HAW HAMBURG

BACHELORTHESIS
Andreas Müller

Chord-Comm: Eine modulare TypeScript Middleware für Chord über WebRTC.

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Andreas Müller

Chord-Comm: Eine modulare TypeScript Middleware für Chord über WebRTC.

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Eingereicht am: 06. Februar 2020

Andreas Müller

Thema der Arbeit

Chord-Comm: Eine modulare TypeScript Middleware für Chord über WebRTC.

Stichworte

Chord, Distrubuted Hash Table, WebRTC, Browser, Framework

Kurzzusammenfassung

Beim Thema der Kommunikation mit anderen Teilnehmern im Internet über einen Browser existieren im wesentlichen zwei Konzepte: *Client-Server* und *Peer-to-Peer*. Bei den *Client-Server* Lösungen agiert immer mindestens ein Server als Koordinator zwischen den einzelnen *Clients* - der Server ist sozusagen der Knotenpunkt. Bei *Peer-to-Peer* Lösungen agieren die einzelnen *Peers* als eigenständige und gleichberechtigte Partner im Netz.
Wenn diese beiden Konzepte jetzt auf ein Informationssystem angewendet werden, welches Informationen speichert und abrufbar macht, werden die Unterschiede der beiden Konzepte sehr deutlich. Wo die Zuordnung der Informationen bei der Client-Server Lösung noch eindeutig war - die Informationen verwaltet der Server - so ist dies in einem *Peer-to-Peer* Netzwerk nicht mehr ganz so eindeutig. Da in einem *Peer-to-Peer* Netzwerk der Server als Knotenpunkt nicht mehr existiert und Informationen nun einem *Peer* statt dem Server zugeordnet werden müssen, braucht es Regeln. Diese Regeln definieren, wie diese Zuordnungen ablaufen. Außerdem wird eine Ordnung in einem *Peer-to-Peer* Netzwerk benötigt, da kein konkreter Server als Koordinator der Kommunikation dient.

In dieser Arbeit wird eine Implementation von „Chord“ im Verbund mit „WebRTC“ in Form einer modularen *Middleware* erstellt. Ziel ist es, die Komplexität der Erstellung eines dezentralen *Peer-to-Peer* Informationssystems für den Nutzer der *Middleware* zu abstrahieren und leicht zugänglich zu machen. Dies wird durch die Veröffentlichung der *Middleware* als Modul in der *Registry* des *Node Package Manager (NPM)* ermöglicht. Somit kann die *Middleware* durch den *NPM* in Projekten eingebunden und genutzt werden.

Andreas Müller

Title of Thesis

Chord-Comm: A modular TypeScript middleware for Chord through WebRTC.

Keywords

Chord, Distrubuted Hash Table, WebRTC, Browser, Framework

Abstract

When it comes to communicating with other participants on the Internet via a browser, there are essentially two concepts: *client-server* and *peer-to-peer*. With client-server solutions, at least one server always acts as a coordinator between the individual clients. With peer-to-peer solutions, the individual peers act as independent and equal partners in the network.

If these two concepts are now applied to an information system that stores and makes information available, the differences between the two concepts become very clear. Where the assignment of the information was clearly defined in the client-server solution - the information is managed by the server - this is no longer quite so clearly defined in a peer-to-peer network. Since the server no longer exists as a coordinating node in a peer-to-peer network and information now has to be assigned to a peer instead of the server, rules are needed. These rules define how these assignments work. In addition, an order in a peer-to-peer network is required, since no concrete server serves as coordinator of the communication.

In this work an implementation of „Chord“ combined with „WebRTC“ in the form of a modular middleware is developed. The aim is to abstract the complexity of creating a decentralized *peer-to-peer* information system for the user of the middleware and to make it easily accessible. This is made possible by the publication of the middleware as a module in the Registry of the *NPM*. Thus, the middleware can be integrated and used easily by the *NPM* in projects.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Bei der im Rahmen dieser Bachelorarbeit entstandenen *Middleware* „ChordComm“ handelt es sich um eine *Middleware*, welche alle Komponenten besitzt, um ein dezentrales Informationssystem auf *Peer-to-Peer* Basis im Browser aufzubauen. Diese Arbeit basiert auf den theoretischen Vorarbeiten von „Chord: A Scaleable Peer-to-peer Lookup Service for Internet Applications“ und der dort vorgestellten Technologie *Chord* [3] .

„ChordComm“ setzt dabei auf die Definition mehrerer Komponenten. Jede Komponente besitzt dabei eine eindeutige Funktion. Die Schnittstellen der Komponenten werden durch Interfaces definiert. Diese eindeutigen Definitionen erlauben einfaches Ersetzen einzelner Komponenten durch neue, ohne dabei in das gesamte System eingreifen zu müssen. Die Zuweisung der Informationen/Schlüssel zu einzelnen *Peers* geschieht dabei wie durch *Chord* beschrieben [3].

„ChordComm“ kümmert sich neben der Ausführung der *Chord*-Logik auch um die Verwaltung der *WebRTC*-Verbindungen inklusive des Austausches der Sitzungsinformationen der einzelnen *WebRTC* Verbindungen über einen Server. Um diese Informationen zwischen den einzelnen *Peers* auszutauschen, wird Server Logik mitgeliefert. Diese Server Logik lässt sich als Modul in einem Node.js Server nutzen.

Um in einem strukturierten *Peer-to-Peer* System wie *Chord* unter Verwendung von *WebRTC* eine stabile Verfügbarkeit der Informationen gewährleisten zu können, nutzt „ChordComm“ eine Replizierungs-Strategie [10, 6].

1.1 Motivation

Die Motivation dieser Arbeit besteht darin, die Komplexität eines dezentralen *Peer-to-Peer* Informationssystems für Browser-Anwendungen mittels einer *Middleware* zu abstrahieren. Das so entstandene Modul soll eine gute Basis für Anwendungsfälle solcher

Systeme liefern. Die Aufteilung der *Middleware* in mehrere Module sorgt außerdem dafür, dass einzelne Komponenten durch Schnittstellengleiche ersetzt werden können. Hierdurch sollen die bestehenden Komponenten ohne Anpassung der Schnittstellen einfach durch neue ersetzt werden können.

1.2 Ziele

Im folgenden werden die Ziele von „ChordComm“ festgelegt.

Da „ChordComm“ durch die Nutzung von *Chord* ein Verteiltes System darstellt, werden die von M. van Steen und A.S. Tanenbaum genannten Design-Ziele [8, Kapitel 1.2] eines Verteilten Systems herangezogen.

- **Ressourcenverteilung [8, S. 7]** Es soll eine Verteilung der Ressourcen (Informationen) auf alle Teilnehmer stattfinden. Außerdem sollen die Ressourcen leicht erreichbar sein. *Chord* definiert hierbei, wie Ressourcen verteilt und gesucht werden können [3, S.2 System Model][2, 4.2].
- **Transparente Verteilung [8, S. 8]** Die Verteilung der Ressourcen soll transparent sein. Der Nutzer soll somit nicht erkennen, ob und wie die Information verteilt ist. *Chord* verlangt hierfür ein *Consistent Hasching* [3, S. 3 Constistent Hashing].
- **Offenheit [8, S. 12]** Es soll eine Offenheit des Systems bestehen. Daraus folgt, dass die einzelnen Komponenten von „ChordComm“ sauber getrennt, möglichst lose gekoppelt und durch Interfaces komplett beschrieben werden.
- **Skalierbarkeit [8, S. 15]** Das System soll auf allen Ebenen skalierbar sein:
  - **Größen-Skalierbarkeit** Neue Nutzer und Ressourcen müssen dem System hinzugefügt werden können, ohne spürbare Performance-Einbußen.
  - **Geographische Skalierbarkeit** Nutzer und Ressourcen können geographisch weit auseinander liegen, ohne dass es spürbare Effekte auf das System hat.
  - **Administrative Skalierbarkeit** Das System muss auch trotz vieler verschiedener administrativen Ebenen skalierbar sein. In *Chord* sind alle Teilnehmer gleichberechtigt und organisieren sich durch das Hashing selbst zu einem strukturierten System [3, S. 4].

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen für die verwendeten Technologien *Chord* und *WebRTC* kurz zusammengefasst. Darüber hinaus wird noch auf die gewählte Replizierungs-Strategien im System eingegangen, welche für die Verteilung der Informationen notwendig sind.

2.1 Chord

In diesem Abschnitt wird die Organisation der Teilnehmer erläutert. Des Weiteren wird das Einbringen und Abrufen von Informationen erklärt.

2.1.1 Chord allgemein

Chord ist eine Distributed Hash Table (DHT), welche durch Kontrollalgorithmen eine Ordnung in einem dezentralen Informationssystem definiert. Jeder *Peer* bekommt dabei durch *Consistent Hashing* [3, 7] eine eindeutige ID im möglichen Bereich von üblicherweise 256 Bit[1] zugewiesen. Informationen besitzen einen Schlüssel, welcher sich im selben Wertebereich befindet. *Chord* hat im Wesentlichen 4 Aufgaben zu bewältigen:

1. Suchen von Schlüsseln und derer zuständige Knoten [3, S. 4 Scalable Key Location].
2. Einfügen von neu hinzukommenden Knoten [3, S. 5 Node Joins].
3. Entfernen von Knoten, die das Netz verlassen [3, S. 8 Failures and Replication].
4. Eigenständige Stabilisierung des Netzes [3, S. 7 Stabilization].

[1] bei Verwendung von SHA 256

Die Suche nach Informationen wird implizit durch die Suche nach einem Schlüssel im Netz abgedeckt. Für einen Schlüssel wird der zuständige Knoten gesucht und somit auch direkt nach der gesuchten Information [2, 4.2].

2.1.2 Stabilisierung des Chord-Ring

Um einen *Chord*-Ring zu stabilisieren sind zwei Abläufe nötig:

- Bilden und halten eines stabilen Rings mit allen Nodes.
- Bilden und halten einer stabilen Fingertable.

Bildung und Stabilisierung des Rings

In der folgenden Abbildung 2.1 ist der Ablauf im *Chord*-Ring gezeigt. Dargestellt sind die Schritte, die sicherstellen, dass die Nodes einen Ring bilden, also ihren passenden Vorgänger (*predecessor*) und Nachfolger (*successor*) kennen.

Ausgangssituation der Abbildung ist, dass *Node A* als aktuellen *successor* die *Node C* führt. *Node A* fragt nun *Node C* nach seinem aktuellen *predecessor*. Daraufhin wird *Node B* als predecessor von *Node C* zurückgeliefert. Da *Node B* für *Node A* ein passenderer *successor* ist, wird dieser nun gespeichert. Anschließend wird der neue *successor* von *Node A* darüber informiert, dass er der neue *successor* von *Node A* ist und überprüft, ob *Node A* nun auch für ihn ein besserer *predecessor* ist. Sofern dies der Fall ist, wird *Node B* ihn als neuen *predecessor* setzen.

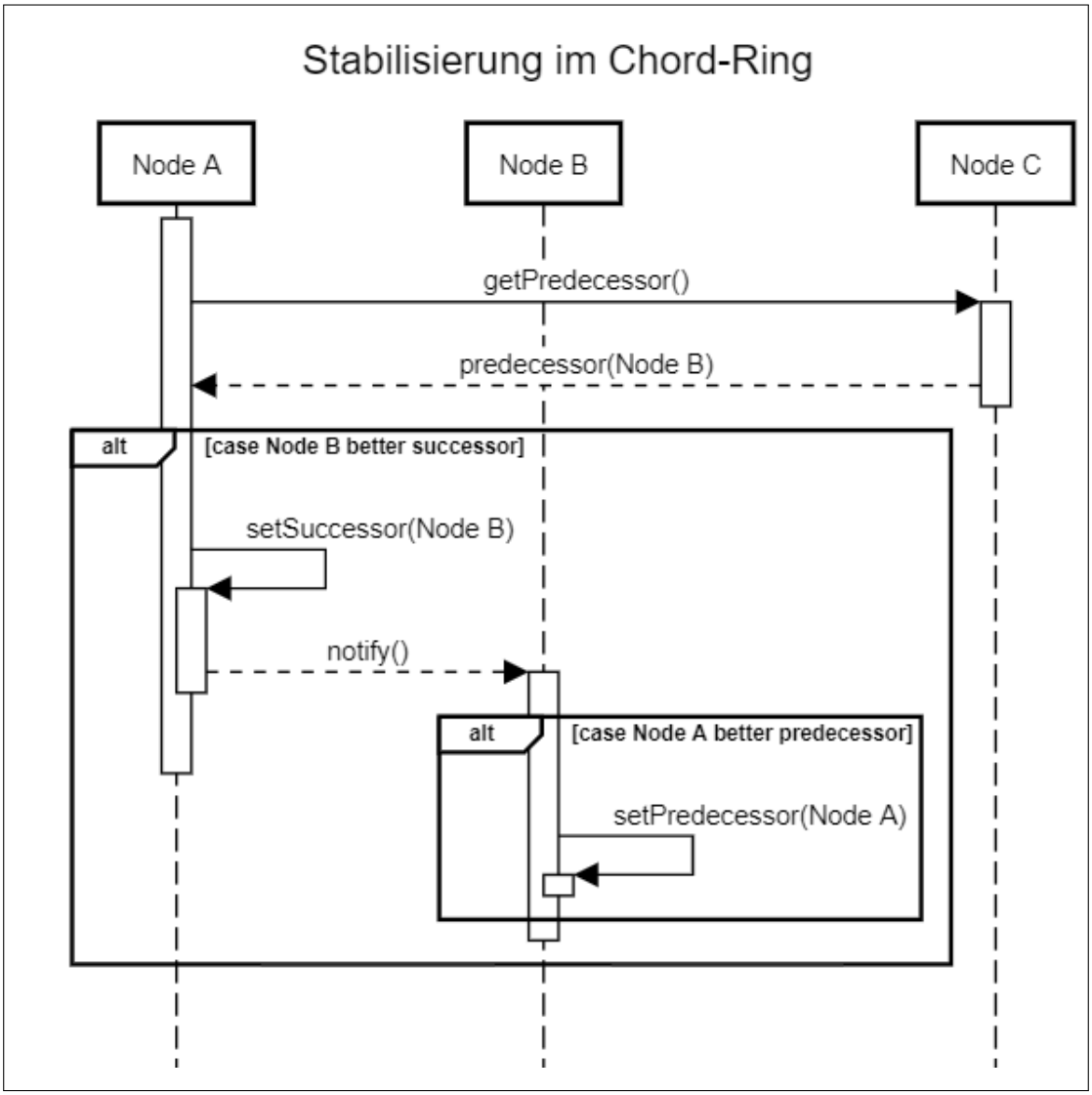

Abbildung 2.1: Stabilisierung im Chord-Ring

Dieser Mechanismus wird in regelmäßigen Abständen von jeder Node ausgeführt und stellt einen geordneten Ring sicher.

Bildung und Stabilisierung der Fingertable

Die Fingertable enthält eine Liste von Indizes und dazu gehörigen *Nodes*. Durch diese lassen sich von der Lokalen *Node* aus die *Nodes* errechnen, die für eine Anfrage am besten

geeignet sind. Die Fingertable stellt eine Erweiterung der Ring-Struktur dar.

Die Fingertable verweist - ausgehend von der ID der eigenen *Node* - auf N andere *Nodes*, wobei N der Länge des Hashwertes entspricht (idR. 256). Die Abstände sind dabei exponentiell mit der Basis 2 festgelegt. Zum Beispiel hat der $k$te Eintrag der Fingertable der *Node* n die Referenz: $Finger_k = (n + 2^{(k-1)}) \bmod 2^m, 1 \leq k \leq m$. Somit ist bei einem Stabilisierten *Chord*-Ring der kürzeste Weg zu einer beliebigen *Node* maximal $O(\log N)$ Schritte lang. [3]

Soll nun die passende *Node* für einen Finger N gefunden werden, so wird der aktuell bekannte naheliegendste *successor* von N ermittelt und die Suchanfrage an diesen weitergegeben. Dieser Vorgang wird so lange rekursiv über die verschiedenen auf dem Weg liegenden *Nodes* wiederholt, bis der direkte *predecessor* X der gesuchten *Node* gefunden wurde. Somit wäre der *successor* von X die für N zuständige *Node*.

2.1.3 Informationen im Ring

Anfrage von Information

Wie bereits bekannt werden Informationen im *Chord*-Ring einem Knoten zugewiesen. Sobald ein Knoten A eine Information anfragen will, ermittelt er über seine lokale *Fingertable* den ihm bekannten *predecessor* B des Schlüssels der Information. Dieser Knoten B prüft nun, ob sein direkter *successor* C für den Schlüssel zuständig ist. Ist der direkte *successor* C für den Schlüssel zuständig, wird die Anfrage an diesen weitergeleitet. Ansonsten wiederholt der Knoten B den ersten Schritt und schickt die Anfrage an den ihm bekannten *predecessor* des Schlüssels weiter. Dieser Vorgang wird wiederholt, bis der Knoten gefunden wurde, welcher für den angefragten Schlüssel zuständig ist.

Dieser ganze Vorgang hat einen Aufwand von $\mathcal{O}(\log N)$, wobei N die Anzahl der Knoten im Ring ist [3, S. 3 4.1].

Ist der für eine Information zuständige Knoten und die Information gefunden, kann diese auf verschiedene Arten zum Anfragenden übertragen werden:

1. Die Information wird über den gleichen Weg, den auch die Suche genutzt hat, zurückgegeben. Somit sind alle Knoten auf dem Weg zur Information auch in den Austausch dieser involviert. Direkter Nachteil ist, dass für einen Prozess, der logisch nur zwei Knoten anspricht, $\mathcal{O}(\log N)$ Knoten genutzt werden und alle Knoten, über

die gelaufen wurde, auf eine Antwort warten müssen. Möglicher großer Nachteil wäre der Wegfall eines Knotens auf dem Weg nach der Anfrage, da der Rückweg zwingend in dieser Variante gleich bleiben muss.

2. Bei der Anfrage der Information wird auch übermittelt, wer diese Information angefragt hat. Der für die Information zuständige Knoten schickt seine Antwort auf die Anfrage nun basierend auf seiner lokalen *Fingertable* zu dem größten bekannten *predecessor* des Anfragenden oder direkt zum Anfragenden, sofern eine direkte Verbindung besteht. Der Aufwand wäre auch hier $\mathcal{O}(\log N)$ und es würden wieder durchschnittlich $\mathcal{O}(\log N)$ Knoten involviert.

3. Sowohl der Anfragende als auch der Information-beherbergende Knoten bauen eine temporäre Verbindung auf. Diese besteht nur für die Dauer der Übertragung der Information. Potentiell sinnvoller Ansatz um andere Knoten bei großen Informationsmengen zu entlasten, jedoch in einem *Peer-to-Peer* Netz auf *WebRTC* schwer umsetzbar. Grund hierfür ist, dass für den Aufbau einer Verbindung vorab viele Informationen ausgetauscht werden müssten. Das wiederum müsste über die Knoten auf dem Pfad geschehen.

In „ChordComm“ wurde der zweite Ansatz gewählt, da er die Interaktion der Knoten auf dem Weg vom Anfragenden zum Ziel Knoten minimal belastet und somit ein Ausfall von Knoten besser gehandhabt werden kann.

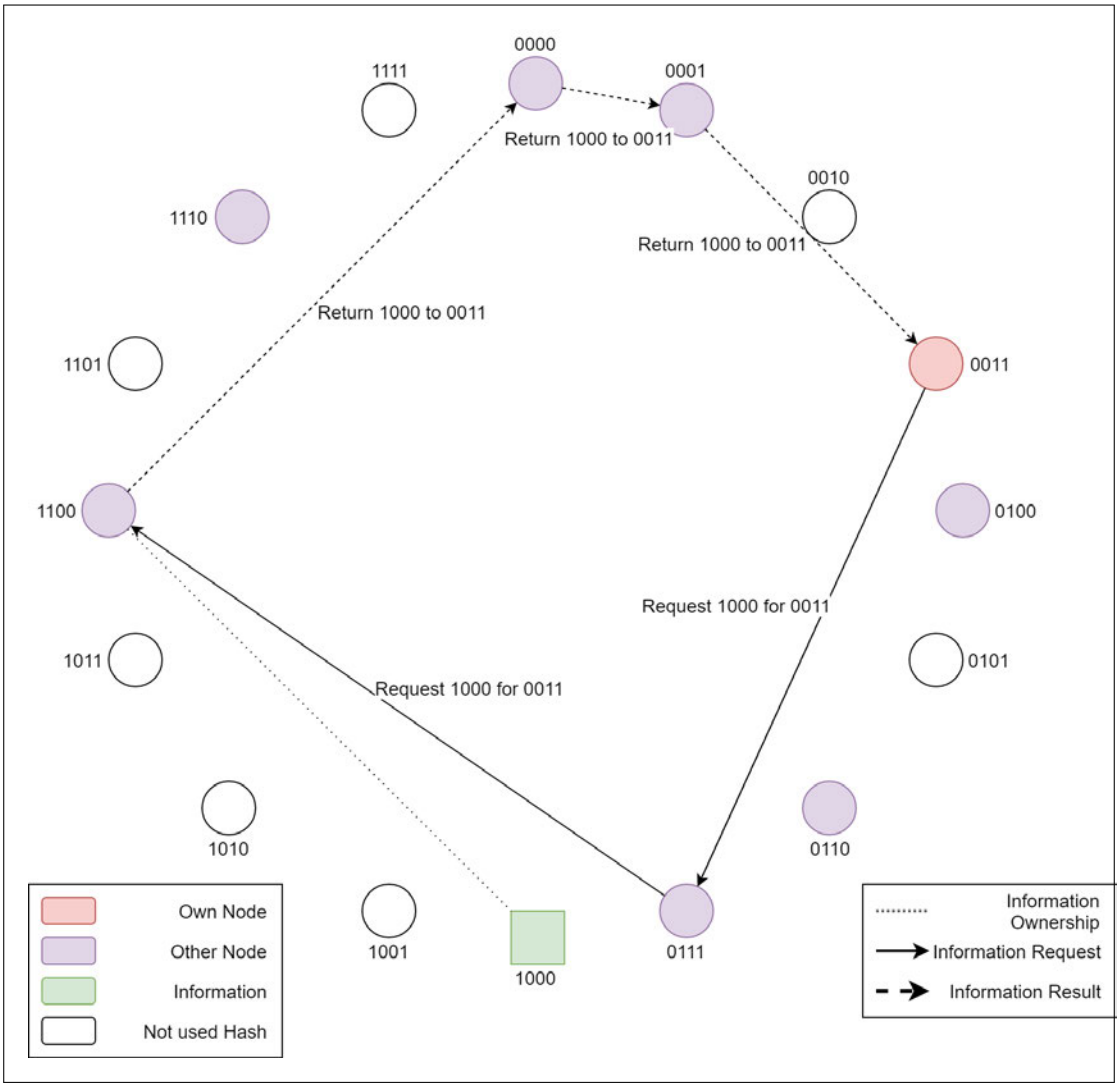

Abbildung 2.2: Anfrage und Rückführung der Information im Ring

In Abbildung 2.2 ist der eben genannte Ablauf der Anfrage einer Information zu sehen. Die Node *0011* (roter Kreis) möchte die Information *1000* (grünes Rechteck) anfragen. Dazu schickt sie eine Anfrage an den ihr bekannten *predecessor* der Information *1000*, welcher die Node *0111* ist. Die Node *0111* sucht aus seiner lokalen *Fingertable* nun den *predecessor* für die Information. In diesem Fall ist die Node selber der bekannte *predecessor* der Information, somit ist der direkte *successor*, die Node *1100*, wahrscheinlich der Besitzer der Information und die Anfrage wird an diesen weitergeleitet. Die Node *1100* besitzt in diesem Fall tatsächlich die Information und schickt diese auf die gleiche Art

und Weise zurück an die Node *0011*. Dabei wird ein neuer Pfad über die Node *0000* und Node *0001* genutzt.

Erstellung von Information

Analog zu der Anfrage einer Information kann über den gleichen Weg auch eine Information in das Netz eingebracht werden. Dafür wird die Information direkt dem passendsten bekannten *predecessor* des Schlüssels mitgegeben. Dieser prüft dann, ob sein direkter *successor* ein *successor* des Schlüssels ist. Sofern das der Fall ist, leitet er die Information direkt an seinen *successor* weiter, ansonsten wieder basierend auf seiner lokalen *Fingertable* zu dem größten *predecessor* des Schlüssels der Information. Wie auch bei der Anfrage gibt es auch hier die Möglichkeit, vorab eine Verbindung zum Ziel-Knoten aufzubauen, um die anderen Knoten auf dem Weg zu entlasten. Dies ist gerade bei großen Informationsmengen sinnvoll. Aus den gleichen Gründen wie bei der Anfrage von Informationen wird auch hier wieder nur der zweite Ansatz gewählt.

Backup von Informationen

Chord definiert in seinen Kontrollstrukturen keine Abläufe, um Informationen für höhere Verfügbarkeit zu replizieren. Auf die in „ChordComm“ genutzte Strategie zur Replizierung von Informationen wird in der Sektion 2.3 „Replizierung der Informationen“ (Seite 17) eingegangen.

Löschen von Informationen

Den Informationen wird als Meta-Informationen auch ein Ersteller zugewiesen, welcher veranlassen kann, dass die Information gelöscht wird. Außerdem kontrolliert jeder Knoten regelmäßig die ihm zugeteilten Informationen und überprüft:

- ob er noch immer der für einen Schlüssel - oder dessen Replik - zuständige Knoten ist.
- ob die Information einen „Valid Until“ Parameter besitzt, welcher abgelaufen ist.

Sofern einer der beiden Fälle eintritt, kann die Information entweder an den passenden Knoten weitergegeben oder gelöscht werden.

2.1.4 IP hashing mit SHA256 als unique identifier

Um eine ordentliche Verteilung der einzelnen *Peers* im Ring zu erlangen, werden die *identifier* der einzelnen *Peers* gehasht. SHA256 ist hierbei das vorgeschlagene [3] Verfahren und die IP-Adresse inklusive Port der vorgeschlagene *identifier*. Denkbar wären jedoch auch andere Verfahren und oder andere Hashlängen. Zu berücksichtigen wäre in dem Fall, dass dies auch Auswirkungen auf die Größe der *Fingertable* und die potentielle Menge an Informationen und Peers im Ring hat.

In den Abbildungen 2.3, 2.4 und 2.5 ist die Verteilung der Hashwerte von IP-Adressen dargestellt. Die IP-Adressen wurden dabei pseudo-zufällig gewählt. Konkret wurde die IP-Adresse und Port in textüblicher Form gehasht, sprich: z.B. „145.54.2.246:12345“. In den Abbilungen wird jeder Hashwert durch einen roten Punkt dargestellt. Die x-Achse repräsentiert dabei die oberen 128 bit des Hashwertes, die y-Achse die unteren 128 bit. Eine Gute Verteilung wird dabei leicht sichtbar. Die Diagramme wurden durch ein Python-Script erstellt und lassen sich leicht reproduzieren.[2]

[2] Das Python-Script ist in der Datei „misc/ip_sha256.py“ zu finden.

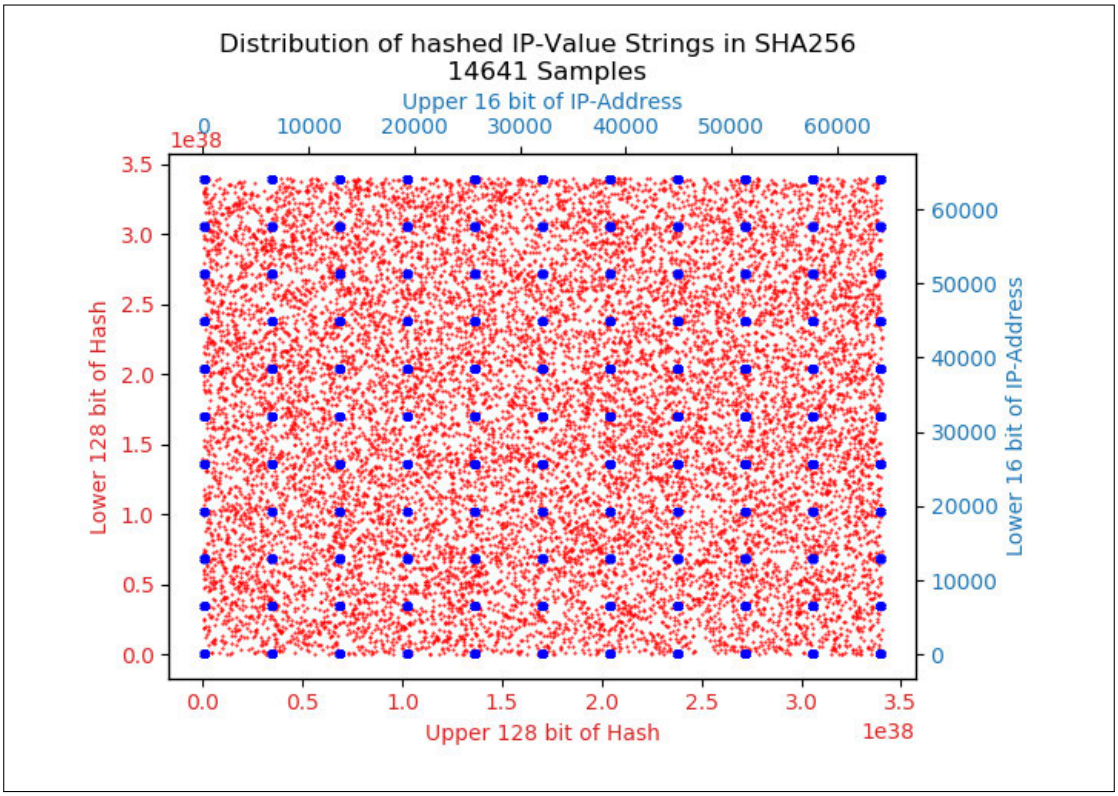

Abbildung 2.3: Verteilung der Hashwerte von 14.641 IP-Adressen

In Abbildung 2.3 wurden 14.641 IP-Adressen inklsuive Port gehasht. Zusätzlich zu den in rot dargestellten Hashwerten durch die untere x-Achse und die linke y-Achse (rote Achsen) sind auch noch die ursprünglichen IP-Adressen in blau durch die obere x-Achse und die rechte y-Achse dargestellt. Jeder der erkennbaren blauen Punkte repräsentiert dabei eine Ansammlung von IP-Adressen (121 relativ dicht beisammen liegende IP-Adressen je Ansammlung). Durch diese Aufteilung ist zu erkennen, dass die IP-Adressen in Textform durch SHA256 sehr gut verteilt werden, sogar wenn sie dicht beisammen liegen.

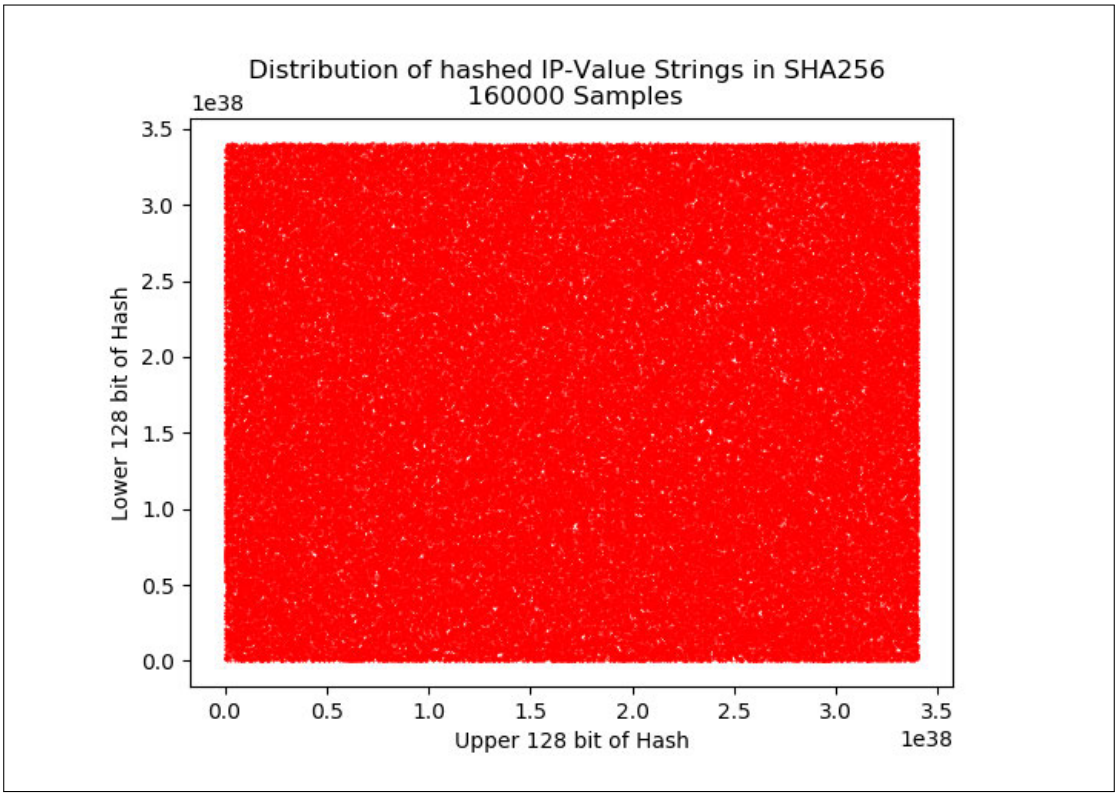

Abbildung 2.4: Verteilung der Hashwerte von 160.000 IP-Adressen

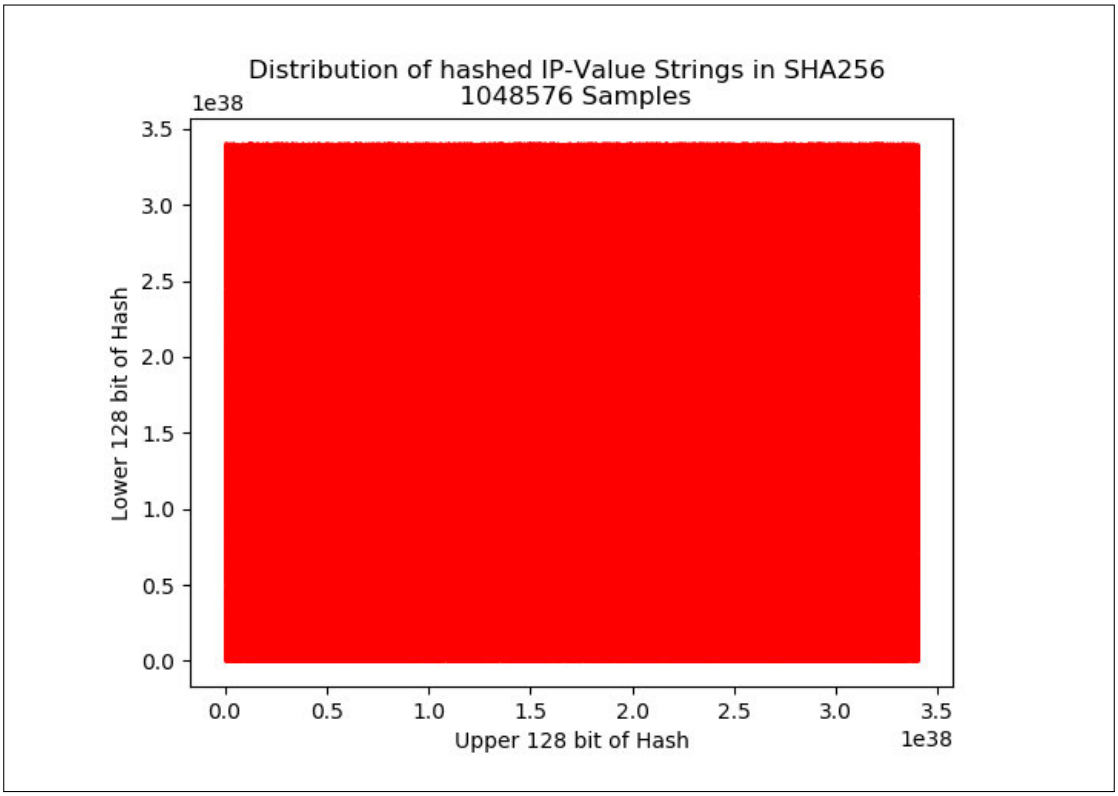

Abbildung 2.5: Verteilung der Hashwerte von 1.048.576 IP-Adressen

In den Abbildungen 2.4 und 2.5 wurde auf die Darstellung der IP-Adressen und Ports verzichtet, da sich das in Abbildung 2.3 gezeigte Raster der IP-Adressen und Ports lediglich verdichtet.

In Abbildung 2.4 wurde die Anzahl der gehashten IP-Adressen auf 160.000 erhöht. Es sind kaum noch Lücken zu erkennen, was wiederum eine Gute Verteilung belegt.

In Abbildung 2.5 wurde die Anzahl der gehashten IP-Adressen abermals auf über eine Million erhöht. Die einzelnen Punkte bilden nun eine durchgängige rote Fläche. Von diesem Punkt an lässt sich keine visuelle Veränderung mehr erkennen, wenn die Anzahl der Werte erhöht wird.

Es lässt sich also zusammenfassend sagen, dass sich alleine durch das Hashen der IP-Adressen inklusive Port in Textform durch SHA256 eine sehr gute Verteilung erreichen lässt. In „ChordComm“ wird dieses Verfahren exakt so angewendet.

2.2 WebRTC

WebRTC ist ein offener Standard, der eine Sammlung von Protokollen und Schnittstellen definiert und implementiert. Diese Protokolle und Schnittstellen ermöglichen die sogenannte „Peer-to-Peer“ Kommunikation. Gemeint ist damit, dass im Browserumfeld nicht mehr nur die Kommunikation von einem Client über den Server, sondern auch direkt zu einem anderen Client - in dem Fall sind beide beteiligte Clients Peers genannt - ermöglicht wird.

Die Firma „Global IP Solutions (GIPS)“ und somit auch WebRTC wurde 2010 von Google erworben und eine Refernz-Implementierung als freie Software veröffentlicht.[3]

2.2.1 WebRTC Kommunikation

Da die Kommunizierenden Peers in der Regel hinter einem NAT sitzen, muss ein Mechanismus genutzt werden, diesen NAT zu überwinden. Die von *WebRTC* genutzte Technik nennt sich Interactive Connectivity Establishment (ICE) und unterstützt sowohl Session Traversal Utilities for NAT (STUN) als auch Traversal Using Relays around NAT (TURN).

In Abbildung 2.6 sind die einzelnen Verbindungspartner, die bei WebRTC vorkommen, gezeigt. Die Peers fragen an den STUN-Servern ihre ICE-Kandidaten an. Anschließend tauschen sie diese Kandidaten über einen Signalling-Server aus. Nach diesem Austausch können die Peers direkt kommunizieren. Sollten jedoch bei einem der Peers keine ICE-Kandidaten über STUN ermittelt werden kann - sofern ein TURN-Server definiert ist - die Kommunikation über einen TURN-Server umgeleitet werden.

Aufgrund der Kosten für einen TURN-Server (Bandbreite und Rechenleistung) und der damit verbreiteten geringen Nutzung wird folgend nur noch auf STUN-Server eingegangen.

[3] Übernahmemeldung von GIPS auf golem.de

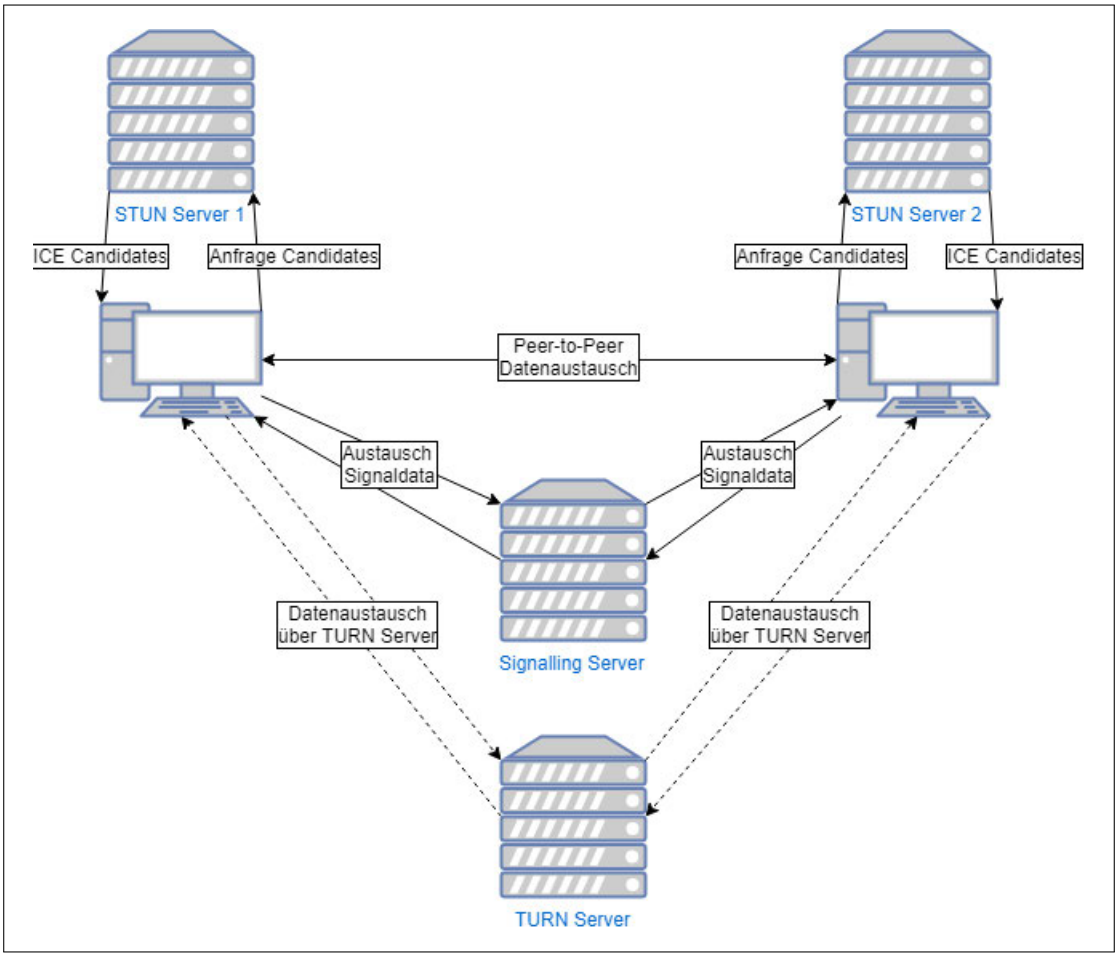

Abbildung 2.6: Ablauf von WebRTC

In Abbildung 2.7 ist der Aufbau einer WebRTC Kommunikation mit einem STUN-Server für zwei Partner gezeigt.

Peer_1 fragt an einem STUN-Server seine ICE-Kandidaten an. Der STUN-Server liefert *Peer_1* einige (oder auch keine, in diesem Beispiel aber mindestens einen) ICE-Kandidaten zurück. *Peer_1* schickt diese ICE-Kandidaten nun über den Signalling-Server an *Peer_2*. Nachdem *Peer_2* die ICE-Kandidaten von *Peer_1* erhalten hat, fragt dieser auch an einem STUN-Server seine ICE-Kandidaten an. Nach Erhalt der ICE-Kandidaten von *Peer_2* werden diese nun über den Signalling-Server an *Peer_1* zurück geschickt. Nach erfolgreichem Austausch der ICE-Kandidaten gilt eine Verbindung als etabliert.

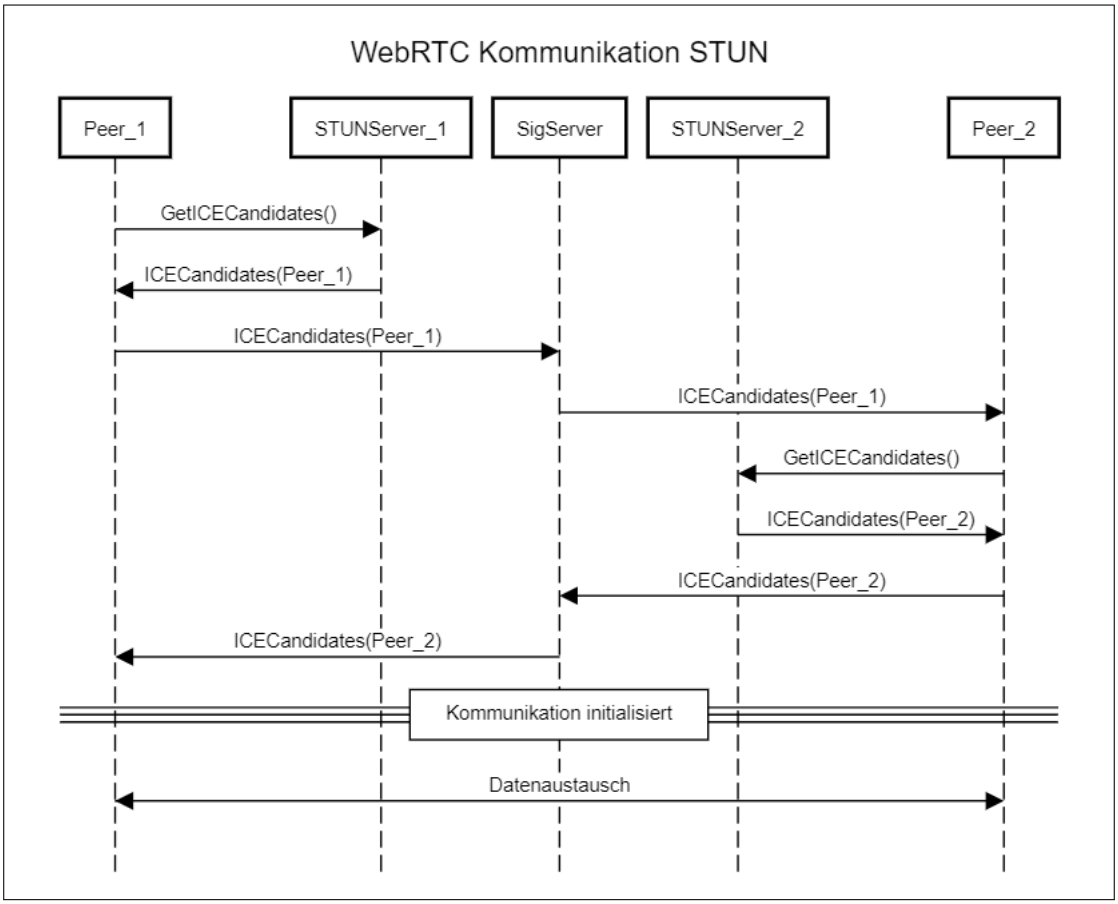

Abbildung 2.7: Ablauf eines Verbindungsaufbaus von WebRTC (STUN)

Nachdem eine Verbindung etabliert wurde, können sogenannte „Data Channel“ erstellt werden, um die eigentlichen Daten auszutauschen. Diese „Data Channel“ nutzen Real-Time Transport Protocol (RTP) in Kombination mit Datagram Transport Layer Securiy (DTLS), um Daten auszutauschen. Neben den „Data Channel“ werden noch andere Typen von Kanälen unterstützt, um verschiedenste Codecs abzubilden. In vielen Fällen wird dann Secure Real-Time Transport Protocol (SRTP) genutzt. In „ChordCom“ werden lediglich Roh-Daten verschickt und somit nur „Data Channel“ genutzt.

2.3 Replizierung der Informationen

Um im fragilen Umfeld der Browserkommunikation die Verfügbarkeit der Informationen gewährleisten zu können, ist eine Replizierungs-Strategie nötig. Die für „ChordComm“ in Frage kommenden Strategien wurden von Matthew Leslie, Jim Davies und Todd Huffman vorgestellt und verglichen [6].

Zur Auswahl standen: „DHash Replication“ [6, S. 37], „Dynamic Replication“ [6, S. 37], „Successor Allocation“ [6, S. 39], „Finger Allocation“ [6, S. 39], „Block Allocation“ [6, S. 39] und „Predecessor Allocation“ [6, S. 39]. Außerdem noch das von Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee und Pete Keleher vorgestellte Stabilisierungsprotokol „Lightweight, Adaptive, system-neutral Replication protocol (LAR)“ [10].

Da durch *Chord* eine starke Ringförmige Struktur vorliegt und jeder *Peer* seinen direkten *predecessor* und *successor* kennt, ist die engere Wahl auf „DHash Replication“, „Successor Allocation“, „Predecessor Allocation“ und „Finger Allocation“ gefallen. Alle anderen zur Wahl stehenden Möglichkeiten wären auch anwendbar, sind von der Komplexität und Menge der Kontrollroutinen entweder für diese Arbeit zu aufwendig oder weniger gut für Browserkommunikation geeignet.

Für „ChordComm“ ist die Wahl auf „Successor Allocation“ gefallen. Bei diesem Verfahren gibt der für eine Information zuständige Knoten diese auch an seinen *successor* weiter, inklusive eines Zählers für die Replikationstiefe. Jeder Knoten, der solch eine Information inklusive dieses Zählers erhält, speichert diese ab und schickt sie dann mit einem um eins dekrementierten Zähler an seinen *successor* weiter.

Vorteil dieses Verfahrens ist es, dass bei Ausfall eines für eine Information zuständiger Knoten logisch gemäß *Chord* sein *successor* nun für diese Information zuständig ist. Durch die „Successor Allocation“ besitzt dieser die Information bereits und es muss keine spezielle Suche nach eine Replik der Information gestartet werden - die *Chord* Logik bleibt Intakt und unberührt von der Replizierungs-Logik.

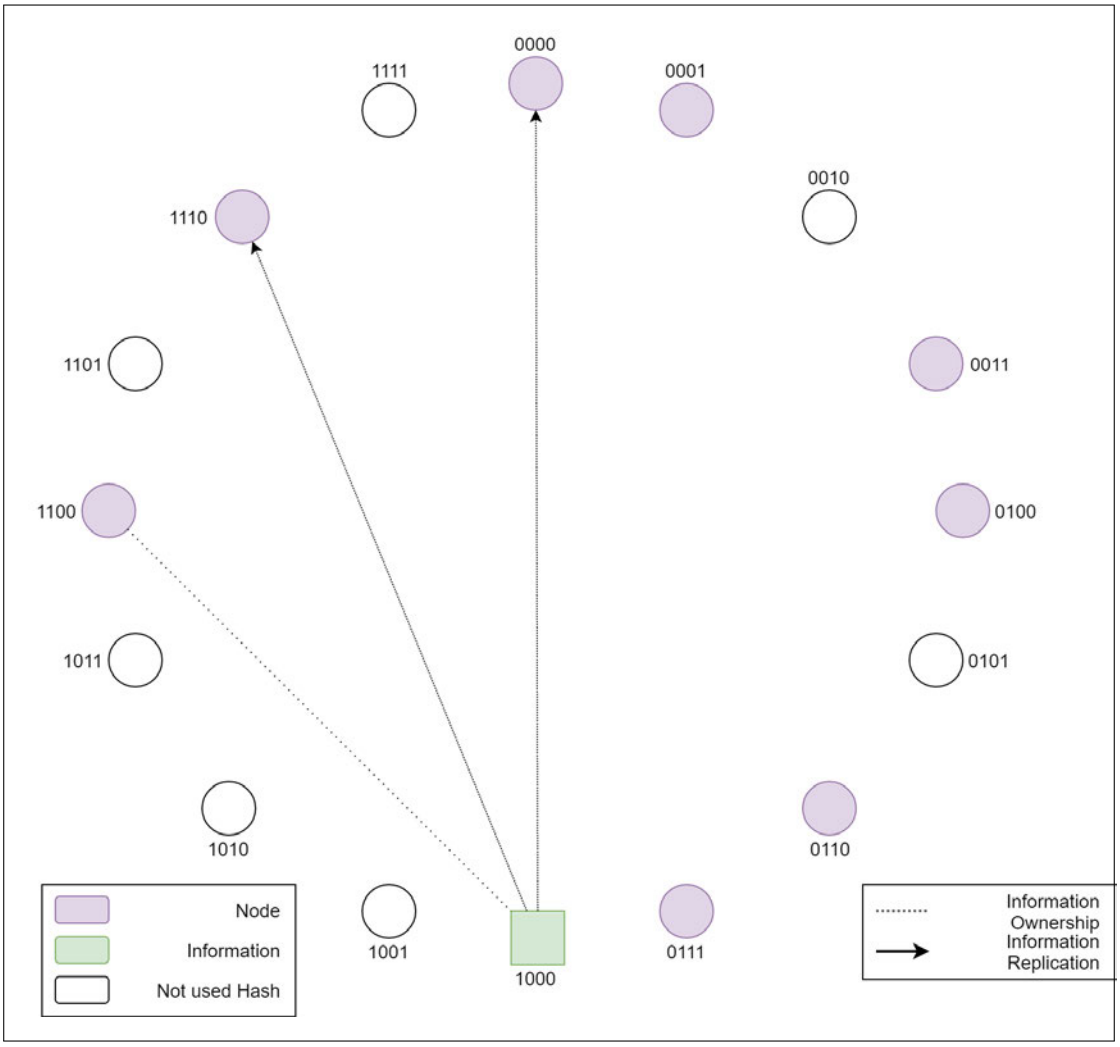

Abbildung 2.8: Replizierung einer Information im Ring mit Faktor 3

In Abbildung 2.8 ist die Information *1000* zu sehen, welche von der Node *1100* verwaltet wird. Zusätzlich wurde die Information von der Node *1100* noch an seinen *successor*, Node *1110*, weitergegeben. Die Node *1110* hat die Information nun auch noch an seinen *successor*, Node *0000*, weitergegeben. Somit hat die Information einen Verfügbarkeits-Faktor von 3. Sollte die Node *1100* nun ausfallen, wäre der neue logische Verwalter der Information die Node *1110*, welche dank der Replizierung bereits die Information besitzt.

2.3.1 Informationsverteilung bei Veränderung

Die Informationen, welche lokal auf einem Knoten gespeichert werden, sind zum Zeitpunkt des Speicherns logisch richtig verteilt. Sollte sich im Nachhinein der Ring verändern, muss geprüft werden, ob die Informationen immer noch logisch richtig verteilt sind.

In Abbildung 2.9 ist der Ablauf einer solchen Prüfung dargestellt. Die dargestellt Prüfung wird in festgelegten Intervallen für jede lokale Information durchgeführt. Sofern die gerade zu prüfende Information eine Replik ist - also der Replizierungsindex ungleich 0 ist - wird die Prüfung vorzeitig abgerochen. Sollte die Information jedoch keine Replik sein, wird geprüft, ob unser *predecessor* passender für die Information ist. Ist dies der Fall, wird die Informaion an den *predecessor* gesendet, der lokale Replizierungsindex um 1 erhöht und mit einer weiteren Verringerung um 1 an unseren *successor* gesendet. Sollte der *predecessor* nicht passend sein, wird geprüft, ob ein neuer *successor* vorliegt (der die Information noch nicht hat). Sollte einer vorliegen, wird die Information mit einem um 1 erhöhten Replizierungsindex an ihn gesendet werden.

Durch dieses Vorgehen werden Informationen logisch auch auf später hinzukommende Knoten richtig verteilt. Für den Fall, dass mehrere Knoten zeitgleich hinzukommen und aus Sicht der bereits vorhanden Information alle besser wären, kann es mehrere Zyklen dauern, bis die Information wieder richtig logisch verteilt wurde.

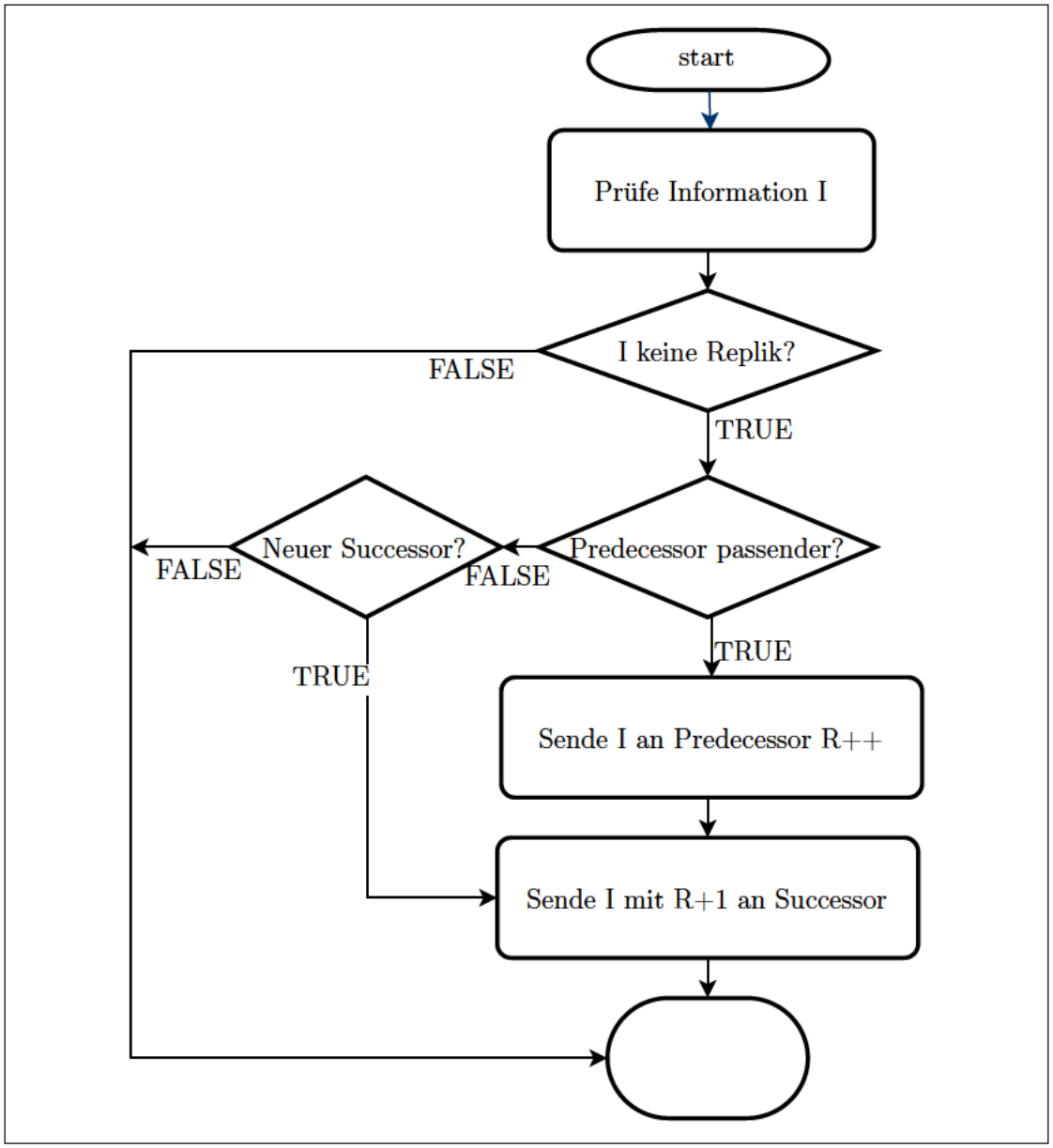

Abbildung 2.9: Prüfen der Informationsverteilung

2.3.2 Replizierung bei Ausfall

Neben der Verteilung bei Veränderungen im Ring müssen auch Vorkehrungen getroffen werden, um auf Ausfälle von Knoten vorbereitet zu sein. Aus Sicht der Replizierung

sind nur zwei Fälle relevant: Der Wegfall unseres *successor* und der Wegfall unseres *predecessor*.

Sollte der *successor* wegfallen, wird die lokale Information mit einem um 1 erhöhten Replizierungsindex an den neuen *successor* gesendet. Dieser Vorgang ist in Abbildung 2.10 dargestellt.

Sollte der *predecessor* wegfallen, wird der lokale Replizierungsindex um 1 verringert - sofern er nicht 0 ist - und die Information mit einem um 1 erhöhten Replizierungsindex an den *successor* gesendet. Dieser Vorgang ist in Abbildung 2.11 dargestellt.

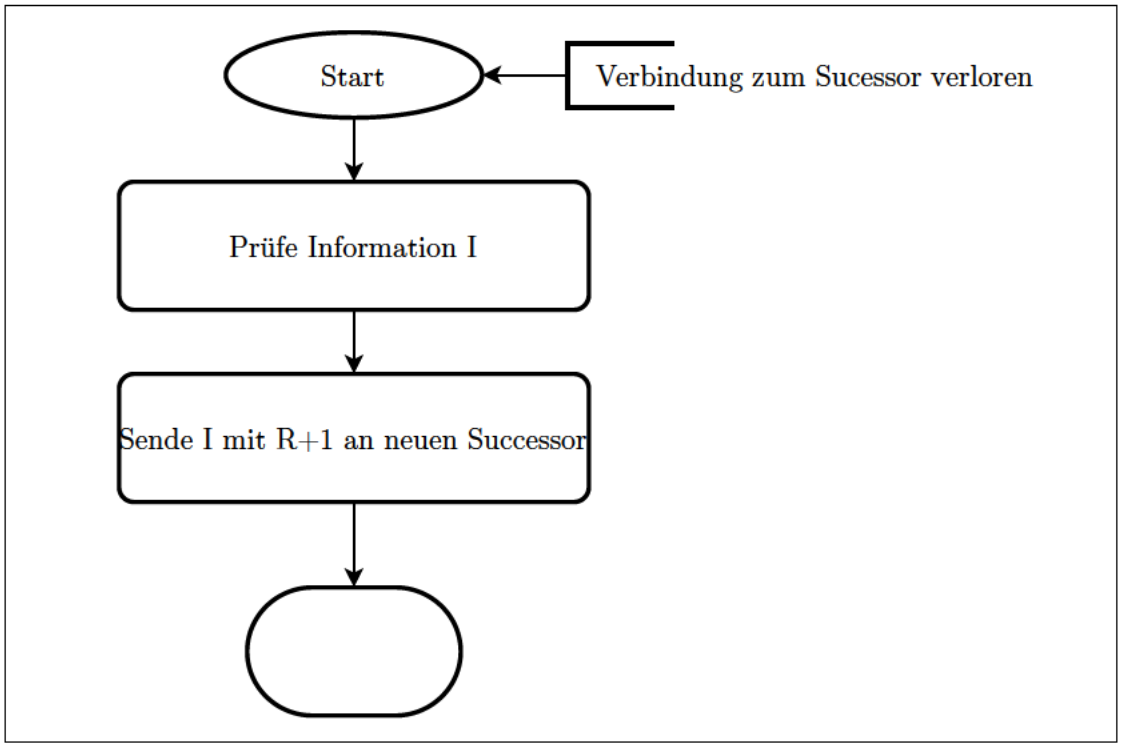

Abbildung 2.10: Replizierung bei Wegfall von Successor

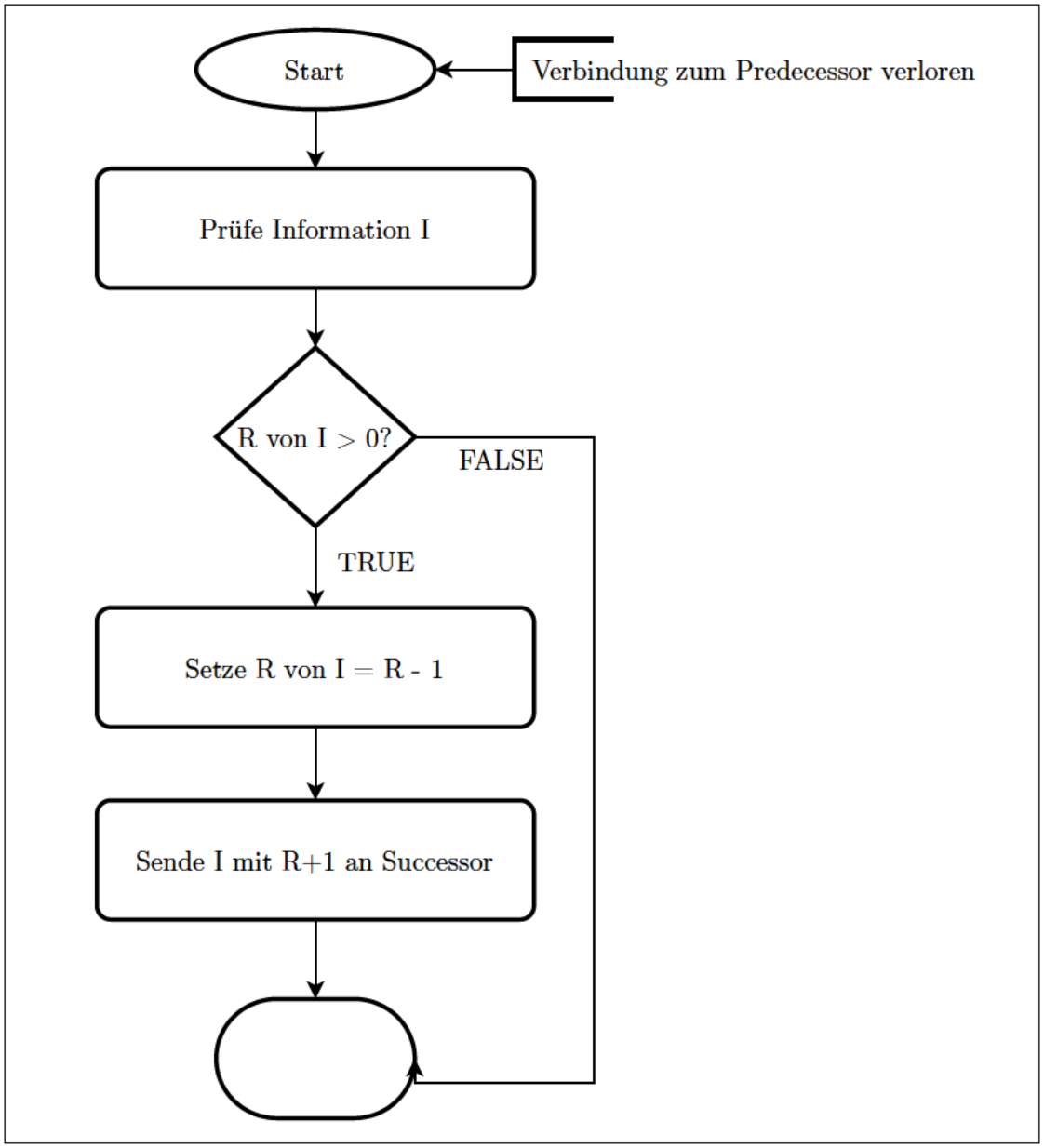

Abbildung 2.11: Replizierung bei Wegfall von Predecessor

3 Ähnliche Arbeiten

Neben dem hier entstehenden „ChordComm“ existieren bereits einige weitere Lösungen, die auch auf *Chord* über *WebRTC* basieren. Im folgenden werden einige dieser Lösungen genannt und die Abgrenzungen zu „ChordComm“ erläuert.

3.1 #1 „webrtc-chord“

„webrtc-chord“[1] Implementiert die „Chord“-Logik im Verbund mit „WebRTC“. Dabei wird für die Koordination der „WebRTC“-Verbindungen das externe Modul „PeerJS“[2] genutzt.

Auf die Verteilungen von Informationen - inkl. der Möglichkeit Informationen zu ändern, löschen und abzurufen - wird nicht explizit eingegangen. Außerdem wird das Modul „webrtc-connectionpool“[3] genutzt, um nur eine gewisse Menge an Verbindungen gleichzeitig aufrecht zu erhalten (standardmäßig 10).

Die Implementierung erfolgte in „JavaScript“ und die letzte Aktualisierung ist ca. 6 Jahre her[4].

[1] https://github.com/tsujio/webrtc-chord von User „tsujio“ (Stand 02.02.2020)
[2] https://github.com/peers/peerjs (Stand 02.02.2020)
[3] https://github.com/tsujio/webrtc-connectionpool von User „tsujio“ (Stand 02.02.2020)
[4] Stand 02.02.2020

3.2 #2 „webrtc-chord“

„webrtc-chord“[5] Implementiert die „Chord“-Logik im Verbung mit „WebRTC“.

Auf die Verteilungen von Informationen - inkl. der Möglichkeit Informationen zu ändern, löschen und abzurufen - wird nicht explizit eingegangen. Größter Unterschied zu „Chord-Comm“ ist, dass das Wegfallen von Nodes im Ring nicht unterstützt und als „TODO“ aufgelistet ist.

Die Implementierung erfolgte in „JavaScript“ und die letzte Aktualisierung des Codes ist ca. 5 Jahre her[6].

[5] https://github.com/daviddias/webrtc-chord von User „daviddias“ (Stand 02.02.2020)
[6] Stand 02.02.2020

4 Architektur und Design

Im folgenden Kapitel wird auf das Systemdesign eingegangen. Neben einer Übersicht des Gesamtsystems wird noch auf die einzelnen Module eingegangen, sowie den Aufbau der Nachrichten und Informationen. Abschließend werden für die Kerninteraktionen der verschiedenen Kommunikationspartner - sowohl Peers als auch Signalling-Server - die Abläufe dargestellt.

4.1 Gesamtsystem

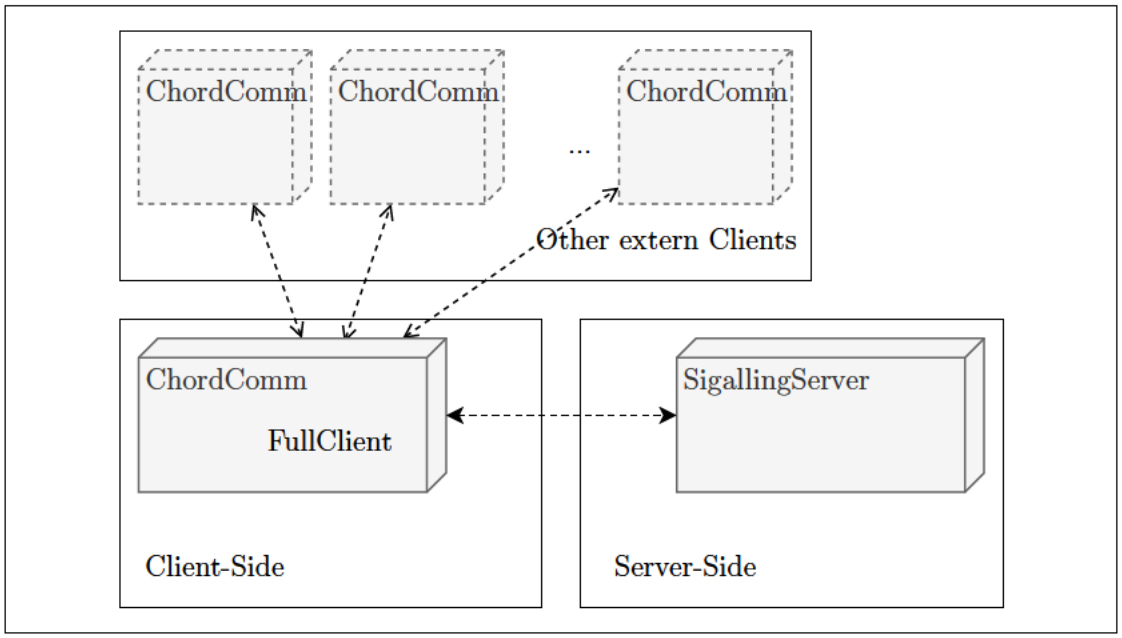

Abbildung 4.1: ChordComm Gesamtsystem

Das Gesamtsystem von „ChordComm“ unterteilt sich in 2 Pakete:

- „ChordComm“-Paket, welches auf der Client-Seite genutzt wird und die gesamte *Chord*- und *WebRTC*-Logik enthält.
- „SignallingServer“-Paket, welches für den Austausch der Verbindungsinformationen der Clients benötigt wird.

In Abbildung 4.1 sind diese Pakete dargestellt. Der Signalling Server wird nur für den Austausch der Verbindungsinformationen benötigt. Die verschiedenen „ChordComm“ Clients kommunizieren direkt über WebRTC miteinander.

4.2 Komponenten

Das Paket „ChordComm“ setzt sich aus 5 Komponenten zusammen. In Abbildung 4.2 sind die Komponenten und ihre Schnittstellen dargestellt. Der „FullClient“ stellt dabei die einzige von „ChordComm“ nach außen sichtbare Schnittstelle zur Verfügung. Diese Schnittstelle wird vom Endnutzer verwendet.

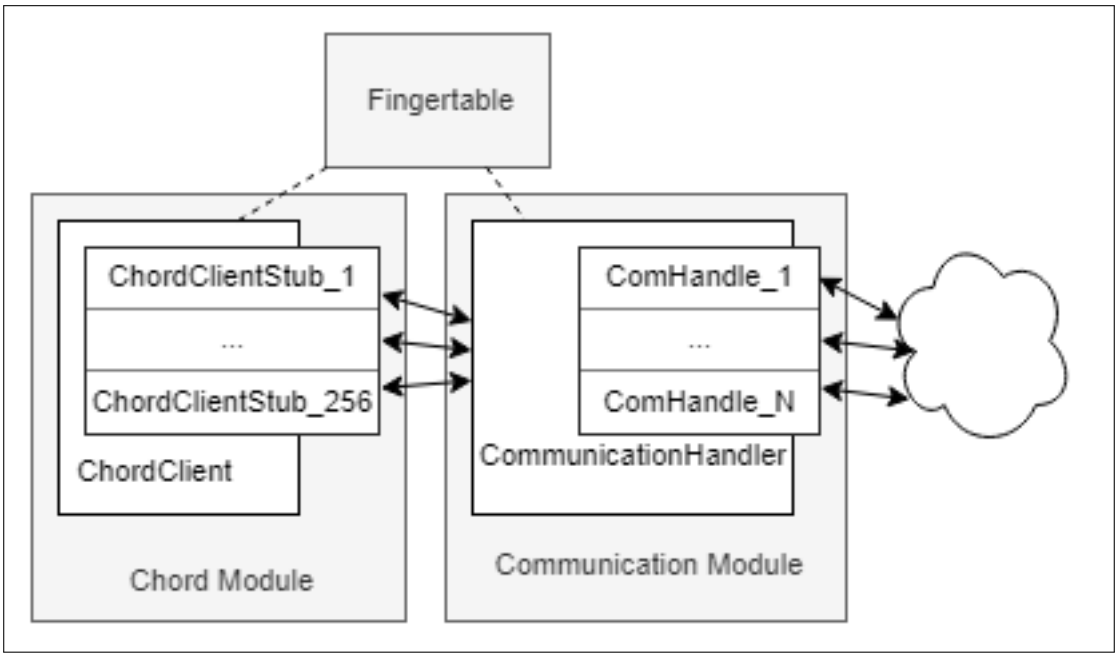

Abbildung 4.3: Überblick der Komponenten.

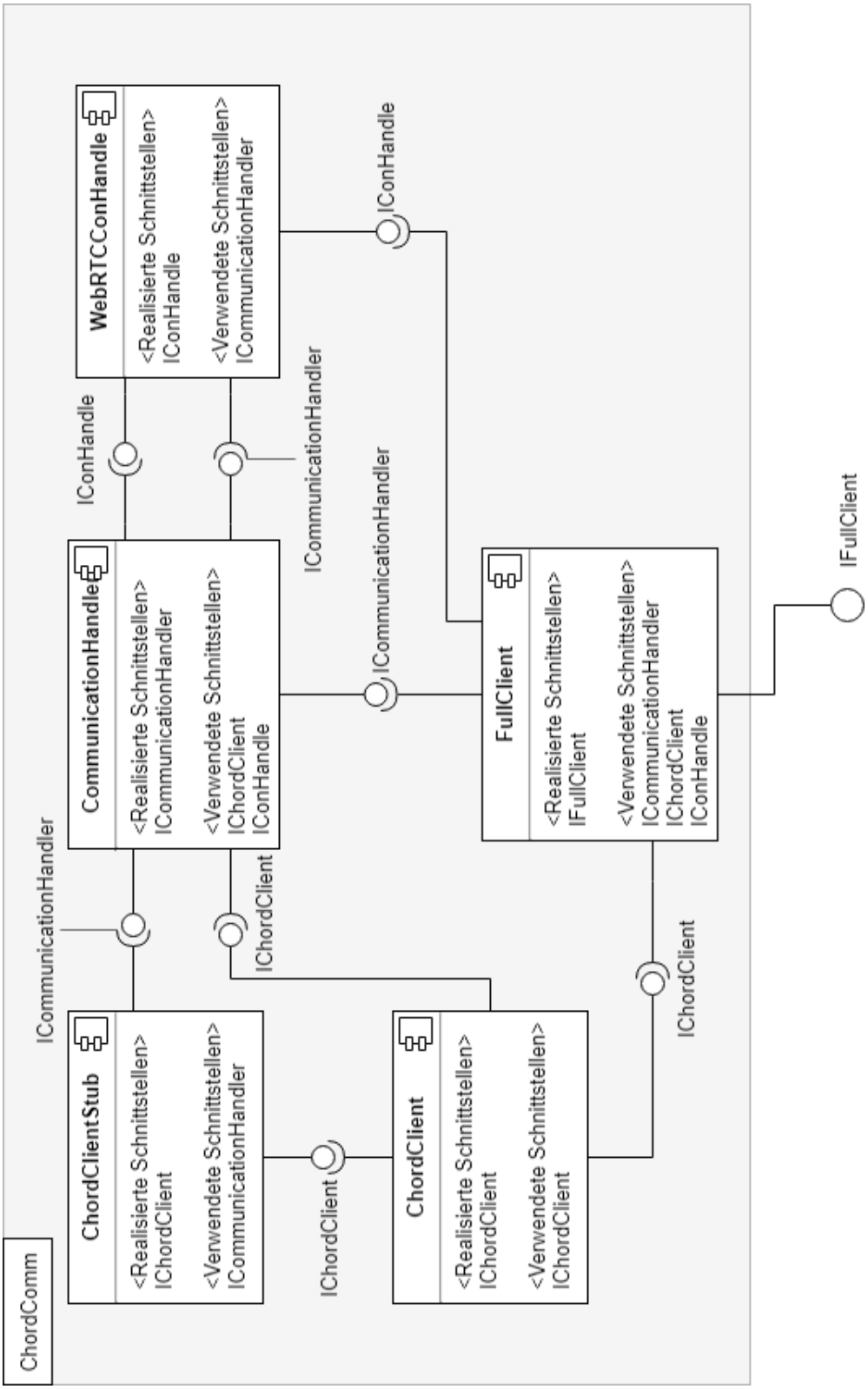

Abbildung 4.2: Komponenten von ChordComm

In Abbildung 4.3 sind die einzelnen Komponenten und ihre Zugehörigkeit zu erkennen. *Chord Module* und *Communication Module* bilden dabei die zwei Komponenten-Gruppen. *Chord Module* besteht dabei aus den einzelnen Komponenten *ChordClient* und *ChordClientStub*. *Communication Module* besteht aus den Komponenten *CommunicationHandler* und *ComHandle*.

Das *Chord Module* ist dabei für die gesamte *Chord*-Logik zuständig und das *Communication Module* für die Abwicklung des Nachrichtenaustausches zwischen den *Peers*.

In den folgenden Sektionen werden die einzelnen Komponenten inklusive ihrer Aufgaben und Schnittstellen genauer beschrieben.

4.2.1 ChordClient

Das Modul ChordClient ist für die gesamte *Chord* Logik zuständig. Hierfür wird ein Interface IChordClient definiert, welches die Methoden aus der *Chord* Definition übernimmt [3]. Dieses Interface wird nun von den Klassen ChordClient und ChordClientStub implementiert.

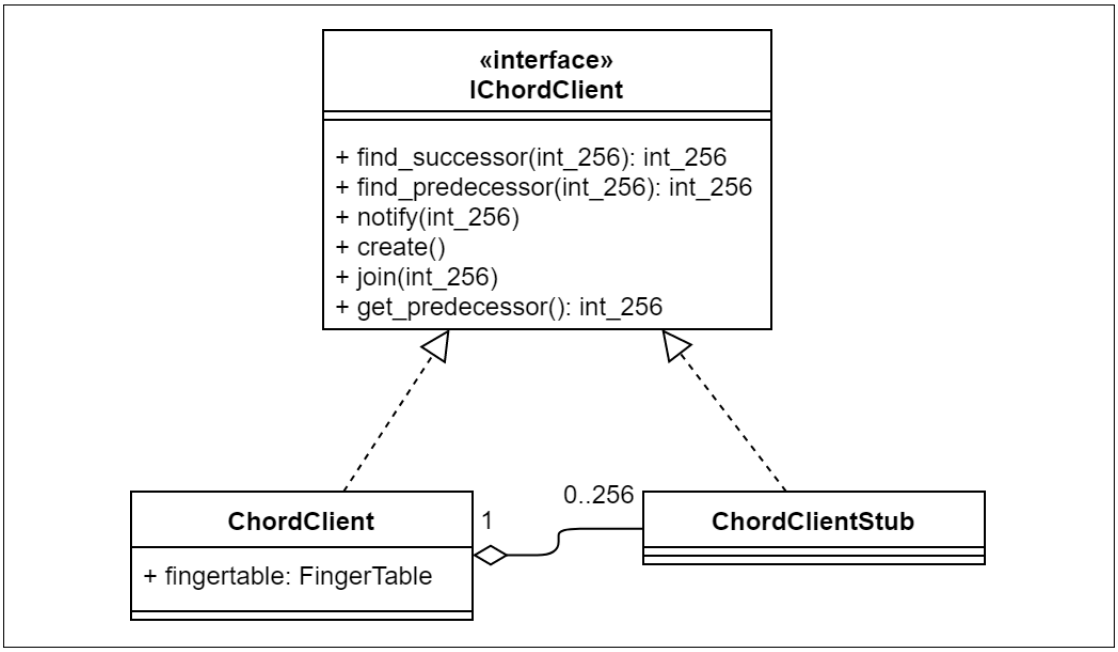

Abbildung 4.4: Klassendiagramm des Moduls ChordClient

ChordClient

ChordClient implementiert die durch das Interface IChordClient definierten Methoden funktionsgemäß des Pseudo-Codes aus [3, S. 5+6]. Da durch die Verteilung des Systems keine direkten Referenzen auf die anderen ChordClient Objekte anderer Nodes möglich sind, um deren Funktionen aufzurufen, existiert eine Klasse ChordClientStub. Diese implementiert auch das Interface IChordClient und wird vom lokalen ChordClient wie direkte Referenzen auf andere ChordClient behandelt.

ChordClientStub

Bei ChordClientStub handelt es sich um einen proxy/stub zur Bindung an einen entfernten, nicht lokalen ChordClient. Da das gesamte ChordClient Modul von der eigentlichen Kommunikation mit anderen Nodes entkoppelt ist, dienen die ChordClientStub als Message-Factory. Es werden also basierend auf den Aufrufen passende Nachrichten zusammengesetzt, welche an das eigentliche Kommunikationsmodul CommunicationHandler weitergereicht werden.

4.2.2 CommunicationHandler

Das Modul CommunicationHandler ist für die Verwaltung aller Verbindungen und Auswertung alles Nachrichten zuständig. Der CommunicationHandler implementiert das Interface ICommunication.

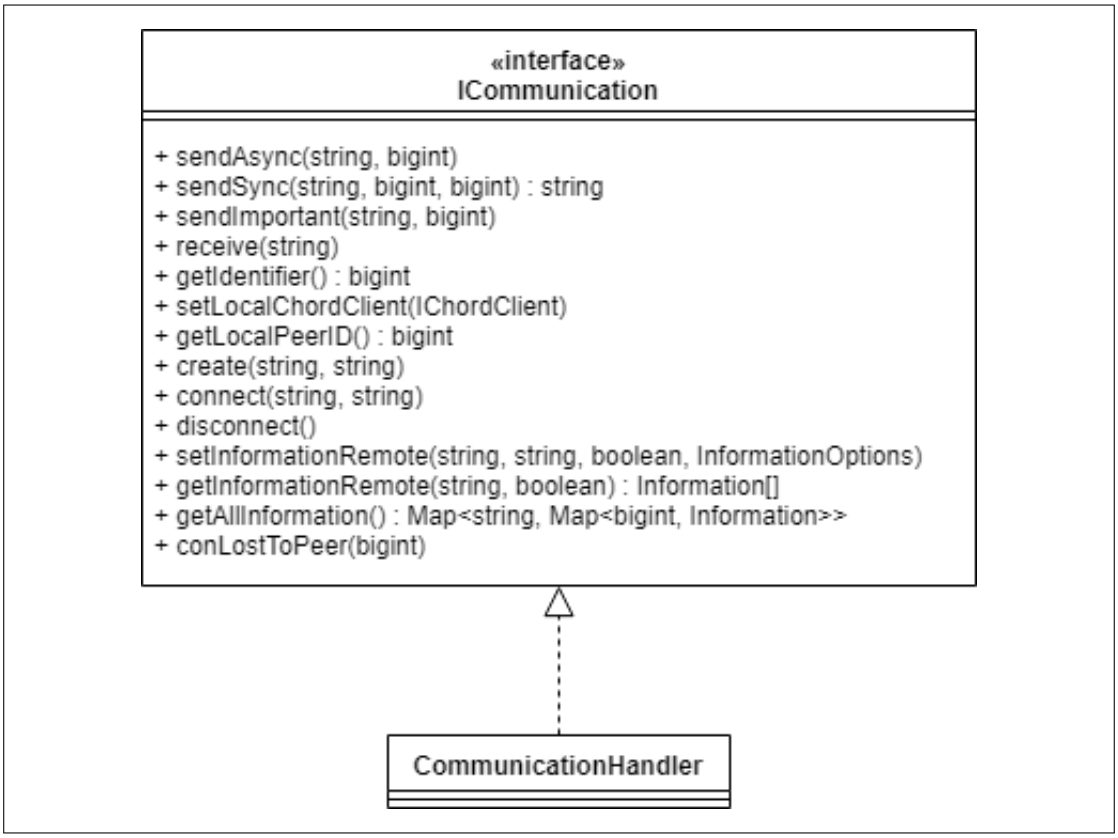

Abbildung 4.5: Klassendiagramm des Moduls CommunicationHandler

Die konkreten Aufgaben des *CommunicationHandlers* umfassen dabei:

1. Verwaltung der Verbindung zum *Signalling* Server.
2. Zuordnung/Verarbeitung der erhaltenen Nachrichten
   a) Falsch empfangene Nachrichten verarbeiten.
   b) Weiterleiten an anderen *Peer* über *ConnectionHandle* (Relay-Funktion).
   c) Richtig empfangene Nachricht auswerten z.B. Ergebnisse einer Anfrage speichern.
3. Verwalten der einzelnen Verbindungen
   a) Neue Verbindungsobjekte erstellen.

b) Verbindungsabbrüche behandeln.

c) Nicht mehr benötigte Verbindungen trennen.

4.2.3 WebRTCConHandle

Das Modul WebRTCConHandle ist für die Verwaltung einer WebRTC-Verbindung zu einem anderen Peer zuständig. Der WebRTCConHandle implementiert das Interface IConnectionHandle.

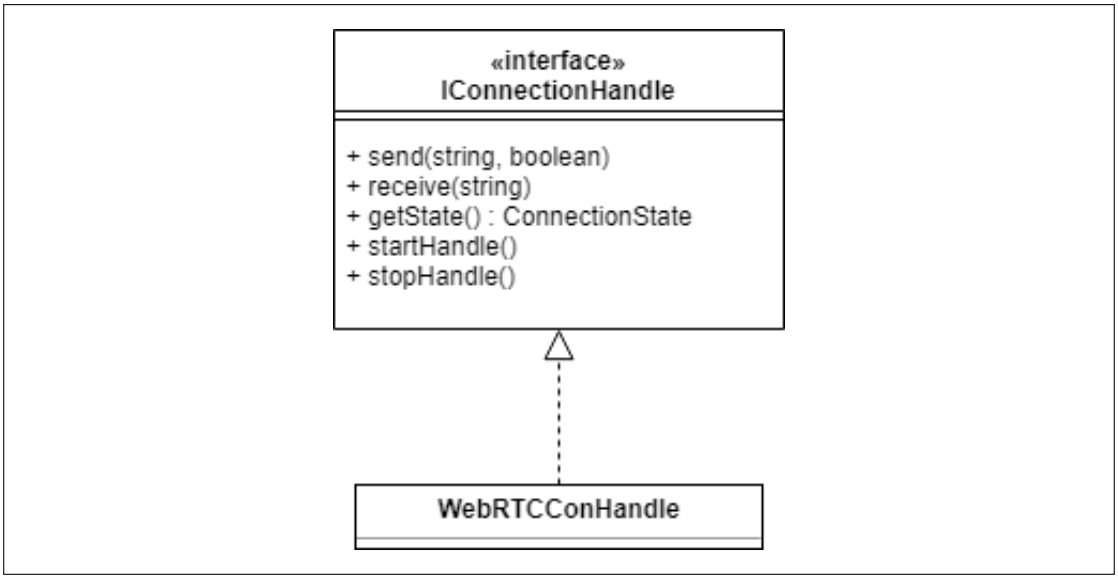

Abbildung 4.6: Klassendiagramm des Moduls WebRTCConHandle

Das Interface präsentiert nach außen nur die stark abstrahierte Funktionsweise eines *ConnectionHandles*. Die gesamte Logik für den Aufbau der Verbindung sowie die Logik um Fehler abzufangen, wird von dem Modul selbst verwaltet. Sofern Nachrichten über „send“ vor Verbindungsaufbau an das Modul übergeben werden, werden diese gebuffert.

4.3 Nachrichten

Im folgenden Abschnitt wird auf die Struktur der Nachrichten eingegangen, die die Nodes austauschen. Neben einer allgemeinen Struktur wird auch auf jeden einzelnen Nachrichtentyp und dessen Felder eingegangen.

Nachrichten sind die primäre Informationseinheit, mit der verschiedene Nodes Kommunizieren. Um Informationen der Anfrage zuordnen zu können, ist eine Verschachtelung notwendig.

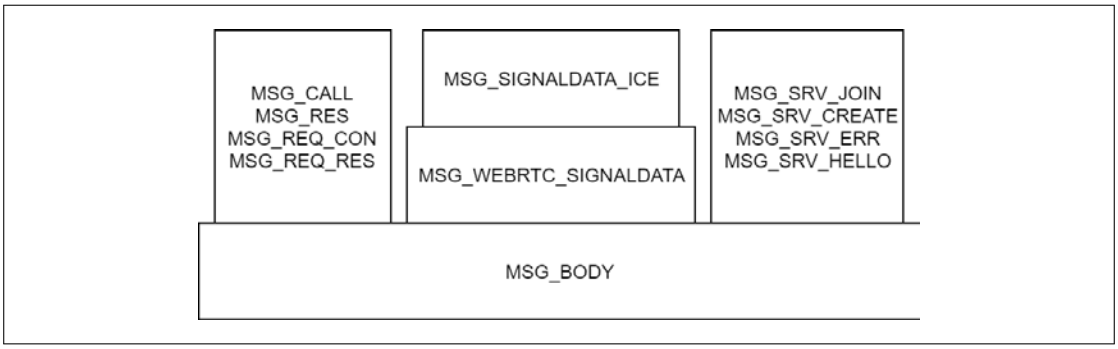

Abbildung 4.7: Verschachtelung der Nachrichten

In Abbildung 4.7 ist diese Verschachtelung zu sehen. *MSG_ BODY* dient jeder Nachricht als Basis - bildet also sozusagen den Umschlag. In diesem Umschlag kann nun eine konkrete Nachricht, wie z.B. ein Funktionsaufruf *MSG_ CALL* stehen. Nachrichten, wie z.B. *MSG_ WEBRTC_ SIGNALDATA* können außerdem noch weiter verschachtelt sein.

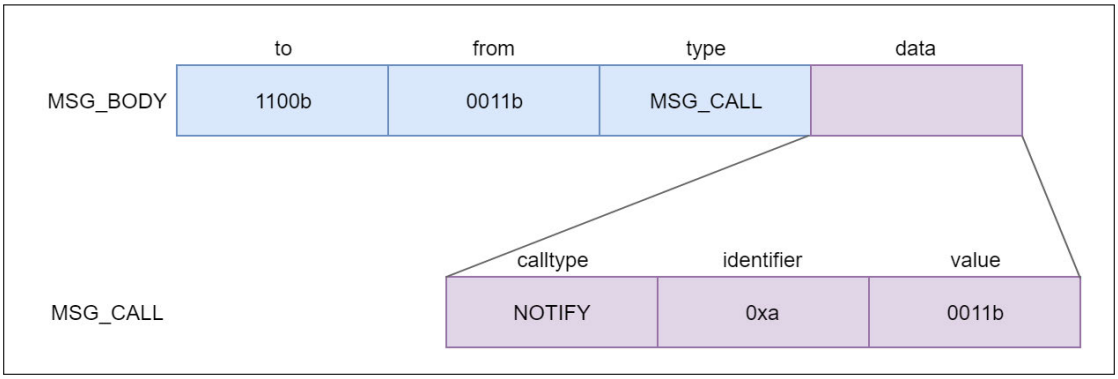

Abbildung 4.8: Beispiel einer ChordComm Nachricht

In Abbildung 4.8 ist beispielhaft eine komplette Nachricht zu sehen. Erkennbar ist hierbei der *MSG_BODY* und als payload ein *MSG_CALL*. Die Abbildung berücksichtigt nicht eventuell vorhandene Umwandlung der Datentypen - wie z.B. die Umwandlung des *MSG_CALL* Objektes in einen string - und dient nur der Veranschaulichung der allgemeinen Verschachtelung.

Durch die gegebenen Standards von WebRTC muss der Austausch von Nachrichten nur auf syntaktischer Ebene beschrieben werden. Um über alle Operationen, inkl. Austausch von Informationen im Ring, einen Standard zu pflegen und die Erweiterbarkeit der Nachrichten auf den verschiedenen verschachtelten Ebenen zu ermöglichen, wird durchgehend JSON verwendet.

4.3.1 MSG_BODY

Der *MSG_BODY* bildet den Umschlag der Nachricht und beinhaltet grundlegende Informationen bezüglich Absender, Empfänger und Typ. Definiert ist *MSG_BODY* durch das Interface *MSG_BODY*.

Alle Nachrichten müssen den *MSG_BODY* nutzen und alle Nachrichten-verarbeitenden Prozesse erwarten auf der untersten Ebene einen *MSG_BODY*.

In Tabelle 4.1 sind die einzelnen Felder von *MSG_BODY*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
to	string	ID des Empfängers
from	string	ID des Senders
type	MSG_TYPES	Typ der Nachricht
data	string	Daten der Nachricht *(Stringified)* z.B. ein MSG_CALL Objekt

Tabelle 4.1: Aufbau des *MSG_BODY* einer Nachricht

4.3.2 MSG_TYPES

Das Enum *MSG_TYPES* definiert alle Nachrichtentypen, welche in einem *MSG_BODY* transportiert werden können.

Um bessere Übersicht zu geben, werden die Nachrichtentypen hier in zwei Tabellen aufgeteilt.

Typ	Bedeutung
CALL	Nachricht enthält einen Funktionsaufruf (*MSG_CALL*).
RESULT	Nachricht enthält das Ergebnis eines Funktionsaufrufes (*MSG_RES*).
REQUESTCONNECTION	Nachricht enthält die Anfrage einer neuen Verbindung (*MSG_REQUESTCONNECTION*)
REQUESTRESPONSE	Nachricht enthält die Antwort auf die Anfrage einer Verbindung (*MSG_REQUESTRESPONSE*)
SIGNALDATA	Nachricht enthält Signalling Informationen (*MSG_WEBRTC_SIGNALDATA*)
SETINFORMATION	Nachricht enthält eine Information, die in den Chord-Ring eingebracht werden soll.
GETINFORMATION	Nachricht enthält eine Anfrage für eine Information.

Tabelle 4.2: Nachrichten Typen *MSG_TYPES* Teil 1.

In Tabelle 4.2 sind alle Nachrichtentypen aufgelistet, die Nodes in einem Chord-Ring untereinander verwenden.

Typ	Zweck
SERVERHELLO	Nachricht enthält eine Begrüßung vom Signalling Server (*MSG_SRV_HELLO*).
SERVERJOIN	Nachricht enthält eine Anfrage einem Chord-Ring beizutreten (*MSG_SRV_JOIN*).
SERVERCREATE	Nachricht enthält eine Anfrage einen neuen Chord-Ring zu erstellen (*MSG_SRV_CREATE*).
SERVERERROR	Nachricht enthält eine Fehlermeldung des Servers (*MSG_SRV_ERR*).

Tabelle 4.3: Nachrichten Typen *MSG_TYPES* Teil 2.

MSG_CALL

MSG_ CALL repräsentiert die Anfrage zum Ausführen einer Methode - also die Informationen für einen Remote Procedure Call (RPC) - und ist durch das Interface *MSG_ CALL* definiert. Für die definition der auszuführenden Methode wird ein Feld vom Typ *MSG_ CALL_ TYPES* genutzt.

In der Tabelle 4.4 sind die einzelnen Felder von *MSG_ CALL*, deren Typen und Bedeutung zu finden.

Key	Typ	Bedeutung
calltype	MSG_CALL_TYPES	Typ der auszuführenden Methode
identifier	string	Wert für die Zuweisung des Ergebnisses zum passenden Aufruf. Wird von jedem *Peer* selber festgelegt. Weitere Informationen hierzu in 4.2.2 „CommunicationHandler“ (Seite 29).
value	string	Funktionsparameter für den Aufruf.

Tabelle 4.4: Aufbau des *MSG_ CALL* Objektes in einer Nachricht

MSG_CALL_TYPES

Das Enum *MSG_ CALL_ TYPES* definiert alle aufrufbaren Methoden. Wird in *MSG_ CALL* verwendet.

In Tabelle 4.5 sind alle möglichen Typen und ihre Bedeutung aufgelistet.

Typ	Bedeutung
GET_PREDECESSOR	ID für den Aufruf der Funktion „get_predecessor“.
NOTIFY	ID für den Aufruf der Funktion „notify“.
FIND_SUCCESSOR	ID für den Aufruf der Funktion „find_successor“.
FIND_PREDECESSOR	ID für den Aufruf der Funktion „find_predecessor“.

Tabelle 4.5: Typen der verschiedenen Aufrufe durch *MSG_ CALL*.

MSG_RESULT

MSG_RESULT enthält das Ergebnis einer durch RPC aufgerufenen Methode. Das Interface *MSG_RESULT* definiert hierbei die einzelnen Felder. Eine Nachricht von Typ *MSG_RESULT* wird nur auf eine vorher empfangene Nachricht vom Typ *MSG_CALL* erstellt.

In der Tabelle 4.6 sind die einzelnen Felder von *MSG_RESULT*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
identifier	string	Wert für die Zuweisung des Ergebnisses zum passenden Aufruf. Wird von jedem *Peer* selber festgelegt. Weitere Informationen hierzu in 4.2.2 „CommunicationHandler“ (Seite 29).
value	string	Ergebnis des Funktionsaufrufes.

Tabelle 4.6: Aufbau des *MSG_RESULT* Objektes in einer Nachricht

MSG_REQUESTCONNECTION

MSG_REQUESTCONNECTION ist ein Nachrichtentyp. Diese Nachricht beinhaltet neben den Informationen, wer sich mit dem Empfänger verbinden will, auch etwaige Daten, die für eine neue Verbindung relevant sind. Bei einer neuen Verbindung mittels *WebRTC* können diese Daten z.B. *ICE-Candidates* sein.

Implementiert ist dieser Nachrichtentyp durch das Interface *MSG_REQUEST_CONNECTION* in der Datei „MSG.ts“.

In der Tabelle 4.7 sind die einzelnen Felder von *MSG_REQUESTCONNECTION*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
from	string	*(Stringified)* ID des *Peers*, der sich neu Verbinden möchte.
data	string	*(Stringified)* Etwaige Daten die für eine neue Verbindung relevant sind.

Tabelle 4.7: Aufbau des *MSG_REQUESTCONNECTION* Objektes in einer Nachricht

MSG_REQUESTRESPONSE

MSG_ REQUESTRESPONSE ist ein Nachrichtentyp. Diese Nachricht beinhaltet die Antwort auf die Anfrage einer neuen Verbindung durch eine *MSG_ REQUESTCONNECTION* Nachricht.

Implementiert ist dieser Nachrichtentyp durch das Interface *MSG_ REQUEST_ RESPONSE* in der Datei „MSG.ts“.

In der Tabelle 4.7 sind die einzelnen Felder von *MSG_ REQUESTRESPONSE*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
from	string	*Stringified* ID des *Peers*, der sich neu Verbinden möchte.
data	string	*Stringified* Informationen als Antwort auf die Anfrage einer neuen Verbindung.

Tabelle 4.8: Aufbau des *MSG_ REQUESTRESPONSE* Objektes in einer Nachricht

MSG_GETINFORMATION

MSG_ GETINFORAMTION ist ein Nachrichtentyp. Diese Nachricht beinhaltet die Anfrage, eine Information aus dem Chord-Ring abzurufen.

Implementiert ist dieser Nachrichtentyp durch das Interface *MSG_ GETINFORAMTION* in der Datei „MSG.ts“.

In der Tabelle 4.9 sind die einzelnen Felder von *MSG_ GETINFORAMTION*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
requester	string	ID des *Peers*, der die Information anfragt.
identifier	string	Wert für die Zuweisung des Ergebnisses zum passenden Aufruf. Wird von jedem *Peer* selber festgelegt. Weitere Informationen hierzu in 4.2.2 „CommunicationHandler“ (Seite 29).
hash	string	ID/Hash der Information.

Tabelle 4.9: Aufbau des *MSG_ GETINFORAMTION* Objektes in einer Nachricht

MSG_SETINFORMATION

MSG_SETINFORMATION ist ein Nachrichtentyp. Diese Nachricht beinhaltet eine neue Information, die dem Chord-Ring hinzugefügt werden soll.

Implementiert ist dieser Nachrichtentyp durch das Interface *MSG_SETINFORMATION* in der Datei „MSG.ts“.

In der Tabelle 4.10 sind die einzelnen Felder von *MSG_SETINFORMATION*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
options	InformationOptions	Options Objekt einer Nachricht. Weitere Informationen zu dem *InformationOption* Typ in 4.4.1 „InformationOption“ (Seite 41).
information	Information	Informations Objekt, welches die eigentliche Information beinhaltet. Weitere Informationen zu dem *Information* Typ in 4.4 „Information“ (Seite 40).

Tabelle 4.10: Aufbau des *MSG_SetInformation* Objektes in einer Nachricht

MSG_WEBRTC_SIGNALDATA

MSG_WEBRTC_SIGNALDATA ist ein Nachrichtentyp. Diese Nachricht beinhaltet Signalisierungsinformationen, welche zwischen verschiedenen Peers ausgetauscht und zum Verbindungsaufbau benötigt werden.

Implementiert ist dieser Nachrichtentyp durch das Inferface *MSG_WEBRTC_SIGNALDATA* in der Datei „MSG.ts“

In der Tabelle 4.11 sind die einzelnen Felder von *MSG_WEBRTC_SIGNALDATA*, deren Typ und Bedeutung zu finden.

Key	Typ	Bedeutung
type	MSG_WEBRTC_SIGNALTYPES	Art der Signalisierungsinformationen.
signaldata	string	Signalisierungsobjekt in JSON-Format als String.

Tabelle 4.11: Aufbau des *MSG_SetInformation* Objektes in einer Nachricht

MSGServer

Nachrichten, die direkt vom *Signalling-Server* kommen, haben immer einen Typen, der mit dem Präfix „MSG_SERVER“ versehen ist. Folgend sind die 4 Nachrichtentypen vom Server aufgeführt.

- *MSG_SERVER_HELLO*

Key	Typ	Bedeutung
uid (optional)	string	Eindeutige ID des Peers.

Tabelle 4.12: Aufbau des *MSG_SERVER_HELLO* Objektes in einer Nachricht

- *MSG_SERVER_JOIN*

Key	Typ	Bedeutung
netid	string	ID des Chord-Rings, dem beigetreten werden soll.
state (optional)	MSG_STATE	Status des Beitritts. Bei Fehlern wird dieser als ERROR für die Rücksteuerung des Fehlers genutzt.
start (optional)	string	Einstiegspunkt in den Chord-Ring.

Tabelle 4.13: Aufbau des *MSG_SERVER_JOIN* Objektes in einer Nachricht

- *MSG_SERVER_CREATE*

Key	Typ	Bedeutung
netid	string	ID des Chord-Rings, der erstellt werden soll.
state (optional)	MSG_STATE	Status des Erstellens. Bei Fehlern wird dieser als ERROR für die Rücksteuerung des Fehlers genutzt.

Tabelle 4.14: Aufbau des *MSG_SERVER_CREATE* Objektes in einer Nachricht

- *MSG_SERVER_ERROR*

Key	Typ	Bedeutung
state	MSG_STATE	Status des Fehlers.
msg	MSG_BODY	Fehlerhafte Nachricht.

Tabelle 4.15: Aufbau des *MSG_SERVER_ERROR* Objektes in einer Nachricht

4.4 Information

Unter einer Information ist im Kontext dieser Arbeit eine von einem im Chord-Ring befindlichen Knoten verwaltete und eindeutig zuweisbare Menge an Daten gemeint.

In der Tabelle 4.16 ist der Aufbau eines Informations-Objektes beschrieben.

Key	Typ	Bedeutung
tag	string	Tag der Information.
hash	string	Hashwert des Tags. Genutzt für die Zuweisung der Information zu einem Knoten im Ring.
creator	string	Ersteller der Information.
datecreated	number	Zeitstempel der Erstellung der Information. UNIX-Timestamp basierend auf Erstellersystem.
data	string	Daten der Information. Format der Daten sind frei wählbar (JSON/string für Information-Objekt).

Tabelle 4.16: Aufbau einer Information

4.4.1 InformationOption

Neben der Information können weitere Optionen für diese mitgegeben werden. Besonderheit ist, dass alle Optionen optional sind.

In der Tabelle 4.17 ist der Aufbau der Optionen einer Information beschrieben.

Key	Typ	Bedeutung
bestbefore	number	Zeitstempel bis zu dem die Information gültig ist. UNIX-Timestamp basierend auf Erstellersystem.
deletedate	number	Zeitstempel zu dem die Information gelöscht werden soll. UNIX-Timestamp basierend auf Erstellersystem.
backupdepth	num	Replizierungsfaktor im Ring.

Tabelle 4.17: Aufbau der Optionen einer Information

4.5 Interaktion

In der folgenden Sektion werden die Interaktionen auf Akteurebene dargestellt.

4.5.1 Create/Join Chord-Ring

In folgender Abbildung 4.9 sind die Nachrichtentypen dargestellt, die die Partner beim Erstellen und Beitreten eines Chord-Ring austauschen.

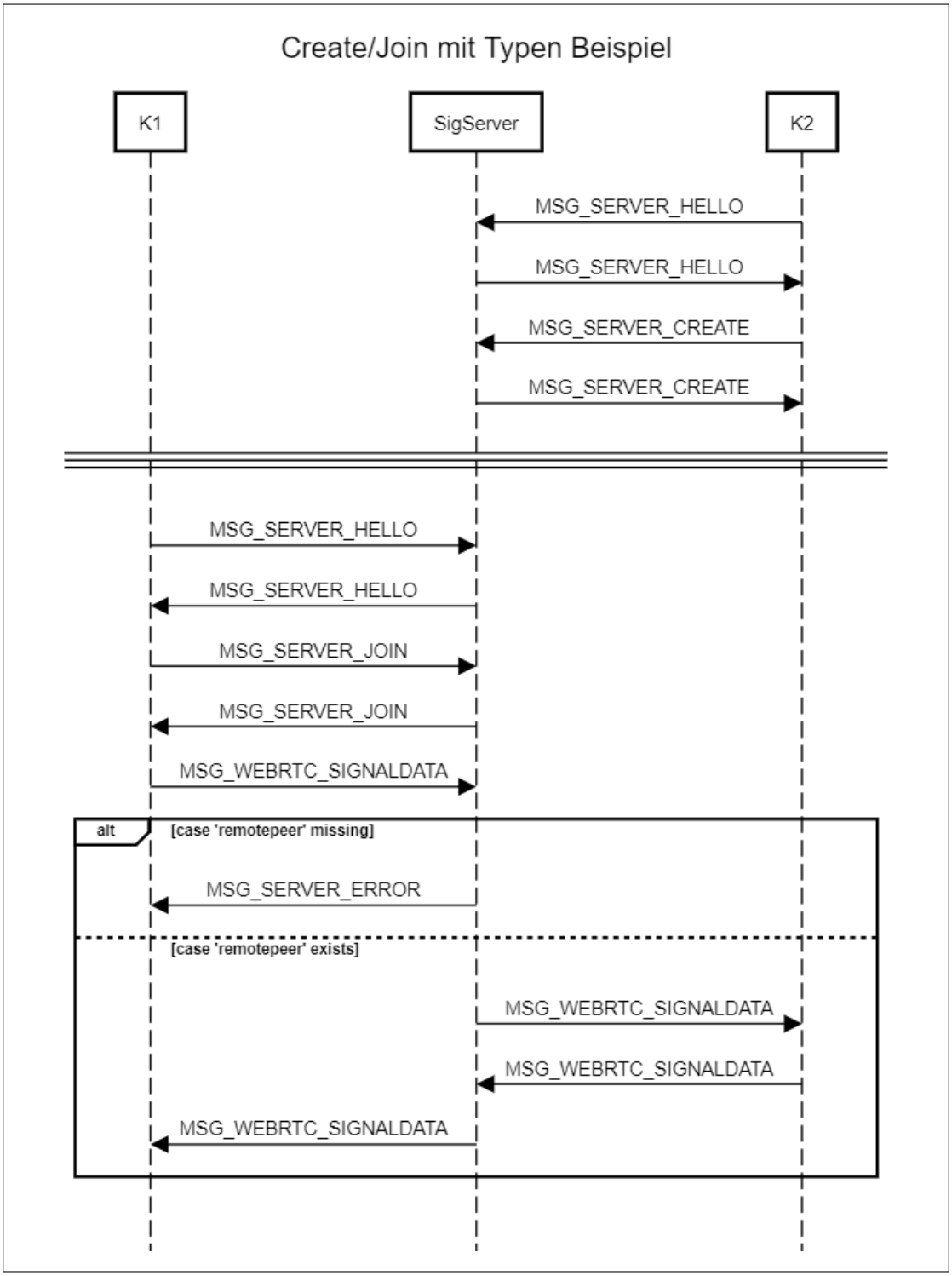

Abbildung 4.9: Interaktion: Create/Join mit Typen

Create Chord-Ring

In Abbildung 4.10 sind die Interaktionen bei der Erstellung eines neuen Chord-Ring dargestellt. Beteiligt hierbei sind immer: Ein Knoten und ein Signalling-Server.

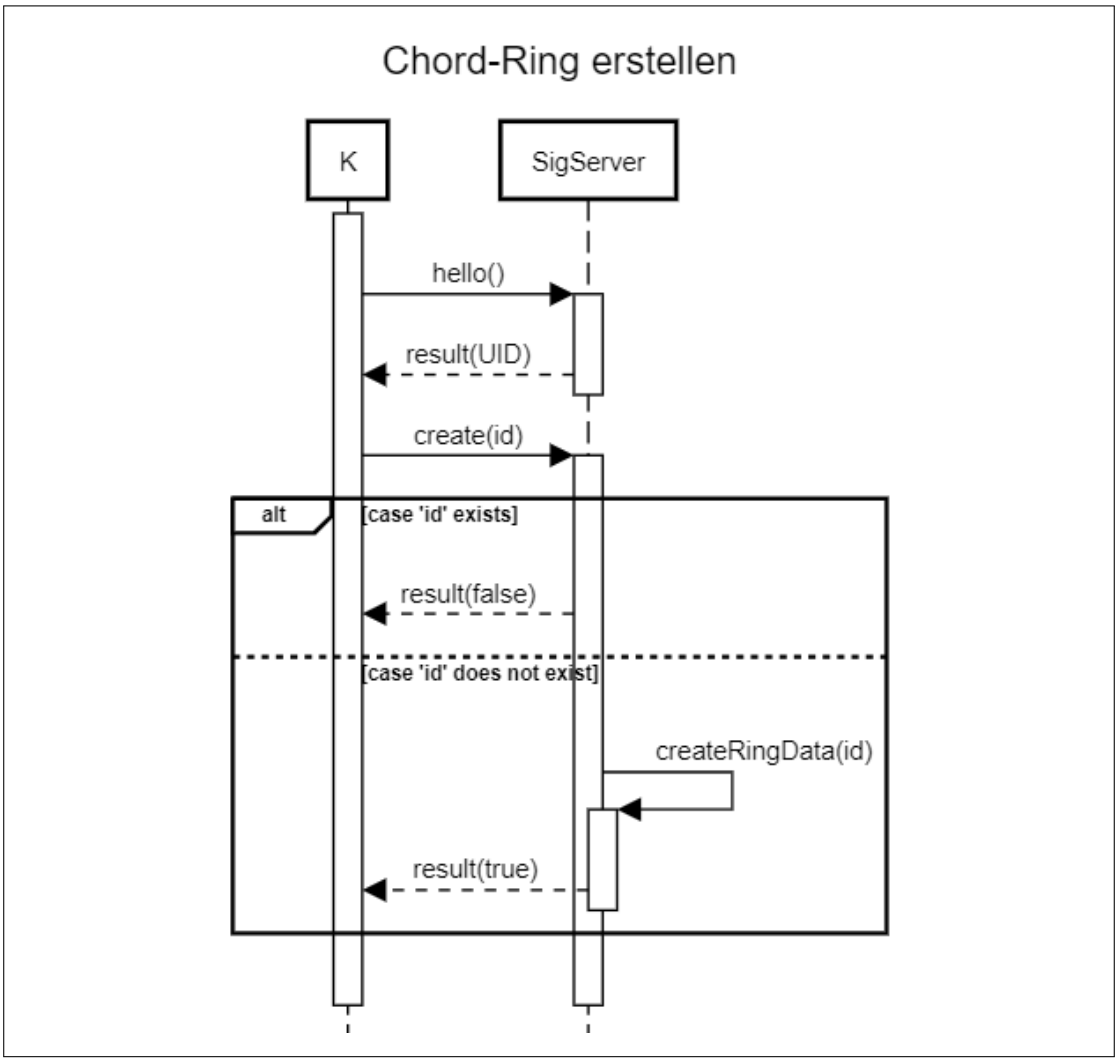

Abbildung 4.10: Interaktion: Create Chord-Ring

Der Knoten K meldet sich beim Signalling-Server *SigServ* mit einem „hello“ und erhält darauf die vom Signalling-Server für ihn festgelegte eindeutige ID zurück. Anschließend teilt K dem Signalling-Server mit einem „create“ und der Ring-ID mit, dass er einen neuen Chord-Ring erstellen möchte. Sofern der Ring schon existiert, wird K mitgeteilt,

dass das Erstellen nicht möglich ist. Andernfals legt der Signalling-Server alle nötigen Informationen zur Verwaltung des Rings an und teilt K den Erfolg mit.

Join Chord-Ring

In Abbildung 4.11 sind die Interaktionen bei der Erstellung eines neuen Chord-Ring dargestellt. Beteiligt hierbei sind immer mindestens: Ein Knoten und ein Signalling-Server. Vorraussetung ist jedoch, dass eine Chord-Ring bereits exisitert, also muss mindestens ein zweiter Knoten existieren.

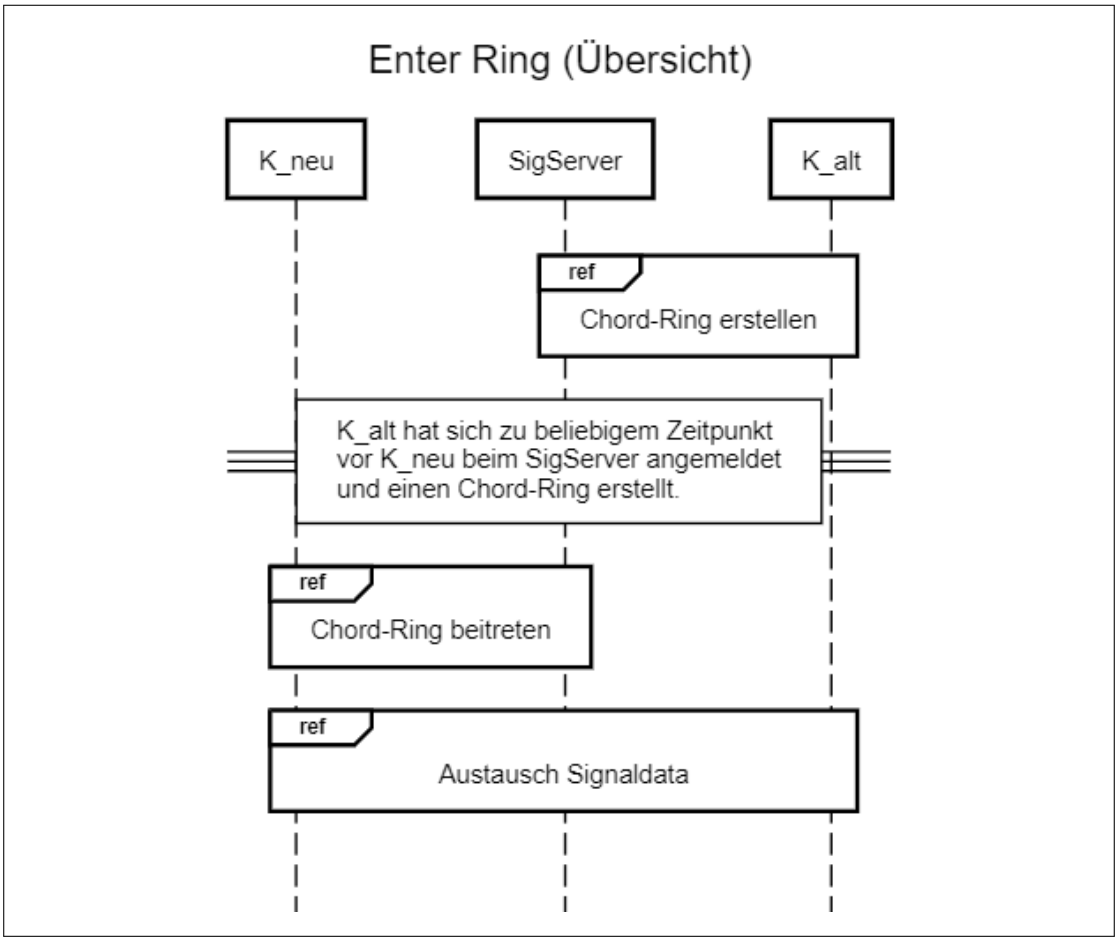

Abbildung 4.11: Interaktion: Join Chord-Ring Übersicht

In Abbildung 4.11 sind die Phasen der Erstellung und des Beitretens eines Chord-Ringes

getrennt. Auf die Erstellung wurde bereits eingegangen, deshalb wird hier jetzt auf den Beitritt eingegangen.

Die Abbildung 4.12 zeigt den Knoten *K*, welcher dem Ring beitreten will, sowie den Signalling-Server *SigServer* und den Knoten *Kentry*, welcher ursprünglich den Ring erstellt hat und nur der übersichtlichkeit halber aufgerührt ist.

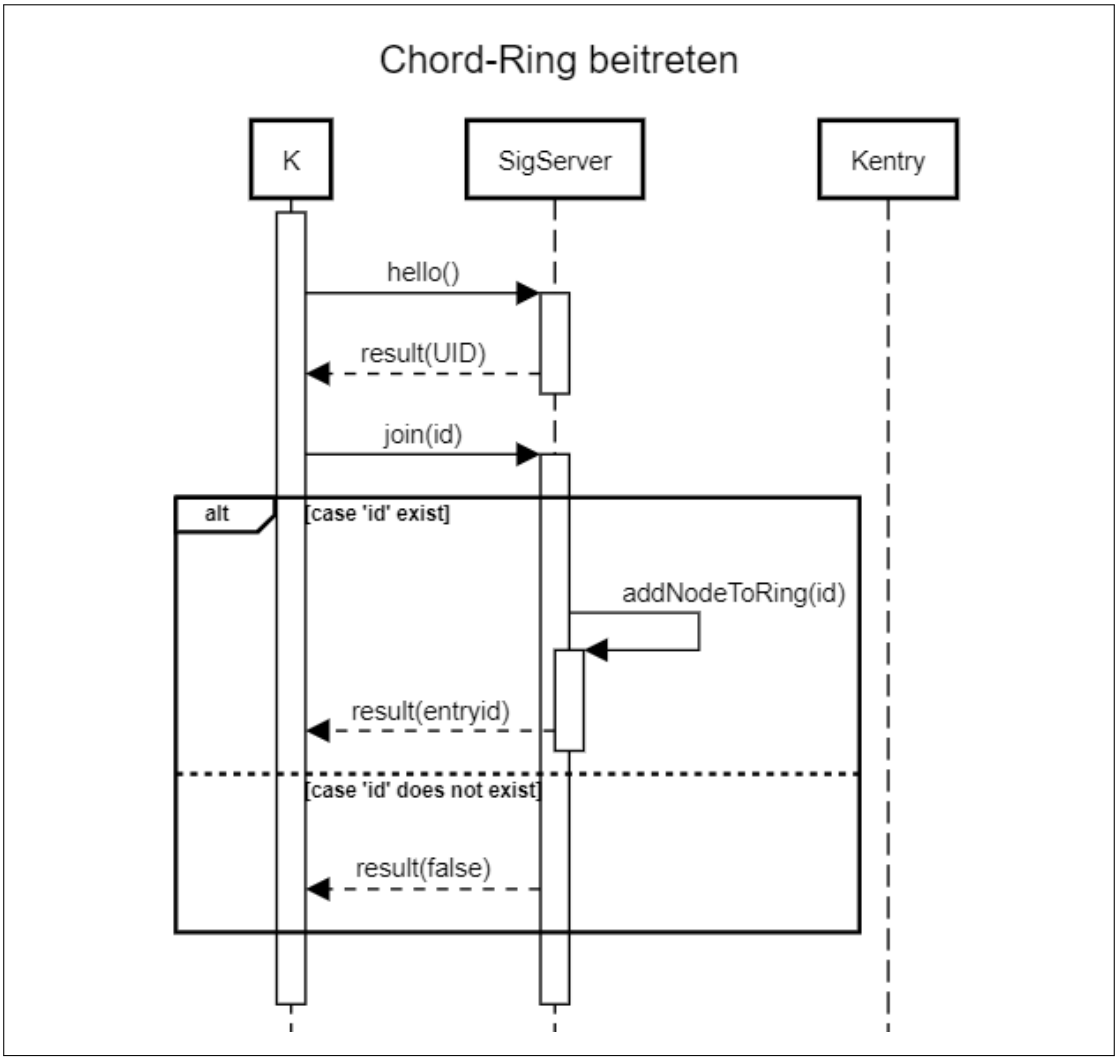

Abbildung 4.12: Interaktion: Join Chord-Ring

Nachdem *K* sich beim *Signalling-Server* anmeldet und seine ID erfährt, meldet er dem *Signalling-Server*, dass er einem Chord-Ring „id“ beitreten möchte. Sofern dieser Ring

bereits exisitert, ergänzt der Signalling-Server seine verwalteten Ring-Informationen um den neuen Knoten K und sendet K die ID des Einstiegsknotens in den Ring zurück. Sollte der Ring nicht exisiteren, wird K ein Fehler mitgeteilt.

4.5.2 Autausch Signaldata

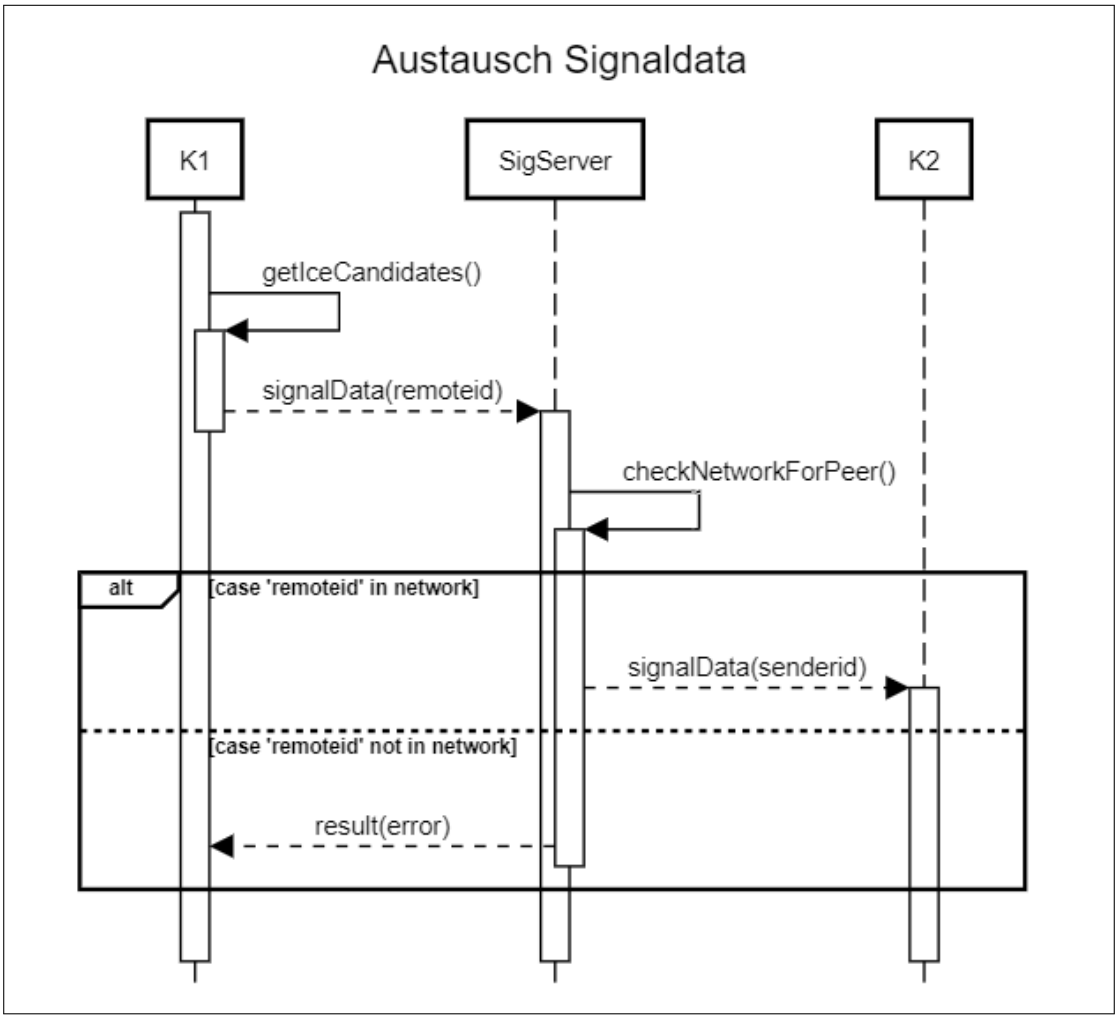

Abbildung 4.13: Interaktion: Austausch Signaldata

4.5.3 Remote Call

Der entfernte Aufruf einer Funktion ist in der folgenden Abbildung 4.15 dargestellt. Die verwendeten Nachrichtentypen sind in Abbildung 4.14 zu erkennen.

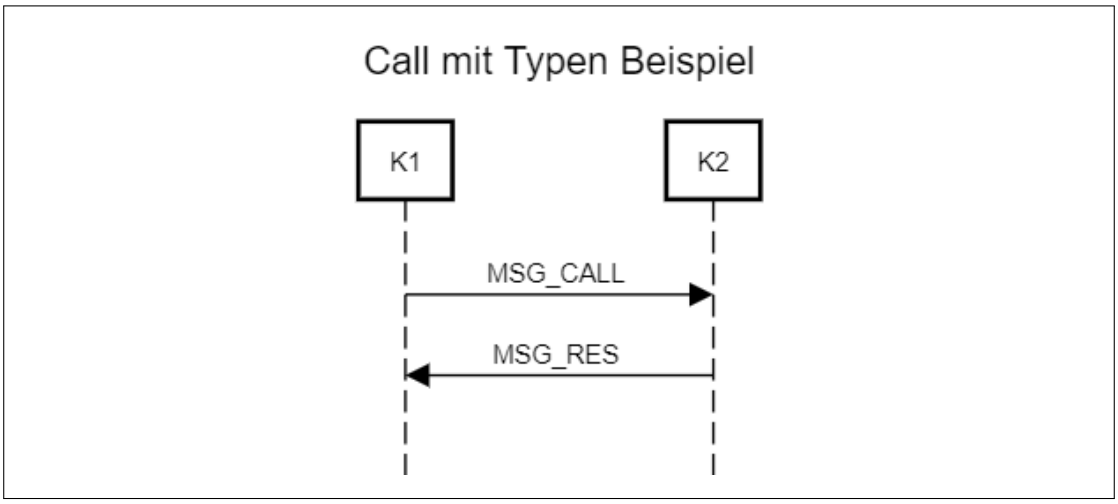

Abbildung 4.14: Interaktion: Remote Call mit Typen

In Abbildung 4.15 ist dargestellt, dass das Ergebnis eines entfernten Aufrufs nicht zwangsläufig direkt zum Anfragenden zurückgesendet wird. Grund hierfür ist, dass die von Chord verwalteten Verbindungen logisch unidirektional behandelt werden und somit nicht zwangsläufig eine direkte logische Verbindung vom Ziel- zum Urspungsknoten existiert.

So ruft hier der Knoten *K1* eine entferne Methode von *K2* auf, *K2* schickt das Ergebnis dieses Aufrufs jedoch nicht direkt zurück, sondern über den ihm bekannten Weg über Knoten *K3*.

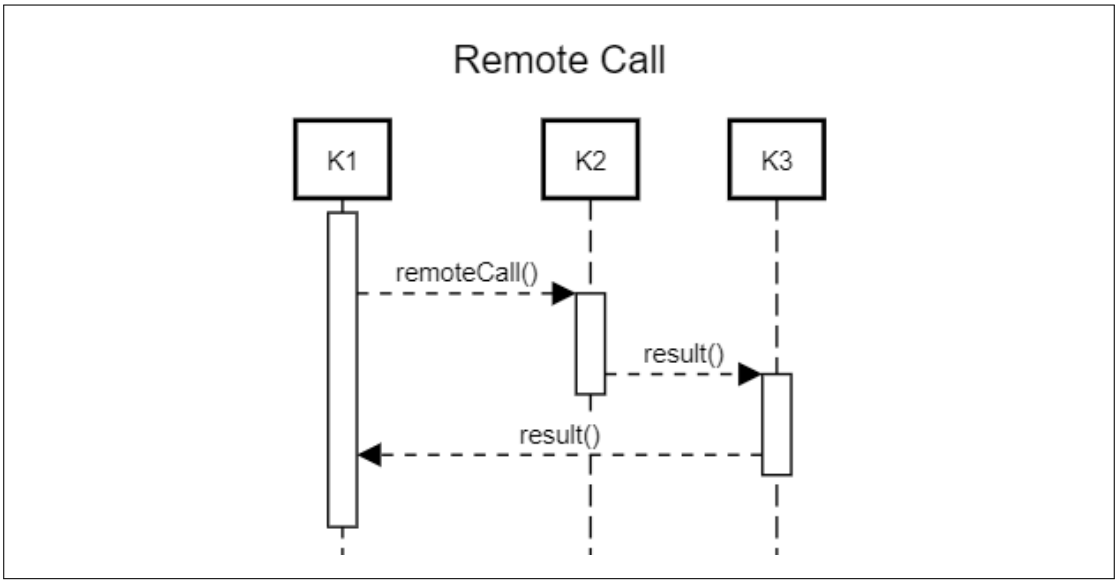

Abbildung 4.15: Interaktion: Remote Call

4.5.4 Set/Get Information

Im folgenden wird der Ablauf beim Einbringen und Abrufen von Informationen im Chord-Ring dargestellt.

In Abbildung 4.16 sind die Nachrichtentypen zwischen den Nodes dargestellt, die beim Abrufen und Einbringen von Informationen verwendet werden.

Im oberen Beispiel fragt *K1* eine Information bei *K2* an, *K2* leitet diese Anfrage an *K3* weiter. *K3* schickt nun das Ergebnis der Abfrage - entweder die Information oder einen Fehler - über einen möglicherweise anderen Weg an den Anfragenden *K1* zurück.

Ähnlich der Abfrage von Informationen ist im unteren Beispiel gezeigt, dass *K1* eine Information in den Chord-Ring einbringt und diese an *K2* sendet. *K2* kennt einen passenderen für die Information zuständigen Knoten und schickt diese Nachricht an ihn weiter.

In beiden Fällen war die Knotenauswahl nur beispielhaft, Informationen können unter Umständen auch beim einbringenden Knoten verbleiben oder über viele Knoten weitergereicht werden. Die Anfrage von Informationen kann ebenfalls je nach Gegebenheiten über viele Knoten laufen.

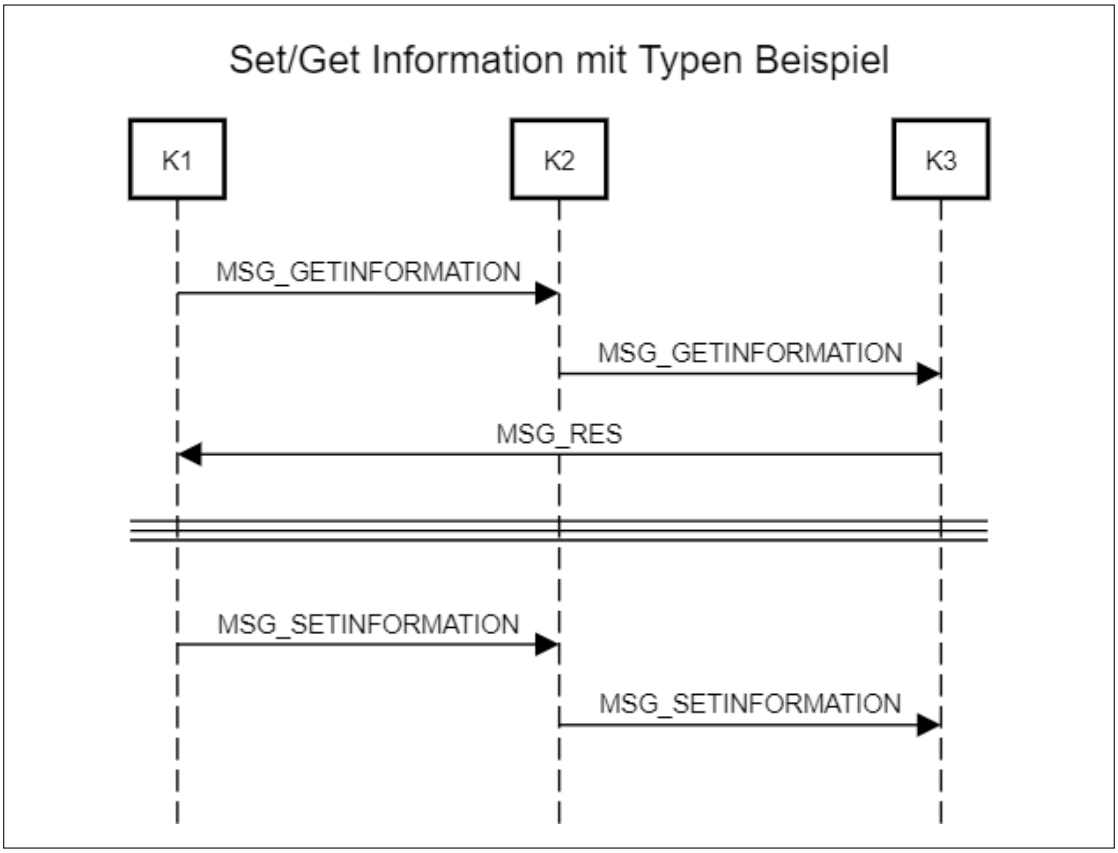

Abbildung 4.16: Interaktion: Set/Get Information mit Typen

In der folgenden Abbildung 4.17 ist der konkrete Ablauf des Einbringens einer Information zu sehen. Der Knoten KS möchte eine Information in den Chord-Ring einbringen und ruft hierzu lokal die Methode „setInformation()“ auf. In dieser Methode wird der lokal bekannte am besten passende Knoten für die Information ermittelt und die Information an diesen weitergereicht. Der Knoten Kx steht beispielhaft für alle auf dem Weg vorkommenden Knoten. Alle Knoten auf dem Weg wiederholen den Vorgang, bis der Knoten erreicht wurde, der sich für die Information verantwortlich sieht (KT) und diese speichert.

Neue Information

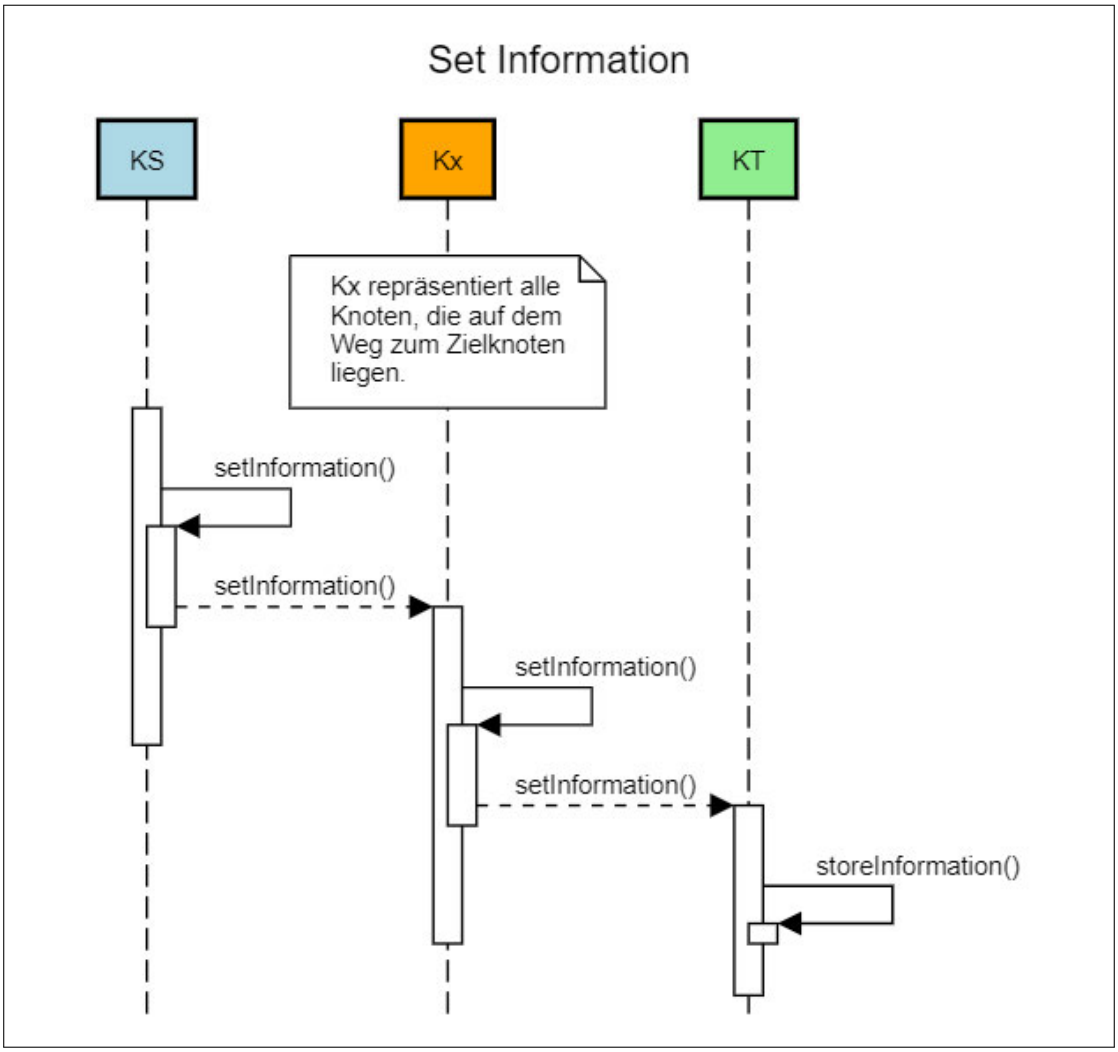

Abbildung 4.17: Interaktion: Set Information

In der folgenden Abbildung 4.18 ist der konkrete Ablauf des Anfragens einer Information zu sehen. Der Knoten *K1* möchte eine Information aus dem Chord-Ring anfragen, ruft hierzu lokal die Methode „getInforamtion(H)“ auf und übergibt den Hashwert der Information. Anschließend wird der bekannte am besten passende Knoten ermittelt, der die Information beherbergen könnte. In diesem Fall wird die Anfrage an die Information an *K2* weitergegeben. *K2* wiederholt den Vorgang und leitet die Anfrage an *K4* weiter. *K4* besitzt die Information und gibt sie über einen passenden Weg an den Anfragenden, *K1*,

zurück. Der Rückweg führt in diesem Beispiel über den Knoten *K3* zurück zu *K1*.

Information abrufen

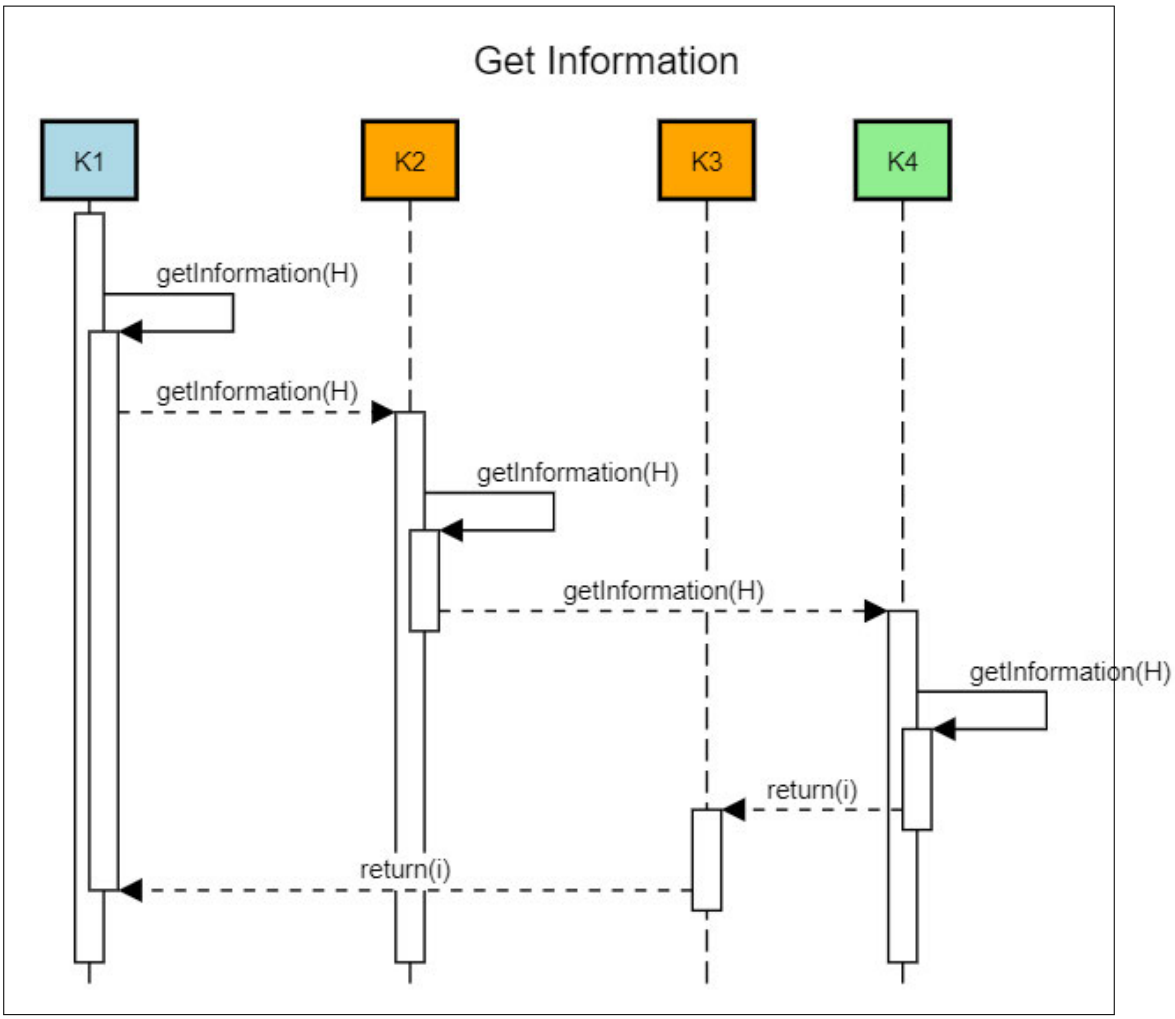

Abbildung 4.18: Interaktion: Get Information

In der Abbildung 4.18 ist der typische Ablauf beim Abrufen eine Information zu sehen. Wie durch *Chord* definiert kann eine solche Anfrage über mehrere Knoten führen[3]. Die Rückführung des Ergebnisses kann auch hier über einen anderen Weg geschehen als die Anfrage.

5 Implementierung

In dem folgenden Kapitel geht es um die konkrete Implementierung von „ChordComm“, die Struktur und die konkrete Verwendung des Projektes. Hierbei wird nicht auf jedes einzelne, sondern nur auf einige spezifische Details eingegangen.

5.1 Projektstruktur

In der Abgabe sind 3 Ordner zu finden.

- An erster Stelle der Ordner „ChordComm“. Dieser beinhaltet das gesamte gleichnamige Paket „chordcomm“. In diesem Ordner befinden sich außerdem die bereits vortranspilierten Dateien. Die Quelldateien sind im Unterordner „src“ zu finden.
- An zweiter Stelle der Ordner „ChordCommExperiments“. Dieser beinhaltet die 3 hier durchgeführten Experimente. Diese sind auch bereits vortranspiliert und gebündelt. Die Quelldateien sind im Unterordner „src“ zu finden.
- Zuletzt der Ordner „misc“. Dieser enthält das Python-Script für die Ausgabe der SHA256-Bilder.

5.2 Implementierungen

Im folgenden Abschnitt wird auf die konkrete Implementierung einiger Funktionalitäten eingegangen. Um Platz zu sparen sind Funktionen oft von den Klassen losgelöst, einzelne Passagen durch „...“ ersetzt und Kommentare entfernt. Auf einige der Codebeispiele wird nur verwiesen, sie sind dann im Appendix zu finden.

5.2.1 CommunicationHandler

Der *CommunicationHandler* bildet das bindende Glied zwischen der Chordlogik *(Chord-Client)* und der konkreten Verbindungslogik. Seine Aufgabe ist es auch, die lokalen Informationen zu verwalten und zu stabilisieren.

Im Folgenden werden die Kerneigenschaften des *CommunicationHandler* anhand der konkreten Implementierung gezeigt. Dabei wird auf die Aspekte der Verarbeitung von Nachrichten, sowohl eingehend als auch ausgehend, eingegangen. Außerdem wird gezeigt, wie neue Verbindungen erstellt und Verbindungsabbrüche behandelt werden. Zuletzt wird noch auf die Informationshaltung im Bezug auf das Verarbeiten von neuen Information und das Stabilisieren von vorhandenen eingegangen.

Verarbeitung eingehender Nachrichten

Eingehende Nachrichten werden immer zuerst in ihr *MSG_BODY* Format geparsed. Anschließend kann basierend des gesetzten Typs der Nachricht, sowie Ziel der Nachricht entschieden werden, wie mit der Nachricht umzugehen ist. In Listing 5.1 ist dieser Ablauf zu sehen.

```
1  async receive(msg : string) : Promise<void> {
2      let msgbody : MSG_BODY = JSON.parse(msg);
3
4      if(BigInt(msgbody.to) == this.localPeerID) {
5
6          // Message is for us
7          if(msgbody.type == MSG_TYPES.RESULT) {
8
9              // Message is a result
10             let msgres : MSG_RES = JSON.parse(msgbody.data);
11             this.resList.set(BigInt(msgres.identifier), msg);
12         } else if(msgbody.type == MSG_TYPES.CALL) {
13
14             // Message is a call
15             let msgcall : MSG_CALL = JSON.parse(msgbody.data);
16             this.remoteCall(msgcall.calltype, BigInt(msgcall.value), BigInt(
       msgbody.from), BigInt(msgcall.identifier));
17         }
18
19         ...
20     }
```

```
21 }
```

Listing 5.1: Implementation CommunicationHandler: receive

Asynchrones Senden von Nachrichten

Eine Nachricht asynchron an einen anderen Peer zu senden, muss dieser lediglich als Verbindung initialisiert sein, oder neu initialisiert werden. In Listing 5.2 ist der gekürzte Ablauf dargestellt.

```
1  sendAsync(msg : string, remotePeerID : bigint): void {
2      ...
3      if(remotePeerID == this.localPeerID) {
4          this.receive(msg);
5          return;
6      }
7
8      if(!this.checkConnection(remotePeerID)) {
9          this.addConnection(remotePeerID, "");
10     }
11
12     if(this.checkConnection(remotePeerID)) {
13         let con = this.conList.get(remotePeerID);
14         if(con) {
15             con.send(msg, false);
16         }
17     }
18 }
```

Listing 5.2: Implementation CommunicationHandler: send async

Synchrones Senden von Nachrichten

Da aus technischer Sicht seitens WebRTC keine synchronen Anfragen definiert sind, wird die asynchrone Sende-Logik um einen Identifier und eine Ergebnisliste - in der alle Antworten/Results landen - ergänzt. In Listing 5.3 ist der Aufruf zum asynchronen Senden, sowie das Eintragen in die Ergebnisliste und Warten auf das Ergebnis zu sehen.

```
1  async sendSync(msg: string, remotePeerID : bigint, ident : bigint) : Promise<
       string> {
2      return new Promise<string>(async (resolve, reject) => {
```

```
3         ...
4         this.sendAsync(msg, remotePeerID);
5
6         while(!this.resList.has(ident)) {
7             ...
8             await delay(250);
9             ...
10        }
11
12        let res : string | undefined = this.resList.get(ident);
13
14        if(res) {
15            this.resList.delete(ident);
16            resolve(res);
17        } else {
18            reject(res);
19        }
20    });
21 }
```

Listing 5.3: Implementation CommunicationHandler: send sync

Hinzufügen neuer Verbindungen

Neue Verbindungen werden der Liste bestehender Verbindungen durch das Hinzufügen eines durch die *WebRTCConHandleFactory* erstellten Verbindungsobjektes realisiert. In Listing 5.4 ist dieser Vorgang zu sehen. Die Referenz auf die *Factory*-Funktion wird dem *CommunicationHandler* bei der Initialisierung übergeben und deifiniert die Art der Verbindung - hier *WebRTC*.

```
1 public addConnection(remotePeerID : bigint, data : string) : boolean {
2     ...
3     if(!this.checkConnection(remotePeerID)) {
4         let remotePeerHandle : IConnectionHandle = this.conHandleFactoryCB(
      this, remotePeerID, this.localPeerID, data);
5         this.conList.set(remotePeerID, remotePeerHandle);
6         remotePeerHandle.startHandle();
7         return true;
8     } else {
9         return false;
10    }
```

```
11 }
```

Listing 5.4: Implementation CommunicationHandler: add connection

Verarbeitung von Verbindungsabbrüchen

Verbindungsabbrüche werden dem *CommunicationHandler* entweder direkt von einem bestehenden *IConHandle* gemeldet, oder durch den Server bekanntgegeben. In Listing 5.5 ist zu sehen, wie der *CommunicationHandler* den *ChordClient* über den Verbindungsabbruch zu einem Peer informiert. Der *ChordClient* kann anschließend seine bestehenden Referenzen zu anderen Peers anpassen.

```
1 conLostToPeer(peerId : bigint) : void {
2     if(this.chordClient) {
3         ...
4         this.chordClient.conLostToPeer(peerId);
5     }
6 }
```

Listing 5.5: Implementation CommunicationHandler: handle disconnect

Information speichern

Für einen gegebenen Hashwert einer Information können mehrere Exemplare verschiedener Ersteller existieren. Sollte eine Information bereits von einerm Ersteller existieren, wird diese überschrieben. In Listing A.1 „Implementation CommunicationHandler: handle information“ (Seite 87) ist die stark gekürzte Implementation des lokalen hinzufügens einer Information zu sehen.

5.2.2 ChordClient und ChordClientStub

Der *ChordClient* ist für die gesamte Chordlogik zuständig. Der *ChordClientStub* implementiert wie auch der *ChordClient* das Interface *IChordClient* und ist als Stub eines entfernten *ChordClient* für die Verarbeitung von Remote-Calls zuständig.

Im Folgenden wird auf die Stabiliserung des Ringes und das Fixen der Finger eingegangen. Außerdem wird auf den Umgang mit Verbindungsabbrüchen eingegangen. Zuletzt wird die Verarbeitung von Remote-Calls über den *ChordClientStub* dargestellt.

Stabilisieren des Rings

Um einen Ring zu stabilisieren muss jeder Knoten seinen korrekten *successor* kennen und sich ihm bekanntmachen. Beim Stabilisieren wird also jeden zyklus geprüft, ob der aktuelle *predecessor* des lokalen *successor* ein besser neuer *successor* wäre. Listing A.2 „Implementation ChordClient: stabilize“ (Seite 88) zeigt diesen Vorgang in gekürzter Form.

Fixen der Finger

Um einen in der Fingertable aufgeführten Finger zu *fixen* muss für diesen Hashwert der *successor*-Knoten gefunden werden. In Listing 5.6 ist dieser Vorgang zu sehen. Durch das *fixen* von zwei Fingern gleichzeitig kann - in der Theorie - im Idealfall die Zeit bis alle Finger *gefixt* sind halbiert werden.

```
1  async fix_fingers() {
2      this.curFingerItvl++;
3      this.curFingerIdx2 = (this.curFingerIdx + (this.size >> 1)) % this.size;
4
5      let x : bigint  = this.fingerTable.table[this.curFingerIdx].hash;
6      let x2 : bigint = this.fingerTable.table[this.curFingerIdx2].hash;
7
8      this.fingerTable.table[this.curFingerIdx].peerID = await this.
       find_successor(x);
9      this.fingerTable.table[this.curFingerIdx].lastUpdate = (new Date()).
       getTime();
10
11     this.fingerTable.table[this.curFingerIdx2].peerID = await this.
       find_successor(x2);
12     this.fingerTable.table[this.curFingerIdx2].lastUpdate = (new Date()).
       getTime();
13
14     this.curFingerIdx  = (this.curFingerIdx + 1) % this.size;
15     this.curFingerIdx2 = (this.curFingerIdx2 + 1) % this.size;
16 }
```

Listing 5.6: Implementation ChordClient: fix fingers

Verarbeitung von Verbindungsabbrüchen

Sollte der *ChordClient* Meldung über einen Verbindungsabbruch erhalten, ist dies nur relevant, sollte der getrennte Knoten sein direkter *predecessor* gewesen sein. Trifft dies zu, muss die Referenz auf diesen sofort entfernt werden. Listing 5.7 zeigt dieses Vorgehen.

```
1 conLostToPeer(peerId : bigint) {
2     if(peerId == this.predecessor) {
3         this.removeStub(this.predecessor);
4         this.predecessor = null;
5     }
6 }
```

Listing 5.7: Implementation ChordClient: handle disconnects

ChordClientStub: Verarbeitung von Remote-Calls

Der *ChordClientStub* implementiert das selbe Interface wie der *ChordClient*. Dadruch agiert der *ChordClientStub* für den *ChordClient* wie ein direkter anderer *IChordClient*. Aufrufe der implementierten Funktionen bewirkten, dass eine neue Nachricht passend zum Aufruf erzeugt und an den *CommunicationHandler* weitergegeben wird. Sollte es sich um einen Aufruf mit Rückgabewert handeln, wird auf die Antwort des synchronen Sendens vom *CommunicationHandler* gewartet. In Listing A.3 „Implementation ChordClient: remote calls“ (Seite 89) ist der volle Ablauf eines solchen Aufrufes mit Rückgabewert zu sehen.

5.2.3 WebRTCConHandle

Der *WebRTCConHandle* realisiert eine konkrete Verbindung zu einem anderen Peer mittels *WebRTC*. Dabei abstrahiert dieser die gesamte Verbindungslogik und kapselt diese nach außen-gehend. Nachrichten die gesendet werden sollen, bevor die Verbindung existent ist, werden gebuffert und nach erfolgreichem Verbindungsaufbau gesendet. Sollte eine Verbindung abbrechen meldet der *WebRTCConHandle* dies dem *CommunicationHandler*.

Buffern der Nachrichten

In dem Listing 5.8 ist das Buffern von Nachrichten und senden des Buffers zu sehen.

```
send(msg : string, b: boolean) : void {
    if(this.channel && this.channel.readyState == "open") {
        this.channel.send(msg);
    } else if(this.state == ConnectionState.FAILED) {
        // Error to CommunicationHandler
        ...
    } else {
        this.buffer.push(msg);
    }
}

sendBuffer() {
    let bufCpy = [...this.buffer];
    this.buffer = [];
    while(bufCpy.length > 0) {
        let curMsg : string | undefined = bufCpy.shift();
        if(curMsg != null) {
            this.send(curMsg, true);
        }
    }
}
```

Listing 5.8: Implementation WebRTCConHandle: message buffer

Verarbeitung von Verbindungsabbrüchen

In dem Listing 5.9 ist die Meldung über einen Verbindungsabbruch zu sehen.

```
private channelOnClose(ev: Event) {
    this.state = ConnectionState.FAILED;
    this.sendBuffer();
    this.comHandler.conLostToPeer(this.localPeerID);
}
```

Listing 5.9: Implementation WebRTCConHandle: handle disconnects

5.2.4 FullClient

Der *FullClient* stellt das Wurzelelement von *ChordComm* dar. Er verbindet alle anderen Module und stellt die Schnittstellen nach außen zur Verfügung.

Im Folgenden wird auf den Funktionsumfang inklusive der Schnittstellen des *FullClient* eingegangen. Außerdem wird seine Funktion als Bootstrapper für dieses Projekt dargestellt.

Funktionsumfang

Das von dem *FullClient* implementierte Interface *IFullClient* und der somit nach außen verfügbare Funktionsumfang ist in dem Listing 5.10 zu sehen.

```
1  interface IFullChordClient {
2      getFingerTable() : FingerTable | undefined;
3
4      addInformation(obj : Information, opt : InformationOptions, isTagHash :
       boolean) : void;
5      getInformation(hash : string, isTagHash : boolean) : Promise<Information
       []>;
6      deleteInformation(information : Information, opt : InformationOptions) :
       void;
7      getAllInformations() : Map<string, Map<bigint, Information>>;
8
9      create(networkid : string, serverurl? : string) : void;
10     connect(networkid : string, serverurl? : string) : void;
11     disconnect() : void;
12
13     init(options : FullClientOptions) : void;
14 }
```

Listing 5.10: Implementation FullClient: functions

Bootstrap

Bei der Erstellung eines neuen *FullClient* werden diesem verschiedene Referenzen auf *Factory*-Funtionen übergeben. Diese erlauben das austauschen gewisser Komponenten. In dem Listing 5.11 ist der Konstruktor, sowie die Referenzen auf alle anderen Komponenten zu sehen.

```
class FullClient implements IFullChordClient {
    chordClient : IChordClient | undefined;
    fingerTable : FingerTable | undefined;
    commHandler : ICommunication | undefined;

    ...

    constructor(ownID : bigint,
                cho : CommunicationOptions = new CommunicationOptions({}),
                conHandleFactoryCB : (comHandler : ICommunication,
    remotePeerID : bigint, localPeerID : bigint, data : string) =>
    IConnectionHandle,
                conHandleHelloCB : (comHandler: ICommunication, ID : bigint)
    => void,
                size : number = 256) {
        this.ownID = ownID;
        this.cho = cho;
        this.conHandleFactoryCB = conHandleFactoryCB;
        this.conHandleHelloCB = conHandleHelloCB;
        this.size = size;
    }

    ...
}
```

Listing 5.11: Implementation FullClient: bootstrap

5.2.5 MSG

In der Datei „MSG.ts“ sind die für die Nachrichten benötigten Interfaces und Enumerierungen implementiert. Sie beschreiben auf Datenebene, wie Nachrichten ausgetauscht werden.

Im Folgenden wird auf den Aufbau der einzelnen Typen und wie diese verschachtelt werden eingegangen. Anschließend wird auf das Indempotente Verhalten der Nachrichtenverarbeitenden Funktionen mit Bezug zu den Typen eingegangen. Zum Schluss wird Anhand eines Beispiel gezeigt, wie diese Datenstrukturen erstellt und verwendet werden.

Aufbau der Typen

```
interface MSG_BODY {
    to : string,
    from: string,
    type : MSG_TYPES,
    data : string,
    identifier? : string
};
```

Listing 5.12: Implementation MSG: form of messages

Verwendungsbeispiel: Information in den Ring einbringen

In dem Listing A.4 „Implementation MSG: add information“ (Seite 90) ist Beispielhaft die Erstellung eines ganzen Nachricht, inklusive der Verschachtelung der einzelnen Nachrichtenschichten zu sehen.

5.2.6 SignallingServer

Der *SignallingServer* dient nebem dem Austausch der Verbindungsinformationen der Peers - ICECandidates etc. - auch dem Verwalten der einzelnen Ringe. Durch bleibende Verbindung zu den Peers im Ring bietet er so den zentralen Einstiegspunkt in den Ring.

Im Folgenden wird auf die Verwaltung der Ringe und Peers eingegangen. Außerdem wird die Verarbeitung von Verbindungsabbrüchen und die Folgen dieser erklärt.

Verwalten von Ringen und Peers

```
const networks : Map<string, Map<bigint, WebSocket>> = new Map;
const conlist : Map<bigint, WebSocket> = new Map;
...
const wss = new WebSocket.Server({
    server: server,
    path: "/socketserver"
});
...
wss.on('connection', (ws: WebSocket, req : Request) => {
    let hasID = false;
    let hash;
    let curPeerID : bigint;
    let curnetid : string | null = null;
```

```
14      let curnet : Map<bigint, WebSocket> | null = null;
15      ...
16      ws.on('message', (message: string) => {
17          const msg_body : MSG_BODY = JSON.parse(message);
18
19          if(msg_body.type == MSG_TYPES.SIGNALDATA) {
20              ...
21          } else if(msg_body.type == MSG_TYPES.SERVERHELLO) {
22              let recMsgHello : MSG_SERVER_HELLO = JSON.parse(msg_body.data);
23
24              if(options.allowOwnIDS && recMsgHello.uid) {
25                  curPeerID = BigInt(recMsgHello.uid);
26              } else {
27                  hash = sha256.digest(`${req.connection.remoteAddress}:${req.
    connection.remotePort}`);
28                  curPeerID = digestToBigInt(hash, size);
29              }
30              conlist.set(curPeerID, ws);
31              ...
32          }
33          ...
34      }
35  }
```

Listing 5.13: Implementation SignallingServer: handle peers and rings

Verarbeitung von Verbindungsabbrüchen und Folgen

```
1  ws.on('close', () => {
2      conlist.delete(curPeerID);
3
4      if(curnet){
5          curnet.delete(curPeerID);
6      }
7  });
```

Listing 5.14: Implementation SignallingServer: handle disconnects

5.2.7 Experimente

Die im Zuge dieser Arbeit durchgeführten Experimente behandeln die wichtigsten Eigenschaften von *ChordComm* im Bezug auf Stabilität und somit auch Verwendbarkeit.

Im folgenden werden die für die Experimente genutzten Verfahren erklärt, durch die die Stabilität im Ring geprüft wurde. Anschließend wird am Beispiel von Experiment 3 der Ablauf eines Experimentes von *ChordComm* gezeigt.

Prüfung der Stabilität des Rings

Dadruch, dass beim Experiement lokal Referenzen auf alle *FullClient* bestehen, kann über diese einfach geprüft werden, ob jeder den richtigen *successor* hat. In dem Listing A.5 „Implementation Experimente: check ring stability“ (Seite 91) ist dieser Ablauf zu sehen.

Beispiel: Peer-Definitionen Experiment 3

Durch festlegen des allgemeinen Präfix und Definition der einzelnen Suffixe, kann über eine Schleife der volle hexadezimale Bezeichner für jeden Knoten des Experimentes gebildet werden. Die Referenz auf jeden Knoten - also jeden *FullClient* - wird lokal gespeichert und kann über den in den abgekürzten Bezeichner aus den Experimenten angesprochen werden (z.B. „BEEF0C“). In dem Listing A.6 „Implementation Experimente: experiment 3“ (Seite 92) ist dieser Vorgang beispielhaft dargestellt.

5.3 Transpilierung

Da *NPM* Module genutzt werden und die *TypeScript*-Quelldateien nicht direkt vom Browser interpretiert werden können, muss vorab eine Transpilierung und Bündelung stattfinden. Die nötigen Schritte können mit den Befehlen aus dem Listing 5.15 durchgeführt werden. Ausgangsort für die Durchführung ist der Ordner „ChordComm“ - für die „ChordComm“ Distribution - und „ChordCommExperiments“ für die Experimente.

```
1 npm install
2 npm run loslos
```

Listing 5.15: Installation, Transpilierung und Bündelung

Anschließend finden sich in den Ordnern neue Ordner mit dem Namen „Dist“.

5.4 Signalling-Server und Experimente

Im „ChordComm“ Ordner kann nun mit dem im Listing 5.16 stehenden Befehl der *Signalling-Server* gestartet werden.

```
1 node ./build/SignallingServer
```

Listing 5.16: Nutzung SignallingServer

Nachdem die im Listing 5.15 gezeigte Bündelung für beide Ordner durchgeführt wurde, können die im Ordner „ChordCommExperiments/app/“ liegenden Experimente direkt im Browser geöffnet werden. In der Konsole lässt sich dann über dan Alias „JAWM“ die Schnittstelle zu den Experimente einsehen und die Phasen aufrufen. Die Bezeichnung richtet sich dabei nach denen der Experimente in dieser Ausarbeitung. Inital müssen jedoch immer die Knoten mittels „JAWM.initClients()“ initialisiert werden. Außerdem muss der *Signalling-Server* laufen.

5.5 NPM

„ChordComm“ wurde auf *NPM* veröffentlich und ist als „chordcomm“ bei *NPM* zu finden.[1] Außerdem kann als *Dependency* ind *NPM*-Projekten einfach „chordcomm“ angegeben werden und das Paket wird mitinstalliert.

[1] https://www.npmjs.com/package/chordcomm

6 Evaluation

In dem folgenden Kapitel werden mehrere Experimente durchgeführt, um die wesentlichen Eigenschaften von „ChordComm“ zu evaluieren. Eingegangen wird in diesen Experimenten dabei auf den Ein- und Austritt von Knoten und Informationen in den Ring, sowie auf die Replizierung vorhandener Informationen im Ring.

Für die meisten Experimente gilt, dass der für die Experimente verwendete Chord-Ring in einem stabilen Zustand ist. Um dies zu gewährleisten, werden die Fingertables der Knoten vorher schon einmal berechnet und während des Aufbaus vom Experiment mit den tatsächlichen abgeglichen. Um dieses Vorgehen bei 256 bit IDs der Knoten zu ermöglichen, wurden die IDs bewusst gut lesbar gewählt und im hexadezimalen dargestellt. Führende Nullen werden weggelassen, sodass die ID *0x0000000000000000000000000000000000000000000000000000000000BEEF00* einfach als *0xBEEF00* dargestellt wird, außerdem wird in den Texten und Diagrammen auf das Hexadezimale Präfix „0x“ verzichtet, somit also nur *BEEF00* geschrieben.

Um konkrete Fälle abprüfen zu können, werden die IDs der Nodes sich auf den Bereich von *BEEF00* bis *BEEFFF* beschränken, also 256 IDs (Die Hashwerte und somit auch die Fingertable werden weiterhin mit dem Standard von 256 bit gebildet). Irrelevante Fingertable Bereiche werden explizit nicht berücksichtigt, aber genannt.

Definitionen und Auswertungen

Um Auswertungen der Experimente vornehmen und Schritte inkl. ihrer Zustände beschreiben zu können, wird der Begriff „Stabil“ genutzt. Unter „Stabil“ wird im Kontext von *Chord* bzw. „ChordComm“ verstanden, dass jeder Knoten seinen logisch richtigen *successor* kennt und eine Verbindung zu ihm aufgebaut hat.

Um diesen Zustand darstellen zu können, wird eine Funktion genutzt, die für jeden Knoten - unter Berücksichtigung aller vorhandenen Knoten - die Richtigkeit prüft.

```
1 === SUCCESSOR-LIST ===
2 'BEEF00' incorrect Successor!
3 'BEEF03' => 'BEEF07'
4 'BEEF07' => 'BEEF0A'
5 'BEEF0A' incorrect Successor!
6 'BEEF0C' => 'BEEF0F'
7 'BEEF0F' incorrect Successor!
8 'BEEF26' => 'BEEF30'
9 'BEEF30' => 'BEEF3A'
10 'BEEF3A' incorrect Successor!
11 'BEEF60' => 'BEEF70'
12 'BEEF70' incorrect Successor!
13 'BEEFF0' => 'BEEFFA'
14 'BEEFFA' => 'BEEF00'
15 === ===
```

Listing 6.1: Beispiel Stabilitäsprüfung

In dem Listing 6.1 ist eine Ausgabe dieser Funktion zu sehen. Zu sehen sind die Knoten und mit einem Pfeil (=>) der zu dem Knoten vorhandene *successor*. Sollte ein Knoten einen falschen logischen *successor* besitzen, wird dies direkt geschrieben.

Wird des Begriff „Stabil“ im Kontext von Informationen verwendet, ist damit gemeint, dass die Information logisch dem richtigen Knoten zugewiesen und - gemäß den Replizierungsoptionen - richtig repliziert wurde.

Testumgebung

Alle Experimente wurden für Messungen in Firefox mit der Version 71.0 (64-Bit) durchgeführt. Die Version des Node.js Servers beträgt 10.15.3 (64-Bit).

6.1 Experiment 1: Ein- und Austritt von Knoten

In diesem Experiment wird der Ein- und Austritt von Knoten in einem kontrolliert aufgebauten Chord-Ring evaluiert.

1. **Fragestellung**
   Stabilisiert sich der Ring:

   a) nach dem Erstellen des Rings mit 8 Knoten?

   b) nach dem gleichzeitigen Hinzufügen 8 weiterer Knoten?

   c) nach dem gleichzeitigen unangekündigten Wegfall von 3 Knoten?

2. **Erwartung**
   Durch den Stabilisierungsmechanismus stabilisiert sich der Ring in allen 3 Szenarien. Dabei dauert die Stabilisierung maximal $N * I * 2$ Sekunden, wobei N der Anzahl der Knoten entspricht, also für a) gleich 8, b) gleich 16, c) gleich 15 und I dem zeitlichen Interval der Stabilisierung von 1000ms entspricht. Maximale Dauer bis stabil: a) 16 Sekunden, b) 32 Sekunden, c) 30 Sekunden.

3. **Planung**
   Um nachvollziehen zu können, ob und in welchem Tempo sich der Chord-Ring stabilisiert, wird die bereits bekannte Prüfungsfunktion in Verbindung mit Zeitstempeln genutzt. Die oben genannten Schritte a, b und c werden folgend als Phasen a, b und c bezeichnet. Bevor Phase b gestartet werden kann, muss Phase a stabil sein. Gleiches gilt für Phase c im Bezug auf b.

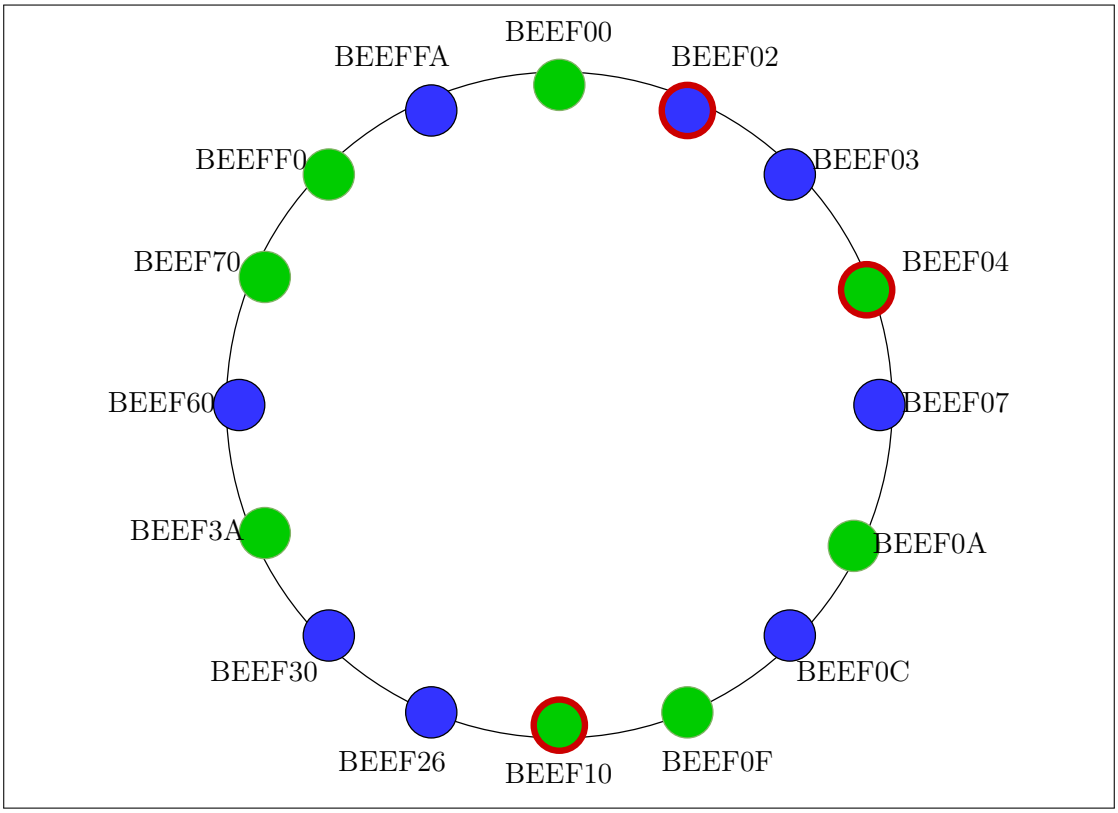

Abbildung 6.1: Planung des Rings für Experiment 1

In Abbildung 6.1 ist der geplante Chord-Ring zu sehen. Die grünen Knoten *(BEEF00, BEEF04, BEEF0A, BEEF0F, BEEF10, BEEF3A, BEEF70, BEEFF0)* werden in Phase a hinzugefügt, wobei *BEEF00* den *Chord-Ring* erstellt. In Phase b werden die blauen Knoten *(BEEF02, BEEF03, BEEF07, BEEF0C, BEEF26, BEEF30, BEEF60, BEEFFA)* hinzugefügt. In der letzten Phase c werden die rot umrandeten Knoten *(BEEF02, BEEF04, BEEF10)* gleichzeitig ohne explizite Benachrichtigung der anderen Knoten entfernt.

4. **Beobachtungen Anmerkung:** Die Listings wurden teilweise gekürzt und eingerückt, um sie besser darstellen zu können. Die in den Listings aufgeführten Ausgaben befinden sich in chronologischer Reihenfolge.

   In dem Listing 6.2 ist zu erkennen, dass der Ring sich in Phase a stabilisieren konnte.

```
1 === SUCCESSOR-LIST ===
```

```
2 'BEEF00' => 'BEEF04'
3 'BEEF04' => 'BEEF0A'
4 'BEEF0A' => 'BEEF0F'
5 'BEEF0F' => 'BEEF10'
6 'BEEF10' => 'BEEF3A'
7 'BEEF3A' => 'BEEF70'
8 'BEEF70' => 'BEEFF0'
9 'BEEFF0' => 'BEEF00'
10 === ===
```

Listing 6.2: Experiment 1: Ausgabe Phase a

In dem Listing 6.3 ist zu erkennen, dass der Ring sich in Phase b stabilisieren konnte.

```
1 === SUCCESSOR-LIST ===
2 'BEEF00' => 'BEEF02'
3 'BEEF02' => 'BEEF03'
4 'BEEF03' => 'BEEF04'
5 'BEEF04' => 'BEEF07'
6 'BEEF07' => 'BEEF0A'
7 'BEEF0A' => 'BEEF0C'
8 'BEEF0C' => 'BEEF0F'
9 'BEEF0F' => 'BEEF10'
10 'BEEF10' => 'BEEF26'
11 'BEEF26' => 'BEEF30'
12 'BEEF30' => 'BEEF3A'
13 'BEEF3A' => 'BEEF60'
14 'BEEF60' => 'BEEF70'
15 'BEEF70' => 'BEEFF0'
16 'BEEFF0' => 'BEEFFA'
17 'BEEFFA' => 'BEEF00'
18 === ===
```

Listing 6.3: Experiment 1: Ausgabe Phase b

In dem Listing 6.4 ist zu erkennen, dass der Ring sich in Phase c stabilisieren konnte.

```
1 === SUCCESSOR-LIST ===
2 'BEEF00' => 'BEEF03'
3 'BEEF03' => 'BEEF07'
4 'BEEF07' => 'BEEF0A'
5 'BEEF0A' => 'BEEF0C'
6 'BEEF0C' => 'BEEF0F'
```

```
'BEEF0F' => 'BEEF26'
'BEEF26' => 'BEEF30'
'BEEF30' => 'BEEF3A'
'BEEF3A' => 'BEEF60'
'BEEF60' => 'BEEF70'
'BEEF70' => 'BEEFF0'
'BEEFF0' => 'BEEFFA'
'BEEFFA' => 'BEEF00'
=== ===
```

Listing 6.4: Experiment 1: Ausgabe Phase c

Messung der Dauer in ms bis Stabilität erreicht wurde			
	Phase a	Phase b	Phase c
Test 1	16075	7854	20702
Test 2	16063	7225	19911
Test 3	16088	8444	19318
Test Langzeit	16077	5032	13389

Tabelle 6.1: Experiment 1: Zeitmessung für Stabilität

5. **Auswertung** In den Listings 6.2, 6.3 und 6.4 sind die Ausgaben der *successor* jedes Knoten nach der Stabilisierung zu sehen. Deutlich erkennbar ist, dass die Knoten einen geordneten Ring bilden und somit - wie erwartet - einen stabilen Ring gebildet haben.

   In der Tabelle 6.1 sind die Stabilisierungszeiten zu sehen.

   Phase a zeigte in jedem Durchlauf ein sehr identisches Verhalten. Die Zeiten lagen leicht über den Erwartungen von 16 Sekunden, wobei die Prüfung auf Stabilität alle 200 ms läuft, somit also bis zu 200 ms vergangen sein können, seit dem der Ring stabil war.

   Phase b zeigt eine breitere Fächerung der Zeiten. Dies ist auf kleine Verzögerungen der Funktionsaufrufe jedes Knoten zurückzuführen, die zu unterschiedlichen Zuständen in jedem Interval führen können. Mit durchschnittlich 7841 ms findet die Stabilisierung deutlich schneller als erwartet (32 Sekunden) statt.

   Phase c zeigt in jedem Durchlauf ein sehr stabiles Verhalten. Mit durchschnittlich 19977 ms ist diese Phase deutlich unter den maximal erwarteten Zeiten von 30 Sekunden.

6. **Fazit** Der Ausgang von Experiment 1 entrpach den Erwartungen. Der *Chord-Ring* hat sich bei dem Hinzufügen und Entfernen von Knoten stabilisiert. Besonders hervorzuheben ist, dass die entfernten Knoten keine anderen Knoten beim Verlassen explizit benachrichtigt haben, sodass das Entfernen einem Verbindungsabbruch gleicht.

6.2 Experiment 2: Hinzufügen, Abrufen und Entfernen von Informationen

In diesem Experiment wird das Hinzufügen, Abrufen und Entfernen von Informationen in einem kontrolliert aufgebauten Chord-Ring evaluiert.

1. **Fragestellung**
   Werden die Informationen erfolgreich dem *Chord-Ring* hinzugefügt und können wieder abgerufen werden? Können die Informationen vom Ersteller gelöscht werden und werden Informationen, deren „deleteDate“ überschritten wurde automatisch gelöscht?

2. **Erwartung**
   Informationen werden richtig im Ring verteilt, gesucht und gelöscht. Experiment 1 zeigte bereits, dass der Ring sich stabilisiert. Das Löschen stellt keinen wesentlichen Unterschied zum Erstellen dar - bisauf ein Flag in den Optionen - weshalb hier auch keine Probleme erwartet werden. Automatisches Löschen von abgelaufenen Nachrichten wird im Intervall abgearbeitet und stellt keine Probleme dar.

3. **Planung**
   Wie auch schon im ersten Experiment werden hier auch wieder die bekannten 16 Knoten verwendet. Auch dieses Experiment ist wieder in mehrere Phasen unterteilt:

   0) Initialisieren und stabilisieren der 16 Knoten.

   1) Einbringen von 9 Informationen inklusive Duplikate verschiedener Ersteller (Siehe Abbildung 6.2).

   2) Löschen aller erstellten Informationen.

   3) Einbringen einer neuen Information mit Ablaufdatum („deleteDate“).

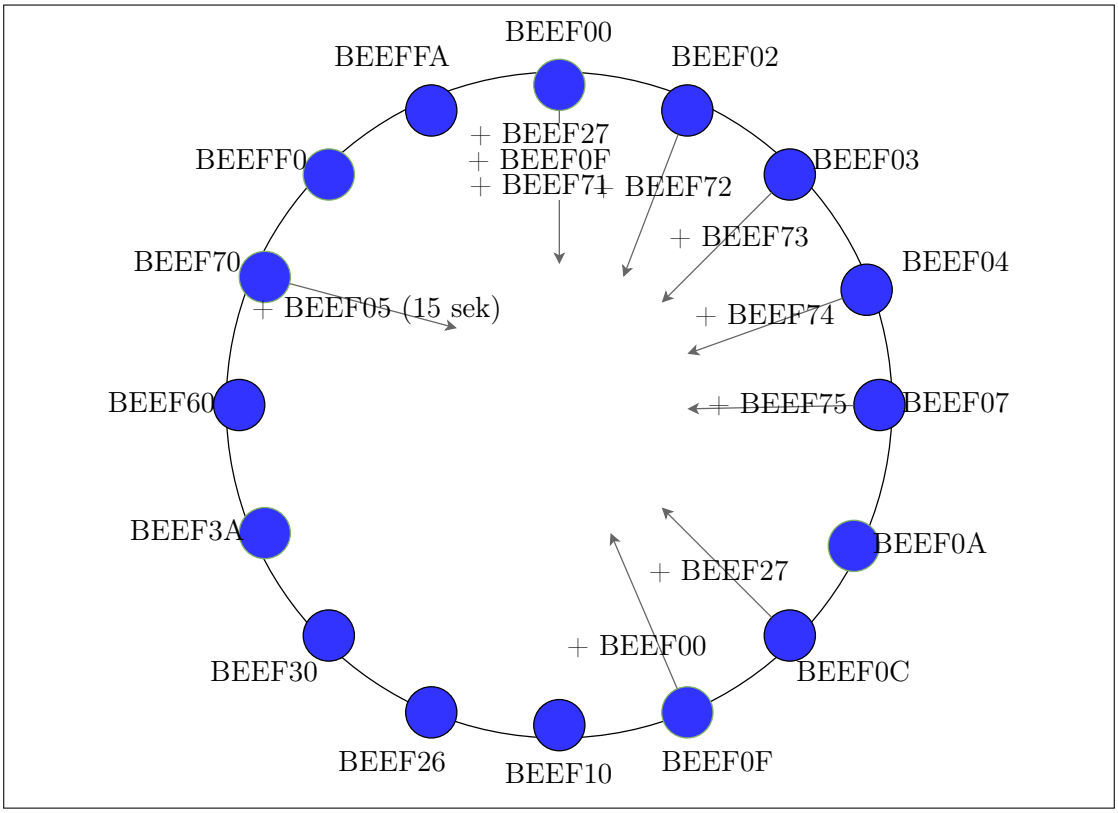

Abbildung 6.2: Planung des Rings für Experiment 2

Phase 1: Die in der Abbildung 6.2 zu erkennenden Knoten und Informationen (exkl. Knoten *BEEF70*) werden in den *Chord-Ring* eingebracht. Anschließend wird überprüft, ob die Informationen bei den logisch richtigen Knoten im *Chord-Ring* liegen. In der Tabelle 6.2 sind die Ersteller, Informationen und Zileknoten der Informationen zu sehen.

Phase 2: Alle Informationen werden von Ihren Erstellern gelöscht bzw. ein Löschauftrag versendet. Anschließend wird geprüft, ob alle Informationen tatsächlich gelöscht wurden.

Phase 3: Der Knoten *BEEF70* bringt die Information „BEEF05“ mit einem Ablaufdatum in den Ring. Das Ablaufdatum wird 15 Sekunden in der Zukunft liegen. Es wird anschließend geprüft, ob die Information bei dem Knoten *BEEF07* angekommen ist und nach Ablauf der 15 Sekunden automatisch gelöscht wird.

Creator(Hash)	Information(Hash)	Target Node(Hash)
BEEF00 (12513024)	BEEF0F (12513039)	BEEF0F (12513039)
BEEF00 (12513024)	BEEF27 (12513063)	BEEF30 (12513072)
BEEF00 (12513024)	BEEF71 (12513137)	BEEFF0 (12513264)
BEEF02 (12513026)	BEEF72 (12513138)	BEEFF0 (12513264)
BEEF03 (12513027)	BEEF73 (12513139)	BEEFF0 (12513264)
BEEF04 (12513028)	BEEF74 (12513140)	BEEFF0 (12513264)
BEEF07 (12513031)	BEEF75 (12513141)	BEEFF0 (12513264)
BEEF0C (12513036)	BEEF27 (12513063)	BEEF30 (12513072)
BEEF0F (12513039)	BEEF00 (12513024)	BEEF00 (12513024)

Tabelle 6.2: Experiment 2: Zuweisung Informationen Phase 1 + 2

4. **Beobachtungen Anmerkung:** Die Listings wurden teilweise gekürzt und eingerückt, um sie besser darstellen zu können. Die in den Listings aufgeführten Ausgaben befinden sich in chronologischer Reihenfolge. Alle Beobachtungen sind Im Anhang zu finden. Die Beobachtungen für dieses Experiment sind:

   A.7 „Experiment 2: Relevante Ausgaben aus Phase 1“ (Seite 93)

   A.1 „Experiment 2: Ausschnitt der lokalen Informationen von BEEF00“ (Seite 94)

   A.2 „Experiment 2: Ausschnitt der lokalen Informationen von BEEFF0“ (Seite 94)

   A.3 „Experiment 2: Ausschnitt der lokalen Informationen von BEEF30“ (Seite 95)

   A.8 „Experiment 2: Relevante Ausgaben aus Phase 2“ (Seite 95)

   A.4 „Experiment 2: Ausschnitt von 3 Knoten nach dem Löschen“ (Seite 95)

   A.9 „Experiment 2: Relevante Ausgaben aus Phase 3“ (Seite 96)

5. **Auswertung** In dem Listing A.7 ist zu erkennen, dass alle erstellten Informationen bei dem richtigen Knoten angekommen sind und der Knoten sie speichert. In den Abbildungen A.1, A.2 und A.3 ist beispielhaft an drei Knoten die Ausgabe der lokalen Informationen gezeigt. Knoten *BEEF00* besitzt dabei eine Information, welche als Hashwert seine eigene Knoten-ID besitzt. Knoten *BEEF* besitzt insgesamt 5 Informationen, jeweils mit unterschiedlichem Hashwert. Der Knoten *BEEF30* besitzt zwei Informationen mit demselben Hashwert, jedoch von anderen Erstellern.

   In dem Listing A.8 ist zu erkennen, dass alle informationshaltenden Knoten diese wieder gelöscht haben. In der Abbildung A.4 sind die drei eben gezeigten Knoten

erneut zu sehen, jedoch ist zu erkennen, dass nach dem Löschen alle Informationen wirklich entfernt wurden.

In dem Listing A.9 ist zu erkennen, dass der Knoten *BEEF07* die Information „BEEF05“ speichert und selbstständig löscht. Die Tatsache, dass das Löschen vor Ablauf der dargestellten 15 Sekunden passiert, lässt sich auf den vorherigen Zeitinterval zur Sicherstellung, dass die Information ankommen konnte, zurückführen.

6. **Fazit** Informationen lassen sich wie erwartet dem *Chord-Ring* hinzufügen, abrufen und entfernen. Duplikate von Informationen funktionieren unter dem Vorbehalt, dass sie von verschiedenen Erstellern sind. Außerdem werden Informationen mit einem Ablaufdatum wie erwartet automatisch entfernt.

6.3 Experiment 3: Replizierung und Verschiebung von Informationen

In diesem Experiment wird die Replizierung und Verschiebung von Informationen in einem kontrolliert aufgebauten Chord-Ring evaluiert.

1. **Fragestellung**
   Werden Informationen richtig im Ring repliziert? Und funktionieren die Mechanismen, um Informationen bei Veränderung des Rings bzw. Wegfall von Knoten zu verschieben?

2. **Erwartung**
   Informationen werden basierend auf ihrem Replizierungsindex richtig repliziert. Bei Hinzukommen von neuen Knoten werden die Informationen richtig verschoben, je nach Konstellation der hinzukommenden Knoten kann dies mehrere Zyklen dauern. Bei Wegfall von Knoten passen die direkt anliegenden Knoten ihre Replizierungsindexe richtig an.

3. **Planung**
   Dieses Experiment besteht aus insgesamt 4 Phasen. Erneut wird nach Beginn jeder Phase - nachdem die Knoten initial hinzugefügt wurden - auf Stabilität des Rings gewartet.

   In Abbildung 6.3 ist der für das Experiment 3 geplante Ring dargestellt. Die Knoten werden dabei in zwei Gruppen eingeteilt: Die grünen Knoten *(BEEF00, BEEF04, BEEF0A, BEEF0F, BEEF10, BEEF3A, BEEF70, BEEFF0)* und die blauen Knoten *(BEEF02, BEEF03, BEEF07, BEEF0C, BEEF26, BEEF60, BEEFFA)*.

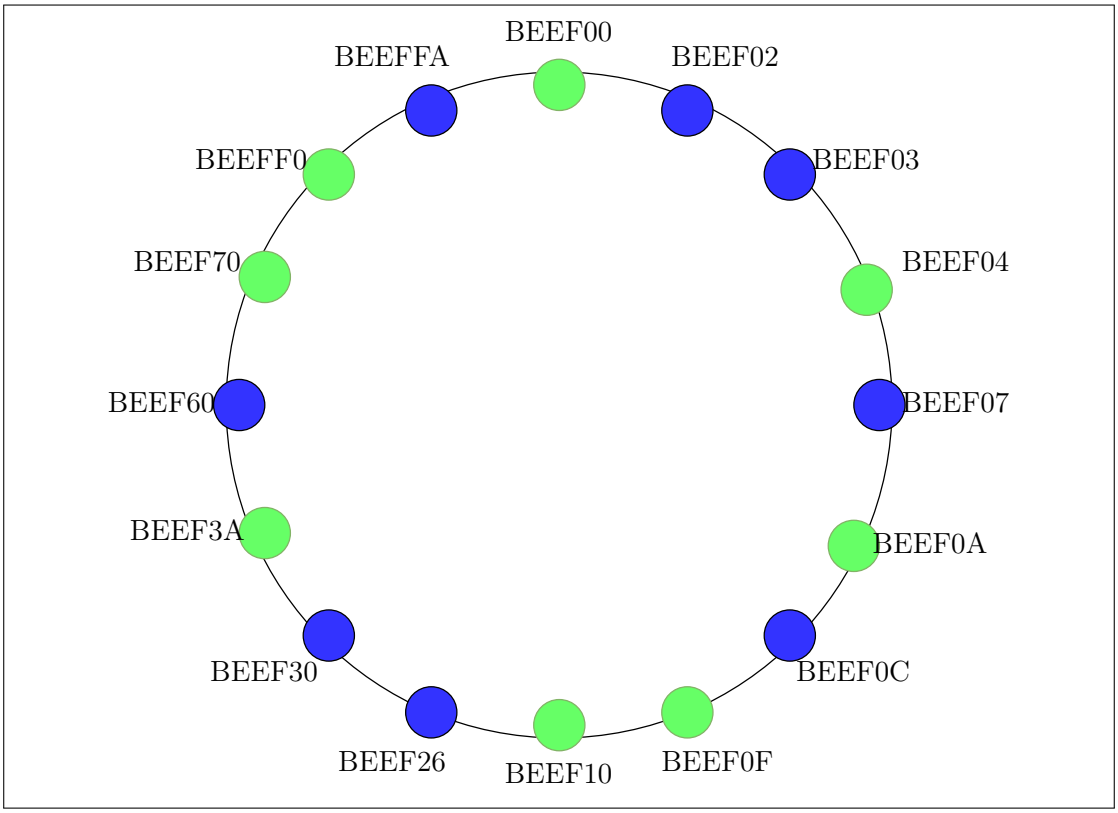

Abbildung 6.3: Planung des Rings für Experiment 3

In Phase 1 geht es darum, zu prüfen, ob eine Information in einem Ring richtig repliziert wird. Der geplante Aufbau für Phase 1 ist in Abbildung 6.4 zu sehen. Dabei wird die Information „BEEF11“ von dem Knoten *BEEF00* mit einem Replizierungsfaktor von 2 erstellt. Die Information wird daraufhin logisch dem Knoten *BEEF3A* zugeordnet. Dieser gibt die Information mit einem um 1 inkrementierten Replizierungsindex an seinen *successor BEEF70* weiter. Dieser wiederholt das Prozedere mit Knoten *BEEFF0*. Zuletzt schickt der Knoten *BEEFF0* die Information mit einem Replizierungsindex von -1 an seinen *successor BEEF00* weiter.

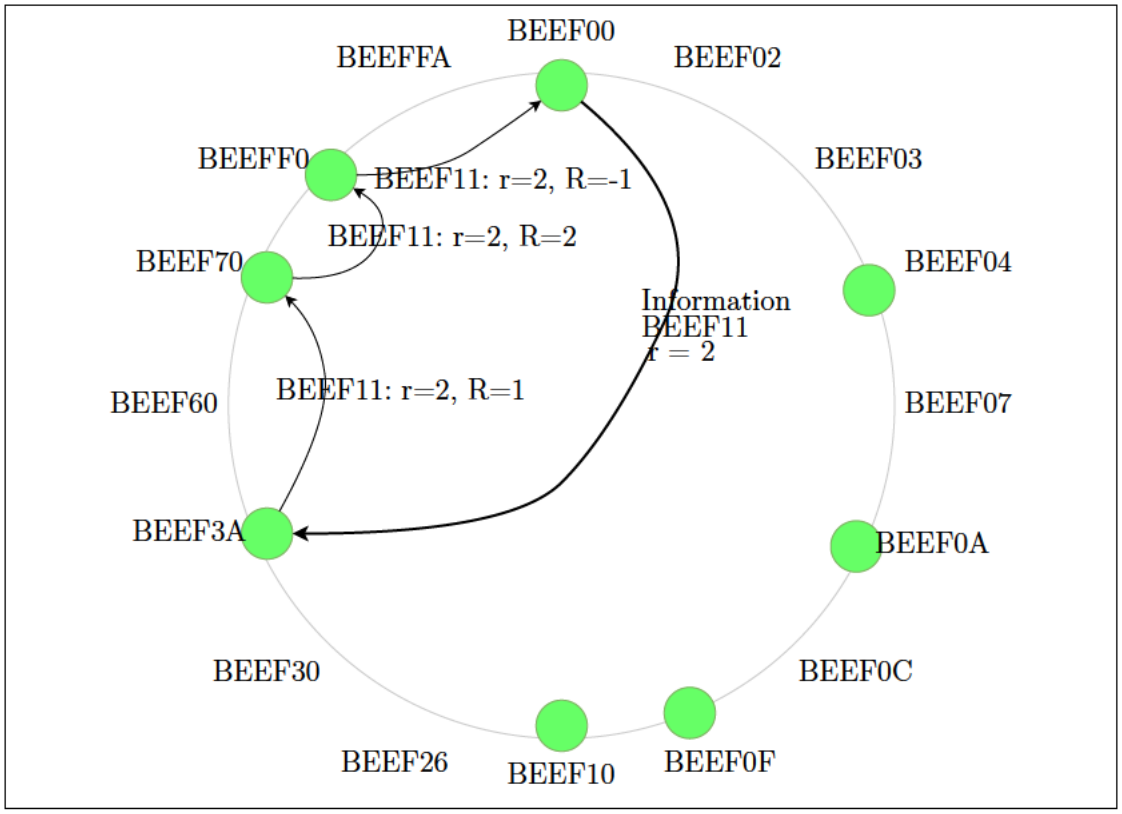

Abbildung 6.4: Experiment 3: Phase 1

Phase 2 baut auf Phase 1 auf. Das heißt, dass die Phase 1 vor der Phase 2 durchlaufen wird, um so die Information in den Ring einzubringen. Danach soll der Knoten der initial für die Information zuständig war entfernt werden. Anschließend soll die Information von dem logischen *successor* des entfernten Knotens bereitgestellt werden. In Abbildung 6.5 ist der geplante Ablauf zu sehen. Der Knoten *BEEF3A* und die bei ihm gespeicherte Information „BEEF11“ werden aus dem Ring entfernt. Logischer *successor* ist nun der Knoten *BEEF70*. Dieser soll die Information nun zur Verfügung stellen. Gleichzeitig soll der Knoten *BEEF70* den Wegfall seines *predecessor*s erkennen, sich selber als neuen Wurzelknoten der Information identifizieren und die Information wieder im Ring stabilisieren.

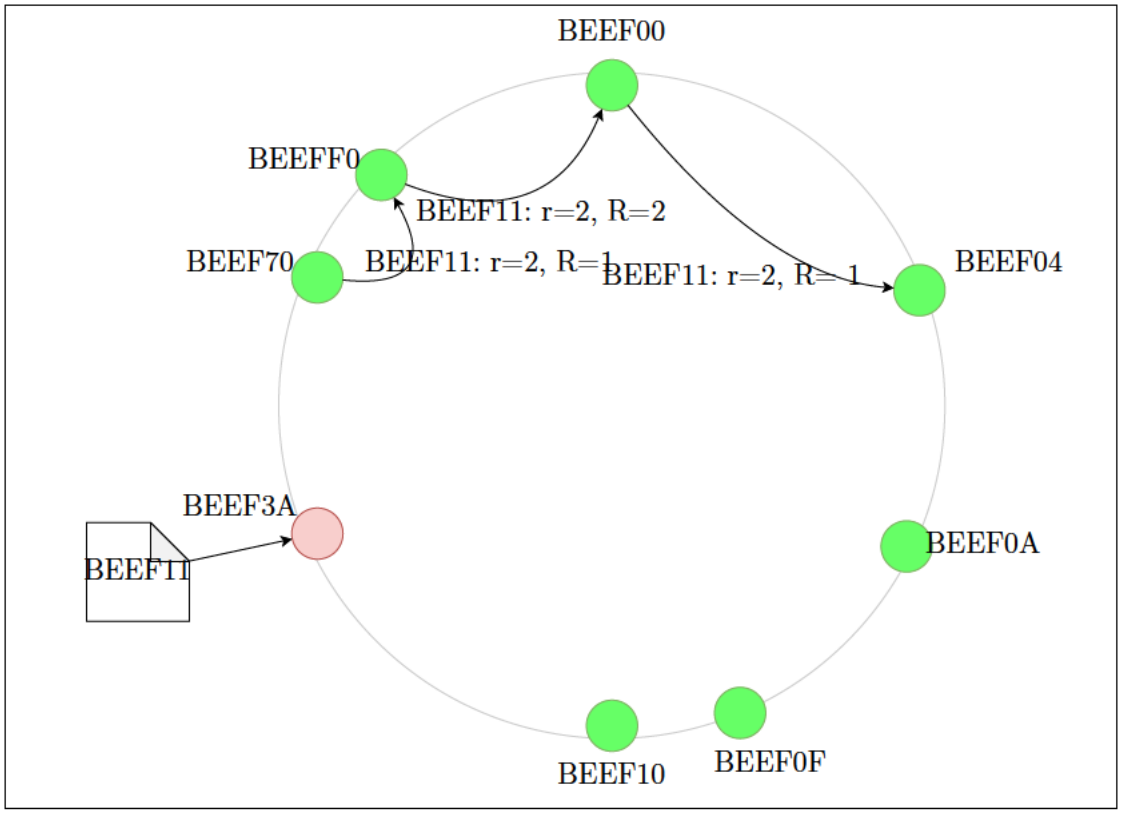

Abbildung 6.5: Experiment 3: Phase 2

Phase 3 baut erneut auf Phase 1 auf. Nachdem die Information in dem Ring der grünen Knoten gespeichert wurde, werden die blauen Knoten hinzugefügt. Anschließend soll der Knoten *BEEF3A* erkennen, dass es passendere Knoten für die Information gibt und eine Stabilisierung der Information einleiten. Ziel ist es, dass der Knoten *BEEF26* der neue Wurzelknoten der Information ist, *BEEF30* das erste und *BEEF3A* das zweite Replikat speichert.

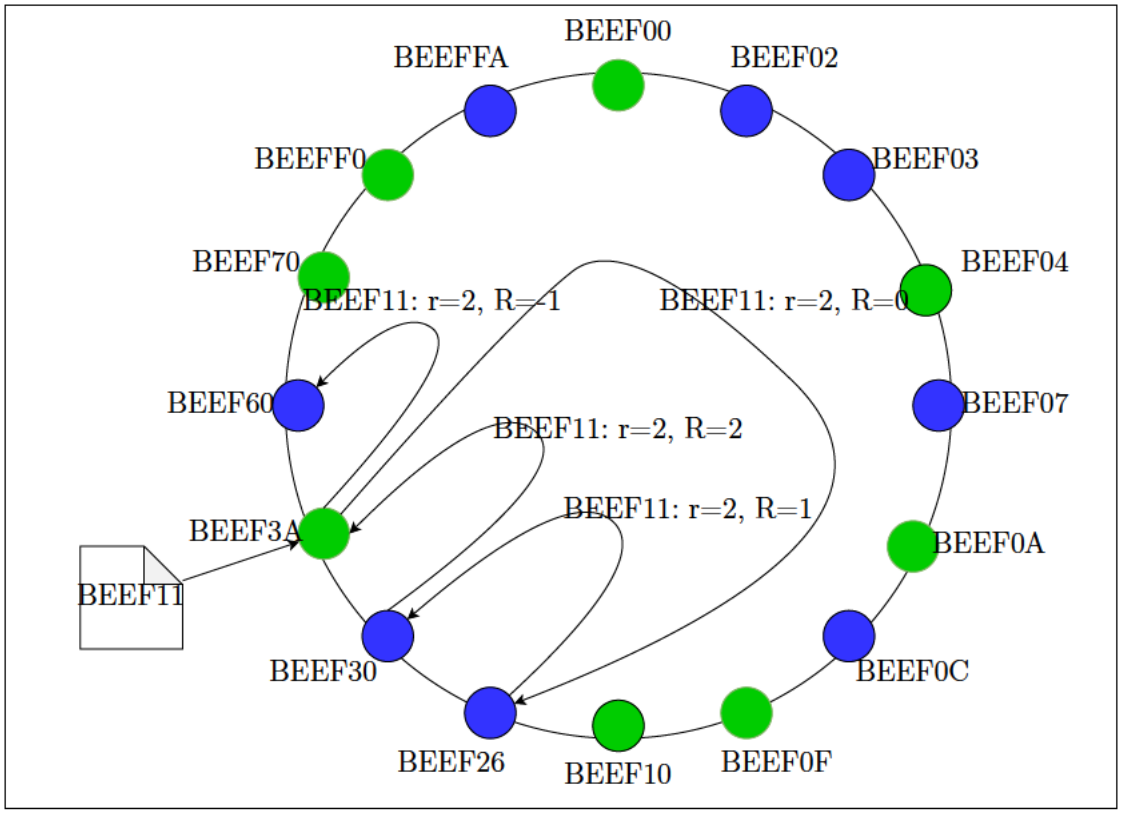

Abbildung 6.6: Experiment 3: Phase 3

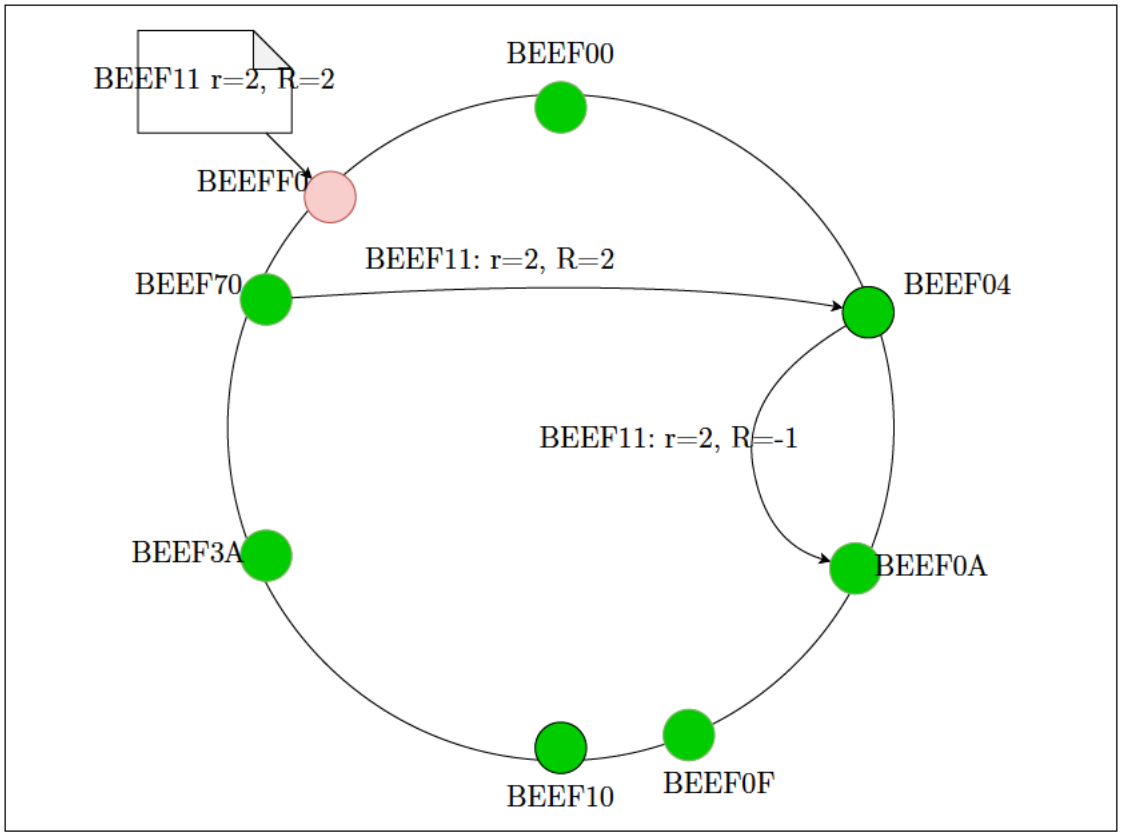

Abbildung 6.7: Experiment 3: Phase 4

4. **Beobachtungen**
   Anmerkung: Die Listings wurden teilweise gekürzt und eingerückt, um sie besser darstellen zu können. Die in den Listings aufgeführten Ausgaben befinden sich in chronologischer Reihenfolge. Alle Beobachtungen sind Im Anhang zu finden. Die Beobachtungen für dieses Experiment sind:

   A.10 „Experiment 3: Relevante Ausgaben aus Phase 1“ (Seite 96)

   A.11 „Experiment 3: Relevante Ausgaben aus Phase 2“ (Seite 96)

   A.5 „Experiment 3: Ausschnitt der Abfrage von Informationen“ (Seite 97)

   A.12 „Experiment 3: Relevante Ausgaben aus Phase 3“ (Seite 97)

5. **Auswertung**
   Phase 1 Die in dem Listing A.10 dargestellten Ausgaben zeigen deutlich, dass

die Information wie erwartet repliziert wurde. Die Information wurde zuerst mit einem Replizierungsindex R von 0 und einem Replizierungsfaktor r von 2 bei dem Knoten *BEEF3A* gespeichert. Anschließend wurde die Information mit einem um 1 erhöhten Replizierungsindex bei dem Knoten *BEEF70* und zuletzt mit einem Replizierungsindex von 2 bei dem Knoten *BEEFF0* gespeichert. Der Knoten *BEEF00* erhielt die Information mit einem Replizierungsindex von -1, was für ihn bedeuten soll, dass die Information bereits maximal oft repliziert wurde und er sie, sofern er sie besitzt, lokal löschen darf. Dies hat der Knoten auch ausgegeben - Information lag lokal nicht vor, aber das Verhalten war richtig - und somit die Phase 1 erfolgreich abgeschlossen.

Phase 2 In dem Listing A.11 sind die relevanten Ausgaben aller beteiligter Knoten nach dem Verlassen des Knotens *BEEF3A* zu sehen. Der Knoten *BEEF10* erkannte erfolgreich, dass sein direkter *successor* (Knoten *BEEF3A*) nicht mehr vorhanden ist und stabilisierte den Ring wieder. Der Knoten *BEEF70* erkannte erfolgreich, dass sein *predecessor* nicht mehr vorhanden und er nun der neue Wurzelknoten der Information ist. Anschließend stabilisierte der Knoten *BEEF70* die Information wieder im Ring.

In der Abbildung A.5 ist der Abruf der Information nach dem Verlassen des Knoten zu sehen. Der Knoten *BEEF0F* konnte erfolgreich die Information abrufen. An der Information ist zu erkennen, dass sie von dem *predecessor* des Knotens *12513264* (dezimal von BEEF70) kommt und dieser Knoten sich als Wurzelknoten der Information sieht (options.backupdepth.idx ist 0).

Phase 2 ist somit erfolgreich abgelaufen wie erwartet.

Phase 3 Die in dem Listing A.12 dargestellten Ausgaben zeigen deutlich, dass die Information nach dem Eintritt der blauen Knoten wie erwartet stabilisiert wurde. Der Knoten *BEEF3A* repliziert die Information zu seinem neuen *predecessor* *BEEF30*, welcher erkennt, dass sein *predecessor*, Knoten *BEEF26*, besser geeignet ist. Der Knoten *BEEF26* erhält die Information nun als neuer Wurzelknoten und stabilisiert diese wie erwartet. Der Knoten *BEEF60* erhält zwei mal die Nachricht, dass er der erste Knoten nach vollständiger Stabilisierung sei, was darauf zurückzuführen ist, dass der Knoten *BEEF3A* diese Nachricht einmal nach Erhalt seiner Replik und einmal aufgrund des Wechsels seines *successor* verschickt. Phase 3 ist genau wie erwartet abgelaufen.

7 Fazit

Eine modulare *Middleware* für *Chord* über *WebRTC* war das Ziel und wurde auch umgesetzt. Das fragile Umfeld von *WebRTC* für ein dezentrales Informationssystem zu nutzen stellte sich im allgemeinen als schwierig dar. Viele Kontrollstrukturen mussten geschaffen werden. Die reine Definition des Ablaufs - wie der *Chord*-Ring stabilisiert wird - musste um Funktionalitäten wie z.B. die Replizierung von Informationen erweitert werden. Verbindungsabbrüche können von verschiedenen Stellen gemeldet werden. Auf noch nicht verarbeitete Nachrichten muss eingegangen werden. Der Wegfall von Knoten darf keine Auswirkungen auf Anfragen haben. Diese sind die Geschehnisse und Funktionalitäten, die es zu bewältigen gab.

Die Aufteilung dieser Kontrollstrukturen in mehrere Module erlaubt es, kontrollierter mit den Fehlern umzugehen. Außerdem können an vielen Stellen durch die Nutzung der *Interfaces* Kontrollstrukturen oder gleich ganze Module ersetzt werden.

Die Definition einer einheitlichen Struktur für alle Nachrichten - über alle Module hinweg - erlaubt einfachen Datenaustausch. Das Nachrichtenformat ist dabei Menschen-lesbar im *JSON*-Format.

Die Experimente haben alle genannten Eigenschaften getestet. Dabei zeigte sich, dass „ChordComm“ sehr Robust ist. Der *Chord*-Ring stabilisiert sich wie erwartet. Informationen werden richtig Repliziert. Bei Wegfall von Knoten bleiben *Chord*-Ring und Informationen stabil.

„ChordComm“ zeigt, dass sich ein dezentrales Informationssystem auf *Chord*-Basis über *WebRTC* mit modularen Komponenten umsetzen lässt.

Literaturverzeichnis

[1] Eng Keong Lua, Marcelo Pias Ravi S. ; Lim, Steven: A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. (2004)

[2] Frank Dabek, David Karger Robert Morris Ion S.: Wide-area cooperative storage with CFS. (2001). – URL http://pdos.lcs.mit.edu/chord

[3] Ion Stoica, David Karger M. Frans Kaashoek Hari B.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. (2001). – URL https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

[4] Karsten, Maxwell Young Aniket Kate Ian Goldberg M.: Practical Robust Communication in DHTs Tolerating a Byzantine Adversary.

[5] Labiod, Salma Ktari ; Artur Hecker ; H.: Empowering Chord DHT overlays. (2009). – URL https://ieeexplore.ieee.org/abstract/document/5307418

[6] Matthew Leslie, Todd H.: A Comparison Of Replication Strategies for Reliable Decentralised Storage. (2006)

[7] Shay Gueron, Jesse W.: SHA-512/256. (2011). – URL https://ieeexplore.ieee.org/abstract/document/5945260. ISBN 978-1-61284-427-5

[8] Steen, M. van ; Tanenbaum, A.S.: *Distributed Systems, 3rd ed.* URL distributed-systems.net, 2017. – ISBN 978-90-815406-2-9

[9] T. Peer Meera Labbai, S.Jothi P.: T2WSN: TITIVATED TWO-TIRED CHORD OVERLAY AIDING ROBUSTNESS AND DELIVERY RATIO FOR WIRELESS SENSOR NETWORKS. (2016). – URL http://www.jatit.org/volumes/Vol91No1/18Vol91No1.pdf

[10] Vijay Gopalakrishnan, Bobby Bhattacharjee Pete K.: Adaptive Replication in Peer-to-Peer Systems. (2004)

[11] W3C: WebRTC 1.0: Real-time Communication Between Browsers. (27 September 2018). – URL https://www.w3.org/TR/webrtc/

A Anhang

A.1 Implementation

A.1.1 CommunicationHandler: Information speichern

```
addInformationLocal(information : Information) : void {
    let infoCpy = information;
    infoCpy.sucessornode = this.fingerTable.table[0].peerID.toString();
    let localDuplicate = this.localInformations.get(information.hash);
    ...
    if(localDuplicate) {
        if(information.options.delete && information.options.delete == true)
    {
            localDuplicate.delete(BigInt(information.creator));

            if(localDuplicate.size == 0) {
                this.localInformations.delete(information.hash);
            }
            return;
        }
        ...
        let localEntry = localDuplicate.get(BigInt(information.creator));
        if(localEntry) {
            if(localEntry.datecreated < information.datecreated) {
                localDuplicate.delete(BigInt(information.creator));
                localDuplicate.set(BigInt(information.creator), infoCpy);
            } else if(localEntry.datecreated == information.datecreated) {
                let infoOpt = information.options;
                if(infoOpt.backupdepth && localEntry.options.backupdepth) {
                    localEntry.options.backupdepth.idx = infoOpt.backupdepth.
    idx;
                }
            }
            return;
```

```
28          }
29          localDuplicate.set(BigInt(information.creator), infoCpy);
30      } else if(information.options.delete == undefined || information.options.
    delete == false){
31          let newInfoMap : Map<bigint, Information> = new Map<bigint,
    Information>();
32          newInfoMap.set(BigInt(information.creator), infoCpy);
33          this.localInformations.set(information.hash, newInfoMap);
34      }
35  }
```

Listing A.1: Implementation CommunicationHandler: handle information

A.1.2 ChordClient: Stabilisieren des Rings

```
1  async stabilize() {
2      let successor = this.fingerTable.table[0].peerID;
3
4      if(successor == this.ownID && successor == this.predecessor) { return; }
5
6      let n0;
7      if(successor != this.ownID) {
8          n0 = this.getStubforID(successor);
9      } else {
10         n0 = this;
11     }
12
13     if(n0) {
14         let x;
15         try {
16             if(successor == this.ownID) {
17                 x = this.predecessor;
18             } else {
19                 x = await n0.get_predecessor();
20             }
21
22             if(x && x >= 0) {
23                 if(this.ownID == successor || boundsEXEX(x, this.ownID,
    successor)) {
24                     let nx = this.getStubforID(x);
25
26                     if(nx && !nx.getActive()) {
27                         n0.notify(this.ownID);
28                         return;
```

```
29                      }
30                      this.fingerTable.table[0].peerID = x;
31                      this.fingerTable.table[0].lastUpdate = (new Date()).
    getTime();
32                      ...
33                      if(!this.hasStubforID(x)) {
34                          this.addStub(x);
35                      }
36                      n0 = this.getStubforID(x);
37                  }
38              }
39
40              if(n0) {
41                  n0.notify(this.ownID);
42              }
43          } catch(err) {
44              this.fingerTable.fixSuccessor(successor);
45              this.removeStub(successor);
46          }
47      }
48      ...
49  }
```

Listing A.2: Implementation ChordClient: stabilize

ChordClientStub: Verarbeitung von Remote-Calls

```
1  async find_successor(key : bigint) : Promise<bigint> {
2      let ident : bigint = this.ch.getIdentifier();
3
4      let msg_call : MSG_CALL = {
5          calltype : MSG_CALL_TYPES.FIND_SUCCESSOR,
6          identifier : ident.toString(),
7          value : key.toString()
8      };
9
10     let msg : MSG_BODY = {
11         to : this.remotePeerID.toString(),
12         from : this.localPeerID.toString(),
13         type : MSG_TYPES.CALL,
14         data : JSON.stringify(msg_call)
15     };
16
17     this.runningCalls++;
```

```
18      let answer;
19
20      try {
21          answer = await this.ch.sendSync(JSON.stringify(msg), this.
        remotePeerID, ident);
22          let ans : MSG_BODY = JSON.parse(answer);
23          let result : MSG_RES = JSON.parse(ans.data);
24          this.runningCalls--;
25
26          if(result.state && result.state == MSG_STATE.ERROR) {
27              this.activeState = false;
28              return Promise.reject("Peer is no longer Connected to Chord-Ring"
        );
29          } else {
30              return BigInt(result.value);
31          }
32      } catch(error) {
33          return Promise.reject("Peer is no longer Connected to Chord-Ring");
34      }
35  }
```

Listing A.3: Implementation ChordClient: remote calls

A.1.3 Verwendungsbeispiel: Information in den Ring einbringen

```
1  async setInformationRemote(information : string, tag : string, fixHash :
       boolean = false, options? : InformationOptions) : Promise<void> {
2       const informationobject : Information = InformationFactory.
       createInformation(
3           tag,
4           this.localPeerID.toString(),
5           information,
6           this.tablesize,
7           fixHash
8       );
9       ...
10      let pre = await this.chordClient.get_predecessor(true);
11      let target : bigint = this.getClosestPreceedingOrExact(BigInt(
       informationobject.hash));
12
13      if(boundsEXIN(target, pre, this.localPeerID) && pre != BigInt(-1)) {
14          this.addInformationLocal(informationobject);
15          return;
16      }
```

```
17      ...
18
19      let msg_info : MSG_SETINFORMATION = {
20          information : informationobject
21      };
22
23      let msg_body : MSG_BODY = {
24          to : target.toString(),
25          from : this.localPeerID.toString(),
26          type : MSG_TYPES.SETINFORMATION,
27          data : JSON.stringify(msg_info)
28      };
29
30      this.sendAsync(JSON.stringify(msg_body), target);
31  }
```

Listing A.4: Implementation MSG: add information

A.1.4 Prüfung der Stabilität des Rings

```
1  async function checkStable() : Promise<boolean> {
2      let first : string | undefined;
3      let previous;
4      let last;
5      let error = false;
6
7      for(const k of Object.keys(beefs)) {
8          if(watched[k]) {
9              if(!first) {
10                 first = k;
11                 previous = k;
12                 continue;
13             }
14
15             if(!previous) return false;
16             let ftp = beefs[previous].fingerTable;
17             let ftk = beefs[k].ownID;
18             if(ftp && ftk && ftp.table[0].peerID != ftk) {
19                 error = true;
20             }
21             previous = k; last = k;
22         }
23     }
24
```

```
25      if(last && first) {
26          let ftp = beefs[last].fingerTable;
27          let ftk = beefs[first].ownID;
28          if(ftp && ftk && ftp.table[0].peerID != ftk) {
29              error = true;
30          }
31      } else {
32          error = true;
33      }
34      ...
35      return !error;
36  }
```

Listing A.5: Implementation Experimente: check ring stability

A.1.5 Beispiel: Peer-Definitionen Experiment 3

```
1  var clientPrefix =
2      "0000000000" +
3      "0000000000" +
4      "0000000000" +
5      "0000000000" +
6      "0000000000" +
7      "00000000" +
8      "BEEF";
9
10 var clientIds : string[] = [
11     "00", "02", "03", "04",
12     "07", "0A", "0C", "0F",
13     "10", "26", "30", "3A",
14     "60", "70", "F0", "FA"
15 ];
16 ...
17 for(let i = 0; i < clientIds.length; i++) {
18     let peerIdStr = parseHexString(clientPrefix + clientIds[i]);
19     let peerId = digestToBigInt(peerIdStr);
20
21     let fullClient = new FullClient(
22         peerId,
23         new CommunicationOptions({}),
24         conHandleFactpryCB,
25         conHandleHelloCB
26     );
27
```

```
28      await fullClient.init({serveraddress : "ws://localhost:8999/socketserver"
        });
29
30      if(fullClient.commHandler)
31          localPeerId = await fullClient.commHandler.getLocalPeerID();
32
33      fullClients.set(peerId, fullClient);
34      beefs["BEEF" + clientIds[i]] = fullClient;
35      ...
36  }
37  ...
38  watched["BEEF00"] = beefs["BEEF00"];
39  beefs["BEEF00"].create("Experiment1");
40  if(beefs["BEEF00"].chordClient && beefs["BEEF00"].chordClient.
        startStabilizationProtocol)
41      beefs["BEEF00"].chordClient.startStabilizationProtocol();
42  await delay(250);
43
44  watched["BEEF04"] = beefs["BEEF04"];
45  beefs["BEEF04"].connect("Experiment1");
46  if(beefs["BEEF04"].chordClient && beefs["BEEF04"].chordClient.
        startStabilizationProtocol)
47  beefs["BEEF04"].chordClient.startStabilizationProtocol();
48  ...
```

Listing A.6: Implementation Experimente: experiment 3

A.2 Experimente

A.2.1 Experiment 2: Beobachtungen

Das Listing A.7 zeigt die Ausgaben der Knoten beim Speichern der Informationen.

```
1 [12513024]: saving information(000...000BEEF00:12513024)
2 [12513072]: saving information(000...000BEEF27:12513063)
3 [12513039]: saving information(000...000BEEF0F:12513039)
4 [12513072]: saving information duplicate (000...000BEEF27:12513063)
5 [12513264]: saving information(000...000BEEF75:12513141)
6 [12513264]: saving information(000...000BEEF74:12513140)
7 [12513264]: saving information(000...000BEEF73:12513139)
8 [12513264]: saving information(000...000BEEF72:12513138)
9 [12513264]: saving information(000...000BEEF71:12513137)
```

Listing A.7: Experiment 2: Relevante Ausgaben aus Phase 1

In Abbildung A.1 sind die Lokalen Informationen von *BEEF00* zu sehen.

Abbildung A.1: Experiment 2: Ausschnitt der lokalen Informationen von BEEF00

In Abbildung A.2 sind die Lokalen Informationen von *BEEFF0* zu sehen.

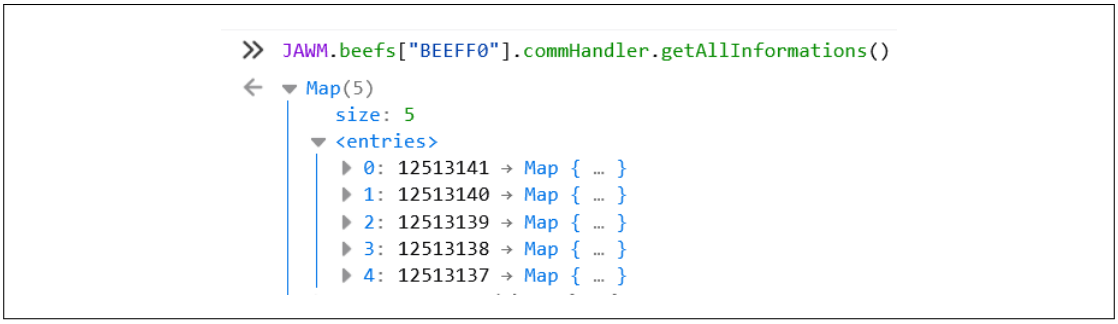

Abbildung A.2: Experiment 2: Ausschnitt der lokalen Informationen von BEEFF0

In Abbildung A.3 sind die Lokalen Informationen von *BEEF30* zu sehen.

Abbildung A.3: Experiment 2: Ausschnitt der lokalen Informationen von BEEF30

Das Listing A.8 zeigt die Ausgaben der Knoten beim Speichern der Informationen.

```
[12513024] deleting local information '12513024'
[12513072] deleting local information '12513063'
[12513039] deleting local information '12513039'
[12513072] deleting local information '12513063'
[12513264] deleting local information '12513141'
[12513264] deleting local information '12513140'
[12513264] deleting local information '12513139'
[12513264] deleting local information '12513138'
[12513264] deleting local information '12513137'
```

Listing A.8: Experiment 2: Relevante Ausgaben aus Phase 2

In Abbildung A.4 sind die lokalen Informationen von den vorherigen 3 Knoten nach dem Löschen zu sehen.

Abbildung A.4: Experiment 2: Ausschnitt von 3 Knoten nach dem Löschen

Das Listing A.9 zeigt die Ausgaben der Ausführung von Phase 3

```
[12513031]: saving information(000...000BEEF05:12513029)
Map { 12513029 -> Map(1) }
Waiting 15 seconds...
[12513031]: deleting 1
... 15 seconds passed!
Map(0)
```

Listing A.9: Experiment 2: Relevante Ausgaben aus Phase 3

A.2.2 Experiment 3: Beobachtungen

In dem Listing A.10 sind die Relevanten Konsolenausgaben der Phase 1 zu sehen.

```
[Phase 1] stable after '16077ms'

[beef00] adding information to Ring (000...000BEEF11)

[beef3a] received data: '000...000BEEF11': 0/2 (R/r)
[beef3a] saving information(000...000BEEF11:beef11)

[beef70] received replication data: '000...000BEEF11': 1/2 (R/r)
[beef70] saving information(000...000BEEF11:beef11)

[beeff0] received replication data: '000...000BEEF11': 2/2 (R/r)
[beeff0] saving information(000...000BEEF11:beef11)

[beef00] received replication data: '000...000BEEF11': -1/2 (R/r)
[beef00] removing local replicate!
```

Listing A.10: Experiment 3: Relevante Ausgaben aus Phase 1

In dem Listing A.11 sind die Relevanten Konsolenausgaben der Phase 2 zu sehen.

```
[beef10] catched stabilization error 'beef3a':
>> "Peer is no longer Connected to Chord-Ring"
[beef10]: new successor 'beef70'

[Phase 2] stable after '1635ms'

[beef70] error retrieving predecessor.predecessor
[beef70] change replication index to 0 as predecessor changed
[beef70] replicating information to successor 'beeff0'
```

```
10
11 [beeff0] received replication data: '000...000BEEF11': 1/2 (R/r)
12
13 [beef00] received replication data: '000...000BEEF11': 2/2 (R/r)
14 [beef00] saving information(000...000BEEF11:beef11)
15
16 [beef04] received replication data: '000...000BEEF11': -1/2 (R/r)
17 [beef04] removing local replicate!
```

Listing A.11: Experiment 3: Relevante Ausgaben aus Phase 2

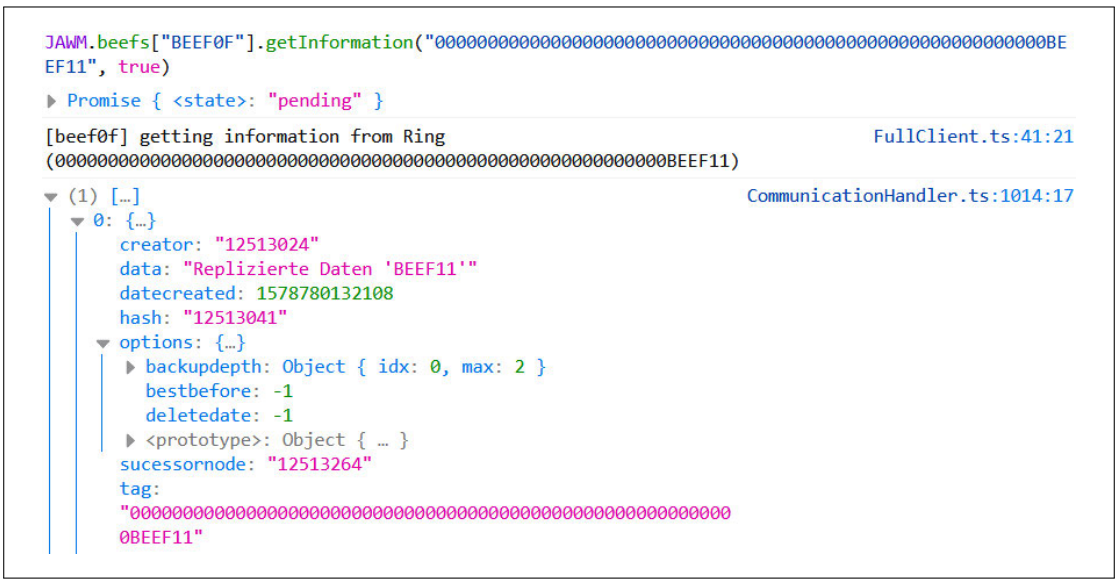

Abbildung A.5: Experiment 3: Ausschnitt der Abfrage von Informationen

In dem Lisiting A.12 sind die relevanten Konsolenausgaben der Phase 3 zu sehen.

```
1 [Phase 3] stable after '6837ms'
2
3 [beef3a] replicating information to predecessor 'beef30'
4
5 [beef26] received data: '000...000BEEF11': 0/2 (R/r)
6 [beef26] saving information(000...000BEEF11:beef11)
7
8 [beef30] received replication data: '000...000BEEF11': 1/2 (R/r)
9 [beef30] saving information(000...000BEEF11:beef11)
10
11 [beef3a] received replication data: '000...000BEEF11': 2/2 (R/r)
```

```
12 [beef3a] updating replication
13
14 [beef60] received replication data: '000...000BEEF11': -1/2 (R/r)
15 [beef60] removing local replicate!
16
17 [beef3a] replicating information to new successor 'beef60'
18
19 [beef60] received replication data: '000...000BEEF11': -1/2 (R/r)
20 [beef60] removing local replicate!
```

Listing A.12: Experiment 3: Relevante Ausgaben aus Phase 3

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Chord-Comm: Eine modulare TypeScript Middleware für Chord über WebRTC.

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort | Datum | Unterschrift im Original