Implementierung und Evaluierung von Netzwerkpartitionierungen und Mergingalgorithmen in Peer-to-Peer Overlays

Bachelorarbeit

von

Ilham Amara

aus

Segangan

vorgelegt am

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

März 2015

Betreuer:
Tobias Amft M.Sc

Abstract

Peer-to-Peer-Systeme generieren heutzutage einen großen Teil des Datenverkehrs im Internet, da sie von vielen Benutzer verwendet werden. Dabei werden diese Systeme hauptsächlich zum Datenaustausch, zum sicheren Speichern von Dateien oder zur Kommunikation innerhalb von kleinen organisierten Gruppen eingesetzt. Die Teilnehmer (Peers) in einem Peer-to-Peer-System bilden eine logische Overlay-Kommunikationsstruktur über eine physisches Netzwerk (dem sogenannten Underlay). Der größte Vorteil der Peer-to-Peer-Systeme, im Gegensatz zu Client-Server-Systemen, ist die gute Skalierbarkeit zur Unterstützung einer gro/ssen Anzahl von teilnehmenden Overlay-Knoten und der Verzicht auf zentrale Server. Somit ist die Verfügbarkeit der Daten meistens redundant, da jeder Knoten zusätzliche Ressourcen bietet. Dadurch sind Peer-to-Peer-Systeme relativ robust und resistent gegen Ausfall einiger Overlay-Knoten. Aus diesem Grund sind solche Systeme gut geeignet, um freien Imformationsaustausch zu realisieren und damit Meinungsfreiheit auszubreiten. Da Speicherung und Austausch der Dateien nicht kontrolliert werden können, sind Peer-to-Peer-Systeme besonders resistent gegen sämtliche Zensurmaßnamen.
In Ländern, in denen keine Demokratie herrscht und somit die Meinungs und Pressefreiheit stark zensiert wird, kann Internet zeitweise abgeschaltet werden. Dadurch kann in Peer-to-Peer-Systemen die Partitionierung des Underlays sowie des Overlays hervorgerufen werden, so dass die übliche Kommunikation, zwischen den Teilnehmern zeitweise nicht möglich ist, selbst dann, wenn Konnektivität wieder hergestellt wurde.

Die vorliegende Arbeit beschäftigt sich zunächst mit der Realisierung der Netzwerkpartitionierung im Underlay, d. h. die Bildung von isolierten Knotengruppen bzw. Regionen, die nicht miteinander kommunizieren können, obwohl die Konnektivität innerhalb der Regionen besteht. Dabei wird ein Konzept für das Underlay entworfen, das mit Hilfe der eingeführten Szenarioeingabedateien verschiedene Anzahl an Knotengruppen mit verschiedenen Gruppennnamen bildet. Mit diesem Konzept wird die Konnektivität zwischen unterschiedlichen Knotengruppen Gruppen (bzw. Regionen) zeitweise eingeschränkt. Die Effekte dieser Netzwerkeinschränkung auf die isolierten Regionen werden analysiert, simuliert und bewertet. Der zweite Teil dieser Arbeit befasst sich mit den benötigten Verfahren, um diese getrennten Overlay-Partitionen zu verschmelzen. Dabei werden bekannte Verfahren bzw. Algorithmen im Overlay implementiert und evaluiert.

Gehe nicht, wohin der Weg führen mag, sondern dahin, wo kein Weg ist, und hinterlasse eine Spur.
Jean Paul

Danksagung

Diese Bachelorarbeit widme ich von ganzen Herzen meinem Mann Salah und meinen Söhne Iyas und Rayan El Hamrouni.
Danke meine Kinder für das Verständnis , die Unterstützung und die guten Leistungen in der Schule, welche mich während der Bachelorarbeit motiviert hat. Ich liebe euch!
Danke Salah, dass du mich ausnahmslos unterstützt, ermutigst, und viel von meiner Aufgaben übernommen hast, damit ich mein Ziel erreiche. Ohne dich wäre das Studium nicht möglich.
Ganz besonderer Dank gilt meinem Betreur, Tobias Amft, für die geduldige und hervorragende Begleitung.Vielen Dank, dass du an mich geglaubt hast und dieses Interessante Thema angeboten hast. Du bist ein Glück für diesen Lehrstuhl.

Aus meinem Freundeskreis danke ich Shadia Shahzad und Hümeyra Yilmaz für die Unterstützung.
Des Weiteren bedanke ich mich bei Frau und Herr Willecke, die sich viel Zeit genommen haben für das Korrekturlesen.

Meinen Eltern und Schwestern in Marokko möchte ich an dieser Stelle auch danken für Ihre emotionale Unterstützung.

Inhalt

Abbildungen

Tabellen

Kapitel 1

Einleitung

1.1 Motivation

schon vor dem Internetzeitalter hat der kanadische Medientheoretiker Marshall McLuhan die Welt als Dorf vorgestellt [1]. Die klassischen Medien darunter das One-to-One-Medium (zum Beispiel die klassische Post, Telegramm und Telefon) und die One-to-Many-Massenmedien haben dabei eine große Rolle gespielt, um Distanzen zwischen den Menschen überall auf der Welt zu minimieren und damit die ersten Bausteine für die moderne Globalisierung vorgelegt. Heutzutage im Internetzeitalter spielen Raum und Zeit tatsächlich keine Rolle mehr.

Die Vorteile, die das Internet mit sich gebracht hat, besonders für die Länder, die nach Demokratie und Meinungsfreiheit streben, haben die autoritären Regime gefordert, die Kontrolle darüber zu verschärfen. Allerdings hat die Zensur als Kommunikationskontrolle auch Wurzeln in der Geschichte. Schon im Mittelalter, im Römischen Reich und in der Islamischen Welt, wurden die meisten Werke der ”Häretiker” in der Öffentlichkeit verbrannt.

Die Erfindung des Drucks Mitte des 15. Jahrhunderts durch Johannes Gutenberg hat die Überwachung der gesellschaftlichen Kommunikation überfordert, denn hunderte bzw. tausende Exemplare konnten von jedem Text hergestellt und verbreitet werden. Dafür wurden 1559 neue Kontrollmaßnahmen eingeführt und ein Index "librorum prohibitorum", der Index der verbotenen Bücher, ausgestellt [2].

Weil die Meinungsfreiheit immer noch als Bedrohung und Gefahr angesehen wird, sowohl von den Machthabern als auch von fundamentalistischen Gruppierungen in den demokratischen Ländern, "Charlie Hebdo", das französische Satire-Magazin als Beispiel, werden in vielen Staaten die Übergangspunkte zu dem Rest der Welt stark kontrolliert und unerwünschte Inhalte herausgefiltert und gesperrt.

Zu den Feinden des Internets, Reporter ohne Grenze (ROG) zufolge, gehörten 2014 China, Saudi-Arabien, Turkmenistan, Russland, Belarus, Syrien, Vietnam und Bahrain. Der Internetverkehr in den Grenzen von China, Syrien und Sudan wurde vorübergehend ganz gestoppt, um die Verbreitung kritischer Informationen zu verhindern. Besonders rigide sind Iran und China. Dort sind nicht nur zahlreiche Seiten blockiert, indem eine Zentralstelle für Filterung eingerichtet wurde, sondern auch die Kommunikation zwischen den Demonstranten wurde mehrmals durch zeitweise Umschaltung der Mobil-Funknetze beispielsweise nach der Präsidentenwahl 2009, eingeschränkt [3].
Die Netzwerkverbindung teilweise oder vollständig abzuschalten ist die einfache Reaktion auf Online-Proteste oder systemkritische Bewegungen mancher Länder, in denen die Internet-Infrastruktur nicht so gut entwickelt ist. Ägypten, als Beispiel, war gegenüber dem Sturm der Inhalte in Soziale-Netzwerke (Facebook, Twitter oder YooTube) während des Arabischen Frühlings überfordert.

In Allgemeinen Client-Server-Systeme ist durch die zentralisierte Architektur, die sie charakterisiert, ein Nährboden für Angriffe und Datenverkehrsüberwachung. Ein gezielter Angriff auf einen Server, der auf ”single point of failure Infrastruktur” basiert, kann den Ausfall des gesamten Netzes verursachen, da die Daten von den Clients nicht mehr greifbar sind [4].

Als Alternative bietet Peer-to-Peer eine robuste Infrastruktur, die resistent gegen jegliche Form der Zensur ist, da Daten nicht zentralisiert sind. Dezentral bedeutet, dass auf keine zentralen Server zurückgegriffen wird. Stattdessen bieten alle beteiligten Endsysteme sowohl Ressourcen und Dienste als auch die in Anspruchnahme von Diensten an.

Das Peer-to-Peer-Modell besteht also aus vielen gleichberechtigten Knoten (den sogenannten Peers), die direkt miteinander kommunizieren, ohne dem Umweg über einen Server zu folgen. Somit skaliert ein Peer-to-Peer-System besser in Bezug auf eine steigende Teilnehmerzahl als eine zentrale Server-Lösung, da erforderliche Berechnungen sowie die Datenablage auf alle beteiligten Knoten aufgeteilt werden können.
Ein anderer Vorteil ist, dass der Ausfall eines Teils eines Peer-to-Peer-Systems (Rechnerausfall, Netzwerkunterbrechung durch Katastrophenszenario oder Autoritäre Regime), nicht unbedingt den Ausfall des Gesamtsystems verursacht. Dennoch sind die Lösungen für Wiedervereinigung getrennter Netzwerke sehr wichtig für Peer-to-Peer-Systeme, so sichert man die automatische Wiederherstellung der Kommunikationsmöglichkeit zwischen verschiedenen Regionen, die sich nicht mehr kennen, nach der Netzwerk Aufspaltung.

In dieser Arbeit werden Mechanismen und Konzepte für die Netzwerkpartitionierung entworfen und Mechanismen für das Peer-to-Peer Overlay-Chord implementiert, die eine automatische Wiedervereinigung getrennter Netzwerke ermöglichen.

1.2 Ziel dieser Arbeit

Das Ziel der Netzwerkpartitionierung ist Regionen bzw. Nutzergruppen zu isolieren, zwischen denen keine Kommunikation mehr existiert bzw. kein Routing mehr möglich ist. Der Verlust der Konnektivität zwischen diesen Regionen kann die Partitionierung des Overlays veranlasst. In diesem Fall werden dann mehrere getrennten Overlay-Partitionen entstehen, die nur Knoten aus der gleichen Region beinhalten.
Die Wiederherstellung der Konnektivität zwischen diesen Regionen wird dann Verfahren bzw. Merging-Algorithmen benötigt, um die getrennten Overlay-Partitionen zusammenzuführen. Diese Algorithmen ermöglichen dabei die automatische Herstellung einer globale Kommunikationsstruktur, die alle Knoten aus allen Partitionen beinhaltet (s. Abbildung 1.1).

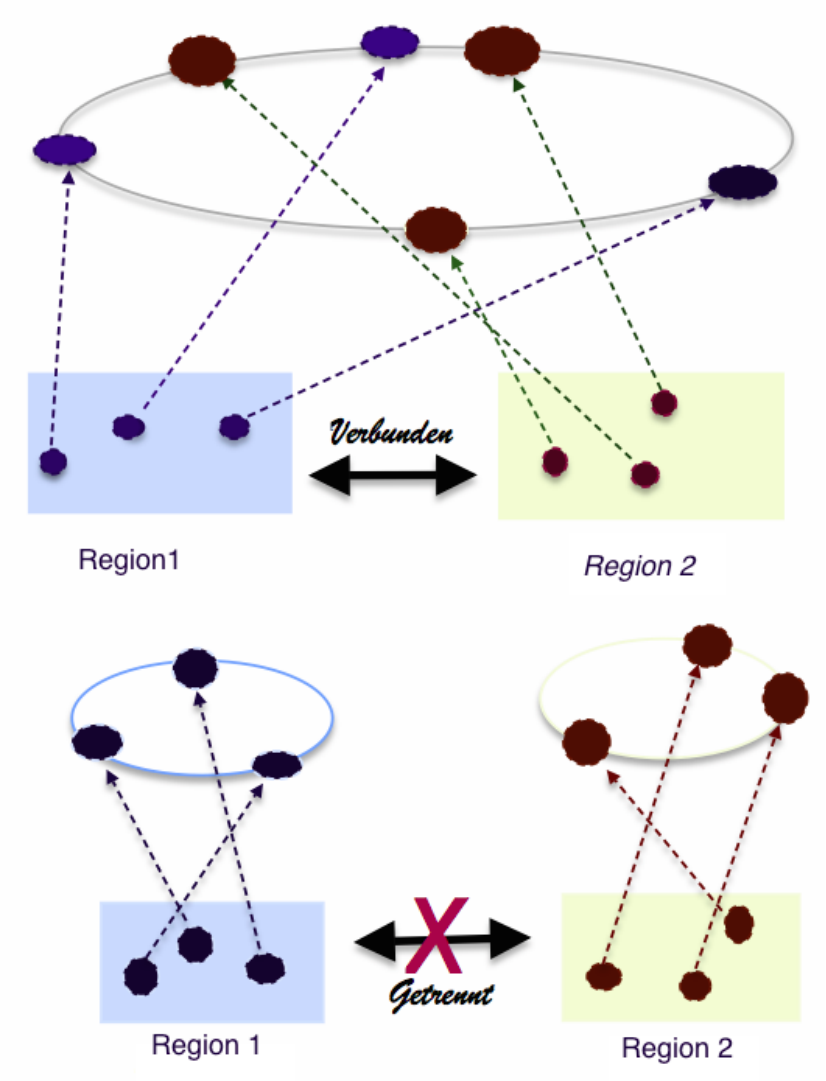


Abbildung 1.1: Netzwerkpartitionierung und die Bildung von Overlay-Partitionen

Diese Arbeit beantwortet dann folgende Fragen:

Wie wird die Netzwerkpartitionierung im Underlay realisiert?
Wie werden die Verschmelzungsverfahren bzw. Merging-Algorithmen im Overlay implementiert? Welche Techniken und Mechanismen werden benötigt, um diese Algorithmen automatische durchzuführen, sodass jeder Knoten die Möglichkeit hat, die Verbindung zu allen Regionen, die nicht mehr erreichbar sind, aufzubauen?

1.3 Gliederung der Arbeit

Kapitel 2 liefert einen Überblick über die Grundlangen, die notwendig für das Verständnis dieser Arbeit sind. Insbesondere wird in die strukturierten Overlay-Netze eingeführt sowie die Merging-Algorithmen vorgestellt und analysiert, die im OverSim implementiert und evaluiert werden.
In *Kapitel* 3 wird die allgemeine Architektur des Overlay-Framworks OverSim vorgestellt, die zur Evaluierung der im Rahmen dieser Arbeit entworfenen und implementierten Konzepte eingesetzt wird. Dabei wird sowohl die Architektur als auch der Simulationsablauf im OverSim betrachtet. Im Anschluss daran befasst sich *Kapitel 4* mit der Netzwerkpartitionierung im Underlay, das ein vorhandenes physisches Netz darstellt. Dabei werden das entworfene und implementierte Konzept, um mehrere Knotengruppen bzw. Regionen zu bilden und dann die Netzwerkverbindung

zwischen den Regionen zeitweise einschränken, vorgestellt und analysiert. Für das Verständnis der Realisierung dieses Konzepts wird zunächst der Aufbau des eingesetzten Underlay näher erläutert.
Darauf basierend werden die implementierten Verfahren bzw. Algorithmen, die benötigt werden, um die getrennten Overlay-Partitionen, zu verschmelzen und die gespaltenen Gruppen zusammenzuführen, in *Kapitel 5* im Detail diskutiert. Dabei werden die Techniken und Mechanismen für die Realisierung dieser Algorithmen vorgestellt. Am Anfang des *Kapitels* werden die benötigte Klassen und Funktionen, die für die Implementierung der Algorithmen verwendet wurden, dargestellt.
In *Kapitel* 6 werden schlie/sslich die Ergebnisse der Evaluierung der in den *Kapiteln 4* und 5 vorgestellten Konzepte und Verfahren erläutert. Zunächst werden alle durchgeführten Szenarien vorgestellt, dann die Ergebnisse der Evaluierung diskutiert und untersucht.
Zum Abschluss folgt in *Kapitel 7* eine Zusammenfassung dieser Arbeit und ein Ausblick auf zukünftige Arbeiten.

Kapitel 2

Grundlagen

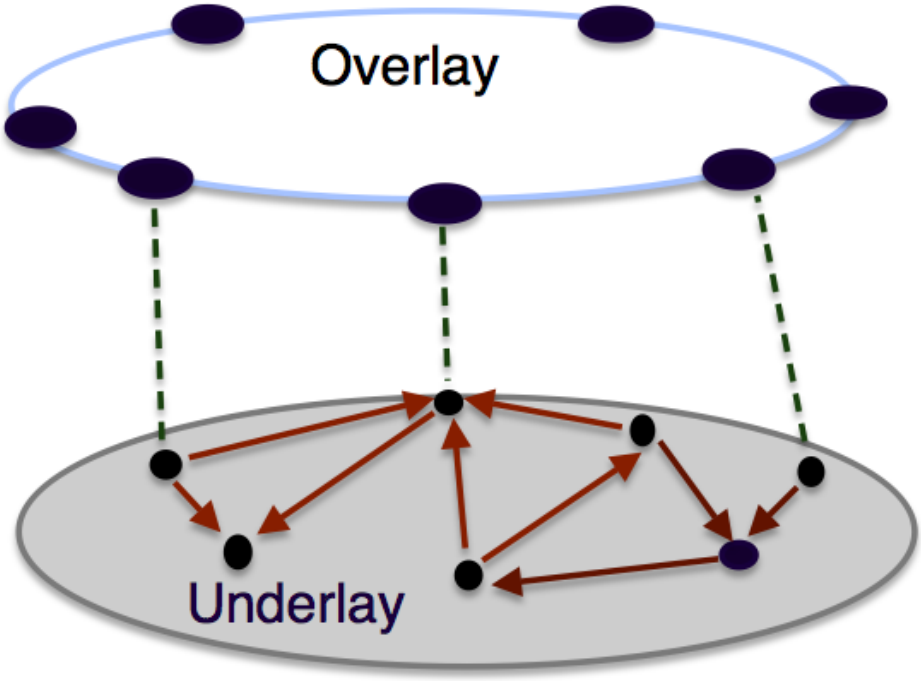


Abbildung 2.1: Overlaynetzwerk als Ringtopologie

Dieses Kapitel soll einen Überblick über die wesentlichen Grundlagen und Techniken, die für das Verständnis der weiteren Arbeit notwendig sind, liefern. Nach einer Einführung in das Thema Peer-to-Peer-Systeme werden in dieser Arbeit im Fokus stehenden strukturierten Overlay-Netze von den unstrukturierten Overlay-Netzen unterschieden. Dabei werden der Aufbau und die Eigenschaften der verteilten Hashtabellen näher erläutert. Die Merging-Algorithmen, die in dieser Arbeit implementiert

und simuliert wurden, werden im Abschnitt 2.2 im Detail diskutiert.

2.1 Peer-to-Peer und Overlay-Netze

Ein Peer-to-Peer-System [5] ist ein Kommunikationsnetzwerk zwischen gleichberechtigten Teilnehmern (Peers), das sowohl Dienste zur Verfügung stellen als auch in Anspruch nimmt.
Also jeder Teilnehmer führt sowohl Client als auch Serveraufgaben durch.
Gegenüber dem Client/Server-Modell, bei dem eine zentrale Instanz existiert (Server), die nur Dienste für mehrere Dienstnehmer anbietet, stellen sich Peer-to-Peer-Systeme als autonome selbstorganisierende Systeme vor, die gemeinsame Ressourcen wie Rechenzeit, Bandbreite oder Speicherplatz teilen. Somit werden mit jedem neuen Endsystem zusätzliche Ressourcen für den Betrieb eingebracht, was der Peer-to-Peer-Infrakstruktur eine hohe Skalierbarkeit und Robustheit ermöglicht.
Zur Kommunikation zwischen den Endsystemen bilden Peer-to-Peer-Netze häufig Overlay-Netze d. h. die Knoten (Peers), die sich im Overlay-Netz befinden, richten eine logische Topologie über ein vorhandenes physisches Netz (Underlay) ein. Das Internet kann als Underlay verwendet werden. Die Verbindungen auf Anwendungsschicht sind die Kanten der Overlay-Topologie [6] (s. Abbildung 2.1). Es gibt verschiedene Overlay-Topologien, die vom Overlay-Protokoll abhängig sind, beispielsweise Ring (z. B. Chord), ein Hyperkubus (z. B. Pastry) oder ein De-Bruijn-Graph (z. B. Broose und Koorde) [5].

Im Folgenden wird ein Überblick über die zwei Klassen von Overlay-Netzen (Strukturierte und unstrukturierte Overlay-Netze) gegeben. Dabei wird auf die Strukturierte Klasse fokussiert, indem die verteilten Tabellen näher erläutert werden.

2.1.1 Unstrukturierte Peer-to-Peer-Systeme

Unstrukturierte Overlay-Netze [7] wurden vor allem in den ersten Peer-to-Peer-Systemen eingesetzt, mit der Hauptaufgabe, Dateien auszutauschen. Diese erste Generation basierte auf einer Server-orientierten Suche. Die Daten werden dabei in einem Server verwaltet, obwohl die Kommunikation zwischen den Peers direkt stattfindet. Genutella sowie dessen Nachfolger Gia sind Beispiele solche Systeme, wo jeder Overlay-Knoten mit einer Reihe weiterer Overlay-Knoten, welche zufällig ausgewählt werden, in Kontakt kommt.
Die Suchanfrage basiert auf dem Prinzip des Flutens, d. h. jeder Overlay-Knoten leitet die Suchnachricht nach lokaler Bearbeitung an alle beteiligten Overlay-Knoten weiter. Wenn eine zuvor festgelegte Anzahl von Weiterleitungen im Overlay-Netz erreicht ist, wird die Suchnachricht verworfen. Sobald ein Overlay-Knoten das gesuchte Datum enthält, antwortet er direkt dem Initiatorknoten. Das Fluten

beeinflusst die Skalierbarkeit dieser Klasse von Peer-to-Peer-Systemen, da es zu enormen Belastungen der Kommunikationsverbindungen führt.

2.1.2 Strukturierte Peer-to-Peer-Systeme

Strukturierte Overlay-Netze [5] , [7] sind nach den unstrukturierten Overlay-Netzen zustande gekommen. Dabei wurden die Skalierbarkeitsprobleme durch geschickte Einführung von weiteren Kanten und Routingalgorithmen vermieden.

Die Overlay-Knoten verfügen über Informationen über die Wegewahl und können dadurch eine effiziente Weiterleitung der Suchnachrichten zum Zielknoten ermöglichen. Jedem Overlay-Knoten wird ein eindeutiger m-Bit Schlüssel (Node-ID) zugewiesen, z. B. durch das Anwenden einer Hashfunktion auf die IP-Adresse. Anhand dieser wird die Position der Overlay-Knoten festgelegt und gezielt Verbindungen zu weiteren Overlay-Knoten aufgebaut, sodass eine bestimmte Overlay-Struktur entsteht (z. B. Ring, Hyperkubus... usw.). Um die für eine Datei zuständigen Peers zu finden, wird die abgelegte Information auf dem gleichen Schlüsselraum abgebildet.
Aufgrund dieses Hash-basierten Ansatzes werden strukturierte Peer-to-Peer-Netzwerke häufig auch als DHTs bezeichnet. Darauf wird im Folgenden näher eingegangen. Zu diesen Systemen gehören unter anderem Chord, Pastry, Kademelia, Bamboo, Koorde und Broose. Da Chord in dieser Arbeit verwendet wurde, wird dieses Overlay-Protokoll vorgestellt.

Verteilte Hashtabellen

Verteilte Hashtabellen (distributed hash tables, DHT) [8], [5] , sind dezentrale Datenstrukturen, die große Mengen von (Schlüssel, Wert) Paaren verwalten. Sie ermöglichen damit eine effiziente Suche nach Daten zu einem bestimmten Schlüssel. Jeder Knoten verwaltet dabei eine Routing-Tabelle mit einer bestimmten Anzahl von Datensätzen für andere Knoten.
Es wird also für jeden Overlay-Knoten sowie jedes Datum ein Schlüssel aus einem linearen Wertebereich, mittels einer Basis-Hashfunktion (SHA-1), erzeugt. Der Knoten bekommt dann eine *Node-ID* durch die Verschlüsselung der entsprechenden IP-Adressen im darunterliegenden (physischen) Netzwerk. Dabei bekommt das Datenobjekt, das in einer verteilten Hash-Tabelle gespeichert werden soll, vorher einen eindeutigen Namen. Um nun das gegebene Paar aus Namen und Datenobjekt zu speichern und später wieder auffinden zu können, wird der eindeutige Name des Datenobjekts auf einen eindeutigen Schlüssel innerhalb des vorhandenen Schlüsselraumes des verwendeten strukturierten Overlay-Netzes abgebildet. Das Wertepaar bestehend, aus Key und zugehörigem Datenobjekt, wird dann in den Knoten gespeichert, welcher den Schlüsselbereich verwaltet.

Der entstehende Schlüsselraum wird auf alle Overlay-Knoten verteilt, somit sind alle Knoten gleichberechtigt.
Eine Lookup-Anfrage wird an den Knoten weitergeleitet, der aus der Routing-Tabelle ausgewählt wird, dessen *Node-ID* näher am Schlüssel ist. Dadurch liegt der Laufzeit eine Lookup Operation in *O(log N)*. Dabei ist zwar der Verwaltungsaufwand eines jeden einzelnen Peers höher als in einem unstrukturierten Peer-to-Peer-Netzwerk. Allerdings fällt dieser durch die Vorteile, die DHT ermöglicht, kaum ins Gewicht. DHT als Konzept zur effizienten Verteilung von Routinginformationen über die Peers hat das Ziel die Suchanfragen zu beschleunigen, ein robustes Netzwerk zu verschaffen und eine gute Skalierbarkeit zu ermöglichen.
Da es nun normalweise weniger Knoten als Schlüssel gibt, muss ein Knoten mehrere Schlüssel-Wert-Paare verwalten, d. h. jeder Knoten verwaltet eine Partition des Schlüsselraumes. In einem Netzwerk mit drei Knoten ist Knoten K1 beispielsweise zuständig für alle Keys von 00.. 00 bis 3F.. FF, Knoten K2 für alle Keys von 40.. 00 bis 7F.. FF und Knote K3 für die Keys von 80.. 00 bis BF.. FF. Eine Nachricht für den Key AC.. CC fällt im obigen Beispiel also in den Bereich von Knoten K3 [9].

Chord

Chord [6] ist ein strukturiertes Overlay-Protokoll, das eine einfache Struktur (Ring) im Overlay-Netz darstellt. Aus diesem Grund wird Chord oft als Standardbeispiel für strukturierte Overlay-Protokolle verwendet.
Alle Knoten werden anhand ihrer eindeutigen Nummer (genannt "Node-ID") aus dem Schlüsselbereich 2^{m-1} in einem logischen Ring platziert, der als unidirektionaler Ring aufgefasst wird, da auf den größten Schlüssel aus der Schlüsselmenge direkt *Node-ID* 0 folgt. Somit werden die Nachrichten nur in eine aufsteigende Richtung geroutet.
Jeder Knoten verwaltet eine Liste von Nachfolgerknoten, die aus aufeinanderfolgenden Knoten (successor) auf dem Ring (die sog successer-Liste) besteht. Mit dieser Liste kann ein Schlüssel in *O(N)* Schritten aufgefunden werden. Um die Laufzeit auf *O(logN)* zu verbessern, verwaltet jeder Knoten zusätzlich noch eine Finger-Tabelle, welche alle Knoten bzw. Finger aufnimmt, für die Finger(i) ist Successor von *(Node-ID* $+2^{m-1}$*, 1<= i <=m)* gilt. Die Verwendung der Finger-Tabelle ermöglicht die Halbierung des Abstands zum gesuchten Schlüssel k in jedem Schritt, womit sich eine Pfadlänge von *O(logN)* ergibt. Der unmittelbare Vorgänger auf den Ring (dem sog. Predecessor) ist auch von jedem Knoten bekannt. Somit ist er für alle Schlüssel verantwortlich, für die er der Nachfolger ist.

Schlüsselsuche (lookup)

Das Auffinden eines Schlüssels im Ring [9] erfolgt entweder ohne Einsetzen der Finger-Tabelle, indem der anfragende Knoten der Lookup-Nachricht an seinen unmittelbaren Nachfolger solange weiterleitet, bis der gesuchte Schlüssel zwischen der eigenen *Node-ID* und dem des unmittelbaren

Nachfolgers liegt, der den gesuchten Schlüssel verwaltet, oder über einen Algorithmus, der sich die Eigenschaften der Finger-Tabellen zu Nutze macht. (s.Abbildung 2.2)

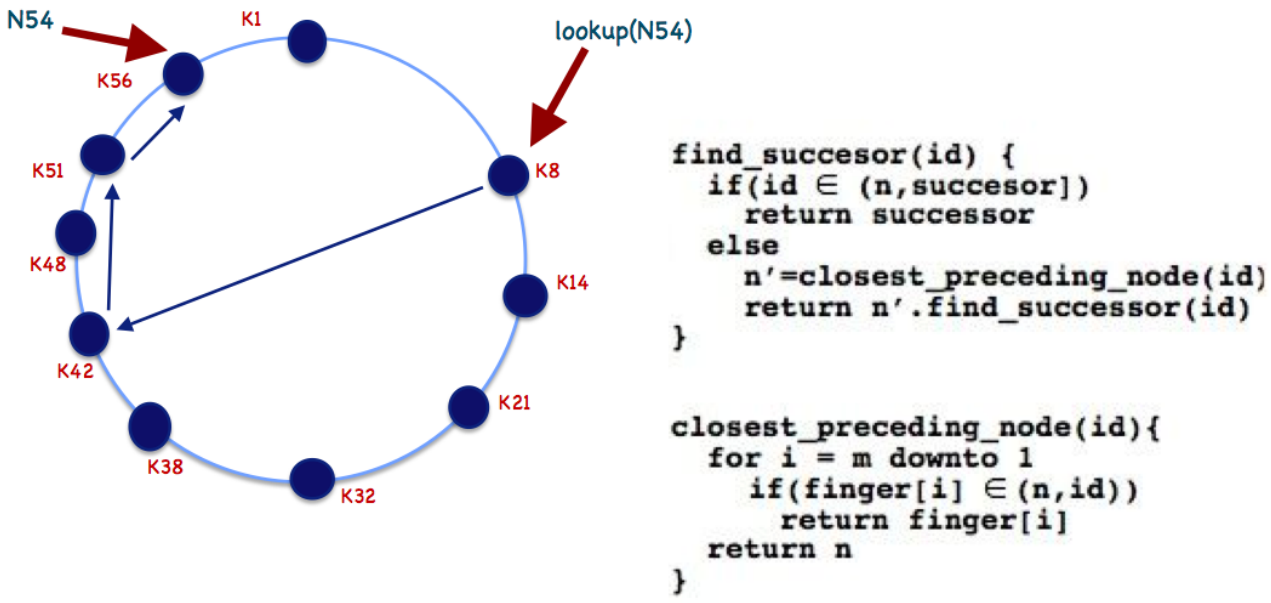


Abbildung 2.2: Lookup im Chord (Schlüsselsuche) mit Finger-Tabelle.

Dabei wird der anfragende Knoten überprüfen, ob der Schlüssel zwischen dem eigenen und dem *Node-ID* des Nachfolgers liegt. Wenn ja, dann ist der gesuchte Knoten bereits gefunden. Ansonsten gibt der Knoten die Suchanfrage an den Knoten in der Finger-Tabelle weiter, der am nächsten am Schlüssel liegt. Dieser Knoten übernimmt die Suchaufgabe und leitet die Anfrage weiter, bis schließlich ein Knoten gefunden wird, dessen Nachfolger für den gesuchten Schlüssel zuständig ist. Dieses Verfahren kann sowohl iterativ als auch rekursiv ausgeführt werden.

Netzbeitritt

Der Beitritt von neuen Knoten [5, 9] in das Chord-Overlay [10] benötigt das Kennenlernen eines Knotens, der bereits im Overlay-Netz ist. Aus diesem Grund wird ein zufälliger Knoten (Bootstrap-Knoten) zur Verfügung gestellt. Dieser Knoten übernimmt dabei die Aufgabe, den zuständigen Schlüssel für den neuen beitretenden Knoten aufzufinden.
Diese Nachricht, die die erzeugte Node-ID enthält, wird dann schrittweise von dem Bootstrap-Knoten bis zu dem Nachfolger von dem Knoten K weitergeleitet. Dieser Knoten antwortet dann direkt dem Knoten K mit seiner Node-ID und wird daraufhin als Nachfolger von K aufgenommen.
Die Abbildung 2.3 illustriert die Schritte für den Netzbeitritt von dem Knoten k15

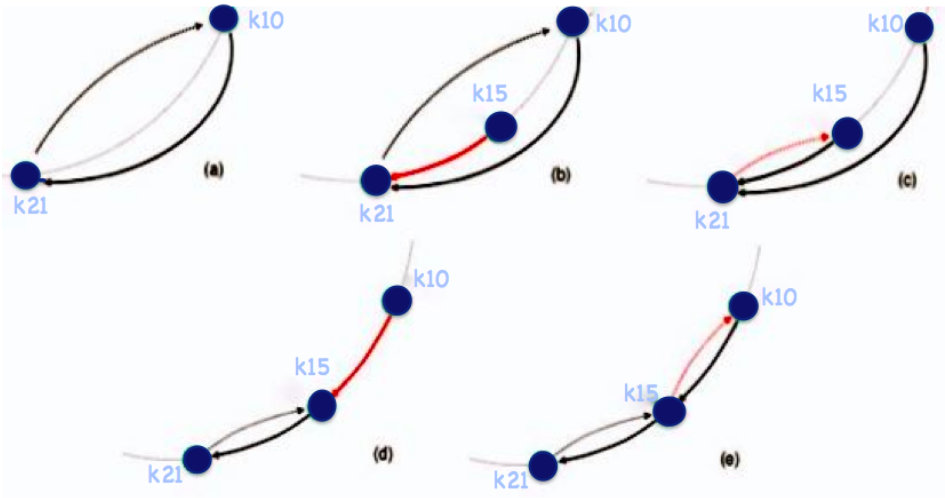


Abbildung 2.3: Der Netzbeitritt eines Knotens im Chord

Um die Routing-Tabelle nach jeder Netzwerkänderung zu aktualisieren, sendet jeder Knoten K in periodischen Intervallen folgenden Nachrichten [5].

stabilize() / notify():
stabilize() fragt den aktuellen Nachfolger N nach dessen Vorgänger V. V wird von K als neuer Nachfolger eingetragen, falls V zwischen K und N liegt d. h. V ist kürzlich dem Netz beigetreten.
Der Nachfolger enthält direkt danach eine Notify-Anfrage von K, damit dieser gegebenenfalls K als neuen Vorgänger eintragen kann.

check-predecessor() :
wird periodisch von jedem Knoten K an seinen Vorgänger gesendet, damit die ausgefallenen Knoten erkannt werden.

fix-fingers():
Mit dieser Prozedur wird die Finger-Tabelle aufrechterhalten und stabilisiert. Dazu werden in regelmäßigen Abständen Anfragen an alle *O(logN)* successor((K. Id+2^i) mod 2^m)) mit i = 0,.. , m geroutet. Die Knoten, die für die Zielschlüssel zuständig sind, werden dann in die Finger-Tabelle eingetragen.

Netzaustritt (leave)

Wenn ein Knoten ausfällt z. B. durch einen Abbruch der Netzverbindung, kümmert sich das Stabilize-Protokoll um die Reparatur des Rings im nächsten Durchgang. Das Verlassen des Rings bzw. der beabsichtigte Austritt des Rings, kann die Leistung des Ring-Protokolls verbessern, da der Knoten dabei seinen Vorgänger und Nachfolger benachrichtigt. Somit wird die Nachricht zur Reparatur übermittelt

d. h. die Zeiger aktualisieren, indem die kontaktierten Knoten die neuen Vorgänger bzw. Nachfolger übernehmen.

Im allgemeinen hat das Beitreten oder das Verlassen des Chordrings keinen Einfluss auf den Anfrage-Aufwand, der mit großer Wahrscheinlichkeit bei O(logN) bleibt, obwohl das Stabilize-Protokoll ständig den Ring reparieren muss [9].

2.2 Merging-Algorithmen

Der Verlust der Konnektivität zwischen mehrer Gruppen von Knoten kann die Partitionierung des Underlays sowie des Overlays hervorrufen. Die Wiederherstellung dieser Konnektivität wird somit Verfahren verlangen, um getrennte Overlay-Partitionen zusammenzuführen.
In diesem Abschnitt werden dann die bekannten Verfahren bzw. Algorithmen, die Konzepte für das Verschmelzen der getrennten Overlay-Topoligien zur Verfügung stellen, diskutiert.
Die bereits entwickelten Algorithmen: Chord-Zip, Ring Unification, Gossip-based Ring Unification, Ring Reunion und seine parallelisierte Version werden im Folgenden vorgestellt.

2.2.1 Chord-Zip

Voraussetzung für Chord-Zip [11] sind die lokale Informationen, die in jedem Knoten gespeichert sind. Dabei ist es nicht notwendig, dass die einzelnen Knoten die gesamte Topologie kennen. Vielmehr muss jeder Knoten nur seine Position im eigenen Ring kennen und den engsten Nachfolger anhand der Node-IDs in einem anderen Ring auffinden. Während der Verschmelzung wird jeder Knoten den alternativen Nachfolger bzw. alternativen Vorgänger im anderen Ring referenzieren. Dabei werden auch die gesendeten Finger-Tabellen und Successor-Listen mit eigenen Tabellen und Listen kombiniert.

Chord-Zip bietet eine weitere Eigenschaft, so dass jeder Knoten die Verschmelzungsnachricht dem alternativen Nachfolger weiterleitet. Somit ist er beauftragt, das Verschmelzen weiterführen. Der Prozess wird abgeschlossen, wenn der Knoten, der den Algorithmus gestartet hat, die Verschmelzungsnachricht wieder bekommen hat.
Im Algorithmus wurde nicht betrachtet, ob ein Initiatorknoten sich selbst als alternativer Nachfolger bekommen hat. Die PING- und PONG-Nachrichten werden trotzdem weitergeleitet. Dadurch wird das Verfahren verlangsamt.

Im Allgemeinen wird der Algorithmus terminiert, wenn der Initiatorknoten wieder die PING-Nachricht

bekommen hat (s. Abbildung 2.4). Dann antwortet er dem Sender mit einer PONG-Nachricht und ändert dabei seinen Zustand. Somit wird das Weiterleiten von Nachrichten gestoppt und der Algorithmus terminiert.

Zur Behandlung von Paketverlusten haben die Autoren von Chord-Zip [11] vorgeschlagen, dass ein Knoten, welcher den alternativen Nachfolger nicht mehr erreichen kann, einen anderen alternativen Nachfolger aus der empfangenen Nachfolgerliste (Succcessorlist) auswählt. Der Knoten wird dann dem neuen alternativen Nachfolger die Merging-Nachricht einreichen.

Falls der Knoten, der die Merging-Nachricht empfangen hat, aus dem Netzwerk ausgetreten ist, ohne die Nachricht weiterleiten zu können, wird der Chord-Zip Mechanismus nach Ablauf eines verwendeten Timeout neu gestartet. Dabei wurde allerdings nicht berücksichtigt, dass der Initiatorknoten am Anfang des Verfahrens die Successor-Liste nicht bekommt hat, da er die PING-Nachricht erst am Schluss erhalten kann.

Im Allgemeinen kann Chord-Zip die doppelten gerouteten Nachrichten nicht unter Kontrolle halten, da er dieses Problem nicht korrekt behandeln kann, was dazu führt, dass der Algorithmus nicht komplett terminiert werden kann. .

Merging

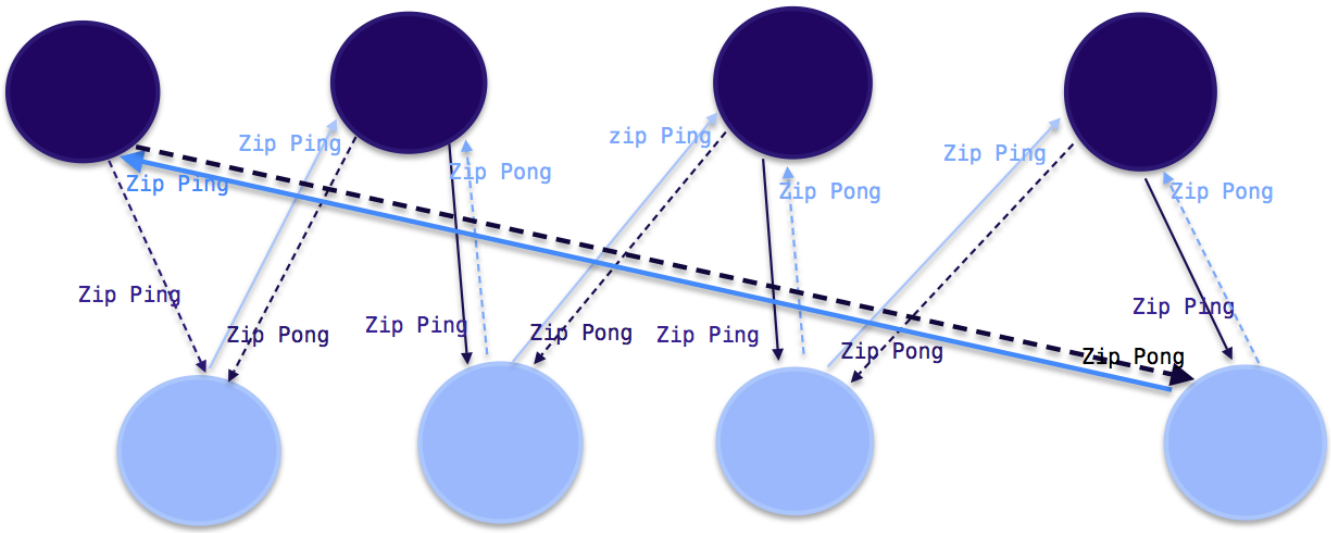


Abbildung 2.4: Terminierungsverfahren vom Zip-Chord

Das Ziel des Chord-Zip Algorithmus [11, 12] ist die Verschmelzung von zwei Chord-Ringen *R1* und *R2*.
Ein Knoten *n* aus *R1* initiiert den Merging Prozess und stellt zunächst eine Verbindung zu einem Kontaktknoten *k* aus *R2* her (s. Listing 5.5 Zeile 1). Dieser startet eine Lookup-Anfrage nach dem Schlüssel von *n* in *R2* (s. Listing 5.5 Zeile 2). Dabei antwortet der alternative Nachfolger direkt dem Knoten *n*. Der Knoten *n* routet dann eine Zip-PING-Nachricht zum alternativen Succcessor (Zeile 4). Daraufhin wird eine Zip-PONG-Nachricht vom Empfänger zurückgesendet und die-Zip-PING-Nachricht weitergeleitet (Zeilen 8-14).

Mit Zip-PING und Zip-PONG-Nachrichten werden die Successor-Listen und Finger-Tabellen ausgetauscht. Dabei werden die neuen alternativen Successor bzw. Predecessor in die Tabellen aufgenommen, nachdem überprüft wird, dass der Nachfolger-ID zwischen dem aktuellen Nachfolger-ID und eigenen-ID liegt. Die Tabellen werden dabei auch miteinander kombiniert. Diese Verfahren werden weitergeführt bis der Initiatorknoten die Zip-PING-Nachricht bekommt. Dann schickt er eine Zip-PONG-Nachricht zurück. Der Initiatorknoten wird daraufhin die Tabellen miteinander kombinieren und den Prozess abschließen.

```
1  receipt of STARTZIPPER (contact) from m at n
2    sendto contact:LOOKUP (n.id)
3    if altSucc is received from contact within time interval g then
4       sendto altSucc:ZIPPING (n.succ,n.succList,n.fingerTable)
5       n.isInitiator:=true
6    end if
7  end event
8
9  receipt of ZIPPING (m.succ,m.succList,m.fingerTable) rom m at n
10   sendto m:ZIPPONG (n.pred,n.succList,n.fingerTable)
11   if n.isInitiator=true then
12      combine()
13      n.isInitiator:=false
14   else
15      sendto m.succ:ZIPPING(n.succ,n.succList,n.fingerTable)
16   end if
17 end event
18
19 receipt of ZIPPONG (m.pred,m.succList,m.fingerTable) from m at n
20   if n.isInitiator=false then
21      combine()
22   end if
23 end event
```

Listing 2.1: Chord-Zip

2.2.2 Ring Unification Agorithm

Im Folgenden werden der Ring Unification Algorithm [12–14]und seine verbesserte Version Gossip based Algorithm präsentiert. Dabei werden deren Funktionalitäten näher erläutert. Basierend auf dem Pseudo-Code von jedem Algorithmus wird am Schluss der Ring Unification Algorithm von seiner verbesserten Variante abgegrenzt.

Die beiden oben genannten Algorithmen werden gestartet, wenn ein Initiatorknoten einen Kontaktknoten in einem anderen Ring aufgefunden hat. Dafür haben die Autoren eine passive Liste vorgeschlagen. Diese Liste wurde neben anderen Listen (s. Abschnitt 5.2 Kapitel 5) im Rahmen dieser Arbeit implementiert und die Algorithmen damit getestet und simuliert. Die im [13] beschriebene Liste enthält alle Knoten, die nicht mehr erreichbar sind. Dabei wird der passive Knoten regelmäßig mit PING-Nachrichten kontaktiert. Wenn dieser wieder erreichbar ist, antwortet er dem anfragenden Knoten mit einer PONG-Nachricht. Daraufhin wird der Merging-Algorithmus in beiden Ringen gestartet und dieser Knoten von der passiven Liste entfernt. Ein Nachteil einer solchen passiven Liste ist, dass nur Knoten zusammengeführt werden können, die bisher bekannt waren. Änderungen der Routing-Informationen in jeder einzelnen Partition könnte den Inhalt der passiven Liste unbrauchbar machen, weil Informationen über frühere Kontaktknoten veraltet sein können und einige Knoten möglicherweise nicht mehr existieren. In diesem Fall könnte ein Systemadministrator einen Kontaktknoten zu einem bestimmten Knoten manuell einfügen und dadurch einen Zusammenführungsalgorithmus starten.

Im Vergleich mit Chord-Zip [11, 12] wurden in Merge Unification Algorithm der Nachrichtenverlust und die gewünschte Sendewiederholungen nicht berücksichtigt. Mit der Einführung der Listen, die alle vorhandenen Konten enthält, kann die Sendewiederholung von Nachrichten nicht unbedingt gebraucht werden. Der Algorithmus känn mit Nachrichtenverlusten gut umgehen, obwohl sie keinen Mechanismus dafür entwickelt haben. Die verwendete passive Liste detektiert die ausgefallenen Knoten. Dabei werden in periodischen Abständen die Knoten mit PING-Nachrichten kontaktiert, und falls keine Antwort zurück gesendet wird, werden die ausgefallenen Knoten in der passiven Liste aufgenommen. Die wiederverfügbaren Knoten werden dann zusammengeführt. Der Algorithmus, wie Ring Reunion Algorithm, kann frühzeitig terminiert werden, wenn der Knoten, der zu verschmelzen ist, schon den Algorithmus ausgeführt hat. Im allgemeinen kann der Algorithmus mehrere Ringe gleichzeitig zusammenführen.

Simple Ring Unification

Im Folgenden wird das Merging-Verfahren dargestellt. Zunächst wählt der Initiatorknoten p (s. Listing 2.2 Zeilen 1-6) den Kontaktknoten q aus der Liste detqueue, dann wird Mlookup(p) bzw. Mlookup(q) aufgerufen. Somit wird das Verfahren in beiden Ringen durchgeführt. Mit MLooKup (id) (Zeilen 6-

17) überprüft der Knoten n, der die Nachricht Mlookup(id) empfangen hat, ob der Id zwischen dem eigenen id und dem Nachfolger id oder dem Vorgänger id liegt. Falls das nicht der Fall ist, wird die Nachricht weitergeleitet, bis der Schlüsssel, der näher am id ist, aufgefunden wird. Dann sendet der Knoten n der Id eine TryMerger() Nachricht, die die Nachfolger bzw. Vorgänger enthält. Dabei wird bestimmt, ob der Schlüssel id sich zwischen dem Knoten n und dem Nachfolger oder dem Vorgänger befindet. In den Zeilen (17-26) wird die Methode TryMerger () dann durchgeführt. Somit wird die Mlookup (id) Nachricht weitergeroutet und die neue passenden Nachfolger und Vorgänger werden in die Tabellen aufgenommen.
In den Zeilen 18 und 22 in dem Listing 2.2 und der Zeilen 23 und 27 in dem Listing 2.3 wird die Methode Mlookup(id) lokal durchgeführt und nicht über die Schnittstelle gesendet. Dadurch könnte eine mögliche Verzögerung vermieden werden.

```
1every alpha time units and detqueue!= 0 at p
2    q:= detqueue.dequeue()
3    sendto p:MLOOKUP (q)
4    sendto q:MLOOKUP (p)
5  end event
6
7  receipt of MLOOKUP (id) from m at n
8    if id 6= n and id=succ then
9       if id 2 (n,succ) then
10         sendto id : TRYMERGE (n,succ)
11      else if id 2 (pred, n) then
12         sendto id:TRYMERGE (pred, n)
13      else
14         sendto closestprecedingnode(id):MLOOKUP (id)
15      end if
16   end if
17 end event
18
19 receipt of TRYMERGE (cpred,csucc) from m at n
20   sendto n:MLOOKUP (csucc)
21   if csucc in (n,succ) then
22       succ := csucc
23   end if
24   sendto n:MLOOKUP (cpred)
25   if cpred in (cpred,n) then
26       pred:=cpred
27   end if
28  end event
```

Listing 2.2: Simple Ring Unification

Gossip-based Ring Unification Algorithm

Mit diesem Algorithmus werden zusätzliche Schritte eingeführt, um die Nachteile vom Simple Ring Unification zu vermeiden. Im Allgemeinen hat der Simple Unification Algorithmus zwei Nachteile. Erstens, der Algorithmus ist langsam und braucht meistens O(N) Schritte, um einen kompletten Ring zu bilden. Zweitens kann der Algorithmus in manchen Szenarien nicht terminieren.

Mit dem Gossip-based Ring Unification Algorithm werden mehrere Instanzen des Algorithmus gestartet. Wenn ein Knoten eine Mlookup(id) Nachricht bekommen hat, wählt er zufällig einen Knoten aus der Routing-Tabelle aus. Dieser Knoten startet dann den Algorithmus, der wie üblich zwei Mlookup-Nachrichten verschickt. Dabei wird ein Fanout F Parameter der Methode Mlookup() übergeben (s. Listing 2.3 Zeilen 2-5). F bestimmt, wie viele Instanzen jedes Mal gestartet werden müssen. Am Anfang wird F mit 4-3 initialisiert, wie von den Autoren vorgeschlagen wurde. Dieser Parameter wird jedes Mal dekrementiert.

Der gossip-based Ring Unification Algorithm bietet die Möglichkeit, mehrere Mergin-Instanzen zu starten. Der Empfänger der MLookup Nachricht bekommt neben dem Id auch einen Fanout-Parameter (F). Dabei wird F dekrementiert, falls F grösser als 1 ist (s. Listing 2.3 Zeilen 8-9). Anschließend wird ein zufälliger Knoten aus der Routing-Tabelle ausgesucht (s. Listing 2.3 Zeile 10) und damit beauftragt, den Algorithmus mit dem neuen Parameter F zu starten. Somit wird die Anzahl der Knoten, die den Algorithmus gleichzeitig durchführen, beschränkt.

Die Durchführung mehrerer Instanzen erhöht die Robustheit des Algorithmus und verhindert manche pathologischen Szenarien, beispielsweise wenn zwei Ringe zusammengeführt werden, in denen alle Knoten auf eigene Nachfolger und Vorgänger im eigenen Ring verweisen, die anderen zusätzlichen Nachfolger zeigen aber auf Knoten in einem anderen Ring. In diesem Fall wird mit einer Mlookup-Nachricht der Algorithmus sofort beendet und der Ring verlassen.

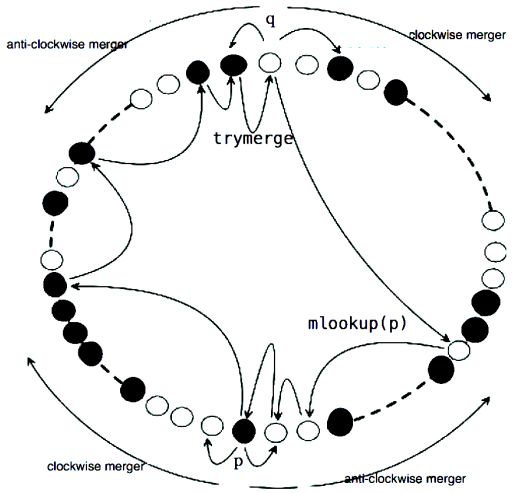


Abbildung 2.5: Der Zusammenführungsprozess des Ring Unificaton Algorithmus in beiden Richtungen

```
every g time units and detqueue 6= 0/ at p
  <q,f>:=detqueue.dequeue()
  sendtop:MLOOKUP(q,f)
  sendtoq:MLOOKUP(p,f)
end event

receipt of MLOOKUP (id,f) from m at n
  if id?n and id ? succ then
   if f >1 then
    f:= f −1
    r:=randomnodeinRT()
    at r:detqueue.enqueue(hid, f i) 12
   end if
   if id(n,succ) then
    sendto id:TRYMERGE (n,succ)
   else if id 2 (pred, n) then
    sendto id:TRYMERGE (pred,n)
   else
    sendto closestprecedingnode(id):
       MLOOKUP (id)
   end if
  end if
end event

receipt of TRYMERGE (cpred,csucc) from m at n
  sendto n : MLOOKUP (csucc)
    if csucc 2 (n, succ) then
       succ:= csucc
    end if
    sendto n : MLOOKUP (cpred)
    if cpred 2 (cpred,n) then
        pred := cpred
    end if
  end event
```

Listing 2.3: Gossip-based Ring Unification Algorithm

2.2.3 Ring Reunion Algorithm

In diesem Unterabschnitt werden zwei Merging-Algorithmen [12] vorgestellt: Ring Reunion Algorithm und seine verbesserte Version. Diese Algorithmen wurden im Rahmen der Masterarbeit von Tobias Amft [12] entwickelt. Dabei stellen die Algorithmen eine erweiterte Version des Chord-Zip-Algorithmus dar. Um Kontaktknoten aufzufinden, werden die passive Liste und die manuelle Liste verwendet. Zusätzlich zu diesen Listen wird eine Liste von zufälligen Knoten, die jeder Knoten von

der Bootstrap-Liste während des Netzbeitritts erhalten kann, angewendet. Die passive Liste wird periodisch aufrechterhalten. Dabei versucht jeder Knoten die Liste mit PING-Nachrichten zu erreichen. Wenn ein Knoten in der Liste erreichbar ist, wird er von dem Initiatorknoten kontaktiert und daraufhin das Merging-Verfahren gestartet. Ob der Kontaktknoten sich innerhalb oder außerhalb des Rings befindet, spielt es dabei keine Rolle. Die passiven Listen haben den Nachteil, dass die Gruppen, die sich in gleichen isolierten Regionen befinden, mit solchen Listen nicht verschmolzen werden können.

Ring Reunion Algorithm

Dieser Algorithmus [12] stellt eine erweiterte Version des Chord-Zip-Mechanismus dar. Wie im Chord-Zip [11]
wird zunächst der Kontaktknoten bestimmt, der vom Initiatorknoten kontaktiert wird, um
eine Lookup-Anfrage im eigenen Ring zu starten. Dabei kann eine passive bzw. aktive Liste verwendet werden. Wenn der alternative Nachfolger (altSucc) vom Initiatorknoten aufgefunden wurde, wird die Merging-Nachricht dem alternativen Nachfolger eingereicht. Dabei wird die Methode Merge (altSucc) durchgeführt (s. Listing 2.4 Zeile 3-4). Diese Methode routet an den Zielknoten, der näher am Schlüssel Id ist, die Merging-Nachricht weiter.

```
1 receipt of STARTMERGER(cont act ) from m at n
2   sendto contact:LOOKUP (n. id)
3     if altSucc is received from contact within time interval g then
4       MERGE (altSucc)
5     end if
6 end event
7
8 receipt of MERGE (altSucc) from m at n
9   pred:= m
10  if n?altSucc then
11     if altSucc 2 (n, succ) then
12     sendto altSucc : MERGE (succ)
13      succ:= altSucc
14   else
15      sendto succ : MERGE (altSucc)
16    end if
17   end if
18 end event
```

Listing 2.4: Ring Reunion Algorithm

Ein Knoten, der die Merging-Nachricht bekommt (s. Listing 2.4 Zeile 8-15), setzt den Absender als neuen Vorgänger ein und den gesendeten alternativen Nachfolger als neuen Nachfolger, falls er näher als der aktuelle Nachfolger ist. Somit wird die richtige Position der zu verschmelzenden Knoten be-

rücksichtigt.
In diesem Algorithmus werden im Gegensatz zum Chord-Zip, die Finger-Tabellen und die Successor-Listen nicht ausgetauscht.

Verbesserte Version vom Ring Reunion Algorithm

Um mehrere Merging-Instanzen zu starten, wurde ein neuer Algorithmus [12] entwickelt. Diese Instanzen führen den Algorithmus gleichzeitig und unabhängig voneinander aus. Dabei wird die Methode distribute von jeder Instanz aufgerufen (s. Listing 2.5 Zeile 2). Der Initiatorknoten fordert den weitesten Knoten aus der Finger-Tabelle auf, den Algorithmus zu starten. Daraufhin wird dieser Knoten den zweiten weitesten Knoten auswählen, um den Algorithmus zu starten. Also jedes Mal, wenn das Verfahren ausgeführt wird, wird ein Counter übergeben, der die Position der Knoten in der Finge-Tabelle bestimmt. Jedes Mal wird der Counter dekrementiert, damit die Anzahl der zusätzlichen Instanzen nicht $2^{maxInstances-1}$ überschritten wird. Dabei ist maxInstances eine Konstante, die am Anfang festgelegt wird.

```
1 receipt of STARTMERGER(cont act ) from m at n
2   DISTRIBUTE (0, contact)
3 end event
4
5 receipt of DISTRIBUTE (cnt, contact ) from m at n
6   sendto contact : LOOKUP (n. id)
7   if altSucc is received from contact within time interval g then
8    while (cnt < maxInstances) do
9      sendto fingerEntry(lastEntryPos-cnt)  : DISTRIBUTE(cnt +1, contact)
10     cnt := cnt + 1
11   end while
12   MERGE (altSucc)
13  end if
14 end event
15
16 receipt of MERGE (altSucc) from m at n
17  pred := m
18  if n ?altSucc then
19    if altSucc in (n, succ) then
20      sendto altSucc : MERGE (succ)
21      succ:=altSucc
22    else
23      sendto succ : MERGE (altSucc)
24    end if
25  end if
26 end event
```

Listing 2.5: Ring Reunion Algorithm with parallelization

2.3 Zusammenfassung

In diesem Kapitel wurde zunächst in die Peer-to-Peer-Systeme eingeführt, dann wurde ein Überblick über die strukturierten Overlay-Netze gegeben. Die verteilten Hashatabellen wurden dabei näher betrachtet. Das in dieser Arbeit verwendete Overlay-Protokoll wurde näher erläutert. Somit werden die Grundlagen der in den kommenden Kapiteln durchgeführten Konzepte gebildet. Im zweiten Abschnitt wurden die Verfahren, die im Rahmen dieser Arbeit implementiert, getestet und bewertet wurden, im Detail diskutiert. Dabei wurde auf die Funktionalität der vorgestellten Merging Algorithmen fokussiert. Ziel ist es, in dem Kapitel 5 basierend auf diesen Grundlagen die Techniken und Mechanismen der Implementierung dieser Algorithmen vorzustellen.

Kapitel 3

Der Simulator OverSim

Das Overlay-Framework OverSim wird in dieser Arbeit eingesetzt, um Knotengruppen im Underlay zu bilden und dann die Konnektivität zwischen Knoten aus unterschiedlichen Knotengruppen zeitweise einzuschränken und dabei die Effekte auf die Overlay-Topologie (Chord) zu untersuchen.
Der Simulator unterstützt bereits mehrere strukturierte und unstrukturierte Overlay-Protokolle wie Chord, Kademlia, Pastry, Bamboo, Koorde, Broose, Gia und Vast.
Alle Protokollimplementierungen können unverändert sowohl für die Simulation als auch für Evaluierung in echten Netzwerken eingesetzt werden. OverSim unterstützt die Entwicklung neuer Overlay-Protokolle, indem es zahlreiche Basis- und Hilfsklassen anbietet.
Mit Hilfe einer grafischen Benutzeroberfläche (s. Abbildung 3.1) kann die Simulation eines Overlay-Netzes im Detail visualisiert und analysiert werden.
Dieses Kapitel gibt einen Überblick über den Overlay-Simulations-Framework OverSim. Zunächst wird die Architektur des Oversim dargestellt. Dabei werden die Konzepte des dem OverSim zugrunde liegenden diskreten Ereignissimulators OMNeT++ vorgestellt. Ein Überblick über den üblichen Simulationsablauf, die verschiedenen austauschbaren Underlay-Modelle und die verwendeten Schnittstellen wird im Folgenden gegeben.

3.1 Architektur

OverSim [15] besitzt eine modulare Architektur, die einem Peer-to-Peer-Netzwerk eine vollständige Abbildung von allen Komponenten ermöglicht. Die einzelnen Komponenten werden im Folgenden näher erläutert. Die Abbildung 3.2 gibt eine Überblick über die Architektur von OverSim.

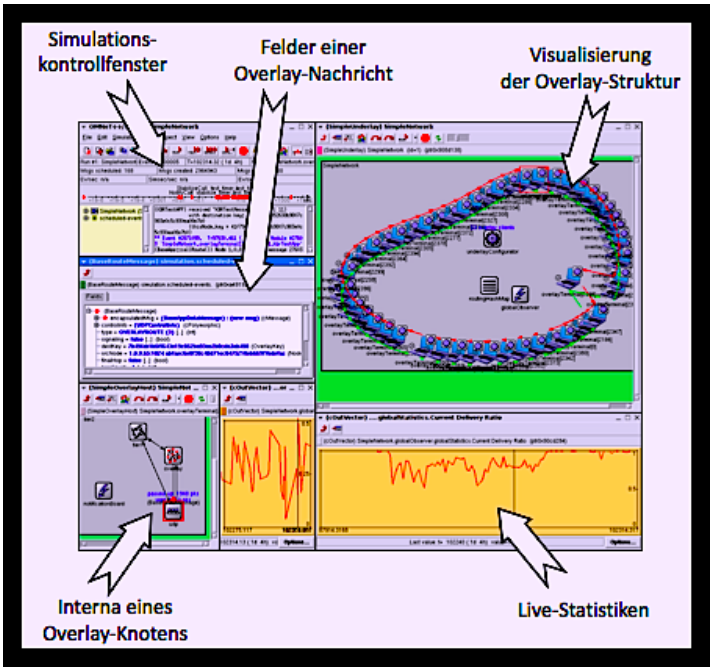


Abbildung 3.1: Grafische Benutzeroberfläche (GUI) des Overlay-Frameworks OverSim [5]

3.1.1 Simulationskern OMNeT++

OMNeT++ [5, 6]. ist wie OverSim quelloffen und verfügt über eine flexible modulare Architektur, welche sich in der Verwendung bereits in C++ geschriebenen Modulen spiegelt. Diese Module sind in der Simulationsklassenbibliothek abgelegt und können beim Aufbau komplexer Systeme verwendet werden.

Der OMNeT++ bereitgestellte Mechanismus hat die Umsetzung der modularen OverSim Architektur stark vereinfacht. Dabei wird die Beschreibungssprache NED verwendet, um Module zu definieren und miteinander zu verknüpfen. Die Module in Omnet++ [16] können sowohl als atomare Module (auch Simple Modules), als auch zusammengesetzte Module (Compound Modules) dargestellt werden. Die Kommunikation unter den Modulen erfolgt durch Versenden von Nachrichten über Kanäle zu diskreten Zeitpunkten.

Mittels der NED-Beschreibungssprache können neben Modulparametern auch Ein- bzw. Ausgabeschnittstellen (Gates) beschrieben werden.

Die Nachrichten können auch als Selfmessage verschickt werden, d. h. ein Modul sendet zu einem Zeitpunkt in der Zukunft an sich selbst eine Nachricht, welche zur Modellierung von Zeitgebern verwendet wird.

Für Nachrichten gibt es auch eine eigene Beschreibungssprache (MSG), mit welcher die Nachrichten-Felder definiert werden. In jedem Modul, welches eine Nachricht empfängt, wird die Methode (handelMessage()) aufgerufen, um die ankommende Nachricht zu verarbeiten.

3.1.2 Underlay Abstraktion

OverSim bietet dem Anwender mehrere austauschbare Underlay Modelle, die die Anbindung an die eigentliche Kommunikation darstellen. Ein Austausch der Underlay-Abstraktion wird nur per Parameter in der Konfigurationsdatei realisiert. Änderungen im Overlay-Protokoll oder Applikation müssen nicht vorgenommen werden.
Es gibt im OverSim drei Underlay-Modelle :
Simple Underlay, Inet-Underlay und Single-Host-Underlay.

Im Folgenden werden die drei Modelle kurz vorgestellt.

Simple Underlay

Das Simple Underlay Modell [15] stellt sich als hochskalierbares Modell zur Simulation von bis zu 100.000 Overlay-Knoten dar. In diesem Modell funktioniert die Simulation auf Nachrichtenebene (message level simulation) und nicht auf Paketebene, d. h. der Nachrichtenaustausch ist direkt zwischen zwei Knoten [5]. Dabei werden die Netzwerkschicht und die darunter liegenden Schichten abstrahiert. Zudem wird durch die Abstraktion die Simulationsgeschwindigkeit deutlich.
Wenn ein Overlay Knoten im SimpleUnderlay erzeugt wird, wird ein zufälliger Punkt aus einem zuvor eingegrenzten Bereich oder aus einer zuvor eingelesenen XML-Datei im mehrdimensionalen Koordinatenraum zugeordnet. Dadurch wird die Paketverzögerung aus dem Abstand der Overlay-Knoten in dem euklidischen Koordinatensystem bestimmt.
Das Simple-Underlay-Modell bietet im Allgemeinen eine effiziente Modellierung realistischer Verzögerungen und eine einfache Möglichkeit für die Modellierung der Knotenmoblität durch einfache Zuweisung neuer Koordinaten, Zugangsnetzeigenschaften und jeweils ein neuen IP-Adressen.

Inet-Underlay

Das Inet-Underlay Modell verwendet das INET-Framework für OMNeT++. Dabei wird für jede Overlay-Knoten alle Netzwerkschichten modelliert. Das INET-Framwork bietet eine Sammlung von Internet-Protokollmodellen einschließlich IP, ICMP, UDP und TCP, die die Modellierung des Underlay-Backbones auf Routerebene mit allen Netzwerkschichten inkl. Sicherungsschicht, heterogenen Zugangsnetzen mit Access-Routern und allen Zwischensystemen (Router) im Netzwerk ermöglicht [5, 6].

Single-Host-Underlay

Das Underlay-Modell [5] bietet die Möglichkeit, dass mehrere Oversim-Instanzen über ein echtes Netzwerk miteinander kommunizieren. Dabei wird jeder Oversim-Instanz nur einen einzelnen Knoten emuliert.
Der Einsatz des Oversim in echten Netzwerken unterscheidet sich von einer Simulation, da die Ereignisse aus der Ereigniswarteschlange während der Simualtionszeit so schnell wie möglich abgearbeitet werden. Somit entspricht die Simulationszeit nicht der Realzeit. Deshalb wird für die Single-Host der OMNeT++-Scheduler durch einen eigenen Scheduler ersetzt. Dadurch werden Simulationszeit und Realzeit synchronisiert [5, 6].

3.1.3 Churn-Generator

OverSim bietet eine Reihe von Churn-Modellen [6], die zur Erzeugung verschiedener Formen von Knotenfluktuation zuständig ist. Diese Knotenfluktuation bzw. ständiger Wechsel der am Overlay beteiligten Knoten, wird als Churn bezeichnet. In OverSim gibt es vier Typen von Churn-Generatoren: RandomChurn, LifetimeChurn, ParetoChurn und NoChurn.

Im Folgenden werden diese Typen kurz vorgestellt:

- **RandomChurn**:
  ermöglicht in periodischen Intervallen einen zufälligen Knoten aus dem Overlay-Netz zu entfernen, dem Overlay-Netz hinzuzufügen oder in ein anderes Zugangsnetz zu migrieren.

- **LifetimeChurn**:
  Jeder neu beitretende Knoten bekommt mit diesem Modell eine zufällige Lebenszeit, die auf einer gegebenen Wahrscheinlichkeitsverteilung basiert. Bei Ablauf der Zeit verlässt der Knoten das Netz, und nach Ablauf einer darauf folgenden Totzeit wird ein neuer Knoten dem Netz beitreten. Die Totzeit basiert dabei auf derselben Wahrscheinlichkeitsverteilung.

- **ParetoChurn**:
  Dieses Modell erzeugt eine ähnliche Knotenfluktuation wie die des LiftimeChurn. Der Churn basiert allerdings auf der Pareti-Verteilung und auf einem 2-stufigen Verfahren.

- **NoChurn**:
  Dieser Churn-Generator generiert keine Knotenfluktuation. Das Overlay-Netz bleibt also nach der Aufbauphase in einem stabilen Zustand. Alle diese Typen können einfach ausgetauscht und miteinander kombiniert werden.

3.1.4 Overlay-Protokolle

OverSim [5, 6] stellt dem Entwickler eine Reihe von gemeinsamen benötigten Funktionalitäten und Schnittstellen, die protokollunabhängig sind, zur Verfügung, um die Implementierung neuer Overlay-Protokolle zu unterstützen und dessen Vergleichbarkeit zu vereinfachen. Dazu zählen:

- Bootstraping
- Statistikdaten sammeln
- Remote Procedure Call (RPC)-Schnittstelle
- Visualisierung der Overlay-Topologie
- Generische iterative und rekursive Routingverfahren

RPC-Dienst stellt eine einheitliche Schnittstelle zum Senden und Empfang von Remote Procedure-Calls (RPCs) zur Verfügung.
Dabei werden asynchrone Nachrichtenübertragungen verwaltet und Fehler aufgehoben und mögliche Sendewiederholungen übernommen.

Der RPC-Protokoll spielt vor allem in strukturierten Overlay-Netzen eine wichtige Rolle,.
Dabei verbessert die effiziente und schnelle Erkennung von Nachrichtenverlusten das Bewerten von Routinglatenz.

Mit RPC können die Nachrichten sowohl direkt als auch durch das Overlay ausgetauscht werden. Dabei werden die Nachrichten statt an eine Transportadresse an einen Schlüssel gesendet. D. h. die Lookup Anfrage nach einem Schlüssel wird dadurch unabhängig vom Overlay-Protokoll erledigt.

Wenn zum Beispiel im Chord ein Overlay-Knoten A erzeugt wird, sendet dieser zu einem Bootstrap-Knoten B eine Call-Nachricht. Dabei übernimmt die RPC-Schnittstelle die Lookup -Aufgabe, um den zuständigen Knoten für den anfragenden Schlüssel zu finden.
Die dafür eingesetzte Methode ist:
sendRouteRpcCall():
Die der Methode übergebenen Parameter sind u. a. die Zielkomponente, der gewünschte Routing-Modus bzw. Transport-Modus, ein Zielschlüssel und/oder eine Transportadresse und die RPC-Call Nachricht.

Der Einsatz eines gemeinsamen Lookup-Verfahrens ermöglicht die Verbesserung der Wartbarkeit und Korrektheit der Implementierung, was zu einer Verbesserung der Vergleichbarkeit führt [5, 6].

Allen simulierten Overlay-Knoten steht ein zentrales Global Observer Modul zur Verfügung. Dieses Compound Modul besteht aus vier atomaren Modulen: GlobalNodeList, GlobalStatistics, GlobalParameters und TraceManager.

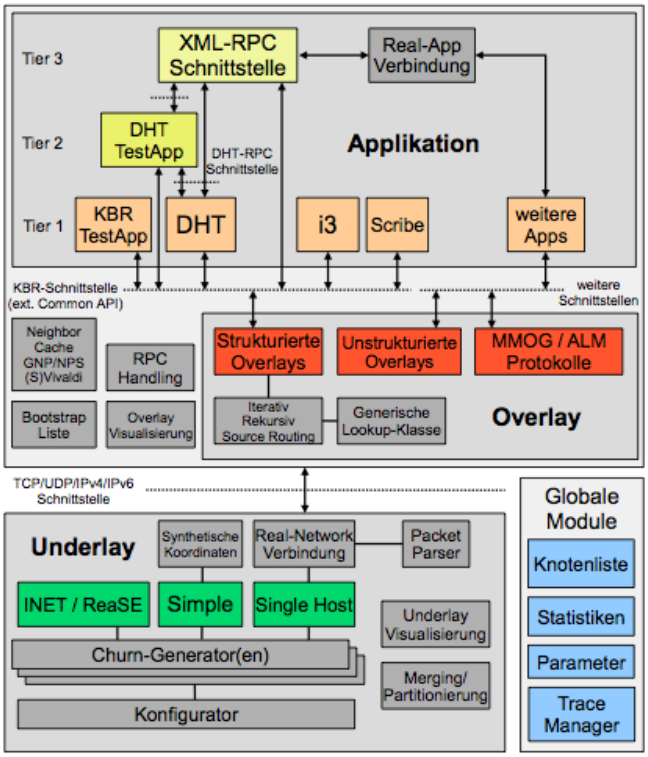


Abbildung 3.2: Architektur des Overlay-Frameworks OverSim [5]

Diese Module werden im Folgenden kurz erläutert:

GlobalNodeList :
Diese Liste speichert alle erzeugten Overlay-Knoten und deren Merkmale (Node ID, NodeHandle, TransportAdress..... usw.) und stellt für BootstrapList zufällige Bootstrap-Knoten bereit. Dieses Modul wird sowohl von Underlayconfigurator als auch von Overlay-Protokoll verwendet.
Globalstatistics :
In diesem Modul erfolgt die globale Erfassung und Sammlung von Statistikdaten. Während der Simulation werden alle aggregierten Daten aller Knoten an dieses Modul übertragen und in den Ergebnisdateien gespeichert.
GlobalParameters:
Mit diesem Modul können für alle Overlay-Knoten gültige Parameter, die während der Simulation eingesetzt wurden, gespeichert werden.
TraceManager:
Dieses Modul kann Trace-Dateien einlesen, die Ereignisse speichern, die zu festgelegten Zeitpunkten

ausgeführt werden sollen.
Methoden all dieser Module können von allen Overlay-Knoten zur Simulationslaufzeit aufgerufen werden.

3.2 Simulationsablauf

In diesem Abschnitt folgt zunächst ein Überblick über die Konfiguration. Dabei werden die Schritte, die notwendig sind, um ein Parameter zu konfigurieren, anhand eines Beispiels veranschaulicht. Die verschiedenen Phasen einer Simulation werden im Folgenden repräsentiert.

3.2.1 Konfiguration

Die Simulation in Oversim benötigt zwei Konfigurationsdateien omnetpp.ini und default.ini. In omnetpp.ini werden für jedes Overlay-Protokoll die verschiedenen Underlay-Modelle beschrieben. Dabei werden hauptsächlich der Churn -Type und zwei Knoten-Parameter (overlayType und tier1Type) festgelegt (s. Beispiel Chord im Listing 3.1).

```
1 [Config Chord]
2 description = Chord (iterative, SimpleUnderlayNetwork)
3 *.underlayConfigurator.churnGeneratorTypes = "oversim.common.NoChurn"
4 **.overlayType = "oversim.overlay.chord.ChordModules"
5 **.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
```

Listing 3.1: Die eingesetzten Parameter für Chord-Overlay in omnetpp.ini

Mit defaut.in werden alle Laufzeitparameter, die vor Begin der Simulation eingelesen werden, eingegeben. Diese Parameter müssen zuerst in der NED-Datei definiert werden, damit sie in der Klasse, die diese Datei implementiert, extrahiert werden.
Das Beispiel im Listing 3.2 zeigt manche Laufzeitparameter, die in dieser Arbeit in Chord eingeführt wurden. Dabei sind die benötigten drei Schritte, um Parameter zu konfigurieren, illustriert.

```
1
2 1) Die Parameter in chord.ned definieren
3
4     bool activeList;
5     bool passiveList;
6     bool manuell;
7     bool zip;
8     bool reunionUnification;
9     int nodeCount;
10    bool ringReunion;
11    bool registrierPointer;
12    bool pointerVergleiche;
13    double pointerdelay;
14    bool distributeConf;
15
16 2) Die Werte in default.ini eingeben
17
18    **.overlay*.chord.manuell =true
19    **.overlay*.chord.zip =false
20    **.overlay*.chord.passiveList=false
21    **.overlay*.chord.activeList=false
22    **.overlay*.chord.reunionUnification=false
23    **.overlay*.chord.ringReunion=true
24    **.overlay*.chord.distributeConf =true
25    **.overlay*.chord.nodeCount=1000
26    **.overlay*.chord.pointerdelay=20
27
28 3)Die Parameter in chord.cc einlesen
29
30    zip=par("zip");
31    activeList=par("activeList");
32    passiveList=par("passiveList");
33    manuell=par("manuell");
34    reunionUnification=par("reunionUnification");
35    ringReunion=par("ringReunion");
36    nodeCount=par("nodeCount");
37    pointerdelay=par("pointerdelay");
38    distributeConf=par("distributeConf");
```

Listing 3.2: Beispiele von in dieser Arbeit gesetzten Parameter

3.2.2 Simulationsphasen

Ein Simulationsdurchlauf startet mit Einlesen der Konfigurationsdateien omnetpp.ini und default.ini, um alle Laufzeitparameter zu setzen.

Initialisierungsphase (init phase)

In dieser Phase werden die Overlay-Knoten nacheinander erzeugt. So werden die Knoten in der Initialisierungs- oder Netzaufbauphase periodisch nacheinander dem Netzwerk beitreten.
Mit dem **initPhaseCreationInterval** Parameter wird das Intervall zwischen zwei Knotenerzeugungen in default.ini festgelegt. Die Anzahl der Knoten in Overlay-Netzwerk wird ebenfalls mit dem Parameter **targetOverlayTerminalNum** gesetzt.
Diese Phase wird dann mit dem Erzeugen des letzten Overlay-Knotens beendet.
Jede Klasse in OverSim implementiert eine Initialisierungsmethode, die die Parameter von den Konfigurationsdateien einliest und die verwendeten Variablen initialisiert.
Beispielsweise wurden die in dieser Arbeit eingeführten Szenariodateien im SimpleUnderlay in der Methode SimpleUnderlayConfigurator::initializeUnderlay (int stage) eingelesen und in den dafür reservierten Listen bzw. Containern gespeichert, damit sie während der Simulation verwendet werden können.

Übergangsphase

Diese Phase bezeichnet den Zeitraum zwischen dem Netzwerkaufbauphasenende und Simulationsende. Der Dauer dieser Phase wird mit dem Parameter **TransitonsTime** festgelegt. Dieser Parameter ist in der Datei UnderlayConfigurator.ned definiert und wird in der Klasse UnderlayConfigurator.cc eingelesen. Die oben beschriebenen Churn-Generatoren werden in dieser Phase mit der Generierung von Knotenfluktuation starten. Die Statistikdateien werden in dieser Phase nicht erfasst.

Datenerhebungsphase

In dieser Phase werden die statistischen Daten gesammelt, und bei Simulationsende einem globalen Modul (GlobalStatistics) übergeben. Mit dem **measurementPhase** Parameter wird dabei die Dauer dieser Phase festgelegt.
Wenn dieser Parameter nicht angegeben wurde, wird bis zu einem manuellen Abbruch weiter simuliert.
Jede Klasse kann eine finish-Methode, die das Ende der Simulation bezeichnet, implementieren.
Zum Beispiel implementiert Chord. cc die Methode `Chord::finishOverlay()` und SimpleUnderlayConfigurator.cc verwendet `SimpleUnderlayConfigurator::finishUnderlay()`.
Dabei werden die Funktionen von dem GlobalStatistic Modul aufgerufen(s. Listing 3.3), das die Daten auffasst und aggregiert. Diese Daten werden dann nach Ende der Simulation in verschiedenen Dateien gespeichert.

Die Statistikdateien werden in OverSim/simulations/results angelegt. Es muss aber vor Beginn der Simulation der Parameter `*.globalObserver.globalStatistics.outputStdDev` in default.ini auf true gesetzt werden, damit die Dateien erstellt werden.

```
1  void Chord::finishOverlay()
2  {
3    bootstrapList->removeBootstrapNode(thisNode);
4    simtime_t time = globalStatistics->calcMeasuredLifetime(creationTime);
5       if (tim <GlobalStatistics::MIN_MEASURED)
6        return;
7
8     globalStatistics->
9     addStdDev("Chord:Sent JOIN Messages/s",joinCount/time);
10    globalStatistics->
11    addStdDev("Chord:Sent NEWSUCCESSORHINT Messages/s",newsuccessorhintCount/time
          );
12    globalStatistics->
13    addStdDev("Chord:Sent STABILIZE Messages/s",stabilizeCount/time);
14    globalStatistics->
15    addStdDev("Chord:Sent NOTIFY Messages/s",notifyCount/time);
16    globalStatistics->
17    addStdDev("Chord:Sent FIX_FINGERS Messages/s",fixfingersCount/time);
18    globalStatistics->
19    addStdDev("Chord:Sent JOIN Bytes/s",joinBytesSent/time);
20    globalStatistics->
21    addStdDev("Chord:Sent NEWSUCCESSORHINT Bytes/s",newsuccessorhintBytesSent/
          time);
22    globalStatistics->
23    addStdDev("Chord:Sent STABILIZE Bytes/s",stabilizeBytesSent/time);
24    globalStatistics->
25    addStdDev("Chord:Sent NOTIFY Bytes/s",notifyBytesSent/time);
26    globalStatistics->
27    addStdDev("Chord:Sent FIX_FINGERS Bytes/s",fixfingersBytesSent /time);
28    globalNodeList->print();
29 }
```

Listing 3.3: finishOverlay Methode im Chord

3.3 Zusammenfassung

Das im Rahmen dieser Arbeit verwendete OverSim-Framework bietet eine Vielzahl an Eigenschaften, die OverSim zu einem hervorragenden Evaluierungswerkzeug zur Analyse von Peer-to-Peer- und Overlay-Netzen macht. Das Framwork stellt eine Reihe von Funktionen für Nachrichtenverwaltung, Visualisierung und generische Lookup-Verfahren bereit. Dadurch können neue Protokolle schnell implementiert und evaluiert werden. Dabei spielt der modulare Aufbau eine wichtige Rolle, da einzelne Komponenten flexibel erweitert oder neuintegriert bzw. ausgetauscht werden. Zusätzlich bietet OverSim eine große Anzahl implementierter Overlay-Protokolle und eine umfangreiche Statistikerfassung. Aufgrund seiner Vorteile und seiner Eigenschaften als Evaluierungswerkzeug für Peer-to-Peer-und Overlay-Netze wird OverSim für die Realisierung und Evaluierung der in dieser Arbeit vorgestellten Konzepte verwendet. Die allgemeinen Eigenschaften und dafür notwendigen Erweiterungen, wie die Integration der Netzwerkpartitionierung im Underlay und der Implementierung der Merging Algorithmen im Overlay, werden in den dazugehörigen Kapiteln beschrieben.

Kapitel 4

Design und Implementierung der Netzwerkpartitionierung im Underlay

In diesem Kapitel wird ein Konzept zur Netzwerkpartitionierung im Underlay vorgestellt, sowie seine Realisierung und Implementierung beschrieben. Dabei wurden Events und Gruppenamen eingeführt. Dieses Konzept wurde im Rahmen dieser Arbeit entwickelt, um Knotengruppen im Underlay mit Hilfe des OverSim Framework zu bilden und dabei die Effekte der Isolierung dieser Knotengruppen mit verschiedenen Szenarien auf die Overlay-Topologie zu untersuchen und analysieren.
Ziel dieser Arbeit ist es, mit diesem Verfahren die Implementierung der Merging-Algorithmen (s. Kapitel 2), die genutzt werden, um die getrennten Overlay-Partitionen wieder zu verschmelzen, zu evaluieren. Gleichzeitig werden die Merge Algorithmen genutzt, um die Netzwerkpartitionierung zu verifizieren.
Als Underlay Abstraktion wurde dafür das SimpleUnderlay verwendet. Darin wurde die Partitionierung implementiert und integriert.
Konkret werden folgende OverSim-Module und -Klassen mit den darin bereitgestellten Funktionen verwendet:

- ***SimpleUnderlayConfigurator**:*
  Das Modul bildet Knotengruppen und führt Events aus.

- ***SimpleUDP**:*
  Das Modul steuert die Konnektivität zwischen Knoten aus unterschiedlichen Knotengruppen.

Im Abschnitt 4-1 werden diese oben genannten Module näher erläutert. Weshalb die Modellierung der Overlay-Knoten in OverSim keine direkten Lösung für die Netzwerkpartitionierung bietet, wird im Abschnitt 4-2 kurz erklärt. Im Abschnitt 4-3 wird dann die entwickelte Lösung vorgestellt.

4.1 Beschreibung

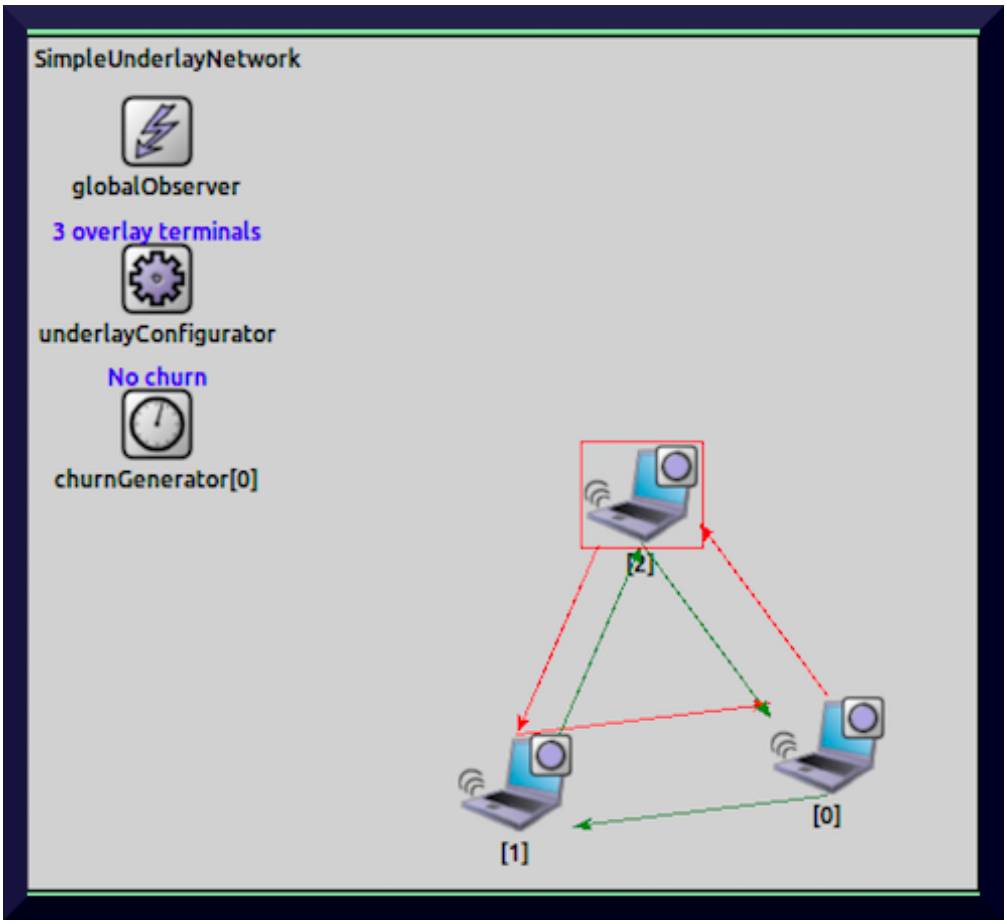


Abbildung 4.1: Die Submodule des SimpleUnderlayNetwork

Zunächst wird der Aufbau des SimpleUnderlay Network beschrieben (s. Abbildung 4.1). Dabei wird die Erzeugung der Overlay-Knoten und die Organisation der Module innerhalb des SimpleUnderlay verdeutlicht. Die Module bzw. die Klassen, die dabei beteiligt sind, werden kurz vorgestellt.
Das Modul SimpleUDP ist auch im SimpleUnderlay implementiert und stellt die Schnittstelle zwischen Underlay und Overlay dar. Darauf wird im Folgenden näher eingegangen.

4.1.1 SimpleUnderlayNetwork

Im Gegensatz zum Inet-Underlay und Single-Host-Underlay wird im SimpleUnderlay das Backbone auf der Routerebene nicht modelliert [5,6]. Somit bietet der SimpleUnderlay in OverSim die einfachste Modellierung.
In diesem Modell werden die Nachrichten direkt von einem Overlay-Knoten zu einem anderen ver-

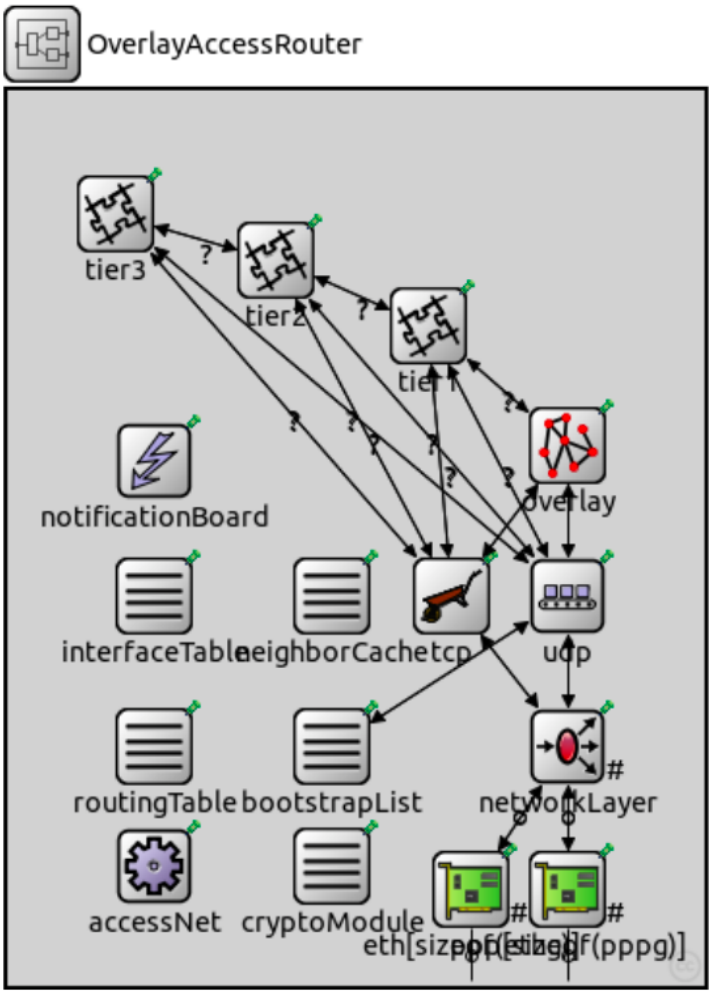


Abbildung 4.2: OverlayAccessRouter in Inet-Underlay

schickt. Dabei wird der Speicheraufwand, der durch die Modellierung der Access Router (s. Abbildung 4.2) hervorgerufen wird, erspart.
Inet-Underlay und Single-Host-Underlay können, im Grunde genommen, verwendet werden, um die Effekte von Overlay-Knoten im Zugangsnetzrouter mit Overlay-Funktionalität zu modellieren [5].
Für das Ziel dieser Arbeit spielen diese Effekte keine entscheidende Rolle. Von daher ist der Einsatz einer einfachen Modellierung wie Simple Underlay für diese Arbeit gut geeignet.

Komponenten des SimpleUnderlay

Im Underlay wird das komplette Design des Framworks entworfen. Mittels der NED-Datei (SimpleUnderlay. ned) werden alle Komponenten definiert (s. Listing 4.1).
Diese Komponenten werden dann in verschiedene Schnittstellen implementiert.

```
network SimpleUnderlayNetwork
{
parameters:
submodules:
  underlayConfigurator: SimpleUnderlayConfigurator {
    @display("p=70, 130");
  }
  churnGenerator[0]: ChurnGenerator {
    @display("p=70, 210, column");
  }
  globalObserver: GlobalObserver {
    @display("p=70, 50");
  }
  OverlayKnoten[0]: SimpleOverlayHost { @dynamic;}
}
```

Listing 4.1: SimpleUnderlay.ned

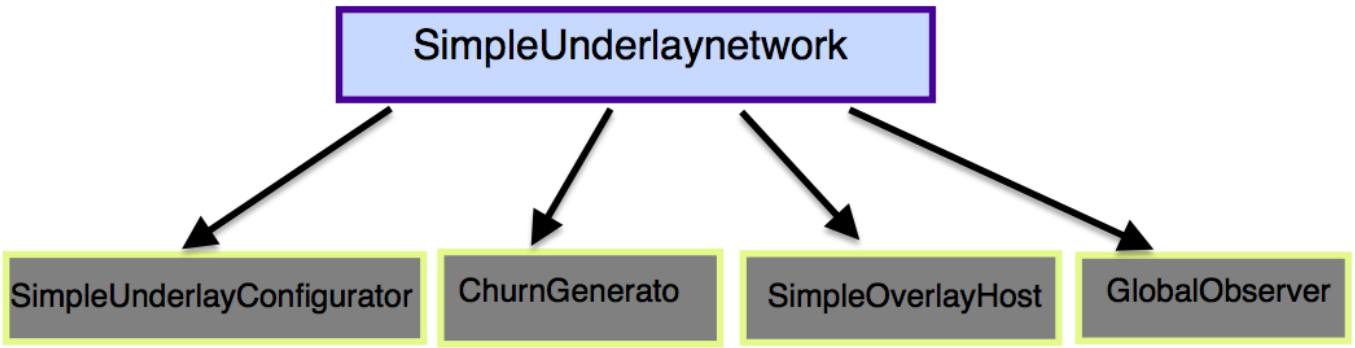


Abbildung 4.3: Komponenten des SimpleUnderlayNetwork

Im Folgenden werden diese Komponenten, die als Submodule im SimpleUnderlay.ned (s. Abbildung4.3) definiert wurden, näher erläutert:

`Der Churngenerator`:

Dieses Modul ist in der Common API implementiert. Der Churngenerator ist für das Erzeugen (bzw. Zerstören) der Overlay-Knoten während der Simulation zuständig.

Dabei werden neben anderen Parametern die Anzahl der Knoten (*.churnGenerator*.targetOverlayTerminalNum) von der Konfigurationsdatei(default.ini) eingelesen, dann wird die Methode CreateNode () aufgerufen, die die Knoten hintereinander erzeugt. Diese Methode ist im SimpleUnderlayConfigurator.cc imple-

mentiert.

`SimpleHostOverlay`:

Das Design von jedem Overlay-Knoten als Compound Modul wird in SimpleHostOverlay (bzw. SimpleMultiOverlay. ned) definiert.

Mit dieser Datei werden die Schichten von jedem Overlay-Knoten beschrieben. Dabei wird auch die Verbindung zwischen jeder Schicht dargestellt (s. Listing 4.3 und Abbildung4.4)).

`SimpleUnderlayConfigurator`:

Dieses Modul ist im SimpleUnderlay implementiert. Die Klasse SimpleUnderlayConfigurator. cc implementiert die Methode CreateNode(). Diese Methode erzeugt die im SimpleHostOverlay.ned beschriebenen Knoten sequentiell. Zunächst wird ein Name für den Knoten definiert. Anhand dieses Namens wird die Omnett++ Methode create() aufgerufen. Somit wird dann ein Compound Modul erzeugt. (s. Listing 4.2)

```
1 cModule* node = moduleType->create(nameStr.c_str(), getParentModule(),
      numCreated +1, numCreated)
```

Listing 4.2: Erzeugen eines Overlay-Knotens

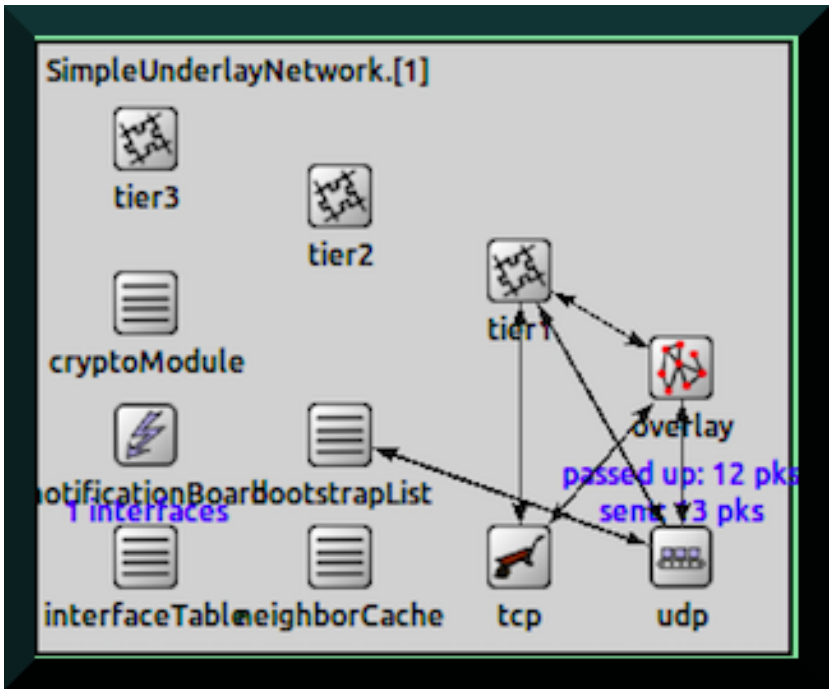


Abbildung 4.4: Komponenten eines Overlay-Knoten

```
module SimpleOverlayHost{
parameters:
  @display("bgb=433, 386;i=device/wifilaptop_l;i2=block/circle_s");
gates:
   input overlayNeighborArrowIn[];
   output overlayNeighborArrowOut[];
submodules:
  tier3: <tier3Type> like ITier {
    parameters:
      @display("p=64, 57;i=block/segm");
  }
  tier2: <tier2Type> like ITier {
    parameters:
      @display("p=179, 90;i=block/segm");
  }
  tier1: <tier1Type> like ITier {
    parameters:
      @display("p=286, 135;i=block/segm");
  }
  overlay: <overlayType> like IOverlay {
    parameters:
      @display("p=382, 193;i=block/network2");
  }
  udp: SimpleUDP {
    parameters:
      @display("p=382, 305");
  }
  tcp: SimpleTCP {
    parameters:
      @display("p=286, 305");
  }
  cryptoModule:CryptoModule {
    parameters:
      @display("p=64, 154");
  }
  notificationBoard:NotificationBoard {
    parameters:
      @display("p=64, 232");
  }
  interfaceTable:InterfaceTable {
    parameters:
      @display("p=64, 305");
  }
  neighborCache:NeighborCache {
    parameters:
      @display("p=179, 305");
  }
  bootstrapList:BootstrapList {
    parameters:
      @display("p=179, 232");
```


Listing 4.3: SimpleOverlayHost.ned

Jedem Konten wird eine IP-Adresse zugewiesen. Zusätzlich bekommt er zufällige Koordinaten aus einem XML Datei. Mit diesen Daten wird ein Eintrag erstellet, welche verwendet wird, um SimpleUDP zu kontaktieren. (s. Listing 4.4)

```
1 SimpleNodeEntry* entry = new SimpleNodeEntry(node, rxChan, txChan,
      sendQueueLength, fieldSize);
2 SimpleUDP* simpleUdpsimpleUdp->setNodeEntry(entry);
```

Listing 4.4: Entry für den Knoten erzeugen und dem SimpleUDP übergeben

`GlobalObserver`:
Dieses Compound Modul ist in der Common API implementiert. Es besteht aus fünf einfachen Module: GlobalNodeList, GlobalParameter, GlobalTraceManager, GlobalFunktion und GlobalStatics (s. Listing 4.5).

```
1    parameters:
2    int numGlobalFunctions;
3    @display("i=block/control");
4   submodules:
5    globalNodeList: GlobalNodeList {
6      parameters:
7        @display("p=60, 60;i=block/control");
8    }
9    globalParameters: GlobalParameters {
10     parameters:
11       @display("p=180, 180;i=block/control");
12   }
13   globalTraceManager: GlobalTraceManager {
14     parameters:
15       @display("p=60, 200;i=block/control");
16   }
17   globalFunctions[numGlobalFunctions]: GlobalFunctions {
18     parameters:
19       @display("p=60, 300, column;i=block/control");
20   }
21   globalStatistics: GlobalStatistics {
22     parameters:
23       @display("p=200, 100;i=block/control");
24   }
25}
```

Listing 4.5: GlobalObserver. ned

Im Global Observer Modul werden globales Wissen über den Simulationsablauf abgelegt. Dabei stellt das Modul GlobalNodeList für neu hinzukommende Overlay-Knoten einen zufälligen Bootstrap-Knoten bereit. Mit dem GlobalStatistics werden die Statistiksdaten erfasst und gespeichert.

4.1.2 Simple Module SimpleUDP

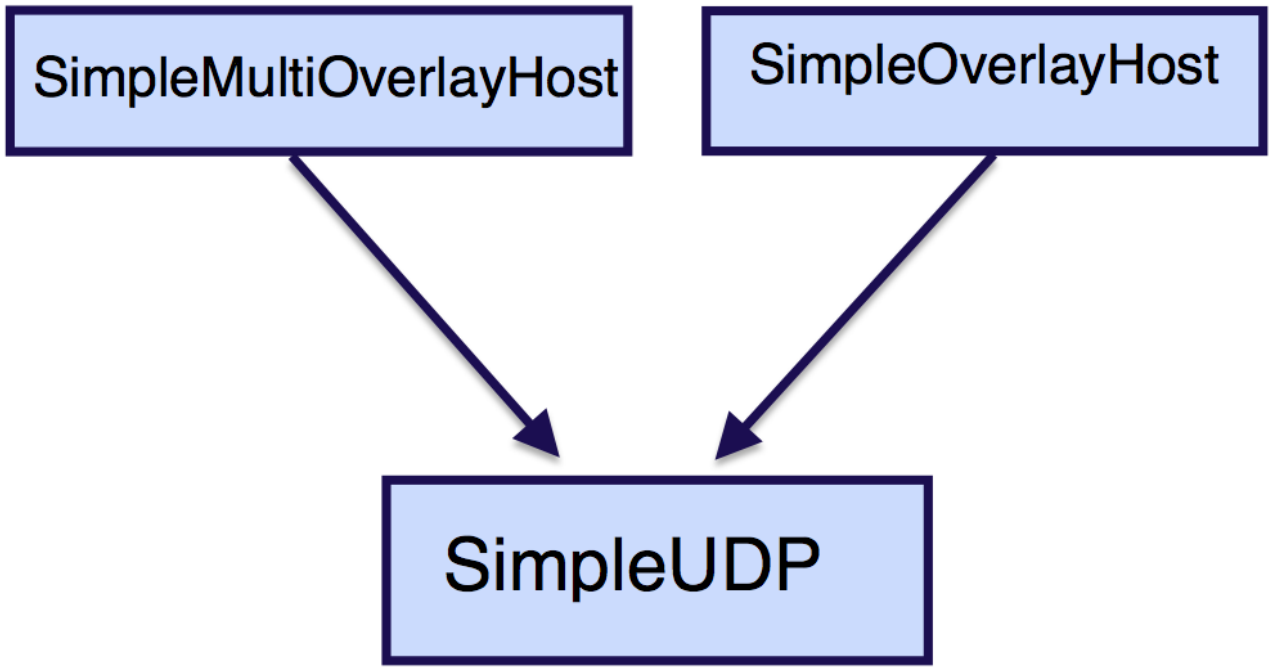


Abbildung 4.5: SimpleUDP ist in SimpleMultiOverlayHost und SimpleOverlayHost NED-Dateien definiert

SimpleUDP ist als einfaches Modul im SimpleUnderlay implementiert (s. Abbildung 4.5). Das Modul ist eine Komponente des Overlay-Knoten (s. Abbildung 4.4) Dabei verbindet das Modul die Overlay-Schicht mit dem Underlay Schicht.

SimpleUDP(oder Simple TCP, wenn TCP anstatt UDP eingeschaltet ist) ist vom UDP Modul abgeleitet, das die Transport-Schicht für alle Underlay-Implementierungen in OverSim abbildet. Das Modul bietet alle wichtigen Methoden, um Nachrichten weiterzuleiten.

UDP ist im inet/src/transport implementiert. Dabei beschreibt die Datei UDP.ned (s. Listing 5.1) die Kommunikationsschnittstellen (Gates), die für die Verbindung zwischen dem Overlay und dem Underlay verwendet werden können.

```
1
2 simple UDP
3{
4
5  parameters:
6
7    @display("i=block/transport");
8
9  gates:
10
11    input appIn[] @labels(UDPControlInfo/down);
12    input ipIn @labels(UDPPacket, IPControlInfo/up);
13    input ipv6In @labels(UDPPacket, IPv6ControlInfo/up);
14    output appOut[] @labels(UDPControlInfo/up);
15    output ipOut @labels(UDPPacket, IPControlInfo/down);
16    output ipv6Out @labels(UDPPacket, IPv6ControlInfo/down);
17
18}
```

Listing 4.6: UDP.ned

SimpleUDP verwendet nur zwei von den im Listing beschriebenen Gates:
appIn für die Kommunikation mit der Applikation und ipOut für die Kommunikation mit anderen Overlay-Knoten (s. Abbildungen 4.6 und 4.7)
ipOut und appOut sind in Inet-Underlay und Single- Host-Underlay eingesetzt, um einen Access Router modellieren.
Dabei steht `appOut` für die Kommunikation zwischen dem Router und der Applikation, während `ipOut` für die Verbindung zwischen dem Underlay und dem Router steht.
Mit dem SimpleUDP Modul werden dann im SimpleUnderlay alle Typen von Nachrichten weitergeleitet:

- `Applikationsnachrichten`:
  Diese erreichen das `AppOut` Gate und werden mit dem Einsatz der von UDP bereitgestellten Protokoll UDPControlInfo weitergeleitet.

- `Knotennachrichten`:
  Sind die zwischen Knoten ausgetauschten Nachrichten. Die erreichen das `IpIn` Gate und werden direkt dem Zielknoten eingereicht.

- `SelfMessage`:
  Sind Nachrichten, die ein Knoten an sich selbst schickt, um ein bestimmtes Ereignis zu einem

festgelegten Zeitpunkt auszuführen.

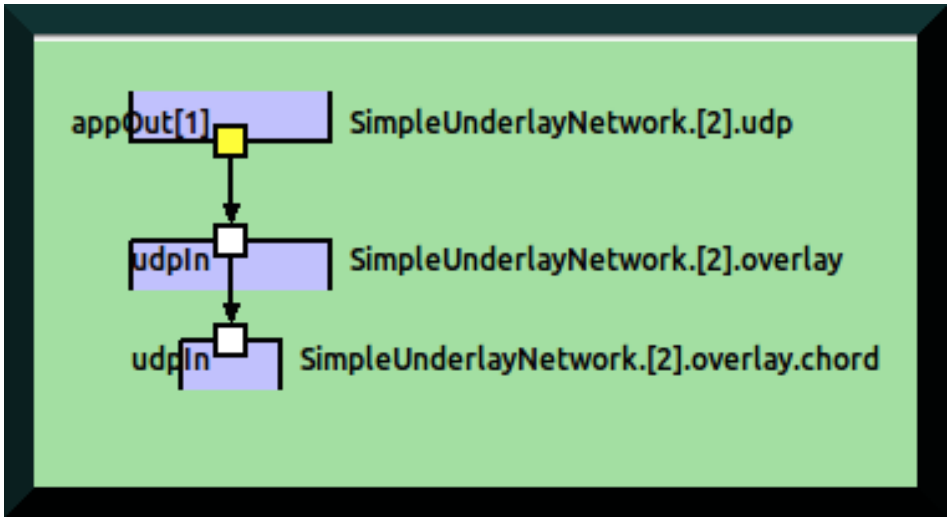


Abbildung 4.6: SimpleUDP appOut Gate

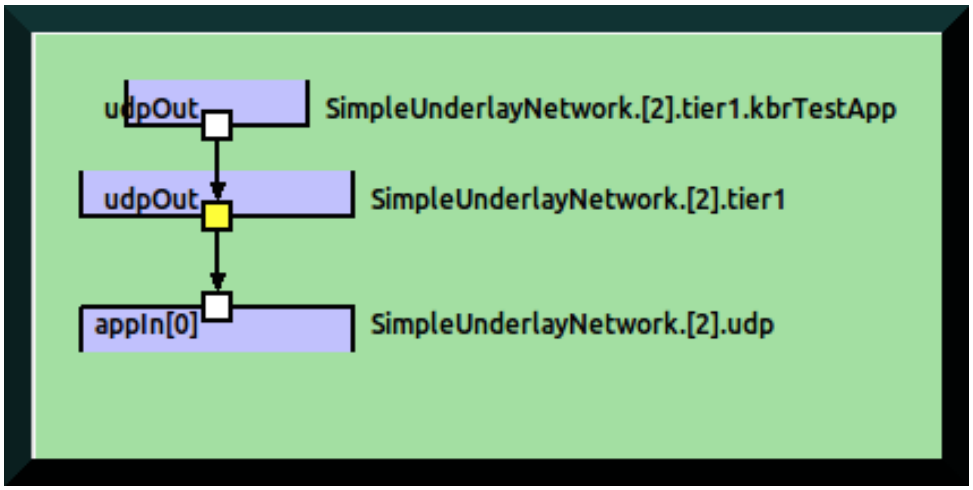


Abbildung 4.7: SimpleUDP appIn Gate

4.2 Problematik

4.2.1 Modellierungsprobleme in OverSim

OverSim

stellt jeden Knoten als Compound Modul dar. Dabei wird ein Schichten-Modell eingesetzt, d. h jede Schicht wird als einfaches Modul repräsentiert.

Mit den NED-Dateien wird genau beschrieben, wie die Schichten miteinander kommunizieren können.

Der Nachteil dieses Aufbaus besteht darin, dass die Netzwerkverbindung zwischen verschiedenen Overlay-Knoten nicht dargestellt wird. Somit besteht keine Möglichkeit, um die Konnektivität zwischen verschiedenen Knoten direkt auf der Ebene der Netzwerkschicht einzuschränken.

PeerfactSim [12]
Overlay-Simulator in Java, der sich ebenfalls an der Common-API-Schnittstelle orientiert, ist in der Lage, verschiedene Gruppen zu einem gegebenen Zeitpunkt für eine festgelegte Zeitdauer zu isolieren.

4.3 Design der Netzwerkpartitionierung im Underlay

Der folgende Entwurf ist nach einer gründlichen Analyse und dem Verständnis von allen Klassen im SimpleUnderlay entstanden.
Vor allem die CreateNode() Methode, die in der Klasse SimpleUnderlayConfigurator.cc implementiert ist, wurde intensiv untersucht, da diese alle wichtigen Eigenschaften und Einträge für den Overlay-Knoten bereitstellt. Von dort werden auch alle anderen Klassen im SimpleUnderlay aufgerufen.
Die Funktionen, die da aufgerufen werden, wurden auch in jeweiligen Klassen analysiert. Nur dadurch wird verstanden, wie die SimpleUDP-Schnittstelle verwendet wird, um die Verbindung zwischen den Knoten herzustellen.
Dieser Aufwand war nötig, da keine ausreichenden Quellen dazu gefunden wurden. Die Schwierigkeit dabei war, dass viele Methoden und Instanzen von Omnet++ übernommen worden sind, und dass die einzige Quelle hierfür die Omnett++ Dokumentation ist. Im Folgenden wird eine mögliche Lösung zur Einführung von Netzwerkpartitionierung vorgestellt.

4.3.1 Erweiterung des SimpleUnderlayConfigurator Moduls

Das SimpleUnderlayConfigurator Modul ist von der Klasse UnderlayConfigurator abgeleitet. Für jedes Underlay Modell existiert ein von dieser Klasse erbendes Modul.
Die Methoden, die in der Klasse SimpleUnderlayConfigurator.cc erweitert wurden, sind:

- **initializeUnderlay ()**:
  Diese Methode liest die initialisierten Parameter von der Konfigurationsdatei default.ini ab.
  Darin werden auch die für diese Arbeit integrierten Dateien (Partition- und Event-Dateien) eingelesen.
  Die extrahierten Daten werden dann in einem assoziativen Container (map) gespeichert.

- **TransportAddress*createNode** ():
  Die Methode wurde von der Klasse UnderlayConfigurator.cc geerbt.
  Hauptsächlich wird diese Methode von der ChurnGenertaor.cc aufgerufen. In dieser Methode werden die Overlay-Knoten hintereinander kreiert.
  Dabei werden alle Eigenschaften und Einträge, die ein neuer integrierter Knoten braucht, um einem Overlay-Netz beizutreten, bereitgestellt.

Diese Methoden wurden neu integriert:

- **handleEventsMessage()**:
  Ist eine neue integrierte Methode, welche die verschickten SelfMessages in CreateNode() bearbeitet. In dieser Methode werden die Ereignisse, die aus der Event-Datei eingelesen wurden, umgesetzt.

- **readNextEvent()**:
  Diese Methode liest die Eventzeilen, die in einem Container im InitializeUnderlay() von der Event-Datei eingelesen wurden. Die Methode wird anhand einer SelfMessage-Nachricht, die schon in der Initialisierungsphase gestartet und periodisch verschickt wurde, aufgerufen.

- **scheduleNextEvent(int groupNum, char *event,double zeit)**:
  Mit dieser Methode werden die Events synchronisiert. Jedes Mal, wenn ein Event eingelesen wurde, wird eine Nachricht dynamisch erzeugt, die dieses Event enthält. Somit werden verschiedene Events zu verschiedenen Zeiten realisiert.

4.3.2 Szenarioeingabedateien

Mittels zweier Eingabedateien (Partition und Event) wird die Partitionierung im Simple-Underlay gestaltet und konfiguriert. Beliebige Knotengruppen mit verschiedenen Eigenschaften können gebildet werden.
Die Netzwerkkonnektivität zwischen verschiedenen Knotengruppen ist dabei gut konfigurierbar, da die Dateien die Möglichkeit bieten, verschiedene Zeiten für jedes Event zu bestimmen.
Somit könnten auch komplizierte Szenarien getestet und bewertet werden. Anpassungen beim Simple-Underlay-Protokoll muss nicht vorgenommen werden.
Diese Dateien sind in dem Simulationsordner angelegt, damit sie direkt während der Simulation eingelesen werden können. Sie können aber auch woanders gespeichert werden. Es muss nur der kompletten Pfad berücksichtigt werden.

Partition-Datei

Um die Partitionierung im Underlay zu realisieren, wurden anhand einer Eingabe-Datei (Partition-Datei) Knotengruppen gebildet.
Die integrierte Datei (Partition-Datei) enthält die Namen der gewünschten Gruppen und die Anzahl der Knoten in jeder Gruppe.

Diese Datei wird im SimpleUnderlayConfigrator.cc in der Methode initializeUnderlay (int stage) zeilenweise eingelesen und in einem Container (map) gespeichert (s. Listing 4.7).
Dabei stellt jeder Zeile eine Gruppe dar.

```
1  ifstream partFile(par("partitionFile"), ios::in);
2  if (!partFile) {
3      cout<< "Partition File not found"<< endl;
4   } else {
5    while (partFile >> name>> anteil) {
6      Region region;
7      strcpy(region.Name, name);
8      region.Anteil=anteil;
9      region.status=status;
10     region.TrennungsZeit=trennungsZeit;
11     reg.insert(make_pair(r, region));
12     r++;
13    }
14   partFile.close();
15   }
```

Listing 4.7: Partition-Datei

Mit der CreateNode() Methode werden die Gruppen hintereinander erzeugt. Dabei werden die gespeicherten Daten von jeder Gruppe (Gruppennummer, Anzahl der Knoten und Status offline oder online) eingelesen. (s. Listing 4.8). Diese Daten werden dann dem SimpleUDP Modul zugewiesen. (s. Listing 4. 7 Zeilen 37 und 38).

```
1    std::string nameStr;
2  SimpleNodeEntry* entry;
3  cModuleType* moduleType = cModuleType::get(type.terminalType.c_str());
4
5  std::map<int, Region>::iterator it =region.begin();
6  while (it != reg.end()) {
7   if(gruppe==l->first) {
8     nameStr=it->second.Name;
9     anzahl=it->second.Anteil;
```

```
10      status=it ->second. status;
11      trennungsZeit=it ->second.TrennungsZeit;
12      break;
13    }
14     it++;
15  }   SimpleUDP* simpleUdp = check_an_dcast <SimpleUDP*> (node->getSubmodule("udp")
        );
16   simpleUdp->setNodeEntry(entry, satus, gruppe);
```

Listing 4.8: Einlesen der Gruppeninformationen in der CreateNode() Methode

Event-Datei

Mit einer `Event`-Datei werden beliebige Ereignissen definiert. Diese werden während der Initialisierung (initializeUnderlay ()) vom SimpleUnderlayConfigurator.cc in einem Vector extrahiert (s. Listing 4.9).
Beim Erzeugen der Knoten in der Methode CreateNode() werden dann die Events eingelesen.

```
1    ifstream eventFile("eventFile", ios::in);
2   if (!eventFile) {
3    std::cout<< "Event File not found"<< std::endl;
4   } else {
5     while (eventFile >>groupNum>>event>>zeit ) {
6      Event e;
7      strcpy(e.event,event);
8      e.gruppeNum=groupNum;
9      e.zeit=zeit;
10     eve.insert(make_pair(pos,e));
11     pos++;
12    }
13   eventFile.close();
14   }
15 nextEvent = new cMessage("NextEvent");
16 scheduleAt(simTime()+1, nextEvent);
```

Listing 4.9: Event-Datei

Die `Event`-Datei kann für jede Gruppe die Netzwerkaustrittzeit (Offline) und/oder Netzwerkbeitritt (Online) bestimmen.
Jede Zeile in der Datei definiert ein Event. Dabei wurden Gruppennummer, Eventnachricht und Zeit definiert. In der Initialisierungsmethode wurde eine SelfMessage "nextEvent"gestartet (s. Listing 4.9 Zeilen 15-16). Die "nextEvent"Nachricht weist daraufhin, dass die Methode readNextEvent() ein

Event einliest, so dass einer Zeile in der Event-Datei entspricht, von der der Container, der den Inhalt dieser Datei umfasst, einliest. Dabei wird die SelfMessage "nextEvent"wieder verschickt und die Methode scheduleNextEvent() ausgerufen. Diese Methode erzeugt dynamisch Nachrichten mit der gelesenen Eventsnachricht. Mit jedem neuen Event wird eine neue Nachricht erzeugt, die dann in der Methode handleEventsMessage bearbeitet wird. Die Aktionen werden dann umgesetzt, indem für jede eingelesene Gruppe eine SelfMesssage in der angegebenen Zeit mit dem entsprechenden Ereignis verschickt wird.
Mit der SelfMessage werden dem SimpleUDP Modul neue Informationen über die Gruppen übergeben. Deswegen wird die Liste, die diese Informationen enthält, jedes Mal aktualisiert.
Somit werden die Knoten in jeder Gruppe mit dem aktuellen Status (Offline oder Online) behandelt.
Mit der neuen integrierten Methode handleEventsMessage() werden die SelfMessages, die beim Einlesen der gespeicherten Events verschickt wurden, behandelt. Es wird die Event-Nachricht überprüft und mit dem passenden Status das SimpleUDP Modul kontaktiert.
Wenn die Nachricht darauf hindeutet, dass die Gruppe aus dem Netzwerk austreten soll (LEAVE), wird der Status auf false gesetzt. Anderenfalls (JOIN) wird der Status auf true geändert. (s. Listing 4.11).

```
1 void SimpleUnderlayConfigurator::readNextEvent(){
2   double zeti;
3   std::map<int,Event>::iterator p=eve.begin();
4   while (p!= eve.end()){
5    if (p->first==position){
6     zei=p->second.zeit;
7       scheduleNextEvent(p->second.gruppeNum,p->second.event,zei);
8     position++;
9     break;
10   }
11   p++;
12  }
13
14  if (position< eve.size() ) {
15  cancelEvent(nextEvent);
16  scheduleAt(simTime()+1, nextEvent);
17  } else {
18  }
19 }
20
21 void SimpleUnderlayConfigurator::scheduleNextEvent(int groupNum, char event[],
      double zeit){
22   cMessage *msg=new cMessage(event);
23    scheduleAt(zeit, msg);
24 }
```

Listing 4.10: Die integrierten Methoden readNextEvent() und scheduleNextEvent()

```
1
2void SimpleUnderlayConfigurator::handleEventMessage(cMessage* msg){
3 if (!msg->isSelfMessage()) {
4  delete msg;
5  return;
6  } else if ( msg == nextEvent){
7    readNextEvent();
8    return;
9  }
10
11 else if( (strstr(msg->getName(),"LEAVE") == msg->getName())||(strstr(msg->getName()
       ,"JOIN") == msg->getName())) {
12  std::vector<NetzwerkTrennung>::iterator it =
13  NetzwerkT.begin();
14  std::map<int,Event>::iterator p=eve.begin();
15  int zeile;
16  while (p!= eve.end()){
17   if (p->first==GRUPPE){
18    zeile=p->second.gruppeNum;
19    break;
20   }
21   p++;
22  }
23  while (it!= NetzwerkT.end()){
24   if(it->gruppe==zeile){
25    if (strstr(msg->getName(),"LEAVE") == msg->getName())  {
26     SimpleUDP* simpleUdp1=check_and_cast<SimpleUDP*>((it->node)->getSubmodule("udp"
           ));
27     simpleUdp1->setNodeEntry((it->entry),false,it->gruppe,true);
28    }else if (strstr(msg->getName(),"JOIN")== msg->getName()){
29     SimpleUDP* simpleUdp2=check_and_cast<SimpleUDP*>((it->node)->getSubmodule("udp"
           ));
30     simpleUdp2->setNodeEntry((it->entry),true,it->gruppe,true);
31    }
32   }
33   it++;
34  }
35  GRUPPE++
36 }
37}
```

Listing 4.11: handleEventMessage()

4.3.3 Erweiterung des SimpleUDP Moduls

Die Hauptmethode im SimpleUDP ist:
handleMessage (cMessage *msg):
Diese Methode behandelt alle ankommenden Nachrichten. Von dort werden auch andere Methoden, die im SimpleUDP implementiert sind, aufgerufen, um die Nachrichten weiterzuleiten.
Die Klasse implementiert auch die Methode **setEntry()**. Diese wird in der SimpleUnderlayConfigurator.cc Klasse aufgerufen, um die eingelesenen Eigenschaften von jeder Knotengruppe im SimpleUDP einzusetzen.
In der Methode **handleMessage (cMessage *msg)** werden dann die Gruppen mit den IP-Adressen, dem jeweiligen Status und der Gruppennummer in einer Liste gespeichert (s. Listing 4.12 Zeilen 15-18).
Zunächst werden die IP-Adresse mit der Omnet++ Methode **IPAddressResolver(). addressOf(node)** von jedem Knoten bestimmt. Vorher wird der Knoten (cModule *node) mit der Methode (auch in Omnet++ implementiert) **getParentModule()** bestimmt (s. Listing 4.12 Zeilen 1-3)
Die IP-Adresse wird zuerst in dem Vektor gesucht. Wenn die vorhanden ist, wird sie gelöscht und durch den aktuellen Eintrag ersetzt (s. Listing 4.12 Zeilen 9-13). Somit wird die Liste nach jeder Änderung aktualisiert.
Die Nachrichten werden dann gefiltert, damit die Verbindung zwischen verschiedenen Gruppen vom Status abhängig hergestellt wird.
Zunächst werden alle angekommenen Nachrichten ausgepackt. Dabei werden die Zieladressen (destAddr) und die Quelladressen (srcAddr) eingelesen(s. Listing 4.12 Zeilen 20-24).
Mit der Liste, die alle Informationen über die Knoten enthält, werden die Adressen zugeordnet und der Status von jeder Adresse bestimmt. Somit wird für die Nachrichten entschieden, ob die weitergeleitet oder entworfen werden.
Dabei werden die Nachrichten so aussortiert:

```
1  if((statusA==true && statusB==true) ||( statusA==false && statusB==false) ){
2      if( statusA==false && statusB==false){
3          if(gruppeB!=gruppeA){
4              ..... Entwerfen
5           delete msg;
6           return;
7           }
8         }
9      ....    Weiterleiten
10 }else{
11       ..... Entwerfen
12     delete msg;
13     return;
14  }
```

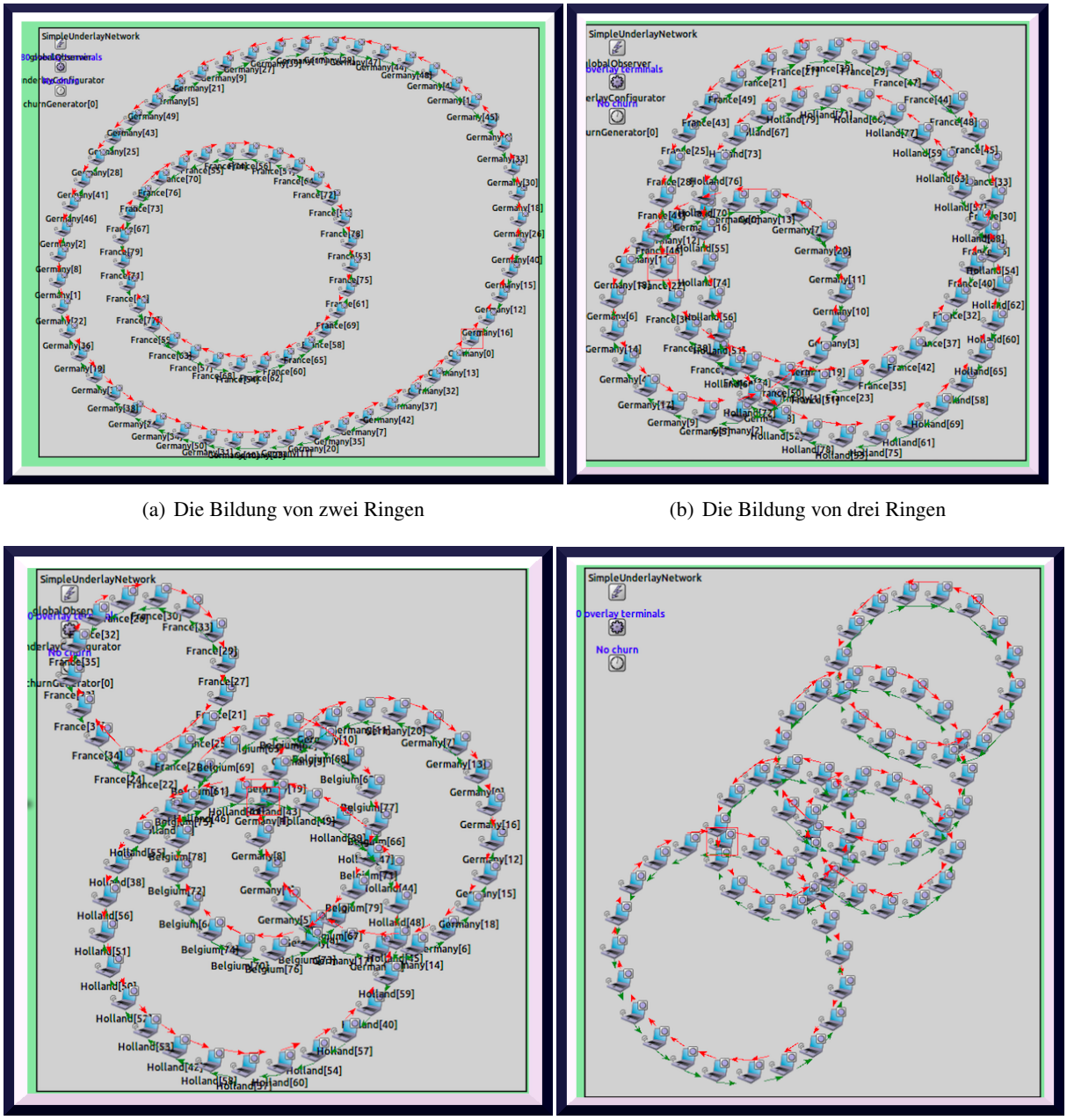
StatusA bzw. StatusB ist der Status der gelesenen Quell bzw. Zieladresse.

```
1
2  cModule *node = getParentModule();
3  IPvXAddress nodeIp;
4  nodeIp=IPAddressResolver(). addressOf(node);
5  Gruppen gruppe;
6  std::vector<Gruppen>::iterator it = vector. begin();
7
8  while (it != vector. end()) {
9     if ((it-> ip)!=nodeIp) {
10      it++;
11    }else{
12      vector. erase(it);
13    }
14 }
15
16 gruppe.ip=nodeIp;
17 gruppe. gruppe =gruppenNum;
18 gruppe. status=gruppenStatus;
19 vector. push_back(gruppe );
20
21 UDPControlInfo *udpCtrl;
22 PvXAddress srcAddr, destAddr;
23 udpCtrl = check_and_cast <UDPControlInfo*>(PK(msg)->getControlInfo());
24 srcAddr = udpCtrl->getSrcAddr();
25 destAddr = udpCtrl->getDestAddr();
26 std::vector<Gruppen> ::iterator k =v. begin();
27
28 while (k != v. end()){
29   if(k->ip==srcAddr0 ){
30    gruppeA=k->gruppe;
31    statusA=k->status;
32    break;
33   }
34   k++;
35  }
36
37  while (k != w. end()){
38    if (k->ip==destAddr0 ){
39     gruppeB=k->gruppe;
40     statusB=k->status
41     break;
42    }
43    k++;
44   }
```

Listing 4.12: Erweiterungen in der Klasse SimpleUDP.cc

4.4 Zusammenfassung

In diesem Kapitel wurde ein Verfahren vorgestellt, das die Partitionierung im Underlay und die Bildung von isolierten Gruppen ermöglicht.
Mit den Eingabedateien werden also Knotengruppen im Underlay gebildet (s. Bilder. 2 Ringe, 3 Ringe, 4 Ringe und 7 Ringe), um die Konnektivität zwischen Knoten aus unterschiedlichen Knotengruppen zeitweise einzuschränken und dabei die Effekte auf die Overlay-Topologie zu untersuchen.
Es hat sich gezeigt, dass die Architektur von OverSim, die von Omnet++ übernommen wurde, viele Vorteile für das Verständnis des Aufbaus der Komponenten innerhalb der Module bietet.
Die Kommunikation zwischen verschiedenen Overlay-Knoten ist zwar nicht berücksichtigt, aber die Verbindung innerhalb eines Knotens (zwischen den unterschiedlichen Schichten) ist mittels der NED-Dateien gut beschrieben. Dadurch können alle ausgetauschten Nachrichten gesteuert werden. Es können beispielsweise bestimmte Adressen (Zieladressen bzw. Quelladressen) gefiltert werden oder bestimmte Schichten isoliert werden, was zu Sicherheitszwecken eingesetzt werden könnte.
Diese Bilder zeigen die Möglichkeit, mehrere Anzahl (3, 4, 5, 7 Ringen) an getrennten Ringen (bzw. Regionen) zu bilden. Dafür wurden verschiedene Parameter in den Eingabedateien eingegeben (Partition und Event-Dateien). Die folgende Tabelle (s.Tabelle 4.1) veranschaulicht die eingesetzten Parameter, um diese Bilder zu bekommen.



(a) Die Bildung von zwei Ringen

(b) Die Bildung von drei Ringen

(c) Die Bildung von vier Ringen

(d) Die Bildung von fünf Ringen

Abbildung 4.8: Die Bildung von isolierten Gruppen im Underlay

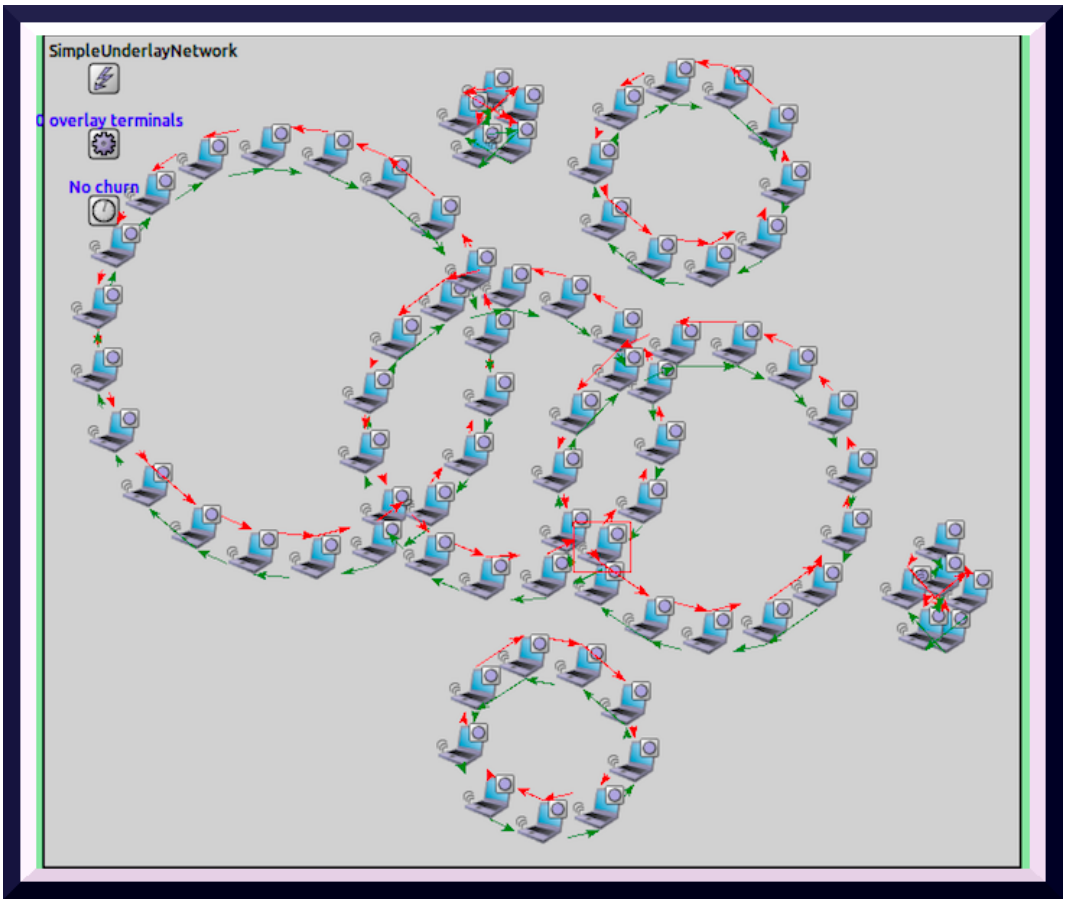


Abbildung 4.9: Die Bildung von sieben Ringen

	Partition-Datei		Event-Datei		
	Gruppenname	Anzahl an knoten	Gruppen Nr	Event	Zeitpunkt (s)
2 Ringe	Germany	30	1	LEAVE	150
	France	70	2	LEAVE	200
3 Ringe	Germany	20	1	LEAVE	150
	France	40	2	LAEVE	200
	Belgium	40	3	LEAVE	250
4 Ringe	Germany	20	1	LEAVE	150
	France	15	2	LAEVE	200
	Belgium	20	3	LEAVE	250
	Holland	25	4	LEAVE	300
5 Ringe	Germany	17	1	LEAVE	150
	France	13	2	LAEVE	200
	Belgium	10	3	LEAVE	250
	Spain	10	4	LEAVE	300
	Holland	15	5	LEAVE	350
7 Ringe	Germany	17	1	LEAVE	150
	France	20	2	LAEVE	200
	Belgium	15	3	LEAVE	250
	Spain	9	4	LEAVE	300
	Holland	6	5	LEAVE	350
	Poland	5	6	LEAVE	400
	Luxembourg	15	7	LEAVE	450

Tabelle 4.1: Die eingegebenen Parameter in den Szenarioeingabedateien

Kapitel 5

Implementierung der Merging-Algorithmen im Overlay

Im vorigen Kapitel wurden die Realisierung der Netzwerkpartitionierung im Underlay betrachtet und dabei die Effekte auf die Overlay-Topologie analysiert. Diese Partitionierung, die durch den Verlust der Netzwerkanbindung hervorgerufen wurde, benötigt Verfahren, um die getrennten Overlay Partitionen nach Wiederherstellung der Konnektivität zu verschmelzen. In diesem Kapitel werden dann die Mechanismen und Konzepte für die Implementierung dieser Verfahren bzw. Algorithmen, welche in Kapitel 2 diskutiert worden sind, näher erläutert.
Für die Umsetzung der Merging-Algorithmen in OverSim Overlay wurde Chord eingesetzt, da dieses Overlay-Protokoll ein einfach strukturiertes, effizientes Routingprotokoll für Peer-to-Peer-Systeme bietet. Zusätzlich wurde das GlobalNodeList Modul erweitert und ein neues Modul integriert.
Konkret wurden die folgenden Klassen und Dateien für die Implementierung verwendet bzw. erweitert.

- **Chord.cc**:
  In dieser Klasse wurden die Merging-Algorithmen implementiert.

- **ChordMessage.msg**:
  Diese Datei wurde für die Definition der von den Algorithmen ausgetauschten Nachrichten verwendet.

- **Chord.ned**:
  Diese NED-Datei wurde eingesetzt, um die Algorithmen konfigurierbar zu gestalten.

- **GlobalNodeList**:
  Das globale Modul wurde erweitert, um Listen für die Simulation und die Evaluierung bereitzustellen.

Ein neues Modul wurde im Chord integriert:

- **gruppenList**:
  Dieses Modul wurde neu implementiert. Es hat die Aufgabe, Kontaktlisten für die Algorithmen zur Verfügung stellen. Das atomare Modul bietet zwei Listen: aktive und passive Listen.
  Dabei wurden Methoden implementiert, um die Listen zu aktualisieren, die Große zu bestimmen und Einträge zu entfernen.

Im Abschnitt 5.1 werden die in diesem Kapitel eingesetzten Module bzw. Klassen näher erläutert. Wie die Merging-Algorithmen im Chord implementiert wurden, wird im Abschnitt 5.2 ausführlich erklärt. Dabei werden die wichtigen Mechanismen und Techniken, die dazu geführt haben, die Algorithmen umzusetzen, zu testen und zu bewerten, präsentiert.

5.1 Beschreibung

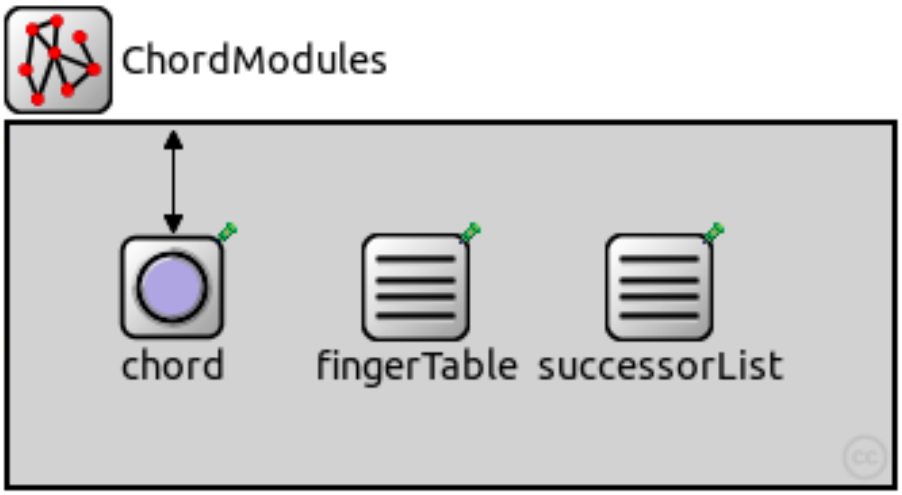


Abbildung 5.1: Chord Moduls

Im folgenden Abschnitt werden alle Module bzw. Klassen, die für die Implementierung der Merging-Algorithmen verwendet wurden, beschrieben. Zunächst wird der Aufbau des Chord Moduls näher erläutert. Anschließend wird das GlobalNodeList Modul mit den Methoden, die erweitert bzw. neu integriert wurden, kurz vorgestellt.

5.1.1 Chord-Implementierung

Chord-Overlay ist in OverSim als Compound Modul dargestellt. Dieses Modul besteht aus drei einfachen Modulen (s. Abbildung 5.2): Chord, FingerTabelle und SuccessorList.

- `Chord`:
  Stellt das Hauptmodul in Chord-Protokoll-Implementierung dar. Die Chord Klasse ist von der BaseOverlay Klasse abgeleitet(s. Abbildung 5.3). Diese Klasse bietet viele Methoden, um die Aufgaben der schlüsselbasierten Routing-Protokolle (Key-based Routing, KBR) zu erledigen. Der KBR-Dienst ist dabei der Routing-Dienst, der die Weiterleitung einer Nachricht N mit dem Zielschlüssel x an den Overlay-Knoten, der für den Schlüssel x verantwortlich ist, ermöglicht. Die Klasse Chord muss nun nur einige wenige Methoden überschreiben, um beispielsweise Routingentscheidungen oder Routing-Metrik zu bestimmen.

- `FingerTable` :
  Diese Klasse enthält die Finger-Tabelle und implementiert Methoden, um die Finger in Chord aufzunehmen und die Tabelle aufrechtzuerhalten.

- `SuccessorList`:
  Das Modul enthält die Nachfolgerliste. Die Tabelle wird periodisch aktualisiert mit der Methode failedNode(), die im Chord aufgerufen wird, wenn ein Nachfolger ausfällt.

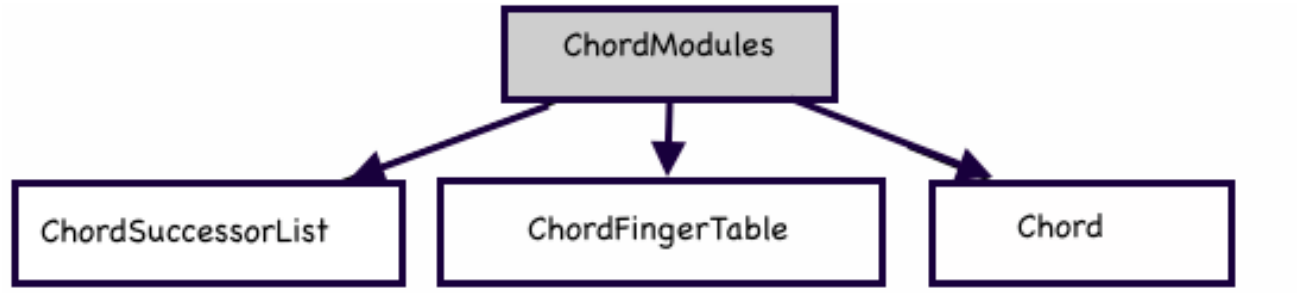


Abbildung 5.2: Chord Moduls

5.1.2 BaseOverlay

Als Basisklasse für alle Overlay-Protokolle im OverSim übernimmt BasisOverlay u.a folgenden Aufgaben:

- Erzeugung von Instanzen verschiedener Lookup-Klassen

- Versenden von Bestätigungsnachrichten.

- Bootstrapping

- Visualisierung der Overlay-Topologie.

Diese Klasse bietet viele Methoden, die von anderen Overlay-Protokollen ubernommen oder überschrieben wurden.

- `handleUDPMessage()`: Mit dieser Methode werden die Nachrichten, die an dem UDP Modul angekommen sind, behandelt. Der definierte Befehl (Command) in der MSG-Datei wird hier abgelesen und daraufhin wird die implementierte Methode, um die Nachrichten zu behandeln, aufgerufen. Diese Methode wird für alle ausgetauschten Nachrichten von den Merging-Algorithmen eingesetzt (s. Listing 5.1).

- `findNode()`: Diese Funktion wird von jeder Overlay-Implementierung zur Verfügung gestellt. Dabei wird der Knoten, der näher am Zielschlüssel liegt, zurückgeliefert.

- `isSiblingFor()`: Diese Methode überprüft, ob der gesuchte Schlüssel in den Zuständigkeitsbereich des Knotens fällt.

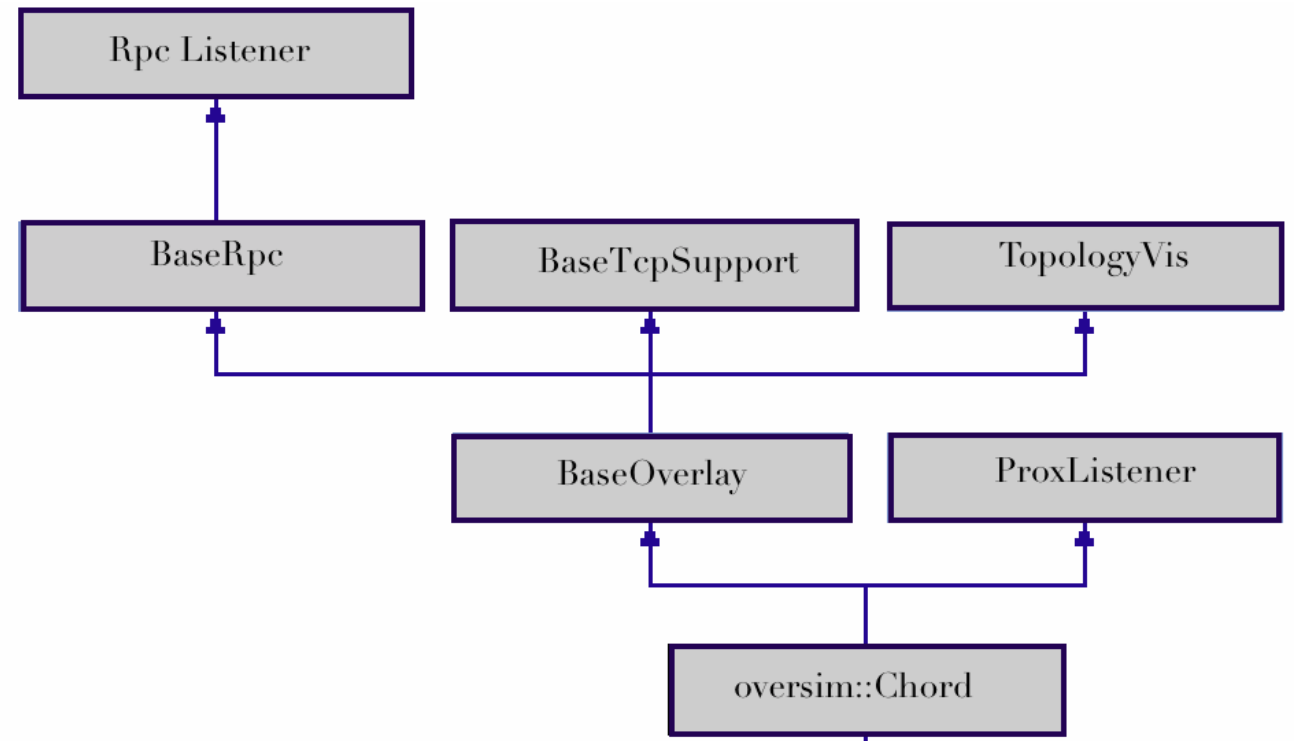


Abbildung 5.3: Ableitungsbaum des Chord

```
1  void Chord::handleUDPMessage(BaseOverlayMessage* msg)
2{
3   ChordMessage* chordMsg = check_and_cast<ChordMessage*>(msg);
4   switch(chordMsg->getCommand()) {
5   case NEWSUCCESSORHINT:
6     handleNewSuccessorHint(chordMsg);
7     break;
8   case ZipPing:
9     handleZipPing(chordMsg);
10    break;
11  case New:
12   handleNewMsg(chordMsg);
13   break;
14  case Mlookup:
15    handleMlookupMessage(chordMsg);
16    break;
17  case TryMerger:
18     handleTryMerger(chordMsg);
19     break;
20  case MergeRing:
21    handlemerge1(chordMsg);
22    break;
23   case Distribute:
24     handleDistribute(chordMsg);
25     break;
26   case ZipPong:
27     handleZipPong(chordMsg);
28     break;
29   case Ping:
30     handlePing(chordMsg);
31     break;
32   case Pong:
33     handlePong(chordMsg);
34     break;
35   case StartUni:
36     handleStartUni(chordMsg);
37     break;
38  default:
39    error("handleUDPMessage(): Unknown message type!");
40    break;
41  }
42  delete chordMsg;
43}
```

Listing 5.1: handleUDPMessage()

5.1.3 GlobalNodeList

Dieses Modul ist im Ordner Common als einfaches Modul (GlobalObserver.ned) implementiert. Das Modul speichert alle im Netzwerk aktuell vorhandenen Knoten und deren Merkmale, u.a. die Transportadresse des Knotens, die dem Knoten gegebene Node-ID und den für den Knoten zugehörige Churn-Generator. Die Transportadresse oder Transport Address bedeutet hierbei die Underlay-Adresse eines Overlay-Knotens (IP-Adresse und den Port des Transportprotokolls). Die Klasse GlobalNodeList.cc bietet viele Funktionen, die für alle anderen Klassen greifbar sind. Die Methoden, die in dieser Arbeit verwendet wurden, sind:

- `GlobalNodeList::getNumNodes():`
  Diese Methode liefert die Anzahl der vorhandenen Knoten zurück. Beispielsweise im Chord wurde sie integriert, um einen bestimmten Knoten zu fordern, eine SelfMessage zu verschicken.

```
1    If(globalNodeList->getNumNodes()==3){
2        scheduleAt(simTime(), merge\_Timer);
3    }
```

  Somit wird der dritte Knoten, der dem Netzwerk beigetreten ist, in der Zeit simTime()+300 eine Aktion mit der SelfMessage merg_Timer ausführen. Dabei liefert die Omnet++ Funktion simTime() den aktuellen Zeitpunkt zurück.

- `GlobalNodeList::getRandomNode():`
  Diese stellt zufällige Knoten aus dem Netzwerk zur Verfügung. Die Methode wurde in dieser Arbeit verwendet, um die aktive Liste, die im Rahmen dieser Arbeit implementiert wurde, zu erstellen.

- `GlobalNodeList::addPeer():`
  Diese Methode wird in der SimpleUnderlyConfigurator.cc Klasse aufgerufen. Somit werden die Knoten, die im Underlay erzeugt worden sind, direkt in der GlobalNodeList registriert. Die Methode wurde in dieser Arbeit verwendet, um die manuelle Liste zu erstellen.

- `TransportAddress* GlobalNodeList::getRandomAliveNode():`
  Diese Funktion wurde oft verwendet, um einen zufällig vorhandenen Knoten zu kontaktieren.

- `NodeHandle* GlobalNodeList::getNodeHandle():`
  Diese nimmt eine Transportadresse als Parameter und gibt eine NodeHandle zurück. Ein NodeHandle ist die vollständige Adresse eines Overlay-Knotens in einem strukturierten Overlay-Netz. Sie enthält die Transportadresse und die Node-ID des Knotens.

Die neu integrierten Methoden sind:

- `const NodeHandle&GlobalNodeList::getNodeManuall(int index)`:
  Die Funktion nimmt einen Index als Parameter. Der Index verweist dabei auf die Gruppennummer, die von der Partition-Datei eingelesen wurde (s. Kapitel 4). Wenn ein Merging-Algorithmus manuell gestartet wird, wird am Anfang der Simulation ein Initiatorknoten festgelegt. Dieser Knoten verschickt dann eine SelfMessage mit einer bestimmten Nachricht. Wenn diese Nachricht empfangen wird, wird ein Knoten von einer anderen Gruppe mit dieser neuen implementierten Methode ausgewählt. Die Liste enthält die ersten Knoten von jeder Gruppe.

- `void GlobalNodeList::addList(NodeHandle&node,const NodeHandle&succ)`:
  Diese Methode wird in Chord aufgerufen, um eine Liste der Pointer von jedem Knoten am Anfang der Simulation zu erstellen. Diese Liste wird als Templat verwendet, um die Pointer der Knoten während der Simulation zu vergleichen

- `int GlobalNodeList::vergleiche(NodeHandle&node,NodeHandle&succep)`:
  Die Methode wurde implementiert, um die Pointer der Knoten während der Simulation zu vergleichen und somit die Funktionalität der Merging-Algorithmen, die gestartet wurden, zu bewerten.

- `void GlobalNodeList :addPointer (int anzahlP, simtime_t zeit)`:
  Sie stellt eine Liste zur Verfügung, die korrekte Pointer in einem bestimmten Zeitpunkt enthält. Diese Methode wird in der GlobalNodeList::vergleiche () aufgerufen. Darauf wird in dem Abschnitt 5.2 näher eingegangen.

5.2 Merging-Algorithmen

Das Chord-Protokoll wurde in OverSim einfach implementiert. Es reicht, dabei zu verstehen, wie Chord im Allgemeinen aufgebaut ist. Als Vorbereitung für diesen Teil der Arbeit wurde das Beispiel in der OverSim Dokumentation [17] (unter OverSimDevelop – How to add a new overlay protocol) bearbeitet und erprobt. Dabei wurde anhand eines Beispiels die Implementierung des Moduls eines Overlay-Protokolls schrittweise erklärt. Das vorherige Bearbeiten des tic toc Tutorial (in Omnet++ Dokumentation) hat grundsätzlich gut geholfen, um den Aufbau der Module im OverSim nachzuvollziehen.
Das Herausfinden, wie das Suchen nach einem Schlüssel (Lookup) im Chord OverSim funktioniert, war das einzige Prob lem, da die Implementierung hierfür von der Lookup-Methode, die im Chord-Paper vorgestellt wurde, sich abgrenzt. Das Verstehen des im Chord Oversim ausgeführten JOIN

Prozesses hat allerdings das Problem rasch gelöst.
Die im Kapitel 2 vorgestellten Merging-Algorithmen wurden also im Rahmen dieser Arbeit in Chord OverSim implementiert und getestet. Nach der Realisierung der Netzwerkpartitionierung im Underlay, die den Effekt der Bildung der Overlay-Partitionen hat, kommen die Merging-Algorithmen zum Einsatz, um diese getrennten Partitionen zu verschmelzen.
Ziel ist es, dass die Mechanismen und Prozesse, die verwendet wurden, um die Algorithmen zu implementieren, eingeführt bzw. erklärt werden.
In diesem Abschnitt wird dann erklärt, wie die Merging-Algorithmen in OverSim umgesetzt werden können. Zunächst wird das neu integrierte Modul gruppenliste vorgestellt. Die darin implementierten Listen werden näher erläutert. Dabei werden sowohl der Aufbau als auch die Funktionalität der Listen diskutiert.

Um einen Merging-Algorithmus zu startet, muss ein Initiatorknoten einen Kontaktknoten in einem anderen Ring kennenlernen. Für diesen Zweck wurden in dieser Arbeit mehrere Listen implementiert:

- `Manuelle Liste`:
  Sie ist eingesetzt, um einen Kontaktknoten in einem anderen Ring auszuwählen. Dabei wird der Initiatorknoten auch manuell bestimmt und gefordert, den ausgesuchten Kontaktknoten zu verschmelzen. Somit werden die beiden Ringe zusammengeführt. Diese Liste wird in das GlobalNodeList implementiert.

- `Passive Liste`:
  enthält die ausgefallenen Knoten. Diese Knoten werden während der Simulation von jedem Knoten in einer Liste aufgenommen. Diese Liste wird von jedem Knoten periodisch mit PING-Nachrichten kontaktiert. Wenn der passiven Knoten wieder verfügbar ist, antwortet der anfragende Knoten mit einer PONG-Nachricht. Daraufhin wird der Knoten als Kontaktknoten aufgenommen und damit der Merging-Prozess gestartet.

- `Aktive Liste`:
  Jeder Knoten enthält diese Liste. Sie besteht aus einer bestimmten Anzahl an Knoten, die im Netzwerk verfügbar waren, als der Knoten dem Netzwerk beigetreten ist. Die aktiven und passiven Listen wurden in dem neuen integrierten Modul Gruppenliste implementiert.

Alle durchgeführten Erweiterungen bzw. neu implementierten Methoden in allen betroffenen Klassen werden im Folgenden besprochen. Dabei werden die drei folgenden Typen der Nachrichten, die von den Algorithmen verwendet wurden, anhand von Beispielen im Detail betrachtet.

- **`Lookup Nachrichten`**:

Diese Nachrichten enthalten eine Suchanfrage nach einem Suchschlüssel aus dem Schlüsselraum des Overlay-Protokolls. Dabei wird der Overlay-Knoten zurückgeliefert, in dessen Verantwortungsbereich der Schlüssel liegt. Chord-Zip und Ring Reunion Algorithmen und seine verbesserte Version werden gestartet, indem eine Lookup-Anfrage nach dem eigenen Schlüssel verschickt wird.

- **`Direkt ausgetauschten Nachrichten zwischen den Overlay-Knoten`**:
Diese Nachrichten werden direkt zu einer bekannten Adresse über UDP-Schnittstelle geschickt.

- **`SelfMessage`**:
sind Nachrichten, die ein Knoten zu einem Zeitpunkt in der Zukunft an sich selbst sendet, welche zur Modellierung von Zeitgebern verwendet werden. Diese Nachrichten sind im Chord hauptsächlich verwendet, um die periodischen Operationen für die Routing-Tabelle durchzuführen. Als Beispiel betrachten wir stabilize_timer Message, die periodisch von jedem Knoten verschickt wird, um die Routing-Tabelle zu stabilisieren. Die Methoden bzw. Klassen, die verwendet werden, um SelfMessages zu verschicken, wurden von Omnet++ übernommen.

  Zuerst wird eine Nachricht in der Methode Chord::initializeOverlay() erzeugt.

```
1        stabilize\_timer = new cMessage("stabilize_timer");
```

Um diese Nachricht dann periodisch zu verschicken, wird die scheduleAt() Methode verwendet. Dabei liefert die simTime() Methode den aktuellen Zeitpunkt und stabilizeDelay den Zeitraum, der von der Configurationsdatei (default. ini) abgelesen wird. Vorher wird die CancelEvent() eingesetzt, um die SelfMessages zu synchronisieren.

```
1            cancelEvent(stabilize_timer);
2            scheduleAt(simTime() + stabilizeDelay, stabilize_timer);
```

handelStabilizeTimerExpired(msg) wird dann aufgerufen, um den Stabilisierungsprozess, wofür der stabilize_timer gestartet wurde, durchzuführen.

```
1   if (msg == stabilize\_timer) {
2    handleStabilizeTimerExpired(msg);
3   }
```

5.2.1 Integriertes GruppenList Modul

Dieses Modul wurde als ein einfaches Modul (SubModule) im Chord-Overlay implementiert. Somit werden in jedem Overlay-Knoten SuccesorList und die Fingertabelle zur Verfügung stehen.
Ziel ist das Bereitstellen von Kontaktlisten (active und passive Listen), damit jeder Knoten über die Möglichkeit verfügt, die Merging-Algorithmen automatisch durchführen. Somit wird eine mögliche Partitionierung auf der Ebene des Overlay vermieden bzw. repariert. Das Modul implementiert also sowohl die passive Liste als auch die aktive Liste, die im Folgenden näher erläutert werden.

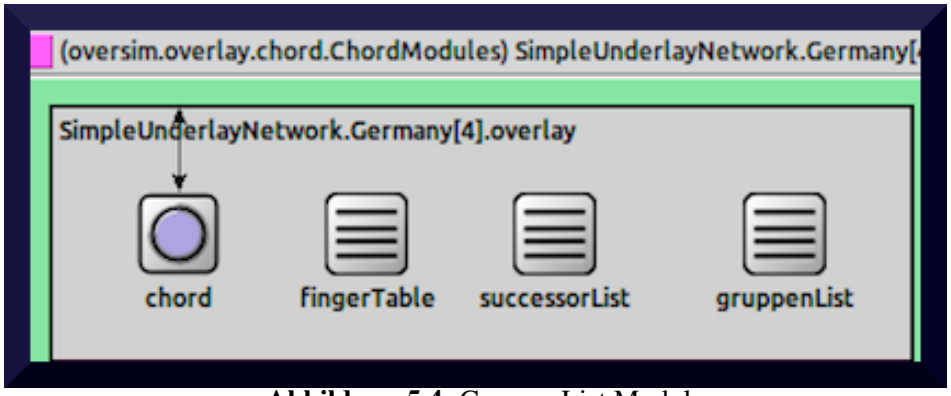


Abbildung 5.4: GruppenList Modul

Aktive Liste

Wenn der Knoten dem Overlay-Netz beitreten möchte, wird eine Liste von dem GloblaNodeList bereitgestellt. Diese Liste wird mit der Methode GlobalNodeList::getRandomNode() erstellt. Die Länge der Liste wird im Voraus festgelegt. Die Methode getRandomNode() sucht zufällige Knoten, die im Overlay-Netz verfügbar sind. Dann wird sie von dem neuen beigetretenen Knoten im Gruppen-List Modul gespeichert. Mit einer SelfMessage wird die Liste in periodischen Abständen mit PING-Nachichten kontaktiert. Jeder antwortende Knoten wird als Kontaktknoten aufgenommen. Somit wird der Merging-Algorithmus ausgeführt. Falls der Kontaktknoten sich im gleichen Ring befindet, wird der Algorithmus meist sofort abgebrochen (Chord-Zip und Ring Reunion) oder spätestens, wenn der Initiatorknoten sich selbst als alternativen Successor bekommen hat (Ring Unification).

Passive Liste

Die Liste empfängt die ausgefallenen Knoten, welche der Overlay-Knoten in der Routing-Tabelle (SuccessorTabelle oder/und Finger-Tabelle) enthalten hat. Chord implementiert die Methode handle-

FailedNode(), die von BasisOverlay zur Verfügung gestellt wird.
`handleFailedNode()`: Die Methode detektiert die Knoten, die mit der Methode findNode() nicht erreichbar sind. Dann sorgt handleFailedNode() für das Entfernen eines ausgefallenen Overlay-Knotens aus den Routing-Tabellen des jeweiligen Overlay-Protokolls. Dabei werden die Methoden `fingerTable->handleFailedNode(failed)` bzw. `successorList->handleFailedNode(failed)` aufgerufen. Daraufhin werden die ausgefallenen Knoten aus den Routing-Tabellen (Finger-Tabelle und SuccessorList) entfernt. In dieser Methode werden dann von jedem Overlay Knoten die ausgefallenen Knoten empfangen und in dem GruppenList Modul gespeichert (s. Listing 5.2). Jeder Knoten startet eine SelfMessage, damit die Liste periodisch kontaktiert wird. Dabei werden die in der Liste gespeicherten Knoten mit PING-Nachrichten gespeist. Wenn der passive Knoten wieder erreichbar ist, antwortet er mit einer PONG-Nachricht. Daraufhin wird der Knoten aus der passiven Liste gelöscht und das Zusammenführen dann gestartet.

```
1bool Chord::handleFailedNode(const TransportAddress& failed)
2 { if (!predecessorNode.isUnspecified() && failed == predecessorNode)
3     predecessorNode = NodeHandle::UNSPECIFIED_NODE;
4     TransportAddress oldSuccessor = successorList->getSuccessor();
5 if (successorList->handleFailedNode(failed))
6   gruppenList->addPassiveList(failed);
7   updateTooltip();
8 if (fingerTable != NULL)
9   fingerTable->handleFailedNode(failed);
10  gruppenList->addPassiveList(failed);
11 if ((!predecessorNode.isUnspecified()) &&
12   oldSuccessor == predecessorNode) {
13   predecessorNode = NodeHandle::UNSPECIFIED_NODE;
14   callUpdate(predecessorNode, false);
15 }
16 if (failed == oldSuccessor) {
17   if (memorizeFailedSuccessor) {
18     failedSuccessor = oldSuccessor;
19   }
20   cancelEvent(stabilize_timer);
21   scheduleAt(simTime(), stabilize_timer);
22 }
23 if (state != READY) return true;
24 if (successorList->isEmpty()) {
25   cancelEvent(stabilize_timer);
26   cancelEvent(fixfingers_timer);
27 }
28 return !(successorList->isEmpty());}
```

Listing 5.2: handleFailedNode()

5.2.2 Erweiterung des Chord Moduls

Alle Merging-Algorithmen, die im Kapitel 2 vorgestellt wurden, wurden in das Chord-Modul im Rahmen dieser Arbeit integriert und implementiert. Dafür wurden hauptsächlich Chord.cc, Chord.h, Chord.ned und chordMessage. msg erweitert.
Alle diese Algorithmen basieren auf Nachrichtenaustausch mit dem Kontaktknoten, der die Aufgabe übernimmt, den Iniatiator im eigenen Ring neu zu positionieren. Dabei werden die Merging-Nachrichten an den anderen Knoten weitergeleitet. Dadurch werden verschiedene Ringe zusammengefügt, und ein globaler Ring wird dann gebildet.
Die Chord-Zip und Ring Reunion Algorithmen werden gestartet, indem der Kontaktknoten ein Lookup nach der Initiator-ID im eigenen Ring durchführt. Die Realisierung des Lookup-Prozesses anhand des Chord-Zip bzw. Ring Reunion Algorithmus wird dann im Folgenden näher betrachtet. Wie die normalen Nachrichten zwischen zwei Knoten vertauscht werden, wird anhand des Unification-Algorithmus beschrieben. Am Schluss werden die SelfMessages, die für die Algorithmen und die implementierten Listen eingeführt wurden, vorgestellt.

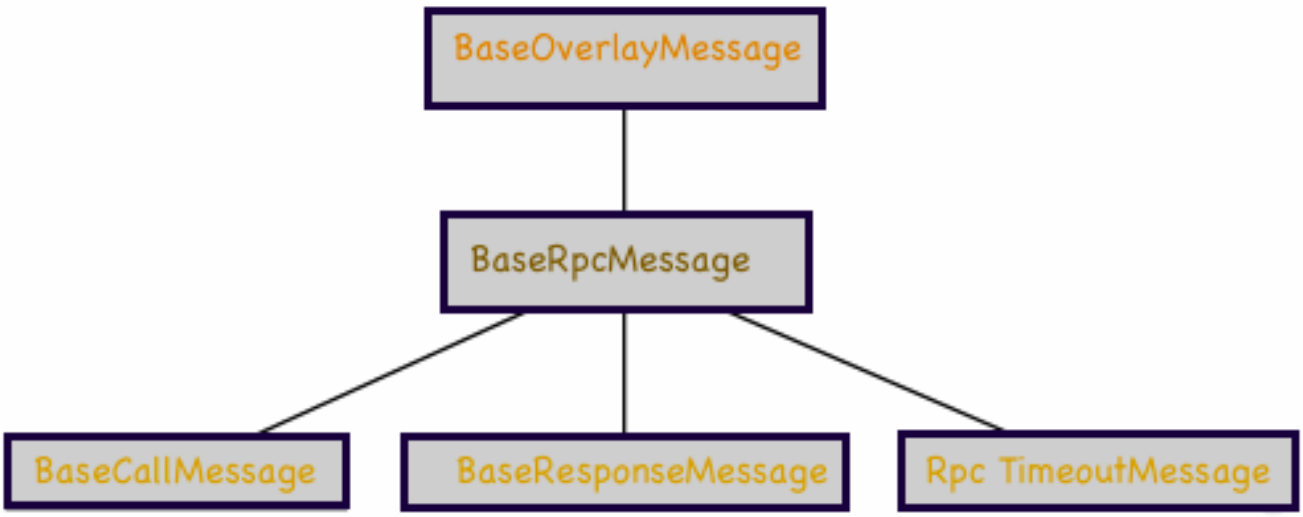


Abbildung 5.5: RPC-Ableitungsbaum.

Lookup() in OverSim

Im Allgemeinen erfolgt das Suchen nach einem Schlüssel im Chord, wie es in der Literatur erläutert wird, mit der Methode, die in dem Listing 5.3beschrieben wird. Im OverSim wird das Auffinden von Schlüsseln durch die von der RPC-Schnittstelle (s. Kapitel 3 Abschnitt3.1.4) übernommen. Diese Schnittstelle (s. Kapitel 2) stellt ein gemeinsames, unabhängiges Protokoll aller strukturierten

Overlay-Protokolle zur Verfügung. Mit dieser Schnittstelle wird dann die Lookup() Prozedur durchgeführt.

```
1 n. find_successor(id)
2  if (id \in (n, successor] )
3   return successor;
4  else
5   n0 = closest_preceding_node(id);
6   return n0.find_successor(id);
7 n. closest_preceding_node(id)
8  for i = m downto 1
9   if (finger[i]\in(n, id))
10   return finger[i];
11  return n;
```

Listing 5.3: Lookup()-Pseudocode in Chord

```
1  void Chord::zipMerge(NodeHandle node){
2  ZipCall* call = new ZipCall("ZipCall");
3  call->setBitLength(ZIPCALL_L(call));
4  RoutingType routingType = (defaultRoutingType == FULL_RECURSIVE_ROUTING ||
5                 defaultRoutingType == RECURSIVE_SOURCE_ROUTING) ?
6                SEMI_RECURSIVE_ROUTING : defaultRoutingType;
7  sendRouteRpcCall(OVERLAY_COMP, node, thisNode.getKey(),
8           call, NULL, routingType, joinDelay);
9
10}
11 void Chord::rpcZip(ZipCall* zipCall)
12{
13  NodeHandle requestor = zipCall->getSrcNode();
14  ZipResponse* zipResponse =new ZipResponse("ZipResponse");
15  zipResponse->setSucNode(successorList->getSuccessor());
16  sendRpcResponse(zipCall, zipResponse);
17  updateTooltip();
18}
19void Chord::handleRpcZipResponse(ZipResponse* zipResponse)
20{
21  NodeHandle reque = zipResponse->getSrcNode();
22  successorList->addSuccessor(reque);
23  isIni=true;
24  if(thisNode!=reque){
25   merge(reque, thisNode, false);
26  }
27updateTooltip();
28}
```

Listing 5.4: Zip-Chord Lookup()

Im Chord-Zip und in dem Ring Reunion Algorithmus generiert der Initiator eine RPC-Nachricht (Call) (s. Listing 5.5 Zeile 1) und wird mit der Methode sendRouteRPCCall () (s. Listing 5.5 Zeile 7) an den Kontaktknoten gesendet. Dabei wird der zuständige Schlüssel für den Initiator gefunden. Die Basis Klasse BaseOverlay generiert dann eine FindNodeCall Nachricht, um den Schlüssel zu finden. Dieser antwortet dem Initiator mit einer RPC-Response (Zeile 20).
Mit der Methode handleRPCZipResponse() (s. Listing 5.5 Zeilen 19-25) erfährt der Initiator seinen alternativen Nachfolger, den Sender der RPC-Nachricht (Zeile 21). Dann wird die Merging-Nachricht verschickt und somit das Zusammenführen gestartet (Zeile 33)
Für die Verarbeitung der RPC-Nachrichten bietet die Schnittstelle im Allgemeinen die folgenden Methoden, die von allen Overlay-Protokollen verwendet werden müssen, um RPC-Nachrichten weiterleiten zu können :
`handleRpcCall()`:
wird bei Empfang einer Call-Nachricht aufgerufen. Dabei werden C++-Makros verwendet. Dadurch wird die Weiterleitung an die dazugehörigen Callback-Methoden vereinfacht.

```
1 Chord::handleRpcResponse(BaseResponseMessage* msg, cPolymorphic* context, int
      rpcId, simtime_t rtt)
2{
3
4  RPC_SWITCH_START(msg)
5   RPC_ON_RESPONSE(Zip ) {
6    handleRpcZipResponse(_ZipResponse);
7    EV << "[Chord::handleRpcResponse() @ " << thisNode.getIp()
8    << " (" << thisNode.getKey().toString(16) << ")]\n"
9    << "  Received a Fixfingers RPC Response: id=" << rpcId << "\n"
10   << "  msg=" << *_ZipResponse << " rtt=" << rtt
11   << endl;
12   break;
13  }
14   RPC_ON_RESPONSE( Ringlookup ) {
15    handleRpcRinglookupResponse(_RinglookupResponse);
16    EV << "[Chord::handleRpcResponse() @ " << thisNode.getIp()
17    << " (" << thisNode.getKey().toString(16) << ")]\n"
18    << "  Received a Fixfingers RPC Response: id=" << rpcId << "\n"
19    << "  msg=" << *_RinglookupResponse << " rtt=" << rtt
20    << endl;
21    break;
22  }
23  RPC_SWITCH_END( )
24}
```

Listing 5.5: Chord-Zip Lookup Verfahren

`handleRpcReponse()`:
wird beim Empfang einer Antwort auf eine zuvor versandte Nachricht aufgerufen. Auch hier können die erwähnten C++- Makros eingesetzt werden.

`handleRpcTimeout()`:
wird beim Verlust einer Nachricht oder der dazugehörigen Antwort bei allen Sendewiederholungen aufgerufen. Ein Nachrichtenverlust wird beim Ablauf eines Zeitgebers angenommen, falls die betreffende Nachricht bis dahin nicht beantwortet wurde.

```
1  bool Chord::handleRpcCall(BaseCallMessage* msg)
2  {
3   if (state != READY) {
4    EV <<"[Chord::handleRpcCall() @" << thisNode.getIp()
5      << "("<< thisNode.getKey().toString(16) <<")]\n"
6      <<"Received RPC call and state != READY"
7      <<endl;
8    return false;
9  }
10  RPC_DELEGATE(Zip, rpcZip);
11  RPC_DELEGATE(Ringlookup, rpcRinglookup);
12  RPC_SWITCH_END( )
13  return RPC_HANDLED;
14 }
15 void Chord::handleRpcTimeout(BaseCallMessage* msg, const TransportAddress&dest,
       cPolymorphic* context, int rpcId, const      OverlayKey&)
16 {
17  RPC_SWITCH_START(msg)
18  RPC_ON_CALL(Ziplookup ) {
19   EV << "[Chord::handleRpcTimeout() @ " << thisNode.getIp()
20   << " (" << thisNode.getKey().toString(16) << ")]\n"
21   << "Zip RPC Call timed out: id=" << rpcId << "\n"
22   << " msg=" << *_ZiplookupCall
23   << endl;
24   if (!handleFailedNode(dest)) join();
25   break;
26 }
27  RPC_ON_CALL( Ringlookup) {
28   EV << "[Chord::handleRpcTimeout() @ " << thisNode.getIp()
29   << " (" << thisNode.getKey().toString(16) << ")]\n"
30   << " RingLookup RPC Call timed out: id=" << rpcId << "\n"
31   << " msg=" << *_RinglookupCall
32   << endl;
33   break;
34  }
35  RPC_SWITCH_END( )
36 }
```

Listing 5.6: Die erweiterten RPC Methoden

Eine RPC-Nachricht stellt grundsätzlich entweder eine Anfrage (BaseCallMessage)(s. Listing 5.7) dar oder liefert eine zugehörige Antwort (BaseResponseMessage). Die beiden Nachrichtenformate werden in der Datei ChordMessage.msg definiert (s. Listing 5.7 für Ziplookup()).

```
1 ZipCall extends BaseCallMessage
2 {
3 NodeHandle initiator;
4 }
5 packet ZipResponse extends BaseResponseMessage
6 {
7 NodeHandle sucNode;
8 NodeHandle ini;
9 }
```

Listing 5.7: RPC-Nachrichtenformat in ChordMessage.msg

Direkter Austausch der Nachrichten zwischen den Overlay-Knoten

Hier wird der Overlay-Knoten direkt kontaktiert und nicht, wie in der Lookup Anfrage, die sich um den Schlüssel kümmert und den zuständigen Knoten findet. Alle implementierten Merging-Algorithmen basieren auf Nachrichtenaustausch mit anderen Knoten. Dabei werden Informationen ausgetauscht, um die Routing-Tabellen zu aktualisieren und den Algorithmus zu terminieren, wenn der Merging-Prozess das Ziel erreicht hat.
Im Folgenden wird die Realisierung dieses Austauschs anhand des Ring Unfication Algorithmus betrachtet. Die Funktionalität von jedem Algorithmus wurde schon im Kapitel 2 erläutert.

Der Ring Unification Algorithmus wird gestartet, indem er zwei Mlookup-Nachrichten verschickt.
Die erste wird an den Kontaktnoten in einem anderen Ring gesendet. Die zweite Mlookup-Nachricht wird lokal weitergeleitet. Diese Nachricht enthält die ID des Initiator. Somit wird das Merging-Verfahren in beiden Ringen initiiert.

```
1 packet MlookupMessage extends ChordMessage handleMlookupMessage (ChordMessage*
      chordMsg)
2 {
3  NodeHandle srcNode;
4  NodeHandle id;
5 }
```

Listing 5.8: Nachrichtenformat der Mlookup-Nachricht

Um eine Mlookup-Nachricht zu erzeugen (s. Listing 5.8 Zeile 7), wird zuerst ein Nachrichtenformat in der Datei ChordMessage.msg definiert (s. Listing 5.8).

Die Informationen, die mit den Nachrichten gesendet werden, werden dann in der ChordMessage.msg Datei gespeichert (s. Listing 5.9 Zeilen 8-10). Mit der Methode **sendMessageToUDP(ChordMessage * chordMsg)** wird die Nachricht an den Zielknoten verschickt.
Die Basis-Klasse BaseOverlay stellt die**handleUDPMessage(ChordMessage * chordMsg)** Methode zur Verfügung, die in Chord überschrieben wurde, um neue Nachrichten zu empfangen, die vorher als Command in der Datei ChordMessage.msg definiert wurden. Wenn die Nachricht empfangen wurde, wird die Methode, die diese Nachricht behandelt, dabei aufgerufen.

Für die Mlookup-Nachricht wird dann **handleMlookupMessage (ChordMessage * chordMsg)** (s. Listing 5.9) aufgerufen. Dabei werden die Nachrichten ausgepackt und die ID abgelesen. Die Methode **getKey()** liefert den Schlüssel zurück und **isBetween()** überprüft die Lokalisierung des Knotens.

Somit wird jedes Mal überprüft, ob die Terminierungsbedingung (Zeile 25) erfüllt ist. Daraufhin wird die ID zwischen dem Nachfolger und dem Vorgänger positioniert und dementsprechend eine TryMerger-Nachricht zum Nachfolger bzw. Vorgänger weitergeleitet (Zeilen 25-30).

Wenn die ID nicht in dem Bereich zwischen dem Knoten und dem Nachfolger oder dem Vorgänger liegt, wird die Methode close() aufgerufen (s. Listing 5.9 Zeilen 35-42). Dabei wird in der RoutingTabelle (FingerTabelle) nach dem nächsten Schlüssel zur ID gesucht. Zu diesem Schlüssel wird dann die Mlookup-Nachricht weitergegeben (Zeile 30).

Mit der sendTryMerger()-Methode wird eine TryMerger-Nachricht erstellt und zur ID gesendet (s. Listing 5.10 Zeile 2). Die neuen Pointer werden auch mitgeschickt, damit der Zielknoten sich neu positionieren. In der handleTryMerger() Methode (s. Listing 5.10 Zeile 11-22) werden dann die Pointer aktualisiert und die Mergin-Nachricht weitergeleitet.

```
1 void Chord::UnifikationStarte(NodeHandle contact, int k){
2   sendMlookup(thisNode, contact, k);
3   sendMlookup(contact, thisNode, k);
4   updateTooltip();
5  }
6 void Chord::sendMlookup(NodeHandle desNode, NodeHandle nodeId){
7   MlookupMessage *mlookupMsg= new MlookupMessage("Mlookup");
8   mlookupMsg->setCommand(Mlookup);
9   mlookupMsg->setSrcNode(thisNode);
10  mlookupMsg->setId(nodeId);
11  if( desNode==thisNode){
12    handleMlookupMessage(mlookupMsg);
13   }
14   else{
15     sendMessageToUDP(desNode, mlookupMsg);
16   }
17   updateTooltip();
18 }
19 void Chord::handleMlookupMessage(ChordMessage* chordMsg){
20   MlookupMessage* lookupMsg =check_and_cast<MlookupMessage*>(chordMsg);
21   NodeHandle id=lookupMsg->getId();
22   NodeHandle succ=successorList->getSuccessor();
23    if (id!=thisNode && succ!=id){
24      if(id.getKey().isBetweenLR(thisNode.getKey(), succ.getKey())){
25        sendTryMerger(id, thisNode, succ);
26     }else if(id.getKey().isBetweenLR(predecessorNode.getKey(), thisNode.getKey())){
27        sendTryMerger(id, predecessorNode, thisNode);
28     }else{
29        NodeHandle close2=close(id.getKey());
30      sendMlookup(close2, id);
31     }
32   }
33   updateTooltip();
34 }
35 NodeHandle Chord::close(const OverlayKey& key){
36   Nodehandle close;
37    for (int i = fingerTable->getSize() -1; i >= 0; i--) {
38      if (fingerTable->getFinger(i).getKey().isBetween(thisNode.getKey(), key)) {
39       close1=fingerTable->getFinger(i);
40       return close;
41    }
42  }
43 }
```

Listing 5.9: Ring Unification (Teil 1)

```
1
2void Chord ::sendTryMerger(NodeHandle des, NodeHandle cpred, NodeHandle csucc){
3   TryMergerMessage *trymergerMsg= new TryMergerMessage("TryMerger");
4    trymergerMsg->setCommand(TryMerger);
5    trymergerMsg->setSrcNode(thisNode);
6    trymergerMsg->setCsucc(csucc);
7    trymergerMsg->setCpred(cpred);
8
9    sendMessageToUDP(des, trymergerMsg);
10   updateTooltip();
11}
12
13void Chord::handleTryMerger(ChordMessage *chordmsg)      {
14   TryMergerMessage* trymergerMsg =check_and_cast<TryMergerMessage*>(chordmsg);
15    NodeHandle succ= successorList->getSuccessor();
16    sendMlookup(thisNode, trymergerMsg->getCsucc());
17
18    if((trymergerMsg->getCsucc()).getKey().isBetweenLR(thisNode.getKey(), succ.getKey
          ())){
19    successorList->addSuccessor(trymergerMsg->getCsucc());
20}
21
22    sendMlookup(thisNode, trymergerMsg->getCpred());
23    if((trymergerMsg->getCpred()).getKey().isBetweenLR(predecessorNode.getKey(),
          thisNode.getKey())){
24   predecessorNode=trymergerMsg->getCpred();
25   }
26}
```

Listing 5.10: Ring Unification (Teil 2)

SelfMessage

In dieser Arbeit wurden diese Nachrichten verwendet, um die Kontaktlisten periodisch mit PING-Nachrichten zu kontaktieren oder die Pointer-Liste (s. nächsten Abschnitt) zu aktualisieren bzw. zu erstellen.
Die SelfMessages werden von jedem Knoten in der Initialisierungsphase verschickt (s. Listing 5.11). Dabei werden die Parameter von der Configurationsdatei eingelesen und dann mit der Methode scheduleAt() die SelfMessages gesendet. In der eingegebenen Zeit wird dann die Aktionen anhand der Methode handlemergeTimerExpired(cMessage*msg) ausgeführt. (s. Listing 5.13).

```
1
2 if((activeList) ){
3  scheduleAt(simTime()+nodeCount, merge_start);
4 }
5
6 if(passiveList ){
7  scheduleAt(simTime()+10, activeList_timer);
8  scheduleAt(simTime()+nodeCount, merge_start);
9 }
10
11 if (registrierPointer ){
12  scheduleAt(simTime()+pointerdelay, registrier_Pointer);
13 }
14
15 if (pointerVergleiche ){
16  scheduleAt(simTime(), pointer_timer);
17 }
```

Listing 5.11: Die eingeführten SelfMessages

```
1
2 void Chord::handlemergeTimerExpired(cMessage*msg){
3
4    if (msg==pointer_timer){
5
6    if ((state != READY) || successorList->isEmpty()){
7     scheduleAt(simTime()+pointerdelay, pointer_timer);
8     return;
9   }
10
11   for(int i=0;i<successorList->getSize();i++){
12     if(i==0){
13      NodeHandle successorp=successorList->getSuccessor(i);
14      globalNodeList->vergleiche(thisNode, successorp, i);
15      break;
16      }
17    }
18
19    cancelEvent(pointer_timer);
20    scheduleAt(simTime()+pointerdelay, pointer_timer);
21
22    globalNodeList->addList(thisNode, succe);
23
24   }
```

Listing 5.12: Die eingeführte Methode um SelfMessages zu behandeln(teil1)

```
1  if(msg==registrier_Pointer){
2     if(simTime()>nodeCount+pointerdelay+10){
3       return;
4      }
5     for(int i=0;i<successorList->getSize();i++){
6       if(i==0){
7       globalNodeList->addList(thisNode, successorList->getSuccessor(i));
8       break;
9      }
10    }
11     scheduleAt(simTime()+pointerdelay, registrier_Pointer);
12  }
13
14 if(msg==activeList_timer){
15  int l=0;
16  if(gruppenList->getSizeActive()==0){
17   for(size_t i=0;i<jount-1 ;i++){
18     if(l<10){
19     NodeHandle node=globalNodeList->getRandomNode();
20       if(thisNode!=node){
21        gruppenList->addAktiveList(node);
22       }
23       l++;
24     }
25     }
26    }
27  }
28  if(msg==merge_start){
29  pingList(h);
30  scheduleAt(simTime()+10, merge_start);
31 }
```

Listing 5.13: Die eingeführte Methode um SelfMessages zu behandeln (teil2)

5.2.3 Erweiterung des GlobalNodeList Moduls

Diese Klasse wurde hauptsächlich verwendet, um die Pointer-Liste und die Pointer-Datei zu erstellen und alle dafür benötigten Methoden zu implementieren. Die manuelle Liste wurde auch in dieser Klasse implementiert.
Was die Pointer-Liste ist und wofür sie implementiert ist, wird im Folgenden beantwortet. Auf die manuelle Liste wird auch näher eingegangen.

Nachfolger-Pointer

Um die Funktionalität der Merging-Algorithmen zu testen, wurde eine Liste in GlobalNodList implementiert. Diese Liste enthält alle Knoten mit den jeweiligen Nachfolgern. Mit dem periodischen Vergleich der Pointer in jedem Knoten während der Simulation wird festgestellt, ob eine Änderung im Overlay-Netz stattgefunden hat. In dieser Arbeit wird die Liste für die Evaluation verwendet. Dabei werden die Anzahl (bzw. Anteile) der Korrekten Pointer während der Simulation periodisch berechnet.
Somit wird getestet, ob ein Merging-Algorithmus das Ziel erreicht hat und die Knoten wieder die gleichen Pointer bekommen haben wie am Anfang der Simulation.
Für diesen Zweck wurden zwei Zeitgeber (SelfMessage) eingeschaltet. Zunächst wird die Liste ausgefüllt (s. Listing 5.14 Zeilen 1-7), indem die Nachfolger von jedem Knoten in der Pointer-Liste (im GlobalNodeList) gespeichert werden. Der zweite Zeitgeber wird periodisch für jeden Knoten ausgeführt. Dabei werden jedes Mal die Pointer abgelesen und mit der Liste verglichen (s. Listing 5.14 Zeilen 10-35).
Nach jedem Vergleich wird die Anzahl der korrekten Knoten mit der aktuellen Zeit in einer Liste gespeichert (s. Listing 5.14 Zeilen 35 &Listing5.15 Zeilen 1-6). Nach Ende der Simulation wird die Methode GlobalNodelist->print() in **Chord::finishOverlay()** aufgerufen. Die Methode print() wurde in GlobalNodeList implementiert (s. Listing 5.15 Zeilen 8-18). Diese liest die Pointer-Liste ein und schreibt den Inhalt in die Datei successorPointer.txt. Diese Datei wird dann verwendet, um die Ergebnisse der Simulation zu bewerten.

```
1  void GlobalNodeList:: addList(NodeHandle& node, const NodeHandle& succ ){
2   std::map<NodeHandle, NodeHandle>::iterator it =
3   gruppe. find(node);
4   if(it!= gruppe. end()) {
5    gruppe. erase(it);
6   }
7    gruppe. insert(std::make_pair(node, succ));
8  }
9  int GlobalNodeList::vergleiche(NodeHandle\& node, NodeHandle\& succep, int pos){
10 std::map<NodeHandle, NodeHandle> ::iterator it =
11 gruppe. begin();
12 if(gruppe. size()>0){
13  gruppe. find(node);
14  if(it==gruppe. end()) {
15    pointer=0;
16    addPointer(0, simTime());
17  }else{
18    if(it!=gruppe. end()){
19      pointer++;
20    }
```

```
21  }
22  std::map<NodeHandle, NodeHandle > ::iterator iter =
23  gruppe.begin();
24  while (iter != gruppe.end()) {
25    if (iter->first== node){
26     if (iter->second==succep) {
27      pointer0++;
28      break;
29     }
30    }
31    iter++;
32  }
33 }
34 if(pointer>=gruppe.size()) {
35   addPointer(pointer0, simTime());
36   pointer0=0;
37   pointer=0;
38  }
39 }
```

Listing 5.14: Die eingeführten Methoden in das GlobalNodeList (Teile 1)

```
1  void GlobalNodeList::addPointer(int anzahlP, simtime_t zeit)
2  {
3   SuccessorPointer p;
4   p.pointerAnzahl=anzahlP;
5   p.zeit=zeit;
6   succPointer.push_back(p);
7  }
8
9  void GlobalNodeList::print()
10 {
11  fstream datei("successorPointer.txt", ios::out);
12  std::vector<SuccessorPointer>::iterator l =succPointer.begin();
13  SuccessorPointer p;
14  while (l!= succPointer.end()){
15    double anz=(l->pointerAnzahl*100)/peerStorage.size();
16    datei<< (l->zeit)/60 << "\t" << anz <<endl;
17    l++;
18  }
19  datei.close();
20 }
```

Listing 5.15: Die eingeführten Methoden in das GlobalNodeList(Teile 2)

Manuelle Liste

Um die Simulation per Hand zu starten, wird ein bestimmter Initiatorknoten am Anfang der Simulation ausgewählt. Dieser Knoten braucht einen Kontaktknoten in einem anderen Ring, um die Knoten von verschiedenen Gruppen zu verschmelzen. Die manuelle Liste bietet dann die Kontaktknoten für die Iniatorknoten, um der Merging-Algorithmen zu starten. Die Liste wurde in der GlobaNodlist Klasse implementiert. Dabei werden die ersten Knoten von jeder Gruppe direkt von der Klasse SimpleUnderlayConfigurator.cc aufgenommen. Die Knoten werden dort mit der Gruppennummer gespeichert. Somit wird in jeder Gruppe per Hand der Merging-Algorithmus gestartet.
Mit der Methode addPeer(), die in der Klasse SimpleUnderlayConfigurator.cc aufgerufen wird, wird die Funktion einem neuen Parameter übergeben. Dieser Parameter bezeichnet dabei den ersten Knoten, der erzeugt wurde. Somit werden die Knoten in der **GlobalNodelist::addPeer()** empfangen und in der Liste gespeichert.

5.3 Zusammenfassung

In diesem Kapitel wurden die Mechanismen, die eingesetzt wurden, um die Merging-Algorithmen im Chord-Overlay in OverSim zu implementieren, vorgestellt. Dabei wurden die verschiedenen implementierten Kontaktknoten, mit denen die Algorithmen gestartet werden, präsentiert. Die Pointer-Listen und Pointer-Datei, die die Lauffähigkeit der Algorithmen überprüfen, wurden auch im Detail diskutiert. Die Algorithmen lassen sich sehr einfach in Chord OverSim implementieren, da Chord viele Funktionen bietet, die das Umsetzen der Algorithmen vereinfachen. Natürlich hat die gute strukturierte Implementierung des Chords in OverSim dabei geholfen. Die periodischen Operationen, die Chord durchführt, um die Routing-Tabelle aufrechtzuerhalten, können für das Verständnis der verwendeten Mechanismen bzw. Prozesse eine wichtige Rolle spielen. Alle diese Operationen verwenden die RPC-Schnittstelle, um Nachrichten weiterzuleiten. Dadurch kann dieses Verfahren gut verstanden werden, besonders, wenn die RPC-Methoden, die dabei aufgerufen wurden, in den eigenen Klassen nachgeschlagen werden.
Bezüglich der Algorithmen wurde eine kleine Erweiterung durchgeführt. Das Testen mit der aktiven Liste hat gezeigt, dass Chord-Zip sich erst terminiert, wenn der Initiatorknoten die PING-Nachricht bekommt hat, (s. Kapitel 2) d. h. das Verfahren wird durchgeführt, auch wenn der Initiatorknoten eigene ID als lookup-Antwort zurückbekommen hat. Da die aktive Liste Knoten, die sich im gleichen Ring befinden, enthalten könnte, werden diese periodisch mit PING-Nachrichten kontaktiert und zusammengeführt. Dadurch wird die Topologie innerhalb des gleichen Rings meistens beeinflusst und das Bilden des globalen Rings verzögert. Als Lösung dafür wurden Chord-Zip und Ring Reunion Algorithmen um einen Zeilen-Code erweitert. Dabei wird überprüft, ob die Antwort der Lookup-Anfrage den eigenen Schlüssel enthält. In diesem Fall wird das Verfahren abgebrochen, und somit werden nur

Knoten, die nicht zum gleichen Ring gehören, verschmolzen.

Kapitel 6

Evaluation

In diesem Kapitel erfolgt eine Evaluation des entworfenen Netzwerkpartitionierungskonzepts im Underlay und der implementierten Merging-Algorithmen im Overlay. Zuerst wird das entworfene Verfahren zur Realisierung der Netzwerkpartitionierung im SimpleUnderlay überprüft.
Dabei werden Szenarien durchgeführt, um das Konzept zu verifizieren und zu evaluieren. Danach werden die im Kapitel 2 vorgestellten Merging-Algorithmen (Chord-Zip, Simple Unification Ring, Gossip-based Ring Unification, Ring Reunion und seine parallelisierte Version) mit der Kombination der verschiedenen implementierten Listen simuliert. Am Schluss werden alle erzielten Ergebnisse präsentiert und diskutiert.
Alle durchgeführten Simulationen werden anhand der Pointer-Datei evaluiert, die nach dem Simulationsende (Erhebungsphase) erstellt wird. Im Folgenden wird diese Datei vorgestellt.

Pointer-Datei

Die Pointer Datei enthält periodisch den Anteil der Knoten, die während des Simulationsdurchlaufs korrekte Pointer haben. Dabei werden zeilenweise die gerechneten Knotenanteile mit dem entsprechenden Zeitpunkt gespeichert. Während der Simulation werden periodisch (der Parameter Pointerdelay=20s wird in default.ini gesetzt) die Pointer von jedem Knoten mit denen, die in einer Liste während der Aufbauphase gespeichert wurden, verglichen. Mit diesem Vergleich wird die Anzahl der Knoten, die korrekte Pointer haben, in jedem Zeitabstand in eine Liste (Pointer-Liste) aufgenommen. Somit wird der Knotenanteil an korrekten Zeigern gerechnet und in der Pointer-Datei mit dem gespeicherten Zeitpunkt geschrieben.
Diese Datei wird dann nach Ende der Simulation erstellt. Dabei wird der Knotenanteil so gerechnet:
Knotenanteil = Knotenanzahl *100/Gesamtanzahl der vorhandenen Knoten.

```
1          double anzahl =(l->pointerAnzahl*100)/peerStorage.size();
```

Parameter	Wert
Anzahl der Knoten im Overlay	1000
Type der Knotenfluktuatuion	Keine (no churn)
Initialisierungsphase	1000 s (jede Sekunde tritt ein Knoten dem Netzwerke bei)
Underlay Modell	Simple Underlay
Länge der Nachfolger	100
Pointerdelay (Zeitabstand für Pointervergleich)	20

Tabelle 6.1: Eingesetzte und integrierte Parametert in default.ini

Im Abschnitt 6.1 werden alle durgeführten Simulationsszenarien vorgestellt. Dabei werden zunächst die gesetzten bzw. die neu integrierten Parameter näher erläutert. Dann werden die Simulationen, die für die Netzwerkpartitionierung und die Merging-Algorithmen durchgeführt wurden, beschrieben.
Im Abschnitt 6.2 werden die Evaluierungsergebnisse präsentiert. Alle diese Ergebnisse werden dabei anhand der erstellten Schaubilder illustriert und erklärt.

6.1 Szenarien und Parameter

Die Evaluation wurde mit dem in Kapitel 3 präsentierten Overlay-Framwork OverSim und in den Kapiteln 4 und 5 integrierten Netzwerkpartitionierung und implementierten Merging-Algorithmen durchgeführt.
Zur Konfiguration der durchgeführten Simulationen wurden neue Parameter in default.ini (s.Abschnitt 3.2) integriert. Somit wurden für jede Simulation verschiedene Merging-Algorithmen und verschiedene Listen festgelegt. Die Tabelle 6.1 gibt eine Übersicht über die in allen durchgeführten Simulationen verwendeten Simulationsparameter bzgl. der simulierten Netzwerkpartitionierung im SimpleUnderlay und der Merging Algorithmen im Overlay. Die Parameter der zu evaluierenden Szenarien werden in den dazugehörigen Abschnitten vorgestellt.
Die simulierten Netzwerke bestehen aus 1000 Overlay-Knoten, welche in der Initialisierungsphase nacheinander dem Netzwerk beitreten. Also benötigt die Initialisierungsphase 1000 s für 1000 Knoten. Die Knotenfluktuation wurde mit dem im Abschnitt 3.1.3 beschriebenen Churn-Generator (No Churn) generiert. Am Anfang wurde mit LifetimeChurn getestet. Ein Problem trat dabei beim Vergleichen der Pointer auf. Mit diesem Churn werden neuen Knoten dem Netzwerk beitreten. Dadurch werden die Pointer von manchen Knoten während der Simulation beeinflusst. Somit können die Merging-Algorithmen nicht gut verifiziert werden, da diese anhand der Pointer der Knoten getestet werden.

Während der Simulation der Netzwerkpartitionierung wurde festgestellt, dass die Länge der Nachfolgerliste bei der Bildung der getrennten Overlay-Gruppen eine entscheidende Rolle spielt. Die Länge der Nachfolgerliste war in OverSim auf 8 gesetzt. Damit wurde auch die Bildung der gewünschten Gruppen im Simple-Underlay bzw. die Netzwerkpartitionierung getestet. Allerdings konnten keine richtigen Gruppen bzw. getrennten Ringe entstehen, obwohl die Netzverbindung zwischen den Gruppen komplett ausgeschaltet wurde.
In diesem Buch [6] wurde gezeigt, dass die Länge der Nachfolgerliste die Overlay-Topologie im Chord beeinflusst. Dabei wurde die Länge der Nachfolgerliste im Chord getestet und evaluiert. Es wurde ausführlich erklärt, wie die Länge der Liste die Ring-Topologie und die Leistungsfähigkeit von Chord beeinflussen kann. Dabei wurde festgestellt, dass der Ring auseinanderbricht, falls die Länge der Nachfolgerliste 2 ist. Aus diesem Grund wurde dieser Parameter geändert bzw. erhöht. Dadurch wurden die Ergebnisse erzielt, die zu erwarten sind.
Mit dem Parameter Pointerdelay wird ein Zeitraum definiert, der die periodischen Vergleiche der Pointer der Knoten während der Simulation bestimmt.
Alle SimpleUnderlaykonfigurationsparameter, die sich auf den Overlay-Knotennamen (overlayTerminal) beziehen, die in OverSim integriert waren, wurden angepasst, so dass diese Parameter dynamisch für jeden neuen Namen eingesetzt werden können.

6.1.1 Netzwerkpartitionierung

Zunächst wird die Netzwerkpartitionierung im SimpleUnderlay anhand der Pointer-Datei, die nach Simulationsende erstellt wird, verifiziert. Diese Datei zeigt periodisch den Anteil der Knoten, die korrekte Pointer haben.
Zur Konfiguration der Netzwerkpartitionierung wurden die Szenarioeingabedateien (s. Kapitel 4) verwendet. Dabei bestimmt die Partitionsdatei den Namen der Gruppen und die Anzahl der Knoten pro Gruppe. Mit der Event-Datei werden die Events definiert. Also wird für jede Gruppe LEAVE als Eventnachricht eingegeben. Somit wird jede Gruppe gefordert, das Netzwerk in der eingegebenen Zeit zu verlassen. Ziel ist es, die Funktionalität des beschriebenen Konzepts zur Netzwerkpartitionierung in dem Kapitel 4 zu verifizieren und zu evaluieren. Dieses Szenario wird für verschiedene Anzahlen an Gruppen (2, 3, 4 und 5 Ringe) getestet und bewertet. Die Tabelle 6.2 gibt eine Übersicht über die eingegebenen Parameter in den Szenarioeingabedateien.
Die folgenden Szenarien werden dabei simuliert:

	Partition-Datei		Event-Datei		
	Gruppenname	Anzahl an Knoten	Gruppen Nr	Event	Zeitpunkt (s)
2 Ringe	Germany	500	1	LEAVE	2000
	France	500	2	LEAVE	2000
3 Ringe	Germany	300	1	LEAVE	2000
	France	400	2	LAEVE	2000
	Belgium	300	3	LEAVE	2000
4 Ringe	Germany	200	1	LEAVE	2000
	France	250	2	LAEVE	2000
	Belgium	300	3	LEAVE	2000
	Holland	250	4	LEAVE	2000
5 Ringe	Germany	250	1	LEAVE	2000 | 2000
	France	200	2	LAEVE	2000 | 2600
	Belgium	100	3	LEAVE	2000 | 3200
	Spain	200	4	LEAVE	2000 | 3800
	Holland	250	5	LEAVE	2000 | 4400

Tabelle 6.2: Die eingegebenen Parameter in den Szenarioeingabedateien (Event und Partition) zur Netzwerkpartitionierung

Einfaches Szenario

Hier werden alle Gruppen gleichzeitig das Netzwerk verlassen, d. h. der eingegebene Zeitpunkt in der Event-Datei ist gleich für alle Gruppen. Also in t= 2000 s werden alle Gruppen gespalten, was zur Bildung von verschiedenen Ringen führt und zwar gleichzeitig. Dieses Szenario wird für 2, 3, 4 und 5 Ringe wiederholt. Um die Anzahl der Ringe zu erhöhen, reicht es eine neue Zeile in der Partitionsdatei zu definieren und die Events für diese Gruppen in der Event-Datei festzulegen.

Kompliziertes Szenario

Dieses Szenario basiert nur auf den Eingaben in der Tabelle 6.2. Dabei wurden keine neuen Parameter eingeschaltet. Die Events werden allerdings für verschiedene Zeiten eingegeben, d. h. die Gruppen werden zu verschiedenen Zeiten aus dem Netzwerk austreten. Also werden die Gruppen hintereinan-

der das Netzwerk verlassen. In der Tabelle sind folgende Zeiten definiert:

Gruppe 1 in t=2000s, Gruppe 2 in t=2600s, Gruppe 3 in t=3200s, Gruppe 4 in t=3800s und Gruppe 5 in t=4400s. Dieses Szenario wird nur mit 5 Ringen simuliert. In der Tabelle wurden dabei zwei Zeiten pro Zeile eingegeben. Die zweite Spalte dabei verweist auf die Zeiten für das komplizierte Szenario.

6.1.2 Merging-Algorithmen

Für die Evaluierung der implementierten Merging-Algorithmen wurden alle benötigten Parameter, die nötig sind, um jeden einzelnen Algorithmus mit der Kombination von verschiedenen Kontaktlisten zu testen und zu evaluieren, in der Configurationsdatei default.ini integriert. Mit der Pointer-Datei, die am Ende der Simulation erstellt wird, wird dann verifiziert, ob die Merging-Algorithmen das Ziel erreicht haben und einen globalen Ring aus mehreren Ringen gebildet haben. Der erste Teil der Arbeit beschreibt den entworfenen und implementierten Mechanismus, der die Netzwerkpartitionierung herstellt und somit mehrere Overlay-Gruppen im Underlay bildet. Es werden die implementierten Merging-Algorithmen (Chord-Zip, Simple Ring Unification,Gossip based Ring Unification, Ring Reunion und seine parallelisierte Version) hintereinander mit verschiedenen Listen kombiniert und eingesetzt, um die Fähigkeit dieser Algorithmen mehrerer Ringe zu verschmelzen, zu testen und zu evaluieren.
Für diesen Zweck wurden folgenden Szenarien durchgeführt:

Manuelle Liste

In der Configurationsdatei default.in wird in diesem Szenario der neu integrierte Parameter manuell auf true gesetzt. Somit wird die manuelle Liste aktiviert, d. h. die Simulation der Merging-Algorithmen wird per Hand gestartet. Dabei wird dann der Initiatorknoten am Anfang der Simulation manuell festgelegt (s. Listing Zeilen 2-3).
Mit der manuellen liste werden alle Merging-Algorithmen simuliert. Also jedes Mal wird ein Algorithmus gesetzt. Dafür wurde in der Configurationsdatei default.ini für jeden Algorithmus ein neuer Parameter integriert (zip für Chord-Zip, Unifikation für Unification Reunion, Unifikation2 für Gossip-based Ring Unification, Reunion für Ring Reunion und Reunion2 für seine parallelisierte Version). Somit wird für jedes Szenario ein Algorithmus aktiviert und unabhängig von den anderen simuliert. Mit der manuellen Liste wurden dann insgesamt 20 Simulationen durchgeführt. Jeder festgelegte Algorithmus wurde mit 500 Knoten für zwei, drei, vier und fünf Ringe simuliert und evaluiert. Die Tabelle 6.3 veranschaulicht die Eingabedateien, die für diese Szenarien verwendet wurden. Dabei wurden diesmal zwei Events definiert. Das LEAVE Event-Nachricht fordert die Gruppen auf, in der gegebener Zeit das Netzwerk zu verlassen. Somit werden getrennte Gruppen entstehen. Mit der JOIN

Event-Nachricht werden die Gruppen wieder dem Netzwerk beitreten. Zum angegebenen Zeitpunkt wird die Verbindung zwischen den verschiedenen Gruppen hergestellt. Der Initiatorknoten wird also manuell für jedes simulierte Szenario bestimmt. Dann wird von jeder Gruppe, die zusammengeführt wird, ein Kontaktknoten von der implementieren Liste (Manuelle Liste) (s. Listing 6.1 Zeile 9) ausgewählt.
Um Kotaktknoten zu definieren, wurde der Code in das Listing 6.1 integriert. Dabei wird in der Initialisierungsmethode in Chord.cc ein Initiatorknoten festgelegt (s. Zeilen 2-3). Dieser verschickt eine SelfMessage in der festgelegten Zeit. In der Methode, die diese SelfMessage behandelt, wird dann ein Kontaktknoten von der anderen Gruppe in der manuellen Liste (s. Listing 6.1 Zeilen 13-21) eingelesen. Der Code wurde dann abhängig von der Anzahl der Ringe, die zusammengeführt werden sollen, erweitert. Diese Simulationen wurden für 2, 3, 4 und 5 Ringe durchgeführt.

```
1  if(manuell){
2     if (globalNodeList::getNumNodes()==1){
3     scheduleAt(simTime()+3000,merge_manuell);
4    }
5   }
6
7 if(msg==merge_manuell){
8    NodeHandle kontaktKnoten=globalNodeList->getNodeManuall(2);
9    if (GlobalNodeList::getNumNodes()==1){
10       if(Unification){
11        starte(kontaktKnoten);
12        }
13        else if(Reunion){
14        lookup(kontaktKnoten);
15        }
16        else if(zip){
17        zipMerge(kontaktKnoten);
18        }
19     }
20 }
```

Listing 6.1: Manuelle Bereitstellung von Kontaktknoten.

Automatische Zusammenführung (Merging)

Im letzten Abschnitt wurden die Algorithmen manuell simuliert. Dabei wurde ein Initiatorknoten per Hand festgelegt und gefordert, einen Kontaktknoten in einem anderen Ring auszusuchen. Somit hat nur ein Knoten pro Gruppe den Merging-Algorithmus gestartet. In den folgenden Szenarien wird das Verfahren mit Hilfe der Listen (passive Liste/aktive Liste), die von jedem Knoten implementiert

	Partition-Datei		Event-Datei		
	Gruppenname	Anzahl an knoten	Gruppen Nr	Event	Zeitpunkt (s)
2 Ringe	Germany	250	1	LEAVE	1000
	France	250	2	LEAVE	1000
			1	JOIN	1500
			2	JOIN	1500
3 Ringe	Germany	200	1	LEAVE	1000
	France	150	2	LAEVE	1000
	Belgium	150	3	LEAVE	1000
			1	JOIN	2000
			2	JOIN	2000
			3	JOIN	2000
4 Ringe	Germany	100	1	LEAVE	1000
	France	150	2	LAEVE	1000
	Belgium	150	3	LEAVE	1000
	Holland	100	4	LEAVE	1000
			1	JOIN	2000
			2	JOIN	2000
			3	JOIN	2000
			4	JOIN	2000
5 Ringe	Germany	250	1	LEAVE	1000
	France	100	2	LAEVE	1000
	Belgium	100	3	LEAVE	1000
	Spain	100	4	LEAVE	1000
	Holland	100	5	LEAVE	1000
			1	JOIN	2000
			2	JOIN	2000
			3	JOIN	2000
			4	JOIN	2000
			5	JOIN	2000

Tabelle 6.3: Die Eingabedateien für die manuellen Szenarien mit den Merging-Algorithmen

wurden, automatisch gestartet. Mit diesen Listen wird jeder Knoten in periodischen Abständen den gewählten Merging-Algorithmus automatisch durchführen. Also jeder Merging-Algorithmus (Chord-Zip, Unification Reunion, Gossip-based Ring Unification, Ring Reunion und seine parallelisierte Version) wird mit der Kombination der aktiven bzw. passiven Liste simuliert.
Im Folgenden werden dann die Szenarien, die sowohl mit der aktiven Liste als mit der passiven Liste durchgeführt wurden, vorgestellt.

Aktive Liste

Um die aktive Liste zu aktivieren, wird der Parameter activeList in der default.ini auf true gesetzt. Somit wird jeder Knoten automatisch den festgelegten Algorithmus (auch hier den entsprechenden Parameter in der Confgurationsdatei auf true setzen) starten, d. h. es werden periodisch die Knoten, die in der aktiven Liste gespeichert sind, zusammengeführt. Die aktive Liste wird erstellt anhand der gleichen Parameter, die gesetzt wurden, um die aktive Liste zu aktivieren. Dabei wird jeder Knoten eine bestimmte Anzahl an vorhandenen Knoten von der GlobalNodelIst bekommen. Die Länge der Liste wurde nicht vorgegeben. Deshalb wurden die Algorithmen mit verschiedenen Längen getestet. Für 1000 Knoten in einem gewählten Szenario wurden die Algorithmen mit den Längen 200, 100, 50 und 10 getestet. Dabei wurde festgestellt, dass die Länge der Liste die Laufzeit der Algorithmen beeinflusst hat. Mit zunehmender Länge der aktiven Liste steigt der Kommunikationsaufwand, da in jeder Merging-Runde auch Knoten, die sich im gleichen Ring befinden, mit PING-Nachrichten kontaktieren und dann zusammengeführt werden. Von daher wurde entschieden, für den Rest den Szenarien die Länge der aktiven Liste auf 10 zu setzen. Mit der aktiven Liste wird dann jeder Algorithmus simuliert und evaluiert. Diese Simulation wird dann für verschiedene Anzahlen an Gruppen wiederholt (2, 3, 4 und 5 Ringe). Die Tabelle 6.4 gibt einen Überblick über die verwendeten Szenarioeingabedateien. Dabei werden alle Gruppen in der Zeit t0= 2000 gleichzeitig das Netzwerk verlassen und in t1=4000 dem Netzwerk wieder beitreten.

Passive Liste

Die gleichen Simulationen, die mit der aktiven Liste durchgeführt wurden, werden mit der passiven Liste wiederholt. Hierfür wird auch der Parameter passivelist auf true gesetzt. Die Länge der passiven Liste muss nicht konfiguriert werden, da diese Länge von dem Knotenm in den Routing Tabellen (Successorliste und Fingertabelle) abhängt. Die passive Liste nimmt alle ausgefallenen Knoten, die schon in der Routingtabelle gespeichert waren, auf. Die Liste wird von jedem Knoten periodisch mit PING-Nachrichten kontaktiert, und alle darauf antwortenden Knoten werden dann zusammengeführt. Meistens wird der Merging-Algorithmus erst gestartet, wenn die Knoten, die in der Liste befinden, wieder Online sind. Diese Knoten werden dann von der Liste entfernt. Dadurch ist der Kommunikationsaufwand im Vergleich mit der aktiven Liste geringer. Diese Simulationen werden dann mit den gleichen Eingaben, die in der Tabelle 6.4 zu sehen sind, wiederholt. Dabei werden die Gruppen gleichzeitig dem Netzwerk beitreten und es verlassen. Wie die Tabelle 6.4 zeigt, werden die gleichen Simulationen für zwei, drei, vier und fünf Ringe durchgeführt.

	Partition-Datei		Event-Datei		
	Gruppenname	Anzahl an Knoten	Gruppen Nr	Event	Zeitpunkt (s)
2 Ringe	Germany	500	1	LEAVE	2000
	France	500	2	LEAVE	2000
			1	JOIN	4000
			2	JOIN	4000
3 Ringe	Germany	300	1	LEAVE	2000
	France	400	2	LAEVE	2000
	Belgium	40	3	LEAVE	2000
			1	JOIN	4000
			2	JOIN	4000
			3	JOIN	4000
4 Ringe	Germany	200	1	LEAVE	2000
	France	150	2	LAEVE	2000
	Belgium	300	3	LEAVE	2000
	Holland	250	4	LEAVE	2000
			1	JOIN	4000
			2	JOIN	4000
			3	JOIN	4000
			4	JOIN	4000
5 Ringe	Germany	250	1	LEAVE	2000
	France	200	2	LAEVE	2000
	Belgium	100	3	LEAVE	2000
	Spain	200	4	LEAVE	2000
	Holland	150	5	LEAVE	2000
			1	JOIN	4000
			2	JOIN	4000
			3	JOIN	4000
			4	JOIN	4000
			5	JOIN	4000

Tabelle 6.4: Die Eingabedateien für die automatische Zusammenführung (Einfaches Szenario)

6.1.3 Kompliziertes Szenario

Dieses Szenario wird sowohl mit der aktiven Liste als auch mit der passiven Liste durchgeführt. Alle Algorithmen (Chord-Zip, Ring Unification, Gossip-based Unifiction, Ring Reunion und seine parallelisierte Version) werden dann mit diesem Szenario hintereinander simuliert und evaluiert. Die Tabelle 6.5 zeigt die Eingaben, die für dieses Szenario verwendet wurden. Diese Simulation wurde nur für 3 Ringe durchgeführt. Dabei werden verschiedene Eintrittszeiten bzw. Austrittszeiten definiert, d. h. die Gruppen werden zu verschiedenen Zeiten aus dem Netzwerk austreten bzw. dem Netzwerk beitreten.

	Partition-Datei		Event-Datei		
	Gruppenname	Anzahl an Knoten	Gruppen Nr	Event	Zeitpunkt (s)
3 Ringe	Germany	300	1	LEAVE	1600
	France	300	2	LAEVE	2200
	Belgium	400	3	LEAVE	2800
			1	JOIN	3400
			2	JOIN	4000
			3	JOIN	4600

Tabelle 6.5: Die Eingabedateien für die automatische Zusammenführung (Kompliziertes Szenario)

6.2 Ergebnisse

In diesem Abschnitt werden die Evaluierungsergebnisse der im vorigen Abschnitt durchgeführten Simulationsszenarien vorgestellt. Diese Ergebnisse werden dann anhand von Grafiken, die von den Pointer-Dateien erstellt wurden, veranschaulicht. Dabei werden die wichtigen Änderungen während der Simulationen beschrieben und interpretiert. Allerdings wird nicht auf alle Details eingegangen, da alle simulierten Szenarien gute Ergebnisse geliefert haben. Dabei wurde gezeigt, dass das entworfene Konzept für die Netzwerkpartitionierung und alle implementierten Merging-Algorithmen gut funktioniert und dadurch die gesetzten Ziele dieser Arbeit erreicht wurden.
Im Folgenden werden die Ergebnisse von jedem vorgestellten Szenario sowohl für die Netzwerkpartitionierung als auch für die Merging-Algorithmen vorgestellt und diskutiert.

6.2.1 Netzwerkpartitionierung

Um die Funktionalität des Konzepts zur Netzwerkpartitionierung zu zeigen, wurden zwei Szenarien durchgeführt: ein einfaches und ein kompliziertes Szenario. Der Fokus dabei war auf die Eventzeiten, die in der Event-Datei eingegeben wurden, gerichtet, d. h. in beiden Szenarien wurden gleiche Gruppen mit gleichen Eigenschaften erstellt, die von der Partition-Datei eingelesen wurden, aber mit unterschiedlichen Eingaben in der Event-Datei. In dem einfachen Szenario werden alle Gruppen gleichzeitig das Netzwerk verlassen, in dem komplizierten hingegen werden die Gruppen hintereinander aus dem Netzwerk austreten.

Die erzielten Ergebnisse dieser Szenarien werden im Folgenden diskutiert.

Einfaches Szenario

Die in Abbildung 6.1 dargestellten Ergebnisse zeigen die Änderung der Knotenanteile an korrekten Pointern während des Simulationsablaufs. Die zusammengestellten Grafiken stellen die Fähigkeit von dem im Kapitel 4 vorgestellten Konzept zur Bildung von mehreren Anzahlen an getrennten Ringen dar. In der Aufbauphase (zwischen t=0 min und t= 16min) steigt dieser Anteil, bis alle Knoten (100 Prozent) die korrekten Pointer enthalten. Zum Zeitpunkt t=33 verlassen alle Gruppen gleichzeitig das Netzwerk. Dadurch nimmt der Anteil an Knoten mit korrekten Pointern zügig ab, was die Trennung der Gruppen und der Bildung von verschiedenen Ringen spiegelt.

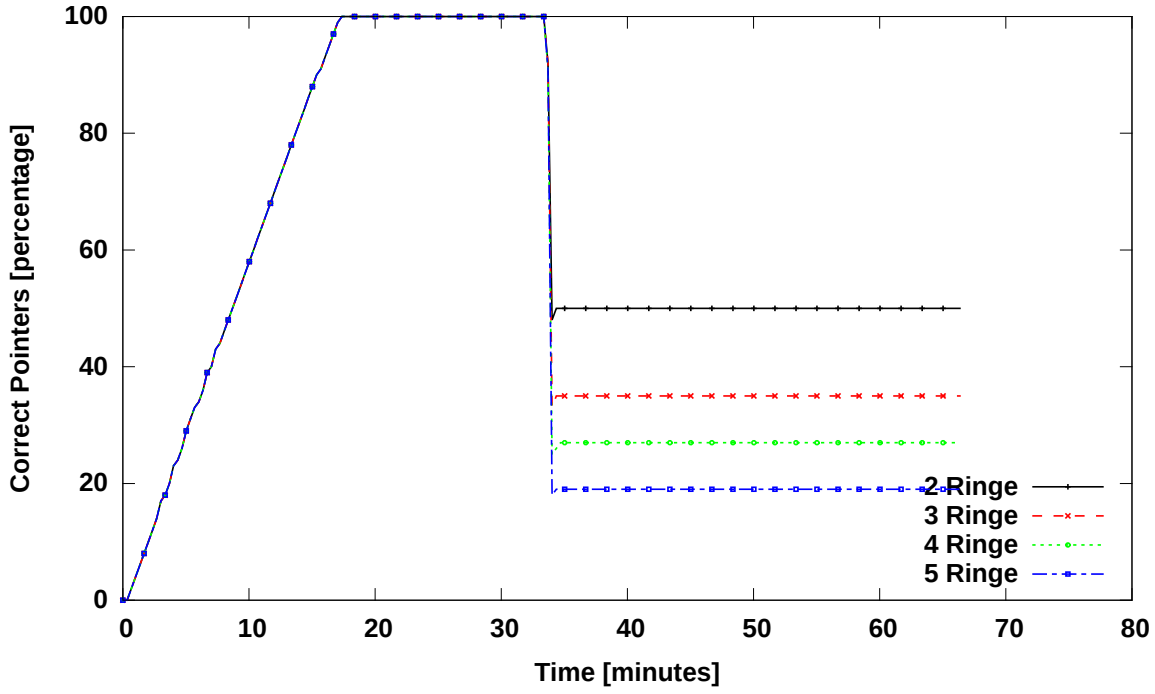


Abbildung 6.1: Einfaches Szenario zur Netzwerkpartitionierung

Für 2 Ringe enthalten 50 Prozent der Knoten, die korrekte Pointer haben, was die Hälfte der Knoten entspricht, d. h. zwei getrennte Ringe sind dadurch entstanden. Für 3 Ringe gibt es 35 Prozent der Knoten mit korrekten Pointern, was einem Drittel der Knoten entspricht, d. h. es sind 3 getrennte Ringe, die gebildet wurden. Für 4 Ringe ist der Anteil 27 Prozent, während für 5 Gruppen nur 19 Prozent der Knoten die korrekten Pointer erreicht haben. Dadurch wurde gezeigt, dass die Effekte des entworfenen und implementierten Konzepts im SimpleUnderlay bezüglich der Netzwerkpartitionierung zur

Bildung von mehreren getrennten Overlay-Gruppen geführt haben.

Kompliziertes Szenario

Die Abbildung 6.2 zeigt die Ergebnisse des Szenarios im Abschnitt 6-1. Dabei wurde ein kompliziertes Szenario durchgeführt, indem die vorhandenen Gruppen das Netzwerk zu verschiedenen Zeiten verlassen. Das Netzwerk dabei besteht aus 5 Gruppen, und die gesamte Anzahl der Knoten ist 1000. Die Gruppen am Anfang der Simulation sind alle im Netzwerk verbunden, wie die Abbildung zeigt. Da sieht man, dass der Knotenanteil mit korrekten Pointern (nach der Aufbauphase) 100 Prozent ist. In diesem Szenario verlässt die erste Gruppe zum Zeitpunkt t=2000 das Netzwerk. Zu diesem Zeitpunkt sinkt in der Abbildung der Knotenanteil und erreicht somit 65 Prozent, d. h. zwei Ringe sind dabei entstanden. Also alle Knoten, die zur ersten Gruppe gehören, bilden miteinander einen Ring, und der Rest bildet einen größeren Ring. Zu dem Zeitpunkt t=2600 s verlässt die zweite Gruppe das

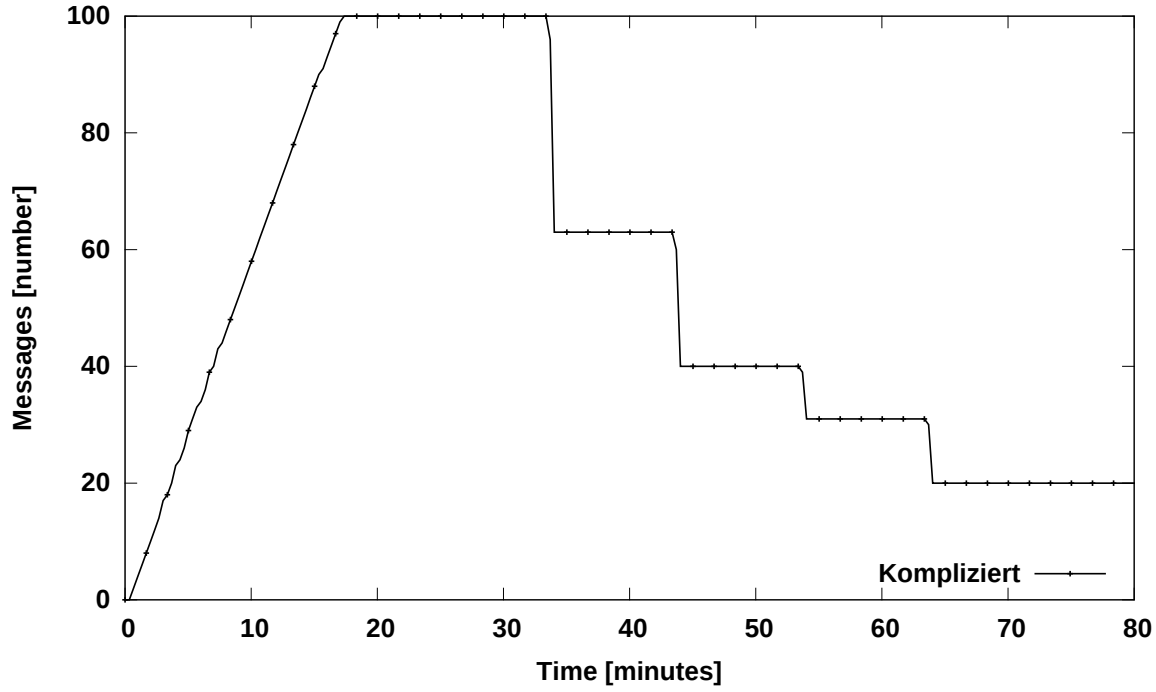


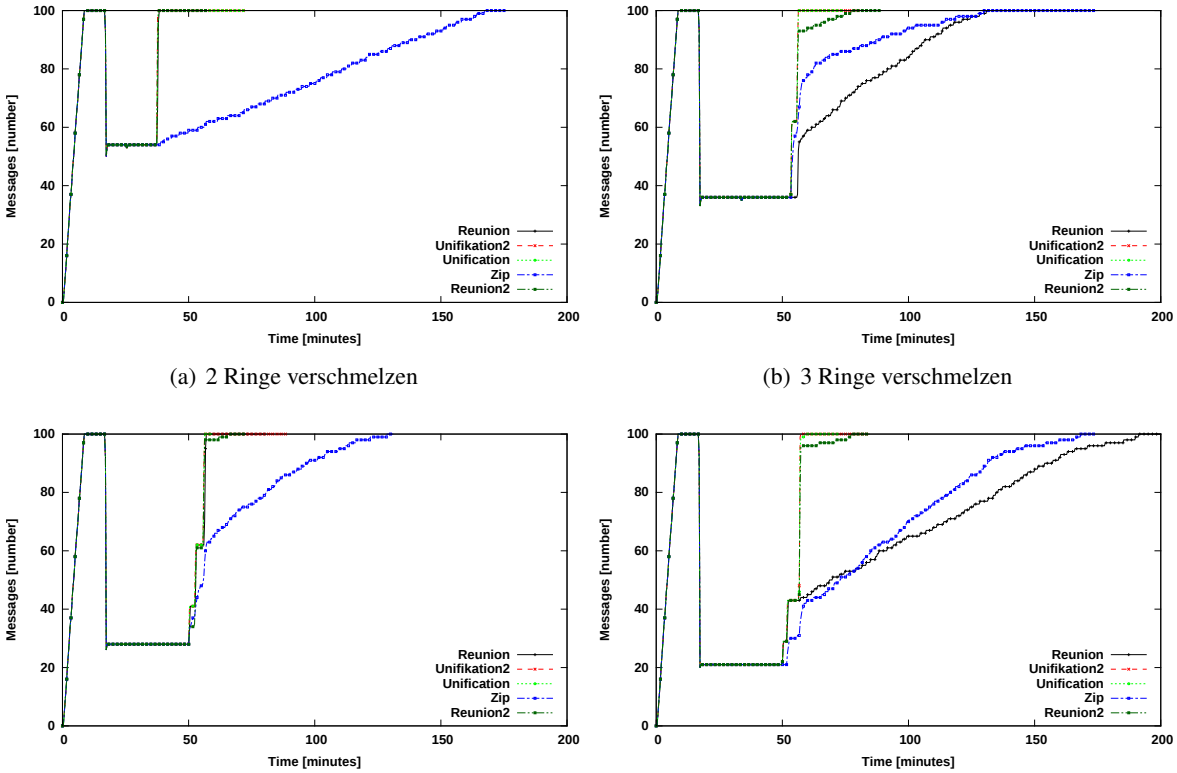
Abbildung 6.2: Kompliziertes Szenario zur Netzwerkpartitionierung

Netzwerk. Somit sinkt der Anteil wieder und erreicht 40 Prozent. Dabei werden drei Ringe entstehen. Ein Ring besteht aus den Knoten der ersten Gruppe, der zweite Ring besteht dann aus den Knoten der zweiten Gruppe, der letzte aus dem Rest der Gruppen. In t=3200s wird die dritte Gruppe aus dem Netzwerk austreten. Dann sinkt der Anteil und erreicht 38 Prozent. Die letzte Änderung ist in der t=3800. Da wird die Netzwerkverbindung zwischen allen Gruppen ausgeschaltet. Der Anteil der

Knoten, die korrekte Knoten haben, erreicht dann nur 20 Prozent d. h. jede Gruppe bildet einen Ring, der von allen anderen Ringen getrennt ist. Somit werden hintereinander mehrere Anzahlen an Ringen in Abhängigkeit von der Austrittszeit der Gruppen aus dem Netzwerk gebildet. Das Konzept hat damit die Möglichkeiten gezeigt, dass mehrere Events zu verschiedenen Zeiten und mit verschiedenen Eventnachrichten mehrere getrennten Gruppen bilden können.

6.2.2 Manuelle Liste

Mit der manuellen Liste wurden alle Merging-Algorithmen simuliert. Diese Simulationen wurden mit mehreren Anzahlen an Gruppen (2, 3, 4 und 5) wiederholt. Die Abbildungen stellen die Ergebnisse dieser Szenarien dar. Das Schaubild 6.3 (a) liefert die Ergebnisse vom Verschmelzen von 2 Ringen. Dabei wird erkannt, dass alle Merging-Algorithmen (Chord-Zip, Simple Unification Ring, Gossip-based Ring Unification, Ring Reunion und seine verbesserte Version) das Zusammenführen erreicht haben. Zum Zeitpunkt t= 2000s haben alle Gruppen gleichzeitig das Netzwerk verlassen, was zum Sinken der Anteile der Knoten in allen Gruppen geführt hat. In der Abbildung sieht man, dass der Anteil zu diesem Zeitpunkt 50 Prozent erreicht hat, d. h. 2 getrennte Ringe sind dabei entstanden. Zum Zeitpunkt t=4000 treten alle Gruppen wieder gleichzeitig dem Netzwerk bei. Da kommt auch die manuelle Liste zum Einsatz, die die Kontaktknoten für den gewählten Initiatorknoten zur Verfügung stellt. Somit werden dann der in default.in gesetzte Merging-Algorithmus gestartet, und die beiden Ringe verschmolzen. Diese Effekte sind in dem Zeitraum zwischen t=50min und t=200min in der Grafik gespiegelt. Da sieht man, dass ab dem Zeitpunkt t= 4000s bzw. 66min der Anteil der Pointer zu steigen beginnt, bis alle Knoten die korrekten Pointer bekommen. Die Schaubilder (b), (c), (d) in 6.3 zeigen das Verschmelzen von 3, 4 bzw. 5 Ringen. Dabei sind ähnliche Ergebnisse erzielt worden wie die beim Zusammenführen von 2 Ringen. Der einzige Unterschied dabei ist der Wert des Knotenanteils, welcher nach der Netzwerkspaltung erreicht wird. Dieser ist abhängig von der Anzahl der gebildeten Ringe.
Alle getrennten Gruppen wurden also mit allen Algorithmen verschmolzen, so dass damit ein globaler Ring gebildet wurde. Der Chord-Zip Algorithmus zeigt in allen Ergebnissen mit der manuellen Liste seine Schwäche. Er hat dabei lange Zeit gebraucht, um die Ringe zu verschmelzen. Weil nur ein Initiatorknoten den Merging-Algorithmus startet, konnten die Nachteile vom Chord-Zip am besten abgelesen werden. Dieser Algorithmus verfügt über keine effizienten Terminierungsverfahren.
Die Knoten leiten die PING-Nachricht, die von dem Initiatorknoten verschickt wurde, einfach weiter, ohne dabei die Position des Senders bzw. Zielknotens zu berücksichtigen. Diese Nachrichten werden also ohne Abbruch entlang der zu verschmelzenden Ringe weitergeleitet, d. h. alle Knoten von allen Ringen werden zusammengeführt, auch wenn sie in der richtigen Position sind. Der Algorithmus terminiert erst, wenn alle Knoten verschmolzen worden sind. Dabei erreicht die PING-Nachricht den letzten Knoten, welcher dem Initialknoten entspricht. Dieser verschickt eine PONG-Nachricht zurück



(a) 2 Ringe verschmelzen

(b) 3 Ringe verschmelzen

(c) 4 Ringe verschmelzen

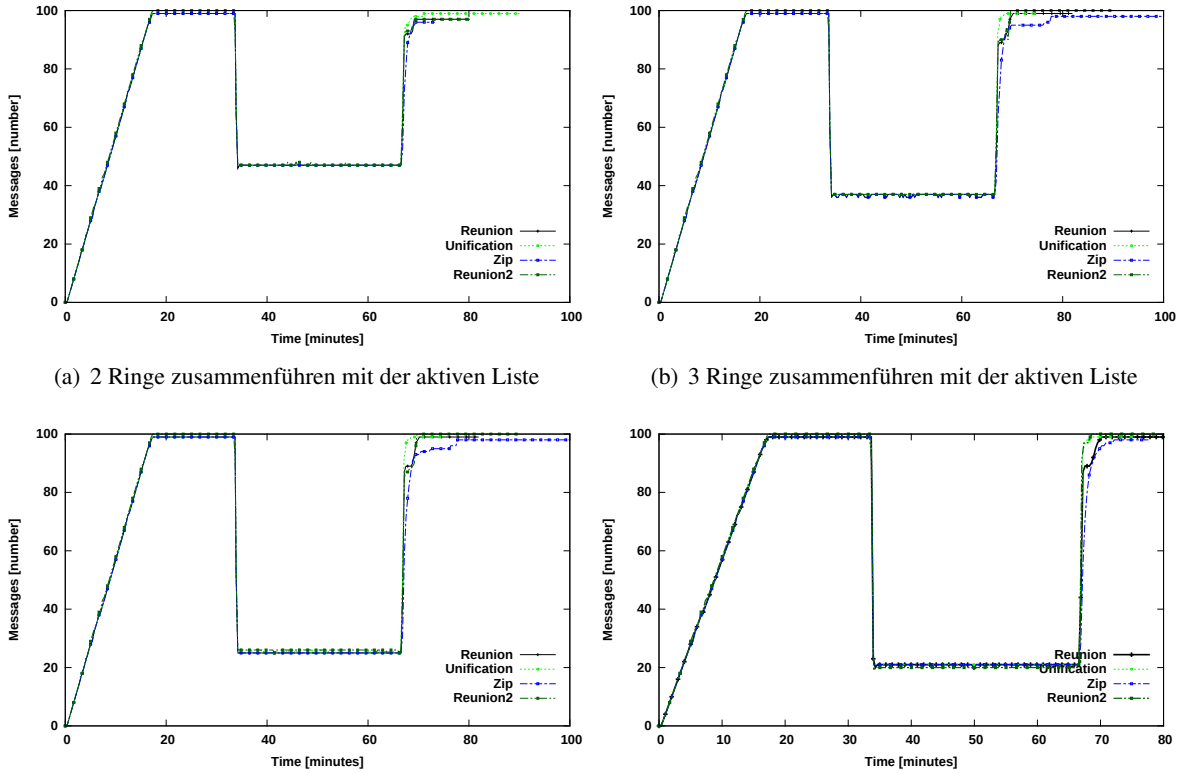
(d) 5 Ringe verschmelzen

Abbildung 6.3: Zusammenführung mit der manuellen Liste

und terminiert den Algorithmus. Dieses Verfahren verlangsamt den Verschmelzungsprozess besonders, wenn nur einzelne Knoten den Algorithmus starten, d. h. es besteht keine Möglichkeit mehrere Instanzen gleichzeitig durchzuführen, was die Laufzeit des Verfahrens beeinflussen bzw. verbessern kann.

6.2.3 Automatische Szenarien

Alle Algorithmen (Chord-Zip, Ring Unification, Gossip-based Unification, Ring Reunion und seine verbesserte Version) wurden sowohl mit der aktiven Liste als auch mit der passiven Liste simuliert, die die Aufgaben dabei übernommen haben, Kontaktknoten für den Initiatorknoten zur Verfügung zu stellen. Diese Simulationen wurden für mehrere Anzahlen an getrennten Ringen (2, 3, 4 und 5 Ringen), die mit dem Konzept der Netzwerkpartitionierung im SimpleUnderlay realisiert wurden, durchgeführt. In diesen Szenarien startet jeder Knoten automatisch anhand der Kontaktknoten (passive bzw. aktive Liste) den gewählten Merging-Algorithmus. Dabei kontaktiert jeder Knoten periodisch die Knoten in der eingesetzten Liste (passive oder aktive Liste), um Kontaktkonten aufzufinden und



(a) 2 Ringe zusammenführen mit der aktiven Liste

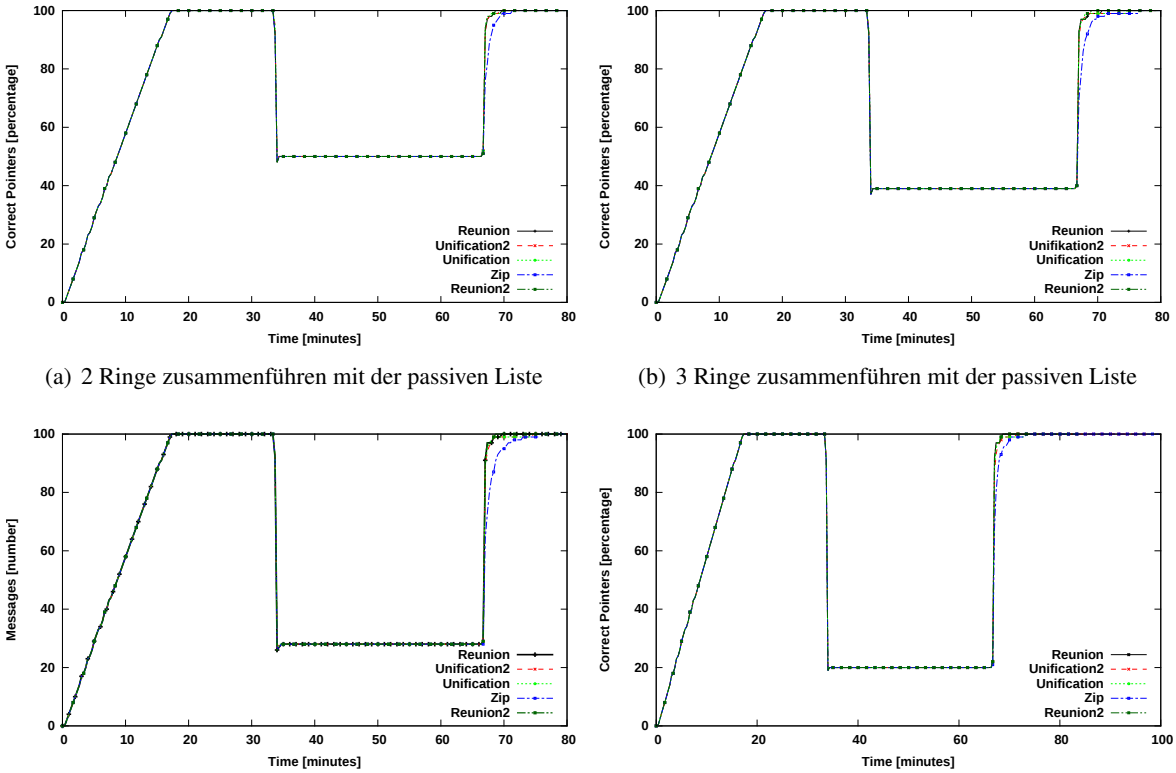
(b) 3 Ringe zusammenführen mit der aktiven Liste

(c) 4 Ringe zusammenführen mit der aktiven Liste

(d) 5 Ringe zusammenführen mit der aktiven Liste

Abbildung 6.4: Einfaches Szenario mit der aktiven Liste

dann den Merging-Algorithmus zu starten. Die Ergebnisse dieser Simulationen zeigen die Abbildungen 6.5 6.4. Dabei erkennt man anhand der Knotenanteile, die über den korrekten Pointer verfügen, die verschiedene Änderungen, die während der Simulation aufgetreten sind. In der Netzwerkaufbauphase zwischen t=0 s und t=1000 s treten die Overlay-Knoten hintereinander dem Netzwerk bei, was mit dem linearen Anstieg des Knotenanteils, die korrekte Pointer haben, zu interpretieren ist. Nach dieser Aufbauphase zum Zeitpunkt t= 1000 enthalten alle Knoten (100 Prozent) die korrekten Pointer, d. h. alle Knoten sind erzeugt und haben die Pointer stabilisiert. Zum Zeitpunkt t=4000 ist dieser Anteil gesunken, bis ein bestimmter Wert, der abhängig von der Anzahl der getrennten Gruppen, die im Netzwerk vorhanden sind, erreicht wird. Diese Änderung reflektiert den Effekt von dem Event LEAVE, das in die Event-Datei eingegeben wurde und dabei den Netzwerkaustritt der Gruppen gefordert hat. Zum Zeitpunkt t=4000 beginnt der Knotenanteil zu steigen, bis 100 Prozent der vorhandenen Knoten korrekte Pointer erhalten werden. Dieser Zeitpunkt entspricht der eingegebenen Zeit in der Event-Datei, um die Eventnachricht JOIN durchzuführen, die darauf hinweist, dass die Gruppen wieder dem Netzwerk beigetreten sind. Dadurch wurde die Fähigkeit aller simulierten Merging-Algorithmen, die die Zusammenführung von verschiedenen Anzahlen an getrennten Ringen erreicht hat, gezeigt. Wenn die Laufzeit der Algorithmen, die das Verschmelzen jeder simulierten Anzahl an bestehenden Ringen (2, 3, 4 und 5 Ringe) erreicht haben, wie die Ergebnisse es zeigen, betrachtet wird, sieht man, dass



(a) 2 Ringe zusammenführen mit der passiven Liste

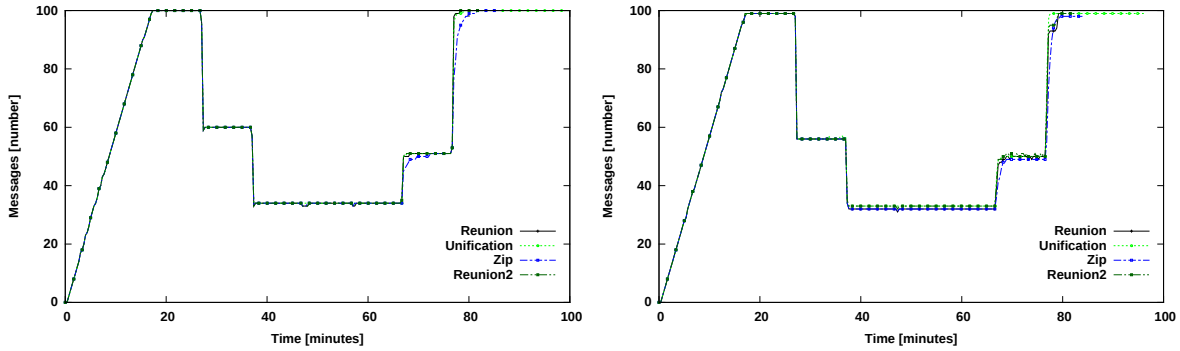
(b) 3 Ringe zusammenführen mit der passiven Liste

(c) 4 Ringe zusammenführen mit der passiven Liste

(d) 5 Ringe zusammenführen mit der passiven Liste

Abbildung 6.5: Einfaches Szenario mit der passiven Liste

kein großer Unterschied bezüglich der Schnelligkeit der Algorithmen festgestellt wurde. Das könnte daran liegen, dass die Änderungen bzw. Erweiterungen bei der Implementierung des Chord-Zip Algorithmus, die durchgeführt wurden, dabei geholfen haben, die Laufzeit im Vergleich mit der erzielten Leistung in PeerfactSim [12] zu verbessern. Der Pseudocode vom Chord-Zip definiert eine Methode combine(), die durchgeführt werden sollte, wenn der Knoten die PING- und PONG-Nachrichten bekommt, welche die alternativen Nachfolger bzw. Vorgänger und der Nachfolger-Tabelle enthält. Was genau diese Methode macht, wurde in dem Code nicht beschrieben. Von daher wurde die Methode so implementiert, dass der Knoten, der PING-Nachrichten mit der Tabelle und den alternativen Nachfolger bekommt, zunächst überprüft, ob der alternative Nachfolger zwischen eigener ID und der Nachfolger-ID liegt. Dabei wird der gesendete alternative Nachfolger in die Routing-Tabelle aufgenommen, wenn er sich dazwischen befindet.
Wenn der Knoten die PONG-Nachricht bekommt, die vom Empfänger der PING-Nachricht zurückgesendet wird, führt er auch die Methode combine() durch. Diesmal aber wird die Position des Vorgängers überprüft, d. h. es wird verglichen, ob der Sender der PONG-Nachricht zwischen dem Vorgänger und der Position des Knotens liegt. Dann wird der Sender als neuer Vorgänger gesetzt, falls seine ID dazwischen passt. Jeder Knoten startet automatisch den gesetzten Algorithmus, d. h. jeder Knoten ist ein Initiatorknoten und versucht, alle Knoten, die in der Liste erreichbar sind, zu verschmelzen.



(a) Kompliziertes Szenario mit der passiven Liste

(b) Kompliziertes Szenario mit der aktiven Liste

Abbildung 6.6: Kompliziertes Szenario mit der automatischen Zusammenführung

Dadurch werden in Chord-Zip mehrere Parallele Instanzen durchgeführt, was dazu führt, das oben beschriebene Terminierungsproblem zu vernachlässigen, wenn nur ein Initiatorknoten den Algorithmus startet. Die Ergebnisse in den Grafiken zeigen, dass der Ring Unification Algorithmus die beste Laufzeit erzielt hat. Das liegt daran, dass dieser Algorithmus ein effektives Verfahren bietet. Dabei sind zwei Merging-Runden gleichzeitig in verschiedenen Ringen gestartet. Dadurch wird die Laufzeit des Algorithmus verbessert, und somit wird das Verschmelzen der verschiedenen getrennten Ringe schneller erreicht.

6.2.4 Komplizierte Szenarien für die Merging-Algorithmen

In diesem Szenario wurde jeder Merging-Algorithmus sowohl mit der aktiven Liste als auch mit der passiven Liste simuliert. Dabei wurden die Austrittszeiten bzw. Eintrittszeiten von jeder im Netzwerk vorhandenen Gruppe betrachtet. Die Gruppen werden also zu verschiedenen Zeiten aus dem Netzwerk austreten und dann wieder hintereinander dem Netzwerk beitreten. Dieses Szenario entspricht einem realistischen Szenario, indem die Knoten bzw. Gruppen von Knoten zu verschiedenen Zeiten dem Netzwerk beitreten. Diese Simulationen wurden nur mit drei Gruppen durchgeführt. Ziel ist es, die Kompatibilität der Algorithmen mit der kombinierten Kontaktliste zu allen möglichen Szenarien zu zeigen. Die dargestellten Ergebnisse in der Abbildungen 6.6 (a) (mit passiver Liste) und (b) (mit aktiver Liste) haben bestätigt, dass die Fähigkeit aller simulierten Algorithmen sowohl mit der aktiven Liste als auch mit der passiven Liste nach der Bildung von mehreren Ringen zu verschiedenen Zeiten besteht, die Ring-Topologie wiederherzustellen. Diese Ringe, die im Netzwerk verfügbar sind, können aber wegen der getrennten Topologie nicht miteinander kommunizieren.

6.3 Zusammenfassung

Dieses Kapitel hat die Ergebnisse der Evaluierung der Netzwerkpartitionierung
und der Merging-Algorithmen im Simulator OverSim geliefert. Die eingeführten Szenarioeingabedateien im Underlay haben ermöglicht das Erstellen von verschiedenen Szenarien, um die Netzwerkpartitionierung und Merging-Algorithmen zu verifizieren. Diese Szenarien wurden dabei anhand der Pointer-Datei evaluiert und bewertet. Somit konnten alle Algorithmen miteinander verglichen und interpretiert werden. Die Evaluierung hat dann gezeigt, dass das entwickelte Konzept zur Isolierung von verschiedenen eingeführten Regionen im Underlay und die implementierten Merging-Algorithmen gut funktioniert haben. Somit wurden alle in dieser Arbeit gesetzten Ziele erfüllt.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden verschiedene Verfahren und Techniken eingeführt, um ein Netzwerkpartitionierungskonzept im Underlay zu entwickeln und Merging-Algorithmen im Overlay zu implementieren.
So wurde in Kapitel 4 das entworfene Netzwerkpartitionierungskonzept vorgestellt. Dieses Konzept hat die Partitionierung im Underlay und die Bildung von isolierten Knotengruppen mit Hilfe des OverSim Framwork ermöglicht. Dabei wurden Szenarioeingabedateien eingeführt, um unterschiedlichen Gruppen, die verschiedene Anzahl an Knoten und zugewiesenem Gruppennamen enthalten können, zu bilden. Um Overlay-Partitionen zu realisieren wurden Events eingeführt, die Konnektivität zeitweise zwischen Knotengruppen einschränken. Somit wurden verschiedene Events durchgeführt, analysiert und bewertet.

Die modulare Architektur in OverSim, die vom Omnet++ übernommen wurde, bietet keine direkte Lösung für die Netzwerkpartionierung, da die Overlay-Knoten unabhängig voneinander implementiert und erzeugt werden. Der Overlay-Knoten im OverSim wird vom Churngenerator, welcher ein Komponent im Underlaynetwork darstellt, erzeugt. Dieses Modul bestimmt alle Parameter, die ein Overlay-Knoten besitzen soll. Dabei werden diese Parameter von dem Nutzer festgelegt, die von der Konfigurationsdatei eingelesen wird. Im Underlay werden dann diese Knoten mit allen dazu gehörigen Schichten dargestellt. Dabei wird genau anhand der NED-Dateien beschrieben, wie die Vernetzung zwischen alle Schichten etabliert, ohne die Kommunikation mit anderen Overlay-Knoten zu berücksichtigen. Dadurch ist keine Underlay-Topologie zu erkennen. Das Problem hat allerdings gefordert eine Lösung zu finden, indem die Information genutzt wird, dass das UDP Modul (bzw.TCP) die Schnittstelle darstellt, die die Kommunikation mit der Underlay Abstraktion und somit mit alle anderen Overlay-Knoten steuert. Jeder Knoten implementiert das UDP Modul (und TCP Modul) als Schnittstelle, um Nachrichten zu empfangen bzw. zu verschicken. Basierend darauf wurde die Lösung entworfen, indem die empfangene bzw. gesendeten Nachrichten anhand der IP-Adressen und aller benötigten Information gefiltert und aussortiert werden. Im Kapitel 5 wurden bekannte Verfahren bzw. Algorithmen, für Chord-Overlay implementiert. Diese implementierten Merging-Algorithmen wur-

den eingesetzt, um die getrennten Overlay-Partitionen wieder zu verschmelzen und zu evaluieren. Gleichzeitig wurden die Merging-Algorithmen genutzt, um die Netzwerkpartitionierung zu verifizieren. Die dabei implementierten Listen um Kontaktknoten aufzufinden, ermöglichen jedem Overlay-Knoten das automatische Starten des Verschmelzungsverfahrens. Diese Listen stellen Kontaktknoten zur Verfügung, um jeden Knoten periodisch den Mergign Algorithmus durchführen zu lassen, und somit die Kommunikation zwischen allen ausgefallenen Knoten wiederherstellen, wenn die wieder erreichbar sind.

Der Einsatz dieser Listen hat den Kommunikationsaufwand beeinflusst. Das hat sich gespiegelt durch der steigenden Anzahl der vertauschten Nachrichten während der Simulation. Besonderes aufwendig war der Simulation mit der aktiven Liste. Diese Liste bekommt zufällige Knoten beim Beitreten zum Netzwerk. Die Knoten, die in der Liste enthalten sind, werden periodisch mit PING-Nachrichten kontaktiert. Alle darauf antwortenden Knoten werden dann zusammengeführt.

Der Unterschied dabei zwischen dieser und der passiven Liste, die das gleiche Verfahren durchführt, liegt darin, dass die passive Liste nur Knoten, die nicht mehr verfühgbar sind, enthält, und nur wenn die wieder erreichbar sind, werden sie zusammengeführt und dann direkt danach gelöscht. Mit der aktiven Liste werden die Knoten nicht entfernt, wenn sie wieder erreichbar sind. D. h. die PING-Nachrichten werden weiterhin gesendet und dann die Knoten zusammengeführt, obwohl sie zu dem gleichen Ring gehören. Dadurch ist der Kommunikationsaufwand deutlich grö/sser als beim Einsatz der passiven Liste. Als Lösung könnte man in der Zukunft die Liste erweitern, so dass überprüft wird, ob die Knoten in gleichen Ringen wie der Initatorknoten und verfügbar sind. In diesem Fall werden diese nicht mit PING-Nachrichten kontaktiert.

Der Parameter successorList, der die Länge der Nachfolgerliste bestimmt, hat beim Testen des Netzwerkpartitionierungskonzepts verschiedene Ergebnisse geliefert, die meistens nicht den Erwartungen entsprechen. Dabei hat dieser Wert einen Einfluss auf die Bildung der getrennten Ringe. Für den Wert 8 hat für Knotenanzahl= 100, die in 3 Gruppen gehören, nur kleine Ringe ergeben. Der Rest des Knotens sind einfach als getrennter Knoten dargestellt. Dabei hat die Anzahl der Knoten in jeder Gruppe auch eine Rolle gespielt. Der Einfluss dieses Parameters auf der Bildung von den isolierten Gruppen wäre sinnvoll, wenn es weiter erforscht würde.

Die Zuweisung der IDs in OverSim basiert auf zufälliger Zuordnung von Koordinaten, die die Position vom Overlay-Knoten bestimmt. Anhand dieser Position mit der IP-Adresse wird die ID erstellt. Somit wierden die Latenzen berechnet und simuliert. Als Idee könnten vielleicht diese Dateien verwendet werden, um Knoten-Partitionen zu erstellen, dann dabei die Latenz zu analysieren und zu simulieren.

Interessant wäre auch wenn dieser Partitionierungen mit Sicherheitsverfahren kombiniert werden, so

dass ein Anonymisierungskonzept für die vertauschten Nachrichten durchgeführt und bewertet werden.

Zusammenfassend kann festgestellt werden, dass die gesetzten Ziele dieser Arbeit erreicht wurden. Einerseits haben die Bildung der getrennten Regionen und dann die zeitweise Einschränkung der Konnektivität zwischen unterschiedlichen Knotengruppen anhand der Szenarioeingabedateien gute Ergebnisse geliefert. Andererseits hat die Implementierung der Merging Algorithmen gut funktioniert und die Kommunikation wurde von alle getrennten Overlay-Ringen wieder hergestellt.

Literaturverzeichnis

[1] Florian Rötzer. DAS GLOBALE GEHIRN.

[2] . ibrorum prohibitorum. http://www.aloha.net/ mikesch/ILP-1559.htm.

[3] . praesidentschaftswahlen-in-iran-12-06-2009. http://www.bpb.de/politik/hintergrund-aktuell/69373/praesidentschaftswahlen-in-iran-12-06-2009.

[4] e-teaching.

[5] Bernhard Heep. *Effizientes Routing in strukturierten P2P Overlays*. KIT Scientific Publishing, 2012.

[6] Ingmar Baumgart. *Verteilter Namensdienst für dezentrale IP-Telefonie*. KIT Scientific Publishing, 2011.

[7] Ralf Steinmetz and Klaus Wehrle. Peer-to-peer-networking &-computing. *Informatik-Spektrum*, 27(1):51–54, 2004.

[8] Christian Schindelhauer. Algorithmen für peer-to-peer-netzwerke. *Vorlesungsskript der Fakultät EIM, Institut für Informatik Universität Paderborn*, 2004.

[9] Alexander Prohaska and Gerhard Weikum. Visualisierung einer verteilten hashtabelle (chord).

[10] Werner Gaulke. Chord und varianten, 2007.

[11] Z Kis and Robert Szabo. Chord-zip: a chord-ring merger algorithm. *Communications Letters, IEEE*, 12(8):605–607, 2008.

[12] Tobias Amft. Design, implementation and evaluation of merging mechanisms for large-scale and dynamically partitioned networks. 2013.

[13] Tallat M Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with network partitions in structured overlay networks. *Peer-to-Peer Networking and Applications*, 2(4):334–347, 2009.

[14] Xuemin Sherman Shen, Heather Yu, John Buford, and Mursalin Akon. *Handbook of peer-to-peer networking*, volume 34. Springer Science & Business Media, 2010.

[15] II OVERLAY-FRAMEWORK OVERSIM. Oversim: Ein skalierbares und flexibles overlay-framework für simulation und reale anwendungen. 2009.

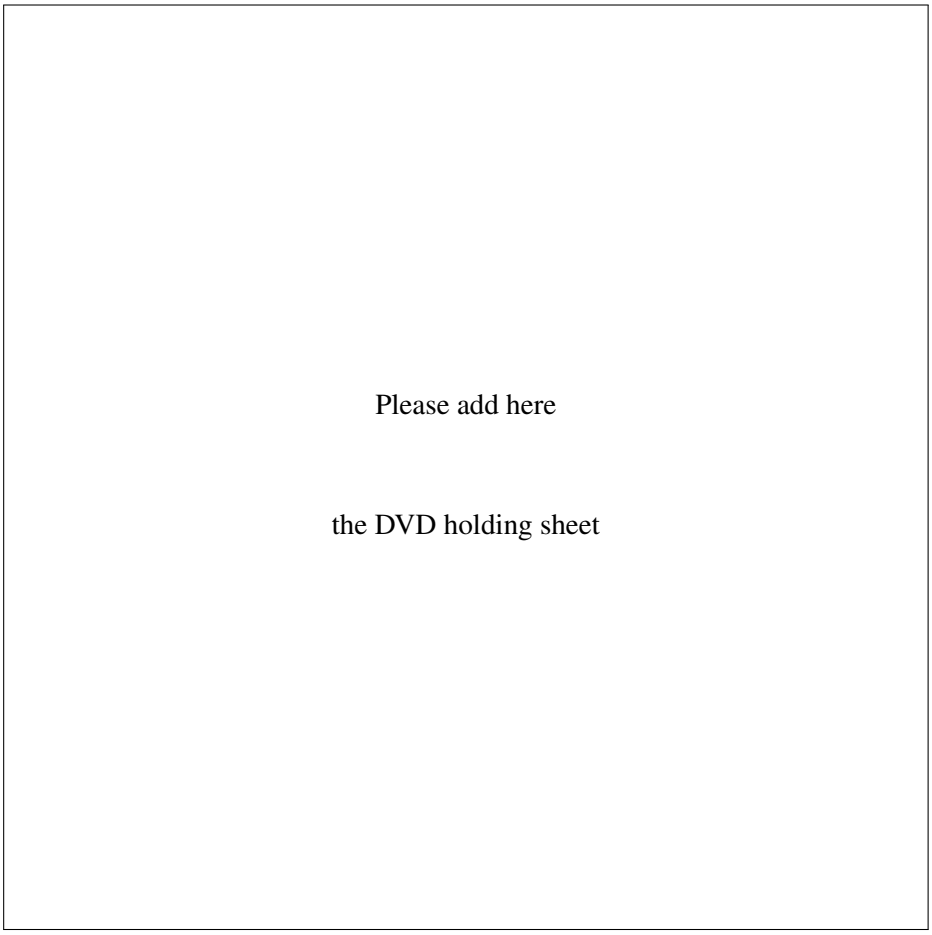
[16] . OMNeT++ documentation and tutorials. http://www.omnetpp.org/pmwiki/index.php?n=Main.Omnetpp4.

[17] . Implementing new overlay modules in OverSim. http://www.oversim.org/wiki/OverSimDevelop.

Ehrenwörtliche Erklärung

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

Düsseldorf, 23.März 2015 Ilham Amara



This DVD contains:

- A *pdf* Version of this bachelor thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- **[adapt]** The source code of the software that was created during the bachelor thesis
- **[adapt]** The measurment data that was created during the evaluation
- The referenced websites and papers