die Nachricht gemäß der zeitlichen Dimension als uninteressant.
Die Auswertung der räumlichen Dimension *Spatial* erfolgt durch die Aufstellung und Auswertung von Bewegungsprofilen. Diese Bewegungsprofile werden vom Smartphone (nach der Abfrage des Einverständnisses des Benutzers) aufgezeichnet und sind Teil des semantischen Profils. Nach Eingang einer Nachricht werden nun das Bewegungsprofil vom

Benutzer und das Bewegungsprofil der Nachricht miteinander verglichen. Wie bereits zuvor erwähnt, ist die spatiale Auswertung der Nachrichten das Thema der Bachelorarbeit von Maximilian Oehme. Weitergehende Details und Codebeispiele können in dieser Bachelorarbeit in Erfahrung gebracht werden.

9.3.1 Schnittstellendefinitionen

9.4 Nutzung

Die Nutzung dieser Komponente erfolgt über die zu diesem Zweck von der *SharkEngine* bereitgestellten Methoden. Sie enthält wie in Abbildung 7.1 dargestellt das Kompositum, welches als Behälter dynamisch Teilfilter aufnehmen, löschen und die Reihenfolge der Ausführung ändern kann. Falls der Entwickler von der App-Ebene heraus Filter hinzufügen wollen, kann er dies über die *SharkNetApi* tun, welche die gewünschten Änderungen an die *SharkEngine* weiterleitet. Das folgende Codebeispiel zeigt diese verhältnismäßig leichte Handhabung:

Listing 9.5: Beispiel für die Anwendung der Filter

```
1 TopicFilter newTopicFilter = new TopicFilter(Dimension.TOPIC);
2 PeerFilter newApproverFilter = new PeerFilter(Dimension.APPROVERS);
3 mApi.addSemanticFilter(newTopicFilter);
4 mApi.addSemanticFilter(newApproverFilter);
5 boolean isInteresting = mApi.executeSemanticFilters(message,
    knowledge, profile);
```

Nachdem die Filter erzeugt und ihre Dimension festgelegt worden sind, werden sie über die *SharkNetApi* der *SharkEngine* hinzugefügt und können über die Methode *executeSemanticFilters()* für die Auswertung benutzt werden. Für die Auswertung muss zusätzlich die Eingansnachricht (*message*), die semantischen Annotationen der Nachricht (*knowledge*) und das aktuelle semantische Profil des Benutzers (*profile*) mit übergeben werden.

9.5 Test

[...]

9.6 Ausblick

Kapitel 10

Sonstiges

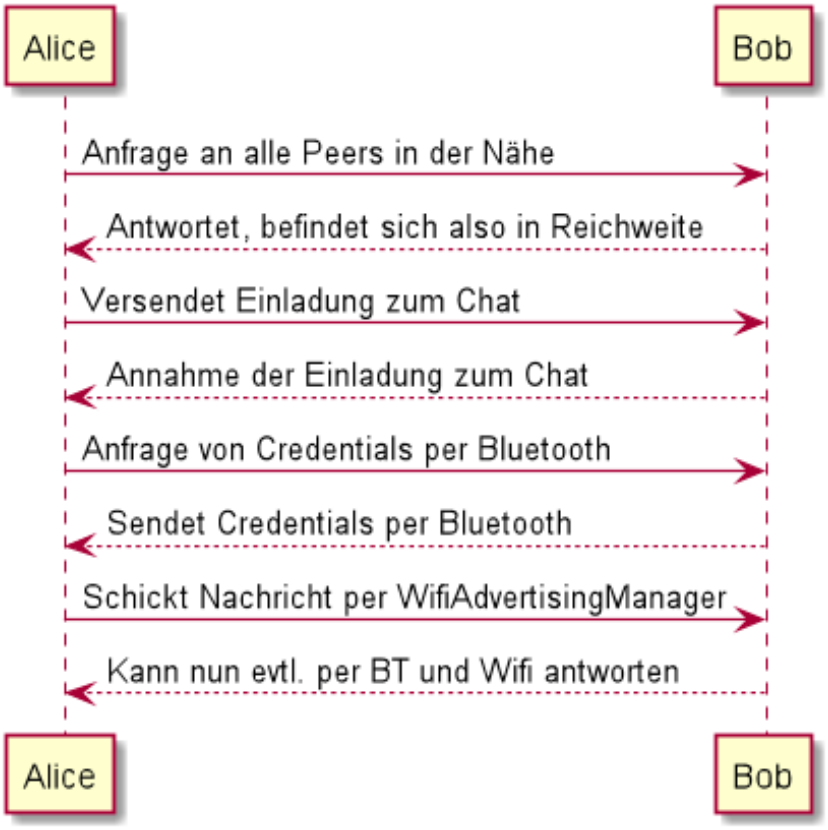

Abbildung 10.1: Kommunikation per Chat