Rundenbasierte Strategiespiele in Android-basierten Opportunistischen Netzwerken

Bachelorarbeit

von

Fabian Weiß

aus

Leverkusen

eingereicht bei

Technik Sozialer Netzwerke
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

Oktober 2017

Betreuer:

Andre Ippisch, M. Sc.

Zusammenfassung

Die Anzahl der Menschen, welche Android-Smartphones täglich nutzen, steigt zunehmend. Viele davon nutzen ebenso täglich Spiele-Apps, um zum Beispiel Wartezeiten zu vertreiben. Mithilfe dieser Geräte ist es außerdem möglich, Opportunistische Netzwerke zu bilden. Diese ermöglichen es, Informationen auch ohne die Nutzung von Mobilfunkkommunikation, welche meistens über dritte Anbieter angeboten wird, zu versenden. In dieser Arbeit wird die Umsetzung von Spiele-Apps auf Basis dieser Opportunistischen Netzwerke behandelt. Die Apps sollen es den Nutzern erlauben, ohne die Nutzung von Datenvolumen und mit mehreren Nutzern gleichzeitig Spiele zu spielen. Dies könnte neue Nutzer motivieren, auch an dem Netzwerk teilzunehmen. Im Zuge der Arbeit wurden sowohl eine Plattform zur Realisierung von rundenbasierten Strategiespielen als auch zwei solcher Strategiespiele entwickelt, welche über Opportunistische Netzwerke nutzbar sind. Die Plattform verbindet sich mit dem Netzwerk via *Opptain*. *Opptain* ist eine App, welche vom Lehrstuhl für Technik sozialer Netzwerke der Heinrich-Heine-Universität Düsseldorf entwickelt wird und anderen Apps die Kommunikation über ein Opportunistisches Netzwerk ermöglicht. Für die Realisierung der Plattform, welche den Namen *GameLayer* trägt, wurde ein Protokoll entwickelt, das in der Arbeit vorgestellt und ausgewertet wird.

Danksagung

Ich möchte mich vor allem bei meiner Familie für die Unterstützung bedanken, welche es ermöglicht hat, diese Arbeit zu verfassen.
Außerdem möchte ich mich bei Andre Ippisch für die Betreuung während der Erstellung dieser Arbeit bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Motivation

In den letzten Jahren ist die Anzahl von privat genutzten mobilen Android-basierten Smartphones stetig gestiegen. So liegt der Marktanteil von Smartphones mit dem Android-Betriebssystem laut Statista im März 2017 bei 86% [Staa]. Diese Smartphones haben bereits mehr Rechenstärke als unsere Computer vor einigen Jahren. Durch die Nutzung von Apps, welche es den Geräten erlauben, an Opportunistischen Netzwerken teilzunehmen, erhalten ihre Nutzer die Möglichkeit, auch ohne Mobilfunkverbindung – teilweise über große Entfernungen – Daten auszutauschen.
Zusätzlich hat vor allem auch die Mobilegaming-Industrie an Bedeutung gewonnen. So gibt es viele Nutzer, welche auf ihren Smartphones Spiele-Apps nutzen. Laut Statista lag die Zahl der mobilen Spieler in den USA in 2015 bei 164.9 Millionen und wurde für das Jahr 2017 auf 192.2 Millionen geschätzt [Stab]. Diese Spiele-Apps nutzen oft eine Verbindung zum Internet, um ihre Dienste anzubieten. Nun stellt sich die Frage, ob man Spiele-Apps auch in Opportunistischen Netzwerken realisieren kann, um damit gegebenenfalls die Attraktivität dieser Netzwerke zu erhöhen. Da diese Peer-basiert sind, wird ihre Funktionalität durch die steigende Anzahl von Nutzern verbessert.
In dieser Arbeit wird die Frage bearbeitet, in welchem Ausmaß sich rundenbasierte Strategiespiele in Android-basierten Opportunistischen Netzwerken realisieren lassen. Dazu werden sowohl Anwendungen, welche die Spiele bereitstellen, als auch ein Protokoll, welches die Kommunikation zwischen den Anwendungen ermöglicht, entwickelt.

1.2 Verwandte Arbeiten

Die Anwendung *Opptain* wurde von Andre Ippisch in seiner Masterarbeit [Ipp15] vorgestellt. Die Anwendung ermöglicht es, mithilfe von Android-basierten Geräten ein Opportunistisches Netzwerk aufzubauen und stellt die Basis der im Rahmen der Arbeit getätigten Implementierungen dar.

1.3 Exposition

Im folgenden zweiten Kapitel werden die Anforderungen festgelegt, welche sich für die realisierbaren Spiele und die Apps, welche die Spiele realisieren, ergeben. Ferner werden die Interaktionen festlegen, die das Protokoll realisieren muss.
Im dritten Kapitel wird die Implementierung präsentiert, welche auf Basis der Anforderungen geleistet wurde. Hierbei wird speziell auf die Kommunikation der Anwendungen und die Datenbank eingegangen.
Im vierten Kapitel wird der Aufbau und die Interaktionen des entwickelten Protokolls beschrieben.
Im fünften Kapitel wird die Implementierung ausgewertet. Hierbei werden sowie problematische Netzwerktopologien diskutiert als auch die verschiedenen Ansätze zur Verbreitung von Informationen zwischen den Nutzern ausgewertet.
Im letzten Kapitel werden die Ergebnisse der Arbeit zusammengefasst und mögliche zukünftige Arbeiten angesprochen.

Kapitel 2

Anforderungen

2.1 Rundenbasierte Strategiespiele

Rundenbasierte Strategiespiele zeichnen sich vor allem dadurch aus, dass sie die Spieler auf strategische bzw. taktische Weise die ihnen bereitgestellten Ressourcen verwalten lassen [Mobb]. Als Beispiel, welches vor allem den Opportunistischen Netzwerken zu gute kommt, lassen sich Post- bzw. E-Mail-Spiele wie zum Beispiel Briefschach anführen. Es wird wie das herkömmliche Schachspiel gespielt. Hier wissen beide Spieler genau, welches Spiel gespielt wird. Die Züge werden dann über Post oder entsprechend E-Mail an den Mitspieler gesendet. Dies ließe sich auch in Opportunistischen Netzwerken [Now17] realisieren, indem die genutzte Anwendung ein Protokoll nutzt, welches es ermöglicht, die Züge der Nutzer den jeweils anderen Nutzern zu übermitteln und so den aktuellen Zustand des Spiels zu synchronisieren.
Wenn das Spiel aber nun mit mehr als zwei Spielern gespielt werden soll, ist es nicht optimal, einzelne Züge zu verschicken, da jeder Spielzug bei jedem Spieler ankommen müsste, bevor dieser erneut ziehen kann, da sonst kein vollständig aktueller Spielstand vorliegen würde. Hier bietet es sich an, den gesamten Zustand des Spiels zu versenden und so immer nur mit seinem Vorgänger zu kommunizieren.
So ergeben sich folgende Anforderungen an die Spiele:

1. Die Spiele müssen rundenbasiert sein. D. h. von Beginn des Spiels an muss klar sein, welcher Spieler zu welchem Zeitpunkt am Zug ist. Es darf immer nur ein Spieler gleichzeitig an der Reihe sein.

2. Der Spielstand muss gespeichert werden können. D. h. der Spielstand muss in Daten dargestellt werden können, damit er an die Mitspieler versandt werden kann.

Diese Anforderungen erfüllen viele Brettspiele, da diese in den meisten Fällen rundenbasiert sind und einen klaren Aufbau des Spielfeldes haben, welchen man in Form von Daten speichern könnte. Ferner fallen unter diese Anforderungen viele Computerspiele wie zum Beispiel die Spiele der Reihen *Civilization* [Moba] oder *Heroes of Might and Magic* [Mobc].

2.2 Anwendung

Zur Realisierung der oben definierten Spiele auf Android-basierten Geräten muss es entsprechend eine Anwendung geben, welche die Spielstände verwaltet. Als Spielstand sollen ein gespieltes Spiel, der aktuelle Zustand dieses Spiels und die damit assoziierten Mitspieler gespeichert werden. Neben der Verwaltung der Spielstände muss die App die Kommunikation mit den Mitspielern umsetzen. Dafür muss sie in der Lage sein, das dafür entwickelte Protokoll umzusetzen. Zur Kommunikation muss die Anwendung außerdem eine Verbindung zu einem Opportunistischen Netzwerk herstellen können. Dafür bietet sich die App *Opptain* an, da sie ein solches Netzwerk und entsprechende Schnittstellen bereitstellt. Hierbei soll die Anwendung dennoch nicht die Spiellogik, die Kodierung der Spielstände und die Darstellung eines Spielstandes umsetzen. Dies ist für jedes Spiel spezifisch und sollte in einer eigenen Anwendung des entsprechenden Spiels umgesetzt werden. Die beschriebene Anwendung wird im Folgenden als Verwaltungs-App bezeichnet. Die getätigte Implementierung der App trägt den Namen *GameLayer*. Sie verwaltet die von nun an so genannten Spiele-Apps.

2.3 Spiele-Apps

Die Spiele-Apps müssen sich mit der Verwaltungs-App verbinden können. Ferner muss sie sowohl die Spiellogik realisieren als auch eine Funktion, Spielstände zu erstellen, diese in Form von Daten zu kodieren und auch wieder aus den entsprechenden Daten herzustellen, anbieten.

2.4 Protokoll

Das Protokoll sollte die Möglichkeit bieten, die unten aufgeführten Interaktionen der Verwaltungs-App zu realisieren und auch in Opportunistischen Netzwerken nutzbar sein. Hierbei ist vor allem Augenmerk darauf zu richten, dass nicht garantiert ist, ob oder wann Nachrichten übertragen werden [Now17]. Im Folgenden werden die Interaktionen aufgeführt, welche das Protokoll realisieren muss.

2.4.1 Kennenlernen von Peers

Wenn sich Peers über *Opptain* kennen gelernt haben, so ist nicht garantiert, dass beide auch die Verwaltungs-App nutzen. So müssen sich die Peers erst entsprechend verifizieren. Hierbei würde es sich empfehlen, auch soziale Informationen wie einen Nickname (Spitznamen) und einen Profiltext auszutauschen, um dem Nutzer eine bessere Lesbarkeit der Kontaktdaten zu bieten.

2.4.2 Erstellen von Spielständen

Wenn nun ein Benutzer ein Spiel mit seinen Peers starten möchte, so kann er dies tun, indem er ein gewünschtes Spiel und die gewünschten Mitspieler auswählt. Danach sollte das Protokoll bei den gewählten Mitspielern anmelden, dass ein neues Spiel gestartet worden ist. Diese sollten die neuen Informationen dann speichern können.

2.4.3 Synchronisierung von Spielständen

Nach dem Zug eines Mitspielers muss dieser über das Netzwerk propagiert werden, um den Spielstand bei den Mitspielern zu synchronisieren. Bei dem Versenden dieser Informationen bieten sich drei Strategien zur Verteilung der Informationen an:

Broadcast-Strategie Wenn das Update nach einem Zug an alle anderen Spieler gesendet wird, so haben diese eine bessere Übersicht darüber, was aktuell im Spiel passiert. So kann

der Verlauf der Spiels auch bei größerer Spielerzahl immer möglichst aktuell mitverfolgt werden. Diese Option könnte durch die hohe Menge an versandten Nachrichten die Performance des Netzwerkes stören.

Einfach-Gezielt-Strategie Zum Erhalten des Spielflusses wäre es nur nötig, dass der Spieler, welcher auch als nächstes am Zug ist, den vergangen Zug erhält, damit er den nächsten Zug tätigen kann. Hierbei würden die Intervalle zwischen den Aktualisierungen vor allem für Spieler, welche nicht als nächstes am Zug sind oder gerade erst gezogen haben, steigen. Allerdings würde sich auch die Last auf dem Netzwerk im Vergleich zur Broadcast-Strategie verringern.

Zweifach-Gezielt-Strategie Genau wie bei der Einfach-Gezielt-Strategie sendet diese Strategie die Aktualisierung nicht an alle Nutzer. Im Gegensatz zur Einfach-Gezielt-Strategie wird die Aktualisierung nicht nur an den jeweiligen Nachfolger, sondern auch an den Vorgänger gesendet.

In Abbildung 2.1 sind die Strategien schematisch dargestellt. Die Zahlen unter den Smartphones zeigen die Zugreihenfolge an. Die Versendungen der Einfach-Gezielt-Strategie sind durch den durchgezogenen Pfeil dargestellt, die der Zweifach-Gezielt-Strategie durch die gestrichelten Pfeile und die der Broadcast-Strategie durch die gepunkteten Pfeile.

Neben den aufgeführten Strategien wären noch andere Ansätze zum Versenden von Nachrichten denkbar, wie beispielsweise die zufällige Auswahl eines Empfängers.

2.4.4 Austreten aus laufenden Spielen

Jedem Nutzer der App sollte die Möglichkeit gegeben werden, aus laufenden Spielen auszutreten. Dies sollte möglich sein, ohne den laufenden Spielfluss zu stören. Ferner sollten die anderen Peers keine Nachrichten mehr an den ausgetretenen Spieler senden, um sein Endgerät nicht weiter zu belasten.

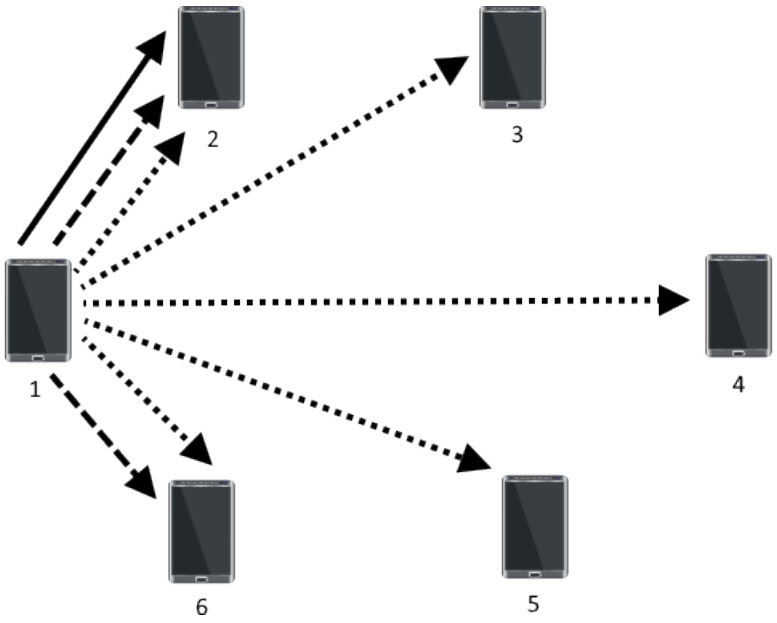

Abbildung 2.1: Die Strategien

2.4.5 Funktionalität in Opportunistischen Netzwerken

Da das Protokoll in einem Opportunistischen Netzwerk verwendet werden soll, muss es, wie oben bereits bemerkt, auch mit Verlust oder Verzögerung der versendeten Nachrichten umgehen können. Dies ist kritisch für die Aktualisierung der Spielstände, da der Erhalt einer Nachricht nicht verlässlich bestätigt werden kann, wenn sowohl die initiale Nachricht als auch die Bestätigung verloren gehen können.
Um dies zu umgehen, müssen die Peers nach gewissen Zeitabständen Aktualisierungen anfordern. So wird das Netzwerk nur belastet, wenn auch ein Update erfragt wird, anstelle dessen, dass die Updates so lange verschickt werden, bis diese bestätigt sind. Dieses Prinzip lässt sich nicht auf das Erstellen von Spielen anwenden, da der Empfänger bei Verlust der Nachricht nicht erfährt, dass er zu einem Spiel hinzugefügt wurde. Hierbei muss vom Empfänger eine Bestätigung versandt werden, wenn eine solche Nachricht erhalten wird. Wenn keine Bestätigung empfangen wird, so werden die Nachrichten erneut versandt, bis eine Bestätigung erfolgt. Dieses Verhalten ist in Abbildung 2.2 dargestellt.

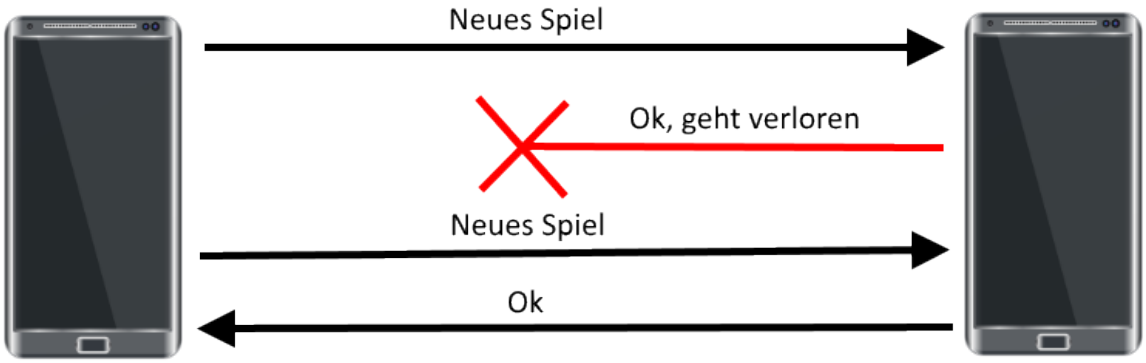

Abbildung 2.2: Verhalten bei Verlust von Nachrichten

Kapitel 3

Implementierung

Die Implementierung erfolgte mithilfe der Entwicklungsumgebung Android Studio [Roh17] in der Programmiersprache Java. Ferner wurde die Opptain-API v1.0.15 verwendet, um die Kommunikation mit der Anwendung *Opptain* zu realisieren. In der Implementierung werden viele Android-Komponenten wie Activities und Servies verwendet, welche in [Roh17] ausführlich beschrieben werden.

3.1 Übersicht

Wie in Abbildung 3.1 sichtbar, setzt sich die Verwaltungs-App *GameLayer* aus mehreren Komponenten zusammen. Diese werden in den folgenden Abschnitten erläutert. Die Verwaltungs-App liegt zwischen *Opptain* und den Spiele-Apps. Hierbei vermittelt sie die Spielstände und synchonisiert diese für die Teilnehmer eines Spiels. Diese Form der Implementierung wurde bewusst gewählt, um die größtmöglichste Flexibilität zu gewährleisten und den Entwicklern der Spiele-Apps möglichst viel Freiraum zu bieten. Der Kontakt zwischen den Apps wird fast ausschließlich mithilfe von Intents realisiert. Intents sind Objekte, welche auf Androidgeräten die Kommunikation zwischen verschiedenen Activities und Apps erlauben [Goof].

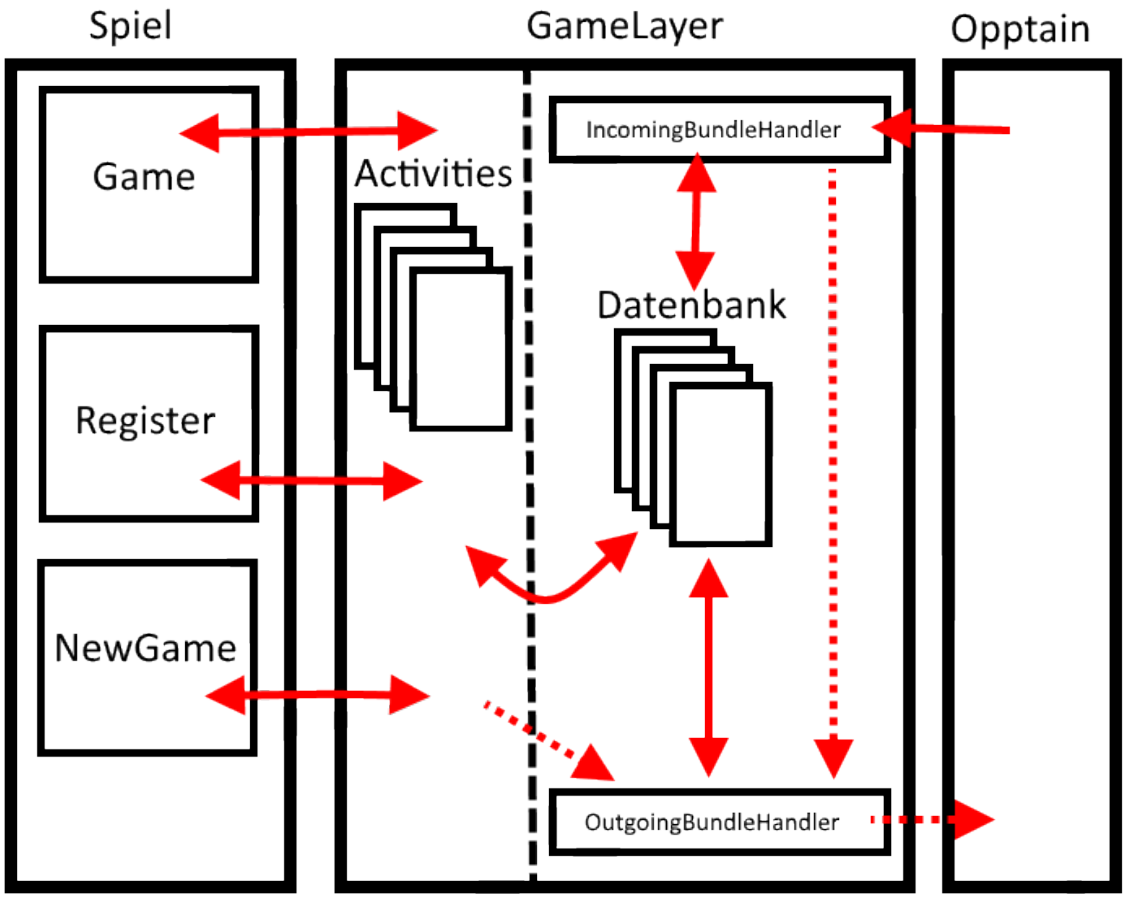

Abbildung 3.1: Übersicht des Konzepts der Implementierung

3.2 Verwaltungs-App

Zunächst gehe ich auf die Verwaltungs-App *GameLayer* ein. Diese setzt sich, wie in Abbildung 3.1 sichtbar, grundlegend aus der Datenbank, ihren Activities und dem Incoming- bzw. OutgoingBundleHandler zusammen. Die einzelnen Komponenten werden in den folgenden Abschnitten beschrieben. Grundsätzlich steuert der Nutzer die App über die Benutzeroberfläche der Activities. Hierbei greifen diese auf die Datenbank zu, in welcher die Informationen über die laufenden Spiele gespeichert werden, um diese darzustellen oder auf Eingabe des Nutzers zu verändern. Die Activities kommunizieren außerdem mit den Spiele-Apps. Das Versenden von Nachrichten als Antwort auf Eingaben realisiert sich durch das Aufrufen des OutgoingBundleHandler. Dies ist ein Service, welcher auf die Datenbank zugreift und Nachrichten versendet. Sein Gegenstück, der IncomingBundleHandler verarbeitet von *Opptain* eingehende Nachrichten. Hierbei modifiziert er auch die Datenbank und kann den OutgoingBundleHandler aufrufen, um das Versenden von Nachrichten anzuordnen.

3.2.1 Beschreibung der Klassen

Im Folgenden werden alle wichtigen Klassen der Implementierung *GameLayer* beschrieben. Genauere Informationen zur Funktionsweise finden sich in der Implementierung selber.

Activities

Die Activites der Verwaltungs-App stellen die Benutzeroberfläche dar und erlauben es, Eingaben zu registrieren und dem Nutzer Daten auszugeben. Wie im AndroidManifest der App definiert, setzt sich die App aus den unten aufgeführten Activities zusammen. Das Android Manifest definiert die Komponenten und andere Metainformationen einer App [Goob].

MainActivity Startactivity der App. Jedes Mal, wenn der Nutzer die App neu startet, wird sie zuerst aufgerufen. Hier muss ab API-Level 23 die Permission, in den externen Speicher zu schreiben, vom Nutzer abgefragt werden, da diese als gefährlich eingestuft ist [Good]. Diese Permission wird benötigt, um zu versendende Nachrichten an *Opptain* weiterzuleiten. In früheren Android-API-Versionen muss diese Permission nicht explizit angefragt werden, wenn sie bereits im Android-Manifest angegeben ist [Gooh]. Ferner startet die Activity im

Hintergrund die ServiceConnection myServiceConnection, um eine Verbindung zu *Opptain* herzustellen. Die Activity bietet dem Nutzer die Möglichkeit an, zu folgenden Activities zu navigieren: SaveGameSelectionActivity, RegisterActivity, ContactActivity, ProfileActivity, ConfigActivity und LoggingActivity.

SaveGameSelectionActivity Die SaveGameSelectionActivity bietet eine Liste mit allen aktiven Spielständen an. Außerdem realisiert sie auch bei Auswahl eines Spielstands den Übergang in die entsprechende Spiele-App. Dabei stellt sie via eines Intents der entsprechenden Spiele-App die Spielstand-Datei bereit und versieht den Intent zusätzlich mit Meta-Informationen zum gewählten Spielstand. Die Activity bietet dem Nutzer einen Übergang zur CreateGameActivity.

CreateGameActivity Die CreateGameActivity lässt den Benutzer einen neuen Spielstand erstellen. Dazu muss der Nutzer Mitspieler und ein Spiel auswählen. Dies geschieht über das Wechseln zur RegisterActivity bzw. ContactActivity. Wenn der Nutzer ein Spiel und Mitspieler gewählt hat, kann er das Spiel erstellen. Nun wird eine leere Datei erzeugt und der Spiele-App wie bei der SaveGameSelectionActivity bereitgestellt. Diese kann einen neuen Spielstand erstellen und in der Datei speichern. Wenn die Spiele-App eine positive Rückmeldung sendet, so erstellt die CreateGameActivity die entsprechenden Datenbankeinträge. Wenn die Rückmeldung negativ ist, so wird die zuvor erstellte Datei wieder gelöscht.

RegisterActivity Die RegisterActivity lässt sich durch einen Intent von außerhalb der App starten. Somit wird es einer fremden App ermöglicht, sich als Spiel zu registrieren. Ferner bietet die Activity die Möglichkeit an, ein Spiel auszuwählen und Informationen über die Auswahl bereitzustellen. Die Activity zeigt alle registrierten Spiele an. Die Einträge der registrierten Spiele lassen sich durch das Halten auf den gewünschten Eintrag wieder löschen.

ContactActivity Die ContactActivity zeigt alle Kontakte an. Sie bietet an, neue Kontakte über die Kontaktliste von *Opptain* auszuwählen und sich mit ihnen bekannt zu machen. Über diese Activity kann außerdem wie in der RegisterActivity die Auswahl eines Kontaktes angefordert werden.

ProfileActivity Hier kann der Nutzer die ihm von *Opptain* zugewiesene DeviceId [Now17] einsehen und seinen Nickname und Profiltext festlegen und ändern. Nickname und Profiltext werden anderen Nutzern angeboten, um den eigenen Client für Menschen lesbar darzustellen. Bevor das Nutzen der App möglich ist, muss hier mindestens ein Nickname angegeben

werden.

ConfigActivity Die ConfigActivity dient als Interface zum Ändern von Einstellungen. Hier lassen sich Konfigurationen für das Verhalten von *GameLayer* festlegen.

LoggingActivity Die LoggingActivity ist eine Activity, welche die gesammelten Daten der App aus der Datenbank liest und darstellt. Hierbei können alle Informationen zu gesendeten oder empfangenen Nachrichten angesehen werden. Zusätzlich ist es möglich, die Datenbank zu exportieren oder zu löschen. Wenn die Datenbank exportiert wird, wird diese im externen Speicherbereich des Geräts gespeichert.

Services

Zusätzlich zu den Activities baut sich *GameLayer* noch aus Services auf. Diese übernehmen Aufgaben, welche dann asynchron bearbeitet werden.

BundleHandler Da der BundleHandler eine abstrakte Klasse ist, ist er weder im Android-Manifest registriert noch wird er instantiiert. Dennoch bietet er den beiden Services IncomingBundleHandler und OutgoingBundleHandler die wichtige Funktion, mit einem einfachem Methodenaufruf OutgoingBundles aus GamePackets zu erstellen und diese an *Opptain* weiterzureichen, damit sie zugestellt werden können. Outgoing- bzw. IncomingBundles sind Klassen, welche von der API von *Opptain* bereitgestellt werden. Sie stellen Objekte dar, welche von *Opptain* versendet bzw. empfangen werden können.

IncomingBundleHandler Der IncomingBundleHandler wird von MyBroadcastReceiver gestartet, wenn ein IncomingBundle von *Opptain* eintrifft. Der IncomingBundleHandler führt dann die durch das Protokoll definierten Interaktionen aus.

OutgoingBundleHandler Der OutgoingBundleHandler wird von verschiedenen Stellen der App gestartet. Von hier aus werden Nachrichten verschickt.

PacketResender Der PacketResender wird durch ein PendingIntent von dem AlarmManager regelmäßig gestartet. Der AlarmManager ist eine Android Systemkomponente, welche es erlaubt, registrierte Intents zu festgelegten Zeitpunkten zu starten [Gooa]. Dieser stellt fest, ob Bundles versendet werden müssen und ordnet dies im entsprechenden Fall auch an.

Der PacketResender wird durch PendingIntents aufgerufen, welche durch die MainActivity beim AlarmManager des Endgeräts hinterlegt werden [Gook]. In der Implementierung wurde die Option gewählt, einen inexakten wiederholenden Alarm zu nutzen. Dieser bietet ab API-Level 19 die Möglichkeit, sich nach beliebigen Zeitintervallen zu wiederholen. Hierbei versucht das Androidsystem, ihn mit anderen Alarmen, welche in einer ähnlichen Zeit ausgeführt werden sollen, zu synchronisieren, um die Gerätelast zu minimieren [Gooc].

3.2.2 Sonstige Klassen

GamePacket Das GamePacket stellt die Basis für die Versendung von Informationen dar. Es beinhaltet mehrere Felder, welche bei Bedarf gefüllt werden können. GamePacket selber implementiert das Interface Serializable, um das Versenden zu vereinfachen.

SaveGame Das SaveGame ist eine Klasse, welche für einen definierten Spielstand sämtliche Informationen aus der Datenbank lädt. Hierbei bietet die Klasse Methoden an, welche das Verwalten von Spielständen vereinfachen und von den Netzwerk-Services dazu genutzt werden, die Ziele für das Versenden von OutgoingBundles festzustellen.

MyServiceConnection MyServiceConnection ist eine Unterklasse von ServiceConnection. Mithilfe dieser Klasse wird eine Verbindung zu einem Service von *Opptain* hergestellt, um die DeviceId des Geräts abzufragen.

DatabaseContract Die Klasse DatabaseContract ist nach der von Google empfohlenen Quellcodepraxis [Gooi] eine statische Klasse, welche den Aufbau der Datenbank definiert.

MyDbHelper Der MyDbHelper ist eine Unterklasse von SQLiteOpenHelper. Hier wird die Datenbank angefordert und verwaltet. Die Klasse speichert hierzu Querys, um die Datenbank zu löschen und wiederherzustellen.

3.3 Versendung eines Packets

Die Versendung eines GamePackets wird in der Klasse BundleHandler realisiert. Um ein GamePacket zu versenden, muss dieses zunächst gepackt werden. Versendete GamePackets

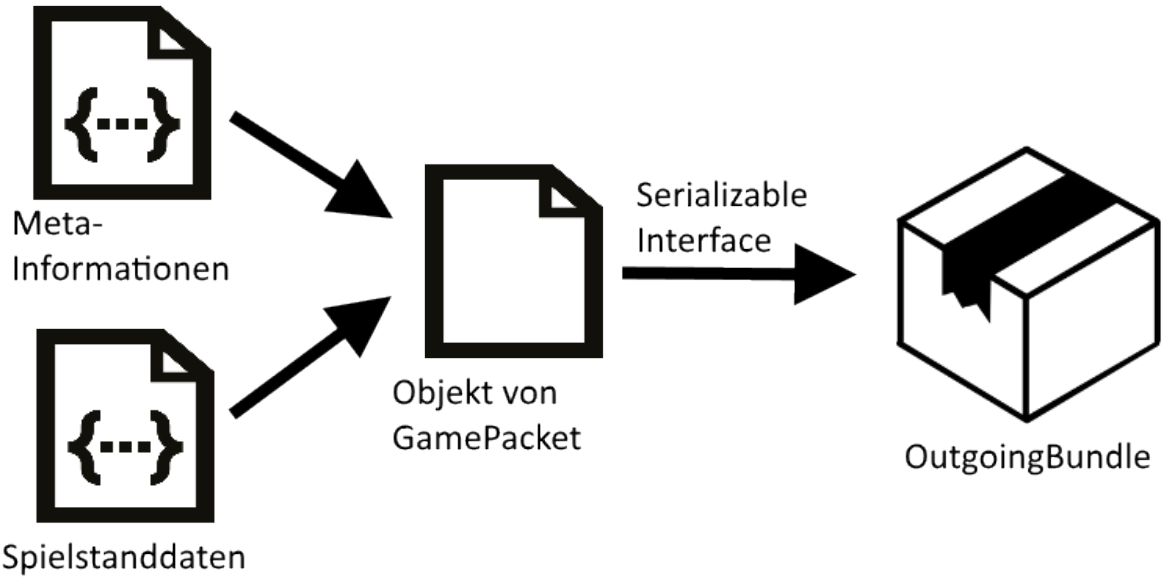

Abbildung 3.2: Packvorgang eines Packets

werden im Folgenden als Packet bezeichnet. Dies ist schematisch in Abbildung 3.2 dargestellt. Wenn ein Packet gepackt wird, so wird zunächst ein Objekt der Klasse GamePacket instantiiert. Dieses wird dann je nach Art des Packets mit den Daten eines Spielstands oder mit Metainformationen zu einem Spielstand oder einem Spieler gefüllt. Ist dies abgeschlossen, so wird das Objekt mithilfe des Serializable-Interface von Java in eine Datei im externen Dateispeicher des Geräts gespeichert. Im Folgenden wird ein OutgoingBundle, welches von der API von *Opptain* bereitgestellt wird, instantiiert. Dieses OutgoingBundle wird mit der zuvor erstellten Datei bestückt und an einen Intent angehängt, welcher dann an *Opptain* gesendet wird. Das Objekt von GamePacket wird als Datei gespeichert und nicht über die nativen Funktionen des OutgoingBundles versendet, da so maximal 1 Megabyte Daten versendet werden könnten [Goog].

3.4 Datenbank

Zur Realisierung der Datenhaltung der Verwaltungs-App wird eine Datenbank genutzt. Android bietet dazu eine interne SQLite Datenbank an. Die Datenbank wird in der Klasse DataBaseContract definiert. Mithilfe des MyDbHelper, welcher eine Unterklasse von SQLiteO-

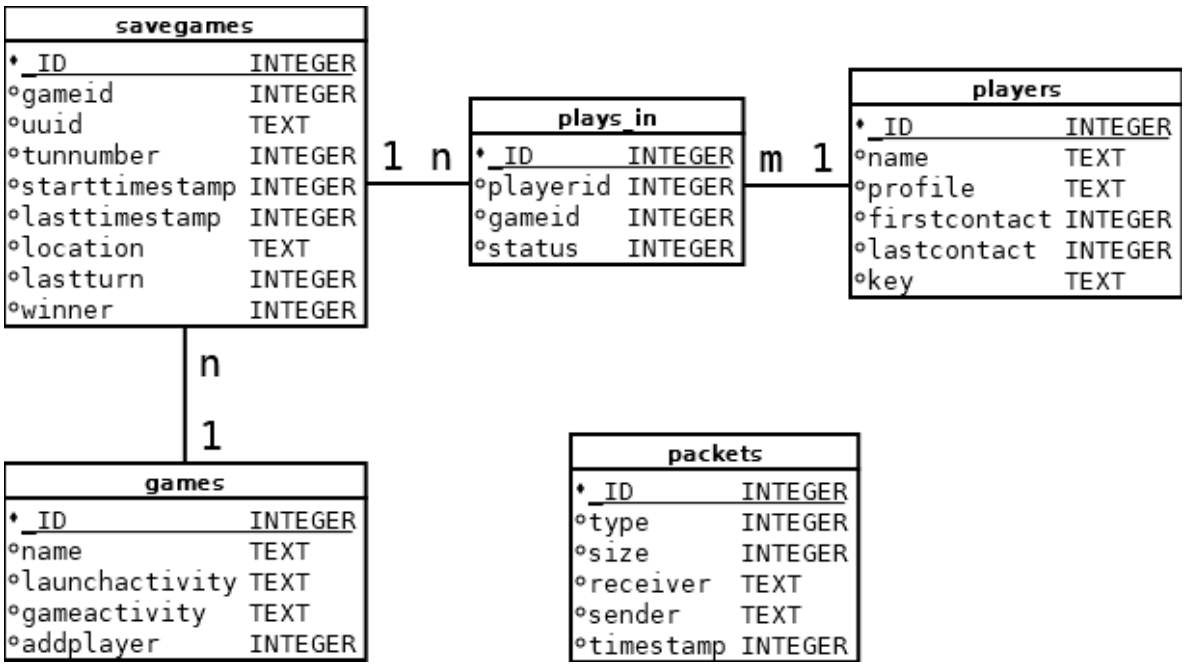

Abbildung 3.3: Entity Relationship Modell der Datenbank

penHelper darstellt, wird die Datenbank verwaltet und der Handle zur Nutzung im Projekt übergeben.

3.4.1 Entitiy Relationship Modell

Das ERM (Entity Relationship Modell) in Abbildung 3.3 beschreibt den Aufbau der Datenbank.

3.4.2 Beschreibung der Elemente

In diesem Abschnitt werden die Spalten der Datenbank beschrieben. Für jede Tabelle gilt, dass die Spalte *_ID* den jeweiligen Primärschlüssel darstellt.

savegames

Die Tabelle *savegames* enthält Informationen über aktuelle Spielstände.

gameid	Fremdschlüssel von Tabelle *games*. Definiert Spiele-App eines Spielstands.
uuid	UUID (Universally Unique Identifier), welche beim Starten des Spiels generiert wird. Dient zur Identifikation des Spielstandes über mehrere Geräte.
turnnumber	Anzahl der gespielten Züge. Steht zu Spielbeginn auf 0. Wird vom spielenden Client nach Beenden eines Zuges erhöht und dient als Identifikation, ob ein erhaltener Spielstand ein Update darstellt.
starttimestamp	Enthält einen Zeitstempel vom Zeitpunkt, an dem der Spielstand erstellt wurde.
lasttimestamp	Enthält einen Zeitstempel des Zeitpunktes, an dem das Spiel zuletzt aktualisiert wurde.
location	Speicherort der Spieldaten auf dem Smartphone.
lastturn	Enthält den Index des Spielers, welcher zuletzt seinen Zug gemacht hat.
winner	Enthält den Index des Spielers, welcher das Spiel gewonnen hat.

players

Die Tabelle *players* enthält Informationen über alle bekannten Spieler.

name	Enthält den gewählten Nickname des Spielers.
profile	Enthält den gewählten Profiltext des Spielers.
firstcontact	Enthält den Zeitstempel des ersten Kontakts mit dem Spieler.
lastcontact	Enthält den Zeitstempel des Zeitpunktes, an welchem die Spielerdaten zuletzt aktualisiert wurden.
key	Enthält die DeviceId, welche dem Spieler von *Opptain* zugewiesen worden ist.

plays_in

Die Tabelle *plays_in* stellt die N:M Beziehung zwischen den Tabellen *players* und *savegames* her.

playerid	Fremdschlüssel der Tabelle *players*.
gameid	Fremdschlüssel der Tabelle *savegames*.
status	Stellt den Status des Spielers in dem Spiel dar. Die verschiedenen Status werden in Kapitel 4.2 definiert.

games

Speichert Informationen über alle registrierten Spiele.

name	Name des Spiels.
launchactivity	Package-Name der Activity, welche das Spiel nutzt, um einen neuen Spielstand zu erstellen.
gameactivity	Package-Name der Activity, welche das Spiel nutzt, um einen Spielstand darzustellen und zu bearbeiten.

packets

Speichert Informationen zu versandten und erhaltenen Packets. Wird zum Debuggen der Anwendung und zu ihrer Auswertung verwendet.

type	Speichert den Typ des gesendeten/empfangenen Packets.
sender	Speichert die DeviceId des Senders.
receiver	Speichert die DeviceId des Empfängers.
timestamp	Speichert den Zeitpunkt des Sendens/Erhaltens des Packets.
size	Speichert die Größe des serialisierten *GamePackets* in Byte.

3.5 Schnittstelle zu Opptain

Wie in Tabelle 3.1 [Ipp15] sichtbar, ist die Verwaltungs-App auf *Opptain* aufgebaut. Sie stellt das Verbindungsstück zwischen Spiele-Apps und *Opptain* beziehungsweise den Schichten unter *Opptain* dar. Diese Kommunikation findet in Form von übergebenen Incoming- bzw. OutgoingBundles statt. Diese enthalten Packets, welche zwischen den verschiedenen Instanzen von *GameLayer* verschickt werden. Die Verwaltungs-App kommuniziert auf drei Wegen mit *Opptain*:

Modell nach OSI	Eigentliche Funktionalität	Bezeichnung
Anwendung	Anwendung	Spiele-App
Anwendung	Anwendung	Verwaltungs-App
Anwendung	Transport	Opptain
Transport	Transport	TCP
Transport	Transport	IP
Transport	Transport	802.11 MAC
Transport	Transport	802.11 PHY

Tabelle 3.1: OSI-Modell [Ipp15]

3.5.1 RemoteService

Beim Start der App wird über die MainActivity eine ServiceConnection gestartet. Diese ServiceConnection verbindet sich mit einem Service von *Opptain*, um so die Verwaltungs-App bei *Opptain* zu registrieren und die DeviceId des Nutzers abzufragen. Letztere wird in den SharedPreferences [Gooj] zur späteren Verwendung gespeichert. Ist dies erfolgreich, so schließt sich die Verbindung wieder.

3.5.2 BroadcastReceiver

Über den BroadcastReceiver werden durch den Intent-Filter festgelegte Broadcasts von *Opptain* angenommen. Dies geschieht, wenn ein für die App prädestiniertes Bundle bei *Opptain* eingetroffen ist. Der BroadcastReceiver startet dann direkt den IncomingBundleService, mit welchem die eintreffenden IncomingBundles asynchron weiterverarbeitet werden.

Name	Typ	Beschreibung
launch	String	Beinhaltet den Package-Name der Aktivität, welche neue Spielstände erstellen kann.
game	String	Beinhaltet den Package-Name der Aktivität, welche das Spiel für den Nutzer darstellt.
name	String	Beinhaltet den Namen, unter welchem die App das Spiel darstellen soll.
addplayer	int	Gibt an, ob einem laufenden Spiel Spieler hinzugefügt werden sollen können. 0 steht für false, 1 für true.

Tabelle 3.2: Felder des Intents zur Registration

3.5.3 Intents

Wie in Abschnitt 3.3 beschrieben, werden fertige Packets an ein OutgoingBundle gehängt und mithilfe eines Intents an *Opptain* gesendet.

3.6 Schnittstelle zur Spiele-App

Die Kommunikation mit der Spiele-App beschränkt sich ausschließlich auf die Übergabe von Intents. Hierbei registriert sich die App via eines Intents für die RegisterActivity. Danach kann die App über das Userinterface der Verwaltungs-App angesteuert werden. Tabelle 3.2 erläutert die akzeptierten Felder des Intents, welcher zur Registration verwendet wird.

3.6.1 Spiele erstellen

Wenn der Nutzer ein neues Spiel erstellen will, so wird eine Activity der Spiele-App gestartet, in welcher der Nutzer alle Einstellungen zum Spielstand festlegen kann, welche von der jeweiligen Spiele-App vorgesehen sind. Die Verwaltungs-App übergibt hierbei eine URI (Uniform Resource Identifier) für eine Datei, in welche der erstellte Spielstand geschrieben werden soll. Die Bereitstellung einer URI über die FileProvider-API ermöglicht es, intern gespeicherte Dateien für fremde Apps zugänglich zu machen [Gool]. Wenn die Erstellung eines Spiels erfolgreich war, muss über einen Rückgabe-Intent RESULT_OK gesendet werden, indem der Rückgabecode entsprechend gesetzt wird. Dieser ist im Android-System als

Name	Typ	Beschreibung
playernumber	int	Beinhaltet die Anzahl an Spielern, welche für das Spiel vorgesehen sind.
data	URI	Beinhaltet die URI zu der Datei, wo der Spielstand gespeichert werden soll.

Tabelle 3.3: Felder des Intents zur Erstellung eines Spielstands

Name	Typ	Beschreibung
myTurn	boolean	Ist true, wenn Nutzer des Smartphones gerade am Zug ist.
activePlayer	String	Beinhaltet die DeviceId des Spielers welcher gerade am Zug ist.
turn	int	Beinhaltet den Index des aktuellen Zugs.
data	URI	Beinhaltet die URI der Datei, in welcher der Spielstand gespeichert ist.

Tabelle 3.4: Felder des Intents zum Starten eines Spielstands

Integer kodiert [Gooe]. Tabelle 3.3 beschreibt die Felder des Intents.

3.6.2 Spiel spielen

Wie bei dem Erstellen eines Spiels startet die Verwaltungs-App via eines Intents die Spieleactivity des entsprechenden Spiels. Hier kann der Nutzer dann seine Interaktionen vornehmen und den Zug beenden. Tabelle 3.4 beschreibt die Felder des Intents.

Wenn der Spieler nun einen Zug beendet hat, so muss die App den aktuellen Spielstand in die Datei schreiben und den Intent wieder zurückgeben. Hierbei ist vor allem wichtig, dass der Status RESULT_OK gesetzt wird. Ferner können dem Rückgabe-Intent die Felder aus Tabelle 3.5 hinzugefügt werden.

Name	Typ	Beschreibung
winner	int	Gibt den Index des gewinnenden Spielers an.
out	String[]	Gibt die DeviceIds der ausgeschiedenen Spieler an.

Tabelle 3.5: Felder des Rückgabe-Intents nach Beendigung eines Zuges

3.6.3 Implementierung von Spielen

Die Spiele-App muss, wie oben bereits erwähnt, die Schnittstellen abdecken. Speziell muss sich die App registrieren können und dabei die richtigen Daten mit dem Intent verschicken. Es liegen folgende Interaktionen in der Verantwortung der Spiele-App:

1. Die Spiele-App muss die oben beschriebenen Intents der Verwaltungs-App empfangen und mit diesen umgehen können. Um die Intents empfangen zu können, muss ein Intent-Filter für die entsprechenden Activities angegeben werden. Dieser muss den MIME-Typ *application/octet-stream* akzeptieren.

2. Die Spiele-App muss empfangene Spielstände lesen und erneut schreiben können. Es wird das Serializable Interface empfohlen.

3. Die App muss die Spieler an ihren DeviceIds erkennen und verwalten können.

3.7 Implementierte Spiele

Im Rahmen der Arbeit wurden zwei Spiele implementiert, welche auf Basis von *GameLayer* arbeiten. *TicTacToe* ist das klassische *Drei gewinnt*. Zweitens *AndroidStrategy*, ein generisches Strategiespiel, an dem unbegrenzt viele Peers teilnehmen können. Ferner ist die Größe des Feldes wählbar, wodurch sich auch die Größe der Speicherdatei und die Länge der Runden beeinflussen lässt. In den Anlagen finden sich Anleitungen zu den Spielen.

3.7.1 TicTacToe

In *TicTacToe* können zwei Spieler gegeneinander spielen. Die Spieler können ihre Zeichen (X und O) abwechselnd auf dem 3x3-Feld platzieren. Der Spieler, der es schafft, drei seiner Zeichen senkrecht oder waagerecht in einer Reihe zu platzieren, gewinnt das Spiel. Abbildung 3.4 zeigt das Spielfeld von *TicTacToe*.

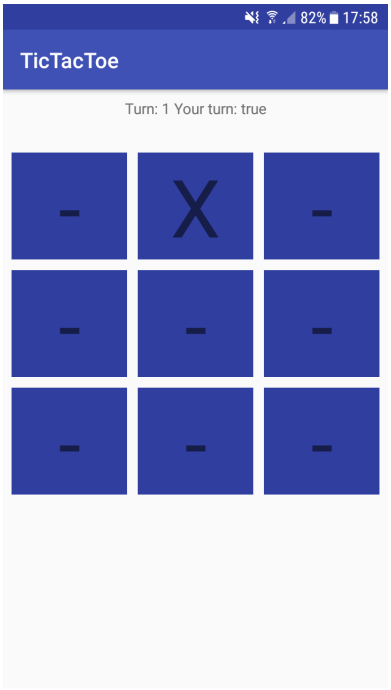

Abbildung 3.4: Spielfeld von *TicTacToe*

3.7.2 AndroidStrategy

Im Spiel *AndroidStrategy* können beliebig viele Spieler auf einer quadratischen Karte gegeneinander spielen. Die Karte besteht aus passierbaren und unpassierbaren Feldern. Auf den Feldern kann auch noch eine der Ressourcen Holz oder Stein liegen, welche von den Spielern eingesammelt werden können. Zu Beginn des Spiels wird eine Karte zufällig generiert und jeder Spieler erhält einen Bauarbeiter auf einem zufälligen Feld, welcher es ermöglicht, auf der Karte Gebäude zu errichten. Spieler können ihren Truppen befehlen, andere Truppen oder Gebäude anzugreifen. Falls ein Spieler nach einem Zug keine Truppen und keine Gebäude mehr auf der Karte haben sollte, so scheidet er aus. Gewonnen hat der Spieler, welcher als letztes noch spielt. Abbildung 3.5 zeigt einen Ausschnitt eines möglichen Spielfelds von *AndroidStrategy*.

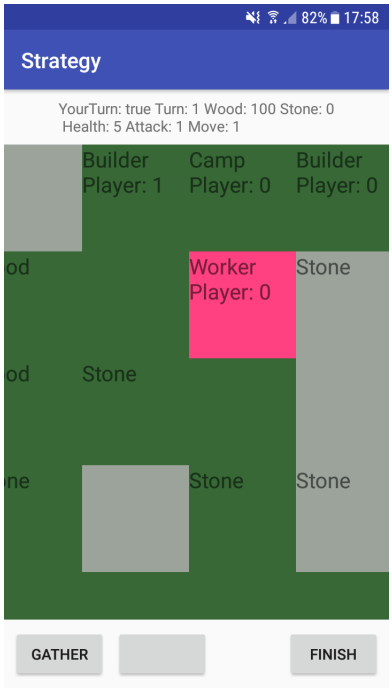

Abbildung 3.5: Ausschnitt des Spielfelds von *AndroidStrategy*

Kapitel 4

Das Protokoll

In diesem Kapitel wird das auf Basis der Anforderungen entwickelte Protokoll erläutert. Die Implementierung des Protokolls wird in den Klassen OutgoingBundleHandler, IncomingBundleHandler und PacketResender vorgenommen.

4.1 Aufbau eines Packets

Wie im Abschnitt 3.2.2 des Kapitels Implementierung bereits beschrieben, wird die Klasse GamePacket für die Kommunikation zwischen den Clients verwendet. Tabelle 4.1 erläutert die beschreibbaren Felder eines Objektes von GamePacket.

4.2 Status eines Spielers

Jedem Spieler wird ein Status zugewiesen. Der Status bestimmt, wie die Peers mit dem Spieler interagieren. Tabelle 4.2 erläutert die Status, welche die Spieler innerhalb eines Spielstandes innehaben können.

Name	Typ	Beschreibung
packetmode	int	Kodiert den Typ des Packets. Die Packet-Typen werden in Abschnitt 4.3 erläutert.
payload	byte[]	Beinhaltet Zustand des Spiels.
nickname	String	Beinhaltet den Namen des sendenden Spielers.
profile	String	Beinhaltet den Profiltext des sendenden Spielers.
turn	int	Beinhaltet den Index des verschickten Zugs.
uuid	String	Beinhaltet UUID des Spiels. Die UUID wird beim Erstellen eines Spiels generiert und dient im Netzwerk als Identifikation.
gamename	String	Beinhaltet den Namen des gespielten Spiels.
playerDeviceId	String[]	Beinhaltet die DeviceIds aller Spieler.
playerStatus	int[]	Beinhaltet die Status der Spieler.
winner	int	Beinhaltet den Index des gewinnenden Spielers.
lastTurn	int	Beinhaltet den Index des Spielers, welcher den letzten Zug gemacht hat.

Tabelle 4.1: Felder eines GamePacket Objekts

4.3 Erläuterung der Interaktionen

Im Folgenden werden die Interaktionen des Protokolls erläutert. Zunächst wird auf die gesamte Interaktion beschrieben, dann die beteiligten Packet-Typen sowie die erwartete Reaktion auf deren Erhalt.

4.3.1 Kennenlernen von Peers

Um Peers kennenzulernen, wird an diese ein HELLO-Packet geschickt. Dieses wird nicht erneut versandt, da kein Erhalt garantiert wird. Wenn die App auf dem anderen Smartphone nicht installiert ist, so liefert *Opptain* das Packet nicht aus. Somit kann auch keine Antwort vom anderen Client erwartet werden. Die Packets enthalten Informationen über den Sender. Hier: Nickname und Profiltext.

Numerischer Wert	Name	Beschreibung
-1	RESEND_NEW_GAME	Der Spieler ist zu einem Spiel hinzugefügt worden, hat allerdings noch nicht geantwortet. Diesen Status besitzt ein Mitspieler nur bei dem Spieler, von dem er dem Spiel hinzugefügt worden ist. Wenn der hinzufügende Spieler ein UPDATE_GAME oder NEW_GAME-Packet versendet, so wird der Spieler im Packet mit dem Standardstatus DEFAULT versehen. An Spieler mit diesem Status werden die NEW_GAME-Packets regelmäßig neu versendet. Wenn der Spieler mit NEW_GAME_OK antwortet, so wird sein Status auf PLAYING gesetzt.
0	DEFAULT	Standardstatus. Keine Interaktion.
1	PLAYING	Status eines Spielers, der aktuell spielen kann. Er wird bei Berechnung des nächsten Spielers berücksichtigt und erhält REQUEST_GAME-Anfragen.
2	OUT	Der Spieler hat das Spiel verloren. Er kann zwar nicht mehr ziehen, aber erhält trotzdem noch Updates über den aktuellen Verlauf des Spiels.
3	DECLINED	Der Spieler hat das Spiel verlassen. Er erhält keinerlei Packets mehr.

Tabelle 4.2: Status des Spieler in einem Spiel

HELLO

Wenn das empfangene Packet mit HELLO bezeichnet ist, so speichert die App zuerst die Informationen über den Sender in der Datenbank, wenn zuvor keine Informationen über ihn enthalten sind. Falls schon ein Eintrag enthalten ist, so wird dieser aktualisiert.
Als Antwort verschickt die App ein HELLO_OK-Packet.

HELLO_OK

Wenn ein HELLO_OK-Packet erhalten wird, so werden die erhaltenen Informationen wie bei HELLO behandelt. Der Unterschied ist, dass auf HELLO_OK keine Antwort erfolgt.

4.3.2 Erstellen von Spielständen

Wenn ein Spieler entscheidet, einen Spielstand zu erstellen, so wird ein NEW_GAME-Packet an alle Mitspieler versendet. Dieses wird an die Clients so lange erneut gesendet, bis diese den Erhalt mit NEW_GAME_OK bestätigen. Das NEW_GAME-Packet enthält alle Meta-Informationen und den Spielzustand des gestarteten Spielstands. Also sind konkret die Felder *uuid*, *turn*, *lastTurn*, *gamename*, *winner*, *playerDeviceId*, *playerStatus* und *payload* gefüllt. Das NEW_GAME_OK-Packet enthält nur die UUID des Spiels, um das Spiel identifizieren zu können.

NEW_GAME

Wenn ein NEW_GAME-Packet empfangen wird, wird zuerst festgestellt, ob die gewünschte Spiele-App registriert ist. Wenn die Spiele-App nicht vorhanden ist, wird mit DECLINE geantwortet. Wenn sie vorhanden ist, werden alle benötigten Einträge in der Datenbank vorgenommen. Wenn bereits ein Spiel mit derselben UUID vorhanden ist, so wird direkt mit NEW_GAME_OK bestätigt. Außerdem wird für alle bisher unbekannten Mitspieler ein Eintrag in der Datenbank erstellt – diese erhalten dann ein HELLO-Packet. Danach wird eine Datei für den Payload angelegt und dieser darin gespeichert. Wenn alles erfolgreich eingerichtet wurde, wird mit NEW_GAME_OK geantwortet. Wenn nicht, mit DECLINE.

NEW_ GAME_OK

Wenn ein NEW_GAME_OK-Packet empfangen wird, so wird der Status vom Sender von RESEND_NEW_GAME auf PLAYING gesetzt. Es erfolgt keine Antwort.

4.3.3 Synchronisierung von Spielständen

Die Synchronisierung eines Spielstands wird durch das Senden einer REQUEST_GAME-Anfrage realisiert. Der Empfänger antwortet dann mit seinem aktuell gespeicherten Zustand des jeweiligen Spiels.

UPDATE_GAME

Das UPDATE_GAME-Packet enthält dieselben Felder wie das NEW_GAME-Packet. Wenn ein UPDATE_GAME-Packet eintrifft, so wird zuerst überprüft, ob ein Spiel mit der enthaltenen UUID vorhanden ist und ob der Sender überhaupt in dem spezifischen Spiel mitspielt. Wenn das Spiel nicht vorhanden ist, so wird das Packet verworfen. Wenn das Spiel vorhanden ist, so werden die Spieler und ihre Status angeglichen. Hierbei kann der Statuscode eines Spielers sich nur erhöhen. Wenn die Version, also entsprechend das Feld *turn* des empfangenen Packets höher ist, als das des gespeicherten Eintrages, so wird die gespeicherte Datei durch die neu erhaltene ersetzt. In diesem Fall wird auch die Datenbank aktualisiert.

REQUEST_GAME

Das REQUEST_GAME-Packet enthält nur die UUID des Spiels. Der sendende Client fordert hierbei die Version des Spielstands an, welche der Empfänger hält. Als Antwort wird ein UPDATE_GAME-Packet erwartet. Wenn das Spiel nicht vorhanden ist, ist die Antwort DECLINE. Dies ermöglicht, dass ein Spieler, welcher aus einem Spiel ausgetreten ist, also keinen Eintrag mit der UUID mehr führt, auch aus den Listen der anderen Spieler gelöscht wird (siehe Abschnitt 4.3.4). Wenn ein REQUEST_GAME-Packet erhalten wird, ein Spiel mit der UUID existiert, aber der sendende Spieler nicht in dem entsprechenden Spiel mitspielt, wird keine Antwort gesendet.

4.3.4 Austreten aus einem Spiel

Das Austreten aus einem Spiel kann aus vielen Gründen nötig werden. Das Spiel kann vom Nutzer über die Benutzeroberfläche gelöscht werden. Dabei werden alle den Spielstand betreffenden Einträge aus der Datenbank gelöscht. Dadurch wird garantiert, dass später erhaltene Packets, welche zu dem Spielstand gehören, mit einem DECLINE-Packet beantwortet werden.

DECLINE

Das DECLINE-Packet enthält nur die UUID eines Spiels. Der Sender fordert dabei den Empfänger dazu auf, seinen Status auf DECLINED zu setzen und so keinerlei Packets mehr über das Netzwerk des definierten Spiels zu empfangen.

4.4 Neusenden von Packets

Da das Protokoll dazu entworfen wurde, in einem Netzwerktyp zu arbeiten, in welchem Verzögerungen oder Packetverlust wahrscheinlich sind, müssen Packets unter Umständen neu gesendet werden. Dies trifft vor allem solche Interaktionen, welche eine Bestätigung benötigen – in diesem Fall NEW_GAME und UPDATE_GAME. Um dies zu umgehen, wird im Falle UPDATE_GAME das Packet initial bei Abschluss des Zugs der ausgewählten Versendungsstrategie entsprechend versandt. Es erfolgt aber keine Neusendung. Alle Clients können nun nach der vergangenen Zeit, welche in der Klasse ProtocolContract definiert ist, nach Erhalt des letzten Updates, REQUEST_GAME-Packets an ihren Vorgänger bzw. an Vorgänger und Nachfolger (Im Falle des Zweifach-Gezielt-Modus) oder an alle Teilnehmer (Im Falle des BROADCAST-Modus) senden. Dieser wird dann seinen aktuellen Stand mitteilen.
Im Falle von NEW_GAME ist dieses Problem nicht so einfach zu beseitigen. Hier kann der Client, welcher das Packet nicht erhalten hat, nicht wissen, dass er zu einem Spiel hinzugefügt worden ist. Also muss der sendende Client das Packet erneut senden, bis der Erhalt des Packets mit NEW_GAME_OK oder DECLINE bestätigt wird. NEW_GAME-Packets werden, wie im ProtocolContract definiert, regelmäßig an alle Spieler gesendet, welche den Status RESEND_NEW_GAME haben. Die durch die Klasse ProtocolContract gebotene Stan-

dardkonfiguration kann mithilfe der ConfigActivity verändert werden. Dies schließt die verwendete Versendungsstrategie, den Intervall zwischen REQUEST_GAME und NEW_GAME Versendung und die minimale Zeit nach der letzten Aktualisierung eines Spielstandes, bis eine REQUEST_GAME-Anfrage erfolgen kann, ein.

4.5 Hinzufügen von Spielern in laufendem Spiel

Wenn eine Spiele-App es vorsieht, so kann ein Nutzer, wenn er am Zug ist, einen Spieler hinzufügen. Hierbei wird der ausgewählte Spieler mit Status RESEND_NEW_GAME in seiner Spielerliste hinzugefügt. Und ihm wird ein NEW_GAME-Packet gesandt. Wenn der Spieler nicht reagiert, so steht es dem einladenden Spieler frei, seinen Zug durchzuführen. Dann werden alle Spieler mit dem Status RESEND_NEW_GAME auf DECLINED gesetzt. Entsprechend darf erst gezogen werden, wenn der neue Spieler entweder dem Spiel hinzugefügt ist oder nicht erreicht werden kann.

Kapitel 5

Auswertung

5.1 Apps

5.1.1 Testaufbau

Neben Tests während der Entwicklung wurden auch drei Tests im Rahmen der Auswertung durchgeführt. Die Logs dieser Tests finden sich im Anhang. Die Aufbauten der Tests werden im Folgenden beschrieben.

1. Der Test wurde mit der Verwaltungs-App und der Spiele-App *TicTacToe* durchgeführt. Die Version von *Opptain* war b6b212b vom 11. August 17:24. Der Test wurde auf einem Motorola RAZR i und einem Samsung Galaxy A3 2016 durchgeführt.
   Im Rahmen des Tests wurden die Geräte über die Verwaltungs-App bekannt gemacht. Danach wurden 5 Spielstände von TicTacToe durch das Samsung-Gerät eröffnet und bis zum Gewinnen eines Spielers mit willkürlichen Zügen gespielt. Die Spiele wurden in vier Fällen im 7. und in dem übrigen im 5. Zug beendet. Es wurde die Standardkonfiguration der Verwaltungs-App verwendet.

2. Der Test wurde mit der Verwaltungs-App und der Spiele-App *AndroidStrategy* durchgeführt. Die genutzte Version von *Opptain* war 3e3af69a vom 13. Oktober um 8:35. Der Test wurde mit 3 Huawei Y3, 1 Samsung A3 2016, 1 Motorola Razr i und einem Samsung J5 6 durchgeführt. Hierbei wurden im Verlauf von ca. 2 Stunden 8 Runden

Packetmodus	Anzahl der Packets	Durchschnittliche Speichergröße in Byte
REQUEST_GAME	212	289
NEW_GAME*	45	˜8977
HELLO*	14	˜263
HELLO_OK*	24	˜263
UPDATE_GAME*	290	˜9033
NEW_GAME_OK	75	289

*- Die Größen einiger Packet-Typen hängen von den versandten Daten ab.

Tabelle 5.1: Gemessene Packetgrößen der Verwaltungs-App

gespielt. Die Züge wurden in unregelmäßigen Abständen durchgeführt, um ein möglichst realitätsnahes Szenario zu bieten. In den Anlagen findet sich ein Protokoll, in welchem die Reihenfolge der Züge und ihr Zeitpunkt genau beschrieben sind. Es wurde die Standardkonfiguration der Verwaltungs-App verwendet.

3. Der Test wurde mit der Verwaltungs-App und der Spiele-App *AndroidStrategy* durchgeführt. Die genutzte Version von *Opptain* war 3e3af69a vom 13. Oktober um 8:35. Der Test wurde mit 3 Huawei Y3 und einem Samsung A3 2016 durchgeführt. Bei diesem Test wurde nach der Erstellung des Spiels, nur nachdem alle Clients von dem Zug erfahren haben, ein neuer gespielt. Hierbei wurde durch die verschiedenen Verteilungsstrategien gewechselt. Es wurden insgesamt 16 Runden gespielt. Die Konfiguration weicht von der Standardkonfiguration insofern ab, dass die Abstände der REQUEST_GAME-Anfragen auf 5 Minuten (300.000 ms) und das Mindestalter eines Spielstands, bis eine solche Anfrage erfolgt, auf 0 gestellt wurde.

Packetgröße

Im Verlauf der Tests wurden alle versandten Packets mitgeschrieben. Hieraus lässt sich eine durchschnittliche Größe der versandten Packets ermitteln. Die ermittelten Werte werden in Tabelle 5.1 dargestellt.

Spielname	Einstellungen	Speichergröße in Byte
TicTacToe	-	211
AndroidStrategy	10x10, 2 Spieler	2983
AndroidStrategy	20x20, 4 Spieler	9163
AndroidStrategy	100x100, 10 Spieler	202203

Tabelle 5.2: Gemessene Speichergrößen der Spiele-Apps

5.1.2 Spiele-Apps

Dateigröße

Da die Dateien, welche den Spielzustand definieren, für die beiden Spiele *TicTacToe* und *AndroidStrategy* mithilfe des Serializable-Interfaces von Java in eine versendbare Form gebracht werden, kann man ihre Größe unmittelbar nach der Erstellung messen. Die Größe kann sich im Verlauf des Spiels ändern, da Felder der serialisierten Objekte mit neuen Werten oder Objekten gefüllt werden können.

In der Tabelle 5.2 wird die Größe, welche auf einem Galaxy Nexus 5 in der VM von Android Studio gemessen wurde, dargestellt. Die Übertragung der Spielstände hat bei allen aufgezählten Konfigurationen in den Tests keine Komplikationen verursacht.

5.2 Definition der Topologie

Um über die Eigenschaften der Topologien zu diskutieren, ist es nötig, diese formal zu definieren. Im Folgenden gelte:

C	=	Menge der mit dem Spiel assoziierten Clients.
C_i	=	i. Client aus dem Spiel.
$C_{P,O}$	=	Menge der Clients aus C, welche den Status PLAYING oder OUT haben.
$t(C_1, C_2)$	=	Minimal benötigte Zeit für die Übertragung eines Packets von C_1 zu C_2.
$u(C_1, C_2)$	=	Minimal benötigte Zeit für die Übertragung einer Aktualisierung von C_1 zu C_2.
n	=	Anzahl der Clients im Spiel.
$vor(C_i)$	=	Vorgänger des Clients C_i aus $C_{P,O}$ nach der Zugreihenfolge.
$meg(C, C_x)$	=	Zeit, welche ein Netzwerk mit Einfach-Gezielt-Konfiguration benötigt, um jeden Spieler aus $C_{P,O}$ zu aktualisieren, wenn C_x der Client ist, von dem die Aktualisierung ausgeht.
$mzg(C, C_x)$	=	Zeit, welche ein Netzwerk mit Zweifach-Gezielt-Konfiguration benötigt, um jeden Spieler aus $C_{P,O}$ zu aktualisieren, wenn C_x der Client ist, von dem die Aktualisierung ausgeht.
$mbg(C, C_x)$	=	Zeit, welche ein Netzwerk mit Broadcast-Konfiguration benötigt, um jeden Spieler aus $C_{P,O}$ zu aktualisieren, wenn C_x der Client ist, von dem die Aktualisierung ausgeht.

5.2.1 Darstellung als Graph

Mit diesen Voraussetzungen lässt sich ein ungerichteter Graph (V, E) erstellen. Hierbei ergibt sich die Menge der Knoten aus der Menge

$$V = C_{P,O}$$

da für die Kommunikation nur die Clients von Interesse sind, welche auch die Information erhalten müssen. Die Kanten sind entsprechend die Verbindungen:

$$E = \{(c_1, c_2) \mid c_1, c_2 \in C_{P,O}\}$$

Das Gewicht der Kanten ergibt sich entsprechend aus der Gewichtungsfunktion

$$u(C_1, C_2)$$

welche oben definiert ist. Wenn die Übertragung einer Aktualisierung auf Grund der TTL (Time to Live) der Packets nicht möglich ist, so wird die Kante entfernt.

5.2.2 Bedeutung

Ist nun ein solcher Graph erstellt, so stellt er ein virtuelles Konstrukt dar und spiegelt nicht unbedingt die wahre Beschaffenheit des Netzwerkes wieder, da Clients, welche nicht im Spiel mitspielen, als Zwischenknoten fungieren können, um die versandten Packets zu vermitteln. So stellt dieser Graph nur das Teilnetzwerk dar, in welchem ein Spiel stattfindet und zeigt, welche Spieler innerhalb des Netzwerks des Spiels untereinander kommunizieren können und welche nicht.

5.3 Ungünstige Topologien

Wenn nun ein Spielstand durch einen Peer eröffnet wird, kann es dazu kommen, dass durch Peerbewegung oder direkt bei der Erstellung eine ungünstige Topologie entsteht, welche die Spielgeschwindigkeit verringert. In Extremfällen kann dies sogar dazu führen, dass das Spiel unterbrochen wird, bis sich der Umstand auflöst.

5.3.1 Einfacher Bruch in der Ringtopologie

Der Spieler, welcher das Spiel eröffnet, legt für das Netzwerk eine Topologie fest: So werden zumindest im gezielten Modus die Packets immer entsprechend der Zugreihenfolge verschickt. Ein Beispiel für ein funktionales Netzwerk findet sich in Abbildung 5.1. Spieler 1 hat das Spiel eröffnet und jeder Knoten erreicht seinen Nachfolger. In den Beispielen wurde zur Vereinfachung davon ausgegangen, dass das Gewicht der beiden Kanten zwischen zwei Knoten gleich ist. Bei der Erstellung selber kann eine ungünstige Topologie entstehen, wenn

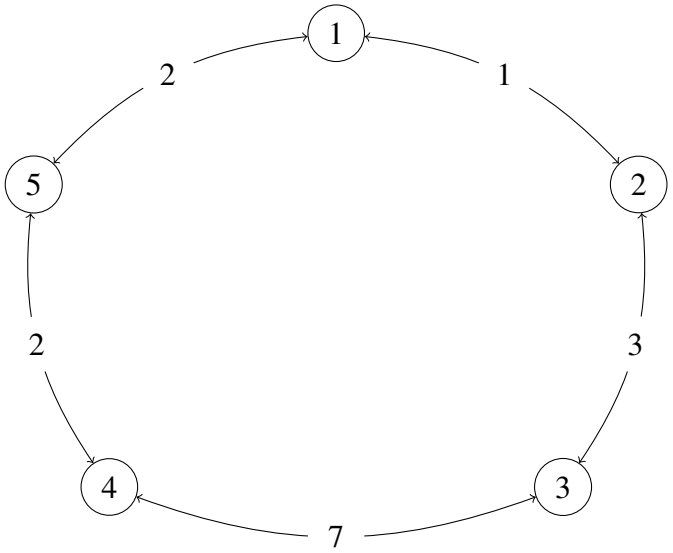

Abbildung 5.1: Ausgangstopologie

nachfolgende Spieler im Netzwerk sich aufgrund der TTL nicht erreichen können. Ein Beispiel dafür ist Abbildung 5.2. Hier können alle Spieler, bis auf Spieler 3 und Spieler 4, ihre Vorgänger beziehungsweise Nachfolger erreichen. Solche Fehlstellungen können außerdem durch die Veränderung des Netzwerks entstehen. Wenn Spieler 3 nun seinen Zug gemacht hat, besitzt er einen Datensatz, in welchem Spieler 4 an der Reihe ist. Spieler 3 fragt regelmäßig den Spielstand von 2 an, dieser ist allerdings veraltet und wird verworfen. Spieler 4 fragt regelmäßig nach einer Aktualisierung durch Spieler 3, diese Anfragen gehen aber verloren. Dies setzt sich so lange fort, bis Spieler 4 einen Spielstand hält, in welchem er am Zug ist. Die Ring-Topologie ist somit durchbrochen und nicht mehr funktionsfähig, bis Spieler 3 Spieler 4 aktualisiert, kann das Spiel nicht weitergehen. Ein Nebeneffekt ist, dass Spieler hinter dem Bruch keine Updates mehr erhalten und nicht über den Fortgang des Spiels informiert werden.
Um dieses Problem zu lösen, gibt es mehrere Ansätze:

Broadcast-Modus Durch das Einschalten des Broadcast Modus bei den Teilnehmern würde die Nachricht auch entgegen der Reihenfolge reisen können. Dadurch könnte im Beispiel der Abbildung 5.2 das Update von Spieler 3 zu Spieler 2 zu Spieler 1 zu Spieler 5 und dann zu Spieler 4 reisen. Dieser könnte seinen Zug dann machen. Zusätzlich könnte dieser Ansatz sehr schnell sein, wenn zwischen den Knoten des Graphen noch andere Kanten existieren würden als nur die in Abbildung 5.2 gezeigten.

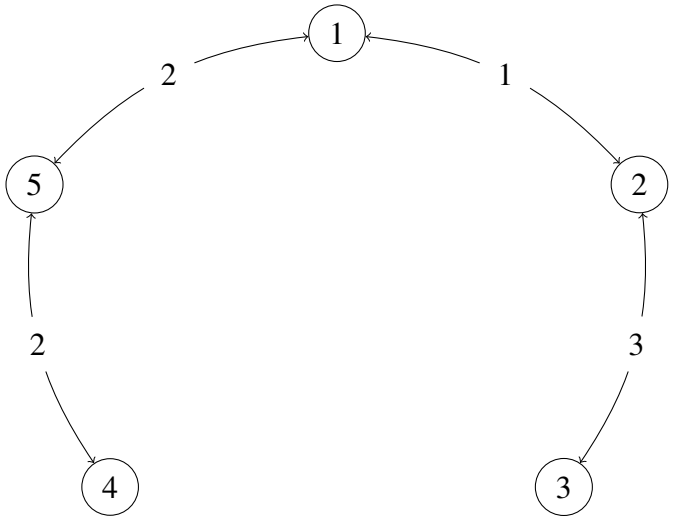

Abbildung 5.2: Bruch im Ring

Anfrage an Nachfolger Wenn die Clients ihre Anfragen nicht nur entgegengesetzt des Ringes verschicken würden, sondern auch entlang des Ringes, so würde sich der aktuelle Spielstand wie im Broadcast-Modus verteilen und das Spiel könnte weitergehen. Hierbei würde sich die Anzahl der versandten Packets in etwa auf das doppelte erhöhen und als Nebeneffekt würden im Falle, dass der Ring nicht gebrochen ist, auch die aktuellen Züge schneller bei Peers ankommen, welche nicht als nächstes am Zug sind. Dieses Verhalten wird durch die Zweifach-Gezielt-Strategie realisiert.

Vermeidung von solchen Topologien Wenn der erstellende Peer Informationen über die Topologie des Netzwerkes hätte bzw. die Entfernungen zwischen den beteiligten Peers abschätzen könnte, so würde es möglich sein, eine Topologie zu errichten, in der die Wahrscheinlichkeit auf solche Fehlstände sehr gering wäre. Da sich die Peers in einem Opportunistischen Netzwerk allerdings ständig bewegen, wäre dies keine zuverlässige Lösung.

Hierbei bietet sich die Aktivierung des Zweifach-Gezielt-Modus am ehesten an, da sich die Menge an versendeten Packets zwar erhöhen würde, doch nicht so stark wie bei der Aktivierung des Broadcast-Modus. Zusätzlich kann sich der Fehlstand durch Veränderungen im Netzwerk mit der Zeit auflösen.

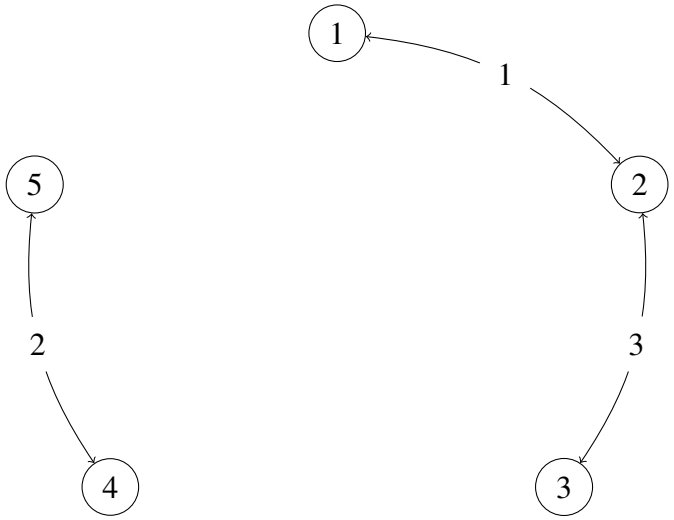

Abbildung 5.3: Doppelter Bruch im Ring

5.3.2 Mehrfacher Bruch in der Ringtopologie

Wenn, wie in Abbildung 5.3 gezeigt, ein doppelter Bruch in der Topologie entstehen würde, so könnte dieses Problem im Gegensatz zum einfachen Bruch nicht mehr mit dem Zweifach-Gezielt-Modus gelöst werden. Um diese Fehlstellung zu beseitigen, muss zuerst der Broadcast-Modus genutzt werden. Ferner müssen mehr Kanten zwischen den Spielern bestehen, die eine Verbindung zum abgeschnittenen Teil des Graphen ermöglichen. Würde beispielsweise eine Kante zwischen Spieler 2 und Spieler 5 bestehen, so würde sich das Update von Spieler 3 zu Spieler 2 verbreiten. Von dort an Spieler 1 und Spieler 5. Nun von Spieler 5 zu Spieler 4. Dieser könnte dann seinen Zug machen. Dies gilt auch für Fehler, in denen es zu mehr Brüchen gekommen ist. Hierbei müssen zur Auflösung wieder alle abgetrennten Ringteile von einem Spieler erreichbar sein.

5.4 Vergleich der Versendungsstrategien

5.4.1 Vergleich der Netzwerkbelastung

Die drei vorgeschlagenen Strategien zur Versendung von Packets verursachen unterschiedliche Belastungen des Netzwerks. In diesem Abschnitt werden diese eingeschätzt und danach

	Start eines Spiels	REQUEST_GAME-Zyklus	Beendung von Zug
Einfach gezielt	$2(n-1)$	$2n$	1
Zweifach gezielt	$2(n-1)$	$4n$	2
Broadcast	$2(n-1)$	$2n(n-1)$	$n-1$

Tabelle 5.3: Erwartete Anzahl von versandten Packets mit den verschiedenen Strategien

mit den Messergebnissen aus den Tests verglichen.

Erwartung

Beim Einschätzen der Belastung wird davon ausgegangen, dass die Packets, welche die Peers senden, immer ankommen und dass die Übertragungszeit eines Updates zwischen den Clients nicht über dem Intervall liegt, in welchem sie erneut via REQUEST_GAME anfragen. Außerdem gehe ich davon aus, dass es mindestens 2 Clients gibt und dass die Clients alle im selben Intervall anfragen und dies zu einem synchronisierten Zeitpunkt tun. In der Tabelle 5.3 werden die Strategien gegenübergestellt. Der Typ der versandten Packets verteilt sich beim Start eines Spiels in die Hälfte NEW_GAME und die Hälfte NEW_GAME_OK, bei einem REQUEST_GAME Zyklus in die Hälfte UPDATE_GAME und REQUEST_GAME und bei Beendigung eins Zuges sind es lediglich UPDATE_GAME-Packets.

Ein REQUEST_GAME-Zyklus ist hierbei ein Zeitintervall, in dem kein Spieler spielt und die Clients je nach Verteilungsprinzip untereinander kommunizieren, ob es eine neue Version gibt. Wenn man nun davon ausgeht, dass in jedem Intervall ein Spieler einen Zug macht und dieser unter den Clients kommuniziert wird, so kann man die Menge an versandten Packets gegenüberstellen.

Wie man in Abbildung 5.4 sieht, ist die Netzwerkbelastung mit der Broadcast-Strategie sehr hoch, vor allem mit einer wachsenden Anzahl von Spielern. Die Zweifach-Gezielt-Strategie und die Einfach-Gezielt-Strategie scheinen jedoch recht gut skalierbar zu sein.

Für die Erstellung von Abbildung 5.4 wurde für die Einfach Gezielt-Methode die Funktion $f(x) = 2(n-1) + (1+2n) * x$ genutzt. Diese setzt sich daraus zusammen, dass das Spiel einmal eröffnet werden muss. Danach wird die Belastung, welche aus den Runden entsteht,

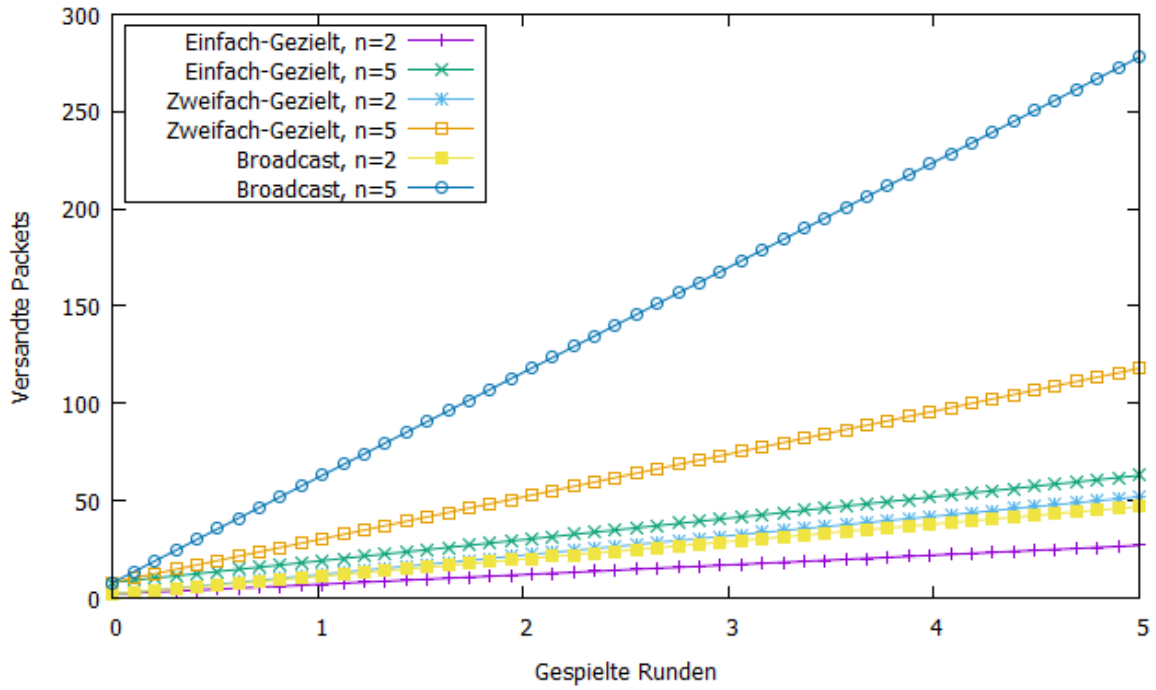

Abbildung 5.4: Vergleich der Strategien

hinzugerechnet. Hierbei wurde der Wert für einen REQUEST_Game-Zyklus und das Versenden eines Spielzuges gewählt.
Die Funktionen für die anderen Strategien wurden nach demselben Prinzip bestimmt. Für Zweifach Gezielt wurde $g(x) = 2(n-1) + (2+4n) * x$ verwendet. Für Broadcast $h(x) = 2(n-1) + (n-1+2n(n-1)) * x$.

Messung

Der in Abschnitt 5.1.1 beschriebene dritte Test liefert Daten, welche die Menge und die Größe der versandten Packets beschreiben. Diese werden in Abbildung 5.5 dargestellt. Hier wurde, wie auf der X-Achse sichtbar, die Verteilungsstrategie während des Spiels bei allen Peers geändert. Man erkennt den Wechsel von der ursprünglichen Strategie Einfach-Gezielt zu Zweifach-Gezielt durch eine erhöhte Steigung des Graphen. Beim Wechsel zur Broadcast-Strategie lässt sich kaum ein Unterschied im Verlauf des Graphen erkennen. Dies ist vermutlich der Fall, da in dem Test nur mit vier Geräten gearbeitet wurde. So würden nach Tabelle 5.3 pro REQUEST_GAME-Zyklus mit der Broadcast-Strategie nur $2*4(4-1) = 24$ Packets versendet werden, was im Vergleich zu $4*4 = 16$ mit der Zweifach-Gezielt-Strategie ein ge-

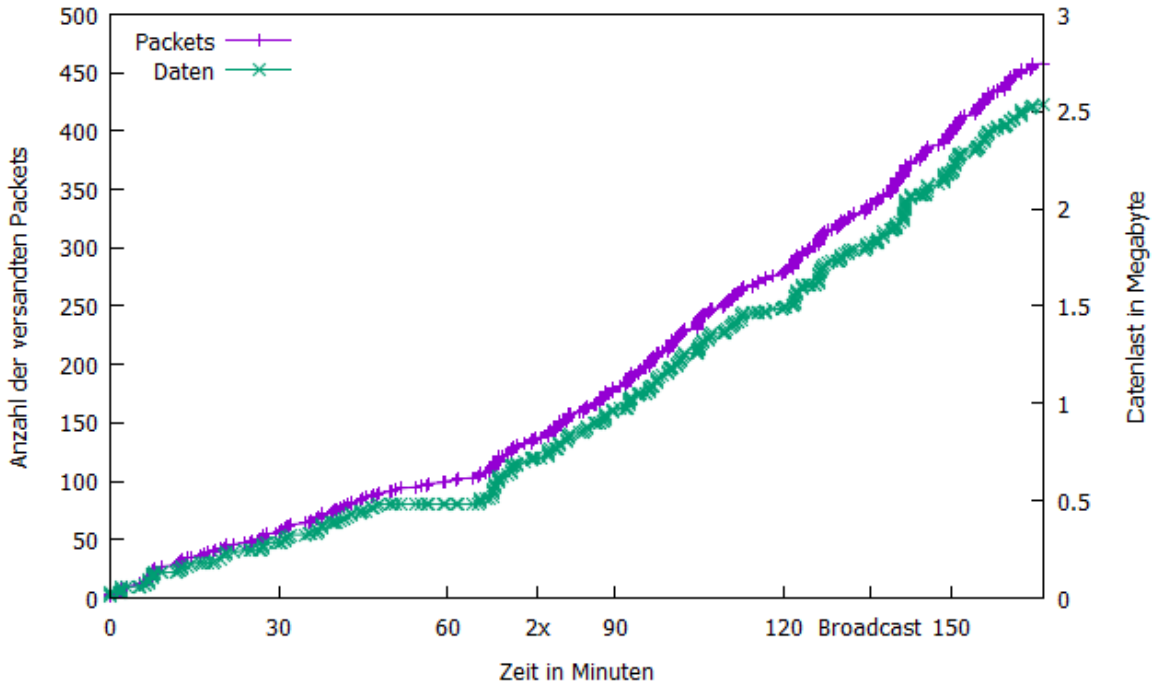

Abbildung 5.5: Verlauf des dritten Tests in Bezug auf Menge und Größe der versandten Packets

ringer Unterschied ist. Die Daten sind in Abbildung 5.6 noch einmal dargestellt. Hierbei wurden lediglich Packets mitgezählt, welche zwischen dem Abschicken des ersten Packets nach dem Vervollständigen eines Zuges und dem Packet, welches den letzten Client aktualisierte, verschickt wurden. Die dazu als Vermutung abgebildete Funktion ist eine Abwandlung der, welche für Abbildung 5.4 verwendet wurde. Sie enthält nicht die Packets, welche beim Erstellen eines Spiels versandt werden.
So liegen die Werte für Einfach-Gezielt bzw. Zweifach-Gezielt deutlich über der Vermutung. Der Grund hierfür könnte sein, dass die Zustellung der Packets sich mit diesen Strategien durch die benötigte REQUEST_GAME-Anfrage der Empfänger so lange verzögert, dass in der Zwischenzeit auch bereits aktualisierte Mitspieler erneut eine solche Anfrage stellen. Der Test wurde zudem mit einer Konfiguration durchgeführt, welche eine sofortige Neuanfrage selbst bei kürzlich aktualisierten Spielständen zulässt, was diesen Effekt verstärkt. Überraschend wirkt, dass die Broadcast-Strategie deutlich unter der vermuteten Menge an versandten Packets liegt. Dies lässt sich durch denselben Umstand begründen: Dadurch, dass an jeden Client ein Packet versendet wird, müssen diese nicht erst eine REQUEST_GAME-Anfrage versenden, bevor sie die Aktualisierung erhalten. Dadurch wird die Runde schneller beendet und den Peers bleibt weniger Zeit, eine REQUEST_GAME-Anfrage zu stellen.

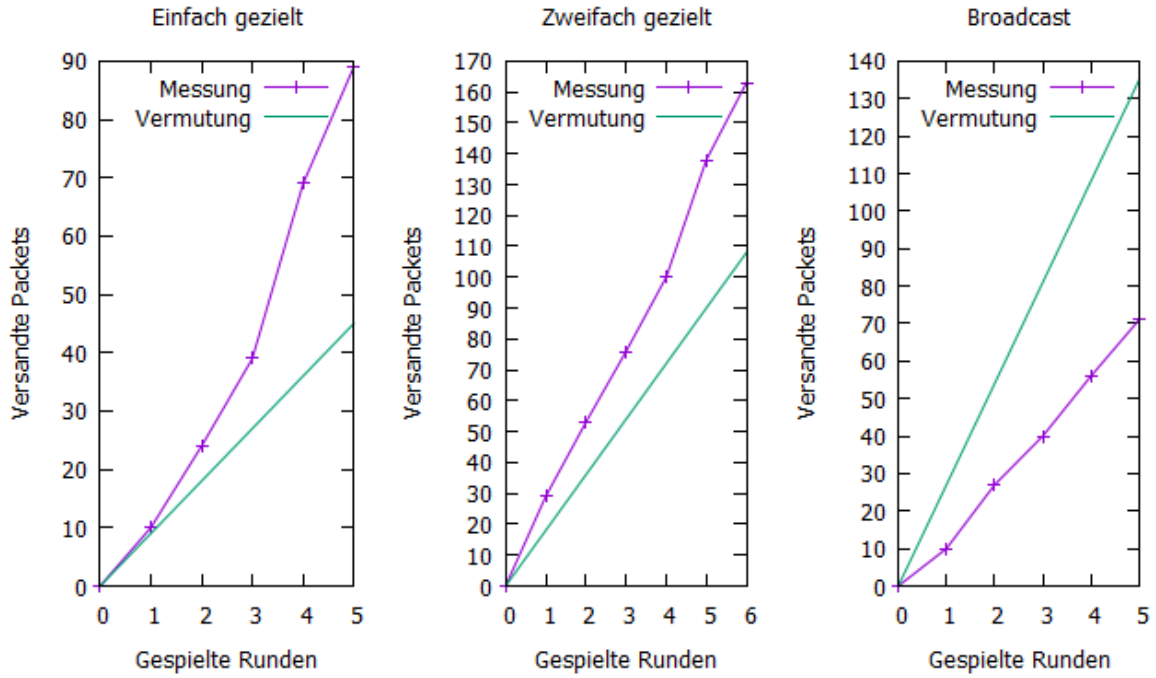

Abbildung 5.6: Vergleich der Strategien in Bezug auf die versandten Packets pro Zug

5.4.2 Aktualisierungsgeschwindigkeit der Spielstände

Wenn ein Spielstand nun von mehreren Spielern bespielt wird, kann es zu langen Aktualisierungszeiten zwischen den Spielern kommen. Dies führt in der Regel zu einem schlechteren Spielgefühl, da die Spieler nicht genau wissen, wer gerade am Zug ist oder was gerade im Spielverlauf geschieht. Hierbei unterscheiden sich die verschiedenen Ansätze, um die Packets zu verteilen. Im Folgenden werden die einzelnen Ansätze bewertet und später die Vermutungen mit Daten aus einem Test überprüft.

Einfach-Gezielt

Die benötigte Zeit, bis sich ein Spielzug in einem Netzwerk verbreitet hat, ergibt sich unter der Bedingung, dass sich während der Übertragung die Beschaffenheit des Netzwerks nicht verändert, aus der unten stehenden Formel. Es wird von mindestens 2 Clients im Netzwerk ausgegangen. Der Client, welcher den Zug vollführt, ist der x. aus C. Die Formel setzt sich daraus zusammen, dass das Update in diesem Fall vom x. Client aus jeden anderen Clienten

in der Zugreihenfolge passieren muss.

$$meg(C,C_x) = \sum_{\substack{c \in C_{P,O} \\ c \neq C_x}} u(vor(c),c)$$

Zweifach-Gezielt

Bei der Einschätzung der Zweifach-Gezielt-Strategie wird von den oben erläuterten Bedingungen ausgegangen. Der Unterschied ist, dass sich die Clients im Zweifach-Gezielt-Modus befinden. Da je nach Netzbeschaffenheit verschiedene Szenarien auftreten können, ist eine Fallentscheidung nötig:

1. Im Fall, dass alle Kanten des Graphen in etwa gleich gewichtet sind, bringt die Zweifach-Gezielt-Strategie eine Zeitersparnis von der Hälfte der Zeit oder weniger, da zwar doppelt so viele Kanten in derselben Zeit abgelaufen werden können, trotzdem die Kante zwischen dem Sender und seinem Vorgänger passiert werden muss.

2. Sind nun innerhalb des Graphen der Topologie eine oder mehrere sehr hoch gewichtete Kanten, so gilt das selbe wie in 1.

3. Ist die Kante, welche den Ausgangsknoten mit seinem Vorgänger verbindet, extrem hoch gewichtet, so kann es sogar sein, dass die Aktualisierungsgeschwindigkeit sich überhaupt nicht verringert.

Unter dem Strich ergibt sich: Zweifach-Gezielt bietet nur unter bestimmten Bedingungen einen Vorteil gegenüber Einfach-Gezielt und zwar genau wenn gilt:

$$u(C_x, vor(C_x)) < meg(C,C_x)$$

So lässt sich die Zweifach-Gezielt-Strategie wie folgt einordnen:

$$\frac{1}{2} meg(C,C_x) \leq mzg(C,C_x) \leq meg(C,C_x)$$

Somit kann Zweifach-Gezielt zwar deutlich schneller sein, muss es allerdings nicht. Auch vorteilhaft ist, dass es die Möglichkeit bietet, einen Bruch in der Ringtopologie auszugleichen. Von daher kann es sinnvoll sein, diese Strategie zu verfolgen.

Broadcast

Im Broadcast-Modus werden die Updates an alle Clients im Spiel gestreut. Diese Strategie verursacht zwar eine hohe Belastung im Netzwerk, kann aber vor allem bei manchen Topologien hilfreich sein, in denen die Mitglieder eines Spiels keine direkte Verbindung untereinander haben. Der größte Vorteil dieser Methode ist, dass Packets auch über Peers versendet werden können, welche nicht am Spiel teilnehmen und nicht zwischen zwei aufeinander folgenden Mitspielern liegen. Es wird wieder von denselben Bedingungen wie oben ausgegangen. Um abzuschätzen, welche Zeitersparnis dies liefert, muss wieder unterschieden werden:

1. Es könnte sein, dass alle Verbindungen abgesehen von denen, welche sich auf dem Ring befinden, extrem hoch gewichtet oder nicht vorhanden sind. In diesem Fall braucht der Broadcast-Ansatz genau so lange wie mit der Zweifach-Gezielt-Strategie.

2. In allen anderen Fällen ist die Broadcast-Methode schneller.

Die Broadcast-Methode dauert so lange wie der längste von den minimalen Aktualisierungszeiten vom Ausgangsknoten zu jedem anderen:

$$mbg(C,C_x) = max(\{t(C_x,c) \mid c \in C_{P,O}\})$$

Dies gilt im Optimalfall, wenn vom Ausgangsknoten alle anderen Knoten erreichbar sind und keine Packets verloren gehen. Wenn nicht, dann dauert die Aktualisierung länger. Maximal aber so lange wie die Zweifach-Gezielt-Methode.
Die Methode kann wie folgt eingeordnet werden:

$$0 < mbg(C,C_x) \leq mzg(C,C_x)$$

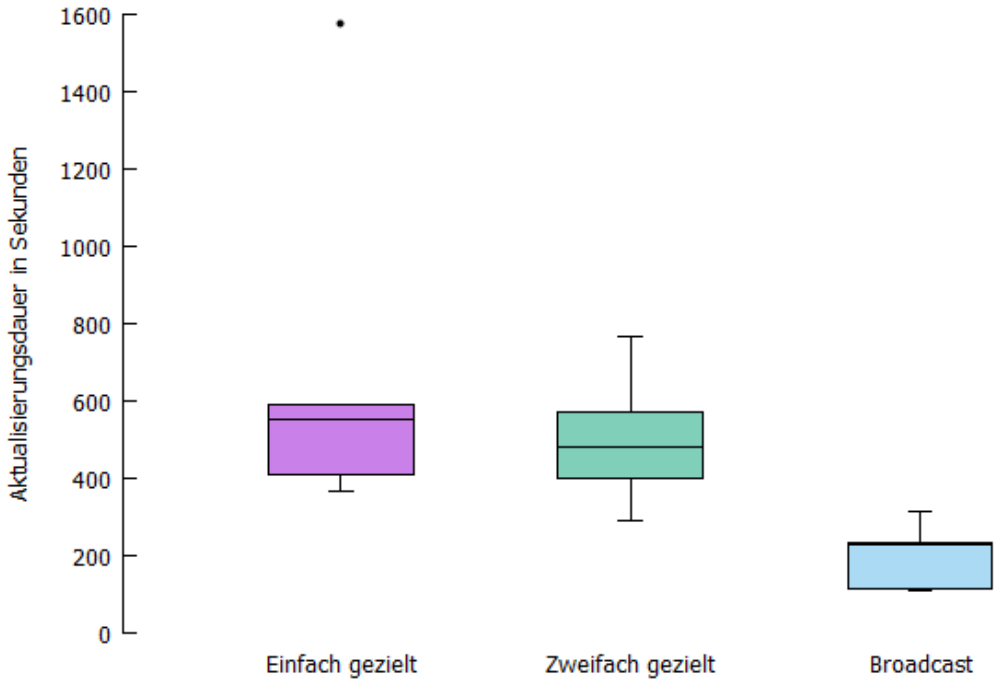

Abbildung 5.7: Vergleich der Strategien in Bezug auf die Aktualisierungsgeschwindigkeit

Messung

Mithilfe vom dritten Aufbau, welcher in Abschnitt 5.1.1 beschrieben wird, wurden auch Daten zur Aktualisierungsdauer erhoben. Diese werden in Abbildung 5.7 dargestellt. In der Abbildung werden die Strategien verglichen. Hierbei ist zu erkennen, dass die Einfach-Gezielt-Strategie im Vergleich zur Zweifach-Gezielt-Strategie – abgesehen von dem abweichenden Wert – kaum langsamer ist. Dies liegt vermutlich an dem Versuchsaufbau, welcher nur vier Geräte vorsieht. Bei der Broadcast-Strategie kann man eine deutliche Verbesserung zu den anderen erkennen. Somit ist diese laut den vorliegenden Daten und der Einschätzung die beste Strategie für Spiele mit einer geringen Spieleranzahl.

5.4.3 Fazit

Die drei Strategien haben alle ihre Vor- und Nachteile. So hält die Einfach-Gezielt-Strategie die Menge der versandten Packets niedrig. Selbst wenn keine erneute Nachfrage mehr entsteht, erhält der Spieler, welcher die Aktualisierung am ehesten benötigt, da er den nächsten

Zug tätigen muss, am ehesten eine Aktualisierung.
Die Zweifach-Gezielt-Strategie sorgt hingegen für eine schnellere Verteilung der Information unter den Spielern und kann im Gegensatz zur Einfach-Gezielt-Strategie auch Probleme wie den Bruch in der Ringtopologie lösen. Hierbei versendet die Zweifach-Gezielt-Methode aber auch deutlich mehr Packets.
Die Broadcast-Strategie ist zwar ein roher Ansatz, trotzdem hat sie in den Tests zu guten Ergebnissen geführt: Sie hatte die geringste Aktualisierungsdauer und hat auch, entgegen der Vermutung, wenig Packets verschickt. Dies lag wie aufgeführt daran, dass die Runden deutlich kürzer waren. So ist die Broadcast-Methode die beste für Spielrunden, in denen eine geringe Übertragungsdauer zwischen den Spielern besteht und diese ihre Züge schnell nach Erhalt einer Aktualisierung tätigen. Zusätzlich bietet sie noch die Option, auch doppelte Brüche in der Ringtopologie zu umgehen. Somit könnte sie auch als Lösung für ungünstige Topologien genutzt werden. Generell gibt es keine *beste* Strategie. Vielmehr ist sie von den Nutzern bzw. ihrer Nutzungsweise abhängig, weswegen die genutzte Strategie in der Implementierung *GameLayer* frei wählbar ist.

5.5 Bewertung der Peer-Verbindung

Um zum Beispiel Spielervorschläge bei der Erstellung des Spiels bieten zu können, ist es wichtig, die Bewertung der Peers, mit denen man spielen kann, zu bieten. Hierbei ergeben sich mehrere Möglichkeiten, dies mit dem bestehenden Protokoll zu realisieren. Die Voraussetzung wäre, dass ein Packet ein anderes anfordert – nach dem Ping/Pong-Prinzip. Hierfür würden sich REQUEST_GAME, UPDATE_GAME und HELLO, HELLO_OK anbieten. Diese Möglichkeiten werden im Folgenden diskutiert.

1. REQUEST_GAME und UPDATE_GAME würden sich zur Auswertung der Peer-Verbindung anbieten, da sie aufeinander folgen. Das Problem hierbei ist, dass ein REQUEST_GAME auch ein DECLINE-Packet als Antwort erhalten kann. Außerdem kann nach Protokoll ein solches Packet bewusst durch den Empfänger ignoriert werden.

2. HELLO und HELLO_OK bieten sich viel eher an. Sie können regelmäßig verschickt werden, sind recht klein und aktualisieren zusätzlich auch Namen und den Profiltext des Empfängers.

Um die Peer-Verbindung nun zu bewerten, müssen alle gesendeten Packets, welche entweder an den Peer ausgingen und vom Typ HELLO sind, oder welche von dem Peer ausgingen und vom Typ HELLO_OK sind, aus der Datenbank ausgelesen werden. Hier muss dann den HELLO-Packets ein HELLO_OK-Packet zugewiesen werden. Dann kann die Zeitdifferenz der Ankunft der Packets errechnet werden. Wenn man nun den Durchschnitt all dieser Werte errechnet, erhält man eine durchschnittliche RTT (Round Trip Time). Leider wären die Werte vor allem im Kontext von Opportunistischen Netzwerken vermutlich nicht aussagekräftig, da sich die RTTs sehr schnell und vor allem häufig ändern können. In diesem Fall wäre es besser, dem Nutzer sowohl eine durchschnittliche RTT als auch eine kürzlich ermittelte zu bieten. Wie aussagekräftig diese Werte sind, wäre dann zu testen.

5.6 Cheating

Das Verschaffen von unfairen Vorteilen in Spielen wird auch als Cheating bezeichnet. Da im Umfang der Arbeit Spiele bzw. eine Plattform für diese entwickelt wurden, wird aufgeführt, wie man ihre Funktionsweise ausnutzen könnte, um sich selber einen Vorteil gegenüber den anderen Spielern zu verschaffen.

5.6.1 Modifikation der App

Um zu cheaten, muss der Nutzer, wenn er keine Bugs oder andere Mechaniken der Applikation missbraucht, Modifikationen an ihr vornehmen. Im Folgenden werden Möglichkeiten beschrieben, dies zu bewerkstelligen. Danach werden einige Angriffe beschrieben, welche unter Ausnutzung der Mechaniken der Applikation oder des Protokolls möglich wären.

Nutzung von Cheatclients

Unter Cheatclients versteht man eine modifizierte Version eines Clients. In dem Fall der entwickelten Apps wäre dies eine repackaged [Now17] Version der Verwaltungs- bzw. Spiele-App oder gar eine komplett neu kreierte Version der Spiele-App, welche sich über denselben Spielnamen wie das Original bei der Verwaltungs-App registriert. Sollte eine sol-

che Version der Verwaltungs-App verwendet werden, so hat der Angreifer volle Kontrolle und kann den Spielstand beliebig editieren. Dies schließt sowohl den Zustand des Spiels als auch die Mitspieler und ihren Status ein. Dieser modifizierte Spielstand wird dann über ein UPDATE_GAME-Packet an einen Mitspieler versendet, welcher diesen dann weiter im Netzwerk verbreitet. Um dies zu verhindern, bietet sich eine Methode, welche dem Onion-Routing [DMS04] nachempfunden ist, an. Es würde anstelle von ganzen Spielständen nur der Ausgangszustand eines Spielstands versendet werden. Dieser würde vorher vom Host elektronisch signiert werden, damit die Mitspieler den Spielstand als von ihm generiert erkennen können. Spielt nun ein Spieler seinen Zug, so könnte er seinen Spielzug dem ursprünglichen Spielstand hinzufügen und die Kombination erneut signieren. Dies würden dann alle Spieler für Änderungen in der Liste der Spieler und ihrer Spielzüge durchführen. Hieraus würde zum einen die Möglichkeit bestehen, zu überprüfen, ob der empfangene Spielzug wirklich durch den Spieler getätigt wurde und ob es überhaupt nach der Logik des Spieles möglich gewesen wäre, den erhaltenen Zug zu spielen. Wird in einem Spiel eine Zufallskomponente wie zum Beispiel ein Würfel verwendet, so wäre auch eine Versendung des Zufallsgenerators nötig, um einen Spielzug zu verifizieren.

Entfernen von anderen Nutzern

Nutzt der cheatende Mitspieler einen solchen Cheatclient, so ist es für ihn möglich, den Status anderer Nutzer auf DECLINED zu setzen. Wenn dies geschieht, wird, sobald sich die Information im Netzwerk verbreitet hat, der Zug des Nutzers übersprungen und er wird auch keine Updates mehr erhalten. Der betroffene Nutzer kann hierbei nichts tun, da sich der Status eines Spielers nur steigern kann und DECLINED der höchstkodierte Status ist. Würden die Veränderungen innerhalb der Spielerliste wie oben beschrieben verifiziert werden, so wäre dies nicht mehr möglich, wenn auch eine Signatur des DECLINE-Packets erfolgen würde.

Überspringen von gegnerischen Zügen

Wenn ein Nutzer, welcher einen Cheatclient verwendet, die Funktion entfernt, die der Spiele-App mitteilt, dass er nicht am Zug ist, so könnte er unbegrenzt viele Züge spielen, ohne dass einer der Mitspieler einen Zug machen dürfte.

5.6.2 Missbrauch der App

Es wäre auch möglich, sich durch den Missbrauch der App einen Vorteil gegenüber den anderen Spielern zu verschaffen oder diese zu belästigen.

Spam mit Spieleinladungen

Ein bösartiger Nutzer könnte die Verwaltungs-App dazu nutzen, einen Spieler gezielt in Spiele mit hoher Spielstandgröße hinzuzufügen, um durch den aufkommenden Datenversand auf ihn wie eine DoS-Attacke (Denial of Service Attacke) zu wirken und sowohl das Endgerät des Ziels als auch das Netzwerk zu belasten. Im schlimmsten Falle könnte der Angreifer dies automatisieren. In diesem Fall werden viele Spielstände bei dem Zielnutzer auftauchen, welche er wieder löschen muss. Zwar handelt es sich in diesem Fall nicht direkt um Cheating, dennoch könnte dieses Verhalten einen Spieler zum Beispiel davon abhalten, in bestimmten Spielständen seinen Zug zu machen. Das Opfer des Angriffs kann die Funktion von *Opponent* nutzen, den Angreifer auf die Blacklist zu setzen, um keine Bundles mehr von ihm zu erhalten.

Hinzufügen von Geisternutzern

Wenn das Spiel es zulässt, ist es möglich, durch das Hinzufügen von neuen Spielern einen Vorteil zu erlangen. So lässt sich durch das ständige Hinzufügen von neuen Nutzern eine Kette starten, wobei der Zug der eigentlichen Mitspieler verzögert wird. In diesen dazugewonnenen Zügen kann der cheatende Nutzer mit den anderen Clients Züge machen, welche sein Spielziel begünstigen.

5.6.3 Fazit

Es gibt viele Möglichkeiten, den Spielfluss zu stören. Hierbei stehen einem Angreifer dann, wenn er eine Spiele-App neu gepackt hat oder eine eigene mit demselben Namen bei sich registriert hat, viele Möglichkeiten zur Verfügung. Um Angriffe zu verhindern, müsste jeder Client die Züge der anderen verifizieren können, was mit dem vorgestellten Lösungs-

vorschlag des Onion-Routings möglich wäre. Die beste Lösung, welche mit der aktuellen Implementierung der Apps nutzbar ist, ist die Nutzer kennenzulernen und nur mit denen zu interagieren, welche sich als vertrauenswürdig erwiesen haben. Bösartige Nutzer können dann durch die Funktion von *Opptain* blockiert werden.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Arbeit ist es gelungen, die App *GameLayer* als Verwaltungsplattform für rundenbasierte Strategiespiele in Opportunistischen Netzwerken zu implementieren. Die App bietet ihren Nutzern auf Basis der Netzwerk-App *Opptain*, die Möglichkeit an, rundenbasierte Strategiespiele auf ihren Android Smartphones zu spielen. Die Spiele selber müssen hierbei keine Netzwerkinteraktionen vornehmen. Somit müssen diese nicht mehr von Entwicklern von Spiele-Apps realisiert werden. Speziell die verschiedenen Strategien zur Synchronisierung des Spielstands unter den Mitspielern einer Partie wurden näher analysiert und werden für den Nutzer frei konfigurierbar angeboten.

Durch das Bestehen einer Plattform wie *GameLayer* könnten in Zukunft mehr Implementierungen von Spielen für die Nutzung mit *Opptain* erfolgen. Durch dieses größere Angebot von Anwendungen könnte die Nutzung von *Opptain* im Alltag attraktiver werden und dessen Netzwerk durch den damit verbundenen Zuwachs von Nutzern gestärkt werden.

6.2 Ausblick

Im Verlauf der Bearbeitung des Themas haben sich einige Anknüpfungspunkte für zukünftige Arbeiten ergeben. So ist, auch wenn das Konzept des Zusammenspiels der Apps sehr frei ist, die Weitergabe von Informationen über Intents ein Sicherheitsrisiko. Generell ist es dem Entwickler einer Spiele-App zur Zeit nicht erlaubt, Modifikationen an der Arbeitsweise der Verwaltungs-App *GameLayer* vorzunehmen. Dies schließt sowohl eine eigene Benutzeroberfläche als auch Einfluss auf die Strategie der Netzwerkbildung ein. Um dem Entwickler noch mehr Freiheit geben zu können, wäre es von Vorteil, die Funktionalität von *GameLayer* als API anzubieten.

Es hat sich herausgestellt, dass sich die Nutzung der unterschiedlicher Versendungsstrategien in unterschiedlichen Szenarien anbietet. Denkbar wäre auch die Implementierung und Testung neuer Strategien. Aktuell muss der Nutzer selber entscheiden, welche der Strategien er verwenden möchte. Dies ist vor allem für Nutzer, welche die Bedeutung der konfigurierbaren Parameter nicht verstehen, von Nachteil. Definitiv könnte die Anwendung automatisch die Parameter anpassen, um das bestmögliche Spielerlebnis für den Nutzer zu bieten. Dies würde die Bewertung der Peer-Verbindungen und die Behandlung von Cheatern mit einschließen. Um gegen Cheater vorzugehen empfiehlt es sich, das vorgeschlagene mehrschichtige Verschlüsselungsprinzip zu implementieren und zu testen.

Aktuell ist *GameLayer* auf rundenbasierte Spiele beschränkt, doch sind auch die Echtzeitspiele sehr verbreitet. Ob es möglich ist, diese Spiele auch auf Basis eines Opportunistischen Netzwerkes zu realisieren, ist vermutlich vom jeweiligen Spielprinzip abhängig. Ist es aber vielleicht möglich, Spielprinzipien zu entwickeln, welche nicht nur unempfindlich gegenüber den Eigenschaften eines Opportunistischen Netzwerkes sind, sondern das Netzwerk an sich fördern? Beispielweise dadurch, dass der Nutzer dafür belohnt wird, als Data-Mule für Nachrichten innerhalb des Netzwerkes zu fungieren.

Literaturverzeichnis

[DMS04] DINGLEDINE, Roger; MATHEWSON, Nick; SYVERSON, Paul: Tor: The Second-Generation Onion Router. 2004. Forschungsbericht.

[Gooa] GOOGLE: *AlarmManager*. https://developer.android.com/reference/android/app/AlarmManager.html, . Zuletzt besucht: 27.10.2017

[Goob] GOOGLE: *App Manifest*. https://developer.android.com/guide/topics/manifest/manifest-intro.html, . Zuletzt besucht: 27.10.2017

[Gooc] GOOGLE: *Dokumentation AlarmManager: setInexactRepeating()*. https://developer.android.com/reference/android/app/AlarmManager.html, . Zuletzt besucht: 27.10.2017

[Good] GOOGLE: *Dokumentation Manifest.permission*. https://developer.android.com/reference/android/Manifest.permission.html, . Zuletzt besucht: 27.10.2017

[Gooe] GOOGLE: *Getting a Result from an Activity*. https://developer.android.com/training/basics/intents/result.html, . Zuletzt besucht: 27.10.2017

[Goof] GOOGLE: *Intents and Intent Filters*. https://developer.android.com/guide/components/intents-filters.html, . Zuletzt besucht: 27.10.2017

[Goog] GOOGLE: *Parcelables and Bundles*. https://developer.android.com/

guide/components/activities/parcelables-and-bundles.html, . Zuletzt besucht: 27.10.2017

[Gooh] GOOGLE: *Requesting Permissions at Run Time*. https://developer.android.com/training/permissions/requesting.html, . Zuletzt besucht: 27.10.2017

[Gooi] GOOGLE: *Saving Data in SQL Databases*. https://developer.android.com/training/basics/data-storage/databases.html, . Zuletzt besucht: 27.10.2017

[Gooj] GOOGLE: *Saving Key-Value Sets*. https://developer.android.com/training/basics/data-storage/shared-preferences.html, . Zuletzt besucht: 27.10.2017

[Gook] GOOGLE: *Scheduling Repeating Alarms*. https://developer.android.com/training/scheduling/alarms.html, . Zuletzt besucht: 27.10.2017

[Gool] GOOGLE: *Sharing a File*. https://developer.android.com/training/secure-file-sharing/share-file.html, . Zuletzt besucht: 27.10.2017

[Ipp15] IPPISCH, Andre: *A fully distributed Multilayer Framework for Opportunistic Networks as an Android Application*, Department of Computer Science, Heinrich-Heine-University Düsseldorf, Masterarbeit, März 2015

[Moba] MOBYGAMES: *Civilization series*. https://www.mobygames.com/game-group/civilization-series, . Zuletzt besucht: 27.10.2017

[Mobb] MOBYGAMES: *Genre Definitions*. http://www.mobygames.com/glossary/genres, . Zuletzt besucht: 27.10.2017

[Mobc] MOBYGAMES: *Heroes of Might and Magic series*. https://www.mobygames.com/game-group/heroes-of-might-and-magic-series, . Zuletzt besucht: 27.10.2017

[Now17] NOWAK, Martin: *Security Threats in Android-based Opportunistic Networks*, Heinrich-Heine-Universität Düsseldorf, Bachelorarbeit, Juni 2017

[Roh17] ROHR, Oliver: *A Filesharing App for Android based on Opportunistic Networks*, Heinrich-Heine-Universität Düsseldorf, Bachelorarbeit, Februar 2017

[Staa] STATISTA: *Installed base of smartphones by operating system from 2015 to 2017 (in million units)*. `https://www.statista.com/statistics/385001/smartphone-worldwide-installed-base-operating-systems/`, . Zuletzt besucht: 27.10.2017

[Stab] STATISTA: *Number of mobile phone gamers in the United States from 2011 to 2020 (in millions)*. `https://www.statista.com/statistics/234635/number-of-mobile-gamers-forecast/`, . Zuletzt besucht: 27.10.2017

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 30. Oktober 2017 Fabian Weiß

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Anwendungen GameLayer, TicTacToe und AndroidStrategy
- die Anleitungen für die Anwendungen
- die zur Auswertung verwendeten Datensätze
- die Websites der verwendeten Internetquellen