A Browser-based Dynamic Overlay for Secure Real Time Communication via WebRTC

Masterarbeit

von

Andreas Disterhöft

aus

Duschanbe, Tadschikistan

vorgelegt am

Lehrstuhl für Technik sozialer Netzwerke
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

Juli 2014

Betreuer:
Jun.-Prof. Dr.-Ing. Kalman Graffi

Zusammenfassung

Diese Arbeit befasst sich mit der Implementierung einer sicheren und browserbasierten Chatplattform. Der recht junge WebRTC Standard [BNBJ13] bietet die Möglichkeit der direkten Kommunikation zwischen Browserinstanzen und dem Austausch von Echtzeitdaten wie Audio und / oder Video. So ist in dieser Arbeit eine Webanwendung entstanden, die ihren Nutzern eine P2P-Chatplattform anbietet. Das Besondere dabei: Der Benutzer muss im Gegensatz zu anderen P2P-Chatlösungen keine weitere Software installieren, da diese dank WebRTC im Browser lauffähig ist.

Das verwendete P2P-Overlay der Chatplattform erweitert das quelloffene OpenChord [LK06] um Funktionalitäten, die für einen authentifizierten und sicheren Chat nötig sind. So kommt ein asymmetrisches Kryptosystem zum Einsatz, welches unter Anderen die verschlüsselte Sicherung von Daten, z.B. Kontaktlisten, im Chord-Ring anbietet. Weiter schafft das System die Möglichkeit einer Zertifikatinfrastruktur, wodurch Benutzer innerhalb des Chord-Rings gesucht werden können. Neben den vielen Chatfunktionalitäten entsteht auch eine Monitoringschicht, welche eine Evaluation des P2P-Overlays möglich macht.

Die Analyse der gesammelten Daten während eines Testlaufs von 60 Minuten schafft einen Einblick in die Leistung des Overlays. So liegt das aggregierte Datenaufkommen pro Knoten im Durchschnitt bei etwa 2,5 $\frac{Kilobyte}{s}$. Der Flaschenhals eines jeden privaten Internetanschlusses ist der Upload. Bei einer maximalen Datenrate von 1 $\frac{Megabit}{s}$ liegt die Auslastung bei etwa acht Prozent. Aktuelle Anschlüsse bieten eine maximalen Uploadrate von 5 $\frac{Megabit}{s}$, womit die Auslastung nunmehr bei 1,6 Prozent liegt. Die durchschnittliche Wartezeit der Zustellbestätigung einer Chatnachricht dauert etwa 805 ms und ist mit unter einer Sekunde ein gutes Ergebnis.

Danksagung

Zuerst möchte ich mich bei Jun.-Prof. Dr.-Ing. Kalman Graffi für die überaus gute und freundliche Betreuung bedanken. Er gab mir stets wertvolle Tipps, die mir bei der Implementierung der Chatplattform halfen.

Weiter möchte ich Dominik Weinhold, Thomas Kampermann, Sandra Golor, Katharina Golor, Jessica Jonda, Ian Webermann, Lilia Weigandt und Marc Ewert für die aufgewendete Zeit für das Testszenario danken. Im besonderen Maße bedanke ich mich bei meinem Bruder Alexander Disterhöft für seine aufgewendete Zeit. Seine Ressourcen machten die Durchführung des Testszenarios erst möglich.

Erneuter Dank gelten Dominik Weinhold, Thomas Kampermann, Sandra Golor und Marc Ewert, die sich zum Ende der Arbeit erneut Zeit genommen haben, um mir Verbesserungsvorschläge und Korrekturhilfen zu geben.

Abschließend bedanke ich mich herzlichst bei meinen Eltern und meiner Freundin Sandra Golor, die mich während meines gesamten Studiums unterstützten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung und Motivation

P2P-Netzwerke ermöglichen eine direkte Kommunikation von Nutzern, nachfolgend *Peers* genannt, ohne Umwege über Server. Im Gegensatz zu einer Client / Server-Struktur skaliert eine P2P-Lösung, da mit zunehmender Anzahl an Peers die Speicherkapazität, Rechenleistung und Bandbreite des Netzwerks zunimmt und somit mächtiger wird. Insbesondere wird die aufkommende Last und die benötigte Speicherkapazität auf die einzelnen Peers des P2P-Netzwerks aufgeteilt. Software, die ein P2P-Netzwerk aufspannt und nutzt, muss von den Benutzern zunächst installiert werden, was für viele Benutzer eine Barriere für die Nutzung dieser darstellt. Ist die Barriere des Herunterladens und Installierens der Software überwunden, so ist bei einem Update der Software wichtig, dass dies alle teilnehmende Peers installieren, da eventuell Inkompatibilitäten mit veralteter Software auftreten können. Schlimmer noch: werden grundlegende Änderungen an der Software vorgenommen, so müsste der Benutzer gezwungen werden diese zu installieren. Weiter sind Anwendungen, die die Möglichkeit schaffen überwachungsfrei und nicht-abschaltbar über das Internet zu kommunizieren, rar. Aktuelle Lösungen sind überwachbar und / oder abschaltbar, da diese zumeist eine Client-Server-Architektur verwenden.

Durch die Einführung des WebRTC Standards [BNBJ13] bietet sich nun die Möglichkeit

an eine verschlüsselte direkte (Multimedia-)Kommunikation zwischen Browserinstanzen aufzubauen. Insbesondere besteht die Möglichkeit ein P2P-Netzwerk auf Basis von Browsern, quasi Browser-to-Browser, zu errichten. Hierdurch fällt die Barriere neue Software installieren zu müssen, da jeder Internetnutzer mindestens einen Browser installiert hat. Weiter ist die clientseitige Installation von Updates nicht notwendig, da gewissermaßen mit dem Laden der Internetseite die Übertragung der Software in Form von HTML5 und Javascript Dateien erfolgt. Die Zahl der Browser, welche den WebRTC Standard implementieren, wächst zudem stetig, sodass Mitte 2014 bereits Chrome, Firefox und Opera diesen implementieren. Weitere Softwarefirmen, die Browser anbieten, arbeiten bereits an der Implementierung.

Da die WebRTC Standardisierung noch nicht komplett durchgeführt ist, sind vergleichbare Projekte, die ein P2P-Overlay mit Browsern aufbauen, aktuell[1] nicht vorhanden. Somit ist der Grundstein für diese Arbeit gelegt.

1.2 Zielsetzung

Nach der Erläuterung der Problemstellung und Motivation im vorangegangenen Abschnitt folgt nun das Ziel dieser Arbeit. Die Zielsetzung beinhaltet die Implementierung einer sicheren und modular aufgebauten Chatplattform auf einem geeignetem P2P-Overlay, welche auf einem Webserver lauffähig sein muss. Weiter soll das P2P-Overlay auf der Kommunikationsschicht WebRTC nutzen und dessen Funktionalitäten wie VoIP und Datenversand der Anwendungsschicht zur Verfügung stellen. Der Kommunikationskanal soll gesichert sein und die Authentizität der Benutzer der Chatplattform soll gegeben sein.

Die Anwendungsschicht, sprich die Chatanwendung, sollte folgende Features unterstützen. Ein Benutzer soll sich anhand eines Benutzernamens und eines Passwortes an der Chatplattform anmelden können. Die Kombination aus Benutzernamen und Passwort ist eindeutig, wodurch sich ein Benutzer gegenüber der Chatplattform authentifiziert. Eine Suche nach registrierten Benutzern der Chatplattform ist notwendig. Zuletzt muss eine

[1] Stand: Juni 2014.

Audio / Video-Chat Funktionalität gewährleistet sein. Hierzu gehört eine Kontaktlistenverwaltung, dessen Kontaktliste verschlüsselt hinterlegt und bei erneutem Login geladen werden soll, sowie Nachrichtenchats und das Audio / Video-Streaming mit Kontakten aus der Kontaktliste.

Abschließend soll eine Evaluation der entwickelten Software stattfinden. Hierfür sind geeignete Metriken zu wählen, um etwa die Performance der Plattform zu prüfen.

In den nächsten Kapiteln gilt es dies umzusetzen.

1.3 Aufbau der Arbeit

Die Ausarbeitung beginnt mit dem Grundlagenkapitel in Abschnitt 2, welches sich zunächst um den Begriff WebRTC beschäftigt, um anschließend die Frage des verwendeten P2P-Overlays zu klären. Abschließend erfolgt die Vorstellung eines geeigneten Frameworks für die Umsetzung der Chatplattform. In Kapitel 3 ist die Architektur der entstandenen Software grob umrissen, sodass der Leser einen Überblick des großen Ganzen erhält. Die Kapitel 4, 5 und 6 enthalten Details der modularen Chatplattform-Implementierung, worüber zuvor ein Überblick verschafft wurde. Im jeweils letzten Abschnitt der Kapitel sind die Beiträge des Autors nochmals gelistet. Die Auswertung dieser Arbeit befindet sich in Kapitel 7, wo ein Testlauf der Chatplattform durchgeführt wird. Die generierten Monitoringdaten bedürfen eine Erläuterung und enthalten außerdem einen Einblick in die Performance der Plattform. Schließlich beinhaltet Kapitel 8 ein Fazit dieser Arbeit mitsamt Ausblick für potentiell aufbauende Arbeiten.

Kapitel 2

Grundlagen und verwendetes Framework

Dieses Kapitel umfasst die Grundlagen der Implementierung einer Browser-to-Browser Chatplattform. Zuvor ist eine Beleuchtung des WebRTC Standards nötig. Dies geschieht in Abschnitt 2.1. Einen Überblick einiger P2P-Overlays, welche als Grundlage für die Chatplattform dienen können, verschafft Abschnitt 2.2. Die Vorstellung des verwendeten Frameworks, welches die Entwicklung der Chatplattform beschleunigen soll, erfolgt in Abschnitt 2.3.

2.1 Was ist WebRTC

WebRTC, Kurzform für *web real time communication*, ist ein offener Standard zur sicheren Echtzeitkommunikation zwischen Browsern. Die *W3C* (World Wide Web Consortium) ist für die API Standardisierung [BNBJ13] und die *IETF* (Internet Engineering Task Force) ist für die Standardisierung der Protokollebene [THT14] zuständig.

Dieser Standard umfasst fundamentale Bausteine für den Transfer von Daten-, Audio- und Video-Streaming, sowie die darunterliegende Signalisierung. Das Besondere dabei: *jegliche* Kommunikation erfolgt verschlüsselt. Hierfür wird DTLS (Datagram Transport

Layer Security) in Kombination mit SRTP (Secure Real Time Protocol: Verschlüsseltes Signierungsprotokoll) verwendet. Eine Authentifizierung ist mit externen Diensten wie OAuth, OpenID oder BrowserID möglich.

Die folgenden Browser beinhalten bereits eine Implementierung des Standards: Googles Chrome, Mozillas Firefox und Opera Softwares Opera. Auf die Funktionalität kann über eine Javascript API zugegriffen werden. Somit ist eine Möglichkeit geschaffen worden Peer-to-Peer- bzw. Browser-to-Browser-Übertragungen von Multimediainhalten zu gewährleisten. Durch den Standard sollen außerdem die Browser interoperabel werden.

Neben der Übertragung von Multimediainhalten zwischen Browsern ist auch eine Nachrichtenübertragung via DataChannels API möglich, sodass das Errichten eines P2P-Overlays auf Basis von Browsern möglich ist. Der Grundstein für eine webbasierte P2P-Chatplattform ist gelegt.

2.2 Auswahl: P2P-Overlay

Durch die Verwendung von WebRTC ist es nun möglich ein P2P-Netzwerk aus Browsern aufzubauen. Welches P2P-Overlay sich für eine Chatplattform eignet und welche Implementierungen dieser Overlays bereits existieren ist Thema dieses Abschnittes.

Zunächst stellt sich die Frage, ob ein strukturiertes oder unstrukturiertes P2P-Overlay verwendet werden soll. Diese unterscheiden sich grundlegend in ihrer Topologie und Informationssuche.

Unstrukturierte Overlays haben keine Topologie und halten Verbindungen zu mehreren, meist zufälligen, Knoten. Die Informationssuche erfolgt durch *flooding*, *random walk*, mithilfe einer hierarchischen Struktur von *super peers* oder von *rendezvous* basiertem *bubblestorm* [TKLB07], welcher die Vorteile von flooding und random walk kombiniert. Nichtsdestotrotz ist der Erfolg einer Suche nur unter Einsatz einer hohen Bandbreite zu garantieren.

Strukturierte Overlays verwenden eine DHT (Distributed Hash Table: verteilte Hashtabelle), um eine exakte Suche nach einem Objekt mit einer bekannten Objekt ID zu unterstützen. Hierfür wird einem Knoten die Zuständigkeit für einen Teil der Hashtabelle zugeordnet. Durch die Strukturierung der Topologie und den Zuständigkeiten der Hashtabelle ist ein Auffinden von Objekten in einem stabilen Overlay in $\mathcal{O}(log(n))$ möglich, wobei *n* die Anzahl an Knoten im Overlay ist.

Die Wahl des Overlaytyps fällt auf das strukturierte Overlay, da dies verglichen zum unstrukturierten Overlay eine besser skalierende Suche anbietet. Außerdem existieren bereits Implementierungen einiger bekannter strukturierter P2P-Overlays, sodass die Entwicklung des Overlays weniger Zeit in Anspruch nehmen sollte. Solche Implementierungen existieren für *Chord* [SMK+01] mit *OpenChord* [LK06], *Pastry* [RD01] mit *FreePastry* [Ric] sowie *SimPastry/VisPastry* [Mic] und *Kademlia* [MM02] mit *Kad* [SENB07].

Die verschiedenen strukturierten P2P-Overlays haben ihre Stärken und Schwächen, welche in [LCP+05] diskutiert werden. Im Rahmen dieser Masterarbeit kommt Chord als P2P-Overlay zum Einsatz. Die bekannteste Chord-Implementierung, OpenChord, bietet eine solide Basis und ist in Java geschrieben. Um die Entwicklung zu beschleunigen, wäre eine Anpassung von OpenChord denkbar, jedoch ist dann ein Framework notwendig, welches die Entwicklung einer Webanwendung mit Java bietet.

Die Antwort bietet Googles *Web Toolkit Framework*, dessen Features sowie Vor- und Nachteile nachfolgend diskutiert sind.

2.3 Google Web Toolkit Framework

Googles Web Toolkit Framework [Goo12a], kurz GWT, ist ein Framework zur Entwicklung von Webanwendungen. Das GWT-SDK beinhaltet mehrere Java API's und GUI-Elemente, sogenannte *Widgets*. Somit erfolgt die Entwicklung der Webanwendung in Java mit einigen Einschränkungen. Die emulierten Java API's und deren Einschränkungen gegenüber der Standard Java API ist Thema des Abschnitts 2.3.1. Beim Kompilierungs-

vorgang wird der in Java geschriebene Code in hoch optimierten Javascript-Code übersetzt und kann somit auf allen Browsern ausgeführt werden.

Darüber hinaus besteht die Möglichkeit mithilfe von GWT-RPC's (Remote Procedure Call: entfernte Methodenaufrufe) AJAX Anwendungen zu entwickeln, die den Austausch von Java-Objekten mit dem Server unterstützen. Hierfür wird ein Webserver benötigt, der Java Servlets implementiert. Beispiele für solche Webserver sind der Open Source Webserver Tomcat [tom14] von Apache Software Foundation oder der ebenfalls in Open Source verfügbare Webserver Jetty [Fou] der Eclipse Foundation. Letzterer wird seit GWT 1.6 im DevelopmentMode eingesetzt. Eine weitere nützliche Debugmöglichkeit ist der von GWT mitgelieferte *codeserver*, welcher im SuperDevMode eingesetzt wird. Dieser ermöglicht das Debugging der Webanwendung mithilfe von *source maps*. Source maps bilden den generierten Javascript Code auf den ursprünglichen Java Code ab, wodurch sich mit einem Browser, welcher source maps unterstützt, sogar Breakpoints im Java Code setzen lassen können. Durch diese Funktionalität wird das Debugging einer Webanwendung sehr vereinfacht.

Ein weiteres Feature von GWT ist die Einbindung von Javascript Code in Java per JSNI (JavaScript Native Interface). Dies bedeutet, dass beliebige Javascript-Bibliotheken eingebunden und in der Webanwendung verwendet werden können. Zudem ist es möglich aus dem Javascript Code heraus auf Attribute und Methoden von Java Objekten zuzugreifen. Ein Wermutstropfen bleibt jedoch: Die Einbettung von Javascript-Bibliotheken bettet gleichzeitig deren Anfälligkeiten für Javascript Angriffe mit ein. Weitere sicherheitsrelevante Details des GWT-Frameworks enthält Abschnitt 2.3.2.

Neben den vielen Features und Vereinfachungen für den Entwickler gibt es einige Einschränkungen bezüglich der Java API's. Diese werden im nächsten Abschnitt geschildert.

2.3.1 Einschränkungen Java API

Das GWT-Framework emuliert die Java API's weitestgehend. Jedoch existieren durch die Übersetzung von Java- in Javascript-Code einige Einschränkungen, die meist der Natur von Javascript bzw. Browserumgebungen zuzuschreiben sind.

Nachfolgend erfolgt eine Diskussion der Einschränkungen gegenüber J2SE 1.5, welche Auswirkungen auf die Implementierung der Chatplattform haben. Eine vollständige Auflistung der Einschränkungen befindet sich unter [Goo12b].

Das erste Problem ist, dass Javascript nur einen numerisch primitiven Datentyp hat. Dieser ist eine 64 Bit lange Fließkommazahl. Hierdurch werden alle primitiven Java Datentypen mit einer 64 Bit langen Fließkommazahl emuliert. Dies hat Nebeneffekte, die im Auge behalten werden sollten. Zum einen führt der Java spezifische Überlauf, beispielsweise einer 8 Bit langen Ganzzahl, *nicht* zu einem Überlauf der emulierten Fließkommazahl. Ist dies gewünscht, so muss dieser Überlauf im Java Code emuliert werden. Zum anderen ist die Genauigkeit des Java Datentyps *float* höher und entspricht der eines *double*. Eine Ausnahme der Emulation bildet der Java Datentyp *long*, welcher eine 64 Bit lange Ganzzahl darstellt. Die Emulation dieser ist mit zwei 32 Bit Ganzzahlen erreicht worden. Das Überlaufverhalten wird in diesem Fall jedoch korrekt emuliert aber dieser Datentyp sollte aus Performancegründen nur als Ausnahme verwendet werden.

Die wohl größte Einschränkung liegt in der Natur des Javascript Interpreters, welcher single-threaded arbeitet. Somit muss die Software komplett single-threaded und eventbasiert programmiert werden, wobei darauf geachtet werden muss, den einzigen Thread nicht zu blockieren. Damit längere Operationen den einzigen Browser-Thread nicht blockieren, bietet die GWT-API eine kleine Abhilfe. Die Klasse *Scheduler* mit der Methode *scheduleIncremental* bietet die Möglichkeit, lang laufende Prozesse in Stücke abzuarbeiten, sodass Jobs inkrementell gerufen werden, falls der lang laufende Prozess nicht fertig ist. Weiter bietet die Methode *scheduleDeferred* die Möglichkeit, eine Abfolge von Befehlen erst auszuführen, wenn die „Browser event loop“ zurückkehrt, d.h. wenn der Browser zunächst keine weiteren Events verarbeiten muss. Durch den Wegfall von Multithreading und da angestoßene Operationen ohne Unterbrechungen durchlaufen, ist keine Synchronisierung notwendig und wird deshalb nicht unterstützt bzw. emuliert.

Die Java Reflection API, welche das dynamische Laden von Klassen unterstützt, konnte aus Effizienzgründen nicht emuliert werden. Dies hat zur Folge, dass der Java Serialisierungsmechanismus, welcher mithilfe der Reflection API realisiert ist, nicht zur Verfügung steht. Stattdessen bietet GWT das *isSerializable* Interface an, welches eine Objektserialisierung für RPC Calls, sprich für die Kommunikation zwischen Clients und

Server, anbietet. Für die Kommunikation zwischen Clients bietet GWT zunächst keinerlei Serialisierungsmöglichkeiten an. Dies hat Auswirkungen auf die Kommunikationsschicht der Chatplattform und wird dort erneut aufgegriffen.

Für die Entwicklung der Chatplattform ist neben den Einschränkungen der Java API auch die Sicherheit des GWT-Frameworks wichtig. Der nächste Abschnitt beschäftigt sich mit eben dieser.

2.3.2 Sicherheit

Das Ziel dieser Arbeit ist die Implementierung einer sicheren Chatplattform. Doch wenn das Framework bereits Sicherheitslücken aufwirft, so ist die Schaffung der Sicherheit auf einer höheren Ebene wirkungslos. In diesem Abschnitt wird die Sicherheit des GWT-Frameworks beleuchtet und Anforderungen an die Implementierung der Chatplattform geschaffen. Einen kompletten Überblick über die in diesem Kapitel aufgezeigten Punkte liefert Googles Internetpräsenz des GWT-Projekts [Goo12c].

Durch die Übersetzung von Java- in Javascript Code seitens GWT sind alle Angriffe auf Javascript von Bedeutung. Um den Kommunikationsweg von Client und Server und somit den Versand der HTML sowie Javascript Codedateien zu schützen, ist die Nutzung von SSL essenziell. Hierzu muss der Webserver mittels SSL und einem gültigen Zertifikat abgesichert werden.

Zu den drei wichtigsten Angriffsklassen auf Javascript zählen das Cross-Site Scripting (XSS), Cross-Site Request Forging (XSRF) und JSON im Zusammenhang mit XSRF. Um diesen Angriffen Paroli bieten zu können, werden GWT-Entwicklern einige Hinweise mitgegeben.

Um **XSS** keine Angriffsfläche zu bieten ist es zunächst wichtig jegliche Eingabe, die von außen eintrifft oder vom Benutzer initiiert wird, zu überprüfen, bevor eine weitere Verarbeitung erfolgt. So sollte beispielsweise niemals die Widget Methode *setInnerHtml* mit einer Benutzereingabe getätigt werden. Hierzu hält das GWT-Framework den sogenannten SafeHtmlBuilder bereit, welcher vor XSS gesicherte Texteingaben in HTML

umwandelt. Weiter ist die Nutzung des GWT-*JSONParser* bei nicht vertrauenswürdigen Zeichenketten gefährlich. Dies beruht darauf, dass der JSONParser die Javascript *eval* Funktion ruft. Diese führt wiederum möglicherweise enthaltenen Scriptcode aus, sodass per JSON XSS Angriffe durchgeführt werden können. Abhilfe schafft die Überprüfung der JSON Zeichenkette mittels *JsonUtils* der GWT-API, bevor diese geparst wird. Die Klassenmethoden *escapeValue* und *safeToEval* überprüfen, ob eine JSON-Zeichenkette gültige JSON-Objekte nach RFC [Cro06] enthält und eine sichere Ausführung der eval Methode gegeben ist. Eine weitere Angriffsfläche bietet selbst geschriebener JSNI Code, welcher unsichere Operationen wie das Rufen der Funktionen eval, document.write(), setInnerHtml, etc. durchführt. Zuletzt sind natürlich alle ins Projekt eingebetteten Javascript-Bibliotheken auf potentielle Sicherheitslücken zu prüfen, da diese einige der oben genannten Richtlinien verletzen könnten.

XSRF Angriffe zielen auf ein fehlerhaftes Session Management des Servers ab. Werden in der Chatplattform keine sensiblen Daten an den Server geschickt bzw. erfolgt der Login ohne den Server, sodass kein Session Management stattfindet, so ist diese Art von Angriff irrelevant. Wann immer eine Anfrage an den Server innerhalb einer Session versendet wird, schafft die Verwendung eines Cookie-Duplikats mehr Sicherheit. Der Server prüft daraufhin das vorliegende Session Cookie und das mitgesendete Duplikat auf Gleichheit. Hierdurch lässt sich diese Angriffsklasse verhindern.

Insgesamt bietet das GWT-Framework eine sehr gute Umgebung, um mit emulierten Java API's eine Chatplattform auf P2P-Basis mithilfe von WebRTC zu schaffen. Chord und dessen Implementierung OpenChord eignet sich hierbei als P2P-Overlay. Diese gilt es unter Berücksichtigung der Einschränkungen, beschrieben in Abschnitt 2.3.1, zu portieren. Weiter ist es zunächst sehr schwer Sicherheit in eine Webanwendung zu bringen, da durch die weite Verbreitung von Javascript viele Angriffe existieren, welche es abzuwehren gilt. Die Tipps an GWT-Entwickler müssen in der Implementierung in jedem Fall berücksichtigt und umgesetzt werden.

Das nächste Kapitel befasst sich mit einem groben Überblick der Architektur der Webanwendung, welche das Produkt dieser Arbeit ist.

Kapitel 3

Architektur

Dieses Kapitel beschäftigt sich mit der Architektur der Software. Ziel war es eine modulare Architektur zu schaffen, welche eine Erweiterung bzw. Auswechslung der Module möglich macht. Einen groben Überblick der Module verschafft Abbildung 3.1, welche das Zusammenwirken der fünf Module Anwendung, Chord, Chat, *Graphical User Interface* (engl.: Grafische Benutzeroberfläche) und Monitoring nahebringt. Insgesamt sind sieben Module definiert, wobei zwei Module der Übersichtlichkeit halber in der Abbildung nicht vertreten sind. Diese Module sind die Konfiguration und das Logging, welche jedes der übrigen fünf Module nutzen. Die Kommunikation zwischen den Modulen geschieht über Interfaces, welche von den jeweiligen Modulen implementiert werden.

Der Kern und ein Großteil der Logik befinden sich im Chordmodul. Dieses Modul bietet eine chatunterstützende Chordimplementierung an. Details werden in Kapitel 4 geschildert.

Das Anwendungsmodul ist der Einstiegspunkt des GWT-Frameworks und dessen Aufgaben sind die folgenden. Zum einen werden hier die gewünschten Module geladen bzw. instanziiert. Weiter hat dieses Modul eine koordinierende Funktion und kümmert sich um das Modul-Management und um die Bekanntmachung der Module untereinander. Schließlich werden Chat-spezifische Anfragen vom Chat- und GUI-Modul an Chord delegiert.

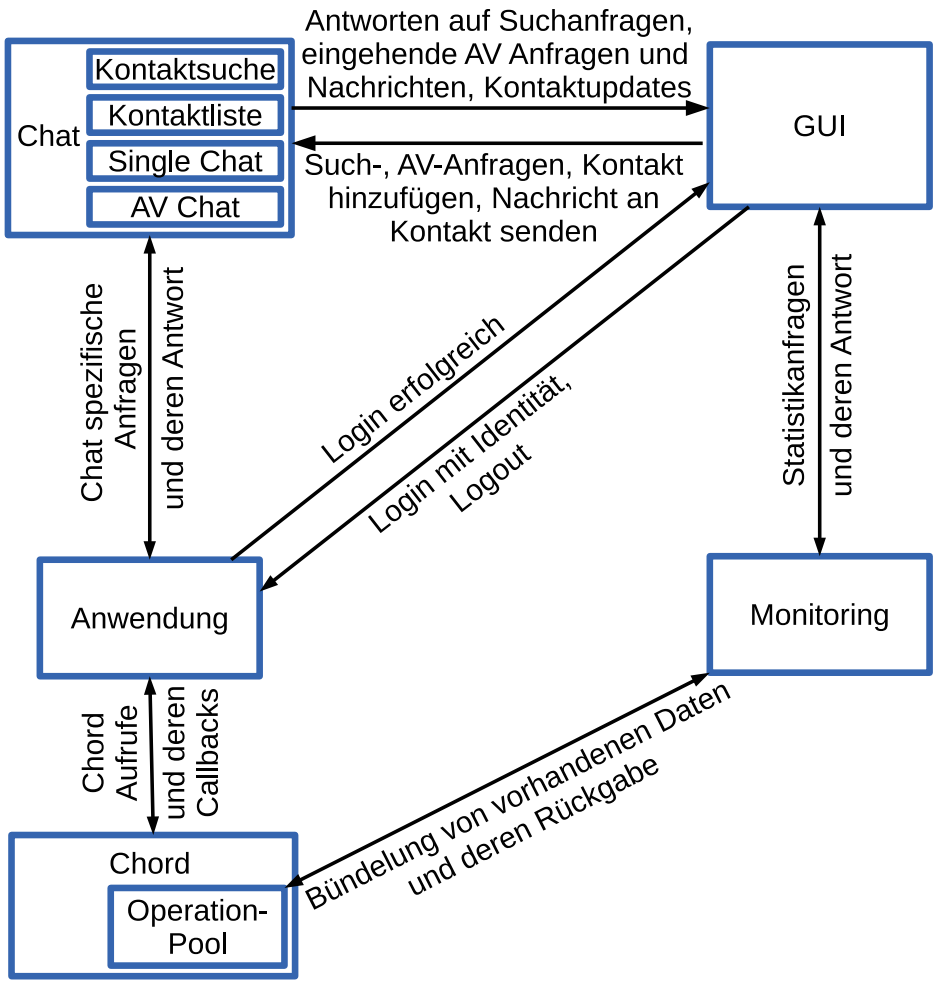


Abbildung 3.1: Architektur der Software.

Das Chatmodul selbst umfasst insgesamt vier Untermodule, welche ebenso modular und erweiterbar implementiert sind. Die Untermodule sind im Einzelnen die folgenden: Kontaktsuche, Kontaktliste, Single- und AV-Chat. Diese kommunizieren über Interfaces mit dem GUI-Modul, um eingehende valide Anfragen an dieses weiterzuleiten bzw. dessen Anfragen zu bearbeiten. Gewisse Anfragen, wie beispielsweise das Senden einer Nachricht oder das Sichern einer Kontaktliste, werden an das Anwendungsmodul weitergeleitet.

Für die grafische Oberfläche ist das GUI-Modul zuständig. Zu den Aufgaben gehören

der Aufbau der Benutzeroberfläche sowie die Verarbeitung aller Benutzereingaben und das Delegieren an verantwortliche Module.

Das letzte Modul der Abbilung ist das Monitoringmodul. Dieses Modul übernimmt Monitoringjobs, falls diese in der Konfiguration definiert wurden. Die dazu benötigte Datenbasis holt sich das Modul direkt vom OperationPool des Chordmoduls. Außerdem werden Statistikanfragen der GUI anhand der Datenbasis des Moduls beantwortet.

Die Implementierung aller sieben Module werden in den folgenden drei Kapiteln dargelegt. Die Reihenfolge der Kapitel ist im Bottom-Up Design, d.h. von unten nach oben. Den Beginn macht das Chordmodul in Kapitel 4. Daran anschließend liefert das Kapitel 5 zwei wichtige Module der Anwendungsschicht. Diese sind das Anwendungsmodul in Abschnitt 5.1 und das Chatmodul in Abschnitt 5.2. Abschließend gibt Kapitel 6 einen detaillierten Einblick in die GUI, die Konfiguration, das Logging und das Monitoring. Diese sind in den Abschnitten 6.1, 6.2, 6.3 und 6.4 zu finden.

Kapitel 4

Modul: Chord

Dieses Kapitel umfasst das Herzstück der Chatplattform, das Chordmodul. Im Rahmen dieser Arbeit ist keine von Grund auf neue Chord-Implementierung entstanden. Als Grundbaustein kam die bekannte Chord-Implementierung OpenChord [LK06], welche in Java geschrieben ist, zum Einsatz. Welche Features diese Implementierung bietet, behandelt der nachfolgende Abschnitt 4.1. Alle Änderungen bzw. neu hinzugefügte Features werden in Abschnitt 4.2 diskutiert und erläutert. In Abschnitt 4.3 findet ein Vergleich der modifizierten Implementierung mit dem ursprünglichen OpenChord statt. Zum Schluss des Kapitels sind in Abschnitt 4.4 alle Beiträge des Autors aufgelistet.

4.1 Was bietet OpenChord?

OpenChord [LK06] ist eine Chord [SMK+01]-Implementierung der Universität Bamberg und ist unter der GNU General Public License lizenziert. Eine Dokumentation, geschrieben von den Entwicklern, befindet sich unter [KL07]. Die Software ist in Java geschrieben und bietet eine erweiterbare und solide Basis, um sie als P2P-Overlay zu nutzen. Nachfolgend wird die Software in Version 1.0.5 betrachtet und anschließend modifiziert, um den Anforderungen des Moduls gerecht zu werden.

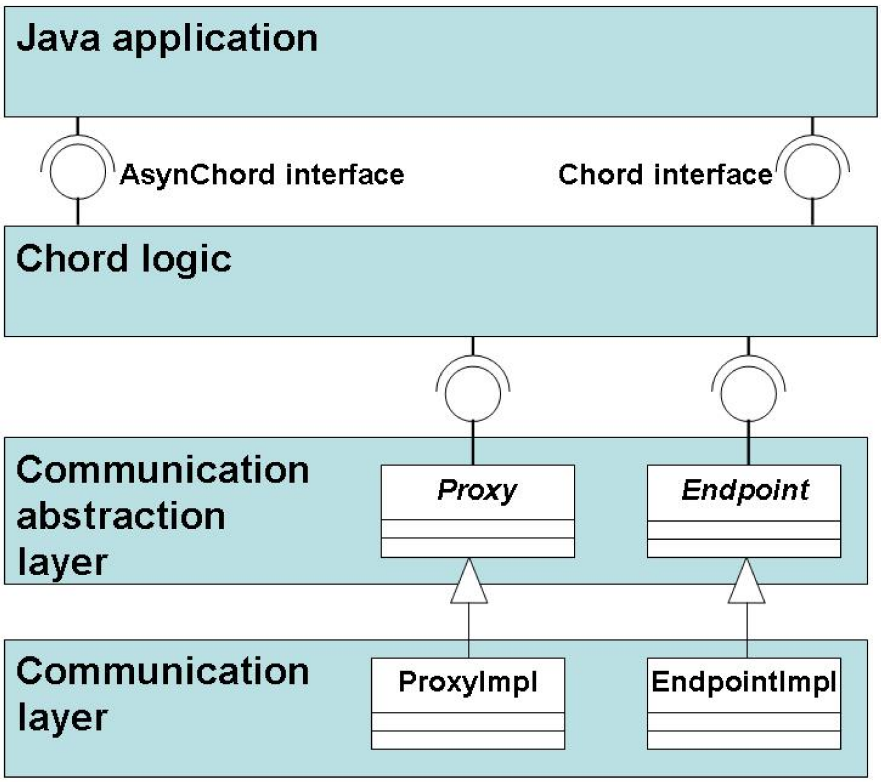


Abbildung 4.1: OpenChord Architektur. Quelle: [KL07]

Einen Überblick der OpenChord Architektur verschafft Abbildung 4.1. Diese Grafik zeigt die drei Schichten der Architektur. Die *Communication abstraction layer* (engl.: abstrahierte Kommunikationsschicht) bietet der darüber liegenden *Chord logic* eine Abstraktion der *Communication layer* (engl.: Kommunikationsschicht), welche die Implementierung der Kommunikationsprotokolle enthält.

Initial sind drei Kommunikationsprotokolle implementiert, welche die Kommunikation über Java-*Sockets*, *Remote Method Invocation (RMI)* (engl.: entfernter Methodenaufruf) und lokal erlaubt. Trotz des Vermerks der Entwickler, dass die Kommunikationsprotokolle nicht einfach ausgetauscht werden können, da diese *hardcoded* (engl.: im Quellcode verankert, nicht durch Konfiguration veränderbar) in einigen Klassen sind, gelang es ein neues Kommunikationsprotokoll zu implementieren. Mehr dazu in Abschnitt 4.2.2.

Die *Serviceschicht* (vgl.: Chord logic) von OpenChord bietet zunächst die Möglichkeit einen Chord-Ring zu erstellen, diesem beizutreten und wieder zu verlassen. Zur Manipulierung der DHT stehen drei Methoden zur Verfügung:

- *insert*: Füge ein Schlüssel / Wert-Paar (key/value pair) hinzu.
- *remove*: Lösche ein angegebenes Schlüssel / Wert-Paar.
- *retrieve*: Hole Wert(e)[1] für einen angegeben Schlüssel.

Vorteilhaft ist, dass jedes Java-*Serializable*-Objekt in der DHT gesichert werden kann. Wie in Abschnitt 2.3.1 beschrieben ist das Serializable Interface im GWT leider nicht nutzbar. Durch den Einsatz des *JsonSerializer* [Kfu11] kann ein ähnliches Verhalten erreicht werden. Weitere Details folgen im nächsten Abschnitt. Sinngemäß bietet OpenChord *custom*-key's (benutzerdefinierte Schlüssel) an, unter welchen Werte in der DHT gesichert werden können. Eine Einschränkung bezüglich der Schlüssel / Wert-Paare existiert jedoch. Die Klassendefinitionen für die Schlüssel- und Wert-Objekte müssen an jedem Knoten vorliegen. Dies ist jedoch nur dann ein Problem, wenn Knoten eines Chord-Rings unterschiedliche Versionen der Software betreiben. Da die Software dieser Arbeit auf einem Webserver läuft, verwenden, nach einer Konfigurationsanpassung (*Property* Datei) des Webservers, alle dem Chord-Ring beitretenden Knoten die gleiche Version. Nähere Informationen bietet Abschnitt 6.2.

Das Routing im Chord-Ring erfolgt *transparent* für den Benutzer. Zum Einsatz kommt eine Abwandlung des *rekursiven* Routings. Eine Anfrage gelangt zunächst rekursiv zum Zielknoten, welcher die Antwort jedoch nicht an den initial anfragenden Knoten schickt, sondern entlang der zurückgelegten Route zurück an den Anfrager leitet. Transparent erfolgt auch die Replikation der DHT und per Konfiguration kann eingestellt werden an wie vielen Nachbarn Daten repliziert werden sollen. Weitere Konfigurationsmöglichkeiten wären unter anderem die Auswahl eines Loggers und die Konfiguration der Zeiten für den Start und der Periode der maintanance-Tasks (Chord-Ring Aufrechterhaltungs-Jobs).

[1] In OpenChord ist es möglich einem Schlüssel mehrere Werte zuzuweisen, sodass ein Schlüssel der DHT auf eine Menge von Werten abbildet.

Eine Einschränkung, die in der Dokumentation der Entwickler genannt wird, ist, dass angenommen wird, dass alle Knoten, die dem Chord-Ring beitreten, vertrauenswürdig sind. So kann beispielsweise jeder Knoten Einträge der DHT manipulieren und somit Einträge von fremden Knoten löschen bzw. modifizieren. Außerdem wird die Replikation von dem Knoten angestoßen, der für ein Datum zuständig ist und nicht von dem Knoten, der das Datum einfügt. Dadurch können bösartige Knoten ihm zugeteilte Daten nicht replizieren, um diese absichtlich aus dem Chord-Ring bei dessen Abzug zu entfernen. Weiter können bösartige Knoten Daten in den Chord-Ring einfügen, die beliebig groß sind und keine Haltbarkeitsdauer aufweisen. Es wird angenommen, dass Knoten nicht mehr verwendete Daten eigenständig aus dem Ring löschen. Der letzte Kritikpunkt bezieht sich auf das automatische Partizipieren an der DHT, sobald ein Knoten dem Chord-Ring beigetreten ist. Es gibt also keine Möglichkeit dem Chord-Ring beizutreten, ohne an der DHT zu partizipieren.

Einige dieser Punkte werden in Abschnitt 4.2.4 aufgegriffen und *teilweise* gelöst.

Somit bietet OpenChord eine sehr gute und erweiterbare Grundlage für die Chatplattform. Der nächste Abschnitt bietet eine detaillierte Beschreibung der modifizierten Implementierung von OpenChord.

4.2 Modifizierte Implementierung

In diesem Abschnitt wird die im Rahmen dieser Arbeit modifizierte OpenChord-Implementierung vorgestellt. Es wurde stets darauf geachtet die bestehende Architektur, siehe Abschnitt 4.1, zu wahren, um vorhandene Arbeitsabläufe nicht zu beeinträchtigen. Nachfolgend werden die Modifikationen der drei vorhandenen Schichten (Abschnitte 4.2.1, 4.2.2 und 4.2.3) erläutert. Zwei neu implementierte Zwischenschichten sind im Rahmen dieser Arbeit entstanden. Diese sind die Sicherheitsschicht in Abschnitt 4.2.4 und die Monitoringschicht in Abschnitt 4.2.5. Die Notwendigkeit neben der grundlegenden Verschlüsselung durch WebRTC eine weitere Sicherheitsschicht zu implementieren wird zu Beginn des entsprechenden Abschnitts erläutert.

Im Anschluss erfolgt in Abschnitt 4.3 ein Vergleich der modifizierten und der ursprünglichen OpenChord-Implementierung.

4.2.1 Abstrahierte Kommunikationsschicht

Die abstrahierte Kommunikationsschicht beinhaltet im Wesentlichen drei abstrakte Klassen, welche von der Kommunikationsschicht in Abschnitt 4.2.2 implementiert werden.

Die abstrakte Klasse *Node* repräsentiert einen Knoten im Chord-Ring mit einer *ID*, welche zunächst frei wählbar ist und von implementierenden Klassen gesetzt werden muss. Erbende Klassen müssen des Weiteren verschiedene Operationen, die jeder Knoten können muss, implementieren. Die Operationen umfassen alle Funktionen, die ein Knoten im Chord-Ring benötigt, und von entfernten Knoten aufgerufen werden können. Nachfolgend eine vollständige Aufzählung der Operationen:

- findSuccessor: Finde den Nachfolger eines spezifizierten Knotens.
- notify / notifyAndCopy: Benachrichtige den direkten Nachfolger über die Anwesenheit des lokalen Knotens und erfrage den aktuellen Vorgänger sowie Nachfolger (und die für den lokalen Knoten relevanten Einträge der DHT).
- ping: Einen Knoten nach einem Lebenszeichen fragen.
- insertEntry / removeEntry: Füge einen übergebenen Eintrag, bestehend aus einen Schlüssel und einem Wert, in die DHT des entfernten Knotens ein. Analog für die Löschung eines Eintrags.
- insertReplicas / removeReplicas: Analog zu insertEntry bzw. removeEntry mit einer Menge an Einträgen.
- retrieveEntry: Erfrage den Eintrag eines bestimmten Schlüssels der DHT des entfernten Knotens.

- leavesNetwork: Informiere einen entfernten Knoten über das Ausscheiden des lokalen Knotens und teile ihm außerdem den Nachfolger des lokalen Knotens mit.
- disconnect: Die Verbindung zu dem entfernten Knoten soll geschlossen werden.
- handleMessage: Sende eine Anwendungsschicht Nachricht an einen entfernten Knoten.
- handleCall: Starte einen Audio / Video- Stream mit dem entfernten Knoten.

Die letzten beiden Operationen sind im Rahmen dieser Arbeit entstanden, um der Anwendungsschicht die nötige Funktionalität des Nachrichtenversands und des Streamings zu bieten. Für jede der oben genannten Operationen existiert ein Callback, um den Aufrufer über den Ausgang der Operation zu informieren. Für die Zuordnung von Operationsaufruf und Ausgang der Operation sorgt die neu eingeführte *OperationID*, welche eine eingeleitete Operation eindeutig identifiziert. Außerdem wird die OperationID verwendet, um Daten für das Monitoring zu sammeln. Mehr dazu in Abschnitt 4.2.5.

Endpoint ist eine weitere abstrakte Klasse, welche ein Knoten-Objekt kapselt und einen Endpunkt repräsentiert. Zu den Aufgaben eines Endpunkts zählen folgende Punkte. Zum einen bietet ein Endpunkt anderen Knoten die Möglichkeit sich mit dem gekapselten Knoten zu verbinden. Wenn man so will, so ist der Endpunkt für ankommende Anfragen zuständig. Zum anderen kann ein Endpunkt verschiedene Status annehmen, um die Bearbeitung von ankommenden Nachrichten zu steuern. Ist ein Endpunkt im Zustand *started*, so lauscht er nicht auf eingehende Nachrichten. Eingehende Anfragen werden im Zustand *listening* entgegen genommen und bearbeitet, sofern diese Anfragen den lokalen Speicher der DHT nicht betreffen. Solche Anfragen wären beispielsweise insertEntry oder retrieveEntry. Erst ab dem Zustand *accept entries* werden auch die Anfragen, die den lokalen Speicher der DHT manipulieren, bearbeitet. Bei Statusänderungen werden jeweils registrierte *listener* über den Statuswechsel benachrichtigt. Außerdem überschreiben Endpoint-Implementierungen Methoden, die bei Statusänderungen des Endpunkts gerufen werden. Hierzu zählt das Starten des Listeners bzw. die Reaktion auf den Statuswechsel auf *accept entries*. Zuletzt muss eine Methode überschrieben werden, die alle Verbindungen zu entfernten Knoten, ob eingehend oder ausgehend, schließt. Über eine Factorymethode wird ein Endpunkt erstellt und zurückgegeben. In

dieser Implementierung gibt es nur ein Kommunikationsprotokoll, weshalb hier immer *B2BEndpoint*-Objekte erzeugt und zurückgegeben werden.

Die dritte abstrakte Klasse heißt *Proxy* und erbt von der weiter oben beschriebenen Klasse Node. Proxy-Objekte repräsentieren entfernte Knoten gegenüber einem Knoten. Ein Proxy muss eine Verbindung zu einem entfernten Knoten, der durch einen Endpunkt repräsentiert wird, aufbauen können. Üblicherweise werden Implementierungen von Proxy verwendet, um Anfragen, definiert in der abstrakten Klasse Node, an den entfernten Knoten zu schicken.

Die Implementierung der eben beschriebenen abstrakten Klassen erfolgt in der Kommunikationsschicht, welche der nächste Abschnitt darstellt.

4.2.2 Kommunikationsschicht

Die Kommunikationsschicht implementiert jene abstrakte Klassen, welche in der abstrahierten Kommunikationsschicht in Abschnitt 4.2.1 vorgestellt wurden. Die Kommunikationsschicht enthält lediglich ein Kommunikationsprotokoll, welches eine Kommunikation zwischen Browsern implementiert. Nachfolgend Browser-to-Browser, kurz B2B, genannt. Dieses Kommunikationsprotokoll ist eine Modifizierung des in OpenChord implementierten Socket-Protokolls und wurde im Rahmen dieser Arbeit entwickelt. Diese Schicht wird *Top-down* vorgestellt, sodass zunächst die Implementierungen *B2BEndpoint* und *B2BProxy* der abstrakten Klassen Endpoint bzw. Proxy vorgestellt werden. Anschließend wird die neu entwickelte untere Kommunikationsschicht mit der Nutzung der *PeerJS*-Bibliothek [MB14c] beschrieben.

Obere Kommunikationsschicht

Zu der Hauptaufgabe von B2BEndpoint gehört die Reaktion auf wechselnde Zustände des Endpunktes. Wird der Zustand auf *listening* geändert, so müssen Schritte eingeleitet werden, um auf eingehende Anfragen reagieren zu können. Hierfür wird zunächst das *B2BConnectionManager*-Singleton-Objekt erzeugt bzw. geholt. Anschließend wird die

Kommunikationseinheit mit der eigenen Chord ID initialisiert und die nötigen Schritte, um auf eingehende Daten- und Audio / Video-Verbindungen reagieren zu können, werden eingeleitet.

Meldet der B2BConnectionManager eine neue eingehende Daten-Verbindung, so wird ein *RequestHandler*-Objekt erzeugt. Bei der Erzeugung des RequestHandler-Objekts wird die Verbindung mittels eines *B2BConnection*-Objekts übergeben, um fortan auf Anfragen des entfernten Endpunkts reagieren zu können und um das Ergebnis der bearbeiteten Anfrage an den entfernten Knoten zu senden.

Der typische Ablauf bei einer eingehenden Anfrage sieht wie folgt aus: Das RequestHandler-Objekt erhält die Anfrage von dem gekapselten B2BConnection-Objekt und prüft, ob die Bearbeitung der Anfrage bei dem aktuellen Endpunkt-Status erlaubt ist. Falls dies nicht der Fall sein sollte, so wird diese Anfrage in einer Warteschlange angereiht, welche beim Wechsel des Status abgearbeitet wird. Als nächstes wird der Anfrage-Typ geprüft. Ist die Anfrage vom Typ *shutdown*, so benötigt der entfernte Endpunkt diese Verbindung nicht mehr, was dem B2BConnection-Objekt gemeldet wird. Andernfalls wird die Anfrage an die lokale Node-Implementierung der Serviceschicht weitergegeben, welche die weitere Bearbeitung anstößt. Per Callback erhält das RequestHandler-Objekt die Antwort der Serviceschicht, um diese an den anfragenden entfernten Endpunkt zurückzuschicken. Es ist nicht unüblich, dass die Antwort der Serviceschicht auf sich warten lässt, denn einige Anfragetypen triggern weitere Anfragen, sodass hier mit einer Verzögerung zu rechnen ist und ein Callback benötigt wird. Beispiele für solche Anfragen sind die des Typs *findSuccessor* und werden in der Serviceschicht in Abschnitt 4.2.3 näher beschrieben.

Weiter wird bei einer eingehenden bzw. ausgehenden Audio / Video-Verbindung ein *CallHandler*-Objekt erzeugt und gestartet. Dieses Objekt wird von der Kommunikationsschicht über weitere Events wie ankommender Stream, abgewiesener lokaler Stream und Terminierung des Calls informiert und leitet diese Informationen an die lokale Node-Implementierung der Serviceschicht weiter.

Zuletzt implementiert der B2BEndpoint eine Methode, um alle aktiven Verbindungen zu schließen und sich selbst vom Chord-Ring zu trennen. Hierfür werden alle erzeugten Re-

questHandler und CallHandler-Objekte über die Trennung informiert. Des Weiteren werden auch alle *B2BProxy*-Objekte über eine B2BProxy Klassenmethode informiert, um alle ausgehenden Verbindungen zu schließen. Nachfolgend wird die B2BProxy Klasse näher betrachtet.

B2BProxy ist die Implementierung der abstrakten Klasse Proxy. Wie weiter oben beschrieben, besteht die Hauptaufgabe der Klasseninstanzen Anfragen an entfernte Peers zu verschicken. Initiiert werden die Operationen von der Serviceschicht. Bei allen Operationen bis auf den Audio / Video-Stream wird zunächst geprüft, ob eine Daten-Verbindung zu dem entfernten Knoten besteht. Diese Prüfung besteht aus einem null-Pointer Check der gekapselten B2BConnection. Ist das B2BConnection Attribut null, so wird das B2B-ConnectionManager Singleton-Objekt geholt und eine Verbindung zu dem entfernten Knoten aufgebaut, wodurch ein B2BConnection-Objekt zurückgegeben wird. Um ankommende Antworten des entfernten Endpunktes zu erhalten, registriert sich das B2B-Proxy-Objekt bei dem B2BConnection-Objekt. Außerdem wird ein *PendingQueueChecker* gestartet, welcher im Zuge der modifizierten Kommunikationsschicht benötigt wird.

Der PendingQueueChecker ist ein periodischer Job und prüft alle x ms, ob es Anfragen gibt, die schon länger als y ms auf eine Antwort warten. Ist dies der Fall, so wird die Anfrage erneut versendet. Nach z Versandwiederholungen wird die Serviceschicht über die erfolglose Zustellung benachrichtigt. Die Parameter x, y und z werden in der Konfiguration, zu finden in Abschnitt 6.2, spezifiziert.

Ist die Daten-Verbindung zu dem entfernten Knoten aufgebaut, so wird B2BProxy von der B2BConnection benachrichtigt und es werden aufgestaute Anfragen an diesen Endpunkt versendet. Eingehende Antworten erhält der B2BProxy von der B2BConnection und kann diese dementsprechend an den Callback der Serviceschicht weiterleiten.

Wird eine Call Operation ausgeführt, so wird dies dem B2BConnectionManager mitgeteilt, wer den Call initiiert. Die Serviceschicht wird über das CallHandler-Objekt über Call-Events informiert.

Signalisiert die Serviceschicht einem B2BProxy-Objekt, dass diese ausgehende Verbindung nicht mehr benötigt wird, so wird dem entfernten Endpunkt eine *shutdown* Anfrage geschickt, welche der entfernte Endpunkt mit der Schließung des Datenkanals

beantwortet. Somit schließt sich der Kreis um B2BProxy-Objekte und RequestHandler, welche die Sende- bzw. Empfangseinheiten der oberen Kommunikationsschicht darstellen.

Die erwähnten Klassen B2BConnectionManager und B2BConnection, die Teil der unteren Kommunikationsschicht sind und der oberen Kommunikationsschicht Dienste anbieten, sind Thema des nächsten Abschnitts.

Untere Kommunikationsschicht

Nachfolgend wird die in dieser Arbeit geplante und entwickelte untere Kommunikationsschicht erläutert, welche die Kommunikation zwischen Browsern gewährleistet.

Hauptbestandteil ist die Nutzung der JavaScript PeerJS-Bibliothek [MB14c] via GWT-JSNI. Diese Bibliothek kapselt die WebRTC-Implementierung des Browsers und stellt eine vereinfachte P2P-Programmierschnittstelle (P2P-API) [MB14a] zur Verfügung. Wie in Abschnitt 2.1 bereits erläutert, verläuft jegliche Kommunikationen per WebRTC verschlüsselt. Die aktuelle Version[2] bietet bereits eine volle Unterstützung für aktuelle Versionen der Browser Google Chrome, Mozilla Firefox und Opera an.

Zum Verbindungsaufbau zweier Browser werden *Session Traversal Utitlites for NAT*-Server (engl.: Protokoll zur Umgehung von NAT's, um eingehende Daten zu empfangen) und ein PeerServer [MB14b] verwendet. Diese dienen lediglich der Vermittlung der Verbindungen. Nach dem Verbindungsaufbau erfolgt der Datenaustausch direkt, also P2P. Der verwendete selbstgehostete PeerServer[3] verwendet SSL mit einem Zertifikat, um die Kommunikation zwischen einem PeerJS-Knoten und dem PeerServer abzusichern.

Die PeerJS-API bietet die Möglichkeit einen neuen Knoten unter der Angabe einer ID, einem PeerServer, sowie weiterer Optionen, wie die Nutzung einer sicheren Leitung über SSL, zu erschaffen. Das entstandene *peer*-Objekt kann nun auf eingehende Events wie

[2] Version 0.3.8: Stand 31.05.2014
[3] Version 0.2.5: Stand 31.05.2014

eine Media- oder Datenverbindung hören oder selbst Verbindungen aufbauen. So können ausgehende Datenverbindungen per *DataConnection connect(ID)* Methode aufgebaut werden. Analog zu den ausgehenden Datenverbindungen können ausgehende Mediaverbindungen mit der *MediaConnection call(ID,localStream)* Methode aufgebaut werden. Über die DataConnection und MediaConnection-Objekte können dann Daten respektive Mediaströme versendet und empfangen, sowie deren Schließung vorgenommen werden. Wird eine Verbindung zu einem entfernten Knoten geschlossen, so bekommt das peer-Objekt dies per Event mitgeteilt.

Die Einbindung der PeerJS-Bibliothek erfolgt in der unteren Kommunikationsschicht, welche aus den folgenden vier Komponenten besteht:

- Der B2BConnectionManager kapselt das PeerJS peer-Objekt und verwaltet bzw. erschafft neue B2BConnection-Objekte bei neuen eingehenden bzw. ausgehenden Verbindungen.

- Die Klasse *DataConnection* kapselt das PeerJS-DataConnection-Objekt und bietet Methoden, um Daten verschlüsselt und signiert zu versenden und zu empfangen.

- Das PeerJS-MediaConnection-Objekt kapselt die *MediaConnection* Klasse und bietet Methoden, um Audio- und Videoanrufe aufzubauen und zu empfangen.

- Die B2BConnection Klasse, welche bereits in der oberen Kommunikationsschicht erwähnt wurde, repräsentiert eine Verbindung zwischen zwei PeerJS-Knoten. Es erfolgt eine Kapselung der verschiedenen Data- und MediaConnections und sorgt für die Auslieferung eingehender Ereignisse an registrierte Callbacks.

Das Kernstück der unteren Kommunikationsschicht ist der B2BConnectionManager, welcher als Singleton implementiert wurde. Die Hauptaufgaben des B2BConnectionManagers sind die Initialisierung eines PeerJS *peer*-Objekts und das hören auf peer-Events. Darüber hinaus initiiert er den Daten- bzw. Media-Verbindungsaufbau zu einem entfernten Knoten unter der Angabe seiner PeerJS ID (= Chord ID).

Behandelte eingehende peer-Events sind die folgenden:

- Connection: Dieses Event informiert über eine eingehende Daten Verbindung mitsamt einem PeerJS-DataConnection-Objekt. Die Reaktion darauf sieht wie folgt aus. Es wird zunächst die DataConnection in einem B2BConnection-Objekt gesichert bzw. ein neues B2BConnection-Objekt für den entfernten Peer mitsamt der DataConnection angelegt.

- Call: Dieses Event zeigt einen eingehenden AV-Ruf mitsamt MediaConnection-Objekt an. In diesem Fall wird dem zugehörigen B2BConnection-Objekt das MediaConnection-Objekt übergeben.

- Close: Wird ausgelöst nachdem das eigene PeerJS peer-Objekt zerstört wurde. Hier wird das B2BConnectionManager-Objekt aufgeräumt.

- Error: Dies sind meist Error Events, die vom PeerJS-Konstruktor geworfen werden. Zum Beispiel wenn die übergebene PeerJS ID vergeben ist oder illegale Zeichen enthält. Als Reaktion auf ein Error Event bleibt nur noch das Aufräumen des B2BConnectionManagers und die Bekanntgabe an den lokalen Endpunkt.

Wird dem B2BConnectionManager die Aufgabe übertragen eine ausgehende Datenverbindung aufzubauen, so wird die PeerJS peer-Methode *connect*, unter Angabe der Ziel ID und mit der Option auf eine zuverlässigen Datenverbindung, gerufen. Anschließend wird ein B2BConnection-Objekt erzeugt und diesem die geschaffene Datenverbindung übergeben. Für die Events einer DataConnection ist das B2BConnection-Objekt bzw. die oben genannte DataConnection Klasse zuständig.

Soll ein Call aufgebaut bzw. ein laufender Call gestoppt werden, so bietet der B2BConnectionManager die dazugehörigen Methoden. Hierbei wird beim Aufbau der Verbindung der registrierte Endpunkt informiert und außerdem wird dem zugehörigen B2BConnection-Objekt die Audio / Video-Call-Anfrage weitergegeben. Beim Abbau der Verbindung wird das zugehörige B2BConnection-Objekt aus dem B2BConnection-Pool gesucht und diese Information weitergegeben.

DataConnection-Objekte, die zweiten Komponente der unteren Kommunkationsschicht, kümmern sich um eine verschlüsselte Datenverbindung zu einem entfernten Knoten. Gekapselt wird das PeerJS-DataConnection-Objekt und dessen Methoden, die das Sen-

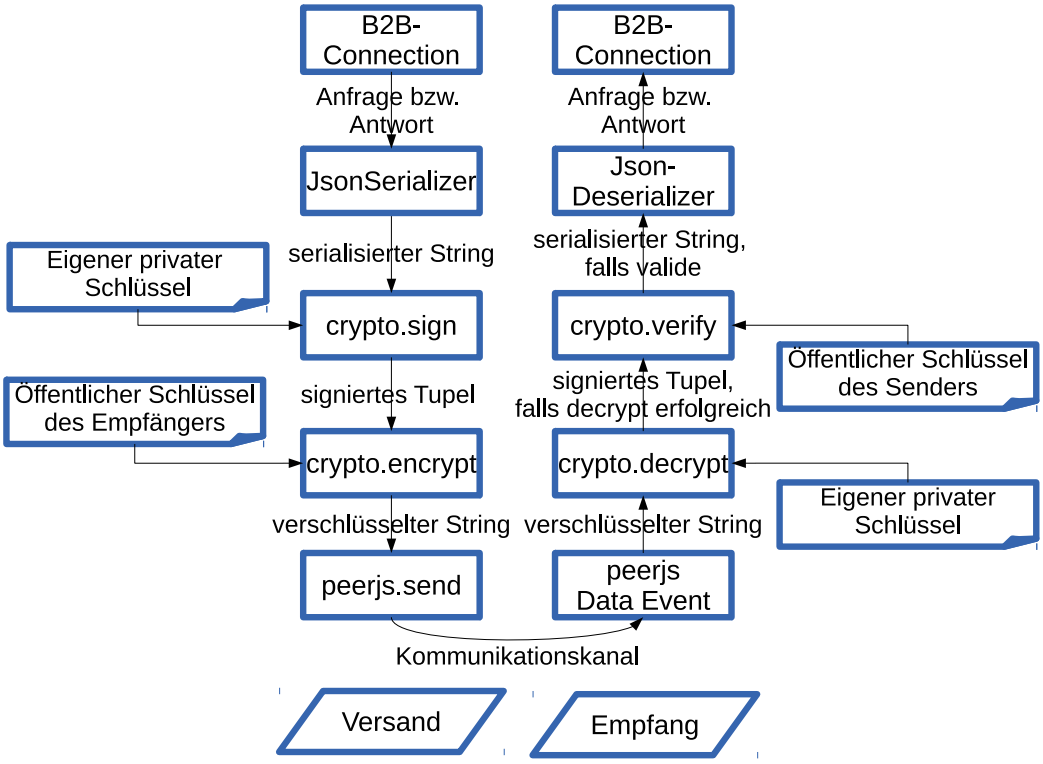


Abbildung 4.2: Nachrichtenversand und -empfang.

den, Empfangen und Schließen der Datenverbindung anbieten. Jedes DataConnection-Objekt wird einer B2BConnection zugeordnet, welchem eingehende Nachrichten mitgeteilt werden. Wird vom B2BConnection-Objekt die startListener Methode gerufen, so wird auf eingehende PeerJS-Events reagiert:

- Data Event: Eingehende Nachricht des entfernten Knotens. Die Nachricht liegt zunächst als verschlüsselter und signierter String vor und muss entschlüsselt, verifiziert und deserialisiert werden.

- Close / Error Event: Reagiere mit dem Aufräumen des DataConnection-Objekts und benachrichtige die übergeordnete B2BConnection.

Bevor der Sende- und Empfangsprozess erläutert wird, muss zunächst die (De-)Serialisierung der Daten sichergestellt werden. Wie in Abschnitt 2.3.1 geschildert, bietet GWT

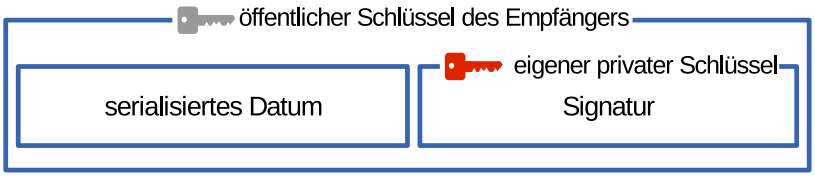


Abbildung 4.3: Aufbau einer Nachricht.

keine Möglichkeit Daten bei der Kommunikation zwischen Peers zu serialisieren bzw. deserialisieren. Die GWT-Bibliothek JsonSerializer [Kfu11] schließt diese Lücke. Durch die Verwendung von GWT's *deferred Binding*, welches eine Antwort auf Java's Reflection ist, werden Klasseninstanzen, deren Klasse das JsonSerializable Interface implementiert und einigen Anforderungen genügen, die Möglichkeit geboten serialisiert und deserialisiert zu werden. Hierbei wird zur Kompilierungszeit für jede Klasse, die das JsonSerializable Interface implementiert, eine (De-)Serialisierungsklasse generiert.

Nachfolgend wird der in Abbildung 4.2 ausgeführte Sende- und Empfangsprozess dargelegt. Bei dem Versand wird dem zuständigen DataConnection-Objekt die Nachricht übergeben. Diese Nachricht wird zunächst vom JsonSerializer serialisiert, wodurch ein Datenstring herauskommt. Anschließend erfolgt die Signierung und Verschlüsselung des Datenstrings. Dem DataConnection-Objekt liegt die *EccCrypto*-Instanz vor, welche unter anderem Methoden für die Verschlüsselung, Entschlüsselung, Signierung und Verifizierung einer Signatur zur Verfügung stellt. Die Kommunikationsschicht kann an dieser Stelle annehmen, dass die Chord ID gleich der PeerJS ID ist. Des Weiteren wird die Chord ID so gewählt, dass sie identisch mit dem öffentlichen Schlüssel der Sicherheitsschicht ist. Dies hat wiederum zur Folge, dass die asymmetrische Verschlüsselung mit der PeerJS ID des Empfängers, sprich dem öffentlichen Schlüssel, erfolgt. Nachdem die übergebene Nachricht serialisiert wurde, erfolgt die Signierung. Hier wird mittels des eigenen privaten Schlüssels der serialisierte String signiert. Heraus kommt ein Tupel mit dem serialisierten String und der Signatur. Anschließend wird dieses Tupel mit Hilfe des öffentlichen Schlüssels des Empfängers, sprich seiner PeerJS ID, verschlüsselt. Abbildung 4.3 veranschaulicht die resultierende Nachricht. Zum Schluss wird die send-

Methode der gekapselten DataConnection-Instanz der PeerJS-Bibliothek mit dem verschlüsselten String als Parameter gerufen.

Die Empfängerseite erhält zunächst das Data Event mitsamt des verschlüsselten Strings. Mithilfe des eigenen privaten Schlüssels wird dieser entschlüsselt. Falls die Daten korrupt sind, d.h. der lokale Knoten ist nicht der richtige Adressat, so wird eine Exception geworfen und dem übergeordneten B2BConnection gemeldet. Das Tupel aus Datenstring und Signatur wird anschließend mit dem öffentlichen Schlüssel des Senders verifiziert. Ist die Signatur korrupt, so wird auch dies der B2BConnection gemeldet. Um nun wie in Abschnitt 2.3.2 erläutert, sicherzugehen, dass die serialisierten JSON Daten keinen ausführbaren und potentiell bösen Code enthalten, erfolgen aufrufe an das *JsonUtils* Paket von GWT. Hierbei prüft die Klassenmethode *safeToEval* mit vorherigen Aufruf von *escapeValue*, ob der Aufbau der JSON Zeichenkette dem RFC [Cro06] entspricht und somit keinen ausführbaren Code enthält. Erst dann erfolgt die abschließende Deserialisierung des Datenstrings, wodurch wir entweder eine Anfrage oder eine Antwort erhalten. In beiden Fällen wird diese der B2BConnection für die weitere Bearbeitung übergeben.

Die dritte Komponente der unteren Kommunikationsschicht bilden die Objekte der MediaConnection Klasse. Sie sind für das Audio / Video-Streaming verantwortlich. Somit bilden diese Objekte den Wrapper für PeerJS-MediaConnection-Objekte eines entfernten Knotens. Ähnlich wie die DataConnection Klasse wird die übergeordnete B2BConnection-Instanz gesichert, um nachfolgende erfolgreich ankommende Events diesem mitzuteilen. Zu den Aufgaben der MediaConnection Klasse gehören der Aufbau, die Beantwortung und das Stoppen des AV-Streams. Wird ein Verbindungsaufbau zu dem entfernten Knoten gewünscht, so wird zunächst der *stream access* des lokalen Benutzers erfragt. Hierbei wird browserabhängig der Zugriff auf Audio- und Video-Geräte erfragt. Wird der Medienzugriff vom Benutzer untersagt, so ist *kein* Call mit PeerJS möglich, da bei einem Rufaufbau ein gültiger Stream notwendig ist. Andernfalls wird der Call aufgebaut und auf MediaConnection Events gehört:

- Das *stream* Event überliefert einen Stream-Blob des entfernten Knotens. Mittels der statischen $window.URL.createObjectURL(blob)$ JavaScript Methode wird aus dem Stream-Blob ein DOMString gebaut, welcher eine URL enthält, der den Stream-

Blob repräsentiert. Diese Objekt-URL kann in einem HTML5 *video*-Element angezeigt werden. Mehr hierzu im GUI-Kapitel 6.1.

- Ein eingehendes *close* Event tritt ein, wenn der entfernte oder lokale Knoten die unterliegende PeerJS-MediaConnection geschlossen hat. Als Reaktion wird das Objekt aufgeräumt und der B2BConnection gemeldet.

Das Beantworten eines eingehenden Rufes erfolgt ähnlich zu dem Rufaufbau. Zunächst wird der Medienzugriff des Benutzers erfragt. Da dies ein eingehender Ruf ist, ist es zulässig keinen Medienzugriff zu gewähren, wodurch ein einseitiges Streaming erfolgt. Analog zum Rufaufbau wird anschließend auf MediaConnection Events gehört.

Die Rufterminierung erfolgt durch einen *close()* des unterliegenden PeerJS-MediaConnection-Objekts. Dieses triggert das close Event auf beiden Seiten.

Nachfolgend wird die vierte Komponente der unteren Kommunkationsschicht vorgestellt: die B2BConnection Klasse. Ein B2BConnection-Objekt repräsentiert eine Verbindung zwischen einem entfernten und dem lokalen Knoten auf einer höheren Ebene. Ein Objekt kapselt alle existierenden Verbindungen, Daten oder Media-Objekte, zwischen diesen Knoten. Außerdem bietet die B2BConnection Klasse eine Art Vermittlungsschicht für eingehende Anfragen bzw. Antworten. Diese werden an registrierte Callbacks weitergeleitet; hier gehen eingehende Anfragen an den registrierten RequestHandler und eingehende Antworten an den registrierten B2BProxy. Sind die Objekte der oberen Kommunikationsschicht nicht mehr an eingehenden Nachrichten interessiert, so können diese sich deregistrieren. Sind keine registrierten Callbacks mehr vorhanden, so wird dies dem B2BConnectionManager mitgeteilt, welcher das B2BConnection-Objekt aufräumt. Weiter bietet es RequestHandler- und B2BProxy-Objekten die Möglichkeit Nachrichten zu versenden. Abschließend werden registrierte B2BProxy-Objekte über einen erfolgreichen DataConnection Aufbau informiert, wodurch diesen das Senden von Nachrichten ermöglicht wird.

Im Kontext der B2BConnection-Objekte werden unterliegende DataConnection-Objekte *pseudo unidirektional* verwendet. Pseudo, da für eingehende Anfragen auch ausgehende Antworten über dieselbe Datenverbindung geschickt werden. Jedoch werden eingehende und ausgehende Anfragen nicht über dasselbe DataConnection-Objekt verschickt. Wes-

halb dies nicht über dasselbe DataConnection-Objekt geschickt wird, wird folgend erläutert.

In der Chord-Implementierung OpenChord werden eingehende und ausgehende Anfragen über unterschiedliche Java Sockets gesendet. Der Versuch für eine Verbindung zwischen einem Knotenpaar eine Datenverbindungen, d.h. ein einziges peerjs DataConnection-Objekt, zu nutzen enthielt Fallstricke. So könnte man an eine Verbindung, die man als lokal unbrauchbar eingestuft hat, d.h. keine zu beantwortenden Anfragen vorhanden und die Serviceschicht meldet, dass dieser Knoten nicht mehr gebraucht wird, so würde man eine Schließung der Datenverbindung veranlassen, indem man das peerjs-Objekt DataConnection close() ruft. Leider benötigt dieses close-event relativ lange, um bei dem entfernten Knoten anzukommen, wohingegen das Event am lokalen Knoten sofort erscheint. In Tests benötigte diese Nachricht etwa fünf Sekunden, wohingegen alle anderen Nachrichten innerhalb weniger Millisekunden ankamen. Dies hat zur Folge, dass man weitere Zustände benötigt, um solch einen Schließungsvorgang zu beschreiben. Würde nun innerhalb dieser fünf Sekunden der entfernte Knoten eine weitere Anfrage absetzen, so gerät man in einen undefinierten Zustand. Der entfernte Knoten würde in wenigen Sekunden das close event erhalten und könnte davon ausgehen, dass der lokale Knoten abgestürzt ist und beendet somit die Verbindung. Da der lokale Knoten die Anfrage erhalten hat, versucht dieser die Anfrage zu beantworten und könnte dies unter Umständen sogar noch bevor das zweite close-event des entfernten Knoten ankommt. Hierdurch könnten sogar Schleifen entstehen, welche es zu verhindern gilt.

Um diesem Phänomen zu entgehen, wurden in dieser Implementierung, wie auch in der OpenChord-Implementierung, Datenverbindungen pseudo unidirektional gehandhabt, sodass eine ausgehende Anfrage und die darauf antwortende Nachricht über einen Kanal geschickt werden. Möchte der entfernte Knoten nun auch eine Anfrage an den lokalen Knoten schicken, so muss dieser zunächst eine weitere Verbindung aufbauen, die aus Sicht des lokalen Knotens eingehend ist und worüber nur Antworten verschickt werden. Eine Schließung einer Verbindung wird nun durch den anfragenden Knoten (Anfragen-Sender: B2BProxy-Objekte) mit dem Versand einer *shutdown* Nachricht eingeleitet, wodurch der Empfänger der *shutdown* Nachricht die Schließung des Kanals veranlasst. Unerwartet eingetroffene close Events lassen nun auf einen Absturz des entfernten Knotens schließen.

Die Kommunikationsschicht ist somit vollständig. Die obere und untere Schicht bieten durch die Implementierung der abstrahierten Kommunikationsschicht einen umfassenden Nachrichtenversand- sowie Empfang an. Weiter ist der Aufbau von Calls mittels WebRTC möglich. Alle Kommunikationen sind durch das WebRTC und die Chordeigene Sicherheitsschicht, die Teil dieser Arbeit ist, abgesichert. Bevor die Sicherheitsschicht in Abschnitt 4.2.4 detailliert vorgestellt wird, folgt die Serviceschicht.

4.2.3 Serviceschicht

In diesem Abschnitt wird die modifizierte *Chord Logic* der Abbildung 4.1 vorgestellt.

In der grundlegenden Form von OpenChord beinhaltet die Serviceschicht die Implementierung des Chord Protokolls, vorgestellt in [SMK+01]. Wie in Abschnitt 4.1 beschrieben, gehören hierzu die *create*, *join* und *leave* Operationen, womit das Eintreten, Erstellen und Verlassen eines Chord-Rings möglich ist. Die create sowie join Prozesse werden nachfolgend erläutert. Weiter sind DHT-Manipulationen mithilfe der *insert* und *remove* Operationen möglich. Diese stellen Mechanismen bereit, um Objekte in den Chord-Ring einzufügen und wieder zu entfernen. Die *retrieve* Operation hingegen beschafft das Objekt, welches sich an einem spezifizierten Platz (= Chord ID) befindet. Das lokale Repositorium der DHT wird von der *References* Klasse verwaltet, welche darüber hinaus die Fingertabelle *finger table* und die Nachfolgerliste *successor list* führt. Die im Chord Protokoll beschriebenen Aufrechterhaltungsprozesse werden periodisch gerufen, um die Ringstruktur zu wahren. Diese sind die folgenden:

- *CheckPredecessor*: Schicke einen Ping an den registrierten Vorgänger. Ist dieser nicht erreichbar, so lösche diese Referenz und entferne diesen als unseren Vorgänger.

- *FixFinger*: Repariere **einen** Eintrag der Fingertabelle. Hierfür wird eine Zufallszahl α im Intervall $[1; \text{length}(\text{ownChord ID})]$ geworfen und anschließend wird eine findSuccessor Operation für die ID $\text{ownChord ID} + 2^{\alpha}$ ausgeführt. Die Antwort liefert uns den zuständigen Knoten für diese ID, welcher in die Fingertabelle eingepflegt wird.

- *Stabilize*: Führe eine notify Operation auf den registrierten direkten Nachfolger aus, um diesem die eigene Anwesenheit zu signalisieren. Die Antwort enthält den Vorgänger und Nachfolger des direkten Nachfolgers. Ist der lokale Knoten nicht als Vorgänger eingetragen, so wird eine notifyAndCopy Operation initiiert, um Einträge für den aktuellen Zuständigkeitsbereich zu erhalten. Alle anderen Nachfolger des direkten Nachfolgers werden der References Klasse übergeben, welche die Nachfolgerliste und Fingertabelle aktualisiert.

Abschließend wird in der grundlegenden Form eine Replikation der Daten auf die Nachfolgerliste getätigt. So wird nach einem Ausfall eines Knotens der neu zuständige Knoten, sprich der Nachfolger, die Replikate seines abgestürzten Vorgängers halten und somit dem Verlust von Daten entgegenwirken. Für die Replikation der Daten ist immer der für die Chord ID zuständige Knoten verantwortlich.

Die Serviceschicht der modifizierten Implementierung enthält nun weitere Funktionen, die das Ziel haben, der darüberliegenden Anwendung eine sichere Chatplattform zu bieten. Streng genommen sorgt die Sicherheitsschicht (siehe Abschnitt 4.2.4) gepaart mit der Verschlüsselung durch WebRTC für die Sicherheit der Serviceschicht. Nachfolgend angesprochene Features sind im Rahmen dieser Arbeit ausgearbeitet und Implementiert worden.

Für eine Chatplattform ist zunächst die Einführung von Identitäten notwendig. Durch die Eingabe eines Benutzernamens und eines Passworts muss eine Identität geschaffen werden, die eindeutig ist und durch andere Identitäten im Chord-Ring aufgefunden werden kann. Zu diesem Zweck findet eine Abbildung eines Benutzernamen / Passwort Tupels auf eine Chord ID statt. Gleichzeitig ist die Chord ID der öffentliche Schlüssel des Kryptomoduls der Identität. Im Abschnitt 4.2.4 wird die Generierung des öffentlichen Schlüssels unter der Angabe einer Identität näher beschrieben. Zunächst bleibt anzunehmen, dass eine Identität immer auf die gleiche Chord ID abgebildet wird.

Zertifikatinfrastruktur

Listing 4.1: Methodenköpfe der abstrakten Klasse CertificateManagement

```
public void insertCertificate(Certificate cert, CertificateCallback callback);
```

```
public void getCertificate(String username, CertificateCallback callback );
public void getCertificate(ID compressedPublicKey, CertificateCallback callback );
public void modifyCertificate(Certificate cert, CertificateCallback callback );
public void deleteCertificate(Certificate cert, CertificateCallback callback );
```

Da Chord ID's im Chord-Ring eindeutig sind, jedoch verschiedene Identitäten den gleichen Benutzernamen verwenden können, werden *Zertifikate* angefertigt. Diese Zertifikate bilden eine Chord ID auf eine Identität ab und beinhalten neben dem Benutzernamen und seinem öffentlichen Schlüssel, Informationen wie Vor-/Nachnamen, Alter, Erstellungsdatum des Zertifikats und die Signatur. Die Validität eines Zertifikats kann geprüft werden indem anhand des darin enthaltenen öffentlichen Schlüssels die Signatur verifiziert wird. Methoden hierfür stellt die Sicherheitsschicht bereit. Für die Zertifikatsverwaltung im Chord-Ring ist das *CertificateManagement* zuständig, welches nachfolgend erläutert wird. Ein Auszug mit allen Methodenköpfen des CertificateManagement ist in Listing 4.1 zu finden. Diese Entität sorgt für die Zertifikatinfrastruktur und bietet Möglichkeiten Zertifikate im Chord-Ring zu sichern, zu suchen und das Zertifikat des Knotens zu modifizieren oder zu löschen bzw. zu deaktivieren. Es sei angemerkt, dass jeder Knoten selbst für das Anlegen und die Pflege des eigenen Zertifikats zuständig ist. Es liegt nämlich im eigenen Interesse von anderen Identitäten erreicht zu werden. Einen Vorteil der Zertifikatinfrastruktur besteht in der Möglichkeit nach erfolgreicher Suche eines Benutzers mit ihm Kontakt aufzunehmen, da der öffentliche Schlüssel gerade der Chord ID entspricht.

Um nun im Chord-Ring nach Benutzern bzw. deren öffentlichen Schlüssel suchen zu können, legen Benutzer ihr angefertigtes Zertifikat an zwei Stellen im Chord-Ring ab.

- hash(publicKey): Das erste Zertifikat wird an der Chord ID des gehashten öffentlichen Schlüssels abgelegt. Dadurch erhält man eine schnelle Suche anhand des öffentlichen Schlüssels eines Benutzers.

- hash*(username.toLowerCase()): Das zweite Zertifikat wird in einer *Benutzernamen-Liste* abgelegt. Diese Liste wird durch die Verkettung von Hashwerten im Chord-Ring realisiert. So beginnt die Liste bei hash(benutzername) und der nächste Eintrag der Liste ist bei hash(hash(benutzername)) zu finden. Die Verkettung setzt sich fort und endet, wenn der Wert der DHT an dieser Stelle *null*, d.h. leer,

ist. So erreichen wir eine einfach verkettete Liste mit DHT Einträgen. Die Suche nach einem Benutzernamen durchläuft diese Liste. Um eine Suche unabhängig von Groß-Kleinschreibung anzubieten, werden Benutzernamen immer in Kleinbuchstaben umgewandelt.

Beim Einfügen eines Zertifikats in die logische Benutzernamen-Liste wird diese zunächst komplett durchlaufen, d.h. bis $\text{hash}^n(\text{Benutzername})$ einen null Wert für ein $n \geq 1$ zurückliefert. An dieser Stelle wird das Einfügen des Zertifikats eingeleitet. Schlägt dies fehl, da ein anderer Nutzer an dieser Stelle sein Zertifikat eingefügt hat, so wird $\text{hash}^{n+1}(\text{Benutzername})$ geprüft und versucht das Zertifikat dort erneut einzufügen. Dies resultiert in einem Wettlauf, wo der Nutzer, dessen Anfrage zuerst beim zuständigen Knoten ankommt, diesen Platz für sein Zertifikat beansprucht.

Folgende Suchstrategien des Zertifikatmanagements für die Suche nach einem gegebenen Benutzernamen wurden theoretisch evaluiert:

1. *Naiv*: Die Benutzernamen-Liste wird Schritt-für-Schritt abgearbeitet mit einer Pipelinelänge von eins, d.h. es wird immer auf eine Antwort gewartet bevor die nächste Anfrage initiiert wird. Sobald das Ende erreicht wird, d.h. die Antwort null ist, stoppt der Suchlauf.

2. *Slow-start*: Ähnlich zum TCP Slow-Start sollen für jede Antwort ungleich null zwei weitere Anfragen initiiert werden. Wurde zuvor das Ende der Liste erreicht bzw. *überschritten* so wird lediglich auf die Antwort bereits initiierter Operationen gewartet.

3. *Modifizierter Slow-start*: Hier wurde der oben genannte Slow-Start modifiziert. Es soll mit einer vordefinierten Anzahl an retrieve Operationen gestartet werden und bei einer Antwort ungleich null sollen zwei bzw. eine weitere Anfrage(n), abhängig von der Anzahl an ausstehenden Anfragen, gestartet werden. Wurde die vordefinierte maximale Anzahl ausstehender Anfragen erreicht, so sende lediglich eine Anfrage, sonst zwei.

Der naive Ansatz ist zwar sehr langsam im Durchlauf längerer Listen, jedoch produziert dieser den geringsten Overhead. Der Slow-Start Ansatz hingegen ist vergleichsweise

schnell im Durchlaufen längerer Listen, produziert jedoch im schlimmsten Fall doppelt so viel Traffic wie der naive Ansatz. Außerdem werden kürzere Listen mit etwa fünf Einträgen nicht signifikant schneller durchlaufen als mit dem naiven Ansatz, sofern man initial nicht mehr als eine Operation initiiert. Abhilfe schafft hier der modifizierte Slow-Start. Durch eine vordefinierte Anzahl an initiierter Operationen werden kürzere Listen schnell durchlaufen und längere, abhängig von der definierten maximalen Anzahl ausstehender Anfragen, ähnlich schnell wie Slow-Start. Der verursachte Overhead ist bei einer optimalen Parameterwahl erwartungsgemäß gering.

Wird die Löschung einer Identität gewünscht, so sollten die hinterlegten Zertifikate idealerweise deaktiviert bzw. gelöscht werden. Hierzu hält das Zertifikatmanagement die Methode *deleteCertificate* aus dem Listing 4.1 bereit. Anders als andere Chord DHT-Einträge werden Zertifikate **nicht** gelöscht. Eine Löschung der DHT-Einträge hätte bei der Benutzername-Liste beispielsweise zur Folge, dass Lücken in dieser Liste fälschlicherweise als Ende der Liste angesehen werden und somit, je nach implementiertem Algorithmus, die Suchanfrage terminiert. Bei einer Löschung einer Identität wird lediglich das Zertifikat auf *inaktiv* gesetzt, sodass diese Zertifikate beispielsweise bei der Suche nicht an die Anwendungsschicht weitergegeben werden. Hierzu wird das eigene Zertifikat mit dem inaktiv Flag versehen und in die DHT eingefügt bzw. überschrieben.

Letztendlich wird der Anwendungsschicht, welche ein Chord-Objekt instanziiert, die Möglichkeit geboten nach Benutzernamen und öffentlichen Schlüsseln zu suchen, sowie Kontaktdetails des eigenen Zertifikats zu modifizieren und seine eigene Identität zu deaktivieren. Details zu den Methodenköpfen können dem Listing 4.1 entnommen werden.

Bereitstellung Anwendungsschicht: Nachrichten und AV-Streaming

Listing 4.2: Methodenköpfe bzw. Interfaces der abstrakten Klasse Chord, die zum Senden und Empfangen von Nachrichten notwendig sind.

```
abstract void sendMsg(ID targetID, String msg, MessageSupport callback);

interface MessageSupport{
    [...]
    void sentMsg(ID toPeer, String msg, MessageSupportFailure failure);
}
```

```
interface MessageReceiver {
    void recvMsg(ID fromPeer, String msg);
}
```

Die nächste wichtige Funktionalität für eine Chatplattform ist der Versand und Empfang von Anwendungsschicht Nachrichten. So soll es möglich sein eine Nachricht an eine angegebene Chord ID zu senden.

Eine Implementierung dieser Funktionalität ist erfolgt und die Methodenköpfe für das Senden einer Nachricht kann dem Listing 4.2 entnommen werden. Diese Nachrichten enthalten einen Absender in Form einer Chord ID und einen Inhalt in Form einer Zeichenkette. Über einen Callback wird der Empfangsstatus der Nachricht gemeldet.

Durch das Zertifikatmanagement erhält man die Chord ID einer gewünschten Identität, weshalb der Sendevorgang *kein* Routing im Chord-Ring durchläuft. Stattdessen wird zunächst anhand der Chord ID ein Proxy-Objekt instanziiert. Anschließend erfolgt eine *handleMessage* Operation auf das Proxy-Objekt, wodurch die Anwendungsschicht der Gegenseite die Nachricht über seine Kommunikationsschicht erhält. Ist die Gegenseite nicht im Chord-Ring angemeldet, so wird dies durch die Kommunikationsschicht signalisiert. Abschließend erfolgt eine Empfangsbenachrichtigung inklusive aufgetretener Fehler an den initial übergebenen MessageSupport Callback.

Um Anwendungsschicht Nachrichten von Chord Teilnehmern empfangen zu können, muss die Anwendungsschicht bei der Instanziierung eines Chord-Objekts einen *MessageReceiver* registrieren. Die MessageReceiver Schnittstelle ist in Listing 4.2 zu finden. Über diese Schnittstelle erhält die Anwendungsschicht eingehende Nachrichten der Chord Serviceschicht.

Listing 4.3: Methodenköpfe bzw. Interfaces der abstrakten Klasse Chord, die zum Starten bzw. Stoppen eines AV-Streams notwendig sind.

```
abstract void startAVStream(ID targetID);
abstract void stopAVStream(ID targetID);
public interface AVSupport{
    [...]
    void incomingStream(ID fromPeer, String othersMediaStream, String myMediaStream);
    void stopStreaming(ID peer, AVSupportFailure failure);
}
```

Neben dem Empfang und Versand von Nachrichten benötigt die Chatplattform eine Möglichkeit Audio / Video-Streams zu starten und zu beenden.

Das Listing 4.3 enthält alle relevanten Methoden der Implementierung dieser Funktionalität. Ein AVSupport Interface wird bei der Instanziierung einer Chord-Instanz benötigt, worüber ankommende Stream-Objekte ankommen und der Wunsch des Stoppens der Gegenseite empfangen wird. Ähnlich zum Versand und Empfang von Anwendungsschicht Nachrichten bedient man sich der Möglichkeit Chord ID's einer gewünschten Identität durch das Zertifikatmanagement zu erlangen. Somit wird beim Starten und Stoppen des AV-Streams zunächst ein Proxy-Objekt anhand der übergebenen *targetID* instanziiert. Anschließend wird auf der soeben erstellten Instanz die *handleCall* Methode gerufen. Die Kommunikationsschicht baut dann, je nach übergebenem Parameter der handleCall Methode, einen Call auf oder ab. Wurde ein Callaufbau gewünscht, so fragt der Browser zunächst den Nutzer, ob auf die lokalen Mediastreams zugegriffen werden darf. Je nach Browser kann man hier Audio- und Video-Quellen getrennt angeben und diese Einstellung für weitere Mediastreamanfragen speichern lassen. Wird diese Anfrage zurückgewiesen, so wird der ausgehende Call unterbrochen. Die Gegenseite erhält nun über die Kommunikationsschicht einen eingehenden Call. Hat diese Zugriff auf seinen Mediastream autorisiert, so erhält der lokale Knoten über das AVSupport Interface per incomingStream den Mediastream des Gegenüber, sowie den Eigenen.

Auf Anwendungsschichtebene sollte eine vorherige Abfrage, ob ein AV-Call aufgebaut werden soll, erfolgen. Dies hätte den Vorteil den Nutzer nicht durch ein plötzlich aufpoppendes Dialogfeld für der lokalen Medienzugriff zu verunsichern bzw. überraschen. Die Beschreibung der Implementierung des AVStream-Moduls der Anwendungsschicht erfolgt in Abschnitt 5.2.4.

Nachdem in diesem Unterabschnitt zwei wichtige Features, nämlich die Anwendungsnachrichten Verarbeitung und AV-Unterstützung, vorgestellt wurden, folgt nun die Be-

schreibung der Anpassungen der Chordmethoden, um den neuen Voraussetzungen gerecht zu werden.

Angepasste Chord Methoden: create, join, insert, retrieve und remove

Um der Chatplattform Möglichkeiten zu bieten sich einloggen bzw. registrieren zu können, wurde der *create* und *join* Prozess von OpenChord angepasst. Es soll genau ein öffentlicher Chord-Ring geschaffen werden, welchem Chatnutzer beitreten können. Ziel war es also einen Login zu schaffen, ohne einen Eintrittsknoten bzw. dessen Chord ID kennen zu müssen, um dem öffentlichen Chord-Ring beizutreten. Erreicht wurde dies durch die Speicherung von Chord ID's von Knoten, die dem öffentlichen Netzwerk eingetreten sind, seitens des Webservers. Hier sei erwähnt, dass **nur** Chord ID's an den Webserver transferiert werden. Es werden keinerlei extra Metainformationen an den Webserver übermittelt.

Außerdem soll es möglich sein einen privaten Chord-Ring zu schaffen, welchem nur Knoten beitreten können, die die Chord ID eines Teilnehmers kennen bzw. diese außerhalb dieser Software beziehen. Teilnehmer, die einen privaten Chord-Ring erstellen oder in einen privaten Chord-Ring eintreten, transferieren ihre Chord ID nicht zu dem Webserver. Die gesicherten Eintritt-ID's sind lediglich für den Eintritt in den öffentlichen Chord-Ring bestimmt.

Infolgedessen erschafft der modifizierte create Prozess einen neuen privaten Chord-Ring. Der modifizierte join Prozess teilt sich je nach angegebenen Parameter in vier Varianten auf:

- Registrierung eines neuen Benutzers und eintreten in den öffentlichen Chord-Ring.
- Registrierung eines neuen Benutzers und eintreten in einen privaten Chord-Ring unter Angabe einer Eintrittsknoten-ID.
- Einloggen als existierender Benutzer in den öffentliche Chord-Ring.
- Einloggen als existierender Benutzer in einen privaten Chord-Ring unter Angabe einer Eintrittsknoten-ID.

Listing 4.4: Create Methodenkopf / Callback der abstrakten Chord Klasse.

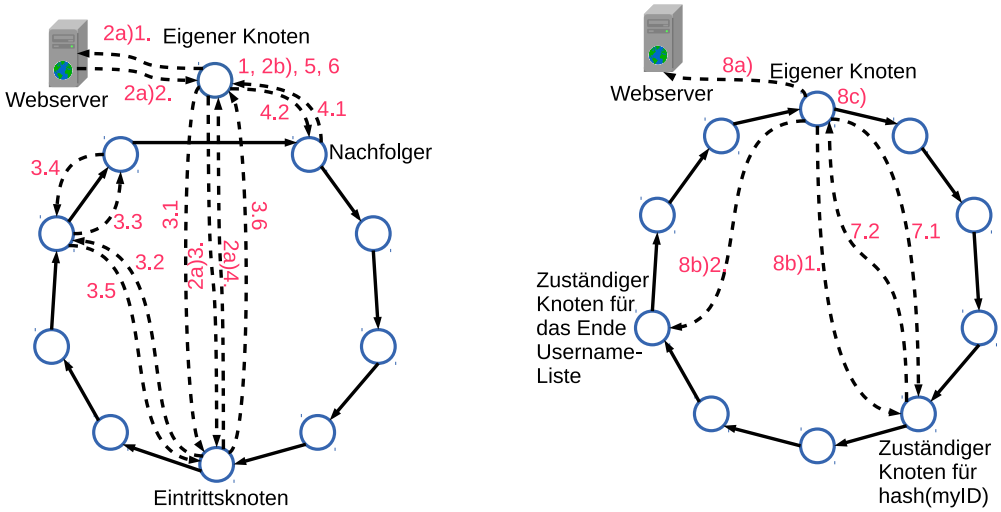
```
public abstract void create(Identity identity,ContactData contactDetails,ChordCallback
    callback) throws ServiceException;
public void joined(Certificate cert, ChordJoinFailure failure);
```

Der Methodenkopf des create Befehls und der Callback nach der Rückkehr der create Methode wird im Listing 4.4 näher gebracht. Der Ablauf des create Prozess sieht folgende Schritte vor. Zunächst wird das Kryptomodul anhand der übergebenen Identität, welche aus einem Benutzernamen und einem gehashten Passwort besteht, initialisiert. Details zu der Initialisierung des Kryptomoduls werden in der Sicherheitsschicht, in Abschnitt 4.2.4, beschrieben. Als nächstes wird die eigene Chord ID auf den soeben generierten öffentlichen Schlüssel des Kryptomoduls gesetzt. Die Referenzen, welche das lokale Repositorium. die Fingertabelle und die Nachfolgerliste bilden, werden initialisiert. Weiter wird der lokale Knoten initialisiert und der lokale Endpunkt wird auf *listen* gesetzt. Ist der listener gestartet, d.h. die Kommunikationsschicht bearbeitet nun eingehende Anfragen, die nicht das lokale Repositorium manipulieren, so werden die Instandhaltungsjobs gestartet und der Endpunkt in den Status *accept entries* versetzt. Nun werden auch Anfragen, die das lokale Repositorium manipulieren, bearbeitet. Außerdem werden angestaute Anfragen, die zuvor nicht bearbeitet werden durften, bearbeitet. Abschließend wird ein Zertifikat mithilfe der übergebenen Kontaktdetails erstellt, welches vom CertificateManagement in den neu geschaffenen privaten Chord-Ring eingefügt wird. Wurde das Zertifikat erfolgreich eingefügt, so wird der *joined* Callback gerufen, um den Aufrufer über den aktualisierten Status mitsamt des eigenen Zertifikat zu berichten.

Listing 4.5: Join Methodenkopf / Callback der abstrakten Chord Klasse.

```
public abstract void join(Identity identity, boolean wantRegister, ContactData
    contactDetails, ID bootstrapID, ChordCallback callback) throws ServiceException;
public void joined(Certificate cert, ChordJoinFailure failure);
```

Der join Prozess ist komplexer, da mehr Schritte als die bloße Initialisierung der Serviceschicht Komponenten und der Kommunikationsschicht unternommen werden müssen. Der angepasste join Befehl der abstrakten Chord Klasse und der dazugehörige Callback können dem Listing 4.5 entnommen werden. Der genaue join Prozess lässt sich anhand Abbildung 4.4 wie folgt beschreiben.



(a) Start Join Prozess

(b) Nach Chord Join

Abbildung 4.4: Join Prozess.

1. Analog zum create Prozess werden zunächst das Kryptomodul, die eigene Chord ID, der lokale Referenzenspeicher und der lokale Knoten initialisiert. Ist der listener gestartet, so wird zunächst geprüft, ob ein privater oder öffentlicher join erwünscht wurde. Dies geschieht anhand der übergebenen bootstrapID. Ist diese null, so soll ein öffentlicher join vollzogen werden.

2. Beschaffung eines Eintrittknotens *bootstrap node*:

   a) Öffentlicher join: Frage die Eintrittsknotenliste vom Webserver an. Da die übergebene Liste absteigend nach der letzten gemeldeten Aktivität der Knoten sortiert ist, werden die Knoten der Liste nach angepingt. Hierbei wird anhand der ID ein Proxy-Objekt gebildet und die Operation *ping* initiiert. Antwortet ein Knoten, so haben wir den gewünschten Eintrittsknoten erreicht.

   b) Privater join: Erstelle ein Proxy-Objekt aus der übergebenen boostrapID.

3. Es erfolgt der reguläre join Prozess, beschrieben in [SMK+01]. Zunächst wird auf den *bootstrap node* die Methode findSuccessor angewandt mit der Angabe der

lokalen ID.

4. Die Antwort des entfernten Knotens (bootstrap node) ist ein weiterer entfernter Knoten, der als potentieller Nachfolger benannt wird. Auf diesen Knoten wird nun die Methode notifyAndCopyEntries angewandt, um diesem die eigene Anwesenheit und die Bereitschaft Referenzen / Einträge zu erhalten kundzutun.

5. Ist die Antwort eingetroffen, so wird zunächst geprüft, ob die eigene Chord ID zwischen den ID's des zugesandten Vorgängers unseres potentiellen Nachfolgers und der ID des potentiellen Nachfolgers liegt. Ist dies nicht der Fall, so wird die Methode notifyAndCopyEntries auf den Vorgänger unseres potentiellen Nachfolgers angewandt. Hierdurch gelangen wir sukzessive zu dem richtigen Nachfolger. Haben wir den richtigen Nachfolger gefunden, so ist der join in den Chord-Ring erfolgreich und es werden alle zugesandten Einträge in das lokale Repositorium übernommen.

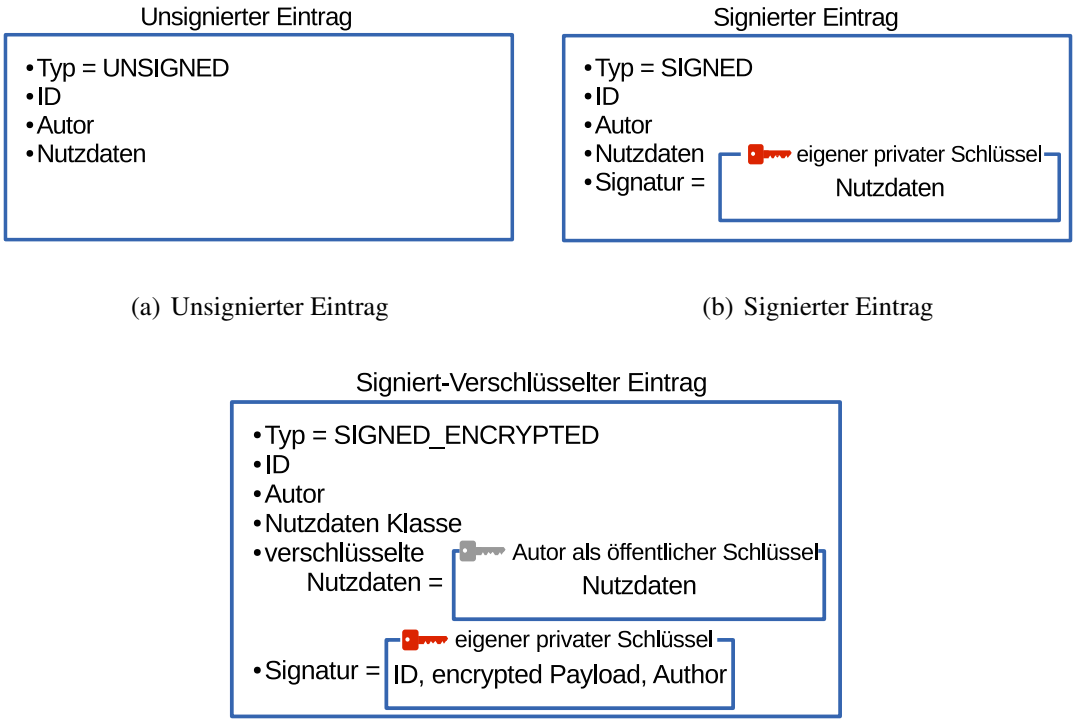
6. Analog zum create Prozess wird nun der Status des lokalen Endpunktes auf *accept entries* gestellt und alle Instandhaltungsjobs gestartet.

7. Mithilfe des Zertifikatmanagers wird nun nach einem vorhandenen Zertifikat an der Stelle des eigenen öffentlichen Schlüssels gesucht.

8. Nachdem die Antwort empfangen wurde, gibt es drei Möglichkeiten:

   a) Liefert der Zertifikatmanager ein gültiges Zertifikat und der Nutzer wollte sich einloggen, so wird abschließend, falls es sich um einen öffentlichen Login handelt, die eigene Chord ID an den Webserver versendet, um neu eintretenden Knoten einen Eintrittsknoten zu stellen. Es erfolgen Aufräumarbeiten und der joined Callback wird gerufen, um dies der Anwendungsschicht zu signalisieren.

   b) Liefert der Zertifikatmanager kein gültiges Zertifikat und der Nutzer wollte sich registrieren, so wird mithilfe des Kryptomoduls ein Zertifikat anhand der contactDetails erstellt und mit Unterstützung des Zertifikatmanagers an zwei Stellen in den Chord-Ring eingefügt. Abschließend wird die eigene



(a) Unsignierter Eintrag

(b) Signierter Eintrag

(c) Signiert-Verschlüsselter Eintrag

Abbildung 4.5: DHT Eintragstypen.

Chord ID gesichert, insofern es sich um ein öffentlichen register join handelt. Nach Aufräumarbeiten wird der joined Callback gerufen, um dies der Anwendungsschicht zu signalisieren.

c) Sonst: Entweder wollte sich der Benutzer registrieren, obwohl bereits solch ein Benutzer existiert, d.h. die Kombination aus Benutzername und Passwort stimmen überein, oder der Benutzer wollte sich einloggen, obwohl dieser Benutzer noch nicht registriert wurde. In beiden Fällen wird nach Aufräumarbeiten der joined Callback gerufen, um der Anwendungsschicht das Fehlverhalten zu berichten. Abschließend wird der Chord-Ring verlassen.

Zuletzt muss der Chatplattform die Möglichkeit gegeben werden Daten verschlüsselt zu sichern. So soll es beispielsweise möglich sein sensible Daten wie Kontaktlisten ver-

schlüsselt in der DHT abzulegen. Um dies zu gewährleisten, wurden drei Kategorien von DHT-Einträgen geschaffen. Alle Kategorien enthalten essenzielle Eigenschaften wie die ID, worunter dieser Eintrag in der DHT abgelegt wird, die ID des Autors und das Objekt selbst, welches unter der ID abgelegt werden soll. Alle implementierten Einträge besitzen Methoden, um die Nutzdaten des Eintrags zu validieren, die Nutzdaten zu extrahieren und prüfen, ob zwei Objekte vom gleichen Autor stammen. Nachstehend werden die Unterschiede und Methodenimplementierungen der drei Kategorien gegenübergestellt.

- *unsigned*: Grundlegender Eintrag, visualisiert durch Abbildung 4.5(a), welcher nicht signiert ist. Dieser Eintrag enthält nur die essenziellen Eigenschaften und ist immer valide. Die Nutzdaten des Eintrags, ein JsonSerializable-Objekt, kann immer aus dem Eintrag extrahiert werden.

- *signed*: Ein signierter Eintrag, zu sehen in Abbildung 4.5(b). Dieser enthält zusätzlich eine Signatur über die enthaltenen Daten. Bei der Validierung wird die Signatur mithilfe der ID des Autors, welche gleichzeitig sein öffentlicher Schlüssel ist, verifiziert. Die Nutzdaten des Eintrags kann, wie bei dem unsignierten Eintrag, immer extrahiert werden. Eine Besonderheit bildet das Zertifikat. Zertifikate werden immer als signierte Einträge gespeichert und bei der Validierung eines Zertifikateintrags wird zusätzlich der Inhaber des Zertifikats mit dem Autor des Eintrags abgeglichen.

- *signed_encrypted*: Ein signiert und verschlüsselter Eintrag, siehe Abbildung 4.5(c). Diese Kategorie enthält neben den essenziellen Eigenschaften verschlüsselte Nutzdaten als Zeichenkette, eine Signatur über den kompletten Eintrag und eine Zeichenkette, die den Namen der Java Klasse der Nutzdaten speichert. Diese ist notwendig, um die Deserialisierung der entschlüsselten Nutzdaten zu gewährleisten. Ähnlich zum signierten Eintrag wird bei der Validierung die Signatur mithilfe der ID des Autors verifiziert. Anders als bei den anderen beiden Einträgen ist es nicht für jeden Knoten möglich die Nutzdaten des Eintrags einzusehen, da die Entschlüsselung den privaten Schlüssel des Autors benötigt. Ist man nicht befugt, so erhält man eine Exception bei dem Versuch diesen zu entschlüsseln.

Listing 4.6: Insert, remove und retrieve Methodenköpfe und deren Callbacks der abstrakten Chord Klasse.

```
public abstract void insert(EntryType type, String entryClass, Key key,
    JsonSerializable entry, ChordCallback callback);
public void inserted(EntryType type, String entryClass, Key key, JsonSerializable
    entry, ChordCallbackFailure failure);

public abstract void remove(EntryType type, String entryClass, Key key,
    JsonSerializable entry, ChordCallback callback);
public void removed(EntryType type, String entryClass, Key key, JsonSerializable entry,
    ChordCallbackFailure failure);

public abstract void retrieve(Key key, ChordCallback callback);
public void retrieved(Key key, JsonSerializable entry, ChordCallbackFailure failure);
```

Um die oben genannten DHT-Einträge in den Chord-Ring einzufügen, löschen und empfangen, stehen die drei Operationen insert, remove und retrieve bereit, dessen Methodenköpfe und Callbacks in Listing 4.6 angegeben sind. Die Abläufe der beiden DHT Manipulierungsprozesse ähneln sich sehr, weshalb folgend die insert bzw. remove Operation vorgestellt wird. Veranschaulicht wird dieser Prozess von Abbildung 4.6;

1. Der übergebene Key wird mit der festgelegten Hashfunktion gehasht. Man erhält die Chord ID für diesen Eintrag. Sei *entryID* diese Chord ID.

2. Der gewünschte EntryTyp wird anhand des übergebenen JsonSerializable-Objekts und der entryID gebildet.

3. Lokaler Knoten führt einen findSuccessor auf die entryID aus.

4. Wurde der Nachfolger für die entryID gefunden, so führe insertEntry bzw. removeEntry auf dem Proxy-Objekt aus.

5. Der zuständige Knoten erhält über seine Kommunikationsschicht die Anfrage und leitet diese an den lokalen Knoten (NodeImpl) weiter.

6. Die Validität und Zulässigkeit der Operation wird durch den zuständigen Knoten geprüft:

    - Ist der Eintrag valide? Prüfung des Eintrags.

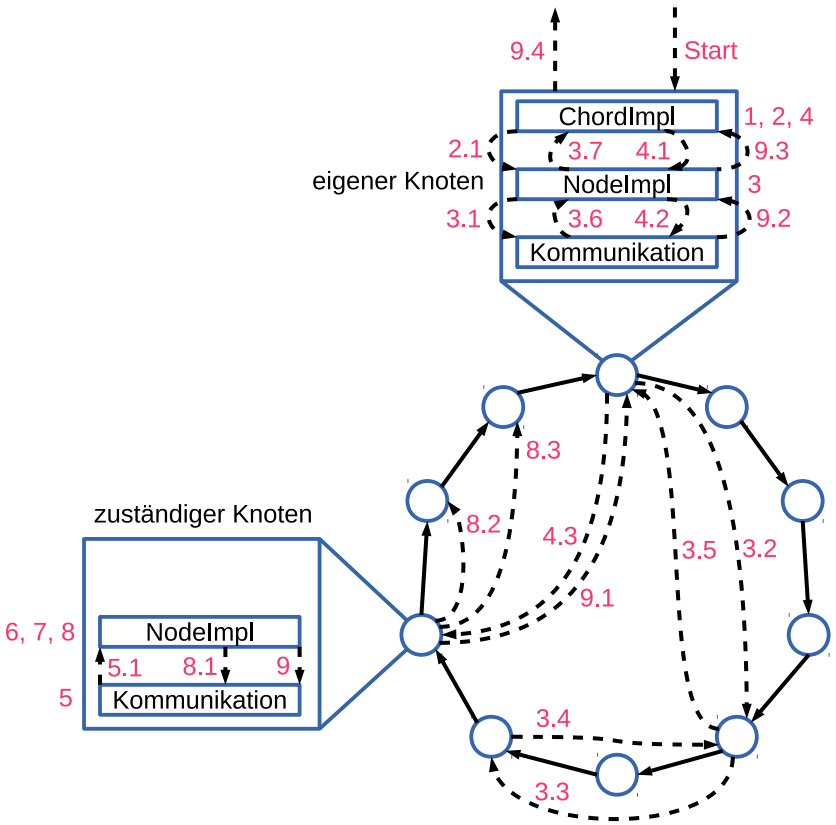


Abbildung 4.6: Insert / Remove Operation.

- Darf der Autor diesen Eintrag in die DHT einfügen bzw. modifizieren? Prüfe, ob der vorhandene Eintrag mit der gleichen ID, falls existent, vom gleichen Autor ist. Unsignierte Einträge sind von dieser Prüfung ausgenommen, da diese jeder Autor modifizieren darf. Durch die Prüfung soll die Manipulation signierter Einträge durch andere Autoren unterbunden werden.

- Zusätzlich werden bei remove Anweisungen Operationen auf Zertifikate untersagt. Der Hintergrund ist, dass die logische Benutzernamenliste hierdurch unterbrochen werden könnte. Näheres dazu in diesem Abschnitt zur Zertifikatinfrastruktur.

7. Unter Umständen haben sich Zuständigkeiten der DHT geändert, sodass diese Anfrage an den Vorgänger weitergeleitet werden muss. Dies ist der Fall, falls unser

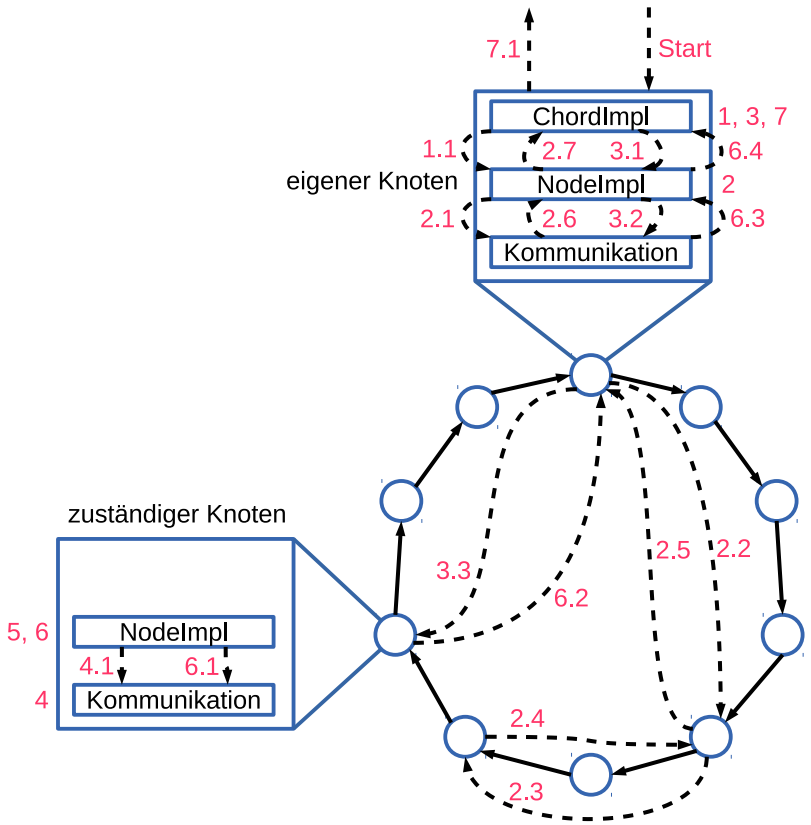


Abbildung 4.7: Retrieve Operation.

Vorgänger $\neq$ null und $entryID \notin]ID_{predecessor}, ID_{my}]$.

8. Füge den Eintrag in das lokale Repositorium ein bzw. entferne es. Initiiere daraufhin ein *insertReplicas* bzw. *removeReplicas* auf meine Nachfolger. Diese prüfen jeweils nochmals die Validität und Zulässigkeit der Operation und nehmen anschließend die Manipulation in ihr lokales Repositorium auf.

9. Abschließend erfolgt eine Bestätigung der Operation oder eine Fehlerberichterstattung an den Initiator.

Der Ablauf der retrieve Operation, veranschaulicht in Abbildung 4.7, ist wie folgt.

1. Der übergebene Key wird mit der festgelegten Hashfunktion gehasht. Man erhält die Chord ID für den gewünschten Eintrag. Sei *entryID* diese Chord ID.

2. Lokaler Knoten führt einen findSuccessor auf die entryID aus.

3. Wurde der Nachfolger für die entryID gefunden, so führe retrieveEntry auf dem Proxy-Objekt aus.

4. Der zuständige Knoten erhält über seine Kommunikationsschicht die Anfrage und leitet diese an den lokalen Knoten (NodeImpl) weiter.

5. Unter Umständen haben sich Zuständigkeiten der DHT geändert, sodass diese Anfrage an den Vorgänger weitergeleitet werden muss. Dies ist der Fall, falls unser Vorgänger $\neq$ null und $entryID \notin]ID_{predecessor}, ID_{my}]$.

6. Abschließend wird der Eintrag an der Stelle entryID aus dem lokalen Repositorium genommen und als Antwort an den anfragenden Knoten geschickt.

7. Der anfragende Knoten erhält die Antwort in Form eines DHT Eintrags. Die Validität des Eintrags wird geprüft und die Nutzdaten des Eintrags werden extrahiert. Der gesicherte Callback wird mit dem Key und den Nutzdaten des Eintrags gerufen. Bei etwaigen Fehlern werden diese ebenfalls per Callback übermittelt.

Schließlich enthält der Anhang A einen Auszug aus der abstrakten Chord Klasse, dessen Implementierung ChordImpl von der ChordFactory instanziiert werden kann. Weiter enthält der Anhang eine Liste mit den Callbacks, die für die Chord Methoden benötigt werden.

Die Serviceschicht ist nun komplett und beinhaltet somit alle notwendigen Operationen, die eine Chatplattform benötigt. Dies reicht von dem Login mit einer Identität, dessen Zertifikat im Chord-Ring mithilfe der Zertifikatinfrastruktur abgelegt und andere Nutzer gesucht werden können, über den Nachrichtenversand und eine AV-Stream Unterstützung bis hin zu verschlüsselten und / oder signierten Chord Einträgen im Ring. Um unter Anderen Chord Einträge verschlüsseln zu können, ist eine Sicherheitsschicht im Chordmodul notwendig. Dessen Präsentation erfolgt im anschließenden Abschnitt.

4.2.4 Sicherheitsschicht

Neben der Sicherheitsarchitektur von WebRTC [Res11] ist die Einführung einer Sicherheitsschicht in Chord notwendig. Die Sicherheitsschicht des Chordmoduls arbeitet ergänzend zu der des WebRTC's und beinhaltet die Sicherheitslogik, welche der Kommunikations- und Serviceschicht angeboten wird. Zunächst folgt eine kurze Übersicht des Kapitels.

Der erste Abschnitt befasst sich mit der Frage wofür eine weitere Sicherheitsschicht notwendig ist und was diese genau können soll, d.h. welche Ansprüche an diese gestellt werden. Der zweite Abschnitt bietet eine Auswahl bestehender Sicherheitsbibliotheken und diskutiert deren Vor- und Nachteile. An dieser Stelle wurden vorhandene GWT-Bibliotheken gesucht und geprüft, ob diese den oben genannten Ansprüchen genügen. Als Ausweichslösung können Javascript-Bibliotheken eingebunden und über das JSNI (JavaScript Native Interface) angesprochen werden. Der dritte und zugleich letzte Abschnitt befasst sich mit der Einbindung der Bibliothek und Implementierung der benötigten Methoden der Kommunikation- und Serviceschicht.

Notwendigkeit und Ansprüche

Begonnen wird mit dem ersten Abschnitt und der Frage nach der Notwendigkeit und den Ansprüchen an diese Schicht. Das Problem der Sicherheitsarchitektur von WebRTC, welche gleichzeitig die Notwendigkeit einer separaten Sicherheitsschicht entspricht, ist der fehlende Zugriff auf die verwendeten Schlüsselpaare der asymmetrischen Kryptografie für weitere Operationen. Eine wichtige Operation wäre die Ver- / Entschlüsselung und Signierung sowie Verifizierung von Chord Einträgen, welche von der Serviceschicht verlangt wird. Darauf baut die Zertifikatinfrastruktur der Serviceschicht und die verschlüsselte Sicherung von Anwendungsdaten im Chord-Ring auf. Weiter soll die Chord ID eines jeden Knotens gleichzeitig sein öffentlicher Schlüssel sein, welcher deterministisch aus einer gegebenen Identität generiert werden muss. Insbesondere ist der öffentliche Schlüssel für eine Identität eindeutig. Hingegen erweist sich die Einbindung des Authentifizierungsprozesses von WebRTC als problematisch, da auch bei einer erfolgreichen

Authentifizierung bei externen Diensten keine Informationen zur Generierung eindeutiger Chord ID's zur Verfügung stehen. Auch ohne die Einbindung des Authentifizierungsprozesses von WebRTC solle mit der Zertifikatinfrastruktur und der Signierung und Verschlüsselung von Nachrichten eine Authentifizierung zwischen Knoten geschaffen werden. So ist die Notwendigkeit einer ergänzenden Sicherheitsschicht auf Chord Ebene geklärt.

Die Ansprüche an die asymmetrische Kryptografie, speziell an die Schlüssellänge, sind die folgenden. Die aus der Identität des Knotens generierte Chord ID sollte nicht zu lang werden, da sonst der Verwaltungsaufwand der Fingertabelle wächst. Sei m die Bitlänge der Hashwerte, dann enthält die Fingertabelle je nach Implementierung $\leq m$ Einträge. So wäre *RSA* beispielsweise eine schlechte Wahl, da hier aus Sicherheitsgründen mindestens 2048 Bit lange Schlüssel verwendet werden sollten. *Elliptic Curve Cryptography*, kurz ECC, hingegen benötigt bei gleichem Sicherheitslevel kleinere Schlüssel als RSA [GS11]. Beispielsweise entspricht eine ECC Schlüssellänge von 256 Bit einer RSA Schlüssellänge von 3072 Bit [nsa09]. Anders als RSA beruht die Sicherheit von ECC auf dem diskreten Logarithmus Problem auf elliptischen Kurven, kurz ECDLP, welches ein schwereres Problem ist als die Faktorisierung ganzer Zahlen. Ein weiterer Nebeneffekt ist, dass Rechenoperationen auf kleineren Schlüsseln einen Geschwindigkeits- und Ressourcenvorteil bieten [nsa09].

Existieren jedoch aktuell Bedenken bezüglich der ECC durch jüngste Ereignisse. Das National Institute of Standards and Technology, kurz NIST, publizierten Koeffizienten für die Weierstraß-Gleichung $y^2 = x^3 + a * x + b \mod p$ für verschiedene p, welche eine sichere elliptische Kurve darstellt. Eine vollständige Liste hält die Standards for Efficient Cryptography Group, kurz SECG, in ihrem Dokument [Bro10], wo die NIST Mitglied ist, fest. An der Entwicklung der geeigneten Koeffizienten der NIST war die National Security Agency, kurz NSA, beteiligt. Nach dem potentiellen Backdoor des Zufallsgenerators *Dual_EC_DRBG* [DS07], welche der NSA angerechnet wird, ist das Vertrauen gegenüber der NSA, nicht zuletzt wegen der *Snowden-Affäre* [GGP13], gesunken [Zet13]. So wird den NIST Kurven eine potentielle Manipulierung der Parameter nachgesagt [BL14], da die Parameter durch das hashen eines nicht weiter erklärten Seeds erzeugt wurden. Andere gelistete elliptische Kurven enthalten jeweils eine Erklärung der Herkunft der Parameter. Die Internetpräsenz [BL14], geführt von Prof. Dr. Daniel J. Bernstein

und Prof. Dr. Tanja Lange (Technische Universiteit Eindhoven), zeigt alternative Elliptische Kurven, welche auf Schwachstellen geprüft wurden. Montgomery Kurven, wie beispielsweise Curve511187 [ABGR13] oder Curve25519 [Ber06], sind nachweislich robuster gegen bekannte Angriffe. Somit ist bei der Nutzung der Elliptic Curve Cryptography auf die Wahl der elliptischen Kurve zu achten.

Auswahl Sicherheitsbibliotheken

Im zweiten Abschnitt wird eine Auswahl an GWT- bzw. Javascript-Bibliotheken, die die oben genannten Voraussetzungen erfüllen, aufgestellt und diskutiert.

GWT Security Bibliotheken gibt es bisher nur eine. Es ist die *gwt-crypto* [RP13] Bibliothek und enthält leider keine ECC. Diese Bibliothek bietet symmetrische, sowie asymmetrische Verschlüsselung an. Für die symmetrischen Verschlüsselung werden *3DES* und *AES* angeboten. Auf der asymmetrischen Seite finden wir lediglich *RSA*.

Javascript-Security-Bibliotheken gibt es hingegen einige. Die Einbettung einer Javascript-Bibliothek in ein GWT-Projekt bedeutet hingegen, dass diese unter Umständen Sicherheitslücken in der GWT-Anwendung öffnen. Die Sicherheit von GWT-Anwendungen wird in Abschnitt 2.3.2 diskutiert. Tabelle 4.1 zeigt die populärsten Javascript-Security-Bibliotheken mit ECC.

JScrypto von Zhi Guan ist leider veraltet und eine Erweiterung der Liste an unterstützten elliptischen Kurven ist ungewiss. Außerdem wurde diese Bibliothek nicht dokumentiert.

JSBN von der Stanford University bietet dagegen eine Dokumentation und mehr Kurven an. Hier sind neben NIST auch Koblitz Kurven vertreten, gekennzeichnet durch *k1* im Namen. Jedoch sind hier keine 'SafeCurves' enthalten und ein Update auf andere elliptische Kurven ist auch hier ungewiss.

Js-NaCl, dessen Motivation im Paper [BLS12] beschrieben wird, bietet als einzige genannte Bibliothek eine SafeCurve an. Verwendet wird hier die Montgomery Kurve

Tabelle 4.1: Javascript-Security-Bibliotheken mit ECC.

Name der Bibliothek	*Dokumentation vorhanden*	*Letztes Update*	*Unterstützte elliptische Kurven*	*Sonstiges*
JScrypto [Gua09]	×	Februar 2009*	secp192r1**	Veraltet, keine updates, keine' SafeCurves'.
JSBN [Wu13]	✓	Juli 2013	secp128r1 secp160r1 secp160k1 secp192r1 secp192k1 secp224r1 secp256r1	Kurven updates ungewiss, keine 'SafeCurves'
SJCL [ESB14] (Stanford Javascript Crypto Library)	✓	Mai 2014 auf GitHub	secp192r1 secp192k1 secp224r1 secp224k1 secp256r1 secp256k1 secp384r1	Relativ junge Bibliothek, aktiver Support, keine 'SafeCurves', jedoch Support von Montgomery Kurven denkbar***
Js-NaCl [GJ14]	✓	Mai 2014 auf GitHub	Curve25519	SafeCurve, aktiver Support

Stand 23.05.2014.
[*]: Sourcecode in Version 0.6.
[**]: Laut Sourcecode.
[***]: Quelle: https://github.com/bitwiseshiftleft/sjcl/issues/158

Curve25519. Auf der GitHub Seite ist eine gute API Beschreibung angegeben und der Code wird weiterhin aktiv aktualisiert.

SJCL ist eine weitere jüngere Bibliothek der Stanford University, welche ihren initialen *commit* unter GitHub im Mai 2010 hatte. Diese Bibliothek enthält eine sehr gute Dokumentation und wird bis dato aktuell gehalten. Die bisher unterstützten Kurven enthalten zwar lediglich NIST und Koblitz Kurven aber der Support weiterer Nicht-Weierstraß Kurven wäre denkbar.

Die Wahl der Sicherheitsbibliothek beschränkt sich auf die Wahl zwischen SJCL- und Js-NaCl-Bibliothek. Der Vorteil der NaCl-Bibliothek gegenüber der SJCL ist die Verwendung der als sicherer geltenden elliptische Kurve. Der Vorteil der SJCL-Bibliothek ist hingegen, dass die Schlüsselpaare für das ver- / entschlüsseln und signieren / verifizieren gleich gewählt werden können. Beispielsweise würde unter Verwendung der Kobilitz Kurve secp256k1 und der Anwendung der nachfolgend erklärten *publicKey-Komprimierung* einen öffentlichen Schlüssel der Länge 256 Bit. Dieser öffentliche Schlüssel kann dazu verwendet werden Nachrichten verschlüsselt an den Besitzer des zugehörigen privaten Schlüssels zu schicken und von diesem unterzeichnete Signaturen zu verifizieren. Die NaCl-Bibliothek benötigt hingegen zwei unterschiedliche Schlüsselpaare zum ver- / entschlüsseln und signieren / verifizieren. Die öffentlichen Schlüssel sind jeweils 256 Bit lang. Da die Voraussetzung vorsieht, dass die Chord ID gleichzeitig der öffentliche Schlüssel ist, erreicht man hier beispielsweise durch Konkatenation der beiden öffentlichen Schlüssel eine Chord ID der Länge 512 Bit.

Letztendlich erscheint die Einbindung beider Bibliotheken sinnvoll. Im Umfang dieser Arbeit wird jedoch nur die Einbindung der SJCL-Bibliothek beschrieben. Im Ausblick, detailliert in Abschnitt 8.2, wird auf die zukünftige Einbindung der Js-NaCl-Bibliothek eingegangen.

Einbindung Sicherheitsbibliothek

Der dritte Abschnitt beinhaltet nun die Einbindung der SJCL-Bibliothek und Realisierung der benötigten Methoden der Kommunikation- und Serviceschicht. Nachdem das SJCL Git Repositorium geholt und mitsamt dem ECC-Modul kompiliert und in das Projekt eingebunden wurde, stellte sich die Frage der verwendeten elliptischen Kurve. Die Wahl fiel auf die Koblitz Kurve secp256k1, welche eine Schlüssellänge von 256 Bit verwendet, was einer RSA Schlüssellänge von 3072 Bit entspricht. Dementsprechend wurde SHA256 als Hashfunktion verwendet, um 256 Bit lange hashes zu generieren.

Da die öffentlichen Schlüssel dieser Art von elliptischen Kurve einem zweidimensionalen Punkt entsprechen, wäre dieser bei einer Konkatenation der beiden Koordinaten zunächst 512 Bit lang. Eine Möglichkeit die Länge des öffentlichen Schlüssels zu hal-

bieren bietet die *publicKey-Komprimierung*, welche nachfolgend erläutert wird. Viele NIST und Koblitz Kurven beschreiben eine elliptische Kurve, welche die folgende Weierstraß-Gleichung erfüllen:

$$y^2 = x^3 + a * x + b \mod p \text{ in } \mathbb{F}_p$$

Wobei $a, b \in \mathbb{Z}$ und p prim die Koeffizienten und somit Bestandteil der Kurvenparameter sind. Da der öffentliche Schlüssel nun ein Punkt auf dieser Kurve bzw. Punktewolke ist, kann für eine gegebene Koordinate x die y-Koordinate abgeleitet werden:

$$\text{Sei } \alpha = x^3 + a * x + b$$

$$y = \sqrt{\alpha} \bmod p$$

$$\text{Falls } p \mid \alpha \Rightarrow \alpha = 0 \Rightarrow y = 0 \text{ in } \mathbb{F}_p$$

$$\text{Sei } p \perp \alpha \text{ und } p \equiv 3 \bmod 4 \tag{4.1}$$

$$\text{Bleibt zu zeigen: } (\alpha^{\frac{p+1}{4}})^2 \equiv \alpha \mod p$$

$$(\alpha^{\frac{p+1}{4}})^2 \equiv \alpha^{\frac{p+1}{2}} \equiv \alpha^{\frac{p-1}{2}} * \alpha \equiv \sqrt{\alpha^{p-1}} * \alpha \bmod p$$

$$\text{Da } \alpha \perp p \Rightarrow^{4} \alpha^{p-1} \equiv 1 \bmod p$$

$$\Rightarrow \sqrt{\alpha^{p-1}} * \alpha \equiv \sqrt{1} * \alpha \bmod p \tag{4.2}$$

$$\sqrt{1} \equiv \pm 1 \bmod p, \forall p \text{ in } \mathbb{F}_p$$

$$\text{Da } p \equiv 3 \bmod 4 \Rightarrow \frac{p+1}{4} \in \mathbb{Z}$$

$$\Rightarrow y \equiv \pm x^{\frac{p+1}{4}} \mod p \tag{4.3}$$

Durch die Bedingung 4.1 fallen Kurven raus, die auf $\mathbb{F}_{2^m}$ operieren. Außerdem erfüllen einige Weierstraß Kurven wie secp224r1 diese Anforderung nicht. Für allgemeine p müsste der *Tonelli-Shanks Algorithmus* implementiert werden.

Anhand der x-Koordinate kann nun die y-Koordinate modulo Vorzeichen mithilfe der Gleichung 4.3 abgeleitet werden. Wenn nun bei der Generierung des Schlüsselpaares

[4] Fermat's kleines Theorem

darauf geachtet wird, dass nur öffentliche Schlüssel verwendet werden, die zum Beispiel $y \equiv 0 \bmod 2$ sind, so kann die y Koordinate vollständig aus der x-Koordinate abgeleitet werden.

Wie wird also aus der Identität, bestehend aus Benutzername und hashed Passwort, ein Schlüsselpaar generiert? Die Generierung des Schlüsselpaares erfolgt im Code durch einen JSNI Aufruf und wird bei der Instanziierung des Crypto-Moduls gerufen. Die gerufene *native* Methode tätigt Aufrufe an die SJCL-Bibliothek. Der genaue Ablauf ist wie folgt zu beschreiben:

1. Generiere einen Schlüssel *key* mithilfe des *pbkdf2* (Password-based Key Derivation Function 2) Algorithmus. Die Länge des Schlüssels ist in der Konfiguration festgehalten und beträgt standardmäßig 1024 Bits. Die Parameter des pbkdf2 enthalten den Benutzernamen, das gehashte Passwort, die Anzahl an Iterationen (per Konfiguration: standardmäßig 1024) und die eben genannte Schlüssellänge.

2. Leite den privaten Schlüssel anhand des eben generierten Schlüssels ab. Dieser hat die Voraussetzung im Intervall $]0; n[$ zu liegen, wobei n ein Kurvenparameter ist. Erreicht wurde dies durch $(key \bmod n-1)+1$.

3. Leite den öffentlichen Schlüssel vom privaten Schlüssel ab. Dies ist eine Multiplikation des Generators (Kurvenparameter) mit dem privaten Schlüssel.

4. Prüfe den generierten öffentlichen Schlüssel: Ist die y Koordinate des generierten Schlüssels $\equiv 0 \bmod 2$ und das *most siginificant byte* $\neq 0$? So ist die Schlüsselgenerierung erfolgreich andernfalls wird die Anzahl an Iterationen des pbkdf2 inkrementiert und zu Schritt 1 zurückgesprungen. Die erste Bedingung stellt sicher, dass der öffentliche Schlüssel auf die Länge der x-Koordinate komprimiert werden kann. Die zweite Bedingung betrifft eher einen Bug in der SJCL-Bibliothek, wo die Dekomprimierung fehlschlägt, wenn die x-Koordinate des öffentlichen Schlüssels ein führendes 0 Byte enthält.

Die Generierung des Schlüsselpaares ist deterministisch, d.h. für eine gegebene Kombination aus Benutzernamen und Passwort wird immer das gleiche Schlüsselpaar generiert. Der Nachteil der deterministischen Schlüsselgenerierung ist, dass der Benutzer

ein großes und sicheres Passwort zu wählen hat, da sonst durch Wörterbuchattacken die Identität leicht übernommen werden kann.

Das Schlüsselpaar ist nun so gewählt, dass die x-Koordinate als Hexadezimalzahl nun für die Chord ID eingesetzt werden kann. Dies entspräche der komprimierten Darstellung des öffentlichen Schlüssels. Um diesen komprimierten öffentlichen Schlüssel zu dekomprimieren, um beispielsweise dem Besitzer des öffentlichen Schlüssels eine verschlüsselte Nachricht zu senden, sind folgende Schritte erforderlich und werden von der Klassenmethode *static getPublicKey(String compressedPublicKey)* der Klasse PublicKey implementiert. Zunächst wird der lokale PublicKeyPool gefragt, ob zuvor solch ein komprimierter öffentlicher Schlüssel erfolgreich dekomprimiert wurde. Ist dies der Fall, so wird das PublicKey-Objekt zurückgegeben. Andernfalls erfolgt die Dekomprimierung wie folgt (JSNI Call). Aus der übergebenen Zeichenkette wird ein sjcl.BN (big number)-Objekt erzeugt. Nun gilt es y aus x abzuleiten. Die Berechnung reduziert sich laut Gleichung 4.3 auf die Berechnung von $\beta \equiv (x^3 + a * x + b)^{\frac{p+1}{4}} \bmod p$. Falls nun $\beta \equiv 0 \bmod 2 \Rightarrow y = \beta \bmod p$. Sonst $y = -\beta \bmod p$. Abschließend erfolgt ein Test, ob der Punkt x / y auch auf der Elliptischen Kurve liegt. Ist dies nicht der Fall, so wird eine Exception geworfen. Ansonsten wird der erfolgreich dekomprimierte öffentliche Schlüssel zurückgegeben.

Die Beschreibung der Methodenimplementierungen, benötigt von der Kommunikations- sowie Serviceschicht, erfolgt nun.

Listing 4.7: Signiermethoden.

```
protected abstract SignedData sign(String data);
public abstract String signEntry(ID id, JsonSerializable value, ID author);
public abstract String signEntry(ID id, String encrypted, ID author);
```

Die *Signierungsmethoden*, aufgeführt in Listing 4.7, beinhalten das allgemeine signieren von Zeichenketten und das signieren von DHT Einträgen. Die *sign* Methode signiert die übergebene Zeichenkette und gibt ein *SignedData*-Objekt zurück, welches die Originaldaten und die Signatur als Hex-String enthält. Für den SJCL sign Aufruf wird ein Wert, welcher *paranoia* genannt wird, benötigt. Dieser Wert wird für den in SJCL implementierten Zufallszahlgenerator (prng = pseudo random number generator) benötigt. Der prng sammelt Daten von Mausbewegungen und Interaktionen mit der GUI, um den

Pool zu füllen. Der paranoia wert zeigt an welche Entropie die Daten enthalten sollen. Es gibt zur Zeit elf Entropielevel (Arrayindex 0-10): [0, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024]. Dies bedeutet, dass die vom prng ausgegebenen Daten eine Entropie von beispielsweise 96, bei Arrayindex 3, haben müssen. Der gewählte Arrayindex wird in der Konfiguration festgehalten und ist auf standardmäßig zehn gesetzt, welcher gleichzeitig der Maximalwert ist. Für den SJCL Call wird zunächst noch ein hash über den übergebenen Datenstring durchgeführt um abschließend den Call durchzuführen: *ownPrivateKey.sign(hash,paranoia)*, wobei ownPrivateKey der eigene private ECDSA Schlüssel ist. Die erzeugte Signatur wird durch einen weitere SJCL Call in einen Hex-String umgewandelt und das benötigte SignedData-Objekt instanziiert und zurückgegeben.

Die beiden signEntry Methoden rufen die eben beschriebene sign Methode auf, nachdem die übergebenen Parameter in ihrer Reihenfolge in der Parameterliste als String verkettet und übergeben wurden. Vom erzeugten SignedData-Objekt wird lediglich die Hex Signatur als String zurückgegeben.

Listing 4.8: Verifizierungsmethoden.

```
protected abstract void verify(SignedData data, PublicKey senderPub) throws
    CorruptDataException;
public abstract void verifyEntry(ID id, Object payload, ID author, String
    hexSignature) throws CorruptDataException;
public abstract void verifyCertificate(Certificate cert) throws CorruptDataException;
```

Die in Listing 4.8 angegebenen *Verifizierungsmethoden* prüfen Signaturen von DHT Einträgen und Zertifikaten. Es wird ein SignedData-Objekt, sowie eine PublicKey-Instanz übergeben, welche den potentiellen PublicKey des Signierers enthält. Ist die Signatur valide und passt zu dem PublicKey, so liefert diese Funktion keine Rückgabe. Ist diese korrupt, d.h. der PublicKey passt nicht zu der Signatur, so wird eine *CorruptDataException* geworfen. Hierfür werden zunächst die Originaldaten gehasht und anschließend per SJCL Aufruf (publicKey.verify(hash,signature)) geprüft.

Die verifyEntry Methode erzeugt zunächst anhand der übergebenen Parameter ein Signed-Data-Objekt. Die übergebene Autor ID wird als komprimierte PublicKey Zeichenkette aufgefasst und anhand der oben genannten PublicKey.getPublicKey(String) wird ein PublicKey-Objekt erzeugt. Achtung: falls kein gültiger PublicKey erzeugt werden konnte, so wird eine CorruptDataException geworfen, d.h. die Daten passen nicht zu der

Signatur! Anschließend wird die oben genannte verify Methode gerufen. Die verifyCertificate Methode arbeitet analog zu der verifyEntry Methode, wonach zunächst ein SignedData und PublicKey-Objekt erzeugt werden, um anschließend die oben genannte verify Methode zu rufen.

Listing 4.9: Verschlüsselungsmethoden.

```
protected abstract String encrypt(SignedData data, PublicKey recvPub);
public abstract String encryptEntry(JsonSerializable value, ID author) throws
    CorruptDataException;
```

Listing 4.9 beinhaltet alle benötigten Verschlüsselungsmethoden. Die encrypt Methode benötigt ein SignedData und ein PublicKey-Objekt des Empfängers der Daten. Zurückgegeben wird der Chiffretext (cipher text) als Zeichenkette. Das SignedData-Objekt wird zunächst mithilfe des JsonSerializer serialisiert und anschließend ein JSNI Call mit den serialisierten Daten und dem PublicKey getätigt. Hier gibt es lediglich einen Call an die SJCL-Bibliothek: sjcl.encrypt(publicKey, DataToEncrypt). Verschlüsselt wird mit der ElGamal Verschlüsselungsmethode.

Die encryptEntry Methode formt aus den übergebenen Parameter ein SignedData- und PublicKey-Objekt und ruft die eben beschriebene encrypt Methode.

Listing 4.10: Entschlüsselungsmethoden.

```
protected abstract SignedData decrypt(String encryptedData) throws CorruptDataException;
public abstract JsonSerializable decryptEntry(String encryptedData, String
    serializedClass) throws CorruptDataException;
```

Die Entschlüsselungsmethoden, gelistet im Listing 4.10, bieten die Entschlüsselung übergebener Chiffretexten. Konnte der Chiffretext anhand des eigenen privaten Schlüssels entschlüsselt werden, so wird eine CorruptDataException geworfen. Die Methode decryptEntry entschlüsselt analog dazu den Chiffretextes unter der Angabe der serialisierten Klasse und liefert das deserialisierte JsonSerializable-Objekt.

Listing 4.11: Mischmethoden.

```
public abstract String signEncrypt(String serializedData, PublicKey recvPub);
public abstract String decryptVerify(String encryptedSignedData, PublicKey senderPub)
    throws CorruptDataException;
```

Schließlich sind in Listing 4.11 zwei Methoden für die Kommunikationsschicht angegeben. So sorgt signEncrypt für die Signierung der übergebenen serialisierten Daten, welche mithilfe des PublicKey-Objekts des Empfängers verschlüsselt werden. Intern werden hier Aufrufe an die sign und encrypt Methode getätigt. Das Produkt ist ein Chiffretext als Zeichenkette. Die decryptVerify Methode hingegen entschlüsselt zunächst den übergebenen Chiffretext mithilfe des eigenen privaten Schlüssels und verifiziert anschließend das Erzeugnis mithilfe des öffentlichen Schlüssels des Empfängers. Bei der Entschlüsselung sowie Verifizierung kann es zu einer CorruptDataException kommen. Das Produkt sind serialisierte Daten als Zeichenkette.

Die Sicherheitsschicht bietet somit alle von der Serviceschicht benötigten Funktionen an. Die Sicherheitsbibliothek SJCL sorgt dabei für die asymmetrische Kryptografie.

4.2.5 Monitoringschicht

Um den Zustand des Chord-Overlays aus lokaler Sicht zu verfolgen und aufzuzeichnen ist eine neue Schicht, die Monitoringschicht, entstanden. Die Monitoringschicht des Chordmoduls ist für das Einsammeln von Informationen bezüglich eines Operationtyps zuständig. Eine Operation ist nachfolgend definiert als ein vom lokalen Knoten initiierter Vorgang in Chord. Zu diesen Vorgängen gehören:

- Alle Chord Methoden: join, leave, retrieve, insert, remove und sendMsg.
- Chord Instandhaltungsjobs: stabilize, check predecessor und fix finger.
- Replikationsvorgänge.
- Schließungswunsch einer Verbindung

Jede der oben beschriebenen Operationen versendet Nachrichten an weitere Knoten, um diese Operation durchzuführen. Diese Nachrichten entsprechen somit den in der abstrahierten Kommunikationsschicht, siehe Abschnitt 4.2.1, beschriebenen Operationen.

Die Ausnahme bildet handleCall, da diese keinen Nachrichtenaustausch auf Chordebene vorsieht.

Die Einsammlung der Informationen, welche die Grundlage für deren Auswertung ist, beginnt mit der *OperationID*. Bei dem Start einer Operation wird zunächst eine OperationID instanziiert, welche eine initiale ID und eine aktuelle ID, sowie einen Zeitstempel enthält. Die aktuelle ID ist in jeder Nachricht, die über die Kommunikationsschicht versendet wird, enthalten. Aus diesem Grund wird diese verwendet, um eingehende Nachrichten in ihrem Kontext zu identifizieren. Der Aufbau der ID's, nachfolgend *SingleID* genannt, besteht aus genau vier *Token*:

- *initiatorID*: Chord ID des Initiators.
- *NumOfOperation*: Der Initiator zählt seinen internen Operation-counter bei jeder neuen Operation hoch, sodass initiatorID und NumOfOperation eindeutig sind.
- *MessageForThisOperation*: Nachrichten, die vom Initiator für diese Operation versendet wurden. Für die Durchführung einer Operation sind unter Umständen mehrere Nachrichten notwendig. Beispiel anhand der Chord Methode *insert*: erst findSuccessor dann insertEntry.
- *HopCountForThisMessage*: Anzahl an Knoten, die diese Nachricht bereits durchlaufen ist. Diese wird von *jedem* Knoten beim Erhalt der Nachricht hochgezählt.

Nun können Knoten anhand der OperationID einer Nachricht dem Verlauf der Nachricht die oben genannten Informationen abgewinnen. Jedoch werden mehr Informationen zu einer Aktion einer Operation benötigt, um Statistiken zu erstellen. Weshalb jeweils beim Empfang und Versand von Nachrichten Details der Aktion gesichert werden. Um solche Details zu sichern, werden *OperationDetail*-Instanzen erzeugt, welche gewisse Informationen für eine SingleID bereithalten.

Zu diesen gehören *i*) ein Zeitstempel *ii*) ein *isOpen* Flag, welches anzeigt, ob die Operation noch offen oder hiermit geschlossen wird *iii*) der *TransceiverStatus*, d.h. ob dies eine sendende oder empfangene Aktion ist oder eben keines von beiden *iv*) der *Nachrichtentyp*, also auf welcher Chord Nachricht wird eine Aktion ausgeführt *v*) die Nachrich-

tengröße in byte, welche der Länge des Chiffretexts entspricht und *vi*) die Krypto-Zeit in ms, welche, je nach Aktion, der Zeit entspricht, die benötigt wird, um das Paket zu verschlüsseln und signieren oder zu entschlüsseln und verifizieren .

Diese Details werden zusammen mit ihrer zugehörigen OperationID in diversen Datenstrukturen des *OperationPools* gesichert. Um veraltete Informationen aus dem OperationPool zu löschen wird periodisch ein *cleaner* gerufen. Das maximale Alter von gesicherten Informationen, sowie die Periode des cleaners wird in der Konfiguration, detailliert in Abschnitt 6.2, hinterlegt.

Die gesammelten Informationen können über Methoden Aufrufe an den OperationPool gebündelt und zurückgegeben werden. Hier sei angemerkt, dass alle Informationen lokal gesammelt werden, eine Aggregation der Daten jedoch möglich ist, da viele Statistiken nur über lokal initiierte Operationen erstellt werden. Vor jeder Informationenanfrage wird zunächst der Cleaner gerufen, um eventuell veraltete Daten zu löschen. Folgende Anfragen sind verfügbar:

1. Anfragen über alle gesammelten Daten abhängig von einem gewähltem Nachrichtentyp: *i*) absolute Anzahl empfangener / versendeter Nachrichten *ii*) empfangene / -versendete Nachrichten pro Sekunde *iii*) empfangene / versendete Byte pro Sekunde *iv*) insgesamte Krypto-Zeit für das empfangen / versenden von Nachrichten und *v*) durchschnittliche Krypto-Zeit für das empfangen / versenden von Nachrichten.

2. Status initiierter Operationen für einen gewählten Operationstyp: *i*) Anzahl initiierter Operationen *ii*) Aufzählung und Anzahl sowie Durchschnitt versendeter Nachrichten *iii*) Anzahl erfolgreich geschlossener Operationen und deren Rate *iv*) Anzahl erfolglos geschlossener Operationen und deren Rate und *v*) Anzahl noch offener Operationen, sowie deren Rate.

3. Hop-Anzahl initiierter Operationen: *i*) Liste von Hop-Anzahlen und deren Durchschnitt eines gewählten Nachrichtentyps *ii*) Liste von Hop-Anzahlen und deren Durchschnitt eines gewählten Operationstyps und *iii*) Liste von Hop-Anzahlen und deren Durchschnitt eines gewählten Operationstyps und Nachrichtentyps.

Insgesamt liefert die Monitoringschicht anderen Modulen an den Zustand des Chord-Overlays aus lokaler Sicht zu verfolgen bzw. eine Momentaufnahme zu erhalten. Genau dies ist die Aufgabe des in Abschnitt 6.4 beschriebenen Monitoringmoduls.

Hiermit ist der Abschnitt um die modifizierte Implementierung von OpenChord abgeschlossen. Es waren zunächst viele grundlegende programmiertechnische Umänderungen notwendig, bedingt durch die Einschränkungen des GWT-Frameworks (siehe Abschnitt 2.3.1). Weiter erfolgte die Implementierung einer neuen Kommunikationsschicht und Einführungen weiterer Schichten. Eine komplette Auflistung aller im Rahmen dieser Arbeit durchgeführten Anpassungen enthält der nächste Abschnitt 4.3.

4.3 OpenChord Anpassungen

Der Grundstein des in Abschnitt 4.2 dargestellten Chordmoduls ist die Chord-Implementierung OpenChord [LK06]. Welche Schritte in den verschiedenen Schichten für diese Arbeit vorgenommen wurden, fasst Tabelle 4.2 zusammen. Alle aufgeführten Punkte wurden mit dem Ziel eine sichere Chatplattform zu erschaffen durchgeführt.

Auf der Kommunikationsschicht wurden zunächst alle ungenutzten Protokolle entfernt. Dies wären die Protokolle *socket*, *RMI* und *local*. Das einzige Kommunikationsprotokoll ist das neu implementierte Browser-to-Browser über PeerJS-Protokoll, welches dem ehemaligen socket Protokoll ähnlich ist. Des Weiteren wurde die URL, welche zur Knotenadressierung verwendet wurde, entfernt. Die Knotenadressierung wird nun auf die Chord ID reduziert, welche gleichzeitig der PeerJS-ID entspricht. Der Nachrichtentyp *connect* wurde entfernt, da der Verbindungsaufbau nun über PeerJS erfolgt. Das Pendant zu dem connect ist das open Event der peer-Instanz.

Neu hinzugefügt wurde die AV-Streaming Unterstützung durch PeerJS. Des Weiteren ist der Nachrichtentyp message eingeführt worden. Dies ist eine Anwendungsschichtnachricht an einen Knoten mit einer angegebenen Chord ID. Als letzte Neuerung der Kommunikationsschicht ist eine Operation ID eingeführt worden, die alle verschickten

Tabelle 4.2: OpenChord Anpassung.

Schicht	*Entfernt / Angepasst*	*Hinzugefügt*
Kommunikation	• veränderte Knotenadressierung • Connect Nachrichten entfernt • Kom-Protokolle entfernt, B2B Protokoll hinzugefügt	• AV-streaming via PeerJS • Message Nachrichtentyp • Nachrichten enthalten Operation ID
Service	• ID aus Identity statt URL • Geänderter join Vorgang • Angepasster create Vorgang • Anpassung DHT Einträge	• Zertifikatinfrastruktur / -management • Anwendungsschicht Nachrichten • AV-Streaming zwischen Peers
Sicherheit		• Verschlüsselte Kommunikation • Operationen auf Zertifikaten • Sicherheit in DHT Einträgen
Monitoring		• Operation Pool für Operation ID's
Alle	• Event-based Callbacks statt Threading	

Nachrichten enthalten. Anhand dieser Operation ID werden Monitoringinformationen für die Monitoringschicht gesammelt.

Die Serviceschicht erfuhr die meisten Anpassungen. Statt die Chord ID durch eine URL abzuleiten, ist es nun eine Identität, die anhand der Sicherheitsschicht eine Chord ID generiert. Die Identität besteht aus einem Benutzernamen und einem gehashten Passwort eines Benutzers. Der join Vorgang wurde dahingehend angepasst, dass es nun vier join Varianten gibt. Hierfür gibt es zwei Unterscheidungsmerkmale. Zum einen soll einem öffentlichen Chord-Ring oder einem privaten Chord-Ring, unter der Angabe einer Bootstrap ID, beigetreten werden können. Zum anderen soll ein Login mit bereits

im Chord-Ring gespeicherten Identitätsmerkmalen in Form eines Zertifikats oder eine Registrierung einer nicht vorhandenen Identität im Chord-Ring vollzogen werden. Bei dem öffentlichen join sei angemerkt, dass der Webserver Bootstrap ID's sammelt, um neu eintretenden Knoten eine Liste mit möglichen Eintrittsknoten zu senden. Als Alternative zum öffentlichen join wäre der private join, wo manuell eine Bootstrap ID eingegeben werden muss, von einem Knoten, der an einem Chord-Ring angemeldet ist. Zur Eröffnung eines neuen privaten Chord-Rings steht der angepasste create Vorgang zur Verfügung. Die letzte Anpassung der Serviceschicht betrifft die DHT Einträge. Zunächst wurde die DHT von einem Mapping Key $\rightarrow$ Set<Value> auf Key $\rightarrow$ Value geändert, sodass für einen Schlüssel maximal ein Wert hinterlegt werden kann. Weiter werden DHT Einträge nun in drei Kategorien eingeteilt: unsigniert, signiert und signiert-verschlüsselt.

Der Serviceschicht wurden drei neue Features hinzugefügt. Zum einen wurde eine Zertifikatinfrastruktur eingeführt, wobei jede Identität ein Zertifikat (an zwei Stellen) im Chord-Ring hinterlegt. Das Zertifikatmanagement bietet eine Suche von Zertifikaten, Modifikation und Löschung des eigenen Zertifikats an. Das zweite Feature bietet einen Nachrichtenversand für Anwendungsschichtnachrichten an eine Chord ID an. Das letzte Feature ermöglicht das starten und stoppen von Mediastreams mit einem entfernten Knoten unter Angabe seiner Chord ID.

Zwei neu eingeführte Schichten sind die Sicherheits- und Monitoringschicht. Die Sicherheitsschicht bietet zunächst die Möglichkeit an eindeutige Schlüsselpaare unter Angabe einer Identität zu generieren. Ein weiteres Feature ist die Verschlüsselung der Kommunikation. Weiter ist das erstellen sowie verifizieren von Zertifikaten möglich. Für die neu erstellten DHT Kategorien stehen außerdem die Ver- / Entschlüsselung und Signatur bzw. Verifizierung der Einträge zur Verfügung.

Die Monitoringschicht hingegen sammelt in einem OperationPool alle Operation ID's ein und bietet eine sortierte Bündelung der Daten an, um gewisse Statistiken zu führen.

Die letzte Anpassung bezieht sich auf alle Schichten. Durch die Ausführung der Software im Browser musste das verwendete Threading in Event-based Callbacks umgebaut werden.

Das Kapitel um das Chordmodul zeigte zunächst den durch OpenChord gelegten Grundstein in Abschnitt 4.1. Auf Basis von OpenChord erfolgte eine modifizierte Implementierung. Zu den Hauptbestandteilen zählen das Hinzufügen eines neuen Kommunikationsprotokolls, wodurch die Kommunikation zwischen Browsern möglich ist, die Anpassung der Serviceschicht, um eine Chatplattform mitsamt Login zu unterstützen und infolge dessen die Einführung zwei weiterer Schichten: der Sicherheitsschicht und der Monitoringschicht. Eine Auflistung der Anpassungen enthält letztendlich der letzte Abschnitt 4.3. Das nächste Kapitel umfasst Module, die das Chordmodul nutzen, um eine Chatplattform zu schaffen.

4.4 Beiträge des Autors

Die Beiträge des Autors in diesem Kapitel sind Thema dieses Abschnitts. Da das Chordmodul auf die OpenChord-Implementierung aufbaut, ist eine strikte Differenzierung notwendig. Die Mechanismen und Architektur von OpenChord wurden bei der Portierung ins GWT-Framework gewahrt und alle Anpassungen sowie neuen Features sind Beiträge des Autors. Details der Anpassung sind Abschnitt 4.3 zu entnehmen. Im Rahmen dieser Anpassungen sind vom Autor drei Bibliotheken eingebettet worden, die nicht im Rahmen dieser Arbeit erstellt wurden. Dies sind die folgenden.

Für die (De-) Serialisierung, welche von der Kommunikationsschicht benötigt wird, sorgt der **JsonSerializer** [Kfu11].

Die **PeerJS**-Bibliothek [MB14c] hat eine Client- und Server-Komponente. Die Client-Komponente kapselt die WebRTC API, um den Aufbau und Unterstützung der Echtzeitkommunikation zwischen zwei Browserknoten zur Verfügung zu stellen. Die Server-Komponente hingegen sorgt für die Vermittlung der Clients untereinander. Diese Komponente vermittelt lediglich seine Clients, sodass der Datenverkehr nach dem Aufbau zweier Knoten direkt, d.h. P2P, erfolgt.

Die letzte Bibliothek wird in der Sicherheitsschicht verwendet und heißt **SJCL** [ESB14]. Die beschriebene PublicKey-Komprimierung (siehe Abschnitt 4.2.4) ist bekannt, die

Einbindung dieser in die SJCL-Bibliothek hat jedoch im Rahmen dieser Arbeit stattgefunden. Weiter ist die Idee der Nutzung des öffentlichen Schlüssels als Chord ID nicht neu. Diese wird in [GPM+08] vorgestellt. Der Beitrag des Autors ist die Implementierung einer deterministischen Generierung eines Schlüsselpaares unter Nutzung des pbkdf2-Verfahrens und der publicKey-Komprimierung.

Kapitel 5

Modul: Anwendung und Chat

Dieses Kapitel erläutert die Aufgaben und Bedeutung der beiden Module Anwendung und Chat. Das Anwendungsmodul, welches in Abschnitt 5.1 behandelt wird, beinhaltet den Einstiegspunkt des GWT-Frameworks und übernimmt aus diesem Grund die Instanziierung und Koordinierung der Module. Das Chatmodul, dessen Spezifizierung in Abschnitt 5.2 erfolgt, ist in viele Untermodule unterteilt, die mithilfe des Chordmoduls die Logik der Chatplattform realisieren. Der letzte Abschnitt 5.3 enthält eine Auflistung der Beiträge des Autors bezüglich des hier beschriebenen Moduls.

5.1 Anwendung

Das Anwendungsmodul ist der Einstiegspunkt der Software. Zum Aufgabenbereich gehört die Erstellung, Bekanntmachung untereinander und Koordinierung der Module.

Zunächst wird der Programmeinstieg beschrieben. Viele Module benötigen bei der Instanziierung eine gültige und geladene Konfiguration. Aus diesem Grund wird zuallererst gemäß dem Konfigurationsmodul (siehe Abschnitt 6.2) die Konfiguration vom Server geladen. Erst wenn dieser Vorgang erfolgreich abgeschlossen ist, folgt das Anschließende. Um dem Benutzer eine Login Maske zu gewähren werden das GUI- und Chordmodul instanziiert und der GUI wird das Laden der Login Maske aufgetragen. Verzeichnet die

GUI einen Login Wunsch, so wird dieser an das Anwendungsmodul via Schnittstelle weitergegeben, woraufhin der zuständige Befehl (join oder create) des Chordmoduls angestoßen wird. Über das Chord joined Callback wiederum erhalten wir eine Rückmeldung des join bzw. create Vorgangs.

Enthält die Rückmeldung eine Fehlermeldung, so wird ein Chord leave veranlasst, falls dieser nicht bereits geschehen ist. Außerdem werden alle Module entladen, wodurch diese Aufräumtätigkeiten durchführen werden können, jedoch kein Datenaustausch mehr möglich ist. Nach einer konfigurierbaren Zeit wird eine Neuladung der HTML Seite veranlasst, wodurch der Nutzer zu dem Einstiegspunkt der Software zurück gelangt.

Ist der join Vorgang jedoch erfolgreich gewesen, so werden zunächst das Chat- und Monitoringmodul instanziiert. Hierbei werden bereits die GUI Schnittstellen übergeben, sodass die Module auch untereinander kommunizieren können. Anschließend werden der GUI jeweils benötigte Chat Schnittstellen übergeben, um Benutzereingaben bezüglich des Chats direkt an das entsprechende Modul weiterleiten zu können. Weiter wird der GUI der joined Zustand kommuniziert, sodass diese die Benutzeroberfläche angemessen ändern kann. Abschließend erhält das Chatmodul den load Befehl, wodurch die Chatmodule ihren Dienst aufnehmen.

Um nun entfernte Knoten oberhalb des Chordmoduls als Objekte auszudrücken, wurde die *Peer* Klasse eingeführt. Jedes Peer-Objekt enthält unter anderen die eindeutige Chord-ID, den Benutzernamen und das Zertifikat, sofern bereits erhalten, welches die Abbildung Chord ID auf Benutzernamen bestätigt. Da die Chord ID eindeutig ist, wird ein Peer anhand seiner ID identifiziert. Bekannte Peers werden in einen *PeerPool* gesichert, die eine Abbildung von Chord ID auf ein Peer-Objekt ist. Treffen nun Zertifikate aus verschiedenen Suchanfragen, initiiert durch das Chatmodul, auf, so wird das zugehörige Peer-Objekt aktualisiert. Dies hat zum Ziel keine bloßen Chord ID's an Module weiterzugeben. So soll aus einer erhaltenen Chord ID eines Chord Callbacks immer zunächst das dazugehörige Peer-Objekt aus dem PeerPool extrahiert bzw. ein neues Peer-Objekt anhand der ID erstellt werden.

Die Chat-spezifischen Anfragen, welche in Abbildung 3.1 zu sehen sind, sind meist delegate Methoden, die an das Chordmodul weitergereicht werden. Hierzu gehören die Suche

nach einem Benutzernamen oder einer ID und deren Antwort in Form eines Zertifikats oder *null*, das Holen eines Zertifikats für ein Peer-Objekt und das Starten bzw. Stoppen eines AV-Streams mit einem Peer. Beim Speichern bzw. Holen der Kontaktliste wird zunächst aus der übergebenen Zeichenkette, die den Schlüssel repräsentieren soll, ein Chord-Key-Objekt gebaut und anbei die eigene Chord ID angehängt. Der Schlüssel hat dann die Form: <*ChatModulSchlüssel*>#<*EigeneChordID*>. Die Kontaktliste soll hierbei als ein signiert-verschlüsselter DHT Eintrag gesichert werden. Schließlich erfolgt ein Aufruf an die insert bzw. retrieve Methode des Chordmoduls und die Rückmeldung wird dem aufrufenden Chatmodul übersandt. Das Format von Nachrichten, die über das Chordmodul versendet bzw. empfangen werden, folgen einem Muster. Dieses Muster sieht wie folgt aus: <*Modulname*>#<*Nachricht*>. Durch die Verwendung wird sichergestellt, dass alle empfangenen Nachrichten durch die Inspektion des ersten Tokens an das richtige Modul weitergeleitet werden können. Zusätzlich teilt das Anwendungsmodul dem Kontaktlisten Untermodul mit, wenn eine Aktivität eines Peers erfolgt, die darauf schließen lässt, dass dieser im Chord-Ring angemeldet ist. Dies erfolgt beispielsweise beim Empfang einer Nachricht des Peers bzw. dem ACK Empfang einer von uns versendeten Nachricht. Dies ist lediglich eine Optimierung, um unnötigen Nachrichtenfluss, initiiert durch das Kontaktlistenmodul, zu verhindern. Mehr Details zum Kontaktlistenmodul folgen im Abschnitt 5.2.2.

Schließlich muss das Anwendungsmodul auf Wunsch des GUI-Moduls bzw. der Benutzereingabe den Nutzer ausloggen können. Hierfür wird allen geladenen Modulen zunächst die Chance gegeben abschließend Informationen z.B. im Chord-Ring zu sichern. Zu diesem Zweck wartet ein *waiter* auf ein *ready* Signal von den Modulen. Um nicht zu lange auf diese Signale zu warten, wird eine konfigurierbare obere Schranke für die Wartezeit eingeführt. Nachdem sich entweder alle Module als bereit gemeldet haben oder die obere Schranke der Wartezeit erreicht ist, erfolgt ein Chord leave. Schließlich werden alle Module entladen und die HTML Seite neu geladen.

Anschließend zum Anwendungsmodul folgt nun das Chatmodul.

5.2 Chat

Das Chatmodul der Implementierung umfasst vier Untermodule. Diese Module sind die Kontaktsuche, Kontaktliste, Single-Chat und Audio / Video-Chat, nachfolgend AV-Chat genannt. Die genannten Untermodule werden in den nachfolgenden Abschnitten 5.2.1, 5.2.2, 5.2.3 und 5.2.4 näher spezifiziert. Um den modularen Aufbau zu wahren, ist die Einbindung weiterer Untermodule möglich. Eingehende Benachrichtigungen des Anwendungmoduls, im Detail der Auslogwunsch und das Laden bzw. Entladen des Moduls, wird an alle Untermodule weitergeleitet, sodass diese auf das Event reagieren können.

Es erfolgt nun die detaillierte Vorstellung der Untermodule.

5.2.1 Kontaktsuche

Das Untermodul Kontaktsuche implementiert eine Schnittstelle des GUI-Moduls, wodurch Kontaktsuchanfragen des Benutzers in diesem Modul eintreffen. Der Methodenkopf ist der folgende *requestSearch(String search, boolean isPublicKeySearch)*.

Die Implementierung des Moduls wird durch eine Suche nach Zertifikaten im Chord-Ring realisiert. Des weiteren sieht diese vor, dass immer nur eine Suche zur selben Zeit verarbeitet wird. Ist somit eine Suche im Gange und requestSearch wird erneut gerufen, so kehrt diese ohne weitere Verarbeitung zurück. Wird gerade keine Suchanfrage ausgeführt, so wird die Suche an das Anwendungsmodul weitergeleitet, welches die Suche im Chord-Ring anstößt und Ergebnisse über die Methode *searchResult(Peer, SearchSupportFailure)* ausgibt. Ein Suchtreffer wird direkt nach dem Eintreffen zur Verarbeitung an die Kontaktsuche weitergegeben. Das Ende der Suche bzw. der letzte Treffer wird durch ein null-Objekt als Peer Parameter signalisiert. Demnach werden Suchtreffer mit einem Peer-Objekt $\neq$ null bei einer Suche nach einem Benutzernamen direkt an das GUI-Modul weitergegeben und signalisiert, dass dies nicht das Ende der Suche ist. Ist das Peer-Objekt null oder handelt es sich um eine Suche nach einem öffentlichen Schlüssel, so wird dem GUI-Modul der Treffer und das gleichzeitige Ende der Suche mitgeteilt.

Wird ein Fehler per searchResult mitgeteilt, so wird dieser lediglich protokolliert. Das Ende der Suche wird hierdurch nicht beeinträchtigt.

Um dem GUI-Modul die Möglichkeit zu bieten, Suchtreffer, die die Kontaktliste bereits beinhaltet, kenntlich zu machen, enthält die Kontaktgesuch-Schnittstelle eine Methode zu eben dieser Prüfung.

5.2.2 Kontaktliste

Das Modul *Kontaktliste* ist für alle Funktionalitäten rund um eine klassische Kontaktliste zuständig. Um die Funktionen zu gewähren, kommuniziert diese über Schnittstellen mit dem GUI-Modul (siehe Abschnitt 6.1) und dem Anwendungsmodul, beschrieben in Abschnitt 5.1. Eingehende und erfolgreich versendete Nachrichten erreichen die Kontaktliste über eine Schnittstelle mit dem Single-Chat-Modul.

Um Modulen wie beispielsweise dem GUI-Modul Änderungen der Kontaktliste mitzuteilen, ist eine Listener Schnittstelle implementiert worden. Interessierte Module können diese Schnittstelle implementieren, um nach einer Registrierung zunächst eine Kopie der Kontaktliste und anschließend Änderungen dieser zu erhalten. Zu den Aktualisierungen gehören Benachrichtigungen über das Hinzufügen eines neuen Kontakts, Kontaktupdates, d.h. Änderung eines Kontaktattributs durch jüngste Ereignisse, sowie eingehende bzw. erfolgreich versendete Nachrichten.

In den nachfolgenden Abschnitten werden die wichtigsten Definitionen und Funktionen der Modulimplementierung beschrieben. Zunächst gilt es einen Kontakt zu definieren und benötigte Eigenschaften zu spezifizieren. Anschließend wird der Vorgang des Ladens bzw. Entladens erörtert. Weiter werden zwei Aufgaben des Moduls mithilfe von periodischen Jobs gelöst. Diese wären die Prüfung des Onlinestatus der Kontakte und die Sicherung der eigenen Kontaktliste. Ein wichtiger Prozess, nämlich die Hinzufügung von neuen Kontakten erfolgt zum Abschluss.

Definition eines Kontakts

Ein Kontakt ist nachfolgend ein der Kontaktliste hinzugefügter Peer, der erweiterte Attribute besitzt. Jeder Kontakt (im Quellcode: *buddy*) kapselt hierzu das darunterliegende Peer-Objekt und fügt diesem drei weitere Attribute hinzu. Diese wären *i*) ein Onlinestatus mit drei Werten: online, offline sowie undefiniert *ii*) ein Zeitstempel, wann dieser Kontakt zuletzt online gesehen wurde und *iii*) ein Flag, welches anzeigt, ob die Freundschaft beiderseits akzeptiert wurde.

(Ent-)laden des Moduls

Erhält die Implementierung des Kontaktlistenmoduls die Anweisung das Modul zu laden, so wird eine potentiell gespeicherte Kontaktliste aus dem Chord-Ring abgerufen. Hierfür wird lediglich eine Methode der Anwendungsmodulschnittstelle gerufen. Der Callback mit der Antwort enthält entweder eine Kontaktliste oder null bzw. einen Fehlerbericht.

Wurde eine Kontaktliste abgerufen, so werden zunächst alle Zertifikate der Kontakte nachgeladen (diese werden nämlich nicht zusammen mit der Kontaktliste gesichert) und registrierten Listenern wird über dieses Event berichtet. Andernfalls wird eine leere Kontaktliste initialisiert. Abschließend werden die nachfolgend beschriebenen periodischen Jobs gestartet.

Bei der Anweisung letzte Schritte vor dem Ausloggen einzuleiten, werden eine sofortige Sicherung der Kontaktliste beantragt und die periodischen Jobs gestoppt. Ist die Sicherung der Kontaktliste geglückt, so wird dem Waiter der Abschluss signalisiert.

Periodischer Job: OnlineChecker

Die Notwendigkeit und Hintergrund des periodischen online checkers ist der folgende. Das Chordmodul bietet keine Funktionalität an, um zu erkennen, ob ein bestimmter Knoten im Chord-Ring angemeldet ist oder nicht. Der online Status ist eine der grundlegenden Informationen, die eine klassische Kontaktliste enthalten muss.

Um diese Funktionalität auf einer höheren Ebene zu realisieren erfolgte die Implementierung eines periodischen Jobs, welcher Kontakte anpingt, deren Anwesenheit zuletzt vor konfigurierbarer langen Zeit festgestellt wurde. So wird mit jedem Aufruf des Jobs das Attribut *zuletzt online* jedes Kontakts der Kontaktliste geprüft. Liegt der gesicherte Zeitstempel laut Konfiguration zu weit in der Vergangenheit, so wird eine Nachricht mit dem Inhalt *buddy_ping* an diesen versendet. Wird als Callback ein Fehler zurückgegeben, so ist der online Status des betroffenen Knotens auf offline zu setzen.

Um das Nachrichtenaufkommen zu verringern, wird jede eingehende bzw. geackte Nachricht als Lebenszeichen des Kontakts aufgenommen und durch das Anwendungsmodul an das Kontaktlistenmodul weitergeleitet, wo das „zuletzt gesehen“ Attribut des jeweiligen Knotens aktualisiert wird. Um ein Online-kommen eines Kontakts zeitnah zu erkennen, werden auch als offline markierte Kontakte in den Job einbezogen. Dies verursacht jedoch kein erhöhtes Nachrichtenaufkommen. In Chord werden Nachrichten des Typs *message* direkt an den Empfänger versendet und ist dieser nicht angemeldet, so signalisiert dies die Kommunikationsschicht von Chord und der Nachrichtenversand bleibt aus. Diese Signalisierung erfolgt durch den Vermittlungsserver von PeerJS.

Periodischer Job: BuddylistSaver

Um die aktuelle Kontaktliste bei jedem Login laden zu können, muss die Sicherung dieser im Chord-Ring aktuell gehalten werden. Durch den Auslogvorgang wird, wie weiter oben in Abschnitt 5.2.2 beschrieben, die aktuelle Kontaktliste im Chord-Ring gesichert. Jedoch passiert dies nicht, wenn der Benutzer beispielsweise den Browser schließt, ohne sich ordnungsgemäß auszuloggen. Aus diesem Grund wurde der periodische Job BuddylistSaver implementiert.

Die periodische Aufgabe prüft nach einer konfigurierbaren Periode, ob sich die Kontaktliste mitsamt Chatlogs seit dem letzten Aufruf geändert hat. Ist dies der Fall, so wird eine Speicherung dieser veranlasst. Hierzu hält der Job eine Referenz auf das zuletzt erfolgreich gespeicherte Objekt und gleicht dieses mit dem aktuellen ab. Zusätzlich gibt es die Möglichkeit die Prüfung auf Abruf zu erzwingen und abschließend den Job abzubrechen. Dies ist beim Ausloggen vonnöten.

Prozess: Kontakt hinzufügen

Eine Kernaufgabe des Kontaktlistenmoduls ist die Möglichkeit einen Kontakt hinzuzufügen. Dieser Prozess wird nun erläutert.

Wird dem Kontaktlistenmodul per Schnittstelle der GUI mitgeteilt, dass ein Peer der Kontaktliste hinzugefügt werden soll, so wird dieser Kontakt anhand des Peer-Objekts erstellt und dessen Freundschaftsflag auf *false* gesetzt. Fortan schickt der OnlineChecker dem Kontakt periodisch eine *requestAdd* Nachricht, bis diese beantwortet wurde. Das Kontaktlistenmodul des Kontakts erhält diese Nachricht und leitet diese Anfrage an das GUI-Modul weiter, welche den Benutzer über diese Kontaktanfrage informiert. Die Antwort des GUI-Moduls erfolgt in Form einer Nachricht an den anfragenden Peer von der Form *responseAdd#true* oder *responseAdd#false*. Erhält letztendlich der anfragende Knoten eine responseAdd Nachricht, so kann dieser Kontakt bezüglich des Freundschaftsflags aktualisiert werden und muss bei einer negativen responseAdd Nachricht wieder aus der Kontaktliste gelöscht werden.

Wie das Chatting mit neu-hinzugefügten Kontakten implementiert ist, enthält der nachfolgende Abschnitt 5.2.3.

5.2.3 Single-Chat

Die Aufgabe des Single-Chat-Moduls ist das Anbieten des Versands einer text-basierten Nachricht an einen Kontakt, sowie der Empfang von Textnachrichten von Kontakten. Die Definition eines Kontakts ist dem Abschnitt 5.2.2 zu entnehmen.

Das Single-Chatmodul bietet der GUI eine Schnittstelle an, worüber Sendewünsche an einen potentiellen Kontakt übermittelt werden. Hierzu prüft die Single-Chatmodul Implementierung zunächst mithilfe einer Schnittstelle zum Kontaktlistenmodul, ob der gewünschte Zielknoten in der Kontaktliste aufgenommen und das Freundschaftsflag gesetzt ist. Ist dies der Fall, so wird ein Sendewunsch mit dem Inhalt *chat_message#<nachricht>* an die Schnittstelle des Anwendungsmoduls weitergegeben. Per Callback erhält der Ab-

sender eine Empfangsbestätigung und kann dem Kontaktlistenmodul per Schnittstelle mitteilen, dass eine Nachricht erfolgreich an einen angegebenen Kontakt versendet wurde.

An das Single-Chat-Modul weitergeleitete Nachrichten sind Textnachrichten eines potentiellen Kontakts. Auch hier findet zunächst die Prüfung statt, ob der Absender Teil der Kontaktliste ist. Ist die Prüfung positiv, d.h. der Absender ist ein Kontakt, so leitet das Modul die empfangene Textnachricht an das Kontaktlistenmodul weiter.

Da die Implementierung des Single-Chat-Moduls keine Initialisierungen benötigt, muss bei dem Laden des Moduls nichts erledigt werden. Außerdem müssen beim Auslogvorgang keine Zustände gesichert werden, sodass dem waiter sofort der Abschluss signalisiert wird.

Anschließend zum Single-Chat wird nun das AV-Chat-Modul vorgestellt.

5.2.4 Audio/Video-Chat

Das Audio / Video-Chat-Modul hat zur Aufgabe AV-Streaming Anfragen korrekt weiterzuleiten, um somit einen AV-Stream zwischen zwei Kontakten aufzubauen.

Wird eine Streaminganfrage des GUI-Moduls vernommen, so findet in der Implementierung zunächst ein *handshake* statt. Hierzu sendet der anfragende Knoten zunächst eine *request_avstream* Nachricht an den gewünschten Kontakt. Dieser leitet die Anfrage an sein GUI-Modul weiter, um diese vom Benutzer beantworten zu lassen. Die Antwort des Benutzers wird in Form einer *response_avstream* Nachricht an den Initiator versendet. Der Initiator reagiert auf die Antwort und startet, falls die Antwort positiv ist, einen AV-Stream mithilfe einer Schnittstelle des Anwendungmoduls. Während der Initialisierung des *Calls* wird der Benutzer vom Browser gefragt, ob und auf welche angeschlossenen Mediengeräte zugegriffen werden darf. Das Dialogfenster ist browserspezifisch und unterscheidet sich somit von Browser zu Browser. Schließlich vermittelt das Anwendungsmodul einen eingehenden Stream des Gegenüber, den eigenen Stream und den Kontakt. Die Streamobjekte sind Zeichenketten, die URL-Blobs enthalten, und können

von HTML5 Video-Elementen wiedergegeben werden. Mehr dazu ist dem GUI-Modul in Abschnitt 6.1 zu entnehmen.

Ist ein AV-Streaming mit einem Kontakt im Gange, so kann ein Stoppen dessen vom GUI-Modul verlangt werden. Diese Anfrage wird dem Anwendungsmodul weitergegeben und nach dem erfolgreichen Abbau wird das Ereignis der GUI mitgeteilt, um den Stream zu schließen. Das Gegenüber des Streams verhält sich ähnlich nach dem Eingang eines stop-Stream-Ereignisses vom Anwendungsmodul.

Abschließend bleibt zu erwähnen, dass ähnlich zum Single-Chat-Modul keine Initialisierungen beim Laden des Moduls benötigt werden. Bei einem Auslogwunsch wird dem waiter somit auch sofort der Abschluss signalisiert.

5.3 Beiträge des Autors

Die Implementierung des Anwendungsmoduls aus Abschnitt 5.1 ist in seiner Gesamtheit ein Beitrag des Autors. Diese dient als Einstiegspunkt der Software und hilft bei der Kommunikation von und zum Chordmodul.

Das Chatmodul ist wie das Anwendungsmodul ein Beitrag des Autors. Alle Untermodule sind für die Bedürfnisse einer Chatplattform angepasst und implementiert.

Kapitel 6

Module: GUI, Konfiguration, Logger, Monitoring

Dieses Kapitel umfasst die letzten vier der insgesamt sieben Module. Diese ist zum einen die GUI, welche dem Benutzer eine grafische Benutzeroberfläche bietet und in Abschnitt 6.1 vorgestellt wird. Anschließend folgt die Konfiguration in Abschnitt 6.2, welche allen Modulen eine Parameterliste anbietet, die initial vom Server geladen wird. Das Loggingmodul, welches in Abschnitt 6.3 detailliert wird, bietet einen Loggerdienst für alle Module an. Im Anschluss folgt das Monitoringmodul in Abschnitt 6.4, welches Informationen über den Status des Chord-Rings erfasst und diese an eine Sammelstelle weitergibt. Abschließend fasst Abschnitt 6.5 die Beiträge des Autors zu diesem Kapitel zusammen.

6.1 Modul: GUI

Das GUI-Modul repräsentiert die grafische Benutzeroberfläche, welche der Benutzer nutzt. Das Ziel dieser ist eine Schaffung einer übersichtlichen Benutzeroberfläche, welche alle Funktionen der übrigen Module nutzt. Benutzerinteraktionen sollen also an die dafür vorgesehene Module weitergeleitet werden.

Im Einzelnen werden folgende Voraussetzungen an die GUI gestellt. Zunächst soll eine Login Maske existieren, welche verschiedene Logineinstellungen bieten soll. Hierzu zählt der Login in bzw. die Registrierung am öffentlichen Netzwerk und der Login in bzw. die Registrierung am privaten Netzwerk. Weiter muss nach einem erfolgreichen Beitritt eine fensterbasierte Chatoberfläche geladen werden. Die Anordnung der Fenster ist (größtenteils) dem Benutzer zu überlassen. Zu den Fenstern zählen mindestens die Kontaktliste, welche alle bekannten Kontakte enthält, eine Suchmaske, um neue Kontakte zu finden, Chatfenster, um mit bekannten Kontakten zu chatten und schließlich AV-Streaming-Fenster, um ein AV-Streaming mit einem bekannten Kontakt zu führen.

Die nächsten Abschnitte beinhalten die im Rahmen dieser Arbeit und gemäß den Voraussetzungen implementierte GUI.

Um die Funktionalitäten der übrigen Module nutzen zu können, enthält die abstrakte Klasse GUI eine Reihe von delegates[1], die die Weiterleitung von Benutzereingaben an die einzelnen Module gewähren. So existiert genau ein delegate pro (Unter-) Modul, welches das jeweilige Modul implementiert. Die Antworten auf die durch Benutzereingaben generierten Anfragen erhält die GUI dank implementierten Schnittstellen der entsprechenden Module. Die Implementierung der *init()* Methode soll eine Login Maske erzeugen und anzeigen. Die *notifyChordJoined()* Methode hingegen muss die Login Maske abbauen und die Chatoberfläche laden. Methodenköpfe für den Auslogwunsch und das Laden bzw. Entladen des Moduls sind analog zu allen bestehenden Chatmodulen zu verstehen.

Nachfolgend werden wichtige Details der Implementierung *GUIImpl* der abstrakten GUI Klasse vorgestellt.

Zunächst ist eine Internationalisierung einer Website d.h. der Benutzeroberfläche wichtig. Zu diesem Zweck ist der i18n[2]-Mechanismus vom GWT-Framework zum Einsatz gekommen. Der GWT-Mechanismus unterscheidet zwischen Konstanten und Nachrichten. Konstanten enthalten reine Mappings von einem Schlüssel auf eine Zeichenkette. Nachrichten enthalten darüber hinaus eine Eingabe, die im Kontext der Nachricht auftritt. Für

[1] Entspricht hier Funktionsweiterleitungen.
[2] i18n: internationalization, da zwischen i und n 18 Buchstaben liegen.

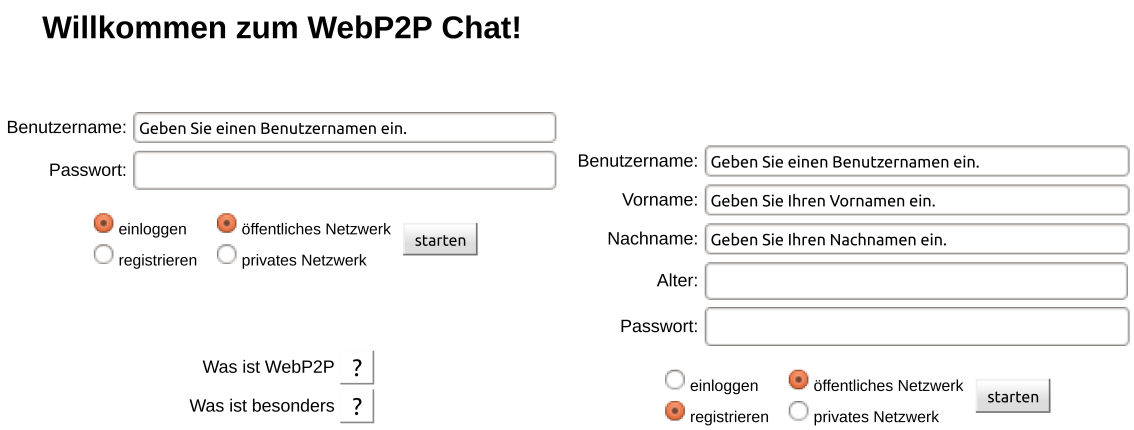
die Unterstützung einer Sprache werden jeweils zwei Java *property* Dateien angelegt. Eine für die Konstanten und die andere für Nachrichten. Per GWT-Skript lassen sich diese property Dateien in Java Interfaces übersetzen, welche per deferred Bindung Mechanismus verfügbar gemacht werden. Im Quellcode kann nun auf definierte Schlüssel zugegriffen werden, die zur Laufzeit auf die vom Browser gewählte Sprache, falls nicht vorhanden auf die als Standard definierte Sprache, abgebildet wird. Bisher unterstützte Sprachen sind Deutsch und Englisch. Weitere Sprachen lassen sich durch das Anlegen und Befüllen weiterer property Dateien leicht integrieren.

6.1.1 ViewController

Die Implementierung GUIImpl erfolgt mithilfe von *ViewControllern*. Um die Aufgaben eines ViewControllers zu beschreiben benötigen wir eine Definition einer *View*. Nachfolgend ist eine View ein Synonym für GWT-Panels mit GWT-Widgets, d.h. eine Zusammenfassung von Widgets in einem Layoutobjekt. Somit sind Views äquivalent zu *composites*. GWT-Widgets sind gängige GUI-Objekte, die vom GWT-Framework zur Verfügung gestellt werden. Zu diesen gehören Textboxen, Radioboxen, Drop-Down-Menüs etc.

Die Aufgaben eines ViewControllers sind nun Views zu instanziieren, anzeigen zu lassen und zu koordinieren. Um Views von der Browserengine rendern zu lassen, werden diese dem RootPanel, dem Wurzelpanel, via *RootPanel.get().add(view)* angehängt. Weiter ist ein ViewController für den Inhalt seiner Views verantwortlich und hält die Model-Objekte für diese. So bietet GWT die Nutzung von sogenannten *DataProvidern* bei *CellList*'s (dynamisch ladende Liste) oder *DataGrid*'s (klassische Tabelle) an. DataProvider halten Daten für registrierte Tabellen bzw. Listen bereit und aktualisieren diese bei Änderungen des Datenbestands. Zuletzt müssen ViewController auf durch Schnittstellen definierte Ereignisse instanziierter Views reagieren.

Speziell sind zwei ViewController entstanden. Der *LoginViewController* und der *ChatViewController*.



(a) LoginView mit Einloggen-Maske. (b) Registrierungsmaske.

Abbildung 6.1: LoginView der Chatplattform.

LoginViewController

Der LoginViewController enthält lediglich die LoginView, welche ein composite ist und aus einem Login Formular besteht. Per Schnittstelle erhält der LoginViewController Beitrittswünsche, welche an das Anwendungsmodul weitergeleitet werden. Ist der Login erfolgreich, so entlädt die GUIImpl diesen Controller, sodass auch die LoginView entladen wird. Passiert ein Fehler während des Vorgangs, so teilt die GUIImpl dies mit. Reagiert wird mit einem Fehlerbericht in der LoginView.

Der Aufbau der LoginView wird in Abschnitt 6.1.2 beschrieben.

ChatViewController

Die Instanziierung des ChatViewControllers erfolgt, sobald der Loginvorgang erfolgreich verlaufen ist. Außerdem werden bei der Instanziierung die diversen delegates von der GUIImpl übergeben, um diese bei Bedarf direkt (ohne Umwege über GUIImpl) zu rufen. Zunächst werden die grundlegenden Views geladen. Die *HeaderView* und *ContactListView*. Eine Beschreibung des Aufbaus beider Views, sowie nachfolgender Views erfolgt in Abschnitt 6.1.2. Die HeaderView ist eine Leiste am Kopfe der Seite, worüber

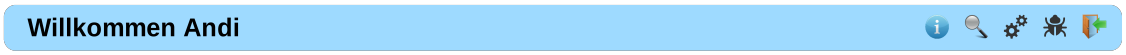


Abbildung 6.2: HeaderView: Kopfleiste der Chatplattform.

die Suche, das Monitoring und das Ausloggen initiiert werden kann. Die ContactListView repräsentiert eine Kontaktliste und nach deren Instanziierung ist eine Registrierung bei dem Kontaktlistenmodul fällig, um Kontaktlisten-updates zu erhalten. Alle Views bis auf die HeaderView, die der ChatViewController verwaltet, sind fensterartig aufgebaut. Diese Fenster enthalten eine Titelleiste mit einem Text und einen Schließen-Button. Mehr dazu folgt im Abschnitt 6.1.2. Eine weitere wichtige Komponente ist der *WindowSpaceDistributor*, welcher unter Anderen für die initiale Platzvergabe eines sich öffnenden Fensters zuständig ist. Weitere Informationen enthält Abschnitt 6.1.3

Nachdem die HeaderView und ContactListView angezeigt sind erfolgen alle weiteren Prozesse ereignisbasiert. So kann per Buttonklick auf das zugehörige Bild in der Headerleiste die Suche geöffnet werden. Nach einer erfolgreichen Suche können Suchtreffer zur Kontaktliste hinzugefügt werden, welche dadurch eine Benachrichtigung in Form eines nachfolgend beschriebenen *BuddyAddRequestViews* erhält. Weiter kann per Doppelklick auf einen Kontakt der Kontaktliste eine *ChatView* geöffnet werden. Allgemein werden nicht durch Benutzerinteraktionen erstellte Fenster durch eine blinkende Titelleiste hervorgehoben, um die Aufmerksamkeit des Benutzers auf diese zu lenken.

Um sich einen Überblick über die verwendeten Views und deren Zusammenspiel durch den ChatViewController zu verschaffen, werden diese nun in Abschnitt 6.1.2 vorgestellt.

6.1.2 Views

Im Kontext der ViewController sind Views ein wichtiger Bestandteil und werden in diesem Abschnitt vorgestellt.

Die LoginView des LoginViewControllers im Abschnitt 6.1.1 hält eine kurze Begrüßung, eine Login-Maske und Informationen über die Chatplattform bereit. Die Login-Maske

Abbildung 6.3: Kontaktliste.

Abbildung 6.4: Beispielzertifikat von Peter.

verändert sein Aussehen je nach Stellung der Radiobuttons. In der Abbildung 6.1(a) etwa zeigt die Login-Maske das Layout für das Einloggen in das öffentliche Netzwerk. Weiter zeigt Abbildung 6.1(b) die Erweiterung der Login-Maske zur Registrierungsmaske, wo neben der Angabe eines Benutzernamens noch zusätzlich der Vor- / Nachname und das Alter eingegeben werden kann. Per Buttonklick auf die Fragezeichen im unteren Drittel, erfolgt ein Popup mit den nötigen Informationen über die Chatplattform und deren Besonderheiten.

Die HeaderView in Abbildung 6.2 beinhaltet einen Willkommenstext, sowie verschiedene Buttons, um Funktionen der Plattform zu erreichen. Zu den Buttons gehören eine kurze Hilfe zur Plattform, die Suchfunktion, Einstellungen (nicht implementiert), Informatio-

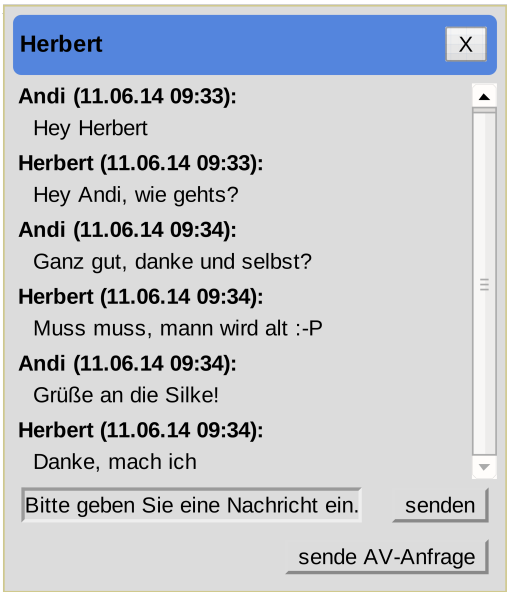


Abbildung 6.5: Chatfenster.

nen zum System sowie Monitoring und der Logout (v.l.n.r). Programmatisch ist die HeaderView ein horizontal ausgerichtetes Panel, welches Buttonklicks an einen registrierten Handler weitergibt.

Die übrigen Views, welche der ChatViewController koordiniert, sind fensterartige Views, die allesamt von der zu diesem Zweck geschaffenen abstrakten Klasse *MoveableAbsolutePanel* erben. Um die Fenster verschiebbar zu gestalten, erbt die MoveableAbsolutePanel Klasse vom AbsolutePanel, sodass diese unter Angabe von absoluten Koordinaten innerhalb des Browserfensters verschoben werden kann. Das Grundgerüst besteht aus einer Titelleiste, die eine per Konstruktor definierte Zeichenkette enthält. Rechtsbündig in der Titelleiste erscheint ein Schließen-Button, welcher den registrierten Handler über den Vorgang benachrichtigt. Das jeweils aktive Fenster wird farblich von den inaktiven hervorgehoben. Unterhalb der Titelleiste befindet sich der *Contentbereich*, welcher von erbenden Klassen befüllt wird. Per Drag and Drop der Titelleiste kann dieses Fenster verschoben werden. Außerdem ist es standardmäßig erlaubt das Fenster in seiner Größe zu verändern, wodurch sich der Cursor in der rechten unteren Ecke des Fensters verändert, um die Skalierbarkeit zu visualisieren. Alle Positionswechsel und Größenveränderungen bedürfen zunächst der Prüfung durch den WindowSpaceDistributor, welcher

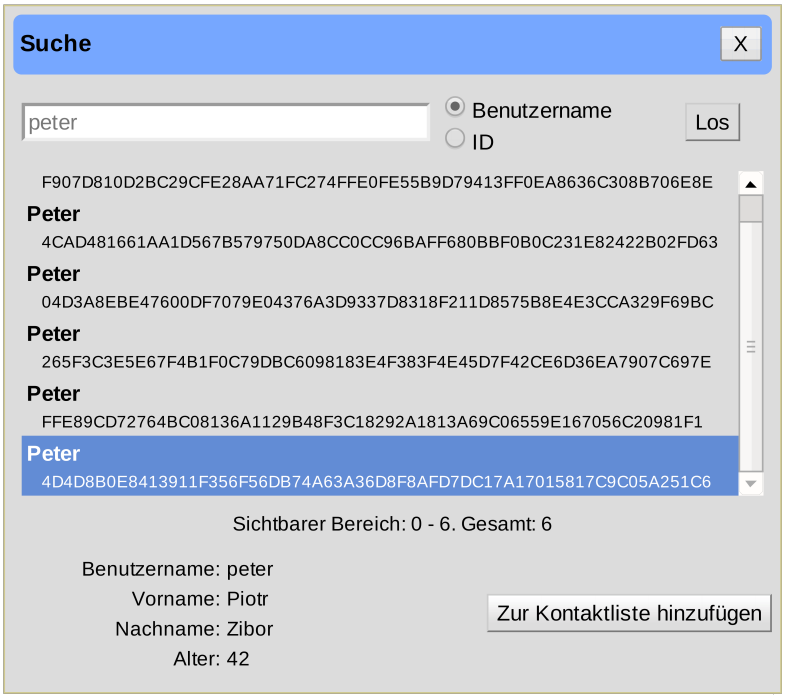


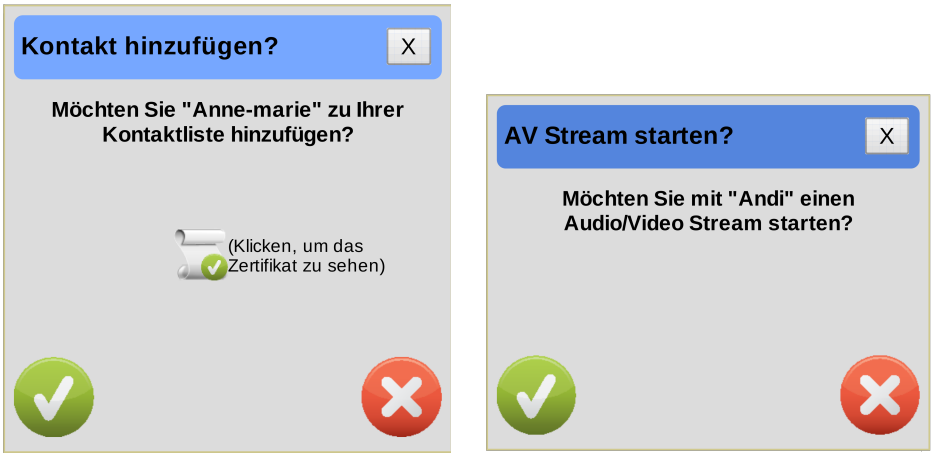
Abbildung 6.6: Suchfenster.

in Abschnitt 6.1.3 weitere Informationen bereithält.

Die von MoveableAbsolutePanel erbende Klassen sind die folgenden.

Abbildung 6.3 trägt die Kontaktliste. Der Contentbereich des Fensters enthält ein *DataGrid*, was einer klassischen Tabelle gleicht. Diese Tabelle zeigt die folgenden Eigenschaften eines jeden Kontakts. Zunächst einmal wird der Benutzername angezeigt. Durch ein Symbol (grüner Haken oder gelbes Fragezeichen) wird die Freundschaft zu diesem Benutzer angezeigt. Lehnt das Gegenüber die Freundschaft ab, so verschwindet der Eintrag aus der Kontaktliste. Weiter zeigt ein Symbol den Zertifikatstatus an. Per Klick auf das Symbol wird dem Benutzer das Zertifikat angezeigt.

Ein Beispielzertifikat zeigt Abbildung 6.4. Das Zertifikat zeigt alle Eigenschaften eines Certificate-Objekts. Die nächste Spalte der Tabelle enthält einen Zeitstempel, wann von diesem Benutzer das letzte Mal ein Lebenszeichen in Form einer Nachricht erhalten wurde. Schließlich zeigt die letzte Spalte den Online Status an. Die Ampelfarben zeigen



(a) Anfrage: Kontakt hinzufügen. (b) Anfrage: AV-Start.

Abbildung 6.7: Anfrage Fenster.

einen online bzw. offline sowie einen undefinierten Status (gelb) an. Der undefinierte Onlinestatus wird beim Laden der Kontaktliste verwendet, da noch keine Aussage über den Onlinestatus getroffen werden konnte.

Per Doppelklick auf einen Kontakt der Kontaktliste öffnet sich die ChatView, visualisiert durch Abbildung 6.5. Der Contentbereich enthält einen HeaderPanel, dessen Footer eine Textbox mit dem Hinweis eine Nachricht einzugeben enthält. Weiter enthält der Footer einen Button, um dem Kontakt per Handler eine AV-Stream-Anfrage zu senden. Der Hauptteil des HeaderPanels enthält eine CellList, welche bei eingehenden oder erfolgreich versendeten Nachrichten automatisch nach unten scrollt. Die Besonderheit hierbei: durch die Verwendung eines *Pagers* werden Nachrichten dynamisch nachgeladen, wenn die CellList bzw. der Chatverlauf nach oben gescrollt wird, sodass ältere Beiträge sichtbar werden. Erhält dieses Fenster den Fokus, so wird dieser auf die Textbox gelegt, um eine Chatnachricht zu verfassen. Nach der Eingabe der Nachricht kann die Enter Taste verwendet werden, um diese dem registrierten Handler zu übergeben.

Das Suchfenster, visualisiert durch Abbildung 6.6, bietet dem Benutzer die Möglichkeit nach neuen Kontakten zu suchen. Ähnlich zur ChatView enthält der Contentbereich auch hier ein HeaderPanel. Der Header des Panels enthält eine Suchmaske: Suchfeld, Radiobutton zur Wahl ob nach Benutzernamen oder öffentlichen Schlüssel gesucht werden

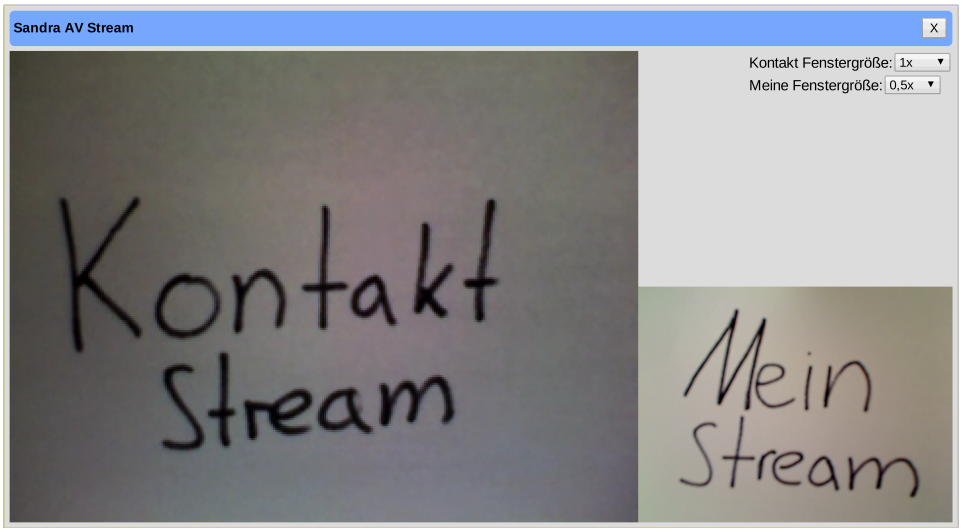


Abbildung 6.8: AV-Stream Fenster.

soll und schließlich ein Startbutton, um dem registrierten Handler den Start der Suche zu signalisieren. Der Hauptteil bzw. die Mitte des HeaderPanels zeigt die Suchtreffer in einer CellList an. Diese Liste lädt Suchtreffer dynamisch nach, sobald nach unten gescrollt wird. Ein Label informiert hierbei über die Anzahl der sichtbaren Treffer und die Gesamtheit der Treffer. Wird ein Suchtreffer markiert, so erscheinen im Footer Details des Treffers. Dies sind Informationen, die dem Zertifikat des Suchtreffers entnommen werden. Per Buttonklick ist es möglich den markierten Kontakt der Kontaktliste hinzuzufügen, was durch den Handler bewerkstelligt wird.

Eingehende Anfragen werden dem Benutzer über Fenster der Abbildung 6.7 sichtbar gemacht. Möchte ein Kontakt den lokalen Benutzer hinzufügen, so erscheint ein Fenster der Art von Abbildung 6.7(a). Durch einen Klick auf das Zeritifkatsymbol wird das Zertifikat des anfragenden Benutzers gezeigt. Weiter kann ein bekannter Kontakt per ChatView eine AV-Anfrage versenden, welche dem Benutzer durch die Abbildung 6.7(b) visualisiert wird. Für beide Fensterarten gibt es einen Akzeptieren- und Ablehnen-Button, wodurch der Handler die ihm gegebene Information verarbeitet.

Nachdem eine eingehende Anfrage, visualisiert durch Abbildung 6.7(b), positiv beantwortet und der Aufbau geglückt ist, öffnet sich ein AV-Stream Fenster, zu sehen in Abbildung 6.8. Bei der Instanziierung wird die AV-Quelle des Kontakts und die eigene

Abbildung 6.9: Monitoring Fenster.

benötigt, da den per *Video.createIfSupported()* geschaffenen HTML5 Video-Objekten jeweils eine Quelle per *void setSrc(String)* angegeben werden soll. Die Aufgabe des Fensters ist simpel. Die linke Seite wird vom Kontakt Stream gebraucht und auf der rechten Seite befindet sich unten der eigene AV-Stream. Oberhalb des eigenen AV-Streams können per Drop-Down-Menüs die AV-Streams skaliert werden. Gewählte Skalierungen, die durch Platzmangel zu einer Überschreitung des Oberflächenplatzes führen, werden so weit runter skaliert, bis diese passen. Durch die automatische Skalierung des Fensters ist eine manuelle Anpassung nicht nötig und wurde dementsprechend deaktiviert.

Das letzte implementierte Fenster ist das Monitoring Fenster, welches Abbildung 6.9 darbietet. Im Contentbereich kommt ein *TabLayoutPanel* zum Einsatz. Hier gibt es eine Tableiste und hinter jedem Tag kann ein eigenes Panel definiert werden, welches den Inhalt des Tabs trägt. Die Tabs *Lokales Repositorium*, welche die lokalen DHT Einträge anzeigt, und *Referenztabellen*, welche die lokale Routingtabelle enthält, tragen jeweils einen Aktualisierungsbutton und eine Textarea, wo der aktualisierte Inhalt angezeigt wird. Der Monitoring Tab enthält zwei Drop-Down-Menüs und einen Aktualisierungsbutton, um die jeweiligen ausgewählten Informationen in der darunterliegenden Textarea anzuzeigen.

Der Aufbau der Benutzeroberfläche der GUI-Implementierung ist hiermit abgeschlossen. Ein weiterer wichtiger Bestandteil der GUI-Implementierung ist der WindowSpaceDistributor, welcher nachfolgend vorgestellt wird.

6.1.3 Platzverwalter

Der Platzverwalter, der sogenannte *WindowSpaceDistributor* beschäftigt sich mit den folgenden Fragen:

- Wenn der ChatViewController ein neues Fenster öffnen möchte, wo soll dieses positioniert werden?
- Sollen Überlappungen von neu positionierten Fenstern stattfinden?
- Was geschieht, wenn das Browserfenster verkleinert bzw. vergrößert wird?
- Dürfen alle Fenster an beliebige Positionen verschoben werden oder sollen Grenzen eingeführt werden?

Die in der Motivation enthaltenen Fragen werden nun in Ziele für die Implementierung des WindowSpaceDistributors gefasst. Zunächst soll eine Positionsfindung implementiert werden, welche einen Platz im Browserfenster findet, sodass keine Schnitte mit anderen Fenstern auftreten. Falls eine schnittfreie Positionierung nicht möglich ist, so soll eine Backup Funktionalität, beispielsweise durch Verkleinerung des zu positionierenden Fensters, angeboten werden. Entstehende Scrollbalken des Browserfensters durch Verschiebung von Fenstern sollen durch Eingrenzung des Browserfensters unterbunden werden.

Die Implementierung WindowSpaceDistributorImpl erfüllt diese Ziele und wird nachstehend dargestellt.

Per Konstruktor werden der Implementierung zunächst die initial verfügbare Größe des Browserfensters, die Höhe des Headers und eine *NoSuitablePlacePolicy* übergeben. Da der Header auch nicht von übrigen Fenstern geschnitten werden sollte, ist dies notwendig.

Die NoSuitablePlacePolicy ist das eingesetzte Backupverfahren, welches verwendet werden soll, wenn keine schnittfreie Position gefunden wurde. Diese besteht aus einem Enum und enthält zur Zeit zwei Methoden: *i*) *none* mit einer Fehlerrückgabe und *ii*) *halving_size* mit der Halbierung der benötigten Größe des zu platzierenden Objekts. Fortan wird der verfügbare Platz mithilfe einer zweidimensionalen Liste mit Integer Einträgen nachvollzogen. Die Werte der Liste sind Gewichte des an dem Pixel befindlichen Fenstern. Je höher das Gewicht, desto wichtiger die darunter liegenden Fenster. Aus diesem Grund werden Fenster bei der Instanziierung in zwei Kategorien eingeordnet: wichtige und weniger-wichtige Fenster. Diese Angabe hat bei der späteren Positionsfindung einen Einfluss.

Weiter benachrichtigen alle MoveableAbsolutePanel Fenster die WindowSpaceDistributorImpl über eine Neupositionierung oder eine Anpassung der Größe. Mithilfe der Liste, die den verfügbaren Platz hält, wird die Auskunft gegeben, ob diese Aktion legal ist und somit nicht über den Rand des Browserfensters hinaus läuft oder den Header schneidet.

Zuletzt folgt eine Beschreibung des verwendeten greedy Algorithmus zur Findung einer Position für ein neu öffnendes Fenster.

Ein grundlegend einfacher greedy Algorithmus wird zunächst für die Findung eines Fensterursprungs mit den Koordinaten x_{origin}/y_{origin} anhand der gegebenen Höhe h und Breite b ermittelt, welcher keine Fenster mit dem Gewicht g überlappt. Der Methodenkopf lautet *Rectangle findFreeSpace(Size, int)*. Hierfür wird die zweidimensionale Liste zeilenweise in hinreichend kleinen Schritten (z.B. 10) in x Richtung gelaufen, bis ein Wert eines Punktes x / y gefunden wird, der g unterschreitet, oder das Ende, d.h. x-b und y-h erreicht ist. Wurde ein potentieller Fensterursprung gefunden, so erfolgt nun die Prüfung, ob für die benötigte Größe des Fensters genügend Platz vorhanden ist. Zu diesem Zweck werden Checkpoints in x und y Richtung gesetzt, welche sequenziell geprüft werden. Diese Checkpoints sind der Anfang, das Ende und dazwischen vielfache der minimalen Fenstergröße. Somit wird sichergestellt, dass sich in dem Bereich kein Fenster mit einem Gewicht $\geq g$ befindet. Wurde nach Durchlauf der Checkpoints keine gewichtete Überschneidung aufgedeckt, so wurde ein Fensterursprung x_{origin}/y_{origin} gefunden und wird dem Aufrufer zurückgegeben. Wurde die komplette Fläche ohne den Fund eines

Fensterursprungs durchlaufen, so wird dem Aufrufer null zurückgegeben.

Ein weiterer grundlegender Algorithmus wird für die Suche eines zufälligen Platzes innerhalb des Browserfensters verwendet. Der Methodenkopf lautet *Rectangle findRandomSpace(Size)*. Hier werden zunächst alle möglichen Werte für den Fensterursprung anhand der Größe des Browserfensters und der benötigten Höhe und Breite des Fensters ermittelt und einem Pool hinzugefügt. Enthält der Pool gültige Werte, so wird ein zufälliger Ursprung gezogen, ansonsten wird null zurückgegeben.

Um einen passenden Platz für Fenster mit gegebener Größe $size_{origin}$ zu finden, werden die obigen Algorithmen folgendermaßen verwendet.

1. Versuche unbenutzten Platz zu finden. Rufe findFreeSpace(size, 0)

2. Falls kein Platz gefunden wurde (Rückgabe null), starte einen erneuten Versuch und suche dabei auch Überschneidungen mit unwichtigen Fenstern. Rufe findFreeSpace(size, Weights.UNIMPORTANT).

3. Falls kein Platz gefunden wurde, d.h. Rückgabe null ist, greift die verwendete NoSuitablePlacePolicy:

   - *none*: Gebe null zurück, was darauf hinweist, dass kein geeigneter Platz für das Fenster gefunden wurde.

   - *halving_size*: Halbiere die Fenstergröße *size*, indem die Höhe und Breite halbiert wird. Sei $size = \frac{size}{2}$

     – Fall: $size > minSize$. Springe zu Schritt 1 zurück und verwende die halbierte Fenstergröße.

     – Fall: $size < minSize$. Suche einen zufälligen Platz für das Fenster. Starte mit dem Aufruf findRandomSpace($size_{origin}$). Schlägt dieser fehl, so halbiere die Größe rekursiv, bis ein Platz gefunden wurde oder die minimale Größe unterschritten ist. Bei einer Unterschreitung wird null zurückgegeben, was darauf hinweist, dass kein geeigneter Platz gefunden wurde.

Der oben beschriebene Algorithmus sucht somit stufenweise nach einem geeignetem Platz für eine gegebene Größe eines Fensters. So ist die erste Stufe einen Platz zu finden, der mit keinem Fenster Überlappungen enthält. Stufe zwei sucht nach einem Platz, wo weniger-wichtige Fenster überlappt werden. Stufe drei ist von der verwendeten NoSuitablePlacePolicy abhängig und unter *halving_size* werden Halbierungen der Fenstergröße unternommen und die Stufen eins und zwei erneut durchlaufen. Ist eine Halbierung aufgrund der Beschränkung einer minimalen Fenstergröße nicht durchführbar, so startet Stufe vier. Die vierte und letzte Stufe prüft rekursiv unter Halbierung der Größe, startend mit der benötigten Originalgröße, ob diese unabhängig von Gewichtungen im Browserfenster einen zufälligen Platz findet.

Da der ChatViewController lediglich die Kontaktliste als wichtiges Fenster deklariert, endet der Algorithmus oft bei Stufe eins oder zwei. Falls der Benutzer seine Kontaktliste sehr groß skaliert, was ihm freisteht, kommt Stufe drei zur Geltung. Die letzte Stufe wird nur erreicht, wenn der Benutzer die Kontaktliste über nahezu das komplette Browserfenster skaliert oder das Browserfenster selbst sehr klein skaliert.

Somit ist das Kapitel rund um die GUI-Implementierung, welche ViewController (Abschnitt 6.1.1) zur Verwaltung ihrer Views (Abschnitt 6.1.2) und einen Platzverwalter (Abschnitt 6.1.3) zur Platzierung und Platzierungsüberwachung verwendet, abgeschlossen. Das nachfolgende Kapitel behandelt das Konfigurationsmodul.

6.2 Modul: Konfiguration

Ziel des Konfigurationmoduls ist es den übrigen sechs Modulen eine Sammelstelle verschiedener Einstellungen anzubieten. Ein wichtiger Aspekt sei hier, dass die Sammelstelle lediglich Informationen raus gibt aber die Änderung dieser nicht unterstützt. Eine weitere Anforderung betrifft die Anpassung verschiedener Konfiguration und deren Auswirkung auf die Knoten des P2P-Overlays. Es soll möglich sein verschiedene Konfigurationsmerkmale dynamisch zur Laufzeit zu verändern, sodass neu eintretende Knoten diese verwenden. Da sich die Konfigurationsanpassung bereits aktiver Knoten im Chord-

Ring als schwierig erweist, sollen der Konfiguration der Knoten nach dem Start bzw. Laden der HTML Seite keinerlei Änderungen widerfahren.

Im Rahmen dieser Arbeit ist ein Konfigurationsmodul entstanden, welches den Anforderungen genügt. Nachstehend folgt die Realisierung des Moduls mit anschließender Betrachtung der Konfigurationsmerkmale.

Realisierung

Die Implementierung des Konfigurationmoduls besteht aus drei Teilen.

Zunächst sind für den Server und alle Clients bzw. Peers des P2P-Overlays die Klasse *ConfigAttributes* zugänglich. Diese enthält alle Attribute bzw. Konfigurationsmerkmale für dieses Modul.

Im zweiten Teil implementiert der Server einen Service, um Clients ein aktuelles ConfigAttributes-Objekt zur Verfügung zu stellen. Ein *ConfigService* Interface mit der Methode *public ConfigAttributes getConfig();*, welches das GWT-Interface RemoteService erweitert, wird von dem Server in der Klasse *ConfigServiceImpl* implementiert. Dies ist das Pattern des GWT-RPC. Der Server hält nun in seinem lokalen Dateisystem die Konfigurationsdatei, welche eine Java-Properties-Datei ist, und kann die enthaltenen Konfigurationsmerkmale anpassen. Kommt ein RPC call bei dem Server an, so wird aus der lokal gesicherten Konfigurationsdatei ein ConfigAttributes-Objekt erstellt und an den Aufrufer zurückgegeben. Ist keine Konfigurationsdatei vorhanden, so wird eine mithilfe von Standardwerten erstellt.

Der dritte Teil bezieht sich auf den Clients Teil der Realisierung. Dieser kann mithilfe der *Config* Klasse auf den eben genannten Service des Servers zugreifen. Die Config Klasse bietet die Klassenmethode *public static void createConfig(final ConfigRetrieval);* zur Erstellung einer Konfiguration an. Die Klassenmethode holt hier das benötigte ConfigAttributes-Objekt vom Server und erstellt ein Config-Objekt, welches das ConfigAttributes-Objekt kapselt. Das erstellte Config-Objekt wird über das ConfigRetrieval Interface an den Aufrufer zurückgegeben. Um dem Client nun den Zugriff auf die Konfigu-

rationsmerkmale zu gewähren, enthält die Config Klasse diverse *delegate* get-Methoden der Klasse ConfigAttributes. Außerdem wird dem Client die Instanziierung eines Config-Attributes-Objekts verwehrt, da eine *RuntimeException* geworfen wird, falls dieser dies versucht.

Der Arbeitsablauf der beschriebenen Teile ist wie folgt. Möchte der Client die Konfiguration abrufen, so erfolgt zunächst der Aufruf an die Klassenmethode createConfig. Nachdem der Server den RPC verarbeitet hat, erhält der Client per Callback ein Config-Attributes-Objekt, anhand dessen ein Config-Objekt erstellt wird. Konfigurationsmerkmale können nun anhand von get-Methoden des Config-Objekts geholt werden. Ändert der Server nun die Konfigurationsdatei, so werden fortan RPC's die Änderung beinhalten und Clients erhalten die aktuelle Konfiguration. Um nun durch eine Anpassung der Konfigurationsdatei die Kompatibilität der Clients mit einer älteren Version und der Neuen zu sichern, sind einige Konfigurationsmerkmale *nicht* durch die Konfigurationsdatei veränderbar. Zu diesen gehören beispielsweise Merkmale der Kommunikationsschicht, wie die verwendete elliptische Kurve oder verwendete Hashfunktion des Chordmoduls. Sollen diese geändert werden, so ist eine Anpassung im Quellcode und somit eine erneute Kompilierung des Projekts und der damit verbundene Neustart des Webservers notwendig.

Die eben beschriebene Realisierung erfüllt die Zielsetzung einer teilweise dynamischen Konfigurationsammelstelle. Im nächsten Abschnitt erfolgt ein Einblick in vorhandene feste und dynamische Konfigurationsmerkmale.

Konfigurationsmerkmale

Dieser Abschnitt beinhaltet eine vollständige Aufzählung der festen und Übersicht der dynamischen Konfigurationsmerkmale. Eine vollständige Aufzählung der dynamischen Konfigurationsmerkmale hält der Anhang B bereit.

Wie im Abschnitt 6.2 erläutert gibt es feste und dynamische Konfigurationsmerkmale des Konfigurationmoduls. Zu den festen Merkmalen zählen all die, welche durch Änderung eine Inkompatibilität von Clients bzw. Peers unterschiedlicher Versionen ausmacht.

So ist die Angabe des PeerJS-Host/Port fest, da die Peers sonst untereinander nicht mehr vermittelt werden können. Weiter sind alle Merkmale der Sicherheitsschicht des Chordmoduls fest. Zu diesen gehören die verwendete Kurve, Hashfunktion, Parameter für die Schlüsselgenerierung per PBKDF2 (Iterationen und Schlüssellänge) und das signierungs paranoia Level, welche die Sicherheit der Signatur beeinflusst. Zuletzt sind alle Webserver Konfigurationen fest. Diese enthalten *i*) die maximale Anzahl an Knoten ID's im Knoten Pool für den öffentlichen Login *ii*) die Anzahl an ID's, die bei einer Loginanfrage an den Knoten versendet werden *iii*) die maximale Dateigröße von log-, monitoring- und Knoten Pool Dateien und *iv*) den Namen der Property Datei, die eben alle dynamischen Konfigurationsmerkmale innehält. Die Beschränkung bezüglich der Dateigröße zeigt die Maximalgröße einer Datei, wonach diese abgespalten und umbenannt wird, um eine Stückelung von beispielsweise Logs zu erhalten.

Einen Auszug der Konfigurationsmerkmale, die dynamisch zur Laufzeit geändert werden können, bietet folgende Liste.

- (De-)Aktivierung des Loggers und die Logger Klasse. Siehe Abschnitt 6.3
- Log Level, d.h. ab welchem Log Level sollen Lognachrichten vom Logger erfasst werden.
- Startzeit und Periode von Aufrechterhaltungsjobs des Chordmoduls[1]
- Zertifikatssuche: Anzahl initialer retrieve Sendungen und maximale Anzahl an unbeantworteten retrieve Sendungen
- Standardgröße aller verwendeter Fenster im GUI-Modul.
- Repräsentierung der ID's. Binär, Dezimal oder Hexadezimal[1]
- ... eine vollständige Aufzählung beinhaltet Anhang B

[1] Aus OpenChord Konfiguration entnommen[KL07].

6.3 Modul: Logging

Das Loggingmodul hat das Ziel den übrigen Modulen einen Logging Dienst zu bieten. Hierbei sind Logger zu implementieren, die abhängig von der Konfiguration die Ausgabe der Logs entweder auf die lokale Javascript Konsole oder zum Server umlenken, wo diese in einer Datei im lokalen Dateisystem gesichert werden.

Realisierung

Zur Umsetzung des Ziels verwenden wir eine modifizierte Version des OpenChord Loggers, da sich dieser sehr gut erweitern lässt, um es auf der Chatplattform nutzbar zu machen. Der OpenChord Logger besteht aus einer abstrakten Klasse *Logger* und beinhaltet eine Klassenmethode *Logger getLogger(String name)* mit dessen Hilfe eine Logger Instanz der in der Konfiguration gespeicherten Loggerklasse per Reflection API gebaut wird. Jede Logger-Implementierung muss nun Methoden für die Ausgabe eines jeden *LogLevels* erstellen. Die folgenden LogLevels stehen zur Verfügung: debug, info, warn, error, fatal.

Im Rahmen dieser Masterarbeit erfolgten folgende Schritte zur Realisierung des Loggingmoduls.

Zum einen war eine Abänderung der getLogger Klassenmethode notwendig, da das GWT-Framework die Java Reflection API nicht emuliert. So ist der dynamische Konstruktoraufruf nun in statische Abfragen umgewandelt, die abfragen, welche Logger-Klasse in der Konfiguration gesichert ist. Bei einem Treffer wird dessen Konstruktor gerufen. Der einzige Nachteil der statischen Abfragen ist der, dass die Implementierung eines neuen Loggers auch eine weitere statische Abfrage nach sich ziehen muss.

Weiter ist die Entfernung aller Implementierungen der abstrakten Logger-Klasse nötig, da diese nicht benötigt bzw. inkompatibel zum GWT-Framework sind.

Die *LogRecord*-Klasse verspricht eine einheitliche Generation von Logeinträgen. Diese Klasse implementiert die *isSerializable* Schnittstelle des GWT-Frameworks, um Daten

an den Server zu versenden bzw. zu (de-)serialisieren. Die Reihenfolge der Log-Elemente und eine Beispielausgabe sieht wie folgt aus:

```
#Timestamp;Loglevel;Username;Loggername;Msg
09.06.2014|20:17:36,648|UTC+2;DEBUG;Andi;GUIImpl;Debugging Code.
09.06.2014|20:17:37,123|UTC+2;INFO;Georg;Application;Some event occured.
09.06.2014|20:17:38,456|UTC+2;WARN;Jan;NodeImpl;Inserting replicas failed! Reason:[...]
09.06.2014|20:17:39.789|UTC+2;ERROR;Jana;OperationPool;Null pointer exception at [...]
09.06.2014|20:17:40,012|UTC+2;FATAL;Herbert;ChordImpl;Join failed! Reason:[...]
```

Die erste Logger-Implementierung, der *ConsoleLogger*, erstellt anhand der vorliegenden Informationen ein LogRecord-Objekt. Um die Ausgabe des Objekts auf die Javascript Konsole umzuleiten, wird die GWT Browser Konsole Klasse *console* und dessen Methode *log(Object)* verwendet.

Die zweite Logger-Implementierung, der *ServerLogger*, erstellt analog zum Console-Logger zunächst ein LogRecord-Objekt. Ein weiterer RPC-Service, der *ServerLogService*, ist notwendig, da diese Objekte an den Server versenden werden sollen. Serverseitig erfolgt die Implementierung des Service durch die Implementierung der ServerLogService-Schnittstelle mitsamt der *void logToServer(LogRecord)* Methode. Der Server erhält das vom Client versendete LogRecord-Objekt und speichert diesen Eintrag in eine lokale Logdatei. Hat die Logdatei die konfigurierbare maximale Größe erreicht, so erfolgt eine Umbenennung der Datei nach dem Muster *log<vierstellige laufende Nummerierung>*.

Mit der Modifizierung der OpenChord Klassenmethode getLogger, Entfernung der Logger-Implementierung, Schaffung der Klasse LogRecord und zuletzt Implementierung zweier Logger ist das Ziel, definiert in Abschnitt 6.3, erreicht.

6.4 Modul: Monitoring

Für die Auswertung der anfallenden Statistiken, die durch den OperationPool, das in Abschnitt 4.2.5 detailliert wird, ist das Monitoringmodul zuständig. Zu den Aufgaben des Moduls gehören die Beantwortung von Statistikanfragen der GUI und die Ausführung eines in der Konfiguration, das in Abschnitt 6.2 spezifiziert ist, definierten Monitoringjobs. Ist ein Monitoringjob definiert, so erfolgt ein periodisches Abrufen der Ope-

rationPool Daten und strukturierter Versand an einen *Collector*, welcher in der Lage ist die gesammelten Daten auszuwerten. Eine Beschreibung enthält Abschnitt 6.4.1.

Zunächst zu den Statistikanfragen der GUI. Das GUI-Modul sendet per Schnittstelle Anfragen an das Monitoringmodul. Die Antworten auf die Anfragen entsprechen sinngemäß den gebündelten Informationen aus Abschnitt 4.2.5. Hinzu kommt eine Monitoring Abfrage bezüglich eines OperationType und MessageType Tupels. Um bei dieser umfangreichen Abfrage die Abarbeitung der Browser-Event-Queue nicht übermäßig zu verzögern bzw. einzufrieren, erfolgt die Antwort asynchron. Wie bereits in Kapitel 3 erläutert, bedient sich das Monitoringmodul direkt vom OperationPool des Chordmoduls. Sobald die gebündelten Informationen vorliegen, leitet das Monitoringmodul diese an die GUI weiter.

Die Implementierung des Monitoringjobs ist im nachstehenden Abschnitt geschildert.

6.4.1 Monitoringjob

Ist in der Konfiguration der MonitoringMode an bzw. *true*, so startet der Monitoringjob bei der Instanziierung des Monitoringmoduls. Verallgemeinert ist dies eine konfigurierbare periodische Erhebung von Monitoringdaten, welche anschließenden an den Collector versendet werden. Hierzu hält die Konfiguration neben dem MonitoringMode zwei weitere Attribute für das Modul bereit. Die Dauer des Jobs, d.h. wie lange sollen periodisch Daten gesammelt und an den Collector versendet werden und die Periode, d.h. wie oft sollen Datenerhebungen stattfinden. Ist die Dauer des Jobs auf 0 gesetzt, so läuft der Job solange, wie man eingeloggt ist. Die anfallenden Daten bei dem Collector sind für die Auswertung 7 interessant, da er die Auswertungsdaten speichert.

Im Folgenden wird erläutert, was bei einem Aufruf des Jobs passiert. Zunächst sind genau zwei Schritt pro Aufruf nötig. Die Einsammlung der Daten und das Verschicken an den Collector.

Zunächst zur Einsammlung der Daten. Hierzu erfolgen Aufrufe an die OperationPool Instanz. Die Ergebnisse werden in insgesamt drei Tabellen gespeichert: eine für Mes-

sageType, eine für OperationType und eine für eine Kombination aus beiden. Pro Aufruf werden maximal 39 Einträge in den drei Tabellen gesichert: Genau 13 in die MessageType Tabelle (eine pro MessageType). Genau eine pro OperationType, d.h. 11, in die OperationType Tabelle. Außerdem legale Kombinationen aus OperationType und MessageType. Ein Beispiel wäre: für Operation Insert werden maximal zwei Nachrichtentypen versendet. Diese wären findSuccessor und insertEntry. Aktuell sind dies etwa 15, was in der Auswertung 7 erörtert wird. Sind alle Einträge gesammelt, so erfolgt der Versand an den Collector. Ist der Empfang der Daten bestätigt worden, so werden die lokalen Tabellen gelöscht, da diese nicht mehr benötigt werden. Außerdem erfolgt eine Prüfung, ob die Dauer des Jobs abgelaufen ist. Ist dies der Fall, so wird dies dem Collector mitgeteilt.

Im Rahmen dieser Arbeit ist eine Collectorimplementierung als Serverdienst via RPC entstanden. Somit erhält der Server die versendeten Daten seiner Clients. Um die Dateneingänge seiner Clients einem anonymen Knoten zuzuordnen, schicken diese eine zufällig generierte aber über den Zeitraum des Jobs feste ID der Länge 20 mit jedem Monitoringschritt mit. Der Server legt bei dessen Initialisierung im Root-Servletkontext einen Monitoring-Ordner an, worin sich ein weiterer finished Ordner befindet. Weiter werden drei Mappings initialisiert, die die geschickten ID's auf Tabellen Daten abbildet. Ein Tabellen Datum enthält die Tabelle selbst sowie eine Dateireferenz, welche die Tabelle im Monitoring-Ordner sichert. Weiter werden pro Client drei Dateien erstellt, welche beim Empfang von Daten der zugehörigen ID aktualisiert wird. Signalisiert ein Client den Abschluss des Monitorings, so werden seine Daten geschlossen, d.h. Datenstrukturen aufgeräumt und dessen Dateien im Dateisystem in den Ordner finished verschoben. Die Auswertung derer kann dann vollzogen werden.

Schlussendlich hat das Monitoringmodul ähnlich wie andere Module Zeit, um auf einen Auslogwunsch zu reagieren. In diesem Kontext erfolgt die Beendung des Monitoringjobs, falls dieser zuvor gestartet wurde. Hierdurch erfolgt ein letzter Datenaustausch mit dem Collector, um diesem die Beendigung zu signalisieren. Ist dieser Vorgang erfolgreich verlaufen, so wird dem Waiter die Bereitschaft gemeldet.

Das Entladen des Moduls führt zu einer Beendigung des OperationPools, sodass diese das Einsammeln von Operationsdaten unterbricht.

Abschließend sei angemerkt, dass dieses Modul vorrangig für die Auswertung der Chatplattform entworfen und implementiert wurde. Diese ist Thema des nächsten Kapitels 7.

6.5 Beiträge des Autors

Die vier vorangegangenen Module sind fast ausschließlich Beiträge des Autors. Ausnahmen werden nachfolgend benannt.

Der im GUI-Modul des Abschnitts 6.1 genannte Internationalisierungsprozess ist ein Werkzeug des GWT-Frameworks. Der Autor hat lediglich gewünschte Sprachdateien anfertigt und diese werden mithilfe des Werkzeugs im Quellcode verfügbar gemacht.

Im Konfigurationsmodul in Abschnitt 6.2 sind einige Konfigurationsmerkmale von OpenChord übernommen worden. Diese sind im Anhang B markiert.

Weiter baut das Loggermodul, beschrieben in Abschnitt 6.3, auf den Logger von OpenChord auf. Hierzu waren Modifizierungen notwendig, um es im GWT-Framework lauffähig zu machen. Jegliche Logger-Implementierungen und die Log-Einträge sind jedoch im Rahmen dieser Arbeit entstanden.

Das Monitoringmodul ist ausnahmslos der Beitrag des Autors.

Kapitel 7

Auswertung

Dieses Kapitel enthält die Auswertung dieser Arbeit. Zunächst werden in Abschnitt 7.1 die Methodik der Auswertung sowie die Erwartungen an die Ergebnisse geschildert. Der anschließende Abschnitt 7.2 enthält die Analyse der gesammelten Daten des zuvor beschriebenen Testlaufs.

7.1 Methodik und Erwartungen

Das Ziel dieser Auswertung ist einen Überblick der Performance der Chatplattform zu schaffen. Hierfür sind eine Reihe von Fragen offen, die es durch die Auswertung zu beantworten gilt.

Zum einen ist das Leistungsverhalten des P2P-Overlay zu bewerten. Wie hoch ist die durchschnittliche Hop-Anzahl von gestarteten Operationen bzw. Nachrichtentypen? Wie verhalten sich die Antwortzeiten von Operationen bzw. Nachrichtentypen? Außerdem ist die Relation von fehlerhaften zu erfolgreichen Anfragen interessant. Weiter sind die Instandhaltungskosten des P2P-Overlays mit den Kosten der darauf laufenden Anwendung zu vergleichen. Die Kosten entsprechen hier der verwendeten Bandbreite. Zuletzt ist die aufgewandte Zeit der kryptografischen Verfahren in der Kommunikationsschicht des P2P-Overlays interessant.

Um die Antworten auf diese Fragen liefern zu können, erfolgte ein Testlauf mit eingeschaltetem Monitoringjob (siehe Abschnitt 6.4.1). Die lokal generierten Datenbestände werden hierbei an den Webserver übermittelt, welcher die Aufbereitung der Daten durchführt. Zu den gesammelten Daten zählen die Anzahl, die Rate und der Bandbreitenaufwand aller versendeter und empfangener Nachrichten. Weiter werden die Hop-Anzahl und Antwortzeiten lokal initiierter Nachrichten nachvollzogen. Hierzu gehört sowohl die gesamte, als auch die durchschnittliche pro Nachricht aufgewandte Zeit kryptografischer Verfahren für das Senden und Empfangen von Nachrichten. Für alle initiierten Operationen werden zudem die Hop-Anzahl und Antwortzeit getrennt gesichert. Außerdem werden die Anzahl initiierter Operationen sowie die absolute Anzahl und prozentual offenen, erfolgreichen und fehlgeschlagenen Operationen übermittelt.

Der Aufbau des Testlaufs mit eingeschaltetem Monitoring sieht wie folgt aus. Zunächst war eine Portierung auf einen lokal laufenden *tomcat7*[1] Webserver nötig. Um den Webserver öffentlich zugänglich zu machen, erfolgte eine Anmeldung der Domain *webp2p.de*. Der Webserver wurde so konfiguriert, dass SSL mit einem Zertifikat ausgestellt von StartSSL [sta13] verwendet wird. Eine PeerJS-Serverinstanz läuft auf der gleichen Maschine wie der Webserver und verwendet ebenfalls das von StartSSL ausgestellte Zertifikat.

Das Testszenario, welches die Chatplattform, erreichbar über *www.webp2p.de* nutzt, sieht wie folgt aus. Insgesamt haben zehn Instanzen während der Laufzeit von 60 Minuten die Chatplattform genutzt. Hierfür erfolgte zunächst eine Anmeldung am öffentlichen Netzwerk mit einer neuen Registrierung. Begonnen wurde mit einem gerade neu gestarteten Webserver, wodurch mit der ersten Registrierung am öffentlichen Netzwerk ein neuer Chord-Ring geschaffen wurde. Nachdem die Benutzer sich eingeloggt haben, wurden jeweils in einem nicht näher spezifizierten Zeitraum fünf der zehn Kontakte hinzugefügt und mit diesen gechattet. Das AV-Streaming wurde für dieses Szenario ausgeschaltet, da das Monitoring den AV-Datenfluss nicht abgreifen kann. Die genaue Konfigurationsdatei, die für dieses Testszenario genutzt wurde, enthält Anhang C.

Die Auswertung und Analyse der generierten Daten enthält der nachfolgende Abschnitt 7.2.

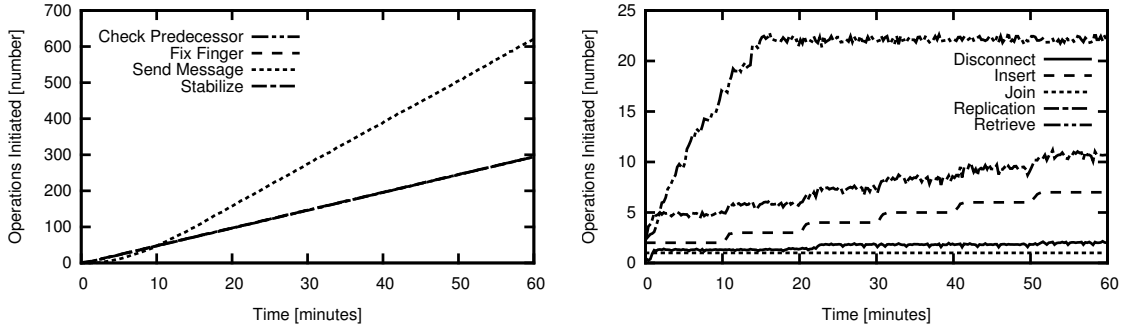
[1] Mit Heartbleed [DBL14] fix.

7.2 Analyse

Die nachfolgenden Abschnitte beantworten die in Abschnitt 7.1 gestellten Fragen mithilfe erstellter Grafiken der Monitoringdaten des Testszenarios. Grafiken, die gezeigt werden, basieren auf kumulativen Datenbeständen und jeder Graph ist das Erzeugnis eines arithmetischen Mittels über die vorliegenden Daten. Die Bündelung aller Graphen in einer Grafik würde diese unübersichtlich machen.

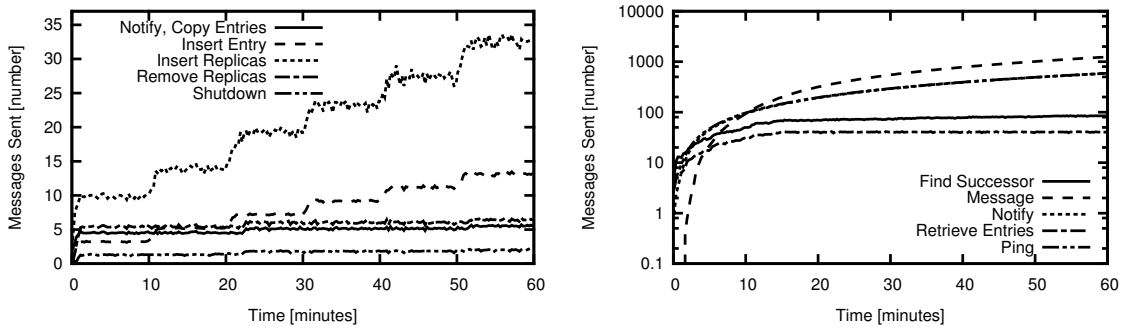
Weiter ist jede Grafik zu Operationen in zwei Abbildungen aufgeteilt: *Operationstypen A* enthält die Instandhaltungsjobs des Chord-Overlays (Fix Finger, Check Predecessor und Stabilize) sowie Send Message. *Operationstypen B* enthält die restlichen Operationen: Disconnect, Insert, Join, Replication und Retrieve. Remove wird nicht verwendet. Die Nachrichtengrafik erfährt eine ähnliche Teilung in zwei Abbildungen: *Nachrichtentypen A* enthält notifyAndCopyEntries, insertEntry, insertReplicas, removeReplicas und shutdown. *Nachrichtentypen B* enthält dementsprechend den Rest: findSuccessor, message, notify, retrieveEntries und ping. Da Operation Remove keine Verwendung findet, wird dementsprechend removeEntry keinem Nachrichtentyp zugeordnet.

Abschnitt 7.2.1 beschreibt zunächst den Verlauf des Szenarios anhand der gestarteten Operationen und deren versendeten Nachrichten. Weiter beschäftigt sich der Abschnitt 7.2.2 mit dem Datenaufkommen während des Testings. Verglichen werden Instandhaltungskosten des P2P-Overlays mit den Kosten der darauf laufenden Anwendung. Anschließend wird in Abschnitt 7.2.3 die aufgewandte Zeit für kryptografische Verfahren der kompletten Laufzeit gegenübergestellt. Im Anschluss beschäftigt sich Abschnitt 7.2.4 mit der Hop-Anzahl von Nachrichtentypen und Operationen. Besonders interessant ist der Hop-Anzahl-Vergleich des Nachrichtentyps findSuccessor mit dem theoretisch ermittelten Wert im Chordpaper [SMK+01]. Zum Schluss befasst sich Abschnitt 7.2.5 mit der Antwortzeit initiierter Operationen und deren Nachrichten unter Berücksichtigung der in den vorherigen beiden Abschnitten ermittelten Ergebnissen.



(a) Operationstypen A (b) Operationstypen B

Abbildung 7.1: Initiierte Operationen.



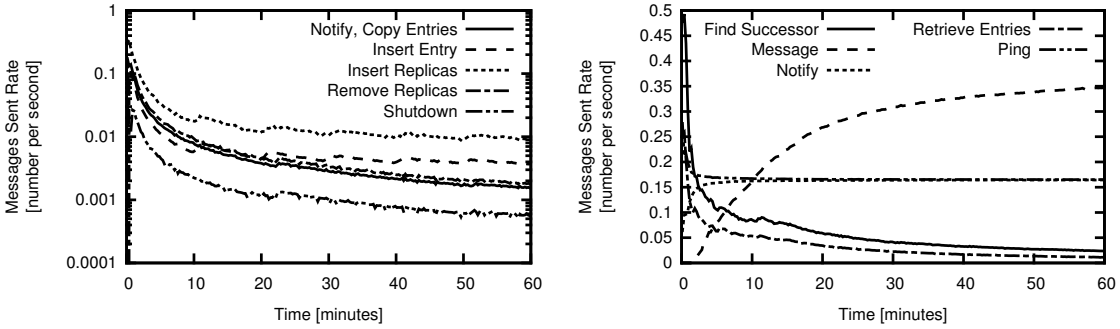
(a) Nachrichtentypen A (b) Nachrichtentypen B

Abbildung 7.2: Versendete Nachrichten.

7.2.1 Verlauf des Szenarios

In diesem Abschnitt erfolgt eine Analyse des Verlaufs. Zunächst sei angemerkt, dass hier nur initiierte Operationen und versendete Nachrichten betrachtet werden. Die Grafiken zu empfangenen Nachrichten sieht der Grafik mit versendeten Nachrichten ähnlich, da relativ wenige Nachrichten verloren gegangen sind. Hierzu mehr am Ende des Abschnitts und im nächsten Abschnitt 7.2.2.

Die Grafiken 7.1, 7.2 und 7.3 beinhalten die Anzahl initiierter Operationen, versendete Nachrichten und die Rate versendeter Nachrichten. Die Rate ist hierbei in Nachrichten



(a) Nachrichtentypen A

(b) Nachrichtentypen B

Abbildung 7.3: Rate versendeter Nachrichten.

pro Sekunde angegeben. Auf der x-Achse befindet sich überall die verstrichene Zeit in Minuten.

Join-Vorgang und Stabilisierung

Da der Monitoringjob erst nach dem Join startet, ist der Knoten ab Minute null im Chord-Netzwerk angemeldet. Die Operations Grafik 7.1(b) zeigt hierbei, dass die durchschnittliche Anzahl an Join Operationen aller Knoten bei eins liegt. Die verwendeten Nachrichten für die Join Operation sind der ping, findSuccessor und notifyAndCopyEntries. Grafik 7.3(b) zeigt eine erwartungsgemäß hohe ping Rate, da potentielle Bootstrap Knoten zu Beginn angepingt werden, um den Join Vorgang zu starten. Den hohen Peak der Versandrate von findSuccessor ist im Rahmen der anschließenden Platzsuche im Chord-Ring der Grafik 7.3(b) zu entnehmen. Nachdem der richtige Platz gefunden wurde, erfolgt der Versand von notifyAndCopyEntries Nachrichten, wodurch Einträge kopiert werden, für die der lokale Knoten zuständig ist. Dieser Peak liegt bei einer Rate von 0,17, d.h. etwa alle fünf Sekunden eine Nachricht. Im Zuge der neuen Einträge fallen Replikationsoperationen an, welche in Grafik 7.1(b) und dem Versand von insertReplica in Grafik 7.1(a) zu verzeichnen sind.

Anschließend folgt das Einfügen von Zertifikaten an zwei Stellen durch das Zertifikatmanagement, beschrieben in Abschnitt 4.2.3. Dies ist anhand der Grafik 7.1(b) ersichtlich,

wo durchschnittlich genau zwei insert Operationen gestartet werden. Der Anstieg der durchschnittlich initiierten Retrieve Operationen ist mit dem Eintragen des Zertifikats in die Benutzernamenliste zu erklären, wo zunächst ein freier Platz gesucht wird. Da die Insert sowie Retrieve Operation findSuccessor Nachrichten versenden, ist der hohe Peak der findSuccessor Senderate mit durchschnittlich 0,492 Nachrichten pro Sekunde zu erklären. Nachdem das Zertifikat eingefügt wurde, ist eine Replikation dessen notwendig und wird von den zuständigen Knoten durch den Versand von insertReplica Nachrichten initiiert.

Es folgt nun ab Minute eins eine verstärkte Stabilisierungsphase, wo das Starten der Disconnect und Replication Operationen vermehrt eintritt. Dies ist den Grafiken 7.1 zu entnehmen. Ausgelöst werden diese durch den Stabilisierungsjob.

Im Zuge einer empfangenen notify Nachricht merkt ein Knoten, dass dieser seinen Vorgänger prüfen sollte. So wird durch den Versand einer potentiellen shutdown Nachricht an den alten Vorgänger die Verbindung zu diesem getrennt und der neue Vorgänger als dieser anerkannt. Außerdem muss eine removeReplica Nachricht an den letzten Nachfolger der Nachfolgerliste gesendet werden, dass Replikate mit einer ID zwischen dem alten und neuen Vorgänger gelöscht werden müssen.

Erhält der lokale Knoten eine Antwort auf eine zuvor versendete notify Nachricht, so wird die erhaltene Referenzliste geprüft. Ist der lokale Knoten nicht als Vorgänger verzeichnet, so werden mit dem Erhalt der Antwort auf eine verschickte notifyAndCopyEntries Nachricht neue Einträge, für die der lokale Knoten nun zuständig ist, registriert. Außerdem werden Replikate mithilfe der Replication Operation eingefügt. Es folgt der potentielle Versand einer removeReplica Nachricht an einen nicht mehr benötigten Nachfolger der Nachfolgerliste. In Grafik 7.1(b) ist die Initiierung des oben genannten Prozesses durch den Anstieg von Disconnect Operationen festzumachen.

Wichtig anzumerken ist hier, dass die Stabilisierung eines Chord-Rings ein kontinuierlicher Prozess und kein Abschnitt ist. In diesem Szenario ist es lediglich ein Abschnitt, da kein Churn besteht und der Chord-Ring durch die Erreichung der Stabilität auch stabil bleibt. Im normalen Gebrauch ist mit hohem Churn zu rechnen, sodass die Stabilisierung (-sphase) weiter fortbesteht.

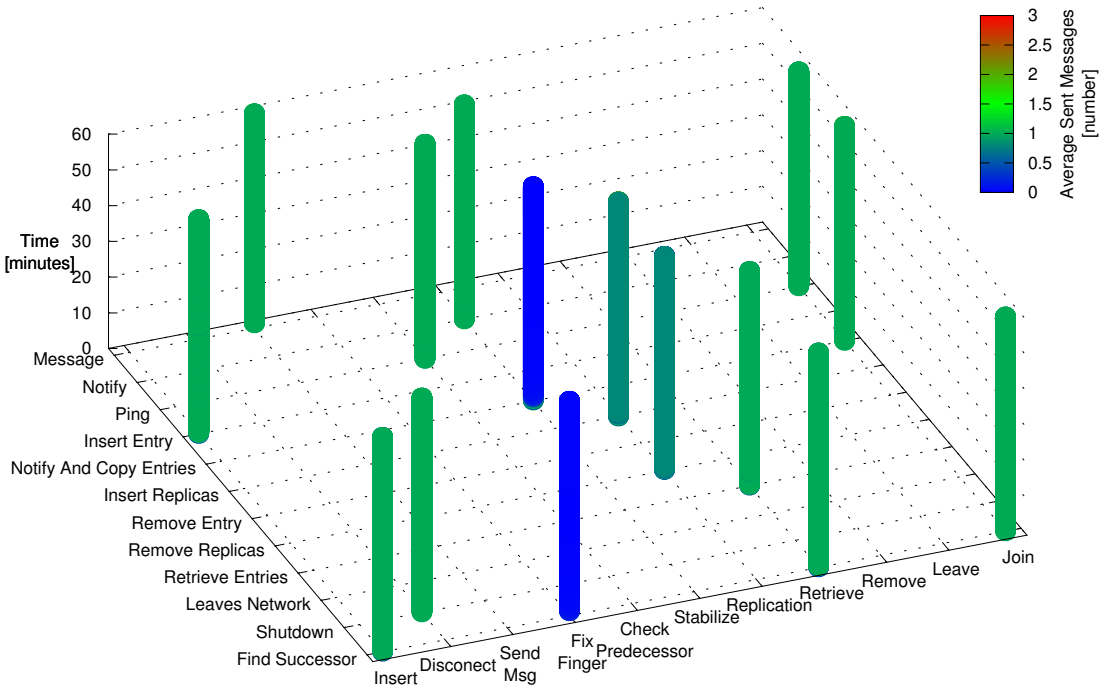


Abbildung 7.4: Operationen / Nachrichten Überblick: durchschnittlich versendete Nachrichten.

Nach Stabilisierung

Nach der Stabilisierung ist ein Abfall der shutdown, notifyAndCopyEntries und removeReplica Senderaten zu verzeichnen. Der Grafik 7.2(a) ist zu entnehmen, dass durchschnittlich zwei shutdown Nachrichten über die gesamte Dauer versendet wurden. RemoveReplica Nachrichten kommen auf durchschnittlich 6,4 und notifyAndCopyEntries auf 5,5 Sendungen.

Weiter zeigt Grafik 7.3(b) die Zunahme der notify Senderate und der Abfall der ping Senderate, da diese im Join Prozess verwendet wurde. Beide Senderaten nähern sich dem erwarteten Wert von 0.167, was einem Nachrichtenversand alle sechs Sekunden entspricht, welche zuvor in der Konfigurationsdatei, siehe Anhang C, festgehalten wurde. Die durchschnittliche Senderaten am Ende des Szenarios betragen 0.164 für notify und 0.165 für ping.

```
Node: 73E5FD766F026579816A39266134640A5B2E9E5ED62D0E52F4821EF87D6B8486
Finger table:
  767744E0FC64882C7C62C1CA411E9DFCC3EC977999A4FA94F088088197F7EFA2 (0-249)
  8634C3CF7C6D8AA1729498A5A238BF729251645BF9E24718985717FC8F5B8181 (250-252)
  AAB693EA7E22F164B817EFED8BBC8E7438A14049E77B87589D78CE05132F3929 (253)
  C1F77CE2273907C9E3B06478A5E5AFD0B2DF70648109593139CE61B99EAB7931 (254)
  671171FB59B08BEFE08BD6227FFDDCE0B170E3155ADCDAD004C016F499DDFA21 (255)
Successor List:
  767744E0FC64882C7C62C1CA411E9DFCC3EC977999A4FA94F088088197F7EFA2
  8634C3CF7C6D8AA1729498A5A238BF729251645BF9E24718985717FC8F5B8181
Predecessor: 671171FB59B08BEFE08BD6227FFDDCE0B170E3155ADCDAD004C016F499DDFA21
```

Abbildung 7.5: Routingtabelle: gespeicherte Referenzen.

Kontaktsuche, Hinzufügen von Kontakten und Chatting

Die Kontaktsuche ist durch den Anstieg der durchschnittlichen Anzahl initiierter Retrieve Operationen in Grafik 7.1(b) festzumachen. Da die Retrieve Operation findSuccessor und retrieveEntries Nachrichten versendet, zeichnet sich ein erhöhter Versand von diesen in Grafik 7.2(b) aus. Die Kontaktsuche endet etwa bei Minute 14, denn es erfolgen danach keine weiteren Initiierungen der Retrieve Operation.

Der danach anhaltende Versand von findSuccessor Nachrichten in Grafik 7.2(b) ist mit dem FixFingerJob zu verbinden. Es werden zwar die periodischen Jobs Stabilize, Check-Predecessor und Fix Finger in etwa gleich oft initiiert (siehe Grafik 7.1(a)), jedoch ist die durchschnittliche Senderate, erfasst durch Abbildung 7.2(b) von findSuccessor gegenüber ping bzw. notify Nachrichten geringer. Die Antwort zeigt Abbildung 7.4. Diese enthält zunächst die Information welche Nachrichten durch eine Operation versendet werden. Die z-Achse enthält die Zeit und die Werte der Metrik werden anhand einer Farbskala abgelesen. Diese Art von Abbildung enthält keine Durchschnittsbildung, d.h. alle vorhandenen Werte werden hier eingespeist und abgebildet. Hier lässt sich feststellen, dass der FixFingerJob durchschnittlich nahezu keine Nachricht versendet. Fix Finger würfelt wohl oft Finger, für die der direkte Nachbar zuständig ist. Die Abbildung 7.5 zeigt die Fingertabelle eines angemeldeten Knotens. So zeigen die ersten 250 Finger auf den direkten Nachfolger. Dies ist richtig, da sich die ID's des lokalen Knotens und des Nachfolgers schon im sechst höchstem Stellenwert unterscheiden. Dies bedeutet wiederum, dass die ersten 256-6 = 250 (0-249) Finger auf den Nachfolger zeigen. Dies ist in einem Chord-Ring mit wenigen Knoten üblich. Die durchschnittliche Senderate der findSuccessor Nachrichten liegt am Ende des Testlaufs bei 0,024, was eine versendete

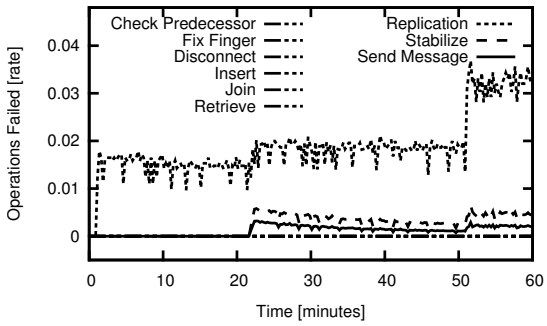


Abbildung 7.6: Rate fehlgeschlagener Operationen.

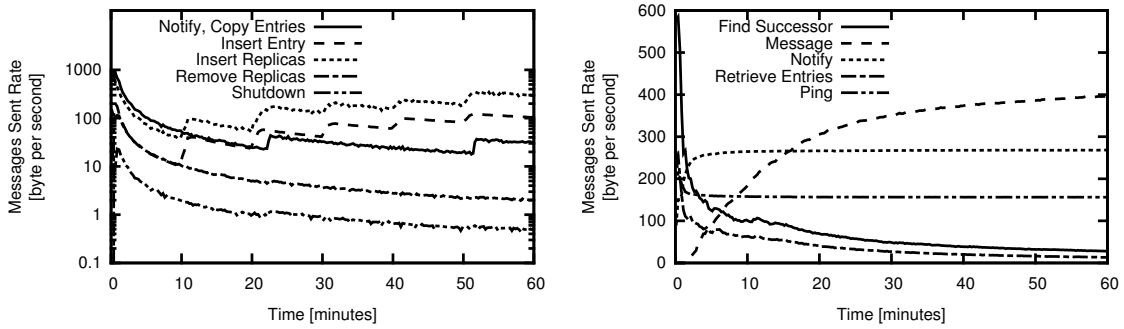
Nachricht alle 42 Sekunden bedeutet.

Der Nachrichtenversand des im Rahmen dieser Arbeit entstandenen Nachrichtentyps message mit Operationstyp Send Message startet ab etwa 1,5 Minuten. Dieser wird für verschiedene Anwendungsschichtprozesse genutzt. Zum einen für diverse Anfragen wie beispielsweise die Kontakthinzufügung aber auch für den periodischen Kontaktlisten-ping. Weiter sind message Nachrichten Träger von Chat Nachrichten, weshalb Abbildung 7.3(b) einen raschen Anstieg der Senderate verzeichnet.

Chatting

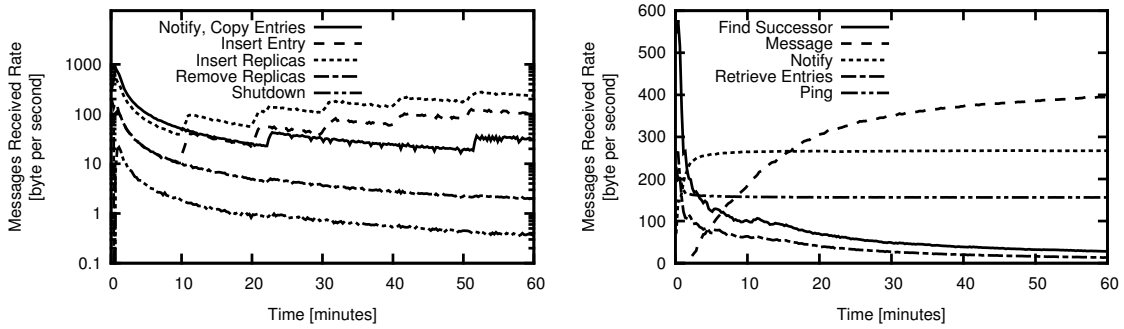
Ab Minute 14 erfolgt nur noch das Chatting. Zuvor ist in Abbildung 7.1(b) bei Minute zehn ein Anstieg um durchschnittlich eine Insert Operation zu verzeichnen. Dies ist der Kontaktlistensicherungsjob, welcher laut Konfigurationsdatei alle zehn Minuten die Kontaktliste und Chatlogs im Chord-Ring sichert. Abbildung 7.2(a) macht den Kontaktlistensicherungsjob sehr gut kenntlich. Alle zehn Minuten geht die Anzahl durchschnittlich versendeter Nachrichten für insertEntries und insertReplicas hoch. Der Nachrichtentyp insertReplicas erfährt eine doppelt so hohe Steigung wie insertEntries, da laut Konfigurationsdatei auf zwei Nachfolgerknoten repliziert werden soll.

Der rasante Anstieg der message Senderate flacht ab etwa Minute 30 ab. Am Ende ist eine durchschnittliche Senderate von 0.347 Nachrichten pro Sekunde bzw. eine Nachricht



(a) Nachrichtentypen A (b) Nachrichtentypen B

Abbildung 7.7: Datenaufkommen versendeter Nachrichten.



(a) Nachrichtentypen A (b) Nachrichtentypen B

Abbildung 7.8: Datenaufkommen empfangener Nachrichten.

alle 2,9 Sekunden zu verzeichnen. Diese umfasst den Kontaktlistenping, falls kein Chat-Nachrichtenverkehr mit diesem herrscht, an durchschnittlich die Hälfte der Online-Kontakte des Knotens. Durchschnittlich ist anzunehmen, dass der lokale Knoten von jedem zweiten Kontakt angepingt wird und die andere Hälfte selbst anpingen muss. So erhält man eine untere Grenze, wenn nämlich kein Chatverkehr herrscht. Diese liegt bei 0,25 Nachrichten pro Sekunde bzw. eine Nachricht alle vier Sekunden bei fünf Online-Kontakten. Hier sei bereits angemerkt, dass dies nicht skaliert und bei größeren Kontaktlisten unter Umständen viele Kontaktpings anfallen. So ist beispielsweise eine dynamische Anpassung der Kontaktping-Periode wünschenswert, da sonst das Nachrichtenaufkommen linear mit der Anzahl an Online-Kontakten in der Kontaktliste steigt.

Tabelle 7.1: Datenaufkommen pro Kategorie.

Kategorie	*Nachrichtentyp*	*Versendet [B/s]*	*Empfangen [B/s]*
Stabilisierungsjobs	notify	268,2	267,2
	ping	156,3	156,3
	findSuccessor	28,0	28,0
Sicherungsjobs	insertEntry	105,4	105,4
	insertReplica	297,5	235,1
Nachrichten	message	396,1	395,2
Rest	notifyCopy	30,9	30,8
	retrieveEntries	13,4	13,4
	removeReplica	2,0	2,0
	shutdown	0,5	0,4
Insgesamt		1298,3	1233,8

Auf das Datenaufkommen während der gesamten Testdauer wird der nächsten Abschnitt tiefer eingehen.

7.2.2 Datenaufkommen

Das Datenaufkommen durch die verschiedenen Operationen ist Thema dieses Abschnitts. Zunächst sei angemerkt, dass das durchschnittliche Datenaufkommen durch das Senden von Daten gleich dem durchschnittlichen Datenaufkommen durch den Empfangen dieser Daten bei keinem Paketverlust entsprechen sollte. Um zunächst einen Überblick über gescheiterte Operationen zu erhaltet, enthält Abbildung 7.6 die Veranschaulichung. Die einzigen Operationstypen, die gescheitert sind, sind die Replication, Stabilize und Send Message Operationen. So erreichen diese zum Schluss eine Fehlerrate von 3,2 %, 0,4 % bzw. 0,2 %. Die gescheiterten Operationen lassen sich sehr wahrscheinlich auf Verbindungsabbrüche eines Teilnehmers zurückführen. Dies ist aufgefallen, da das Streaming über Skype jeweils ebenso unterbrochen wurde. Die Auswirkung der gescheiterten Replikationen wird in diesem Abschnitt deutlich.

Abbildungen 7.7 und 7.8 zeigen die durchschnittlich gesendeten bzw. empfangenen Bytes

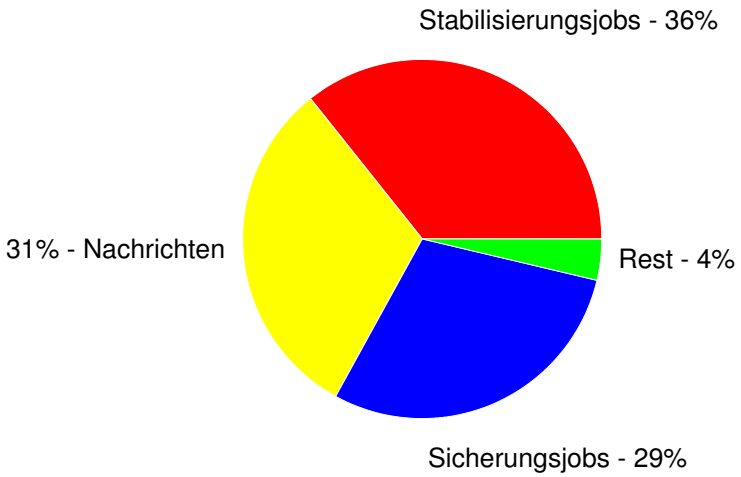


Abbildung 7.9: Prozentuale Aufteilung Datenaufkommen.

pro Sekunde. Zu Beginn sind die versendeten und empfangenen Byte pro Sekunde sehr hoch, da beim Join einige Operationen aufeinander folgen. Dies relativiert sich nach dem Join. Der Abbildung 7.7(a) ist außerdem zu entnehmen, dass insertReplicas für das meiste Datenaufkommen verantwortlich ist. Dies ist klar, da durchschnittlich alle zehn Minuten zwei insertReplica Nachrichten an beide Nachfolger versendet werden. Mit stetig wachsenden Chatlogs, werden auch die insertReplica Nachrichten größer. Dies stellt ein Problem dar. Inkrementelle Sicherungen der Chatlogs wären für nachfolgende Arbeiten denkbar, da dadurch keine *full updates* sondern lediglich *delta updates* geschrieben werden müssen. Außerdem würden geschickte delta updates einer fairen Datenaufteilung im Chord-Ring zugute kommen. Mehr dazu in Kapitel 8.

Tabelle 7.1 enthält das durchschnittliche Datenaufkommen über die gesamte Dauer, sortiert nach Kategorien. Auffällig ist, dass mehr Daten durch die Nachrichtentypen notify, insertReplica und message versendet als empfangen wurden. Vor allem weist insertReplica eine Differenz von 62,4 Byte pro Sekunde auf, sodass etwa 21 % der versendeten Daten nicht beim Empfänger ankommen. Vergleicht man die gesendeten Bytes für den insertReplica Nachrichtentyp bei Minute 50 in Abbildung 7.7(a) mit den empfangenen Bytes für den gleichen Nachrichtentyp der Abbildung 7.8(a), so fällt auf, dass sehr viel mehr Daten gesendet als empfangen wurden. Man beachte hierbei, dass die Abbildungen

Tabelle 7.2: Durchschnittlicher Zeitaufwand von Kryptoverfahren der Kommunikationsschicht von Chord.

Nachrichtentyp	*Zeit beim Senden [ms]*	*Zeit beim Empfangen [ms]*
notifyAndCopyEntries	168	157
insertReplica	113	157
insertEntry	107	139
removeReplica	107	125
retrieveEntries	100	124
notify	98	123
ping	97	123
message	96	125
findSuccessor	95	124
shutdown	94	116

eine logarithmische Skalierung der y-Achse aufweisen. Dies ist genau der Punkt, wo die Replication Operation fehlschlägt und die Daten erneut versendet werden müssen.

Ein Benutzer erzeugt in diesem Szenario ein durchschnittliches Datenaufkommen von 2532,1 Byte pro Sekunde für eingehende und ausgehende Daten. Zum Vergleich: nach 6,9 Minuten ist ein Megabyte Traffic geflossen. Die prozentuale Aufteilung der Datenaufkommen enthält das Kreisdiagramm 7.9. So machen die Stabilisierungsjobs, Sicherungsjobs und der Nachrichtenversand jeweils etwa ein Drittel des Gesamtdatenaufkommens aus. Die Reduzierung des Datenaufkommens durch den Nachrichtenversand und des Sicherungsjobs ist mit den oben genannten Modifikationen sicherlich möglich. Das Datenaufkommen von Stabilisierungsjobs würde sich hingegen in einer Umgebung mit mehr Churn sicherlich erhöhen, da bei Zuständigkeitswechsel Repositorien versendet werden müssen und für deren Replikation gesorgt werden muss. Ist der Chord-Ring hingegen hinreichend groß, so relativiert sich dies im Durchschnitt wieder.

Nach der Analyse des Datenaufkommens enthält der nachfolgende Abschnitt 7.2.3 die Analyse des Zeitaufwands verursacht durch kryptografische Verfahren des Chordmoduls.

Tabelle 7.3: Absoluter Zeitaufwand von Kryptoverfahren der Kommunikationsschicht von Chord.

Nachrichtentyp	*Zeit beim Senden [s]*	*Zeit beim Empfangen [s]*
message	119,6	155,7
notify	57,2	71,8
ping	57,0	71,9
findSuccessor	7,6	9,9
retrieveEntries	3,8	4,8
insertReplica	3,4	4,3
insertEntry	1,4	2,0
notifyAndCopyEntries	0,6	0,8
removeReplica	0,6	0,7
shutdown	0,2	0,2

7.2.3 Zeitaufwand: Krypto

Dieser Abschnitt analysiert den Zeitaufwand, der durch kryptografische Verfahren der Kommunikationsschicht des Chordmoduls entsteht. Die Tabellen 7.2 und 7.3 fassen die Abbildungen aus Anhang D zusammen. Die Werte aus den Tabellen entsprechen jeweils dem Enddatum der jeweiligen Grafik.

Tabelle 7.2 zeigt die durchschnittliche Zeit, die ein Knoten benötigt, um eine gegebene Nachricht zum Senden bzw. Empfangen zu verschlüsseln und signieren bzw. zu entschlüsseln und verifizieren. Da eine asymmetrische Verschlüsselung verwendet wird, ist zu erwarten, dass das Empfangen länger dauert als das Senden. Hier fällt bereits auf, dass größere Nachrichten auch länger Zeit in Anspruch nehmen. So enthält beispielsweise eine notifyAndCopyEntries Nachricht mehr Inhalt als eine ping Nachricht. Über alle Nachrichten gemittelt benötigt ein Knoten rund 107 ms für die Sendungsaufbereitung und 131 ms für die Empfangsaufbereitung. Wichtig ist natürlich auch wie oft ein Nachrichtentyp versendet wird. So sind die Nachrichtentypen message, ping und notify die am öftesten verschickten Nachrichten (siehe Abbildung 7.2(b)). Den größten Einfluss auf den Zeitaufwand hat die CPU des Teilnehmers. So besteht zwischen der schwächsten verwendeten CPU (Intel T2400 CoreDuo, 1. Generation, Januar 2006) und der stärk-

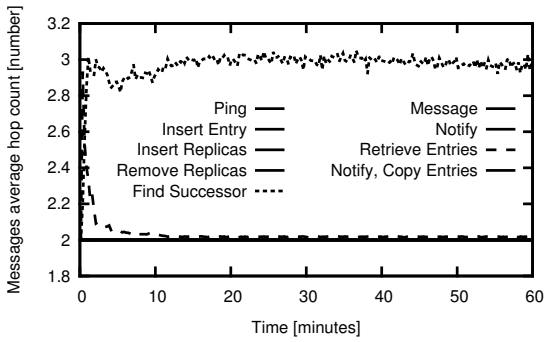


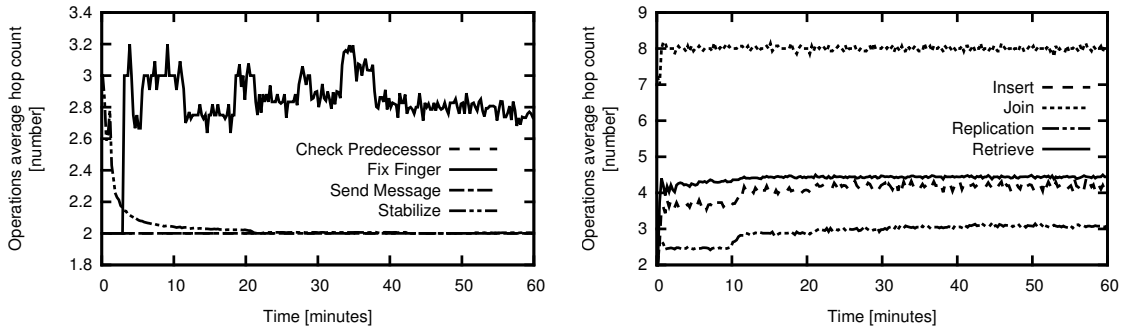
Abbildung 7.10: Durchschnittliche Hop-Anzahl pro Nachrichtentyp.

sten (Intel i5, 3. Generation, Juni 2012) ein Unterschied mit dem Multiplikator sechs. In Zahlen bedeutet dies: die stärkste CPU bereitet eine Nachricht zum Versand in durchschnittlich 41 ms vor. Die schwächste schafft dies erst nach 257 ms.

Tabelle 7.3 zeigt den über alle Teilnehmer absoluten gemittelten Zeitaufwand. So macht die Masse der versendeten message Nachrichten den Großteil der aufgewendeten Zeit aus. In der Summe werden 251,5 Sekunden für das Aufbereiten zum Senden benötigt. Rechnet man dies auf die 60 Minuten Gesamtdauer hoch, so sind dies rund 7 % der Zeit. Analog sind es beim Empfang rund 9 %. Insgesamt werden durchschnittlich 15,9 % der gesamten Zeit für das Aufbereiten verwendet. Dies ist ein relativ hoher Wert, welcher jedoch sehr stark CPU abhängig ist. So verbringt die schwächste CPU 34 % der gesamten Zeit für das Aufbereiten, wohingegen die stärkste lediglich 6,8 % benötigt.

Als Fazit bleibt hier zu erwähnen, dass alte CPUs (zum Zeitpunkt des Tests acht Jahre alt) relativ viel mit dem Aufbereiten durch kryptografischen Verfahren beschäftigt werden. Dies entspricht rund einem Drittel der gesamten Zeit, was sogar zu regelmäßigen Blockierungen der GUI führt. Die stärkste CPU, die zum Zeitpunkt des Tests zwei Jahre alt ist, schafft dies rund fünf mal besser, sodass eine Blockierung der GUI ausbleibt.

Der in diesem Abschnitt analysierte Zeitaufwand spielt auch eine Rolle für die Antwortzeit-Analyse im letzten Abschnitt 7.2.5. Eine weitere wichtige Rolle in der Antwortzeit-Analyse spielt die im nachfolgenden Abschnitt 7.2.4 analysierte Hop-Anzahl.



(a) Operationstypen A (b) Operationstypen B

Abbildung 7.11: Durchschnittliche Hop-Anzahl von Operationen.

7.2.4 Hop-Anzahl

Zunächst ist es wichtig die Hop-Anzahl (engl. *hop count*) zu definieren. Die Hop-Anzahl ist nachfolgend die Anzahl an Nachrichten, die infolge einer Anfrage von Knoten versendet werden. Wie in Abschnitt 4.1 beschrieben, wird eine Abwandlung des rekursiven Routings verwendet. Im Gegensatz zum rekursiven Routing erhält der anfragende Knoten die Antwort nicht direkt vom antwortenden Knoten, sie wird stattdessen entlang des zurückgelegten Pfads an diesen zurück geleitet. Dies hat zur Folge, dass die Pfadlänge einer Anfrage $\frac{Hop-Anzahl}{2}$ beträgt.

Die einzigen Nachrichten, die weitergeleitet bzw. geroutet werden, sind findSuccessor, insertEntry, removeEntry und retrieveEntries. Für das Routing selbst, d.h. die Auffindung des Nachfolgers eine Chord ID, ist findSuccessor zuständig. Die drei weiteren Nachrichtentypen werden aus Performancegründen nicht verworfen, wenn der empfangene Knoten für die enthaltene Chord ID nicht zuständig ist, sondern an den Vorgänger weitergeleitet, da dieser mit großer Wahrscheinlichkeit nun für diese ID zuständig ist. Andernfalls wäre eine erneute findSuccessor Nachricht seitens Initiator notwendig, um den neuen zuständigen Knoten aufzufinden.

Interessant wäre der Vergleich der durchschnittlichen Hop-Anzahl der Routingnachricht findSuccessor gegenüber der im Chordpaper [SMK+01] enthaltenen Formel für die Pfadlänge. Dort heißt es: Sei N die Anzahl an Knoten im Ring, so ist die Pfadlänge $\frac{1}{2}\log_2 N$.

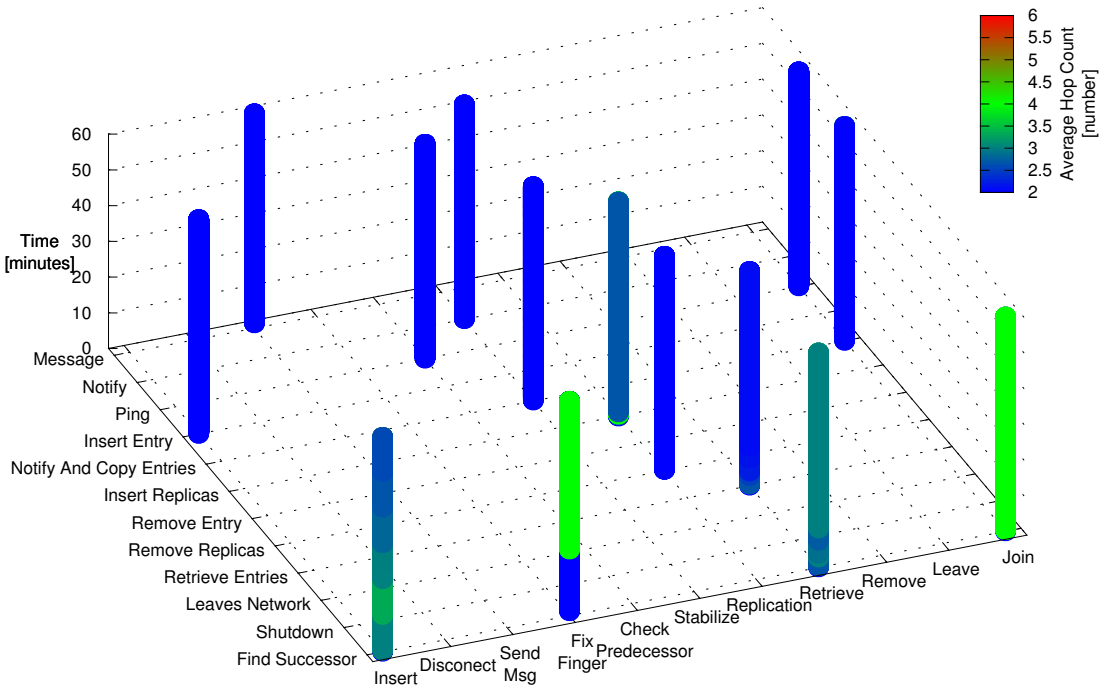


Abbildung 7.12: Operationen / Nachrichten Überblick: durchschnittliche Hop-Anzahl.

Für $N = 10$ entspräche dies einer durchschnittlichen Pfadlänge von 1,66.

Hop-Anzahl von Nachrichtentypen

Zunächst stellt sich die Frage, wie groß die Hop-Anzahl der einzelnen Nachrichtentypen ist. Wie zuvor erwähnt, wird ein Großteil der Nachrichten direkt an den Empfänger versendet, sodass die Hop-Anzahl zwei und die Pfadlänge eins ist, vorausgesetzt der Empfänger wurde zuvor mithilfe der findSuccessor Nachricht gefunden. Abbildung 7.10 enthält, bis auf shutdown und removeEntry, alle Nachrichtentypen und deren durchschnittliche Hop-Anzahl gemittelt über alle Teilnehmer. Der Nachrichtentyp shutdown, dessen Versand durch die Operation Disconnect initiiert wird, ist hier nicht vertreten, da aus Performancegründen nicht auf den Erhalt der ACK-Nachricht gewartet, sondern die Verbindung direkt nach dem Versand abgebaut wird. Der Nachrichtentyp removeEntry hingegen wird von dieser Implementierung nicht benötigt.

Tabelle 7.4: Durchschnittliche Hop-Anzahlen von Operationen.

Operationstyp	*Durchschnittliche Hop-Anzahl*
Join	8
Retrieve	4,4
Insert	4,2
Replication	3
Fix Finger	2,7
Stabilize	≈ 2
Send Message	2
Check Predecessor	2

Abbildung 7.10 zeigt, dass retrieveEntries Nachrichten aufgrund von Zuständigkeitsänderungen zu Beginn öfters weitergeleitet werden. Nachdem das Netz seine Stabilität erreicht hat, fällt der Hopcount auf einen Endwert von 2,09 ab. Bei insertEntry kam dieser Fall nicht vor, sodass dieser durchgängig eine durchschnittliche Hop-Anzahl von zwei hat. FindSuccessor, der wohl interessanteste Nachrichtentyp, erreicht am Ende eine durchschnittliche Hop-Anzahl von 2,97, was einer durchschnittlichen Pfadlänge von 1,49 entspricht. Dementsprechend wurde die theoretische durchschnittliche Pfadlänge von 1,66 unterschritten. Dies liegt wahrscheinlich an dem relativ kleinen Chord-Ring, welcher sich nach kurzer Zeit stabilisiert hat.

Einfluss auf Operationstypen

In diesem Abschnitt bleibt der Einfluss der Hop-Anzahlen von Nachrichtentypen auf die der Operationen zu klären. Tabelle 7.4 liefert die durchschnittlichen Hop-Anzahlen am Ende des Laufs, visualisiert durch Abbildung 7.11.

Operationen, die nur eine Nachricht versenden, sind für eine weitere Analyse uninteressant. Zu diesen zählen Check Predecessor und Send Message, welche der Hop-Anzahl des jeweiligen Nachrichtentyps entsprechen. Für die Analyse der anderen Operationen hilft Abbildung 7.12. Diese enthält Informationen über die durch Operationen versendeten Nachrichten und deren Hop-Anzahlen. Anders als vorherige Grafiken sind die Werte

keine Durchschnitte sondern enthalten alle Hop-Anzahlen der Teilnehmer. Dies bedeutet, dass jeder der dargestellten “Türme” die Informationen von zehn Graphen enthält.

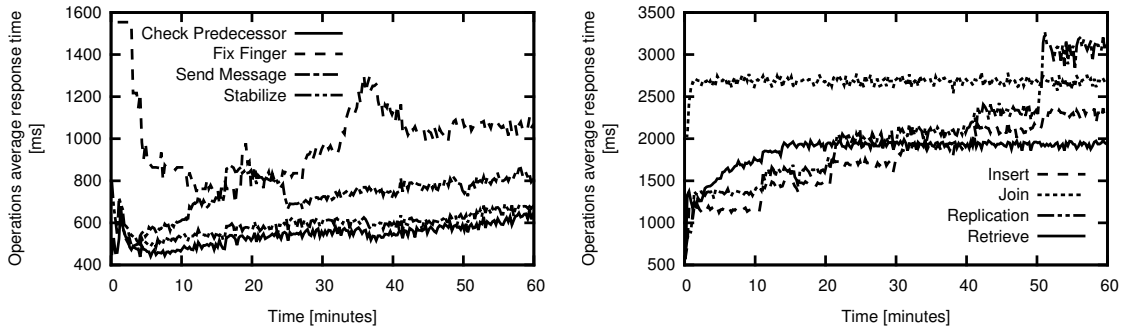
Operation **Join**, welche ping, findSuccessor und notifyAndCopyEntries Nachrichten versendet, hat eine durchschnittlichen Hop-Anzahl von acht (laut Tabelle 7.4) erreicht. Abbildung 7.12 liefert die genaue Aufteilung unter den Nachrichten: ping und notifyAndCopyEntries benötigten zwei Hops und findSuccessor durchschnittlich vier.

Die Operation **Retrieve** verwendet findSuccessor mit Hop-Anzahlen zwischen zwei und drei. RetrieveEntries benötigt zu Beginn etwa drei Hops, was sich nach der Stabilisierung in Richtung zwei bewegt. Insgesamt ergibt diese Operation eine durchschnittlichen Hop-Anzahl von 4,4 und ist über die Dauer hinweg sehr konstant.

Insert setzt sich aus einem findSuccessor, welcher zu Beginn zwei bis 3,5 Hops aufbringt und anschließend in Richtung zwei tendiert und insertEntry, welcher im Durchschnitt zwei Hops benötigt, zusammen. Der Verlauf der Hop-Anzahlen, visualisiert durch Abbildung 7.11(b), zeigt einen Sprung bei Minute zehn und bleibt dann bis zum Ende konstant. Der Sprung lässt sich auf die Ausführung des Sicherungsjobs zurückführen.

Das gleiche Verhalten zeigt auch die **Replication** Operation, welche durchschnittlich zwei Hops für removeReplica und zwischen zwei und vier Hops für insertReplica benötigt. Wie kommt jedoch der Wert aus Tabelle 7.4 zustande? Für die Klärung ist es wichtig in Erfahrung zu bringen, wann die Operation Replication gerufen wird. Dies erfolgt *i*) bei der Verdrängung eines Nachfolgers in der Nachfolgerliste, sodass eine removeReplica Nachricht an diesen geschickt wird (2 Hops) *ii*) bei einem neuen Nachfolger in der Nachfolgerliste, welchem Replikate per insertReplica Nachricht übermittelt werden (2 Hops) und *iii*) bei Eintragssicherung für einen Eintrag, für den der lokale Knoten zuständig ist: sende insertReplica an alle Knoten der Nachfolgerliste (zwei Einträge: 2 x 2 Hops). Die beiden ersten Fälle treten nur zu Beginn auf, der letzte dagegen immer wenn der lokale Knoten einen neuen Eintrag speichern muss. Einige Knoten müssen dies alle zehn Minuten, sodass dies auf drei Hops gemittelt ist.

Operation **Fix Finger** nutzt ausschließlich den Nachrichtentyp findSuccessor. Hier kommen Werte zwischen zwei und sechs vor. Hop-Anzahlen mit zwei bis vier überwiegen jedoch. Somit ergibt dies einen Durchschnitt von 2,7. Fix Finger Aufrufe, die keinen wei-



(a) Operationstypen A (b) Operationstypen B

Abbildung 7.13: Durchschnittliche Antwortzeit von Operationen.

teren Knoten kontaktieren mussten, da der gesuchte Knoten der Nachfolger ist, liefern keine null-Hops für diesen Aufruf. Die initiierte Operation wird zwar gezählt, jedoch nicht als null-Hop gekennzeichnet, sodass diese Aufrufe keinen Einfluss auf die Durchschnittliche Hop-Anzahl haben.

Die letzte verbliebene Operation **Stabilize** nutzt überwiegend den Nachrichtentyp notify, welcher immer zwei Hops benötigt. Zu Beginn jedoch, wenn ein neuer Nachfolger gefunden wurde, so wird neben der notify Nachricht eine notifyAndCopyEntries Nachricht verschickt, sodass diese Stabilisierungsoperation insgesamt vier Hops benötigt. Abbildung 7.11(a) visualisiert dies durch den Peak bei den ersten Minuten. Anschließend erfolgt kein weiterer Versand der notifyAndCopyEntries Nachrichten, sodass die Durchschnittliche Hop-Anzahl auf 2,004 abfällt. Nachdem die Stabilize Operation bei Minute 50,8 bei einem Knoten fehlgeschlagen ist (siehe Abbildung 7.6), wird erneut eine notifyAndCopyEntries Nachricht verschickt, was den minimalen Anstieg erklärt.

Im nächsten Abschnitt 7.2.5 erfolgt die Analyse der Antwortzeiten während des Tests.

7.2.5 Antwortzeiten

Die Antwortzeit ist die Zeit, die verstreicht, bis eine Anfrage eine Antwort erhält. Die Anfrage kann eine einzige Nachricht sein, oder aber eine Operation, welche eine Rei-

Tabelle 7.5: Durchschnittliche Antwortzeit von Operationen.

Operationstyp	*Durchschnittliche Antwortzeit [ms]*
Replication	3105
Join	2687
Insert	2313
Retrieve	1937
Fix Finger	1065
Send Message	805
Stabilize	679
Check Predecessor	643

he von Nachrichten auslösen kann. Die Antwortzeit hängt von der im vorherigen Abschnitt 7.2.4 analysierten Hop-Anzahl, der im Abschnitt 7.2.3 analysierten Kryptozeit und der Internetanbindung des Teilnehmers ab.

Den Verlauf der Antwortzeiten für alle Operationen über den Testzeitraum zeigt Abbildung 7.13. Das Enddatum zu jeder Operation, d.h. den Wert zum Zeitpunkt t = 60 Minuten, enthält Tabelle 7.5. Was fällt nun zum Verlauf in Abbildung 7.13 auf? Die Antwortzeit der Join Operation bleibt über den Verlauf konstant und beträgt Durchschnittlich 2,6 Sekunden, was ein guter Wert in Hinblick auf eine Hop-Anzahl von 8 ist. Das Zittern des Join Graphen kommt durch die Intervall- und Durchschnittsbildung der Werte in diesem Intervall zustande.

Die Retrieve Operation beginnt mit einer Antwortzeit von 500 ms, welche jedoch im Verlauf neu ankommender Knoten auf den Wert 1,9 Sekunden korrigiert wird. Check Predecessor und Stabilize sind bis auf den Beginn relativ Konstant, da nur der Nachfolger bzw. Vorgänger kontaktiert wird. Es ist jedoch bei beiden Operationstypen eine leichte Steigung von etwa 100 ms von Minute fünf bis zum Ende zu verzeichnen. Bei Send Message ist das gleiche Phänomen zu beobachten. Zunächst pendelt sich die Antwortzeit bei etwa Minute 25 ein und steigt dann bis zum Ende um etwa 100 ms an. Woran dies liegt konnte abschließend nicht geklärt werden, da viele Faktoren hierfür eine Rolle spielen könnten. So könnte es sein, dass einzelne Knoten mit den ankommenden Daten überfordert sind

und so eine Warteschlange für Anfragen entsteht, wodurch sich die Antwortzeit verlängert.

Die Operation Fix Finger hat auf den ersten Blick überraschend gute Antwortzeiten, denn kehrt die Fix Finger Operation ohne jeglichen Nachrichtenversand zurück (gesuchter Finger ist Nachfolger), so wird keine null-Antwortzeit hinzugerechnet. Mit einer mittleren Hop-Anzahl von 2,9 Hops, wie im vorherigen Abschnitt 7.2.4 erläutert, sollte die Antwortzeit dementsprechend nur wenig schlechter als die von Check Predecessor und Stabilize Operationen sein. Genau dies zeigt auch Abbildung 7.13(a). Die anfänglich hohe Antwortzeit ist mit der Instabilität des Chord-Rings zu verbinden. Die über die gesamte Dauer gemittelte Antwortzeit liegt bei 1065 ms.

Abbildung 7.13 zeigt weiter, dass die Antwortzeiten der Operationen Insert und Replication über den gesamten Verlauf stetig zunehmen. Zum einen könnte dies an der Kryptozeit liegen, die bei größeren Datenmassen ebenfalls zunimmt. Zum anderen zeigt Abbildung 7.6 bei Minute 20,8 und 50,8 jeweils fehlgeschlagene Replication Operationen an, wodurch eben diese größeren Sprünge zustande kommen können. Der Datenbestand spricht dafür: bei Minute 20,8 gibt es eine maximale Antwortzeit von 4,4 Sekunden für eine Replication Operation. Weiter gibt Minute 50,8 erneut eine maximale Antwortzeit von 10,5 Sekunden für eine Replication Operation an. Ein weiterer Faktor ist die Begrenzung des Upload Kanals eines jeden privaten Internetanschlusses, wodurch sich die Antwortzeit mit wachsenden Nachrichtengrößen verlängert.

Insgesamt wird eine durchschnittliche Wartezeit von 1,65 Sekunden, gemittelt über alle Operationen, vernommen. Viele Operationen laufen jedoch im Hintergrund, welche der Benutzer nicht mitbekommt bzw. realisiert. Antwortzeiten von Operationen, die Benutzer direkt betreffen, da diese eine Antwort einer GUI-Interaktion sind, sind besonders kritisch. So beträgt die durchschnittliche Antwortzeit beispielsweise einer Chatnachricht (via Send Message Operation) 805 ms für die Rückmeldung vom Gegenüber, sodass der Benutzer keine ganze Sekunde auf die Rückmeldung der GUI warten muss.

Die Analyse und somit auch die Auswertung ist nun abgeschlossen. Weitere Abbildungen, die nicht näher beschrieben wurden, enthält Anhang E. Die Erkenntnisse der Analyse werden nun im letzten Kapitel 8 aufgegriffen und diskutiert.

Kapitel 8

Fazit

Das Fazit enthält zunächst in Abschnitt 8.1 eine Zusammenfassung mitsamt Ergebnis dieser Arbeit. Den Abschluss dieser Arbeit bietet Abschnitt 8.2 mit einem Ausblick auf mögliche aufbauende Arbeiten.

8.1 Ergebnis

Das Ziel dieser Arbeit war die Implementierung einer sicheren und browserbasierten Chatplattform mithilfe des WebRTC-Standards. Im Rahmen der Arbeit ist eine modulare P2P-Chatplattform entstanden, die den gestellten Anforderungen gerecht wird und anders als andere P2P-Software keine weitere Installation benötigt. Voraussetzung ist lediglich eine (aktuelle) Version eines Browsers, welche den WebRTC-Standard unterstützt. Aktuell[1] sind dies Google Chrome, Mozilla Firefox und Opera.

Dies wurde durch die Nutzung eines P2P-Overlays realisiert, welches eine Modifikation der Chord-Implementierung OpenChord ist. Durch die Modifikation sind Funktionalitäten entstanden, wodurch dem Chord-Ring unter Eingabe eines Benutzernamens und Passworts beigetreten und mithilfe automatisch erstellter Zertifikate andere Nutzer im Chord-Ring gesucht werden können. Neben der kompletten Verschlüsselung der Kom-

[1] Stand: 27.06.2014

munikation zwischen Browsern seitens WebRTC ist eine weitere Sicherheitsschicht im Chordmodul entstanden, welche die verschlüsselte Sicherung von Anwendungsschichtdaten im Chord-Ring ermöglicht. So werden zum Beispiel Kontaktlisten gesichert, um diese bei einem erneuten Beitritt laden zu können. Die Chatanwendung, welche das P2P-Overlay nutzt, bietet eine Kontaktlistenverwaltung, Kontaktsuche, Single-Chat und Audio / Video-Streaming an.

Durch die Implementierung einer Monitoringkomponente ist es möglich Informationen zum Zustand des Netzwerks zu sammeln. So erfolgte ein Testlauf mit zehn Instanzen, die eine Datenbasis für die Evaluation generierten. Hierfür sieht das Szenario einen Join, die Kontaktsuche und das anschließende Chatten vor. Aufbauend auf der erzeugten Datenbasis erfolgte eine Analyse, die mithilfe verschiedener Metriken die Performance der Chatplattform prüft.

Die Ergebnisse der Analyse sind die folgenden. Das aggregierte Datenaufkommen für den Up- und Downstream betrug pro Knoten im Durchschnitt 2532,1 $\frac{Byte}{s}$. Dieses Datenaufkommen ist für aktuelle Verhältnisse nicht hoch. Der Flaschenhals bei privaten Internetanschlüssen liegt beim Upload, verwendet für den Nachrichtenversand, welcher bei etwa 1 bis 5 $\frac{Megabit}{s}$ und somit 125 bis 625 $\frac{Kilobit}{s}$ liegt. Im Szenario betrug der Datenversand durchschnittlich etwa $1,3\frac{Kilobyte}{s} = 10,4\frac{Kilobit}{s}$ und entspricht somit einer etwa acht prozentigen Auslastung bei einem 1 $\frac{Megabit}{s}$ Upload. Bei steigender Knotenzahl würde diese Implementierung jedoch nicht skalieren, da *i*) periodische Vollsicherungen der Chatlogs durchgeführt werden und *ii*) Kontaktlistenpings linear zur Anzahl an Online-Kontakten wachsen. Dies gilt es in nachfolgenden Arbeiten zu lösen. Der nächste Abschnitt 8.2 beschäftigt sich nochmals hiermit.

Bevor Nachrichten versendet bzw. empfangene Daten weitergeleitet werden, werden diese durch die Sicherheitsschicht des Chordmoduls aufbereitet, was Zeit kostet. Die beiden einzigen Faktoren, die die verwendete Zeit beeinflussen, sind die Leistung der verwendeten CPU und die Nachrichtengröße. Die Analyse liefert einen durchschnittlichen Aufwand von 15,9 % der gesamten Zeit, sodass die CPU rund $\frac{1}{6}$ der Gesamtdauer für die Nachrichtenaufbereitung aufwendet. Etwa acht Jahre alte CPUs wenden rund 34 % der Gesamtdauer dieser Aufgabe zu, sodass GUI-Blockierungen im Millisekunden- bis Sekunden-Bereich auftreten können. Rund zwei Jahre alte CPUs hingegen benötigen nur

6,8 % der Gesamtdauer für die Aufbereitung, wo GUI-Blockierungen ausbleiben.

Die Analyse der Hop-Anzahl liefert eine Unterschreitung der theoretischen Pfadlänge für die Routingnachricht findSuccessor. Die theoretische Pfadlänge $\frac{1}{2}\log_2 N$ bringt bei dem Szenario einen Wert von 1,66: erreicht wurden durchschnittlich 1,49.

Abschließend liefert die Analyse der durchschnittlichen Wartezeit einen gemittelten Wert von 1,65 Sekunden pro Operation. Kritische Operationen wie das Senden einer Chatnachricht, die vom Benutzer selbst initiiert werden, sind jedoch durchschnittlich schneller. So gelingt der Versand einer Anwendungsschichtnachricht und dessen Bestätigungsempfang in etwa 805 ms.

Die Weiterentwicklung der entstandenen Software ist durch den modularen Aufbau in diverse Richtungen möglich. Einige der Möglichkeiten benennt und erläutert der anschließende Ausblick.

8.2 Ausblick

Einige Möglichkeiten die im Rahmen dieser Arbeit entstandenen Software zu erweitern wurden bereits genannt und würden eine Verbesserung der Performance bringen.

So ist eine dynamische Anpassung der Kontaktlistenping-Periode denkbar, da momentan Kontaktpings linear zur Anzahl an Online-Kontakten in der Kontaktliste versendet werden müssen. Dynamisch bedeutet in diesem Zusammenhang, dass die Ping-Periode von einigen Faktoren abhängig gemacht werden könnte. Ein *lazy* Ansatz wäre die Onlineprüfung eines Kontakts erst anzufordern, wenn der Benutzer ein Chatfenster geöffnet hat. Ein weiterer Ansatz könnte aktuell sichtbare Kontakte der Kontaktliste periodisch pingen.

Da in der aktuellen Implementierung Vollsicherungen der Kontaktliste und Chatlogs getätigt werden und diese unnötig Chord Kapazitäten belegen, ist die Implementierung einer inkrementellen Sicherung denkbar. So könnte diese analog zu der Zertifikatsliste für Benutzernamen, die in der Zertifikatinfrastruktur verwendet wird, implementiert wer-

den. Hierfür könnte die inkrementelle Sicherung als Liste durch die verkettete Anwendung der Hashfunktion auf einer definierten ID realisiert werden.

Weiter könnte durch die Einführung einer Ende-zu-Ende-Verschlüsselung ein potentielles *route-poisoning* unterbunden werden. Durch den Einsatz des abgewandelten rekursiven Routings kann potentiell jeder Knoten zwischen dem antwortenden und anfragenden Knoten die Antwort fälschen, da hier lediglich eine Hop-to-Hop Verschlüsselung erfolgt. Nichtsdestotrotz ist diese Hop-to-Hop Verschlüsselung für viele Nachrichtentypen eine Ende-zu-Ende-Verschlüsselung, da diese direkt an den Empfänger versendet werden. Eine genaue Auflistung enthält Abschnitt 7.2.4. Ein möglicher Ansatz wäre die Einführung eines iterativen Routings. Hierdurch erhält der Anfrager eine größere Kontrolle über den Routingvorgang und kann diesen steuern.

Eine weitere Idee wäre eine Erweiterung der aktuellen Chatfunktionalitäten durch einen Datentransfer und / oder Gruppenchat.

Zuletzt ist eine Portierung auf mobile Geräte interessant. Hierfür bietet PhoneGap [GP12] die Möglichkeit Webanwendungen auf gängige mobile Betriebssysteme wie Android, iOS, WindowsPhone, etc. zu portieren. Dies scheint attraktiv in Hinblick auf die ständig wachsende Smartphone-Sparte.

Anhang A

Chord Methoden

Das nachfolgende Listing A.1 zeigt einen Auszug der abstrakten Klasse Chord, welche alle notwendigen Methodenköpfe enthält. Das anschließende Listing A.2 zeigt ebenso einen Auszug der abstrakten Klasse Chord, welche alle notwendigen Callbacks in Form von Schnittstellen enthält.

Listing A.1: Auszug aus der abstrakten Klasse Chord

```
enum EntryType{
    UNSIGNED, SIGNED, SIGNED_ENCRYPTED;
}

abstract ID getID();
abstract void setID(ID nodeID) throws IllegalStateException;
abstract void create(Identity identity,ContactData contactDetails,ChordCallback
    callback) throws ServiceException;
abstract void join(Identity identity, boolean wantRegister, ContactData contactDetails,
    ID bootstrapID, ChordCallback callback) throws ServiceException;
abstract void leave() throws ServiceException;
abstract Identity getIdentity();
abstract void retrieve(Key key, ChordCallback callback);
abstract void insert(EntryType type, String entryClass, Key key, JsonSerializable
    entry, ChordCallback callback)
abstract void remove(EntryType type, String entryClass, Key key, JsonSerializable
    entry, ChordCallback callback);
abstract void sendMsg(ID targetID, String msg, MessageSupport callback);
abstract void startAVStream(ID targetID);
abstract void stopAVStream(ID targetID);
abstract void searchContact(String username, SearchSupport callback);
abstract void searchContact(ID compressedPubKey, SearchSupport callback);
abstract void modifyContactDetails(ContactData details, ModifyContactSupport callback);
abstract void deleteMyCertificate(ModifyContactSupport callback);
```

Listing A.2: Auszug aus den Callback Interfaces der abstrakten Klasse Chord

```
interface ChordCallback {
        enum ChordCallbackFailure{
            NONE, UNKNOWN_TYPE, COMMUNICATION_ERROR, NOT_ALLOWED,
            INVALID_RETRIEVAL,UNKNOWN;
        }
        enum ChordJoinFailure{
            NONE, BOOTSTRAP_NODES_OFFLINE, CERTIFICATE_CHECK_FAILED,
            REGISTER_BEFORE_JOIN, USER_ALREADY_EXISTS,
            CERT_INSERT_FAILED, COM_INVALID_API_KEY,
            COM_INVALID_PEER_ID, COM_SERVER_UNREACHABLE,
            COM_ALREADY_LOGGED_IN, COM_OTHER;
        }
        void retrieved(Key key, JsonSerializable entry, ChordCallbackFailure failure);
        void inserted(EntryType type, String entryClass, Key key, JsonSerializable
    entry, ChordCallbackFailure failure);
        void removed(EntryType type, String entryClass, Key key, JsonSerializable
    entry, ChordCallbackFailure failure);
        void joined(Certificate cert, ChordJoinFailure failure);
}

interface MessageSupport{
        enum MessageSupportFailure{
            NONE, COMMUNICATION_ERROR, NODE_PROBABLY_OFFLINE,
            NODE_OFFLINE, UNKNOWN;
        }
        void sentMsg(ID toPeer, String msg, MessageSupportFailure failure);
}

interface MessageReceiver {
        void recvMsg(ID fromPeer, String msg);
}

interface AVSupport{
        enum AVSupportFailure{
            NONE, COMMUNICATION_ERROR, NODE_PROBABLY_OFFLINE,
            NODE_OFFLINE, STOPPED_STREAMING, REFUSED_STREAMING,
            UNKNOWN;
        }
        void incomingStream(ID fromPeer, String othersMediaStream, String
    myMediaStream);
        void stopStreaming(ID peer, AVSupportFailure failure);
}

interface SearchSupport{
        enum SearchSupportFailure{
            NONE,ALREADY_SEARCHING, INVALID,
            FAILED, UNKNOWN;
        }
```

```
        void searchResult(Certificate cert, String searchedFor, SearchSupportFailure
    failure);
}

interface ModifyContactSupport{
        enum ModifyContactSupportFailure{
            NONE, STILL_MODIFYING, STILL_DELETING,
            FAILED, UNKNOWN;
        }
        void modifiedContactDetails(ContactData details, ModifyContactSupportFailure
    failure);
        void deletedOwnContact(ModifyContactSupportFailure failure);
}
```

Anhang B

Dynamische Konfigurationsmerkmale

Tabelle B.1 zeigt eine vollständige Aufzählung der dynamischen Konfigurationsmerkmale gruppiert nach deren Zugehörigkeit.

Tabelle B.1: Dynamische Konfigurationsmerkmale.

Modul	*Merkmal*
Logging	(De-)Aktivierung des Loggers per Boolean Wahl des Loggers: ServerLogger, ConsoleLogger oder DummyLogger LogLevel. Ab welchen LogLevel sollen Lognachrichten erfasst werden
PeerJS Server	Wann wird der PeerJS Server als nicht erreichbar eingestuft (Angabe in ms)
Chord	Periode des QueueCheckers (Angabe in ms) Ab wann werden Nachrichten vom QueueChecker erneut versendet. (Angabe in ms) Anzahl an Versandwiederholungen, bis der QueueChecker den Knoten als unerreichbar einstuft Startzeit und Periode aller Aufrechterhaltungsjobs* (Angabe in ms) Anzahl an Nachfolgern* Maximales Alter von Monitoringdaten (Angabe in ms) Anzahl an initialen retrieve Sendungen bei Suche in Benutzernamenliste Maximale Anzahl an unbeantworteten retrieve Sendungen
Anwendung	Benutzername regex** Passwort regex** Zeit in ms, die den Modulen bei einem Auslogwunsch gewährt wird, um deren Zustände zu sichern Zeit in ms nachdem nach einem Ausloggen die HTML Seite neugeladen wird
GUI	Repräsentierung der ID's [1]. Mögliche Werte: 0: binär, 1: dezimal, 2: hexadezimal In ChatView: Anzahl der beim Öffnen sichtbaren Einträge der Chat History In ChatView: Anzahl an zu ladenen Elementen bei der Erreichung des scrollcaps In SearchView: Anzahl an Suchtreffern, die initial angezeigt werden sollen In SearchView: Anzahl an Elementen, die beim runterscrollen, und erreichen des scrollcaps, der Trefferliste dynamisch nachgeladen werden sollen Für alle Views: Standardgröße der View Für alle Views: minimale Größe Für alle Views: Blinkintervall (in ms) der Titelleiste von Views, die auf sich aufmerksam machen
Monitoring	Boolean Angabe, ob sich die Software im monitoring Modus befindet. Falls im monitoring Modus: Abtastrate in ms der Erhebung der Monitoringdaten Falls im monitoring Modus: Dauer in ms des monitoring Modus

[*]: Aus OpenChord-Konfiguration entnommen [LK06].
[**]: Welche Zeichen können bzw. sollen enthalten sein.

Anhang C

Auswertung: Konfigurationsdatei

Die verwendete Konfigurationsdatei des Testszenarios enthält Tabelle C.1.

Tabelle C.1: Konfigurationsdatei Testszenario.

Attribut	*Wert*
LOG_LEVEL	WARN
INITIAL_PAGE_SIZE	10
DEFAULT_SIZE_SEARCH_VIEW_HEIGHT	500
AGE_OF_MONITORING_DATA	10800000
LOGGER_CLASS_NAME	ServerLogger
MONITORING_SAMPLING_RATE	5000
DEFAULT_SIZE_SEARCH_VIEW_WIDTH	450
CONNECTION_TIMEOUT	7000
MAX_PENDING_GET_CERT_REQUESTS	20
START_GET_CERT_REQUESTS	5
DEFAULT_SIZE_REQUEST_ADD_VIEW_HEIGHT	250
FIXFINGER_TASK_START	0
FIXFINGER_TASK_INTERVAL	12
DEFAULT_SIZE_CONTACTLIST_VIEW_WIDTH	500
RELOAD_PAGE_AFTER_ERROR_OR_LOGOUT	8000
DEFAULT_SIZE_MONITORING_VIEW_WIDTH	450
DEFAULT_SIZE_SETTINGS_VIEW_HEIGHT	500
DEFAULT_SIZE_MONITORING_VIEW_HEIGHT	500
CHECKPREDECESSOR_TASK_INTERVAL	12
DEFAULT_SIZE_REQUEST_AV_VIEW_WIDTH	250
WAIT_FOR_FINALIZING_MODULES	5000
MONITORING_TIME	3600000
USERNAME_REGEX	[a-zA-Z0-9].2,20
NUM_SUCCESSORS	2
MINIMAL_VIEW_SIZE_WIDTH	200
PASSPHRASE_REGEX	[a-zA-Z].1,20
INCREMENT_PAGE_SIZE	5
RESEND_INTERVAL	15000
IN_MONITORING_MODE	true
TRIES_BUDDY_PING_DECLARE_OFFLINE	1
DEFAULT_SIZE_CHAT_VIEW_WIDTH	300
DISPLAYED_REPRESENTATION	2
INITIAL_LAST_SHOWN_ELEMENTS	20
DEFAULT_SIZE_CONTACTLIST_VIEW_HEIGHT	600
LOGGING_OFF	false
DEFAULT_SIZE_REQUEST_AV_VIEW_HEIGHT	200
STABILIZE_TASK_INTERVAL	12
DECREMENT_PAGE_SIZE	10
CHECKPREDECESSOR_TASK_START	6
PERIODIC_BUDDY_PING_INTERVAL	10000
DEFAULT_SIZE_AV_STREAM_VIEW_WIDTH	320
RESENT_THRESHOLD	3
STABILIZE_TASK_START	12
FLASHING_HEADER	1000
DEFAULT_SIZE_SETTINGS_VIEW_WIDTH	600
PERIODIC_BUDDY_LIST_SAVER_INTERVAL	600000
CHECK_INTERVAL	8000
DEFAULT_SIZE_CHAT_VIEW_HEIGHT	350
DEFAULT_SIZE_REQUEST_ADD_VIEW_WIDTH	250
DEFAULT_SIZE_AV_STREAM_VIEW_HEIGHT	240

Anhang D

Auswertung: Kryptozeit

Die in der Auswertung zur Kryptozeit erstellten Tabellen bauen auf die Abbildungen D, D.1, D.1 und D.2 auf.

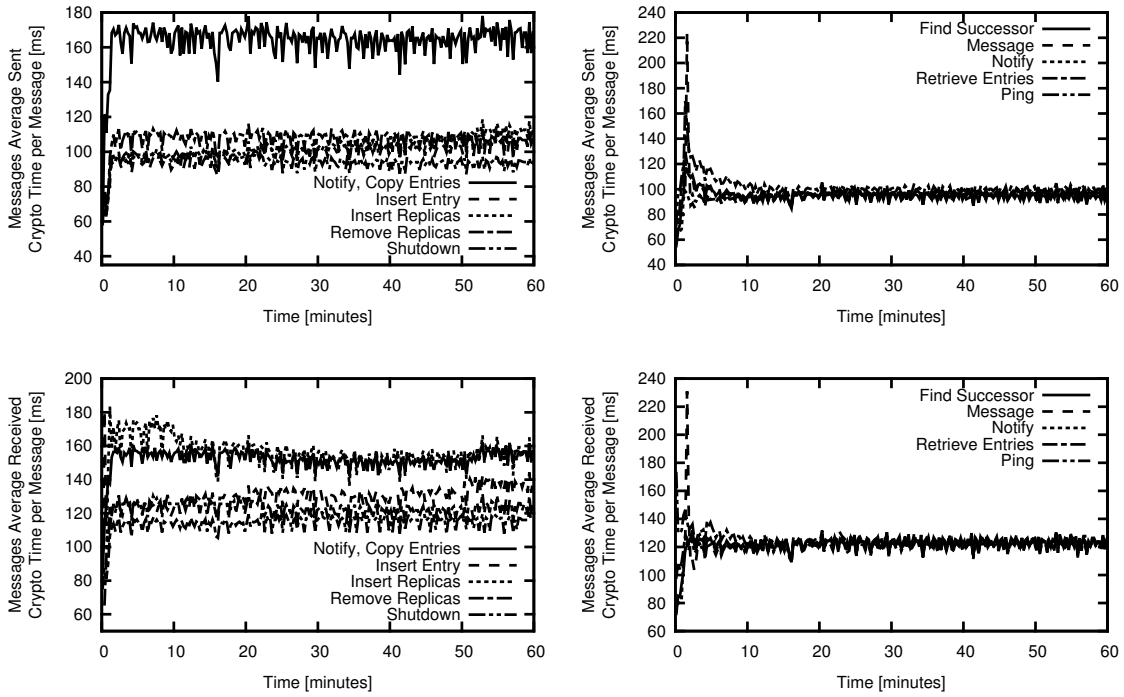


Abbildung D.1: Durchschnittliche Kryptozeit pro versendetem / empfangenem Nachrichtentyp.

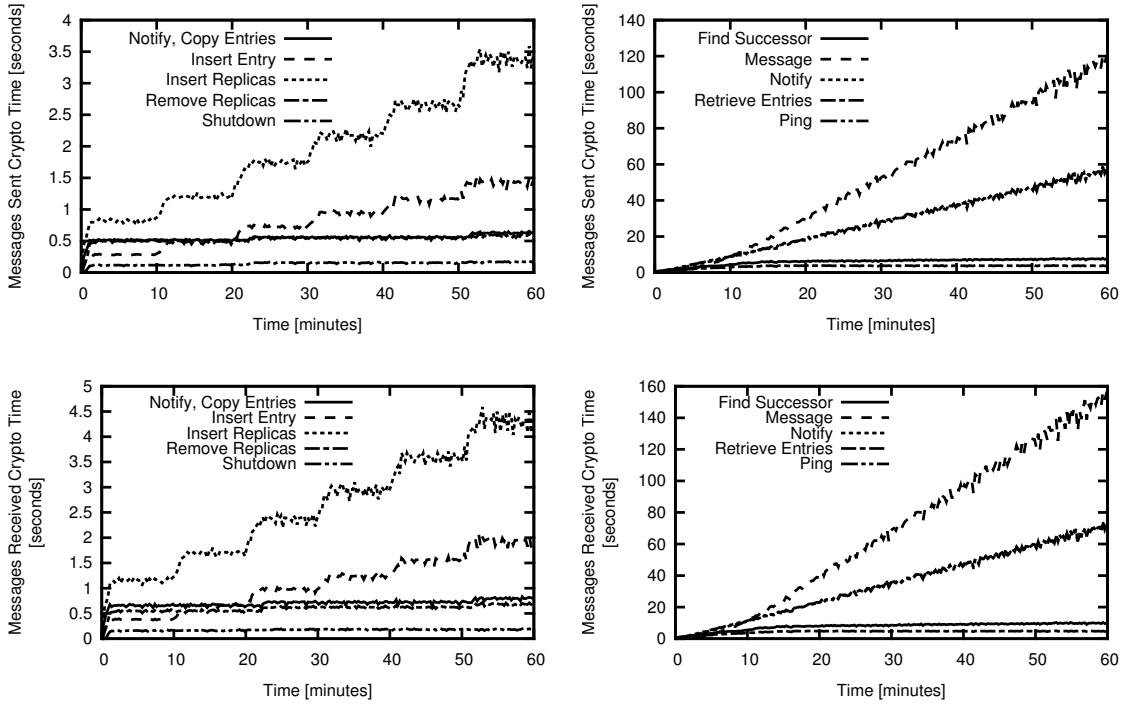


Abbildung D.2: Gesamte Kryptozeit versendeter / empfangener Nachrichten.

Anhang E

Auswertung: weitere Abbildungen

Dieser Anhang enthält weitere Abbildungen, auf die in der Auswertung nicht näher eingegangen wurde.

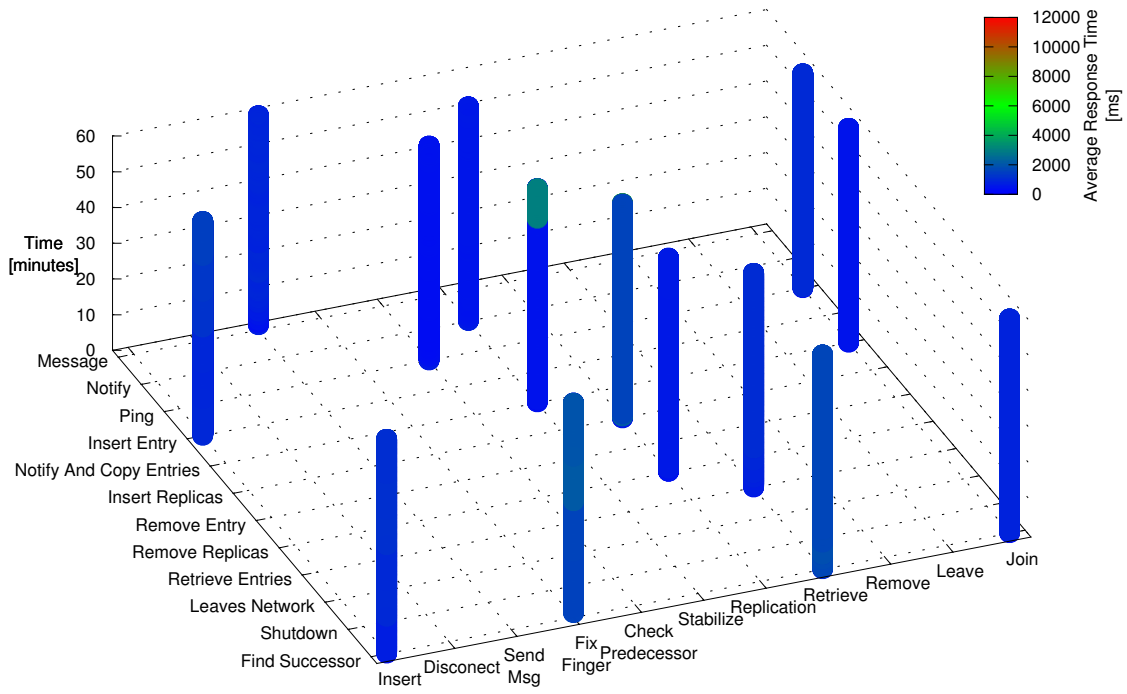


Abbildung E.1: Operationen / Nachrichten Überblick: durchschnittliche Antwortzeit.

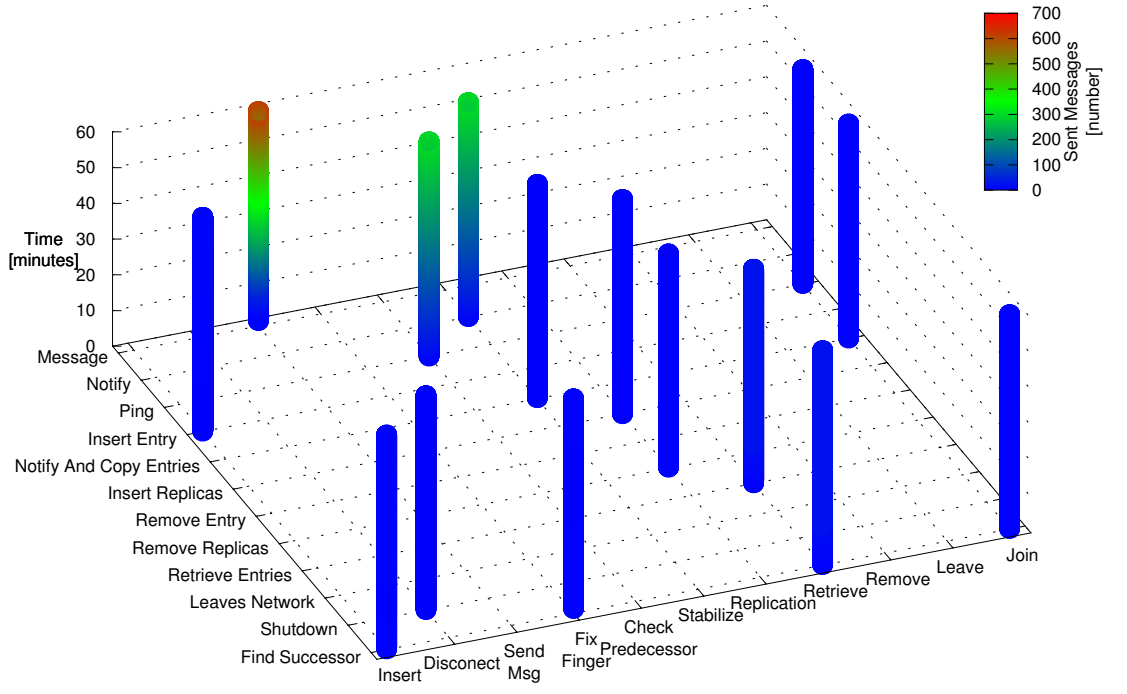


Abbildung E.2: Operationen / Nachrichten Überblick: versendete Nachrichten.

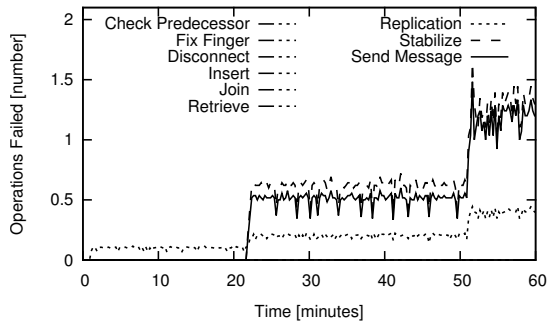


Abbildung E.3: Durchschnittliche Anzahl fehlgeschlagener Operationen.

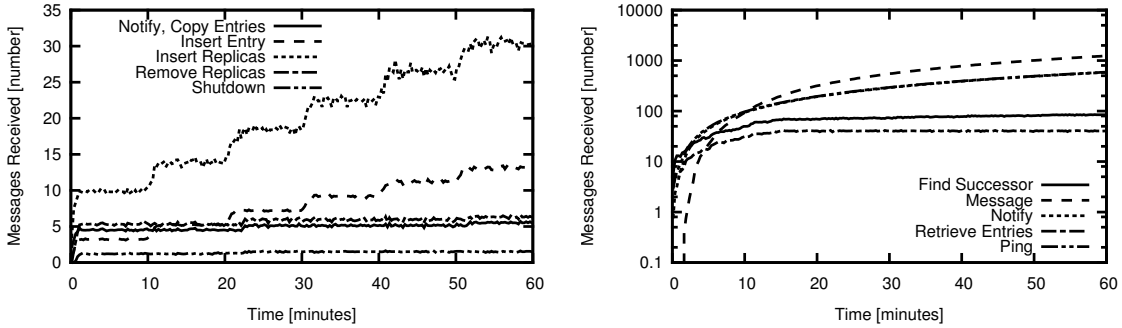


Abbildung E.4: Empfangene Nachrichten.

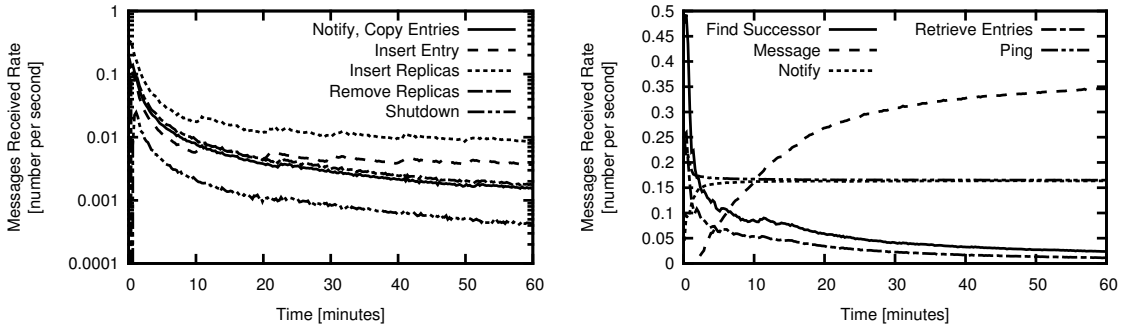


Abbildung E.5: Rate empfangener Nachrichten.

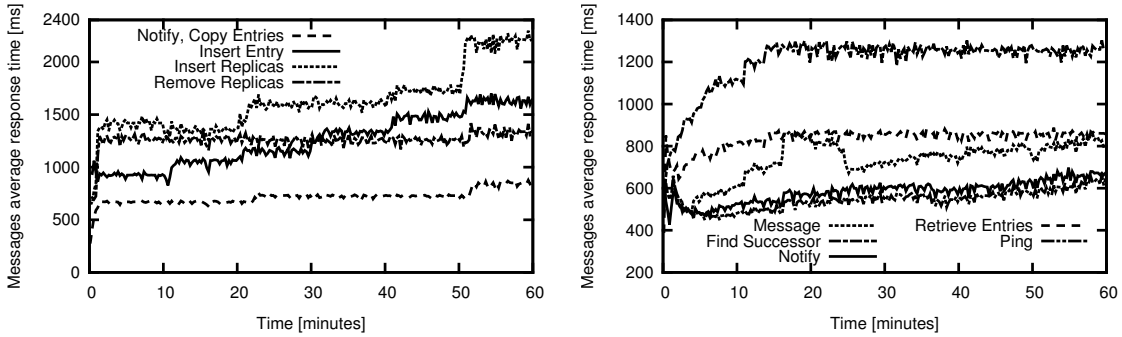


Abbildung E.6: Durchschnittliche Antwortzeit von Nachrichten.

Literaturverzeichnis

[ABGR13] ARANHA, Diego F.; BARRETO, Paulo S. L. M.; GEOVANDRO, C. C. F. P.; RICARDINI, Jefferson E.: A Note on High-Security General-Purpose Elliptic Curves. In: *IACR Cryptology ePrint Archive* 2013 (2013), S. 647.

[Ber06] BERNSTEIN, Daniel J.: Curve25519: New Diffie-Hellman Speed Records. In: *Proceedings of Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography.*, 2006, S. 207–228.

[BL14] BERNSTEIN, Daniel J.; LANGE, Tanja: *SafeCurves: Choosing Safe Curves for Elliptic-Curve Cryptography*. http://safecurves.cr.yp.to/, 2014.

[BLS12] BERNSTEIN, Daniel J.; LANGE, Tanja; SCHWABE, Peter: The Security Impact of a New Cryptographic Library. In: *Proceedings of the Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago.*, 2012, S. 159–176.

[BNBJ13] BURNETT, Daniel; NARAYANAN, Anant; BERGKVIST, Adam; JENNINGS, Cullen: WebRTC 1.0: Real-Time Communication Between Browsers / W3C. 2013. W3C Working Draft. . http://www.w3.org/TR/2013/WD-webrtc-20130910/

[Bro10] BROWN, Daniel R. L.: *SEC 2: Recommended Elliptic Curve Domain Parameters*. http://www.secg.org/download/aid-784/sec2-v2.pdf, 2010.

[Cro06] CROCKFORD, D.: *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). `http://www.ietf.org/rfc/rfc4627.txt`. Version: Juli 2006 (Request for Comments).

[DBL14] BSI: Heartbleed Bug Ist Kritisch. In: *Datenschutz und Datensicherheit* 38 (2014), Nr. 6, S. 425.

[DS07] DAN SHUMOW, Niels F.: *On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng*. http://rump2007.cr.yp.to/15-shumow.pdf, 2007.

[ESB14] EMILY STARK, Mike H.; BONEH, Dan: *SJCL: Stanford Javascript Crypto Library*. http://bitwiseshiftleft.github.io/sjcl/, 2014.

[Fou] FOUNDATION, Eclipse: *Jetty*. http://www.eclipse.org/jetty/, .

[GGP13] GLENN GREENWALD, Ewen M.; POITRAS, Laura: *Edward Snowden: the Whistleblower behind the NSA Surveillance Revelations.* https://www.chebayadkard.com/uploadfile/english-article574.pdf, 2013.

[GJ14] GARNOCK-JONES, Tony: *Emscripten-Compiled Javascript Version of NaCl, the Networking and Cryptography Library*. https://github.com/tonyg/js-nacl, 2014.

[Goo12a] GOOGLE: *GWT Project: Google Web Toolkit.* http://www.gwtproject.org/, 2012.

[Goo12b] GOOGLE: *GWT Project: Google Web Toolkit - DevGuide*. http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsCompatibility.html, 2012.

[Goo12c] GOOGLE: *GWT Project: Google Web Toolkit - Security*. http://www.gwtproject.org/articles/security_for_gwt_applications.html, 2012.

[GP12] GHATOL, Rohit; PATEL, Yogesh: *Beginning PhoneGap: Mobile Web Framework for JavaScript and HTML5*. Berkely, CA, USA : Apress, 2012. http://phonegap.com/

[GPM+08] GRAFFI, Kalman; PODRAJANSKI, Sergey; MUKHERJEE, Patrick; KOVACEVIC, Aleksandra; STEINMETZ, Ralf: A Distributed Platform for Multimedia Communities. In: *Proc. of IEEE ISM '08*, 2008.

[GS11] GUPTA, Kamlesh; SILAKARI, Sanjay: *ECC over RSA for Asymmetric Encryption: A Review*. 2011.

[Gua09] GUAN, Zhi: *Jscrypto: A JavaScript Cryptography Library*. https://code.google.com/p/jscryptolib/, 2009.

[Kfu11] KFUNTAK: *GWT Professional JSON Serializer*. https://code.google.com/p/gwtprojsonserializer/, 2011.

[KL07] KAFFILLE, Sven; LOESING, Karsten: Open Chord version 1.0.4 User's Manual / Lehrstuhl für Praktische Informatik. 2007 (Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik 74). Forschungsbericht. 31 S. : Ill.

[LCP+05] LUA, Eng K.; CROWCROFT, Jon; PIAS, Marcelo; SHARMA, Ravi; LIM, Steven: A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. In: *IEEE Communications Surveys and Tutorials* 7 (2005), Nr. 1-4, S. 72–93.

[LK06] LOESING, K.; KAFFILLE, S.: *OpenChord*. http://sourceforge.net/projects/open-chord/, 2006.

[MB14a] MICHELLE BU, Eric Z.: *PeerJS API*. http://peerjs.com/docs/#api, 2014.

[MB14b] MICHELLE BU, Eric Z.: *PeerJS Server: Brokering PeerJS Peers*. https://github.com/peers/peerjs-server, 2014.

[MB14c] MICHELLE BU, Eric Z.: *PeerJS simplifies WebRTC peer-to-peer data, video, and audio calls.* http://peerjs.com/, 2014.

[Mic] MICROSOFT: *SimPastry*. http://research.microsoft.com/en-us/downloads/33F10BF8-42BD-4C7F-B24C-AD73D4CE2FF7/default.aspx, .

[MM02] MAYMOUNKOV, Petar; MAZIÈRES, David: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: *IPTPS '02: Proc. of the Int. Workshop on Peer-To-Peer Systems* Bd. 2429, Springer, 2002 (LNCS).

[nsa09] *The Case for Elliptic Curve Cryptography*. http://www.nsa.gov/business/programs/elliptic_curve.shtml, 2009.

[RD01] ROWSTRON, Antony I. T.; DRUSCHEL, Peter: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: *Proceedings of the Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, 2001, S. 329–350.

[Res11] RESCORLA, Eric: *Proposed WebRTC Security Architecture*. http://www.ietf.org/proceedings/82/slides/rtcweb-13.pdf, 2011.

[Ric] RICE UNIVERSITY, HOUSTON, USA: *FreePastry*. http://www.freepastry.org/FreePastry/, .

[RP13] RAMEJAN, Mooreds; PAPPIN, John: *Crypto Module for the Google Web Toolkit (GWT).* https://code.google.com/p/gwt-crypto/, 2013.

[SENB07] STEINER, Moritz; EN-NAJJARY, Taoufik; BIERSACK, Ernst W.: A Global View of KAD. In: *ACM SIGCOMM'07: Proc. of the ACM Conf. on Internet Measurement*, 2007.

[SMK+01] STOICA, Ion; MORRIS, Robert; KARGER, David; KAASHOEK, M. F.; BALAKRISHNAN, Hari: Chord: A Scalable Peer-to-Peer Lookup Service for In-

ternet Applications. In: *SIGCOMM '01: Proc. of the Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM, 2001. ISBN 1–58113–411–8.

[sta13] *StartSSL PKI*. https://www.startssl.com/, 2013.

[THT14] TED HARDIE, Cullen J.; TURNER, Sean: *Real-Time Communication in Web-Browsers Status Pages*. http://tools.ietf.org/wg/rtcweb/, 2014.

[TKLB07] TERPSTRA, Wesley W.; KANGASHARJU, Jussi; LENG, Christof; BUCHMANN, Alejandro P.: Bubblestorm: Resilient, Probabilistic, and Exhaustive Peer-to-peer Search. In: *SIGCOMM Comput. Commun. Rev.* 37 (2007), August, Nr. 4, 49–60. http://dx.doi.org/10.1145/1282427.1282387. DOI 10.1145/1282427.1282387. ISSN 0146–4833.

[tom14] *Apache Tomcat*. http://tomcat.apache.org/, 2014.

[Wu13] WU, Tom: *JSBN: RSA and ECC in JavaScript*. http://www-cs-students.stanford.edu/tjw/jsbn/, 2013.

[Zet13] ZETTER, Kim: *How a Crypto 'Backdoor' Pitted the Tech World Against the NSA*. http://www.wired.com/2013/09/nsa-backdoor/, 2013.

Ehrenwörtliche Erklärung

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

Düsseldorf, 2. Juli 2014 Andreas Disterhöft

Diese DVD enthält:

- Eine *pdf* Version der Masterarbeit
- Alle LaTeXund Grafik Dateien sowie entsprechende Skripte, die verwendet wurden
- Der Quelltext der Software, welcher während der Masterarbeit entstanden ist
- Messdaten, die während der Auswertung angefallen sind
- Referenzierte Webseiten und wissenschaftliche Publikationen