Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Bachelorarbeit

Ad-hoc-Kommunikation zwischen aktiven Komponenten auf mobilen Geräten

Julian Kalinowski

8kalinow@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 6053975
Fachsemester 7

Erstgutachter: Prof. Dr. Winfried Lamersdorf
Zweitgutachter: Dr. Alexander Pokahr

Inhaltsverzeichnis

Kapitel 1

Einleitung

Nirgends ist der technologische Fortschritt so schnell und direkt nachvollziehbar wie aktuell bei mobilen Geräten [Ern11b]. Notebooks, Netbooks, Mobiltelefone, Smartphones und Tablets: Leistungsfähige Hardware wird immer kompakter, es werden ständig neue Geräteklassen definiert [Sch11]. Vollständige Computer mit Internetzugang in der Größe eines Mobiltelefons ermöglichen den ortsunabhängigen Zugriff auf Daten und bieten ihren Nutzern eine nie dagewesene Flexibilität. Eine große Rolle spielen dabei die vielen unterschiedlichen Anwendungen, die dem Nutzer diese neuen Möglichkeiten komfortabel zugänglich machen.

Die verteilten Anwendungen für diese Plattformen beschränken sich jedoch meist auf ein altbekanntes Schema: die Client/Server-Architektur. Dazu zählen Dienste wie E-Mail, das World Wide Web, Kartendienste und Fahrplanauskunftsprogramme. Selbst vermeintlich innovative Augmented Reality-[1], Social Network- oder Chat-Anwendungen nutzen das klassische Client/Server-Prinzip als grundlegende Kommunikationsarchitektur.

Im Bereich des Mobile Computing könnte also ein großes Potential brachliegen: Die Nutzung eben dieser Geräte in einer anderen Systemarchitektur, beispielsweise in einem Peer-to-Peer-System.

Peer-to-Peer-Systeme bestehen aus mehreren gleichberechtigten Rechnern, die standortunabhängig durch Nachrichtenaustausch zusammenwirken [DEF+08]. Darauf aufbauend können Anwendungen entwickelt werden, die von der Zusammenarbeit der unterschiedlichen Geräte profitieren, aber unabhängig von zentralen Kontrollinstanzen sind.

Eingebettet in verteilte Anwendungen können mobile Geräte aufgrund ihrer Portabilität Kontextdaten beitragen, indem sie mittels Sensoren und Antennen ihre variable Umgebung wahrnehmen und erfassen. Dazu stellen z.B. Smartphones Beschleunigungs-, Positions-, Licht- und Lautstärkesensoren bereit. Außerdem können sie selbst durch Ad-hoc-Kommunikation zwischen gleichberechtigten Geräten ein verteiltes Peer-to-Peer-System aufbauen, in dem sich wiederum andere Anwendungen realisieren lassen. Ad-

[1] Diese Anwendungen blenden in ein Kamerabild zusätzliche Informationen ein, z.B. Positionen von Sehenswürdigkeiten oder virtuelle Objekte

hoc-Vernetzung ermöglicht den infrastrukturlosen Austausch von Daten und somit den Aufbau eines verteilten Peer-to-Peer-Systems ohne zentrale Kontrolle [BMJ+98].
Vor allem Letzteres bietet Anwendungsszenarien (siehe Kapitel 2), die nicht ohne weiteres durch klassische Client/Server-Architekturen realisiert werden können.

Die Grundvoraussetzungen zur Entwicklung verteilter Anwendungen auf mobilen Geräten sind erfüllt: Die Geräte bieten durch verschiedenste Funktechnologien eine hohe Konnektivität, sind leistungsfähig und eine große Verbreitung von einheitlichen Plattformen wie *iOS* [App11] und *Android* [Ope11b] ermöglicht den Einsatz in der Praxis.

1.1 Motivation

Verteilte Anwendungssoftware unterscheidet sich im Wesentlichen dadurch von herkömmlichen Anwendungen, dass ihre Komponenten auf unterschiedlichen Systemen ausgeführt werden [DEF+08]. Diese echte Nebenläufigkeit führt dazu, dass nicht über einen gemeinsamen Speicher kommuniziert werden kann, sondern nur durch das Versenden von Nachrichten, und bedeutet einen hohen Aufwand für den Entwickler. In [PB11] wird der Ansatz der *aktiven Komponenten* für die Entwicklung verteilter Systeme vorgeschlagen, der drei bekannte Konzepte vereint: *Softwareagenten*, *Services* und *Komponenten*. Für die Entwicklung von Peer-to-Peer-Anwendungen im Kontext mobiler Geräte bietet sich dieser Ansatz an, denn im Unterschied zum klassischen Client/Server-Konzept steht hier die Zusammenarbeit von autonomen, aktiven Komponenten in einem gemeinsamen System im Vordergrund. Die Eigenschaften der Autonomie und Kooperation der aktiven Komponenten sind dem Softwareagentenparadigma entnommen, weshalb dieses die Basis für die weitere Motivation ist.
Ein Softwareagent ist nach [Jen00] definiert als autonome Softwareeinheit, die in einer Umgebung agiert und mit ihr interagiert, um festgelegte Ziele zu erreichen.
Jennings und Wooldridge definieren weiterhin in der *Weak Notion of Agency* [JW98] Agenten über vier wesentliche Eigenschaften:

Autonomie Agenten agieren ohne direktes Einwirken anderer und haben die Kontrolle über ihre Aktionen und ihren Zustand

Soziale Fähigkeiten Agenten kommunizieren mit dem Anwender oder anderen Agenten

Reaktivität Agenten nehmen ihre Umwelt wahr und reagieren auf Veränderungen

Proaktivität Agenten können selbstständig, auf eigene Initiative hin, Aktionen ausführen, um ihre Ziele zu erreichen

Vergleicht man die genannten Eigenschaften mit denen eines typischen Mobilgeräts, fallen Parallelen auf:

- Ein Mobilgerät muss autonom agieren können, da die Möglichkeit besteht, dass eine bestehende Infrastrukturverbindung aufgrund von Umwelteinflüssen abbricht; außerdem ist der Nutzer nicht ständig für Eingaben bereit

- Es kann durch kabellose Übertragungstechniken wie Infrarot, Bluetooth oder WLAN mit anderen Geräten Daten austauschen und so mit ihren kommunizieren

- Die Umwelt des Gerätes verändert sich ständig und es gibt Sensoren, die dies wahrnehmen können

Die *Proaktivität* ist keine grundsätzliche Eigenschaft eines mobilen Gerätes sondern abhängig von der Aufgabe, die eine Anwendung erfüllen soll.

Die Übereinstimmung in den Eigenschaften legt eine Repräsentation des mobilen Gerätes als Agent bzw. die Ausführung agentenbasierter Software nahe. Um Entwicklern wiederkehrende Aufgaben wie den Nachrichtenversand und -empfang und die Behandlung der Nebenläufigkeit abzunehmen, wurden verschiedene Plattformen entworfen, die diese und weitere Aufgaben erledigen.

Ein auf stationären Systemen nicht vorhandener Aspekt, der deshalb in den gängigen Plattformen nicht berücksichtigt wird, ist die Kommunikation zwischen räumlich benachbarten Agenten. Für diese Kommunikation, die Voraussetzung für soziale Fähigkeiten von Agenten auf mobilen Plattformen ist, ist die Ad-hoc-Vernetzung zwischen mobilen Plattformen notwendig. Da das Konzept der aktiven Komponenten gegenüber dem der Softwareagenten weitere Vorteile wie Konfigurierbarkeit und dynamische Dienstbindung mit sich bringt (vgl. Abschnitt 3.1.2), soll dieser Ansatz genutzt werden.

1.2 Zielsetzung

Die Arbeit soll im Kontext der aktiven Komponenten und mobiler Geräte Möglichkeiten zur autonomen, infrastrukturlosen Kommunikation untersuchen und eine für Entwickler sowie Benutzer einfach zu verwendende Implementation auf Basis von *Jadex* und der Geräteplattform *Android* hervorbringen.
Im Rahmen dessen sollen die nötigen Schnittstellen zur Integration der Ad-hoc-Kommunikation in die Jadex-Plattform und die Architektur von *Jadex-Android* entworfen werden. Für die Kommunikation der Komponenten via Bluetooth muss eine Architektur entwickelt werden, welche den Ansprüchen der Kommunikation zwischen Jadex-Plattformen und darauf laufenden Komponenten genügt sowie den Aufbau und die Verwaltung von Ad-hoc-Verbindungen ermöglicht.

Auf Basis der Implementation sollen auf Mobilgeräten Komponenten genutzt werden können, die auf anderen Geräten laufende Komponenten automatisiert auffinden und mit ihnen kommunizieren können. Außerdem sollen Deployment, Konfiguration und Monitoring von entfernten mobilen Plattformen berücksichtigt werden. Die Möglichkeit

einer Vermaschung zur Vergrößerung der Reichweite soll geprüft und ein dafür geeignetes Routing-Verfahren gefunden werden.

Komponenten, die das entwickelte Framework nutzen, sollen über eine einfache Schnittstelle alle in Reichweite befindlichen Komponenten abrufen können und mit diesen kommunizieren, ohne sich um die Details des Verbindungsaufbaus kümmern zu müssen.

Auf Basis der Implementation soll eine Beispielanwendung erstellt werden, die die Funktionalität im Hinblick auf die Kommunikation zwischen mobilen Plattformen veranschaulicht.

1.3 Aufbau der Arbeit

Nach dieser allgemeinen Einführung und Motivation soll der Zweck einer Ad-hoc-Kommunikation zwischen mobilen Geräten durch verschiedene Anwendungsszenarien (Kapitel 2) erläutert werden.
Anschließend werden die Grundlagen, die für die Realisierung notwendig sind, dargestellt. Dazu zählt die Jadex-Plattform an sich und die von ihr zur Kommunikation verwendeten Mechanismen, sowie Grundlagen zur Ad-hoc-Kommunikation wie Funktechnologien und Routing-Protokolle.
Im Rahmen dieser Arbeit werden im Kapitel 4 auch verwandte Arbeiten betrachtet und auf Überschneidungen geprüft.

Unter Verwendung der Grundlagen werden im Kapitel 5 Anforderungen aus den Anwendungsszenarien ermittelt um daraus eine Architektur abzuleiten, die den Ansprüchen gerecht werden kann. Darauf aufbauend wird in Kapitel 6 der Verlauf der Implementierung sowie der Einsatz in einem Anwendungsbeispiel mitsamt aufgetretener Probleme und deren Lösungen beschrieben.

Schließlich fasst das Kapitel 7 diese Arbeit und ihr Ergebnis zusammen und bietet einen Ausblick auf weitere Entwicklungsmöglichkeiten.

Kapitel 2

Anwendungsszenarien

Um den Nutzen der in der Einleitung beschriebenen, im Laufe dieser Arbeit noch zu entwickelnden Lösung zu veranschaulichen, werden in diesem Kapitel mögliche Anwendungsszenarien vorgestellt. Entsprechend dem primären Einsatzkontext aktiver Komponenten setzen alle Szenarien auf einer verteilten Anwendung auf. Dabei profitieren sie auf unterschiedliche Art und Weise von den Konzepten der aktiven Komponenten und sind auf die zu entwickelnde Ad-hoc-Kommunikation angewiesen.

2.1 Mehrspielerbingo

Als erstes Beispiel sei ein Mehrspielerbingo für Smartphones genannt, bei dem sich alle Spieler in einem Raum befinden, in dem die Zahlen oder Begriffe des Bingospiels genannt werden. Die Smartphones müssen sich untereinander finden und ein Ad-hoc-Netzwerk aufbauen, sodass den Spielern alle in der nahen Umgebung laufenden Bingospiele zum Beitreten angezeigt werden können bzw. sie neue Spiele eröffnen können.

Die Nutzung zusätzlicher Infrastruktur, wie z.B. eines Servers im Internet, bringt weitere Anforderungen (Internetverbindung, Austausch eines Passwortes) und Fehlerquellen (Verbindungsabbruch, Serverausfall), aber keinen direkten Mehrwert mit sich. Im Gegenteil führt die Verwendung der Funkvernetzung eine räumliche Beschränkung ein, die in diesem Fall sinnvoll ist, da sich das Bingo auf einen bestimmten räumlichen Kontext bezieht (z.B. auf in einer Vorlesung verwendete Ausdrücke). Durch die Verwendung von *Jadex-Android* muss der Abwicklung der Verbindungen und dem Auffinden möglicher Spielpartner keine Beachtung geschenkt werden. Die Kommunikationsarchitektur für das Spiel kann auf einem höheren Abstraktionsniveau entwickelt werden.

2.2 RoboCup

Der RoboCup [Rob11] ist ein von der *RoboCup Federation* ausgeschriebener internationaler Wettbewerb, in dem es in verschiedenen Bereichen um die Leistungsfähigkeit von intelligenten Robotern geht. Der erste und nach wie vor größte Bereich ist der *RoboCup*

Soccer, bei dem Roboter im Fußball gegeneinander antreten. In den drei größten der vier vorgegebenen Größenklassen müssen die Roboter vollständig autonom handeln und dürfen lediglich untereinander kommunizieren, was sie ideal als Anwendungsszenario für den Einsatz einer mobilen Plattform für aktive Komponenten qualifiziert.
Da beim Fußball das Zusammenspiel und die Teamfähigkeit eine große Rolle spielt, ist eine problemlose Ad-hoc-Kommunikation zwischen den Teampartnern wichtig und ihre *sozialen Fähigkeiten* sind entscheidend für die Leistung des Teams. Weiterhin ist durch die vorgeschriebene *Autonomie* die Nutzung der Agentenorientierung für die Implementierung der Robotersoftware naheliegend, zumal auch die weiteren Kriterien der *Weak Notion of Agency* erfüllt werden: *Proaktivität* ist notwendig, da sonst kein Roboter die Initiative ergreifen und als erster den Ball spielen würde, *Reaktivität* begründet sich beispielsweise aus der Abwehrreaktion des Torwartes oder der Verfolgung eines gegnerischen Spielers im Ballbesitz.

Als direkter Nutzen für den RoboCup kann *Jadex-Android* auf einem Mobiltelefon oder Tablet als Recheneinheit der Roboter dienen.

2.3 Datensynchronisation und verteiltes Forum

Mobile Geräte speichern heutzutage große Mengen an Daten. Meist handelt es sich um Kontaktdaten, Fotos oder Videos. Für all diese Daten ist ein Ad-hoc-Austausch mit anderen Nutzern denkbar und meistens auch bereits möglich.

Beim Kontaktdatenaustausch scheint der Nutzen einer intelligenten, automatisierten Lösung durch Verwendung von aktiven Komponenten am offensichtlichsten: Eine verteilte Synchronisationslösung könnte Telefonnummern und E-Mail-Adressen nicht nur wie bisher bereits üblich mit einem Desktop-Rechner synchron halten, sondern auch mit anderen mobilen Geräten.

Dabei kann das Potenzial der aktiven Komponenten dazu verwendet werden, lediglich gleiche, auf beiden Geräten vorhandene Kontakte abzugleichen und dabei veraltete durch neue Daten zu ersetzen.

Auf ähnliche Art und Weise kann ein mobiles Forum bzw. ein soziales Netzwerk entwickelt werden, welches von Nutzern geschriebene Beiträge im Netzwerk verteilt. Optional ist eine Verschlüsselung denkbar, die ähnlich wie in anonymen Netzen wie Freenet [fre11] zwar alle Daten möglichst weit im Netzwerk verteilt, sie aber nur für den eigentlichen Adressaten lesbar macht.
Mit diesem Konzept sind weitere Szenarien denkbar, die eine verteilte Datenbasis nutzen.

2.4 Gemeinsame Positionserfassung

Die Positionserfassung durch GPS ist für mobile Geräte mit relativ hohem Energieaufwand verbunden. Dazu kommt die Zeit, bis eine initiale Positionsbestimmung erfolgt ist. Wird

der GPS-Sensor vom Benutzer aktiviert, wünscht dieser jedoch eine möglichst schnelle Anzeige der Position, sei es für die Navigation zu Fuß oder mit dem Auto, oder zur Bestimmung in der Nähe befindlicher Ziele wie Geschäfte oder Restaurants.

Um den Energieaufwand und insbesondere die Zeitverzögerung zu minimieren, kann eine von einem Gerät erfasste Position mit anderen Geräten geteilt werden, für die dadurch der Aufwand für die Bestimmung der Position reduziert wird. Dabei ist die räumliche Beschränkung einer Ad-hoc-Funkverbindung wiederum hilfreich, da so die erfasste Position eines Gerätes maximal um den Betrag der Funkreichweite von den Positionen der anderen erreichbaren Geräte abweicht.

2.5 Zusammenfassung

Die in diesem Kapitel genannten Szenarien benötigen eine Peer-to-Peer-Kommunikation und profitieren von der Entwicklung auf Basis von aktiven Komponenten. Dabei stellen sie unterschiedliche Ansprüche an das zu entwickelnde Framework, die im weiteren Verlauf der Arbeit berücksichtigt werden.

Neben der Ermittlung der Anforderungen aus den Anwendungsszenarien müssen für den Entwurf weiterhin die Funktionsweise von Plattformen für aktive Komponenten sowie die technischen Möglichkeiten der Ad-hoc-Kommunikation bekannt sein, weshalb sich das nächste Kapitel mit diesen Grundlagen beschäftigt.

Kapitel 3

Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die zur Realisierung der Ad-hoc-Kommunikation zwischen aktiven Komponenten auf mobilen Geräten notwendig sind. Dazu gehört die Laufzeitplattform Jadex-Standalone, deren Bestandteile im Kontext von Agentenplattformen dargestellt werden, sowie die Grundlagen zur Ad-hoc-Funkkommunikation. Dazu werden WLAN und Bluetooth vorgestellt, zwei Technologien, die sich grundsätzlich für die infrastrukturlose, spontane Vernetzung mobiler Geräte eignen.

3.1 Plattformen für aktive Komponenten und Agenten am Beispiel von Jadex

Bereits in der Einleitung wurde das Paradigma der aktiven Komponenten für die Entwicklung verteilter Anwendungen vorgestellt. Als Grundlage für die Entwicklung aktiver Komponenten wird eine Plattform benötigt, die hier am Beispiel von Jadex [BP11b] betrachtet wird. Sie stellt die Basis für ein System aus verschiedenen Komponenten dar und übernimmt die Behandlung der Nachrichtenkommunikation, das Erstellen und Zerstören von Komponenten und bietet Abstraktion in Bezug auf die Nebenläufigkeit der Komponenten.

3.1.1 Jadex

Das an der Universität Hamburg entwickelte Jadex war ursprünglich als *BDI*[1] *Reasoning Engine*[2] für Agentenplattformen wie *JADE* [Til11] gedacht, bringt aber mittlerweile mit *Jadex-Standalone* seine eigene Plattform mit [BP11b]. Neben dem ursprünglichen BDI-Modell bringt Jadex weitere *Kernel* mit, die zur Ausführung von verschiedenen Agententypen und aktiven Komponenten genutzt werden können. Dabei ergeben sich unterschiedliche Ansprüche an Speicherkapazität und Rechenleistung, sodass die Art der genutzten Kernel

[1] Beliefs, Desires, Intentions: Ein Modell für Softwareagenten
[2] Reasoning bezeichnet den Prozess der Entscheidungsfindung [PBL05].

Kernel	Modellierung
BDI	durch Beliefs, Desires (Ziele), Intentions (Pläne)
BPMN (Business Process Modelling Notation)	durch Geschäftsprozesse
BDIBPMN	durch BDI, Pläne als BPMN
GPMN (Goal-oriented Process Modelling Notation)	durch die in [BPJ+10] vorgestellte BPMN Erweiterung
Micro	direkt in Java
Component	als *aktive Komponenten* (Abschnitt 3.1.2)

Tabelle 3.1: Übersicht Jadex-Kernel

und Modellierungsarten an die zur Verfügung stehende Leistung des Gerätes angepasst werden muss.

Die Tabelle 3.1 zeigt die Modellierungssprachen, die von den verschiedenen Kerneln unterstützt wird. Die BDI-Modellierung setzt auf eine Kombination aus XML-Dateien, in denen die Beliefs, Desires und Intentions deklariert werden, und Java Code, der die Intentions (Pläne) ausformuliert.
Die BPMN und GPMN [BPJ+10] Kernel erlauben eine Modellierung der Agenten als Geschäftsprozesse [Whi04] und extrahieren dabei den internen Programmfluss eines Agenten als Diagramm. Als Kombination der beiden genannten Modellierungsarten stellen BDIBPMN-Agenten ihre Beliefs und Desires in XML-Dateien dar, während die Ausformulierung der Pläne in BPMN-Diagrammen stattfindet.

Micro-Agenten werden direkt in Java implementiert, sind leichtgewichtig ausgelegt und benötigen kein aufwendiges Reasoning, sodass sie eine schnelle Ausführung auch auf leistungsschwächeren Geräten versprechen, aber auch die geringste Abstraktion und Flexibilität bieten.

Der Component Kernel stellt zusammen mit den BDI, BPMN und GPMN Kerneln die flexible und komplexe Seite von Jadex dar. Die entsprechende Modellierung der *aktiven Komponenten* [PB11] wird im folgenden Abschnitt erläutert.

3.1.2 Aktive Komponenten

Die vom Component Kernel ausgeführten Komponenten stellen ein neues Konzept dar, welches das Softwareagentenparadigma mit herkömmlichen Entwicklungsansätzen für verteilte Systeme als *aktive Komponenten* vereint [PB11].

Genau wie ein Softwareagent ist eine aktive Komponente eine autonome Einheit, die proaktiv und reaktiv agieren kann und mit anderen aktiven Komponenten interagiert. Weiterhin kann eine aktive Komponente *Dienste* anbieten und nutzen. Wie in der *Ser-*

vice Oriented Architecture (SOA) [DEF+08] werden Dienste dynamisch, also zur Laufzeit, gebunden und dienen der Kommunikation zwischen aktiven Komponenten, die durch Nachrichtenaustausch oder entfernte Methodenaufrufe stattfindet.

Dem Komponentenparadigma [McI68] entlehnt ist die mögliche Komposition mehrerer (Sub-) Komponenten zu einer gemeinsamen, umfangreicheren Komponente sowie die externe Konfiguration durch *properties* und *configurations*. Abbildung 3.1 zeigt eine aktive Komponente mit den genannten Eigenschaften eingebettet in die *Management Infrastructure*, die von der Jadex Plattform und ihren Plattformdiensten (vgl. Abschnitt 3.1.3) bereitgestellt wird.

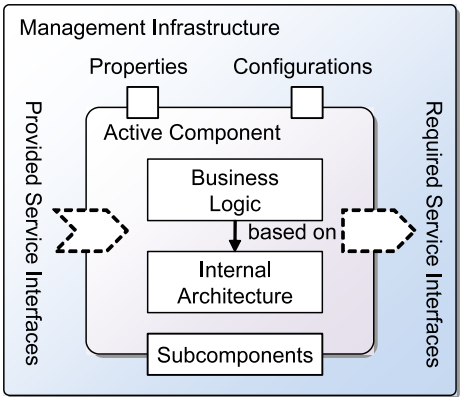

Abbildung 3.1: Struktur einer aktiven Komponente, aus [BP11a]

Die aktiven Komponenten beinhalten alle Konzepte der Softwareagenten und werden deshalb in dieser Arbeit als deren Obermenge verstanden.

3.1.3 Die Jadex-Standalone-Plattform

Wie bereits erwähnt, ist die Jadex Reasoning Engine auf eine Plattform angewiesen, die Jadex in Form von *Jadex-Standalone* mitliefert [BP11c]. Jadex-Standalone ist frei konfigurierbar. Das Verhalten der in Abbildung 3.2 skizzierten Plattform ergibt sich aus den gestarteten Diensten und Komponenten, zu denen auch die in Tabelle 3.1 gezeigten Kernel zählen.

Die grundlegenden Plattformdienste von *Jadex-Standalone* orientieren sich, wie die Dienste der ursprünglich genutzten Plattform JADE, an den FIPA[3]-Spezifikationen für Agentenplattformen [FIP04] und sind folgende:

CMS (Component Management System) Das *CMS*, abgeleitet vom FIPA *Agent Management System (AMS)*, bietet Operationen für die Lebenszyklusverwaltung von Komponenten, wie das Erstellen, Starten, Beenden und Zerstören einer Komponente, liefert Informationen über die laufenden Komponenten und kann Komponenten anhand von Suchkriterien finden.

[3]Die Foundation for Intelligent Physical Agents verfolgt u.a. das Ziel, agentenbasierte Kommunikation zu standardisieren und propagiert in diesem Zusammenhang einen Nachrichtenstandard.

DF (Directory Facilitator) Der *DF* ist ein Registrierungsdienst für die von Komponenten angebotenen Dienste. Er bietet die An- und Abmeldung von Diensten, die Modifikation von bereits vorhandenen Diensteinträgen und das Suchen nach vorhandenen Diensteinträgen.

MS (Message Service) Der *Message Service* bietet das Senden und Zustellen von Nachrichten an. Beim Versenden von Nachrichten an eine entfernte Plattform wird je nach Empfänger und Möglichkeiten des Netzwerks ein passender *Transport* (vgl. Abschnitt 3.1.4) ausgewählt, der auf Netzwerkebene mit der entfernten Plattform kommuniziert. Die Zustellung an die Zielkomponente wird schließlich wieder vom *MS* der entfernten Plattform übernommen.

Neben diesen Plattformdiensten werden auch alle Kernel, die die Plattform unterstützen soll, sowie eine Reihe von zu ladenden Standardkomponenten in der Plattformkonfiguration festgelegt. Dazu gehören der *Awareness Management Agent* (siehe Abschnitt 3.1.5), das *Jadex Control Center (JCC)* als grafische Oberfläche zur Plattformverwaltung, und der *Remote Service Management Service (RMS)*, der entfernte Serviceanfragen ermöglicht.

Der für die Kommunikation entscheidende Message Service vermittelt ausgehende Nachrichten an einen oder mehrere Zielkomponenten (*sendMessage*). Dabei nutzt er die in der Plattformkonfiguration eingetragenen *Transports* [BP11b]. Er nimmt außerdem eingehende Nachrichten von Transports entgegen, um sie an die lokale Zielkomponente zuzustellen (*deliverMessage*).

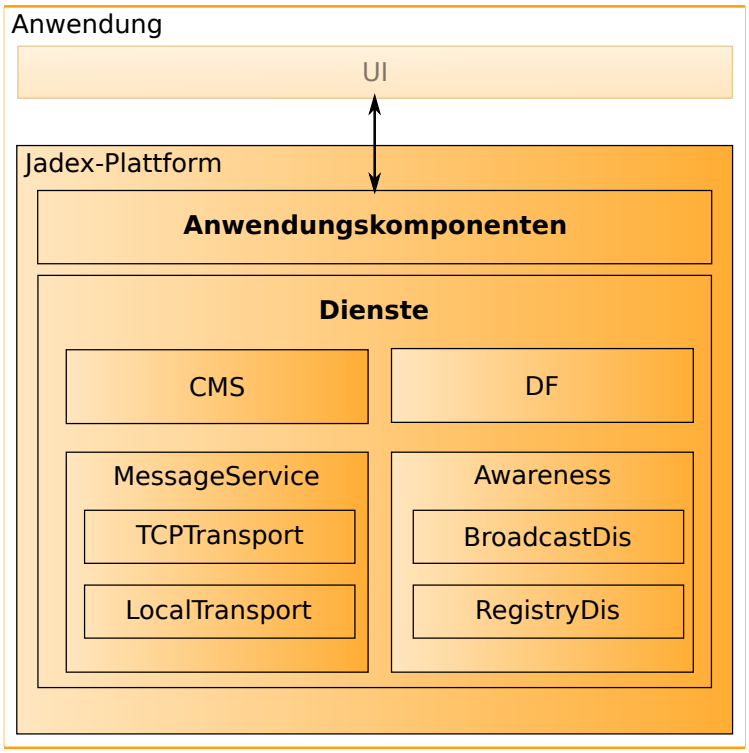

Abbildung 3.2: Architektur der Jadex-Plattform

3.1.4 Transports

Jeder Transport unterstützt ein selbst definiertes Adressierungsschema. Wenn der Transport vom Message Service aufgerufen wird, prüft er zunächst seine Zuständigkeit anhand der Zieladresse und beginnt anschließend mit der Übermittlung der Nachricht [BP11b].

Jadex bringt zwei verschiedene Transports mit: Zum einen den *LocalTransport*, der Nachrichten plattformlokal zustellen kann. Dabei ruft er lediglich die *deliverMessage*-Methode des Message Service auf.
Der *TCPTransport* dagegen startet, sobald die Plattform initialisiert wird, einen TCP Server, um eingehende Nachrichten anderer Plattformen im Netzwerk zu empfangen. Beim Senden von Nachrichten verbindet er sich mit dem TCP Server der Zielplattform, der die Nachricht entgegen nimmt und durch die *deliverMessage*-Methode an den Message Service übergibt. Von dort wird die Nachricht wiederum an die Zielkomponente weitergeleitet.

Durch das Zusammenspiel von Message Service und Transports (siehe Abbildung 3.2) ist bei der Nachrichtenvermittlung eine Netzwerktransparenz gegeben. Jedoch muss einer Plattform manuell mitgeteilt werden, welche entfernten Komponenten unter welcher Adresse verfügbar sind. Um diese Konfiguration zu vermeiden, wurde die im nächsten Abschnitt beschriebene *Jadex Awareness* entworfen.

3.1.5 Jadex Awareness

Damit eine Jadex-Plattform automatisch über angebotene Dienste anderer erreichbarer Plattformen Kenntnis hat, bedarf es eines Mechanismus, der diese Daten mit anderen Plattformen im Netzwerk austauscht. Zu diesem Zweck wurde die *Awareness* entwickelt, die auf jeder Plattform durch einen *Awareness Management Agent* repräsentiert wird [BP11b]. Dieser nutzt je nach Konfiguration verschiedene Discovery-Agenten (vgl. Abbildung 3.2), um sich mit anderen Plattformen auszutauschen.
Als Discovery-Agent kann z.B. der *Broadcast Discovery Agent* genutzt werden, der über eine IP-Broadcast-Adresse[4] mit anderen Agenten des gleichen Typs kommuniziert, oder der *Registry Discovery Agent*, der die Plattform in einem zentralen Verzeichnis anmeldet und von dort auch die Informationen über andere Plattformen abrufen kann.
Der spätere Nachrichtenaustausch findet über den *Message Service* und die von ihm genutzten *Transports* statt.

Die Jadex-Awareness legt den Grundstein für eine konfigurationslose Kommunikation verschiedener Plattformen untereinander, ohne Kenntnis von Adressen oder Netzwerkdetails.

[4] IP-Pakete, die an eine Broadcast-Adresse gesendet werden, werden von allen Teilnehmern im Netz empfangen.

3.2 Ad-hoc-Netzwerke

Um die Netzwerkfähigkeiten von Jadex auch in mobilen Umgebungen einzusetzen, benötigt man ein Funknetzwerk, welches die mobilen Geräte miteinander verbindet. Um weiterhin den Anwendungsszenarien (vgl. Kapitel 2) gerecht zu werden, darf es dabei keine feste Infrastruktur geben. Ein solches spontanes, infrastrukturloses Funknetzwerk heißt *Ad-hoc-Netzwerk* [MM04]. Um die Jadex Awareness (vgl. Abschnitt 3.1.5) auch auf mobilen Geräten durch drahtlose Kommunikation nutzbar zu machen und den Vorteil der Konfigurationslosigkeit beizubehalten, muss die Netzwerkschicht ebenfalls konfigurationslos funktionieren. Sie soll also nach Möglichkeit ohne die Eingabe von Parametern wie Name, Passwort oder Adressen auskommen.

Im Detail bedeutet dies für das benötigte Funknetzwerk: Es gibt keine feste, vorgegebene Infrastruktur in Form von Access Points, Switches oder Routern. Vielmehr wird das Netz erst bei Bedarf aufgebaut und die Infrastruktur besteht lediglich aus den Teilnehmern selbst [MM04, S.193]. In welcher Art und Weise die Teilnehmer miteinander kommunizieren können und welche Einschränkungen es dabei gibt, hängt von der verwendeten Funktechnologie und den darauf aufsetzenden Protokollen ab und wird im Folgenden anhand der weit verbreiteten Technologien *WLAN* (Abschnitt 3.2.1) und *Bluetooth* (Abschnitt 3.2.2) dargestellt.

Dabei werden die technischen Details der Funkübertragung außer Acht gelassen und die für eine Nutzung der Technologien aus Entwicklersicht interessanten Aspekte erläutert.

3.2.1 WLAN

Als *Wireless Local Area Network (WLAN)* wird in dieser Arbeit ein Funknetz bezeichnet, das dem *IEEE*[5] *802.11-Standard* [IEE11a] entspricht, welches der de-facto Standard für WLANs ist. Er wurde entwickelt, um die bisher kabelgebundenen *Local Area Networks (LANs)* zu ersetzen bzw. zu ergänzen [MM04, S. 63]. WLAN hat nach seiner breiten Einführung in Notebooks durch die fortschreitende Miniaturisierung auch in kleineren mobilen Geräten wie Smartphones Einsatz gefunden, sodass es dort mittlerweile eine sehr große Verbreitung vorweisen kann.

Der IEEE-802.11-Standard spezifiziert die *Physische-* und *Medienzugriffsschicht*, also die beiden untersten Schichten des OSI-Referenzmodells [Rot05, S.17ff] und nutzt dabei dasselbe Adressierungsschema wie im herkömmlichen *Ethernet*-Protokoll für kabelgebundene LANs. Er unterscheidet zwei Betriebsarten: Den Infrastruktur- und den Ad-hoc-Modus [MM04, S. 69].

Im Infrastrukturmodus agiert ein *Access Point (AP)* als zentraler Zugangspunkt für die mobilen WLAN Stationen, die *Member Stations (MTs)*. Die MTs kommunizieren nicht direkt untereinander, sondern über den AP als zentralen Vermittler. Access Points können über

[5] Die *Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA)* entwickelt Standards in naturwissenschaftlichen Forschungsgebieten und der Informatik.

herkömmliche kabelgebundene Schnittstellen mit einer bestehenden LAN-Infrastruktur verbunden werden und dank der gemeinsamen Adressierung auf Medienzugriffsschicht als Brücke zwischen LAN- und WLAN-Geräten agieren.

Im Ad-hoc-Modus wird auf einen zentralen Zugangspunkt verzichtet, stattdessen kommunizieren mehrere MTs direkt miteinander. Eine *Vermaschung*, also eine Erweiterung der Reichweite des Netzwerks durch die Weiterleitung von Nachrichten, wird im IEEE-802.11s-Standard [IEE11b] beschrieben, der erst kurzfristig im Laufe dieser Arbeit vervollständigt wurde [Hei11] und daher nicht mehr berücksichtigt werden konnte. In der Praxis sind aber ohnehin noch keine Geräte verfügbar, die den neuen Standard unterstützen.
Somit sind Ad-hoc-Netze auf IEEE 802.11-Basis auf eine geringe Reichweite beschränkt, in der sich alle beteiligten Geräte befinden müssen, oder auf Vermaschung in höheren Protokollschichten angewiesen [WXA09, S. 8f].
Selbst im Ad-hoc-Modus bedarf es beim WLAN einer vorherigen Konfiguration: Alle Stationen müssen denselben *Service Set Identifier (SSID)* sowie dieselben Verschlüsselungseinstellungen nutzen.
In der Praxis unterstützen einige WLAN-Geräte nur den Infrastruktur-Modus, beispielsweise Geräte auf Android-Basis[6].

3.2.2 Bluetooth

Bluetooth ist ein Industriestandard für Kurzstreckenfunk, der von der *Bluetooth Special Interest Group* spezifiziert wird [Blu10]. Die Reichweite einer Verbindung beträgt je nach Geräteklasse 10-100m, wobei die meisten mobilen Geräte eine Reichweite von 10m (Klasse 3) erreichen [Rot05, S. 136]. Im Gegensatz zu WLAN dient Bluetooth nicht in erster Linie zur drahtlosen Anbindung von mobilen Geräten an eine Infrastruktur, sondern zum Aufbau von Punkt-zu-Punkt Verbindungen zwischen zwei Geräten und bietet die Vernetzung mit einer Infrastruktur gar nicht erst an. Es wurde mit Blick auf den begrenzten Energievorrat und das eingeschränkte Speicherangebot mobiler Geräte entwickelt [Rot05, S.136].

Bluetooth erlaubt die Vernetzung von bis zu acht Geräten in einem *Personal Area Network (PAN)*, im Bluetooth-Jargon als *Piconetz* bezeichnet [MM04, S.91]. Ein Piconetz orientiert sich an der klassischen Client/Server-Architektur. Jegliche Kommunikation findet nur zwischen einem *Master* und einem der sieben aktiven *Slaves* statt. Eine direkte Kommunikation der Slaves untereinander ist nicht vorgesehen sondern muss auf höherer Protokollschicht implementiert werden.

Während ein Gerät nur in einem Piconetz als Master agieren kann, kann es in mehreren Piconetzen als Slave teilnehmen. Dadurch werden sogenannte *Scatternetze* möglich, die aus mehreren, sich überlappenden Piconetzen bestehen (Abbildung 3.3). Die Scatternetze legen den Grundstein für eine mögliche Vermaschung. Anders, als man annehmen würde, spezifiziert der Bluetooth-Standard jedoch kein Nachrichten-Routing zwischen verschie-

[6] siehe http://code.google.com/p/android/issues/detail?id=82

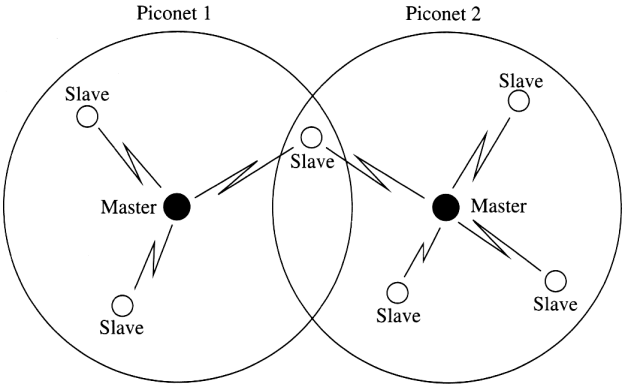

Abbildung 3.3: Bluetooth Scatternetz, aus [MM04, S.95]

denen Piconetzen [Blu10]. Dies kann nur durch eigene Routing-Protokolle ermöglicht werden (siehe Abschnitt 3.2.3).

In den nächsten Teilabschnitten werden Details des Verbindungsaufbaus und der zur Kommunikation notwendigen Profile sowie die Bluetooth-Authentifizierung beschrieben.

Verbindungsaufbau

Da Bluetooth für die spontane Vernetzung zwischen mobilen Geräten entwickelt wurde, erfordert der Verbindungsaufbau keine gemeinsame Vorkonfiguration der Geräte [MM04, S.94]. Bluetooth definiert eine Reihe von *States*, die den aktuellen Zustand eines Geräts repräsentieren. Ein Bluetooth-Gerät muss sich zum Verbindungsaufbau im *Inquiry State* befinden, der in drei Unterklassen eingeteilt ist.

Inquiry. In diesem Modus sendet der potentielle Master Inquiry-Pakete an die Umgebung um mögliche Slaves zu finden und wartet auf deren Antwort.

Inquiry Scan. Ein Slave-Gerät befindet sich in diesem Modus, wenn es von einem Master gefunden werden soll und horcht auf Inquiry-Pakete.

Inquiry Response. Der Slave sendet eine Antwort auf das Inquiry-Paket vom Master und meldet sich so als verfügbar.

Nachdem der Master Informationen über die verfügbaren Slaves in der Umgebung erhalten hat, wechselt er in den *Page State*, der ebenfalls slave-seitige Entsprechungen hat.

Page. Im Page State sendet der Master ein Page-Paket an einen Slave, den er in sein Piconetz aufnehmen will und wartet auf eine Antwort.

Page Scan. Ein Slave im Page-Scan-Modus wartet auf ein Page-Paket vom Master.

Page Response. Der Slave sendet eine Antwort auf das Page-Paket vom Master und ist damit Teil seines Piconetzes.

Sind einem Master bereits Geräte aus früheren Inquiry-Phasen bekannt, kann er auch direkt in den Page-Modus wechseln. Ist die Verbindung etabliert (*Active Mode*), wird sie durch das *Link Manager Protocol* verwaltet, über das unter anderem auch ein Rollentausch vorgenommen werden kann, bei dem der Master seine Rolle an einen Slave übergibt, um selbst Slave zu werden.

Der Bluetooth-Verbindungsaufbau ermöglicht zunächst lediglich eine Kommunikation der untersten Schichten des Bluetooth-Stacks. Damit Anwendungen die Verbindung nutzen können, müssen sie sich der *Profile* bedienen.

Profile

Bluetooth definiert eine Reihe von Anwendungsprofilen, die die Zusammenarbeit von Anwendungen auf unterschiedlichen Geräten garantieren [Blu10, Vol.1, S.93]. Ein Profil schreibt dabei die benötigten Features der Hardware, Software sowie den Ablauf der Kommunikation inklusive verwendeter Datenformate vor. Alle Profile beschreiben außerdem, wie ein entferntes Gerät sich unter Nutzung des Profils verbinden und angebotene Dienste finden kann (siehe **Service Discovery Protocol (SDP)**) und ob dafür ein vorheriges Pairing (siehe Abschnitt **Pairing**) notwendig ist.

Eine Sonderrolle nimmt dabei das *Generic Access Profile (GAP)* ein, ein Basisprofil, welches von allen Bluetooth-Geräten implementiert wird und die Basis für weitere Profile ist. Es beinhaltet den Aufbau und die Verwaltung von Verbindungen zwischen Master und Slaves wie im Abschnitt **Verbindungsaufbau** beschrieben.

Die übrigen Profile lassen sich nach [MM04, S.98f] in 3 Klassen gliedern:

- **Telefonieprofile** dienen der Verbindung von Telefon und Headset oder der direkten Sprachverbindung zwischen zwei Geräten
- **Netzwerkprofile** ermöglichen Bluetooth-Geräten den Zugang zu LANs oder die Freigabe von mobilen Internetverbindungen an andere Geräte
- **Serielle und OBEX-Profile** emulieren eine serielle Schnittstelle (*Radio Frequency Communication, RFCOMM*) oder ermöglichen den kabellosen Dateitransfer (Object Exchange)

Von den genannten Profilen ist RFCOMM für die Entwicklung mit Bluetooth besonders interessant. Es emuliert serielle Schnittstellen für die transparente Nutzung von Bluetooth durch Anwendungen und erlaubt eine zuverlässige, paketbasierte Datenübertragung über Input/Output Streams [Blu10, Vol1, S. 42f].

Neben den vorgesehenen Profilen lassen sich eigene definieren, die dann genau wie die vorgegebenen Profile mithilfe des *Service Discovery Protocols* von anderen Geräten gefunden werden können.

Service Discovery Protocol (SDP)

Das *Service Discovery Protocol* definiert die Dienstabfrage (einschließlich Bluetooth-Profilen) von einem SDP-Client an einen auf jedem Bluetooth-konformen Gerät laufenden SDP-Server [Blu10]. Der SDP-Server führt im SDP-Register eine Liste von *Service Records*, die sich anhand einer UUID[7] identifizieren lassen. Neben Bluetooth-Profilen lassen sich Dienste beim SDP-Server registrieren, die ein bereits vorhandenes Profil nutzen. Eine Anwendung kann auf diese Weise unter Verwendung eines Profils wie RFCOMM einen Dienst beim SDP-Server registrieren, sodass dieser für andere Geräte sichtbar ist. Ein entferntes Gerät kann per SDP entweder unter Nutzung einer bekannten UUID einen Dienst ansprechen oder per *Browsing* alle verfügbaren Dienste abrufen.

Die Profile, die im SDP-Register eingetragen sind, bestimmen, ob für den Verbindungsaufbau ein vorheriges *Pairing* notwendig ist, dessen Ablauf im nächsten Teilabschnitt erläutert wird.

Pairing

Beim *Pairing* werden zwei Bluetooth-Geräte miteinander vertraut gemacht und tauschen ein gemeinsames Geheimnis aus, sodass sie anschließend verschlüsselte Verbindungen aufbauen können. Die *Security-Manager*-Schicht im Bluetooth Stack behandelt das *Pairing* von Geräten und die damit verknüpfte Verschlüsselung von Verbindungen [Blu10].
Die Pairing-Informationen werden vom Gerät gespeichert, sodass bei einem wiederholten Verbindungsaufbau kein erneutes Pairing erforderlich ist.

Nachdem das Pairing vom Nutzer bzw. einer Anwendung initiiert wurde gibt es folgende Methoden zum Geheimnisaustausch, die sich an den Ein-/Ausgabemöglichkeiten der Geräte orientieren [Blu10, Vol 2, S. 1094]:

Just Works Dieser Mechanismus erfordert keine Nutzerinteraktion. Er wird von Geräten ohne Eingabemöglichkeit genutzt und bietet keinen Schutz vor *Man-in-the-middle*-Angriffen[8].

Numeric comparison Wenn beide beteiligten Geräte zumindest ein Display und eine Taste zum Bestätigen haben, kann ein sechsstelliger Zahlencode generiert und auf beiden Geräten angezeigt werden. Die Nutzer müssen den angezeigten Code vergleichen und ihn bestätigen. So kann ein Man-in-the-middle-Angriff verhindert werden.

Passkey Entry Hat ein Gerät ein Display, das andere Display und Tastatur, generiert das erste Gerät einen Code, der auf dem zweiten eingegeben werden muss. Haben beide

[7] Universal Unique Identifier: Weltweit eindeutige Kennzeichnung, im Fall von SDP Service Records müssen diese nur SDP-Server-eindeutig sein.

[8] Bei einem man-in-the-middle-Angriff fängt ein Angreifer den Verbindungsaufbau ab, baut seinerseits eine Verbindung mit dem Ziel auf und leitet alle Daten weiter. Dabei kann er sie mitschneiden oder verändern.

Geräte eine Tastatur, müssen beide Geräte denselben Code eingeben. Auch hier wird ein Man-in-the-middle-Angriff verhindert.

Out-of-Band Bei dieser Methode wird ein anderer Kanal zum Austausch des Geheimnisses genutzt, beispielsweise die *Near Field Communication (NFC)*. Dabei geht der Schutz vor Man-in-the-middle-Angriffen nur soweit, wie der der genutzten Out-of-Band-Übertragung.

In allen Fällen wird aus dem so erhaltenen numerischen *Temporary Key (TK)* ein Schlüssel generiert, mit dem die Verbindung verschlüsselt wird. Ob ein Pairing notwendig ist und wenn ja, welche Art vorausgesetzt wird, entscheidet die im genutzten Profil hinterlegte Sicherheitsanforderung.

In diesem Abschnitt der Grundlagen wurde die Funktechnologie Bluetooth mitsamt Details zum Verbindungsaufbau, den Bluetooth-Profilen und dem Pairing zwischen Geräten erklärt. Um mit Bluetooth ein Netzwerk aufzubauen, braucht es aber mehr als die Verbindung zwischen zwei Geräten. Die Beschränkung auf acht vernetzte Geräte (vgl. Abschnitt 3.2.2) lässt sich durch die Kopplung mehrerer Piconetze umgehen. Damit Slaves untereinander und Teilnehmer unterschiedlicher Piconetze miteinander Nachrichten austauschen können, bedarf es eines Routing-Protokolls.

3.2.3 Routing

In jedem Netzwerk wird zur Nachrichtenvermittlung ein Routing-Protokoll benötigt, das dafür sorgt, dass jeder Teilnehmer mit jedem anderen Nachrichten austauschen kann [Rot05, S. 164].
Im einfachsten Fall ist dies trivial - z.B. in sternförmigen Netzen, wo ein zentraler Knotenpunkt alle anderen Teilnehmer erreichen kann. Dies ist bei Bluetooth-Piconetzen der Fall, nicht jedoch, wenn es gilt, mehrere Piconetze zu einem Scatternetz zusammenzufassen.

In kabellosen, mobilen Ad-hoc-Netzwerken gibt es zusätzliche Aspekte, die ein Routing-Protokoll berücksichtigen muss [MM04, S.299ff]:

Mobilität Die Topologie eines kabellosen Ad-hoc-Netzwerkes ist hochdynamisch. Knoten können sich bewegen und so bestehende Pfade unterbrechen. Zwar können auch herkömmliche Routing-Protokolle neue Pfade finden, sie konvergieren jedoch nur langsam[9].

Bandbreitenbeschränkung Im Vergleich mit kabelgebundenen Netzwerken ist die Übertragungsgeschwindigkeit durch die beschränkte Bandbreite in Funknetzen deutlich geringer, worauf auch die Routing-Protokolle Rücksicht nehmen müssen. Die Kenntnis der gesamten Netztopologie stellt zwar für kabelgebundene Netze einen Vorteil dar, durch die Dynamik in Ad-hoc-Netzen kann die dafür zu übertragende Datenmenge aber zum Nachteil werden.

[9] Konvergenz bezeichnet den Zustand, in dem die Routing-Informationen in allen Knoten nach einer Änderung des Netzes wieder aktuell sind.

Ressourcenbedarf Auf mobilen Geräten stellen Akkukapazität und Rechenleistung stark beschränkte Ressourcen dar, die ein ideales Routing-Protokoll ebenfalls berücksichtigen muss.

In dieser Auflistung unberücksichtigt bleiben Probleme durch die Qualitätsschwankungen der Funkübertragung an sich und solche, die unter anderem durch Überlappung der Sendereichweite und -frequenz zustande kommen, beispielsweise das *hidden-station*-Problem[10] sowie die hohe Fehlerquote [MM04, S.207]. Sie sind nur durch Protokolle auf hardware-naher Ebene zu adressieren und deshalb für diese Arbeit nicht relevant, die auf bestehenden Funkstandards und -Verbindungen aufsetzen soll. Sowohl WLAN [MM04, S.72ff] als auch Bluetooth [Rot05, S.137] nutzen bereits auf der Verbindungsschicht entsprechende Mechanismen.

Routing-Verfahren, die für mobile Ad-hoc-Netzwerke geeignet sind, müssen, um der Mobilität gerecht zu werden, *adaptiv* sein, sich also bei Topologieveränderungen anpassen können.
Man unterscheidet *proaktive* und *reaktive* Verfahren. Proaktive Routing-Verfahren halten in jedem Knoten Tabellen zur Wegfindung zu allen Knoten im Netzwerk vor, während reaktive Verfahren den Weg zum Zielknoten erst bei Bedarf ermitteln [Rot05, S. 165].

Weiterhin unterscheidet man verschiedene Klassen von Routing-Verfahren wie die *Distance-Vector-*, *Link-State-* [Rot05, S.165], *Source-Routing-* [Rot05, S.175] und *Link-Reversal-*Verfahren [Rot05, S.191].

Nachfolgend werden exemplarisch drei unterschiedliche Ansätze vorgestellt:
DSDV als Vertreter der Distance-Vector-Protokolle, *DSR*, ein Source-Routing-Protokoll, bei dem die gesamte Route in jedem Paket übertragen wird und *TORA*, ein Link-Reversal-Protokoll, das eine lokale Rekonfiguration erlaubt.

Bei Link-State-Verfahren werden die Informationen über direkte Nachbarn an alle Knoten im Netz verschickt, sodass jeder Knoten die gesamte Netztopologie kennt und anhand dieser den optimalen Pfad wählen kann. Das macht Link-State-Protokolle sehr aufwendig in Bezug auf den Bedarf an Rechenleistung und Speicher [Rot05, S.165f] und somit ungeeignet für mobile Geräte, weshalb an dieser Stelle kein Beispiel genannt wird.

DSDV

Das *Destination-Sequenced-Distance-Vector*-Routing-Protokoll (DSDV) [PB94] ist ein proaktives Verfahren [MM04, S.308]. Jeder Knoten führt eine Tabelle mit allen erreichbaren Knoten im Netz, die die Kosten des Pfades (üblicherweise in Hops[11]), den nächsten Knoten auf dem Pfad und eine Sequenznummer enthält, die die von diesem Knoten verschickten Tabellenupdates zählt, um Schleifen zu vermeiden und das *count-to-infinity*-Problem [Rot05,

[10] Das hidden-station-Problem entsteht, wenn eine sendewillige Station A wartet, bis die Funkfrequenz vermeintlich frei ist, aber eine andere Station B außerhalb der Reichweite ebenfalls zu diesem Zeitpunkt sendet und sich so die Signale beim Empfänger C, der in Reichweite beider Stationen A und B ist, überlagern.
[11] Ein Hop ist der Übergang von einem Knoten zum nächsten.

S.169] zu lösen.

Wann immer ein Knoten die Verbindung zu einem seiner Nachbarn verliert oder einen neuen Nachbarn bekommt, sendet er den neuen Tabelleneintrag an alle seine Nachbarn. Eine verlorene Verbindung bekommt dabei den Kosteneintrag ∞. In jedem Eintrag ist die Sequenznummer enthalten, die ein Knoten beim Verschicken einer Updatenachricht erhöht.
Empfängt ein Knoten einen neuen Tabelleneintrag und ist die Sequenznummer des neuen Eintrags höher als die des alten, aktualisiert er den betroffenen Eintrag und verschickt seinerseits ein Update an alle Nachbarn.
Zusätzlich zu diesem ereignisgesteuerten Senden von Routing-Informationen sendet jeder Knoten periodisch seine gesamte Routing-Tabelle an alle Nachbarn. Zusammen mit einem Zeitstempel und einem Timeout für die Gültigkeit von Routen vermeidet dies ungültige Einträge beim Ausfall eines Knotens [Rot05, S.170].

Vor- und Nachteile Der Vorteil im DSDV-Protokoll liegt darin, dass jeder Knoten den Weg zu jedem Ziel kennt, ohne erst andere Knoten befragen zu müssen. Außerdem kennt jeder Knoten alle erreichbaren Knoten im Netz.
Die Nachteile sind viele Update-Nachrichten bei hoher Mobilität der Knoten, die sich proportional zur Knotenanzahl verhalten.

DSR

Das *Dynamic-Source-Routing*-Protokoll [Joh94] ist ein reaktives Protokoll [MM04, S.317]. Es fordert Pfade zum Ziel erst dann aus dem Netzwerk an, wenn sie gebraucht werden, also beim Senden eines Pakets. Im Unterschied zu Distance-Vector-Algorithmen, enthält jedes Paket den vom Sender ermittelten kompletten Pfad zum Ziel im Header (*Source Routing*).

Zum Anfordern eines Pfades dient der **Route-Discovery**-Mechanismus des Protokolls: Um den Pfad zum Ziel zu finden verschickt der Sender ein *Route-Request*-Paket an seine Nachbarn. Diese leiten das Paket ebenfalls an alle Nachbarn weiter (*Flooding*), bis der Zielknoten schließlich die Anfrage erhält. Eine Sequenznummer im *Route Request* dient der Vermeidung von Schleifen, des Weiteren wird der Pfad, den ein *Route Request* zurücklegt, in der Anfrage gespeichert. Der Zielknoten sendet ein *Route Reply* auf dem umgekehrten Pfad zurück, wodurch der Sender jetzt die Route zum Zielknoten kennt und die eigentlichen Daten verschicken kann.
Da jedes Datenpaket die gesamte Route zum Ziel enthält, können Knoten, die ein Paket weiterleiten, dem Paket die Routing-Informationen entnehmen und zwischenspeichern, um so zukünftige *Route Requests* beantworten zu können, bevor sie das Ziel erreicht haben (*Caching*).

Der zweite Teil des Protokolls, die **Route Maintenance**, dient der Behandlung von Verbindungsabbrüchen. Wenn der nächste Knoten auf dem Weg eines Pakets nicht mehr erreichbar ist, wird eine *Route-Error*-Nachricht an den Sender geschickt. Alle Knoten auf

dem Weg des *Route-Error*-Pakets löschen den ungültigen Eintrag dann aus ihrem Cache, der Sender muss ein neues *Route-Request*-Paket verschicken.

Vor- und Nachteile Der reaktive Ansatz vermeidet das häufige Senden von Tabellen-Updates, wie es in DSDV nötig ist. Eine Route wird nur gebildet, wenn sie benötigt wird; trotzdem werden durch das Caching in den Knoten Mehrfachanfragen vermieden.
Ein Nachteil des *Route Maintenance* ist, dass defekte Verbindungen in angrenzenden Knoten nicht durch gültige Routen ersetzt werden. Veraltete Routing-Informationen können außerdem zu Inkonsistenzen führen, da nicht alle Knoten über ausgefallene Verbindungen informiert werden [MM04, S.320].

Insgesamt wird der Verzicht auf lokale Routing-Tabellen mit einer erhöhten Verbindungsaufbauzeit und größeren Paketheadern, die die gesamte Route enthalten müssen, erkauft. In Netzen mit hoher Mobilität bricht die Performance von DSR ein, da die *Route Discovery* dann einen Großteil des Datenverkehrs ausmacht [MM04, S.320].

TORA

Der *Temporally Ordered Routing Algorithm (TORA)* ist ein reaktiver *Link-Reversal*-Algorithmus [Rot05, S.201], bei dem jedem Knoten eine Höhe in Bezug auf einen Zielknoten zugeordnet wird. Betrachtet man den Paketfluss als Wasser und die Verbindungen zwischen Knoten als Rohre, erschließt sich die Funktionsweise des Algorithmus: Ein Paket wird immer durch eine Verbindung aus einem Knoten geleitet, die in Richtung eines Knotens mit niedrigerer Höhe führt.

TORA lässt sich in drei Funktionalitäten unterteilen [MM04, S.323]:
Das **Route Establishing** wird von einem Knoten angestoßen, wenn lokal noch keine Information über das Ziel bekannt ist. Ein *Query*-Paket mit der Zieladresse wird an alle Nachbarknoten verschickt und von diesen weitergeleitet, bis es das Ziel oder einen Knoten, der das Ziel kennt, erreicht. Daraufhin wird eine Antwort, das *Update*-Paket generiert, welches die Entfernung zum Zielknoten enthält - am Zielknoten ist diese 0. Jeder Knoten, der das *Update* erhält, erhöht den Entfernungszähler, trägt die Entfernung zum Ziel in seine lokale Tabelle ein, und leitet das Paket weiter.
Auf diese Weise wird ein gerichteter, zyklenfreier Graph erstellt, der die möglichen Routen zum Ziel beschreibt. Die Kanten des Graphs verlaufen in Richtung der kleineren Höhe.

Wenn ein Knoten feststellt, dass eine Route ungültig geworden ist, erfolgt ein *Link Reversal*. Dabei wird die Höhe des Knotens, bei dem der Verbindungsfehler aufgetreten ist, auf das lokale Maximum der Nachbarn gesetzt, so dass alle Pfade von ihm weg weisen. Auch die Nachbarn passen ihre Höhen an, sodass alternative Pfade benutzt werden. Wenn bei diesem **Route Maintaining** eine Partitionierung des Graphen festgestellt wird, sendet der erkennende Knoten eine *Clear*-Nachricht, die alle Pfadinformationen bezüglich des Ziels in der Partition löscht. Dieser Vorgang wird als **Route Erasing** bezeichnet.

Vor- und Nachteile Hauptvorteil von *TORA* ist, dass ein Verbindungsausfall nur lokale Auswirkungen hat, da der Algorithmus alle vorhandenen Routen berücksichtigt [MM04, S.325]. So wird viel Routing Overhead eingespart und eine schnelle Rekonfiguration ermöglicht.
Durch die kurzzeitige Partitionierung kann es bei der Vermittlung jedoch zu Schleifen kommen, des Weiteren wird durch die lokale Rekonfiguration nicht immer die optimale Route gewählt, was zu höheren Paketlaufzeiten führt.

3.3 Zusammenfassung

In diesem Kapitel wurden die Grundlagen für die Realisierung eines drahtlosen Ad-hoc-Netzes zur Kommunikation zwischen aktiven Komponenten erklärt.

Wie Abschnitt 1.1 bereits motiviert wurde, eignen sich aktive Komponenten zur Entwicklung verteilter Anwendungssoftware, weshalb in diesem Kapitel Jadex als Plattform für aktive Komponenten vorgestellt wurde.

Damit mobile Plattformen untereinander kommunizieren können, ist eine drahtlose Verbindung notwendig. Dafür wurden die Funktechnologie WLAN und Bluetooth mitsamt Details zum Verbindungsaufbau, den Bluetooth-Profilen und dem Pairing zwischen Geräten erklärt.

WLAN bietet einen eigenen Ad-hoc-Modus, der sich aber nur eingeschränkt nutzen lässt, zum einen, da er eine Konfiguration erfordert, zum anderen, da er von Android-Geräten nicht unterstützt wird.
Bluetooth ist energiesparend ausgelegt und bietet außerdem eine Reichweitenvergrößerung durch die Scatternetze an. Dabei ist die Nutzung eines Routing-Protokolls notwendig, von denen beispielhaft DSDV, DSR und TORA beschrieben wurden. Ein Bluetooth-Gerät führt ein Dienstregister, in das eigene Dienste eingetragen werden müssen, damit sie von anderen Geräten gefunden und genutzt werden können.

Auf Basis dieser Grundlagen kann eine Architektur entworfen werden, die die drahtlose Kommunikation mobiler Jadex-Plattformen und deren Komponenten untereinander ermöglicht (vgl. Kapitel 5).

Zunächst sollen jedoch verwandte Arbeiten zu diesem Thema vorgestellt werden, um thematische Überschneidungen zu erkennen und zu nutzen.

Kapitel 4

Verwandte Arbeiten

Nachdem im Kapitel 3 bereits die notwendigen Voraussetzungen für einen Entwurf geschaffen wurden, sollen im Folgenden Arbeiten untersucht werden, die sich thematisch mit dieser überschneiden. Das Thema dieser Arbeit umfasst im Wesentlichen zwei Bereiche: aktive Komponenten bzw. Agenten auf mobilen Geräten sowie die Peer-to-Peer-Kommunikation. Als verwandte Arbeiten werden im daher solche vorgestellt, die sich mindestens einem der beiden Bereiche widmen.

4.1 Agenten auf mobilen Geräten

In der Vergangenheit gab es bereits viele Ansätze, Agenten auf mobile Geräte zu bringen [CB03]. In den meisten Fällen wurden dabei bestehende Plattformen stark angepasst und dabei eingeschränkt, aber es existieren auch speziell für mobile Geräte entworfene Plattformen.

Jade-LEAP

Das *Java Agent Development Framework (Jade)* [Til11] bietet eine FIPA-konforme quelloffene Plattform für Softwareagenten. Ein Jade-Agent ist zunächst eine einfache Java-Klasse. Jade ist somit eine reine Plattform für individuell ausformulierte Agenten. Es kann jedoch eine externe Reasoning Engine wie Jadex oder *JESS* [Ern11a] eingebunden werden, um komplexere Verhaltensweisen abzubilden.

Ursprünglich wurde *LEAP*[1] [CP03] als alternative Plattform für Jade entwickelt, mit dem Ziel, es auf mobilen Geräten lauffähig zu machen. Seit Version 4.0 ist Jade-LEAP jedoch eine für Java-ME-Geräte[2] angepasste Version der Jade-Plattform und mit *Jade-LEAP-Android* [Til11] existiert auch eine Implementation für Android-basierte Mobilgeräte (siehe **Jade-LEAP-Android**).

[1] LEAP: Lightweight Extensible Agent Platform

[2] Die Java Micro Edition beinhaltet Spezifikationen für Java auf Geräten, die stark in ihren Ressourcen beschränkt sind [Ora11]. Sie findet z.B. bei Handys Einsatz, wird aber durch die Verbreitung von Smartphones immer unbedeutender.

Im Fall von *Jade-LEAP* kann lediglich ein kleiner Teil der Anwendung, das *Frontend*, auf einem Java-ME-kompatiblen Gerät laufen, während der *Jade*-Hauptcontainer auf einem entfernten Server ausgeführt wird, zu dem eine permanente Verbindung gehalten werden muss.
Dieser sogenannte *Split Execution Mode* lagert alle Plattformdienste (siehe Abschnitt 3.1.3) sowie möglichst viele Agenten auf den Hauptcontainer aus. Lediglich die Agenten, die der Interaktion mit dem Benutzer dienen, werden mobil ausgeführt.
Diese Aufteilung ist durch die stark begrenzten Ressourcen der meisten Java-ME-Geräte begründet.

Jade-LEAP-Anwendungen sind also von einer Infrastruktur in Form des Hauptcontainers abhängig und eine Ad-hoc-Kommunikation ist nicht vorgesehen.

Jade-LEAP-Android

Die Android-Version von Jade-LEAP erlaubt die Ausführung eines kompletten Jade-Hauptcontainers, sodass infrastrukturunabhängige Anwendungen möglich sind [Til11]. Die Kommunikation zwischen verschiedenen Plattformen über einen zentralen Jade-Container wird unterstützt, jedoch wird die Ad-hoc-Kommunikation zwischen mobilen Plattformen auch in Jade-LEAP-Android nicht berücksichtigt.

MicroFIPA-OS

Ein anderes Beispiel ist das seit 2001 nicht mehr weiterentwickelte *MicroFIPA-OS* [FIP11], welches für PocketPCs entwickelt wurde. Es basiert auf FIPA-OS [PBH00], das als erste Referenzimplementation der FIPA-Standards entwickelt wurde und bei deren Weiterentwicklung helfen sollte.

FIPA-OS bietet, wie im FIPA-Modell beschrieben, einen Agent Management Service, den Directory Facilitator und einen Message Transport zur Nachrichtenvermittlung inklusive Konversationsmanagement an. Eigene Agenten werden als Erweiterung der sogenannten *Agent Shell* erzeugt, die die Grundfunktionalitäten wie das Nutzen von Plattformdiensten bereitstellt. Alternativ kann die angebotene *JESS-Shell* genutzt werden, die JESS [Ern11a] als Reasoning Engine nutzt.

Die MicroFIPA-OS-Plattform führt dabei keine Restriktionen ein, sondern ist lediglich an die eingeschränkte Java ME API angepasst worden. FIPA-OS erlaubt das Einrichten einer verteilten Plattform, bei der AMS und DF Dienste (siehe Abschnitt 3.1.3) lediglich auf einem Rechner des verteilten Systems laufen und physisch getrennte Agenten miteinander kommunizieren können, eine Ad-hoc-Kommunikation ist aber auch hier nicht vorgesehen.

MicroJIAC

Die bisher genannten Plattformen sind durch Anpassungen bestehender Software an die eingeschränkte Umgebung mobiler Geräte entstanden. Bei der Entwicklung von

MicroJIAC (*Java-based Intelligent Agent Componentware*, [PT09]) gingen die Autoren den umgekehrten Weg: Ausgehend von den Anforderungen der am meisten eingeschränkten Java-ME-Konfiguration, der *Connected Limited Device Configuration (CLDC) [Ora11]*, wurde eine Plattform entworfen, die diese Einschränkungen berücksichtigt.
Dazu werden z.B. XML-Konfigurationsdateien schon während des Kompilierens ausgewertet, sodass ein ressourcenaufwendiges Parsing zur Laufzeit nicht mehr notwendig ist. In MicroJIAC ist die Laufzeitumgebung für einen oder mehrere Agenten ein *Node*, vergleichbar mit der *Plattform* im FIPA-Modell. Eine Anwendung besteht aus einem oder mehreren Nodes, die neben der Basisfunktionalitäten wie dem Erstellen und Ausführen von Agenten durch *Node Components* erweitert werden können. So können, ähnlich wie in Jadex, verschiedene Kommunikationssysteme als Node Components umgesetzt werden.

Ein Agent besteht in MicroJIAC aus *sensors*, *actuators*, *reactive behaviours* und *active behaviours*. *Sensors* legen fest, welche Daten aus der Umgebung der Agent wahrnehmen soll, während *actuators* die Umgebung beeinflussen. Ein weiteres Konzept verknüpft diese Daten mit den *behaviours*: das Kurz- und Langzeitgedächtnis des Agenten. Der Entwickler entscheidet explizit, welche Sensordaten in welchem Gedächtnis gespeichert werden, wobei das Kurzzeitgedächtnis Informationen, die nicht durch ein *behaviour* genutzt werden, verwirft, um die Speicherbelegung minimal zu halten.

Als Besonderheit gegenüber den anderen hier vorgestellten Arbeiten ist MicroJIAC echtzeitfähig. Auch MicroJIAC lässt die Ad-hoc-Kommunikation zwischen Plattformen auf mobilen Geräten unberücksichtigt.

Zusammenfassung

Die betrachteten mobilen Agentenplattformen sind alle aus bestehenden Projekten hervorgegangen. Dabei unterscheiden sich jedoch die Ansätze: Während Jade-LEAP für den Betrieb auf mobilen Geräten stark eingeschränkt wurde, lag bei der Entwicklung von MicroFIPA-OS und MicroJIAC der Fokus auf einer vollwertigen mobilen Agentenplattform. Während MicroJIAC das umfangreichste Plattformkonzept mitbringt und zudem echtzeitfähig ist, hält sich MicroFIPA-OS streng an die FIPA-Standards ohne die Konzepte zu ergänzen.

Mit Jade-LEAP-Android besteht die einzige Weiterentwicklung einer der genannten Plattform für modernere, leistungsfähigere mobile Geräte. Wie in den vorher genannten Plattformen besteht aber auch hier keine Möglichkeit zur Ad-hoc-Kommunikation.

4.2 Ad-hoc-Kommunikation zwischen mobilen Geräten

Der in mobilen Agentenplattformen bisher nicht vorhandenen Ad-hoc-Kommunikation widmen sich die im Folgenden betrachteten Arbeiten.
Beide hier vorgestellten Ansätze bauen auf einer der im Abschnitt 4.1 genannten Plattformen auf. Dabei liegt der Fokus im ersten Fall auf der Einbindung in eine FIPA-Plattform,

im zweiten Ansatz wird die Kooperation von mobilen Plattformen näher betrachtet.

Pirker, Berger & Watzke (2004)

In [PBW04] stellen Pirker, Berger und Watzke einen Ansatz zur FIPA-kompatiblen *Agent Discovery* mit *JXTA* [Sun03] vor. JXTA ist ein Peer-to-Peer-Netzwerkprotokoll welches auf TCP/IP Verbindungen aufsetzt und für dynamische Netze geeignet ist. Es bietet eine verteilte Discovery auf Basis von XML-Beschreibungen der angebotenen Dienste, nach denen im Netzwerk gesucht werden kann.

Diesen Mechanismus nutzen die Autoren in einem WLAN-Netzwerk zur verteilten Service Discovery, indem sie den DF-Dienst einer Jade-LEAP-Plattform erweitern, sodass dieser registrierte Dienste unter Verwendung von JXTA im Netzwerk bekannt gibt und Suchanfragen ebenfalls über JXTA abwickelt.
So kann eine bestehende FIPA-Plattform um Peer-to-Peer Fähigkeiten erweitert werden, während die Abstraktion des Nachrichtenversands für die Agenten erhalten bleibt. Da JXTA eine TCP/IP Verbindung benötigt, kann dieser Ansatz nicht ohne weiteres mit Bluetooth Verbindungen genutzt werden, ist also zunächst auf WLAN beschränkt.

Lawrence (2002)

Ebenfalls auf der Basis von Jade-Leap wurde in [Law02] eine Erweiterung der Plattform für den Einsatz in Ad-hoc-Netzwerken eingeführt. Dazu ist mit der *Platform Discovery* ein Mechanismus ähnlich der Jadex Awareness (vgl. Abschnitt 3.1.5) entwickelt worden, der andere Plattformen findet. Dabei wird nicht festgelegt, ob die Plattform Discovery an eine Bluetooth-Kommunikation und die zugehörige Bluetooth Service Discovery (vgl. Abschnitt 3.2.2) gebunden ist oder auf einer unabhängigen P2P-Schicht wie JXTA aufsetzt.

Des Weiteren wurden der AMS sowie der DF als notwendige Plattformbestandteile entfernt und werden nur noch optional ausgeführt, unter der Annahme, dass auf den meisten mobilen Geräten lediglich ein Agent laufen wird. Dennoch ist die Integration eines DF möglich: Wird eine entfernte Plattform gefunden, die einen DF ausführt, meldet die lokale Plattform all ihre Agenten bei dem entfernten DF an. Führen mehrere Plattformen DF-Instanzen aus, so arbeiten diese zusammen, damit alle dieselben Informationen erhalten. Diese Arbeit ist auf die Discovery und die Integration in FIPA-Plattformen fokussiert und lässt die P2P-Schicht außen vor.

4.3 Zusammenfassung

Die hier vorgestellten Arbeiten betrachten jeweils Teilaspekte der in dieser Arbeit behandelten Themen. Die vorgestellten Agentenplattformen sind, abgesehen von Jade-LEAP-Android, für die bei aktuelleren mobilen Geräten nicht mehr relevante Java Micro Edition

entwickelt worden und damit an leistungsschwächere Geräte angepasst. Sie schränken die Nutzung auf verschiedene Art und Weise gegenüber den Standardplattformen ein.

Die Funktionalität der Ad-hoc-Kommunikation, die ja im Gegensatz dazu eine Erweiterung speziell für mobile Geräte darstellt, wurde in keiner der betrachteten Plattformen behandelt.

Die betrachteten weiterführenden Arbeiten, die sich mit der Ad-hoc-Kommunikation beschäftigen, beschränken sich auf die Integration eines Discovery-Dienstes in eine Agentenplattform, beleuchten aber die Kommunikationsmöglichkeiten eines mobilen Geräts nicht näher.

Im nächsten Abschnitt wird daher eine eigene Lösung entworfen, die alle notwendigen Aspekte zur Realisierung der Ad-hoc-Kommunikation zwischen aktiven Komponenten berücksichtigt.

Kapitel 5

Entwurf

In dieser Arbeit wurden bereits mögliche Anwendungsszenarien und die Grundlagen behandelt, auf denen die Ad-hoc-Kommunikation zwischen aktiven Komponenten auf mobilen Geräten aufbauen kann. Um die Anwendungsszenarien zu realisieren, müssen neben den einzusetzenden Technologien weiterhin die Anforderungen an die Implementation geklärt werden, um anschließend daraus die Architektur abzuleiten.

Dabei muss, bevor die Kommunikationsarchitektur entworfen wird, zunächst die Jadex-Plattform in das Android-System integriert werden. Dazu werden verschiedene Möglichkeiten abgewogen (Abschnitt 5.1), um anschließend die dazu passende Ad-hoc-Kommunikationsarchitektur entwerfen zu können (Abschnitt 5.2). Im Abschluss dieses Kapitels wird die endgültige Architektur vorgestellt, die beide Teile vereint (Abschnitt 5.3).

5.1 Integration von Jadex in Android-Anwendungen

Bislang ist Jadex lediglich unter Ausführung mit der *Java Standard Edition (SE)* lauffähig [BP11b]. Das Android *Software Development Kit (SDK)* bietet zwar die Entwicklung von Android-Anwendungen in einer Java-Syntax an, allerdings sind in der Laufzeitumgebung auf einem Android-Gerät nicht alle von Java SE bekannten Bibliotheken verfügbar [Ope11b]. Um auf die Besonderheiten der virtuellen Maschine eines Android-Gerätes Rücksicht zu nehmen, müssen Teile des Quellcodes von Jadex geändert oder entfernt werden; auch genutzte Bibliotheken sind betroffen.
Nachdem der Portierungsweg skizziert wurde (Abschnitt 5.1.1), sollen verschiedene Möglichkeiten zur Integration der Jadex-Plattform in Android-Anwendungen diskutiert werden (Abschnitt 5.1.2).

5.1.1 Portierung

Um zu gewährleisten, dass *Jadex-Android* mit der schnellen Entwicklung von Jadex mithalten kann, ist eine systematische Portierung wünschenswert. Die einfache Anpassung des

Quellcodes entfällt somit, sie würde es nicht ermöglichen, schnell und automatisiert auf eine neue Jadex-Version zu wechseln.
Eine mögliche Lösung ist die Verwendung von Interfaces zur Abstraktion, zusammen mit verschiedenen Implementationen, die je nach Plattform genutzt werden. Dies ist jedoch durch die verwendete Android Platform Library, die sich in einigen Schnittstellen von der Java SE System Library unterscheidet und nicht alle Pakete aus Java SE beinhaltet sowie eigene hinzufügt, nicht möglich.

Als Lösung bietet sich ein Präprozessor an, der anhand von Anweisungen im Quellcode diesen verändert, bevor er kompiliert wird. Dies kann dann zur *Conditional Compilation* genutzt werden, also zum Ein- oder Ausschließen von Quellcode aufgrund einer festgelegten Bedingung wie z.B. der gewünschten Zielplattform [ADMTH09].
Im Gegensatz zu anderen Programmiersprachen [Ker88, S. 210] bietet Java keinen integrierten Präprozessor an, so dass man auf externe Lösungen zurückgreifen muss.
Zur Auswahl werden zwei frei verfügbare Präprozessoren verglichen: *munch-maven-plugin* [Son11] und *Prebop* [Sou11]. Das *munch-maven-plugin* ist ein Plugin für das Build-System *Maven* [Apa11]. Es erzeugt aus speziell kommentierten Quelldateien verschiedene Varianten, aus denen es die jeweils nicht benötigten Teile löscht. Dabei arbeitet es innerhalb eines Maven-Projekts, kann also nicht auf externe Dateien zugreifen.
Ein Einsatz im Jadex-Projekt hätte Änderungen an der Projektstruktur zur Folge, da der mit den Präprozessoranweisungen versehene Quellcode nicht mehr direkt kompilierbar ist: Er enthält alle eingearbeiteten Versionen gleichzeitig.

Die zweite Lösung, Prebop, arbeitet ebenfalls mit speziellen Kommentaren, die ausgewertet werden. Jedoch löscht Prebop den nicht benötigten Code nicht bei der Ausgabe, sondern kommentiert ihn aus, so dass alle Präprozessoranweisungen erhalten bleiben. Die Ausgabedatei ist sowohl für den Compiler als auch (wieder) für Prebop verwendbar. Außerdem nimmt Prebop Eingabedateien projektübergreifend entgegen, sodass es insgesamt möglich wird, den Originalcode durch Kommentare zu ergänzen, die den ursprünglichen Jadex-Build-Prozess nicht behindern, und die Konfiguration des Präprozessors in ein eigenes Projekt auszulagern.

5.1.2 Integration ins Android-System

Nach der Portierung von Jadex stellt sich die die Frage, wie es in Android-Anwendungen integriert werden soll. Aufgrund der Beschränkungen der Android-Plattformbibliothek und der eingeschränkten Ressourcen und Bildfläche der mobilen Geräte ist zunächst klar, das der typische Anwendungsfall sich von der Desktop-Nutzung unterscheidet.

In einer Java-SE-Umgebung startet die Jadex-Plattform zuerst das *Jadex Control Center (JCC)*, eine grafische Oberfläche zur Plattformverwaltung, welches dann den Zugriff auf Jadex-Anwendungen und deren Start erlaubt. Außerdem bietet es den Jadex-Anwendungen eine grafische Darstellung an, mit der Vorgänge auf der Plattform visualisiert werden und durch die der Nutzer mit der Plattform interagieren kann [BP11b].

Da sich das Android-Bedienkonzept jedoch durch den kleineren Bildschirm und der meist fehlenden herkömmlichen Eingabegeräte wie Tastatur und Maus wesentlich von einer Desktop-Umgebung unterscheidet, ist ein Äquivalent zum JCC hier nicht sinnvoll. Damit unterscheidet sich der Ablauf beim Starten einer Anwendung vom oben beschriebenen Ablauf von Jadex-Desktop-Anwendungen: Der Einstiegspunkt in eine Android-Anwendung ist immer eine sogenannte *Activity* [Ope11b]. Diese bestimmt über die Darstellung auf dem Bildschirm und kann während der Initialisierung eine Jadex-Plattform starten.

Danach muss sie die Möglichkeit haben, eigene Komponenten auf der Plattform auszuführen und mit ihnen zu kommunizieren, um eine Kopplung zwischen Anzeige und Ausführung der aktiven Komponenten auf der Jadex-Plattform zu erreichen.

Eine technische Anforderung dabei ist der möglichst gering zu haltende Ressourcenbedarf bei der Integration der Plattform, zu der im Folgenden zwei Alternativen vorgestellt und verglichen werden.

Die Jadex-Plattform als Android Service

Android bietet mit den *Services* eine Möglichkeit, gemeinsam genutzte Funktionalität auszulagern und zur Nutzung durch verschiedene Anwendungen freizugeben [Ope11b]. Dabei kann die Kommunikation mit einem Service von einer Anwendung durch das Bekanntgeben eines *Intents* initiiert werden. Intents stellen Nachrichten dar und sind als passiver Datencontainer zu verstehen [Ope11b].

Eine Android-Anwendung kann sich mit einem *IntentFilter* für den Empfang von Intents eines bestimmten Typs anmelden. Mit bestimmten Intents kann das Android-System selbst angesprochen werden. Auf diesem Weg kann eine Anwendung einen Service starten und sich mit ihm verbinden. Läuft der angeforderte Service bereits, wird ein Zugriffsobjekt zurückgegeben, das die vorab in der *Android Interface Definition Language (AIDL)* definierten Service-Schnittstellen bereitstellt. Über dieses Objekt kann die Anwendung mit dem Service kommunizieren, indem entfernte Methoden aufgerufen und deren Ergebnisse zurückgegeben werden (*Remote Procedure Call (RPC)*).
Läuft der angesprochene Service noch nicht, wird er durch das Android-System gestartet. Auf diese Weise können mehrere Anwendungen mit einem Service kommunizieren, der die Jadex-Plattform beinhalten könnte.

Anwendungen, die die Jadex-Plattform nutzen wollen, müssten dann der Plattform ihre eigenen aktiven Komponenten zur Verfügung stellen. Das kann über einen gemeinsamen Zugriff auf eine externe Datei geschehen, deren Pfad der Plattform mitgeteilt wird. Die Plattform kann die externe Komponente starten, die dann in einem von der Android Activity unabhängigen Prozess abläuft. Um eine Kommunikation zwischen Activity und Komponente zu ermöglichen, muss also auf Interprozesskommunikation zurückgegriffen werden, entweder in Form der Intents, oder über die AIDL-Schnittstelle zum Plattform-Service. Die Kommunikation zwischen Activity, Plattform und Komponente für letzteren

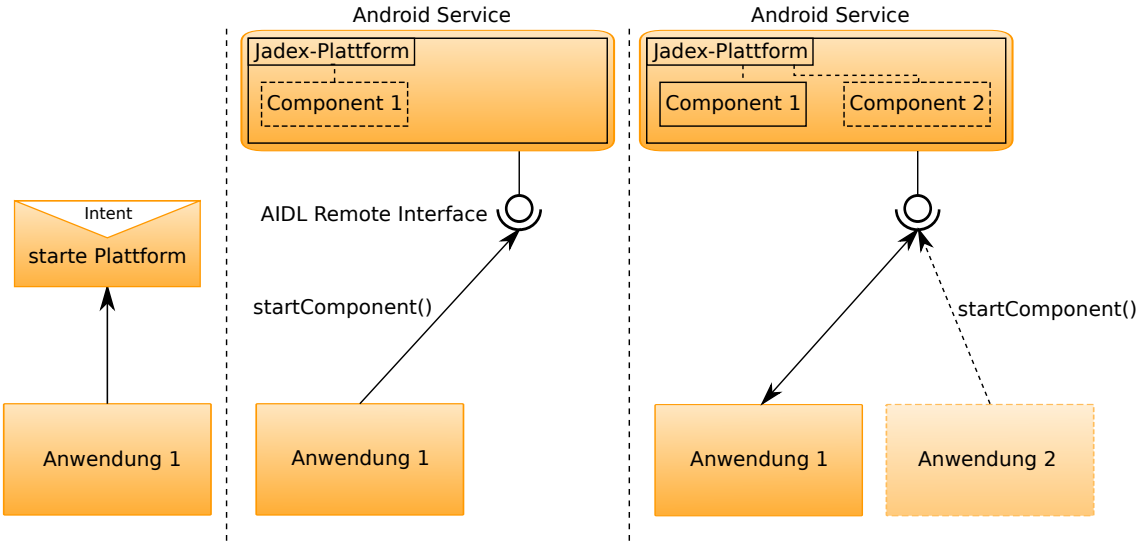

Abbildung 5.1: Zugriff auf Jadex-Plattform als Android Service

Fall ist in Abbildung 5.1 dargestellt, in der jede Verbindung zwischen zwei ausgefüllten Rechtecken als Interprozesskommunikation zu interpretieren ist.

Der Vorteil der Implementierung der Plattform als Service ist die Nutzung einer gemeinsamen Jadex-Plattform für alle Anwendungen, die aktive Komponenten nutzen. Damit geht ein geringer Speicherbedarf einher, da die Plattform nur einmal geladen werden muss. Da die Plattform dann unabhängig von den nutzenden Anwendungen ist, kann sie separat gepflegt und aktualisiert werden.
Zwar wäre es möglich, die Plattform als Service in jeder Anwendung mitzuliefern, jedoch schwinden dann die Vorteile des geringen Speicherbedarfs und der separaten Pflege.

Daraus ergibt sich direkt der größte Nachteil: Die Plattform muss unabhängig von den Anwendungen auf das mobile Gerät gebracht werden. Anwendungen müssen zudem sicherstellen, das die von ihnen benötigte Version vorliegt; Versionskonflikte, etwa wenn die Schnittstellen sich ändern oder Features verworfen werden, sind vorprogrammiert.
Dazu kommt die Notwendigkeit einer flexiblen Schnittstelle zwischen Activity, Service und Komponente, die alle Varianten der Interaktion abbildet und eines Containerformats für Agenten, die die Anwendung dem Service als Datei zur Verfügung stellen muss. Durch den höheren Kommunikations-Overhead aufgrund der vielfachen Interprozesskommunikation wird der Leistungsgewinn durch den geringeren Speicherbedarf dieser Lösung in Frage gestellt.

Jadex als Laufzeitbibliothek

Die zweite Variante bindet die Jadex-Plattform als Bibliothek in die zu entwickelnde Anwendung ein. Dadurch können Jadex-Klassen sowie die Plattform genau wie bei einer herkömmlichen Desktop-Anwendung, die die Jadex-Plattform nutzt, direkt im Quellcode referenziert und genutzt werden. Die *Jadex-Android*-Bibliothek kann dabei in die

fertige Android-Anwendung eingebaut werden oder zur Laufzeit von einem gemeinsam zugänglichen Speicherplatz auf dem Gerät geladen werden.

Beim Laden zur Laufzeit entsteht weiterer Aufwand: Vor dem Starten der eigentlichen Anwendung muss geprüft werden, ob die *Jadex-Android*-Bibliothek bereits vorhanden ist, wenn nicht, muss sie nachgeladen werden. Weiterhin bildet das dynamische Nachladen ein Sicherheitsrisiko, da die Bibliothek durch ein schadhaftes Exemplar ersetzt werden könnte.

In beiden Fällen wird der Laufzeitspeicherbedarf der Anwendung um den Speicherbedarf der Plattform vergrößert, was bei mehreren parallel laufenden Anwendungen mit jeweils einer eigenen Plattform Auswirkungen haben kann. Für den Fall, dass die Bibliothek in die Android-Anwendung mit eingebaut wird, verhält sich die Kommunikation zwischen Activity, Plattform und Komponenten wie in Abbildung 5.2 und kommt gänzlich ohne Interprozesskommunikation aus.

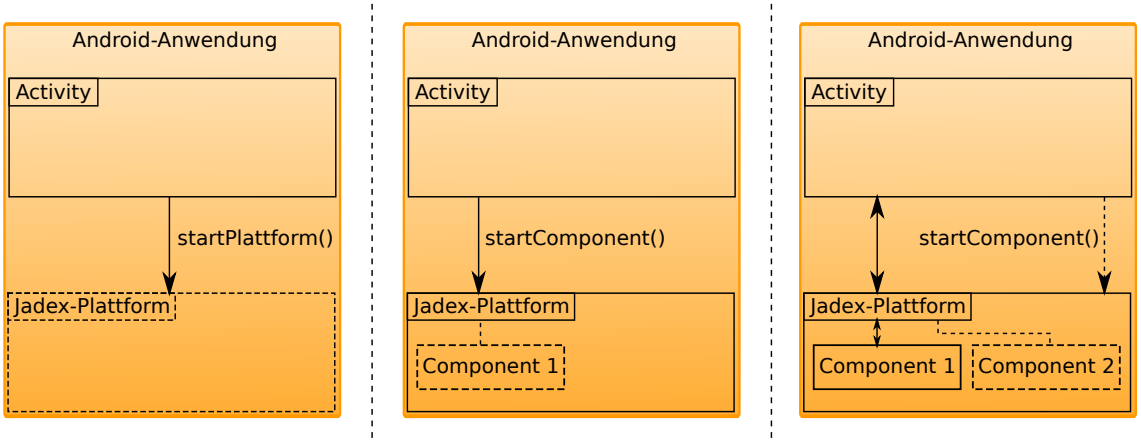

Abbildung 5.2: Die Jadex Plattform als Bibliothek integriert in Android-Anwendungen

Aus dieser Variante ergeben sich folgende Vorteile: Die Schnittstelle zwischen Activity und Plattform ist die übliche Jadex-Plattform-Schnittstelle und bietet somit die volle Flexibilität, die auch mit Jadex-Desktop-Anwendungen möglich ist. Da die Anwendung die Jadex-Plattform sowie benötigte aktive Komponenten mitliefert und integriert, kann sie wie andere Android-Anwendungen einzeln ausgeliefert werden. Damit werden auch Versionsprobleme umgangen: Gibt es mehrere Anwendungen, die auf verschiedene Versionen der Jadex-Plattform basieren, stellt dies keinen Konflikt dar.

Gleichzeitig steigt aber bei mehreren parallel verwendeten *Jadex-Android*-Anwendungen der Speicherverbrauch um das Vielfache des Jadex-Plattform-Speicherbedarfs an.

Da bei dieser Lösung die Vorteile überwiegen, wird Jadex in dieser Arbeit als Laufzeitbibliothek eingebunden werden.

Nach diesem Entwurfsschritt existiert eine Einbettung von Jadex in Android-Anwendungen, die allerdings noch keine Ad-hoc-Kommunikation mit anderen Instanzen zulässt. Dazu wird eine passende Kommunikationsinfrastruktur benötigt, die im folgenden Abschnitt diskutiert wird.

5.2 Ad-hoc-Kommunikation mit Android

Um den Ansprüchen der in Kapitel 2 vorgestellten Anwendungsszenarien gerecht zu werden, muss die drahtlose Kommunikation zwischen den aktiven Komponenten verschiedener Jadex-Plattformen nachfolgend genannte Kriterien erfüllen.

Reichweite. Die Reichweite der Funkübertragung muss ausreichen, um mit Geräten in unmittelbarer Umgebung (ca. 10m) kommunizieren zu können. Das entspricht einem Seminarraum bzw. dem Spielfeld bei einem RoboCup-Spiel. Wünschenswert ist eine größere Reichweite, um größere Räume abzudecken.

Die Mindestreichweite ist durch die beiden in Abschnitt 3.2 gezeigten Technologien gegeben, wobei WLAN generell eine höhere Reichweite aufweist. Die Bluetooth-Reichweite kann durch Vermaschung gesteigert werden.

Konfiguration. Der Aufbau des Netzwerks muss für den Anwender und möglichst auch für den Entwickler konfigurationslos sein, um die spontane Vernetzung zu ermöglichen.

Kommunikation. Jede Komponente auf einer im Netzwerk teilnehmenden Plattform muss in der Lage sein, mit jeder Komponente auf jeder anderen teilnehmenden Plattform Nachrichten auszutauschen.

Diese Peer-to-Peer-Kommunikation ist in Ad-hoc-WLANs bereits gegeben und muss in Bluetooth-Netzwerken anhand eines Routing-Protokolls zur Nachrichtenweiterleitung integriert werden.

Dynamik. Teilnehmer können aufgrund der Mobilität jederzeit dem Netzwerk beitreten oder es verlassen, worauf das Netzwerk entsprechend reagieren muss.

Ein entscheidendes Kriterium ist weiterhin der Ad-hoc-Betrieb an sich, der auf eine Infrastruktur verzichtet. Diese Anforderung ist mit WLAN nur durch den Ad-hoc-Modus zu erfüllen, der jedoch auf Android-Geräten nicht unterstützt wird. Des Weiteren geht Bluetooth sparsamer mit den Energieressourcen der mobilen Geräte um, sodass die Entscheidung in diesem Entwurf auf die Nutzung von Bluetooth fällt.
Da Bluetooth allerdings nicht per se alle genannten Anforderungen erfüllt, muss eine Bluetooth-Kommunikationsarchitektur entworfen werden.

5.2.1 Bluetooth-Kommunikationsarchitektur

Die *Android Bluetooth API*[1] [Ope11a] sieht den Betrieb eines Bluetooth Servers sowie eines Clients vor und abstrahiert dabei von den Master/Slave-Rollen des Bluetooth-Protokolls. Damit eine Verbindung von einem beliebigen Gerät aufgebaut werden kann, muss die

[1] API: Application Programming Interface, bietet Anwendungsentwicklern Zugang zu Programm- oder Geräteschnittstellen

Architektur auf jedem Gerät sowohl einen Server als auch einen Client einsetzen. Das Starten eines Servers beinhaltet die Eintragung in das SDP-Register des Bluetooth-Geräts (siehe Abschnitt 3.2.2), wodurch der Server von Clients per SDP gefunden werden kann.

Zum Verbindungsaufbau muss dabei lediglich die UUID, die beim Eintragen in das SDP-Register genutzt wurde, bekannt sein. Diese kann bereits durch den Entwickler festgelegt werden, so dass eine Verbindung zu anderen Geräten weitgehend konfigurationslos aufgebaut werden kann, was in diesem Entwurf angestrebt wird.

Für die Verbindung wird das RFCOMM-Profil (vgl. Abschnitt 3.2.2) genutzt, da die Erstellung vollständiger, eigener Profile von der Android API nicht unterstützt wird. Android verlangt vor dem Verbindungsaufbau per RFCOMM mit einem anderen Gerät ein vorheriges Bluetooth Pairing (vgl. Abschnitt 3.2.2). Dies steht im Gegensatz zur geforderten Konfigurationslosigkeit (vgl. Abschnitt 5.2), ist allerdings lediglich einmalig pro Gerät notwendig [Ope11a] und erübrigt eine eigene Lösung zur Authentifizierung von Teilnehmern.

Eine Vermaschung von verschiedenen Piconetzen dient der Reichweitenvergrößerung, ist aber ohnehin notwendig, um durch den Aufbau eines Scatternetzes die Beschränkung von acht Geräten im Netzwerk aufzuheben (vgl. Abschnitt 3.2.2). Die Vermaschung bzw. der Aufbau eines Scatternetzes entsteht durch die gleichzeitige Verbindung eines Bluetooth Clients mit mehreren Bluetooth Servern. Dieser mit mehreren Servern verbundene Client, der somit an mehreren Piconetzen gleichzeitig teilnimmt, vermittelt dann zwischen ihnen.

Beim Scatternetzaufbau müssen weitere Anforderungen beachtet werden [WSL04]:

Piconetzanzahl Die Zahl der Piconetze sollte so klein wie möglich gehalten werden, um schnelleres Routing zu ermöglichen. Dazu sollte sich ein Gerät jeweils nur mit anderen Geräten verbinden, wenn noch keine Verbindung zwischen den beiden über einen dritten Knoten im Netzwerk besteht. Durch diese Maßnahme wird außerdem der Verbindungsaufwand reduziert, allerdings leidet die Ausfallsicherheit, sodass eine spätere Erweiterung auf mehrere redundante Verbindungen denkbar ist.

Verwaltungs-Overhead Der Aufbau des Scatternetzes soll möglichst wenig Overhead generieren. Da Bluetooth mit dem RFCOMM-Kanal eine zuverlässige Datenübertragung garantiert [Rot05, S.133], ist beim Verbindungsaufbau kein zusätzlicher Aufwand nötig, um dies zu erreichen.

Routing Zur Vermittlung von Nachrichten zwischen verschiedenen Piconetzen ist ein Routing-Protokoll notwendig, da der Bluetooth-Standard allein keine solche Nachrichtenvermittlung vorsieht (vgl. Abschnitt 3.2.2).

Welches der in Abschnitt 6.2.4 vorgestellten Protokolle für den Einsatz in dieser Arbeit geeignet ist, wird im folgenden Abschnitt erörtert.

5.2.2 Routing-Protokoll

Aus Sicht der Anwendung muss ein Routing-Protokoll nur Eines leisten, nämlich jede Nachricht jederzeit an den richtigen Empfänger zuzustellen.
Dazu sind die im Abschnitt 3.2.3 genannten Eigenschaften mobiler Ad-hoc-Netze zu berücksichtigen. Zwei dieser Eigenschaften können durch Simulationen untersucht werden: die Eignung für beschränkte Bandbreiten, gemessen durch den *Overhead*, also der zusätzlich zu den Nutzdaten übertragenen Datenmenge, den ein Protokoll einführt, und die Tauglichkeit in mobilen Netzen durch die Messung der Paketverluste.
Der Ressourcenbedarf lässt sich anhand der verwendeten Algorithmen und Techniken abschätzen.

In [BMJ+98] wurden die im Abschnitt 3.2.3 vorgestellten Protokolle in verschiedenen Szenarien simuliert und der Routing Overhead sowie die Paketverluste gemessen. Zusätzlich wurde eine Abwandlung von DSDV eingesetzt, in der auf einige Updates verzichtet wird.

Die 900-sekündige Simulation besteht aus 50 Knoten in einem 1500m $\times$ 300m großen Raum, die Funkreichweite jedes Knotens beträgt 250m. Dabei wurden Maximalgeschwindigkeiten von 1m/s bzw. 20m/s sowie Haltezeiten zwischen den Bewegungen der Knoten von 0, 30, 60, 120, 300, 600 und 900 Sekunden angenommen. Die Anzahl der sendenden Knoten beträgt 10, 20 oder 30 und die Senderate 4 Pakete pro Sekunde.
Zu Beginn der Simulation verharren alle Knoten für die Länge der Haltezeit am zufälligen Startort, bewegen sich dann zu einem zufällig gewählten Ort mit einer zufälligen Geschwindigkeit zwischen 0 und der Maximalgeschwindigkeit, und wiederholen dies, bis die Simulation beendet ist.

Für diese Arbeit sind, bezogen auf die in Kapitel 2 dargestellten Anwendungsszenarien, die Messungen mit 1m/s Maximalgeschwindigkeit, die nur mit 20 sendenden Knoten durchgeführt wurden, besonders relevant. In diesen Messungen liegt der Paketverlust aller Protokolle unter 0,015%, sodass der Routing Overhead das entscheidende Kriterium wird.
Tabelle 5.1 zeigt die Zahlenwerte für eine Haltezeit von 100s. Dabei ist zusätzlich noch eine Variante von DSDV gemessen worden, hier DSDV* genannt, die Updates nur sendet, wenn sich neben der Sequenznummer auch die Distanz im Eintrag ändert. Dadurch werden Pakete auf Kosten einer schnellen Ausfallerkennung eingespart [BMJ+98, S.11].

TORA setzt in der gemessenen Implementation von Boch et al. auf dem *Internet MANET Encapsulation Protocol (IMEP)* auf, welches die von TORA vorausgesetzte zuverlässige und geordnete Übertragung in mobilen Ad-hoc-Netzen (*MANETs*) garantiert. IMEP generiert jedoch durch das periodische Senden von Präsenznachrichten viel Overhead, der Autor der Simulation beziffert den Minimal-Overhead auf 45.000 Pakete in der Simulationszeit, was die hohe Anzahl der Routing-Pakete mit *TORA* erklärt.

Im Vergleich schneidet *DSR* am besten ab. Es ist für dynamische Netze sehr gut geeignet, da es keine Routeninformationen im Voraus austauscht, sondern nur bei Bedarf, und durch

Messung	DSDV	DSR	TORA	DSDV*
Verlorene Pakete	0,005%	0%	0,005%	0,05%
Routing-Pakete	41.000	2.000	58.000	10.000

Tabelle 5.1: Vergleich von Routing-Protokollen, Daten aus [BMJ+98]

das Caching trotzdem den Routing Overhead in Grenzen hält.

DSDV bietet den Vorteil, dass jeder Knoten durch die lokalen Routing-Tabellen jeden anderen Netzwerkteilnehmer kennt. Das ist für diese Arbeit von Bedeutung: Zwar können alle Netzwerkteilnehmer ihre Adressen bei Bedarf austauschen, für das konfigurationslose Zusammenspiel aller beteiligten Geräte im Sinne der Anwendungsszenarien ist eine Kenntnis aller verfügbaren Teilnehmer aber essenziell. Andernfalls wäre es nicht möglich, automatisiert verfügbare Plattformen nach angebotenen Diensten abzufragen, da überhaupt nicht bekannt wäre, welche Plattformen im Netzwerk existieren.

Da diese Funktionalität in anderen Protokollen durch Präsenznachrichten implementiert werden müsste, wodurch sich sowohl der Overhead als auch die Komplexität erhöht, und die bereits genannten Anwendungsszenarien (vgl. Kapitel 2) keine hohe Mobilität bedingen, wird in dieser Arbeit DSDV zum Einsatz kommen.

5.3 Integration und Ergebnis

Damit die Jadex-Plattform über die bisher entworfene Kommunikationsarchitektur Nachrichten mit anderen Plattformen austauschen kann, muss schließlich noch die Schnittstelle zu den Jadex-Plattformdiensten (siehe Abschnitt 3.1.3) geschaffen werden.
Dabei sind zwei Dienste anzupassen: Der Message Service, dem ein neuer Bluetooth Transport (vgl. Abschnitt 3.1.4) angeboten werden muss, sowie die Awareness, die durch das Einsetzen eines neuen Discovery-Agenten an die Bluetooth-Kommunikation angepasst wird.

Der zu entwickelnde Bluetooth Transport nimmt die Adressierung mittels der Bluetooth-Adressen vor, die für jedes Gerät einmalig sind. Die vom Message Service übermittelten Daten werden unverändert an den Empfänger zugestellt, so dass die Bluetooth-Kommunikation für den Message Service transparent ist.

Der *BluetoothDiscoveryAgent* sendet Awareness-Informationen über die Kommunikationsschicht an alle verfügbaren Geräte im Netzwerk und erhält seinerseits Informationen über andere aktive Plattformen. Diese übergibt er, wie die bereits vorhandenen Discovery-Agenten auch, dem *AwarenessManagementAgent*, so dass andere Jadex-Komponenten auf die verfügbaren Plattformen zugreifen können.

Um Konflikte durch mehrere auf einem Gerät laufende Jadex-Plattformen zu vermeiden, wird die Bluetooth-Kommunikationsarchitektur als Android Service über eine AIDL-

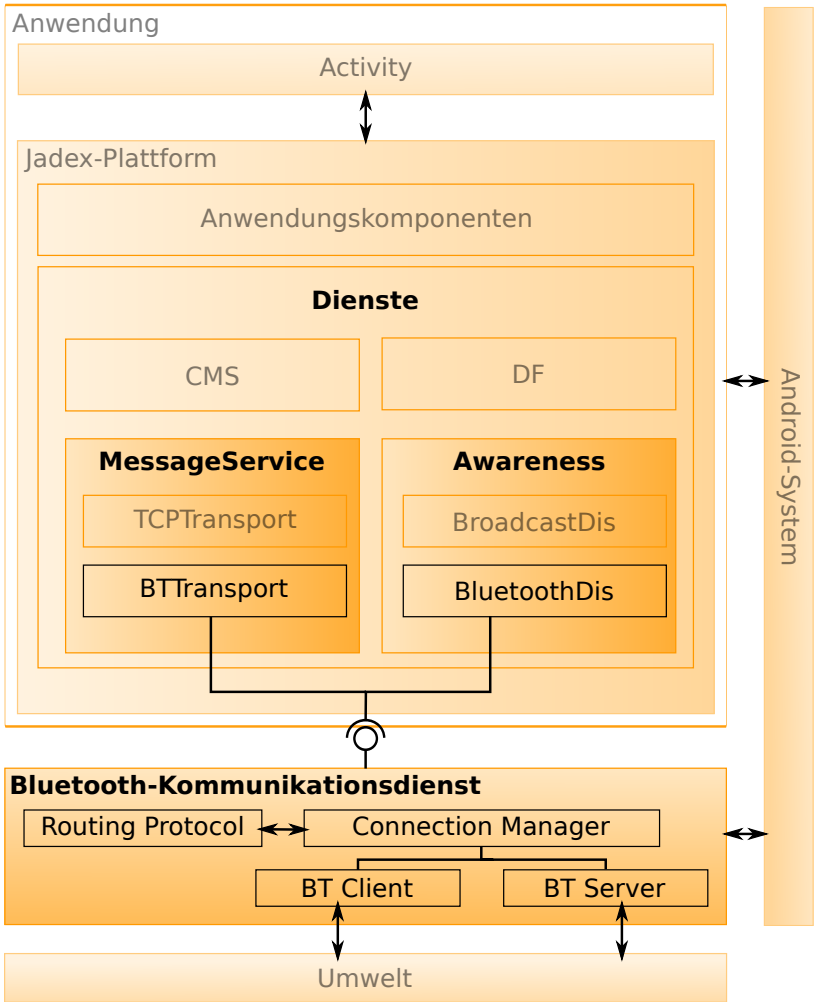

Abbildung 5.3: Jadex-Android mit Bluetooth Kommunikation: Architektur

Schnittstelle von der Plattform angesprochen. So nimmt ein Gerät immer nur einmal an einem Ad-hoc-Netzwerk teil, egal wieviele Jadex-Plattformen auf dem Gerät gestartet werden.

Durch diese Art der Integration in die Jadex-Plattform ist es ohne weiteres möglich, *Jadex-Android* auch ohne die Bluetooth-Kommunikation einzusetzen, indem einfach der Kommunikationsdienst mitsamt der angepassten Jadex-Komponenten aus der Anwendung und der Plattformkonfiguration entfernt wird.

Das Ergebnis des Entwurfs ist in Abbildung 5.3 dargestellt, die bereits die vollständige Architektur inklusive der Integration der Ad-hoc-Kommunikationsschicht in Jadex zeigt.

5.4 Ablauf der Kommunikation

Um die Funktionsweise der entworfenen Bluetooth-Kommunikation im Zusammenspiel mit der Jadex-Plattform zu veranschaulichen, wird an dieser Stelle noch einmal der gesamte geplante Ablauf zusammengefasst.

Beim Starten der Android Activity wird auch die Jadex-Plattform gestartet. In Ihrer Konfiguration ist der BluetoothDiscoveryAgent als Subkomponente des AwarenessManagementAgents eingetragen und wird mit ihm gestartet. Der BluetoothDiscoveryAgent startet den Bluetooth-Kommunikationsdienst, auf den mehrere Anwendungen zugreifen können (vgl. Abschnitt 5.3).

Dieser versucht, sich mit einem bestehenden P2P-Netzwerk zu verbinden, indem er alle bekannten bzw. gekoppelten Geräte (vgl. Abschnitt 3.2.2) auf Erreichbarkeit prüft und sich bei Erfolg mit diesen verbindet. Dabei wird er über das Routing-Protokoll (vgl. Abschnitt 6.2.4) von ebenfalls im Netzwerk erreichbaren Knoten in Kenntnis gesetzt und meldet diese per Interprozesskommunikation an den BluetoothDiscoveryAgent.

Im nächsten Schritt werden, wie in der Desktop-Version von Jadex üblich [BP11b], alle verfügbaren Plattformen nach Komponenten und angebotenen Diensten durchsucht, damit diese von der lokalen Plattform aus erreichbar sind.

Der gerade beschriebene Bluetooth-Discovery-Vorgang wird periodisch durchgeführt, um neue Geräte zu finden.

Die Nachrichtenkommunikation zwischen entfernten Komponenten läuft dabei genauso ab, wie auf Jadex-Desktop-Plattformen: Der Message Service nutzt den zur Zieladresse passenden Transport, der die Nachricht übermittelt.
Der in diesem Fall verwendete Bluetooth Transport nutzt, wie der Discovery Agent, den Bluetooth-Kommunikationsdienst, um die Nachricht über das P2P-Netzwerk zu übertragen. Hier kommt dann der gewählte Routing-Mechanismus ins Spiel, der den nächsten Knoten auf dem Pfad zum Ziel ermittelt und die Nachricht dorthin verschickt.
Hat die Nachricht ihr Ziel erreicht, nimmt der Bluetooth Transport auf Empfängerseite die Nachricht entgegen und leitet sie an den Message Service weiter, der sie der Zielkomponente zustellt.

5.5 Zusammenfassung

In diesem Kapitel wurde unter Einbeziehung der Anwendungsszenarien (Kapitel 2) und der bereits erarbeiteten Grundlagen (Kapitel 3) eine Ad-hoc-Kommunikationsarchitektur für Jadex auf Android-Geräten entworfen.

Für die zunächst notwendige Portierung konnte mit der Nutzung eines Präprozessors ein geeigneter, systematischer Weg gefunden werden, der eine Abspaltung von *Jadex-Android* vom Hauptentwicklungszweig vermeidet. Um die portierte Jadex-Plattform in das Android-System zu integrieren, fiel die Entscheidung zugunsten der Einbindung von Jadex als Laufzeitbibliothek, da die Vorteile hinsichtlich Auslieferung, Wartung und Flexibilität bei dieser Lösung überwiegen.

Beim Entwurf der Ad-hoc-Kommunikation wurden aus den Anwendungsszenarien Anforderungen an die Architektur abgeleitet, die dann – soweit es die Android API erlaubte – umgesetzt wurden. Entstanden ist eine Architektur, die durch die Nutzung eines Ad-hoc-

Routing-Protokolls und Scatternetzbildung die spontane Vernetzung von mobilen Geräten erlaubt. Ein Gerät verbindet sich lediglich mit bereits gekoppelten Geräten in Reichweite, was den Sicherheitsanforderungen der Android API bezüglich Bluetooth-Kommunikation geschuldet ist und einerseits keinen vollständig konfigurationslosen Netzwerkaufbau ermöglicht, andererseits aber möglichen Sicherheitsrisiken beim Verbinden mit unbekannten Geräten vorbeugt.

Die Einzelkomponenten des Entwurfs – zum einen die *Jadex-Android*-Plattform, zum anderen die Ad-hoc-Kommunikation – wurden im Abschnitt 5.3 zusammengeführt, indem an zwei Stellen in die Jadex-Plattform eingegriffen wurde: Der *MessageService* bekommt einen neuen *BluetoothTransport* zum Nachrichtentransport über das Bluetooth-P2P-Netz und der *AwarenessManagementAgent* bedient sich eines *BluetoothDiscoveryAgents*, um die Jadex Awareness (vgl. Abschnitt 3.1.5) auch im Bluetooth-P2P-Netz einsetzen zu können.

Zum Abschluss des Kapitels wurde noch einmal der Ablauf einer *Jadex-Android*-Anwendung dargestellt, die den entwickelten Kommunikationsdienst nutzt.

Kapitel 6

Implementierung

Nachdem im vorherigen Kapitel das Konzept für die Implementierung erarbeitet wurde, folgt nun die Darstellung der prototypischen Implementierung selbst. In diesem praktischen Teil der Arbeit wurde weitestgehend nach dem Entwurf vorgegangen; Abweichungen oder Erweiterungen davon werden ebenfalls erläutert.

Die Implementierung bestand zunächst aus der Portierung von Jadex auf Android (Abschnitt 6.1) und der Integration ins Android-System. Anschließend musste die in Abschnitt 5.2 entworfene Bluetooth-Kommunikationsarchitektur auf dem Android-System implementiert werden (Abschnitt 6.2), wobei auf Nutzung außerhalb der Android-Umgebung Rücksicht genommen wurde. Abschließend wird die Integration der Kommunikationsarchitektur in Jadex anhand von Nutzungsbeispielen gezeigt (Abschnitt 6.4) und die Leistungsfähigkeit des Frameworks evaluiert (Abschnitt 6.5).

6.1 Portierung

Zur systematischen Portierung wurde in dieser Arbeit aufgrund der in Abschnitt 5.1.1 beschriebenen Vorteile *Prebop* [Sou11] als Präprozessor genutzt. Die Anpassung an Android wurde so nicht in einen separaten Entwicklungszweig verlagert, sondern fand im Hauptentwicklungszweig von Jadex statt, sodass die zukünftige Entwicklung parallel verläuft.

Ein Beispiel für die Syntax der von Prebop ausgewerteten Kommentare ist in den Listings 6.1 und 6.2 zu sehen. Deutlich wird dabei auch die Reversibilität des Preprocessings: Mit dem Aufruf von Prebop und dem Setzen der Prebop-Variable *android* auf **true** bzw. **false** kann der Quelltext zwischen den beiden in der Abbildung gezeigten Varianten hin- und hertransformiert werden.
Dabei werden lediglich die Kommentarzeichen verändert, sodass jeweils die richtige Variante aktiv ist. Durch das Syntax-Highlighting gebräuchlicher Quelltexteditoren ist auf einen Blick erkennbar, welche Variante aktiv ist.

Alle von Jadex genutzten externen Abhängigkeiten waren entweder bereits in einer Android-Version oder im Quelltext verfügbar, sodass diese ebenfalls an die Android API

Listing 6.1: Quellcodeausschnitt mit Prebop-Kommentaren: Java SE

```
/* $if !android $ */
      Class clz = sun.reflect.Reflection.getCallerClass(ix);
      /* $else $
      Class clz = this.getClass();
      $endif $ */
```

Listing 6.2: Quellcodeausschnitt mit Prebop-Kommentaren: Android

```
/* $if !android $
      Class clz = sun.reflect.Reflection.getCallerClass(ix);
      $else $ */
      Class clz = this.getClass();
      /* $endif $ */
```

angepasst werden konnten.

Auf diese Weise wurden die Jadex-Plattform sowie die verschiedenen Jadex-Kernels (vgl. Abschnitt 3.1.1) portiert. Im nächsten Abschnitt wird der zweite Schritt der Implementation, die Entwicklung der Bluetooth-Kommunikation, beschrieben.

6.2 Bluetooth-Kommunikation

Die in Abschnitt 5.2 skizzierte Architektur wurde zunächst ohne Anbindung an *Jadex-Android* entwickelt. Dabei wurde eine einfache Beispielanwendung implementiert, die das Kommunikationsframework als Android Service anspricht, der später in Jadex-Android integriert werden konnte.

Das Kommunikationsframework besteht aus einer Basisklasse, dem `BTP2PConnector`, der den Bluetooth Chip des Gerätes aktiviert und die Verbindungen initialisiert. Dabei startet er zunächst, wie im Entwurf beschrieben, einen *BluetoothServer*, der mit der Android-API-Funktion `BluetoothDevice.listenUsingRfcommWithServiceRecord()` eine RFCOMM-Schnittstelle öffnet (vgl. Abschnitt 3.2.2). Dabei verwendet er eine vorgegebene UUID, um den Dienst ins SDP-Register einzutragen.

Da jeder `BluetoothServerSocket` jedoch nur eine Verbindung annehmen kann, müssen mehrere davon mit jeweils unterschiedlichen UUIDs geöffnet werden. Die UUIDs wurden einmalig generiert und sind nun im Programmcode als Konstanten enthalten. Weil die Bluetooth API von Android keinen Einfluss auf die Master/Slave-Rollenverteilung zulässt [Ope11a], ist an dieser Stelle unklar, wie viele Bluetooth Server Sockets geöffnet werden sollten. Das Piconetzlimit von acht Geräten muss nicht angewandt werden, da ein Slave ja mit mehreren Piconetzen verbunden sein kann (vgl. Abschnitt 3.2.2). In der Implementation zu dieser Arbeit wurde dennoch ein Limit von sieben eingehenden Verbindungen gewählt. Durch spätere Evaluation könnten die Auswirkung dieses Limits auf die Netzwerkleistung geprüft werden, mangels ausreichender Testgeräteanzahl war

dies jedoch im Verlauf dieser Arbeit noch nicht möglich.

6.2.1 Verbindungsaufbau

Nachdem der `BluetoothServer` das Android-System für eingehende Verbindungen vorbereitet hat, beginnt der `BTP2PConnector` mit der Verbindungsaufnahme zum P2P-Netz. Dazu wird periodisch die Liste mit den Pairing-Einträgen des eigenen Geräts abgerufen, da ein Pairing vor dem Verbindungsaufbau notwendig ist (vgl. Abschnitt 5.2.1), und zu denjenigen Geräten, die noch nicht über das Netzwerk erreichbar sind, eine Verbindung aufgebaut.

Leider existiert keine öffentlich zugängliche API-Funktion, die es erlaubt, die Erreichbarkeit oder das SDP-Register eines entfernten Gerätes abzufragen. Daher muss der Verbindungsaufbau, wie in Listing 6.3 dargestellt, „blind“ unter Nutzung einer UUID erfolgen. Ist unter der UUID kein Dienst erreichbar (`IOException`), kann dies entweder bedeuten, dass das entfernte Gerät nicht in Reichweite ist bzw. keine Instanz des Bluetooth-Kommunikationsframeworks läuft, oder, dass der RFCOMM-Kanal mit der versuchten UUID bereits von einer anderen Verbindung belegt ist. In letzterem Fall muss die nächste UUID probiert werden (`uuidNum++`).

Listing 6.3: Verbindungsaufbau

```
while (true) {
        try {
          socket = device.createRfcommSocketToServiceRecord(BTServer.↩
              UUIDS[uuidNum]);
          socket.connect();
          manageConnectedSocket(socket);
          break;
        } catch (IOException e) {
          uuidNum++;
          if (uuidNum == BTServer.UUIDS.length) {
            // connection could not be established
            break;
          }
        }
}
```

Sind alle UUIDs durchprobiert, muss der Verbindungsversuch als gescheitert angesehen werden.

Da der Aufruf zum Verbindungsaufbau mit `IBluetoothSocket.connect()` eine Zeit lang dauert und um nicht für jedes Gerät sieben Verbindungsversuche durchführen zu müssen, wird eine UUID für die Erreichbarkeitsprüfung reserviert. Verbindungen, die zum Server unter dieser UUID hergestellt werden, werden sofort wieder abgebaut und dienen nur der Prüfung der Erreichbarkeit. Antwortet ein Gerät nicht auf dieser UUID, wird kein weiterer Verbindungsversuch unternommen. Durch diese Maßnahme verringert sich die Zeit zum Verbindungsaufbau mit dem Netzwerk erheblich, vor allem bei vielen

Pairing-Einträgen.

Nachdem die Verbindung von der Android-Bluetooth-Schicht aufgebaut wurde, wird ein Handshake[1] mit den Nachrichtentypen `CONNECT_SYN` bzw. `CONNECT_ACK` durchgeführt. War der Handshake erfolgreich, wird der zugehörige `BluetoothSocket` an den `ConnectionManager` übergeben, der sich um die Verwaltung der Verbindungen kümmert.

6.2.2 Verbindungsverwaltung

Der `ConnectionManager` unterscheidet nicht zwischen ein- und ausgehenden Verbindungen, sondern verwaltet lediglich eine gemeinsame Liste sowie Beobachter, die über neue und abgebrochene Verbindungen informiert werden. Weiterhin kann jeder Verbindung, ebenfalls über das Beobachtermuster, ein oder mehrere Nachrichtenempfänger zugewiesen werden. Über einen Standardbeobachter werden die protokollinternen Nachrichten (siehe Abschnitt 6.2.3) verarbeitet, sowie Nachrichten, die das Ziel noch nicht erreicht haben, dem Routing-Mechanismus zugeführt.
Während des Verbindungsaufbaus können über den Beobachtermechanismus individuelle Beobachter verwendet werden, die bei einem missglückten Verbindungsversuch z.B. das betroffene Gerät von der Liste der zu kontaktierenden Geräte entfernen.

Jede Verbindung wird durch eine Klasse mit dem Interface `IConnection` repräsentiert, die zwei Threads startet. Einer dient dem Nachrichtenempfang und setzt in mehreren Teilstücken empfangene Nachrichten wieder zusammen, der andere behandelt das Senden von Nachrichten und liest dazu Nachrichten aus einer Sendewarteschlange.

Beim Senden der Nachricht wird in der zu dieser Arbeit entwickelten prototypischen Implementierung keine Prüfung vorgenommen, ob die Nachricht angekommen ist. Es ist lediglich durch die im Nachrichtenformat enthaltene Nachrichten-ID sichergestellt, dass keine Nachricht mehrfach verwendet wird.

Das beim Senden verwendete Nachrichtenformat wird im Folgenden näher beschrieben.

6.2.3 Nachrichtenformat

Um das Senden und Empfangen von Nachrichten durch Paket-Routing (vgl. Abschnitt 5.2.2) vorzunehmen, sind neben den Nutzdaten weitere Informationen notwendig. Das Jadex-Nachrichtenformat beinhaltet zwar ebenfalls einige dieser Informationen, wichtige Felder, wie z.B. die Nutzdatenlänge, fehlen aber. Um für größtmögliche Transparenz zu sorgen und den Bluetooth-Nachrichtendienst unabhängig von der Art der übertragenen Nachrichten zu machen, wurde daher ein eigenes Nachrichtenformat implementiert.
Die Jadex-Nachrichten werden später als Nutzdaten transportiert. Die Tabelle 6.1 stellt das implementierte Nachrichtenformat `DataPacket` mit seinen Feldern dar.

[1] Ein Handshake ("Handschlag") dient der Bestätigung einer erfolgreich aufgebauten Verbindung für beide Seiten.

Typ	Feldname	Beschreibung
byte	`type`	Der Typ der Nachricht
String	`src`	Adresse des Absenders
String	`dest`	Adresse des Ziels
String	`pktId`	Paketidentifikationsnummer
byte	`hopCount`	Hop-Zähler
short	`dataSize`	Länge der Nutzdaten
byte[]	`data`	Nutzdaten

Tabelle 6.1: Felder im verwendeten Nachrichtenformat

Der Wert von `type` kann den Wert einer der folgenden Konstanten annehmen: Zur Verbindungserhaltung dienen die Pakettypen `PING` und `PONG`, zum Aufbau der Verbindung `CONNECT_SYN` und `CONNECT_ACK`, für den Austausch von Routing-Informationen der Typ `ROUTING_INFORMATION` und für die Datenübertragung schließlich der Typ `DATA`.

Für die Adressierung über Verbindungsgrenzen hinweg, also beim Routing über mehrere Knoten, dienen die Felder `src` und `dest`. Als Adressen werden dabei die Bluetooth-MAC-Adressen[2] genutzt.

Eine zufällige ID im Feld `pktId`, die für jedes Paket neu generiert wird, vermeidet, dass doppelt übertragene Nachrichten auch doppelt verwendet werden. Der `hopCount` dient der Vermeidung von Schleifen beim Routing und das Feld `data` enthält schließlich die Nutzdaten und `dataSize` deren Länge.

Im Verlauf der Implementierung stellte sich heraus, dass die übertragenen Daten durch die `InputStream.read()` Methode auf Empfängerseite bei einer Länge von 1008 Byte abgeschnitten werden und der Rest des Paketes erst bei einem erneuten Lesen aus dem Stream ausgegeben wird[3]. Das Feld `dataSize` dient also auf Empfängerseite dazu, zu erkennen, wann ein Paket komplett aus dem Stream gelesen wurde.

Als Nutzdaten können beliebige Byte Arrays versendet werden, wobei die Länge durch den Datentyp `short` des `dataSize`-Feldes auf 32KiB beschränkt ist. Diese Beschränkung wurde zunächst willkürlich aufgestellt, eine spätere Analyse der Paketgrößen zeigte aber, dass selbst eine maximal zulässige Paketgröße von 4KiB für den Jadex-Nachrichtenaustausch ausreichend ist. Werden größere Datenmengen auf einmal versendet, beispielsweise bei der Übertragung von Bildern, muss die erlaubte Maximalgröße entsprechend angepasst werden.

Während Jadex selbst Mechanismen zur Serialisierung von Objekten für den Versand enthält [BP11b], wurde für die Verwaltungsnachrichten der P2P-Kommunikation, insbe-

[2] MAC bezeichnet die Medienzugriffsschicht (Media Access Control), auf der bereits eine Adressierung vorgenommen wird. MAC-Adressen sind weltweit eindeutig [Blu10, Vol.2, S.68].

[3] Die Spezifikation der Java Klasse *InputStream* besagt lediglich, das als Rückgabewert die Anzahl der tatsächlich gelesenen Bytes zurückgegeben wird, nicht aber, dass der Stream gelesen wird, bis keine Daten mehr gesendet werden.

sondere für das Verschicken von Routing-Informationen, auf die Google Protocol Buffers [Var08] zurückgegriffen. Deren Compiler erzeugt aus einer simplen Beschreibungssprache z.B. Java-Klassen, die anschließend eine effiziente Serialisierung und Deserialisierung ermöglichen. Das Format ist dabei unabhängig von Programmiersprache und Umgebung, sodass ein Austausch mit Desktop-Anwendungen möglich bleibt.

Sobald ein `DataPaket` erstellt wurde und alle Datenfelder mit Werten gefüllt wurden, kann es dem Routing-Mechanismus zur weiteren Verarbeitung übergeben werden.

6.2.4 Routing und Nachrichtenversand

Vor der eigentlichen Datenübertragung wird das `DataPacket` zunächst einer Implementierung von `IPacketRouter` übergeben, die anhand eines Routing-Algorithmus entscheidet, an welches verbundene Gerät die Nachricht gesendet werden muss, um ihr Ziel zu erreichen.

Klassen, die das `IPacketRouter` Interface implementieren, bekommen eingehende Routing-Informationen zugestellt und können ihrerseits welche versenden. Dazu werden sie über neue verbundene Geräte und abgebrochene Verbindungen informiert. Durch diese Struktur können leicht verschiedene Routing-Verfahren (vgl. Abschnitt 6.2.4) implementiert werden.

In der Entwicklungsphase wurde dabei zunächst ein `FloodingPacketRouter` genutzt, der alle im Netz erreichbaren Geräte kennt. Ist das Ziel unter diesen Geräten vorhanden, aber nicht direkt erreichbar, wird ein Paket einfach an alle direkt verbundenen Geräte geschickt, andernfalls nur an den Empfänger.
Schließlich wurde auch das im Abschnitt 3.2.3 erläuterte DSDV-Verfahren implementiert, dessen Vorteile gegenüber dem Flooding ab drei Geräten offensichtlich sind.

Um die Routing-Mechanismen unabhängig von der Bluetooth API zu implementieren, wird dem PacketRouter unter dem Interface `IPacketSender` eine Klasseninstanz bereitgestellt, die sich um das Senden der Nachricht unter Verwendung der richtigen Verbindung kümmert. Die Methode `IPacketSender.sendMessageToConnectedDevice()` sendet Daten an ein Ziel, welches direkt verbunden ist. Der `ConnectionManager` implementiert diese Interface-Methode. Zum Senden wählt er die richtige Verbindung aus und reiht die zu sendende Nachricht in die Sendewarteschlange der Verbindung ein.

Hat eine Nachricht ihr Ziel erreicht, werden die Nutzdaten vom Routing-Verfahren an den `BTP2PConnector` weitergegeben. Dort können sich Objekte als Nachrichtenempfänger entweder für ein bestimmtes entferntes Gerät oder für alle eingehenden Nachrichten anmelden.

Schließlich können sich auch externe Anwendungen über die Android-Service-Schnittstelle als Nachrichtenempfänger anmelden, was für das Zusammenspiel mit der Jadex-Plattform notwendig ist (vgl. Abschnitt 6.3).

Bei der Implementierung der Routing-Verfahren und dem Nachrichtenversand wurden Abhängigkeiten zu Android-spezifischen Klassen vermieden, um die Portierbarkeit der

Architektur zu gewährleisten.

6.2.5 Desktop-Kompatibilität

Das in den vergangenen Teilabschnitten beschriebene Bluetooth-Kommunikationsframework nutzt die Bluetooth API [Ope11a] von Android, um mit anderen Geräten zu kommunizieren. In der Java Standard Edition, die auf Desktop-Geräten eingesetzt wird, ist weder diese noch eine andere Bluetooth API enthalten. Um Bluetooth-Geräte mit der Java SE anzusprechen müssen Fremdbibliotheken, wie z.B. *BlueCove* [blu11], eingesetzt werden.

Da eine Integration von Desktop-Geräten eine denkbare Erweiterungsmöglichkeit für diese Arbeit darstellt, wurden alle Funktionalitäten, die direkt die Android-spezifische Bluetooth API ansprechen, gekapselt, um einen Austausch zu vereinfachen. Dabei kam z.B. das Entwurfsmuster *Fabrikmethode* [GHJJ96, S. 131] bei der Instanziierung von Android-spezifischen Klassen zum Einsatz. Konkret wurde es zur z.B. Kapselung des Bluetooth-Adapters genutzt, der den Zugriff auf die Bluetooth Hardware erlaubt.

Hier wird die Instanziierung des Bluetooth-Adapters in der Methode einer Fabrik (`AndroidBluetoothAdapterFactory`) gekapselt, die jederzeit durch eine andere ausgetauscht werden kann. Die Fabrik wird nur unter ihrem Interface `IBluetoothAdapterFactory` genutzt, wodurch keine Abhängigkeiten zur konkret genutzten Fabrik entstehen. Die hier beschriebene Klassenstruktur ist in Abbildung 6.1 als UML-Klassendiagramm dargestellt.

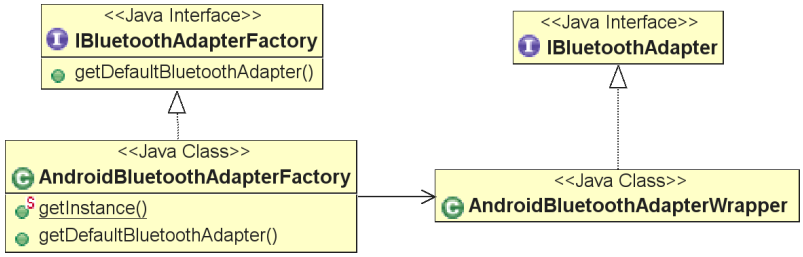

Abbildung 6.1: Klassendiagramm Fabrikmethodenmuster

Durch diese Maßnahmen ist der entstandene Code weitgehend unabhängig von der Bluetooth API und könnte in Zukunft auch auf Desktop-Rechnern mit einer Bluetooth Bibliothek wie BlueCove genutzt werden.

Zunächst soll das Framework jedoch ausschließlich zusammen mit *Jadex-Android* zum Einsatz kommen und muss daher in die Jadex-Plattform integriert werden.

6.3 Integration des Kommunikationsdienstes in Jadex

Die Integration verläuft im Wesentlichen wie im Abschnitt 5.3 bereits beschrieben. Dem in Jadex integrierten AwarenessManagementAgent wird eine neue Discovery-Komponente,

der `BluetoothDiscoveryAgent`, hinzugefügt.
Dieser regelt lediglich den Empfang und Versand der Jadex-Präsenznachrichten, für den Datenaustausch zwischen Komponenten der Plattform ist der neue `BluetoothTransport` zuständig, der als Komponente vom MessageService genutzt wird.

Diese beiden neuen Komponenten kommunizieren per Interprozesskommunikation mit dem Bluetooth-Kommunikationsdienst und starten diesen per Intent, falls notwendig (vgl. Abschnitt 5.1.2). Der Discovery Agent wird über im Netzwerk verfügbare Geräte informiert und versendet periodisch an alle Teilnehmer Präsenznachrichten. Empfangene Präsenznachrichten werden dem AwarenessManagementAgent übergeben und von diesem interpretiert. Eingehende Nachrichten werden dem `BluetoothTransport` übermittelt, der sich dafür als Nachrichtenempfänger beim Kommunikationsdienst anmeldet, dem er auch ausgehende Nachrichten per Interprozesskommunikation übergibt.

Listing 6.4 zeigt die Anbindung an einen Android-Service per Intent, wie sie in den beiden beschriebenen Komponenten auftritt.
Der Aufruf von `Context.bindService()` mit dem Parameter `BIND_AUTO_CREATE` führt dazu, dass der vom Intent angesprochene Dienst vom Android-System gestartet wird, falls er noch nicht läuft.
Das der Methode `onServiceConnected()` übergebene Binder-Objekt kapselt die Interprozesskommunikation und wurde aus einer AIDL-Datei generiert (vgl. Abschnitt 5.1.2).

Listing 6.4: Service-Anbindung der Jadex-Kommunikationskomponenten

```
Intent intent = new Intent(context, ConnectionService.class);
sc = new BTServiceConnection();
context.bindService(intent, sc, Activity.BIND_AUTO_CREATE);

class BTServiceConnection implements ServiceConnection {
    public void onServiceConnected(ComponentName arg0, IBinder arg1)
    {
      binder = ICommServiceConnection.Stub.asInterface(arg1);
      ...
    }
    ...
}
```

Bis hierhin wurden die verschiedenen Entwicklungsschritte hin zu einer Jadex-Plattform beschrieben, die per Bluetooth mit anderen mobilen Jadex-Plattformen kommunizieren kann. Im nächsten Abschnitt wird auf die Nutzung der hier beschriebenen Implementation in eigenen Anwendungen eingegangen.

6.4 Nutzung der Implementierung

Eine Android Activity, die Jadex inklusive der Bluetooth-Kommunikation nutzen soll, muss von der Klasse `JadexBluetoothActivity` abgeleitet werden. So wird sicherge-

stellt, dass der Android-Anwendungskontext, der zum Starten von Android Services benötigt wird, dem `BluetoothDiscoveryAgent` zur Verfügung steht.

Weiterhin muss in der Datei `AndroidManifest.xml`, die den Namen, Version, angebotene Dienste und genutzte API-Funktionen einer Android-Anwendung deklariert, der Bluetooth-Kommunikationsdienst als Android-Service deklariert und die Berechtigungen zur Nutzung von Bluetooth angefordert werden (Listing 6.5).

Listing 6.5: Zusatzeinträge in AndroidManifest.xml

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<service android:name="jadex.android.bluetooth.service.↩
   ConnectionService"/>
```

Als Beispiel zur Nutzung wurde die Chat-Beispielanwendung von Jadex [BP11b] auf Android portiert. Dabei mussten mit Ausnahme der neu zu implementierenden grafischen Oberfläche keine Änderungen am Programmcode vorgenommen werden. Die Chat-Anwendung besteht aus einem `ChatAgent` sowie dem dazugehörigen `ChatService` samt Interface. Der Agent deklariert den ChatService sowohl als *RequiredService*, um andere Chat-Agenten zu finden und mit ihnen zu kommunizieren, als auch als *ProvidedService*, da er ihn selbst auch anbietet (Listing 6.6).

Da die Plattformen durch das Awareness-Verfahren über andere erreichbare Plattformen informiert sind, können Jadex-Anwendungen die von Jadex angebotenen Mechanismen zum Auffinden von Diensten im Netzwerk nutzen. Dazu wird in der Deklaration des vom Agenten benötigten Dienstes der Skopus auf *global* gesetzt, wie in Listing 6.6 zu sehen ist.

Listing 6.6: Dienstdeklaration des Chat-Agenten

```
@ProvidedServices(@ProvidedService(type=IChatService.class))
@RequiredServices({
  @RequiredService(name="chatservices", type=IChatService.class, ↩
     multiple=true,
    binding=@Binding(dynamic=true, scope=RequiredServiceInfo.↩
       SCOPE_GLOBAL))
})
```

Der Agent kann durch den Methodenaufruf `IInternalAccess.getServiceContainer().getRequiredServices()` die Liste aller erreichbaren ChatServices anfordern. Beim diesem Aufruf startet die Jadex-Plattform eine Suchanfrage an alle (durch die Awareness) bekannten Plattformen und liefert gefundene Dienste zurück. Auf diese Art und Weise können Agenten bzw. Komponenten miteinander kommunizieren, ohne Kenntnis von der Bluetooth-Architektur zu haben.

Neben der bisher dargestellten Nutzung von *Jadex-Android* ist es ebenfalls möglich, die Jadex-Plattform in einen Android Service einzubinden. Dabei lässt sich der Vorteil ausnutzen, das Android Services im Hintergrund aktiv sein können, während Activities

Typ	Beschreibung	Größe (Bytes)
SYN/ACK	wird beim Verbindungsaufbau benötigt	0
PING/PONG	wird zum Verbindungserhalt verwendet	0
DSDV-Update	Routing-Informationen des DSDV Protokolls	129
Awareness	Awareness-Information von Jadex	393
4 Byte-Nachricht	Senden von 4 Byte Nutzdaten über Jadex	835

Tabelle 6.2: Typische Nachrichtengrößen (ohne den Header von 60 Byte)

bei Inaktivität beliebig vom Betriebssystem pausiert werden dürfen.
In jedem Fall wird die Bluetooth-Kommunikation in einem separaten Service abgewickelt, sodass keine Verbindungsabbrüche durch das Pausieren einer Activity entstehen.

Nachdem in den bisherigen Abschnitten die Vorgehensweise bei der Implementierung beschrieben wurde und in diesem Abschnitt die Nutzung im Kontext von Anwendungen für *Jadex-Android* gezeigt wurde, folgt im nächsten Schritt eine kurze Evaluation der Übertragungsleistung des Frameworks.

6.5 Evaluation

Neben dem erfolgreichen Austausch von Nachrichten zwischen Jadex-Komponenten über das entwickelte Bluetooth P2P Framework kann die damit zu erreichende Leistungsfähigkeit relevant sein. Beim Nachrichtenversand ist üblicherweise die Verzögerung von Interesse, mit der die Nachricht den Empfänger erreicht.

Mindestens in einem Anwendungsszenario, der Datensynchronisation (vgl. Abschnitt 2.3), ist der auch der Datendurchsatz von Bedeutung, so dass dieser hier ebenfalls betrachtet werden soll.
Bei allen im Folgenden vorgenommenen Messungen beträgt die Distanz zwischen den Geräten weniger als einen Meter, das Netzwerk bestand jeweils aus zwei bis drei Geräten.

6.5.1 Paketgrößen

Um zunächst eine Vorstellung für die Größen der übertragenen Pakete zu bekommen, wurde der Datenverkehr zwischen verschiedenen Jadex-Plattformen analysiert um die typischen Paketgrößen für verschiedene Nachrichten zu ermitteln.

Die Tabelle 6.2 zeigt die unterschiedlichen Paketgrößen, die im Betrieb auftreten. Die Daten beziehen sich auf ein Netzwerk, das aus lediglich zwei Geräten besteht. Außerdem wurden die zum Versand benötigten Paket-Header mit einer festen Größe von 60 Byte nicht in die Größenangaben einbezogen.

Da die Bluetooth-Spezifikation in der aktuell verbreiteten Version eine Übertragungsrate von bis zu 3Mb/s erlaubt [Blu10, Vol.1, S. 124], sind die in der Tabelle gezeigten Paketgrößen unkritisch.

Paketgröße	Ø-Übertragungszeit	Geschwindigkeit
32 Byte	1010 ms	32 Byte/s
512 Byte	1010 ms	507 Byte/s
1 KiB	1059 ms	967 Byte/s
2 KiB	1275 ms	1606 Byte/s
10 KiB	2027 ms	5052 Byte/s
20 KiB	2478 ms	8265 Byte/s

Tabelle 6.3: Datendurchsatz bei direkter Verbindung, Zeit bis zur Bestätigung

6.5.2 Roundtrip-Zeit

Für die Praxistauglichkeit der Nachrichtenübertragung ist die Verzögerung, mit der eine Nachricht den Empfänger erreicht, von Bedeutung. Dabei ist es sinnvoll, die *Roundtrip-Zeit* zu messen, da sie eine Zeitsynchronisation zwischen Sender und Empfänger überflüssig macht [Com03, S. 159f].

Als Roundtrip-Zeit wird die Zeit gemessen, die benötigt wird, bis eine Nachricht den Empfänger erreicht hat und die Empfangsbestätigung des Empfängers beim ursprünglichen Sender angekommen ist. Als Nachricht wird dabei die in Tabelle 6.2 angegebene 4-Byte-Nachricht in beide Richtungen übertragen.

Im konkreten Fall wurde der gesamte Weg der Nachricht von einer Jadex-Komponente zur Zielkomponente auf der entfernten Plattform und wieder zurück einbezogen. Dieses Vorgehen erlaubt zwar keinen Rückschluss auf die Dauer des Nachrichten-Parsings oder der von Jadex genutzten Kompressionsverfahren, bietet aber die beste Abschätzung der Nachrichtenverzögerung beim realen Einsatz von Jadex-Android, da die Zeiten auf Anwendungsebene gemessen werden.

Nach der Implementierung einer Minimalanwendung, die die Roundtrip-Zeit misst, entstanden bei zwei direkt verbundenen Geräten Messwerte im Bereich von $600 - 1200ms$, mit einem Mittelwert von $1010ms$ über 100 Messungen. Bei einem über eine Zwischenstation erreichbarem Gerät liegt die Verzögerung erwartungsgemäß höher, im Bereich von $1500 - 2600ms$ (Mittelwert über 100 Messungen: $1900ms$).

6.5.3 Durchsatz

Um den Durchsatz zu messen, wird wiederum die Roundtrip-Zeit auf Anwendungsebene gemessen. Dabei werden nun allerdings größere Datenmengen verschickt, die randomisiert erzeugt werden. Als Quittung kommt in allen Varianten eine kleine 4 Byte-Nachricht zum Einsatz. Die Tabelle 6.3 zeigt die Ergebnisse der Messungen in einem Netzwerk mit zwei Geräten (direkte Verbindung), Tabelle 6.4 Ergebnisse von Messungen mit Routing über ein Drittgerät. Beide Tabellen zeigen jeweils Mittelwerte über 100 Messungen.

Betrachtet man den Weg eines Pakets auf der zweiten Messstrecke, so durchläuft diese

Paketgröße	Ø-Übertragungszeit	Geschwindigkeit
32 Byte	1900 ms	17 Byte/s
512 Byte	1918 ms	327 Byte/s
1 KiB	2193 ms	721 Byte/s
2 KiB	1949 ms	1363 Byte/s
10 KiB	1967 ms	6376 Byte/s
20 KiB	2211 ms	11222 Byte/s

Tabelle 6.4: Datendurchsatz bei Verbindung über Drittgerät, Zeit bis zur Bestätigung

zwar auf allen Geräten die Routing-Schicht, jedoch nur beim Sender und beim Empfänger die Jadex-Plattform. Durch den Vergleich der Messungen könnte man also auf den Flaschenhals der Implementierung stoßen.

Ein Vergleich bei kleineren Paketgrößen, die über eine Zwischenstation knapp doppelt so lange zur Übertragung benötigen, legt die Vermutung nahe, dass der Flaschenhals in der Implementation des Routing-Protokolls zu suchen ist. Betrachtet man jedoch die größeren Pakete, kommt man zum entgegengesetzten Schluss: Die Übertragungszeiten nähern sich denen der Messungen mit zwei Geräten an, also kann der Übertragungsweg nicht der Flaschenhals sein. Da Messungen dieser Art, gerade bei Funkübertragungen, stark durch die Umgebung beeinflusst werden, ist die Aussagekraft jedoch begrenzt.

Die insgesamt hohen Messwerte lassen sich durch die vielen Zwischenschritte (Jadex Message Service, Serialisierung als XML, Datenkompression, Bluetooth-Dienst, DataPacket, Bluetooth) erklären. Da die Leistungsfähigkeit im Zusammenspiel dieser Komponenten kein prinzipbedingtes Problem ist und es sich bei der Implementierung um einen Prototypen handelt, wird das eventuell vorhandene Optimierungspotential an dieser Stelle nicht weiter berücksichtigt. Für eine Steigerung der Leistungsfähigkeit in zukünftigen Weiterentwicklungen bietet sich der Einstieg durch weitere Messungen, z.B. der Dauer der Nachrichtenkodierung, an.

6.6 Zusammenfassung

In diesem Kapitel wurde die prototypische Implementierung einer Ad-hoc-Kommunikation zwischen aktiven Komponenten auf mobilen Android-Geräten mit Bluetooth vorgestellt. Die gezeigte und umgesetzte Implementation basiert auf den bereits in Kapitel 5 getroffenen Entwurfsentscheidungen: Sie besteht aus einem Kommunikationsdienst, der von mehreren Android-Anwendungen genutzt werden kann, und bindet die Jadex-Plattform fest in die Anwendung mit ein.

Die Portierung von Jadex auf Android bestand im Wesentlichen aus der Anpassung der von Android nicht unterstützten Code-Segmente und der Einbindung der Jadex-Abhängigkeiten in Android-kompatiblen Versionen.

Der Peer-to-Peer-Kommunikationsdienst besteht aus mehreren Threads, die auf eingehende Verbindungen warten, sowie einem Verfahren, welches erreichbare Geräte auffindet. Bei diesem automatischen Verbindungsverfahren werden allerdings nur solche Geräte gefunden, mit denen bereits ein Bluetooth Pairing besteht (vgl. Abschnitt 5.2.1). Die Verbindungen werden pro Gerät zentral verwaltet, sodass kein Unterschied zwischen ein- oder ausgehender Verbindung gemacht wird. Beim Senden und Empfangen von Paketen wird das Routing-Verfahren DSDV eingesetzt (siehe Abschnitt 3.2.3).

Für die Übertragung der Nachrichten wurde ein eigenes Paketformat entwickelt, was Datenfelder enthält, die z.B. die Größe der transportierten Nutzdaten und eine einmalige ID speichern. Dadurch können Nachrichten aus mehreren empfangenen Teilstücken wieder zusammengesetzt werden und der doppelte Empfang von Nachrichten wird verhindert.

Wie im Entwurf bereits dargestellt, reichen zwei in die Jadex-Plattform eingebundene Komponenten aus, um den Nachrichtentransport von Jadex über die Peer-to-Peer-Schicht zu ermöglichen. Dabei wird bei Bedarf der Kommunikationsdienst gestartet.

Um die in dieser Arbeit entwickelte Jadex-Plattform für Android mitsamt der Bluetooth-Kommunikation zu nutzen, bedarf es der Einbindung der entstandenen *Jadex-Android*-Bibliotheken sowie kleinerer Anpassung an der Anwendungsbeschreibungsdatei `AndroidManifest.xml`. Bei der Entwicklung der Android-Anwendung muss allerdings im Gegensatz zu Jadex-Desktop-Anwendungen für die Kopplung von Benutzeroberfläche und Agenten selbst gesorgt werden.

Die Evaluation der Leistungsfähigkeit am Ende des Kapitels hat gezeigt, dass dem praktischen Einsatz der vorgeschlagenen Lösung in den meisten Anwendungsszenarien nichts im Wege steht, zumal vermutlich an einigen Stellen noch Optimierungspotential besteht. Für einige Szenarien, wie z.B. dem Einsatz beim RoboCup, sind die Roundtrip-Zeiten der prototypische Implementierung jedoch zu hoch.

Dieser Abschnitt schließt den praktischen Teil der Arbeit ab. Nachdem nun von den Grundlagen über den Entwurf bis zur Implementation eine Lösung zur Ad-hoc-Kommunikation aktiver Komponenten vorgestellt wurde, folgt im nächsten und letzten Kapitel eine Schlussbetrachtung mit Zusammenfassung und Ausblick auf mögliche Weiterentwicklungen.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel gibt eine Zusammenfassung über die gesamte Arbeit, die aus den Grundlagen, dem Entwurf und der Implementation der Ad-hoc-Kommunikation aktiver Komponenten auf mobilen Geräten besteht. Die Zusammenfassung soll außerdem eine abschließende Betrachtung des Themas geben. Weiterhin werden offene Punkte und mögliche weiterführende Arbeiten im Abschnitt **Ausblick** (7.2) aufgeführt.

7.1 Zusammenfassung

In dieser Bachelorarbeit wurde die anhaltende Steigerung der Leistungsfähigkeit mobiler Geräte zum Anlass genommen, alternative und moderne Softwareentwicklungsparadigmen auch für mobile Geräte einzusetzen. Als Kombination der Service-, Komponenten-, und Agentenorientierung wurde das Konzept der aktiven Komponenten (vgl. Abschnitt 3.1.2) aufgegriffen und anhand der Jadex-Plattform für mobile Geräte umgesetzt. Aktive Komponenten bieten insbesondere die Verwendung von agentenorientierter Softwareentwicklung für die Entwicklung ihrer Funktionalität an. In Kombination mit der Konfigurierbarkeit der Komponentenarchitektur und der dynamischen Dienstanbindung der Serviceorientierung unterstützen aktive Komponenten damit (Software-)Komposition, Nebenläufigkeit und Nachrichtenkommunikation, was vor allem bei der Entwicklung von verteilten Anwendungen vorteilhaft ist.

Der Fokus der Arbeit lag auf der Realisierung der Ad-hoc-Kommunikation zwischen mobilen Plattformen, da dieser Aspekt in Desktop-Plattformen wie Jadex zwar keine Relevanz hat, für die Kommunikation zwischen mobilen Plattformen aber umso wichtiger ist (vgl. Abschnitt 1.1). Eine solche Ad-hoc-Kommunikation mit Geräten, die sich in der näheren Umgebung befinden, ist für viele Anwendungsszenarien notwendig (vgl. Kapitel 2).

Neben der Jadex-Plattform ist die dafür benötigte Funktechnologie eine weitere Grundlage. Im Abschnitt 3.2 wurden die zwei auf mobilen Geräten gängigsten Lösungen, WLAN und Bluetooth, vorgestellt. Im Entwurf dieser Arbeit (Kapitel 5) wurde Bluetooth gewählt, da unter anderem der WLAN-Ad-hoc-Modus nicht auf allen Mobilgeräten unterstützt

wird. Bluetooth erlaubt die Bildung von Piconetzen mit bis zu acht Teilnehmern sowie den Aufbau von Scatternetzen, die aus mehreren Piconetzen bestehen, aber ein eigenes Routing-Verfahren benötigen. Scatternetze können weiterhin die Reichweite des aufgebauten Peer-to-Peer-Netzes erhöhen.

Von den verschiedenen, in den Grundlagen vorgestellten Routing-Verfahren (vgl. Abschnitt 3.2.3) wurde DSDV für die Implementation gewählt. Beim Destination-Sequenced-Distance-Vector-Verfahren speichert jeder Knoten im Netzwerk eine Routing-Tabelle, die periodisch und bei Veränderungen des Netzes an alle Nachbarn verschickt wird. Durch die geschickte Wahl einer Sequenznummer wirkt das Routing-Verfahren Problemen mit variablen Verbindungen entgegen und ist gut für Ad-hoc-Netzwerke geeignet.

Der Entwurf (Kapitel 5) gliederte sich in drei Abschnitte: Die Portierung und Integration der Jadex-Plattform in das für diese Arbeit gewählte Android-Betriebssystem, den Entwurf eines Bluetooth-Kommunikationsdienstes, der die beschriebene Bluetooth-Technologie nutzt, sowie die Einführung neuer Plattformkomponenten, die die Schnittstelle zwischen Jadex-Komponenten und Kommunikationsdienst darstellen.
Jadex wurde dabei als Laufzeitbibliothek in Android-Anwendungen integriert, um auf Kosten des Speicherverbrauchs größtmögliche Flexibilität zu gewährleisten. Die Erweiterungen der Jadex-Plattform beschränken sich auf einen neuen Message Transport sowie einen weiteren Discovery Agent (vgl. Abschnitt 5.3). Diese beiden Komponenten vermitteln zwischen Jadex-Plattform und dem Bluetooth-Kommunikationsdienst, den sie als Android Service ansprechen, um die Nachrichtenvermittlung abzuwickeln.

Die dem Entwurf folgende prototypische Implementation (Kapitel 6) nutzt einen Präprozessor, um die für eine lauffähige Android-Version notwendigen Anpassungen in Jadex vorzunehmen. Die so modifizierten Quelltexte flossen in den Hauptentwicklungszweig ein, sodass eine Weiterentwicklung von *Jadex-Android* nach Abschluss dieser Bachelorarbeit möglich sein wird.
Neben der reinen Umsetzung des Entwurfs war es während der Implementation notwendig, ein eigenes Nachrichtenformat zu erstellen, welches die zur Übertragung und zum Routing notwendigen Datenfelder enthält (vgl. Abschnitt 6.2.3).
Um die mobile Jadex-Plattform mitsamt der Bluetooth-Kommunikation zu verwenden, reichen wenige Anpassungen der Android-Anwendung aus. Das Ad-hoc-Netzwerk wird daraufhin automatisch mit allen gekoppelten Geräten aufgebaut, die sich in Reichweite befinden.

Letzteres stellt die größte Einschränkung der gewünschten Konfigurationslosigkeit (vgl. Abschnitt 5.2) dar: Geräte müssen einander auf Bluetooth-Ebene bekanntgemacht werden, bevor sie dem Netzwerk beitreten können. Diese Einschränkung ist jedoch insofern erträglich, als das eine Kopplung lediglich einmal pro Gerät vorgenommen werden muss. Außerdem reicht es aus, wenn jedes Gerät eine Kopplung zu *einem* weiteren, in Reichweite befindlichem Gerät besitzt, um am Netzwerk teilzunehmen.

Der Entwurf, der in dieser Arbeit entstanden ist, zeigt eine Möglichkeit auf, wie die

Kommunikation zwischen aktiven Komponenten auf mobilen Geräten aussehen kann. Die verwendeten Technologien, Bluetooth und Android, sind bereits weit verbreitet und laden, zusammen mit der Leistungsfähigkeit moderner mobiler Geräte, zur Nutzung neuer Konzepte der Softwareentwicklung ein.
Durch die Verwendung von aktiven Komponenten auf mobilen Geräten reduzieren sich die für den Entwickler sonst schwierigen Aufgaben, wie Nebenläufigkeit und Nachrichtenaustausch, auf das Nutzen eines vorhandenen Frameworks.

Insgesamt stellt diese Arbeit neue Möglichkeiten in der Entwicklung verteilter Anwendungen für mobile Geräte in Aussicht, wobei die prototypische Implementierung die Realisierbarkeit zeigt und für zukünftige Entwicklungen als Basis dienen kann.

7.2 Ausblick

Wie bereits im vorherigen Abschnitt erwähnt wurde, ist die Entwicklung, die während dieser Arbeit stattgefunden hat, in den Jadex-Hauptentwicklungszweig eingeflossen. Die reine Portierung von Jadex auf das Android-Betriebssystem ist bereits als stabil zu betrachten und kann zur Entwicklung eingesetzt werden. Es bleiben allerdings noch Verbesserungsmöglichkeiten offen, beispielsweise hinsichtlich der Leistungsfähigkeit beim Nachrichtenversand.

Offen ist auch ein sinnvolles Muster zur Kopplung zwischen aktiven Komponenten und Benutzungsschnittstelle auf Android-Geräten. Während der ansonsten problemlosen Entwicklung der ersten Beispielanwendung für das entstandene Framework ist die Art und Weise, in der sich Jadex-Desktop-Anwendungen von typischen Android-Anwendungen unterscheiden, negativ aufgefallen:
Da eine *Jadex-Android*-Plattform aus einer Android Activity heraus gestartet wird, anstatt – im Fall von Jadex-Desktop-Anwendungen – andersherum, hat die Plattform keinen direkten Zugriff auf die Benutzungsoberfläche. Dieser Zugriff wird in üblichen Jadex-Anwendungen jedoch genutzt, um Komponenten auf die grafische Ausgabe wirken zu lassen.
Außerdem dürfen viele Zugriffe auf Android-Ressourcen, wie Dateien, Bluetooth-Schnittstelle oder andere Android-spezifische Funktionen, nur aus dem Kontext einer Activity stattfinden. Da die Jadex-Plattform kein Wissen über die sie umgebende Activity hat, kann sie demnach auf solche Funktionen nicht (direkt) zugreifen. Ein Lösungsansatz ist in Form eines Jadex-Dienstes, der den Zugriff auf Android Ressourcen ermöglicht, denkbar.

Dieser Ausblick zielt weiterhin auf die Ad-hoc-Kommunikation, die vielfältige Alternativlösungen und Verbesserungsmöglichkeiten zulässt.

Denkbar wäre beispielsweise eine Ablösung des Awareness-Verfahrens durch ein engeres Zusammenspiel zwischen Jadex und der Bluetooth-Architektur. Da das DSDV-Verfahren bereits Präsenznachrichten verschickt, könnten diese um Jadex-spezifische Daten, wie dem Plattformnamen, ergänzt werden. Für den geringeren Verwaltungsauf-

wand opfert man dadurch allerdings die Entkopplung und Transparenz, mit der in der hier implementierten Variante Jadex-Nachrichten vom Bluetooth-Kommunikationsdienst gehandhabt werden.

Neben der hier genutzten Bluetooth-Übertragung können weitere Technologien geprüft werden, beispielsweise die Nutzung von Wi-Fi Direct [Wi-11], welches das im Abschnitt 3.2.1 beschriebene WLAN um Funktionen für mobile Ad-hoc-Vernetzung erweitert. Zukünftige Verfahren könnten außerdem ein separates Routing-Verfahren überflüssig machen und somit die Komplexität der hier vorgestellten Lösung herabstufen.
Die Entscheidung für das Routing-Verfahren DSDV in dieser Arbeit erfüllt die Ansprüche der Anwendungsszenarien an die Dynamik des Netzes und ermöglichte die einfache Umsetzung der Jadex Awareness. Für weitere, zukünftige Anwendungen könnten andere Verfahren aber besser geeignet sein. Die Implementation und Evaluation weiterer Lösungen kann also nützlich sein, um bei der Anwendungsentwicklung aus verschiedenen Verfahren wählen zu können.

Nicht zuletzt ist eine Implementation des Kommunikationsdienstes für Java SE möglich, sodass auch mobile bzw. stationäre Geräte ohne Android direkt in das Ad-hoc-Netzwerk eingebunden und in die Kommunikation einbezogen werden können.

Literaturverzeichnis

[ADMTH09] ADAMS, Bram ; DE MEUTER, Wolfgang ; TROMP, Herman ; HASSAN, Ahmed E.: Can we refactor conditional compilation into aspects? In: *Proceedings of the 8th ACM international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2009 (AOSD '09), 243–254

[Apa11] APACHE SOFTWARE FOUNDATION: *Maven*. `http://maven.apache.org/index.html`. Version: Oktober 2011

[App11] APPLE: *iOS Dev Center*. `http://developer.apple.com/devcenter/ios`. Version: August 2011

[Blu10] BLUETOOTH SPECIAL INTEREST GROUP: *Specification of the Bluetooth System*. 4.0, Juni 2010. `https://www.bluetooth.org/Technical/Specifications/adopted.htm`

[blu11] *BlueCove*. `http://www.bluecove.org/`. Version: Oktober 2011

[BMJ+98] BROCH, Josh ; MALTZ, David A. ; JOHNSON, David B. ; HU, Yih-Chun ; JETCHEVA, Jorjeta: A performance comparison of multi-hop wireless ad hoc network routing protocols. In: *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA : ACM, 1998 (MobiCom '98), S. 85–97

[BP11a] BRAUBACH, Lars ; POKAHR, Alexander: Addressing Challenges of Distributed Systems using Active Components. In: *In Proceedings of 5th International Symposium on Intelligent Distributed Computing (IDC-2011)*, 2011

[BP11b] BRAUBACH, Lars ; POKAHR, Alexander: *Jadex*. `http://jadex-agents.informatik.uni-hamburg.de/`. Version: August 2011

[BP11c] BRAUBACH, Lars ; POKAHR, Alexander: *Jadex Standalone Platform Guide*, August 2011

[BPJ+10] BRAUBACH, Lars ; POKAHR, Alexander ; JANDER, Kai ; LAMERSDORF, Winfried ; BURMEISTER, Birgit: Go4Flex: Goal-oriented Process Modelling. In: AL., M. E. (Hrsg.) ; Intelligent Distributed Computing IV, SCI 315 (Veranst.): *Proc. 4th International Symposium on Intelligent Distributed Computing (IDC-2010)* Intelligent Distributed Computing IV, SCI 315, Springer-Verlag, 9 2010, S. 77–87

[CB03] CARABELEA, Cosmin ; BOISSIER, Olivier: Multi-Agent Platforms on Smart Devices: Dream or Reality ? In: *Proceedings of the Smart Objects Conference (SOC03), Grenoble, France*, 2003, S. 126–129

[Com03] COMER, Douglas: *TCP/IP: Konzepte, Protokolle, Architekturen*. Mitp, 2003

[CP03] CAIRE, Giovanni ; PIERI, Federico: *LEAP User Guide*, 2003

[DEF+08] DUNKEL, Jürgen ; EBERHART, Andreas ; FISCHER, Stefan ; KLEINER, Carsten ; KOSCHEL, Arne: *Systemarchitekturen für Verteilte Anwendungen*. München : Hanser, 2008

[Ern11a] ERNEST FRIEDMAN-HILL: *JESS, the Rule Engine for the JavaTM Platform*. `http://herzberg.ca.sandia.gov/jess/`. Version: September 2011

[Ern11b] ERNST, NICO: Nvidia Tegra 3 als mobiler Quad-Core für Tablets. (2011). `http://www.golem.de/1101/80946.html`

[FIP04] FIPA: *FIPA Agent Management Specification (SC00023K)*, März 2004

[FIP11] FIPA: *FIPA-OS Agent Platform*. `http://fipa-os.sourceforge.net/`. Version: September 2011

[fre11] *Freenet Project*. `http://freenetproject.org/`. Version: Oktober 2011

[GHJJ96] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; JOHN, Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. Bonn : Addison-Wesley, 1996. – Design Patterns, 1995, Deutsche Übersetzung von Dirk Riehle

[Hei11] HEISE: WLAN-Mesh-Standard endlich fertig. (2011), Oktober. `http://heise.de/-1352670`

[IEE11a] IEEE-SA: *IEEE 802.11: Wireless Local Area Networks (LANs)*, Oktober 2011

[IEE11b] IEEE-SA: *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, Amendment 10: Mesh Networking*, Oktober 2011

[Jen00] JENNINGS, N.R.: On agent-based software engineering. In: *Artificial Intelligence* 117 (2000), Nr. 2, S. 277–296

[Joh94] JOHNSON, D.B.: Routing in ad hoc networks of mobile hosts. In: *Mobile Computing Systems and Applications, Proceedings*, 1994, S. 158 –163

[JW98] JENNINGS, N.R. ; WOOLDRIDGE, M.J.: *Agent technology: foundations, applications, and markets*. Springer, 1998

[Ker88] KERNIGHAN, Brian W. ; RITCHIE, Dennis M. (Hrsg.): *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988

[Law02] LAWRENCE, Jamie: LEAP into Ad-hoc Networks. In: *Proceedings of the First International Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices (AAMAS)*, 2002

[McI68] MCILROY, M. D.: Mass-produced software components. In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968)

[MM04] MURTHY, C. Siva R. ; MANOJ, B.S.: *Ad Hoc Wireless Networks: Architectures and Protocols*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2004

[Ope11a] OPEN HANDSET ALLIANCE: *Android Bluetooth API Documentation*, September 2011. `http://developer.android.com/guide/topics/wireless/bluetooth.html`

[Ope11b] OPEN HANDSET ALLIANCE: *Android developer docs*. `http://developer.android.com/`. Version: August 2011

[Ora11] ORACLE: *Java for Mobile Devices Documentation*. `http://download.oracle.com/javame/mobile.html`. Version: August 2011

[PB94] PERKINS, Charles E. ; BHAGWAT, Pravin: Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In: *Proceedings of the conference on Communications architectures, protocols and applications*. New York, NY, USA : ACM, 1994 (SIGCOMM '94), S. 234–244

[PB11] POKAHR, Alexander ; BRAUBACH, Lars: Active Components: A Software Paradigm for Distributed Systems. In: *In Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2011)*, 2011

[PBH00] POSLAD, Stefan ; BUCKLE, Phil ; HADINGHAM, Rob: *The FIPAOS agent platform: Open Source for Open Standards*. 2000

[PBL05] POKAHR, Alexander ; BRAUBACH, Lars ; LAMERSDORF, Winfried: Jadex: A BDI Reasoning Engine. In: R. BORDINI, J. D. M. Dastani D. M. Dastani (Hrsg.) ; SEGHROUCHNI, A. El F. (Hrsg.): *Multi-Agent Programming*, Springer Science+Business Media Inc., USA, 9 2005, S. 149–174. – Book chapter

[PBW04] PIRKER, Michael ; BERGER, Michael ; WATZKE, Michael: An Approach for FIPA Agent Service Discovery. In: *agent service discovery in mobile ad hoc environments. In Proceedings of the Conference on Autonomous Agents and Multiagent Systems - Workshop on Agents for Ubiquitous Computing, New York, USA*, 2004

[PT09] PATZLAFF, Marcel ; TUGULDUR, Erdene-Ochir: MicroJIAC 2.0 - The Agent Framework for Constrained Devices and Beyond / DAI-Labor, Technische Universität Berlin. 2009 (TUB-DAI 07/09-01). – Forschungsbericht

[Rob11] ROBOCUP FEDERATION: *Robocup Website*. `http://www.robocup.org/`. Version: August 2011

[Rot05] ROTH, Jörg: *Mobile Computing - Grundlagen, Technik, Konzepte (2. Aufl.)*. dpunkt.verlag, 2005. – I–XII, 1–494 S.

[Sch11] SCHULZE, Ronny: Notebook, Netbook, Tablet und jetzt auch noch Ultrabook. (2011), Juni. `http://digitallife.germanblogs.de/archive/2011/06/28/notebook-netbook-tablet-und-jetzt-auch-noch-ultrabook-was-brauche-ich-wofuer.htm`

[Son11] SONATYPE: *Munge Maven Plugin*. `http://sonatype.github.com/munge-maven-plugin/`. Version: September 2011

[Sou11] SOURCEFORGE: *Prebop Preprocessor*. `http://prebop.sourceforge.net/`. Version: September 2011

[Sun03] SUN MICROSYSTEMS: *Project JXTA v2.0: Java Programmer's Guide*, März 2003

[Til11] TILAB: *Jade - Java Agent DEvelopment Framework*. `http://jade.tilab.com/index.html`. Version: September 2011

[Var08] VARDA, Kenton: Protocol Buffers: Google's Data Interchange Format / Google. Version: 6 2008. `http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html`. 2008. – Forschungsbericht

[Whi04] WHITE, S.A.: *Introduction to BPMN*, 2004

[Wi-11] WI-FI ALLIANCE: *Wi-Fi Direct*. `http://www.wi-fi.org/Wi-Fi_Direct.php`. Version: Dezember 2011

[WSL04] WANG, Yu ; STOJMENOVIC, I. ; LI, Xiang-Yang: Bluetooth scatternet formation for single-hop ad hoc networks based on virtual positions. In: *Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on* Bd. 1, 2004, S. 170 – 175 Vol.1

[WXA09] WANG, Jungfang ; XIE, Bin ; AGRAWAL, Dharma P.: Journey from Mobile Ad Hoc Networks to Wireless Mesh Networks. In: MISRA, Sudip (Hrsg.) ; MISRA, Subhas C. (Hrsg.) ; WOUNGANG, Isaac (Hrsg.): *Guide to Wireless Mesh Networks*. Springer London, 2009 (Computer Communications and Networks), S. 1–30

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.
Die Arbeit wurde vorher nicht in einem anderen Prüfungsverfahren eingereicht und die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____