Fachhochschule Köln
University of Applied Science Cologne

Fakultät für Informatik- und Ingenieurwissenschaften

Bachelorarbeit

“YourSights” –

Konzeptionierung und Realisierung einer Geodaten-gestützten Informationsanwendung für Smartphones

Vorgelegt an der Fachhochschule Köln

Campus Gummersbach

Im Studiengang Allgemeine Informatik

Ausgearbeitet von:

Marcel Stratmann

Kevin Ruthmann

Erster Prüfer: Prof. Dr. Faeskorn-Woyke

Zweiter Prüfer: Prof. Dr. Victor

Gummersbach, im 03/2012

Inhaltsverzeichnis

1. Einleitung 5

1.1 Analyse Projektstand: Praxisprojekt 6

1.2 Zielsetzung 8

2. Projektplanung 9

2.1 Meilensteine 9

2.2 Projektabgrenzung 10

3. Revision Praxisprojekt 11

3.1 Mapsforge 11

3.2 Sonstige Änderungen 13

4. Anforderungsanalyse 14

5. Evaluation von potenziellen Kommunikationstechnologien 16

5.1 NIO-Frameworks 17

5.2 Verworfene Alternativen 20

6. Netty 21

7. Kommunikationsmodell 26

7.1 Anforderungen und Anforderungsanalyse 27

7.2 Design 28

8. Übergreifende Entwicklung 33

8.1 Implementierung des Protokolls 34

8.2 Implementierung von *Netty* 43

9. Entwicklung – Server 51

9.1 Anforderungen – Server 51

9.2 Softwaremodell – Server 52

10. Entwicklung Client 61

10.1 ProgressListener 61

10.2 Grafische Oberfläche – App 66
11. Projektstand 73
11.1 Projektstand – Server 73
11.2 Projektstand – Client 73
12. Fazit 75
13. Ausblick 76
14. Verzeichnisse 78
14.1 Abkürzungsverzeichnis 78
14.2 Abbildungsverzeichnis 79
14.3 Tabellenverzeichnis 80
14.4 Codeverzeichnis 81
14.5 Literaturverzeichnis 82
15. Glossar 85
16. Anhang 90

1. Einleitung

Der Markt für mobile Endgeräte unterliegt einem stetigen Wachstum. Unter anderem ist dies auf die vielseitigen Einsatzmöglichkeiten zurückzuführen, die sich durch die Entwicklung immer leistungsfähigerer und dabei noch sparsamerer Prozessoren ergeben.

Smartphones übernehmen Aufgaben, die normalerweise den Einsatz eines PCs oder Notebooks erfordern; vor allem soziale Netzwerke, wie Facebook oder Twitter profitieren von diesem Fortschritt. Aus einer Forsa-Umfrage im Zeitraum zwischen 2010 und 2011 geht hervor, dass im Jahr 2011 knapp jeder vierte Deutsche im Besitz eines Smartphones ist. Kaum eine Werbepause vergeht ohne die Anpreisung von den mobilen Alleskönnern oder Applikationen, die auf ihnen installiert werden können. Mit einer Stagnation des Marktes kann in naher Zukunft nicht gerechnet werden. Die anhaltende Verbreitung und die technischen Möglichkeiten, die Smartphones heutzutage bieten, motivierten dazu, sich mit der Entwicklung von mobilen Anwendungen genauer auseinanderzusetzen.

Dabei wurde im Praxisprojekt „YourSights – Konzeptionierung und Realisierung einer Geodaten-gestützten Informationsanwendung für Smartphones“ eine mobile Anwendung entworfen und zu einem Großteil implementiert, mit welcher es möglich sein sollte, Städteführer, Messeführer, Zoo-Rundgänge usw. zu erstellen und wiederzugeben. Solche Touren stellen eine geordnete Sammlung von Wegpunkten, so genannte *Points of Interest*, dar, welche dem Anwender über Beschreibungen und Fotos Informationen vermitteln. Dabei steht es dem Anwender frei, die Wegpunkte in der festgelegten Reihenfolge aufzusuchen oder das Gebiet eigenständig zu erkunden. Diese Idee wurde um einen sozio-technischen Aspekt erweitert, der Anwender durch das Teilen, Bewerten und Kommentieren von erstellten Touren miteinander verbindet. Letztendlich sah das fertige Konzept die Entwicklung einer prototypischen Applikation für das Smartphone, eine Desktopapplikation für das nachträgliche Bearbeiten von Touren, sowie den Einsatz einer Datenbank für die zentrale Speicherung und als gemeinsame Basis für den Informationsaustausch vor.

Im Rahmen der Bachelor-Thesis wird der aktuelle Stand analysiert. Aufgetretene Probleme und Entwicklungslücken werden dabei aufgezeigt und priorisiert. Ziel ist die konzeptionelle Entwicklung von Lösungen und deren prioritätsbedingte Implementierung.

1.1 Analyse Projektstand: Praxisprojekt

Grundlage dieses Kapitels bilden die Praxisprojektdokumentation und der Programmcode vom 19.01.2012. Die Smartphone- und Desktop-Applikation werden differenziert betrachtet. Gemessen an der Funktionsübersicht (Vgl. Kapitel 4.3 der Praxisprojektdokumentation) wird zunächst auf den Projektstand eingegangen, wobei hier nur die nicht implementierten Funktionalitäten Beachtung finden. Das Grundkonzept der bereits umgesetzten Funktionen wird sich nicht ändern. Falls notwendig, erfolgt im Rahmen dieser Arbeit eine Anpassung der Implementierung.

Es folgen eine Bewertung der Mängel und eine grobe Priorisierung für deren Beseitigung. Auf die Datenbank wird an dieser Stelle nicht eingegangen, da das entwickelte Datenbankdesign vollständig ist und alle für die Anwendungsentwicklung erforderlichen Strukturen und Inhalte bereitstellt. Die Implementierung der für die Datenbank bereits geplanten Funktionserweiterungen ist noch ausstehend, für die Entwicklung des Anwendungsprotoypen aber von untergeordneter Bedeutung.

1.1.1 Analyse der Smartphone-Applikation

Auf dem Smartphone soll dem Client die volle Programmfunktionalität zur Verfügung stehen. Dies umfasst den Bezug und die Wiedergabe von Touren, als auch deren Erstellung und Veröffentlichung. Voraussetzung hierfür sind die noch nicht implementierten Funktionen für die Authentifizierung und Registrierung von Anwendern. Zu den ebenfalls noch nicht gänzlich implementierten Funktionen zählen das Suchen, Sortieren und Filtern der im Paketbrowser angezeigten Pakete. Der Download einer Tour warf Fehler auf, dessen Ursprung sich in der Verbindung zur Datenbank findet. Dies ist auf den verwendeten JDBC-Treiber zurückzuführen. Die Bewertungs- und Kommentarfunktionalität ist für die Erstellung oder Wiedergabe einer Tour nicht von Nöten und wurde deshalb vorerst nicht umgesetzt. Dies gilt auch für die Wegpunktsteuerung unter dem Punkt „Wiedergabe“, sowie die Verarbeitung von akustischen Informationen im Allgemeinen. Die Veröffentlichung von Touren warf ebenfalls verbindungsbedingte Probleme auf. Die Bearbeitung von Touren (serverseitig oder lokal) wurde zeitlich bedingt hintenangestellt und letztendlich nicht umgesetzt. Durch den Einsatz einer Karte und einer evtl. später folgenden Navigation wurde auf die Erfassung einer Wegbeschreibung pro Wegpunkt verzichtet. Die Möglichkeit, Einstellungen am Programm vorzu-

nehmen, um bspw. eine Standardschriftgröße festzulegen oder die Wiedergabelautstärke von Audio-Dateien zu verändern, wurde vorerst nicht implementiert.

Die Einteilung der fehlenden Funktionalitäten in Prioritätsgruppen erfolgt aufgrund der Einschränkungen, die für den Anwender daraus resultieren. Die eigentliche Idee und das Konzept, die dem Projekt „YourSights“ zugrunde liegt, darf dabei nicht außer Acht gelassen werden.
Der Bezug und die Veröffentlichung von Touren sind ein essentieller Bestandteil dieser Idee und genießen bei der Implementierung höchste Priorität. Der sozio-technische Aspekt darf dabei nicht aus den Augen verloren werden. Daher wurde die Bewertungs- und Kommentarfunktion direkt auf der nächsten Prioritätsstufe angesiedelt. Der wahrscheinliche Wunsch des Anwenders, eine Tour nachträglich zu bearbeiten, lokal oder serverseitig, sorgt für eine Ansiedlung auf der Folgestufe. Die übrigen genannten Ausstände sind nicht von wesentlicher Bedeutung für das Projekt oder weisen lediglich einen geringen Nutzen für den Anwender auf und erhalten daher die niedrigste Prioritätsstufe.

1.1.2 Desktop-Applikation

Die Desktopanwendung soll dem Zweck dienen, Touren zu bearbeiten und zu veröffentlichen. Kurzfristig wurde aber auch entschieden, dass der Bezug von Touren möglich sein soll. Grund dafür ist die Annahme, dass der PC über eine höhere Bandbreite verfügen kann als der Client und ein Download somit schneller vonstattengeht.
Im Rahmen der Bearbeitung von bereits erstellten Touren soll es möglich sein, Informationen aus mehreren Touren zu kombinieren. Die Funktionalität und die Möglichkeit, Audio-Dateien einzufügen wurde zeitlich bedingt nicht mehr implementiert. Auf eine Authentifizierung und Registrierung wurde vorerst verzichtet, da auch ohne diese Funktionalität die Anwendung noch für die Bearbeitung von Touren genutzt werden kann. Eine Veröffentlichung darf ohne gültige Authentifizierung allerdings nicht erfolgen.
Dies ist jedoch ein essentieller Bestandteil der Desktop-Anwendung. Insofern genießt die Umsetzung der Authentifizierung die höchste Priorität direkt gefolgt von der Bearbeitungsmöglichkeit, Touren zu kombinieren oder zu beziehen. Andere, noch nicht implementierte Funktionen, sind auf der nächsten Prioritätsstufe angesiedelt.

1.2 Zielsetzung

Beruhend auf der Analyse des aktuellen Projektstands, sowie allgemeiner Überlegungen, wurde entschieden, dass für eine erfolgreiche Umsetzung der Projektidee von „YourSights“ die Entwicklung eines Servers als Organisationseinheit nötig ist. Erste Überlegungen in diese Richtung manifestierten sich bereits gegen Ende des Praxisprojekts und wurden im Fazit beschrieben. Der Server soll Anfragen des Clients entgegennehmen, ggfs. aufbereiten und den Client mit den angeforderten Informationen versorgen. Dabei greift er, wenn nötig, auf die in der Datenbank hinterlegten Informationen zurück.

Durch diesen Schritt verschlankt die Anwendungslogik der Smartphone- und Desktop-Applikation. Zudem wird der direkte Zugriff des Clients auf die Datenbank verhindert, was die Sicherheit erhöht und zeitgleich Kommunikationsprobleme, die auf den JDBC-Treiber zurückzuführen sind, beseitigt.

Anstelle der direkten Kommunikation mit der Datenbank ist somit die Entwicklung und Implementierung eines Kommunikationsmodells für eine Client/Server-Architektur Voraussetzung für ein erfolgreiches Gelingen. Des Weiteren muss ein Konzept erarbeitet werden, dessen Aufgabe die Verarbeitung von Anfragen sein wird.

Die Prioritäten bei der Entwicklung orientieren sich dabei am Kontext der Projektidee. Der Server soll effizient und zuverlässig arbeiten. Wichtig hierfür sind die Evaluation verschiedener Technologien und ein adäquates Softwaremodell. Dies soll sich in einer prototypischen Implementierung wiederspiegeln.

Auf Clientseite gilt es die Anwendungen, entsprechend der neuen Umgebung, anzupassen. Dies erfordert einen Austausch oder Umstrukturierung der bislang eingesetzten Kommunikationskomponente.

Zudem soll der Funktionsumfang des Clients erweitert werden. Dafür werden die in Kapitel 1.1.1 aufgezählten, fehlenden Funktionalitäten entsprechend ihrer Priorität vorbereitet und implementiert.

Aufgrund des gestiegenen Projektumfangs durch den Server muss bereits an dieser Stelle eine erste Projektabgrenzung erfolgen. Für die prototypische Entwicklung ist es wichtig, ein umfassendes Gesamtkonzept zu erarbeiten und vor allem clientseitig die Funktionalitäten zu

gestalten. Aus diesem Grund wird der Entwicklung der Smartphone-Applikation der Vorzug gegeben.

2. Projektplanung

Wichtig bei der anfänglichen Planung ist die Identifizierung der Aufgaben. Besonders, da es sich hierbei um eine Gruppenarbeit handelt, welche das Aufteilen bzw. parallele Bearbeiten von Aufgaben ermöglicht. Die Aufgabe dieser Arbeit besteht darin, dass Projekt „YourSights“ weiter zu führen. Der letzte fehlende Bestandteil des Softwaresystems ist hierbei der Server samt Kommunikation. Der Plan des Projekts sieht daher eine Erarbeitung der Aufgabe in vier Phasen vor.

Abb. 1 – Vier Phasen Modell der Projektplanung

Jede dieser Phasen baut dabei aufeinander auf. Zunächst wird das Problem und dessen Anforderung analysiert. Darauf folgt eine Evaluierung der möglichen Lösungen bzw. Kommunikationstechnologien. Anschließend soll ein darauf aufbauendes Modell entwickelt werden, mit dessen Unterstützung man dann die Software implementieren kann. Um einen besseren Überblick über die vier Phasen zu gewinnen, wurde für jede Phase ein Meilenstein definiert, der beschreibt, was in jeder Phase exakt zu tun ist.

2.1 Meilensteine

Meilenstein 1: Anforderungsanalyse & Revision Praxisprojekt

In einem ersten Schritt muss das Problem analysiert werden. Zu diesem Zwecke soll eine Anforderungsanalyse durchgeführt werden. Hier soll beschrieben werden, welche Eigenschaften der spätere Server besitzen soll. Es ist vorgesehen, dass die Eigenschaften durch Qualitätsmerkmale beschrieben werden, sodass man mögliche Lösungen bewerten und untereinander vergleichen kann.

Neben der Analyse soll parallel der vorliegende Projektstand einer Revision unterzogen werden. Deren Ziel ist hierbei eine Inspektion und Qualitätssicherung der zugrundeliegenden Software, sodass eine Weiterentwicklung ohne Schwierigkeiten durchgeführt werden kann.

Als Ergebnis des ersten Meilensteins soll somit eine Liste der Qualitätsmerkmale und eine überarbeitete einsatzfähige Version der zugrunde liegenden Software vorliegen.

Meilenstein 2: Evaluation potentieller Kommunikationstechnologien

Nach der Analyse des Problems sollen mögliche Kommunikationstechnologien recherchiert und anschließend anhand der festgelegten Qualitätskriterien bewertet werden. Ziel des Meilensteins ist eine Auswertung der möglichen Lösungen, sodass eine der Kommunikationslösungen als Grundlage für weitere Arbeitsschritte ausgewählt werden kann.

Meilenstein 3: Erstellung eines Kommunikationsmodells

Die Planung sieht vor, dass vor der eigentlichen Implementierung ein Kommunikationsmodell entworfen wird. Dieses Modell soll festlegen, in welcher Weise die beiden Kommunikationspartner miteinander in Kontakt treten. Die in der Kommunikation auftretenden Anwendungsfälle sollen durch dieses Modell gänzlich abgedeckt werden. Dieser Meilenstein ist essentiell, um Implementierungen in der ausgewählten Kommunikationstechnologie vornehmen zu können. Sie dient außerdem als Konvention der beiden Partner über die Kommunikation, was eine separate Entwicklung der Komponenten Client und Server möglich macht.

Meilenstein 4: Implementierung Server & Client

Auf Grundlage des Kommunikationsmodells und der ausgewählten Technologie soll nun die Client-Server-Kommunikation implementiert werden. Bestandteil dieses Meilensteins ist zum einen die Erstellung eines Servers samt Datenbankanbindung und zum anderen die Etablierung der Kommunikation zwischen der Android App und dem entwickelten Server.

2.2 Projektabgrenzung

Innerhalb dieses Projekts soll eine mögliche Servertechnologie gefunden werden, welche der Aufgabe angemessen ist. Letztendlich soll ein funktionaler Prototyp vorliegen. Essentiell ist es, durch ihn zu zeigen, dass die Kommunikationstechnologie als mögliche Lösung für das Problem eingesetzt werden kann. Des Weiteren umfasst dieses Projekt die Weiterführung der Entwicklung der Android App. Die App und vor allem ihre grafische Oberfläche sollen erweitert und optimiert werden. Kein Bestandteil dieses Projekts ist wiederum die Durchführung von intensiven Testreihen bezüglich der Kommunikation. Ebenso soll auch keine Veröffentlichung der Software innerhalb dieses Projekts erfolgen.

3. Revision Praxisprojekt

Zur Ausgangslage dieses Projekts gehört die bereits entwickelte App für das Android Betriebssystem. Damit die, mit dem Server einhergehenden, notwendigen Änderung an der Software ohne größere Probleme gemacht werden konnten, musste sich die App einer Revision unterziehen und anschließend auf den neuesten Stand gebracht werden. Hierbei waren größere und kleinere Änderungen vorzunehmen, die in diesem Kapitel näher beschrieben werden. Die hier aufgeführten Änderungen haben jedoch nichts direkt mit der Implementierung der Kommunikation zu tun, sondern sind als reine vorbereitende und auffrischende Maßnahmen zu verstehen. Eine grundlegende Änderung fand sich im Bereich des genutzten Open-Street-Map Frameworks. Hier musste das Framework Mobile Maps durch Mapsforge[1] ersetzt werden.

3.1 Mapsforge

Im Zuge der Revision war festzustellen, dass die Entwicklung und der Support der Mobile Maps API von Ericsson Labs eingestellt wurden. Das Mobile Maps Framework ist daher auf lange Sicht für einen Einsatz im Projekt ungeeignet. Auf der Suche nach einer äquivalenten Alternative stieß man auf ein Projekt der Freien Universität Berlin mit dem Namen Mapsforge. Das im Jahr 2008 initiierte Projekt zielt darauf ab, ein freies und offenes (Lizenz: LGPL3) Werkzeug für die Open-Street-Map Community zu schaffen, mit dem es einfach möglich sein sollte, neue, auf OSM basierte, Software zu entwickeln. Der Fokus lag dabei auf eine Entwicklung für die mobile Plattform Android und somit exakt für den Anwendungsbereich des Projekts. Einige der Vorteile dieses Frameworks sind somit die auf Smartphones ausgelegte Performance, Unterstützung von Multitouch-Gesten und die Möglichkeit, das Kartenmaterial, alternativ zum partiellen Download, als Datei auf dem Endgerät persistent zu speichern. So ließ sich bspw. eine Karte für Deutschland (ca. 1 GB groß) auf der Homepage des Mapsforge Projekts herunterladen. Eine exakte Übersicht[2] der Updates von Mapsforge ließ zudem erkennen, dass in regelmäßigen Abständen das Framework und seine Dokumentation, im Gegensatz zu Mobile Maps, aktualisiert und gepflegt wird.

[1] Mapsforge: http://code.google.com/p/mapsforge/ (05.02.2012)
[2] Mapsforge: IssueList, url: http://code.google.com/p/mapsforge/issues/list (07.02.2012)

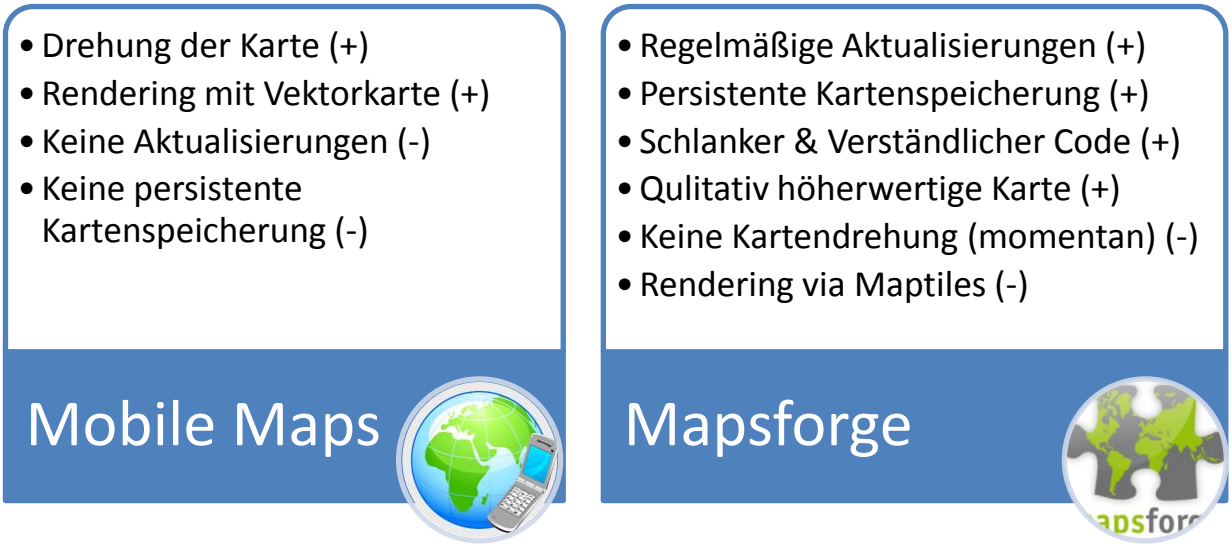

Abb. 2 – Vor- und Nachteile von Mobile Maps und Mapsforge

Startpunkt der Arbeiten war es, die App, bzw. genauer gesagt, die Map-Activity zu Mapsforge zu portieren. Wichtig war es hierbei, dass der Umfang der Funktionalität soweit möglich beibehalten wurde. Ein Nachteil jedoch, der durch die Nutzung von Mapsforge entstand, war die Tatsache, dass sich die Karte derzeit nicht drehen ließ. Eine Ausrichtung der Karte anhand des Kompasses war also nicht wie bisher möglich. Alternativ wurde beschlossen den Blickkegel auf die Kompassrichtung auszurichten, anstatt den Blickkegel zu fixieren und die Karte zu drehen. Die Funktionalität des Drehens der Karte wird jedoch eventuell in einem späteren Mapsforge Release eingeführt werden und kann dann nachträglich in die Activity eingepflegt werden. Weitere Einzelheiten zu den durch Mapsforge einhergehenden visuellen Änderungen können dem Kapitel 10.2 entnommen werden. Die restlichen Funktionalitäten konnten ansonsten fast eins zu eins portiert werden. Dabei war festzustellen, dass der Code durch die Nutzung der neuen API schlanker und verständlicher wurde, was wiederum als ein Vorteil anzusehen ist. Den Arbeitsschritt der Portierung wurde zudem genutzt, um geringfügige Änderungen an der visuellen Gestaltung der Karte vorzunehmen. Die Karte erfuhr somit durch die Änderungen und den Wechsel zu Mapsforge eine Qualitätssteigerung.

3.2 Sonstige Änderungen

Neben der großen Änderung von Mobile Maps zu Mapsforge wurden zudem viele kleine Änderungen eingeführt und Bugfixing betrieben. Ein betroffener Bereich hiervon war das Paket der Datenklassen. Neben der Einführung einer Versionierung der *RAWContainer* und *RAWWaypoint* Klasse, wurde das Konzept der *RAWOperation* Klasse im Hinblick auf die Android App abgeändert. War es beim alten Stand noch so, dass eine leicht modifizierte Version in der App eingesetzt wurde, so ist es jetzt der Fall, dass stattdessen eine Unterklasse von *RAWOperation* benutzt wird. Die abgeleitete Klasse deckt die Detailunterschiede bezüglich des Android Dateisystems ab. Ein weiterer wichtiger Punkt war die Anpassung der Kontexmenüs im Hinblick des jeweiligen Anwendungsfalls. So sollten nur die Optionen für das Listenelement angezeigt werden, welche auch relevant sind. Bspw. sollte für eine bereits heruntergeladene Tour kein Menüpunkt „Download“ vorhanden sein. Ebenfalls verbessert wurde die *onCreate*-Methode der Hauptaktivität. Hier wurden einige kleinere Fehler korrigiert und ein Teil der Programmierung so modifiziert, dass er erst nach Beendigung der vom System aufgerufenen Methoden *onCreate*, *onResume* und *onStart* ausgeführt wird. Dies war notwendig, um einige selten auftretende Exceptions zu vermeiden.

4. Anforderungsanalyse

Bevor die eigentlichen Recherchen für eine mögliche Server-Technologie beginnen konnten, musste der vorliegende Anwendungsfall im Hinblick auf die Kommunikation analysiert werden. So war festzustellen, welche Eigenschaften ein möglicher Server mitbringen muss, um in diesem Kontext eingesetzt werden zu können. Demnach mussten zunächst Qualitätsmerkmale aufgestellt werden, welche die Anforderungen des Anwendungsfalles wiederspiegelten. Das Hauptaugenmerk lag dabei auf einer hohen Qualität bei der Kommunikation zwischen der App und dem Server. Die Kommunikation zur Desktopanwendung ist in dieser Betrachtung hingegen zweitrangig.

4.1 Qualitätsmerkmale

- Geschwindigkeit

  Geschwindigkeit stellt ein wichtiges Kriterium im Bereich der mobilen Anwendungen dar. Wichtig deshalb, da mobile Anwendungen in ihrer Ausführungszeit die Eigenschaft besitzen, kurzweiliger Natur zu sein. Eine langsame Übertragung der Daten wäre in diesem Falle entgegen der Intention des Anwenders und würde daher zu einer negativen Bewertung der App führen. Dieser Bereich umfasst deshalb nicht nur die Geschwindigkeit der Übertragung der Daten, sondern auch Merkmale, wie die Reaktionszeit auf Anfragen und die Dauer bis zum Verbindungsaufbau. Es soll somit vermieden werden, dass der Anwender unnötig lange auf Daten des Servers warten muss.

- Effizienz

  Der Server soll sowohl im Bereich der verwendeten Hardware, als auch bei den genutzten Protokollen effizient sein. Ziel ist es, Overhead zu vermeiden und die transferierte Datenmenge gering zu halten. Dies steht ebenfalls im Interesse des Anwenders, da er bei mobilen Geräten für die transferierten Datenmengen zahlen muss. Eine gute Effizienz trägt zudem zur Geschwindigkeit bei.

- Zuverlässigkeit

  Der Server sollte im Rahmen seiner Arbeit zuverlässig und stetig erreichbar sein. Zudem sind Daten wie Tourpakete ordnungsgemäß zu übertragen. Etwaige Fehler bei der Kommunikation sollen erkannt und behandelt werden können. Außerdem soll der Server in der Lage sein, selbst bei größerer Belastung, die Anfragen in angemessener Zeit zu verarbeiten.

- Skalierbarkeit

  Da die Idee der Software darauf ausgelegt ist, dass sich eine stetig wachsende Community bildet, muss der Server eine schnell größer werdende Menge an Verbindungen und Anfragen handhaben können. So sollte es bis zu einem gewissen Punkt möglich sein, durch Erweiterungen der Hardware, das Problem der wachsenden Community zu lösen. Eine durch die Größe der Community erzwungene gänzliche Neuentwicklung der Serversoftware soll somit vermieden werden.

- Anpassung an den Anwendungsfall mobiler Anwendungen (Android)

  Es ist bei der Auswahl der Technologie darauf zu achten, dass diese auf einem Android Smartphone in einem geeigneten Maße lauffähig ist und in ihrer Art und Weise für mobile Anwendungen geeignet ist. So wäre es bspw. wünschenswert, wenn die verwendete Technologie das Wiederaufnehmen von Verbindungen unterstützen würde, da Verbindungsabbrüche zum Alltag in mobilen Geräten gehören.

- Zugänglichkeit

  Es soll möglich sein, sich in die verwendete Technologie und die dabei verwendete Sprache und Konzepte in einem angemessenen Rahmen einzuarbeiten. Java ist hierbei als Programmiersprache zu bevorzugen, da das restliche System bereits auf Java entwickelt wurde und zusätzlich die Einarbeitung in eine neue Sprache entfällt. Zudem ist Android auf eine Entwicklung in Java ausgelegt. Einige Indizien für eine gute Zugänglichkeit sind bspw. eine gute Dokumentation, viel transparenter Beispielcode und ein guter, zeitnaher Support.

Dies sind die Kriterien, nachdem die möglichen Technologien bewertet werden sollen. Eine gute Verwendbarkeit bei Smartphones (Android), eine gute Geschwindigkeit, sowie Effizienz stehen dabei im Vordergrund der Bewertung und erhalten deshalb bei der Auswahl der Technologie eine erhöhte Priorität.

5. Evaluation von potenziellen Kommunikationstechnologien

Bei der Evaluation der potenziellen Kommunikationstechnologien bedienten wir uns den im Kapitel 4.1 beschriebenen Qualitätsmerkmalen. Erste Recherchen zeigten, dass sich die möglichen Technologien in zwei Richtungen einteilen ließen. Es musste entschieden werden, ob die Implementierung eher auf Anwendungsebene oder eher auf Transportebene realisiert werden soll. Man hatte die Wahl, ein bestehendes Protokoll wie „http“ zu nutzen oder in einer tieferen Schicht im OSI-Referenz-Modell arbeitet und ein eigenes Protokoll zu entwerfen. Zur Entscheidungsfindung wurden erneut die Anforderungen betrachtet, die für eine Kommunikationsanwendung im mobilen Bereich relevant sind. Zusammenfassend konnte man nun einige essentielle Anforderungen an das System definieren:

- Geringer Overhead
- Flexible Anpassbarkeit
- Verwendbarkeit im Android Betriebssystem

Entsprechend des Kriteriums eines geringen Overheads wurde versucht, diesen und somit das Datenvolumen so gering wie möglich zu halten. Dies führte dazu, dass der Entwicklung eines eigenen Protokolls der Vorzug gegenüber der eines bestehenden Protokolls gegeben wurde. Da man in diesem Bereich auf eine zuverlässige Kommunikation angewiesen ist, bot sich das TCP-Protokoll, im Gegensatz zum UDP-Protokoll, als Grundlage an. Dem Vorteil des geringen Overheads bei einem eigenen Protokoll steht ein erhöhter, programmiertechnischer Aufwand gegenüber, den es zu leisten gilt, um die Kommunikation zu etablieren. Vorteilig wiederum ist die speziell an den Anwendungsfall zugeschnittene Lösung.

Da die bereits bestehende Software in Java entwickelt wurde und Java den Vorteil einer plattformunabhängigen Programmiersprache mitbringt, wurde zunächst in dieser Sprache nach einer passenden Möglichkeit gesucht, einen Server und ein eigenes Protokoll zu entwickeln. Fündig wurde man im Java New Input/Output Framework oder auch kurz NIO, wel-

ches seit der Java Version 1.4 ein Teil der Java Standard Edition ist. NIO ist ebenfalls ein Teil des Android Betriebssystems und kann daher für eine Entwicklung genutzt werden. Java NIO bietet den Zugang zu den low-level I/O-Operationen des Betriebssystems und erlaubt somit eine Kommunikation auf der Transportebene. Tiefergehende Recherchen zeigten jedoch, dass eine direkte Nutzung des Frameworks nicht effizient wäre. Begründet liegt dies darin, dass die NIO-API keine gute Zugänglichkeit bietet. Zudem stehen bereits sehr gute NIO-Frameworks zur Verfügung, welche das eigentliche Framework wegkapseln. Bedingt durch den zeitlich begrenzten Rahmen des Projekts wurde deshalb darauf geachtet, dass, wenn möglich, eine der bestehende Lösung genutzt wurde, anstatt eine gänzliche Neuentwicklung auf Basis von Java NIO zu betreiben.

5.1 NIO-Frameworks

Im nächsten Schritt wurde zunächst die Auswahl der möglichen Frameworks durch zwei notwendigen Eigenschaften beschränkt:

- Lauffähigkeit auf Android
- Frei / Kostenlos

Diese Eigenschaften wurden durch die Rahmenbedingungen gegeben und sind daher verbindlich für die verwendete Servertechnologie. Es stellte sich heraus, dass es drei wesentliche freie Frameworks im Bereich Java-NIO zu finden gab, welche qualitativ miteinander auf einer Stufe standen. Diese sollten nun genauer betrachtet und analysiert werden.

Apache MINA

Apache MINA ist eine ereignisbasierte Programmierschnittstelle für asynchrone Kommunikation. Sie wird von der Apache Software Foundation entwickelt. MINA unterstützt mehrere Kommunikationsprotokolle wie TCP und UDP und bietet ein anpassbares Thread-Modell. MINA unterstütz zudem die Sicherheitsmechanismen SSL, TLS und StartTLS.

Grizzly

Grizzly ist ein Java.net Projekt und ist zudem ein Teil der GlassFish-Server-Software. Der Fokus der Programmierschnittstelle liegt auf einer robusten und skalierbaren Kommunikationslösung. Zudem unterstützt Grizzly neben TCP und UDP Protokolle wie http/s oder Comet.

Netty

Netty, ehemals Teil von JBOSS.org ist ähnlich wie Apache MINA eine ereignisbasierte Programmierschnittstelle für asynchrone Kommunikation. Netty unterstützt von Grund auf Protokolle wie http, WebSocket, SSL, FTP, StartTLS, Google Protobuf und andere. Zudem ist es in Netty möglich, Daten bspw. via gzip zu komprimieren.

Da die Funktionalität der einzelnen Frameworks nicht groß voneinander abwich, bestand die Aufgabe des nächsten Schritts darin, festzustellen, inwiefern sich die Performance der einzelnen Frameworks unterschied. Bei den hierzu angestellten Recherchen stieß man auf einen Benchmark-Test[3], welcher unterschiedliche Frameworks miteinander verglich. Unter den verglichenen Frameworks fanden sich auch Mina, Grizzly und Netty. Innerhalb des Tests wurden Pakete fester Größer von einem Client zu einem simplen Echo-Server geschickt. Es wurde dabei untersucht, wie viele Daten pro Zeiteinheit übertragen wurden. Die veränderte Bedingung, auf der X-Achse markiert, war hierbei nicht die Zeit, sondern die Anzahl der konkurrierenden Verbindungen zum Server. Die Ergebnisse, welche mit verschiedenen Netzwerkkonfigurationen und Paketgrößen gemacht wurden, waren eindeutig. Vor allem im Bereich der Skalierbarkeit war die Performance beim Framework Netty am besten.

[3] Trustin Lee: „Performance Comparison between NIO Frameworks“, url: http://gleamynode.net/articles/2232/ (08.02.2012)

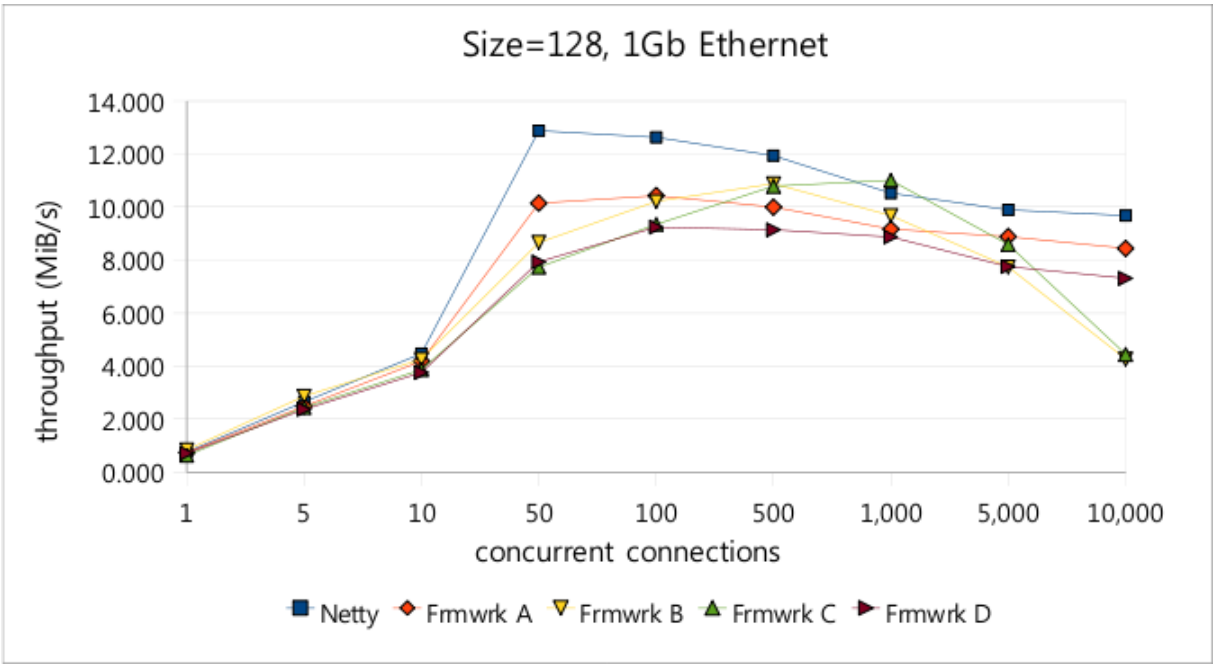

Abb. 3 – Vergleich von NIO-Frameworks (1GB Netzwerk, Paketgröße 128 bytes)

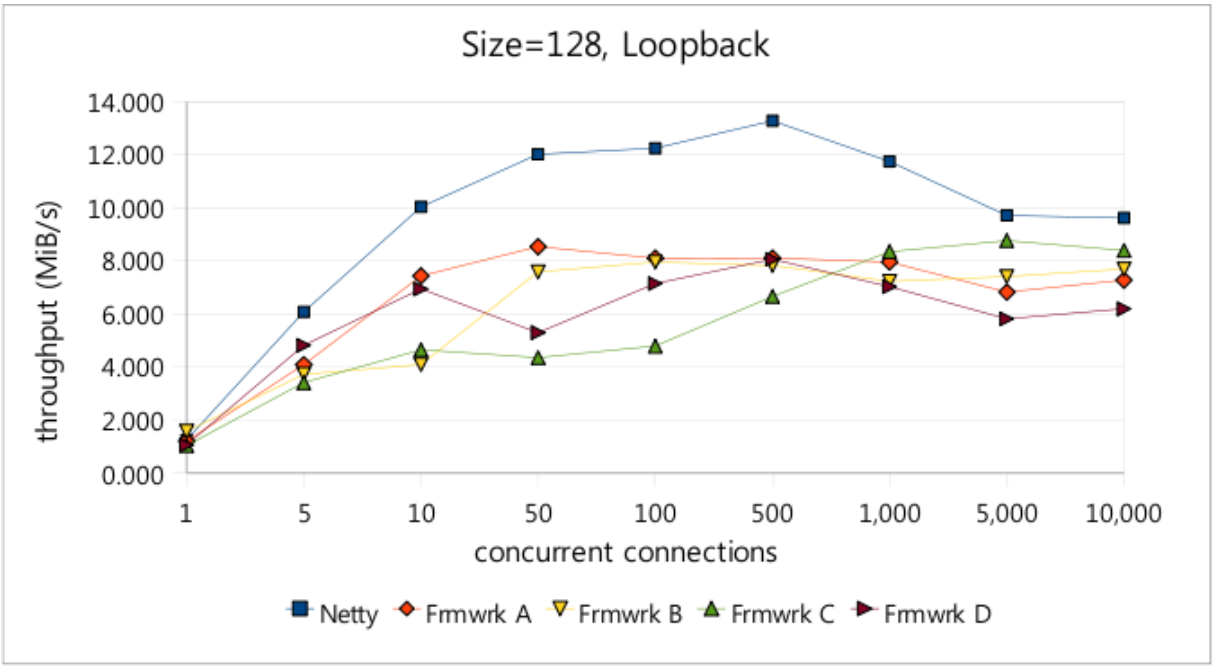

Abb. 4 – Vergleich von NIO-Frameworks (Loopback, Paketgrößen 128 bytes)

Das in der Anforderungsanalyse aufgestellt Qualitätsmerkmal der Skalierbarkeit war hier mit ausschlaggebend für die Entscheidung. Da Netty unter den NIO-Frameworks die beste Skalierbarkeit an den Tag legte und es vom Funktionsumfang den Anderen in nichts nach stand, wurde Netty als Grundlage weiterer Entwicklung ausgewählt. Um die Ergebnisse des Benchmarks nochmalig zu bestätigen wurden weitere Recherchen angestellt, welche die

Ergebnisse jedoch bekräftigten[4]. Der in den Recherchen gefundene Artikel kam im Weiteren zu dem Ergebnis, dass Netty gegenüber seinen Konkurrenten zu bevorzugen sei, da Netty in den Bereichen Performance, Speicherverbrauch und Funktionalität seinen Gegenspielern leicht voraus ist.

5.2 Verworfene Alternativen

Neben Java NIO existieren selbstverständlich noch andere Technologien zur Realisierung einer Client-Server-Kommunikation. In diesem Abschnitt werden die Vor- und Nachteile einiger dieser Alternativen beschrieben. Zudem wird darauf eingegangen, warum man sich stattdessen gegen diese Alternativen entschieden hat.

5.2.1 jWebSocket

jWebSocket[5] ist ein im Frühjahr 2010 in die Beta-Phase gegangenes Projekt, welches mithilfe der WebSocket Technologie schnelle bidirektionale Kommunikation im Web ermöglicht. jWebSocket ist gänzlich in Java entwickelt und bietet auch Client-Software für Android an. Da die WebSocket Technologie auf der Basis von TCP arbeitet, ist sie gegenüber anderen reinen http Ansätzen im Vorteil. Sie ist daher deutlich effizienter. Sie weist nicht den Overhead von http auf und benötigt statt zweier Verbindungen für eine Kommunikation nur einen Kanal. Dieser kann dazu genutzt werden um Daten gleichzeitig in beide Richtungen zu versenden. jWebSocket kann zudem leicht in einen bestehenden Java-EE Server wie JBOSS integriert werden. Nachteilig an dieser Lösung ist die schlechte Zugänglichkeit. Weder eine klare Dokumentation, noch guter Beispielcode ließen sich finden. Zudem sind die Erfahrungsberichte über diese Technologie rar gesät. Begründet liegt dies in dem jungen Alter der Technologie. Sie ist ein interessanter Ansatz, jedoch durch ihren jungen Projektstand und der dadurch resultierenden Mängel im Bereich der Zugänglichkeit, derzeit für den Anwendungsfall ungeeignet.

[4] Nicholas Hagen: Scalable NIO Servers – Part 1 – Performance, url: http://www.znetdevelopment.com/blogs/2009/04/07/scalable-nio-servers-part-1-performance/ (09.02.2012)
[5] jWebSocket: http://jwebsocket.org/ (05.02.2012)

5.2.2 Webservice

Alternativ ließe sich die Kommunikation auch über einen Webservice realisieren. Die Vorteile eines Webservice liegen in der Interoperabilität, da die Schnittstellen zu den Webservices in XML beschrieben werden. So ist es möglich, zwischen unterschiedlicher Hard- und Software zu kommunizieren, da alle die XML Definition der Schnittstelle lesen und verstehen können. Dieser Vorteil käme vor allem dann zu tragen, wenn das Projekt auf weitere Plattformen wie Apples iOS oder Windows Phone erweitert werden würde. Eine Kommunikation über einen Webservice ließe sich bspw. via Java API for XML Web Services (JAX-WS) realisieren. JAX-WS sendet hierbei die Nachrichten über das SOAP Protokoll. Ebenso möglich wäre die Nutzung eines Java Platform - Enterprise Edition (Java EE) Servers. Dieser bietet ebenfalls die Möglichkeit über Webservices auf Enterprise Java Beans (EJB) zuzugreifen. Zu den frei verfügbaren Java EE Implementierung zählen neben Anderen, Server wie JBOSS Application Server, GlassFish oder auch Apache Geronimo. Nachteilig an den Webservices ist der hohe Overhead, der durch das Parsen der XML-Dokumente entsteht. Zudem wird oftmals http selbst oder ein auf http aufbauendes Protokoll zur Übertragung verwendet. Der entstehende Overhead ist für den Anwendungsfall mobiler Anwendungen eher ungeeignet. Zudem ist im Rahmen dieses Projekts keine Ausweitung auf andere Systeme geplant, sodass der Vorteil der Interoperabilität wenig ins Gewicht fällt.

6. Netty

Zur Vorbereitung der Entwicklung mit Netty wurde ein genauerer Blick auf die Funktionsweise des Frameworks geworfen. Im Folgenden wird der grobe Aufbau beschrieben, um zu verdeutlichen, an welcher Stelle man mit der Entwicklung ansetzen kann. Zudem soll hier verdeutlicht werden, wie Netty intern funktioniert. Als Informationsquelle diente hierzu die offizielle Dokumentation[6]. Zu beachten ist bei Netty in erster Linie, dass alle I/O-Operationen asynchron erfolgen. Das bedeutet, dass Aufrufe ohne Garantie der fertigen Übertragung sofort zurückkehren. Stattdessen erhält man ein *ChannelFutur* Objekt, welches Auskunft über den aktuellen Stand der Operation gibt.

[6]Netty Project 3.2 User Guide: http://docs.jboss.org/netty/3.2/guide/pdf/netty.pdf (25.02.2012)

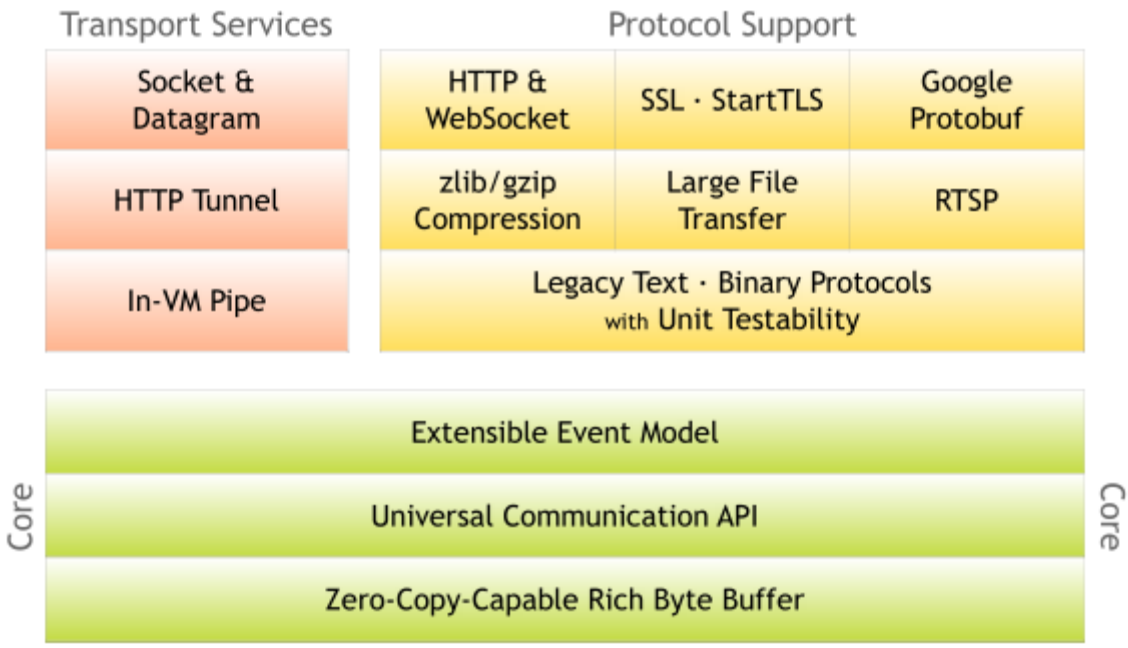

Abb. 5 – Netty-Architektur Übersicht

Nettys Kern baut sich in drei Schichten auf. Auf der untersten Ebene findet man die *Rich Byte Buffer*, welche alternativ zu den NIO *ByteBuffer* von Netty entwickelt wurde um Mängel an den NIO *ByteBuffer* auszumerzen und diese zu Optimieren. Der neu entwickelte *ByteBuffer* ist oftmals sogar schneller als das Java Pendant. Darauf aufgesetzt findet sich die Universal Communication API. Diese API kapselt mittels des Interfaces Channel alle Punkt-zu-Punkt Kommunikationen. Dies soll es dem Entwickler ermöglichen nachträglich auf ein anderes Transportprotokoll umzusteigen, ohne die gesamte Netzwerkschicht der Anwendung neu zu implementieren. Für einen Umstieg auf ein anderes Protokoll sind lediglich wenige Codezeilen nötig. So kann bspw. leicht von TCP auf UDP gewechselt werden oder aber auch von Javas OIO auf NIO. Die Universal Communication API dient als Grundlage für die dritte Netty Kernschicht, dass *Event Model*. Das *Event Model* ist ein erweiterbares *Event Model* und bedient sich hierzu einer Abwandlung des Intercepting Filter Pattern. Grundlage des Modells ist eine *ChannelPipeline,* welche eine Liste von *ChannelHandlern* verwaltet. Das Prinzip hierbei ist recht simpel. Sobald eine Nachricht, eine sogenannte *ChannelMessage*, gesendet oder Empfangen wird, durchläuft die Nachricht die Pipeline und damit alle Handler, welche der Pipeline hinzugefügt wurden. Hierbei sind die Reihenfolge und Flussrichtung essentiell.

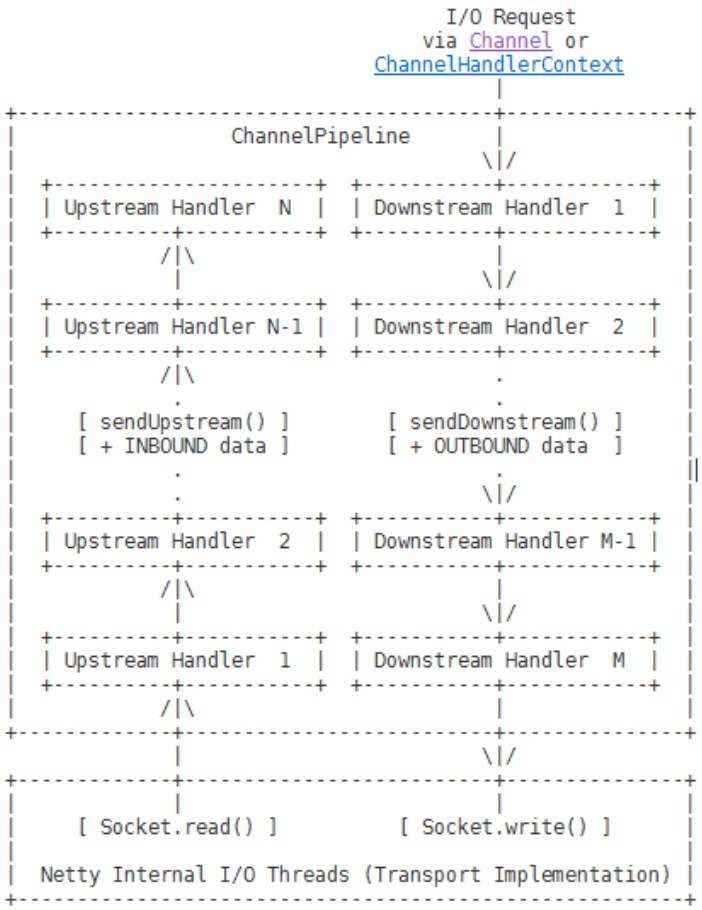

Abb. 6 – Netty ChannelPipeline

Die Art der Handler kann von Fall zu Fall variieren. Bspw. können sie zur Komprimierung verwendet werden oder aber auch um Daten zu verschlüsseln. Neben einer Palette für die gängigsten Anwendungsfälle ist es einfach möglich, selbst Handler zu implementieren und diese untereinander zu mischen. Das Verbindungsstück zur eigentlichen Programmlogik kann ebenfalls durch einen Handler realisiert werden. Bei der in Netty verwendeten Methode muss lediglich darauf geachtet werden, dass die Handler in der richtigen Reihenfolge in die Pipeline geschrieben werden. Die Verwendung ist somit leicht zugänglich und nachvollziehbar.

Netty unterstützt von sich aus eine breite Palette an typischen Handlern. Unter ihnen zu finden sind protokollabhängige Handler wie der *HttpContentDecoder* und Handler für bestimmte Verfahren, etwa die Verschlüsselung via SSL. Die Handler finden sich innerhalb der folgenden Netty Paketen:

- org.jboss.netty.handler.codec.base64
- org.jboss.netty.handler.codec.compression
- org.jboss.netty.handler.codec.embedder
- org.jboss.netty.handler.codec.frame
- org.jboss.netty.handler.codec.http
- org.jboss.netty.handler.codec.http.websocket
- org.jboss.netty.handler.codec.protobuf
- org.jboss.netty.handler.codec.replay
- org.jboss.netty.handler.codec.rtsp
- org.jboss.netty.handler.codec.serialization
- org.jboss.netty.handler.codec.string
- org.jboss.netty.handler.execution
- org.jboss.netty.handler.logging
- org.jboss.netty.handler.queue
- org.jboss.netty.handler.ssl
- org.jboss.netty.handler.stream
- org.jboss.netty.handler.timeout

6.1 Echo-Server Beispiel

Um die Funktionsweise von Netty zu verdeutlichen, soll an dieser Stelle die Entwicklung eines trivialen Echo-Servers durchgegangen werden. Der hier verwendete Beispiel-Code stammt aus der Dokumentation von Netty selbst[7].

```
ServerBootstrap bootstrap = new ServerBootstrap(
                  new NioServerSocketChannelFactory(
                          Executors.newCachedThreadPool(),
                          Executors.newCachedThreadPool()));
```

Code 1 – Erstellung eines ServerBootstrap-Objekts

Im der Main Methode wird zunächst ein *ServerBootstrap* Objekt erzeugt. Dieser *ServerBootstrap* ist eine Helferklasse, welche einen neuen *Channel* erzeugt und dann auf Verbindungen von außen wartet. Der *Channel* ist hierbei ein Verknüpfungspunkt zu einem Socket oder einer anderen IO fähigen Komponente. Er kapselt die Kommunikation zum Socket weg und

[7] JBOSS: http://docs.jboss.org/netty/3.2/xref/org/jboss/netty/example/echo/package-summary.html (12.02.2012)

wird mithilfe einer *ChannelFactory* erzeugt. Der Typ der *ChannelFactory* bestimmt die Art der Kommunikation. In diesem Fall wird die *NioServerSocketChannelFactory* genutzt und somit eine NIO Socketverbindung aufgebaut. Die Factory bedient sich bei ihrer Arbeit zweier *Executors*, einen Boss- und einen Worker-*Executor*. Die *Executors* haben den Zweck *Runnables* auszuführen. Dies wird hier in beiden Fällen durch *ThreadPools* realisiert. So legt der Sever dann für jeden geöffneten Port ein Boss-Thread an, welcher ankommende Verbindungen annimmt und diese an einen Worker-Thread weitergibt. Der Worker-Thread wiederum führt dann die Schreib- und Leseoperationen durch.

```
bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
            public ChannelPipeline getPipeline() throws Exception {
                return Channels.pipeline(new EchoServerHandler());
            }
        });
```

Code 2 – Erstellung einer ChannelPipeline

Im nächsten Schritt wird dann die Pipeline durch eine weitere Factory angelegt. In diesem simplen Echo-Server wird ihr lediglich ein *Handler* zugewiesen. Der *EchoServerHandler* ist in diesem Fall eine eigene geschriebene Klasse, welche vom *SimpleChannelUpstreamHandler* erbt. An dieser Stelle müssen die jeweiligen Handler in der richtigen Reihenfolge in die Pipeline geschrieben werden, damit der Server ordnungsgemäß funktioniert. Dieser Umstand ist hier zu vernachlässigen, da es sich nur um einen Handler handelt. Zum Abschluss der Main-Methode wird dann der Bootstrap an den entsprechenden Port gebunden und somit ein Boss-Thread angelegt.

```
bootstrap.bind(new InetSocketAddress(8080));
```

Code 3 – Öffnen eines neuen Ports

Hauptbestandteil des Handlers ist die Methode *MessageReceived(…)*, welcher neben dem Kontext eine Nachricht in Form eines *MessageEvent* übergeben wird. Da es sich hierbei um einen simplen Echo-Server handelt, wird die Nachricht direkt wieder in den Channel geschrieben und somit zum Client zurück gesendet. Ein richtiger Handler sollte an dieser Stelle stattdessen die Nachricht verarbeiten oder möglicherweise für andere *Handler* aufbereiten.

```
@Override
public void messageReceived(
                ChannelHandlerContext ctx, MessageEvent e) {
            // Send back the received message to the remote peer.

        e.getChannel().write(e.getMessage());
       }
```

Code 4 – Implementierung der messageReceived methode

Der Client ist ähnlich aufgebaut wie der Server. Im Unterschied zum Server nutzt dieser jedoch einen *ClientBootstrap* anstatt eines *ServerBootstrap*. Dieser gibt dem Client nach dem Verbinden das *ChannelFuture* Objekt zurück. Mit diesem Objekt ist der Client in der Lage, den Status der Verbindung zu überwachen und etwaige Aktionen durchzuführen, bspw. wenn die Verbindung abgebrochen ist. Nun muss der Client nur eine Nachricht absenden und erhält sie umkehrend zurück.

7. Kommunikationsmodell

Wie schon im Kapitel „Praxisprojekt“ angedeutet und im Kapitel „Zielsetzung“ formuliert, erfordert die Umsetzung der Idee von YourSights den Einsatz einer Client-Server-Architektur. Für den Aufbau einer verständlichen Kommunikation zwischen den Partnern war es von Nöten, die Inhalte, der zu sendenden Daten zu bestimmen und Regeln zu definieren, wie diese Inhalte ausgetauscht werden. Es musste ein Protokoll entwickelt werden. Die Entwicklung selbst unterteilt sich dabei in vier Phasen[8]:

1. Anforderungen und Anforderungsanalyse
2. Design
3. Implementierung
4. Testen und verifizieren

Miroslav Popovic beschreibt die Aufgabe der ersten Phase anhand von zweier rhetorischer Fragen[9]: *„What must be done? How can the solution be verified?“*
In dieser Phase müssen also die eigentlichen Aufgaben der Kommunikation bestimmt werden, genauso wie Überlegungen angestrengt werden müssen, welche Fragen formuliert und welche Antworten erwartet werden.

[8] *Vgl. Popovic (Communication Protocol Engineering, 2006)*
[9] *Vgl. Popovic (Communication Protocol Engineering, 2006), Seite 9*

In der Design-Phase wird aus den Ergebnissen der ersten Phase ein Modell synthetisiert, welches die Kommunikation wiederspiegelt. Auch das Verhalten wird in dieser Phase genau definiert. Nachdem diese rein theoretischen Arbeiten abgeschlossen wurden, kann die technische Umsetzung erfolgen, welche in der letzten Phase getestet und verifiziert wird. In diesem Kapitel wird nur auf die ersten beiden Phasen eingegangen. Die Entwicklung wird in Kapitel 8 beschrieben und auf die letzte Phase wird im Kapitel über die Serverumgebung eingegangen.

7.1 Anforderungen und Anforderungsanalyse

Für die Bestimmung der Anforderungen mussten alle Funktionalitäten des Clients unter die Lupe genommen werden. Hierzu wurde auf die Funktionsübersicht[10] aus Kapitel 4 der Praxisprojektdokumentation zurückgegriffen. Die folgende gefilterte Tabelle enthält Prozesse, deren Ausführung sowohl Client, als auch Server involviert. Dabei entspringen die Handlungen des Servers der logischen Notwendigkeit für eine erfolgreiche Ausführung des Prozesses.

Client – Aktion	Server – Reaktion
Veröffentlichen	Nimmt eine neue Tour entgegen, speichert sie und stellt sie anderen Nutzern zur Verfügung
Touren bearbeiten	Eine vorhandene Tour wird durch den Client aktualisiert
Touren anzeigen (Paketbrowser)	Liefert eine Liste von verfügbaren Touren
Informationen anzeigen	Liefert detaillierte Informationen zu einer Tour
Tour anfordern	Liefert dem Client eine bestimmte Tour
Registrierung	Erzeugt einen neuen Nutzer im System
Authentifizierung	Verifiziert die Identität des Nutzers
Bewerten und Kommentieren	Speichert eine Bewertung und/oder Kommentar zu einer Tour
Sortieren, Filtern und Suchen	Liefert eine sortierte und/oder gefilterte Liste von verfügbaren Touren

Tab. 1 – Übersicht Aktion-Aufgabe von Client und Server

[10] *Vgl. Ruthmann/Stratmann (2012), ab S. 20*

7.1.1 Frage-Antwort-Paradigma

Grundlegend für die Entwicklung des Protokolls für die Kommunikation war die Frage, wer die Kommunikation initiiert, was gesendet wird und wie sich die Kommunikationspartner in jeder Situation verhalten.

Eine Betrachtung der notwendigen Reaktionen des Servers führte zu dem Schluss, dass kein Bedarf an einer Initiierung der Kommunikation durch den Server besteht. Diese entspringt stets dem Aufgabenbereich des Clients. Aus diesem Grund wurde der Client als alleiniger Initiator jeglicher Kommunikation bestimmt. Der Server verarbeitet demnach die Anfragen des Clients und vermittelt, wenn nötig, aufbereitete Informationen zwischen dem Client und einer hinterlegten Datenbank.

Es wird bei allen Anfragen eine Antwort seitens des Servers erwartet, welche über den Erfolg oder Misserfolg der Verarbeitung informiert. Dabei kann es sich um eine einfache Bestätigung (bspw. über den Erfolg/Misserfolg der Eintragung eines neuen Kommentars) oder aber auch um eine detaillierte Antwort handeln (Rückgabe einer gefilterten Liste von Touren). Eine tiefergehende Beschreibung der einzelnen Anfragen und Antworten kann den nachfolgenden Kapiteln entnommen werden. An dieser Stelle sei nur angemerkt, dass eine Anfrage stets eine Antwort auslöst. Trifft eine Antwort gar nicht oder mehrfach ein, oder kann eine eintreffende Antwort nicht einer entsprechenden Anfrage zugeordnet werden (siehe Kapitel *10.1*), befindet sich die Kommunikation in einem fehlerhaften Zustand.

7.2 Design

Die Tabelle aus dem vorherigen Kapitel dient als Vorlage für das Kommunikationsdesign. An dieser Stelle werden den Funktionen entsprechende Fragen und Antworten zugeordnet.

- Veröffentlichen

  Meint die Aktion des Clients, eine erstellte Tour anderen zugänglich zu machen. Dafür muss sie zentral durch den Server gespeichert werden. An dieser Stelle wird die eigentliche Tour an den Server gesendet. Zugehörige Meta-Informationen, wie bspw. der Tourname und der Uploader, bzw. Autor der Tour müssen ebenfalls mitgesendet werden. Hier muss die Identität des Clients verifiziert werden, um einen Missbrauch auszuschließen. Eine Antwort seitens des Servers signalisiert dem Client, ob die Operation erfolgreich war.

- Touren bearbeiten

  Meint die Aktion, eine bereits existierende Tour zu ändern. Eine Online-Bearbeitung, also eine direkte Bearbeitung einer Tour auf dem Server ist nicht möglich, da der Server hierfür die entsprechende Tour laden und öffnen müsste. Zusätzlich müsste eine Steuerung entwickelt werden, die es dem Client erlaubt, die gewünschten Änderungen vorzunehmen. Dies ist nicht Bestandteil der ursprünglichen Projektidee und wird somit verworfen. Eine andere mögliche Lösung läge in einem Upload einer Tour, welche eine bereits vorhandene ersetzt. Da dies dem Punkt „Veröffentlichen“ sehr ähnelt, wird dieser Punkt in der Entwicklung nicht berücksichtigt und hier nur der Vollständigkeit halber aufgeführt.

- Touren anzeigen (Paketbrowser)

  Hier wird dem Client eine Liste mit verfügbaren Touren offeriert. Ein Eintrag dieser Liste besteht aus den Meta-Informationen, die für eine Tour hinterlegt wurden. Entsprechend dem Datenbankschema[11] aus der Praxisprojektdokumentation wurden diese Informationen um bspw. eine Bewertung, einem Datum oder die Anzahl der Downloads erweitert, was für den Client durchaus von Interesse sein kann. An dieser Stelle wurde außerdem eine Grundsatzentscheidung getroffen. So stellte sich anfangs die Frage, ob eine angeforderte Tourliste persistent auf dem Handy gespeichert oder stets bei Bedarf neu angefordert wird. Unter der Vermutung, dass die Anzahl von uninteressanten Touren, die Anzahl der interessanten Touren bei weitem übersteigt, wurde sich für die letzte Herangehensweise entschieden. Dadurch wird auch die unnötige Belegung von Speicherplatz verhindert und die Tourliste muss nicht in regelmäßigen Abständen aktualisiert werden.

- Informationen anzeigen

  Damit ist die Anzeige von detaillierten Informationen zu einer Tour gemeint. Der Client muss an dieser Stelle dem Server mitteilen, welche Tour für ihn von Interesse ist und eine entsprechende Antwort erhalten. Diese umfasst die genaue Beschrei-

[11] Vgl. Ruthmann/Stratmann (2012), Seite 37

bung und ein Bild. Da der Down- bzw. Upload zunächst eine höhere Priorität genießt, wurde auf diese Funktionalität aus Zeitgründen vorerst verzichtet.

- Tour anfordern:

  Der Client fordert den Download einer bestimmten Tour vom Server an. Die Antwort des Servers muss neben dem eigentlichen Tourpaket, welches der Client für die Ausführung benötigt, auch die Meta-Informationen dieser Tour beinhalten.

- Registrierung

  Der Authentifizierung muss eine Registrierung vorrausgehen, die eine Eintragung von Informationen über den neuen Nutzer auslöst. Der Inhalt besteht dabei aus den Daten, die im Datenbankschema für einen Nutzer vorgesehen sind. Eine Bestätigung für eine Registrierung muss durch den Server erfolgen, bzw. ein Fehlerbericht gesendet werden, aus welchem Grund die Registrierung nicht erfolgreich war. Dies könnte bspw. daran liegen, dass der gewünschte Benutzername bereits vergeben ist oder nicht unterstütze Zeichen in ihm enthalten sind.

- Authentifizierung

  Gerade für Funktionen, in denen personenbezogene Daten verarbeitet werden, muss eine Authentifizierung erfolgen. Diese enthält dabei nur den Nutzernamen und das Passwort des Benutzers. Über eine einfache Bestätigung oder Ablehnung dieser Kombination wird der Client informiert.

- Bewerten und Kommentieren

  Die Funktionalität gibt dem Client die Möglichkeit, seine Meinung über eine Tour zu äußern oder mit anderen Nutzern in Kontakt zu treten. Dabei soll eine Bewertung aus zwei Teilen bestehen, einem numerischen Wert, der die Tour bewertet und eine textuelle Beschreibung der Bewertung. Eine Antwort seitens des Servers sollte die erfolgreiche Eintragung der Bewertung quittieren oder über den Fehlschlag informieren. Dabei ist ein detaillierter Fehlerbericht vorerst nicht von Nöten. Da eine Bewer-

tung im Grunde genommen ein Kommentar ist, welcher um einen numerischen Wert erweitert wurde, erfolgt im Rahmen dieser Arbeit nur die Entwicklung der Bewertung.

- Sortieren, Filtern und Suchen

  Da die Liste der verfügbaren Touren schnell wachsen und unübersichtlich werden kann, muss dem Nutzer die Möglichkeit gegeben werden, nach bestimmten Touren zu suchen, also die Menge zu filtern und die Ergebnisse zu sortieren. Durch ihren sachlichen Bezug zum Anfordern einer Tourliste, werden diese Kriterien auch Teil dieser Nachricht. Wenn keine Menge von Touren bestimmt werden kann, für die diese Kriterien zutreffend sind, muss der Client informiert werden.

7.2.1 Nachrichtenaustausch

Im folgenden Diagramm wird der Ablauf einer möglichen Kommunikation dargestellt. Dieser umfasst die Aktionen Registrierung, Login, bzw. Authentifizierung, sowie Anfordern einer Tourliste, eines Tourpakets und zugehörigen Bewertungen.
Ebenfalls Bestandteil ist das Abgeben einer Bewertung für ein Paket und das Vorbereiten und die Durchführung eines Uploads. Dabei wurden den gesendeten Nachrichten möglichst eindeutige Bezeichnungen zugeordnet. Um einer Verwirrung durch die verwendete Terminologie zu verhindern sei an dieser Stelle angemerkt, dass die Termini „Nachricht“ und „Paket“ stets eine Nachricht bezeichnen, die zwischen Client und Server ausgetauscht wird und analog verwendet werden können. Der abwechselnde Gebrauch soll lediglich den Bezug zur informellen oder technischen Ebene verdeutlichen.

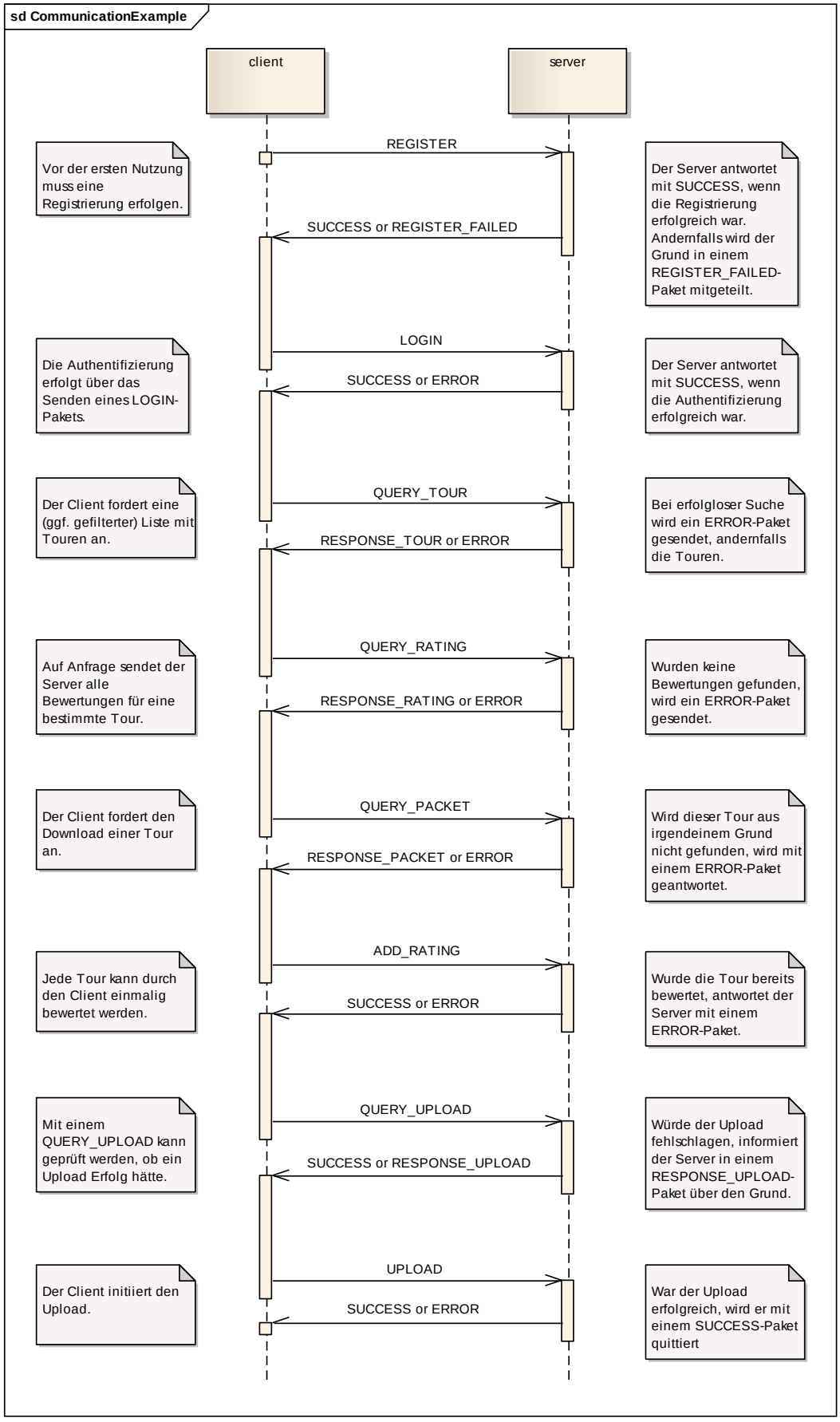

Abb. 7 – Kommunikationsbeispiel Client - Server

8. Übergreifende Entwicklung

Dieses Kapitel beschreibt die erste Stufe der Entwicklung. Es dokumentiert die Implementierung des Kommunikationsprotokolls, bzw. die Entwicklung der Kommunikationskomponenten an sich. Da der Begriff „Kommunikation“ eine doch sehr weitläufige Bedeutung inne hat, wird an dieser Stelle zuerst beschrieben, was die Kommunikationskomponente leisten, bzw. welche Aufgaben sie übernehmen soll. Dabei wurde versucht, einen kurzen Überblick[12] der zwischenmenschlichen Kommunikation von Bernhard Sandkühler auf die Kommunikation zwischen zwei Rechnersystemen zu übertragen.

Er sieht in den wichtigsten Bestandteilen zunächst einen Sender und einen Empfänger. Entsprechend dem Frage-Antwort-Paradigma lassen sich die Beteiligten mit Client und Server beschreiben. Es muss also ein Aufgabenteil sein, diese Rollen zu schaffen. Dabei ist der Client die Instanz, welche sich mit dem Server verbindet, ihn kontaktiert, Anfragen sendet und seine Antworten entgegen nimmt. Dies geschieht über die zur Verfügung stehende Netzwerkverbindung. Dabei kann es sich um eine WLAN-Verbindung, eine Verbindung über das integrierte Modem des Smartphones oder eine jegliche, kabelgebundene Verbindung handeln, die das IP-Protokoll[13] unterstützt (auf Grund der Datenübertragung über ein Netzwerk wurde der Paketname *net* gewählt). Eine Sende- und Empfangsbereitschaft des Servers und des Clients muss sichergestellt werden. Wurden diese Punkte erfüllt, können **Daten gesendet und empfangen** werden.

Ein weiterer wichtiger Bestandteil des Kommunikationspaketes sind die Nachrichten, bzw. die Pakete, die über das Netzwerk gesendet und empfangen werden sollen. Es muss die Möglichkeit geschaffen werden, **Kommunikations-Inhalte zu erzeugen**. Die geschieht in der dritten Phase von Popovics Protokoll-Entwicklung[14], die im Anschluss beschrieben wird.

Die Verknüpfung dieser beiden zentralen Aufgaben erfolgt durch den dritten und letzten Bestandteil des Kommunikationspaketes, die **Aufbereitung von Daten**. Die Programmlogik der übrigen Programmkomponenten hält und verarbeitet Daten üblicherweise in einer anderen Form, als sie für die Übertragung über das Netzwerk benötigt wird. Demnach muss ein Verantwortlicher die Formatierung der Daten übernehmen und ihre Anordnung innerhalb

[12] In Anlehnung an Sandkühler, (Bestandteile von Kommunikation, 2005)
[13] Vgl. RFC 791, bzw. RFC 2460
[14] Vgl. Popovic (Communication Protocol Engineering, 2006), Seite 9

der Kommunikation festlegen, damit der Kommunikationspartner auch in der Lage ist, die ankommende Information richtig einzuordnen. Diese Funktionalität wird nicht alleine durch einen Programmteil bewerkstelligt, sondern ist Bestandteil des Protokolls und *Netty* selbst. Im Rahmen der Implementierung des Protokolls und der Einbettung des *Netty*-Frameworks wird dies detaillierter beschrieben.

8.1 Implementierung des Protokolls

Im Folgenden soll der Paketaufbau der einzelnen Frage- und Antwortpakete detailliert beschrieben werden. Der Vollständigkeit halber sei zu erwähnen, dass lediglich der Aufbau der Pakete auf Applikationsebene beschrieben wird. Ihnen vorangestellt sind die benötigten Header der Übertragungsprotokolle. Zum einen ist hier der TCP-Header (Transmission Control Protocol) zu nennen. Da wir eine Stream-Socket-basierte Kommunikation verwenden geht eine Übertragung über das TCP-Protokoll damit einher. Die Größe eines TCP-Headers umfasst maximal 60 Byte[15] und ist nicht in der Größenangabe der folgenden Pakete enthalten. Ein *BinaryPacket* (s. u.) bildet den Payload eines TCP-Pakets. Dem fertigen TCP-Paket wird noch ein IP-Header (Internet Protocol) vorangestellt. Dieser umfasst ebenfalls max. 60 Byte[16] (in IPv4) und ist nicht in den Größenangaben enthalten.

8.1.1 Paketverpackung – Das BinaryPacket

Für den Informationsaustausch zwischen Client und Server wurde eine Vielzahl von Frage- und korrespondierenden Antwortpaketen erstellt. Da der Inhalt der Pakete i.d.R. nicht gleich ist, und somit auch keine einheitliche Signatur aufweist, ist es von Nöten, sie in einem vorangestellten Paket zu verpacken, welches die relevanten Informationen über den enthaltenen Pakettyp bereitstellt. Die Alternative läge im versuchten Type-Cast[17] auf jedes mögliche Paket, was äußerst ineffizient wäre.

Die Klasse BinaryPacket dient als „Verpackung“ für die Frage- und Antwortpakete und enthält Informationen über das verpackte Paket. Diese Informationen umfassen folgende Felder:

[15] Vgl. (Transmission Control Protocol, RFC 793)
[16] Vgl. (Internet Protocol, RFC 791)
[17] Ein Type-Cast meint eine Typenumwandlung. Hier wäre es der Versuch aus der Byte-Menge ein Nachrichtenpaket zu generieren, wobei es von Nöten wäre, alle Pakete durchzuprobieren, bis eine Umwandlung Erfolg hat. Dabei kann es allerdings auch passieren, dass eine Umwandlung Erfolgreich war, doch es sich nicht um das richtige Paket handelt und somit auch falsche Informationen enthalten sind.

- *Typ* (1 Byte): Dieses Feld identifiziert die Art des verpackten Paketes. Anhand des Typs wird versucht, die entsprechenden Nutzdaten zu decodieren.
- *Version* (1 Byte): Das Versionsfeld nimmt die Versionsnummer des verpackten Pakets auf. Im Laufe der Entwicklung sind Änderungen oder Erweiterungen am Paketaufbau nicht unwahrscheinlich. Zugunsten der Abwärtskompatibilität wurde dieses Feld eingerichtet und informiert die entsprechende Instanz, welche Decoder-Version für die Dekodierung herangezogen werden muss.
- *PacketID* (4 Byte): Dieses 4 Byte große Integer-Feld enthält eine eindeutige Nummer für die Identifizierung und Zuordnung des Pakets. Anhand dieser Nummer kann auf Client-Seite ein wartender Listener informiert werden. Der Server setzt die *PacketID* seines Antwortpaketes auf den Wert der *PacketID* der Anfrage.
- *Payload_Length* (4 Byte): Dieses Integer-Feld enthält die Länge des verpackten Paketes.
- *Payload* (max. 2^{31}-1 Byte): Das eigentliche Paket mit den relevanten Nutzdaten in Byte-Form.

Die Enkodierung eines BinaryPackets erfolgt mittels eines *ChannelBuffers*, welcher von Netty bereitgestellt wird und die zu sendenden Nutzdaten aufnimmt. Er erlaubt das Lesen und Schreiben von Basisdatentypen, wie Zahlen, einzelnen Zeichen und Bytes. Dies kann als Voraussetzung für die „Verpackung“ betrachtet werden. Komplexere Datentypen müssen manuell vorbereitet werden. Anhand einer Montageanleitung, dem *BinaryPacketEncoder*, ist Netty in der Lage, ein gegebenes BinaryPacket in eine übertragbare Form zu konvertieren. Auf das Encoder-/Decoder-Prinzip wird in Kapitel 8.2 genauer eingegangen.
Entsprechend der oben genannten Bestandteile sieht der Aufbau eines BinaryPackets folgendermaßen aus:

Typ (1 Byte)	**Version (1 Byte)**	**PacketID (4 Byte)**	**Payload_Length (4 Byte)**	**Payload (max. 2^{31}-1 Byte)**

Tab. 2 – Paketaufbau BinaryPacket

Die Dekodierung erfolgt analog anhand einer Demontageanleitung, durch welche Netty in der Lage ist aus dem Datenstrom wiederum ein BinaryPacket zu erstellen.

Es wäre auch möglich das BinaryPacket als ganzes Objekt mittels der Serialisierungs-Technologie zu versenden, welche Java von Haus aus bereitstellt. Deren wesentlicher Nachteil liegt jedoch in dem massiven Overhead, der von Java erzeugt wird. Dieser ergibt sich aus zusätzlichen Informationen über die einzelnen Bestandteile, auf die bei der Wiederherstellung zurückgegriffen wird. Die von uns verwendete Art der Informationskodierung erfordert nur minimalen Overhead, nämlich das BinaryPacket, und ist somit die effizienteste. Der Nachteil liegt darin, dass die Montage- und Demontageanleitung vom Programmierer erstellt werden müssem. Bzgl. der Geschwindigkeitsvorteile hat Bruno de Carvalho in seinem Netty-Tutorial[18] Stellung bezogen und diese empirisch belegt. Bei der Implementierung kommt ein nützlicher Bestandteil *Nettys*, der *ReplayingDecoder* zum Einsatz, auf den später genauer eingegangen wird.

8.1.2 Paketübersicht

Zur Wiederholung werden hier nochmals alle Pakete aufgeführt, die im Rahmen der Entwicklung modelliert und implementiert wurden. Die Pakete wurden dem Sequenzdiagramm aus Abb. 7.2.1.1 entnommen und die Bezeichnungen beibehalten.

Client	Beschreibung
REGISTER	Dient der Registrierung eines neuen Anwenders
LOGIN	Authentifiziert einen Nutzer am System
QUERY_UPLOAD	Prüft ob eine Tour veröffentlich werden kann
UPLOAD	Dient dem Veröffentlichen einer Tour
ADD_RATING	Dient der Bewertung einer Tour
QUERY_TOUR	Fordert eine (gefilterte und sortierte) Liste von Touren an
QUERY_RATING	Fordert die Bewertungen zu einer Tour an
QUERY_PACKET	Fordert eine bestimmte Tour an

Tab. 3 – Übersicht Anfragepakete (Client)

[18] Carvalho, (Netty tutorial – High Speed custom codecs with ReplayingDecoder, 2010)

Der Server antwortet mit entsprechenden Paketen:

Server	Beschreibung
SUCCESS	Bestätigt einer Aktion des Clients
ERROR	Informiert den Client über einen Fehlschlag
REGISTER_FAILED	Informiert den Client über den Grund einer fehlgeschlagenen Registrierung
RESPONSE_TOUR	Liefert eine Liste von Touren
RESPONSE_RATING	Liefert eine Liste von Bewertungen zu einer Tour
RESPONSE_PACKET	Liefert eine Tour
RESPONSE_UPLOAD	Informiert den Client warum ein Upload fehlschlagen würde

Tab. 4 – Übersicht Antwortpakete (Server)

Da ein BinaryPacket seine Payload stets in Form eines Byte-Arrays erwartet, ist in jedem Paket eine entsprechende Kodierungsanleitung enthalten. Die Dekodierung erfolgt durch einen Konstruktor in jeder Paketklasse, der versucht, aus einem Byte-Array eine Instanz seiner Klasse zu erzeugen. Zudem enthält jedes Paket seinen Typ und seine Versionsnummer. Daher schreiben die Interfaces *ClientPacket* und *ServerPacket* die Implementierung der Methode *createBinaryPacket()* vor. Diese Methode stößt, wenn nicht bereits geschehen, die Kodierung zu einem Byte-Array an und erzeugt ein BinaryPacket, welches später versendet werden kann. Der Unterschied zwischen den Interfaces liegt in der parametrisierten Version der *createBinaryPacket*-Methode im *ServerPacket*-Interface. Ihr wird die Packet-ID aus dem BinaryPacket der Anfrage übergeben, damit die Antwort auf Seite des Clients zugeordnet werden kann.

8.1.3 Paketdetails (Client)

Alle folgenden Pakete werden vom Client generiert; niemals vom Server. Sämtliche Pakete sind dennoch beiden Seiten bekannt, um eine Dekodierung vornehmen zu können.

REGISTER (Version 1)

Ein REGISTER-Paket fordert den Server auf, einen neuen Nutzer anzulegen. Ein Nutzer besteht zurzeit aus den drei Strings *Nickname*, *Password* und *Email*. Das Passwort sollte nicht im Klartext versendet werden. Vor dem Versenden sollte das Passwort nach einem gängigen Hashing-Verfahren kodiert werden. Die Längen der drei Felder werden in Form von Integer-Werten gespeichert, um sie korrekt aus dem Byte-Strom wiederherstellen zu können.

Aufbau:

sizeOfNick-name (4 Byte)	sizeOfPassword (4 Byte)	sizeOfEmail (4 Byte)	Nickname (n Byte)	Password (n Byte)	Email (n Byte)

Tab. 5 – Paketaufbau REGISTER

LOGIN (Version 1)

Ein Login-Paket dient mehr der Verifikation der Anmeldedaten, als einer Session-erzeugenden Authentifizierung. Aufgrund des eher geringen Maßes an Kommunikation zwischen Client und Server kamen wir zu dem Schluss, dass eine, auf einer Session beruhende Kommunikation und der damit einhergehende Verwaltungsaufwand seitens des Servers, keinen Mehrwert mit sich bringt. Dennoch birgt ein Login Vorteile für den Client. Durch die Prüfung der Anmeldedaten kann auf eine aufwendige Fehlerbehandlung bzgl. fehlerhafter Anmeldedaten innerhalb des Programms verzichtet werden. Ein LOGIN-Paket besteht lediglich aus dem Nutzernamen und dem Passwort und den zugehörigen Feldern, die deren Länge enthalten. Aufbau:

sizeOfNickname (4 Byte)	sizeOfPassword (4 Byte)	Nickname (n Byte)	Password (n Byte)

Tab. 6 – Paketaufbau LOGIN

QUERY_UPLOAD (Version 1)

Ein QUERY_UPLOAD-Paket sollte stets vor einem UPLOAD-Paket gesendet werden. Es enthält die Felder *Nickname, Password* und *Tourname*. Der Zweck des Paketes ist es, beim Server anzufragen, ob das Eintragen einer Tour mit dem Namen *Tourname* und den entsprechenden Anmeldeinformationen möglich ist. Da ein Upload einer Tour ggf. viel Zeit kosten

kann, kann auf diese Weise zunächst die Erfolgsaussicht bestimmt werden. Es ist jedoch nicht zwingend notwendig einem UPLOAD-Paket ein QUERY_UPLOAD-Paket vorrauszuschicken. Dies hat neben dem wegfallenden Verwaltungsaufwand auch den Vorteil, dass ein bspw. unterbrochener Upload direkt erneut gestartet werden kann, ohne ihm wieder ein QUERY_UPLOAD-Paket voraus zuschicken. Aufbau:

sizeOf-Nickname (4 Byte)	Nickname (n Byte)	sizeOfPassword (4 Byte)	Password (n Byte)	sizeOf-Tourname (4 Byte)	Tourname (n Byte)

Tab. 7 – Paketaufbau QUERY_UPLOAD

UPLOAD (Version 1)

Ein UPLOAD-Paket entspricht in Form und Aufbau größtenteils dem eines QUERY_UPLOAD-Pakets mit der einzigen Erweiterung, dass die eigentliche Tour und ein zugehöriger Wert, der die Größe speichert, hinten angefügt wird. Aufbau:

sizeOf-Nickname (4 Byte)	Nickname (n Byte)	sizeOfPassword (4 Byte)	Password (n Byte)	sizeOf-Tourname (4 Byte)	Tourname (n Byte)	sizeOfPayload (4 Byte)	Payload (n Byte)

Tab. 8 – Paketaufbau UPLOAD

ADD_RATING (Version 1)

Ein ADD_RATING-Paket enthält die Bewertung für eine Tour. Diese umfasst eine numerische Bewertung und einen zugehörigen Kommentar. Die Bewertung kann anhand der Tour-ID einer bestimmten Tour zugeordnet werden. Da eine Bewertung nur von einem registrierten Benutzer, und auch nur einmal, abgegeben werden darf, muss wiederum eine Authentifizierung erfolgen. Die hierfür benötigten Daten werden ebenfalls mit dem Paket mitgesendet. Aufbau:

sizeOf-Nickname (4 Byte)	Nickname (n Byte)	sizeOf-Password (4 Byte)	Password (n Byte)	tourID (4 Byte)	sizeOf-Rating (4 Byte)	rating (n Byte)	sizeOf-Comment (4 Byte)	comment (4 Byte)

Tab. 9 – Paketaufbau ADD_RATING

QUERY_TOUR (Version 1)

Ein QUERY_TOUR-Paket fordert vom Server eine Liste mit Touren an, die zum Download bereit stehen. Dieses Paket ermöglicht es auch dem Benutzer, die Suche anhand von Suchbegriffen einzugrenzen. Momentan kann nach einem Tourbezeichner, einem Autor oder einem Ort gesucht werden. Dabei wird standardmäßig absteigend sortiert. Die Sortierreihenfolge kann jedoch ebenfalls geändert werden. Sortiert werden kann außerdem nach dem

Tournamen, Autor, Ort und Einstellungsdatum. Ob und welche Suchparameter genutzt werden und welche Sortierreihenfolge verwendet werden soll, wird in einer Bit-Maske (Key) gespeichert.

Aufbau (die Felder *name, author* und *location* sind optional und werden nur geschrieben, wenn auch ein entsprechender Suchbegriff vorgegeben ist):

key (4 Byte)	sizeOf-Name (4 Byte)	Name (n Byte)	sizeOf-Author (4 Byte)	Author (n Byte)	sizeOf-Location (4 Byte)	Location (n Byte)

Tab. 10 – Paketaufbau QUERY_TOUR

QUERY_RATING (Version 1) & QUERY_PACKET (Version 1)

Ein QUERY_RATING-Paket fordert vom Server eine Liste mit Bewertungen zu einer bestimmten Tour an. Die Tour wird durch ihre Tour-ID identifiziert. Aufbau:

tourID (4 Byte)

Tab. 11 – Paketaufbau QUERY_RATING bzw. QUERY_PACKET

Eine Tour wird ebenfalls über ihre ID vom Server angefordert. Daher unterscheidet sich ein QUERY_PACKET-Paket nicht von einem QUERY_RATING-Paket. Die Differenzierung erfolgt durch den Typ, welcher im BinaryPacket gesetzt wird.

10.1.4 Paketdetails (Server)

Ähnlich wie auf dem Client, erfolgt eine Generierung dieser Pakete nur Server-seitig.

SUCCESS (Version 1)

Ein SUCCESS-Paket wird vom Server als Bestätigung einer erfolgreichen Operation gesendet, wenn der Client keine anderen Informationen erwartet. Dies ist bspw. bei einem Login, dem Abgeben einer Bewertung usw. der Fall. Ein SUCCESS-Paket enthält keine Nutzdaten und wird vom Client nicht ausgewertet. Lediglich der Typ *SearchKey.SUCCESS* als Teil des BinaryPackets ist für den Client von Interesse.

ERROR (Version 1)

Das Gegenteil zu einem SUCCESS-Paket stellt das ERROR-Paket da. Ein ERROR-Paket wird vom Server stets dann gesendet, wenn eine Operation aus einem unbestimmten Grund nicht abgeschlossen werden konnte oder keine detaillierte Fehlermeldung übermittelt werden soll, bzw. die Fehlerquelle nicht eindeutig ist. So wird ein fehlgeschlagener Login und Upload

mit einem ERROR-Paket beantwortet. Dies geschieht bei fehlerhaften Benutzerdaten oder einem bereits vergebenen Tournamen. Letzterer Fehler sollte durch das Senden eines QUERY_UPLOAD-Paketes nicht auftreten. Schlägt das Eintragen einer Bewertung fehl, kann dies auch nur aus dem Umstand resultieren, dass der Nutzer bereits eine Bewertung für diese Tour abgegeben hat.

Ebenfalls werden sämtliche fehlerhaft empfangenen Pakete mit einem ERROR-Paket quittiert, genauso wie ein abgebrochener Download.

Wie auch das SUCCESS-Paket enthält dieses Paket keine Nutzdaten und wird nicht ausgewertet.

REGISTER_FAILED (Version 1)

Ein REGISTER_FAILED-Paket informiert den Client, aus welchem Grund eine Registrierung nicht erfolgreich war. Dies kann verschiedene Ursachen haben:

- Benutzername ist bereits vergeben
- Email-Adresse existiert bereits
- Email-Adresse ist falsch formatiert
- Passwort ist zu unsicher
- Benutzername, Passwort oder Email-Adresse enthält nicht unterstützte Zeichen.

Ähnlich wie bei einem QUERY_UPLOAD-Paket werden die Ursachen als Flag in einem Integer gesetzt. Aufbau:

key **(4 Byte)**

Tab. 12 – Paketaufbau REGISTER_FAILED

RESPONSE_TOUR (Version 1)

Als Antwort auf ein QUERY_TOUR-Paket erfolgt bei erfolgreicher Verarbeitung ein RESPONSE_TOUR-Paket, das die angeforderten Touren enthält. Die Touren werden als String-Array in einer Array-List gespeichert. Eine Tour enthält momentan folgende Informationen: Tour-ID, Tourname, Autor, Einstellungsdatum, Durchschnittsbewertung, Anzahl Downloads. Für die Übertragung werden sämtliche Touren und deren Inhalte in einem String zusammengeführt. Trennzeichen signalisieren bei der Rekonstruierung des Paketes Anfang und Ende eine Tour, bzw. der enthaltenen Spalte. Für den Fall, dass eine Suchanfrage keine Treffer erzielt, war zuerst geplant, eine leere Liste zu senden. Während der Entwicklung wurde dieser An-

satz aber überarbeitet. Nun wird in diesem Fall ein ERROR-Paket gesendet, welches über den Fehlschlag der Suchanfrage informiert.

Da nur eine ggf. große Zeichenkette gesendet wird, ist ihre Länge nicht von Interesse und wird nicht mitgesendet. Aufbau:

touren (n Byte)

Tab. 13 – Paketaufbau RESPONSE_TOUR

RESPONSE_RATING (Version 1)

Ein RESPONSE_RATING-Paket enthält eine Liste von Bewertungen für eine bestimmte Tour. Eine Bewertung enthält momentan die Informationen Autor (der Bewertung), Einstellungsdatum, abgegebene Bewertung und einen zugehörigen Kommentar.

Wie auch ein RESPONSE_TOUR-Paket werden die Bewertungen in Form eines ggf. großen Strings kodiert und versendet. Wurden noch keine Bewertungen abgegeben, wird eine leere Liste gesendet. Aufbau:

ratings (n Byte)

Tab. 14 – Paketaufbau RESPONSE_RATING

RESPONSE_PACKET (Version 1)

Ein RESPONSE_PACKET enthält als Antwort auf ein QUERY_PACKET-Paket die angeforderte Tour und zugehörige Metadaten. Die Metadaten bestehen aus der Tour-ID, Tourname, Autor, Einstellungsdatum, Durchschnittsbewertung, Anzahl Downloads. Die Tour selbst wird innerhalb eines Byte-Arrays kodiert. Aufbau:

tourID (4 Byte)	sizeOf-Tourname (4 Byte)	tourName (n Byte)	sizeOf-Author (4 Byte)	Author (n Byte)	sizeOfDate (4 Byte)	Date (n Byte)	sizeOf-Rating (4 Byte)	Rating (n Byte)	Downloads (4 Byte)
sizeOf-Payload (4 Byte)	Payload (n Byte)								

Tab. 15 – Paketaufbau RESPONSE_PACKET

RESPONSE_UPLOAD (Version 1)

Einem UPLOAD-Paket sollte ein QUERY_UPLOAD-Paket vorrausgeschickt werden, um sich über den wahrscheinlichen Erfolg eines Uploads zu erkundigen. Ein RESPONSE_UPLOAD-Paket wird als Antwort auf ein QUERY_UPLOAD-Paket versendet, wenn ein Upload fehl-

schlagen würde und informiert den Client, welche Gründe dies hat. Mögliche Gründe wären, dass eine Tour mit dem gewünschten Tournamen bereits existiert oder der Tourname unerlaubte Zeichen enthält. Diese Gründe werden als Flags, analog zu einem REGISTER_FAILED-Paket, gesetzt. Aufbau:

key (4 Byte)

Tab. 16 – Paketaufbau RESPONSE_UPLOAD

8.2 Implementierung von *Netty*

Dieses Kapitel widmet sich der eigentlichen Einbettung von *Netty* in das Kommunikationspaket und damit in das Server- bzw. Client-Programm. Dabei wird auf die Client- und Server-Konfiguration eingegangen, die sowohl den Verbindungsaufbau und Starten des Servers umfasst, als auch die notwendige Programmlogik für die Verarbeitung von ausgehenden und ankommenden Daten. Dafür muss ein BinaryPacket in einem Byte-Strom kodiert und dekodiert werden. Bei der Dekodierung kommt das *ReplayingDecoder*-Konzept zum Einsatz.

8.2.1 Client- und Serverkonfiguration

Serverseitig sind für die Initialisierung der Kommunikation folgende Schritte von Nöten:

1. Eine *NioServerSocketChannelFactory (SCF)*, basierend auf Javas *NewInputOutput*-System, wird erzeugt. Dabei wird ein nicht-blockierender *ServerSocketChannel* verwendet. Der SCF wird bei der Erzeugung zwei *ChachedThreadPools*[19] übergeben. Dabei dient der erste für die Verwaltung von *Boss-Threads.* Für jeden geöffneten Port einer Verbindung wird ein neuer *Boss-Thread* erzeugt (siehe Kapitel 6).

```
ChannelFactory factory = new NioServerSocketChannelFactory(
                             Executors.newCachedThreadPool(),
                             Executors.newCachedThreadPool());
```

Code 5 – Erzeugen einer ChannelFactory

2. Die Factory wird einem erzeugten *ServerBootstrap* zugeordnet. Diese Hilfsklasse akzeptiert eingehende Verbindungen und ist nur für die Verwendung von verbindungsorientierten Protokollen, wie z.B. TCP, gedacht.

```
ServerBootstrap bootstrap = new ServerBootstrap(factory);
```

Code 6 – Erzeugen eines ServerBootstraps

[19] Ein ThreadPool ist eine Menge von Threads, die von der Java Virtual Machine erzeugt und verwaltet werden. Dabei werden abzuarbeitende Aufgaben durch die JVM an einen freien verfügbaren Thread delegiert. Der Aufwand für das ständige Anlegen eines neuen Threads entfällt somit.

3. Dem *ServerBootstrap* wird im Weiteren eine *ChannelPipelineFactory* zugeordnet. Sie dient dazu, jeder neuen Verbindung, für die ein so genannter *child channel* erzeugt wird, eine *ChannelPipeline* (siehe Kapitel 9) zuzuordnen. Dabei enthält die Pipeline momentan drei Handler: Den *BinaryPacketEncoder,* für die Enkodierung eines BinaryPackets in einen Byte-Strom, den *BinaryPacketDecoder,* für dessen Dekodierung und einen *ClientServerHandler*, welcher ein empfangenes BinaryPacket weiterverarbeitet.

```
bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
      public ChannelPipeline getPipeline() {
            return Channels.pipeline(new BinaryPacketDecoder(),
                                     new BinaryPacketEncoder(),
                                     new ClientServerHandler());
                                    }});
```

Code 7 – Anlegen einer ChannelPipeline

4. Optional können dem Bootstrap noch verschiedene Optionen übergeben werden. In diesem Fall werden die Standardoptionen *child.tcpNoDelay* und *child.keepAlive* eingeschaltet. Erstere bewirkt, dass Pakete trotz geringer Größe direkt versendet werden und nicht gewartet wird, bis neuer Content verfügbar ist. Ein *SUCCESS*- oder *ER-*

```
bootstrap.setOption("child.tcpNoDelay", true);
bootstrap.setOption("child.keepAlive", true);
```

Code 8 – Eigenschaften eines Bootstrap setzen

ROR-Paket würde ohne diese Option niemals direkt gesendet werden.

5. Kommunikationsbereit ist der Server nach der Erzeugung eines Channels. Dieser Channel akzeptiert Verbindungen, die an einen bestimmten Port gerichtet sind.

```
bootstrap.bind(new InetSocketAddress(1337));
```

Code 9 – Neuen Channel erzeugen und einem Port zuweisen

Die Client-Konfiguration durchläuft dieselben Schritte wie die des Servers und unterscheidet sich lediglich in der Erzeugung der benötigten Objekte.

1. Anstelle einer *NioServerSocketChannelFactory* erzeugt der Client eine *NioClientSocketChannelFactory.* Dabei hat der *Boss-Thread* die Aufgabe Verbindungen aufzubauen. Wurde eine Verbindung hergestellt, wird sie an einen *Worker-Thread* übergeben.
2. Die *NioClientSocketChannelFactory* wird einem *ClientBootstrap* zugeordnet.
3. Analog zum Server erfolgt die Zuweisung einer *ChannelPipeline* mit den BinaryPacket-Handlern. Doch wird der *ClientServerHandler* durch einen *ClientHandler* ersetzt, welcher über den Client-seitigen Werdegang des empfangenen BinaryPackets entscheidet.
4. Anstelle eines *bind*-Aufrufs folgt ein *connect*:

```
bootstrap.connect(new InetSocketAdress("IP or HOSTNAME", 1337));
```

Code 10 – connect-Methode des Clients

Um überprüfen zu können, ob der Server bereit ist, Verbindungen anzunehmen, bzw. der Client, Verbindungen zum Server aufzubauen, liefert die *bind(...) und* die *connect(...)*-Methode ein *ChannelFuture*-Objekt als Rückgabewert. Einem *ChannelFuture*-Objekt kann einem *ChannelFutureListener* übergeben werden, dessen Methode *operationComplete(...)* nach Beendigung der Operation aufgerufen wird. In dieser ist es üblich, den Status der Operation abzufragen und entsprechend zu handeln.

```
ChannelFuture.addListener(new ChannelFutureListener() {
      @Override
      public void operationComplete(ChannelFuture future)
      throws Exception {
          if(future.isDone() && future.isSuccess())
              System.out.println("Connection attempt succeeded!");
          else System.out.println("Connection attempt failed!");
      }
});
```

Code 11 – Ergebnisabfrage eines ChannelFuture-Objekts

8.2.2 ReplayingDecoder-Konzept

Bei der Übertragung von Paketen innerhalb eines Netzwerkes kann es vorkommen, dass Pakete von externen Kommunikationsteilnehmern, bzw. durch Betriebsmittel des Netzwerks, fragmentiert werden[20]. Dies hängt i.d.R. mit der von der Netzwerkschnittstelle erzwungenen, maximalen Paketgröße zusammen, der MTU (Maximum Transmission Unit). Die MTU legt fest, wie viel Bytes maximal innerhalb eines Pakets über diese Netzwerkschnittstelle versendet werden können.

Möchte ein gesendetes Paket eine Netzwerkschnittstelle passieren, deren MTU kleiner als die Paketgröße ist, wird das Paket fragmentiert oder verworfen. Die Fragmentierung hat zur Folge, dass der Empfänger das Paket nicht direkt als Ganzes empfängt und daher Probleme bei der Verarbeitung der enthaltenen Daten auftreten können. In der Regel wird das Problem auftreten, dass weniger Daten bislang eingetroffen sind, als eigentlich erwartet werden. In der *Netty*-Dokumentation findet sich ein anschauliches Beispiel[21]:

„In a stream-based transport such as TCP/IP, packets can be fragmented and reassembled during transmission even in a LAN environment. For example, let us assume you have received three packets:

```
+-----+-----+-----+
| ABC | DEF | GHI |
+-----+-----+-----+
```

because of the packet fragmentation, a server can receive them like the following:

```
+----+-------+---+---+
| AB | CDEFG | H | I |
+----+-------+---+---+„
```

Um diese Pakete zu defragmentieren und für die Applikationslogik verständlich zu machen, gibt es in *Netty* die Möglichkeit einen *FrameDecoder* oder einen *ReplayingDecoder* einzusetzen.

---

[20] Cisco: Resolve IP Fragmentation, MTU, MSS, and PMTUD Issues with GRE and IPSEC, url: http://www.cisco.com/en/US/tech/tk827/tk369/technologies_white_paper09186a00800d6979.shtml#topic1

[21] Netty Dokumentation: http://docs.jboss.org/netty/3.1/api/org/jboss/netty/handler/codec/frame/FrameDecoder.html

Die Anwendung eines *FrameDecoders* stellt sich als relativ trivial raus. Verwendet wird er in Form einer von ihm erbende Subklasse, die als Handler in die *ChannelPipeline* eingetragen wird. Zentraler Bestandteil dieser Klasse ist die Methode *decode(...ChannelBuffer buf...)*, die es zu überschreiben gilt. Dieser Methode wird ein *ChannelBuffer* übergeben, der den Inhalt der angekommenen Pakete enthält. Nachdem die Struktur der Daten innerhalb des Byte-Stroms bekannt sein muss, können die Daten an dieser Stelle entsprechend ausgelesen und formatiert werden.

Bei der Rekonstruktion eines Paketes wiederholen sich für jedes enthaltene Element die folgenden Schritte:

1. Prüfen, ob die Anzahl an zur Verfügung stehenden Bytes ausreichend ist, um das ausstehende Element zu rekonstruieren. Ist dies nicht der Fall, wird die aktuelle Ausführung abgebrochen.
2. Stehen genügend Bytes zur Verfügung werden die Bytes gelesen und das Element rekonstruiert.

In der Regel enthält ein Paket aber komplexere Elemente, wie bspw. einen String. Da ein String aber normalerweise keine feste Länge besitzt, wird ihm eine Zahl vorangestellt, deren Wert die Anzahl an Bytes enthält, welche die folgende Zeichenkette repräsentieren. Daher wird i.d.R. wie folgt vorgegangen:

```
protected Object decode(ChannelHandlerContext ctx,
                        Channel channel,
                        ChannelBuffer buf) throws Exception {

      if (buf.readableBytes() < 4) {                  // 1.
            return null;
      }

      buf.markReaderIndex();                          // 2.

      int length = buf.readInt();                     // 3.

      if (buf.readableBytes() < length) {             // 4.

            buf.resetReaderIndex();
            return null;
      }

      byte[] data = new byte[length];                 // 5.
      data = buf.readBytes(length);
      return new String(data);
}
```

Code 12 – Bespiel decode()-Methode eines FrameDecoders

Bei Punkt eins wird geprüft, ob genügend Bytes für einen Integer gelesen werden können. Falls nicht, wird die Bearbeitung abgebrochen. Sind genügend Bytes vorhanden, so wird der aktuelle Lese-Index im *ChannelBuffer buf* markiert (Punkt zwei). Sollten nicht genügend Bytes für den folgenden String gelesen werden können, kann der Integer bei einem erneuten Aufruf von *decode(...)* wieder gelesen und der String in diesem Anlauf eventuell rekonstruiert werden. Hier drin liegt der eigentliche Clou des *FrameDecoders*. Sollte die theoretisch enthaltene Datenmenge nicht vollständig angefordert werden, wird der Lese-Vorgang in einem fehlerhaften Zustand abgebrochen und erneut gestartet, wenn weitere Daten auf diesem Kanal eintreffen. In Punkt drei wird die Anzahl von Bytes, die den folgenden String repräsentieren, in der Variable *length* gespeichert. Sind nach dem Lesen der Länge nicht mehr genügend Bytes im Puffer um den eigentlichen String zu lesen (Punkt vier), wird der Lese-index des Puffers zurück vor den Integer geschoben und die Methode abgebrochen. Andernfalls (Punkt fünf) können die nötigen Bytes gelesen und der String erzeugt werden.

Für ein komplexeres Protokoll, wie z.B. das beschriebene BinaryPacket, ist ein *FrameDecoder* jedoch sehr unhandlich und nicht die richtige Wahl. Die manuelle Index-Verwaltung ist aufwendig und zudem handelt es sich bei einem *FrameDecoder* um einen blockierenden[22] I/O-Prozess.

Bei einem *ReplayingDecoder* handelt es sich um eine spezialisierte Version eines *FrameDecoders*. Bei jedem fehlgeschlagenen Lesen wird eine *Error*-Nachricht geworfen und der *ReplayingDecoder* übernimmt die Steuerung. Der Lese-Index wird wieder auf die Initialstellung zurückgestellt und die *decode(...)*-Methode erneut aufgerufen, wenn weitere Daten angekommen sind. Durch diese Technik wird der Thread nicht blockiert und kann andere Aufgaben ausführen. Gerade bei einem Empfang von großen Datenmengen stellt sich dies als äußerst praktisch heraus. Der *ReplayingDecoder* erlaubt zudem eine weitere Optimierung. Da es unvorteilhaft wäre, bereits gelesene Elemente bei jedem Aufruf der *decode(...)*-Methode erneut zu lesen, können Checkpoints gesetzt werden, die den aktuellen Stand markieren. Dafür wird ein Enum eingesetzt, welches die verschiedenen Status enthält. Der Klasse *BinaryPacketDecoder* erweitert die Klasse *ReplayingDecoder* und verwendet das Enum *DecodingState*:

[22] Ein „blockierender Prozess“ meint, dass der aufrufende Thread solange durch die Schreib- Leseoperation blockiert, bis ein Ergebnis bekannt ist und zurückgegeben wird.

```
@Override
protected Object decode(ChannelHandlerContext ctx, Channel channel,
                        ChannelBuffer buffer, DecodingState state)
                       throws Exception {

switch (state) {
      case VERSION:
            this.message.setVersion(Version.fromByte(buffer.readByte()));
            checkpoint(DecodingState.TYPE);

      case TYPE:
            this.message.setType(Type.fromByte(buffer.readByte()));
            checkpoint(DecodingState.PAYLOAD_LENGTH);

      case PACKET_ID:
            this.message.setPacketID(buffer.readInt());
            checkpoint(DecodingState.PACKET_ID);

      case PAYLOAD_LENGTH:
            int size = buffer.readInt();
            if (size <= 0) {
                  throw new Exception("Invalid content size");
            }
            byte[] content = new byte[size];
            this.message.setPayload(content);
            checkpoint(DecodingState.PAYLOAD);

      case PAYLOAD:
            buffer.readBytes(this.message.getPayload(), 0,
                             this.message.getPayload().length);

            try {
                  return this.message;
            } finally {
                  this.reset();
            }

      default:
            throw new Exception("Unknown decoding state: " + state);
}}
```

Code 13 – decode()-Methode des BinaryPacketDecoders

Bis auf den letzten Status *PAYLOAD* wird nach jedem erfolgreich gelesenen Element ein Checkpoint gesetzt, an dessen Stelle der Lesevorgang wieder ansetzt, sollte beim nächsten Element ein Lesefehler auftreten. Mit der *reset()*-Methode wird die interne Status-Verwaltung des *ReplayingDecoders* zurückgesetzt und ein weiteres *BinaryPacket* kann auf diesem Kanal empfangen werden. Würde die Status-Verwaltung nicht zurückgesetzt werden, würde die Dekodierung stets wieder bei beim letzten Checkpoint beginnen und somit vermutlich falsche Daten lesen, bzw. scheitern.

Der *default*-Fall aus dem *switch*-Konstrukt sollte niemals auftreten, sondern die Ausführung vorher durch einen auftretenden Fehler oder die Rückgabe des fertigen Pakets beenden.

8.2.3 Der BinaryPacketEncoder

An dieser Stelle sei noch kurz der Vollständigkeit halber auf die Enkodierung eines BinaryPackets in einen Byte-Strom eingegangen. Im Falle eines *DownStreamEvents*, also wenn eine Nachricht gesendet werden soll, wird ein passender Handler gesucht, der die Nachricht in einen *ChannelBuffer* konvertieren kann. Für die in diesem Projekt verwendeten BinaryPackets ist der Handler ein *BinaryPacketEncoder*. Dabei erbt diese Klasse direkt von der Mutterklasse *OneToOneEncoder.* Die Aufgabe eines Encoders ist es, eine zu sendende Nachricht zu formatieren[23]. Dabei werden die *encode(...)*-Methoden aller Encoder in einer *ChannelPipeline* aufgerufen. Die Hauptsache ist, dass am Ende ein *ChannelBuffer* zum Versenden bereit steht; andernfalls schlägt das Senden fehl.

Da der *encode(...)*-Methode die Nachricht als Typ *Object* übergeben wird, wurde die eigentliche Logik in die Hilfsmethode *encodeMessage(BinaryPacket message)* ausgelagert:

```
public static ChannelBuffer encodeMessage(BinaryPacket message)
                              throws IllegalArgumentException {
      // verify that no fields are set to null
      ...

      // Define size of buffer
      // version(1b) + type(1b) + packetID(4b) +
      // payload length(4b) + payload(nb) = 10

      int size = 10 + message.getPayload().length;
      ChannelBuffer buffer = ChannelBuffers.buffer(size);

      buffer.writeByte(message.getVersion().getByteValue());
      buffer.writeByte(message.getType().getByteValue());
      buffer.writeInt(message.getPacketID());
      buffer.writeInt(message.getPayload().length);
      buffer.writeBytes(message.getPayload());

      return buffer;
}
```

Code 14 – encodeMessage()-Methode des BinaryPacketEncoders

[23] Netty Dokumentation:
http://docs.jboss.org/netty/3.2/api/org/jboss/netty/handler/codec/oneone/OneToOneEncoder.html

9. Entwicklung – Server

Die Entwicklung des Servers umfasst eine Formulierung von Anforderungen, deren Einhaltung notwendig ist, um die Erwartungen des Clients an den Server zu erfüllen. Darauf folgt die Dokumentation des Softwaremodells und der Implementierung. Dabei dürfen die Anforderungen an die Kommunikation aus Kapitel 4 nicht außer Acht gelassen werden. Der bei Abschluss erreichte Projektstand wird in Kapitel 11 beschrieben.

9.1 Anforderungen – Server

Die Grundlage hierfür bildet der Aufgabenbereich des Servers, der sich aus den Antwortpaketen, die in Kapitel 8.1.4 beschrieben wurden, ergibt. Aus diesem resultieren vier Anforderungen, die von jeder Reaktion seitens des Servers erfüllt werden müssen. Dabei lassen sich diese Anforderungen ebenfalls auf den Client übertragen:

- Syntaktisches Verständnis

  Das syntaktische Verständnis ist gegeben, wenn der Empfänger eines Datenpakets in der Lage ist, den ankommenden Byte-Strom zu dekodieren. Dies geschieht durch die Rekonstruktion des gesendeten Pakets auf der Anwendungsebene. Somit muss die Nachricht zunächst als BinaryPacket rekonstruiert und anhand der Meta-Informationen und des Payloads das korrekte Paket wiederhergestellt werden. Ist dieser Schritt nicht möglich, liegt eine fehlerhafte Implementierung des Protokolls seitens des Senders oder Empfängers vor. Ebenfalls möglich ist ein Fehler im Protokoll an sich oder eine fehlerhafte Übertragung unterhalb der Anwendungsebene.

- Semantisches Verständnis

  Nach der Rekonstruktion der ursprünglichen Nachricht muss mit dem Inhalt entsprechend verfahren werden. Auf die Anfrage des Clients muss die passende Reaktion folgen. Die Zuordnung Anfrage → Reaktion wird durch das Protokoll spezifiziert. Wurde die Spezifikation durch den Programmierer korrekt implementiert, kann die *totale Korrektheit*[24] als gegeben betrachtet werden.

[24] Vgl. Manfred Broy, (Informatik. Eine grundlegende Einführung: Band 2. Systemstrukturen und Theoretische Informatik, Springer-Verlag, 15.10.1998, ab Seite 364)

- Semantische Mitteilungsfähigkeit

  Wie auch beim semantischen Verständnis muss der Empfänger einer Anfrage in der Lage sein, seine Antwort so zu formulieren, dass der Gesprächspartner darauf reagieren kann. Hier kommt abermals das Protokoll zum Einsatz.

- Syntaktische Mitteilungsfähigkeit

  Kann vom syntaktischen Verständnis abgeleitet werden. Die Antwort auf eine Anfrage muss dem Gesprächspartner in einer Form mitgeteilt werden, die dieser zu verstehen bzw. zu dekodieren vermag.

9.2 Softwaremodell – Server

Dieses Kapitel behandelt die Herleitung des vom Server verwendeten Softwaremodells, dessen Bestandteile sich grob in zwei Komponenten unterteilen. Ausgangspunkt hierfür war das Kommunikationsmodell und der Aufgabenbereich des Servers.
Dabei übernimmt eine Komponente die Kommunikation mit dem Client, die andere die Kommunikation mit der Datenbank. Das folgende Komponentendiagramm zeigt den generellen Aufbau des Servers.

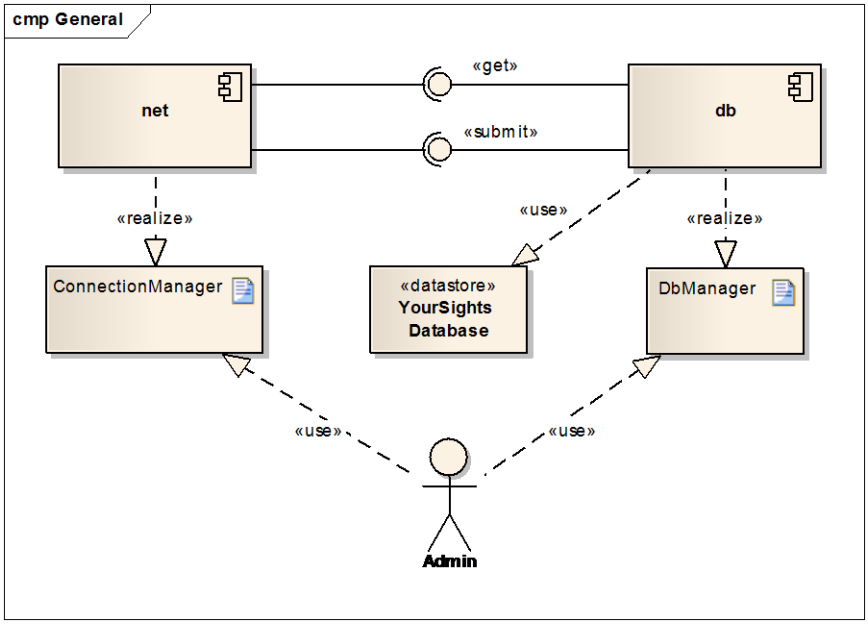

Abb. 8 – Komponentendiagramm: Gesamtüberblick - Server

Das *net*-Paket ist verantwortlich für die externe Kommunikation mit dem Client. Es nimmt Anfragen entgegen und leitet sie an das *db*-Paket weiter, wo sie verarbeitet werden. Im Anschluss kann die fertige Antwort dort vom *net*-Paket abgeholt und versendet werden. Diese Aufgabenverteilung hat mehrere Gründe. Dabei muss der rudimentäre Ablauf betrachtet werden, der aus diesen Aufgaben besteht:

1. Anfrage rekonstruieren
2. Anfrage interpretieren und aufbereiten
3. Anfrage verarbeiten
4. Antwort generieren
5. Antwort versenden

Die Rekonstruktion der Anfrage lässt sich nur dem *net*-Paket zuordnen. Die Interpretation und Aufbereitung könnte in einer eigenständigen Komponente ausgelagert werden, deren Aufgabe das Auslesen und Formatieren der enthaltenen Information wäre. Diese würden dann durch das *db*-Paket verarbeitet werden. Beruhend auf dem aktuellen Konzept, ist jedoch für jede Anfrage ein Austausch mit der Datenbank erforderlich. Daher wurde entschieden, diesen Schritt in die *db*-Komponente einzugliedern, genauso wie das Verfassen der Antwort. Der Versand erfolgt wiederum durch das *net*-Paket.

Die Verwaltung des Servers erfolgt durch zwei konkrete Instanzen der Klassen *ConnectionManager* und *DbManager*. Die nächsten zwei Kapitel gehen auf die Beschreibung der beiden Komponenten genauer ein.

9.2.1 net-Komponente

Die Bestandteile dieser Komponente haben folgende Aufgaben:

- Herstellen/Abschalten der Kommunikationsbereitschaft
- Empfangen von Nachrichten
- Verarbeiten von Nachrichten
- Senden von Nachrichten

Für die Beschreibung dieser Vorgänge wurden die wichtigsten Bestandteile in einem Komponentendiagramm zusammengefasst:

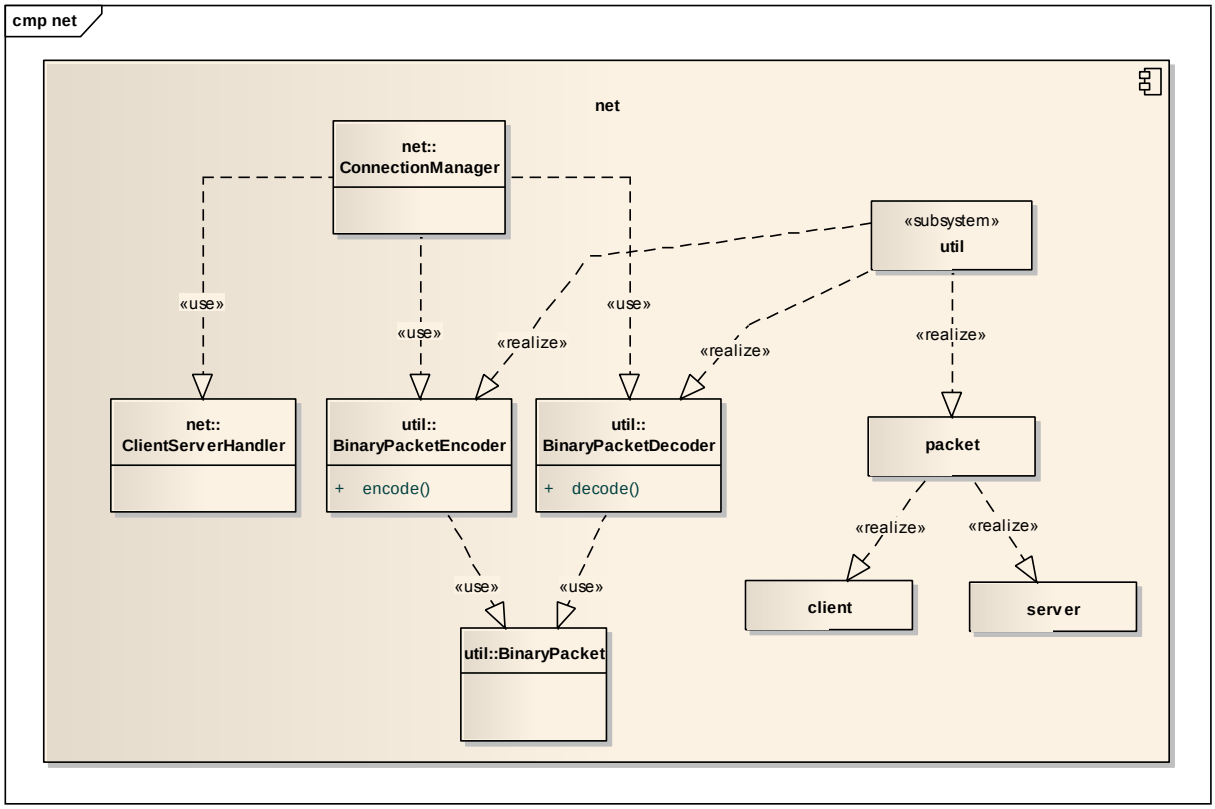

Abb. 9 – Komponentendiagramm: net

Herstellen/Abschalten der Kommunikationsbereitschaft

Für diese Vorgänge wird bei Programmstart eine Instanz der Klasse *ConnectionManager* erzeugt. Diese Klasse wurde als Singleton implementiert, da über sie auch die Bindung der Anwendung an einen Port erfolgt. Eine Bindung von mehreren Anwendungen an einen Port ist nicht erlaubt, da für das Betriebssystem eine deterministische Zuordnung von ankommenden Datenpaketen an ein, sich in Ausführung befindendes Programm, möglich sein muss.

Das Herunterfahren der Kommunikationsbereitschaft besteht aus mehreren Teilschritten. Zuerst werden alle offenen Verbindungen geschlossen. Dabei wird noch eine gewisse Zeit versucht, sich noch im Puffer befindende Daten zu senden. Nachdem alle Verbindungen geschlossen wurden, werden externe Ressourcen freigegeben werden. Dies sind bspw. die Threadpools, die bei der Herstellung der Verbindung angelegt wurden (siehe Kapitel 10.2.1).

Empfangen von Nachrichten

Nachdem ein Client eine Verbindung hergestellt hat, steht ihm ein *Channel*-Objekt zur Verfügung, über welches er Nachrichten an den Server senden und empfangen kann. Treffen beim Server Datenpakete ein, werden sie durch die Socket-Verwaltung von Netty an die neu erzeugte *ChannelPipeline* weitergeleitet.

Verarbeiten von Nachrichten

Diese Datenpakete werden von einem Handler in der *ChannelPipeline* verarbeitet und an den nächsten Handler weitergereicht. Der erste Handler ist hierbei der *BinaryPacketDecoder*. Er erstellt ein neues BinaryPacket und füllt es mit den ankommenden Daten. Wurde die Nachricht vollständig empfangen, leitet er das Paket an eine Instanz des *ClientServerHandlers* weiter. An dieser Stelle könnte eine erste Auswertung der Nachricht erfolgen und ein Verarbeitungsprozess angestoßen werden. Dies wäre bei Prozessen sinnvoll, die nicht auf den Datenbestand der Datenbank zurückgreifen müssen, bspw. die Erzeugung eines Tokens im Rahmen einer Session-Verwaltung. Da in dieser prototypischen Implementierung solche Prozesse noch nicht existent sind, leitet der *ClientServerHandler* die Nachricht direkt an eine Instanz der Klasse *MessageCallable* aus dem *db*-Paket weiter. Diese Klasse wird im nächsten Kapitel genauer beschrieben.

Senden von Nachrichten

Wurde die Anfrage erfolgreich bearbeitet wird die korrespondierende Antwort in den selben *Channel* geschrieben, auf dem die Anfrage eingegangen ist. Dabei wird die entgegengesetzte Richtung eingeschlagen und die Antwort, kodiert als BinaryPacket, durchläuft zuerst die *Channelpipeline*. An der Stelle des *BinaryPacketEncoder* wird die Nachricht als Byte-Folge in einen *ChannelBuffer* geschrieben. An dieser Stelle übernimmt *Netty* und sendet die enthaltenen Daten über den Socket an den Client.

9.2.2 db-Komponente

Der größte Teil der Anwendungslogik des Servers liegt in dieser Komponente. In ihrer Zuständigkeit liegt die Zuordnung von Anfrage → Reaktion und der daraus resultierenden Verarbeitung. Dies beinhaltet auch die Kommunikation mit der Datenbank. Demnach muss sie:

- die Verbindung mit der Datenbank verwalten
- Anfragen entgegennehmen und bearbeiten
- Unter Zuhilfenahme des Datenbestands die fertige Antwort formulieren

Die wichtigsten Bestandteile dieser Komponente finden sich im folgenden Komponentendiagramm wieder:

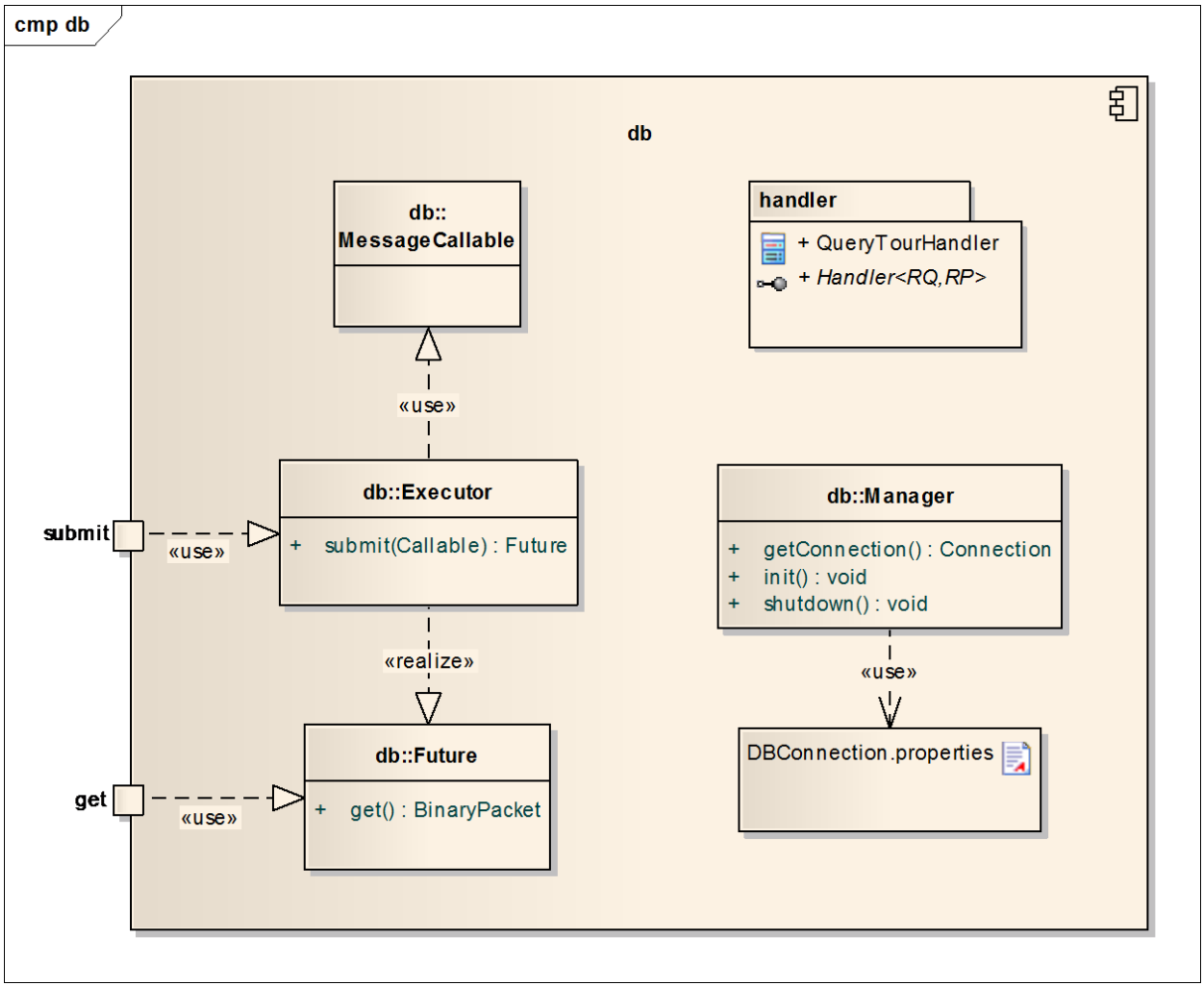

Abb. 10 – Komponentendiagramm: db

Datenbankverbindung verwalten

Das Verwalten der Datenbankverbindung beinhaltet den Verbindungsaufbau, Aufrechterhaltung und Abbau der Verbindung. An diesem Punkt musste entschieden werden, wann eine Verbindung aufgebaut wird und wie viele es sein sollen. Da der Aufbau und die Bereitstellung einer Datenbankverbindung eine zeitaufwendige Prozedur[25] ist, wurde entschieden, dass beim Starten des Servers eine Verbindung mit der Datenbank erfolgen muss. Dies spart enorm viel Zeit bei der Verarbeitung von Anfragen und schont Ressourcen.

Wichtig hierbei ist jedoch die Aufrechterhaltung der Verbindung. Es darf nicht sein, dass nach längerer Inaktivität die Verbindung gekappt wird und keine Anfrage mehr an die Datenbank gestellt werden kann. Hier müsste ein eigener Thread die Verbindung überwachen

[25] Sun Developer Network:
http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html

und bei einem Abbruch erneut aufbauen. Eine Problemlösung wurde schließlich unter Zuhilfenahme des *Timer*-Konzepts von Java gefunden. Dabei stellt die JVM die Möglichkeit zur Verfügung, zu einer bestimmten Zeit oder in Intervallen eine bestimmte Aktion durchzuführen. Die Ausführung dieser Aktion erfolgt in einem dedizierten Thread und blockiert somit nicht die momentane Ausführung eines Prozesses. So war vorgesehen, in einem regelmäßigen Intervall (alle zehn Sekunden) eine Anfrage mit minimalen Aufwand (*SELECT 1 FROM DUAL;)* an die Datenbank zu stellen. Die Implementierung wurde allerdings schon in einem frühen Stadium aufgrund einer weiteren Fragestellung verworfen. Dabei ging es darum, wie viele Verbindungen aufgebaut werden sollten. Sollte es eine feste Anzahl an Verbindungen sein, die sich alle verbundenen Clients teilen (*ConnectionPooling)* oder sollte für jeden Client eine Verbindung verfügbar sein. Letztere Lösung würde allerdings voraussetzen, dass eine enorme Menge an Verbindungen verwaltet werden müssten, die ggf. gar nicht benötigt werden. Oder eine bestimmte Anzahl an Verbindungen wird vorgehalten und dynamisch erhöht, wenn mehr Verbindungen benötigt werden.

Die effizienteste Lösung[26] bietet aber das Konzept des *ConnectionPoolings.* Hierbei wird initial eine bestimmte Menge an Verbindungen aufgebaut und aufrecht erhalten. Viele Clients teilen sich eine kleine Menge an Verbindungen, was sich äußerst ressourcenschonend auf den Server auswirkt. Diese Lösung eignet sich dabei aber nur für Server, die viele, aber kurze Anfragen an die Datenbank stellen müssen. Muss eine Verbindung dauerhaft belegt werden, steht sie anderen nicht zur Verfügung.

Die interne Arbeitsweise des *ConnectionPoolings* beruht auf der Verknüpfung von physikalischen und künstlichen Verbindungen. Dabei werden eine bestimmte Menge von „echten" Datenbankverbindungen aufgebaut, ebenso wie eine Menge von Pseudo-Verbindungen. Diese werden mit den echten Verbindungen verknüpft und können wie diese benutzt werden. Zum Beispiel können Anfragen gesendet und/oder *ResultSets* angefordert und ausgelesen werden. Alle Verbindungen werden in einem *ConnectionPool* verwaltet. Wird eine Verbindung von diesem Pool angefordert, stellt der Pool eine Pseudo-Verbindung bereit. Sind alle physikalischen Verbindungen in Benutzung, wird der aufrufende Thread blockiert. Der wesentliche Unterschied zwischen der physikalischen und der künstlichen Verbindung liegt in der *close()*-Methode. Wurde eine physikalische Verbindung geschlossen, ist ein Öffnen nicht mehr möglich und es muss eine neue Verbindung aufgebaut werden. Soll solch eine

[26] dPunkt Verlag: http://www.dpunkt.de/java/Programmieren_mit_Java/Java_Database_Connectivity/48.html

Verbindung mehrfach benutzt werden, muss ein entsprechender Aufruf also verhindert werden. Von essentieller Wichtigkeit ist allerdings der Aufruf der *close()*-Methode bei einer künstlichen Verbindung. Durch ihn wird die Verbindung zurück in den Pool gegeben und steht anderen wieder zur Verfügung. Nachdem also das Ergebnis der Anfrage feststeht, sollte umgehend ein entsprechender *close()*-Aufruf erfolgen.

Der Oracle JDBC-Treiber stellt eine Möglichkeit für das *ConnectionPooling bereit,* sodass keine manuelle Implementierung nötig ist. Das Feature kann über die Methode *setConnectionCachingEnabled(boolean)* eines *OracleDataSource*-Objekt aktiviert werden. Über ein Property-File, dass der Methode *setConnectionCacheProperties* übergeben wird, können die Eigenschaften des Pools oder der Verbindungen geändert werden. So lassen sich z.B. die minimale oder maximale Anzahl an Verbindungen oder ein Timeout einstellen. Die physikalischen Verbindungen stellt der Pool ähnlich her, wie bei dem Aufbau einer Standardverbindung mit dem JDBC-Treiber. Dem *OracleDataSource*-Objekt wird dazu die Verbindung in Form einer URL, der Benutzername und das Passwort übergeben. Die Klasse *Manager*, die das *ConnectionPooling* initialisiert, entnimmt diese Informationen dabei der Property-Datei *DBConnection.*

Die Verwendung des *ConnectionPoolings* durch den JDBC-Treiber wird von Oracle nicht mehr empfohlen und die entsprechenden Methoden wurden als *deprecated* gekennzeichnet. Oracle empfiehlt[27] die Verwendung einer hauseigenen Lösung, die nicht Bestandteil des JDBC-Treibers ist, den *Oracle Universal Connection Pool*[28]*.* Nachdem der Einsatz dieser Lösung eine längere Einarbeitungszeit mit sich gebracht hätte und die veraltete Lösung für eine prototypische Anwendung ausreichend ist, wurde ihr der Vorzug gegeben. Bei den bisherigen Versuchen konnten keine Fehler festgestellt werden.

Anfragen entgegennehmen und bearbeiten

Die Bearbeitung einer Anfrage erfordert die Bestimmung der Verfahrensweise für diese Anfrage und das Formulieren der Anfrage an die Datenbank. Darunter fällt auch die Zuordnung von Anfrage→ Reaktion, die durch das Protokoll in Kapitel 9 beschrieben wird. Der Ablauf sieht wie folgt aus:

[27] Oracle Database JDBC Developer's Guide and Reference:
http://docs.oracle.com/cd/B14117_01/java.101/b10979/conncache.htm
[28] Oracle: http://www.oracle.com/technetwork/database/enterprise-edition/downloads/ucp-112010-099129.html

Die *net*-Komponente hat die Anfrage in Form eines BinaryPackets wiederhergestellt und übergibt es einer neu erstellten Instanz der Klasse *MessageCallable.* Innerhalb die Klasse wird die eigentliche Anfrage anhand des Typs bestimmt. Ein neu erzeugter zugehöriger Handler für diesen Typ formuliert die Datenbankanfrage, welche innerhalb des *MessageCallable*-Objekts an die Datenbank gesendet wird. An dieser Stelle musste entschieden werden, in welcher Form Daten bei der Datenbank angefragt werden. Es existieren zwei verschiedene Ansätze.

Der erste sieht vor, eine SQL-Abfrage algorithmisch aus der Anfrage zu generieren und an die Datenbank zu senden.

Der andere besteht in der Bestimmung der Parameter für eine, in der Datenbank hinterlegten *Stored Procedure*.

Die Verwendung von *Stored Procedures* bringt mehrere Vorteile[29] mit sich:

1. Die eigentliche Anfrage an die Datenbank wird nicht mehr innerhalb der Anwendung erstellt. Somit muss im Falle einer Aktualisierung oder Umstrukturierung des Datenbankschemas oder der Berechnungsroutine nur die gespeicherte Prozedur angepasst werden, solange der Rückgabewert identisch ist. Eine Anpassung des Servers ist nicht erforderlich, was die Flexibilität erhöht.
2. Der Traffic zwischen Server und Datenbank wird bei langen SQL-Anfragen reduziert, weil sie auf einen einzigen Prozeduraufruf gekürzt werden.
3. Die Verarbeitung kann beschleunigt werden, da das RDMS i.d.R. auf einem leistungsfähigeren System installiert ist.

Im bestehenden Konzept müssen allerdings nur einfache Anfragen verarbeitet werden. Die Implementierung eines Algorithmus für die Erstellung dieser SQL-Anfrage kostete weniger Zeit und führte zum selben Ergebnis. Daher wurde im Rahmen der prototypischen Entwicklung diese Variante bevorzugt. Nachdem sich die SQL-Abfragen im Aufbau bei den verschiedenen einzelnen Anfragen der Clients kaum unterscheiden, kann durch die Verwendung von vorgefertigten Abfragen die Performanz zudem erhöht werden[30].

Genau wie auch bei der Art der Datenbankabfrage musste bei der serverseitigen Verarbeitung von Anfragen entschieden werden, wer sie verarbeitet. Sollte es nur einen verarbeiten-

[29] Vgl. Mike Chapple: SQL Server Stored Procedures, url: http://databases.about.com/od/sqlserver/a/storedprocedure.htm

[30] Oracle: http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

den Prozess geben, der Anfragen entgegen nimmt und über die Handler eine Antwort generiert, wäre er der Flaschenhals der Anwendung und die Überlegungen bzgl. des *ConnectionPoolings* wären somit obsolet. Stattdessen wird ähnlich verfahren und eingehende Anfragen werden von einem eigenen Thread verarbeitet. Dieser wird einem Thread-Pool entnommen, in welchen er nach erledigter Arbeit zurückkehrt.

Ein auftretendes Problem dabei ist, dass nicht festgestellt werden kann, welcher Thread jetzt genau welches Paket bearbeitet. Dies ist jedoch notwendig, um die korrekte Antwort entgegennehmen zu können. Als Lösung hierfür bietet sich die Verwendung eines *ExecutorService* an. Dieser legt einen Thread-Pool an und verwaltet ihn selbständig. Auf Anforderung wird ein freier Thread des Pools geliefert, der anstehende Arbeiten übernehmen kann. Steht gerade kein freier Thread zur Verfügung, wird ein neuer erzeugt.

Die Anforderung eines Threads erfolgt über die Methode *submit(...),* der ein Objekt vom Type *Callable* oder *Runnable* übergeben wird. Der Rückgabewert von *submit(...)* ist stets ein Future-Objekt, mit dem das Ergebnis der ausgeführten Arbeit abgefragt werden kann. Über dieses Objekt kann im Falle der Übergabe eines *Callable*-Objekts der Rückgabewert der Methode *call()* angefordert werden.

Die Methode *call()* muss von jedem *Callable*-Objekt implementiert werden und enthält die, vom Thread auszuführenden, Anweisungen.

So wird in dieser Methode zum einen der passende Handler für die Anfrage erzeugt und die resultierende Abfrage an die Datenbank gestellt.

Antwort formulieren

Zum anderen wird in der *call()*-Methode die entsprechende Antwort generiert. Dies erfolgt unter Zuhilfenahme des Abfrageergebnisses, welches in einem *ResultSet* gespeichert wird. Dieses wird der Methode *buildResponse(ResultSet)* übergeben, ausgelesen und das passende Antwort-Paket erzeugt. Nach dem Auslesen des *ResultSets* wird die künstliche Datenbankverbindung wieder zurückgegeben. Das Antwort-Paket wird in einem BinaryPacket verpackt. Dafür greift die *net*-Komponente auf die *get()*-Methode des *Future*-Objekts zurück.

Anmerkung

Da sowohl der Empfang einer Nachricht, als auch ihre Verarbeitung jeweils in einem eigenen Thread abläuft, entsteht kein Flaschenhals und es können theoretisch viele Anfragen parallel

verarbeitet werden. Ebenso erfolgt kein Zugriff auf gemeinsame Objekte, wodurch die Thread-Sicherheit gegeben ist.

10. Entwicklung Client

10.1 ProgressListener

Netty ist ein asynchrones, damit einhergehendes, ereignisgesteuertes Kommunikationsframework. Über einen Handler ist es möglich, eingehende Nachrichten zu verarbeiten, doch fehlt die Möglichkeit, sich über den Fortschritt einer eingehenden Nachricht zu informieren. So werden lange Nachrichten nicht am Stück gesendet, sondern in fragmentierter Form. Demnach wäre es wünschenswert sich bspw. über den Stand eines Downloads informieren zu können. Der wohl nächstliegende Anwendungsfall ist der Einsatz eines Fortschrittbalkens. Das hierfür entwickelte Interface *ProgressListener* definiert vier Methoden, die von einer Unterklasse implementiert werden müssen:

- *getPacketID()/setPacketID(int pid)*: Nachdem die Kommunikation einem Frage-Antwort-Paradigma unterliegt wurde nach Konvention vereinbart, dass die *PacketID* der Antwort, der *PacketID* der Anfrage entspricht. Auf diese Weise ist es möglich, genau den *ProgressListerner* zu informieren, der auf eine entsprechende Antwort wartet.

  Die Verbindung *Anfrage* → *ProgressListener* ← *Antwort* erfolgt über das Setzen und Vergleichen der *PacketID*, welche im Listener gesetzt werden muss und ebenfalls in der Antwort enthalten ist.
- *progressChange(ProgressState dim)*: Mit jedem Eintreffen eines Antwortfragments wird versucht, die gesamte Nachricht zu lesen. Ist dies nicht möglich, weil die Antwort noch nicht vollständig eingetroffen ist, wird diese Methode aufgerufen und der Status der eintreffenden Antwort übergeben. Dies erfolgt über ein *ProgressState Obejekt*, welches die Anzahl der erwarteten Bytes den bereits empfangenen gegenüber stellt. Eine prozentuale Berechnung der bereits empfangenen Daten ist hiermit möglich, um etwaige Ausgaben für den Anwender zu machen.
- *finish(BinaryPacket data)*: Diese Methode wird aufgerufen, wenn die Nachricht vollständig eingetroffen ist. Ihr wird das eingetroffene *BinaryPacket,* zwecks Weiterverarbeitung, als Parameter übergeben. Zudem wird der *ProgressListener* aus dem

ProgressListenerPool entfernt. Nachdem diese Methode durch den entsprechenden *Handler* aufgerufen wurde, ist der Vorgang der Anfrage abgeschlossen.

Zur Zuordnung des *ProgressListeners* zur entsprechenden Anfrage, wird der *ProgressListener* beim Senden einer Nachricht der entsprechenden *write()*-Methode übergeben und im *ProgressListenerPool* registriert. Beim Eintreffen neuer Nachrichten wird der *ProgressListenerPool* informiert, welcher anhand der *PacketID* der eintreffenden Nachricht den wartenden *ProgressListener* informiert.

10.1.1Einbettung in Android

Zur besseren Einbettung in Android wurde die abstrakte Klasse *SimpleProgressListener* geschrieben, welche als Basisklasse für weitere Implementierungen fungiert. Der *SimpleProgressListener* stellt einige Grundfunktionalitäten bereit, welche bei allen *ProgressListenern* gleich sind. Die konkreten Klassen wurden dann für den jeweiligen Anwendungsfall als anonyme Klassen implementiert. Alternativ wäre es möglich, für jeden Anwendungsfall einen konkreten *ProgressListener* zu schreiben. Da die *ProgressListener* in den meisten Fällen jedoch nur an einer Stelle im Programm definiert werden, brächte diese Alternative keinen Mehrwert. Begründet durch die durchaus zeitintensiven Anfragen und die damit ausgeschlossene Möglichkeit die Ausführung im Main-Thread laufen zu lassen, da die grafische Oberfläche nicht blockiert werden darf, musste ein Weg gefunden werden, die Views der Oberfläche zu aktualisieren. Zum Beispiel um einen Fortschrittsbalken in Form einer *Progressbar* über den Downloadfortschritt zu Informieren. Zur Problemlösung wurde hierbei die Technik des *Handlers* verwendet. Gemeint ist hier nicht das aus Netty bekannt Prinzip, sondern ein Android internes Konstrukt. Ein Handler in Android ist ein Objekt, dem von anderen Objekten Nachrichten geschickt werden können. Diese Nachrichten können dann dazu dienen, die Oberfläche zu aktualisieren, da das Empfangen der Nachricht im Thread der Aktivität abläuft, in welcher der Handler erstellt wurde. In diesem Fall wäre dies die Main-Activity *Act*. Der zu diesem Zwecke entwickelte *Handler* mit dem Namen *MainGUIHandler* implementiert nur die Methode *handleMessage(Message msg)*, in der zunächst überprüft wird, um welche Nachricht es sich handelt, um diese dann entsprechend zu verarbeiten. Eine Message enthält hierbei immer einen Integer-Wert, welcher Auskunft über die Art der Nachricht gibt. Zusätzlich können der Nachricht weitere Informationen mitgegeben werden. Neben zwei grundsätzlich mitgelieferten zusätzlichen Integer-Werten kann der

Message hierzu ein *Bundle* übergeben werden. Das Bundle kann unterschiedliche Informationen enthalten. So kann bspw. auch ein serialisiertes Objekt mitgesendet werden.

Die in der App verankerten *ProgressListener* unterscheiden sich in der Implementierung der Methoden *progressChange* und *finish.* In der *progressChange* Methode informiert der *ProgressListener* den *MainGUIHandler* mittels einer dem Anwendungsfall entsprechende Nachricht darüber, dass sich der Fortschritt geändert hat. Dies findet bspw. Verwendung, um den Downloadfortschritt einer Tour aktuell zu halten. In Anwendungsfällen, die keinen Fortschritt visualisieren, bleibt diese Methode oftmals leer. In der *finish(BinaryPacket data)* Methode ist das anfängliche Vorgehen der *ProgressListener* ähnlich. Zunächst wird geprüft, welchen Typ das *data* Packet enthält. Ist der Typ des Pakets bekannt, so wird ein Objekt des Typs aus dem Payload des *BinaryPackets* erzeugt. Abhängig von der nun vorhandenen Nachricht verarbeitet der *ProgressListener* diese weiter. Handelt es sich bspw. um ein Fehlerpaket, so kann der *ProgressListener* eine Warnmeldung ausgeben. Wie genau sich der jeweilige *ProgressListener* verhält ist abhängig von der Anfrage, die gesendet wurde.

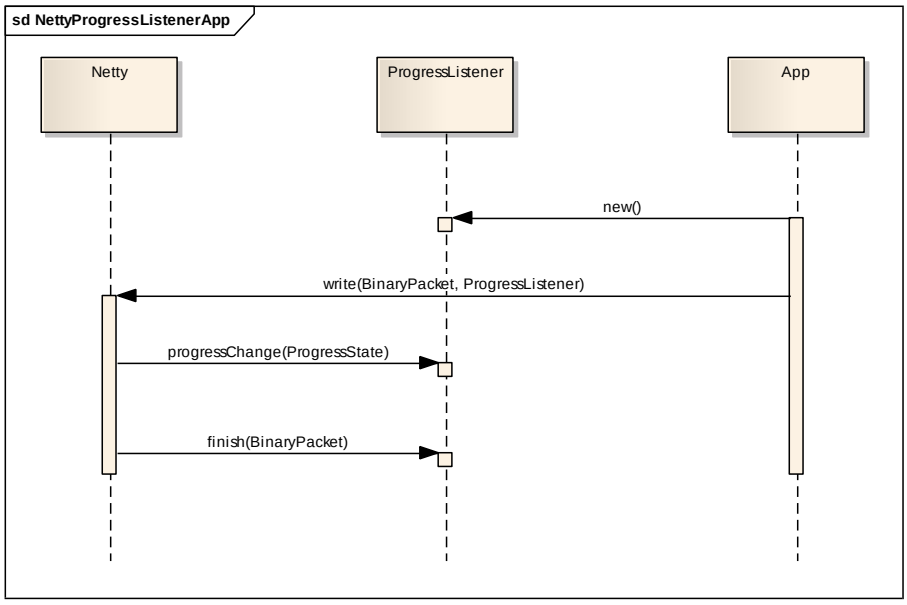

Abb. 11 – Kommunikation zwischen App, Netty und dem ProgressListener

10.1.2Exemplarischer ProgressListener

Zur besseren Veranschaulichung soll hier beispielhaft eine mögliche Implementierung eines *ProgressListeners* beschrieben werden. Der Anwendungsfall eines Downloads soll an dieser Stelle dazu dienen, einen genaueren Überblick über die Mechanik zu gewinnen. Da es sich hier um einen Download handelt, soll in der *progressChanged* Methode dafür gesorgt werden, dass der Stand des Downloads angezeigt wird.

```
@Override
public void progressChange(ProgressState dim) {

simpleTourWrapper.updateProgress(dim.getExpectedSize()/dim.getCurrentSize());
MainGUIHandler.sendEmptyMessage(Act.DOWNLOAD_PROGRESS_CHANGED);


}
```

Code 15 – progressChange-Methode des ProgressListeners

Der *progressChange* Methode wird hierzu ein *ProgressState* Objekt übergeben, welches Auskunft darüber gibt, wie groß die herunterladende Datei ist und wie viele Bytes bereits eingetroffen sind. Dies wird dazu genutzt, den Fortschritt in Prozent zu errechnen und in der Tour zu vermerken. Anschließend wird die GUI über ihren MainGUIHandler über die Veränderung informiert, sodass die Oberfläche neu gezeichnet werden kann. Beim Neuzeichnen wird dann der Wert aus der Tour ausgelesen und angezeigt.

```
if(data.getType()==Type.RESPONSE_PACKET)
{
try {
RESPONSE_PACKET packet= new RESPONSE_PACKET(data.getPayload());

RAWContainerv2 raw=
(RAWContainerv2) RAWOperationAndroid.toObjectFromByteArray(
                                                    packet.getPayload()
                                                    );

String path = Environment.getExternalStorageDirectory().getAbsolutePath()
            +"/"+raw.getTitle()+".tdpkg";
File f=null;
try {
f= new File(path);

RAWOperationAndroid.saveToFile(raw, f.getAbsolutePath());
} catch (Exception e) {
Log.e(ACTIVITY_TAG,"Writing error. Device not ready.");
return;
}

simpleTourWrapper.setPath(f.getAbsolutePath());
simpleTourWrapper.deactivateProgress();

MainGUIHandler.sendEmptyMessage(Act.DOWNLOAD_FILE_FINISHED);
} catch (InstantiationException e) {
Log.e(ACTIVITY_TAG,"Error. InstantiationException!");
return;
}
return;
}
```

Code 16 – Verarbeitung eines eingetroffenen Pakets in der finish-Methode des ProgressListeners

Die *finish*(…)-Methode wiederum ist zuständig für die Weiterverarbeitung der eigentlichen Daten. Sie muss zunächst prüfen, ob auch das richtige Paket eingetroffen ist. Dies geschieht über das if-Statement mit der Abfrage: *data.getType()==Type.RESPONSE_PACKET*. Handelt es sich nun wirklich um ein *RESPONSE_PACKET*, so wird aus dem *BinaryPacket* zunächst ein *RESPONSE_PACKET* und anschließend ein *RAWContainerv2*-Objekt erzeugt. Das Container-Objekt wird dann in das Dateisystem geschrieben und der Pfad im *simpleTourWrapper* vermerkt. Die Anzeige des Fortschritts wird zudem im *simpleTourWrapper* deaktiviert und abschließend die GUI benachrichtigt. Alternativ könnte es sich bei dem *BinaryPacket* jedoch auch um ein *ERROR_PACKET* handeln. In diesem Falle wird das Paket anders gehandhabt. Bspw. könnte der *Listener* versuchen, das Paket erneut anzufordern oder dem Anwender melden, dass ein Fehler aufgetreten ist.

10.2 Grafische Oberfläche – App

Im Rahmen der Bachelorarbeit war es vonnöten, die grafische Oberfläche der App zu erweitern und zu verändern. Diese Änderungen und Erweiterungen sollen in diesem Kapitel erläutert und begründet werden. Hauptbestandteile der Änderung umfassen zum einen die geänderte Oberfläche der Karte durch die Mapsforge API und zum anderen die Anpassungen des Paketbrowser an die Kommunikation zum Server. Die hiesige Reihenfolge der Betrachtungen beruht hierbei auf der Reihenfolge der Benutzung und wird daher mit dem Paketbrowser beginnen.

10.2.1Paketbrowser

Die erste Änderung, die im Paketbrowser gemacht werden musste, war eine Anpassung der Seitenreiter, auch Tabs genannt. Statt der Standardausführungen wurden individualisierte Tabs eingeführt, um in ihnen eine *Progressbar* anzeigen zu können. Diese *Progressbar* ist so konzipiert, dass sie zunächst unsichtbar ist und erst dann angezeigt wird, wenn neue Daten vom Server abgerufen werden. Zum Beispiel wenn eine Liste der verfügbaren Touren herunter geladen wird. Tritt dieser Fall ein, so visualisiert die *Progressbar* durch eine kreisende Animation im jeweiligen Tab diesen Vorgang (siehe Abb. 8).

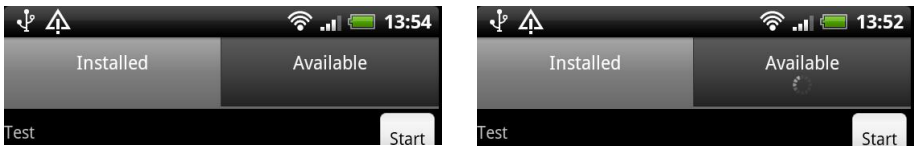

Abb. 12 – Zwei Tabs in den Zuständen „nicht ladend“ (links) und „ladend“ (rechts)

In einem weiteren notwendigen Schritt wurden die Listenelemente des Available-Tabs neu gestaltet. Der simple Text-Button zum Downloaden wurde durch einen Icon-Button ersetzt. Zudem wurde ein sogenannter *Footer* eingeführt, der als eine Art Button dazu dient, die nächsten Elemente zu laden. Dieser *Footer* befindet immer am Ende der Liste.

Sowohl die Download-, als auch die Upload-Vorgänge werden von der GUI visualisiert. Hierzu wird, in Falle eines Downloads, das Element der *Installed* Liste hinzugefügt und zum Installed-Tab gewechselt. In beiden Fällen wird dann durch eine horizontale *ProgressBar* der Fortschritt dargestellt. Ebenfalls angezeigt wird der prozentuale Fortschritt durch eine *TextView*. Nach Beendigung des Vorgangs wird der Benutzer über ein PopUp-Fenster informiert, dass der Vorgang beendet wurde. Eine

Abb. 13 – Downloadfortschritt einer Tour

gedownloadete Tour wechselt dann zu ihrer normalen Darstellung und lässt sich ganz normal verwenden.

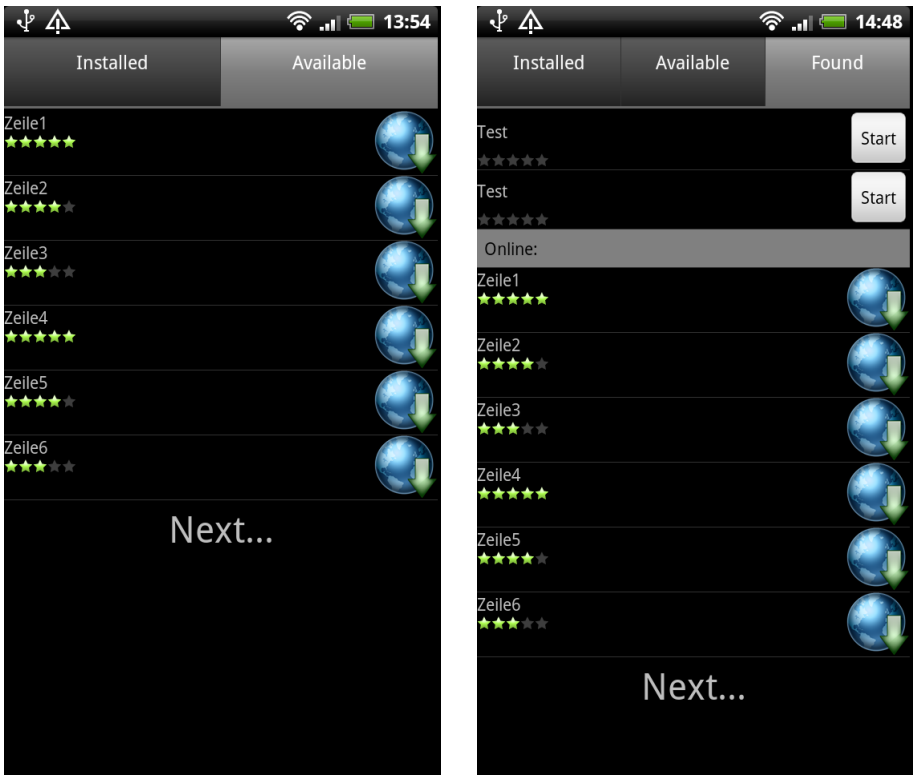

Abb. 14 – Available Tab (links) und Found Tab (rechts)

Zudem wurde der Found-Tab gänzlich neu konzipiert. Er dient nun nicht mehr wie bisher der Darstellung gefundener Touren auf dem Smartphone, sondern auch der Darstellung gefundener Touren auf dem Server. Die Liste wurde insofern erweitert, als dass sie nun zwei Gruppen von Elementen darstellen kann und diese durch einen Separator voneinander trennt. Die auf dem Gerät verfügbaren Touren finden sich im oberen Teil der Liste, da diese die geringere Anzahl von Elementen aufweist und sie im Gegensatz zum anderen Teil nicht regelmäßig erweitert wird. Zudem ist es für den Anwender angenehmer, wenn er seine Touren zuerst und somit schneller findet. Auch hier wurde der *Footer* eingesetzt, um die Liste zu vergrößern, bzw. weitere Elemente vom Server zu laden.

Auch der letzte Seitenreiter, der Info-Tab, wurde im Rahmen der Anpassungen an die Kommunikation mit dem Server erweitert. Dieser beherbergt nun neben einem Button zum Downloaden der Tour eine Liste von Bewertungen. Zusätzlich ist es hier ebenfalls möglich, selbst eine Bewertung abzugeben. Jedes Element der Bewertungen enthält den Benutzernamen, seine in Sternen ausgedrückte Bewertung und einen Kommentar. Der Bewertungsliste vorangestellt ist eine Überschrift und ihr nachfolgend der bereits bekannte *Footer*. Der

Downloadbutton hat zudem die Eigenschaft, dass er nur angezeigt wird, sollte es sich um eine online Tour handeln. Bei bereits heruntergeladenen Touren wird er ausgeblendet. Besonders zu erwähnen ist die Tatsache, dass an dieser Stelle zwei ineinander geschachtelte *ScrollViews* eingesetzt wurden. Sowohl der Tab-Inhalt als auch die Liste der Bewertungen hat eine eigene *ScrollView*. Dieser Umstand ist besonders hervorzuheben, da dieser Zustand in Android vom System aus nicht direkt unterstützt wird. Aufgrund eines Fehlers in der Berechnung der Listenmaße, musste sie manuell erfolgen, um das gewollte Ausgabeergebnis zu erreichen. Berechnet man die Maße der Liste nicht manuell, kollabiert sie. Die Implementierung einer kleinen Utility-Methode schaffte hier Abhilfe. Das Eintragen einer neuen Bewertung erfolgt hier analog zu anderen Eingaben in einem Popup-Fenster. Wie in Abb. 12.5 rechts zu sehen, erfolgt die Eingabe des Kommentars über ein *EditText* Feld und die Festlegung der Bewertung über eine *Ratingbar*. Nach dem Absenden über den Button wird die Bewertung in die Liste eingefügt und gleichzeitig im Hintergrund an den Server übertragen.

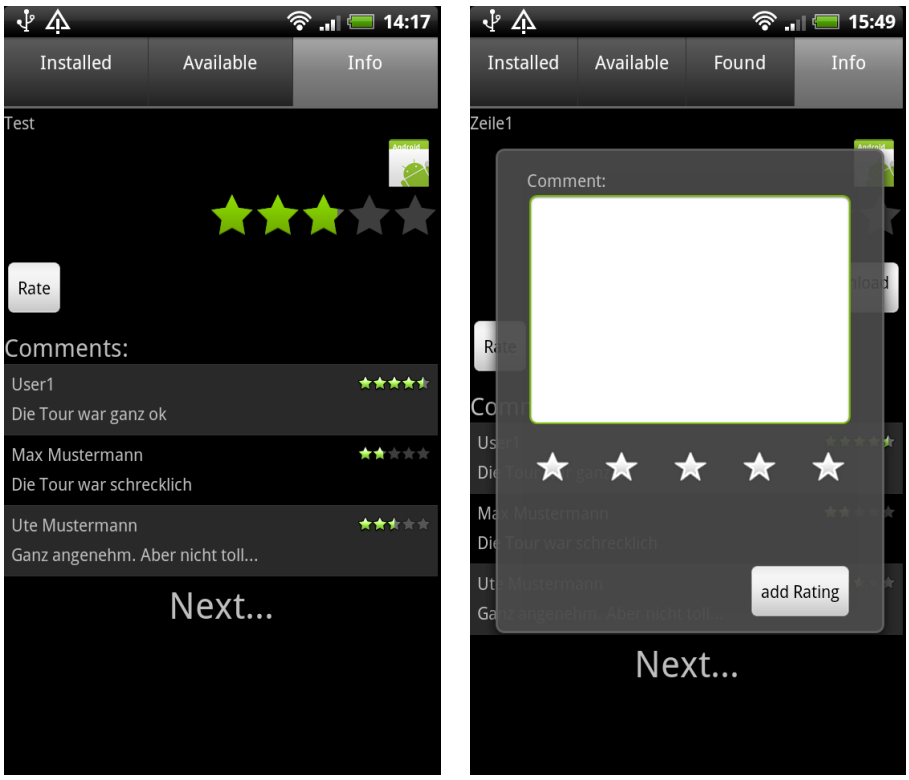

Abb. 15 – Informations-Tab (links) und addRating-Menü (rechts)

Neben diesen elementaren Änderungen wurden noch einige kleine Änderungen vorgenommen. So wurde das Contextmenü geändert, sodass es nun unterschiedliche Menüs für Online- und Offlineitems anzeigt. Des Weiteren wurden die Such- und Filter-PopUp-Fenster erweitert. Diese wurden mit der Möglichkeit erweitert, nach unterschiedlichen Eigenschaften auf dem Server zu suchen bzw. zu filtern.

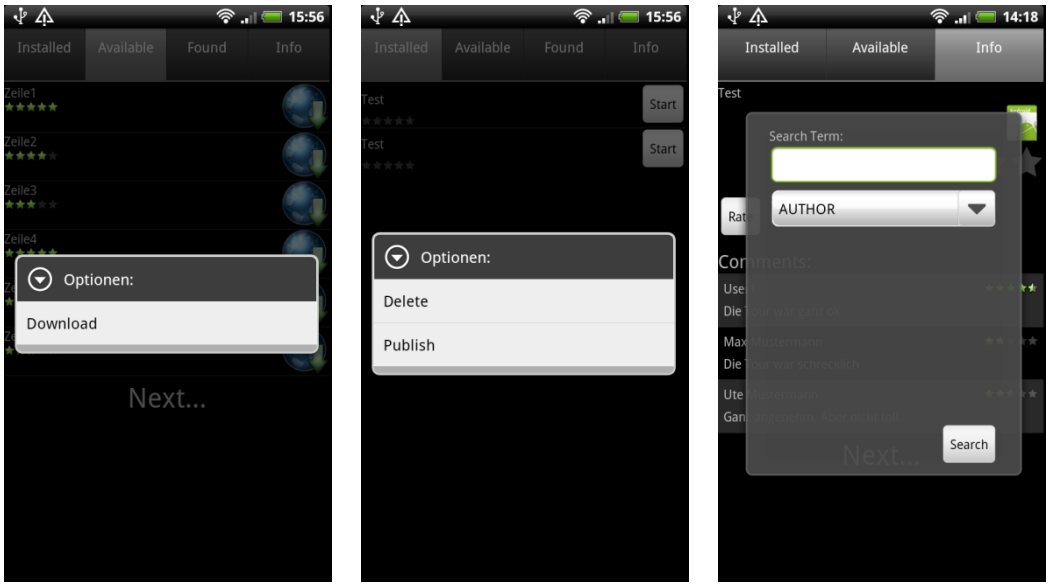

Abb. 16 – Von links nach rechts: Kontextmenü-Available, Kontextmenü-Installed, Search-PopUp-Fenster

10.2.2 Die Karte

Wie schon im Kapitel 3 erwähnt erfuhr die Karte durch die Umstellung zu Mapsforge eine größere Änderung. Zudem wurden jedoch auch Feinjustierungen und leichte Änderungen an der grafischen Oberfläche gemacht, welche ebenfalls in diesem Abschnitt genauer beschrieben werden. Erste auffälliger Punkt im Unterschied zu Mobile Maps, der aber nichts mit der eigentlichen GUI zu tun hat, jedoch aber dem Anwender direkt beim Starten der Karte positiv auffällt, ist die Tatsache, dass sich die Karte deutlich schneller lädt. Ein für den Anwender angenehmer Fall, der durch die Nutzung von Map-Dateien möglich wird. Das Laden der Daten direkt vom internen Speicher des Geräts erfolgt zwangsläufig schneller, als das Laden aus dem Internet und ist auch bei keiner bestehenden Internetverbindung möglich. Hierzu muss jedoch die passende Karte im Vorfeld herunter geladen und auf dem Gerät gespeichert werden. Die derzeit einzig nutzbare Karte, seitens der App, ist eine Deutschlandkarte. Eine Erweiterung wäre mit einem kleinen Aufwand jedoch möglich. Hierzu müsste ein neues Menü geschaffen werden, indem der Anwender die passende Map-Datei auswählen kann.

Die erste Auffälligkeit nach dem Laden ist die verbesserte Qualität der Karte. Sie ist im Gegensatz zur alten Darstellung deutlich schärfer und die Konturen der Objekte werden besser erfasst. Dies liegt auch darin begründet, dass die meisten Objekte wie etwa die Straßen jetzt einen schwarzen Umriss haben. Die Mapsforge Karte wirkt zudem nicht so blass wie die Mobile Maps Karte, da die in ihr verwendeten Farben sich deutlicher voneinander abgrenzen. Überdies verfügt die neue Karte über mehr Details. Beispielhaft hierfür sind die Waldgebiete, welche eine gewisse Textur aufweisen, anstatt über eine matte grüne Fläche dargestellt

zu werden. Neu ist auch, dass bestimmte Orte bereits markiert sind. So sind die Fachhochschule, ihre Café-Bar und ihre Mensa in der Karte verzeichnet. Als zusätzliche Steuerung ist es nun neben der bekannten Pinch-Geste möglich, mit einer Doppeltap-Geste oder den durch Mapsforge eingebauten Zoom-Buttons (siehe Abb. 16 rechts) die Größe des Kartenausschnitts zu ändern.

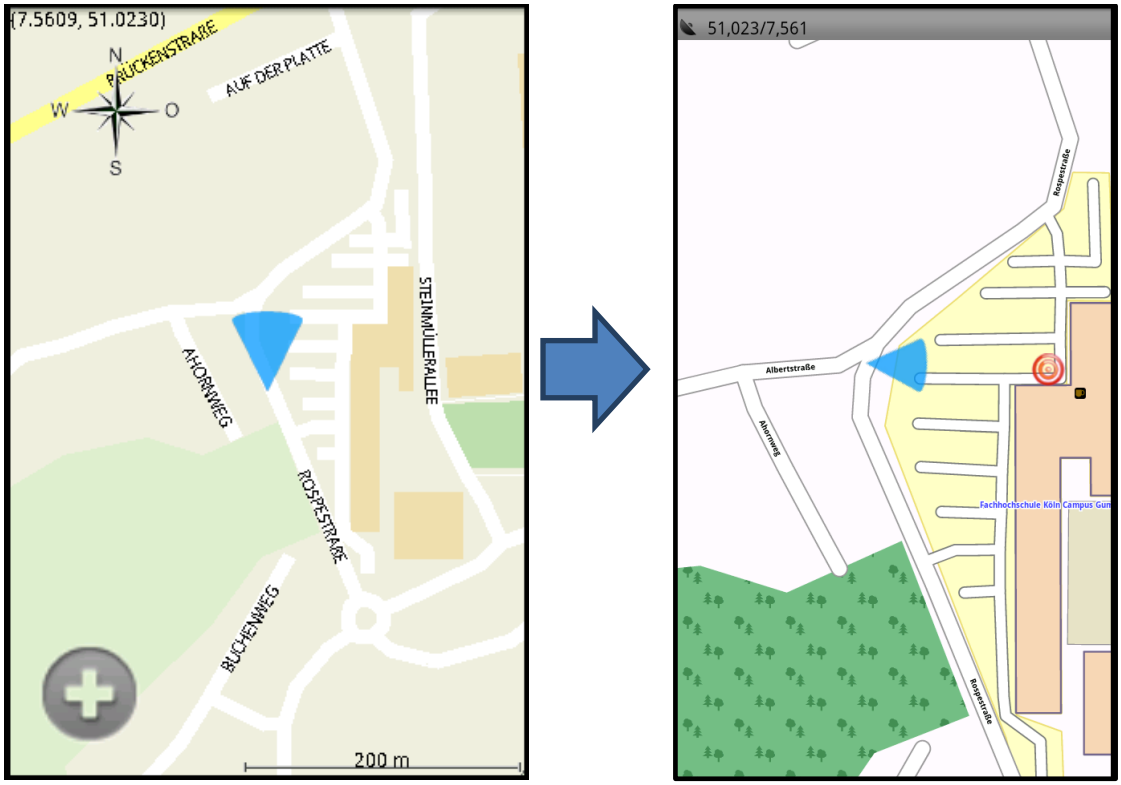

Abb. 17 – geänderte Qualität der Mobile Maps Karte (links) zur Mapsforge Karte (rechts)

Als größte eigene Neuerung wurde der obere Rand der Karte verändert. An dieser Stelle wurde eine Statusleiste eingeführt, die nun die Informationen zur Position und zur Entfernung enthält. Da die Karten-Aktivität in einem Vollbildmodus ausgeführt wird, wurde zudem ein Symbol in die obere linke Ecke platziert, welches Ausschluss darüber gibt, ob das Smartphone derzeit ein GPS-Signal empfängt. Zudem wird nur die aktuelle Position angezeigt, sollte GPS-Empfang bestehen. Im Passiv- und im Aufnahmemodus wird zudem die Entfernung zum Ziel ausgeblendet.

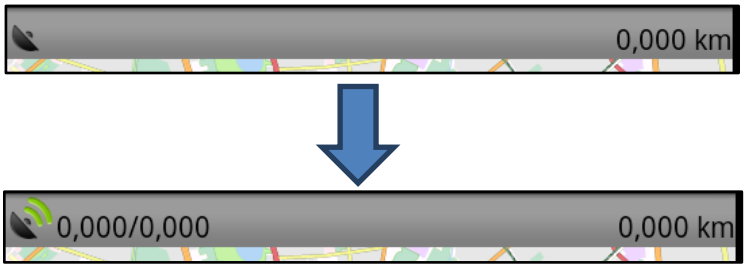

Abb. 18 – Statusleiste mit GPS aus (oben) und GPS an (unten)

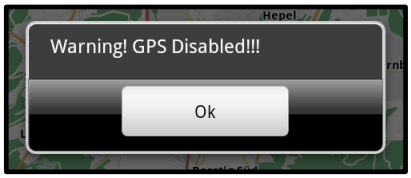

Abb. 12. 1 GPS deaktiviert Warnung

Sollte das GPS zu Beginn der Karten-Aktivität deaktiviert sein, so wird der Anwender mit einer PopUp Nachricht darüber informiert. Daraufhin kann er entsprechend seinen Wünschen eigenhändig oder über einen Menüpunkt des Optionsmenüs das GPS aktivieren.

Hierzu wurde das Optionsmenü ebenfalls erweitert. Neben der Funktionalität die Aktivität zu beenden, wurden zwei weitere Einträge hinzugefügt. Es ist es dem Anwender nun möglich, die Karte auf die eigene Position zu zentrieren. Dies ist notwendig, da es nun mit der Einführung von Mapsforge möglich ist, die Karte zu verschieben. Die Visualisierung der eigenen Position ist nun nicht mehr an die Mitte des Bildschirms, sondern an die momentane GPS Position des Smartphones gebunden. Zudem wird sie anhand des Kompasses ausgerichtet, anstatt die gesamte Karte zu drehen (siehe Abb. 13 rechts). Der Anwender ist hierdurch in der Lage, dass zu erkundende Gebiet näher zu betrachten, bevor er selbst dort ist. Dies versetzt ihn in die Lage, einfacher eine passende Route zu finden. Der zweite neue Menüpunkt ermöglicht es dem Anwender, das GPS an- und auszuschalten. Hierzu wird eine entsprechende Aktivität des Android-Systems aufgerufen. Ein direktes Manipulieren des GPS-Moduls ist, bedingt durch Android, hingegen nicht möglich.

Abb. 19 – Optionsmenü der Karten-Aktivität

Um der durch Mapsforge verbesserten Karte gerecht zu werden, wurde die Grafik für die Zielpunkte ebenfalls überarbeitet. Die Zielpunkte werden jetzt durch ineinander geschachtelte konzentrische rote Kreise dargestellt, welche die Ähnlichkeit mit einer Zielscheibe haben. Diese Optik wurde ebenfalls verwendet um den nicht eindeutigen Button zur Aufnahme eines neuen Wegpunkts zu ersetzen. Die Kreise wurden hierfür vergrößert und um ein grünes rundes Plussymbol erweitert (siehe Abb. 16 links). Diese Darstellung wurde verwendet, um dem Anwender einen intuitiven Zugang zur Funktion zu gewähren. Durch den Wiedererkennungswert der Kreise wird hier der Kontext „Wegpunkt“ dargestellt und durch das Plussymbol die Funktion verdeutlicht. Zusammenfassend kann man sagen, dass die Karte einen

deutlichen Qualitätssprung nach oben gemacht hat und somit die Wiedergaben und Aufnahmen von Touren verbessert hat.

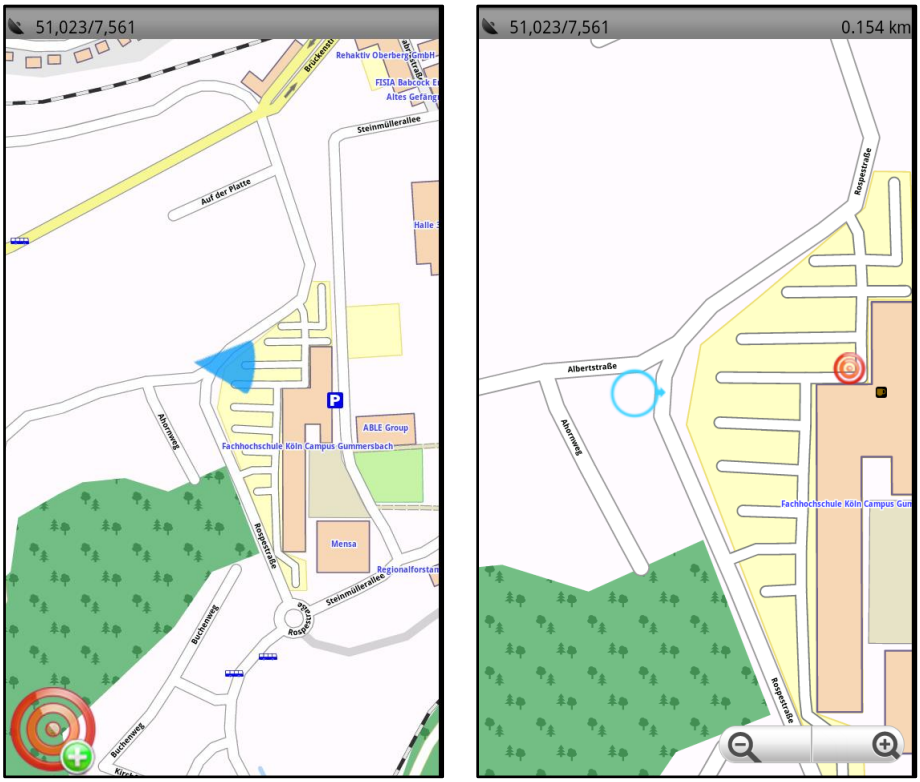

Abb. 20 – Karte im Modus „Aufnahme“ (links) und Karte im Modus „Aktiv“ (rechts)

11. Projektstand

11.1 Projektstand – Server

Dieses Kapitel beschreibt den Projektstand bei Abgabe der Bachelor-Thesis. An erster Stelle stand die Entwicklung eines Konzepts für den Server, was eine Evaluation von verfügbaren Technologien einschließt. Mit der Entscheidung für die Verwendung von *Netty* wurde letzter Punkt erfüllt.

Bei der Konzeption orientierte man sich an einigen Aspekten des *Extreme Programmings.* In der Implementierungsphase wurden letztendlich die wichtigsten Bestandteile der Software realisiert. Dies umfasst zum einen die Kommunikation über das Netzwerk. Dazu gehört die Implementierung eines Frameworks, wobei *Netty* zum Einsatz kam, als auch der eigentliche Inhalt, in Form eines Protokolls.

Ein anderer wichtiger Bestandteil bestand in der Implementierung der Kommunikation mit der Datenbank. An dieser Stelle musste sich zwischen der Verwendung von *Stored Procedures* und regulären SQL-Abfragen entschieden werden. Dabei wurde der Variante mit den SQL-Abfragen der Vorzug gegeben. Im Rahmen einer weiteren Entwicklung oder gar einem Public-Release sollten die *Stored Procedures* implementiert werden. Ebenso sollte eine Implementierung des *Oracle Universal Connection Pools er*folgen.

Den wahrscheinlich wichtigsten Part bildet die Anwendungslogik. Hierfür wurde ein Modell entwickelt, welches die Annahme einer Anfrage, deren Verarbeitung bis hin zur Generierung der korrespondierenden Antwort umfasst. Dabei wurden alle benötigten Klassen implementiert, sowie ein Handler für ein *QUERY_TOUR*-Paket. Somit kann demonstrativ eine Tourliste angefordert werden. Dabei kann auch nach einem Autoren- oder Tournamen gesucht und die Ergebnisse nach Wunsch sortiert werden. Hierfür wurde ein kleiner Bestand an Testdaten in die Tour-Tabelle eingepflegt.

11.2 Projektstand – Client

Der Stand der Software des Clients ist in Bezug auf die Kommunikation zu einem guten Teil fertiggestellt. Die nötigen *Progresslistener* wurden für jeden Anwendungsfall konzipiert und anschließend implementiert. Lediglich die Fehlerbehandlung ist noch nicht bei allen Anfragen in Gänze umgesetzt. Die Listen und ihre Darstellung wurden, wie dem Kapitel 12 zu entnehmen, bezüglich der Kommunikation angepasst und erweitert. Das eigens entwickelte

Paket *Net* wurde in das bestehende Programm eingebettet und ersten Testläufen unterzogen. Neben dem Paketbrowser wurde auch die Map verändert und zur Mapsforge API portiert. Ihre visuelle Gestaltung wurde in diesem Zuge angepasst und verbessert. Die App ist somit in der Lage einen Großteil des zu Beginn des Praxisprojekts festgelegten Umfangs zu leisten. Eine Auswahl der wichtigsten Funktionalitäten der App wäre:

- Effiziente und schnelle Kommunikation mit dem Server
- Einloggen und Registrieren am Server
- Publizieren und Downloaden von Touren
- Persistente Speichern und Verwalten von Touren
- Suchen und Filtern von Touren auf dem Smartphone und auf dem Server
- Einsehen von Tour-Informationen und Bewertungen
- Bewerten von Touren
- Durchführen von Touren in den zwei Modi Aktiv und Passiv
- Aufnahme eigener Touren samt Fotos
- Nutzung von herunterladbaren Kartenmaterial
- Steuern der Karte via Gesten (Bewegen, Zoomen)
- Langlebiger Passivmodus
- Benachrichtigung via Notifications

12. Fazit

Das Projekt YourSights findet nun mit diesen letzten Worten seinen vorläufigen Abschluss. Grundlegend bestand die Aufgabe des Projekts darin, ein Softwaresystem zu entwickeln, welches die Ausführung, Aufnahme und Veröffentlichung von Touren möglich macht. Sinn dieses Projekts ist es zudem, eine Community zu ermöglichen und hat somit einen soziologischen Aspekt. Es sollten daher Rahmenbedingungen innerhalb der Applikation für eine Entstehung einer solchen Community geschaffen werden. So wurde innerhalb der letzten Monate eine App für Android entwickelt, welche den Anwender in die Lage versetzt, Touren durchzuführen oder diese selbständig zu erstellen. Der Zugang zur Community wurde hierbei durch einen durchdachten Paketbrowser gelöst. So wurde durch die Funktionalität Touren bewerten zu können eine Möglichkeit geschaffen, dass sich Anwender untereinander austauschen können. Zudem entsteht durch die Bewertungen ein Anreiz, neue qualitativ hochwertige Touren zu veröffentlichen.

Es ist uns zudem gelungen, die Darstellung der Touren hierbei durch eine von Open-Street-Map zur Verfügung gestellten Karte zu realisieren. Die verwendete Lösung mithilfe von Mobile Maps musste zudem nach Mapsforge portiert werden. Wir waren hierdurch in der Lage, die Karte optisch deutlich aufzuwerten. Zentraler Ankerpunkt für die Community bildet der abschließend implementierte Server, welcher in Hinblick auf eine hohe Performance entwickelt wurde. Dieser mit einer Datenbankanbindung versehene Server basiert auf dem Java NIO Framework Netty, welches zu einem der leistungsfähigsten und skalierbarsten NIO-Frameworks zählt. Es ist somit denkbar möglich, bei ausreichender Hardware, zehntausende Nutzer gleichzeitig zu verarbeiten. Zur Steigerung der Performance wurde zusätzlich ein eigenes Protokoll für die Kommunikation entwickelt, anstatt mit einem auf XML oder ähnlichen Protokollen basierenden System zu arbeiten. So konnte ein perfekt auf unseren Anwendungsfall zugeschnittener Server entwickelt werden.

Neben der von uns erarbeiteten Software ist es uns zudem gelungen, Informationen zu mehreren Servertechnologien zu gewinnen. Unter den im Laufe des Projekts begutachteten und bewerteten Technologien fand sich auch eine interessante, recht neue Technologie mit dem Namen Websockets. Diese stellt eine neue Art der Kommunikation im Web dar. Ihre Weiterentwicklung und ihr Einfluss auf den mobilen Sektor sollte beobachtet werden, da sie viele Vorteile gegenüber anderen Systemen verspricht. Vor allem im Bereich der Performance ist sie gegenüber anderen Ansätzen, die rein auf http beruhen, im Vorteil. Sollte sie es schaffen

in den Bereichen der Zugänglichkeit noch weiter zu gewinnen und sollten weitere speziell auf Android oder iOS zugeschnittene Clients veröffentlicht werden, so könnte WebSockets als eine Kommunikationslösung für den Mobilen Sektor vermehrt interessant werden. Das Alexander Schulze, Gründer des jWebSockets-Frameworks, als einer der Sprecher der Mobile Developer Conference 2012[31] auftritt, zeigt, dass es sich um ein sehr aktuelles Thema handelt. Festzuhalten gilt, dass im Rahmen des Projekts eine Software erstellt wurde, welche ein breites Spektrum an unterschiedlichen Kenntnissen erforderte. Neben der Einarbeitung in das Android Betriebssystem und der dort verwendeten Mechanismen musste zusätzlich ein Server entwickelt werden, welcher in performanter Weise die Anwender untereinander verknüpft. Abgerundet wurde dies noch durch die Verwendung der Open Street Map Karte, welche durch zwei unterschiedliche Frameworks in das System eingebaut wurde.

13. Ausblick

YourSights ist ein Projekt, welches sich mit aktuellen Themen aus den Bereichen GPS-Ortung, mobile Anwendung und Datenbanken, sowie dem Themenbereich Server beschäftigt. Zieht man noch den soziologischen Aspekt hinzu, so ist YourSights ein Projekt mit Zukunft. Innerhalb dieses Projekts wurde ein erster Prototyp entwickelt, welcher die Grundidee von YourSights verkörpert. Für eine Weiterentwicklung müsste jedoch zunächst ein Finanzierungskonzept erstellt werden, damit ein Anreiz dafür geschaffen wird, dieses Projekt weiterzuführen, zumal durch die Betreibung eines Servers Kosten entstehen, welche gedeckt werden müssen. Ein bekanntes mögliches Modell ist die Verwendung von Werbung innerhalb der App. Dieses und andere Modelle müssten jedoch im weiteren Verlauf geprüft und evaluiert werden. Sollte sich dann ein Erfolg der App abzeichnen, so könnte man zusätzlich versuchen, den Kundenkreis zu erweitern. Die Erstellung eines Internetportals oder die Portierung der App auf andere Systeme wie iOS oder Windows Phone sind mögliche Vorgehensweisen. Durch die bereits enorme Größe des Projekts wäre in jedem Fall ein Zuwachs des Projektteams wichtig. Interessant wäre es hierbei auch, Personen mit anderen Qualifikationen mit an Bord zu haben. So könnte bspw. eine optische Aufwertung der App durch einen Grafiker erfolgen. Themen wie eine geleitete Routenführung oder die Verwendung einer GSM-Ortung anstatt des GPS-Systems, wären mögliche zukünftige Erweiterungen der Software

[31] Mobile Developer Conference: http://www.mobile-developer-conference.de/ (06.03.2012)

und bieten somit Raum für weitere Entwicklungen. Ob eine Weiterentwicklung der Software wirklich betrieben wird, ist zu diesem Zeitpunkt noch ungewiss. Der Bereich der mobilen Anwendungen bleibt jedoch, bedingt durch sein enormes Wachstum, ein interessantes und zukunftsträchtiges Anwendungsfeld. Wir wagen an dieser Stelle die Prognose, dass sich dies in naher Zukunft nicht ändern, sondern eher verstärken wird. Verursacht durch die steigende Leistungsfähigkeit mobiler Geräte, wird der Marktanteil mobiler Anwendungen noch weiter steigen.

14. Verzeichnisse

14.1 Abkürzungsverzeichnis

API	Application Programming Interface
DB	Datenbank
EJB	Enterprise Java Beans
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JAX-WS	Java API for XML Web Services
JDBC	Java Database Connectivity
LGPL3	GNU Lesser General Public License V3
NIO	New Input/Output
OIO	Old Input/Output
OSI	Open Systems Interaction Modell
OSM	Open Street Map
RFC	Request For Comment
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
XML	Extensible Markup Language

14.2 Abbildungsverzeichnis

Abb. 1 – Vier Phasen Modell der Projektplanung 9
Abb. 2 – Vor- und Nachteile von Mobile Maps und Mapsforge 12
Abb. 3 – Vergleich von NIO-Frameworks (1GB Netzwerk, Paketgröße 128 bytes) 19
Abb. 4 – Vergleich von NIO-Frameworks (Loopback, Paketgrößen 128 bytes) 19
Abb. 5 – Netty-Architektur Übersicht 22
Abb. 6 – Netty ChannelPipeline 23
Abb. 7 – Kommunikationsbeispiel Client - Server 32
Abb. 8 – Komponentendiagramm: Gesamtüberblick - Server 52
Abb. 9 – Komponentendiagramm: net 54
Abb. 10 – Komponentendiagramm: db 56
Abb. 11 – Kommunikation zwischen App, Netty und dem ProgressListener 63
Abb. 12 – Zwei Tabs in den Zuständen „nicht ladend“ (links) und „ladend“ (rechts) 66
Abb. 13 – Downloadfortschritt einer Tour 66
Abb. 14 – Available Tab (links) und Found Tab (rechts) 67
Abb. 15 – Informations-Tab (links) und addRating-Menü (rechts) 68
Abb. 16 – Von links nach rechts: Kontextmenü-Available, Kontextmenü-Installed, Search-PopUp-Fenster 69
Abb. 17 – geänderte Qualität der Mobile Maps Karte (links) zur Mapsforge Karte (rechts) 70
Abb. 18 – Statusleiste mit GPS aus (oben) und GPS an (unten) 70
Abb. 19 – Optionsmenü der Karten-Aktivität 71
Abb. 20 – Karte im Modus „Aufnahme“ (links) und Karte im Modus „Aktiv“ (rechts) 72

14.3 Tabellenverzeichnis

Tab. 1 – Übersicht Aktion-Aufgabe von Client und Server 27
Tab. 2 – Paketaufbau BinaryPacket 35
Tab. 3 – Übersicht Anfragepakete (Client) 36
Tab. 4 – Übersicht Antwortpakete (Server) 37
Tab. 5 – Paketaufbau REGISTER 38
Tab. 6 – Paketaufbau LOGIN 38
Tab. 7 – Paketaufbau QUERY_UPLOAD 39
Tab. 8 – Paketaufbau UPLOAD 39
Tab. 9 – Paketaufbau ADD_RATING 39
Tab. 10 – Paketaufbau QUERY_TOUR 40
Tab. 11 – Paketaufbau QUERY_RATING bzw. QUERY_PACKET 40
Tab. 12 – Paketaufbau REGISTER_FAILED 41
Tab. 13 – Paketaufbau RESPONSE_TOUR 42
Tab. 14 – Paketaufbau RESPONSE_RATING 42
Tab. 15 – Paketaufbau RESPONSE_PACKET 42
Tab. 16 – Paketaufbau RESPONSE_UPLOAD 43

14.4 Codeverzeichnis

Code 1 – Erstellung eines ServerBootstrap-Objekts 24
Code 2 – Erstellung einer ChannelPipeline 25
Code 3 – Öffnen eines neuen Ports 25
Code 4 – Implementierung der messageReceived methode 26
Code 5 – Erzeugen einer ChannelFactory 43
Code 6 – Erzeugen eines ServerBootstraps 43
Code 7 – Anlegen einer ChannelPipeline 44
Code 8 – Eigenschaften eines Bootstrap setzen 44
Code 9 – Neuen Channel erzeugen und einem Port zuweisen 44
Code 10 – connect-Methode des Clients 45
Code 11 – Ergebnisabfrage eines ChannelFuture-Objekts 45
Code 12 – Bespiel decode()-Methode eines FrameDecoders 47
Code 13 – decode()-Methode des BinaryPacketDecoders 49
Code 14 – encodeMessage()-Methode des BinaryPacketEncoders 50
Code 15 – progressChange-Methode des ProgressListeners 64
Code 16 – Verarbeitung eines eingetroffenen Pakets in der finish-Methode des ProgressListeners 65

14.5 Literaturverzeichnis

1. Broy, Manfred, Informatik. Eine grundlegende Einführung:
   Band 2. Systemstrukturen und Theoretische Informatik, Springer-Verlag (15.10.1998)

2. Carvalho, Bruno, Netty tutorial – High speed custom codecs with ReplayingDecoder,
   URL: http://biasedbit.com/netty-tutorial-replaying-decoder/ (abgerufen: 13.02.2012)

3. Cisco: Resolve IP Fragmentation, MTU, MSS, and PMTUD Issues with GRE and IPSEC,
   URL: http://www.cisco.com/en/US/tech/tk827/tk369/
   technologies_white_paper09186a00800d6979.shtml (abgerufen 12.03.2012)

4. dPunkt Verlag, Connection Pooling,
   URL: http://www.dpunkt.de/java/Programmieren_mit_Java/
   Java_Database_Connectivity/48.html (abgerufen: 10.03.2012)

5. JBOSS, Simple Echo Server Example,
   URL: http://docs.jboss.org/netty/3.2/xref/org/jboss/netty/example/echo/
   package-summary.html (abgerufen: 12.02.2012)

6. jWebSocket,
   URL: http://jwebsocket.org/ (abgerufen: 05.02.2012)

7. Mapsforge,
   URL: http://code.google.com/p/mapsforge/ (abgerufen: 05.02.2012)

8. Mapsforge, IssueList,
   URL: http://code.google.com/p/mapsforge/issues/list (abgerufen: 07.02.2012)

9. Mobile Developer Conference,
   URL: http://www.mobile-developer-conference.de/ (abgerufen: 06.03.2012)

10. Netty Dokumentation,
URL: http://docs.jboss.org/netty/3.1/api/overview-summary.html
(abgerufen: 10.03.2012)

11. Netty Project 3.2 User Guide,
URL: http://docs.jboss.org/netty/3.2/guide/pdf/netty.pdf (abgerufen: 25.02.2012)

12. Nicholas Hagen, Scalable NIO Servers – Part 1 – Performance,
URL: http://www.znetdevelopment.com/blogs/2009/04/07/
scalable-nio-servers-part-1-performance/ (abgerufen: 09.02.2012)

13. Oracle Database JDBC Developer's Guide and Reference,
URL: http://docs.oracle.com/cd/B14117_01/java.101/b10979/conncache.htm
(abgerufen: 12.03.2012)

14. Oracle, Universal Connection Pool,
URL: http://www.oracle.com/technetwork/database/enterprise-edition/downloads/
ucp-112010-099129.html (abgerufen: 10.03.2012)

15. Popovic, Miroslav, Communication Protocol Engineering, Boca Raton (2006)

16. RFC 791 – INTERNET PROTOCOL (1981),
URL: http://www.rfc-editor.org/rfc/rfc791.txt (abgerufen: 10.03.2012)

17. RFC 793 – Transmission Control Protocol (1981),
URL: http://www.rfc-editor.org/rfc/rfc793.txt (abgerufen: 10.03.2012)

18. RFC 2460 – Internet Protocol, Version 6 (IPv6)(1998),
URL: http://www.rfc-editor.org/rfc/rfc2460.txt (abgerufen: 10.03.2012)

19. Ruthmann/Stratmann: Praxisprojektdokumentation (2012)

20. Sandkühler, Bernhard, Bestandteile von Kommunikation,
URL: http://www.bernhard-sandkuehler.de/Kommunikation.html, (03.03.2012)

21. Sun Developer Network, Chapter 8 Continued: Connection Pooling
URL: http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html (abgerufen: 09.03.2012)

22. Trustin Lee, Performance Comparison between NIO Frameworks,
URL: http://gleamynode.net/articles/2232/ (abgerufen: 08.02.2012)

15. Glossar

A

Android
Android ist ein Betriebssystem für mobile Endgeräte, wie z.B. Smartphones oder Net-books. Entwickelt wird es von der *Open Handset Alliance*, deren Hauptmitglied das Unternehmen *Google* ist.

Android-Market
Die Software Android-Market dient dem Bezug von weiterer Software auf Geräten mit dem Android-Betriebssystem.

Apache
Kurz für Apache Software Foundation. Eine ehrenamtliche Organisation zur Förderung der Apache-Softwareprojekte.

API
Ein application programming interface ist eine Softwareschnittstelle, die der Anbindung weiterer Software dient.

App
Kurz für application, bzw. Applikation oder Anwendung. App steht hier stellvertretend für die Smartphoneapplikation.

App Store
Apples Pendant zu Googles *Android-Market.*

Ausführer
Bezeichnet eine Gruppe, bzw. eine Person aus der Gruppe „Ausführer“. Anwender, die Pakete laden und ausführen gehören dieser Gruppe an.

B

Bada
Bada ist ein Betriebssystem, welches für die Verwendung auf Smartphones konzipiert wurde und von Samsung Electronics entwickelt wird.

Beschleunigungssensor
Oder auch Accelerometer genannt, misst die Beschleunigung des beherbergenden Smartphones.

BLOB
Ein binary large object bezeichnet einen Datentyp in *Oracle* und anderen Datenbanksystemen. Dieser dient dem Speichern von großen, binären Objekten, wie z.B. einem Tourpaket.

Browser
Der Browser ist Teil der App und der Desktopanwendung und dient dem Anzeigen von verfügbaren oder bereits installierten Paketlisten. Über ihn können auch Pakete heruntergeladen, oder bereits installierte wieder entfernt werden.

D

Datenbankschema
Ist eine formale Beschreibung der Struktur von Daten.

Desktopanwendung
Oder auch Desktopapplikation bezeichnet eine Software, welche auf einem PC ausgeführt wird.

E

Editbox
Bezeichnet ein Feld zur Eingabe von Text und ist Teil des Android-Betriebssystems.

Enterprise Java Beans
Bezeichnet Komponenten innerhalb eines Java-Anwendungsservers die der Entwicklung komplexer Softwaresysteme in Java dienen.

Ersteller
Bezeichnet eine Gruppe, oder eine Person aus der Gruppe „Ersteller“. Ersteller nehmen Touren auf oder Bearbeiten sie.

F

Facebook
Eines der beliebtesten sozialen Netzwerke weltweit. Gegründet wurde es 2004 von Mark Zuckerberg.

Feature
Eine nennenswerte Funktionalität einer Software.

Forsa
Die „Forsa Gesellschaft für Sozialforschung und statistische Analysen mbH“ ist ein führendes Markt- und Meinungsforschungsinstitut.

Framework
Ein Programmiergerüst aus der Softwaretechnik, welches dem Programmierer oftmals einen einfacheren Zugang zu Bibliotheken ermöglicht und/oder Erweiterungen enthält.

G
Geocaching
Eine Variante der bekannten Schnitzelsuche, erweitert um den Einsatz von GPS-Empfängern.

GlassFish
Eine Implementierung eines Anwendungsservers nach dem Java-EE Standard, der auch auf das *Grizzly*-Framework zurück greift

Geronimo
Implementierung eines Anwendungsserver,
nach dem Java-EE Standard.

GoogleEarth
Eine kostenlose Software aus dem Hause *Google*, welche die Welt auf einem virtuellen Globus anhand von Satelliten- und Luftbildern darstellt. Über GPS-Koordinaten können Punkte direkt angesteuert werden.

GoogleMaps
Eine kostenlose Software aus dem Hause *Google*, die auf dem selben Kartenmaterial wie *Google Maps* beruht, zusätzlich aber auch das Verkehrsnetz darstellt. Eine Unterstützung von GPS-Koordinaten ist ebenfalls gegeben.

GPS
Das ist Global Positioning System ist ein globales Satellitennavigationssystem, welches vom US-Verteidigungsministerium entwickelt wurde.

GPX
Das GPS Exchange Format wurde von der Firma *TopoGrafix* entwickelt, basiert auf XML und speichert GPS-Daten.

Grizzly
Ein Framework für die Netzwerkkommunikation unter Java.

GUI
Steht für graphical user interface, eine grafische Benutzeroberfläche. Sie dient der Interaktion zwischen Anwender und der Maschine.

H
HTTP
Das Hypertext Transfer Protocol dient der Datenübertragung in einem Netzwerk und ist das Standardprotokoll für die Übertragung von Webseiten.

I
Intercepting Filter Pattern
Ein Vorgehensmuster aus den Core J2EE Paterns. Zeigt eine mögliche Designstrategie für die Verarbeitung eines Objekts, welches viele Instanzen durchlaufen und ggf. bearbeitet werden muss.

iOS
Ein von Apple entwickeltes Betriebssystem für die mobilen Endgeräte iPhone, iPod Touch und iPad.

iPhone
Bezeichnet eine Smartphoneserie aus dem Hause Apple, welche mit Apples iOS als Betriebssystem ausgeliefert wird.

J
Java
Bezeichnet eine objektorientierte Programmiersprache von der Firma Sun Microsystems (gehört mittlerweile zu Oracle). Das besondere dieser Sprache ist die Plattformunabhängigkeit.
Java Enterprise Edition
Ein standardisierte Softwarearchitektur für die Ausführung von An-

wendung, die in Java geschrieben wurden.

JAX-WS
Als Teil der *Java Enterprise Edition* dient die *Java API for XML Web Services* der Erstellung von Webservices.

JBoss Application Server
Implementierung eines Anwendungsserver, nach dem Java-EE Standard.

jWebSocket
Eine auf Java- und JavaScript-basierte Implementierung des HTML5-WebSocket-Protokolls.

L

Loopback
Ein Konstrukt aus der Netzwerktechnik, bei der Sender und Empfänger gleich sind.

M

Map-Tile
Ein Bildausschnitt einer Karte. Werden von Mobilen Geräten via Internet geladen. Das Gerät muss somit die Karte nicht selbst Rendern.

Mapsforge
Eine API der Freien Universität Berlin, die das einfache Einbinden von Kartenfunktionalitäten in eine Android-App ermöglicht.

Mina
Ein Framework für die Netzwerkkommunikation unter Java.

Minecraft
Ist ein von Markus Persson entwickeltes Computerspiel, welches den Spieler in eine riesige 3D-Landschaft, bestehend aus quadratischen Klötzen, versetzt. Dort kann der Spieler Rohstoffe abbauen, mit einander kombinieren und Bauwerke errichten.

Mobile Maps
Eine API der Ericsson Labs, die das einfache Einbinden von Kartenfunktionalitäten in eine Android-App ermöglicht. Die API wird seit dem 02.02.2012 nicht mehr angeboten.

Multi-Touch
Touchscreens, welche multi-touch fähig sind, erkennen Berührungen von zwei oder mehr Fingern.

N

NIO
Java *New Input/Output.* Meint die Schreib- und Leseerweiterungen des java.io-Pakets seit Version 1.4.

Netty
Ein besonders effizientes Framework für die Netzwerkkommunikation unter Java.

Nokia
Ein finnischer Telekommunikationsriese, der zu den größten Mobiltelefon-Herstellern der Welt zählt.

Notificationbar
Ein Element der grafischen Oberfläche des Android Betriebssystems. Es dient dazu, den Anwender über verschiedene Dinge zu informieren. Bspw. wenn ein Anruf verpasst wurde.

O

OIO
Java *Old Input/Output.* Meint die Schreib- und Lesefunktionalitäten des java.io-Pakets bis Version 1.4.

OpenMaps
Ein kostenloser Dienst, welcher größtenteils Kartenmaterial von Osteuropa bereitstellt.

OpenStreetMap
Ein Projekt, welches Kartenmaterial unter der „Creative Commons Attribution-Share Alike 2.0“-Lizenz sammelt und bereitstellt.

Overlay
Ein Element der GUI, welches sich über ein anderes Element legt und somit die Aufmerksamkeit des Anwenders von den dahinterliegenden Elementen weg auf das neue bewegt.

P

Paket
Bezeichnet eine Tour, welche innerhalb einer einzelnen Datei abgespeichert wird. Dieses Paket wird als BLOB in einer Datenbank gespeichert und kann über den Paketbrowser bezogen werden.

Paketbrowser
siehe Browser.

Payload
Bezeichnet die Nutzlast eines Datenpaketes, welches die eigentliche Information enthält. Wird manchmal auch als Body eines Datenpakets bezeichnet.

Pinch-Geste
Beschreibt eine Bewegung von zwei Fingern, welche sich auf einem Touchscreen aufeinander zu- oder wegbewegen. Dies bewirkt eine Vergrößerung, bzw. eine Verkleinerung der dargestellten Information.

POI
Ein Point Of Interest bezeichnet einen Punkt auf einer Karte, der für den Anwender von potentiellem Interesse ist.

Protobuf
Protocol Buffers bezeichnet ein Datenformat von Google Inc. zur Realisierung von Objekten.

Protocol
Bezeichnet Regeln für die Formatierung von Nachrichten und deren Austausch zwischen Kommunikationspartnern.

Public-Release
Beschreibt eine Version einer Software, die der Öffentlichkeit, also auch Personen, die keinen Bezug zum Projekt haben, zur Verfügung steht

R

Roaming
Bezeichnet das Durchleiten von Daten durch ein fremdes Netz. Der Eigentümer des Fremdnetzes stellt diesen Dienst i.d.R. in Rechnung.

Route
Eine Route beschreibt einen Weg zwischen zwei oder mehr Wegpunkten.

S

Secure Sockets Layer (SSL)
siehe *Transport Layer Security (TLS).*

Smartphone
Bezeichnet ein Mobiltelefon, welches mehr Computerfunktionalität und –konnektivität als ein Standardmobiltelefon zur Vefügung stellt.

Socket
Eine Softwarekomponenten mit deren Hilfe sich ein Programm mit einem Netzwerk verbindet. In Rahmen dieser Dokumentation sind stets Stream-Sockets gemeint, welche i.d.R. das TCP-Protokoll einsetzen.

Symbian
Ist ein Betriebssystem für Smartphones der Marke *Nokia*.

T

Tour
Eine Tour bezeichnet eine Menge von Wegpunkten, deren Sammlung einem übergeordneten Zweck zugrunde liegt.

TCP
Das *Transmission Control Protocol* ist ein verbindungsorientiertes Übertragungsprotokoll für Netzwerke.

Transport Layer Security (TLS)
Nachfolgebezeichnung von SSL. Ein hybrides Verschlüsselungsprotokoll zur sicheren Datenübertragung im Internet.

Twitter
Bezeichnet ein soziales Netzwerk, in welchem Privatpersonen oder Organisationen Nachrichten veröffentlichen.

U

UDP
Das *User Datagram Protocol* ist ein verbindungsloses Übertragungsprotokoll für Netzwerke.

V

View

View steht hier für eine Oberfläche der GUI bzw. eine Sammlung von GUI Elementen.

W

Webinterface

Bezeichnet eine Schnittstelle, mit welcher der Anwender über das Inter- oder Intranet, über einen Webbrowser oder einer anderen Software kommunizieren kann.

Webservice

Ein Webservice unterstützt den direkten Informationsaustausch von Anwendungen über Internetprotokolle.

Wegpunkt

Bezeichnet eine Zusammenstellung von GPS-Koordinaten und nützlichen Informationen, die dem Anwender an einer bestimmten Stelle angezeigt werden sollen.

X

XML

Mit der *Extensible Markup Language* können hierarchisch strukturierte Daten textuell dargestellt werden.

16. Anhang

Datenträger mit folgenden Inhalten:

- Dokumentation im PDF-Format
- Praxisprojektdokumentation
- Programmcode Server
- Programmcode Client

Erklärung über die selbstständige Abfassung der Arbeit

Wir versichern, die von uns vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, haben wir als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die wir für die Arbeit benutzt haben, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, den 15.03.2012

_____	_____
Kevin Ruthmann	Marcel Stratmann