Diplomarbeit

Join in Peer-to-Peer Systems: The Steady State Statistics Service Approach

Michael Gähwiler

Betreuung: Ruedi Arnold
Leitung: Prof. Dr. Roger Wattenhofer

Distributed Computing Group
Departement Informatik
ETH Zürich

Sommer 2003

Zusammenfassung

Diese Diplomarbeit stellt den Steady-State-Statistics-Service, kurz *4S*, vor. *4S* ist ein dezentraler Dienst, der approximativ statistische Informationen über ein Peer-to-Peer, kurz P2P, System sammelt und diese Informationen in einem P2P-System propagiert.

Als beispielhafte Applikation wird gezeigt, wie man *4S* verwenden kann, um Peers in ein P2P-System einzufügen. Es werden dabei zwei Join-Algorithmen vorgestellt: *Number Join* und *Depth Join*. Beide Algorithmen fügen neue Peers deterministisch in ein P2P-System ein, indem Peers gezielt in bestimmte Regionen des P2P-Systems geleitet werden.

Für die Simulation der Algorithmen wird ein Framework auf der Basis einer Event-Driven-Simulation erstellt. Das Framework erlaubt eine einfache Integration zusätzlicher Applikationen. Der modulare Aufbau ermöglicht darüber hinaus einen einfachen Austausch von Modulen und dadurch die Simulation an andere Sachlagen anzupassen.

Durch Simulationen mit realitätsnahen Parametern kann gezeigt werden, dass Einfüge-Operationen auf der Basis von *4S* zu einem balancierten P2P-System führt.

Wir sehen *4S* als einen Grundbaustein in einem P2P-System, welcher von unterschiedlichsten Applikationen verwendet werden kann.

Abstract

This diploma thesis introduces the Steady-State-Statistics-Service *(4S)*. *4S* is a decentralised service which collects approximate statistical information about a Peer-to-Peer (P2P) system and propagates this information into a P2P system.

An example application illustrates how *4S* can be used in order to insert peers into a P2P system. Two Join-Algorithms are introduced: *Number Join* and *Depth Join*. Both Algorithms deterministically insert new peers into a P2P system in which the peers are guided towards "sparse areas".

A framework for the simulation of algorithms is drawn up on the basis of an Event-Driven-Simulation. The framework allows for an easy integration of additional applications. The modular construction makes a simple exchange of modules possible, thereby making the simulation suitable for other situations.

Simulations using realistic data show that insertion-operations based on *4S* lead to a balanced P2P system. In this system the join algorithms transfer an imbalanced peer into a balanced P2P system.

4S can be described as a basic tool of a P2P system which in particular can be used by different applications.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Peer-to-Peer Systeme

Kaum ein Haushalt, Schule oder auch Arbeitsplatz, welcher keine Verbindung zum Internet aufweist. Die Zahl der über das Medium Internet verbundenen Rechner geht in die Millionen. Rechner nützen in der Regel nur einen Bruchteil ihrer Ressourcen und die dabei brachliegende Rechen- und Speicherkapazität ist enorm.

Peer-to-Peer (P2P) Systeme ziehen Nutzen aus dieser Tatsache, indem sie Rechner und Netzwerk zu einem System mit *enormer Rechner- und Speicherkapazität* zusammenfassen. Bekannteste Anwendung dieser Philosophie ist der Austausch von Multimediadateien, wie sie bei Napster und Gnutella vorzufinden ist. Es sind aber auch Bestrebungen im Gange, P2P-Systeme weg vom einfachen Dateitausch hin zu ausgefeilteren Aufgaben zu führen.

In einem P2P-System teilen sich alle Peers Ressourcen und Dienste, ohne dass dabei eine zentrale Instanz eingreift. Ein P2P-System ist vollständig *dezentral* und *symmetrisch* aufgebaut, d.h. jeder Peer führt Software aus, die exakt gleich in ihrer Funktionalität ist. Obwohl das System aus unzuverlässigen Desktop-Rechnern aufgebaut ist, kommt es durch den dezentralen Aufbau zu keinem *"single point of failure"*. P2P-System sind deshalb für eine robuste und zuverlässige Speicherung von Daten und rechenintensive Operationen geeignet.

1.1.1 Such-Algorithmen

Ein zentrales Problem bei verteilten Systemen ist das Suchen von Objekten in einem System. Auch bei P2P-Systemen gehört dies zu den prominentesten Problemen.

Eine Möglichkeit Objekte zu finden ist, Schlüssel und Speicherort des Objekts auf einem zentralen Server abzuspeichern. Sucht man ein bestimmtes Objekt, sendet man eine Suchanfrage an den zentralen Server, welcher dann als Antwort den Speicherort zurückliefert. Napster schlägt diesen Weg ein, um Musikdatei auszutauschen. Durch den zentralen Server ist ein so aufgebautes P2P-System aber nicht skalierbar und bei einem Ausfall des Servers ist das ganze System blockiert.

Um eine bessere Skalierbarkeit zu erreichen, ist anstelle eines einzelnen zentralen Servers, auch eine Gruppe von Servern —eingebettet in eine Hierarchie— einsetzbar. Der Domain Name Service[1] (DNS) basiert auf diesem Prinzip. Eine Suchanfrage beginnt an der Spitze der Hierarchie. Falls die Suchanfrage nicht erfolgreich ist, wird sie an die nächste Stufe weitergeleitet. Da ein Objekt seinen festen Platz in der Hierarchie hat, verläuft eine Suche geradlinig durch die Hierarchie der DNS-Server. Der Nachteil liegt wiederum beim Ausfall und in einer grösseren Last eines Servers der weit oben in der Hierarchie angesiedelt ist.

Gnutella schlägt einen gänzlich anderen Weg ein. Sucht man ein Objekt, werden die Nachbarn eines Peers angefragt. Ist das gesuchte Objekt auf einem Nachbar vorhanden, wird das Objekt zurückgesendet. Ansonsten führt der Nachbar die gleiche Suchanfrage mit seinen Nachbarn aus. Durch diesen Mechanismus wird ein P2P-System regelrecht mit Suchanfragen überflutet. Der Gewinn an Stabilität des Systems ("no single point of failure") wird durch den Verlust an Skalierbarkeit erkauft.

Napster und DNS sind in der Lage ein Objekt gezielt zu suchen, da Objekte in eine feste Struktur eingebettet sind. Gnutella besitzt keine solche Struktur; man sucht wahllos auf allen Rechner die man greifen kann. Dafür —bedingt durch den symmetrischen Aufbau— ist es nicht anfällig gegenüber Rechnerausfällen.

Um *Struktur* und *Symmetrie* zu vereinigen wird in vielen P2P-Systemen die Abstraktion einer Distributed Hash Table (DHT) verwendet. Objekte erhalten einen Schlüssel und werden anhand des Schlüssels in das P2P-System eingefügt. Die Peers sind zu einer wohldefinierten Struktur zusammengeschlossen, die ein Suchen über den Schlüssel in $O(log(n))$-Schritten (bei n Peers) ermöglicht.

Dieser Ansatz wird von P2P-Systemen wie CAN[6], Cord[10], Tapestry[11], Pastry[7], Kademlia[3] und von dieser Diplomarbeit angewendet. Eine Übersicht einiger P2P-Systeme, die diesen Ansatz wählen, findet man in [2].

1.1.2 Einfügen von Peers

Eine wichtige Anwendung in P2P-Systemen ist das Einfügen von neuen Peers in das P2P-System. Da P2P-Systeme keinen zentralen Dienst besitzen, der das Einfügen koordinieren kann, werden neue Peers zufällig eingefügt: Einem neuen Peer wird ein zufälliger Schlüssel zugeteilt. Der Peer führt nun eine Suchanfrage für diesen Schlüssel aus. Sobald der Schlüssel aufgefunden wurde, fügt der Verwalter des Schlüssels den Peer ins P2P-System ein.

Man würde jetzt annehmen, dass beim zufälligen Einfügen mit uniformen Schlüsseln die Peers gleich verteilt sind und deshalb alle Peers gleich viele Objekte verwalten. Das trifft aber nicht zu. Beim zufälligen Einfügen tritt das "Ball into Bins" Problem [5] zu Tage. Dieses sagt, dass mit hoher Wahrscheinlichkeit ein hochbelasteter Peer mehr Objekte besitzt, als ein Peer mit durchschnittlich vielen Objekten.

[1] DNS ist wohl nicht das, was man sich unter P2P vorstellt. Trotzdem wird es, als eine Anwendung der verteilten Systeme, erwähnt.

Um das Problem eines unbalancierten P2P-Systems zu lösen, wird in dieser Diplomarbeit ein Einfüge-Algorithmus vorgestellt, der Peers nicht zufällig, sondern gezielt einfügt. Dazu wird ein neuer, dezentraler Dienst eingeführt, der Daten über ein P2P-System sammelt und allen Peers zugänglich macht: der *Steady-State-Statistics-Service*, kurz *4S*.

1.2 Ablauf der Diplomarbeit

Ausgangspunkt für diese Diplomarbeit war eine kurze Beschreibung der Arbeit mit wohldefinierten Aufgaben, welche aber nicht als fixe Ziele verstanden werden sollten. Der Ablauf war dann auch dynamisch und iterativ. Ausgehend von der Idee des Steady-State-Statistics-Service, wurden stückweise Ideen ausformuliert, implementiert und getestet. Die daraus erzielten Resultate flossen dann in einen nächsten Schritt ein, bei dem eventuell neue Wege eingeschlagen und neue Ziele definiert wurden. Die ganze Arbeit kam schlussendlich in einem "Paper" [1] zum Ausdruck, welches die Algorithmen und Resultate in kompakter Form vorstellt.

Die Vorgehensweise war nicht theoretisch, sonder praktisch orientiert. Die praktische Vorgehensweise zeigt sich am deutlichsten in der Erstellung eines Frameworks zur Simulation der hier beschriebenen Ideen. Der Aufbau einer Simulation war dann auch ein wesentlicher Bestandteil der Diplomarbeit, obwohl nicht ausdrücklich in der DA-Beschreibung formuliert. Mit Hilfe der Simulation sollten alle Ideen bestätigt und erhärtet werden. Im Mittelpunkt der Arbeit stand dabei immer die Idee des Steady-State-Statistics-Service.

1.3 Aufbau des Berichts

Das erste Kapitel (dieses hier) gibt einen Abriss über P2P-Systeme, sowie den Ablauf der Diplomarbeit und den Aufbau des Berichts wieder. Danach wird das dieser Diplomarbeit zugrunde liegende P2P-Systems erklärt. Begriffe wie Präfixgruppe, Distributed Hash Table und Routing werden hier aufgegriffen und erläutert. Kapitel 3 geht dann auf die Idee des Steady-State-Statistics-Service *(4S)* ausführlicher ein. Wer wissen will, wie man durch einen rein lokalen Informationsaustausch zu einer globalen Sicht eines P2P-System gelangen kann, ist hier am richtigen Ort. Das nächste Kapitel beschäftigt sich mit dem Einfügen von neuen Peers mit Hilfe des *4S*. In Kapitel 5 wird die Simulationsumgebung vorgestellt. Die Funktionsweise der Simulation und das dazugehörige Framework finden hier eine Erklärung. In Kapitel 6 wird auf die Ergebnisse der verschiedenen Algorithmen des *4S* und Joins eingegangen. Das ganze wird im letzten Kapitel durch ein Fazit abgerundet.

4S und die Join-Algorithmen sind zwar eng miteinander verwoben, trotzdem werden sie als eigenständige Kapitel behandelt, zum Teil mit Bezug zueinander. Dabei greifen diese zwei Kapitel schon auf die Simulation der Algorithmen vor, da gewisse Punkte erst bei der Simulation zum Vorschein kamen.

Kapitel 2

Aufbau des P2P-Systems

2.1 Überblick

Dieses Kapitel stellt das den Algorithmen zugrunde liegenden P2P-System vor. Das in dieser Diplomarbeit verwendete P2P-System entspricht weitestgehend dem P2P-System Kademlia[3].

Im ersten Abschnitt wird die Distributed Hash Table und das Mapping zwischen Objekten und Peers erläutert. Der zweite Abschnitt erläutert wie man durch Unterteilung des P2P-Systems in Präfixgruppen eine Topologie bildet. Der letzte Teil erklärt dann, wie der Routing-Mechanismus funktioniert.

2.2 Distributed Hash Table: Mapping von Daten auf Peers

Der grundlegende Baustein für den Aufbau des in dieser Diplomarbeit verwendeten P2P-Systems ist eine **Distributed Hash Table**, kurz DHT. DHT bildet die Basis für viele P2P-Systeme und findet sich unter anderem in den Systemen CAN[6], Cord[10], Tapestry[11], Pastry[7] und Kademlia[3].

In DHT erhalten alle Objekte, die es in einem P2P-System zu verwalten gilt, einen Schlüssel innerhalb eines festgelegten Adressraumes, d.h. es werden Paare bestehend aus Schlüssel und Objekt gebildet. Zur Erstellung des Schlüssels kommt typischerweise ein Verfahren wie das SHA-1[8] zur Anwendung. Das SHA-1-Verfahren nimmt als Input ein Objekt und berechnet daraus einen Hashwert. Der Hashwert stellt dann den Schlüssel eines Objektes dar. Peers, die am System teilhaben wollen, erhalten eine eindeutige ID, welche aus demselben Adressraum wie der Schlüssel des Objektes gewählt wird. Diese ID wird als **Overlay-ID** bezeichnet. Ein Peer ist nun für diejenigen Objekte verantwortlich, bei denen die grösste Übereinstimmung in Schlüssel und Overlay-ID des Peers besteht. Was aber bedeutet grösste Übereinstimmung genau? Dazu betrachtet man die binäre Darstellung des Schlüssels und der Overlay-ID[1]. Die grösste Übereinstimmung besteht nun bei demjenigen Peer, mit dem längsten gemeinsamen Präfix von Overlay-ID und Schlüssel. Schlüssel/Objekt-Paare werden

[1] Im weiteren Verlauf wird davon ausgegangen, dass die Overlay-ID immer ein Bitstring ist.

demzufolge auf dem Peer, dessen Overlay-ID möglichst nahe beim Schlüssel liegt, abgespeichert.

Beispiel: Dieser Bericht erhält durch Berechnung der SHA-1 Funktion den 160-Bit langen Hashwert 00110..., welcher als Schlüssel dient. Im System befinden sich momentan zwei Peers mit den Overlay-IDs 001 und 0011. Der Bericht wird nun auf dem Peer mit Overlay-ID 0011 abgespeichert, da Overlay-ID 0011 in den ersten vier Bits, Overlay-ID 001 hingegen nur in den ersten drei Bits mit dem Schlüssel übereinstimmt.

2.3 Topologie durch Präfixgruppen

Die Overlay-IDs der Peers kann man als Blätter in einem binären Baum darstellen. Bild 2.1 zeigt wie Peer 0011 in einem beispielhaften binären Baum abgebildet wird. Der Baum wird nun aus der Sicht eines Peers in **Präfixgruppen** $p_i, (i = 1, \ldots, k)$ unterteilt. Die Unterteilung in Präfixgruppen geschieht folgendermassen: Ausgehend vom kürzesten Präfix der Overlay-ID eines Peers wird bei jedem Schritt der Präfix um eins verlängert und das letzte Bit im Präfix invertiert. Alle Peers deren Overlay-ID mit diesem Präfix p_i beginnen, bilden zusammen eine Gruppe. Auf diese Art und Weise gebildete Gruppen bezeichnen wir als Präfixgruppen eines Peers. Beispielsweise besitzt der Peer $p = 0011$, wie in Abbildung 2.1 dargestellt, die Präfixgruppen $p_1 = 1$, $p_2 = 01$, $p_3 = 000$ und $p_4 = 0010$.

Als **komplementäre Präfixgruppe** $\overline{p_i}$ bezeichnen wir die Peers, die über einen gleichen Präfix, aber dessen letztes Bit invertiert ist, verfügen. Die komplementären Präfixgruppen zum letzten Beispiel sind $\overline{p_1} = 0$, $\overline{p_2} = 00$, $\overline{p_3} = 001$, und $\overline{p_4} = 0011$.

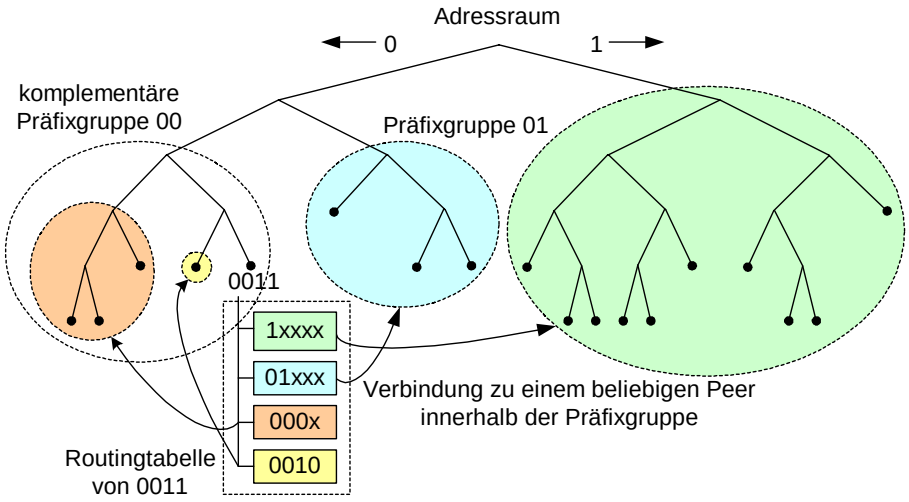

Abbildung 2.1: Aufbau der Topologie: Jeder Peer unterteilt ein P2P-System in Präfixgruppen und unterhält Verbindungen zu diesen Gruppen. Zu jeder Präfixgruppe gibt es eine komplementäre Präfixgruppe. Die Unterscheidung liegt im letzten Bit des Präfix, z.B. hat Präfixgruppe $p_2 = 0\mathbf{1}$ die komplementäre Präfixgruppe $\overline{p_2} = 0\mathbf{0}$.

Um einen Zusammenhalt unter den Peers zu gewährleisten, unterhält jeder Peer eine **Routingtabelle**, welche Verbindungen zu mindestens einem Peer innerhalb der jeweiligen Präfixgruppe abspeichert. Diese Verbindungen nennen wir **Nachbarn** eines Peers. Dabei handelt es sich um unidirektionale Verbindungen. Die Grösse der Routingtabelle eines Peers entspricht der Anzahl Präfixgruppen und besitzt durchschnittlich $log_2(n)$-Einträge, wobei n die Anzahl Peers im P2P-System ist. Werden mehrere Verbindungen pro Präfixgruppe erstellt, erhöht sich die Grösse der Routingtabelle entsprechend. Abbildung 2.1 stellt das wiederum beispielhaft für den Peer 0011 dar.

2.4 Routing-Mechanismus

Die Unterteilung in Präfixgruppen und die damit aufgebauten Verbindungen zwischen den Peers ermöglichen einen eleganten Routing-Mechanismus.

Ein Peer p_A findet einen anderen Peer p_B, indem man die Overlay-ID des gesuchten Peers p_B mit den Präfixgruppen des Peers p_A vergleicht. Die Suchanfrage für den Peer p_B wird dabei an einen Peer p_C innerhalb der Präfixgruppe mit grösster Übereinstimmung weitergeleitet. Dadurch stimmt die Overlay-ID von p_B in mindestens einer Stelle mit p_C überein. p_C starte nun seinerseits eine Suchanfrage für p_B und durch diesen Vorgang konvergiert die Suche eines Peers in einer Präfixgruppe mit nur einem einzigen Peer. Dieser stellt dann den gesuchten Peer p_B dar.

Dieser Mechanismus stellt sicher, dass man mit maximal $O(\log_2(n))$-Schritten jeden Peer lokalisieren kann. Mit folgender Beweisidee kann das gezeigt werden: Bei einem System mit n Peers, braucht es maximal $\log_2(n)$ Bits für die Darstellung der Overlay-ID. Bei jedem Schritt wird die Länge der Übereinstimmung des Präfix um eins erhöht, d.h. die Anzahl Bits in der Übereinstimmung nimmt um eins zu. Daraus folgt, dass man mit maximal $\log_2(n)$ Schritten zum Ziel gelangt.

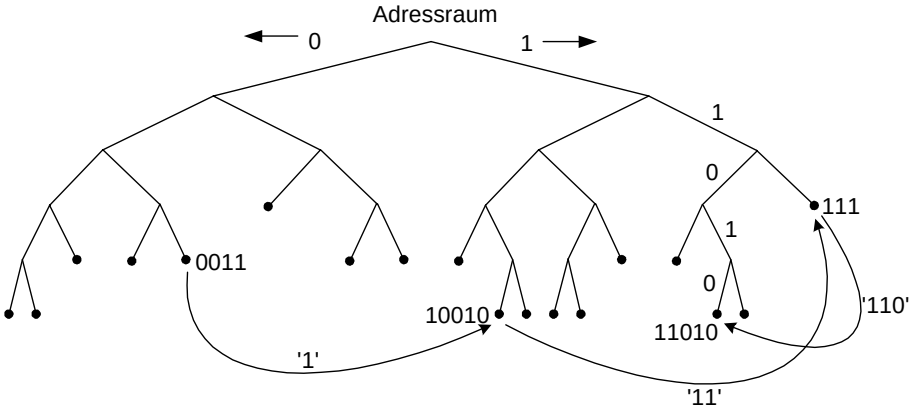

Abbildung 2.2: Beispiel des Routing-Mechanismus: Peer 0011 lokalisiert den Peer 11010 in drei Schritten, indem die Suchanfrage von einer Präfixgruppe zur nächsten weitergeleitet wird.

Beispiel: Peer 0011 sucht den Peer 11010. Der Vergleich der Präfixgruppen zeigt, dass die Suchanfrage an die Präfixgruppe 1 weitergeleitet werden muss. Peer 0011 besitzt eine Verbindung zum Peer 10010 und schickt die Anfrage zu diesem Peer. Die Overlay-ID stimmt in der ersten Stelle (**1**) mit der gesuchten Overlay-ID überein. Peer 10010 startet ebenfalls eine Suchanfrage für den Peer 11010 und leitet die Suchanfrage an den Peer 111 weiter. Dies erhöht die Übereinstimmung auf zwei Stellen (**11**). Der Vorgang wird nun solange wiederholt, bis es zu einer vollständigen Übereinstimmung der Overlay-IDs kommt, d.h. bei Peer 11010 ist das Ziel erreicht. Abbildung 2.2 illustriert den ganzen Ablauf der Suchanfrage.

Da jedes Objekt auf dem Peer mit der grössten Übereinstimmung im Schlüssel und Overlay-ID abgespeichert ist, spielt es keine Rolle, ob das Routing für Peers oder Objekte angewendet wird. Im Beispiel endet eine Suchanfrage für Objekt dessen Schlüssel mit 11010 beginnt, gleichermassen bei Peer 11010.

Kapitel 3

Steady-State-Statistics-Service (*4S*)

3.1 Überblick

Der Steady-State-Statistics-Service, kurz *4S*, stellt einen Dienst dar, der Informationen über ein P2P-System, wie z.B. die Anzahl Peers, verwaltet und darüber hinaus diese Informationen allen Peers zugänglich macht. Dabei handelt es sich um einen dezentralen Dienst, der gleichermassen auf allen Peers vorzufinden ist. *4S* dient als Grundlage für die in Kapitel 4 vorgestellten Join-Algorithmen. Im ersten Teil des Kapitels wird zunächst auf das Grundprinzip des *4S* eingegangen. Der zweite Teil stellt drei Implementierung, die einen Informationsaustausch zwischen den Peers ermöglichen, vor.

3.2 Merkmale und Funktion des *4S*

4S ist ein Dienst, der approximative Informationen über ein P2P-System zur Verfügung stellt. Durch Verwendung von *4S* wissen Peers, wie der momentane Zustande eines P2P-System aussieht. Zustandsinformationen sind dabei z.B. die Anzahl Peers, die Betriebszeit einzelner Peers oder die Bandbreiten im P2P-System. Anhand dieser Informationen kann ein Peer Rückschlüsse über das P2P-Systems ziehen und gezielt eine Applikation basierend auf diesen Information ausführen.

4S ist als dezentraler Dienst ausgelegt und somit auf allen Peers vorhanden. Informationen des *4S* sind nur lokal auf den Peers vorhanden. Berechnungen für den *4S* werden ebenfalls lokal durchgeführt. Ein direkter Datenaustausch findet nur lokal statt, d.h. jeder Peer tauscht *4S* Informationen nur mit seinen Nachbarn aus. Durch den rein lokalen Austausch ist die Skalierbarkeit des *4S* gegeben.

Um zu einem globalen Wissen zu gelangen, kommt folgender Mechanismus zum Einsatz: Jeder Peer p ist ein Experte seiner komplementären Präfixgruppen $\overline{p_i}$, da ja Peers innerhalb einer Präfixgruppe "näher" sind und über genauere Informationen verfügen. Peers in einer komplementären Präfixgruppe $\overline{p_i}$ übernehmen deshalb Informationen eines Peers der zugehörigen Präfixgruppe p_i.

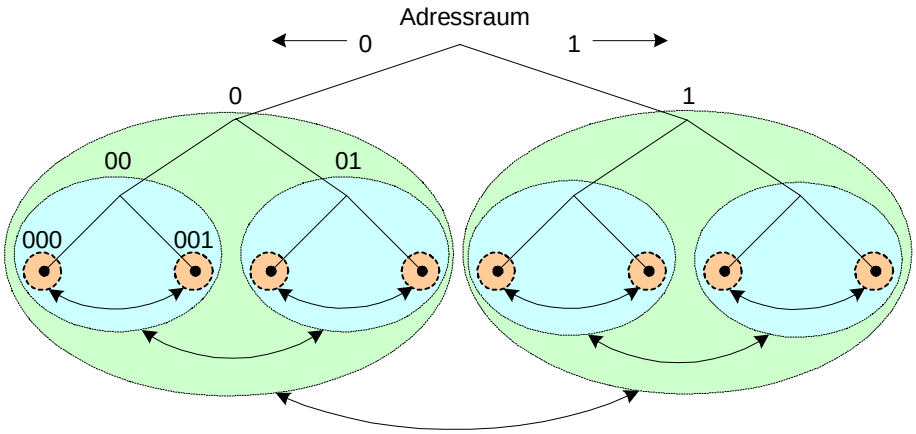

Abbildung 3.1: Der Datenfluss in einem P2P-System. Informationen gelangen immer von einer Präfixgruppe zu einer komplementären Präfixgruppe, da Peers innerhalb einer Präfixgruppe ein "grösseres Wissen" über die lokalen Verhältnisse besitzen und damit ein Experte über die Präfixgruppe ist.

Peers werden nun zu Experten über das ganze P2P-System indem sie Informationen in einer "Buttom-Up"-Manier sammeln. Wir wollen diesen Prozess anhand eines Beispiels (siehe auch Abbildung 3.1) erläutern:

Beispiel: Peer 000 will globales Wissen über das ganze P2P-System erlangen und dadurch zu einem Experten des P2P-Systems werden. Als erstes tauscht Peer 000 Informationen mit dem Peer 001 aus. Zusammen mit den Informationen, die er über sich selber hat (Peer 000 ist Experte der Präfixgruppe 000), ist Peer 000 in der Lage, Informationen über die Präfixgruppe 00 abzuleiten; Informationen über 000 und 001 ergeben zusammen Informationen der Präfixgruppe 00. Peer 001 hat dabei die gleiche Möglichkeit und daher sind Peer 000 und 001 Experten über die Präfixgruppen 00. Um ein Experte der Präfixgruppe 0 zu werden, werden Informationen mit einem Peer der Präfixgruppe 01 ausgetauscht. Informationen der Präfixgruppe 00 und 01, erlauben es, das Peer 000 ein Experte der Präfixgruppe 0 wird. Im letzten Schritt werden noch Informationen mit einem Peer der Präfixgruppe 1 ausgetauscht, danach besitzt der Peer 000 globales Wissen über das P2P-System. Da der Austausch der Werte immer gegenseitig statt findet, werden auch allen anderen Peers zu Experten des P2P-Systems.

Wie man mit diesem Mechanismus, die Anzahl Peers und die minimale Tiefe im P2P-System ermittelt, wird in Kapitel 4.1 resp. Kapitel 4.2 erläutert.

Denkbare Applikationen des *4S* sind z.B.: mit Hilfe des *4S* ein unbalanciertes P2P-System durch einen gezielten Umbau in ein balanciertes P2P-System zu überführen, potente Peers und auch Teilbereiche des P2P-Systems mittels *4S* zu lokalisieren (z.B. anhand der Betriebszeit, Bandbreite, Rechnerkapazität, usw.) und gemäss deren Fähigkeiten einzusetzen, die Gesamtzahl Peers im P2P-System zu ermitteln oder neue Peers gezielt einzufügen. Das Einfügen von Peers stellt dabei die Applikation die im Mittelpunkt dieser Arbeit steht dar.

4S und die damit verbundene Applikation sind eng miteinander verwoben.

Eine Applikation setzt auf den *4S* auf, aber der *4S* muss seinerseits auf die Applikation abgestimmt sein[1]. So ist der *Depth Join* Algorithmus, welcher in Kapitel 4.2 vorgestellt wird, ohne dass der *4S* eine minimale Tiefe liefert nicht umsetzbar.

3.3 Implementierung des *4S*

Die Aktualität der *4S* Daten hängt stark von der Implementierung des Datenaustauschs zwischen den Peers ab. In einem statischen P2P-System ist kein Austausch nötig um exaktes Wissen zu erlangen. In einem P2P-System das hingegen starken Änderungen unterworfen ist, muss der Datenaustausch oft erfolgen. Es werden im folgenden die drei Mechanismen *Periodic 4S*, *Adaptive 4S* und *Piggyback 4S*, welche einen Datenaustauch ermöglichen, vorgestellt.

3.3.1 Periodischer Datenaustausch (*Periodic 4S*)

Funktionsweise

Eine Möglichkeit, Daten aus den entsprechenden Präfixgruppen zu erhalten, ist durch folgenden einfachen Mechanismus gegeben: Jeder Peer im P2P-System sendet in regelmässigen Zeitintervallen Anfragen an jeweils einen in seinen Präfixgruppen enthaltenen Nachbarn ab. Die Anfrage spezifiziert dabei, welche Daten, für welche Präfixgruppe angefordert werden. Nachbarn, die eine Anfrage erhalten, senden darauf die angeforderten Daten dem Peer zurück, welcher dann die Daten entsprechend speichert. Dieser Vorgang wird in regelmässigen Zeitintervallen von allen Peers und für alle Präfixgruppen wiederholt.

Beispiel: Peer 0011 wünscht die neusten minimalen Tiefen[2] seiner Präfixgruppen. Der Peer sendet dazu vier Anfragen an jeweils einen Nachbarn aus den Präfixgruppen 1, 01, 000 und 0010 ab. Diese senden dann ihrerseits die minimale Tiefe zurück. Periodisch wiederholt der Peer 0011 die Anfrage.

Eigenschaften

Dadurch, dass jeder Peer alle seine Nachbarn ohne irgendwelche Einschränkungen und in regelmässigen Zeitintervallen Δt abfragt, ist ein bestimmter *4S*-Wert in $log_2(n)\Delta t$ Zeiteinheiten allen Peers bekannt. n ist die Anzahl Peers im P2P-System.

Beim *Periodic 4S* ist das Nachrichtenaufkommen erheblich und umso grösser, je kürzer das Zeitintervall ist. Das Nachrichtenaufkommen msg_{4S} für alle Peers im System kann mit der Formel

$$msg_{4S} = \sum_{t=1}^{T/\Delta t} N(t)D(t) = \sum_{t=1}^{T/\Delta t} at\Delta t \log_2(at\Delta t) \tag{3.1}$$

[1] Informationen, die der *4S* liefert, sind dabei natürlich nicht alleine einer bestimmten Applikation vorbehalten, sondern sind für allen Applikationen offen.

[2] Die Tiefe eines Peers bezieht sich auf die Tiefe in einem binären Baum und die minimale Tiefe auf eine Präfixgruppe. In Abbildung 2.1 hat Peer 0011 die Tiefe 3. Die minimale Tiefe der Präfixgruppe 00 ist 2.

berechnet werden. Wobei $N(t) = at\Delta t$ die Peerzahl und $D(t) = \log_2(N) = \log_2(at\Delta t)$ die mittlere Tiefe der Peers zum Zeitpunkt t ist. T ist die totale Simulationszeit, a die mittlere Einfügerate pro Zeiteinheit und Δt das Zeitintervall des Datenaustauschs.

Vergleicht man dies mit dem Nachrichtenaufkommen beim Einfügen von neuen Peers,

$$msg_{Join} = \sum_{t=1}^{T/a} D(t) = \sum_{t=1}^{T/a} \log_2(at) \tag{3.2}$$

sieht man schnell, dass der Aufwand um einiges grösser ist. Bei einem Zeitraum von $T = 50'000$ Zeiteinheiten, einem Zeitintervall von 90 Zeiteinheiten und einer Einfügerate von 0.0111 pro Zeiteinheit ergeben sich die Werte 2'600'000 für den *4S*-Algorithmus und 5'000 für das Einfügen von neuen Peers. Die Belastung des P2P-Systems durch einen periodischen Austausch der Daten ist demzufolge beträchtlich.

3.3.2 Adaptiver Datenaustausch (*Adaptive 4S*)

Funktionsweise

Bei diesem Ansatz werden Daten nur dann ausgetauscht, falls sich die lokalen *4S* Daten eines Peers geändert haben. Im Detail funktioniert der Mechanismus folgendermassen: Jeder Peer überprüft in regelmässigen Zeitintervallen, ob eine Änderung in den Daten der komplementären Präfixgruppen vorliegt. Sobald dies der Fall ist, sendet der Peer die geänderten Daten an den Nachbarn der entsprechenden Präfixgruppe.

Eigenschaften

Nimmt man die minimale Tiefe als die Daten die zwischen den Peers ausgetauscht werden, erwarten wir ein erheblich reduziertes Nachrichtenaufkommen gegenüber dem *Periodic 4S*. Um diese Behauptung zu motivieren, betrachten wir folgendes Beispiel wie es auch in Abbildung 3.1 dargestellt ist: Alle Peers weisen die Tiefe 3 auf, d.h. die minimale Tiefe ist ebenfalls 3. Um die minimale Tiefe der Präfixgruppe 0 zu ändern, muss man 2^3 Peers einfügen. Danach haben alle Peers der Präfixgruppe 0 die Tiefe 4 und somit ist auch die minimale Tiefe 4. Erst nachdem 8 neue Peers eingefügt sind, kommt es zu einem Austausch der *4S* Daten.

3.3.3 Datenaustausch durch Piggybacking (*Piggyback 4S*)

Eine weitere Variante einen Datenaustausch unter den Peers zu ermöglichen und dabei einen möglichst kleinen Nachrichtenaufwand zu erzeugen, ist ein Piggyback Mechanismus. Dabei werden einfach alle *4S* Daten an "normale" Nachrichten (z.B. eine Suchanfrage für ein Objekt) angehängt und werden dann mit diesen mitgesendet. Dadurch findet kein separater *4S* Nachrichtenaustausch statt und das Nachrichtenaufkommen für den *4S* liegt bei Null. Piggybacking kann in Kombination mit den anderen vorgestellten Varianten verwendet werden.

Kapitel 4

Einfügen von Peers

Dieses Kapitel stellt drei Algorithmen für das Einfügen von neuen Peers in ein P2P-System vor. Die Algorithmen *Number Join (NJ)* und *Depth Join (DJ)* greifen dabei auf den Steady-State-Statistics-Service zurück. Der dritte Algorithmus *Random Join (RJ)* schlägt den klassischen Weg anderer P2P-Systeme wie z.B. CAN[6], Cord[10], und Kademlia[3] ein und soll als Vergleich zu den anderen zwei Algorithmen dienen.

4.1 Einfügen über die Anzahl Peers *(Number Join)*

Die Idee hinter dem Einfügen eines Peers mit Hilfe von Information über die Anzahl Peers besteht darin, dass ein neuer Peer in die Regionen mit kleinster Anzahl Peers geleitet wird. Damit eine Entscheidung getroffen werden kann, wohin ein Peer geleitet werden muss, weiss jeder Peer, wie viele Peers es in den jeweiligen Präfixgruppen gibt. Diese Information wird durch den Steady-State-Statistics-Service zur Verfügung gestellt.

4.1.1 Funktionsweise

Anzahl Peers durch *4S*

Um einen neuen Peer in die richtige Präfixgruppe weiterleiten zu können, weiss jeder Peer, wie viele Peers wo im P2P-System vorkommen. Dazu wird ein *4S* verwendet, das die Anzahl Peers berechnet und durch das P2P-System propagiert. *4S* baut für jeden Peer eine Struktur wie sie in Abbildung 4.1 abgebildet ist auf.

Ein Peer[1] p unterteilt das ganze P2P-System in Präfixgruppen. Für jede Präfixgruppe wird abgespeichert, wie viele Peers es in einer Präfixgruppe p_i resp. in seiner komplementären Präfixgruppe $\overline{p_i}$ gibt. Um die Anzahl Peers abzuspeichern, wird die Routingtabelle um diese zwei Einträge erweitert. Die Anzahl Peers einer Präfixgruppe p_i wird dabei mit Hilfe der in den Kapitel 3.3.1, 3.3.2 und 3.3.3 vorgestellten Techniken von den jeweiligen Nachbarn eruiert.

[1] Da die Unterteilung aus der Sicht eines Peers beschrieben wird, verwenden wir hier die Begriffe Peer und *4S* gleichermassen.

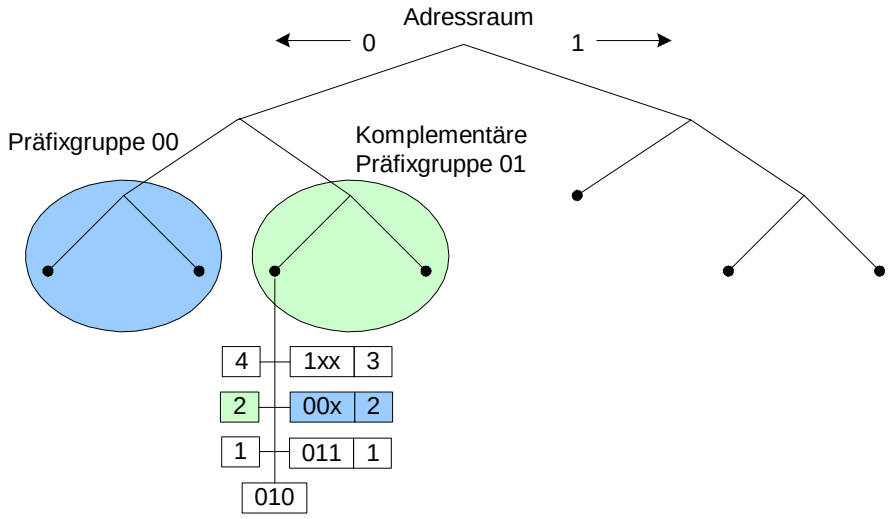

Abbildung 4.1: Der Peer $p = 010$ besitzt die drei Präfixgruppen $p_1 = 1$, $p_2 = 00$ und $p_3 = 011$. Auf der linken Seite der Routingtabelle steht die Anzahl Peers für die komplementäre Präfixgruppe $\overline{p_i}$, rechts neben dem Eintrag der Präfixgruppe die Anzahl Peers der Präfixgruppe p_i selber.

Die Anzahl Peers der komplementären Präfixgruppe $\overline{p_i}$ lässt sich mit folgender Formel berechnen:

$$\overline{n_i} = n_{i+1} + \overline{n_{i+1}}, (i = 1, \ldots, k-1) \tag{4.1}$$

Dabei weisen die Variablen folgende Bedeutung auf:

$\overline{n_{i+1}}$ **:** Anzahl Peers der komplementären Präfixgruppe $\overline{p_{i+1}}$.

n_{i+1} **:** Anzahl Peers der Präfixgruppe p_{i+1}.

Die Formel bedeutet nichts anderes als das die Anzahl Peers einer komplementären Präfixgruppe $\overline{p_i}$ gleich der Summe der nächsten Präfixgruppe p_{i+1} und deren komplementären Präfixgruppe $\overline{p_{i+1}}$ ist. Jede Unterteilung in Präfixgruppen endet des weiteren in einer komplementären Präfixgruppe, die nur noch einen Peer besitzt; dem Peer der die Unterteilung vornimmt. Es gilt deshalb $\overline{n_k} = 1$. Tabelle listet die Werte für den Peer 010 aus dem Beispiel der Abbildung 4.1 auf:

Tiefe i	$\overline{p_i}$	$\overline{n_i}$	p_i	n_i
1	0	$2 + 2 = 4$	1	3
2	01	$1 + 1 = 2$	00	2
3	010	1	011	1

Ablauf des Algorithmus

Anhand der gespeicherten Anzahl Peers ist ein Peer p nun in der Lage einen neuen Peer p_{new} entsprechend zu behandeln. Dabei geschieht im einzelnen folgendes: Der neue Peer p_{new} meldet sich bei einem beliebigen Peer p_A an. Dieser Peer bezeichnet man auch als Bootstrap-Knoten[2]. Die Overlay-ID vom Peer p_{new} ist noch gänzlich undefiniert. Peer p_A vergleicht nun die Anzahl Peers einer Präfixgruppe p_i mit der Anzahl Peers der komplementären Präfixgruppe $\overline{p_i}$, beginnend mit dem kürzesten Präfix der Präfixgruppe p_1 und $\overline{p_1}$. Peer p_A entscheidet nun, was mit dem neuen Peer p_{new} zu geschehen hat und hat dabei drei Möglichkeiten:

1. Anzahl Peers in der Präfixgruppe p_i ist kleiner als $\overline{p_i}$: Die Overlay-ID von p_{new} wird gleich der Präfixgruppe p_i gesetzt. p_A leitet p_{new} an einen Peer p_B aus der Präfixgruppe p_i weiter, d.h. p_B übernimmt die Rolle von p_A.

2. Anzahl Peers in der Präfixgruppe p_i ist grösser oder gleich $\overline{p_i}$ und es gibt noch weitere Präfixgruppen p_{i+1}: In diesem Fall fährt p_A mit dieser nächsten Präfixgruppe p_{i+1} weiter.

3. Anzahl Peers in der Präfixgruppe p_i ist grösser oder gleich $\overline{p_i}$ und es gibt keine weiteren Präfixgruppen p_{i+1}: Dies bedeutet, dass p_{new} nicht mehr weitergeleitet werden kann. p_A wird daher p_{new} ins P2P-System einfügen. Das Einfügen geschieht, indem p_A seine Overlay-ID aufteilt, d.h. die Overlay-ID wird um eine Stelle verlängert und mit einer Null resp. Eins versehen. Ist ein Aufteilen der Overlay-ID nicht möglich, da der Adressraum schon ausgereizt ist, wird p_{new} nicht eingefügt (siehe dazu Erklärung in 4.1.2). Der Algorithmus endet an dieser Stelle.

Ein **Beispiel** soll den Algorithmus illustrieren (Abbildung 4.2): Ein neuer Peer meldet sich bei Peer 010 an. Die Overlay-ID des neuen Peers ist zuerst noch unbekannt. Peer 010 stellt fest, dass die Präfixgruppe 1 weniger Peers als die komplementäre Gruppe 0 hat und leitet den neuen Peer zu Peer 110 innerhalb der Präfixgruppe 1 weiter. Die erste Stelle der Overlay-ID wird auf **1** gesetzt. Damit wird sichergestellt, dass der neue Peer nicht mehr in die Präfixgruppe 0 zurückgeleitet wird. Peer 110 schaut nun seinerseits nach, welche Gruppe die kleinste Anzahl Peers hat. Dabei wird die Präfixgruppe 0 nicht mehr betrachtet, da der neue Peer von dort her kam. Die kleinste Anzahl Peers besitzt die Präfixgruppe 10 mit einem Peer. Eine weitere Stelle der Overlay-ID wird gesetzt und lautet nun **10**. Der neue Peer wird zum Peer 10 weitergeleitet. Peer 10 kennt keine weiteren Präfixgruppen und ist deshalb verantwortlich den neuen Peer ins P2P-System einzufügen. Die Overlay-ID wird nun in zwei neue Overlay-IDs 100 und 101 aufgeteilt, je einem Peer zugewiesen und der neue Peer ist Teil des P2P-Systems. Der Algorithmus ist damit beendet.

[2]Wie Peer p_{new} zu einem Bootstrap-Knoten gelangt, wird hier nicht weiter erklärt. In der Simulation des *Number Join* Algorithmus wird ein Bootstrap-Knoten zufällig aus den im P2P-System befindlichen Peers gewählt.

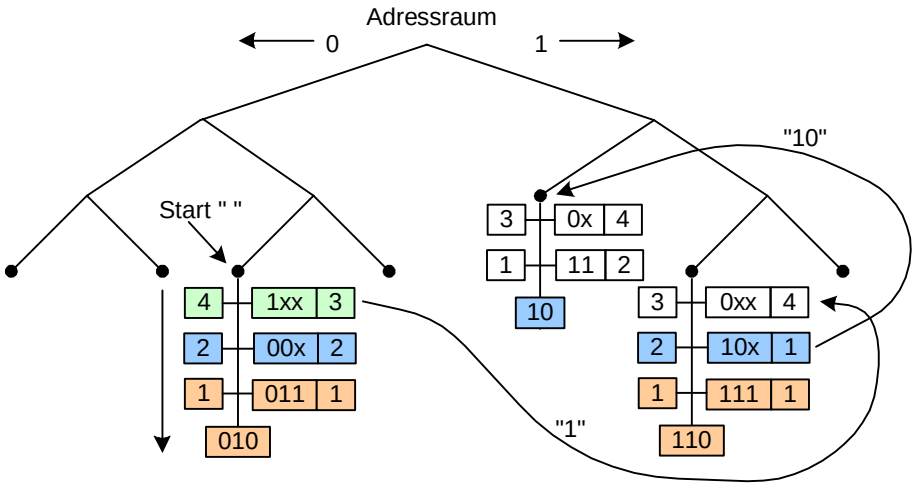

Abbildung 4.2: Ausgehend von Peer 010 wird ein neuer Peer zu Peer 10 geroutet. Bei jedem Schritt wird ein Teil der Overlay-ID des neuen Peers festgelegt. Peers vergleichen deshalb nur die farblich gekennzeichneten Präfixgruppen. Pfeil links der Routingtabelle des Peers 010 kennzeichnet die Reihenfolge der Vergleiche.

4.1.2 Eigenschaften

Skalierbarkeit

Der *Number Join* Algorithmus fügt einen neuen Peer bei einer Gesamtzahl von n Peers in maximal $log_2(n)$ Schritten ein, d.h. der Algorithmus skaliert logarithmisch mit der Anzahl der Peers. Dies gilt, da bei jedem Routen die Overlay-ID um mindestens eine Stelle verlängert wird und die maximale Länge einer Overlay-ID bei n Peers $log_2(n)$ beträgt.

Verhinderung von Zyklen

Beim Routen eines neuen Peers von einem Peer zu einem anderen, wird die Overlay-ID fortlaufend festgelegt. Ein Peer kann dadurch feststellen, aus welcher Präfixgruppe ein Peer zugeschickt wurde und ignoriert beim Vergleich der Anzahl Peers diese Präfixgruppen. Zyklenbildung und ein Zurückleiten wird auf diese Weise verhindert.

Eindeutigkeit der Overlay-ID

Overlay-IDs werden immer eindeutig vergeben. Jeder neue Peer wird durch einen Peer eingefügt, dessen Overlay-ID eindeutig ist. Durch das Aufteilen in zwei neue Overlay-IDs, die um eine Stelle länger sind als die ursprüngliche Overlay-ID und sich nur in der letzten Stelle unterscheiden, entstehen wiederum zwei eindeutige Overlay-IDs. Geht man von einem P2P-System bestehend aus einem Peer mit der leeren Overlay-ID[3] aus, ist eine Eindeutigkeit der Overlay-ID

[3]Falls nur ein Peers existiert ist dieser für alle Objekte verantwortlich und deshalb ist noch keine Overlay-ID nötig. Eine Overlay-ID wird erst bei zwei Peers zugeordnet.

zu jedem Zeitpunkt gegeben. Auch im Fall, dass zwei neue Peers "gleichzeitig" bei einem Peer eintreffen, kann es nicht zu einer doppelten Overlay-ID Vergabe kommen. Beide Peers werden ja von demselben Peer eingefügt. Dieser behandelt "gleichzeitig" eingetroffene Peers nacheinander. Eine Eindeutigkeit ist erst dann nicht mehr gegeben, falls es zu einer Partitionierung des P2P-System kommt.

Einfügen durch Peers mit maximaler Overlay-ID-Länge

Jede Overlay-ID weist eine bestimmte Länge auf. Die minimale Länge kann dabei nie kürzer wie eins sein, aber auch nicht länger wie eine durchs System vorgegebene Konstante. In der Simulation wird z.B. eine maximal Länge von 24 Bits verwendet. Wird ein neuer Peer zu einem Peer dessen Overlay-ID die maximal Länge aufweist geroutet, kann der Peer nicht ins P2P-System aufgenommen werden, da eine Aufteilung der Overlay-ID nicht mehr möglich ist. Die Wahrscheinlichkeit ist bei genügend grosser Tiefe des Adressraums und mit einer Einschränkung der Anzahl Peers auf wenige Peers aber klein. In einem echten P2P-System ist aber wünschenswert, durch geeignete Mechanismen dies gänzlich auszuschliessen. Als Möglichkeiten kann man sich z.B. vorstellen, dass bei einem Fehlschlag, der Algorithmus mit einem anderen Bootstrap-Knoten wiederholt oder Backtrapping beim Routen durchgeführt wird. Bei der Simulation wird darauf verzichtet.

Unbalancierte P2P-Systeme

Der Algorithmus stellt sicher, dass ein neuer Peer immer in eine Präfixgruppe mit weniger Peers geroutet wird. Dadurch hat eine Präfixgruppe und sein komplementäres Gegenstück immer etwa gleich viele Peers, was zu einem balancierten[4] P2P-System führt. Bei genauerem Hinsehen stellt man aber fest, dass gleich viele Peers in einer Präfixgruppe und der komplementären Präfixgruppe nicht automatisch ein balanciertes P2P-System bedeutet. Es kann durchaus vorkommen, das zwar gleich viele Peers da sind, aber die Verteilung der Peers gänzlich anders aussieht, d.h. die minimale und maximale Tiefe im Baum ist unterschiedlich. Man kann sich folgendes vorstellen (Abbildung 4.3):

Die Präfixgruppe 0 besitzt weniger Peers als die Präfixgruppe 1, aber seine minimale Tiefe ist grösser als die von 1. Damit es zu einem balancierten P2P-System kommt, müssen Peers in die Präfixgruppe 1 eingefügt werden. Meldet sich nun ein Peer bei einem beliebigen Peer an, wird dieser aber in die Präfixgruppe 0 eingefügt, da diese über weniger Peers verfügt. Peers werden nun solange in die Präfixgruppe 0 eingefügt, bis diese über gleich viele Peers wie Präfixgruppe 1 verfügt.

[4]In einem balancierten P2P-System besitzt jeder Peer gleich viele Schlüssel/Objekt-Paare, in einem unbalancierten hingegen nicht. Ziel ist es, eine gleiche Verteilung der Schlüssel/Objekt-Paare in einen P2P-System zu haben.

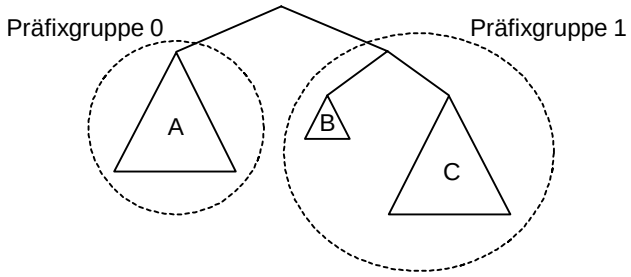

Abbildung 4.3: Teilbaum A besitzt weniger Peers als Teilbäume B und C zusammen. Der *Number Join* Algorithmus fügt deshalb neue Peers bei A ein. Das P2P-System bleibt weiterhin unbalanciert.

4.2 Einfügen über die minimale Tiefe *(Depth Join)*

Die Idee beim *Depth Join*, kurz *DJ*, Algorithmus ist im Prinzip die gleiche wie beim *Number Join* Algorithmus. Ein neuer Peer meldet sich wiederum bei einem Bootstrap-Knoten an und wird dann zu einem Peer weiter geleitet, welcher den neuen Peer ins P2P-System einfügt. Der Unterschied liegt in der Entscheidung, wohin ein Peer geleitet wird. Beim *Number Join* Algorithmus ist es die Anzahl Peers einer Präfixgruppe. Beim *Depth Join* Algorithmus ist es hingegen die minimale Tiefe einer Präfixgruppe. Der Algorithmus geht das Problem mit unbalancierten P2P-Systemen an und stellt eine verbesserte Version des ursprünglichen Algorithmus dar.

4.2.1 Funktionsweise

Minimale Tiefe durch *4S*

Jeder Peer im P2P-System baut wiederum eine Struktur, wie sie unter 4.1.1 beschrieben wird, auf. Anstelle der Anzahl Peers der Präfixgruppen p_i und der komplementären Präfixgruppen $\overline{p_i}$, werden aber die minimalen Tiefen der Präfixgruppen abgespeichert. Die minimale Tiefe hat dabei folgende Bedeutung: Betrachtet man alle Peers in einer Präfixgruppe, wählt man denjenigen Peer aus dieser Gruppe aus, dessen Tiefe die kleinste ist. Die Tiefe des gewählten Peers, gibt die minimale Tiefe d_i der Präfixgruppe p_i. Die minimale Tiefe $\overline{d_i}$ einer komplementären Präfixgruppe $\overline{p_i}$ kann über folgende Formel ermittelt werden:

$$\overline{d_i} = min(d_{i+1}, \overline{d_{i+1}}), (i = 1, \ldots, k-1) \tag{4.2}$$

Wobei $\overline{d_{i+1}}$ die minimale Tiefe der komplementären Präfixgruppe $\overline{p_{i+1}}$ und d_{i+1}, die minimale Tiefe der Präfixgruppe p_{i+1} darstellt. Die minimale Tiefe d_i einer Präfixgruppe p_i wird wiederum durch Austausch der *4S*-Daten mit dem Nachbarn ermittelt. Folgende Tabelle listet die minimalen Tiefen für das Beispiel der Abbildung 4.1 auf:

Tiefe i	$\overline{p_i}$	$\overline{d_i}$	p_i	d_i
1	0	$min(2,3) = 2$	1	3
2	01	$min(3,3) = 3$	00	2
3	010	3	011	3

Ablauf

Der Ablauf des Algorithmus entspricht grundsätzlich dem des *Number Join* Algorithmus. Der einzige Unterschied liegt beim Vergleich der Präfixgruppen. Anstelle der Anzahl Peers findet ein Vergleich der minimalen Tiefen statt. Wir verzichten deshalb auf eine genauere Beschreibung des Algorithmus.

4.2.2 Eigenschaften

Die Eigenschaften des Algorithmus entsprechen denjenigen des *Number Join* Algorithmus. Wir verweisen deshalb auf das entsprechende Kapitel 4.1.2.

Eine Ausnahme bildet lediglich das Problem bei unbalancierten P2P-Systemen. Dadurch das beim *Depth Join* Algorithmus die Grundlage nicht die Anzahl Peers, sondern die minimale Tiefe ist, wird durch das Einfügen von neuen Peers ein unbalanciertes P2P-System in ein balanciertes überführt. Der *Depth Join* stellt also eine Verbesserung des *Number Join* Algorithmus dar.

4.3 Zufälliges Einfügen *(Random Join)*

P2P-Systeme wie CAN[6], Cord[10] oder Kademlia[3] fügen neue Peers über eine zufällig gewählte Overlay-ID in ein P2P-System ein. Das Vorgehen ist dabei immer das gleiche. Peers erhalten eine eindeutige Overlay-ID und werden anhand der Overlay-ID durch das P2P-System geroutet (analog dem Routen bei einer Suchanfrage) und schlussendlich eingefügt. Die Overlay-IDs werden dabei uniform über den Adressraum generiert. Der hier vorgestellte Algorithmus bildet diese Vorgehensweise ab. Der *Random Join*, kurz *RJ*, Algorithmus dient als Vergleich zu den Algorithmen *Number Join* und *Depth Join*.

4.3.1 Funktionsweise

Ein neuer Peer p_{new} generiert in einem ersten Schritt eine Overlay-ID durch das Werfen einer fairen Münze. Bei jedem Wurf wird eine Stelle der Overlay-ID festgelegt, d.h. es wird eine Null oder Eins geschrieben. Dieser Vorgang wird solange wiederholt, bis die Overlay-ID die maximal mögliche Länge einer Overlay-ID hat. Die Overlay-ID ist vorerst nur provisorisch. Wie viele Bits der Overlay-ID schlussendlich die endgültige Overlay-ID ausmachen, wird fortlaufend beim Routen festgelegt. Danach meldet sich p_{new} bei einem beliebigen Bootstrap-Knoten p_A an. p_A entscheidet nun anhand der Overlay-ID, was mit p_{new} geschieht. Dabei hat p_A zwei Möglichkeiten:

1. p_A verfügt über eine Präfixgruppe, die näher bei der Overlay-ID des neuen Peers p_{new} liegt als seine eigene. p_A fixiert eine Stelle der Overlay-ID

und leitet p_{new} zum Peer p_B innerhalb der entsprechenden Präfixgruppe weiter.

2. p_A verfügt über keine Präfixgruppe, die näher bei der Overlay-ID liegt als seine eigene. p_A ist zuständig für das Einfügen des Peers p_{new}. Das Einfügen geschieht, indem p_A seine Overlay-ID um eine Stelle erweitert und mit einer Null oder Eins versehen. Weist die Overlay-ID von p_{new} an dieser Stelle eine Null auf, wird eine Eins zugewiesen und umgekehrt. p_A fixiert eine weitere und letzte Stelle der Overlay-ID von p_{new}. Ist ein Aufteilen der Overlay-ID nicht möglich, da der Adressraum schon ausgereizt ist, wird der neue Peer nicht eingefügt. Der Algorithmus endet hier.

4.3.2 Eigenschaften

Kein Nachrichten-Overhead durch *4S*

Der *Random Join* Algorithmus ist auf keinen Steady-State-Statistics-Service angewiesen, d.h. die Belastung des Systems durch den *4S* fällt weg. Das ist ein grosser Vorteil des *Random Join* Algorithmus gegenüber den anderen vorgestellten Algorithmen. Jeder Algorithmus der über einen Steady-State-Statistics-Service operiert, ist nur dann bezüglich des Nachrichtenaufkommens kompetitiv, falls das Nachrichtenaufkommen des *4S* im Bereich des Nachrichtenaufkommen für das Einfügen von neuen Peers liegt.

Uniformität der Overlay-ID

Die durch diesen Algorithmus erzeugte uniforme Verteilung der Peers ist nur bedingt richtig. Bei genauerem Hinsehen stellt man fest, dass es Bereiche mit mehr oder weniger Peers gibt, d.h. die uniforme Verteilung ist nicht ganz gegeben. Das kann mit folgendem Vergleich gezeigt werden: n Bälle werden in der Reihe nach zufällig auf m Behälter aufgeteilt. Die Wahrscheinlichkeit, dass ein Behälter leer ist, beträgt

$$(1 - \frac{1}{m})^n \approx e^{-n/m} \tag{4.3}$$

da die Wahrscheinlichkeit einen Behälter zu treffen $1/m$ beträgt und dies genau n mal zu geschehen hat. Dieses Problem ist unter dem Namen “Ball into Bins”[5] bekannt. Die gleiche Wahrscheinlichkeit tritt nun bei der Erzeugung von Overlay-IDs ein. Ein P2P-System, welches zufällig Peers einfügt, weist mit hoher Wahrscheinlichkeit Bereiche auf, die um den Faktor $\Theta(\log(n))$ mehr Schlüssel/Objekt-Paare als der Durchschnitt haben.

Kapitel 5

Implementierung der Simulation

5.1 Allgemeines

Dieses Kapitel zeigt, wie die Simulation aufgebaut ist. Es wird auf die Funktion der verschiedenen Klassen und Module eingegangen und der Zusammenhang zwischen den einzelnen Elementen erklärt. Die ganze Implementierung ist modular gegliedert, sodass ein Ausbau oder auch ein Umbau einfach zu bewerkstelligen ist. Die Simulation wurde in Java unter JDK 1.4.1 und der Bibliothek JDSL [9], welche für die "PriorityQueue" verwendet wird, implementiert.

5.2 Übersicht

Die ganze Simulation gliedert sich in drei Hauptmodule (Abbildung 5.1); die Simulationsumgebung, die Algorithmen und das Hilfsmodul. Die **Simulationsumgebung** besteht aus den zwei Modulen *P2P-System* und *Event-Driven-Simulation* und zusammen stellen sie eine Umgebung (Framework) zur Verfügung in der die eigentlichen Algorithmen umgesetzt werden. Mit Hilfe des Moduls **Algorithmen** werden die zu testenden Algorithmen umgesetzt. Das **Hilfsmodul** erweitert die Simulationsumgebung um Funktionalitäten, wie Netzwerkgenerierung und Auswertung der Simulation.

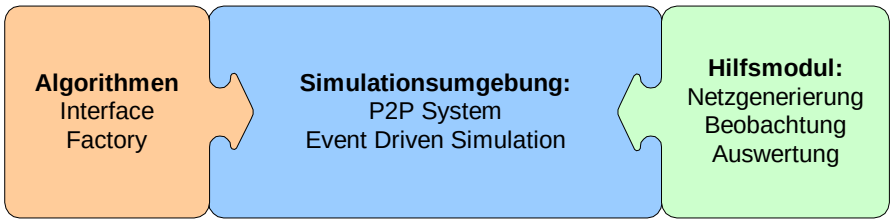

Abbildung 5.1: Unterteilung des Frameworks in die Hauptmodule Simulationsumgebung, Algorithmen und Hilfsmodul.

5.3 Simulationsumgebung

5.3.1 Übersicht

Die Simulationsumgebung stellt ein Framework für die Implementierung und das Testen von P2P Algorithmen zur Verfügung. Es ist zwar auf die Bedürfnisse dieser Diplomarbeit ausgerichtet, kann aber auch für andere Simulationen im P2P-Bereich genutzt oder leicht daran angepasst werden.

Die Simulationsumgebung besteht aus den zwei Modulen *P2P-System* und *Event-Driven-Simulation* (Abbildung 5.2). *P2P-System* stellt das eigentliche P2P-System, auf dem die Algorithmen operieren, dar. Das Modul *Event-Driven-Simulation* steuert den zeitlichen Ablauf der Simulation. Alle Zustandsänderungen des P2P-Systems werden über dieses Modul vorgenommen, indem eine entsprechende Methode eines Algorithmus aufgerufen wird.

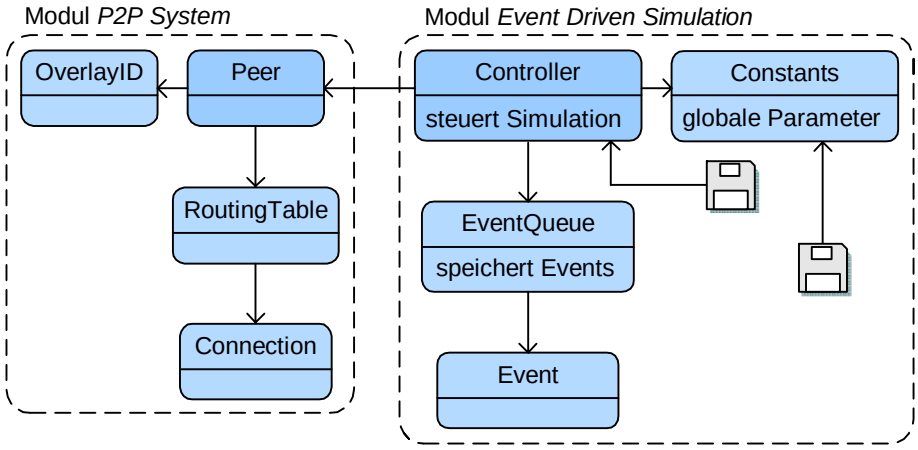

Abbildung 5.2: Grafik bildet alle Klassen des Hauptmoduls Simulationsumgebung ab und stellt sie in ihrem Kontext dar. Das Diskettensymbol symbolisiert Lesezugriffe auf eine Datei.

5.3.2 Modul *Event-Driven-Simulation*

Dieses Modul stellt das Herzstück der ganzen Simulation dar. Alle im System befindlichen Peers werden hier verwaltet. Parameter, die für einen Simulationslauf benötigt werden, werden hier aus einer Datei von der Festplatte ausgelesen und entsprechend gesetzt. Des weiteren stellt das Modul eine Infrastruktur für den zeitlichen Ablauf der Simulation zur Verfügung. Die Infrastruktur verwendet dabei das Modell einer Event-Driven-Simulation.

Funktionsweise

Die Funktionsweise einer Event-Driven-Simulation, wird hier anhand eines Beispiels[4] aufgezeigt und erklärt. Relevante Begriffe sind zu Beginn hervorgehoben. Eine Erklärung findet dann im Verlauf des Beispiels statt. Das Beispiel simuliert einen Flughafen mit Starts und Landungen von Flugzeugen auf einer

Landebahn. Möglich **Ereignisse** sind dabei: *Landing-requst*, *Landing-complete*, *Take-off-request* und *Take-off-complete*. Die Landebahn kann den Zustand *Landing*, *Take-off* und *Idle* aufweisen. *Take-off* und *Landing* benötigen dabei jeweils 3 Zeiteinheiten bis sie abgeschlossen sind.

1. Der Ausgangzustand der Simulation ist wie folgt:
   - Landebahn ist frei(*Idle*), d.h. es finden keine Starts und Landungen statt.
   - Die globale **Simulationszeit** ist auf Null gesetzt.
   - Das erste *Landing-request* Ereignis findet zur Zeit 3 statt und das erste *Take-off-request* Ereignis zur Zeit 5. Diese zwei Ereignisse werden quasi von Hand in eine **Priorityqueue** eingefügt, damit der ganze Prozess überhaupt ins Rollen kommt. Dieser Vorgang wird auch Bootstrapping[1] einer Simulation genannt. Zu Beginn der Simulation befinden sich also zwei Ereignisse in der Priorityqueue.
2. Lies das erste Ereignis (*Landing-request*) aus der Priorityqueue und verarbeite dieses Ereignis. Verarbeiten heisst, es müssen Aktionen wie folgende durchgeführt werden:
   - Sperren der Landebahn für alle anderen Flugzeuge (*Landing*).
   - Generieren eines neuen *Landing-request* Ereignisses. Bei der Bearbeitung eines Ereignisses werden evtl. weitere Ereignisse ausgelöst. Beim *Landing-request* ist es sinnvoll, dass weitere *Landing-request* generiert werden. Damit wird die Simulation weiter angetrieben und es kommt nicht zu einem Stillstand. Zu welchem Zeitpunkt dieses Ereignis in die Priorityqueue eingefügt wird, kann auf verschiedene Arten modelliert werden. Mögliche Modelle sind z.B. feste Zeitintervalle, rein zufällig oder auch über ein Poisson-Modell. In unserem Beispiel findet der nächste *Landing-request* zur Zeit 4 statt, d.h. das Ereignis wird in der Priorityqueue vor dem Ereignis *Take-off-request* eingefügt.
   - Generieren eines *Landing-complete* Ereignisses und einfügen des Ereignisses in die Priorityqueue.
   - Erstellen eines "Schnappschusses" der Simulation, d.h. sammle und speichere alle Information, die es zur Auswertung nach Abschluss der Simulation braucht. Hier z.B. Inkrementierung eines Zähler für die Anzahl Landungen, Anzahl Passagiere, Flugnummer, usw.
3. Lies das nächste Ereignis aus der Priorityqueue. Dabei handelt es sich um einen zweiten *Landing-request* zum Zeitpunkt 4. Leider ist die Landebahn gesperrt, da schon ein anderes Flugzeug am Landen ist. Ereignisse, die nicht sofort verarbeitet werden können, werden nicht in die Priorityqueue zurück geschrieben, sondern werden in einer separaten Warteschlange abgelegt. In diesem Beispiel ist es eine Warteschlange die von der Landebahn

[1] Nicht zu verwechseln mit einem Bootstrap-Knoten beim Einfügen von neuen Peers.

verwaltet wird. Sobald die Landebahn den Zustand *Idle* (freie Landebahn) hat, wird ein Ereignis aus dieser Warteschlange gelesen und verarbeitet.

4. Lies das nächste Ereignis aus der Priorityqueue. Dabei handelt es sich um einen *Take-off-request* zum Zeitpunkt 5. Die Landbahn ist aber immer noch gesperrt, d.h. auch dieses Ereignis kommt in die Warteschlange der Landebahn.

5. Lies das nächste Ereignis aus der Priorityqueue. Dabei handelt es sich um einen *Landing-complete* zum Zeitpunkt 6. Abbildung 5.3 zeigt die Simulation zu diesem Zeitpunkt. Dieses Ereignis löst folgende Aktionen aus:

   - Die Landebahn wird freigegeben (*Idle*).
   - Es wird wieder ein Schnappschuss der Simulation erstellt.
   - Prüfe ob sich noch unbearbeitete Ereignisse in der Warteschlange der Landebahn befinden. Falls das der Fall ist, verarbeite erstes Ereignis in der Warteschlange. In diesem Beispiel ist es das Ereignis *Landing-request*, welches eigentlich zur Zeit 4 vorgesehen war.

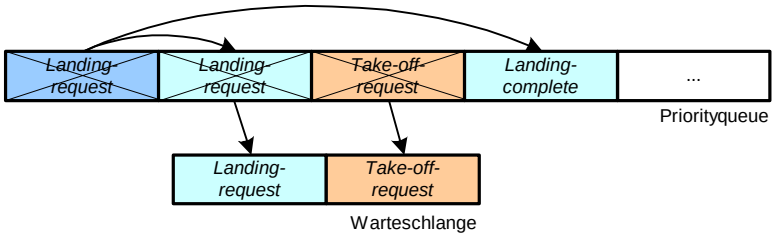

Abbildung 5.3: PriorityQueue und Warteschlange zum Zeitpunkt 6. Die ersten drei Ereignisse sind schon aus der Priorityqueue entfernt. Der erste *Landing-request* ist vollständig behandelt, der zweite *Landing-request* und der *Take-off-request* sind hingegen in die Warteschlange der Landebahn geschrieben worden. Diese werden bearbeitet, sobald die Landebahn frei ist.

6. Der ganze Prozess läuft so lange bis eine festgelegte Bedingung erfüllt ist. Das kann z.B eine festgelegte Simulationszeit sein, aber auch die Anzahl erfolgter Landungen oder Starts von Flugzeugen.

7. Jedes Ereignis ist mit einem Zeitstempel versehen. Die Zeit stellt dabei eine absolute Zeit dar, d.h. ist die Zeit die vom Simulationsbeginn bis zum jetzigen Ereignis vergangen ist. Sobald ein Ereignis aus der Priorityqueue ausgelesen wird, wird eine interne Simulationszeit nachgeführt, dei Simulationszeit übernimmt die Zeit, die im Ereignis gespeichert ist. Die Simulationszeit läuft dabei nicht kontinuierlich ab, sondern wird immer gleich der Zeit des letzten Ereignisses gleichgesetzt. Das bedeutet, das in der Simulation nicht geschaut wird, ob zu einen bestimmten Zeitpunkt ein Ereignis vorliegt, sondern es wird einfach das nächste Ereignis ausgelesen und danach die Simulationszeit gesetzt. Der Name Event-Driven (Ereignisgesteuert) Simulation leitet sich direkt aus dieser Tatsache ab.

Klassenbeschreibung

Bei den Beschreibungen der Klassen wird nur auf die wesentlichsten Aspekte eingegangen. Eine detaillierte Beschreibung alle Klassen findet sich im Javadoc der Diplomarbeit.

Event stellt ein Ereignis in der Simulation dar. Jeder `Event` ist mit einem Zeitstempel und der Art des Ereignisses (`ActionID`) versehen. Anhand der `ActionID` ruft der `Controller` einen entsprechenden Algorithmus auf. Des weiteren wird im `Event` festgehalten welcher Peer das Ereignis generiert hat (`SourceID`) und welcher Peer es behandeln muss (`DestID`).

EventQueue speichert alle noch nicht behandelten Ereignisse der Simulation. Bei der `EventQueue` handelt es sich, wie der Name schon sagt, um eine Prioritäten-Warteschlange. Ein Ereignis wird anhand seines Zeitstempels in die Schlange eingefügt, d.h. Ereignisse mit einer kleineren Simulationszeit werden durch den Controller zuerst behandelt als solche mit einem grossen Zeitstempel. Zur Realisierung dieser Schlange wurde die Java Data Structure Libary JDSL[9] verwendet.

Controller ist der Startpunkt für die Simulation und steuert alle für die Simulation relevanten Vorgänge. Die Verwaltung aller im System befindlichen Peers geschieht im `Controller`. Der `Controller` liest `Event`s aus der `EventQueue` und ruft eine entsprechende Methode zur Behandlung des Ereignisses auf, welches zu einer Zustandsänderung des P2P-Systems führt und die Simulation durch Generierung neuer Ereignisse weiter treibt. Der `Controller` führt auch die Simulationszeit nach und bricht die Simulation nach einer vorgegebenen Simulationszeit ab. Alle Parameter der Simulation liest der `Controller` von einer externen Parameterdatei aus. Beim Aufruf des `Controller`s wird als einziges Argument der Dateiname benötigt. Anhang B.1 zeigt den Aufbau der Datei und erklärt alle Parameter.

Constants hält alle systemweiten Parameter fest. Um eine gewisse Flexibilität bei den Parametern zu erreichen, werden gewisse Parameter wiederum aus der Parameterdatei ausgelesen.

5.3.3 Modul *P2P-System*

Das Modul *P2P-System* wiederspiegelt das zugrunde liegende P2P-System, wie es unter Kapitel 2 vorgestellt wurde. Dabei findet eine direkte Abbildung des Modells auf entsprechende Klassen statt, z.B. finden sich Peers in der Klasse `Peer` wieder. Das Modul *P2P-System* speichert dabei: (i) die aktuelle Topologie des P2P-Systems, (ii) den aktuellen Zustand der Peers, (iii) alle für den Steady-State-Statistics-Service benötigten Informationen, wie z.B. die minimale Tiefe einer Präfixgruppe.

Klassenbeschreibung

Peer repräsentiert einen Peer im P2P-System. Diese Klasse stellt das Herzstück des *P2P-Systems* dar und verschmilzt die restlichen Klassen zu einer Einheit. Jede `Peer`-Instanz hat eine eindeutige Peer-ID. In einem echten P2P-System entspricht das der IP-Adresse. Ein Peer wird über die Peer-ID via Controller angesprochen und manipuliert. Jede `Peer`-Instanz hält den Zustand des Peers fest, wie z.B. Peer ist Teil des P2P-Systems oder Peer wird im Moment eingefügt. Jede `Peer`-Instanz hat des weiteren Referenzen zu einer Routingtabelle und zur Overlay-ID[2] des Peers.

OverlayID stellt die Overlay-ID eines Peers dar und verfügt darüber hinaus folgende Funktionalitäten: Lesen resp. Schreiben der `OverlayID`, Vergleichsoperationen zwischen `OverlayID`s, Manipulation der `OverlayID` wie z.B. das Aufteilen einer `OverlayID` in zwei neue `OverlayID`s.

RoutingTable hat zwei Funktionen inne. Zu einem speichert die `RoutingTable` Verbindungen zu Nachbar-Peers und zum anderen alle Informationen bezüglich des Steady-State-Statistics-Service. Jede Präfixgruppe eines Peers besitzt einen `Hashmap`, indem eine beliebige Zahl von Verbindungen gespeichert werden kann. Eine `Routingtable` besitzt insgesamt $\log_2(n)$ `Hashmap`s, was der Anzahl der Präfixgruppen eines Peers entspricht. `RoutingTable` speichert des weiteren alle Informationen, die es für den *4S* braucht, d.h. minimale Tiefen und die Anzahl Peers werden hier gespeichert.

Connection ist die eigentliche Verbindung zu einem Nachbar-Peer. `Connection` speichert zusätzlich alle Informationen bezüglich des Steady-State-Statistics-Service, die direkt vom zugehörigen Nachbar-Peer der `Connection` abrufbar sind. Der Unterschied zur `Routingtable` ist folgender: Die `Routingtable` speichert Werte die für eine ganze Präfixgruppe Gültigkeit haben. Die Werte des `Connection` sind hingegen nur für einen Peer innerhalb der Präfixgruppe gültig. D.h. Werte in der `RoutingTable` sind gemittelte Werte aller in einer Präfixgruppe enthaltenen Verbindungen.

5.4 Implementierung von Algorithmen

5.4.1 Überblick

Dieses Kapitel beschreibt, wie Algorithmen in die Simulationsumgebung integriert und wie Algorithmen mit Hilfe des Event-Driven-Simulationsmodells umgesetzt werden. Die Klasse `AlgorithmFactory` und das Interface `Algorithm` dient dabei als Schnittstelle zur Simulationsumgebung und ermöglicht die Integration eines Algorithmus in diese. In konkreten Realisierungen des Interfaces `Algorithm` werden die eigentlichen Algorithmen implementiert. Dabei gibt es

[2]Die Peer-ID wird bei der Instanzierung des `Peer` erzeugt und ist nicht veränderbar. Die `Overlay-ID` wird hingegen durch einen Algorithmus zugewiesen und ist veränderbar.

drei Arten von konkreten Realisierungen[3]. In den `JoinAlgorithm` werden Algorithmen für das Einfügen von Peers implementiert. Die `4SAlgorithm` bilden den *4S* Mechanismus ab. Die `TriggerAlgorithm` werden zum Starten der Algorithmen verwendet. Abbildung 5.4 stellt diese Klassen dar.

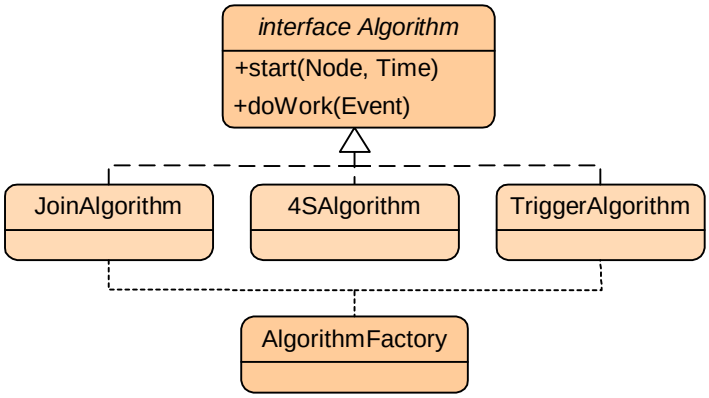

Abbildung 5.4: Join-, *4S*- und und Trigger-Algorithmen implementieren jeweils das Interface `Algorithm`. Die `AlgorithmFactory` liefert eine Instanz des Typs `Algorithm`.

5.4.2 Integration in die Simulationsumgebung

Um eine Integration in die Simulationsumgebung zu erreichen, wurde einerseits das Interface `Algorithm` und anderseits die Fabrik `AlgorithmFactory`, die konkrete Realisierungen des Interfaces liefert, erstellt.

Das Interface `Algorithm` bietet die zwei Methoden `start(Peer, double)` und `doWork(Event)` an. `start(Peer, double)` startet den Algorithmus zu einem vorgegeben Zeitpunkt (`double`) für den durch `Peer` bestimmten Peer. `doWork(Event)` nimmt ein Ereignis entgegen und führt entsprechende Aktionen zur Behandlung des Ereignisses durch. Die Aufgabe des `Controller`s besteht nun einerseits darin, den Algorithmus zu starten und anderseits ein Ereignis einem Algorithmus zuzuordnen und die entsprechende `doWork`-Methode aufzurufen. Die Zuordnung eines Ereignisses zu einem Algorithmus geschieht dabei über die Art des Ereignisses, d.h. über die `ActionID` der Klasse `Event`. Diese Zuordnung ist deshalb nötig, da der `Controller` für jede Simulation jeweils einen `Join-`, `4S-` und `TriggerAlgorithm` erzeugt. Jede Realisierung des Interface `Algorithm` besitzt dabei eine eigene `doWork`-Methode.

Konkrete Realisierungen des Interface `Algorithm` erzeugt der `Controller` über die Methode `getAlgorithm(String)` der Klasse `AlgorithmFactory`. Diese Methode nimmt als Parameter den Factory-Namen eines Algorithmus entgegen. Wie man eine Liste aller zur Zeit bestehender Factory-Namen erhält, wird im Anhang A.2 erklärt.

[3]Diese Aufteilung ist nicht zwingend durch das Interface gegeben und wurde ausschliesslich zur besseren Lesbarkeit des Programms eingeführt

5.4.3 Umsetzung einesJoin-Algorithmus

In der konkreten Realisierung des Interfaces `Algorithm` wird die Logik eines Algorithmus in Form von Methoden und Variablen abgebildet. Jeder Algorithmus muss entsprechend dem Event-Driven-Simulationsmodells umgesetzt und implementiert werden. Wie man nun einen Algorithmus mit diesem Ansatz umsetzt, wird im folgenden anhand eines generischen Join-Algorithmus erläutert.

Jeder Algorithmus wird in einem ersten Schritt in Ereignisse unterteilt. Ein Ereignis liegt dann vor, wenn folgende Bedingung gilt:

- Ein Peer sendet Daten zu einem anderen Peer.
- Ein Peer muss nach Ablauf einer gewissen Zeit, eine Aktion, die ihn selber betrifft, durchführen.

Ein Ereignis findet also dann statt, falls eine Veränderung des Simulationszustandes erfolgen muss, aber eine örtliche oder zeitliche Distanz dazwischen liegt. Dabei gilt immer, dass eine Zustandsänderung eines Peers ausschliesslich durch den Peer selber durchzuführen ist.

Im Beispiel finden wir folgende Ereignisse: ROUTETO, QUERY, NOSPACE, INSERT und TIMEOUT. Abbildung 5.5 zeigt wie diese Ereignisse zusammenhängen.

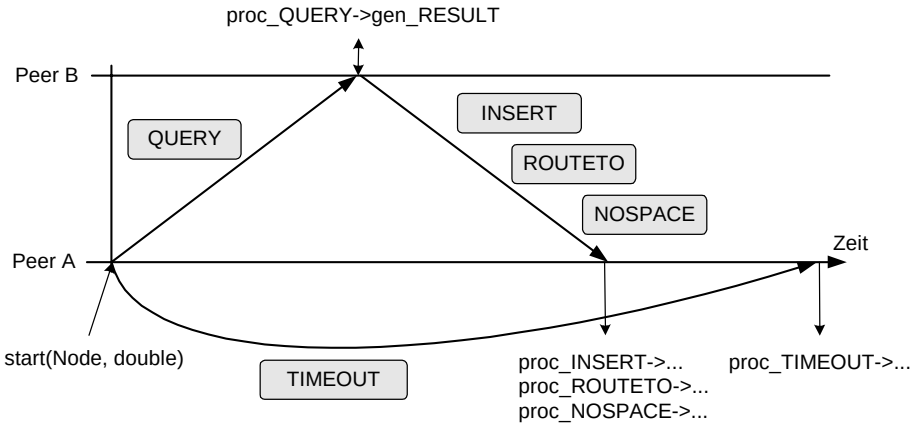

Abbildung 5.5: Ereignisse und zeitlicher Ablauf eines generischen Join-Algorithmus. `gen_X (X steht z.B. für RESULT)` sind Methoden, die ein Ereignis auslösen. Durch die Methoden `proc_X` werden Ereignisse behandelt.

Das Herauskristallisieren der Ereignisse aus dem Algorithmus stellt die grösste Hürde bei der Implementierung eines Algorithmus dar. Jedem Ereignis wird nun ein Methodenpaar zugeordnet. Eine Methode generiert das Ereignis und die andere behandelt es.

Generierende Methode: In der Methode, die ein Ereignis generiert ,wird festgelegt, welcher Peer das Ereignis auslöst (`SourceID`), welcher Peer (`DestID`) es zu welchem Zeitpunkt (`EventTime`) behandeln und welche Aktion (`ActionID`) dabei durchgeführt werden muss. Diese Werte werden im Konstruktor der Klasse `Event` festgelegt. Alle weiteren Werte, die zur Behandlung nötigt sind, wer-

den mit der Methode `addData(Key, Object)` einem Ereignis angehängt. Mit `getData(Key)` kann dieser Wert dann wieder ausgelesen werden. Diese Methode erlauben es, beliebige Objekte abzuspeichern und auszulesen. Das Ereignis wird zum Schluss über `Controller.getEvents().addEvent(Event)` in die `EventQueue` eingefügt. Startet in unserem Beispiel ein neuer Peer eine Anfrage bei einem bestehenden Peer, werden gleich zwei Ereignisse generiert und zwar ein QUERY-Ereignis für die Anfrage und ein TIMEOUT-Ereignis im Falle, dass ein Peer nicht innerhalb einer bestimmten Zeit antwortet.

Behandelnde Methode: Sobald der `Controller` ein Ereignis aus der Priorityqueue ausgelesen hat, tritt die zu behandelnde Methode in Aktion. Diese beinhalten den grössten Teil der Logik des Algorithmus und erzeugt seinerseits weitere neue Ereignisse. Wird z.B das Ereignis QUERY aus der Priorityqueue gelesen, muss der zu behandelnde Peer beispielsweise beim *Depth Join* Algorithmus folgende Aktionen durchführen: Der Peer schaut in seiner Routingtabelle, welche Präfixgruppe die kleinste Tiefe hat. Ist die kleinste Tiefe beim Peer selber und es findet sich noch ein Platz für den neuen Peer, wird ein INSERT-Ereignis generiert, ansonsten ein NOSPACE-Ereignis. Ist die kleinste Tiefe hingegen in einer anderen Präfixgruppe, leitet der Peer den neuen Peer zu dieser Gruppe weiter, d.h. es wird ein ROUTETO-Event ausgelöst.

5.4.4 Triggers und Poisson-Strom

Über die `TriggerAlgorithm` werden neue Peers in ein P2P-System eingefügt, resp. sie lösen einen entsprechenden `JoinAlgorithm` aus. Der zeitliche Verlauf des Einfügens wird dabei über eine Poissonverteilung, welche in der Klassen `EventGenerator` implementiert ist, modelliert. Die Klasse `EventGenerator` erzeugt einen Strom von Ereignissen, welcher zusammengefasst eine bestimmte Ankunftsrate aufweist. Dies entspricht einem realitätsnahen Modell.

Ein Ereignisstrom, der eine Poisson-Verteilung aufweist, kann wie folgt erzeugt werden:

$$\Delta t_i = -\mu \ln(random) \tag{5.1}$$

wobei die Variablen folgende Bedeutung haben:

Δt_i **:** Zeit, bis sich das nächste Ereignis ereignet.

$\mu = 1/\lambda$ **:** Gibt die durchschnittliche Zeit, bis zum nächsten Ereignis an.

λ **:** Durchschnittliche Ereignisrate für einen bestimmten Zeitraum. Meist wird der Zeitraum gleich 1 gesetzt.

$random \in [0 \ldots 1]$ **:** Eine uniform verteilte Zufallszahl zwischen 0 und 1.

5.5 Hilfsmodul

Die Simulationsumgebung wird durch diverse Hilfsklassen ergänzt. Abbildung 5.6 zeigt alle Klassen im Überblick.

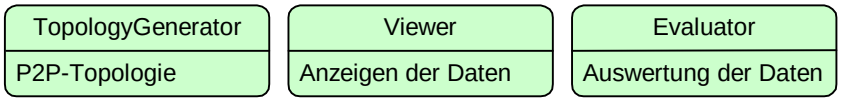

Abbildung 5.6: Übersicht der Hilfsklassen

5.5.1 Generierung der P2P Topologie

Die Klasse `TopologyGenerator` erzeugt eine Topologie, welche als Ausgangspunkt für die Simulation dient. Mit Hilfe dieser Klasse ist es möglich balancierte und unbalancierte P2P-Systeme zu generieren. Bei jedem Simulationsdurchlauf prüft der `Controller`, ob eine initiale Topologie gewünscht ist und erzeugt via dem `TopologyGenerator` eine entsprechende Topologie. Alle Parameter, die zur Generierung nötig sind, werden aus der Parameterdatei ausgelesen. Mehr zu dieser Parameterdatei findet sich im Anhang B.

Das Erzeugen eines balancierten P2P-System geschieht, indem jeder neue Peer im binären Baum feststellt, welcher Teilbaum weniger Peers hat. Sobald ein Peer bei einem Blatt angekommen ist, wird dieses Blatt in zwei Blätter aufgeteilt. Ein Blatt für den schon bestehenden Peer, das andere für den neuen Peer. Dieser Vorgang wird solange fortgesetzt bis die gewünschte Anzahl Peers im P2P-System vorhanden sind.

Ein unbalanciertes P2P-System wird über das Werfen einer unfairen Münze erzeugt. Die Overlay-ID eines neuen Peers ist zuerst noch unbestimmt. Dann wirft man eine Münze und das Resultat, Kopf oder Zahl, ergibt die erste Stelle in der Overlay-ID. Dieser Vorgang wird solange wiederholt, bis man auf ein Blatt im Baum stösst und eine Aufteilung in zwei Blätter findet statt. Dabei kann es aber durchaus vorkommen, dass ein neuer Peer die maximale Tiefe, ohne auf ein Blatt zu stossen, erreicht. Der Vorgang wird in diesem Fall nicht wiederholt, sondern der Peer fällt einfach aus dem System.

5.5.2 Sammeln und Auswerten der Simulationsdaten

Die Klasse `Evaluator` sammelt alle für eine Messung relevanten Daten, die während dem Ablauf einer Simulation anfallen. Diese Daten werden nach Ablauf einer Simulation ausgewertet und ausgegeben. Daten werden einerseits nach jedem Ereignis, anderseits in fixen Zeitabständen gesammelt:

- Nach jedem Ereignis das in der Simulation ausgeführt wird, ruft der `Controller` die Methode `Evaluator.update(event)` auf. Der `Evaluator` führt dadurch das Ereignis in seiner Statistik mit.

- Um einen Schnappschuss der Simulation zu bestimmten Zeiten zu erhalten, fügt der `Evaluator` `SNAPSHOT_EVENT`-Ereignisse in regelmässigen Zeitintervallen in die `EventQueue` ein. Das Zeitintervall wird in der Parameterdatei festgelegt. Damit wird sicher gestellt, das ein Schnappschuss der Simulation in regelmässigen Zeitintervallen erstellt wird. Der Schnappschuss speichert alle Peers und deren Tiefe ab. Damit ist es möglich einen zeitlichen Verlauf dieser Daten zu erstellen.

5.5.3 Monitoring der Simulation

Die Klasse `Viewer` vermittelt einen visuellen Eindruck der einzelnen Algorithmen während der Simulation. Die Visualisierung beschränkt sich dabei auf eine textuelle Darstellung relevanter Daten, wie z.B. die Tiefe der Peers im P2P-System. Obwohl es sich dabei um eine simple Methode handelt, vermittelt dies schon einen recht guten Eindruck des jeweiligen Algorithmus. Der Abgleich der Daten geschieht über die Methode `Viewer.update()`, welche der `Controller` in regelmässigen Zeitintervallen aufruft. Das Zeitintervall wird wiederum in der Parameterdatei festgelegt.

Kapitel 6

Simulation und Ergebnisse

6.1 Ablauf einer Simulation

Dieses Kapitel zeigt den zeitlichen Ablauf einer Simulation auf und erklärt, was in den einzelnen Schritten geschieht.

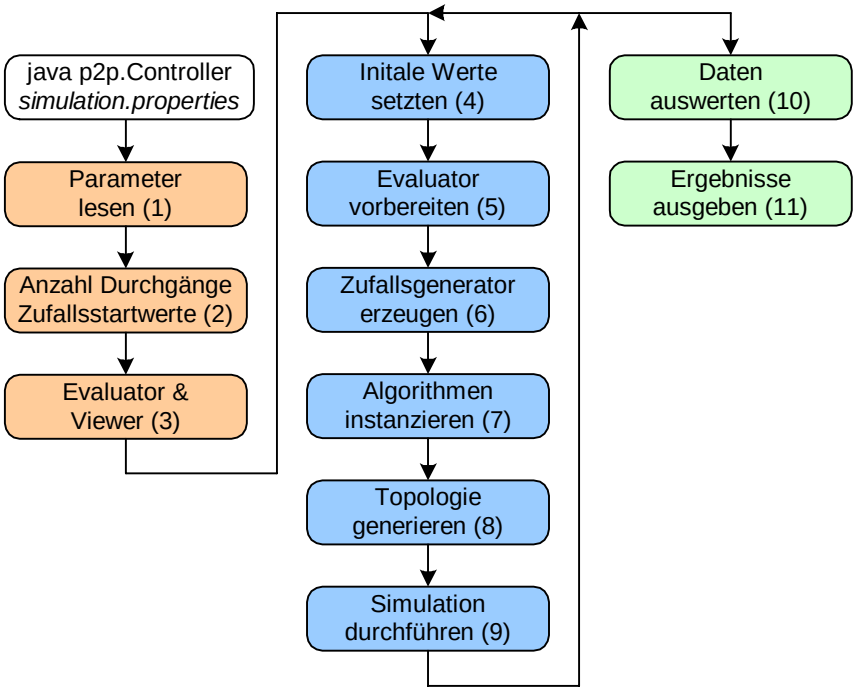

Abbildung 6.1: Der zeitliche Ablauf einer Simulation. Die mittlere Spalte wird gemäss der Anzahl Läufe wiederholt.

Nach dem Start des `Controllers`, werden zuerst alle Parameter von einer Datei eingelesen. Diese Datei enthält alle für eine Simulation nötigen Werte. In ihr wird z.B. festgehalten, welche Algorithmen zum Einsatz kommen, wie lange eine Simulation dauert oder in welchem Zeitintervall ein Schnappschuss erfolgen soll. Ausführliche Erklärungen zu den einzelnen Parametern findet sich

im Anhang B (1).

Aus der Parameterdatei wird auch die Anzahl Simulationsläufe ausgelesen. In der Regel setzt sich eine Simulation aus mehreren Läufen zusammen. Damit ist es möglich gemittelte Werte zu berechnen. Jeder Lauf wird mit einem anderen Startwert für den Zufallsgenerator durchgeführt. Diese Startwerte stammen ebenfalls aus der Parameterdatei. Dadurch wird garantiert, dass Simulationen reproduzierbar sind und Simulationen mit unterschiedlichen Algorithmen, nicht aber mit unterschiedlichen Zufallszahlen ablaufen (2).

Als nächster Schritt werden Instanzen der Klassen `Viewer` und `Evaluator` erstellt, welche eine Beobachtung resp. eine Auswertung der Simulation ermöglichen (3).

Jetzt beginnt die Vorbereitung für einen Simulationslauf. Zunächst werden alle Referenzen auf einen initialen Wert gesetzt (4). Dem `Evaluator` wird via `Evaluator.newRun()` mitgeteilt, dass ein neuer Lauf anliegt (5). Es wird ein neuer Zufallsgenerator mit dem nächsten Startwert erzeugt (6). Danach werden neue Instanzen der Algorithmen und der Triggers via der `AlgorithmFactoy` erzeugt (7). Als letzter Schritt in der Vorbereitung wird eine neue P2P-Topologie generiert (8). Diese Topologie dient als Ausgangslage für eine Simulation und wird bei jedem Lauf aus demselben Startwert[1] für den Zufallsgenerator gebildet, d.h. die Topologie ist bei jedem Lauf identisch. Der Startwert stammt ebenfalls aus der Parameterdatei.

Ein Lauf der Simulation wird nun gestartet und läuft bis eine definierte Simulationszeit erreicht ist (9). Danach werden die Vorbereitungen wiederholt und der nächste Lauf geht über die Bühne. Dieser Vorgang wird solange durchgeführt bis die Anzahl Läufe erreicht ist.

Der letzte Schritt in einer Simulation besteht darin, dass die gesammelten Werte durch den `Evaluator` ausgewertet (10) und die Ergebnisse im `Viewer` dargestellt werden (11).

6.2 Auswahl der Simulationsparameter

Die Simulation soll möglich realitätsnah durchgeführt werden. Da P2P-Systeme in der Regel aus einigen zehntausend oder hunderttausend Entitäten bestehen, muss aber ein Kompromiss zwischen der Realität und der Kapazität der Simulation resp. des verwendeten Rechners gefunden werden. Deshalb laufen Simulationen nur mit einigen hundert oder tausend Peers ab. Es wurden aber auch Simulationen mit bis zu 40'000 Peers durchgeführt und es hat sich dabei gezeigt, dass sich an den Aussagen der Resultate nichts ändert. Die hier aufgelisteten Parameter zeigen die für die Simulation verwendeten Werte.

Anzahl der Läufe : *10.* Alle Simulationen wurden mit jeweils 10 Durchgängen durchgeführt und die Resultate über diese 10 Durchgänge gemittelt.

[1] Man unterscheidet zwischen zwei Startwerten: (i) Ein Startwert zur Bildung der Topologie, (ii) ein Startwert für einen Lauf der Simulation. (i) bleibt während der Simulation unverändert. (ii) ist beim jedem Lauf unterschiedlich. (i) und (ii) sind aber unabhängig voneinander.

Delay : *0.5 Zeiteinheiten.* Stellt die Verzögerung zwischen dem Senden und Empfangen einer Nachricht dar. Ping-Messungen mit diversen Webservern zeigten einen Durchschnittswert von etwa 500ms.

Timeout : *4.0 Zeiteinheiten.* Dieser Wert wurde so gewählt, dass er mindestens das doppelte des Delays beträgt. Damit wird sichergestellt, dass ein Timeout nur bei Peerausfällen erfolgt und nicht auch bei zu langen Roundtrips beim Nachrichtenaustausch.

Länge der Overlay-ID : *24.* Ein Grossteil der Simulationen wird mit etwa 2000 Peers durchgeführt. Um 2000 Peers darstellen zu können, wird eine Overlay-ID von mindestens $\log_2(2000) \approx 11$ benötigt. Der Wert wurde nun auf die doppelte Länge gesetzt, damit Peers nicht durch einen Überlauf bei der Overlay-ID aus der Simulation fallen.

Join-Rate : 40/3600 *pro Zeiteinheit.* Um ein realitätsnahes Modell für das Einfügen von neuen Peers zu haben, wurde ein Poissonprozess verwendet.

Bootstrap-Knoten : Der Bootstrap-Knoten für das Einfügen von einem neuen Peer wird gleichverteilt zufällig aus allen im P2P-System beteiligten Peers gewählt.

Für die Simulation wird ein Rechner mit folgenden Eckdaten verwendet: Pentium IV 2.4GHz, 512MB Speicher, Windows XP und JDK 1.4.1. Der Rechner benötigt für eine Simulation mit 1000 Startpeers, einer Simulationszeit von 100'000 Zeiteinheiten und 10 Durchgängen rund 30 Minuten und belegt etwa 50 MB Arbeitsspeicher.

6.3 Messkriterien

Um quantifizierbare Aussagen über ein P2P-System zu machen, benötigt man Kriterien, die ein messbares Resultat liefern. Anhand dieser Messkriterien ist es dann möglich Aussagen über die Güte eines Systems zu erstellen. Für P2P-Systeme gibt es deren viele, wie z.B. Lastverteilung, Skalierbarkeit, Latenzzeit, verfügbare Bandbreite, Redundanz der Daten, Routingdistanz, Kosten für Look-up Operationen oder auch Sicherheitskriterien. Diese Diplomarbeit verwendet zwei einfach zu messende Kriterien: Die Tiefe und das Nachrichtenaufkommen der Peers. Das Tiefenkriterium wird bei der Beurteilung der Join-Algorithmen und *4S*-Algorithmen eingesetzt. Das Nachrichtenaufkommen wird hingegen nur bei den *4S*-Algorithmen angewendet.

6.3.1 Tiefenmass

Ein Ziel in einem P2P-Systems ist es, die Datenlast pro Peer gleich zu halten. Ein Ansatz ist, die Daten uniform über alle im System befindlichen Peers zu verteilen, d.h. jeder Peer ist im Besitz von gleich vielen Schlüssel/Objekt-Paaren wie sie unter Kapitel 2.2 vorgestellt wurden [2]. Bezogen auf den binären Baum,

[2]Bei diesem Ansatz wird davon ausgegangen, dass jeder Peer über die gleichen Kapazitäten verfügt, was natürlich ein naiver Ansatz ist. Dieses Problem kann aber z.B. über die Einführung

wie in Kapitel 2.3 vorgestellt, bedeutet das, dass eine gleichmässige Verteilung der Daten vorliegt, falls sich alle Peers auf der gleichen Tiefe befinden. Diese **optimale Tiefe** D_{Opt} hängt direkt von der Anzahl der im System befindlichen Peers ab und berechnet sich wie folgt:

$$D_{Opt} = \lfloor \log_2(n) \rfloor \tag{6.1}$$

wobei n die Anzahl Peers ist.

Das erste Messkriterium ist die **minimale Tiefe** D eines Peers. Peers, die eine kleine Tiefe aufweisen, sind in einem P2P-System nicht erwünscht, da sie potentiell eine hohe Datenlast tragen müssen. Je näher sich die Peers bei der optimalen Tiefe befinden, desto balancierter ist das P2P-System.

Das zweite Messkriterium bezeichnen wir als **Verteilungsmass (Balance Measure)** B und lautet:

$$B = \sum_{i \in V} 2^{-2d_i} > 0 \tag{6.2}$$

ist V das Set alle Peers und d_i die Tiefe des Peers i. Bei diesem Kriterium wird die Anzahl Peers und deren Tiefe berücksichtigt, wobei Peers mit kleiner Tiefe mehr ins Gewicht fallen als Peers mit grosser Tiefe. Damit man einen Vergleich zwischen Algorithmen mit unterschiedlicher Anzahl Peers ziehen kann, wird B_{Alg} für einen Algorithmus Alg mit der optimalen Verteilung B_{Opt}[3] normalisiert:

$$\rho_{Alg} = \frac{B_{Opt}}{B_{Alg}} > 0 \tag{6.3}$$

Ein optimaler Algorithmus Opt weisst dabei ein ρ_{Opt} von 1.0 auf.

6.3.2 Nachrichtenaufkommen

Ein *4S*-Algorithmus muss nicht nur in der Lage sein aktuelle Daten zu liefern, er sollte das auch mit einem möglichst geringen Nachrichtenaufkommen erreichen. Dadurch kommt es nicht zu einer zusätzlichen Belastung des P2P-Systems durch den *4S*. Um zu sehen wie viele Nachrichten ein *4S*-Algorithmus sendet, wird als weiteres Messkriterium das Nachrichtenaufkommen verwendet.

In der Simulation wird festgehalten, wie viel mal ein bestimmtes Ereignis aufgerufen wird, d.h. das Nachrichtenaufkommen M wird nicht pro Peer, sondern pro Ereignisart ermittelt. Daraus lässt sich dann das durchschnittliche Nachrichtenaufkommen $\widehat{M}$ pro Peer und Ereignisart ermitteln. Dabei wird die Grösse einer Nachricht und die dafür benötigte Zeit zum Senden und Empfangen nicht berücksichtigt.

---

von virtuellen Peer gelöst werden. Potente Peers im System repräsentieren dabei mehrere virtuelle Peers mit jeweils eigener Overlay-ID.

[3] Bei einer optimalen Verteilung B_{Opt} haben alle Peers die Tiefe $\lfloor \log_2(n) \rfloor$ oder $\lceil \log_2(n) \rceil$. Daraus lässt sich B_{Opt} folgendermassen berechnen: $B_{Opt} = (2^{\lceil \log_2(n) \rceil} - n)2^{-2\lfloor \log_2(n) \rfloor} - 2^{\lceil \log_2(n) \rceil}2^{-2\lceil \log_2(n) \rceil}$

6.4 Auswertung der Join-Algorithmen

Für die Join-Algorithmen *Random Join (RJ)*, *Number Join (NJ)* und *Depth Join (DJ)* werden jeweils drei Simulationen mit unterschiedlicher Ausgangslage durchgeführt; (i) 2 Peers, (ii) 995 Peers mit einer balancierten Verteilung und (iii) und 995 Peers mit einer unbalancierten Verteilung. Die Simulationszeit beträgt 100'000 Zeiteinheiten, was zu etwa 1'000 neuen Peers führt. Für den Datenaustausch des *4S* verwenden alle Simulationen den *Periodic 4S* Algorithmus.

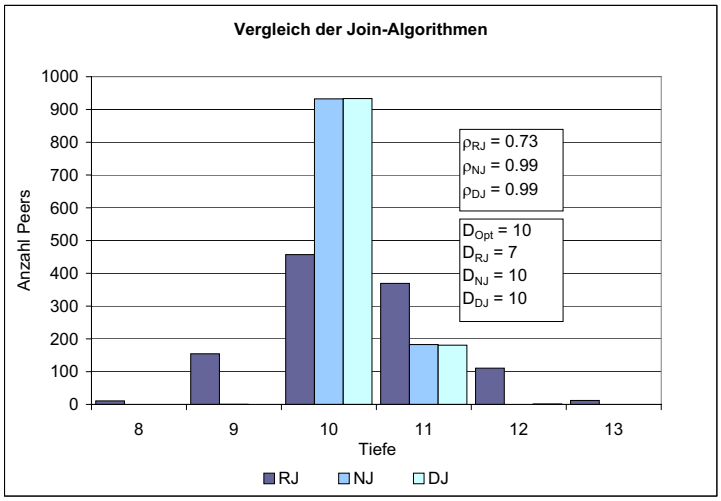

Abbildung 6.2: Vergleich der Join-Algorithmen mit einem initialen P2P-System bestehend aus zwei Peers.

Bei der Ausgangslage (i) zeigt sich, dass der *NJ*-Algorithmus mit einem $D_{NJ} = 10$ und $\rho_{NJ} = 0.99$ die besten Werte aufweist. Der *DJ*-Algorithmus ist nur unwesentlich schlechter, hingegen zeigt sich beim *RJ* das "Ball-into-Bins" Problem ganz deutlich. Peers finden sich auf den Tiefen 7 bis 13, bei einer optimalen Tiefe D_{Opt} von 10, wieder. Alle Resultate sind in der Abbildung 6.2 aufgelistet.

Bei der Ausgangslage (ii) zeigt sich bei den Ergebnissen keinen Unterschied zu (i), d.h. solange Peers in ein balanciertes P2P-System eingefügt werden, bleibt die Güte der Algorithmen auf gleichem Niveau (Abbildung 6.3).

Bei der letzten Simulation mit der Ausgangslage (iii) zeigt sich hingegen die Schwäche des *NJ*-Algorithmus. Das in Kapitel 4.1.2 skizzierte Problem beim Einfügen von Peers in unbalancierte P2P-Systeme zeigt sich hier in aller Form. Die minimale Tiefe $D_{NJ} = 6$ ist wesentlich kleiner als die optimale Tiefe $D_{Opt} = 11$. Zwar werden die meisten Peers bei der optimalen Tiefe eingefügt, trotzdem bleiben Bereiche des P2P-System unangetastet und verändern ihre Tiefe nur unwesentlich. Der *DJ*-Algorithmus verhält sich bei unbalancierten P2P-Systemen ungleich besser, wie man aus der Abbildung 6.4 herauslesen kann.

Aus den durchgeführten Simulationen lässt sich ableiten, dass der *DJ*-

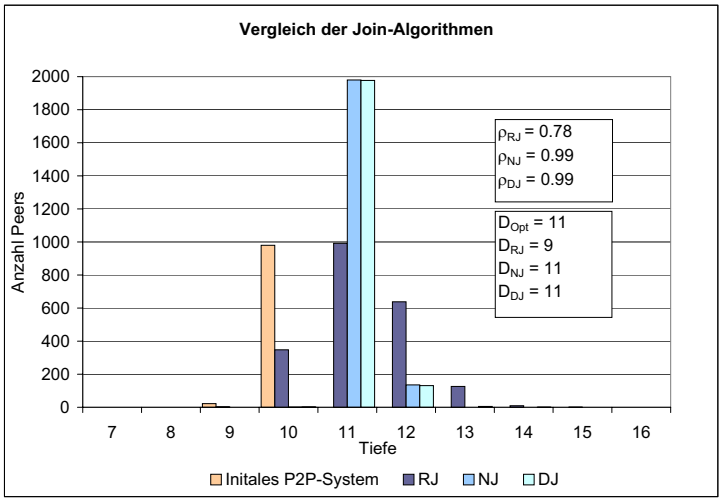

Abbildung 6.3: Vergleich der Join-Algorithmen. Ausgangslage sind 995 Peer in einem balancierten P2P-System.

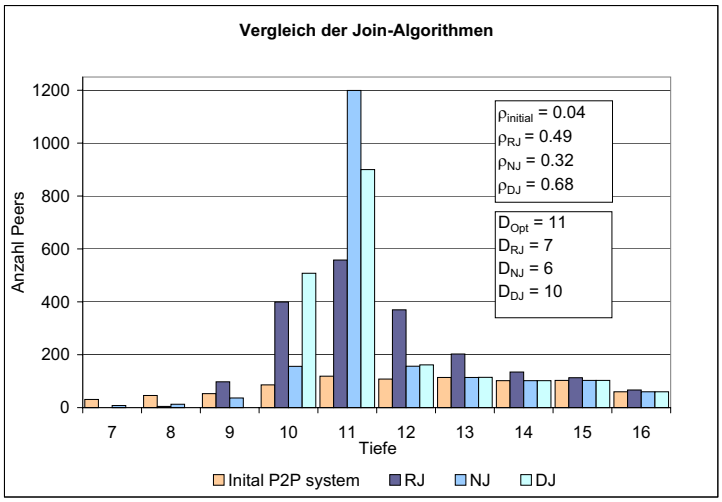

Abbildung 6.4: Vergleich der Join-Algorithmen. Ausgangslage sind 995 Peer in einem unbalancierten P2P-System.

Algorithmus den Algorithmen *RJ* und *NJ* überlegen ist, vor allem bei unbalancierten P2P-Systemen.

6.5 Auswertung der *4S*-Algorithmen

Bei den Simulationen der *4S*-Algorithmen werden die Algorithmen *Periodic 4S*, *Adaptive 4S* und *Piggyback 4S* getestet. Alle Algorithmen werden in Kombination mit dem *Depth Join* Algorithmus durchgeführt. Die Ausgangslage ist jeweils ein P2P-System mit 2 Peers. Die Simulationszeit beträgt 50'000 Zeiteinheiten,

wodurch etwa 500 neue Peers eingefügt werden.

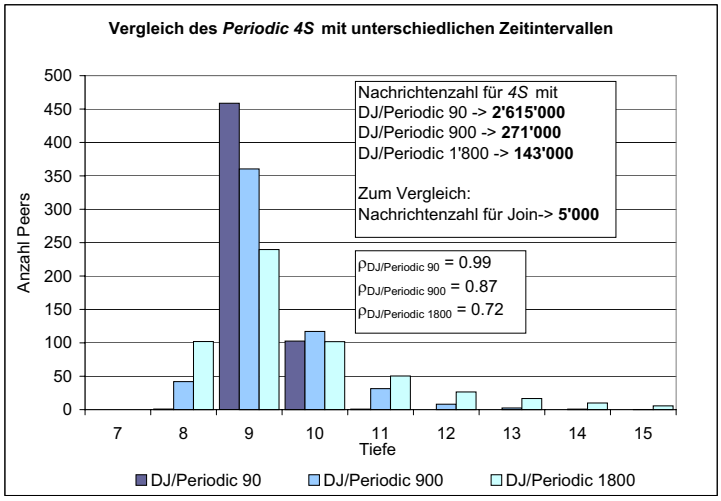

Abbildung 6.5: Vergleich des *Periodic 4S* Algorithmus mit unterschiedlichen Zeitintervallen.

Beim ersten Algorithmus, dem *Periodic 4S* Algorithmus, werden Daten ohne Einschränkung in regelmässigen Zeitintervallen zwischen den Nachbar-Peers ausgetauscht. Bei der Simulationen des *Periodic 4S* werden deshalb verschiedene Zeitintervalle getestet. Ein Überblick der Resultate ist in Abbildung 6.5 dargestellt. Es zeigt sich, dass bei einem kleinen Zeitintervall die Genauigkeit der *4S*-Daten hoch ist und mit zunehmendem Zeitintervall abnimmt. In der Simulation kommt das durch D_{DJ} und ρ_{DJ} zum Ausdruck. Bei einem Intervall von 90 Zeiteinheiten beträgt der Wert für $D_{DJ} = 9$ und $\rho_{DJ} = 0.99$. Beim Intervall 1800 sind die Werte lediglich noch $D_{DJ} = 8$ und $\rho_{DJ} = 0.72$. Die optimale Tiefe liegt bei $D_{Opt} = 9$. Je kleiner das Zeitintervall ist, desto exakter sind die *4S*-Daten. Betrachten man aber den Nachrichtenaufwand M, wird dieses umso grösser, je kleiner das Zeitintervall ist. Bei 90 werden insgesamt $M = 2.6$ Millionen Nachrichten bei 563 Peers ausgetauscht. Das Nachrichtenaufkommen für das Einfügen von 563 Peers beträgt hingegen lediglich ein paar Tausend.

Um das Nachrichtenaufkommen zu reduzieren wurden die Algorithmen *Adaptive 4S* und *Piggyback 4S* eingeführt. Zur Erinnerung: *Adaptive 4S* tauscht *4S*-Daten nur bei einer Änderung der *4S*-Daten aus, *Piggyback 4S* sendet die *4S*-Daten innerhalb des normalen Nachrichtenaufkommens. Vergleicht man den *Adaptive 4S* mit dem *Periodic 4S*, sieht man das die Verteilung der Peers zwar abnimmt, aber der Nachrichtenaufwand massiv reduziert wird (Abbildung 6.6). Beim *Piggyback 4S* reduzieren sich die Werte von D_{DJ} und ρ_{DJ} nochmals, aber die Anzahl der Nachrichten liegt bei Null. Abbildung 6.6 zeigt alle Strategien im Vergleich.

Die letzte Simulation ist ein Vergleich zwischen dem *Random Join* und dem *Depth Join* Algorithmus. Wobei beim *DJ*-Algorithmus der *Adaptive 4S*, resp. *Piggyback 4S* zur Anwendung kommt. Um die Skalierbarkeit der Algorithmen zu zeigen, werden in dieser Simulationen einige Tausend Peers eingefügt. Ab-

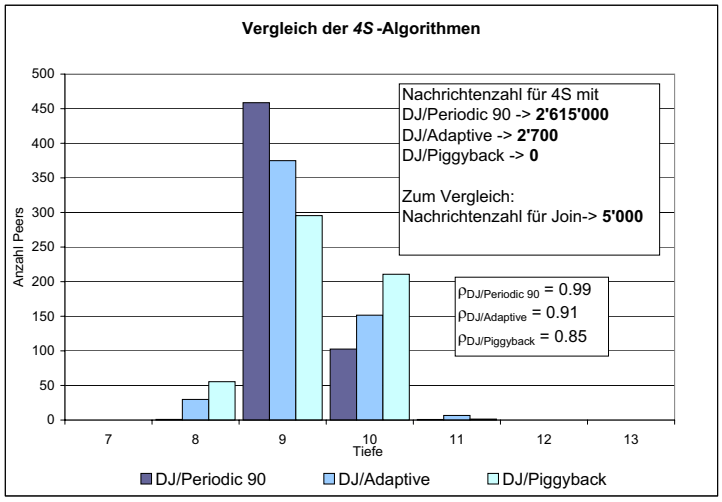

Abbildung 6.6: Vergleich der *4S*-Algorithmen.

bildung 6.7 illustriert, dass der *DJ/Piggyback 4S* zwar etwas schlechtere Werte als der *DJ/Adaptive 4S* liefert, aber gegenüber dem *RJ* die Nase vorn hat.

Der *Piggyback 4S* Algorithmus nützt in den Simulationen lediglich das Nachrichtenaufkommen des Join-Algorithmus, stellt dies doch die einzige Applikation dar. Beim Einsatz von weiteren Applikationen werden die Werte des *4S* noch exakter und damit auch die Güte der Join-Algorithmen. Wir glauben, das ein P2P-System das lediglich den *Piggyback 4S* Algorithmus einsetzt, zum gleichen Resultat wie mit dem *Adaptive 4S* Algorithmus führt.

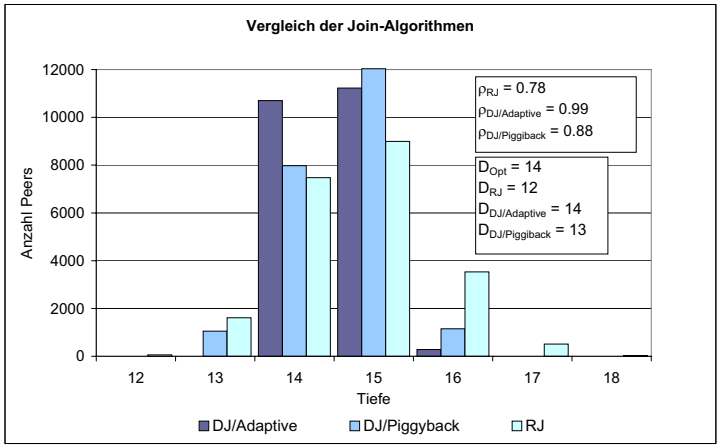

Abbildung 6.7: Anbildung zeigt die Verteilung der Peers nachdem 22'211 Peers eingefügt wurden.

Kapitel 7

Fazit und Erweiterung

7.1 Fazit

Diese Diplomarbeit befasst sich mit der Entwicklung von *4S*- und Join-Algorithmen und deren Implementierung in einer P2P-Simulation.

Der Steady-State-Statistics-Service sammelt statistische Daten über ein P2P-System und propagiert diese im P2P-System. In einem statischen P2P-System sind die Daten exakt, in einem dynamischen System werden hingegen nur approximative Werte geliefert. Ausgehend von den Präfixgruppen, werden Daten in einer "Buttom-Up"-Manier zwischen den Peers ausgetauscht. Wir sehen *4S* als einen Grundbaustein in einem P2P-System, um von unterschiedlichsten Applikationen verwendet zu werden.

Als beispielhafte Applikation für den *4S* wurden zwei Join-Algorithmen entwickelt: *Number Join* und den *Depth Join*. Beide Algorithmen fügen neue Peers deterministisch in ein P2P-System ein, indem Peers gezielt in "Regionen mit kleiner Dichte" weiter geleitet werden. Als Dichtemass kommt dabei die Anzahl Peers resp. die minimale Tiefe zum Einsatz. Dabei hat sich gezeigt, dass der *Depth Join* auch mit unbalancierten P2P-System umgehen kann und gegenüber herkömmlichen Join-Algorithmen (*Random Join*) bessere Ergebnisse liefert.

Beim Datenaustausch des *4S* wurden drei Mechanismen implementiert: *Periodic 4S*, *Adaptive 4S* und *Piggyback 4S*. Der *Periodic 4S* liefert die genauesten Daten, erreicht dies aber mit einem ungleich hohen Nachrichtenaufkommen. Der *Periodic 4S* ist deshalb kein geeigneter Mechanismus für den Datenaustausch. Die Algorithmen *Adaptive 4S* und *Piggyback 4S* liefern *4S* Daten, die immer noch genügend exakt für die Join-Algorithmen sind. Dabei ist das Nachrichtenaufkommen beim *Adaptive 4S* um etliche Zehnerpotenzen kleiner als beim *Periodic 4S*. Der *Piggyback 4S* kommt gänzlich ohne zusätzliches Nachrichtenaufkommen aus und belastet das System deshalb nicht zusätzlich.

Alle hier vorgestellten Algorithmen wurden in eine Simulation integriert und getestet. Für die Simulation wurde ein Framework erstellt. Das Framework erlaubt eine einfache Integration zusätzlicher Applikationen. Durch den modularen Aufbau ist es darüber hinaus einfach, Teile auszutauschen und die Simulation an andere Situationen anzupassen.

Im Zentrum dieser Diplomarbeit stand die Idee des Steady-State-Statistics-

Service. Ziel war es aus der Idee eine erste Applikation zu entwickeln und zu testen. Das Leitmotiv war dabei, die Idee des *4S* zu überprüfen und zu bestätigen. Anhand der durchgeführten Simulationen kann ein positives Fazit gezogen werden. Der Steady-State-Statistics-Service ist umsetzbar und führt beim Einfügen von Peers zu besseren Ergebnissen als mit konventionellen Methoden.

7.2 Erweiterung

Durch den erhöhten Nachrichtenaustausch der regulären Applikationen, wird auch der *4S* Datenaustausch erhöht. Es ist deshalb sinnvoll weitere Applikationen (z.B. Suchanfragen) in die Simulation zu integrieren, um die Genauigkeit der *4S*-Daten des *Piggyback 4S* bei erhöhtem Nachrichtenaufkommen zu verbessern.

Diese Diplomarbeit hat gezeigt, dass der Steady-State-Statistics-Service ein sinnvoller Grundbaustein in einem P2P-System ist und grundsätzlich funktioniert und machbar ist. Eine Simulation bleibt aber eine Simulation. Es ist interessant zu sehen, wie sich die hier vorgestellten Algorithmen in einem echten P2P-System verhalten. Eine Integration in ein bestehendes P2P-System ist deshalb ein möglicher nächster Schritt für den Steady-State-Statistics-Service.

Anhang A

Simulationsumgebung

Die Simulation wurde mit Hilfe von Java JDK1.4 unter Windows XP entwickelt. Getestet wurde das Programm unter JDK1.4 und JDK1.3, sollte aber ohne Anpassung auch unter JDK1.2 lauffähig sein.

A.1 Starten der Simulation

Um die Simulation zu starten, geht man folgendermassen vor:

1. Öffnen einer Konsole/Eingabeaufforderung.
2. Wechseln ins Verzeichnis in dem die Java-Klassen der Simulation gespeichert sind:
   `>cd [Installationspfad]/Implementation/`
3. Starten der Simulation:
   `>java -Xmx400m -cp .;jdsl/lib/jdsl.jar p2p.Controller simulation.properties`

Die Simulation benötigt als einzigen Parameter den Namen einer Datei (z.B. `simulation.properties`), welche alle weiteren Parameter für die Simulation enthält. Weitere Informationen zu dieser Parameterdatei finden sich im Anhang B.

A.1.1 Optionen für die Java Virtual Machine

-Xmx400m : Jeder Peer in der Simulation benötigt rund 7KB Speicher. Ab einigen tausend Peers im System reicht die standardmässig zugewiesene Heapgrösse(rund 70 MB) für die Java VM nicht mehr aus. Mit dieser Option wird die maximale Heapgrösse erhöht. Zum Beispiel setzt -Xmx400m die maximale Heapgrösse auf 400MB.

-cp : Diese Option setzt den Klassenpfad der benötigten Java-Klassen. Das Semikolon (;) trennt dabei die einzelnen Pfade. Der Punkt (.) setzt den Klassenpfad gleich dem momentanen Verzeichnis. Mit `jdsl/lib/jdsl.jar` setzt man den Klassenpfad auf die externe Bibliothek JDSL 2.0[9]. Diese Bibliothek wird für die Eventverwaltung der Simulation verwendet.

Des weiteren muss der Klassenpfad auf das JDK selber gesetzt sein, welches aber in der Regel der Fall ist.

A.2 Auflistung aller Algorithmen

Um eine aktuelle Liste aller implementierten Algorithmen zu bekommen, startet man die Simulation mit dem Parameter -listAlgos:

```
>java -cp .;jdsl/lib/jdsl.jar p2p.Controller -listAlgos
```

Anhang B

Die Parameterdatei zur Simulation

In der Parameterdatei werden alle für die Simulation benötigten Werte abgespeichert. Dabei handelt es sich um eine ASCII-Datei, in welcher pro Zeile ein Schlüssel/Wert-Paar steht. Gross- und Kleinschreibung wird dabei nicht beachtet. Alle Zeilen die mit einem Hashsymbol (#) beginnen, werden als Kommentar behandelt.

B.1 Die Parameter

Im folgenden werden alle Parameter alphabetisch aufgelistet und erklärt. Typische Werte zu den einzelnen Parametern finden sich im Anhang B.2.

4SALGORITHM : Gibt den Namen eines Steady-State-Statistics-Service-Algorithmus an. Anhand des Namens wird über die `AlgorithmFactory` die entsprechende Instanz generiert. Es werden folgende Algorithmen angeboten:

Periodic4S : Periodic 4S

Adaptive4S : Adaptive 4S

Adaptive4SGlobalUpdate : Adaptive 4S with global update

Piggyback4S : Piggyback 4S

DEPTH : Gibt die maximale Länge der Overlay-ID an. Dies ist implementierungsbedingt auf 31 beschränkt, da Peer-IDs durch `Integer`-Werte repräsentiert werden.

DELAY : Gibt die durchschnittliche Übertragungszeit in Zeiteinheiten zwischen zwei Peers an.

FAILURERATE : Legt die Rate fest, in welcher Peers ausfallen. Die Rate bezieht sich dabei auf 3600 Zeiteinheiten und nicht auf eine Zeiteinheit.

JOINALGORITHM : Gibt den Namen eines Join-Algorithmus an. Anhand des Namens wird über die `AlgorithmFactory` die entsprechende Instanz generiert. Es werden folgende Algorithmen angeboten:

NJ : Number Join Algorithm

DJ : Depth Join Algorithm

DJPiggyback : Depth Join Algorithm with Piggyback 4S

RJ : Random Join Algorithm

JOINRATE : Legt die Rate fest, in welcher neue Peers dem P2P-System hinzugefügt werden. Die Rate bezieht sich dabei auf 3600 Zeiteinheiten und nicht auf eine Zeiteinheit.

NOISE : Der Wert der Übertragungszeit zwischen zwei Peers (DELAY) kann mit dem Parameter NOISE zufällig verändert werden werden. NOISE bewegt sich zwischen 0.0 und 1.0. Ein Wert von 0.0 ändert den DELAY nicht. Bei 1.0 wird zum DELAY einen Wert der Übertragungszeit selber addiert oder subtrahiert. Beispiel: DELAY = 4.0, NOISE=0.5. Die Übertragungszeiten bewegen sich nun im Bereich von 4.0 ± 2.0 Zeiteinheiten.

NOISE_LEVEL : Algorithmen protokollieren ihre Aktivitäten in eine Konsole. NOISE_LEVEL ermöglicht, die Ausgabe einzuschränken oder ganz auszublenden. Setzt man diesen Parameter auf Null, erfolgt keine Ausgabe, bei einem Wert von vier, wird alles ausgegeben. Werte dazwischen erzeugen mehr oder weniger Ausgabe.

Eine Simulation kann mit einer Basis-Topologie gestartet werden. Folgende vier Parameter werden für das Erzeugen dieser Topologie verwendet:

PRESIM-NODENUMBER : Legt die Anzahl Peers fest. Die Basis-Topologie besteht immer aus mindestens zwei Peers, auch falls man hier den Wert Null angibt. Wählt man als Strategie 'dice' kann es sein, dass man nicht exakt die gewünschte Anzahl Peers erhält. Dies gilt insbesondere bei unbalancierten P2P-Systemen oder falls das P2P-System fast gesättigt ist. Der Grund liegt im der Implementierung des Algorithmus. Eine Beschreibung des Algorithmus findet sich in Kapitel 5.5.1.

PRESIM-STRATEGY : Legt die Strategie der Generierung einer Topologie fest. Es gibt zwei Strategien:

balance : Erzeugt eine Topologie, in dem Overlay-IDs uniform verteilt sind.

dice : Erzeugt eine unbalancierte Topologie, d.h. es werden entweder Overlay-ID mit mehr Nullen oder Einsen generiert. Der Grad der Unbalanciertheit wird über PRESIM-PROBABILITY gesteuert.

PRESIM-PROBABILITY : Damit wird der Grad der Unbalanciertheit festgelegt. Ein Wert von 0.5 erzeugt etwa gleichviel Nullen und Einsen in der Overlay-ID. Werte näher bei 0.0 resp. 1.0 erzeugen mehr Nullen resp. Einsen. Bei der Strategie 'balance' hat dieser Parameter keine Bedeutung.

PRESIM-RANDOMSEED : Gibt den Startwert für den Zufallsgenerator an.

PROPAGATION_INTERVALL : Einzelne Steady-State-Statistics-Service-Algorithmen werden in festgelegten Zeitintervallen gestartet. PROPAGATION_INTERVALL legt diesen Wert fest.

RUNS : Gibt an wie viele Simulationsläufe ausgeführt werden. Werte die nach Ablauf aller Simulationen ausgegeben werden, sind gemittelt über diese Anzahl.

SEEDS : Startwerte für den Zufallsgenerator einer Simulation. Jeder Simulationslauf hat seinen eigenen Startwert, d.h. die Anzahl RUNS und die Anzahl SEEDS müssen übereinstimmen.

SHOW_NODES_BEFORE_SIMULATION : Mit diesem Parameter legt man fest, ob alle vorhandenen Peers zu Beginn der Simulation ausgegeben werden. Wird vor allem für die Fehlersuche benötigt. Mögliche Werte sind 'yes' und 'no'.

SHOW_NODES_AFTER_SIMULATION : Mit diesem Parameter legt man fest, ob alle vorhandenen Peers nach Abschluss der Simulation ausgegeben werden. Wird vor allem für die Fehlersuche benötigt. Mögliche Werte sind 'yes' und 'no'.

SIMULATIONTIME : Legt die Laufzeit einer Simulation fest. Dabei handelt sich um eine 'virtuelle' Zeit, d.h. gibt man eine Zeit von 50000 an, wird die Simulation 50000 Zeiteinheiten laufen, wobei eine Zeiteinheit losgelöst von der 'echten' Zeit ist. Um ein Bild zu bekommen, in welchem Zeitrahmen die Simulation operiert, ist es aber hilfreich, sich Sekunden darunter vorzustellen.

SNAPSHOT_INTERVALL : Legt fest, nach wie vielen Zeiteinheiten ein Schnappschuss der Simulation erfolgt. Alle Daten der Simulation werden dabei festgehalten und am Schluss der Simulation angezeigt.

TIME_OUT : Join-Algorithmen benötigen einen Time out. Die Zeit nach dem der Time out abläuft, wird mit dem Parameter TIME_OUT festgelegt.

VIEWER_UPDATE_INTERVALL : Legt fest nach wie vielen Zeiteinheiten, der `Viewer` nachgeführt wird. Kleine Werte erzeugen sehr viele Updates, aber verlangsamen die Simulation beträchtlich. Als guter Wert zur Beobachtung der Simulation hat sich 100 herausgestellt. Möchte man die Simulation während eines Durchgangs nicht beobachten, sondern ist nur an den Endresultaten interessiert, setzt man einen hohen Wert ein.

B.2 Ein Beispiel

Die hier abgebildete Parameterdatei stellt ein vollständiges Beispiel dar und ist zur weiteren Illustration der Parameterdatei wiedergegeben:

```
#Key = Value
RUNS = 10
SEEDS = -147411 101 454487894 42 -147411 5987451259874 47894653
-465 7884 12456 -789
SIMULATIONTIME = 50000
JOINALGORITHM = DJ
4SALGORITHM = Adaptive4S
JOINRATE = 40
FAILURERATE = 0
# parameters that relate to pre-simulation build network
PRESIM-NODENUMBER  = 0
PRESIM-STRATEGY = dice
PRESIM-PROBABILITY = 0.9
PRESIM-RANDOMSEED  = 101
# waiting time between propagation queries [time units]
PROPAGATION_INTERVALL = 90.0
# overlay address range
DEPTH = 24
# base delay when sending data between two peers [time units]
DELAY = 0.5
# time out [time units]
TIME_OUT = 4.0
# indicate how much noise is added to the delay [base delay times
+-noise]
NOISE = 0.0
# time between snapshots of a simulation [time units]
SNAPSHOT_INTERVALL = 10000
# time between updates of the viewer [time units]
VIEWER_UPDATE_INTERVALL = 100
# noise level for monitor (replacement for System.out.println)
0=whisper, 4=scream NOISE_LEVEL = 0
# list all peers in the system prior to simulation
SHOW_NODES_BEFORE_SIMULATION = no
# list all peers after simulation is finished
SHOW_NODES_AFTER_SIMULATION = no
```

Anhang C

Klassendiagramm

Die hier abgebildeten Klassen zeigen die wichtigsten Klassen und ihre Abhängigkeit untereinander. Jeder gestrichelt umrandete Bereich stellt einen der in Kapitel 5.2 vorgestellten Module (Simulationsumgebung, Algorithmen und Hilfsmodul) dar. Das Diskettensymbol symbolisiert das Laden von externen Werten aus einer Datei.

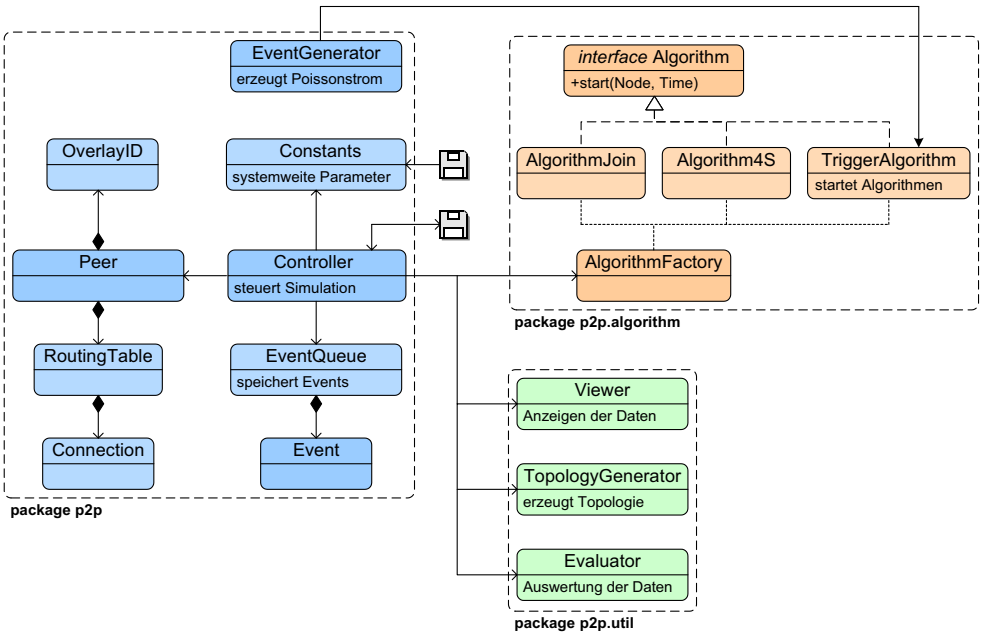

Abbildung C.1: Klassenübersicht

Anhang D

Implementierung eines neuen Algorithmus

Die hier aufgeführten Punkte stellen eine Checkliste dar, welche zeigt was alles zu erledigen ist um einen neuen Algorithmus in die Simulation einzufügen.

1. Der grösste Anteil stellt die Implementierung des neuen Algorithmus dar. Dieser muss das Interface `Algorithm` implementieren. Nur so ist es dem `Controller` möglich einen Algorithmus auch zu verwenden, da der `Controller` alle Algorithmen über dieses Interface anspricht. Wie man einen Algorithmus umsetzt, wird in Kapitel 5.4 erklärt.

2. Die `AlgorithmFactory` wird mit dem Algorithmus- und dem Klassennamen (inklusive Packetname) ergänzt, z.B. "MYALGO" und "p2p.algorithm.join.MyAlgo". Der Algorithmus kann danach über den Namen "MYALGO" angesprochen werden.

Diese zwei Punkte stellen das Minimum für die Implementierung eines neuen Algorithmus dar. Will man einen Algorithmus mit neuen, zusätzlichen Ereignissen realisieren, muss man auch noch folgende Punkte beachten:

1. Die Klasse `Constants` wird um die neuen Ereignisse resp. `ActionID`s ergänzt. Dazu wird eine neue statische und globale Referenz definiert und die Methoden `mapActionIDToString` und `getEventTypes` entsprechend erweitert. `Constants` verwaltet alle Ereignistypen im System, d.h. alle `ActionID`s der Simulation sind dort aufgelistet. `ActionID`s werden dadurch an einem zentralen Ort verwaltet und eine doppelte Verwendung von `ActionID`s wird verhindert.

2. Der Switch-Block der Methode `simulate` in der Klasse `Controller` wird um die neuen Ereignisse ergänzt. Damit wird sichergestellt, dass der `Controller` den richtigen Algorithmus aufruft.

3. Controller erzeugt in der Methode `init` je eine Instanz eines Join- resp. *4S*-Algorithmus und ebenso eine Instanz für das Auslösen(Trigger) von Joins. Will man noch einen weiteren Algorithmustyp oder Trigger im System haben, muss der `Controller` noch um diese erweitert werden.

4. Will man noch zusätzliche Auswertungen über die Simulation haben, muss die Klasse `Evaluator` entsprechend angepasst werden.

5. Dasselbe gilt für die Klasse `Viewer`, falls man zusätzliche Werte im `Viewer` dargestellt haben will.

Literaturverzeichnis

[1] Keno Albrecht, Ruedi Arnold, Michael Gähwiler, and Roger Wattenhofer. Join and leave in peer-to-peer systems: The steady state statistics service approach. Technical Report TR 411, ETH Zurich Switerland, July 2003.

[2] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46, 2003.

[3] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge, USA*, March 2002.

[4] Xiannong Meng. Csci6337 (simulation) Courseware, University of Texas - Pan American, http://www.cs.panam.edu/~meng/course/cs6337/.

[5] Martin Raab and Angelika Steger. "balls into bins" — A simple and tight analysis. *Lecture Notes in Computer Science*, 1518, 1998.

[6] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[7] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[8] Secure Hash Standard SHA-1. National Institute of Standards and Technology, http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[9] Java Data Structures Library JDSL 2.0 Standard. Brown University, http://www.cs.brown.edu/cgc/jdsl/.

[10] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[11] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.