UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR TELEMATIK

Universität zu Lübeck
Institut für Telematik

Bachelorarbeit

Design und Implementierung eines Peer-to-Peer-basierten Overlaynetzwerks

von
Nils Rohwedder

Aufgabenstellung und Betreuung:

Dr.-Ing. Dennis Pfisterer
Daniel Bimschas, M.Sc.

Lübeck, den 29. Dezember 2011

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, den 29. Dezember 2011

Aufgabenstellung

Im Rahmen der Europäischen FIRE-Initiative („Future Internet Research and Experimentation“, http://www.ict-fireworks.eu), die zum Großbereich „Future Internet“ (http://www.future-internet.eu) gehört, geht es um die Entwicklung von Testinfrastrukturen für die Erforschung von Technologien für das Internet der Zukunft. Das Institut für Telematik war als Konsortialführer des inzwischen abgeschlossenen WISEBED-Projekts an FIRE beteiligt. In WISEBED wurde die Infrastruktursoftware *Testbed Runtime* entwickelt, welche es erlaubt eine weltweite Testumgebung für Algorithmen, Protokolle und Anwendungen für drahtlose Sensornetzwerke zu betreiben. Testbed Runtime arbeitet dabei verteilt auf einer Vielzahl von Infrastrukturkomponenten der PC-, Server- oder Netbook-Klasse und verwendet zur Kommunikation zwischen den Komponenten ein selbstentwickeltes Overlay-Netzwerk.

Im Rahmen dieser Bachelorarbeit soll eine Middleware-Lösung für ein solches Overlay-Netzwerk entwickelt werden. Hierbei soll die zu entwickelnde Middleware verschiedene Muster des Nachrichtenaustauschs (*Message Exchange Pattern*) wie z.B. Unicast, Multicast und RPC zwischen den Peer-Hosts des Overlay-Netzwerks erlauben und realisieren. Die entwickelte Lösung soll dabei sowohl das aktuelle Overlay-Netzwerk in Testbed Runtime ersetzen, jedoch auch in anderen verteilten Anwendungen einsetzbar sein. Zu diesem Zweck sollen daher im Rahmen dieser Arbeit gängige Message Exchange Pattern analysiert, beschrieben und implementiert werden. Es soll darauf geachtet werden ausgereifte, bereits existierende Technologien zu verwenden. Die Funktionalität sowie ausreichende Performanz der Implementierung soll schließlich durch eine Evaluation gezeigt werden.

Inhaltsverzeichnis

1 Einleitung

Das Ziel dieser Bachelorarbeit ist es, eine Middleware-Lösung zu entwickeln, welche einem Nutzer erlaubt ein Overlay-Peer-to-Peer Netzwerk aufzubauen, um mit Hilfe von *Message Exchange Patterns* einen Nachrichtenaustausch zu ermöglichen.

1.1 Herangehensweise

Im Rahmen dieser Arbeit wird dabei im ersten Schritt die benötigten Message Exchange Patterns definiert und spezifiziert. Dabei handelt es sich um Nachrichtentypen der Kommunikationsform *Unicast* und *Multicast* sowie *RPC*. Im zweiten Schritt wird mittels der Definition ein Protokoll entworfen, welches die Kommunikation der verschiedenen Nachrichtentypen ermöglicht. Im dritten Schritt werden Technologien gesucht, welche zum einen den Aufbau eines Peer-to-Peer Netzwerk und zum anderen den Nachrichtenaustausch mit Hilfe der Protokollschicht ermöglichen. Im vierten Schritt wird eine Architektur entworfen und implementiert, welche sich an der gewählten Technologie sowie der entwickelten Protokollschicht orientiert. Im fünften und letzten Schritt werden zunächst Evaluationskriterien definiert und schließlich die Implementierung anhand dieser auf ihre Performanz evaluiert.

1.2 Gliederung der Arbeit

Nach der oben erfolgten Einführung in die Herangehensweise, ist die Arbeit im Folgenden in 5 Kapitel aufgeteilt.

Im Kapitel 2 wird dem Leser ein Überblick über die Grundlagen gegeben, auf denen diese Arbeit basiert. Zunächst wird in Abschnitt 2.1 das Konzept der Peer-to-Peer-Architektur erklärt. Was ist die Besonderheit gegenüber dem klassischen Client-Server-Architektur-Ansatz, was für Topologien gibt es und was sind Peer-to-Peer-Overlay-Netzwerke? Danach wird das Verfahren der Remote Procedure Calls in Abschnitt 2.2 erläutert. Anschliessend werden in den Abschnitten 2.3, 2.4 und 2.5, das Framework JBoss Netty, das Google-Protobuf-Projekt und das Uberlay-Projekt erklärt, welche die Basis der entwickelten Bibliothek bilden. Abschließend wird in den Abschnitten 2.6 und 2.7 noch kurz auf das Build-Tool Maven sowie das Test Framework Mockito eingegangen.

Das Kapitel 3 gibt dem Leser einen Überblick über den Entwurf der Imple-

mentierung. Dazu werden in Abschnitt 3.1 die zugrundeliegenden Message Exchange Patterns definiert und spezifiziert. Anschließend wird in Abschnitt 3.2 das Architektur-Konzept beleuchtet, um abschliessend das der Bibliothek zugrundeliegende Protokoll in Abschnitt 3.3 zu definieren und zu beschreiben.

Im Kapitel 4 wird die Implementierung des im Kapitel 3 beschriebenen Entwurfs vorgestellt. Dazu wird zunächst in Abschnitt 4.1 eine Einleitung sowie ein kurzer Überblick darüber gegeben, wie die Implementierung aufgebaut ist. Anschließend wird in Abschnitt 4.2 die Implementierung des Protokolls beschrieben. Danach wird der Kern der Implementierung in Abschnitt 4.3 vorgestellt. Dazu werden im einzelnen die Schnittstellen, die Implementierungen einzelner Komponenten, die Implementierung des Messaging-Systems sowie die Konfiguration einzelner Peers vorgestellt. Anschließend wird anhand von Beispielen in Abschnitt 4.4 veranschaulicht, wie die Implementierung benutzt wird. Abschließend wird in Abschnitt **??** noch kurz auf zukünftige Erweiterungen eingegangen.

Im Kapitel 5 wird die Evaluation der Implementierung beschrieben. Dazu werden zunächst die verwendeten Netzwerk-Topologien und anschließend die Evaluationskriterien erläutert. Abschließend werden die gemessenen Ergebnisse präsentiert und bewertet.

Das Kapitel 6 liefert eine kurze Zusammenfassung dieser Bachelorarbeit, einen kurzen Ausblick über die Zukunft von diesem Projekt sowie einen Überblick in welchen Anwendungsbereichen die Implementierung verwendet werden kann.

2 Grundlagen

Im diesem Kapitel werden die technischen Grundlagen dieser Arbeit erörtert, welche im Entwurf und der daraus resultierenden Implementierung angewendet werden. Im Abschnitt 2.1 wird das Peer-to-Peer Architektur-Konzept vorgestellt, welches die Basis dieser Arbeit bildet. Anschließend wird im Abschnitt 2.2 das Verfahren von Remote Procedure Calls erläutert, welches einen wesentlicher Bestandteil dieser Arbeit ist. Danach wird im Abschnitt 2.3 das Netty-Framework beschrieben, welches zusammen mit dem Uberlay Projekt, beschrieben in Abschnitt 2.5, die Basis des Architektur-Entwurfes bildet. Im Abschnitt 2.4 wird das Google Protobuf Projekt beschrieben, welches die Protokoll-Schicht dieser Arbeit bildet. Danach wird in Abschnitt 2.6 das Projektverwaltungs- und Buildtool Maven vorgestellt und erläutert in wieweit sich dieses Tool in dieser Arbeit wiederfindet. Abschließend wird die Java-Programmbibliothek Mockito vorgestellt, welche in dieser Arbeit zum Testen der Implementierung verwendet wurde.

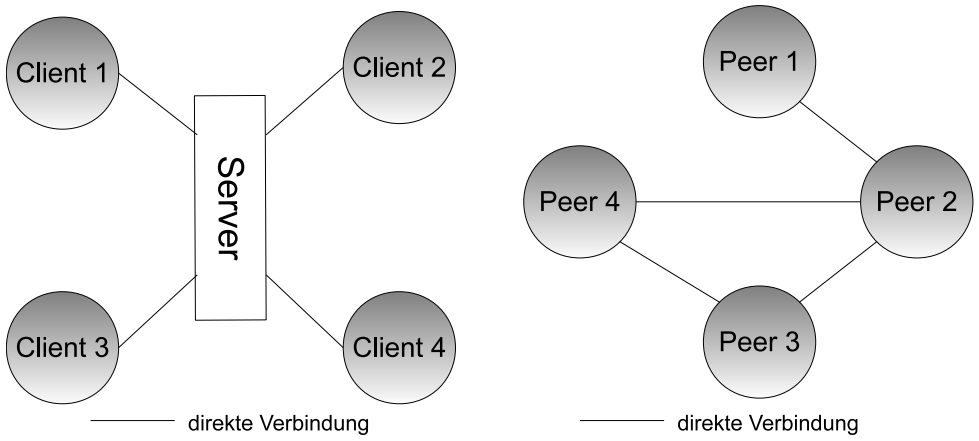
2.1 Peer-to-Peer Architektur

[3] Die klassische Internetkommunikation basiert auf der Client-Server-Architektur. Ausgewählte Server (meist abhängig von Leistungsfähigkeit) bieten Dienste, Dateien, Inhalte etc. an, die Clients konsumieren diese Ressourcen. Der Vorteil hierbei: Kennen Clients einen ausgewählten Server, so lassen sich Dienste, Daten oder Inhalte leicht auffinden. Allerdings besitzt diese Architektur einige Nachteile:

- Ist ein Server vorübergehend nicht erreichbar, so sind deren sämtlich angebotene Ressourcen ebenfalls nicht verfügbar. Dies entspricht dabei dem System des *Single Point of Failure*, d.h. der Ausfall des Servers sorgt für den Zusammenbruch des ganzen Systems (Netzwerk). Dies kann zwar durch z.B. redundante Datenhaltung verhindert werden, verschiebt dabei aber nur das eigentliche Problem des Systems und löst es nicht.
- Die theoretisch zur Verfügung stehenden Ressourcen von Clients sind für andere Clients nicht verfügbar. So ist z.B. das *Filesharing*, bei dem Nutzer untereinander Dateien teilen oder das Nutzen der *Grid Computing* Technologie in dem Rechenleistung geteilt wird, nicht oder nur bedingt möglich.

2.1.1 Was ist Peer-to-Peer?

Die Peer-to-Peer-Architektur ist ein alternatives Konzept um diese Nachteile aufzulösen.



(a) Client-Server Architektur (b) Peer-to-Peer Architektur

Abbildung 2.1: Client-Server vs. Peer-to-Peer

Peer-to-Peer-Netzwerke sind verteilte Systeme, bestehend aus mehreren Knoten, die *Peers* genannt werden. Jeder Peer im Netzwerk ist ein gleichwertiger Partner (*Peer:* engl. für *Gleichgestellter*) und kann sowohl Client- als auch Server-Aufgaben ausführen. Peers eines Netzwerks kommunizieren unmittelbar miteinander, (siehe Abbildung 2.1) dies bedeutet aber *nicht*, dass jeder Peer mit jedem weiteren Peer des Netzwerks verbunden sein muß. Es bedeutet lediglich, dass Peers keine Server benötigen, um Informationen auszutauschen. So ist im Beispiel in Abbildung 2.1 einzig Peer 1 mit Peer 2, Peer 2 mit Peer 3 und Peer 4 und Peer 3 mit Peer 4 verbunden. Das Peer-to-Peer-Architektur-Konzept erlaubt aber trotzdem die Kommunikation zwischen sämtlichen Peers des Netzwerk (wie z.B. zwischen Peer 1 und Peer 4).

2.1.2 Routing

Um die Kommunikation zwischen Peers eines Netzwerks, unabhängig ihrer Topologie, zu ermöglichen, müssen die Informationen, wo und wie einzelne Peers zu erreichen sind, für jeden Peer verfügbar sein; (möchte, bezogen auf Abbildung 2.1(b), z.B. Peer 1 mit Peer 4 kommunizieren). Diese Informationen wo und wie Peers erreicht werden können, werden in sogenannten *Routing-Tabellen* gespeichert. Die Such- und Weiterleitungsverfahren, welche die Routing-Tabellen zum Auffinden von Knoten in Netzwerken benötigen, nennt man dann *Routingverfahren*. Das Routingverfahren eines Netzwerks ist implementierungsabhängig, das bedeutet, jedes Peer-to-Peer Netzwerk kann sein eigenes Routingverfahren besitzen. Bekannte Routingverfahren in Peer-to-Peer-Netzwerken sind z.B.

CAN (Content Addressable Network) [2] oder *Chord* [2] welche auf der Datenstruktur *DHT (Distributed Hash Table)* [2] aufsetzen.

Ein Peer-to-Peer Netzwerk ist hoch-dynamisch. Da Knoten sich jederzeit vom Netzwerk an und wieder abmelden können oder Knoten vorübergehend nicht verfügbar sein können, müssen sich die Routing-Tabellen dynamisch an das Netzwerk anpassen. D.h. Peer-to-Peer Netzwerke organisieren sich selbst. Daraus resultiert eine hohe Fehlertoleranz, da selbst durch den Ausfall von einem oderer mehrerer Knoten die Kommunikation weiterhin gewährleistet bleibt.

2.1.3 Peer-to-Peer Overlay-Netzwerk

Die Kommunikation eines Peer-to-Peer Overlay-Netzwerks findet auf einer eigenen Netzwerktopologie, dem Overlay, statt. Diese Topologie setzt auf eine bestehende physikalische Netzwerktopologie, dem Underlay, auf. Der Vorteil hierbei ist, dass das Routen und Addressieren von Peers unabhängig von der pysikalischen Netzwerkstruktur arbeitet. Aus diesem Grund besitzen Overlay-Netzwerke typischerweise eigene Addressierungen und Routing-Verfahren.

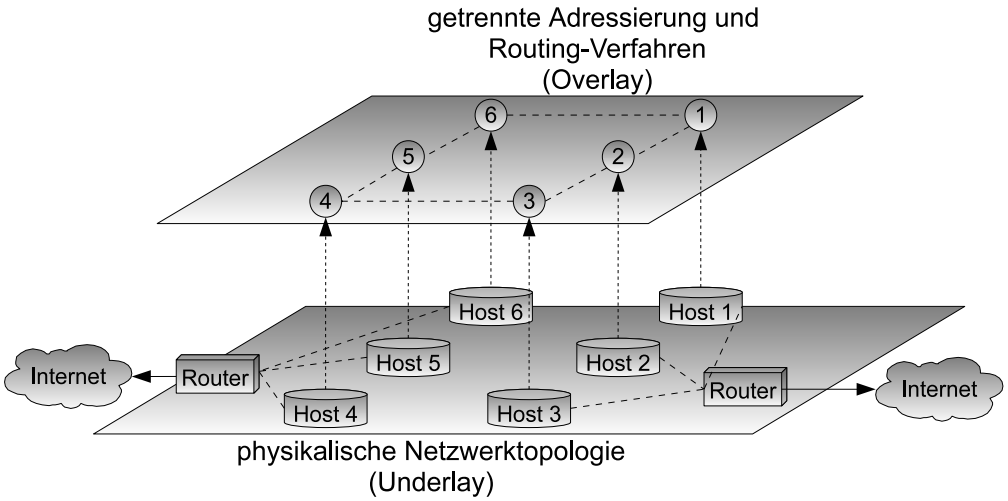


Abbildung 2.2: Netzwerk mit einer Overlay- und einer Underlay-Topologie

Ein Peer-to-Peer-Overlay-Netzwerk mit einer darunterliegenden Underlay Topologie ist in Abbildung 2.2 anhand eines Beispiels zu erkennen. Die physikalische Netzwerktopologie (Underlay) besteht dabei aus 6 Hosts, wobei jeweils 3 Hosts mittels einem Router über das Internet miteinander verbunden sind. Das Overlay wiederum besteht aus 6 Peers. Jedem Peer wird dabei jeweils ein Host des Underlay zugeordnet. Die Peers sind ringförmig miteinander verbunden und besitzen eigene Addressierungs- sowie Routingverfahren für Nachrichten, die unabhängig von der Struktur des Underlay arbeiten.

2.1.4 Topologien

Es wird zwischen zwei Arten von Peer-to-Peer-Overlay-Netzwerk-Topologien unterschieden: *Single-Hop* und *Multi-Hop*. Ein *Hop* entspricht dabei einer Etappe von einem Netzknoten zum nächsten. Im Folgenden wird nun der Unterschied dieser beiden Modelle erklärt.

Single-Hop Single-Hop-Topologie bedeutet: Bei der Kommunikation von einem Peer des Netzwerks zu einem beliebig anderen Peer wird *immer nur ein* Hop, also eine Etappe zurückgelegt.

Multi-Hop Multi-Hop-Topologie bedeutet: Bei der Kommunikation von einem Peer des Netzwerks zu einem beliebig anderen Peer, wird *immer mindestens ein* Hop zurückgelegt. Das bedeutet, ein Großteil der Daten die übertragen werden, laufen über Peers die als Zwischenstationen fungieren. So ist z.B. die in Abbildung 2.2 beschriebene Overlay-Topologie eine Multi-Hop-Topologie, da der Peer 1 eine Nachricht zum Peer 3 über den Peer 2 versenden muss.

2.2 Remote Procedure Call (RPC)

Remote Procedure Call (RPC) ist ein Verfahren zur Abwicklung von Interprozesskommunikation welches den Austausch von Daten in verteilten Systemen erlaubt. Dafür muss der Aufruf, der von einem Client zum aufrufenden Server übertragen wird, in eine Nachricht serialisiert werden. Diese Serialisierung nennt man *Marshalling*. Der serialisierte Aufruf wird dann über einen Kommunikationskanal übertragen und beim Server deserialisiert, die Deserialisierung heißt *Unmarshalling*. Der Ablauf ist dabei in Abbildung 2.3 zu erkennen.

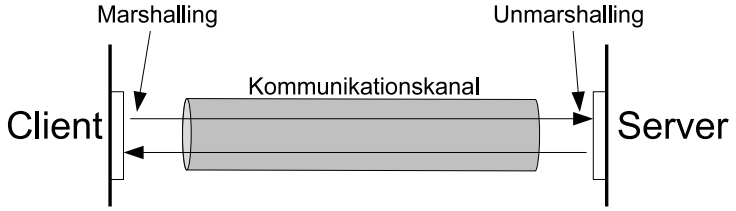


Abbildung 2.3: Übertragung eines Remote-Procedure-Call

Dieses Verfahren erlaubt den Aufruf von entfernten Prozeduren (z.B. auf einem verbundenen Remote-Peer). Der Konsument wird dabei im Allgemeinen als Client, der Dienstleister als Server bezeichnet.

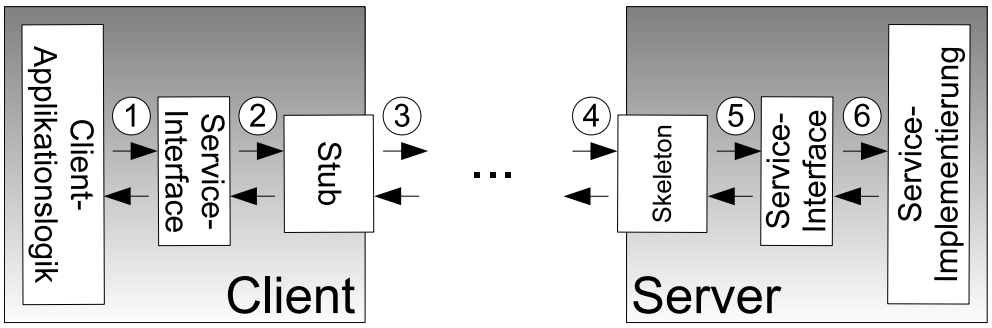


Abbildung 2.4: Ablauf eines Remote-Procedure-Call

2.2.1 Ablauf

Der Ablauf eines Remote Procedure Calls sieht wie folgt aus:

Auf dem Client wird für eine Applikationslogik (1) mittels einem Service-Interface (2) ein Stub (Stellvertreter) generiert. Dieser Stub serialisiert den Aufruf (Marshalling), welcher dann über einen Kommunikationskanal an den Server übertragen wird (3). Auf dem Server wird dann die serialisierte Nachricht vom Skeleton (4) in einen Aufruf deserialisiert (Unmarshalling). Mittels des Service-Interface (5) (dasselbe wie am Client) wird dann die entsprechende Service-Implementierung aufgerufen (6). Das Ergebnis wird über den Server-Skeleton serialisiert, an den Stub des Clients übertragen und anschliessend wieder deserialisiert. Das Ergebnis kann schließlich in der Applikationslogik am Client verarbeitet werden.

Die Abbildung 2.4 zeigt dabei den oben beschriebenen Ablauf eines RPC.

2.2.2 Realisierungen

Für die Verwendung von RPC gibt es mehrere verschiedene Arten der Realisierung. Im Zusammenhang anwendungssorientierter Middleware wäre da zum einen *CORBA* [9] zu nennen, eine Spezifikation für objektorientierte Programmiersprachen. Im Bereich kommunikationsorientierter Middleware wäre da *Web services* [12] zu nennen, ein System um den Aufruf verteilter Webanwendungen zu unterstützen. In dieser Implementierung wird die vom Google-Protobuf-Projekt [4] mitgelieferte RPC-Variante realisiert (siehe dazu Abschnitt 2.4).

2.3 Netty

Die Implementierung dieser Arbeit basiert auf Netty [8], einem asynchronen *network application framework*. Die Architektur setzt sich dabei aus drei Teilen zusammen:

- **Pipelines:** Der Transport in Netty findet über Sockets statt. Jeder Socket ist mit einer *ChannelPipeline* verbunden, welche *ChannelHandler* bereithält, die für die Verarbeitung der Nachrichten sorgen. Die Nachrichten-Pakete werden dabei mittels einem *Event-Driven* Client-Server-Model verarbeitet.
- **Codecs:** Die zu übermittelnden Objekte werden mittels entsprechender Decoder und Encoder in Nachrichten-Pakete umgewandelt.
- **Buffers:** Die Nachrichten-Pakete werden mittels des *Zero-Copy Mechanismus* in *ChannelBuffer* verpackt und dann versendet.

Im Folgenden werden zunächst die beiden eingeführten Begriffe, Event-Driven Architektur und Zero-Copy Mechanismus erklärt, bevor anschliessend die Architektur selbst beschrieben wird.

Event-Driven Architektur

Normalerweise wird auf Multi-Thread-Servern beim Empfangen von Requests ein Thread pro Request erzeugt. Dies birgt mehrere Nachteile:

- Da der Thread die meiste Zeit mit dem Warten auf I/O-Operationen verbringt, befindet sich der Thread den großteil seiner Zeit im Zustand des Leerlaufs. Daraus ergibt sich eine Verschwendung von Ressourcen.
- Außerdem ist das Erzeugen von Threads teuer und bringt den Seiteneffekt von Verzögerungszeiten mit sich. Das Verwenden von Threadpools kann dies zwar verbessern, aber nicht verhindern.

Genau hier setzt die *Event-Driven Architektur* an. Jeder Aufruf einer I/O-Operation führt zum auslösen eines nicht-blockierenden Events. So wird z.B. ein Event ausgelöst, wenn Daten auf ein Socket geschrieben werden. Das Verbinden bzw.Trennen von Kommunikationskanälen führt ebenfalls zum auslösen eines Events. Dabei wird das Verarbeiten eines Events an einen sogenannten *Worker Thread*, aus einem Pool von Threads, weitergeleitet, welcher dann das Event an die Handler in der Pipeline weiterreicht. Dabei können sich verschiedene Events einen Worker Thread teilen.

Zero-Copy Mechanismus

Beim Versenden von Nachrichten-Paketen liegen die zu versendenden Daten typischerweise auf der Festplatte oder im Kernel-eigenen Cache vor. Im Folgenden soll davon ausgegangen werden, das die Daten im Kernel-space vorliegen.

Für das Schreiben der Daten auf einem Client-Socket, müssen diese allerdings im Socket-Buffer vorliegen. Normalerweise würde dies über einen Kopiervorgang geschehen, welcher die Daten aus dem Kernel-space über den Userspace in den Socket-Buffer kopiert. Allerdings hat der Kopiervorgang vom Kernel-space in den User-space den Nachteil der schlechten Performance, zumal sich der Inhalt

nicht verändert hat.

Anstatt diese Daten nun hin und her zu kopieren, wird beim Zero-Copy-Ansatz ein virtueller Puffer im User-space erzeugt, welcher einzig *Pointer* auf den tatsächlichen Inhalt im Kernel-space generiert. Wird dann beim Schreiben der Daten in den Socket-Buffer auf den Inhalt zugegriffen, wird lediglich der Pointer gelesen und der Speicherinhalt vom Kernel-space via *Direct Memory Access* in den Socket-Buffer geschrieben. Derselbe Mechanismus greift dann ebenso beim Empfangen von Nachrichten am Server, also beim Verschieben von Daten-Paketen aus dem Socket-Buffer in den Kernel-space.

Eine ausführlichere Beschreibung der beiden Mechanismen – Event-Driven und Zero-Copy Mechanismus – ist auf der Dokumentationsseite von Netty unter http://www.jboss.org/netty/documentation.html zu finden.

In dieser Arbeit spielt der Zero-Copy Mechanismus für Nachrichten von geringer Größe eine ungeordnete Rolle, gewinnt aber an Bedeutung, je größer die zu versendenden Nachrichten werden.

2.3.1 Architektur

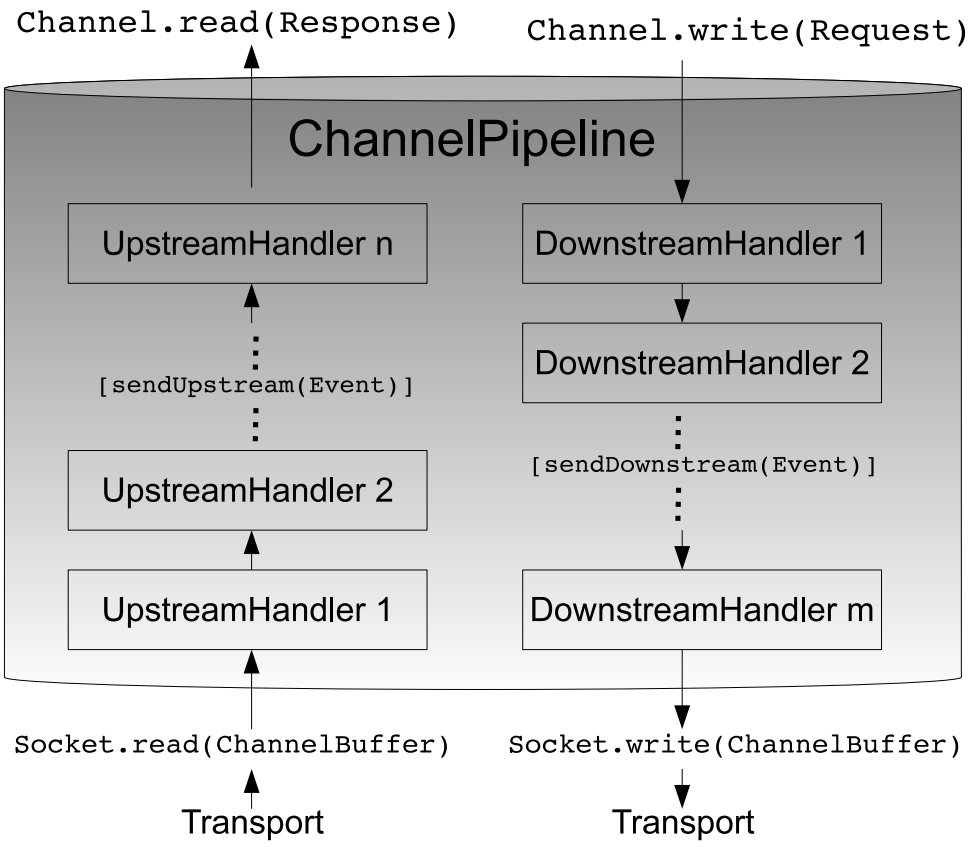


Abbildung 2.5: Vereinfachte Netty-Pipeline-Architektur

Im folgenden soll nun auf die wesentlichen, für diese Arbeit relevanten Aspekte der Netty-Architektur eingegangen werden. Hierfür soll die Abbildung 2.5 die Architektur des Netty-Frameworks veranschaulichen. Sie spiegelt dabei aber nicht die tatsächliche Realisierung in Netty wieder.

Für weiterführende Information, die über diese Arbeit hinaus gehen, steht die Netty-Projektseite zur Verfügung. (http://www.jboss.org/netty)

Pipelines und Buffers

Das Netty-Framework bietet die Möglichkeit Nachrichten von einem Client-Socket zu einem Server-Socket zu senden. Jeder Socket besitzt eine **Channel-Pipeline**. Die Kommunikation findet dabei über einen **Channel** statt, welcher zwischen einem Client-Socket und einem Server-Socket aufgebaut wird. Wie bereits erwähnt, basiert Netty auf der Event-driven Architektur, d.h. die Nachrichten werden als **ChannelEvents** verschickt. Dabei gibt es verschiedene Arten von ChannelEvents. Am wichtigsten hierbei ist das **MessageEvent**, welches den Empfänger eines Events, den **ChannelFuture** (also das Ergebnis der asynchronen Verarbeitung) sowie die Nachricht selbst enthält.

Wie in Abbildung 2.5 zu erkennen ist, enthält die ChannelPipeline beliebig viele **ChannelHandler**, die dann für die Verarbeitung der Nachricht sorgen. Das Prinzip hierbei ist wie folgt: Die Nachrichten werden in der Pipeline von einem Handler zum nächsten weitergereicht. Für die Richtung ist dabei entscheidend, ob eine Nachricht downstream (sendend) oder upstream (empfangend) verarbeitet wird. Downstream werden die Nachrichten von oben nach unten (top-down) in der Pipeline weitergereicht, upstream von unten nach oben (bottom-up). Ist ein Handler für eine Nachricht verantwortlich, so wird diese dort verarbeitet und anschließend weitergereicht, wenn nicht wird sie direkt, ohne Verarbeitung, weitergereicht. ChannelHandler können dabei unterschiedliche Aufgaben erfüllen, wie z.B. das De-/Encodieren von Nachrichten oder das Abarbeiten einer Applikationslogik. Die ChannelHandler in der Pipeline sind positionsbezogen. Beim Versenden muss schlußendlich der letzte, für den Nachrichtentyp verantwortliche ChannelHandler in der ChannelPipeline, dafür sorgen, dass die Nachricht serialisierbar ist, d.h. vom Framework über den Channel versendet werden kann.

Es gibt dabei zwei Arten von ChannelHandlern: Klassen die das Interface **ChannelDownstreamHandler** implementieren sind für das Verarbeiten von versendeten Nachrichten und Klassen die das Interface **ChannelUpstream-Handler** implementieren für das Verarbeiten von empfangenden Nachrichten zuständig.

Die Nachricht selbst, wird in einen **ChannelBuffer** verpackt und dann übertragen. Dabei wendet Netty für den Kopiervorgang den bereits beschriebenen Zero-Copy Mechanismus an. Der ChannelBuffer selbst besteht aus einem zusammengesetzten byte-array und hält für die primitiven Datentypen eigene Methoden wie, *readByte*, *readInt*, *writeByte*, *writeInt*, ... bereit.

Codecs

Die Serialisierung bzw. Deserialisierung von Nachrichten findet bei Netty über Encoder (Serialisierer) bzw. Decoder (Deserialisierer) statt. Die Dekodierte Nachricht, also die eigentliche Protokoll-Nachricht, wird dann über die Channels mittels der ChannelEvents versendet.

Für das De- bzw. Enkodieren von Nachrichten gibt es zwei Möglichkeiten: Zum Einen das Verwenden von Netty bereitgestellten Dekodern und Enkodern (wie z.B. **HttpRequestEncoder**, **HttpRequestDecoder** und **HttpResponseEncoder**, **HttpResponseDecoder**), zum Anderen das Implementieren von eigenen De- bzw. Enkodern. Die Implementierung von eigenen De- bzw. Enkodern soll hier nicht näher beschrieben werden, zumal dies anhand von Beispiel-Applikationen auf der Netty-Homepage nachvollzogen werden kann. Die genutzten De- und Enkoder müssen dann in der ChannelPipeline hinzugefügt werden.

In dieser Implementierung werden die Protokoll-Nachrichten unter Verwendung des Google-Protobuf-Projekts [4] versendet. Um diese nutzen zu können, müssen als De- bzw. Enkoder die von Netty mitgelieferten **ProtobufDecoder** und **ProtobufEncoder** der Pipeline hinzugefügt werden.

Im folgenden wird nun auf die Google-Protobuf-Nachrichten eingegangen.

2.4 Google Protocol Buffers

Google Protocol Buffers (protobuf) ist ein Format zur Serialisierung von strukturierten Daten (wie z.B. Nachrichtenprotokollen) mittels einer *Interface Definition Language* (IDL). Google Protobuf erlaubt dabei eine programmiersprachenunabhängige und effiziente Protokollnachrichten-Serialisierung.

Die Effizienz wird unter anderem dadurch erreicht, dass für die Serialisierung von zu übertragenen Daten eine Kodierung variabler Länge, sogenannte *Varints*, verwendet werden. Dabei gilt die Regel: Je größer die Zahl, desto mehr Bytes werden benötigt, d.h. die Anzahl der Bytes ist abhängig von der Größe der Zahl. Dabei wird diese Form der Serialiserung eben nicht nur für Zahlen (wie z.B. *int32*, *int64*), sondern auch für andere Datentypen (wie z.B. für Längenfelder von *Strings*) verwendet [5]. Eine genaue Erklärung wie Varints enkodiert bzw. dekodiert werden, findet sich auf der entsprechenden Dokumentationsseite von protobuf [5].

Des weiteren wurde protobuf als Binärformat konstruiert und ist deshalb schneller als andere Formate die nicht Binärdaten enthalten, aber dasselbe Prinzip verfolgen, wie z.B. textbasierte Serialisierungsformate wie XML.

Um nun strukturierte Daten zu serialisieren, muß man diese mittels einer *.proto*-Datei beschreiben. Diese *proto*-Dateien werden dann mit Hilfe des Protobuf-eigenen IDL-Compilers *protoc* generiert. Dies bietet den Vorteil der einfachen

Erweiterbar- bzw. Austauschbarkeit von Protokoll-Nachrichten.

Die Erweiterbarkeit bzw. Austauschbarkeit ist dadurch gegeben, dass zusätzlich benötigte Felder in der proto-Datei hinzugefügt bzw. ausgetauscht werden können und die Datei dann nur noch erneut kompiliert werden muß.

2.4.1 Nachrichten-Generierung

Im folgenden (Listing 2.1) soll eine Beispiel-*Proto*-Nachricht zum Versenden von Buchbestellungs-Nachrichten den Aufbau von Google Protobuf-Nachrichten veranschaulichen. Hierbei handelt es sich um ein Protokoll, bestehend aus einem *BookOrderRequest*- und *BookOrderResponse*-Nachrichtentyp. Die Nachrichtentypen bestehen dabei aus erforderlichen (*required*), optionalen (optional) und mehrfachen (repeated) Attributen, die wiederum einen Typ, einen Bezeichner und eine Reihenfolge besitzen. Die Typen bestehen dabei aus von protobuf mitgelieferten Datentypen, aus enums sowie aus den eigenen definierten Nachrichtentypen *Customer* und *Book*.

```
1  message BookOrderRequest{
2    required Customer customer = 1;
3    required Book book = 2;
4    required bool deliverToCustomer = 3;
5  }
6
7  message BookOrderResponse{
8    required uint64 orderNumber = 1;
9  }
10
11 message Customer{
12   required uint32 customerNumber = 1;
13   enum Gender{
14     UNKNOWN = 0;
15     MALE = 1;
16     FEMALE = 2;
17   }
18   required Gender gender = 2 [default = UNKNOWN];
19   required string name = 3;
20   required string address = 4;
21   optional uint32 phone = 5;
22 }
23
24 message Book{
25   required string title = 1;
26   repeated string authors = 2;
27 }
```

Listing 2.1: Aufbau einer proto-Nachricht

2.4.2 Protobuf und RPC

Des weiteren bietet protobuf die Möglichkeit RPC-Services (siehe dazu Abschnitt 2.2) zu generieren, wie im folgenden Beispiel (Listing 2.2) erläutert.

Dafür wird im ersten Schritt die Option *java_generic_services* (für Java-Applikationen) gesetzt — für weitere Programmiersprachen siehe [6]. Anschließend folgt die Beschreibung des Service, der in diesem Beispiel einen RPC-Aufruf *order* erzeugt, welcher einen BookOrderRequest übergeben bekommt und einen BookOrderResponse zurückliefert.

```
27 option java_generic_services = true;
28
29 service BookOrderService {
30  rpc order(BookOrderRequest) returns(BookOrderResponse);
31 }
```

Listing 2.2: Erweiterung der *proto*-Nachricht für RPC

Dieser Service muss dann nur noch der proto-Datei angehängt werden.

2.5 Uberlay

Das Uberlay-Projekt [7] ist ein *high-performance small-scale overlay network*. Entwickelt wurde das Uberlay-Projekt am Institut der Telematik an der Universität zu Lübeck. Es basiert auf dem Netty-Framework (siehe Abschnitt 2.3) sowie dem Google-Protobuf-Projekt (siehe Abschnitt 2.4).

Uberlay unterstützt den Aufbau von Multi-Hop Netzwerken (siehe Abschnitt 2.1.4). Da in dieser Arbeit Nachrichtentypen des Message Exchange Pattern versendet werden, musste das Uberlay dahingehend angepasst werden, dass die Nachrichtentypen, der verwendeten Multi-Hop Topologie, entsprechend übermittelt werden. Dabei wurde die Uberlay Architektur, wie in Abbildung 2.6 beschrieben, nicht verändert.

2.5.1 Architektur

Das Uberlay-Netzwerk lehnt sich konzeptuell stark an IP an. Hierzu wurde für den Paket-Transport ein einfaches (mittels Google Protobuf erstelltes) Protokoll entwickelt: das Uberlay-Protokoll (UP).

Anwendungen, welche das Uberlay-Netzwerk verwenden, hängen sich als ChannelHandler in die dafür bestimmte ChannelPipeline (**ApplicationChannelPipeline**) ein (siehe Abbildung 2.6). Dabei entspricht die ApplicationChannelPipeline einer Netty-ChannelPipeline (siehe Abschnitt 2.3) zur Verarbeitung von Nachrichten durch entsprechende Handler. Nachrichten werden, ähnlich wie bei UDP, an andere Uberlay-Hosts gesendet, indem in den versendeten Protokoll-Nachrichten zusätzlich die Adresse des Ziel-Knotens im Overlay-Netzwerk angegeben werden.

Die Klasse **UberlayNexus** verbindet nun die ChannelPipeline der Applikation, mit einer Menge von Verbindungen zum Overlay-Netzwerk. Sie implementiert eine Router-Funktionalität zum Empfangen (**UPRouter**) und Versenden

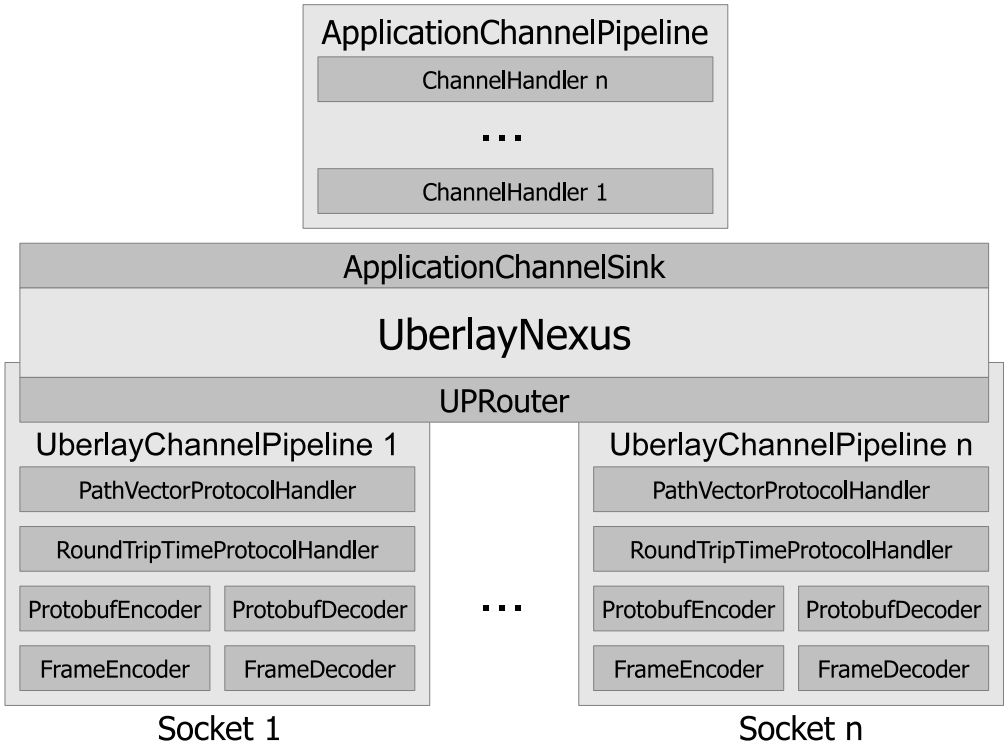


Abbildung 2.6: Uberlay Layer Architektur

von Nachrichten (**ApplicationChannelSink**) für UP und wählt je nach Zieladresse den nächsten Channel zum nächsten Hop des Multi-Hop Netzwerk. Der zu benutzende Channel wird mit Hilfe einer Routing-Tabelle ermittelt. Die Routing-Tabelle wird dabei mittels eines implementierten Pfadvektorprotokolls (siehe Abschnitt 2.5.2) erzeugt bzw. aktualisiert. Die Metrik der Routing-Tabellen wiederum wird durch das Round Trip Time Protokoll (siehe Abschnitt 2.5.3) bestimmt.

Die von Uberlay konfigurierten Encoder und Decoder, in der ChannelPipeline von Uberlay (**UberlayChannelPipeline**), sorgen dann für die Serialisierung bzw. Deserialisierung der Protokollnachrichten. Sollte die Zieladresse der eigene Netzknoten sein, so wird die Nachricht vom UPRouter wieder an die ChannelPipeline der Applikation hoch gesendet (Loopback-Funktionalität).

2.5.2 Pfadvektor Protokoll

Wie bereits in Abschnitt 2.1.2 erwähnt, besitzen Overlay-Netzwerke, zur Adressierung von Peers, Routingverfahren. Das *Pfadvektor Routing*, welches in Uberlay verwendet wird, wird nun im Folgenden erläutert, wobei dieses Routingverfahren kein Overlay-spezifisches Verfahren ist.

Algorithmus

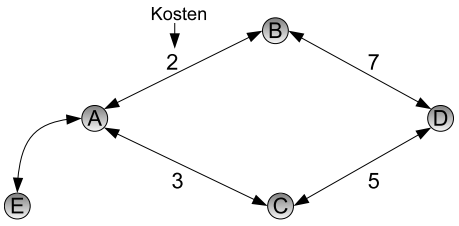
Das Uberlay-Peer-to-Peer Netzwerk unterstützt die Nutzung von Multi-Hop-Topologien (siehe Abschnitt 2.1.4). Das bedeutet, sind Peers nicht direkt miteinander verbunden, so müssen Nachrichten über dazwischenliegende Peers übertragen werden. Die Entscheidung welche Peers für die Weiterleitungen verwendet werden sollen, trifft das implementierte Routing-Protokoll. Das hier verwendete Pfadvektor Protokoll ist eines von diesen und gehört zur Familie der *Distanzvektor Routing Protokolle*.

Distanzvektor Protokolle dienen dabei zur Berechnung des kürzesten Weges von einem Start- zu einem Zielknoten. Der Algorithmus sorgt dafür, das in periodischen Abständen eine Kopie der eigenen Routing-Tabelle an die Nachbarknoten weitergegeben wird, um mittels dieser Tabellen den kürzesten Weg zu berechnen bzw. zu aktualisieren.

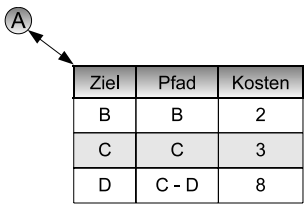
Der *Pfadvektor Protokoll Algorithmus* arbeitet wie folgt:

Jeder Knoten speichert in seiner Routing-Tabelle nicht nur die von ihm erreichbaren Zielknoten, sondern zusätzlich die möglichen Pfade zu diesem Knoten. Die so entstehende Routing-Tabelle entspricht dann dem *Spannbaum* des Netzwerks. Da sich aber sowohl der benötigte Speicherplatz als auch die benötigte Suchzeit im Vergleich zu einem Spannbaum deutlich vergrößert, wird häufig bei Implementierungen, und so auch bei Uberlay, der *minimale Spannbaum* verwendet. Eventuell ausfallende Pfade werden mittels einem Aktualisierungsintervall ersetzt, wobei dieser an die zu erwartende Stabilität des Netzwerks angepasst werden kann. Dies ist, bezogen auf die Zeit- und Platzkomplexität, im Allgemeinen kostengünstiger als die Verwendung der gesamten Spannbäume. So wird z.B. in Abbildung 2.7(b) für den Zielknoten D einzig der Pfad $A \leftrightarrow C \leftrightarrow D$ in der Routing-Tabelle gespeichert und eben nicht der auch mögliche Pfad $A \leftrightarrow B \leftrightarrow D$.

Anhand dieses Pfades kann nun der günstigste Weg von einem Ziel- zu einem Endknoten in einem Multi-Hop-Netzwerk gefunden werden.



(a) Pfadvektor-Kostenberechnung



Ziel	Pfad	Kosten
B	B	2
C	C	3
D	C - D	8

(b) RoutingTabelle von Knoten A

Abbildung 2.7: Pfadvektor Protokoll-Beispiel

Beispiel Mittels der in Abbildung 2.7(b) dargestellten Pathvector-Routing-Tabelle kann nun z.B. folgende Frage beantwortet werden:

- Frage von E: Wie komme ich am schnellsten von A nach D?
- Antwort von A: Über die Route A $\leftrightarrow$ C $\leftrightarrow$ D, da:
  $3 + 5 = \underline{8 < 9} = 2 + 7$

2.5.3 Round Trip Time Protokoll

Das Round Trip Time Protokoll in Uberlay berechnet und aktualisiert gegebenfalls die Kosten (Metrik) der Routing-Tabellen-Einträge. Die Metrik entspricht dabei der Zeitspanne die vergeht, bis ein Datenpaket von einer Quelle zu einem Ziel und wieder zurück zur Quelle gereist ist. In Abbildung 2.8 ist die Berechnung einer Round Trip Time (RTT) der Route A $\leftrightarrow$ B zu erkennen.

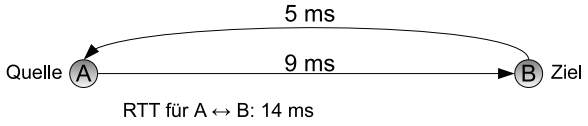


Abbildung 2.8: Berechnung einer Round Trip Time

2.6 Maven

Die Entwickler des Maven-Projekts [1] sehen Maven als *software project management and comprehension tool*. Maven wurde entwickelt, um den Build-Prozess eines Projekts zu vereinfachen, weshalb Maven primär auch heute meist als Build-Werkzeug verwendet wird. Maven bietet Entwicklern von Projekten inzwischen aber auch eine Vielzahl von weiteren Möglichkeiten:

- Erzeugen eines Build-Systems, mit dessen Hilfe ganze Projekte gebaut oder einzelne Build-Phasen durchgeführt werden können, wie z.B. das Validieren auf Korrektheit der Struktur des Projektverzeichnis und Verfügbarkeit aller benötigten Projekt-Informationen oder das Testen aller integrierten Software-Tests.
- Deklaratives Erzeugen und Verwalten von relevanten Projektinformationen wie z.B. das Verwalten von Abhängigkeiten zu anderen Projekten.
- Versionierung von Projekt-Releases, einschließlich der automatisierten Veröffentlichung auf der Projektseite.

2.6.1 Funktionsweise

POM

Das Grundgerüst von Maven ist das Konzept des **P**roject **O**bject **M**odel, basierend auf einer Projektbeschreibung mittels XML in der Datei **pom.xml** im Wurzelverzeichnis des Projekts. Sämtliche für das Projekt relevanten Informationen werden in dieser POM, bzw. in anderen Dateien die auf diese POM verweisen, gehalten. Den Aufbau einer POM ist im Listing 2.3 anhand eines leicht angepassten Ausschnittes des Core-Moduls der Implementierung dieser Arbeit zuerkennen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0␣
                http://maven.apache.org/xsd/maven-4.0.0.xsd">
5
6   <modelVersion>4.0.0</modelVersion>
7
8   <groupId>de.uniluebeck.itm.ubermep</groupId>
9   <artifactId>core</artifactId>
10  <name>Ubermep :: Core</name>
11  <packaging>jar</packaging>
12  <version>0.1</version>
13
14  <dependencies>
15   <dependency>
16    <groupId>de.uniluebeck.itm.uberlay</groupId>
17    <artifactId>core</artifactId>
18    <version>0.1</version>
19   </dependency>
20
21   <dependency>
22    <groupId>commons-logging</groupId>
23    <artifactId>commons-logging</artifactId>
24    <type>jar</type>
25    <scope>compile</scope>
26   </dependency>
27  </dependencies>
28
29 </project>
```

Listing 2.3: Aufbau einer POM

Dabei haben die XML-Tags die folgende Bedeutung:

- **groupId** enthält die eindeutige Produktgruppe unter welcher das Projekt entwickelt wird. (meist in der Form einer umgekehrten Domain wie: de.uniluebeck.
- **artifactId** enthält die eindeutige Kennung (ID) des Artefakts in der Produktgruppe. Artefakte dienen dabei dem Bauen von Projekten, z.B. beim Erzeugen einer JAR, ist der Dateiname aus `<articfactId>-<version>.jar` zusammengesetzt.

- **name** beschreibt den vollen ausgeschriebenen Namen des Projekts.
- **packaging** definiert den Erstellungstyp des Artefakts des Projekts, wie z.B. **jar**, **war**, **ear** oder **pom**. Um das Generieren kümmern sich Maven-Plugins.
- **version** enthält die aktuelle Versionsnummer des Artefakts, welche zur Versionierung von Projekten benötigt werden.
- **dependencies** enthalten die Abhängigkeiten auf Artefakte anderer Projekte.

Die komplette XML-Schema-Datei für Maven 4.0.0 ist unter `http://maven.apache.org/xsd/maven-4.0.0.xsd` verfügbar.

Verzeichnisstruktur

Maven erzeugt die Verzeichnisstruktur eines Softwareprojekts entsprechend des Prinzips *Convention over Configuration*, d.h. es verwendet Konventionen. In einem übergeordneten POM, dem Super POM, werden die Konventionen definiert. Alle Untermodule erben die Eigenschaften des Super POMs. Die Untermodule, die selbst jeweils ein POM enthalten, können geerbte Attribute überschreiben, Ausnahme hierbei, die nicht im Super POM definierten Attribute: **modelVersion** (Version der Maven-Implementierung), **groupId**, **artifactId** und **version**. Aus diesem Grund sind diese Attribute auch die einzigen, die beim Anlegen eines neuen Projekts angegeben werden müssen. Hieraus ergibt sich eine Standard-Verzeichnisstruktur wie sie im Listing 2.6.1 anhand eines Ausschnitts des Ubermep-Core Unter-Moduls zu erkennen ist.

```
ubermep-core
|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
|   |   |   `-- de
|   |   |       `-- uniluebeck
|   |   |           `-- itm
|   |   |               `-- ubermep
|   |   |                   |-- mep
|   |   |                   |   |-- Peer.java
|   |   `-- resources
|   |       |-- log4j.properties
|   |       `-- mep.proto
|   `-- test
|       `-- java
|-- target
|   |-- classes
|   |   |-- de
|   |   |   `-- uniluebeck
|   |   |       `-- itm
|   |   |           `-- ubermep
|   |   |               |-- mep
|   |   |               |   |-- Peer.class
|   |   |-- log4j.properties
```

```
26 |   |   `-- mep.proto
27 |   |-- core-0.1.jar
28 |   `-- test-classes
```

Im Hauptverzeichis befindet sich in der `pom.xml` das POM. Im Unterverzeichnis `src/main/java` befindet sich der Quellcode. Die Quellcode-Dateien werden dann in Verzeichnissen entsprechend der Namen der packages abgelegt. Zusätzlich benötigte Ressourcen finden sich im Verzeichnis `src/main/resources`. Das Verzeichnis `target/classes` enthält schließlich die compilierten Quellcode-Dateien, sowie das gesamte compilierte gepackte Projekt. Das Ausführen einzelner Build-Phasen, sogenannter *Goals*, erfolgt dabei im Wurzelverzeichnis des Projekts. Im Falle von *compile* erfolgt dies durch das Ausführen des Befehls `mvn compile`.

2.6.2 Maven in dieser Arbeit

Diese Arbeit verwendet Maven um die verschiedenen teils voneinander abhängigen Module zu trennen. Hierbei gibt es eine **Parent POM**: das **Ubermep-Parent**. Diese Parent-POM ist zuständig für sämtliche Untermodule wie **Uberlay-Parent** oder **Ubermep-Core**.

2.7 Mockito

Mockito [10] ist eine Java-Programmbibliothek zum Erstellen von sogenannten *Mock*-Objekten in Unit-Tests. *Mock*-Objekte dienen dabei als Platzhalter für tatsächliche Java-Objekte. Diese Mock-Objekte helfen dann dem Entwickler dabei das korrekte Verhalten von Interfaces zu testen. Der Vorteil hierbei: die zu testenden Interfaces können isoliert von ihren Abhängigkeiten, also ihrer Umgebung getestet werden. Diese *Mock*-Objekte können dabei *Testgetriebene Entwicklung* (TDD) bzw. *Verhaltensgetriebene Softwareentwicklung* (BDD) vereinfachen.

2.7.1 Mockito in dieser Arbeit

In dieser Arbeit wird Mockito dafür eingesetzt, um zu überprüfen und sicherzustellen, dass ausgewählte Vorgänge das tun, wofür sie vorgesehen sind. Dies wird sowohl über die korrekten Rückgabewerte als auch das Überprüfen interner Aufrufe gewährleistet. Das folgende vereinfachte Beispiel in den Listings 2.4 und 2.5 soll dabei die hier verwendete Funktionsweise veranschaulichen:

```
1 public class PhoneBook{
2     private final List<PhoneBookEntry> entryList;
3
4     public PhoneBook(List<PhoneBookEntry> entryList) {
5         this.entryList = entryList;
6     }
```

```
7
8       public void add(PhoneBookEntry entry){
9           this.entryList.add(entry);
10      }
11  }
```

Listing 2.4: Klasse PhoneBook

Es wird angenommen, es existiert das Interface *PhoneBookEntry*. Jetzt kann das korrekte Verhalten der Methode *add(PhoneBookEntry entry)* der Klasse *PhoneBook* wie folgt getestet werden:

```
1   @RunWith(MockitoJUnitRunner.class)
2   public class PhoneBookTest {
3       @Mock
4       List<PhoneBookEntry> entryList;
5
6       @Mock
7       PhoneBookEntry entry;
8
9       PhoneBook phoneBook; //zu testendes Objekt
10
11      @Test
12      public void testAdd(){
13          phoneBook = new PhoneBook(entryList);
14
15          phoneBook.add(entry);
16          verify(entryList, times(1)).add(entry);
17                                  // Bestätigen das intern die
18                                  // Methode entryList.add(entry)
19                                  // einmal aufgerufen wird
20      }
21  }
```

Listing 2.5: Testklasse PhoneBookTest

Der Vorteil hierbei: *nur* das Verhalten der Klasse *PhoneBook* wird getestet, nicht das der Interfaces *PhoneBookEntry* bzw. *List*.

2.7.2 Weitere Funktionen

An dieser Stelle soll auf die weiteren vielfältigen Funktionalitäten von Mockito nicht weiter eingegangen werde. Es sei hier nur auf die Dokumentationsseite von Mockito [11] verwiesen, wo diese näher beschrieben und anhand von Beispielen leicht nachzuvollziehen sind.

3 Entwurf

In diesem Kapitel wird der Entwurf der in Kapitel 4 beschriebenen Implementierung erklärt. In Abschnitt 3.1 werden die Message Exchange Pattern definiert und anschließend näher beschrieben.

Diese Patterns liefern die Basis für den im Abschnitt 3.2 dargestellten Architekturentwurf der Bibliothek und für das in Abschnitt 3.3 spezifizierte Protokoll.

3.1 Message Exchange Pattern

Message Exchange Pattern (MEP) ist eine Bezeichnung für ein Kommunikationsmodell, welches den Austausch von Nachrichten spezifiziert. Aufgrund dieser Patterns kann ein Protokoll, also eine Vereinbarung zwischen Knoten definiert werden, welche Typen von Nachrichten ausgetauscht werden.

Im Folgenden werden nun die Message Exchange Pattern, die in dieser Arbeit Verwendung finden, spezifiziert, wobei der Aufbau dabei wie folgt ist:

{**Pattern**}

- {Anfrage-Typ}: {Anzahl}
- {Antwort-Typ}: {Anzahl}

Anmerkung: Die in der Definition verwendete Zahl n entspricht dabei einer unbekannten, aber endlichen Zahl.

Der Anfrage-/Antwort- Typ hat dabei die folgende Bedeutung:

- *Message*: Nachricht mit der Struktur: **adresse**, **inhalt**
- *Confirmation*: Bestätigung mit der Struktur: **adresse**
- *Request*: Anfrage mit der Struktur: **adresse**, **inhalt**
- *Response*: Antwort mit der Struktur: **adresse**, **inhalt**

3.1.1 Definition

One-way:

Unicast Eine Unicast Nachricht wird an eine Adresse verschickt. Diese Nachricht beinhaltet einen Inhalt. Optional wird *eine* Bestätigung zurückgeschickt. Dabei enthält eine Bestätigung *keinen* Inhalt.

Multicast Eine Multicast Nachricht wird an mehrere Adressen verschickt. Alle Nachrichten beinhalten den selben Inhalt. Optional wird *jeweils eine* Bestätigung zurückgeschickt. Dabei enthalten die Bestätigungen *keinen* Inhalt. Eine Multicast Nachricht wird erst zum letzt möglichen Verteiler vervielfältigt, entspricht also *nicht* n Unicast Nachrichten.

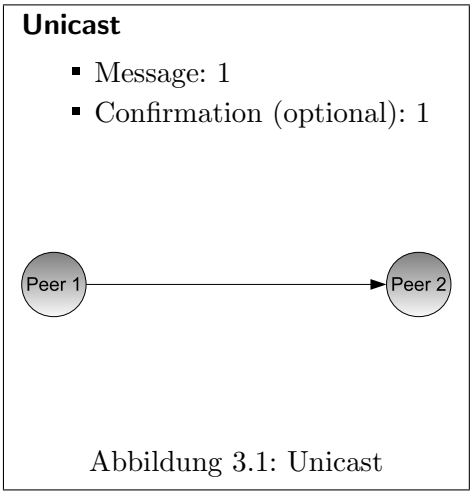


Abbildung 3.1: Unicast

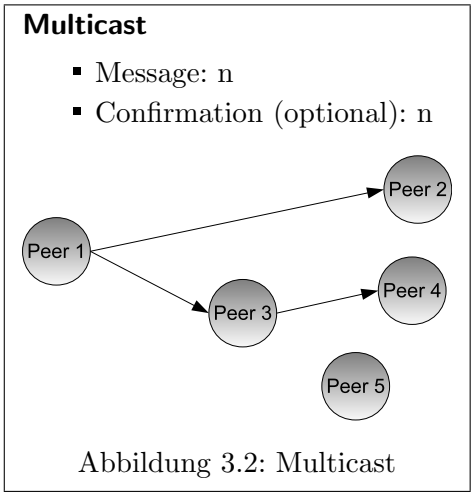


Abbildung 3.2: Multicast

Request-Response:

Single Request Single Response Eine Single Request Single Response Nachricht entspricht einer Unicast Nachricht, die *eine* Antwort zurückgeschickt. Diese Antwort enthält ebenfalls einen Inhalt.

Single Request Multi Response Eine SingleRequestMultiResponse Nachricht entspricht einer Unicast Nachricht, die *mehrere* Antworten zurückgeschickt. Diese Antworten enthalten ebenfalls jeweils einen Inhalt.

Multi Request Multi Response Eine MultiRequestMultiResponse Nachricht entspricht einer Multicast Nachricht, die *jeweils eine oder mehrere* Antworten zurückgeschickt. Diese Antworten enthalten ebenfalls jeweils einen Inhalt.

Single Request Single Response

- Request: 1
- Response: 1

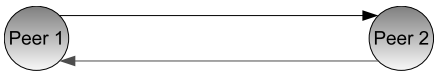


Abbildung 3.3: Single Request Single Response

Single Request Multi Response

- Request: 1
- Response: 1..n

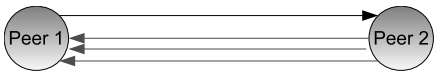


Abbildung 3.4: Single Request Multi Response

Multi Request Multi Response

- Request: 1..n
- Response: 1..m

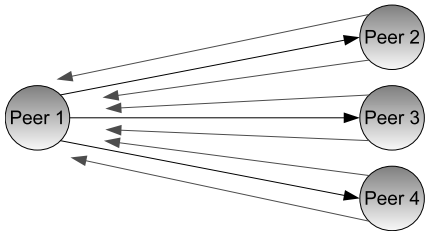


Abbildung 3.5: Multi Request Multi Response

Aus der oben beschriebenen Definition kann nun, mittels der Unterscheidung zwischen *Unreliable* (Anfrage ohne Bestätigung bzw. Antwort) und *Reliable* (Anfrage mit Bestätigung bzw. Antwort), das folgende komplette Pattern abgeleitet werden:

- Unreliable Messaging
  - Unreliable Unicast
  - Unreliable Multicast
- Reliable Messaging
  - Reliable Unicast
  - Reliable Multicast
  - Single Request Single Response
  - Single Request Multi Response
  - Multi Request Multi Response

Anmerkung

Bei der Betrachtung der oben beschriebenen Definition könnte nun die Frage aufkommen, warum hier zwischen dem *Unreliable* und *Reliable* Messaging (z.B. bei Unicast-Nachrichten) unterschieden wird. Bei jedem Nachrichtenversand eines Reliable Messaging Pattern wird jeweils eine Bestätigung bzw. Antwort zum Sender zurückgeschickt. Dieses bedeutet allerdings sowohl einen zusätzlichen Netzwerkverkehr als auch eine zusätzliche Wartezeit auf die Antwort, obwohl diese gegebenfalls gar nicht benötigt wird.

3.2 Architektur

Wie bereits erwähnt setzt diese Bachelorarbeit auf dem Netty-Framework (Abschnitt 2.3) auf. Dementsprechend orientiert sich die gewählte Architektur an diesem Framework. Die Architektur setzt sich dabei aus der Channel-Architektur und der Pipeline-Architektur zusammen.

3.2.1 Aufbau

Die gewählte Channel-Architektur ist aus den folgenden Komponenten zusammengesetzt:

- **Unicast / Multicast Channel**: wird einzig zum Versenden von Unicast und Multicast Nachrichten verwendet.
- **Single Request Single Response Channel**: wird einzig zum Versenden von Single Request Single Response Nachrichten verwendet.
- **Multi Response Channel**: wird einzig zum Versenden von Multi Response Nachrichten, namentlich Single Request Multi Response und Multi Request Multi Response verwendet.
- **RPC Channel**: wird einzig zum Aufruf von Remote Procedure Calls verwendet.

Die Zusammensetzung der einzelnen Komponenten ist in Abbildung 3.6 veranschaulicht, dabei ist die Transportschicht durch das Uberlay-Projekt (Abschnitt 2.5) gegeben. Die Channel sind mittels einer Multiplexer / Demultiplexer - Logik, mit der Transportschicht verbunden. Jeder Channel besitzt eine eigene Pipeline, die wiederum Handler enthält, die für die Verarbeitung der Nachrichten sorgen.

3.2.2 Erläuterung

Die gewählte Architektur, welche oben beschrieben ist,orientiert sich an dem Aufbau der Netty-Architektur. Unicast bzw. Multicast Nachrichten liefern Con-

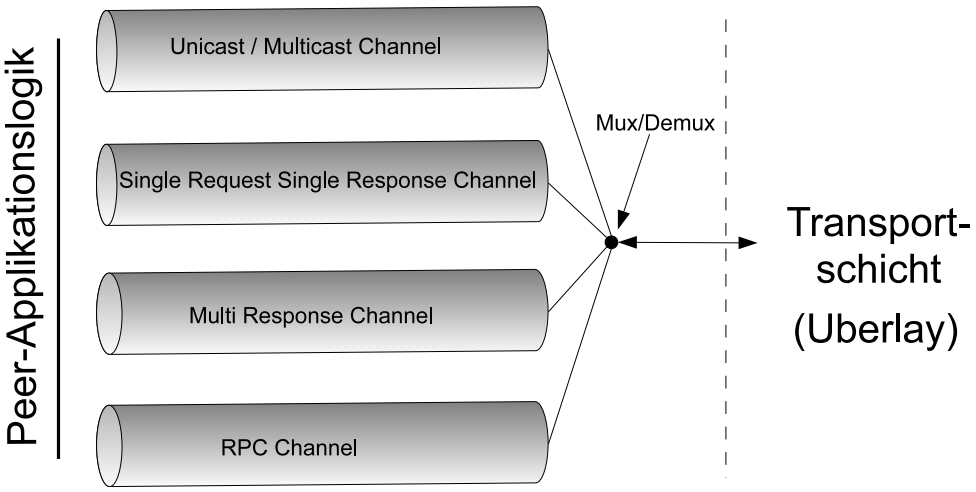


Abbildung 3.6: Vereinfachte Channel-Architektur

firmations zurück, die keinen Payload enthalten. Responses des Request-Response-Pattern, namentlich Single Request Single Response, Single Request Multi Response und Multi Request Multi Response liefern Responses zurück, welche über eigene ChannelFuture-Objekte zurückgeliefert werden. Der RPC Channel wiederum ermöglicht die Benutzung von Protobuf (Abschnitt 2.4).

3.3 Protokoll

Im Folgenden Abschnitt werden nun die, im Abschnitt 3.1 spezifizierten Message Exchange Pattern in ein Protokoll umgewandelt. Mittels diesem Protokoll können dann Nachrichtentypen des Message Exchange Pattern über den entsprechenden Channel versendet werden.

3.3.1 Struktur

Dieser Abschnitt zeigt den schematischen Aufbau des verwendeten Protokolls und beschreibt deren benötigte und optionale Felder. In Abbildung 3.7 ist die Struktur des Protokolls zu sehen. Dabei sind die optionalen Felder mit * gekennzeichnet. Diese Felder werden nur übertragen, wenn sie benötigt werden.

ID	Typ	Reliable	Akt. Nachricht*	Anz. Nachrichten*	Payload

Abbildung 3.7: Schematischer Aufbau des Protokolls

Die in Abbildung 3.7 beschriebenen Felder haben die folgende Bedeutung:

- **ID**: ermöglicht die eindeutige Identifizierung einer Nachricht.
- **Typ**: besteht aus einem der folgenden möglichen Nachrichtentypen:
  - UNICAST
  - MULTICAST
  - SINGLE_RESPONSE_REQUEST
  - MULTI_RESPONSE_REQUEST
  - SINGLE_RESPONSE
  - MULTI_RESPONSE
  - RPC_REQUEST
  - RPC_RESPONSE
- Das Flag **Reliable** wird gesetzt falls eine Nachricht zu dem *Reliable Messaging* Pattern gehört.
- **Akt. Nachricht**: wird *nur* für MultiResponse-Nachrichten übertragen. Dieses Feld zeigt an welche Nummer die entsprechende Response besitzt. Beispiel: *(Aktuelle Nachricht)* **2**, von *(Anzahl Nachrichten)* 4.
- **Anz. Nachrichten**: wird *nur* für MultiResponse-Nachrichten benötigt und beschreibt die zu erwartenden Responses.
- **Payload**: enthält den zu übertragenen Inhalt.

3.3.2 Nachrichten

Im Folgenden wird nun die oben spezifizierte Struktur auf die Nachrichtentypen des Message Exchange Pattern angewendet. Dabei wurden in der Beschreibung die Felder *ID* und *Payload* nicht berücksichtigt, da diese auf die, in diesem Entwurf dargestellten, unterschiedlichen Nachrichtentypen keinen Einfluss haben. Des weiteren werden die Elemente x und - eingeführt. Sie besitzen die folgende Bedeutung:

x := gesetzt
\- := nicht gesetzt

Unreliable Messaging

Unreliable Unicast

Typ	Reliable
UNICAST	-

(a) Unreliable Unicast-Request

Unreliable Multicast

Typ	Reliable
MULTICAST	-

(b) Unreliable Multicast-Request

Reliable Messaging

Reliable Unicast

Typ	Reliable
UNICAST	x

(c) Reliable Unicast-Request

Reliable Multicast

Typ	Reliable
MULTICAST	x

(d) Reliable Multicast-Request

Single Request Single Response

Typ	Reliable
SINGLE_RESPONSE_REQUEST	x

(e) Single Request Single Response-Request

Typ	Reliable
SINGLE_RESPONSE	x

(f) Single Request Single Response-Response

Multi Response

Im Folgenden werden Single Request Multi Response und Multi Request Multi Response Nachrichten in einem *Multi Response*-Typ zusammengefasst. Die Unterscheidung hierbei wird über das Feld ID geregelt.

Typ	Reliable
MULTI_RESPONSE_REQUEST	x

(g) Multi Response-Request

Typ	Reliable	Akt. Nachricht	Anz. Nachrichten
MULTI_RESPONSE	x	x	x

(h) Multi Response-Response

Remote Procedure Call

Typ	Reliable
RPC_REQUEST	-

(i) RPC-Request

Typ	Reliable
RPC_RESPONSE	-

(j) RPC-Response

4 Implementierung

In diesem Kapitel wird die Implementierung, genannt *Ubermep*, des in Kapitel 3 vorgestellten Architekturentwurfs erläutert.

In Abschnitt 4.1 wird zunächst ein Überblick über die Struktur der Implementierung gegeben. Danach wird erläutert wo und wie die im Grundlagenteil in Kapitel 2 vorgestellten Technologien Anwendung finden. Anschließend wird das im Entwurfskapitel spezifizierte Protokoll in Abschnitt 4.2 sowie die entwickelte Bibliothek, also der Kern der Implementierung in Abschnitt 4.3 genauer beleuchtet.

In Abschnitt 4.4 wird dann anhand von Beispielen gezeigt, wie ein Peer-to-Peer Overlay-Netzwerk mittels der Implementierung erzeugt werden kann. Weiter wird gezeigt wie man über das Netzwerk Nachrichten des Message Exchange Pattern versendet, sowie Remote Procedure Calls über das Netzwerk ausführt.

Abschließend wird in Abschnitt **??** eine Auswahl von möglichen zukünftigen Erweiterung der Implementierung vorgestellt.

4.1 Einleitung und Überblick

Wie bereits in Abschnitt 2.6 erläutert, wurde diese Arbeit mittels dem Build-Tool Maven entworfen. Daraus ergibt sich eine Projektstruktur bestehend aus verschiedenen Maven-Untermodulen welche im Folgenden Abschnitt erläutert werden.

Struktur und Komponenten

Diese Implementierung besteht aus 5 verschiedenen Untermodulen welche in einem sogenannten *Multi Module Project* zusammengefasst werden. Jedes Untermodul erbt von der Parent POM des **Ubermep-parent** Moduls. Die Aufteilung der verschiedenen Module ist hierbei im Listing 4.1 zu erkennen.

```
1 ubermep-parent
2 |-- uberlay
3 |   `-- uberlay-core
4 |-- ubermep-cmdline
5 |-- ubermep-core
6 |-- ubermep-example
7 `-- ubermep-gui
```

Listing 4.1: Ausschnitt aus der Ubermep-parent pom.xml

Im Folgenden nun eine kurze Beschreibung der einzelnen Module:

- **ubermep-parent** ist das Parent-Modul dieser Arbeit, welches den Parent-POM enthält. In seiner `pom.xml` werden dabei die enthaltenen Untermodule, sowie die in den Untermodulen benötigten Abhängigkeiten definiert. Diese Abhängigkeiten werden dann an die Untermodule vererbt.
- **ubermep-cmdline** enthält den Einstiegspunkt zum starten der Applikation aus der Kommandozeile.
- **ubermep-core** enthält die Bibliothek von ubermep. Dieses Modul hält unter anderem das zentrale Interface **Peer** bereit, welches benötigt wird um ein Overlay-Netzwerk zu erzeugen und Nachrichten des Message Exchange Pattern auszutauschen. Dieses Modul wird in Abschnitt 4.3 detailiert erklärt.
- **ubermep-example** enthält Beispiel-Anwendungen wie ubermep-core genutzt werden kann.
- **ubermep-gui** enthält eine grafische Oberfläche zum Erzeugen eines Overlay-Netzwerks sowie zum Senden von Nachrichten des Message Exchange Pattern.

Nur das Modul **Ubermep-core** enthält die Bibliothek von ubermep und deshalb wird in diesem Kapitel, in Abschnitt 4.3, auch nur auf dieses Modul näher eingegangen. Im Folgenden aber zunächst die Implementierung des in Abschnitt 3.3 vorgestellten Protokolls.

4.2 Protokoll

Wie im Abschnitt 2.4 bereits erläutert, wird zum Generieren der Protokoll-Nachrichten Google-Protobuf verwendet. Im Listing 4.2 ist die mittels der Protobuf-IDL erstellten Protokoll-Datei zu erkennen, die auf der Spezifikation des Protokoll-Entwurfs (3.3) des Entwurfskapitel 3 basiert. Diese kann unter Benutzung des Protobuf-Compiler als serialisierte Protokoll-Nachricht über den Kommunikationskanal versendet werden.

MEPPacket

Das serialisierbare Protokoll dieser Applikation, das *MEPPacket*, ist wie folgt aufgebaut:

```
1 message MEPPacket {
2     required bool reliable = 1;
3     required MessageType messageType = 2;
4     required bytes payload = 3;
5
6     optional uint32 requestID = 4;
7     optional bool exceptionOccurred = 5;
8     optional uint32 currentMessageNumber = 6;
9     optional uint32 totalMessageNumber = 7;
```

```
10      optional RPCMessage rpcMessage = 8;
11  }
```

Listing 4.2: MEP.proto

Es besteht aus den benötigten *(required)* Feldern:

- **reliable** spezifiziert ob eine Unicast bzw. Multicast-Nachricht *reliable* und unreliable ist. Request-Response-Nachrichten sind immer *reliable*.
- **messageType** beschreibt den Pattern - Typ der Nachricht. Eine genauere Beschreibung findet sich weiter unten.
- **payload** enthält den Inhalt einer Nachricht.

sowie den optionalen *(optional)* Feldern, die nur übertragen werden, wenn benötigt:

- **requestID** wird intern für die Abarbeitung von Request-Response-Nachrichten benötigt. Jeder Request bekommt beim Aufruf eine RequestID zugewiesen. Wird die über die RequestID gekennzeichnete korrespondierende Response empfangen, so wird der Request als erfolgreich ausgeliefert gewertet.
- **exceptionOccured** zeigt an ob Server-seitig, also auf der Seite des Empfängers, beim Abarbeiten des *payloads* eine Ausnahme aufgetreten ist.
- **currentMessageNumber** zeigt an, welche Nummer die Antwort besitzt. Dieses Feld wird ausschließlich für MultiResponse-Responses benötigt.
- **totalMessageNumber** zeigt an, wieviel Antworten versendet bzw. erwartet werden. Dieses Feld wird ausschließlich für MultiResponse-Responses benötigt.
- **rpcMessage** beschreibt eine RPC-Message. Eine genauere Erläuterung findet sich weiter unten.

MessageType Der Aufzählungstyp *MessageType* spezifiziert den Pattern -Typ der Nachricht und ist wie folgt aufgebaut:

```
12  enum MessageType {
13      UNICAST = 1;
14      MULTICAST = 2;
15      SINGLE_RESPONSE_REQUEST = 3;
16      MULTI_RESPONSE_REQUEST = 4;
17      SINGLE_RESPONSE = 5;
18      MULTI_RESPONSE = 6;
19      RPC_REQUEST = 7;
20      RPC_RESPONSE = 8;
21  }
```

Listing 4.3: MEP.proto

Der MessageType enthält die folgenden Nachrichten-Typen:

- **UNICAST** ist zuständig für Unicast Nachrichten; hierbei findet keine Unterscheidung zwischen unreliable bzw. reliable Unicast statt. Dies wird über das Flag **reliable** im **MEPPacket** spezifiziert.

- **MULTICAST** ist zuständig für Multicast Nachrichten; auch hierbei findet keine Unterscheidung zwischen unreliable bzw. reliable Multicast statt.
- **SINGLE_RESPONSE_REQUEST** beschreibt einen Single Request Single Response -Request.
- **MULTI_RESPONSE_REQUEST** beschreibt einen Single Request Multi Response -Request sowie für ein Multi Request Multi Response -Request.
- **SINGLE_RESPONSE** ist zuständig für einen Single Request Single Response -Response.
- **MULTI_RESPONSE** ist zuständig für einen Single Request Multi Response -Response sowie für einen Multi Request Multi Response -Response.
- **RPC_REQUEST** beschreibt einen RPC-Request.
- **RPC_RESPONSE** beschreibt einen RPC-Response.

RPCMessage Die **RpcMessage** wird, wie oben bereits erwähnt für den Aufruf eines Remote Procedure Calls benötigt und ist we folgt aufgebaut:

```
22 message RPCMessage {
23     required string serviceName = 1;
24     required string methodName = 2;
25     required ServiceType serviceType = 3;
26 }
```

Listing 4.4: MEP.proto

Der *RpcMessage*-Typ besteht aus den benötigten *(required)* Feldern:

- **serviceName** enthält den Klassen-Namen des aufzurufenden RPC-Services.
- **methodName** enthält den Methoden-Name des aufzurufenden RPC-Services.
- **serviceType** beschreibt den Typ des aufzurufenden Service. Eine genauere Beschreibung findet sich weiter unten.

Da die Parameter immer mittels eigens generierten Protobuf-RPC-Requests übertragen werden (siehe Abschnitt 2.4), brauchen diese nicht in der RPCMessage integriert zu werden.

ServiceType In ubermep wird zwischen einem blockierendem und nicht-blockierendem RPC-Aufruf unterschieden. Aus diesem Grund muss der **ServiceType** für die Beschreibung des *RpcMessage*-Typ übertragen werden. Dieser ist wie folgt aufgebaut:

```
27 enum ServiceType {
28     SERVICE = 1;
29     BLOCKING_SERVICE = 2;
30 }
```

Listing 4.5: MEP.proto

Der *ServiceType* besteht aus den Feldern:

- **SERVICE** wird gesetzt wenn der RPC-Service als nicht-blockierender Aufruf verwendet wird.
- **BLOCKING_SERVICE** wird gesetzt wenn der RPC-Service als blockierender Aufruf verwendet wird.

4.3 Ubermep-core

Das Modul **Ubermep-core** enthält zum Einen das oben beschriebene Protokoll, als auch wie bereits erwähnt die Kernimplementierung dieser Arbeit. Es enthält das Interface **Peer** zum Erzeugen von Peers. Mit diesem ist der Aufbau eines Peer-to-Peer Overlay-Netzwerks möglich. Über das Netzwerk können dann Nachrichten von einem Peer zu einem weiteren Peer, bzw. mehreren weiteren Peers gesendet werden. Des weiteren ist es möglich, über einen Peer des Netzwerks, einen Remote Procedure Call (siehe Abschnitt 2.2) auf einem anderen Peer des Overlay-Netzwerks auszuführen. Im Folgenden wird nun erläutert, wie die Struktur des Moduls **Ubermep-core** aufgebaut ist.

4.3.1 Einleitung

Ein Peer in ubermep besitzt die Aufgabe, ein Overlay-Netzwerk zu erzeugen oder sich mit einem bestehenden Netzwerk zu verbinden bzw. ggf. sich von diesem zu trennen Hierfür wird die Möglichkeit des Startens und Stoppens eines Peers benötigt. Des weiteren wird neben dem Senden von unterstützten Nachrichtentypen des Message Exchange Pattern, die Möglichkeit zum Verarbeiten von empfangenden Nachrichteninhalten benötigt. Abschließend ist das Ausführen von Remote-Procedure Calls sowie des Hinzufügen von zusätzlichen Handlern zum Verarbeiten von eigenen Nachrichtentypen erforderlich.

4.3.2 Schnittstellen

Im Folgenden die entscheidenden Schnittstellen des Ubermep-core Moduls, welche die oben beschriebenen Aufgaben definieren. Anschließend folgen die entsprechenden Implementierung der Schnittstellen sowie die Abhängigkeiten der Schnittstellen untereinander.

Peer

Aus der Beschreibung zum Erzeugen, Verbinden bzw. Trennen eines Peers ergibt sich die folgende Schnittstelle:

- *getLocalUPAddress()* gibt die lokale UPAdresse zurück, welche die eindeutige Adressierung für Nachrichten in einem Overlay-Netzwerk ermöglicht.

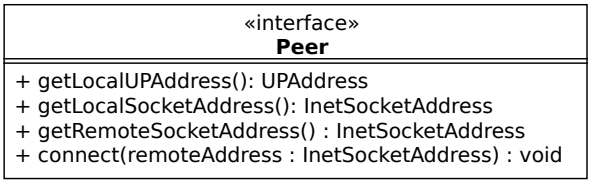


Abbildung 4.1: Aufbau des Peer-Interface

Sie ist vom Typ **UPAddress**, kommt aus dem Uberlay-Projekt (2.5) und ist eine Unterklasse vom Typ **SocketAddress** aus dem Package *java.net*. Die typische Addressierung in einem Peer-to-Peer Overlay-Netzwerk erfolgt dabei z.B. über eine **URN** (Uniform Resource Name), welche wie folgt aufgebaut ist: *urn:namensraum:id*, also z.B. *urn:itm:1* für den ersten Knoten im itm-Namensraum.

- *getLocalSocketAddress()* gibt die lokale Socket-Adresse zurück. Sie ist vom Typ **InetSocketAddress** aus dem Package java.net, enthält einen *hostname*, typischerweise eine IPv4 oder eine IPv6 Adresse und einen Port. An diese lokale Socket-Adresse wird der Netty-Channel zum Nachrichtenaustausch gebunden.
- *getRemoteSocketAddress()* gibt die remote Socket-Adresse vom Typ **InetSocketAddress** zurück, mit welcher der Peer beim Start verbunden wird.
- *connect(InetSocketAddress remoteAddress)* verbindet den Peer mit einem weiteren Peer welcher zu der übergebenen *remoteAddress* gehört.

Die weiteren Aufgaben eines Peers in ubermep sind nun in den folgenden Service-Interfaces definiert.

Service

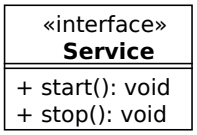


Abbildung 4.2: Aufbau des Service-Interface

Die Aufgaben eines Services sind:

- *start()* startet einen Peer und erzeugt bzw. verbindet diesen mit einem Netzwerk.
- *stop()* stoppt einen Peer und trennt diesen vom Netzwerk.

UbermepService

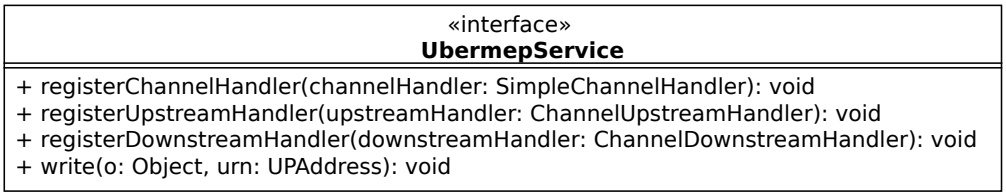


Abbildung 4.3: Aufbau des UbermepService-Interface

Der UbermepService hat die Aufgabe eigene hinzugefügte Nachrichtentypen zu unterstützen.

Unterstützung eigener Nachrichtentypen Für die Unterstützung von eigens hinzugefügten Nachrichtentypen sind die folgenden Methoden von Bedeutung:

- *registerChannelHandler(SimpleChannelHandler handler)* fügt einen implementierten SimpleChannelHandler (aus dem Netty-Framework 2.3) an einem Peer hinzu. SimpleChannelHandler können sowohl empfangende, als auch gesendete Nachrichten verarbeiten.
- *registerUpstreamHandler(ChannelUpstreamHandler handler)* fügt einen implementierten ChannelUpstreamHandler (aus dem Netty-Framework 2.3) an einem Peer hinzu. ChannelUpstreamHandler sind einzig für das Verarbeiten von Empfangenden Nachrichten zuständig.
- *registerDownstreamHandler(ChannelDownstreamHandler handler)* fügt einen implementierten ChannelDownstreamHandler (aus dem Netty-Framework 2.3) an einem Peer hinzu. ChannelDownstreamHandler sind einzig für das Verarbeiten von gesendeten Nachrichten zuständig.

Für das Nutzen von eigenen Nachrichtentypen müssen dann eigene entsprechende Handler implementiert und an einem Peer mittels der entsprechenden *register*-Methode registriert werden. In den eigens implementierten ChannelHandlern müssen die jeweiligen zuständigen Methoden überschrieben bzw. implementiert werden. Diese ChannelHandler werden der ChannelPipeline der entsprechenden Peers gemäß der Architektur des Netty-Frameworks hinzugefügt.

Für die Implementierung gibt es dabei drei verschiedene Varianten, wobei aber jeweils immer für das Verarbeiten von empfangenden Nachrichten die Methode *handleUpstream(ChannelHandlerContext ctx, ChannelEvent e)*, sowie fürs Verarbeiten von versendeten Nachrichten die Methode *handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)* überschrieben werden muss:

- **Variante 1**: Umbau einer Empfangenden / Gesendeten Nachricht in einen unterstützten Nachrichtentyp des Message Exchange Pattern wie z.B. in eine SingleRequestSingleResponse-Nachricht.

- **Variante 2**: Hinzufügen von eigenen oder mitgelieferten Netty- Decodern bzw. Encodern (2.3)
- **Variante 3**: Hinzufügen von eigenen Protokoll-Nachrichten mittels des Google-Protobuf Projekts (2.4). Zusätzlich muss dann ein ProtobufDecoder für die entsprechende Protokoll-Nachricht der Pipeline hinzugefügt werden.

Anschliessend können dann diese Nachrichten mittels der folgenden Methoden versendet werden:

- *write (Object object, UPAddress urn)* versendet eine *One-way* -Pattern-Nachricht an einen Empfänger
- *write(Object object, UPAddress urn, Class channelFutureClass)* versendet eine *Request-Response* -Pattern-Nachricht und bekommt eine Antwort mittels der als Parameter übergebenen ChannelFuture-Klasse zurück. Die übergegebene ChannelFuture-Klasse muss eine Implementierung der Abstrakten Klasse **UbermepAbstractChannelFuture** aus dem Package *mep.channel.future* sein. Wobei einzig und allein der Konstruktur implementiert werden muss.

UbermepUnreliableService

Abbildung 4.4: Aufbau des UbermepUnreliableService-Interface

Die Aufgabe des UbemepUnreliableService liegt in der Versendung von Unreliable Nachrichten (siehe Kapitel 3.1) und hat dementsprechend nur die eine Methode:

- *send(UnreliableRequest request)* versendet einen UnreliableRequest.

UbermepReliableService

Abbildung 4.5: Aufbau des UbermepReliableService-Interface

Die Aufgabe des UbermepReliableService liegt in der Versendung von Reliable Nachrichten (siehe Kapitel 3.1) und hat dementsprechend nur die eine Methode:

- *send(ReliableRequest request)* versendet einen ReliableRequest und gibt den Response als ListenableFuture-Objekt zurück.

UbermepRpcService

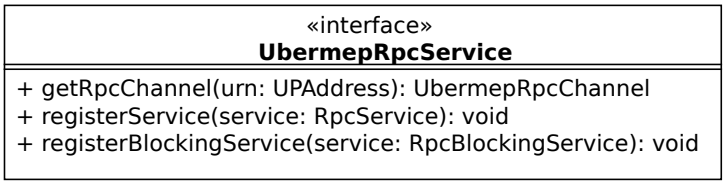


Abbildung 4.6: Aufbau des UbermepRpcService-Interface

Die Aufgabe des UbermepRpcService liegt in der Verarbeitung von Remote Procedure Calls (RPC). Für das Ausführen von RPC's sind dabei die folgenden Methoden von Bedeutung:

- *getRpcChannel(UPAddress urn)* erzeugt einen RpcChannel zu einer Adresse eines Peers des Overlay-Netzwerk.
- *registerService(RpcService service)* registriert einen RpcService an einem Peer, so dass dieser von anderen Peers aufgerufen werden kann,
- *registerBlockingService(RpcBlockingService service)* registriert einen RpcBlockingService an einem Peer zum Ausführen eines RPC.

Der Ablauf ist dabei wie folgt: zuerst registriert man einen RpcService an einem Peer. Anschliessend lässt man sich einen RpcChannel zu einer Adresse eines Peers erzeugen. Über diesen Channel kann man dann einen RPC ausführen. Der Unterschied zwischen einem (Non-Blocking) RpcService und einem RpcBlockingService ist der, das ein RpcService ein nicht-blockierender Aufruf ist, welcher den Rückgabewert in einem RpcCallback zurückgibt. Der RpcBlockingService ist ein blockierender Aufruf, der den Rückgabewert über den Methodenauruf zurückbekommt.

Das Erzeugen und Aufrufen eines RPC ist im Abschnitt 4.4 anhand eines Beispielaufrufes zu erkennen.

RequestListenerService

Zum Verarbeiten von Requests werden *Listener* verwendet. Diese Listener werden mittels der folgenden Methoden an einem Peer registriert:

- *addRequestListener(UnicastMulticastRequestListener listener)* fügt ein UnicastMulticastRequestListener an dem jeweiligen Peer hinzu.
- *addRequestListener(SingleRequestSingleResponseRequestListener listener)* fügt ein SingleRequestSingleResponseRequestListener an dem jeweiligen Peer hinzu.

Abbildung 4.7: Aufbau des RequestListenerService-Interface

- *addRequestListener(MultiResponseRequestListener listener)* fügt ein MultiResponseRequestListener an dem jeweiligen Peer hinzu.

Diese Listener arbeiten dabei nach dem Pattern der *Chain of responsibility*. Das bedeutet, die Listener eines Typs werden hintereinander in einer Kette angeordnet. Dann durchläuft eine Anfrage die Kette. Ist ein Listener dabei, der auf die Anfrage reagiert, wird dieser verwendet und der Aufruf beendet. Wenn nicht, wird weiter nach diesem Muster die Kette abgearbeitet. Reagiert kein Listener auf die Anfrage, gilt das Verarbeiten eines Nachrichten-Inhalts als fehlgeschlagen.

Im Folgenden nun die verschiedenen Listener-Schnittstellen:

UnicastMulticastRequestListener

Abbildung 4.8: Aufbau des UnicastMulticastRequestListener-Interface

- *handleUnicastMulticastRequest(String senderUrn, byte[] payload)* verarbeitet den Inhalt von Unicast und Multicast Requests. Es wird *true* zurückgeliefert, wenn der Inhalt erfolgreich gelesen wurde, sonst *false*.

SingleRequestSingleResponseRequestListener

Abbildung 4.9: Aufbau des SingleRequestSingleResponseRequestListener-Interface

- *handleSingleRequestSingleResponseRequest(String senderUrn, byte[] requestPayload)* verarbeitet den Inhalt von SingleRequestSingleResponse-Requests.

Es wird ein Response-Inhalt als *byte[]* zurückgeliefert, wenn der Request-Inhalt erfolgreich verarbeitet wurde, sonst *null*. Falls bei Verarbeitung eine *Exception* auftritt, wird ein *MEPSingleResponseExceptionEvent* geworfen.

MultiResponseRequestListener

Abbildung 4.10: Aufbau des MultiResponseRequestListener-Interface

- *handleMultiResponseRequest(MultiResponseHandle responseHandle, String senderUrn, byte[] requestPayload)* verarbeitet den Inhalt von *einem* Multi-Response-Request. Es wird *true* zurückgeliefert, wenn der Inhalt erfolgreich verarbeitet wurde, sonst *false*. Falls bei der Verarbeitung eine *Exception* auftritt, wird ein *MEPMultiResponseExceptionEvent* geworfen.

Für das Erzeugen der einzelnen Antworten eines MultiResponseRequests, muß jeweils die folgende Methode des MultiResponseHandle-Interface aufgerufen werden:

Abbildung 4.11: Aufbau des MultiResponseHandle-Interface

MultiResponseHandle

- *handleSingleResponse(byte[] payload, int current, int total)* erzeugt *eine* einzelne MultiResponse mit dem entsprechenden Inhalt und sendet diese zurück. Der Parameter *current* entspricht dabei der Zahl der aktuellen Nachricht und *total* der Gesamt-Anzahl der zurückzusendenden Nachrichten, wobei zu beachten ist, dass der Wert *total* für eine MultiResponse immer gleich bleiben sollte.

Für die Listener *SingleRequestSingleResponseRequestListener* und *MultiResponseRequestListener* gilt dabei: ist an einem Peer kein entsprechender RequestListener registriert, bzw. fühlt sich für den Inhalt einer Nachricht kein RequestListener verantworlich, so wird von der Applikation default-mäßig eine *RequestListenerNotFoundException* geworfen. Diese wird einem *MEPExceptionEvent* übergeben, welches dann zu einer ErrorResponse zusammengesetzt wird, die dann an den Sender der Request-Response-Nachricht übertragen wird.

Des weiteren gibt es client-seitig einen zusätzlichen Listener für den Eingang von MultiResponse-Responses:

ProgressListenerRunnable

Abbildung 4.12: Aufbau des ProgressListenerRunnable-Interface

- *progress(String senderUrn, byte[] payload, int current, int total)* wird bei Eingang einer einzelnen MultiResponse aufgerufen.
- *run()* wird bei Eingang aller einzelnen MultiResponses aufgerufen.

Ein ProgressListenerRunnable kann dann über das ResponseFuture eines ReliableRequests über die Methode *addListener(ProgressListenerRunnable listener, Executer executer)* hinzugefügt werden.

4.3.3 Implementierung und Abhängigkeiten

PeerImpl

Im Folgenden werden nun die Abhängigkeiten des oben beschriebenen Peer-Interfaces, die **PeerImpl**, genauer beleuchtet. Die entsprechende Struktur ist dabei in Abbildung 4.13 zu erkennen. Anmerkung: Um das Schaubild nicht unnötig zu verkomplizieren sind dabei alle Methoden weggelassen. Die weggelassenen Methoden wurden bereits oben in der entsprechenden Schnittstellenbeschreibung erklärt.

Die PeerImpl besitzt eine Aggregationsabhängigkeit zum UberlayModule aus dem Uberlay-Projekt. Des weiteren ist sie die Oberklasse des AbstractPeer welche die Adressierung eines Peer steuert. Der AbstractPeer wiederum, ist die Oberklasse des Interface Peer, welches wiederum die Interfaces Service, ServiceHandler, UbermepService, UbermepUnreliableService sowie UbermepReliableService implementiert. Die Implementierungen sämtlicher Service-Interfaces sind dabei in der PeerImpl zu finden.

In den Folgenden Erläuterungen der Methoden wird von einer *Transport-Schicht* für ubermep gesprochen. Diese Transport-Schicht entspricht dem Uberlay-Projekt. Das Zusammenspiel von ubermep und uberlay ist im Entwurfskapitel 3 in Abschnitt 3.2 beschrieben.

Bevor hier nun die ServiceHandler-Interfaces beschrieben werden, soll nun kurz auf den AbstractServiceHandler eingegangen werden. Der AbstractServiceHand-

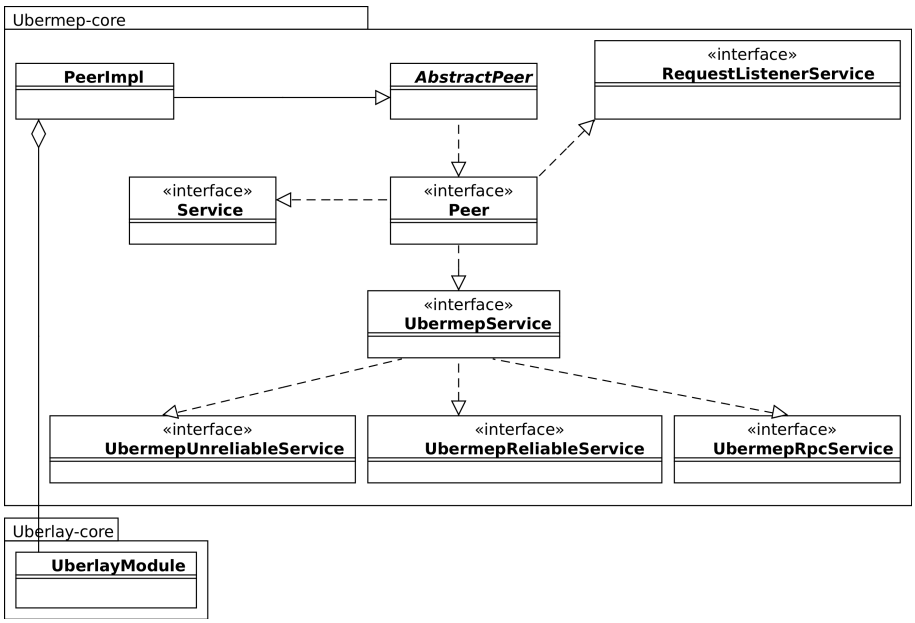


Abbildung 4.13: Vererbungshierarchie und Abhängigkeiten der PeerImpl

ler bildet die Oberklasse der nun folgenden ServiceHandler, welche sich für den Austausch von Nachrichten des Message Exchange Pattern verantwortlich zeichnen. Der AbstractServiceHandler implementiert dabei das bereits beschriebene RequestListenerService-Interface sowie die SimpleChannelHandler-Klasse aus dem Netty-Framework. Die Abhängigkeiten sind dabei in der Abbildung 4.14 veranschaulicht. Hierbei wird aber nur auf die wichtigen Methoden des AbstractServiceHandler eingegangen.

- *messageReceived(ChannelHandlerContext ctx, ChannelEvent e)* wird aufgerufen, wenn eine Protokoll-Nachricht im entsprechenden Handler empfangen wurde.
- *handleUpstream(ChannelHandlerContext ctx, ChannelEvent e)* behandelt ein event, das upstream empfangen wurde.
- *handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)* behandelt ein event, das downstream empfangen wurde.

UnreliableServiceHandler

Des weiteren besitzt die bereits oben beschriebene PeerImpl eine Aggregationsabhängigkeit zum UnreliableServiceHandler, welcher wiederum eine Oberklasse des AbstractService ist. Im UnreliableServiceHandler sind folgende Methoden von Bedeutung:

- *send(UnreliableRequest request, Channel channel)* sendet ein UnreliableRequest über den im Parameter angegeben *channel*. Sie gibt keinen Rück-

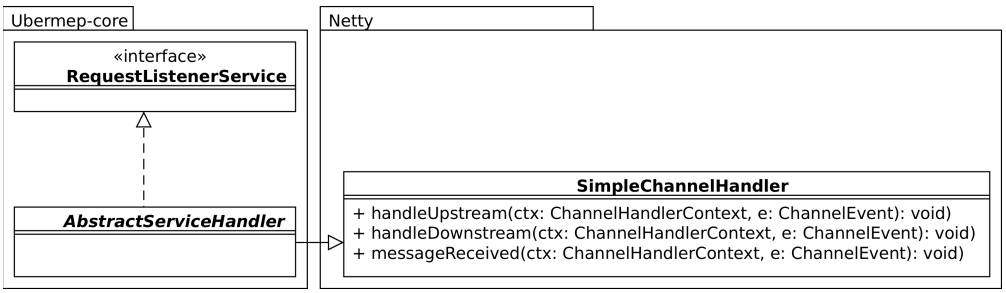


Abbildung 4.14: Abhängigkeiten des AbstractServiceHandler

gabewert zurück.

- *messageReceived(ChannelHandlerContext ctx, MessageEvent e)* empfängt einen UnreliableRequest und ruft für die registrierten *UnicastMulticastRequestListener* die Methode *handleUnicastMulticastRequest(String senderUrn, byte[] payload)* auf. Auch hier wird kein Rückgabewert zurückgegeben.

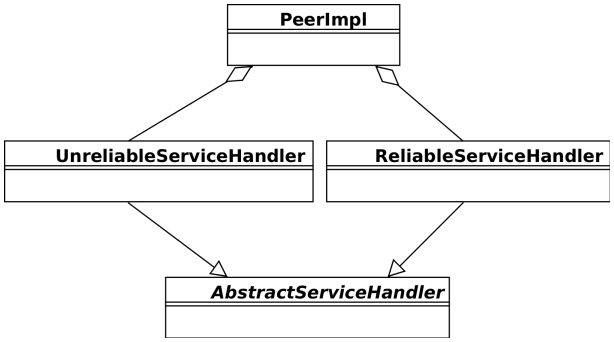


Abbildung 4.15: Abhängigkeiten des Unreliable- und ReliableServiceHandler

ReliableServiceHandler

Wie bereits in Kapitel 3.1 beschrieben, enthält das *R*eliable Messaging von Uberlay Nachrichten des One-way Pattern und des Request-Response Pattern. Dementsprechend verarbeitet der ReliableServiceHandler Nachrichten des One-way Pattern, namentlich Reliable Unicast und Reliable Multicast, sowie Nachrichten des Request-Response Pattern, namentlich Single Request Single Response, Single Request Multi Response und Multi Request Multi Response. Im Folgenden werden nun der ReliableServiceHandler, SingleRequestSingleResponseServiceHandler, SingleRequestMultiResponseServiceHandler sowie MultiRequestMultiResponseServiceHandler näher beschrieben. Die Abhängigkeiten für Nachrichten des One-way Pattern sind dabei in Abbildung 4.15 über das Interface ReliableServiceHandler veranschaulicht. Die Abhängigkeiten für Nachrichten des Request-Response Pattern sind in Abbildung 4.16 veranschaulicht.

Im ReliableServiceHandler sind dabei folgende Methoden von Bedeutung:

- *send(ReliableRequest request, Channel channel)* sendet ein ReliableRequest über den im Parameter angegeben *channel*. Gibt eine Response als Future-Objekt zurück. Eine genau Beschreibung der zurückgelieferten Response findet sich weiter unten im Abschnitt 4.3.4. Falls der Request eine Request-Response-Pattern-Nachricht ist, wird diese an den korrespondierenden Channel weitergeleitet. Eine ausführliche Beschreibung hierfür findet sich weiter unten im Unterabschnitt *RequestResponseChannel*.
- *messageReceived(ChannelHandlerContext ctx, MessageEvent e)* empfängt ReliableUnicast- bzw. ReliableMulticast-Protokoll-Nachrichten und ruft für die registrierten *UnicastMulticastRequestListener* die Methode *handleUnicastMulticastRequest(String senderUrn, byte[] payload)* auf.
- *handleUpstream(ChannelHandlerContext ctx, ChannelEvent e)* empfängt einen ReliableRequest upstream. Für den Fall dass es sich um einen SingleRequestSingleResponse-Request handelt, wird die Nachricht upstream an den *SingleRequestSingleResponseServiceHandler* weitergereicht. Falls es sich um einen MultiResponse-Request handelt, wird die Nachricht upstream an den *SingleRequestMultiResponseServiceHandler* bzw. *MultiRequestMultiResponseServiceHandler* weitergereicht.
- *handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)* empfängt einen ReliableRequest downstream. Für den Fall dass es sich um einen SingleRequestSingleResponse-Request handelt, wird die Nachricht downstream an den SingleRequestSingleResponseServiceHandler weitergereicht. Falls es sich um einen MultiResponse-Request handelt, wird die Nachricht downstream an den *SingleRequestMultiResponseServiceHandler* bzw. *MultiRequestMultiResponseServiceHandler* weitergereicht. Sonst wird die oben beschriebene *send(request, channel)* aufgerufen.

SingleRequestSingleResponseServiceHandler Im SingleRequestSingleResponseServiceHandler werden nur SingleRequestSingleResponse-Nachrichten verarbeitet. Dabei sind die folgenden Methoden von Bedeutung:

- *messageReceived(ChannelHandlerContext ctx, MessageEvent e)* empfängt SingleRequestSingleResponse-Protokoll-Nachrichten. Für die registrierten *SingleRequestSingleResponseRequestListener* wird die Methode *handleSingleRequestSingleResponseRequest(String senderUrn, byte[] requestPayload)* aufgerufen und eine Response zurückgesendet.
- *handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)* empfängt einen SingleRequestSingleResponse-Request downstream im SingleRequestSingleResponseChannel, wandelt diese in eine Protokoll-Nachricht um und sendet diese über die Transport-Schicht. Der genaue Ablauf hierbei ist weiter unten im Unterabschnitt des *SingleRequestSingleResponseChannel* zu finden.

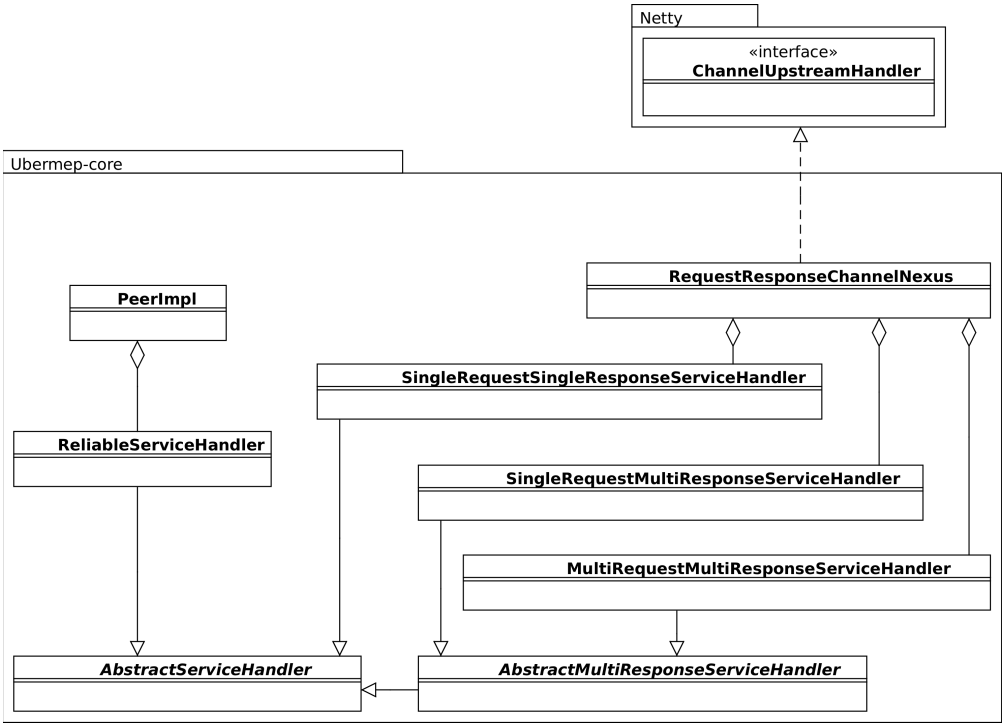


Abbildung 4.16: Abhängigkeiten der RequestResponse-ServiceHandler

SingleRequestMultiResponseServiceHandler Im SingleRequestMultiResponseServiceHandler werden nur SingleRequestMultiResponse-Nachrichten verarbeitet. Dabei sind die folgenden Methoden von Bedeutung:

- *messageReceived(ChannelHandlerContext ctx, MessageEvent e)* empfängt SingleRequestMultiResponse-Protokoll-Nachrichten. Für die registrierten *MultiResponseRequestListener* wird die Methode *handleMultiResponseRequest(MultiResponseHandle responseHandle, String senderUrn, byte[] requestPayload)* aufgerufen. Über das MultiResponseHandle werden dann einzeln Responses zurückgesendet.
- *handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)* empfängt einen SingleRequestMultiResponse-Request downstream im MultiResponseChannel, wandelt diese in eine Protokoll-Nachricht um und sendet diese über die Transport-Schicht. Der genaue Ablauf hierbei ist weiter unten im Unterabschnitt des *SingleRequestSingleResponseChannel* zu finden.

MultiRequestMultiResponseServiceHandler Im MultiRequestMultiResponseServiceHandler werden nur MultiRequestMultiResponse-Nachrichten verarbeitet. Dabei ist einzig die folgenden Methode von Bedeutung:

- *handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)* empfängt einzig MultiRequestMultiResponse-Requests. Diese werden intern in einzelne SingleRequestMultiResponse-Requests umgewandelt und downstream

in der MultiResponse-Pipeline an den SingleRequestMultiResponseServiceHandler weitergereicht.

RpcServiceHandler

Ubermep unterstützt den Aufruf von Remote Procedure Calls (RPC) (siehe Abschnitt 2.2 im Grundlagenkapitel) auf einzelnen Peers des Peer-to-Peer-Netzwerks. Dazu muss an einem Peer eine UbermepRpcChannel generiert werden, über welchen dann der RPC aufgerufen werden kann. Die nähere Beschreibung des UbermepRpcChannel ist weiter unten im Unterabschnitt *UbermepRpcChannel* zu finden. Im Folgenden wird nun der *RpcServiceHandler*, welcher sich für die Abarbeitung der Remote Procedure Calls verantwortlich zeichnet, näher beschrieben.

Im RpcServiceHandler werden nur *Remote Procedure Call* -Nachrichten verarbeitet. Dabei sind die folgenden Methoden von Bedeutung:

- *messageReceived(ChannelHandlerContext ctx, MessageEvent e)* empfängt RPC-Protokoll-Nachrichten, ruft die entsprechende Methode auf und sendet das Ergebnis als Response-Protokoll-Nachricht zurück.
- *callRPCAndReturnResponse(Channel channel, UPAddress destUrn, ...)* wird vom *UbermepRpcChannel* aufgerufen, um einen RPC in eine RPC-Protokoll-Nachricht umzubauen und diese über die Transport-Schicht zu versenden. Gibt die RPC-Response-Protokoll-Nachricht in dem im Parameter übergeben Callback zurück.

Überblick

Bevor nun die verschiedenen Channel erläutert werden, soll ein kurzer Überblick über das Zusammenspiel der oben beschriebenen verschiedenen ServiceHandler gegeben werden. Dies soll anhand eines Sequenzdiagramm für den Aufruf einer Single Request Single Response Nachricht veranschaulicht werden. Dabei ist in der Abbildung 4.17 der Ablauf für einen Empfang Server-seitig vereinfacht dargestellt, wobei in der Abbildung dafür das Wort *SingleRequestSingleResponse* als SRSR abgekürzt ist.

Der Ablauf ist dabei wie folgt: Der Server-Socket liest bytes und reicht diese an den entsprechenden (Protobuf)-Decoder weiter. Dieser dekodiert die Nachricht und erzeugt daraus eine Protokoll-Nachricht. Diese wird als ChannelEvent upstream über die registrierten ChannelHandler an den SingleRequestSingleResponseServiceHandler weitergereicht. Der verantwortliche registrierte SingleRequestSingleResponseRequestListener bekommt den Payload übergeben, arbeitet den Request-Payload ab und gibt einen Response-Payload an den SingleRequestSingleResponseRequestListener zurück, der daraus eine korrespondierende Response-Protokoll-Nachricht erzeugt. Diese Nachricht wird dann über die Handler der Pipeline downstream zum (Protobuf-)Dekoder durchgereicht. Der

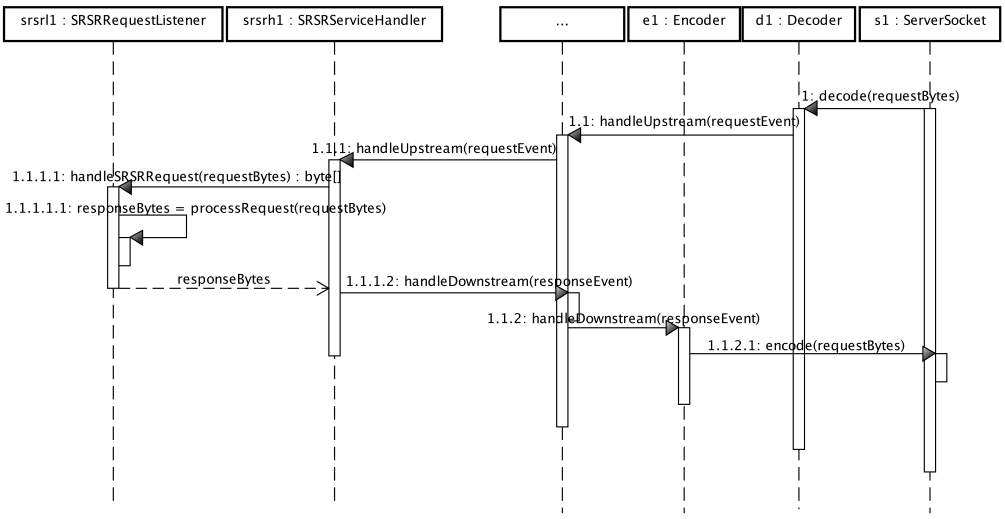


Abbildung 4.17: sequenz

Dekoder serialisiert die bytes und sendet diese an den Client-Socket zurück.

Der Ablauf für die weiteren Nachrichtentypen des Message Exchange Pattern ist dabei analog, wobei sich jeweils nur der ServiceHandler und RequestListener verändert.

RequestResponseChannel

Wie bereits in der ChannelArchitektur im Abschnitt 3.2.1 des Entwurfskapitel beschrieben, besitzt Ubermep neben dem UnicastMulticastChannel eigene RequestResponseChannel, namentlich *SingleRequestSingleResponseChannel* bzw. *MultiResponseChannel*. Im Folgenden werden nun diese beiden Channels näher beschrieben. Das Zusammenspiel der verschiedenen Channels ist dabei in Abbildung 4.18 veranschaulicht.

SingleRequestSingleResponseChannel Über den SingleRequestSingleResponseChannel können ausschließlich SingleRequestSingleResponseRequests versendet werden. Dabei ist einzig die folgende Methode von Bedeutung:

- *write(SingleRequestSingleResponseRequest request)* sendet einen SingleRequestSingleResponseRequest über den SingleRequestSingleResponseChannel. Der Response wird als Future-Objekt zurückgegeben. Eine genaue Beschreibung der zurückgelieferten Response findet sich weiter unten im Abschnitt 4.3.4.

MultiResponseChannel Über den MultiResponseChannel können ausschließlich MultiResponseRequests, namentlich SingleRequestMultiResponseRequest

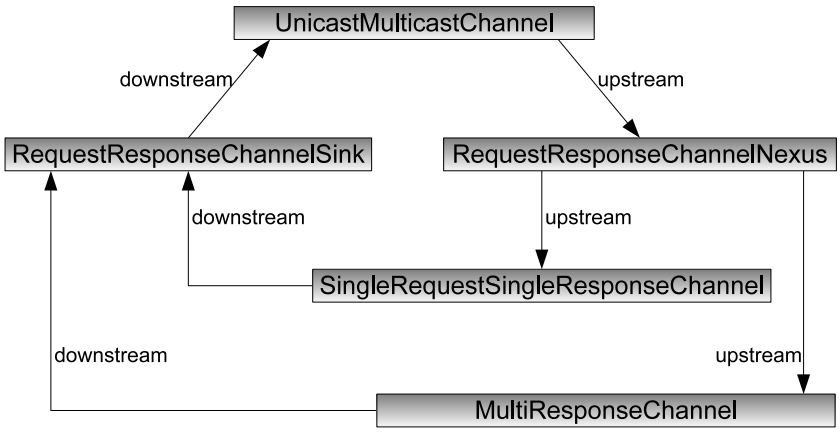


Abbildung 4.18: Handling der RequestResponse-Events in Ubermep

bzw. MultiRequestMultiResponseRequest versendet werden. Dabei ist einzig die folgenden Methode von Bedeutung:

- *write(Request request)* sendet einen MultiResponseRequest über den MultiResponseChannel. Der Response wird als Future-Objekt zurückgegeben. Eine genaue Beschreibung der zurückgelieferten Response findet sich weiter unten im Abschnitt 4.3.4.

UbermepRpcChannel

Ubermep besitzt einen eigenen Channel für den Aufruf von Remote Procedure Calls, den UbermepRpcChannel. Mittels diesem Channel können Stubs generiert werden, über die ein RPC aufgerufen werden kann. Die, für den Aufruf entscheidenen Methoden, bezieht der UbermepRpcChannel aus dem Protobuf-Projekt. Die Abbildung 4.19 veranschaulicht hierbei die Abhängigkeiten. Die Methoden haben dabei die folgende Bedeutung:

- *callBlockingMethod(Descriptors.MethodDescriptor method, ...)* ruft einen blockierenden Remote Procedure Call auf und gibt die Antwort als Protokoll-Nachricht zurück. Blockierend bedeutet, das diese Methode solange den Aufrufer *blockiert* bis eine Antwort empfangen wurde.
- *callMethod(Descriptors.MethodDescriptor method, ...)* ruft einen nicht-blockierende Remote Procedure Call auf und gibt die Antwort als Protokoll-Nachricht in dem als Parameter übergebenen Callback zurück, wobei der Aufrufer hierbei *nicht blockiert* wird.

4.3.4 Messaging

Im Folgenden Abschnitt werden nun die von Ubermep unterstützten Nachrichten-Pattern spezifiziert. Zuerst werden die Nachrichten des UnreliableMessaging-

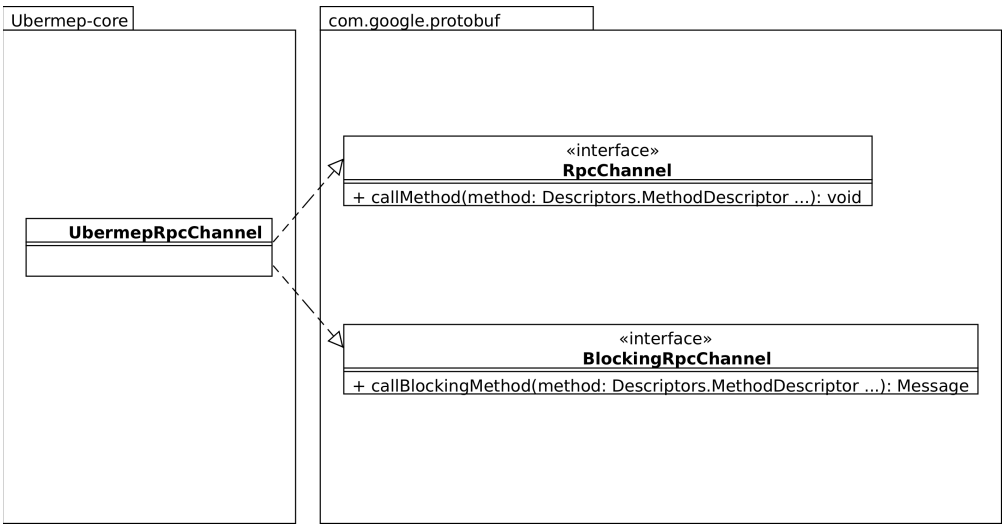


Abbildung 4.19: Abhängigkeiten des UbermepRpcChannel

Pattern beschrieben danach die Nachrichten des ReliableMessaging.

In der nun folgenden Spezifikation haben dabei die verwendeten Eigenschaften die folgende Bedeutung:

- **Parameter**: die der Nachricht zu übergebenen Parameter
- **PayloadListener**: der verantwortliche PayloadListener Server-seitig zum Lesen des Request-Inhalts und Schreiben des Response-Inhalts.
- **Response**: die zu erwartende Response bei erfolgreicher Auslieferung der Nachricht.
- **Optionale Parameter**: die der Nachricht optional zu übergebenen Parameter, wobei diese immer über entsprechende *Setter* gesetzt werden.
- **Erfolgreich ausgeliefert bei**: gibt an, wann eine Nachricht als *erfolgreich ausgeliefert* gilt. Nur dann wird auch die entsprechende Response zurückgegeben.
- **ProgressListener**: das zu implementierende Interface Client-seitig.
- **Zurückgeliefert bei**: gibt an, wann eine Nachricht an den Sender zurückgeliefert wird.

UnreliableRequest

In diesem Abschnitt werden nun die Nachrichten des UnreliableMessaging-Pattern beschrieben.

UnreliableUnicastRequest Ein UnreliableUnicastRequest ist wie folgt spezifiziert:

Parameter	remoteAddress : UPAddress payload : byte[]
PayloadListener	UnicastMulticastRequestListener (lesend)
Response	keine

UnreliableMulticastRequest Ein UnreliableMulticastRequest ist wie folgt spezifiziert:

Parameter	remoteAddresses : Collection<UPAddress> payload : byte[]
PayloadListener	UnicastMulticastRequestListener (lesend)
Response	keine

ReliableRequest

In diesem Abschnitt werden die Nachrichten des ReliableMessaging-Pattern spezifiziert.

ReliableUnicastRequest Ein ReliableUnicastRequest ist wie folgt spezifiziert:

Parameter	remoteAddress : UPAddress payload : byte[]
optionale Parameter	timeOut : int timeOutUnit : TimeUnit
PayloadListener	UnicastMulticastRequestListener (lesend)
Response	ReliableUnicastResponse
Erfolgreich ausgeliefert bei	Eingang der Nachricht am Empfänger-Peer

ReliableMulticastRequest Ein ReliableMulticastRequest ist wie folgt spezifiziert:

Parameter	remoteAddresses : Collection<UPAddress> payload : byte[]
optionale Parameter	timeOut : int timeOutUnit : TimeUnit
PayloadListener	UnicastMulticastRequestListener (lesend)
Response	ReliableMulticastResponse
Erfolgreich ausgeliefert bei	Eingang der Nachricht am Empfänger-Peer

Anmerkung: Eine ReliableMulticastResponse enthält die Responses als Map. Die Responses werden dabei in der Map-üblichen Weise, also als *Key,Value*-Paar der Response-Map hinzugefügt. Der *Key* entspricht dabei der UPAddress des geantworteten Peers, der *Value* einer einzelnen Response vom Typ *ReliableMulticastResponse.SingleReliableMulticastResponse*.

SingleRequestSingleResponseRequest Ein SingleRequestSingleResponseRequest ist wie folgt spezifiziert:

Parameter	remoteAddress : UPAddress payload : byte[]
optionale Parameter	timeOut : int timeOutUnit : TimeUnit
PayloadListener	SingleRequestSingleResponseRequestListener (lesend und schreibend)
Response	SingleRequestSingleResponseResponse
Erfolgreich ausgeliefert bei	Eingang der Nachricht am Empfänger-Peer **und** erfolgreicher Verarbeitung eines PayloadListener

SingleRequestMultiResponseRequest Ein SingleRequestMultiResponseRequest ist wie folgt spezifiziert:

Parameter	remoteAddress : UPAddress payload : byte[]
optionale Parameter	timeOut : int timeOutUnit : TimeUnit
PayloadListener	MultiResponseRequestListener (lesend und schreibend)
ProgressListener	ProgressListenerRunnable
Response	SingleRequestMultiResponseResponse
Erfolgreich ausgeliefert bei	Eingang der Nachricht am Empfänger-Peer **und** erfolgreicher Verarbeitung eines PayloadListener

MultiRequestMultiResponseRequest Ein MultiRequestMultiResponseRequest ist wie folgt spezifiziert:

Parameter	remoteAddresses : Collection<UPAddress> payload : byte[]
optionale Parameter	timeOut : int timeOutUnit : TimeUnit
PayloadListener	MultiResponseRequestListener (lesend und schreibend)
ProgressListener	ProgressListenerRunnable
Response	MultiRequestMultiResponseResponse
Erfolgreich ausgeliefert bei	Eingang der Nachricht am Empfänger-Peer **und** erfolgreicher Verarbeitung eines PayloadListener

Anmerkung: Eine MultiResponse-Response enthält die Responses als Map. Die Responses werden dabei in der Map-üblichen Weise, also als *Key,Value*-Paar der Response-Map hinzugefügt. Der *Key* entspricht dabei der UPAddress des geantworteten Peers, der *Value* einer einzelnen Response vom Typ *SingleMultiResponseResponse*.

ReliableResponse

Einzig Nachrichten des oben beschriebenen ReliableMessaging-Pattern liefern Responses zurück. Dabei gibt es, wie oben spezifiziert, für einen erfolgreich ausgelieferten ReliableRequest einen korrespondierenden ReliableResponse. Des weiteren gibt es für diesen Request ReliableResponses bei *nicht erfolgreicher* Übertragung. Im Folgenden werden nun diese beschrieben.

ErrorResponse Eine ErrorResponse ist wie folgt spezifiziert:

Parameter	request : Request localAddress : UPAddress cause : Throwable
Zurückgeliefert bei	Fehlerhafter Übertragung einer Nachricht

TimeOutResponse Eine TimeOutResponse ist wie folgt spezifiziert:

Parameter	request : Request localAddress : UPAddress
Zurückgeliefert bei	TimeOut während der Auslieferung, ausgelöst durch: TimeOut des Request wenn gesetzt, sonst DefaultTimeOut (siehe dazu Abschnitt 4.3.5)

4.3.5 Konfiguration

Die Konfiguration von Ubermep erfolgt über das konfigurieren der einzelnen Peers des Netzwerks. Dies geschieht über Variablen der Klasse *PeerConfig*. Die nun folgenden Spezifikationen sind dabei wie folgt aufgebaut:

{**Variablenname**}

- **{Verwendung}**
- **{Typ}**
- **{DefaultValue}**

Konfiguration eines Peers

Die Konfiguration eines Peers ist wie folgt spezifiziert:

CORE_POOL_SIZE

- **Verwendung**: Die maximale Größe der von Uberlay verwendeten Thread-Pools
- **Typ**: int

- **DefaultValue**: 10

DEFAULT_TIMEOUT

- **Verwendung**: Die DefaultTimeOut für ReliableRequests.
- **Typ**: int
- **DefaultValue**: 30

DEFAULT_TIMEOUT_TIMEUNIT

- **Verwendung**: Die Default-Zeiteinheit der DEFAULT_TIMEOUT
- **Typ**: TimeUnit
- **DefaultValue**: TimeUnit.SECONDS

UberlayModule.RTT_REQUEST_INTERVAL

- **Verwendung**: Das Interval in dem die RoutingTabellen eines Peers mittels des RoundTripTime-Protokolls von Uberlay aktualisiert werden. (siehe dazu Abschnitt 2.5.3 im Grundlagenkapitel)
- **Typ**: int
- **DefaultValue**: 10

UberlayModule.RTT_REQUEST_INTERVAL_TIMEUNIT

- **Verwendung**: Die Default-Zeiteinheit des RTT_REQUEST_INTERVAL
- **Typ**: int
- **DefaultValue**: TimeUnit.SECONDS

4.4 Beispiele

Im Folgenden soll nun anhand von Beispielen die Funktionalität von Ubermep veranschaulicht werden. Dazu wird ein kleines Peer-to-Peer Netzwerk mit 3 Knoten erzeugt, über die Nachrichten des Message Exchange Pattern versendet werden können. Die Abbildung 4.20 zeigt dabei den Aufbau des Netzwerks.

4.4.1 Aufbau eines Peer-to-Peer Netzwerk

Im nun folgenden Beispiel wird ein Peer-to-Peer Netzwerk wie folgt aufgebaut:

Der entsprechende Quellcode sieht dabei wie folgt aus:

```
1 // Peers erzeugen ...
2 Peer server = new PeerImpl(new UPAddress("urn:itm:1"),
```

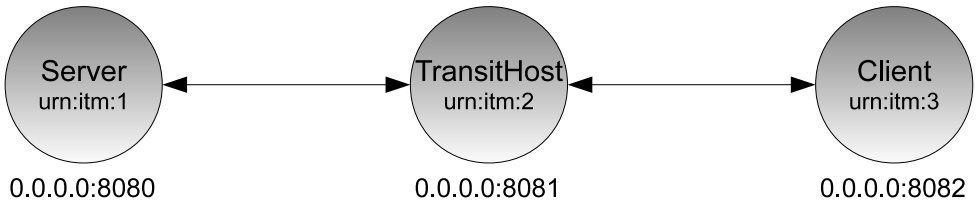


Abbildung 4.20: Aufbau eines einfachen Peer-to-Peer Netzwerk

```
3      new InetSocketAddress("0.0.0.0", 8080));
4  Peer transitHost = new PeerImpl(new UPAddress("urn:itm:2"),
5      new InetSocketAddress("0.0.0.0", 8081),
6      new InetSocketAddress("0.0.0.0", 8080));
7  Peer client = new PeerImpl(new UPAddress("urn:itm:3"),
8      new InetSocketAddress("0.0.0.0", 8082),
9      new InetSocketAddress("0.0.0.0", 8081));
10
11 // Peers starten ...
12 server.start();
13 transitHost.start();
14 client.start();
15
16 // ... kommuniziere ein wenig ...
17
18 // Peers stoppen
19 client.stop();
20 transitHost.stop();
21 server.stop();
```

Listing 4.6: Aufbau eines einfachen Peer-to-Peer Netzwerk

4.4.2 Kommunikation

Nachdem nun das Netzwerk aufgebaut ist, kann über die Knoten kommuniziert werden. In den nun folgenden Beispielen sollen Nachrichten erzeugt und versendet werden, um ausgewählte Peers (Nodes) zu *flashen*. *Flashen* bedeutet hier: Das Aufspielen einer neuen Firmware auf einem Knoten. Diese Funktionalität spielt z.B. eine Rolle in *Wireless Sensor Networks (WSN's)*. Dafür werden in den folgenden Beispielen der String `flashNode` als ByteArray übertragen und ausgewertet.

Unreliable Messaging

Um Unicast- bzw. Multicast-Requests für das flashen von Nodes verarbeiten zu können, muss zuerst der entsprechende UnicastMulticastRequestListener wie folgt erzeugt werden, dabei sei angenommen das eine entsprechende Methode *flashNode()* mit der oben beschriebenen Funktionalität vorhanden ist:

```
1  UnicastMulticastRequestListener
       flashNodeUnicastMulticastRequestListener = new
```

```
    UnicastMulticastRequestListener() {
  public boolean handleUnicastMulticastRequest(String senderUrn,
      byte[] payload) {
    if (new String(payload).equals("flashNode")) {
      // aufrufen von flashNode()
      flashNode();
      return true;
    } else {
      return false;
    }
  }
};
```

Listing 4.7: Erzeugen eines UnicastMulticastRequestListener

Nun muss noch der *flashNodeUnicastMulticastRequestListener* am entsprechenden Peer Server-seitig hinzugefügt werden.

```
server.addRequestListener(
    flashNodeUnicastMulticastRequestListener);
```

Listing 4.8: Hinzufügen eines UnicastMulticastRequestListener

UnreliableUnicastRequest Anschließend kann ein entsprechender UnreliableUnicastRequest an den Server wie folgt versendet werden:

```
UnreliableRequest request = new UnreliableUnicastRequest(server.
    getLocalUPAddress(), "flashNode".getBytes());
client.send(request);
```

Listing 4.9: Senden eines UnreliableUnicastRequest

UnreliableMulticastRequest Der Aufruf eines UnreliableMulticastRequest erfolgt analog zu dem eines UnreliableUnicastRequest.

```
List<UPAddress> urns = new ArrayList<UPAddress>() {{
  add(server.getLocalUPAddress());
  add(...);
}};

UnreliableRequest request = new UnreliableMulticastRequest(urns,
     "flashNode".getBytes());
client.send(request);
```

Listing 4.10: Senden eines UnreliableMulticastRequest

Reliable Messaging

Nachrichten des Reliable Messaging Patterns können als blockierende als auch als nicht-blockierende Aufrufe versendet werden. Im Folgenden Beispiel werden nun die beiden Aufrufe dargestellt:

```
final ListenableFuture<Response> responseFuture = client.send(
    request);

```

```
3  //blockierender Aufruf
4  Response response = responseFuture.get();
5
6  //nicht-blockierender Aufruf
7  responseFuture.addListener(new Runnable() {
8    public void run() {
9      Response response = responseFuture.get();
10   }}, new ScheduledThreadPoolExecutor(1));
11 ...
```

Listing 4.11: Blockierender und nicht-blockierender Nachrichtenempfang

In den folgenden Beispielen werden zur Vereinfachung nur blockierende Aufrufe beschrieben.

ReliableUnicastRequest Mit Hilfe eines ReliableUnicastRequest kann nun überprüft werden, ob die Methode *flashNode()* tatsächlich aufgerufen wurde.

```
1  ReliableRequest request = new ReliableUnicastRequest(server.
       getLocalUPAddress(), "flashNode".getBytes());
2  Future<Response> responseFuture = client.send(request);
3
4  Response response = responseFuture.get();
5  if (response instanceof ReliableUnicastResponse){
6    // tu was mit Response
7  }
```

Listing 4.12: Senden eines ReliableUnicastRequest als blockierender Aufruf

ReliableMulticastRequest Der Aufruf eines ReliableMulticastRequest geschieht analog zu dem eines ReliableUnicastRequest.

```
1  List<UPAddress> urns = new ArrayList<UPAddress>() {{
2    add(transitHost.getLocalUPAddress());
3    add(...);
4  }};
5
6  ReliableRequest request = new ReliableMulticastRequest(urns, "
       flashNode".getBytes());
7  Future<Response> responseFuture = client.send(request);
8
9  Response response = responseFuture.get();
10 // "uberprüfe Responses ... über: response.getResponses()
```

Listing 4.13: Senden eines ReliableMulticastRequest

SingleRequestSingleResponseRequest Um einen SingleRequestSingleResponse-Request für das flashen eines Nodes verarbeiten zu können, muss zuerst der entsprechende SingleRequestSingleResponseRequestListener Server-seitig wie folgt erzeugt und hinzugefügt werden, dabei sei auch hier angenommen, dass eine entsprechende Methode *flashNode()* mit der beschriebenen Funktionalität vorhanden ist:

```
SingleRequestSingleResponseRequestListener
    flashNodeSingleRequestSingleResponseRequestListener = new
    SingleRequestSingleResponseRequestListener() {
  public byte[] handleSingleRequestSingleResponseRequest(String
      senderUrn, byte[] payload) throws UbermepExceptionEvent {
    if (new String(payload).equals("flashNode")) {
      try {
        // aufrufen von flashNode()
        flashNode();
        return "flashNode: Done".getBytes();
      } catch (Exception e){
        throw
          new UbermepSingleResponseExceptionEvent(e.getCause());
      }
    } else {
      return null;
    }
  }
};

server.addRequestListener(
    flashNodeSingleRequestSingleResponseRequestListener);
```

Listing 4.14: Erstellen eines SingleRequestSingleResponseRequestListener

Anschließend kann nun mit Hilfe eines SingleRequestSingleResponseRequest überprüft werden, ob die Methode *flashNode()* tatsächlich aufgerufen wurde:

```
ReliableRequest request = new SingleRequestSingleResponseRequest
    (server.getLocalUPAddress(), "flashNode".getBytes());
Future<Response> responseFuture = client.send(request);

Response response = responseFuture.get();
if (response instanceof SingleRequestSingleResponseResponse){
  // tu was mit Response, z.B. payload überprüfen
}
```

Listing 4.15: Senden eines SingleRequestSingleResponseRequest

SingleRequestMultiResponseRequest Um einen MultiResponseRequest für das Flashen eines Nodes verarbeiten zu können, muss zuerst der entsprechende MultiResponseRequestListener Server-seitig wie folgt erzeugt und hinzugefügt werden, dabei sei auch hier angenommen, dass eine Methode *flashNode()* mit der bereits beschriebenen Funktionalität vorhanden ist, wobei die Methode hier den Fortschritt in Prozent als Integer-Wert zurückgibt:

```
MultiResponseRequestListener
    flashNodeMultiResponseRequestListener = new
    MultiResponseRequestListener() {
  public boolean handleMultiResponseRequest(MultiResponseHandle
      responseHandle, String senderUrn, byte[] payload) throws
      UbermepExceptionEvent {
    if (new String(payload).equals("flashNode")) {
      // aufrufen von flashNode()
      int percentDone = flashNode();
      while(percentDone < 50){;}
      // wenn flashNode() zu 50% fertig
```

```
8         responseHandle.handleSingleResponse("flashNode:50%␣done".
              getBytes(), 1, 2);
9         while(percentDone < 100){;}
10        // wenn flashNode() fertig
11        responseHandle.handleSingleResponse("flashNode:100%␣done".
              getBytes(), 2, 2);
12        return true;
13      } else {
14        return false;
15      }
16    }
17  };
18
19  server.addRequestListener(flashNodeMultiResponseRequestListener)
        ;
```

Listing 4.16: Hinzufügen eines MultiResponseRequestListener

Anschließend kann mit Hilfe eines SingleRequestMultiResponseRequest überprüft werden, ob die Methode *flashNode()* tatsächlich aufgerufen wurde und wie der Fortschritt ist.

```
1 ReliableRequest request = new SingleRequestMultiResponseRequest(
      server.getLocalUPAddress(), "flashNode".getBytes());
2 Future<Response> responseFuture = client.send(request);
3
4 Response response = responseFuture.get();
5 if (response instanceof SingleRequestMultiResponseResponse){
6   // überprüfe Responses ... über: response.getResponses()
7 }
```

Listing 4.17: Senden eines SingleRequestMultiResponseRequest

Das Hinzufügen eines ProgressListeners für eine SingleRequestMultiResponse-Request-Nachrichten ist weiter unten erklärt.

MultiRequestMultiResponseRequest Der Aufruf eines MultiRequestMulti-ResponseRequest geschieht analog zu dem eines SingleRequestMultiResponse-Request.

```
1  List<UPAddress> urns = new ArrayList<UPAddress>() {{
2    add(server.getLocalUPAddress());
3    add(...);
4  }};
5
6  ReliableRequest request = new MultiRequestMultiResponseRequest(
       urns, "flashNode".getBytes());
7  Future<Response> responseFuture = client.send(request);
8
9  //auf Response warten
10 Response response = responseFuture.get();
11 if (response instanceof MultiRequestMultiResponseResponse){
12   // "uberprüfe Responses ... über: response.getResponses()
13 }
```

Listing 4.18: Senden eines MultiRequestMultiResponseRequest

Ein ProgressListener für MultiResponse-Nachrichten läßt sich wie folgt Clientseitig hinzufügen:

```
1  //erzeuge ProgressListenerRunnable
2  ProgressListenerRunnable progressListenerRunnable = new
       ProgressListenerRunnable() {
3    public void progress(String senderUrn, byte[] payload, int
         current, int total) {
4      //tu etwas mit einzeln empfangener Response z.B.
5      log.info("received:␣" + current + "␣of␣" + total);
6    }
7
8    public void run() {
9      //tu etwas mit komplett empfangener MultiResponse z.B.
10     Response response = responseFuture.get();
11   }
12 };
13
14 // ProgressListener dem ResponseFuture hinzufügen
15 responseFuture.addListener(progressListenerRunnable, new
       ScheduledThreadPoolExecutor(2));
```

Listing 4.19: Verwenden eines ProgressListener

RPC

In dem folgenden Beispiel soll veranschaulicht werden, wie Remote Procedure Calls in Ubermep aufgerufen werden können. Dies wird mittels der bereits beschriebenen Funktionalität des flashens von Nodes veranschaulicht. Dazu muß zuerst mittels der Protobuf-IDL (siehe Abschnitt 2.4 im Grundlagenkapitel) ein *FlashNodeServiceProtokoll* erzeugt werden. Das Listing 4.20 zeigt dafür ein Beispiel-Protokoll, welches anschließend mittels des protoc-Compiler in eine Java-Klasse übersetzt wird. Der Aufruf der Methode *flashNode* erfolgt dabei über ein FlashNodeRequest, welcher einen Parameter *delay* enthält. Das Ergebnis des Aufrufs wird über den FlashNodeResponse zurückgegeben, wobei dafür ein Parameter *success* hinzugefügt wurde.

```
1  option java_generic_services = true;
2
3  service FlashNodeService {
4    rpc flashNode(FlashNodeRequest) returns (FlashNodeResponse);
5  }
6
7  message FlashNodeRequest{
8    required uint32 delay = 1;
9  }
10
11 message FlashNodeResponse{
12   required bool success = 1;
13 }
```

Listing 4.20: FlashNodeServiceProtocol.proto

Anschließend wird eine Implementierung des Service-Interface erzeugt, wobei im Folgenden Beispiel eine Implementierung des BlockingInterface verwendet

wurde. Im Folgenden sei auch hier angenommen, dass eine Methode *flashNode()* mit der bereits beschriebenen Funktionalität vorhanden ist ,wobei die Methode hier als Parameter einen *delay* als Verzögerung des Aufrufs übergeben bekommt.

```
1  public class FlashNodeBlockingServiceImpl implements
       FlashNodeServiceProtocol.FlashNodeService.BlockingInterface,
        RpcBlockingService{
2    @Override
3    public FlashNodeServiceProtocol.FlashNodeResponse flashNode(
         RpcController controller, FlashNodeServiceProtocol.
         FlashNodeRequest request) throws ServiceException {
4      // aufrufen von flashNode(delay)
5      flashNode(request.getDelay());
6      return FlashNodeServiceProtocol.FlashNodeResponse.newBuilder
           ().setSuccess(true).build();
7    }
8
9    @Override
10   public BlockingService getRpcBlockingService() {
11     return FlashNodeServiceProtocol.FlashNodeService.
           newReflectiveBlockingService(this);
12   }
13 }
```

Listing 4.21: Implementierung des FlashNodeBlockingService

Im Folgenden soll nun ein RPC, über das bereits oben erzeugte Peer-to-Peer Netzwerk, vom Client auf dem Server ausgeführt werden. Hierfür muss zuerst die *FlashNodeBlockingServiceImpl* am entsprechenden Peer Server-seitig hinzugefügt werden. Anschließend kann ein entsprechender FlashNodeRequest mittels des RPC-Channel auf dem Server wie folgt aufgerufen werden:

```
1  //Service am Server registrieren
2  server.registerService(new FlashNodeBlockingServiceImpl());
3
4  // erzeuge RPC-Request mit delay=10
5  FlashNodeServiceProtocol.FlashNodeRequest request =
       FlashNodeServiceProtocol.FlashNodeRequest.newBuilder().
       setDelay(10).build();
6
7  // hole RPC-Channel
8  UbermepRpcChannel blockingServerRpcChannel = client.
       getRpcChannel(blockingServerUrn);
9  // erzeuge neuen Blocking-Stub für FlashNodeService mittels RPC-
       Channel
10 FlashNodeServiceProtocol.FlashNodeService.BlockingInterface
       blockingService = FlashNodeServiceProtocol.FlashNodeService.
       newBlockingStub(blockingServerRpcChannel);
11
12 try {
13   //rufe flashNodeService auf
14   FlashNodeServiceProtocol.FlashNodeResponse rpcResponse =
         blockingService.flashNode(new RpcControllerImpl(), request
         );
15
16   // tu was mit RPC-Response, z.B. success überprüfen
17 } catch (ServiceException e) {
18   // tu was mit ServiceException
```

```
19 }
```

Listing 4.22: Aufruf des FlashNodeBlockingService

Der Aufruf eines (non-blocking)-Service ist analog und soll deshalb hier nicht weiter vertieft werden. Nur soviel: der Service-Aufruf benötigt eine zusätzliche (non-blocking)-Service-Implementierung sowie einen Callback, der aber einfach über:

```
1 MEPRpcCallback<FlashNodeServiceProtocol.FlashNodeResponse>
    callback = new RpcCallbackImpl<FlashNodeServiceProtocol.
    FlashNodeResponse>();
```

Listing 4.23: Erzeugung eines FlashNodeServiceCallback

erzeugt werden kann.

5 Evaluation

5.1 Netzwerk-Topologien

In den Abbildungen 5.1 und 5.2 ist der Aufbau der Evaluationsszenarios zu sehen. Dabei handelt es sich zum Einem um Single-Hop- und zum Anderen um Multi-Hop-Netzwerke. Die Nachrichten zur Evaluation werden dabei immer von einem Peer A zu einem Peer B und ggf. zurück gesendet. Jeder Peer wird dabei lokal an einem Rechner über ein Loopback-Interface gestartet um Störeinflüße realer Netzwerk-Kommunikation auszuschließen. Damit die Evaluationsergebnisse mit Szenarien der realen Umwelt vergleichbar sind, wurde für die Evaluation eine Übertragungsrate von 16 MBit/s emuliert.

Abbildung 5.1: Single-Hop-Topologie

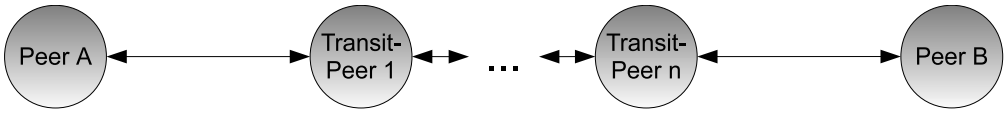


Abbildung 5.2: Multi-Hop-Topologie

5.2 Evaluationskriterien

Zur Evaluation der Implementierung wurden zwei Kriterien verwendet:

- Übertragungsdauer in Abhängigkeit der Payload-Größe
- Übertragungsdauer in Abhängigkeit der Hop-Anzahl

Im Folgenden werden nun diese beiden Kriterien näher erläutert.

5.2.1 Übertragungsdauer in Abhängigkeit der Payload-Größe

Das Kriterium *Übertragungsdauer in Abhängigkeit der Payload-Größe* zeigt das Verhältnis der Übertragungsdauer zu der Größe des übertragenen Payloads. Dabei wurden Single Request Single Response-Nachrichten in einem Single-Hop-Netzwerk versendet. Die gemessene Übertragungsdauer entspricht dabei der Zeit vom Versand einer Nachricht bis zum Empfang der Antwort. Dabei variiert die Größe des Request-Payloads, der Response-Payload aber ist immer jeweils 1 Byte groß. Mittels dieses Kriteriums lässt sich feststellen wie sich die Übertragungsdauer, in Abhängigkeit der Größe einer Nachricht, verhält.

5.2.2 Übertragungsdauer in Abhängigkeit der Hop-Anzahl

Das Kriterium *Übertragungsdauer in Abhängigkeit der Hop-Anzahl* berechnet das Verhältnis der Übertragungsdauer zu der Anzahl an Hops in einem Netzwerk. Dabei wurde die Übertragungsdauer für ein Single-Hop-Netzwerk, sowie Multi-Hop-Netzwerke der Größe 2, 3, 4 und 5 Hops gemessen. Des weiteren wurde die Übertragungszeit für den Nachrichtenversand mittels Uberlay, sowie für den Nachrichtenversand mittels Unicast, SingleRequestSingleResponse und RPC gemessen. Dafür wurden Nachrichten mit einer Größe von 1 Byte über das entsprechende Netzwerk gesendet. Die gemessene Übertragungsdauer für Uberlay und Unicast entspricht dabei der Zeit vom Versand auf Peer A bis zum Empfang der Nachricht auf Peer B. Für SingleRequestSingleResponse und RPC entspricht die gemessene Übertragungsdauer der Zeit vom Versand bis zum Empfang der Antwort auf Peer A, wobei für RPC ein BlockingService verwendet wurde. Mittels dieses Kriteriums lässt sich feststellen wie sich die Übertragungsdauer eines Nachrichtentyps zu der Anzahl an Hops verhält.

Bei beiden Kriterien wurde die Übertragungszeit in bereits verbundenen Netzwerken gemessen, d.h. die benötigte Zeit zum Aufbau eines Netzwerks ist hier nicht inkludiert.

5.3 Testergebnisse

Für das Generieren der Testläufe wurden Testprogramme mit Java geschrieben, welche die oben beschriebenen Netzwerk-Topologien erzeugen und über diese die Nachrichten versenden, sowie die Übertragungsdauer messen. Diese Ergebnisse werden in sogenannten *Boxplots* dargestellt. Im Folgenden werden nun Boxplots erläutert.

5.3.1 Boxplot

Ein Boxplot ist ein Diagramm, das zur grafischen Darstellung der Verteilung von Daten verwendet werden kann. Der Boxplot vermittelt dabei schnell einen

Überblick, in welchem Bereich die Daten liegen und wie sie sich in diesem verteilen. Ein Beispiel-Boxplot ist dabei in Abbildung 5.3 zu erkennen. Die Verteilung wird dabei mittels eines Medians, zwei Quartilen und den beiden Whiskern dargestellt.

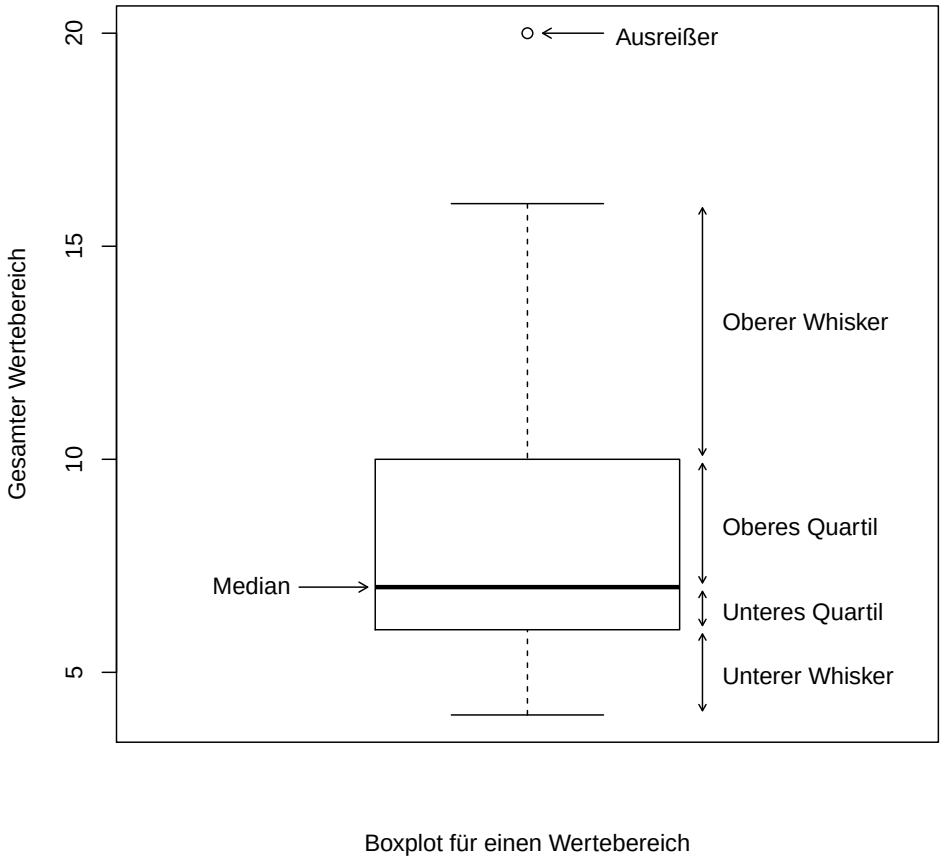


Abbildung 5.3: Boxplot-Beispiel

Quartil Das Quartil, begrenzt durch das *Obere Quartil* und *Untere Quartil*, beschreibt die mittleren 50 % der Werte. Die Länge vom oberen Quartil zum unteren Quartil entspricht dabei dem Interquartilsabstand (IQR), welcher die Streuung der Werte zeigt.

Whisker Der Whisker, begrenzt durch den *Oberen Whisker* und *Unteren Whisker*, beschreibt den gesamten Wertebereich, sofern die Werte dabei nicht außerhalb des 1,5 -fachen IQR zum jeweiligen Quartil liegen. Sofern ein Wert ausserhalb des 1,5-fachen IQR liegt, wird dieser als *Ausreißer* deklariert.

Median Ein Median ist der *Zentralwert*, welcher die Mitte eines Wertebereichs zeigt. Dafür wird der Wertebereich der Größe nach geordnet. Der Wert in der Mitte entspricht dann dem Median. Ist die Anzahl der Werte gerade, wird das arithmetische Mittel der beiden mittleren Werte berechnet. Aus diesem Grund hat der Median die Eigenschaft, dass die eine Hälfte der Werte über diesem und die andere darunter liegt.

Die nun im Folgenden dargestellten Ergebnisse entsprechen einer Messung von 100 versendeten Nachrichten. Zu Testzwecken wurden auch bis zu 1.000 Nachrichten versendet und deren Übertragungszeit gemessen. Dabei sind die entscheidenden Ergebnisse, wie Quartile, Whisker und Medians nahezu identisch, einige Ausreißer allerdings sorgten für eine unübersichtliche Darstellung, weshalb die Messung auf 100 Nachrichten beschränkt wurde.

5.3.2 Übertragungsdauer in Abhängigkeit der Payload-Größe

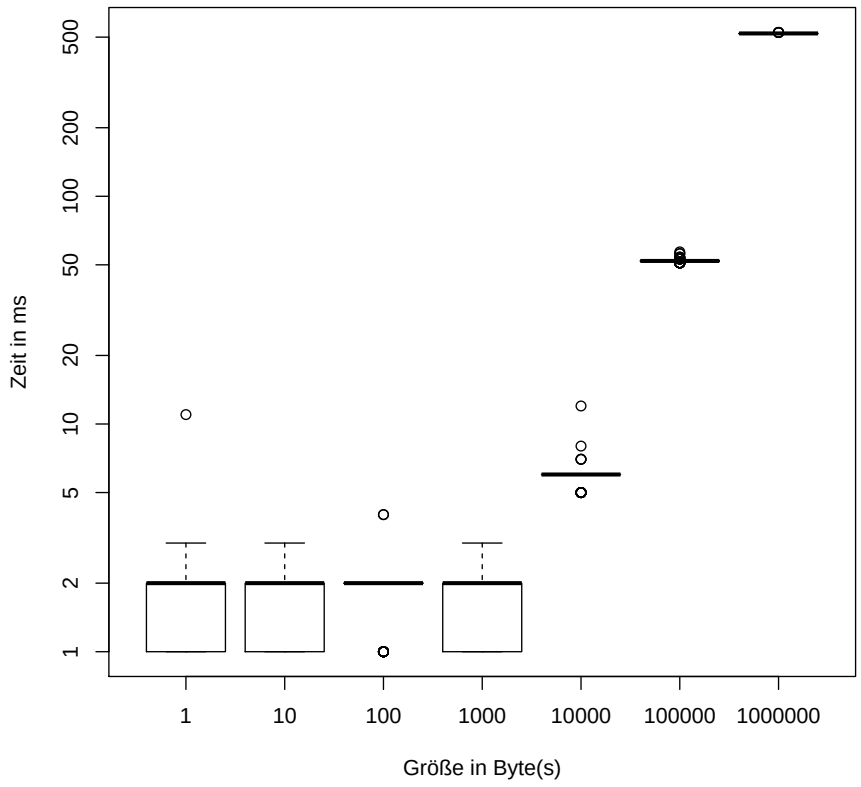


Abbildung 5.4: Ergebnisse: Übertragungsdauer in Abhängigkeit der Payload-Größe

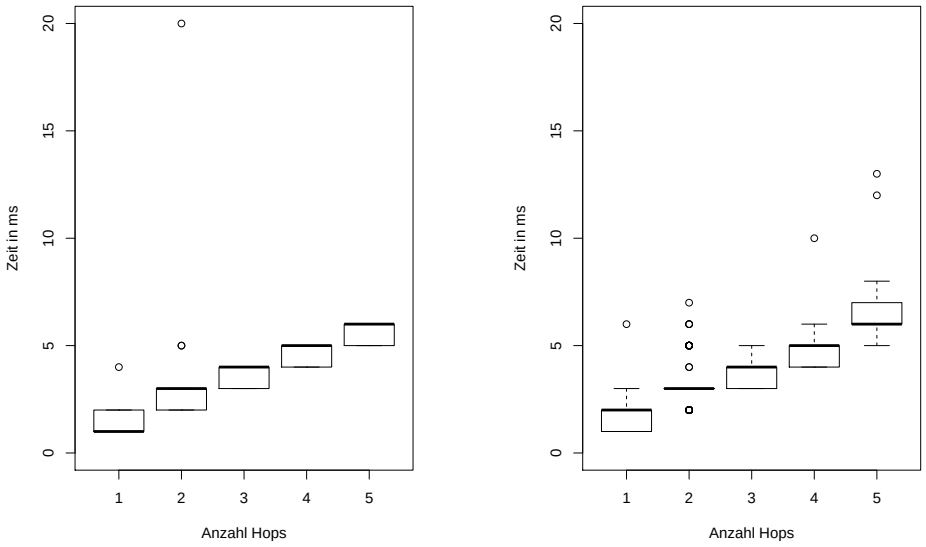
Die Abbildung 5.4 zeigt die gemessenen Ergebnisse des Kriteriums *Payload zu*

Übertragungsdauer.

Die x-Achse zeigt die Größe der versendeten Nachricht, wobei die Größe logarithmisch steigt. Die y-Achse zeigt die Übertragungsdauer in Millisekunden.

Bei der Auswertung der Ergebnisse schien zunächst verwunderlich, das sich die Übertragungszeiten für Payloads <= 1 KB scheinbar linear bei 2 Millisekunden bewegen. Da sich aber die Übertragungszeiten für Nachrichten der Payload-Größe < 1 KB, bei einer emulierten Übertragungsrate von 16 MBit/s im Nanosekundenbereich verändern, lässt sich dies auf Messungenauigkeiten und Rundungsfehler zurückführen. Für Payload-Größen > 1KB, bei der sich die Übertragungszeit im Millisekundenbereich verändert, ist zu erkennen: erhöht sich die Größe des Payloads, so steigt mit ihr linear die Übertragungsdauer.

5.3.3 Übertragungsdauer in Abhängigkeit der Hop-Anzahl

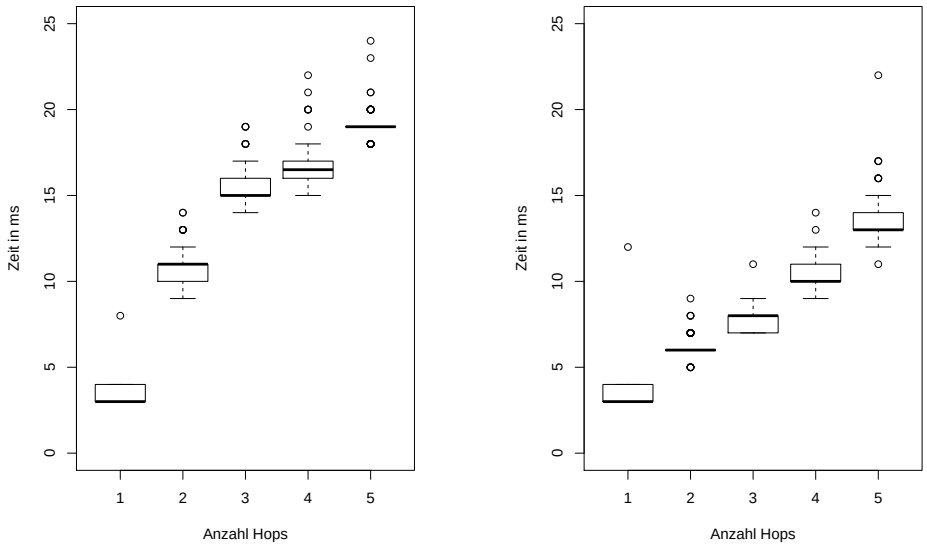


(a) Versand mittels Uberlay

(b) Versand von Unicast

Abbildung 5.5: Ergebnisse: Übertragungsdauer in Abhängigkeit der Hop-Anzahl für den Nachrichtenversand mittels Uberlay sowie den Versand von Unicast-Nachrichten

Die Abbildungen 5.5 und 5.6 zeigen die Ergebnisse des Kriteriums *Übertragungsdauer in Abhängigkeit der Hop-Anzahl*. Die Abbildung 5.5 zeigt dabei die Ergebnisse für den Nachrichtenversand mittels Uberlay und dem Message Exchange Pattern Unicast. Die Abbildung 5.6 zeigt die Ergebnisse des Kriteriums *Übertragungsdauer in Abhängigkeit der Hop-Anzahl* für den Nachrichtenversand mittels dem Message Exchange Pattern Single Request Single Response



(a) Versand von Single Request Single Response Nachrichten

(b) Aufruf mittels RPC

Abbildung 5.6: Ergebnisse: Übertragungsdauer in Abhängigkeit der Hop-Anzahl für den Versand von Single Request Single Response Nachrichten sowie den Aufruf mittels RPC

und den Aufruf für ein RPC. Die x-Achse zeigt jeweils die Anzahl der Hops, die für einen Nachrichtenversand passiert werden. Sendet beispielsweise der Peer A eine Unicast-Nachricht über ein 3-Hop Netzwerk an Peer B, so muss diese Nachricht 3 Hops passieren. Für Single Request Single Response-Nachrichten, sowie den Aufruf mittels RPC kommt dieselbe Anzahl für die Antwort hinzu. In der Summe werden insgesamt also 6 Hops passiert. Auf der y-Achse ist die Übertragungsdauer des Nachrichtenaufrufs in Millisekunden abgebildet.

Aus den Ergebnissen der Abbildung 5.5 lässt sich einerseits feststellen, dass der zu Uberlay äquivalente Nachrichtentyp Unicast, minimalen zusätzlichen Overhead im Vergleich zum Nachrichtenversand mittels Uberlay produziert. Die vergleichbaren Mediane unterscheiden sich um maximal 1 ms, das bedeutet, die Übertragungsdauer ist nahezu identisch. Des weiteren ist zu erkennen, dass die Übertragungsdauer wie erwartet linear mit der Anzahl der zu passierenden Hops steigt.

Aus den Ergebnissen der Abbildung 5.6 lässt sich feststellen dass sich der Versand der vergleichbaren Nachrichtentypen Single Request Single Response und RPC nahezu identisch verhält. Beide Typen benötigen eine ähnliche Übertragungsdauer und steigen linear mit der Anzahl der passierten Hops. Der zusätzliche Overhead zu Nachrichtentypen des One-Way Pattern resultiert aus der Wartezeit auf die Response, welche vor allem durch die Eigenschaft herbei-

geführt wird, dass diese Nachrichten die doppelte Anzahl an Hops passieren.

5.3.4 Schlussfolgerung

Aus den ermittelten Ergebnissen der beiden Evaluationskriterien *Übertragungsdauer in Abhängigkeit der Payload-Größe* und *Übertragungsdauer in Abhängigkeit der Hop-Anzahl* lassen sich nun abschließend, für den Nachrichtenversand mittels Ubermep, folgende Schlussfolgerungen ziehen:

- Ubermep produziert mininalen zusätzlichen Overhead zu vergleichbaren Nachrichtenübermittlungen mittels Uberlay.
- Die Übertragungszeit der Nachrichtentypen von Ubermep steigt linear in Abhängigkeit der Anzahl der passierten Hops.
- Die Übertragungszeit steigt linear in Abhängigkeit der Größe des übertragenen Payloads.

Des weiteren läßt sich sagen, dass die Ergebnisse beider Kriterien eine gute Performanz der entwickelten Lösung zeigen.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Bachelorarbeit wurde eine Middleware-Lösung für den Aufbau eines Peer-to-Peer-basierten Overlaynetzwerks entwickelt. Die entworfene Middleware, namens *Ubermep*, erlaubt Peers den Nachrichtenaustausch anhand von *Message Exchange Pattern*.

Dazu wurden zu Beginn der Arbeit die benötigten Message Exchange Pattern: *Unicast*, *Multicast*, *Single Request Single Response*, *Single Request Multi Response*, *Multi Request Multi Response* sowie *RPC* identifiziert. In der Entwurfsphase wurde dann ein Protokoll entwickelt welches die verschiedenen Message Exchange Pattern abbildet. Anschließend wurde die Architektur der Middleware auf Basis des Frameworks JBoss Netty, einem Framework zur Erstellung von netzwerkbasierten Applikationen, und Uberlay, einem hochperformanten Overlay-Netzwerk für kleine Netzwerke, entworfen. Im Anschluss wurde das Protokoll sowie die Java-Bibliothek anhand der konzipierten Architektur implementiert und getestet. Abschließend wurde die Implementierung auf Basis verschiedener Performanzkriterien evaluiert.

Die Evaluation der Implementierung zeigte dass die Anforderungen an gute Performanz und niedrige Latenzen erfüllt sind. Darüber hinaus gibt es nur einen geringen zusätzlichen Overhead gegenüber dem Nachrichtenaustausch ohne Message Exchange Patterns bei direkter Verwendung von Uberlay.

Das Ergebnis der Arbeit liefert daher eine performante Middleware-Lösung, welche Entwickler verteilter Anwendungen eine Java-Bibliothek bietet, die Nachrichten der oben genannten Muster in einem Overlay-Netzwerk versendet.

Die Einsatzgebiete von Ubermep, wie in der Aufgabenstellung definiert, sind vielfältig und reichen von der Verwendung für den Nachrichtenaustausch kleinerer Datenmengen in der Infrastruktursoftware *Testbed-Runtime* bis zur Verwendung für den Nachrichtenaustausch mittlerer bis großer Datenmengen in File-Sharing-Netzwerken. Die Unterstützung entfernter Funktionsaufrufe mittels RPC, ermöglicht des weiteren die Nutzung verteilter Rechenresourcen z.B. im Bereich des Grid-Computing.

Für die Zukunft ist geplant das Publish-Subscribe Pattern für verteilte Systeme, in Ubermep zu integrieren. Betrachtet man die Peers im Overlay-Netzwerk als Produzenten und Konsumenten von Nachrichten so senden in der derzeitigen Implementierung von Ubermep die Produzenten die Nachrichten direkt an die Konsumenten. Im Publish-Subscribe Pattern werden diese voneinander entkoppelt. Ein Sender (sogenannter Publisher) sendet eine Nachricht an einen Vermittler (Broker). Empfänger (sogenannte Subscriber) bekunden Interesse an

ausgewählten Informationen, z.B. bestimmt über Filter auf sogenannte *Topics*. Der Broker übernimmt dann den Transport der Nachricht zu den registrierten (also interessierten) Subscribern. Die Realisierung in Ubermep könnte dabei über die bereits exisitierenden Nachrichtentypen der Message Exchange Pattern erfolgen. Zusätzlich würde aber die Funktionalität des Brokers benötigt werden.

Abbildungsverzeichnis

Abkürzungsverzeichnis

BDD	Behavior Driven Development
CAN	Content Addressable Network
DHT	Distributed Hash Table
IDL	Interface Definition Language
IQR	Interquartilsabstand
MEP	Message Exchange Pattern
POM	Project Object Model
RPC	Remote Procedure Call
TDD	Test Driven Development
UP	Uberlay Protocol
URN	Uniform Resource Name
WSN	Wireless Sensor Network

Bibliographie

[1] Apache, S. F. Maven. http://maven.apache.org/.

[2] Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Looking up data in p2p systems. *Communications of the ACM - Web science* (2003).

[3] Dunkel, J., Eberhart, A., Fischer, S., Kleiner, C., and Koschel, A. *System-Architekturen für verteilte Anwendungen*. Hanser, 2008.

[4] Google. Google Protocol Buffers. http://code.google.com/p/protobuf/.

[5] Google. Google Protocol Buffers. http://code.google.com/apis/protocolbuffers/docs/encoding.html.

[6] Google. Google Protocol Buffers. http://code.google.com/intl/de-DE/apis/protocolbuffers/docs/proto.html#options.

[7] Institut der Telematik an der Universität zu Lübeck. Uberlay. https://github.com/itm/uberlay.

[8] JBoss. JBoss-Netty. http://www.jboss.org/netty.

[9] OMG. Common Object Request Broker Architecture (CORBA)-Specifications. http://www.omg.org/spec/CORBA/.

[10] Szczepan Faber. Mockito. http://code.google.com/p/mockito/.

[11] Szczepan Faber. Mockito. http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html.

[12] World Wide Web Consortium. Web services. http://www.w3.org/TR/ws-arch/#relwwwrest.