

Universität Hannover
Institut für Informationssysteme
Fachgebiet Wissensbasierte Systeme

Bachelorarbeit

**Simulation eines schema-basierten Super-Peer-
Netzwerkes**

Verfasser: Marc Herrlich

Erstprüfer: Prof. Dr. techn. Dipl.-Ing. Wolfgang Nejdl

Zweitprüfer: PD Dr. Friedrich Steimann

Betreuer: Dipl.-Inform. Wolf Siberski

Zusammenfassung

Im Zuge der Weiterentwicklung von P2P Netzwerken als Teil des *Semantic Web* und der Entwicklung intelligenterer P2P-Protokolle auf Basis von schema-basierten Metadaten Standards, wie RDF oder OWL, steigt gleichzeitig die Nachfrage nach Testmöglichkeiten dieser neuen Entwicklungen. Bedingt durch den hohen Aufwand, mit dem Tests in realen Netzwerken verbunden sind, ist es oftmals nicht möglich solche Tests mit der nötigen Anzahl von Teilnehmern durchzuführen, die für aussagekräftige Daten notwendig wäre. Hier erscheint das Mittel der Simulation oftmals als angebrachter oder zumindest als geeigneter, um einen ersten Überblick zu gewinnen.

In dieser Arbeit wird eine Java Simulationsumgebung auf Basis des *Scalable Simulation Framework* zur Simulation schema-basierter Super-Peer-Netzwerke vorgestellt.

Erklärung

Hiermit erkläre ich, diese Arbeit selbständig verfasst und keine außer den angegebenen Quellen und Hilfsmitteln verwendet zu haben.

Hannover, Oktober 2003.

Marc Herrlich

Danksagung

Mein besonderer Dank richtet sich an Wolf Siberski für seine Betreuung während dieser Arbeit.

Inhaltsverzeichnis

1	Einleitung	7
1.1	Motivation	7
1.2	Verwendete Software und Bibliotheken.....	8
2	Diskrete Simulation	8
2.1	Definition Simulation.....	8
2.2	Modellbildung / Diskrete und kontinuierliche Simulation.....	9
2.3	Wichtige Konzepte der diskreten Simulation	9
3	Das Scalable Simulation Framework (SSF)	10
3.1	Die Klassen des SSF	11
3.1.1	Entity	12
3.1.2	Event	12
3.1.3	process.....	12
3.1.4	inChannel / outChannel.....	12
4	Das Modell.....	13
4.1	Modellierung der Schemainformation.....	13
4.2	Modellierung der Netzwerkverbindungen	15
4.3	Modellierung der Peers	15
4.3.1	Die Consumer-Peers.....	15
4.3.2	Die Provider-Peers.....	16
4.3.3	Die Super-Peers.....	16
4.4	Modellierung der Topologie.....	17
5	Die Implementierung	17
5.1	Die wichtigsten Klassen und ihre Aufgaben im Überblick.....	17
5.1.1	Simulation	18
5.1.2	SimConfig.....	18
5.1.3	AbstractPeer / SuperPeer / Peer	18
5.1.4	ProcessContainer.....	20
5.1.5	Message / InformationMsg / Query / Response	20
5.1.6	Topology	21
5.1.7	SchemaConfig / SchemaConfigurator	22
5.1.8	Schema / Property / SchemaCapability	23
5.2	Die Konfiguration der Simulation	23
5.2.1	Aufbau der Datei „simulation.xml“	24
5.2.2	Aufbau der Datei „schema.xml“	27
5.2.3	Aufbau der Datei „distribution.xml“	30
5.3	Die Peers.....	30
5.3.1	Das Senden und Empfangen von Nachrichten	31
5.3.2	Die Lösung des Prozess-Problems	32
5.3.3	Die Consumer-Peers.....	33
5.3.4	Die Provider-Peers.....	35
5.3.5	Die Super-Peers.....	37
5.4	Die Topologien	39
5.4.1	Die Broadcast-Topologie.....	39
5.4.2	Die Hypercube Topologie.....	40
5.5	Die statistische Auswertung und Ausgabe	42

6	Beispiel Szenario	44
7	Leistungsfähigkeit des Simulators / Ausblick.....	48
8	Anhang	49
8.1	DTDs	49
8.2	Referenzen.....	51

1 Einleitung

Im Zuge der Entwicklung des Semantic Web [1] in den letzten Jahren und dem damit verbundenen Bedarf an Sprachen und Standards zur Beschreibung und Verwaltung von Metadaten, beginnt sich auch in anderen Bereichen, wie Peer-to-Peer Anwendungen, eine ähnliche Entwicklung und ein Verschmelzen mit dem HTML basierten Web abzuzeichnen. Dabei steht bei diesen Anwendungen in erster Linie nicht das Tauschen von Dateien, sondern die Suche nach Information im Vordergrund. Verschiedene Projekte weltweit, insbesondere das Edutella-Projekt [2], auf welches sich diese Arbeit in vielen Teilen bezieht, beschäftigen sich inzwischen mit verschiedenen Ansätzen, um leistungsfähige Peer-to-Peer Anwendungen zu entwickeln, die Metadaten einsetzen, um Informationen zu klassifizieren und darauf effiziente Suchalgorithmen aufbauen zu können. Vielfach kommen dabei schema-basierte Standards, wie das vom W3C entwickelte Resource Description Framework (RDF) [3], zum Einsatz. Die Edutella Peers beispielsweise verwenden solche schema-basierte Metadaten, um das Routing von Anfragen und Broadcasts im Netzwerk besonders effizient und Ressourcen schonend zu erledigen. So werden Anfragen nur an Peers weitergeleitet, deren Schemainformationen mit denen der Anfrage möglichst übereinstimmen, um überflüssige Anfragen an Peers, die diese auf keinen Fall beantworten können, zu vermeiden (siehe dazu auch [4]). Ein weiterer wichtiger Schritt auf dem Weg zu besseren, d.h. performanteren und stabileren Peer-to-Peer Lösungen, ist der Schritt von reinen Peer-to-Peer Netzwerken hin zu Super-Peer-Netzwerken, bei denen spezielle, besonders leistungsfähige Peers, zu Super-Peers werden, die die Aufgabe haben, das Netzwerk zu organisieren und für das Routing der Anfragen zuständig sind, also einen so genannten „Backbone“ bilden (siehe dazu auch [4]).

1.1 Motivation

Ein wichtiger Punkt bei der Entwicklung von schema-basierten Peer-to-Peer Netzwerken sind effiziente Such- und Routingalgorithmen. So haben die Topologie des Netzwerkes und das Routing natürlich einen starken Einfluss auf Netzwerklast, Zuverlässigkeit des Netzwerkes, Dauer der Suche und andere Parameter. Um die Entwicklung solcher Algorithmen weiter voranzutreiben, bedarf es neben theoretischen Überlegungen auch praktischer Tests, um Implementierungen zu überprüfen und empirische Daten zu gewinnen. Da im Gegensatz zu den bekannteren Peer-to-Peer Netzwerken, wie z.B. Gnutella (siehe dazu auch [5] und [6]), die aktuellen schema-basierten Peer-to-Peer bzw. Super-Peer Netzwerke nur im Testbetrieb mit wenigen teilnehmenden Peers laufen und es sicher auch nicht möglich wäre, bei tausenden von Benutzern ständig neue Algorithmen zu testen, muss auf das Mittel der Simulation

zurückgegriffen werden, um dem oben genannten Bedarf nach Daten gerecht zu werden.

Ziel dieser Arbeit ist es daher, eine Simulationsumgebung zu schaffen, um Daten über das Verhalten verschiedener Routingalgorithmen in einer schema-basierten Super-Peer Umgebung zu gewinnen.

Eine besondere Schwierigkeit bei der Simulation von Peer-to-Peer Netzwerken besteht in der großen Zahl von Peers, die dabei simuliert werden muss, um wirklich aufschlussreiche Daten über die Grenzen der eingesetzten Algorithmen zu gewinnen. Daher wird auf eine zu detaillierte Simulation der unteren Netzwerkschichten verzichtet, um mit mehr Peers arbeiten zu können. Die vorliegende Arbeit greift dabei auf Methoden der diskreten Ereignis-Simulation [7] zurück, da sich die zwischen den Peers ausgetauschten Nachrichten in natürlicher Weise als Ereignisse, die den Systemzustand verändern, interpretieren lassen. Eine kurze Einführung in die diskrete Simulation und die in dieser Arbeit verwendeten Methoden folgt in Abschnitt 2.

1.2 Verwendete Software und Bibliotheken

Die in dieser Arbeit vorgestellte Simulationsumgebung verwendet die Java 2 Plattform in der Version 1.4.2 und die darin enthaltenen Bibliotheken [8], sowie eine Implementierung des Scalable Simulation Framework (SSF) [9] der Renesys Corporation in der Version 1.2b03 [10], verschiedene Bibliotheken für Wahrscheinlichkeitsverteilungen und statistische Auswertungen aus der Colt Distribution des CERN und Log4J in der Version 1.2.8. Eine kurze Beschreibung des Scalable Simulation Framework bzw. der verwendeten Implementierung findet sich in Abschnitt 3.

2 Diskrete Simulation

2.1 Definition Simulation

„Simulation ist die Nachbildung der Abläufe eines Prozesses oder Systems aus der realen Welt über der Zeit.“ (Übersetzt aus [7])

2.2 Modellbildung / Diskrete und kontinuierliche Simulation

Da die reale Welt im Allgemeinen zu komplex ist, um sie direkt zu simulieren, muss ein Modell des zu simulierenden Systems entwickelt werden, das sowohl einfach genug ist, um damit arbeiten zu können, als auch komplex genug, um alle im Hinblick auf die Zielsetzung der Simulation wichtigen Parameter des Systems abbilden zu können. Zu Anfang des Modellbildungsprozesses müssen daher die für das Modell wichtigen Teile des Systems gefunden und ihre Aktivitäten innerhalb des Systems untersucht werden, um sie im Modell in entsprechend vereinfachter Weise abzubilden. Es kann daher bei komplexen Systemen kein einzig richtiges Modell geben, sondern nur ein für die jeweils aktuelle Fragestellung passendes Modell. Das in dieser Arbeit verwendete Modell eines Peer-to-Peer bzw. Super-Peer-Netzwerkes wurde dem entsprechend im Hinblick auf den Schwerpunkt der Simulation der Routingalgorithmen entwickelt und im Netzwerkteil daher teilweise stark vereinfacht. Eine detaillierte Beschreibung des verwendeten Modells findet sich in Abschnitt 4.

Man unterscheidet grundsätzlich zwischen diskreten und kontinuierlichen Systemen. In diskreten Systemen ändert sich der Zustand des Systems zu diskreten Zeitpunkten, während sich in kontinuierlichen Systemen eine Zustandsänderung über einen kontinuierlichen Zeitraum vollzieht. Dementsprechend befasst sich die diskrete Simulation mit der Simulation diskreter Systeme. Da ein Peer-to-Peer Netzwerk seinen Zustand jeweils zu diskreten Zeitpunkten, nämlich mit dem Eintreffen bzw. Absenden von Nachrichten bzw. Netzwerkpaketen ändert, kann es mit den Mitteln der diskreten Simulation modelliert und simuliert werden.

Auch bei den Modellen unterscheidet man zwischen diskreten und kontinuierlichen Modellen, allerdings ist es unter Umständen durchaus möglich, ein diskretes System durch ein kontinuierliches Modell bzw. ein kontinuierliches System durch ein diskretes Modell abzubilden. Ein Modell, das zum Teil auf Zufallsvariablen beruht, bezeichnet man als stochastisches Modell, im Gegensatz zu einem deterministischen Modell, das keine Zufallsvariablen enthält. Da wie in Abschnitt 4 nachzulesen ist, dass das in dieser Arbeit entwickelte Modell eines Peer-to-Peer Netzwerkes Zufallsvariablen verwendet, handelt es sich also auch um ein stochastisches Modell.

2.3 Wichtige Konzepte der diskreten Simulation

In der diskreten Simulation besteht ein System aus Entitäten (engl.: Entities), in diesem Fall z.B. den Peers, die über einen Zeitraum zusammenarbeiten, um bestimmte Ziele zu erreichen. Der Zustand des Systems wird zu diskreten Zeitpunkten durch das Eintreffen von Ereignissen (engl.: Events) verändert. Die Ereignisse werden zusammen mit einem Zeitstempel in einer Ereignisliste gespeichert und in der durch diese Zeitstempel vorgegebenen Reihenfolge

verarbeitet. Auf diese Weise können Ereignisse auch zur Kommunikation zwischen Entitäten verwendet werden, indem eine Entität ein entsprechendes Ereignis erzeugt und in der Liste ablegt, welches dann zu einem späteren Zeitpunkt durch eine andere Entität verarbeitet wird und so den neuen Zustand der Entität bzw. des Systems festlegt.

Die grundlegenden Funktionen zur Verwaltung der Ereignisse, wie oben beschrieben, werden im Fall des dieser Arbeit zu Grunde liegenden Programms durch das Scalable Simulation Framework im Hintergrund ausgeführt. Daher befasst sich diese Arbeit hauptsächlich mit der Modellbildung und der Implementierung des Modells.

Weiterhin wird in der diskreten Simulation zwischen verschiedenen Weltansichten unterschieden. Die beiden wichtigsten stellen die ereignisorientierte Modellierung und die prozessorientierte Modellierung dar. Im Falle der ereignisorientierten Modellierung wird die Dynamik des Systems allein durch die Ereignisse modelliert, während die Entitäten im Wesentlichen passiv sind. Die prozessorientierte Modellierung dagegen ordnet den Entitäten eine Art Lebenszyklus zu und modelliert das System durch Suspendierung und Reaktivierung dieser Prozesse. Durch diese Zuordnung lassen sich insbesondere komplexe Systeme oftmals einfacher modellieren und überblicken. Interessant ist, dass die simulierte Zeit in einem prozessorientierten Modell durch die passiven Phasen eines Prozesses und nicht durch die aktiven vergeht. Eine eingehende Beschreibung beider Modellierungsarten findet sich z.B. in [7] und soll nicht Gegenstand dieser Arbeit sein. Wichtig ist nur festzuhalten, dass in dieser Arbeit eine Mischung aus beiden Modellierungsarten Verwendung findet, je nachdem welche Sichtweise für welche Aufgabe jeweils zweckmäßiger erscheint.

Wie oben schon angedeutet, ist es äußerst wichtig zwischen der simulierten Zeit und der realen Zeit zu unterscheiden. So nehmen zwar viele Berechnungen Rechen- und damit reale Zeit ein, aus Sicht der Simulation können diese aber durchaus gleichzeitig geschehen. Die simulierte Zeit startet bei einem bestimmten Zeitindex, z.B. „0“, und schreitet dann bis zu dem nächsten Zeitindex, der einem Ereignis in der Ereignisliste zugeordnet ist, voran. Das dem Zeitindex zugeordnete Ereignis wird dann entsprechend verarbeitet, woraus meist neue Ereignisse resultieren, die wiederum für die entsprechenden Zeitpunkte in der Ereignisliste vorgemerkt werden und die simulierte Zeit schreitet wiederum bis zum nächsten Ereignis voran.

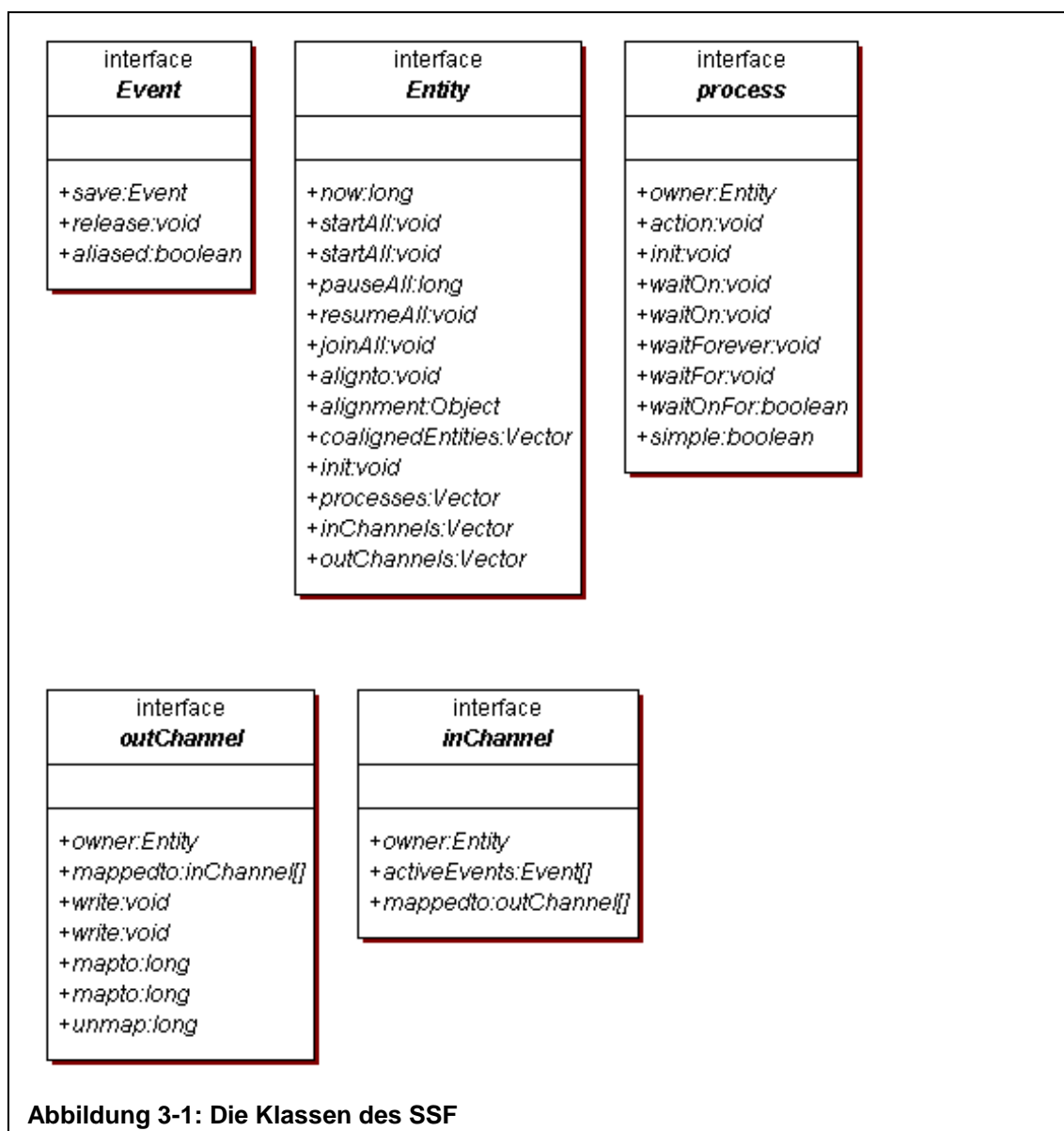
3 Das Scalable Simulation Framework (SSF)

Das Scalable Simulation Framework ist ein offener Standard für die diskrete Ereignis-Simulation. Er spezifiziert APIs für Java und C++, auf denen Simulationen bzw. ganze Frameworks aufsetzen können, wie z.B. die SSF Network Models [9]. Das Scalable Simulation Framework ist so angelegt, dass darauf basierende Simulationen implementierungsunabhängig aufgebaut

werden können. Dabei deckt das Framework alle grundlegenden Funktionen ab, die zur diskreten Ereignis-Simulation benötigt werden (siehe Abschnitt 2.3). Für die in dieser Arbeit beschriebene Simulation wurde eine Implementierung der Firma Renesys verwendet, die sich durch hohe Performance und Unterstützung für Multiprozessor-Systeme auszeichnet [10].

In den nächsten Abschnitten dieses Kapitels folgt eine kurze Übersicht über die Spezifikationen des Scalable Simulation Framework und die Semantik der wichtigsten Methoden und Klassen. Eine eingehende Erläuterung aller Klassen findet sich in [11]. Details zur Modellierung mit diesen Klassen finden sich in den entsprechenden Abschnitten zum Modell (siehe Abschnitt 4) und zur Implementierung (siehe Abschnitt 5).

3.1 Die Klassen des SSF



3.1.1 Entity

Die *Entity* im SSF entspricht im Wesentlichen einer Entität der diskreten Simulation (siehe Abschnitt 2.3), d.h. einer *Entity* können Prozesse zugeordnet werden, mittels derer sie Ereignisse verarbeiten kann. Die *Entity* ist die Zentrale Klasse einer SSF basierten Simulation. Sie verfügt über die entsprechenden Methoden zum Starten und Stoppen einer Simulation. Durch Ableiten von dieser Klasse werden die entsprechenden Entitäten des Modells implementiert.

3.1.2 Event

Ein *Event* ist nichts anderes als ein Ereignis, das den Zustand des Systems verändert und auch zur Kommunikation zwischen Entitäten dienen kann. Allgemein gilt, dass gerade im Hinblick auf die Nebenläufigkeit von verschiedenen Prozessen jegliche Zustandsänderung über Ereignisse und nicht direkt erfolgen sollte, da alleine für Ereignisse Garantien in Bezug auf Verarbeitungsreihenfolge usw. durch das Framework gegeben sind. So sind sämtliche Nachrichten, die verschiedene Entitäten austauschen, durch Ableiten von *Event* zu implementieren.

3.1.3 process

Ein *process* entspricht einem Prozess der prozessorientierten Modellierung (siehe Abschnitt 2.3). Einer *Entity* können mehrere Prozesse zugeordnet werden. Durch die verschiedenen *wait()*-Methoden kann sich ein Prozess selbst suspendieren, um simulierte Zeit verstreichen zu lassen. Um das Verhalten von Prozessen zu implementieren, muss die *action()*-Methode entsprechend überschrieben werden.

3.1.4 inChannel / outChannel

Diese Klassen stellen die Kommunikationskanäle zwischen den Entitäten dar. Mittels der Methode *mapto()* kann jeweils ein *outChannel* mit einem *inChannel* verbunden werden, d.h. Ereignisse die in den *outChannel* geschrieben werden, können am verbundenen *inChannel* abgerufen und verarbeitet werden. In einen *outChannel* werden Ereignisse, d.h. von *Event* abgeleitete Klassen, mittels der *write()*-Methode geschrieben, wobei optional auch noch eine Zeitverzögerung in simulierter Zeit spezifiziert werden kann. Durch einen Aufruf der Methode

activeEvents() können alle am *inChannel* wartenden Ereignisse abgerufen werden. In Verbindung mit den verschiedenen *wait()*-Methoden eines *process* kann so auf eingehende Ereignisse gewartet und können diese dann verarbeitet werden, um den neuen Systemzustand zu erreichen. Eine *Entity* kann über mehrere eingehende und ausgehende Kanäle mit beliebigen Verbindungen verfügen.

4 Das Modell

Im nachfolgenden Abschnitt wird das der in dieser Arbeit entwickelten Simulationsumgebung zugrunde liegende Modell näher erläutert. Das Modell eines Peer-to-Peer bzw. Super-Peer-Netzwerkes lässt sich in verschiedene Teilmodelle zerlegen, auf die unten im Einzelnen eingegangen wird. Grundsätzlich lässt sich dabei zu allererst eine Unterteilung in die Modellierung von eher technischen Aspekten, also z.B. von Netzwerkkomponenten sowie von Verhaltensaspekten, also z.B. die Verarbeitung und die Reaktion der Peers auf bestimmte Nachrichten, und natürlich die Modellierung der äußeren Einflüsse, wie z.B. Benutzerinteraktion, treffen. Weiterhin muss natürlich auch die Anordnung der Peers im Netzwerk, also die Netzwerk-Topologie, und die Verteilung von Information, hier modelliert werden.

4.1 Modellierung der Schemainformation

In dem Modell eines schema-basierten Super-Peer-Netzwerkes spielt natürlich die Modellierung der Schemainformation eine wichtige Rolle. Ein Schema ist in diesem Zusammenhang als eine Menge von Attributen oder Properties zu verstehen. Das heißt, jedem Schema werden bestimmte Properties zugeordnet und jede Property gehört auf der anderen Seite zu genau einem Schema. Welcher Standard dafür in der realen Anwendung Verwendung findet, spielt an dieser Stelle eine eher untergeordnete Rolle, da sich das Modell in dieser Arbeit auf einige grundsätzliche Eigenschaften von Schemadefinitionen bezieht, die den meisten Standards gemeinsam sind. Weiterführende Fähigkeiten einiger Beschreibungssprachen, wie z.B. Vererbung, werden daher an diesem Modell nicht beachtet. Das Ziel ist hier zunächst eine möglichst einfache Modellierung von Schemas und Properties, um eine effiziente Simulation zur Gewinnung von grundlegenden Daten über schema-basierte Peer-to-Peer Netzwerke zu ermöglichen, wobei das Modell und die Implementierung sich später um weitere Fähigkeiten erweitern lassen sollten.

Schemainformationen sind in einem schema-basierten Super-Peer-Netzwerk hauptsächlich an drei Stellen wichtig. Die Bereitsteller von Informationen, also z.B. Dokumenten, so genannte „Provider“, benutzen Schemainformationen, um

eine Zusammenfassung ihrer „Fähigkeiten“ zu geben. Hier ist es wichtig zwischen diesen beiden Informationsarten zu unterscheiden, also zum einen der Information, die in Dokumenten abgelegt ist, und der Schemainformation, die eine Metainformation über die Dokumente darstellt, z.B. Properties wie Titel, Autor oder ähnliches (siehe z.B. [12]). Provider geben also eine Beschreibung ihrer „Fähigkeiten“ ab, indem sie eine Zusammenfassung über die in ihrer Dokumentenbasis verwendeten Schemas bzw. Properties erstellen und weitergeben. Mit „Fähigkeiten“ ist in diesem Zusammenhang also die prinzipielle Fähigkeit zur Beantwortung von Anfragen nach bestimmten durch Schemas klassifizierten Informationen gemeint, die zweite wichtige Stelle, an der Schemainformationen in einem schema-basierten Peer-to-Peer Netzwerk verwendet werden. In solchen Anfragen oder Queries von Peers, die nach Informationen suchen, so genannten „Consumern“, wird also nach bestimmten Properties oder Inhalten bestimmter Properties gefragt, welche die Provider dann mit ihrer Informationsbasis abgleichen und gegebenenfalls darauf antworten. Wie in [4] aufgezeigt, lassen sich solche Anfragen in unterschiedlichen Feinheitsgraden stellen. In diesem Modell wird allerdings nur der Grad der Properties modelliert, da sich die anderen Feinheitsgrade, im Hinblick auf das Routing kaum voneinander unterscheiden, aber wesentlich aufwendiger zu modellieren wären. Das Routing ist der dritte wichtige Punkt, an dem Schemainformationen Verwendung finden und im Zusammenhang dieser Arbeit auch der wichtigste, da das Hauptziel die Simulation verschiedener Routing bzw. Suchalgorithmen in einem Super-Peer-Netzwerk ist. Das Routing ist Aufgabe der Super-Peers, die die Schemainformationen über die Provider verwenden, um ein möglichst effizientes Routing von Anfragen zu gewährleisten und so die Gesamtperformance des Netzwerkes zu erhöhen. Ziel ist also, ein Modell für Schemainformation zu finden, das den oben genannten Anforderungen und Verwendungsmöglichkeiten genügt. Um die oben genannten Anforderungen unter der Beschränkung auf die Ebene der Properties zu erfüllen, müssen sowohl Schemas als auch Properties unterscheidbar sein und es muss eine Zuordnung zwischen Schemas und Properties getroffen werden können. Ein Schema wird daher hier durch eine numerische Id und eine Menge von Properties modelliert, eine Property wiederum durch eine numerische Id und eine Referenz auf das zugehörige Schema.

Des Weiteren muss natürlich auch die Verteilung der Schemas und Properties auf Provider und Queries modelliert werden. Die durch Untersuchungen von Webseiten Zugriffsstatistiken bekannt gewordene Zipf- oder auch Zeta-Verteilung [13] erscheint auch für die Verteilung der Schemas und Properties geeignet, da einige wenige bekannte Schemas, z.B. Dublin Core, oft verwendet und demnach auch nachgefragt werden, während viele Schemas nur von einer kleinen Nutzergruppe verwendet und demnach entsprechend wenig nachgefragt werden. Trotzdem muss darauf hingewiesen werden, dass dem Autor zum Zeitpunkt dieser Arbeit leider keine repräsentative Untersuchung über die Nutzung von Schemas bekannt ist, die hier weitere Anhaltspunkte liefern würde, dennoch erscheint eine Zipf-Verteilung zumindest nicht unplausibel und wird daher an dieser Stelle zur Modellierung der Schema-Verteilung verwendet.

4.2 Modellierung der Netzwerkverbindungen

Da es Ziel dieser Arbeit ist, Routing- bzw. Suchalgorithmen möglichst unabhängig von der zu Grunde liegenden Netzwerkarchitektur zu simulieren, ist das verwendete Modell des Netzwerkes ein stark vereinfachtes. So werden lediglich einige wesentliche Parameter zur Beschreibung einer Netzwerkverbindung in das Modell übernommen. In diesem Modell wird eine Netzwerkverbindung durch die Verzögerung und die Bandbreite beschrieben. Weiterhin wird angenommen, dass die Verteilung dieser Parameter auf die einzelnen Verbindungen annähernd einer Normalverteilung entspricht, mit einem Erwartungswert und einer Standardabweichung, die vom Benutzer variabel vorgegeben werden können. Als kleinste Einheiten, auf die sich Verzögerung und Bandbreite beziehen, dienen hier die einzelnen Nachrichten, die zwischen den Peers versendet werden, auf eine Simulation der tieferen Netzwerkschichten, z.B. der IP-Pakete, wird verzichtet. Die Nachrichten können dabei direkt als spezielle Ereignisse abgebildet werden.

4.3 Modellierung der Peers

Alle verschiedenen Arten von Peers können direkt als Entitäten abgebildet werden, die Netzwerknachrichten in Form von Ereignissen erhalten. Details zur Implementierung finden sich in Abschnitt 5.3. In den folgenden Unterabschnitten werden die Besonderheiten der verschiedenen Klassen von Peers aufgezählt. Es werden 3 Klassen unterschieden: Consumer, die Information nachfragen, Provider, die Information bereitstellen, sowie Super-Peers, die Routing- und Forwarding-Aufgaben übernehmen. Bei realen Netzwerken ist außerdem oftmals der Fall anzutreffen, dass einige Peers, sowohl Consumer als auch Provider sind. Die Trennung dieser beiden Arten von Peers in der Simulation hat den Vorteil, dass das Verhältnis zwischen nach Information suchenden und Information bereitstellenden Peers sehr fein modelliert werden kann. Trotzdem erlaubt das in dieser Arbeit entwickelte Framework auch hybride Peers.

4.3.1 Die Consumer-Peers

Der Consumer-Peer ist der am einfachsten zu modellierende Peer-Typ. Seine einzige Aufgabe besteht darin, in regelmäßigen Abständen Anfragen an das Netzwerk zu stellen. Modelliert werden diese Anfragen durch Folgen von sog. „Querybursts“. Das heißt in bestimmten Intervallen werden in relativ kurzen Zeitabständen mehrere Anfragen abgeschickt. Die Intervalllängen und die

Anzahl der Anfragen sind dabei frei vom Benutzer der Simulation wählbar. Außerdem ist jedem Consumer-Peer eine Lebenszeit zugeordnet, nach deren Ablauf er „stirbt“, d.h. aus dem Netzwerk entfernt wird, um wechselnde Consumer zu simulieren.

4.3.2 Die Provider-Peers

Provider-Peers sind die Informationsbereitsteller. Die eigentliche Information wird dabei allerdings nicht modelliert, sondern lediglich die Zusammenfassung der Metainformationen, also die Schemas und Properties, zu denen ein Provider Informationen bereitstellen kann. Ob eine Query dann tatsächlich beantwortet wird, wird durch die Vorgabe einer Wahrscheinlichkeit simuliert. Auch einem Provider wird eine bestimmte Lebenszeit zugeordnet, nach deren Ablauf er das Netzwerk verlässt.

4.3.3 Die Super-Peers

Super-Peers bilden das „Rückgrat“ des Netzwerkes, d.h. sie sind für die Vermittlung der Nachrichten zwischen den Peers zuständig. In einem schema-basierten Super-Peer-Netzwerk hängen die Wege der Nachrichten sowohl von den enthaltenen Schemainformationen als auch von der grundsätzlichen Topologie des Netzwerkes ab. Die Schemainformation wird dabei durch spezielle Indices in den Super-Peers ausgewertet, während die Topologie implizit durch die Art, wie sich die Super-Peers miteinander verbinden, vorgegeben wird. In dem vorliegenden Modell wird zwischen diesen beiden Einflüssen deutlich unterschieden. Die Topologie wird hierbei als eigenständiges Konzept modelliert, wodurch verschiedene Topologien leicht gegeneinander ausgetauscht werden können, ohne am Modell der Super-Peers etwas zu verändern. Genaueres dazu findet sich im Abschnitt 4.4. Auch das Verhalten der Schemaindices kann leicht durch verschiedene Implementierungen ausgetauscht werden. Näheres dazu findet sich in den Abschnitten über die Implementierung (siehe Abschnitt 5). Da für reale Super-Peers angenommen werden kann, dass die Verbindungen zwischen den Super-Peers und deren Anzahl relativ statisch sind, wird das Modell dahingehend vereinfacht, dass sowohl die Zahl als auch die Konfiguration, d.h. die Topologie, der Super-Peers als für die Dauer der Simulation fest angenommen wird.

4.4 Modellierung der Topologie

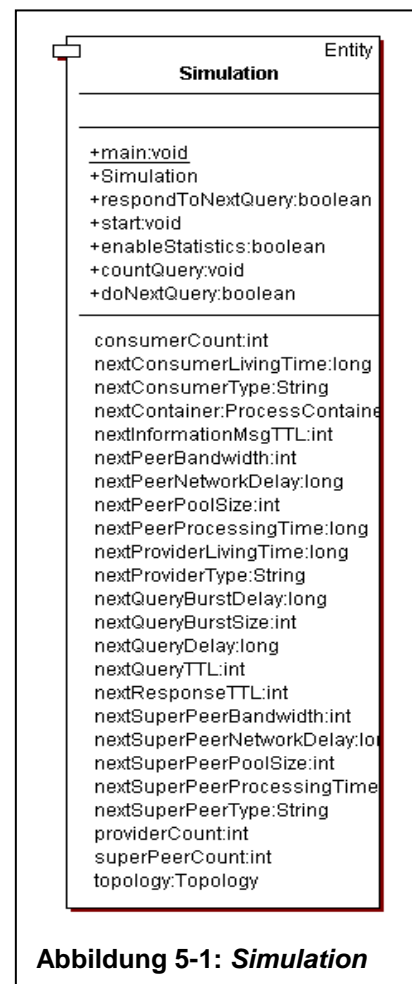
Die Topologie des Super-Peer-Netzwerkes bestimmt die Art, auf die die Super-Peers untereinander verbunden sind. Hier wird die Topologie daher direkt durch die Wege abgebildet, auf denen die Nachrichten, die wie bereits ausgeführt nichts anderes als spezielle Ereignisse der diskreten Ereignissimulation sind, zu den Super-Peers gelangen bzw. zwischen diesen ausgetauscht werden. Details zum konkreten Aufbau der Topologien in der Simulation finden sich in den entsprechenden Abschnitten 5.4.1 und 5.4.2.

5 Die Implementierung

In den folgenden Abschnitten dieses Kapitels werden die Details der Implementierung, der dieser Arbeit zugrunde liegenden Simulationsumgebung für schema-basierte Super-Peer-Netzwerke, beschrieben. Ein Überblick über die dazugehörige API kann über die Seiten des Edutella-Projektes [2] herunter geladen oder unter [16] online eingesehen werden.

5.1 Die wichtigsten Klassen und ihre Aufgaben im Überblick

Hier folgt nun ein kurzer Überblick über die zentralen Klassen und Interfaces der Simulationsumgebung. Die Details zur Implementierung der Konfiguration, Peers, Super-Peers und der Topologien sowie der Gewinnung der statistischen Daten und der Ausgabe folgen dann jeweils in eigenen Unterabschnitten.



5.1.1 Simulation

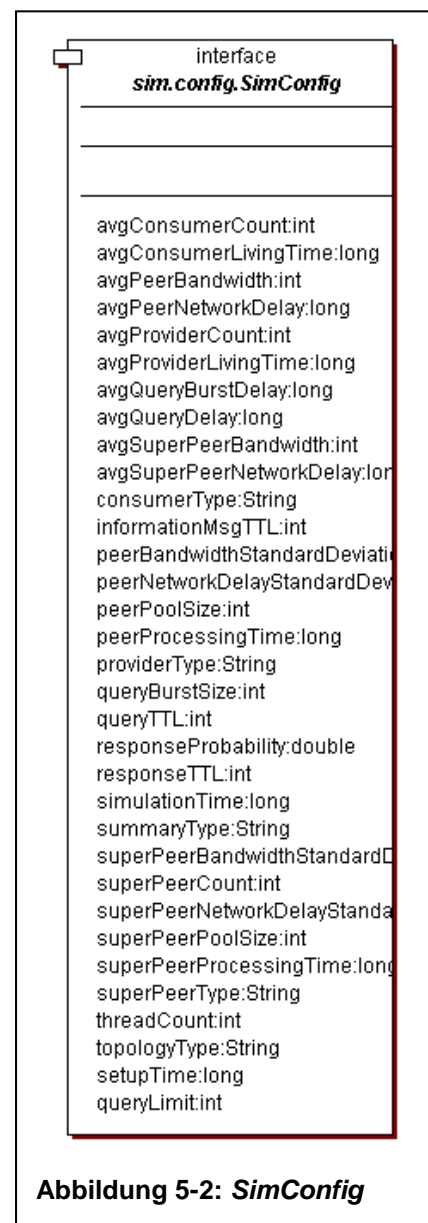
Die Klasse *Simulation* ist die zentrale Klasse der Simulationsanwendung. Mit Hilfe dieser Klasse wird die Konfiguration der Simulation geladen, die Simulation in den Anfangszustand versetzt und gestartet. In dieser Klasse werden auch die Wahrscheinlichkeitsverteilungen für Bandbreite und Verzögerung der Netzwerkverbindungen von Peers und Super-Peers initialisiert und über entsprechende „get-Methoden“ zur Verfügung gestellt. *Simulation* ist von der SSF Klasse *Entity* abgeleitet, um Zugriff auf wichtige Methoden zur Steuerung der Simulation zu haben. Vor dem Start eines Simulationslaufes wird durch diese Klasse die Topologie der Super-Peers mit Hilfe spezieller Factory-Klassen erzeugt und initialisiert und eine durch die Konfiguration festgelegte Anzahl von *ProcessContainern* für die Aufnahme von Peers und Super-Peers bereitgehalten.

5.1.2 SimConfig

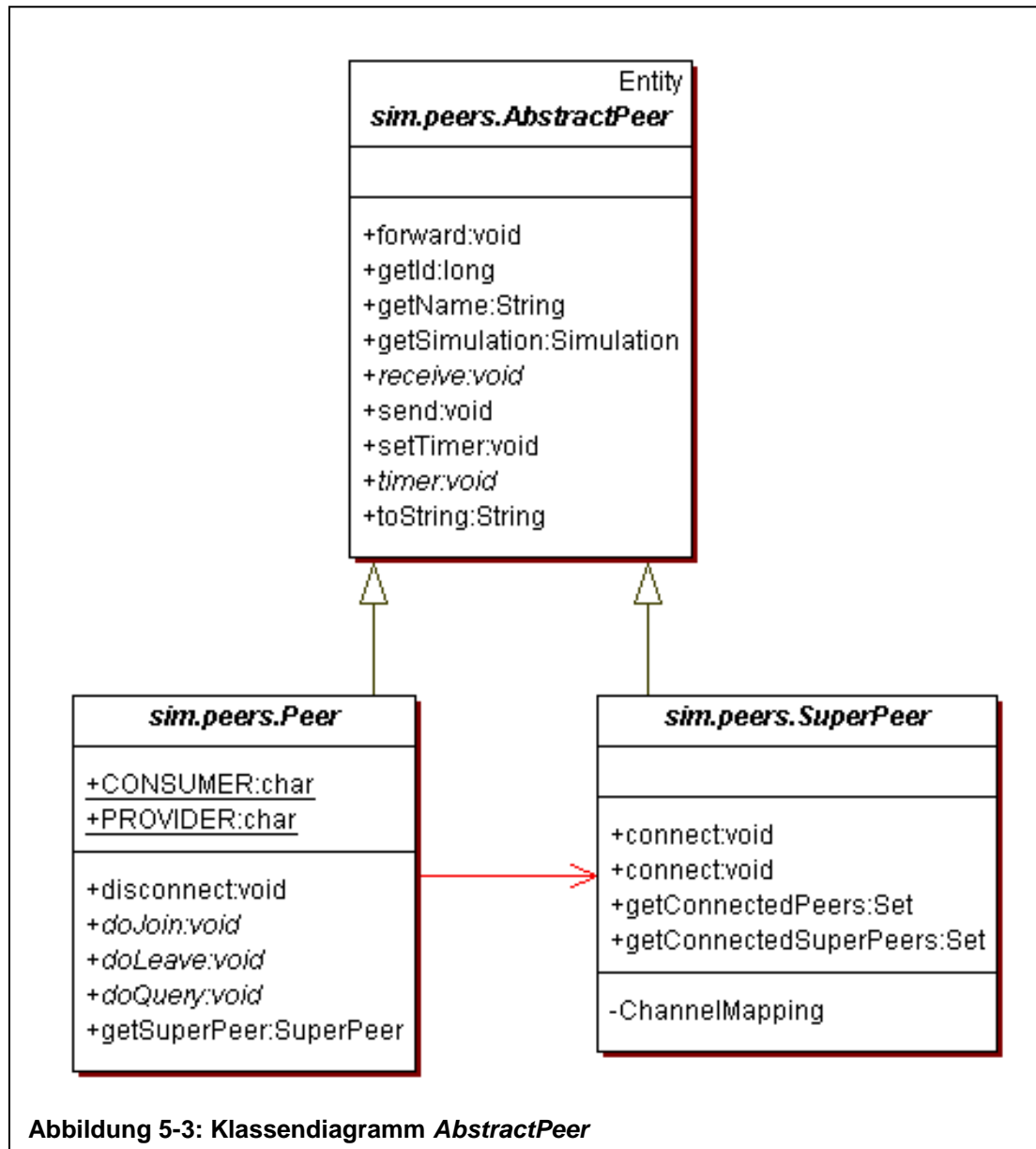
Das Interface *SimConfig* definiert die Methoden zum Zugriff auf die Konfiguration einer Simulation. Durch diese Kapselung können leicht verschiedene Implementierungen von Konfigurationsobjekten für verschiedene Konfigurationsdateiformate gegeneinander ausgetauscht werden. Für diese Arbeit wurden zwei Implementierungen dieses Interfaces entwickelt. Die erste ist zu Testzwecken gedacht und arbeitet ohne ein Konfigurationsfile, die zweite verwendet ein für diese Arbeit entwickeltes XML-Dateiformat. Das im Abschnitt 5.2 beschrieben wird.

5.1.3 AbstractPeer / SuperPeer / Peer

Die Klasse *AbstractPeer* stellt zusammen mit den abgeleiteten Klassen *SuperPeer* und *Peer* das Grundgerüst für verschiedene Super-Peer bzw. Peer Implementierungen zur Verfügung. In diesen Klassen wird das darunter liegende Modell gekapselt, so dass sich die Implementierung konkreter Peers völlig auf das Verhalten dieser Peers konzentrieren kann.



Dazu muss lediglich eine Klasse von *Peer* oder *SuperPeer* abgeleitet und die entsprechenden abstrakten Methoden implementiert werden. Außerdem stellen diese Klassen Methoden zum Senden und Weiterleiten von Nachrichten und einen Timer-Mechanismus bereit. Die Verwendung dieser Klassen wird im Zusammenhang mit der Vorstellung der in dieser Arbeit entwickelten Implementierungen für schema-basierte Peers und Super-Peers erklärt. Die Klasse *AbstractPeer* als Basisklasse aller Peers ist von *Entity* abgeleitet, alle Peers stellen also Entitäten im Sinne der diskreten Ereignissimulation dar.



5.1.4 ProcessContainer

ProcessContainer ist eine Klasse, die keine direkte Entsprechung im theoretischen Modell hat, sondern aus Performance Gründen eingeführt wurde. Wie in den vorherigen Kapiteln erklärt, verfügt eine Entität in der Regel über einen oder mehrer Prozesse für die Verarbeitung von Ereignissen. Solche Prozesse werden im SSF Framework auf Java Threads abgebildet. Da jeder Peer mindestens einen, in der vorliegenden Implementierung sogar mehrere, Prozesse zur Verarbeitung der Nachrichten benötigt, würde die Simulation schon bei einer relativ kleinen Anzahl zu simulierender Peers aufgrund von Speicher- und Betriebssystemgrenzen nicht mehr in der Lage sein, genügend Threads zu erstellen, wenn tatsächlich jedem Peer ein exklusiver Prozess zugeordnet werden würde. Daher wird in dieser Arbeit ein anderer Weg beschritten. Jeder *ProcessContainer* stellt für eine ganze Gruppe von Peers die benötigten Prozesse zur Verfügung. Die Anzahl der verwendeten Gruppen kann dabei vom Benutzer in der Konfiguration vorgegeben werden. Mittels spezieller interner Events werden Peer exklusive Prozesse simuliert. Der genaue Ablauf wird in Abschnitt 5.3 erklärt.

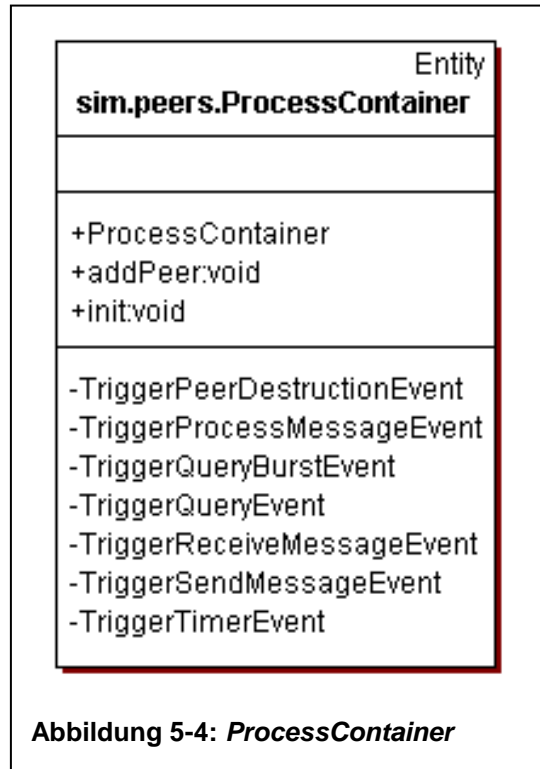
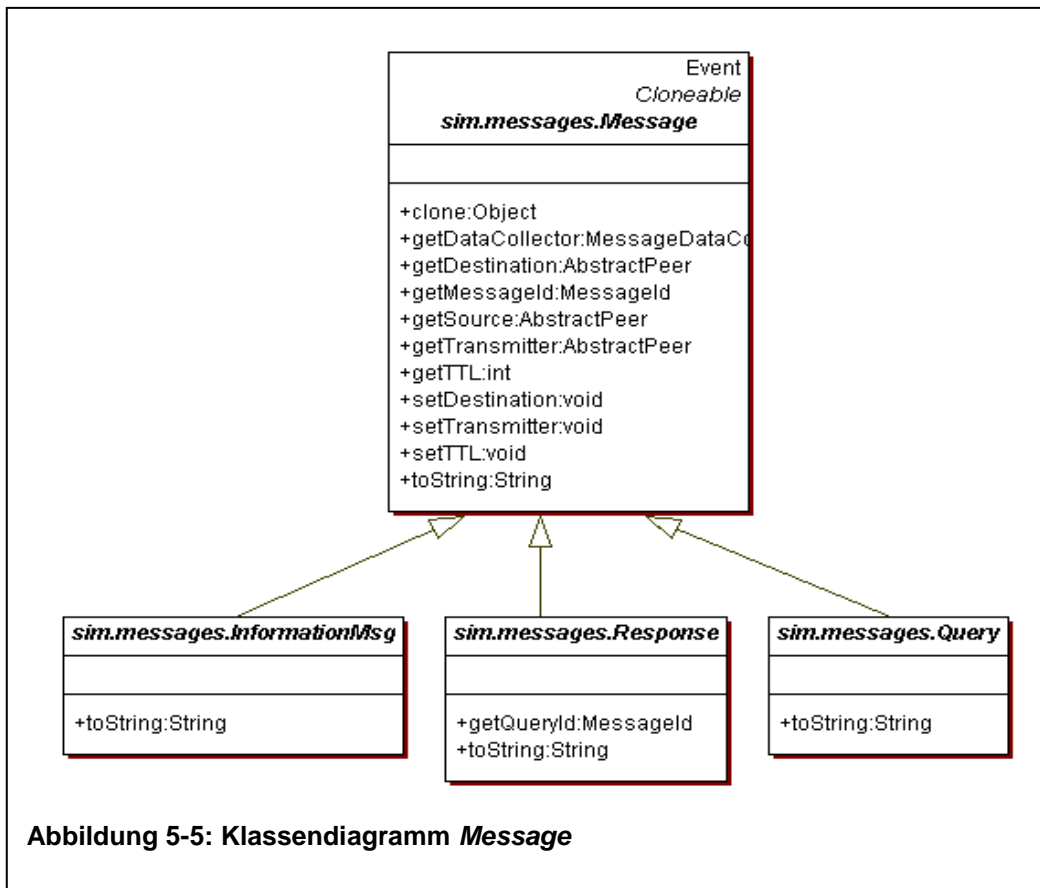


Abbildung 5-4: *ProcessContainer*

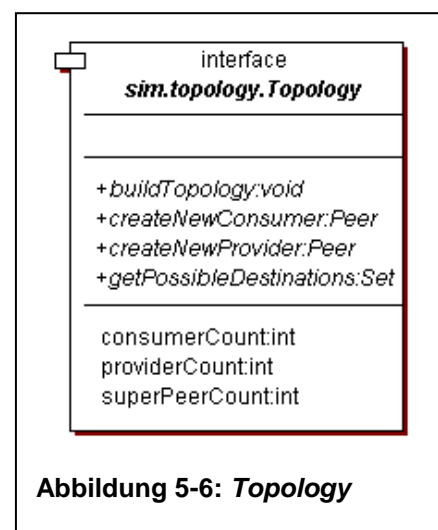
5.1.5 Message / InformationMsg / Query / Response

Die Klasse *Message* ist die Basisklasse für alle in der Simulation zwischen den Peers ausgetauschten Nachrichten. Sie ist von *Event* abgeleitet. Mit Hilfe der von *Message* abgeleiteten Klassen *InformationMsg*, *Query* und *Response* werden die Nachrichten grob in drei Klassen unterteilt. Nachrichten, die der Informationsweitergabe zwischen den Peers dienen, also im Allgemeinen Steuernachrichten, wie z.B. Join- und Leave-Nachrichten, werden von *InformationMsg*, Anfragen werden von *Query* und Antworten auf Anfragen werden von *Response* abgeleitet. Alle drei Klassen sind abstrakt, da es Aufgabe der jeweiligen Implementierung ist, den speziellen Erfordernissen angepasste Nachrichten-Typen zu definieren.



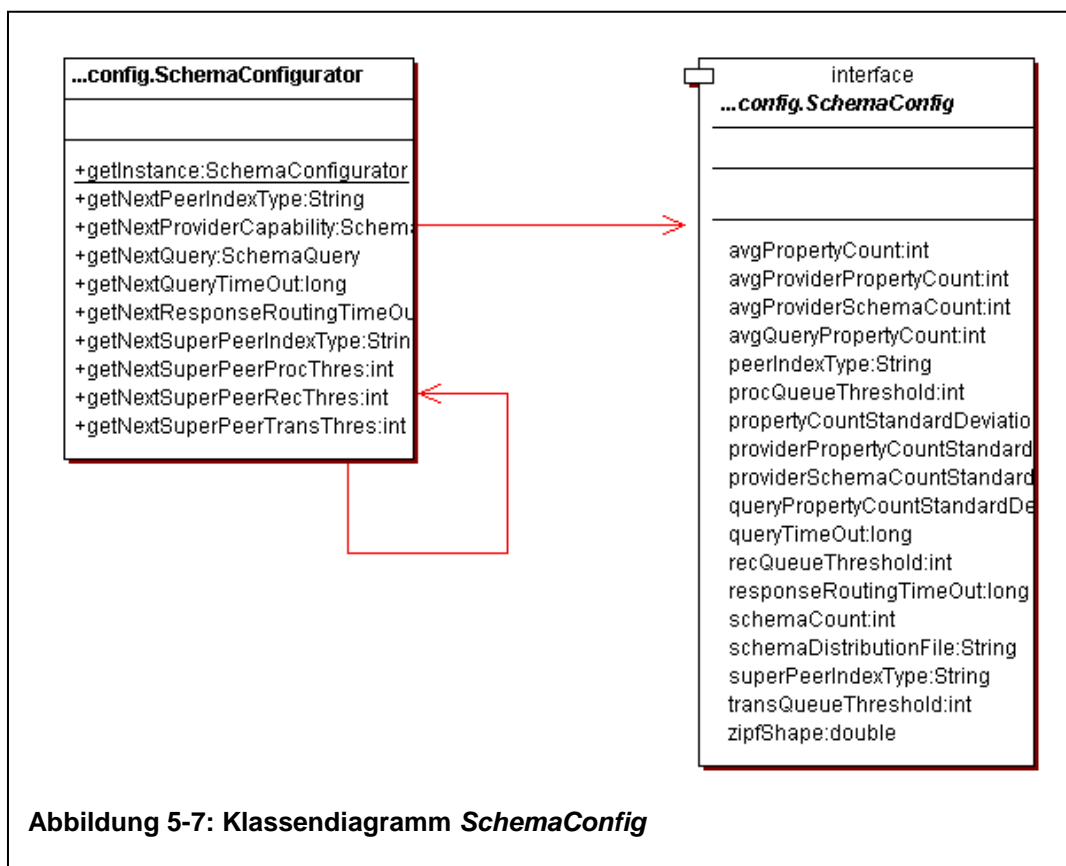
5.1.6 Topology

Das Interface *Topology* definiert die Schnittstelle zu verschiedenen Topologie-Modellen. Eine Topologie-Implementierung besitzt Methoden zum Aufbau der Super-Peer-Topologie und zur Integration von Consumer- und Provider-Peers. Wie im Abschnitt über das Modell beschrieben, beeinflusst die Topologie ähnlich einem Filter den Weg von Nachrichten durch das Super-Peer-Netzwerk. Diese in der Realität der Implementierung der Super-Peers inhärente Filterung wird im Modell in die Topologie-Klassen verlagert, um verschiedene Topologien besser gegeneinander austauschbar zu machen. In dieser Arbeit wurden beispielhaft zwei Topologien implementiert. Eine einfache Broadcast-Topologie und eine auf dem in [14] vorgestellten Hypercube-Protokoll basierende. Näheres dazu findet sich in den Abschnitten 5.4.1 und 5.4.2.



5.1.7 SchemaConfig / SchemaConfigurator

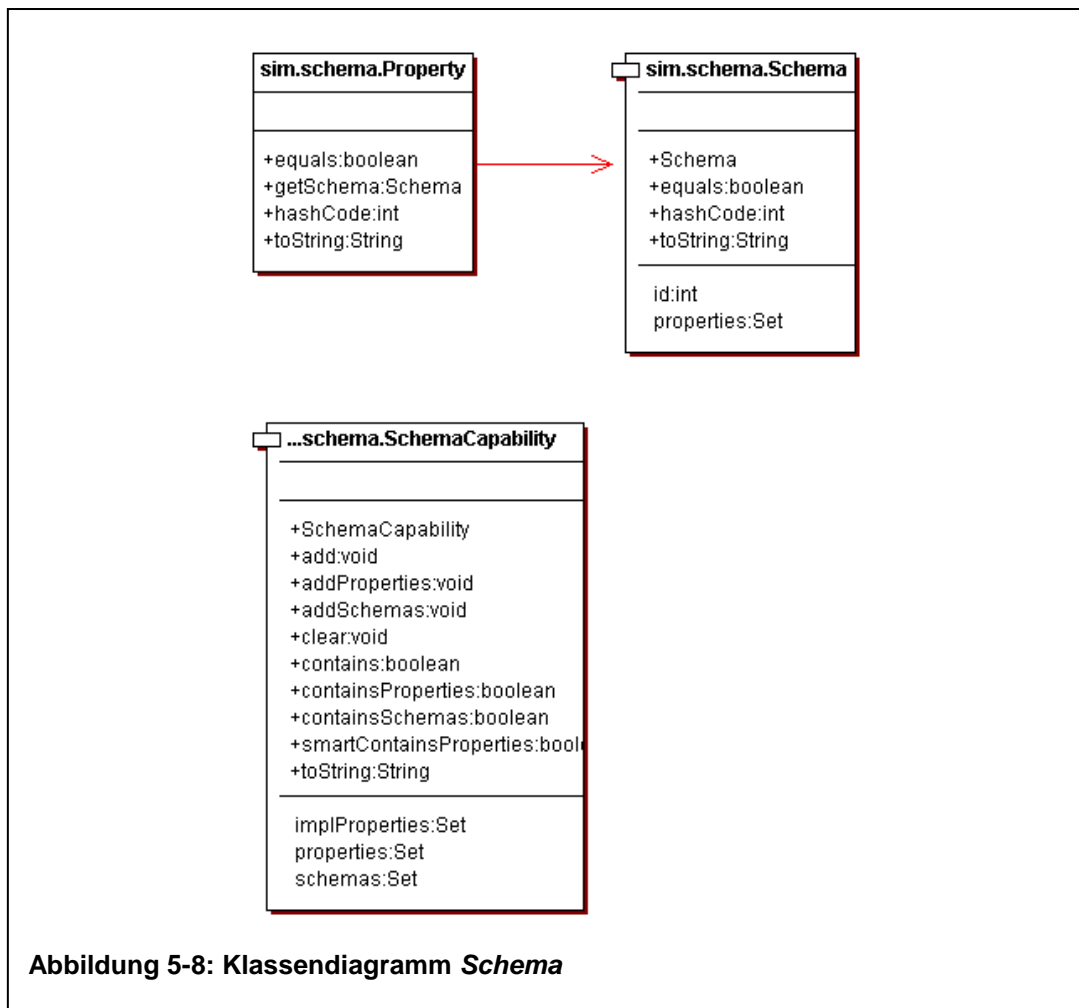
Die Klasse *SchemaConfigurator* ist die zentrale Klasse zur Konfiguration des schema-basierten Teils der Simulation. Während über die Klasse *Simulation* in Verbindung mit dem Interface *SimConfig* bzw. der verschiedenen Implementierungen dieses Interfaces die allgemeinen Simulationsparameter festgelegt werden, die unabhängig von der Implementierung der Peers sind, erfüllt *SchemaConfigurator* in Verbindung mit Implementierungen des Interfaces *SchemaConfig* diese Aufgabe für Parameter, die speziell für die schema-basierte Implementierung der Peers benötigt werden. Ein *SchemaConfig*-Objekt



dient dabei als so genanntes *Data-Access-Object*, um verschiedene Dateiformate zu kapseln. In dieser Arbeit wurde eine Testimplementierung, die ohne Konfigurationsdatei arbeitet, sowie eine mit einem XML-Dateiformat arbeitende Implementierung entwickelt, welche genauer in Abschnitt 5.2 beschrieben sind. Die Wahrscheinlichkeitsverteilungen für die Schemas und Properties werden ebenfalls in der Klasse *SchemaConfigurator* berechnet. Um die Vergleichbarkeit von verschiedenen Simulationsläufen zu gewährleisten, kann diese Verteilung auch in einem XML-Format abgelegt und zwischen den verschiedenen Läufen gespeichert werden.

5.1.8 Schema / Property / SchemaCapability

Die Klassen *Schema* und *Property* implementieren direkt das in Kapitel 4.1 vorgestellte Modell der Schemainformationen. Die Klasse *SchemaCapability* fasst die Schemas und Properties eines Providers bzw. einer Query zusammen und stellt Methoden bereit, um verschiedene Vergleiche durchzuführen. Näheres dazu findet sich im Abschnitt 5.3.



5.2 Die Konfiguration der Simulation

Die Konfiguration der Simulation erfolgt durch die drei Dateien „simulation.xml“, „schema.xml“ und „distribution.xml“.

5.2.1 Aufbau der Datei „simulation.xml“

Die Datei „simulation.xml“ ist die Hauptkonfigurationsdatei der Simulationsumgebung. Die zugehörige DTD findet sich in der Datei „simulation.dtd“ (siehe Anhang). Das Root-Element *simulation* enthält drei Unterabschnitte. Im Abschnitt *topology* werden die Topologie-Parameter konfiguriert, im Abschnitt *superpeers* alle Parameter der Super-Peers, im Abschnitt *peers* die Parameter für Consumer und Provider und schließlich folgt noch ein Abschnitt *summary*, in dem die statistische Auswertung bzw. die Zusammenfassung konfiguriert werden kann. Das Element *simulation* enthält die Attribute *setuptime*, *simtime* und *threadcount*. Durch den Parameter *setuptime* kann eine Vorlaufzeit in Millisekunden angegeben werden, *simtime* spezifiziert dann die eigentliche Simulationsdauer, ebenfalls in Millisekunden.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE simulation SYSTEM "simulation.dtd">
<simulation setuptime="1000000" simtime="100000" threadcount="10">
  <topology type="sim.topology.HyperCubeTopology">
    <TTL informationmsg="-1" query="-1" response="-1"/>
  </topology>
  <superpeers type="sim.schema.peers.SchemaBasedSuperPeer"
    count="100">
    <network>
      <delay avg="100" deviation="1.0"/>
      <bandwidth avg="1000" deviation="1.0"/>
    </network>
    <processing poolsize="10" time="10"/>
  </superpeers>
  <peers>
    <network>
      <delay avg="100" deviation="1.0"/>
      <bandwidth avg="100" deviation="1.0"/>
    </network>
    <processing poolsize="5" time="10"/>
    <consumer type="sim.schema.peers.SchemaBasedConsumerPeer"
      count="10000" livingtime="400000">
      <query limit="1000" burstdelay="100000" burstsize="2"
        querydelay="30000"/>
    </consumer>
    <provider type="sim.schema.peers.SchemaBasedProviderPeer"
      count="10" livingtime="800000">
      <response probability="0.3"/>
    </provider>
  </peers>
  <summary type="sim.schema.statistic.SchemaBasedSummary"/>
</simulation>
```

Abbildung 5-9: Ausschnitt aus „simulation.xml“

Die gesamte Simulationszeit ergibt sich dann aus der *setuptime* zuzüglich der *simtime*. Während der Vorlaufzeit werden keinerlei statistische Daten erhoben und auch keine Anfragen gestellt. Auf diese Weise werden die Statistiken nicht durch nur am Anfang der Simulation auftretende Nachrichten verfälscht und es kann der eingeschwungene Zustand gemessen werden. Wie der Parameter *setuptime* genau zu bemessen ist, hängt von der Anzahl der Peers und Randbedingungen wie z.B. der Netzwerkverzögerung ab. Der Parameter *threadcount* gibt die Anzahl der für die Simulation zu verwendenden *ProcessContainer* bzw. Java Threads an. Das *topology* Element enthält ein Attribut *type*, welches die Java-Klasse spezifiziert, in der die Topologie implementiert ist. Es enthält einen optionalen Unterabschnitt *TTL*, in dem gegebenenfalls getrennt nach Nachrichtenkategorie eine *Time-to-Live* für die Nachrichten vorgegeben werden kann. Je nach verwendeter Topologie sollten solche TTLs angegeben werden oder auch nicht. Während z.B. im Falle der Hypercube-Topologie TTLs keinen Sinn machen, sind sie im Zusammenhang mit der Broadcast-Topologie durchaus sinnvoll einzusetzen. Wird der Abschnitt *TTL* weggelassen, werden keine TTLs verwendet. Durch die Angabe einer TTL kleiner als Null, ist es außerdem möglich, die TTLs gezielt nur für einzelne Kategorien auszuschalten. Das Element *superpeers* enthält auch wieder ein Attribut *type*, welches die zu verwendende Super-Peer-Implementierung spezifiziert und außerdem ein Attribut *count*, um die Anzahl der Super-Peers einzustellen. Zu beachten ist hierbei, dass diese Anzahl je nach Topologie-Implementierung eventuell durch die Simulation modifiziert wird, um die Topologie effizienter simulieren zu können. Die Unterabschnitte *network* und *processing* definieren wichtige Netzwerkparameter bzw. Parameter, die zur Simulation der Verarbeitungsgeschwindigkeiten von Peers und Super-Peers notwendig sind. Sowohl die *superpeers* als auch *peers* enthalten diese Elemente, um unterschiedliche Parameter für Peers und Super-Peers konfigurieren zu können. Eine Netzwerkverbindung wird durch vier Parameter bestimmt: Die durchschnittliche Netzwerkverzögerung in Millisekunden, spezifiziert im Element *delay* mittels des Attributs *avg*, die durchschnittliche Netzwerkbandbreite in Nachrichten pro Sekunde, spezifiziert im Element *bandwidth* mittels des Attributs *avg*, sowie die Standardabweichung für beide Parameter, jeweils definiert durch *deviation*. Im Element *processing* wird durch das Attribut *poolsize* die Anzahl der Nachrichten angegeben, die ein Super-Peer bzw. Peer in der durch *time* vorgegeben Zeit in Millisekunden verarbeiten kann. Das Element *peers* enthält zusätzlich zu den oben schon erwähnten Abschnitten für Netzwerkverbindung und Verarbeitungszeiten noch die Elemente *consumer* und *provider*. Beide Element verfügen jeweils über die Attribute *type*, *count* und *livingtime*. *type* spezifiziert dabei in gleicher Weise wie für die Super-Peers und die Topologie die zu verwendende Implementierung für Consumer- bzw. Providerpeers. *count* definiert die Anzahl von Consumern bzw. Providern, die sich ständig im Netzwerk befinden sollen. Im Unterschied zu den Super-Peers allerdings bleibt hier nur die Anzahl annähernd konstant, während ständig Peers das Netzwerk verlassen bzw. neue hinzukommen. Die Zeit, die ein Consumer bzw. Provider dem Netzwerk angehört, wird durch das Attribut *livingtime* in Millisekunden bestimmt. Die zum Simulationszeitpunkt „0“ erzeugten Peers, werden mit einer im Intervall $]0, livingtime]$ gleichverteilten

Lebenszeit erstellt, um das Austauschen aller Peers zum gleichen Zeitpunkt zu verhindern. Über die Elemente *query* bzw. *response* kann das Verhalten der Consumer bzw. Provider in Bezug auf die Anfragen und Antworten definiert werden. Das Attribut *limit* gibt eine Anzahl Anfragen vor, nach der keine weiteren Anfragen mehr erzeugt werden. Mit Hilfe von *burstdelay*, *burstsize* und *querydelay* kann ein burst-artiges Anfrageverhalten der Consumer simuliert werden. *burstdelay* bestimmt die Zeit in Millisekunden, die zwischen zwei Anfragebursts vergeht. *burstsize* bestimmt die Anzahl der Anfragen, aus denen jeweils ein Burst zusammengesetzt ist und *querydelay* spezifiziert die Verzögerung in Millisekunden, mit der die Anfragen innerhalb des Bursts gestellt werden. Das Attribut *probability* des Elements *response* bestimmt schließlich die Wahrscheinlichkeit, mit der ein Provider auf eine eintreffende Anfrage antwortet. Tabelle 5-1 gibt eine Übersicht über die genannten Parameter.

Tabelle 5-1: Übersicht Simulationsparameter

Element	Attribut	Bedeutung
simulation	setuptime	Vorlaufzeit bis zur eigentlichen Simulation.
simulation	threadcount	Anzahl der zu benutzenden Java Threads.
topology	type	Typ der Topologie.
TTL	informationmsg	TTL für Informationsnachrichten.
TTL	query	TTL für Anfragen.
TTL	response	TTL für Antworten.
superpeers	type	Typ der Super-Peers.
superpeers	count	Anzahl der Super-Peers.
delay	avg	Durchschnittliche Netzwerkverzögerung.
delay	deviation	Standardabweichung der Netzwerkverzögerung.
bandwidth	avg	Durchschnittliche Bandbreite der Netzwerkverbindung.

bandwidth	deviation	Standardabweichung der Bandbreite der Netzwerkverbindung.
processing	poolsize	Anzahl der parallel verarbeitbaren Nachrichten.
processing	time	Dauer der Verarbeitung.
consumer	type	Typ der Consumer-Peers.
consumer	count	Anzahl der Consumer-Peers.
consumer	livingtime	Lebensdauer der Consumer-Peers.
query	limit	Anzahl der maximal ausgeführten Anfragen.
query	burstdelay	Abstand zwischen zwei „Bursts“.
query	burstsize	Anzahl der Anfragen in einem „Burst“.
query	querydelay	Abstand zwischen zwei Anfragen innerhalb eines „Bursts“.
provider	type	Typ der Provider-Peers.
provider	count	Anzahl der Provider-Peers.
provider	livingtime	Lebensdauer der Provider-Peers.
response	probability	Wahrscheinlichkeit einer Antwort.
summary	type	Typ der „Summary“.

5.2.2 Aufbau der Datei „schema.xml“

Die Datei „schema.xml“ ist die Konfigurationsdatei für den schema-basierten Teil der Simulation. Die zugehörige DTD findet sich in der Datei „schema.dtd“ (siehe Anhang). Das Root-Element *schema* enthält die Elemente *schemadistribution*, *superpeer* und *consumer*. Das Element *schemadistribution* legt mit seinen Attributen und Unterelementen alle Parameter für die Erzeugung

und Verteilung von Schemas und Properties auf Provider und Anfragen fest. Durch das Attribut *file* wird der Name der für die Speicherung der Verteilung verwendeten Datei angegeben. Ist eine solche Datei vorhanden, werden die darin gespeicherten Parameter statt der in „schema.xml“ spezifizierten verwendet. Falls die Datei noch nicht vorhanden ist, wird sie neu erzeugt unter Verwendung der aktuellen Parameter in „schema.xml“. Das Element *schema* definiert mittels des Attributs *count* die Gesamtanzahl der zu erzeugenden Schemata, die als Basis für Provider und Anfragen zur Verfügung stehen. Durch das Element *properties* wird in ähnlicher Weise die Anzahl der Properties pro Schema festgelegt, allerdings spezifiziert *count* hier nur einen Durchschnittswert mit der entsprechenden Standardabweichung *deviation*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema SYSTEM "schema.dtd">
<schema>
  <schemadistribution file="distribution.xml">
    <schemas count="3"/>
    <properties count="5" deviation="1.0"/>
    <providerschemas count="1" deviation="1.0"/>
    <providerproperties count="5" deviation="1.0"/>
    <queryproperties count="3" deviation="1.0"/>
    <zipf shape="0.4"/>
  </schemadistribution>
  <superpeer routingtimeout="30000">
    <superpeerindex type="sim.schema.index.DummyPeerIndex"/>
    <peerindex type="sim.schema.index.OptimisticPeerIndex"/>
    <statistics thresholdrec="100" thresholdproc="100"
thresholdtrans="100"/>
  </superpeer>
  <consumer querytimeout="30000"/>
</schema>
```

Abbildung 5-10: Ausschnitt aus „schema.xml“

providerschemas, *providerproperties* und *queryproperties* bestimmen analog dazu die Anzahl der Schemas bzw. Properties, die in der Providerinformation, bzw. den Anfragen Verwendung finden. Das Element *zipf* legt durch das Attribut *shape* die sog. Steilheit der zugrunde liegenden Zipf-Verteilungen fest. Das Element *superpeer* definiert wichtige Parameter, die spezifisch für die Implementierung der schema-basierten Super-Peers sind. *routingtimeout* definiert die Zeitspanne, für die sich ein Super-Peer den Weg einer erhaltenen Anfrage merkt, um eventuelle Antworten weiterleiten zu können. Näheres dazu findet sich im Abschnitt 5.3.3. *superpeerindex* und *peerindex* bestimmen mittels *type* jeweils den Typ des zu verwendenden SuperPeer/SuperPeer bzw. SuperPeer/PeerIndex. Details zur Verwendung und Aufgabe dieser Indices finden sich in [4], bzw. in Abschnitt 5.3.3. Im Element *statistics* können für jede der Warteschlangen eines Super-Peers Schwellwerte definiert werden, deren Überschreitung in der Zusammenfassung der Statistiken besonders ausgewiesen wird. Im Element *consumer* kann mittels *querytimeout* schließlich noch die Zeitspanne angegeben werden, die für jede Anfrage auf Antworten gewartet wird. Tabelle 5-2 gibt eine Übersicht über die genannten Parameter.

Tabelle 5-2: Übersicht Schemaparameter

Element	Attribut	Bedeutung
schemadistribution	file	Name der Datei, in der die Verteilung gespeichert wird.
schemas	count	Gesamtanzahl der Schemas.
properties	count	Durchschnittliche Anzahl der Properties pro Schema.
properties	deviation	Standardabweichung der Anzahl der Properties pro Schema.
providerschemas	count	Durchschnittliche Anzahl der Schemas pro Provider.
providerschemas	deviation	Standardabweichung der Anzahl der Schemas pro Provider.
providerproperties	count	Durchschnittliche Anzahl der Properties pro Provider.
providerproperties	deviation	Standardabweichung der Anzahl der Properties pro Provider.
queryproperties	count	Durchschnittliche Anzahl der Properties pro Anfrage.
queryproperties	deviation	Standardabweichung der Anzahl der Properties pro Anfrage.
zipf	shape	Shape-Faktor der Zipf-Verteilungen.
superpeer	routingtimeout	Maximale Zeitspanne, über die eine Anfrage in der Routing-Tabelle gespeichert wird.
superpeerindex	type	Typ des SP/SP-Index.
peerindex	type	Typ des SP/P-Index.
statistics	thresholdrec	Schwellwert für die <i>ReceiveQueue</i> .
statistics	thresholdproc	Schwellwert für die <i>ProcessingQueue</i> .
statistics	thresholdtrans	Schwellwert für die <i>TransmitQueue</i> .
consumer	querytimeout	Zeitspanne, über die ein Consumer-Peer auf Antworten zu einer Anfrage wartet.

5.2.3 Aufbau der Datei „distribution.xml“

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
  <object class="sim.schema.config.DistributionStorageBean">
    <void property="propertyCount">
      <array class="int" length="3">
        <void index="0">
          <int>3</int>
        </void>
        <void index="1">
          <int>5</int>
        </void>
        <void index="2">
          <int>5</int>
        </void>
      </array>
    </void>
    <void property="schemaCount">
      <int>3</int>
    </void>
  </object>
</java>
```

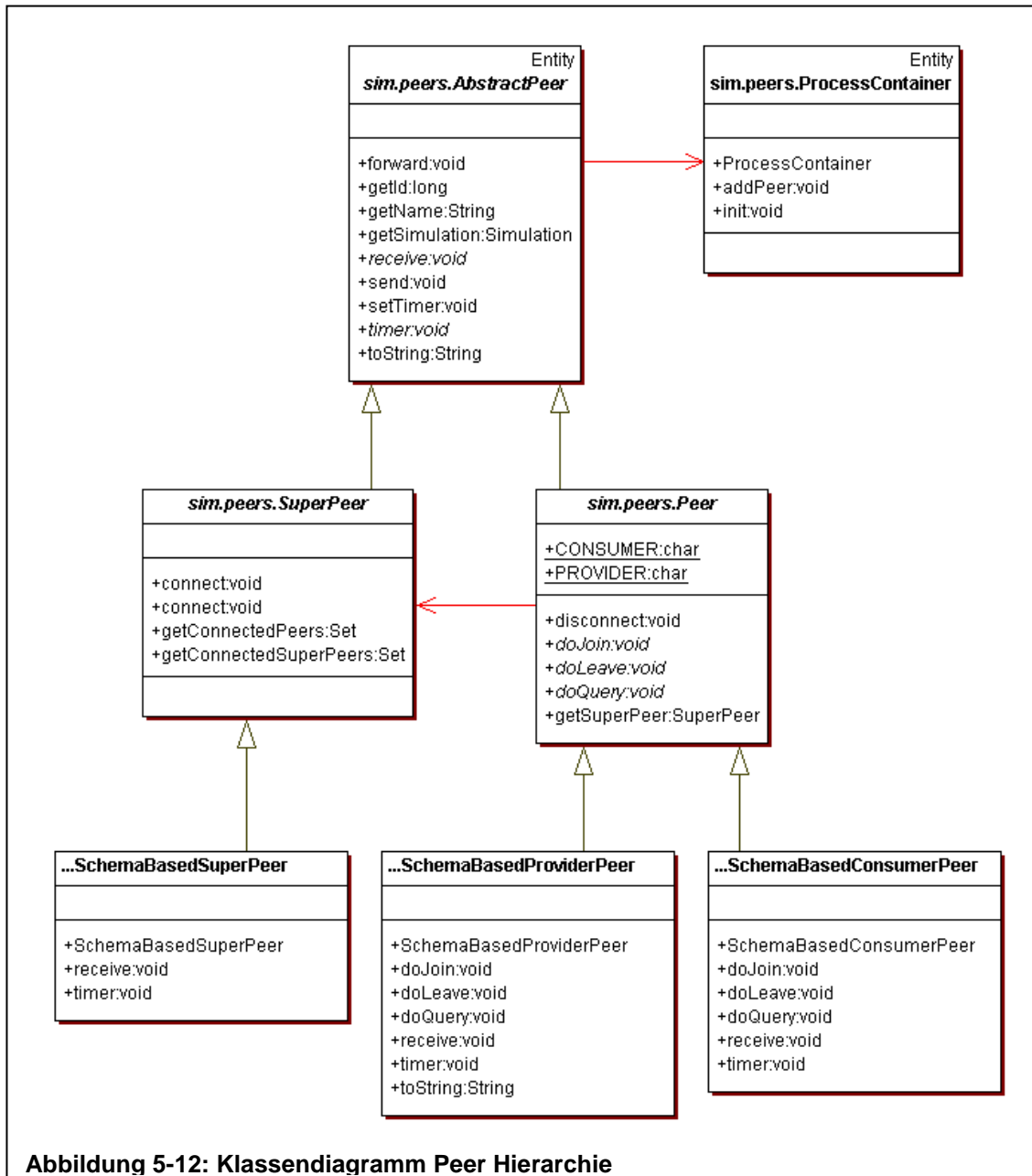
Abbildung 5-11: Ausschnitt aus „distribution.xml“

Die Datei „distribution.xml“ wird verwendet, um die Verteilung der Schemas und Properties zwischen verschiedenen Simulationsläufen zu speichern und die gewonnenen Daten so besser vergleichbar zu machen. Dazu wird als Hilfsklasse eine spezielle Java-Bean [15] verwendet, die mittels der im Java SDK enthaltenen Klassen *XMLEncoder* bzw. *XMLDecoder* als XML-Datei ausgegeben werden kann. Da die Verteilung nur von der Anzahl der Schemas bzw. der Anzahl der Properties pro Schema abhängt, genügt es statt der vollständigen Verteilung nur diese Daten zu speichern und daraus dann die Verteilung zu rekonstruieren.

5.3 Die Peers

Zunächst soll hier der prinzipielle Aufbau erläutert werden. Wie in Abschnitt 5.1 schon kurz angesprochen, wird in der Implementierung ein dreistufiges Design umgesetzt. Die Klasse *AbstractPeer* stellt, wie im Klassendiagramm zu sehen, die oberste Stufe dar. Sie definiert Methoden und Variablen, die allen Peers gemeinsam sind. Auf der mittleren Stufe folgen dann die Klassen *Peer* und *SuperPeer*, welche das durch *AbstractPeer* definierte Gerüst um spezifische Methoden und Variablen für einfache Peers bzw. Super-Peers erweitern. Die dritte Stufe stellen die konkreten Implementierungen der Peers da. Im Rahmen dieser Arbeit wurden schema-basierte Consumer, Provider und Super-Peers implementiert, in den Klassen *SchemaBasedConsumerPeer*,

SchemaBasedProviderPeer und *SchemaBasedSuperPeer*. Die Klasse *ProcessContainer* nimmt eine Sonderstellung ein. Sie ist keiner Stufe zugeordnet, sondern ist eine Hilfsklasse, um das im Überblick schon angesprochene Problem der großen Zahl der Peer-Prozesse zu lösen.



5.3.1 Das Senden und Empfangen von Nachrichten

Die Peers sind als Entitäten modelliert und kommunizieren über spezielle Ereignisse, die die Nachrichten modellieren. Im Scalable Simulation Framework

werden Ereignisse durch *channel* zwischen den Entitäten übertragen. Damit ein Ereignis übertragen werden kann, müssen die *channel* der Entitäten miteinander verknüpft werden. Die Kapselung dieser Verknüpfung erfolgt durch die beiden *connect()* Methoden der Klasse *SuperPeer*. Diese verbinden die beiden *channel* und speichern außerdem einen Zeitstempel, der angibt, wann diese Verbindung bereit sein wird. Jeder Peer verfügt über drei Nachrichtenwarteschlangen: Die *receiveQueue*, die *procQueue* und die *transmitQueue*. Die *receiveQueue* modelliert zusammen mit der *transmitQueue* die Netzwerkverbindung der Peers, während die *procQueue* die Verarbeitung im Peer selber modelliert. Das Senden von Nachrichten funktioniert, indem mittels der *send()* Methode die zu sendende Nachricht an die *transmitQueue* übergeben wird. Durch die Übergabe dieser Nachricht, wird im zugehörigen *ProcessContainer* ein spezielles internes Ereignis erzeugt, welches der Container nach einer durch die entsprechenden Simulationsparameter für Netzwerkverzögerung und Bandbreite vorgegebenen Zeitspanne an sich selbst sendet. Durch den Empfang dieses Ereignisses wird dann die zugehörige Nachricht aus der *transmitQueue* entnommen und mittels der Methode *sendToChannel()* über die *channel* gesendet. Dieser Vorgang ist für die eigentliche Implementierung auf der dritten Stufe völlig transparent. Auch der Empfang von Nachrichten funktioniert nach einem ähnlichen Prinzip. Die Implementierung der dritten Stufe muss dazu lediglich die abstrakte Methode *receive()* implementieren. Intern wird *receive()* durch den jeweiligen *ProcessContainer* aufgerufen. Ausgelöst wird dies wiederum durch ein spezielles internes Ereignis, das erzeugt wird, wenn eine Nachricht auf dem Eingangskanal empfangen und zunächst in der *receiveQueue* zwischengespeichert wurde. Dadurch kann auch auf Eingangsseite eine begrenzte Bandbreite simuliert werden.

5.3.2 Die Lösung des Prozess-Problems

Das Problem, dass der enorme Bedarf an echten Threads die maximale Anzahl der simulierbaren Peers extrem einschränken würde, wenn jeder Peer exklusive Prozesse erhalten würde, wird, wie im obigen Abschnitt schon angedeutet, gelöst, indem die Prozesse in einer Hilfsklasse, dem *ProcessContainer*, untergebracht werden und Pseudoparallelität durch spezielle interne Ereignisse nachgebildet wird. Zwar werden dadurch deutlich mehr Ereignisse erzeugt, als im echt parallelen Fall, aber der dadurch entstandene Mehraufwand wird durch die Einsparung der Threads mehr als kompensiert. Um trotzdem von einer gewissen Parallelisierung profitieren zu können, ist es möglich die Peers auf mehrere *ProcessContainer* aufzuteilen.

5.3.3 Die Consumer-Peers

Die Klasse *SchemaBasedConsumerPeer* implementiert neben der in *AbstractPeer* deklarierten abstrakten Methode *receive()* auch die in *Peer* deklarierten Methoden *doJoin()*, *doLeave()* und *doQuery()*. Alle vier Methoden sind Callback Methoden, die zu definierten Zeitpunkten von der Simulationsumgebung aufgerufen werden. *doJoin()* und *doLeave()* werden

```
public void doJoin() {
    send(
        new SchemaPeerJoinMsg(
            this,
            getSuperPeer(),
            new MessageId(nextMessageId++, getId()),
            Peer.CONSUMER,
            getSimulation().getNextInformationMsgTTL());
}

public void doLeave() {
    send(
        new SchemaPeerLeaveMsg(
            this,
            getSuperPeer(),
            new MessageId(nextMessageId++, getId()),
            Peer.CONSUMER,
            getSimulation().getNextInformationMsgTTL());
}
```

Abbildung 5-13: Ausschnitt aus „SchemaBasedConsumerPeer.java“

genau einmal am Anfang und Ende der Lebenszeit eines Peers aufgerufen, um Nachrichten zum Anmelden bzw. Abmelden im Netzwerk zu versenden. Für diesen Zweck wurden die Klassen *SchemaPeerJoinMsg* und *SchemaPeerLeaveMsg* implementiert. In Abbildung 5-13 ist die Implementierung des Anmelde- und Abmeldevorgangs des *SchemaBasedConsumerPeer* dargestellt. Beide Nachrichtentypen enthalten

```
public void doQuery() {
    Query query =
        schemaConfigurator.getNextQuery(
            this,
            getSuperPeer(),
            new MessageId(nextMessageId++, getId()),
            getSimulation().getNextQueryTTL());
    responseCounts.put(query.getMessageId(), new Integer(0));
    timeStamps.put(query.getMessageId(), new Long(now()));
    send(query);
    setTimer(
        schemaConfigurator.getNextQueryTimeOut(),
        query.getMessageId());
}
```

Abbildung 5-14: Ausschnitt aus „SchemaBasedConsumerPeer.java“

neben den Standardfeldern wie *Source*, *Destination*, *Id* und *TTL* ein zusätzliches Feld, um zwischen Consumern und Providern zu unterscheiden. Die Methode *doQuery()* wird entsprechend der in der Konfiguration abgelegten Parameter aufgerufen, wenn eine Anfrage gestellt werden soll. In Abbildung 5-14 ist die Implementierung von *doQuery()* dargestellt. Durch den *SchemaConfigurator* wird hier eine neue Query mit entsprechenden Properties konstruiert und abgeschickt. Zusätzlich wird der durch *AbstractPeer* zur Verfügung gestellte Timer-Mechanismus benutzt, um ein Timeout für diese neue Anfrage einzustellen. Um diesen Timeout zu registrieren, muss die entsprechende Callback Methode wie in Abbildung 5-15 implementiert werden.

```
public void timer(Object object) {
    MessageId id = (MessageId) object;
    Integer responseCount = (Integer) responseCounts.get(id);
    if (responseCount != null && responseCount.intValue() > 0) {
        [...]
    } else {
        [...]
    }
    responseCounts.remove(id);
    timeStamps.remove(id);
}
```

Abbildung 5-15: Ausschnitt aus „SchemaBasedConsumerPeer.java“

Schließlich muss noch die Methode *receive()* so implementiert werden, dass der Consumer auf eintreffende Antworten reagiert (siehe Abbildung 5-16).

```
public void receive(Message message) {
    if (message instanceof SchemaResponse) {
        SchemaResponse response = (SchemaResponse) message;
        [...]
    }
}
```

Abbildung 5-16: Ausschnitt aus „SchemaBasedConsumerPeer.java“

Bei der Erzeugung der Anfrage durch den *SchemaConfigurator* wird ein zweistufiges Verfahren eingesetzt. Durch die Zipf-Verteilung wird jedem Schema eine Wahrscheinlichkeit zugeordnet.

Tabelle 5-3

	Schema 1	Schema 2	Schema 3
Wahrscheinlichk.	0.6275	0.2378	0.1348

In Tabelle 5-3 ist eine typischen Zipf-Verteilung für drei Schemas dargestellt. Mit Hilfe dieser Verteilung wird zunächst ein Schema bestimmt, aus welchem dann wiederum durch eine Zipf-Verteilung eine Property ausgewählt wird. Dies

wird solange fortgesetzt, bis eine wiederum zufällig ermittelte Anzahl an Properties ausgewählt wurde.

```
private SchemaCapability sampleQueryCapability() {
    [...]
    for (int i = 0; i < propCount; i++) {
        int nextSchema = schemaDist.nextInt();
        p.add(properties[nextSchema].get(propDist[nextSchema].nextInt(
    )));
        i = p.size();
    }
    [...]
}
```

Abbildung 5-17: Ausschnitt aus „SchemaConfigurator.java“

In Abbildung 5-17 ist ein Ausschnitt aus der entsprechenden Methode dargestellt.

5.3.4 Die Provider-Peers

Die Provider-Peers implementieren prinzipiell die gleichen Methoden wie auch die Consumer-Peers. Dadurch ist es möglich, in einer Klasse sowohl die Consumer-, als auch die Provider-Funktionalität unterzubringen. In der vorliegenden Implementierung des *SchemaBasedProviderPeer* wurde allerdings darauf verzichtet, um so das Verhalten von Provider und Consumern klar zu trennen. Beim *SchemaBasedProvider* ist die Methode *doQuery()* daher ohne Funktion (siehe Abbildung 5-18).

```
public void doQuery() {
    // This type of peer does not perform any queries.
}
```

Abbildung 5-18: Ausschnitt aus „SchemaBasedProviderPeer.java“

doJoin() und *doLeave()* wurden in ähnlicher Weise implementiert, wie schon beim *SchemaBasedConsumerPeer* gezeigt, mit dem Unterschied, dass das Typ-Feld hier den Peer als Provider kennzeichnet und ein zusätzliches Feld zur Übergabe der Schemainformationen verwendet wird (siehe Abbildung 5-19). Die Schemainformation des Providers wird bei dessen Konstruktion durch den *SchemaConfigurator* erzeugt. Das Verfahren ist dabei das gleiche wie das zur Erzeugung der Schemainformation für die Anfragen, mit dem einzigen Unterschied, dass Providern auch ganz Schemas zugeordnet werden können.

Ob eine Anfrage beantwortet werden kann, wird in der *receive()* Methode anhand der vom Benutzer vorgegebenen Wahrscheinlichkeit entschieden.

```

public void doJoin() {
    send(
        new SchemaPeerJoinMsg(
            this,
            getSuperPeer(),
            new MessageId(nextMessageId++, getId()),
            Peer.PROVIDER,
            schemaCapability,
            getSimulation().getNextInformationMsgTTL());
}

public void doLeave() {
    send(
        new SchemaPeerLeaveMsg(
            this,
            getSuperPeer(),
            new MessageId(nextMessageId++, getId()),
            Peer.PROVIDER,
            getSimulation().getNextInformationMsgTTL());
}

```

Abbildung 5-19: Ausschnitt aus „SchemaBasedProviderPeer.java“

Vorher wird noch überprüft, ob die Anfrage von der Schemainformation her überhaupt zu den Fähigkeiten des Providers passt. In der vorliegenden Implementierung werden zwar durch die Super-Peers Anfragen überhaupt nur an solche Provider weitergeleitet, die über entsprechenden Schemainformationen verfügen, aber hier sind natürlich auch andere Implementierungen möglich, so dass eine solche Überprüfung sinnvoll erscheint, um zu realistischen Ergebnissen zu gelangen. Die Implementierung der *receive()* Methode ist in Abbildung 5-20 dargestellt.

```

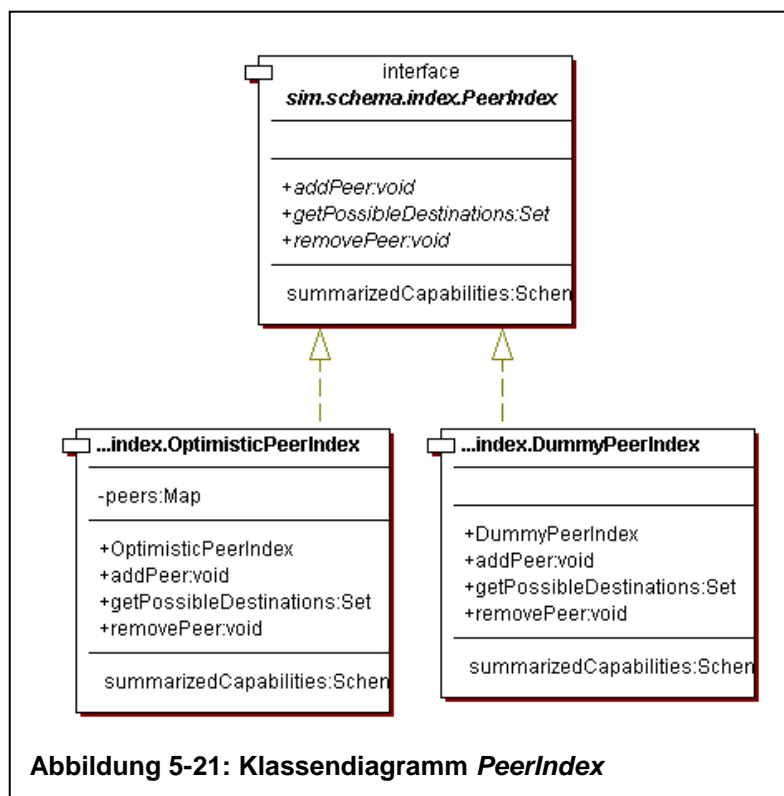
public void receive(Message message) {
    if (message instanceof SchemaQuery) {
        SchemaQuery query = (SchemaQuery) message;
        if (schemaCapability
            .smartContainsProperties(query.getSchemaCapability())
            && getSimulation().respondToNextQuery()) {
            send(
                new SchemaResponse(
                    this,
                    query.getTransmitter(),
                    new MessageId(nextMessageId++,
                        getId()),
                    query.getMessageId(),
                    getSimulation().getNextResponseTTL());
                [...]
            }
        }
    }
}

```

Abbildung 5-20: Ausschnitt aus „SchemaBasedProviderPeer.java“

5.3.5 Die Super-Peers

Die Super-Peers, implementiert in *SchemaBasedSuperPeer*, werden in der Klassenhierarchie von *SuperPeer* abgeleitet. Grundsätzlich muss nur die *receive()* Methode implementiert werden, diese ist aber wesentlich komplexer als bei den einfachen Peers. Daraus folgt natürlich, dass Super-Peers hier lediglich auf von außen eintreffende Nachrichten reagieren, was aber auch realistisch ist. Die im Rahmen dieser Arbeit implementierten Super-Peers verwenden spezielle Schemaindices, um über das Routing der Nachrichten, welches ja die Hauptaufgabe der Super-Peers darstellt, zu entscheiden. Die Implementierung dieser Indices orientiert sich in wesentlichen Punkten an der in der Edutella-Architektur verwendeten, ist aber natürlich den Erfordernissen der Simulation entsprechend etwas vereinfacht. Die Aufgabe der Indices ist es, die



Schemainformationen über angeschlossene Provider und Super-Peers zu nutzen, um das Routing der Nachrichten an Ziele zu vermeiden, von denen keine Antwort zu erwarten ist. Details zu Schemaindices und deren Aufgaben finden sich in [4]. Das Routing wird außer durch die Indices noch durch die Topologie des Netzwerkes beeinflusst. In der Realität entsteht diese Topologie implizit durch die Art und Weise, auf die sich Super-Peers in das Netzwerk integrieren. In der Simulation wird die Topologie dagegen von außen „aufgeprägt“. Auf diese Weise können verschiedene Topologien getestet werden, ohne die Implementierung der Super-Peers anpassen zu müssen. Das

Routing der Nachrichten durch die Super-Peers ist demnach ein zweistufiger Prozess. In der ersten Stufe ermittelt der Super-Peer zunächst die durch die Topologie gegebenen möglichen Empfänger einer Nachricht und trifft daraus dann entsprechend seiner Schemaindices eine Auswahl von Empfängern, an die die Nachricht schließlich weitergeleitet wird. Durch die Kapselung der Indices mittels des Interfaces *PeerIndex* ist es möglich verschiedene Index-Implementierungen einfach auszutauschen. Durch entsprechende Factory-Klassen für Topologien und Indices geschieht dies zur Laufzeit nach den

```

public void receive(Message message) {
    if (message instanceof SchemaPeerJoinMsg) {
        [...]
    } else if (message instanceof SchemaPeerLeaveMsg) {
        [...]
    } else if (message instanceof SchemaQuery) {
        [...]
        Set destinationProviders =
            peerIndex.getPossibleDestinations(this, query);
        Set destinationSuperPeers =
            superPeerIndex.getPossibleDestinations(this,
query);
        destinationSuperPeers.retainAll(
            getSimulation().getTopology().getPossibleDestinations(
                this,
                query));
        [...]
    } else if (message instanceof SchemaResponse) {
        [...]
    } else if (message instanceof SchemaSPSPIndexUpdateMsg) {
        [...]
    }
}

```

Abbildung 5-22: Ausschnitt aus „SchemaBasedSuperPeer.java“

Vorgaben der Konfiguration. Um verschiedene Topologien und Index-Strategien zu testen, müssen also keine Änderungen am Programmcode selbst vorgenommen werden. Auch lässt sich die Simulation leicht um weitere Topologien und Index-Typen erweitern. Die Beschreibung der verschiedenen in dieser Arbeit entwickelten Topologien erfolgt in Abschnitt 5.4. In Abbildung 5-21 ist das *PeerIndex*-Interface dargestellt. Im Rahmen dieser Arbeit wurden zwei Indices entwickelt und implementiert. Ein *DummyPeerIndex* der keinerlei wirkliche Funktionalität bereitstellt, sondern dazu dient, die Index-Funktionalität auszuschalten und ein *OptimisticPeerIndex*. Auf die Aufgabe des *DummyPeerIndex* wird genauer in Abschnitt 5.4.2 eingegangen. Der *OptimisticPeerIndex* vergleicht die Anforderungen einer Anfrage mit den möglichen Zielen und liefert diejenigen zurück, die zu den Anforderungen passen. Er ist dabei insofern optimistisch, als dabei nicht nur die vorhandenen Informationen über die Properties eines Providers einbezogen werden, sondern auch aus den Informationen über die Schemas zusätzliche Properties berechnet werden, die der Provider eventuell unterstützt. In Abbildung 5-22 ist

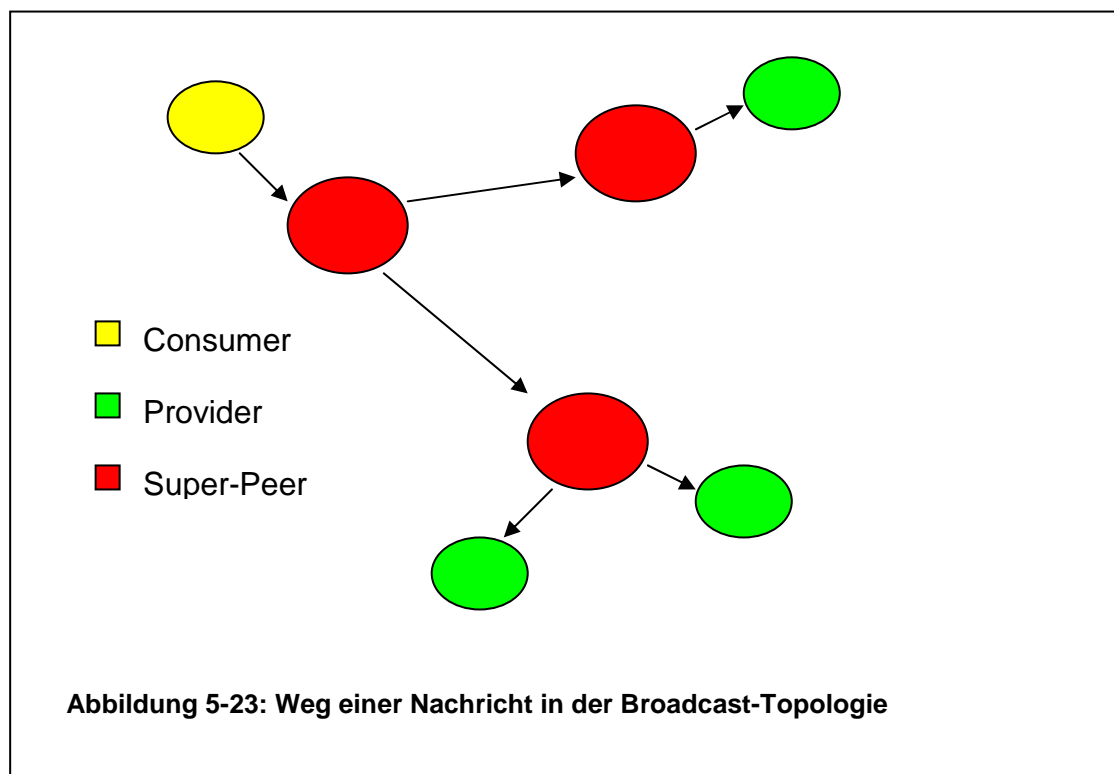
verkürzt die Implementierung der `receive()` Methode des `SchemaBasedSuperPeer` dargestellt.

5.4 Die Topologien

Für die Simulationsumgebung wurden im Rahmen dieser Arbeit zwei Topologien implementiert. Eine einfache Broadcast-Topologie und eine auf dem Hypercup Protokoll (siehe [14]) basierende.

5.4.1 Die Broadcast-Topologie

Die Broadcast-Topologie ist in der Klasse `BroadcastTopology` implementiert. In dieser Topologie existiert zwischen allen Super-Peers eine direkte Verbindung. Eine Nachricht läuft also immer über maximal zwei Super-Peers. Damit es in Verbindung mit dieser Topologie nicht zu einer Überflutung des Netzwerkes kommt, sollten hier sinnvolle TTLs für die Nachrichten vorgesehen werden. Diese normalerweise in Bezug auf Netzwerklast sehr ineffiziente Topologie kann durch den Einsatz der Schemaindices auf Peer und Super-Peer Ebene



deutlich verbessert werden. Nachrichten werden in dieser Topologie durch die Super-Peers weitergeleitet, indem diese einfach an alle Super-Peer-Nachbarn und angeschlossene Peers gesendet, also per *Broadcast* verteilt wird. Die Auswahl der Nachbarn kann dabei durch den Super-Peer/Super-Peer-Index eingeschränkt werden, die der angeschlossenen Peers durch den Super-Peer/Peer-Index. Der Weg einer Nachricht durch das Netzwerk ist in Abbildung 5-23 dargestellt.

5.4.2 Die Hypercube Topologie

Die Klasse *HyperCubeTopology* implementiert die in [14] vorgestellte Hypercube-Topologie für Peer-to-Peer Netzwerke. Die vorliegende Implementierung unterstützt nur vollständige Hypercubes. Die Anzahl der Super-Peers wird automatisch entsprechend angepasst, um den in Bezug auf die Dimension jeweils nächst liegenden vollständigen Hypercube zu konstruieren. Der Aufbau der Topologie geschieht in der Methode *buildTopology()*, die in Abbildung 5-24 dargestellt ist.

```

public void buildTopology(Simulation simulation)
    throws TopologyException, PeerCreationException {
    [...]
    dim =
        (int) Math.round(
            Math.log((double) simulation.getSuperPeerCount())
                / Math.log(2.0));
    superPeerCount = 1 << dim;
    [...]
    for (int i = 0; i < sPeers.size(); i++) {
        SuperPeer peer = (SuperPeer) sPeers.get(i);
        LinkInfo info = (LinkInfo) superPeers.get(peer);
        for (int j = 0; j < dim; j++) {
            int next = i ^ (1 << j);
            SuperPeer neighbour = (SuperPeer) sPeers.get(next);
            info.addLink(j, neighbour);
            peer.connect(neighbour);
        }
        simulation.getNextContainer().addPeer(peer);
    }
}

```

Abbildung 5-24: Ausschnitt aus „HyperCubeTopology.java“

Zuerst wird hier die Dimension des Hypercubes berechnet und die Anzahl der zu erstellenden Super-Peers entsprechend angepasst. Danach wird jeder Super-Peer in jeder Dimension mit seinem Partner verbunden. Die Position des Partners ergibt sich, indem in der Bit-Darstellung der eigenen Position des Super-Peers das Bit, welches der zu verbindenden Dimension entspricht, invertiert wird.

Super-Peer Position: $0 = (00)_2$
 Position des Partners für Dimension 0: $(01)_2 = 1$
 Position des Partners für Dimension 1: $(10)_2 = 2$
 ...

Abbildung 5-25: Beispiel

Das besondere an der Hypercube-Topologie ist die Art und Weise, wie Nachrichten weitergeleitet werden. Der Algorithmus zum Weiterleiten einer Nachricht funktioniert dabei im Prinzip wie folgt. Im ersten Schritt wird die Nachricht an alle Nachbarn gesendet. Dabei ist durch die Topologie jedem Nachbarn eine Dimension zugeordnet. Die Nachbarn wiederum senden die Nachricht dann ihrerseits nur über die Verbindungen weiter, die einer höheren Dimension zugeordnet sind, als die, über welche die Nachricht empfangen wurde. Auf diese Weise wird sichergestellt, dass jeder Super-Peer im Netzwerk die Nachricht genau einmal erhält. Details dazu finden sich in [14]. In Abbildung 5-26 ist ein vollständiger Hypercube der Dimension drei mit Kennzeichnung der Dimensionen dargestellt.

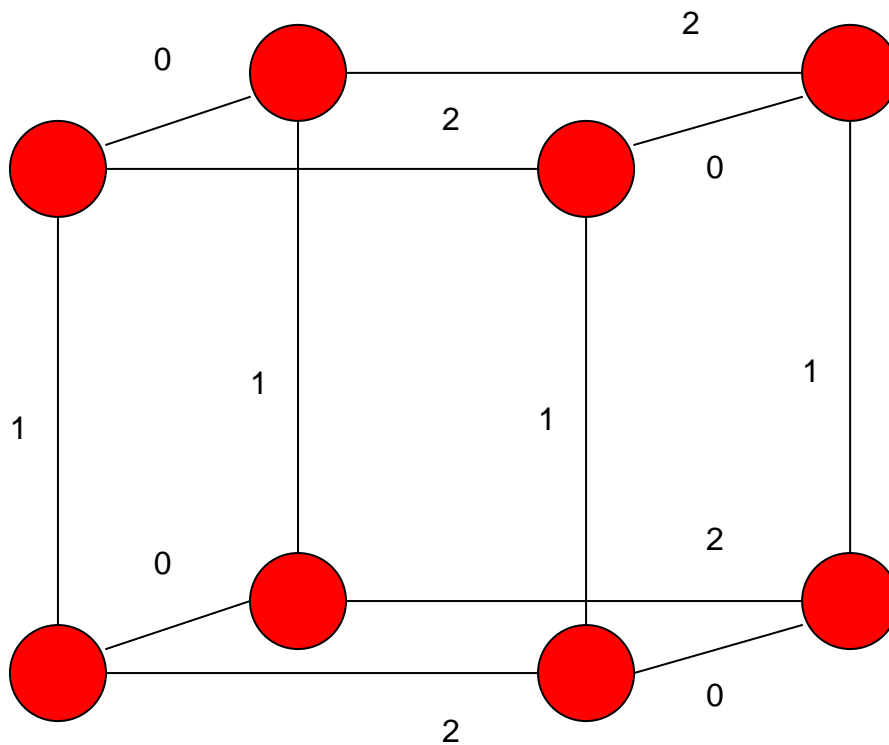
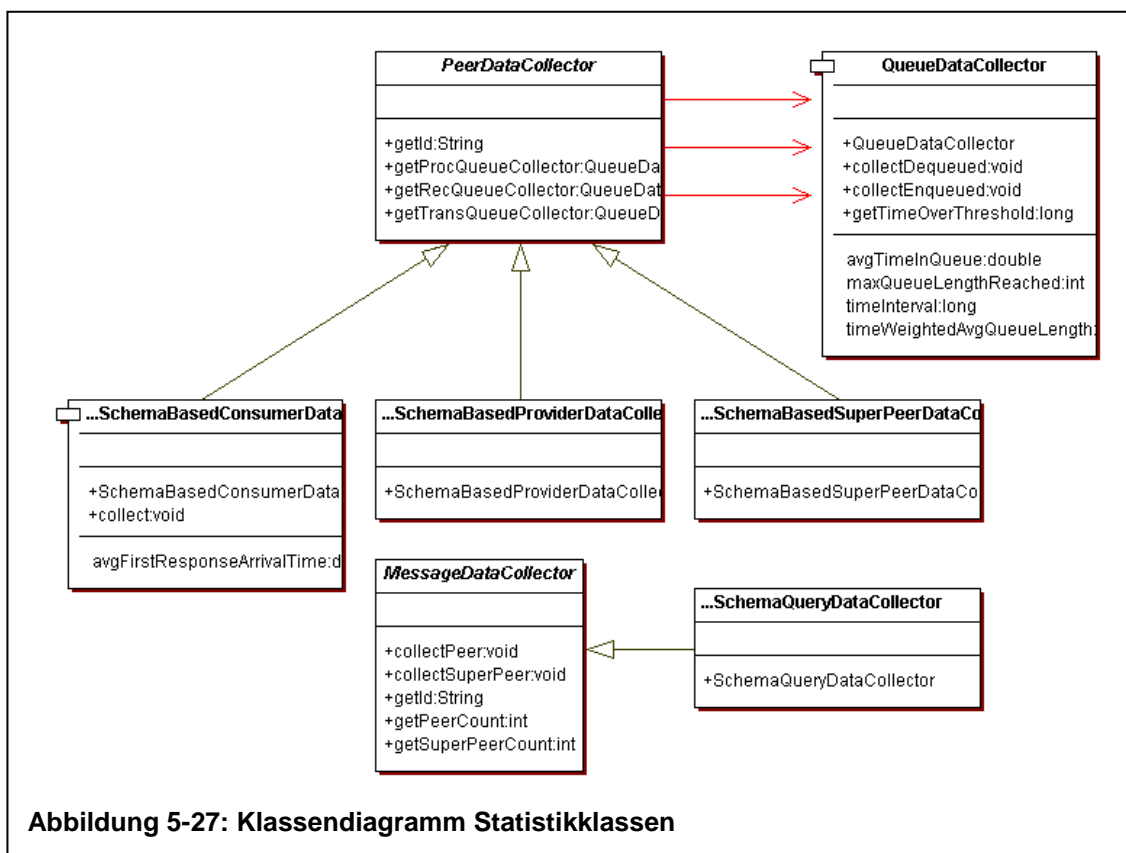


Abbildung 5-26: Hypercube

Zusammen mit der Hypercube-Topologie macht das Setzen von TTLs - außer zu Testzwecken - normalerweise keinen Sinn, es sei denn, um auf diese Weise lokal begrenzte Anfragen auszuführen. Die Verbindung der Hypercube-Topologie mit Schemaindices ist nicht trivial. Der in [14] angeführte Ansatz für Ontologien, für jede Gruppe von Peers, die man anhand der Schemainformationen gleich klassifizieren würde, einen eigenen Hypercube zu konstruieren, ist durch die Dynamik mit der z.B. die Provider wechseln und der Vielzahl der Kombinationsmöglichkeiten von Schemas und Properties hier nicht durchführbar. Da ein Super-Peer den Weg nicht kennt, den eine Anfrage in der Hypercube-Topologie nimmt, bleibt nichts anderes übrig, als auf den SuperPeer/SuperPeer-Index zu verzichten, bzw. mit Hilfe des *DummyPeerIndex* die Index-Funktionalität auszuschalten. Der SuperPeer/Peer-Index bleibt davon natürlich unberührt.

5.5 Die statistische Auswertung und Ausgabe

Statistische Daten über die Simulation werden mittels spezieller implementierungsspezifischer „Collector“-Klassen gesammelt. Diese werden von einer der drei Basisklassen *QueueDataCollector*, *MessageDataCollector*



oder *PeerDataCollector* abgeleitet. *QueueDataCollector* ist die Basisklasse für die Erfassung Warteschlangen-bezogener Daten. Von *MessageDataCollector*

abgeleitete Klassen sammeln Daten über spezielle Nachrichtentypen und *PeerDataCollector* ist die Basisklasse für alle Klassen, die Datensätze über Peers und Super-Peers sammeln. Die „Collector“-Klassen werden zu Beginn der Simulation bei speziellen Beobachter-Klassen registriert, die die Daten zusammenfassen und auswerten. Diese Auswertung wird dann an eine „Summary“-Klasse übergeben, die für die Darstellung der Ausgabe verantwortlich ist und schließlich entweder direkt auf der Konsole oder durch Hilfsklassen zur formatierten Ausgabe ausgegeben. Im Rahmen dieser Arbeit wurden entsprechenden Klassen für schema-basierte Peers, Super-Peers und schema-basierte Anfragen implementiert. Um Daten über die Warteschlangen der Peers zu gewinnen, enthält jede Nachrichtenwarteschlange, wie Eingangs schon erwähnt, einen *QueueDataCollector*. Da hier die Basisklasse schon alle benötigten Funktionen bereitstellt, wird diese hier direkt eingesetzt. Die Nachrichtenwarteschlange ruft bei jeder Operation die entsprechenden Methoden des „Collector“-Objektes auf. Die Basisklasse *PeerDataCollector* fasst die drei *QueueDataCollector*-Objekte pro Peer zusammen. Von ihr wurden die Klassen *SchemaBasedConsumerDataCollector*, *SchemaBasedProviderDataCollector* und *SchemaBasedSuperPeerDataCollector* abgeleitet, um zusätzlich noch für den jeweiligen Peertyp spezifische Daten zu erfassen. Ebenfalls wurde die von *MessageDataCollector* abgeleitete Klasse *SchemaQueryDataCollector* implementiert, um Daten über den Nachrichtentyp *SchemaQuery* zu sammeln. In Abbildung 5-27 sind die wichtigsten Klassen in der Übersicht dargestellt. Durch den Einsatz solcher implementierungsspezifischer Klassen, lassen sich die Erfassung und Auswertung der Daten sehr gezielt den Bedürfnissen des Benutzers anpassen. Durch die in dieser Arbeit entwickelte Implementierung werden folgende statistische Daten erhoben:

- Für jede Warteschlange:
 - Die maximal erreichte Länge.
 - Die durchschnittliche Länge.
 - Die durchschnittliche Verweildauer einer Nachricht.
 - Die Zeit, in der die Länge über einem bestimmten Grenzwert lag.
- Für die schema-basierten Consumer-Peers:
 - Die Durchschnitte von maximaler Länge, durchschnittlicher Länge und Verweildauer je Warteschlange.
 - Die durchschnittliche Zeit bis zur ersten Antwort auf eine Anfrage,
- Für die schema-basierten Provider-Peers:
 - Die Durchschnitte von maximaler Länge, durchschnittlicher Länge und Verweildauer je Warteschlange.
- Für jeden schema-basierten Super-Peer:
 - Die maximale und durchschnittliche Länge, sowie die Verweildauer und die Zeit über einem bestimmten Grenzwert je Warteschlange.
- Für jede *SchemaQuery*:
 - Die Anzahl der besuchten Peers, getrennt nach „einfachen“ Peers und Super-Peers, ohne den Consumer, der die Anfrage gestellt hat.

Die Angaben über die Peers werden am Ende der Simulation als Zusammenfassung auf der Konsole und als HTML-Datei ausgegeben, die Daten über die Anfragen werden in einem einfachen Textformat getrennt gespeichert, um sie einfach weiterverarbeiten zu können. Von der ganzen Simulation wird außerdem ein detailliertes Log erstellt, in dem alle Abläufe nachvollzogen werden können. Beispiele dazu finden sich in Abschnitt 6.

6 Beispiel Szenario

Im folgenden Abschnitt soll ein Simulationslauf exemplarisch vorgeführt werden. Die verwendeten Konfigurationsdateien sind unten abgebildet.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE simulation SYSTEM "simulation.dtd">
<simulation setuptime="10000" simtime="60000" threadcount="32">
  <topology type="sim.topology.HyperCubeTopology">
    <TTL informationmsg="-1" query="-1" response="-1"/>
  </topology>
  <superpeers type="sim.schema.peers.SchemaBasedSuperPeer"
count="32">
    <network>
      <delay avg="80" deviation="10.0"/>
      <bandwidth avg="1000" deviation="100.0"/>
    </network>
    <processing poolsize="10" time="30"/>
  </superpeers>
  <peers>
    <network>
      <delay avg="120" deviation="10.0"/>
      <bandwidth avg="100" deviation="15.0"/>
    </network>
    <processing poolsize="5" time="30"/>
    <consumer type="sim.schema.peers.SchemaBasedConsumerPeer"
count="1000" livingtime="600000">
      <query limit="100" burstdelay="120000" burstsize="3"
querydelay="30000"/>
    </consumer>
    <provider type="sim.schema.peers.SchemaBasedProviderPeer"
count="100" livingtime="3600000">
      <response probability="0.3"/>
    </provider>
  </peers>
  <summary type="sim.schema.statistic.SchemaBasedSummary"/>
</simulation>
```

Abbildung 6-1: Beispiel „simulation.xml“

Simuliert werden soll ein Netzwerk mit 32 in der Hypercube-Topologie angeordneten Super-Peers, sowie 1000 Consumer- und 100 Provider-Peers. Die Netzwerkverzögerung für Super-Peer/Super-Peer-Verbindungen beträgt durchschnittlich 80 ms, mit einer Standardabweichung von 10 ms. Die

Bandbreite beträgt durchschnittlich 1000 Nachrichten pro Sekunde, mit einer Abweichung von 100 Nachrichten pro Sekunde. Super-Peers können 10 Nachrichten parallel innerhalb von 30 ms verarbeiten. Die Netzwerkverzögerung der Super-Peer/Peer-Verbindungen beträgt durchschnittlich 120 ms, mit einer Abweichung von 10 ms und die Bandbreite eines Peers durchschnittlich 100 Nachrichten pro Sekunde mit einer Abweichung von 15 Nachrichten pro Sekunde. „Einfache“ Peers können 5 Nachrichten parallel in 30 ms verarbeiten. Die Lebensdauer eines Consumer-Peers beträgt 10 min, die eines Provider-Peers 1 Stunde. Ein Consumer stellt alle 2 Minuten in einem Abstand von 30 Sekunden jeweils 3 Anfragen. Es werden maximal 100 Anfragen simuliert und mit einer Wahrscheinlichkeit von 30% beantwortet, wenn diese bei einem Provider eintreffen. Es sollen 60 Sekunden simuliert werden, die Setup-Time soll 10 Sekunden betragen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema SYSTEM "schema.dtd">
<schema>
  <schemadistribution file="distribution.xml">
    <schemas count="5"/>
    <properties count="5" deviation="1.0"/>
    <providerschemas count="1" deviation="1.0"/>
    <providerproperties count="10" deviation="1.0"/>
    <queryproperties count="3" deviation="1.0"/>
    <zipf shape="0.4"/>
  </schemadistribution>
  <superpeer routingtimeout="60000">
    <superpeerindex type="sim.schema.index.DummyPeerIndex"/>
    <peerindex type="sim.schema.index.OptimisticPeerIndex"/>
    <statistics thresholdrec="50" thresholdproc="50"
thresholdtrans="50"/>
  </superpeer>
  <consumer querytimeout="30000"/>
</schema>
```

Abbildung 6-2: Beispiel „schema.xml“

Im gesamten Netzwerk werden 5 verschiedene Schemas eingesetzt, von denen jedes durchschnittlich 5 Properties enthält. Jeder Provider-Peer unterstützt durchschnittlich ein ganzes Schema und 10 Properties, eine Anfrage enthält durchschnittlich 3 Properties. Super-Peers speichern eingegangene Anfragen für 1 Minute in ihrer Routing-Tabelle und Consumer warten 30 Sekunden auf Antworten zu einer Anfrage. In der Zusammenfassung soll für die Super-Peers ausgewiesen werden, wie lange eine der Warteschlangen mehr als 50 Nachrichten enthielt. Die Simulation eines Netzwerkes mit den genannten Parametern dauert auf einem handelsüblichen PC ca. 10 Sekunden, als Ausgabe erhält man neben der Zusammenfassung auf der Konsole eine HTML-Datei und zwei Text-Dateien, deren Inhalt unten auszugsweise wiedergegeben ist. Wie an der durchschnittlichen Zeit bis zur ersten Antwort zu sehen, reagiert das Netzwerk sehr schnell. Zum Vergleich, ergab eine Simulation der Broadcast-Topologie mit ansonsten gleichen Parametern ergab eine Zeit von ca. 836,41 ms.

[...]

Summary (SchemaBasedSuperPeer#10):

Average queue length (ReceiveQueue): 0.010090080341926584
 Average time spent in queue (ReceiveQueue): 1.090643274853801
 Maximal queue length (ReceiveQueue): 3
 Time over threshold (ReceiveQueue): 0

Average queue length (ProcessQueue): 0.21474714166013462
 Average time spent in queue (ProcessQueue): 23.230994152046783
 Maximal queue length (ProcessQueue): 9
 Time over threshold (ProcessQueue): 0

Average queue length (TransmitQueue): 0.1844028221065009
 Average time spent in queue (TransmitQueue): 6.524517087667162
 Maximal queue length (TransmitQueue): 20
 Time over threshold (TransmitQueue): 0

Summary (Provider):

Average queue length (ReceiveQueue): 0.005758314532665084
 Average time spent in queue (ReceiveQueue): 2.1993080252677975
 Average maximal queue length (ReceiveQueue): 1.2475247524752475

Average queue length (ProcessQueue): 0.07069909002296747
 Average time spent in queue (ProcessQueue): 28.216848898160805
 Average maximal queue length (ProcessQueue): 1.9306930693069306

Average queue length (TransmitQueue): 0.0012850687364505846
 Average time spent in queue (TransmitQueue): 1.5291815963409099
 Average maximal queue length (TransmitQueue): 1.3663366336633664

Summary (Consumer):

Average queue length (ReceiveQueue): 0.0035539376904794766
 Average time spent in queue (ReceiveQueue): 24.371162169747684
 Average maximal queue length (ReceiveQueue): 0.34012400354295835

Average queue length (ProcessQueue): 0.0021424059081805898
 Average time spent in queue (ProcessQueue): 23.781137884048285
 Average maximal queue length (ProcessQueue): 0.1612046058458813

Average queue length (TransmitQueue): 8.589858980289759E-6
 Average time spent in queue (TransmitQueue): 1.0
 Average maximal queue length (TransmitQueue): 0.27723649247121346

Average time until first response: 775.0816326530612

Abbildung 6-3: Ausschnitt aus „summary.html“

```

Query SuperPeerCount PeerCount
[...]
7671 32 63
8201 32 3
9461 32 14
2821 32 79
8691 32 28
9491 32 79
7741 32 18
7991 32 46
8791 32 39
[...]

```

Abbildung 6-4: Ausschnitt aus „queries.txt“

```

Schema Distribution
[0.5427775017308292, 0.20567421282563603, 0.11658752452655087,
0.07793595292095049, 0.057024807996033475]

HypeCubeTopology

SuperPeerCount(modified): 32
Dimension: 5

SchemaBasedSuperPeer#26 has 5 neighbour(s):
Link 0: SchemaBasedSuperPeer#25
Link 1: SchemaBasedSuperPeer#23
Link 3: SchemaBasedSuperPeer#21
Link 2: SchemaBasedSuperPeer#7
Link 4: SchemaBasedSuperPeer#2

[...]

(10249) SchemaBasedConsumerPeer#418: Send SchemaQuery(5491) [[]
[Property (3) 3, Property (2) 2, Property (2) 1]] to
SchemaBasedSuperPeer#26.
(10257) SchemaBasedSuperPeer#2: Received SchemaQuery(11041) [[]
[Property (1) 1]] from SchemaBasedConsumerPeer#973.
(10257) Container(14): Removed SchemaBasedConsumerPeer#490.
(10257) Container(14): Created SchemaBasedConsumerPeer#1021(600000).
(10258) SchemaBasedConsumerPeer#1021: Send SchemaPeerJoinMsg to
SchemaBasedSuperPeer#27.
(10287) SchemaBasedSuperPeer#2: Forwarding query "11041".

[...]

(69434) Container(12): Removed SchemaBasedConsumerPeer#744.
(69434) Container(12): Created SchemaBasedConsumerPeer#1131(600000).
(69435) SchemaBasedConsumerPeer#1131: Send SchemaPeerJoinMsg to
SchemaBasedSuperPeer#25.
(69563) SchemaBasedSuperPeer#25: Received SchemaPeerJoinMsg from
SchemaBasedConsumerPeer#1131.
(69593) SchemaBasedSuperPeer#25: Processed SchemaPeerJoinMsg from
SchemaBasedConsumerPeer#1131.

```

Abbildung 6-5: Ausschnitt aus „simlog.txt“

7 Leistungsfähigkeit des Simulators / Ausblick

Der in dieser Arbeit entwickelte Simulator ist in der Lage ein schema-basiertes Super-Peer-Netzwerk auf Nachrichtenebene zu simulieren. Dabei können flexibel verschiedene Topologie- und Index-Strategien getestet werden. Durch den verwendeten Aufbau, besonders die Kapselung der wichtigen Klassen durch Interfaces und die Nutzung von Vererbung und anderer Techniken, ist es möglich die bestehenden Implementierungen für das Verhalten der Peers schnell und einfach zu erweitern oder gänzlich neue zu entwickeln. Die Verwendung des Scalable Simulation Frameworks ermöglicht dabei eine hohe Performance durch den Zugriff auf bestehende ausgereifte Implementierungen des Simulationskerns. Insgesamt zeichnet sich die Simulationsumgebung durch ihre offene Architektur aus. Im Hinblick auf die zukünftige Entwicklung sind verschiedene Erweiterungen denkbar. Vor allem ein Visualisierungsmodul und eine Erweiterung der vorhandenen statistischen Datenerfassung wären sicherlich von Interesse. Ein wichtiger, wenn nicht der wichtigste nächste Schritt ist außerdem die Verifikation und Validation der Ergebnisse des Simulators, auf welche in dieser Arbeit nicht eingegangen wird, da es zu diesem Zeitpunkt so gut wie keine Vergleichsdaten gibt, die aber natürlich ein wichtiger Bestandteil jeder Simulation sind. Die im Rahmen dieser Arbeit entwickelten Implementierungen für Peers, Topologien und Indices sind daher auch keinesfalls als endgültig und ausgereift zu betrachten, sondern als Grundgerüste und Bausteine für die Entwicklung aussagekräftiger Simulationen zur Forschung im Bereich der schema-basierten Peer-to-Peer Netzwerke.

8 Anhang

8.1 DTDs

```

<!ELEMENT simulation (topology, superpeers, peers, summary)>
<!ATTLIST simulation
    setuptime      CDATA #REQUIRED
    simtime        CDATA #REQUIRED
    threadcount    CDATA #REQUIRED
>
<!ELEMENT topology (TTL?)>
<!ATTLIST topology
    type           CDATA #REQUIRED
>
<!ELEMENT superpeers (network, processing)>
<!ATTLIST superpeers
    type           CDATA #REQUIRED
    count          CDATA #REQUIRED
>
<!ELEMENT peers (network, processing, consumer, provider)>
<!ELEMENT network (delay, bandwidth)>
<!ELEMENT consumer (query)>
<!ATTLIST consumer
    type           CDATA #REQUIRED
    count          CDATA #REQUIRED
    livingtime     CDATA #REQUIRED
>
<!ELEMENT provider (response)>
<!ATTLIST provider
    type           CDATA #REQUIRED
    count          CDATA #REQUIRED
    livingtime     CDATA #REQUIRED
>
<!ELEMENT TTL EMPTY>
<!ATTLIST TTL
    informationmsg CDATA #REQUIRED
    query          CDATA #REQUIRED
    response       CDATA #REQUIRED
>
<!ELEMENT processing EMPTY>
<!ATTLIST processing
    poolsize       CDATA #REQUIRED
    time           CDATA #REQUIRED
>
<!ELEMENT delay EMPTY>
<!ATTLIST delay
    avg            CDATA #REQUIRED
    deviation      CDATA #REQUIRED
>...

```

Abbildung 8-1: simulation.dtd

```

<!ELEMENT bandwidth EMPTY>
<!ATTLIST bandwidth
    avg          CDATA #REQUIRED
    deviation    CDATA #REQUIRED
>
<!ELEMENT query EMPTY>
<!ATTLIST query
    limit        CDATA #REQUIRED
    burstdelay   CDATA #REQUIRED
    burstsize    CDATA #REQUIRED
    querydelay   CDATA #REQUIRED
>
<!ELEMENT response EMPTY>
<!ATTLIST response
    probability  CDATA #REQUIRED
>
<!ELEMENT summary EMPTY>
<!ATTLIST summary
    type         CDATA #REQUIRED
>

```

Abbildung 8-2: Fortsetzung simulation.dtd

```

<!ELEMENT schema (schemadistribution, superpeer, consumer)>
<!ELEMENT schemadistribution (schemas, properties, providerschemas,
providerproperties, queryproperties, zipf)>
<!ATTLIST schemadistribution
    file          CDATA #REQUIRED
>
<!ELEMENT superpeer (superpeerindex, peerindex, statistics)>
<!ATTLIST superpeer
    routingtimeout CDATA #REQUIRED
>
<!ELEMENT consumer EMPTY>
<!ATTLIST consumer
    querytimeout  CDATA #REQUIRED
>
<!ELEMENT schemas EMPTY>
<!ATTLIST schemas
    count         CDATA #REQUIRED
>
<!ELEMENT properties EMPTY>
<!ATTLIST properties
    count         CDATA #REQUIRED
    deviation     CDATA #REQUIRED
>
<!ELEMENT providerschemas EMPTY>
<!ATTLIST providerschemas
    count         CDATA #REQUIRED
    deviation     CDATA #REQUIRED
>...

```

Abbildung 8-3: schema.dtd

```

<!ELEMENT providerproperties EMPTY>
<!ATTLIST providerproperties
    count          CDATA #REQUIRED
    deviation      CDATA #REQUIRED
>
<!ELEMENT queryproperties EMPTY>
<!ATTLIST queryproperties
    count          CDATA #REQUIRED
    deviation      CDATA #REQUIRED
>
<!ELEMENT zipf EMPTY>
<!ATTLIST zipf
    shape          CDATA #REQUIRED
>
<!ELEMENT superpeerindex EMPTY>
<!ATTLIST superpeerindex
    type           CDATA #REQUIRED
>
<!ELEMENT peerindex EMPTY>
<!ATTLIST peerindex
    type           CDATA #REQUIRED
>
<!ELEMENT statistics EMPTY>
<!ATTLIST statistics
    thresholdrec   CDATA #REQUIRED
    thresholdproc  CDATA #REQUIRED
    thresholdtrans CDATA #REQUIRED
>

```

Abbildung 8-4: Fortsetzung schema.dtd

8.2 Referenzen

- [1] W3C-Consortium. W3C Semantic Web. <http://www.w3.org/2001/sw/>, 2001.
- [2] The Edutella Project. <http://edutella.ixta.org/>.
- [3] W3C Consortium. Resource Description Framework. <http://www.w3.org/RDF/>.
- [4] Ingo Brunkhorst. Mediation und Indexing in P2P-basierten Informationssystemen, Januar 2003.
- [5] Demetris Zeinalipour-Yatzi und Theodoros Folias. A Quantitive Analysis of the Gnutella Network Traffic.
- [6] Mihajlo A. Jovanovic. Modeling Large-scale Peer-to-Peer Networks and a Case Study of Gnutella, June 2000.
- [7] Jerry Banks, John S. Carson II, Barry L. Nelson und David M. Nicol. Discrete-Event System Simulation Third Edition. Prentice Hall, 2001.
- [8] Sun Microsystems. <http://java.sun.com/>.
- [9] The Scalable Simulation Framework. <http://www.ssfnet.org/>.
- [10] Renesys Corporation. <http://www.renesys.com/>.
- [11] James H. Cowie. Scalable Simulation Framework API Reference Manual, 1999.

- [12] Dublin Core Metadata Initiative. Dublin Core Metadata Element Set, Version 1.1. <http://dublincore.org/documents/dces/>.
- [13] Wentian Li. References on Zipf's law. <http://linkage.rockefeller.edu/wli/zipf/>.
- [14] Mario Schlosser, Michael Sintek, Stefan Decker und Wolfgang Nejdl. HyperCuP – Hypercubes, Ontologies and Efficient Search on P2P Networks.
- [15] Sun Microsystems. JavaBeans Specification. <http://java.sun.com/products/javabeans/docs/spec.html>
- [16] Marc Herrlich. API Dokumentation der Simulationsumgebung. <http://www.learninglab.de/~herrlich/research/sim/api/>.
- [17] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario Schlosser, Ingo Brunkhorst und Alexander Löser. Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. 12th International World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003.