Technische Universität Darmstadt
Fachbereich Informatik

Implementierung eines Peer-to-Peer Wikis

Bachelorarbeit von
Markus Günther

Betreuer: Dipl.-Inform. Christof Leng

Eingereicht bei: Prof. Alejandro P. Buchmann, Ph. D.
Fachgebiet Datenbanken und Verteilte Systeme
Fachbereich Informatik
TU Darmstadt

31. März 2008

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 31.März 2008 _____

Danksagungen

Ich möchte mich an dieser Stelle zunächst ganz herzlich bei Christof Leng für sein Stück des Weges bedanken. Er hat mir durch diese Arbeit ermöglicht, einen tiefgreifenden Einblick in ein faszinierendes Themengebiet zu erhalten. Die zahlreichen Diskussionen habe ich sehr geschätzt und nicht zuletzt sind auch einige seiner Hinweise in diese Arbeit eingeflossen.

Des Weiteren möchte ich mich bei den Teilnehmern des Middleware-Praktikums im Sommersemester 2007 bedanken. Die dort gestellte Aufgabe war es, ein Peer-to-Peer Wiki auf Basis eines strukturierten Peer-to-Peer Systems zu entwickeln. Obwohl jene Aufgabe sehr stark von dieser Arbeit differiert, waren die Abgaben der einzelnen Gruppen gerade zu Beginn meiner Arbeit ein steter Quell an Inspiration für mich.

Ebenfalls zu Dank verpflichtet bin ich Paul Bächer und Stephan Mehlhase. Sie haben diese Arbeit auszugsweise gelesen und mich auf Unzulänglichkeiten und mögliche Erweiterungen hingewiesen.

Zu guter Letzt möchte ich mich bei den Menschen bedanken, die mir nahe stehen. Meinen Eltern, für die sicher nicht selbstverständliche finanzielle Rückendeckung. Und nicht zu vergessen Alisa, die mich während der Anfertigung dieser Bachelorarbeit leider viel zu oft vermissen musste, aber mich dennoch mit viel Geduld und Standhaftigkeit daran erinnert hat, dass es auch noch ein Leben jenseits der Arbeit gibt.

Zusammenfassung

Wiki-Applikationen haben sich zu populären Kollaborationsplattformen für das verteilte Schreiben im Internet entwickelt. Die Mehrheit der verfügbaren Implementierungen basiert auf dem Client-Server-Konzept, was unter Umständen eine Reihe von Problemen mit sich bringt, beispielsweise hinsichtlich der Kosten für eine adäquate Skalierbarkeit des Systems. Diese Arbeit beschäftigt sich daher mit der Realisierung einer Peer-to-Peer basierten Wiki-Applikation, deren Basis durch eine prototypische Implementierung des am Fachgebiet entwickelten BubbleStorm-Netzwerks gebildet wird.

Das vorrangige Ziel dieser Arbeit ist es, die Entwicklung von verteilten Anwendungen mit dem Peer-to-Peer System BubbleStorm praktisch zu erproben. In Anbetracht der Realisierung einer Wiki-Applikation stellt sich eine weitere Herausforderung durch die Integration einer rudimentären, aber praxistauglichen Versionskontrolle.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einleitung

> Ein ungeprüftes Leben ist für den Menschen nicht lebenswert.
>
> *Sokrates*

Das Internet erfährt seit nun mehr als drei Jahrzehnten eine stetige Evolution, vorangetrieben durch neue Technologien, die den Zugriff auf verteilte Inhalte vereinfachen und die Kommunikation der Benutzer untereinander fördern. Die Entwicklung der letzten Jahre zeigt einen Trend, der den Benutzer selbst in den Mittelpunkt der Informationsbeschaffung stellt und präsentiert Systeme, die es Teilnehmern erlauben in gleichberechtigter Art und Weise Informationen zu offerieren und zu erhalten. Das Paradigma dieser Entwicklung lässt sich unter den Begriffen *user generated content* und *Web 2.0* zusammenfassen. Kollaborationsplattformen, wie beispielsweise Wikipedia, sind der beste Beweis für die hohe Akzeptanz und den Erfolg derartiger Dienste. Sie schaffen Raum für die Selbstbestimmung der Benutzer, indem teilnehmenden Individuen Möglichkeiten an die Hand gegeben werden, über die Veröffentlichung und den Austausch eigener Inhalte selbst zu entscheiden.

Eine Vielzahl dieser Dienste werden auf Basis von Client-/Server Architekturen realisiert. Konträr zu diesem Modell bietet das *Peer-to-Peer Computing* Lösungen, die einerseits robuster gegen Fehler sind und andererseits kostengünstiger skalieren. Die Entwicklung über die letzten Jahre in diesem Bereich hat durchaus gezeigt, dass Peer-to-Peer Systeme eine praktikable Alternative sind, um verteilte Dienste zu entwickeln. Die wesentlichen Aspekte dieser Systeme lassen sich kurz zusammenfassen. Ein Peer agiert gleichzeitig als Anbieter (engl. provider) und Anforderer (engl. requester). Zwischen einzelnen Peers können Daten direkt ausgetauscht werden. Jeder Peer hat Zugriff auf den verteilten Datenbestand, kann also implizit direkt auf die Daten anderer Peers zugreifen.

Grundsätzlich lassen sich Peer-to-Peer Systeme anhand der Struktur des zugrunde liegenden Netzwerks kategorisieren. Unstrukturierte Systeme basieren auf Netzwerken, die ohne ausgezeichnete Anordnung der einzelnen Peers funktionieren. Die erste Generation solcher Systeme verfolgt den Ansatz, die Kommunikation mit anderen teilnehmenden Peers über einen zentralen Index-Server zu steuern (hybrides Peer-to-Peer). Dieser Server kapselt Metadaten zu den Daten, die von den Peers angeboten werden[1]. Die nachfolgende Generation präsentierte einen Ansatz, der gänzlich ohne zentralen Server auskommt und die angesprochenen Aspekte von Peer-to-Peer in ihrer Reinform umsetzt (reines Peer-to-Peer). Kommunikation erfolgt hierbei typischerweise über Flooding-basierte Algorithmen. Beiden Ansätzen ist gemein, dass die Sys-

[1] Napster ist wohl der erste und bekannteste Vertreter dieser Kategorie.

teme bei zunehmender Anzahl von Peers äußerst schlecht skalieren. Dies förderte die weitere Entwicklung unter der Motivation, die Widrigkeiten unstrukturierter Systeme zu eliminieren und folgte schließlich zu einem strukturierten Ansatz, der über verteilte Hashtabellen - kurz DHTs - realisiert wird. Strukturierte Systeme adressieren die Performanzprobleme ihrer unstrukturierten Pendants und zeichnen sich durch hohe Effizienz aus. Die gewonnene Effizienz bringt allerdings weitere Probleme mit sich. Aufgrund der Natur von Hashtabellen werden Daten basierend auf einem *key-value*-Paar eingepflegt. Daten können folgerichtig sehr effizient adressiert werden[2], jedoch gestaltet sich die Evaluierung einer komplexen Suchanfrage als äußerst schwierig. Beispielsweise können invertierte Indizes dazu genutzt werden, um eine Volltextsuche zu ermöglichen. Allen Ansätzen ist jedoch gemein, dass sie eine zu hohe Komplexität in die Evaluation einbringen, so dass eine Realisierung nicht praxistauglich ist.

Gegenstand aktueller Forschungsarbeit ist es, die Möglichkeiten sowohl des strukturierten als auch des unstrukturierten Ansatzes weiter zu untersuchen. In diesem Kontext stellen die Arbeiten von Terpstra, Kangasharju, Leng und Buchmann das unstrukturierte Peer-to-Peer-System *BubbleStorm* vor [16, 17]. Dieses System basiert auf einer Reihe interessanter Lösungsvorschläge zu den bereits angeführten Problemen. Das Ziel der vorliegenden Arbeit ist es, die prototypische Implementierung von BubbleStorm in Bezug auf die Anwendungsentwicklung zu betrachten. Konkret soll dabei ein verteiltes Wiki realisiert werden. Der Fokus liegt einerseits auf der Konzeption der Anwendung, andererseits soll gleichermaßen herausgestellt werden, inwiefern spezifische Features wie Versionsmanagement und Volltextsuche, profitierend durch die unterliegende Netzwerkstruktur, implementiert werden können.

Im Folgenden soll zunächst die Motivation für diese Arbeit dargelegt werden (Kapitel 2). Die anschließende Diskussion verwandter Arbeiten betrachtet Problemstellungen und Lösungsansätze für Aktualisierungen von Daten innerhalb eines Peer-to-Peer Systems (Kapitel 3). Eine Vorstellung des Peer-to-Peer Systems BubbleStorm (Kapitel 4) rundet den einführenden Teil dieser Arbeit ab. Darauf aufbauend soll eine Anforderungsanalyse für die Entwicklung eines verteilten Wikis (Kapitel 5) präsentiert werden, bevor auf die Möglichkeiten von Versionsmanagementsystemen mit Bezug auf die Integration in ein Wiki (Kapitel 6) detailliert eingegangen wird. Erforderlich für die Netzwerkkommunikation ist eine gemeinsame Basis von Nachrichten, die in Form eines einfachen Applikationsprotokoll vorgestellt werden (Kapitel 7). Nachfolgend wird das Design der Wiki-Applikation besprochen (Kapitel 8). Die daraus resultierende Referenzimplementierung bildet die Basis für eine abschließende Evaluation (Kapitel 9). Den Abschluss dieser Arbeit bildet die kritische Diskussion der Ergebnisse sowie ein Ausblick auf mögliche Erweiterungen und künftige Entwicklungen in diesem Segment (Kapitel 10).

[2] Der Lookup eines Datums liegt in der Komplexität O(log n).

Kapitel 2

Motivation

> Die Neugier steht immer an erster Stelle eines Problems, das gelöst werden will.
>
> *Galileo Galilei*

Client-/Server-basierte Architekturen skalieren hervorragend, wenn eine hinreichend große Anzahl von Servern bereitgestellt werden kann, welche jedoch teuer erkauft werden müssen. Im Gegensatz hierzu bieten Peer-to-Peer Systeme eine Kompromisslösung zwischen Kosten und Nutzen und sind ferner leicht an die Gegebenheiten einer bestimmten Applikation anzupassen. Aus der kostengünstigen Umsetzung und der robusten Struktur resultiert nicht zuletzt die Attraktivität, die für die hohe Beliebtheit und Akzeptanz solcher Systeme sorgt.

Die vorgestellten Ansätze der strukturierten und unstrukturierten Systeme bieten aber dennoch eine Reihe von Nachteilen, die sie für Netzwerke ab einer bestimmten Größe unrentabel erscheinen lassen. Dieser Umstand wird im Wesentlichen durch die zur Verfügung stehenden Suchstrategien beeinflusst, da sie verantwortlich für den Erfolg einer konkreten Anwendung sind. Der Aufwand, ein gesuchtes Datum in einer Menge von Datensätzen über einer verteilten Datenbasis zu finden und daraus zu extrahieren, muss effizient und damit praktikabel sein. Dass dieses Problem nach einer nicht-trivialen Lösung verlangt, zeigen die vielen forschungsorientierten Arbeiten, die sich dieser Aufgabe gewidmet haben. Die Etablierung von strukturierten Systemen mittels DHT ermöglichte die Entwicklung von skalierbaren Algorithmen, die deutliche Effizienzvorteile gegenüber dem naiven Flooding von Anfragen, wie es in unstrukturierten Systemen geläufig ist, aufzeigen. Dies führt jedoch zu einem weiteren Problem, denn eine derartige Struktur ist anfällig gegen den Wegfall von Peers oder anderweitigen Ursachen, die den Netzbetrieb stören oder gar schädigen. Ferner sind durch die strikte Kopplung von Netzwerktopologie und Datenstruktur nur eingeschränkte Abfragen möglich.

Die Idee, die letzten Endes auch zu der Ausgestaltung von BubbleStorm führte, ist es, die Suchstrategie unabhängig von der eingesetzten Netzwerktopologie zu verwirklichen. Es obliegt demnach der Entscheidung des Anwendungsentwicklers, in welcher Form eine Suchstrategie implementiert wird. Diese Strategie kann dabei beliebig komplex ausfallen und reicht von einfachem, exaktem Matching über Suchschlüssel bis hin zu der Integration einer ausgereiften Volltextsuche.

Ein Prototyp des Peer-to-Peer Systems BubbleStorm wurde bereits implementiert und ist lauffähig. Die Motivation dieser Arbeit ist es, diesen Prototypen praktisch zu erproben. Im Zuge dessen soll eine einfache verteilte Applikation in Form eines Wikis realisiert werden, die Bubble-

Storm als zugrundeliegende Netzwerkkomponente nutzt. Die Arbeitsweise eines Wikis erfordert es ferner, eine adäquate Versionierung in die Applikation zu integrieren. Dies steht konträr zu den bisherigen Erfahrungen im Umgang mit Peer-to-Peer Systemen, die sich größtenteils auf die Verteilung statischer Inhalte beschränken. Die Unterstützung für dynamische Inhalte stellt aus diesem Grund eine weitere Herausforderung dar, für die im Zuge dieser Arbeit eine rudimentäre, praxistaugliche Lösung gefunden werden soll.

Kapitel 3

Verwandte Arbeiten

> The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.
>
> *Sir William Bragg*

In Kapitel 2 wurde bereits angedeutet, dass die meisten Peer-to-Peer Systeme Daten verteilen, die unveränderlich sind. Für viele Anwendungsszenarien - wie bespielsweise verteilten Kalendern, Peer-to-Peer basierten Foren oder Wikis - genügt dies jedoch nicht [6]. Diesen Szenarien liegt ferner die Annahme zugrunde, dass Daten häufig geändert werden. Um eine möglichst hohe Verfügbarkeit von Daten zu erhalten, wählen viele Peer-to-Peer Systeme den Ansatz, Daten redundant im Netzwerk zu verteilen (Replikation[3]). Replizierende Peer-to-Peer Systeme müssen jedoch nicht nur dafür sorgen, dass diese Daten eine hohe Verfügbarkeit aufweisen, sondern auch sicherstellen, dass Änderungen an den Inhalten eines Datums jeden verantwortlichen Peer erreichen, damit eine konsistente Datenhaltung ermöglicht werden kann. In diesem Kontext werden Arbeiten diskutiert, die sich grundlegend mit veränderlichen (versionierten) Daten in replizierenden Peer-to-Peer Systemen beschäftigen.

3.1 Datta et al.

Datta et al. beschäftigen sich in [6] mit Änderungsanfragen in unzuverlässigen und replizierenden Peer-to-Peer Systemen. Die Grundannahmen in diesem Szenario sind ein hoher Replikationsfaktor, die geringe Wahrscheinlichkeit der Online-Verfügbarkeit von einzelnen Peers sowie die Tatsache, dass generell kein globales Wissen über den Netzwerkstatus von einzelnen Peers aggregiert werden kann. Um diesen Umständen gerecht zu werden, entfernt sich die Arbeit von strikter Konsistenz und bewegt sich hin zu statistischen Zusicherungen, die innerhalb dieses verteilten Szenarios durchaus als ausreichend betrachtet werden können. Der vorgeschlagene Algorithmus basiert auf dem sozialen Phänomen des *rumor spreading*, welches von der technischen Seite betrachtet durch modifizierte Flooding-Algorithmen umgesetzt wird. Der Algorithmus verfolgt hierbei einen hybriden push/pull-Ansatz, was es Peers, die über längere Zeit offline gewesen sind, ermöglichen soll, nach dem Wiedereintritt in ein Peer-to-Peer System nach Aktualisierungen zu fragen. Für die Verbreitung von Updates muss jeder Peer eine Teilmenge der replizierenden Peers (RP) kennen, die ebenfalls eine Originalversion des selben Datums besitzen.

[3] Replikation steht nicht im Fokus dieser Arbeit und wird daher nur erwähnt, aber nicht weiter betrachtet.

3.1.1 Funktionsweise

Die Verbreitung von Updates arbeitet prinzipiell in konsekutiven push- und pull-Phasen, die sich zeitlich betrachtet jedoch überschneiden können. Während der rundenbasierten push-Phase kann ein Peer p ein Update der Form (U, V, R_f, t) erhalten, wobei U dem aktualisierten Datum entspricht, V die derzeitige Version angibt, R_f einer Menge von Peers repräsentiert, die das Update bereits erhalten haben und t angibt, in welcher Runde sich das push-Verfahren befindet. Im Folgenden wählt p unter einer Wahrscheinlichkeitsfunktion eine zufällige Teilmenge aus den ihm bekannten, replizierenden Peers R_p und sendet ein Update der Form $(U, V, R_f \cup R_p, t + 1)$ an die Menge der Peers $R_p \setminus R_f$. Die Terminierung des push-Verfahrens ergibt sich implizit aus der trivialen Anforderung, dass jeder RP nur ein einziges Mal ein Update weitergibt. Sobald ein Peer erneut dem Netzwerk beitritt oder eine pull-Anfrage bekommt und sich nicht sicher ist, ob seine lokalen Daten aktuell sind, kann er in die pull-Phase wechseln, um seinen lokalen Datenbestand mit anderen RPs zu synchronisieren.

3.1.2 Diskussion

Die push-Phase des vorgestellten Algorithmus ähnelt in ihren Ansätzen dem Flooding-Algorithmus des Peer-to-Peer Systems Gnutella, welches bekanntermaßen in relativ umfangreichen Netzwerkstrukturen nicht mehr hinreichend skaliert [13, 14]. Dadurch, dass die Verbreitung in einer push-Runde nur an eine Teilmenge von replizierenden Peers erfolgt, wird der Overhead an Nachrichten im Netzwerk limitiert, was die Effizienz und Skalierbarkeit des Algorithmus natürlich steigert. Jedoch ist der Overhead nach wie vor als signifikant einzustufen. Des Weiteren wird in [6] nicht diskutiert, wie die Aufrechterhaltung der Teilmenge von replizierenden Peers konkret erfolgen kann.

3.2 Wang et al.

Die Arbeit [18] beschäftigt sich ebenfalls mit Änderungsanfragen in unzuverlässigen und replizierenden Peer-to-Peer Systemen. Die Grundannahmen über der Netzwerkstruktur und das Verhalten einzelner Peers sind ähnlich gewählt wie in Abschnitt 3.1. Der Algorithmus von Wang et al. verfolgt ebenfalls einen push/pull-Ansatz, erreicht aber eine weitere Reduktion des Overheads an Nachrichten. Grundlegende Unterschiede zu [6] basieren im Wesentlichen auf der Vorgehensweise: Für jede Datei, die im Netzwerk repliziert wird, existiert eine sogenannte Replikationskette, die konstruiert und verwaltet werden muss. Die Replikationskette kann als Ansammlung von replizierenden Peers verstanden werden, die ebenfalls diese Datei speichern. Updates können demnach darüber erfolgen, dass eine Änderung (ausgehend von einem RP), entlang der Replikationskette weitergereicht wird.

3.2.1 Funktionsweise

Zu jeder replizierten Datei existiert eine bidirektionale Replikationskette, die aus N verantwortlichen RPs - die entweder on- oder offline sein können - besteht. Zu jedem RP wird eine eindeutige ID ($1 \leq ID \leq N$) und die aktuelle IP-Adresse gespeichert. Ein RP kennt lediglich die

k-nächsten RPs in jede Richtung, die in diesem Kontext auch als *probe nodes* bezeichnet werden. Ein solcher Peer muss demnach nur über ein partielles Wissen der Replikationskette verfügen. Sofern ein RP i eine Aktualisierung der replizierten Datei vorgenommen hat, initiiert er den push-Prozess in beide Richtungen entlang der Replikationskette. RP i reicht das Update mittels einer sogenannten *probe message* jedoch nur an seine probe nodes weiter. Die am weitesten entfernte online probe node (einer Richtung) hat die Aufgabe, das Update an ihre probe nodes entlang der selben Richtung weiterzureichen. Die restlichen probe node von RP i empfangen das Update lediglich, senden es jedoch nicht weiter. Die maximale Anzahl von Nachrichten, um ein Update zu propagieren, liegt somit in der Anzahl der RPs zu der replizierten Datei (also N), da jeder Peer in der Replikationskette höchstens eine probe message empfängt. Sofern ein RP über einen gewissen Zeitraum offline gewesen ist, kann er etwaige Updates sowie Änderungen an der Struktur der Replikationskette (beispielsweise wenn sich die IP-Adresse einer anderen RP, die innerhalb der k-nächsten RPs liegt, geändert hat) versäumt haben. Sobald dieser Peer erneut online ist, muss er sich aus diesem Grunde mit anderen online RPs synchronisieren. Hat sich die IP-Adresse des wieder eingetretenen RP geändert, so muss die neue IP-Adresse mitsamt der ID dieses RP zu all seinen probe nodes gesendet werden, um die Konsistenz der Replikationskette zu wahren.

3.2.2 Diskussion

Die Arbeit von Wang et al. kann durchaus als Verbesserung des ersten Ansatzes von Datta et al. [6] verstanden werden. Die Struktur der Replikationskette ermöglicht es, dass ein Update nur über eine probe node weitergereicht werden muss. Dies reduziert den Overhead an Nachrichten deutlich[4]. Die richtige Wahl für k probe nodes einer Richtung ist jedoch entscheidend für diese Einsparung und letzten Endes auch den praktischen Erfolg des Algorithmus. Wird k zu klein gewählt, so kann es unter Umständen passieren, dass das Update nicht an alle probe nodes weitergereicht werden kann, weil diese nicht online sind. Wird k zu groß gewählt, so überwiegt die Aufrechterhaltung der Replikationskette, da diese bei einer Änderung $2k$ Nachrichten benötigt, um wieder einen konsistenten Zustand zu erreichen.

[4] Bis zu 70%, vgl. [18].

Kapitel 4

BubbleStorm

> Einfachheit ist der Schlüssel zu erfolgreicher wissenschaftlicher Forschung.
>
> *Stanley Milgram*

Das Peer-to-Peer System BubbleStorm bildet die Grundlage für die Netzwerkebene des zu implementierenden Wikis und soll innerhalb dieses Kapitels in seiner Funktionsweise beschrieben werden. BubbleStorm induziert ein Overlay-Netzwerk, dessen Struktur einem randomisierten Multigraphen gleicht. Im Gegensatz zu bisherigen Systemen ermöglicht es eine erschöpfende Suche über das Overlay-Netzwerk und unterstützt Abfragen beliebiger Komplexität, deren Evaluation grundsätzlich von der vorherrschenden Netzwerkstruktur getrennt und somit ohne zusätzliche Indirektionsebenen möglich ist. Das System greift das sogenannte *Rendezvous*-Problem auf und bietet eine Lösung dafür an, die statistische Zusicherungen über die erfolgreiche Beantwortung einer Abfrage liefert.

Dieses Kapitel thematisiert die Grundannahmen, die BubbleStorm voraussetzt und gibt Aufschluss über das Suchverhalten und den Replikationsmechanismus des Systems. Die Diskussion der Programmierschnittstelle, die für Anwendungsentwickler bereitgestellt wird, um eigene Applikationen auf BubbleStorm aufzusetzen, runden die Ausführungen zu diesem Kapitel ab. Eine ausführliche Systembeschreibung nebst Simulation und analytischen Ergebnissen finden sich in [16, 17].

4.1 Suchverhalten

Das statistische Modell von BubbleStorm basiert auf dem Rendezvous-Problem. Dieses Problem entstammt der mathematischen Spieltheorie und stellt die Frage auf, was die beste Strategie ist, so dass sich zwei Personen an einem bestimmten Ort gegenseitig auffinden können. Die Frage nach der besten Strategie kann unter Zuhilfenahme des Geburtstagsparadoxon gelöst werden. Das Paradoxon besagt, dass bei zwei zufällig gewählten Personengruppen die Wahrscheinlichkeit, dass zwei Personen am gleichen Tag Geburtstag haben (unter Vernachlässigung des Jahrgangs), entgegen einer intuitiven Anname sehr hoch ist. Selbst bei Personengruppen geringer Mächtigkeit (ca. 25 Menschen) lässt sich dieses Verhalten bereits mit einer Wahrscheinlichkeit von $p > 0.5$ beobachten.

Im Kontext des Rendezvous-Problems betrachte man nun die Verteilung von Abfragen und Da-

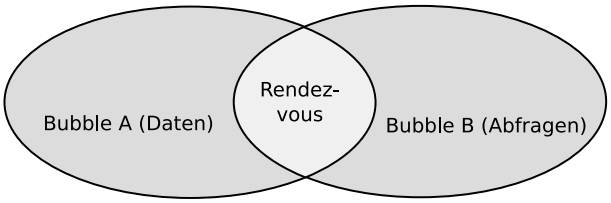


Abbildung 4.1: Der Schnitt beider Bubbles wird als Rendezvous bezeichnet

ten in BubbleStorm. Sofern ein Peer ein neues Datum (Nachrichtentyp A) über das Netzwerk einpflegt, repliziert das System dieses Datum auf eine zufällig gewählte Menge von Knoten, in dem Nachrichten vom Typ A verteilt werden. In Analogie zu dem Geburtstagsparadoxon lässt sich festhalten, dass jede Replika einer Person und jeder Knoten einem Geburtstag entspricht. Gleiches gilt für jede Abfrage, die über das Netzwerk verteilt wird (Nachrichtentyp B). Basierend auf der Annahme des Paradoxon ist es nun wahrscheinlich, dass im Netzwerk Knoten existieren, die sowohl Nachrichtentyp A als auch Nachrichtentyp B empfangen haben und somit die Abfragen gegen die empfangenen Daten evaluieren können. Die beste Strategie um das Rendezvous-Problem zu lösen, liegt folglich darin, für eine hinreichende und zufällige Verteilung von Daten und Abfragen im Netzwerk zu sorgen.

BubbleCast ist eine neue Kommunikationsprimitive, die von BubbleStorm genutzt wird, um dieses Problem zu lösen. Die Aufgabe des BubbleCast ist es, Informationen - wie beispielsweise Daten oder Abfragen - im Netzwerk zu replizieren. Um dies zu gewährleisten bedient sich ein BubbleCast der Vorteile von Random Walks und Flooding, jedoch ist es aus Performanzgründen notwendig, die Explorationstiefe zu kontrollieren. Ein hinreichendes Kriterium hierfür ist das Verhältnis von gewünschter Replika-Anzahl in Abhängigkeit zum exponentiellen Verteilungsgrad der korrespondierenden Nachricht. Ein BubbleCast hat somit ein Gewicht w, welches die Anzahl der Replika repräsentiert, sowie einen Teilungsgrad s, der angibt an wieviele Nachbarn eines Knoten die Nachricht weitergereicht wird. Die Information bezüglich des Gewichts wird dabei lokal aktualisiert. Das heißt, wenn der ursprüngliche Knoten mit $w = 18$ beginnt und der Verteilungsgrad $s = 3$ ist, dann zieht dieser Knoten den Wert 1 von w ab und teilt anteilsmäßig (entsprechend des Verteilungsgrads) um die verbleibenden Gewichte für die Nachbarknoten zu ermitteln. Abbildung 4.2 illustriert diese Vorgehensweise und verdeutlicht, dass der BubbleCast einen Subgraphen über der Netzwerkstruktur induziert. Dieser Subgraph wird im Kontext von BubbleStorm als *Bubble* bezeichnet. Grundsätzlich existieren zwei unterschiedliche Klassen für Bubbles: Solche, die durch einen BubbleCast induziert werden, der Daten im Netzwerk repliziert (Bubble-Klasse A) und solche, die durch einen BubbleCast induziert werden, der Abfragen im Netzwerk verteilt (Bubble-Klasse B). Damit die Überschneidung zweier Bubbles verschiedener Klassen unter einer gewünschten Wahrscheinlichkeit auftritt, werden BubbleCast-Nachrichten des Weiteren mit einem sogenannten Certainty-Wert c parametrisiert. In [16] wird gezeigt, dass bereits für $c = 2$ die Wahrscheinlihckeit einer Überschneidung sehr groß ist ($\approx 98.2\%$). Dabei hängen Netzwerktraffic und Certainty-Wert proportional voneinander ab, das heißt, um die statistische Zusicherung für einen höheren c-Wert zu gewährleisten, muss ebenso erhöhter Netzwerktraffic aufgewendet werden. Darüberhinaus lässt sich das Größenverhältnis einer Bubble über einen sogenannten Balance-Faktor bestimmen, mit dem eine BubbleCast-Nachricht parametrisiert wird. Um dies zu verdeutlichen, stelle man sich folgendes Szenario vor, in dem eine Menge von Netzwerkknoten mit fixer Anzahl vorliegt und zwei BubbleCasts initiiert wurden,

die unterschiedliche Bubble-Klassen A und B induzieren. Die Balance-Faktoren seien $b_A = 1$ und $b_B = 1$ für die jeweilige Bubble-Klasse. In diesem Fall werden zwei Bubbles induziert, die in etwa die gleiche Anzahl von Netzwerkknoten abdecken sollten. Betrachte man nun den Fall, dass diese Werte unterschiedlich gewählt werden, beispielsweise $b_A = 2$ und $b_B = 0.5$. Hier würde die Bubble von Klasse A im Verhältnis zur Bubble von Klasse B deutlich mehr Netzwerkknoten abdecken. Je nach Anwendungsszenario kann die Wahl dieser Werte zu Resultaten unterschiedlicher Qualität führen. Denkbar ist auch, dass die Anwendung die Balance-Faktoren einzelner BubbleCast-Nachrichten in Anbetracht des vorherrschenden Netzwerkstatus dynamisch anpasst. Zu beachten gilt jedoch, dass $b_A \times b_B$ stets 1 ergeben sollte [16].

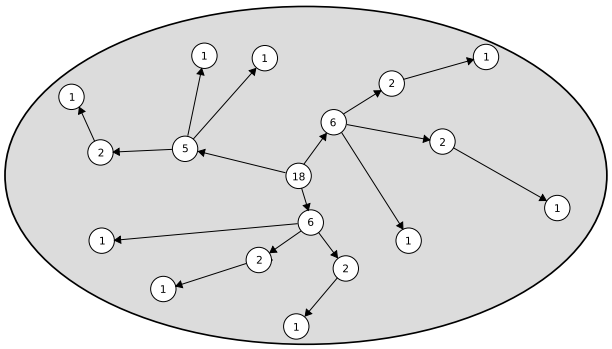


Abbildung 4.2: BubbleCast mit $w = 18$ und $s = 3$

4.2 Replikation

Der Replikationsalgorithmus von BubbleStorm ist für die aktive Stabilisierung von einer angestrebten Anzahl an Replikaten für ein Datum zuständig. Ausgehend von einer Initialverteilung, die durch die auf BubbleStorm aufsetzende Applikation erzeugt wird, ist es das Ziel des Algorithmus, diese angestrebte Anzahl unter Einfluss von dynamischen Änderungen an der Netzwerkstruktur (bedingt durch permanentes Verlassen und Beitreten von Peers) zu erreichen. Die Merkmale des Algorithmus sind, dass er vollkommen dezentral arbeitet, also kein Meta-Wissen über das Netzwerk und einzelnen Peers aggregiert und das kein designierter Besitzer für replizierte Daten gewählt wird. Entscheidend für den Erfolg ist, dass die aufgesetzte Applikation klar trennt, welche Daten in der aktuellen Sitzung repliziert werden dürfen. Geschieht diese Kontrolle nicht, kann es zu einer Überbevölkerung eines Replikats kommen, da zwischenzeitliche Verluste von dem Replikationsalgorithmus bereits ausgeglichen sein könnten. Eine detaillierte Beschreibung des Algorithmus nebst zugrunde liegendem mathematischen Modell und Evaluation findet sich in [1].

4.3 Programmierschnittstelle

Dieser Abschnitt widmet sich der Programmierschnittstelle (Application Programming Interface, kurz API), die in erster Linie für Anwendungsentwickler interessant ist und benötigt wird,

um Applikationen auf Basis von BubbleStorm zu implementieren. Die API definiert verschiedene Schnittstellen, die sich grob in die Verantwortungsbereiche Netzwerkmanagement und Applikationsprotokoll einteilen lassen und unabhängig voneinander diskutiert werden. Die in diesem Abschnitt angesprochenen Schnittstellen werden ausführlich beschrieben und durch konkrete Ausprägungen exemplarisch in Form von Quelltext erläutert. Die Ausführungen beziehen sich auf den zum Zeitpunkt der Anfertigung dieser Arbeit aktuellen Versionsstand des BubbleStorm Prototypen und lehnen sich an die entsprechenden Beschreibungen in [15] an.

4.3.1 Netzwerkmanagement

Das Interface `IRouterService` definiert die Schnittstelle zu dem Routing-Mechanismus von BubbleStorm, der für die Verwaltung und Kommunikation der Anwendung als Knoten in einem BubbleStorm-Netzwerk verantwortlich ist. Um die Anwendung als BubbleStorm-Knoten zu repräsentieren, genügt es, eine Instanz von `RouterService` zu erzeugen, die unter dem statischen Typ dieser Schnittstelle erscheint.

Nach dem Erzeugen des Knotens ist es möglich, über die Schnittstelle grundlegende Operationen wie Beitreten des Netzwerks, Verlassen des Netzwerks, Versenden von BubbleCast-Nachrichten und das Erzeugen von Benutzerverbindungen auszuführen.

Einem BubbleStorm-Netzwerk beitreten

Die Schnittstellte `IRouterService` definiert mehrere Möglichkeiten, einem Netzwerk beizutreten. Grundsätzlich kann ein Knoten auch als *Client*[5] beitreten.

Der Aufruf der Methode `join(boolean client)` bewirkt, dass der lokale Hostcache[6] genutzt wird, um einem BubbleStorm-Netzwerk beizutreten. Ferner besteht die Möglichkeit, `join(String file, boolean client)` zu nutzen. Der Wert des übergebenen String-Parameters wird als Dateiname interpretiert, der einen lokalen Hostcache repräsentiert. Die dritte Möglichkeit besteht darin, `join(URL url, boolean client)` zu benutzen. In diesem Fall repräsentiert die URL die Adresse zu einem entfernten Hostcache. Zu guter Letzt kann der Netzwerkbeitritt durch die direkte Angabe einer Host-Adresse erfolgen, indem `join(InetSocketAddress addr, boolean client)` genutzt wird.

Sofern die Angabe einer Host-Adresse oder das Nutzen eines bereits vorhandenen Hostcaches nicht dazu führt, dass der Netzwerkbeitritt stattfindet, so erzeugt die Anwendung ein eigenes BubbleStorm-Netzwerk.

[5] Jeder Peer in einem BubbleStorm-Netzwerk agiert als gleichberechtigter *super node*. Einem super node können jedoch mehrere Clients zugeordnet werden. Der Beitritt als Client ist beispielsweise dann sinnvoll, wenn ein Host aufgrund einer schwachen Netzwerkanbindung nicht die volle Arbeitslast eines super node handhaben kann.

[6] Der Hostcache ist eine Sammlung von bekannten Hosts, die in einem BubbleStorm-Netzwerk agieren.

Ein BubbleStorm-Netzwerk verlassen

Die Schnittstellte `IRouterService` stellt die Methode `leave()` bereit, die das leave-Protokoll von BubbleStorm initiiert, so dass der Knoten korrekt aus dem Netzwerk entfernt wird.

4.3.2 Applikationsprotokoll

Typischerweise nutzen Anwendungen, die auf BubbleStorm aufsetzen, BubbleCasts um Anfragen im Netzwerk zu verteilen. Diese Anfragen werden für gewöhnlich per Unicast beantwortet. Der Unicast wird innerhalb des Systems durch sogenannte Benutzerverbindungen realisiert, auf die separat eingegangen wird.

4.3.2.1 BubbleCast-Nachrichten

Um die Vorzüge von BubbleStorm zu nutzen und BubbleCast-Nachrichten in der darüberliegenden Applikation zu verwenden, existieren die Interface-Klassen `IBSTypeHandler` und `IBubbleCastMessageType`. Diese Schnittstellen stellen grundlegende Methoden bereit, um BubbleCast-Nachrichten zu verschicken und zu verarbeiten. Jeder BubbleCast-Nachrichtentyp muss durch Realisierung dieser beiden Schnittstellen implementiert werden.

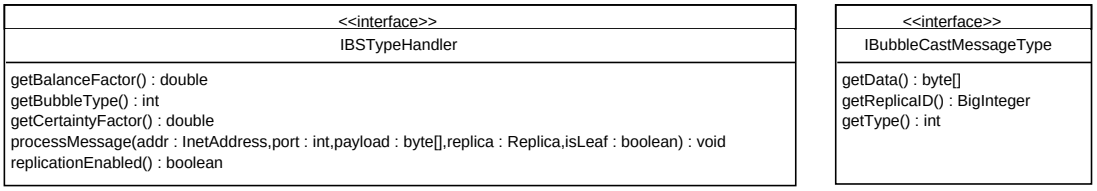


Abbildung 4.3: Schnittstellen zur Verwendung von BubbleCast-Nachrichten

Die Methode `bubblecast(IBubbleCastMessageType msg)` der Klasse `RouterService` ist für das Verteilen von BubbleCast-Nachrichten im Netzwerk verantwortlich. Dazu muss die konkrete Realisierung der Schnittstelle `IBubbleCastMessageType` für einen spezifische BubbleCast-Nachrichtentyp existieren. Diese Realisierung wird von der Methode unter dem statischen Typ der Schnittstelle konsumiert. Jeder Nachrichtentyp sollte seinen eigenen Bubble-Typen zugeordnet bekommen. Dies geschieht durch Zuweisung eines applikationsweit eindeutigen Bezeichners in Form eines Integer-Wertes zu jedem Nachrichtentyp. Nachfolgendes Listing demonstriert die beispielhafte Umsetzung eines Nachrichtentypes unter Verwendung des angesprochenen Interfaces.

```
1 public class GenericQueryMessage implements IBubbleCastMessageType {
2     private String payload;
3
4     public GenericQueryMessage(String payload) {
5         this.payload = payload;
6     }
7
```

```
8       public byte[] getData() {
9           Charset c = Charset.forName("UTF-8");
10          return c.encode(payload).array();
11      }
12
13      public BigInteger getReplicaID() {
14          Random r = new Random(System.currentTimeMillis());
15          return new BigInteger(Constants.replicaIdentifierBytes * 8 - 1, r);
16      }
17
18      public int getType() {
19          return 3;
20      }
21  }
```

Listing 4.1: Beispielhafte Umsetzung einer Nachrichtenklasse

Um empfangene BubbleCast-Nachrichten weiterzuverarbeiten, ist es notwendig einen Handler für jeden genutzen Nachrichtentypen einer Instanz von `RouterService` zuzuordnen. Dies kann durch Aufruf der Methode `register(IBSTypeHandler handler)` erfolgen. `RouterService` kümmert sich um die Behandlung von eingehenden Nachrichten unter dem statischen Typ `IBubbleCastMessageType` und kann anhand der Methode `getType` eine Zuordnung zu dem entsprechenden Handler über den Bezeichner des Nachrichtentyps herstellen. Die Nachricht kann demzufolge an den Handler zur weiteren Verarbeitung weitergereicht werden. Es ist wichtig, dass die korrespondierenden Realisierungen der oben genannten Schnittstellen für die selbe Nachricht den gleichen, applikationsweit eindeutigen, Bezeichner zurückliefern, da sonst keine Zuordnung gefunden werden kann. Die Methoden `getBalanceFactor` und `getCertaintyFactor` sind nachrichtenspezifisch zu betrachten. Es liegt in der Verantwortung des Anwendungsentwicklers, Parameter für diese Methoden anhand der Beschreibung des Applikationsprotokolls zu definieren. Sofern eine Zuordnung gefunden werden konnte, wird die Methode `processMessage` des zugewiesenen Handlers aufgerufen. Diese Methode konsumiert eine Reihe von Parametern, die dem Anwendungsentwickler Informationen über die Herkunft der Nachricht (`InetAddress originAddress` und `int originPort`) geben und nicht zuletzt den Inhalt (`byte[] payload`) in Form eines Byte-Array beinhalten. Die Dekodierung des Byte-Array erfolgt dabei anwendungsspezifisch und ist somit Aufgabe des Anwendungsentwicklers. Nachfolgendes Listing zeigt die beispielhafte Umsetzung eines Handlers für die beschriebene Nachrichtenklasse in Listing 4.1.

```
1   public class GenericQueryHandler implements IBSTypeHandler {
2       private Application application;
3
4       public GenericQueryHandler(Application application) {
5           this.application = application
6       }
7
8       public double getBalanceFactor() {
9           return 1.0;
10      }
11
12      public int getBubbleType() {
13          return 3;
14      }
15
```

```
16      public double getCertaintyFactor() {
17          return 3.0;
18      }
19
20      public void processMessage(InetAddress originAddress, int originPort, byte[] payload,
            Replica replica, boolean leafNode) {
21          application.processQuery(originAddress, originPort, payload);
22      }
23
24      public boolean replicationEnabled() {
25          return true;
26      }
27  }
```

Listing 4.2: Handler für GenericQueryMessage-Nachrichten

Die Herkunftsinformationen einer BubbleCast-Nachricht sind relevant, sofern auf eine empfangene BubbleCast-Nachricht eine Antwort verschickt werden soll. Das Adressierungsverfahren für die Antwort ist abhängig von der Gestaltung des konkreten Applikationsprotokolls. In der Regel wird ein BubbleCast jedoch mit einem Unicast beantwortet.

Die nachfolgenden Listings zeigen, wie ein BubbleCast konkret im Quelltext initiiert und der zugehörige Handler zu dem BubbleCast-Nachrichtentyp registriert werden kann.

```
1  IRouterService router = ...;
2  router.bubblecast(new GenericQueryMessage(data));
```

Listing 4.3: Senden einer BubbleCast-Nachricht

```
1  IRouterService router = ...;
2  try {
3    router.register(new GenericQueryHandler());
4  } catch (IOException e) {
5    /* exception handling */
6  }
```

Listing 4.4: Registrieren eines Handlers zu einem BubbleCast-Nachrichtentyp

4.3.2.2 Benutzerverbindungen

Benutzerverbindungen werden im Kontext von BubbleStorm dazu genutzt, um direkte Verbindungen zwischen zwei Hosts innerhalb eines BubbleStorm-Netzwerkes herzustellen. Der BubbleStorm Prototyp stellt hierfür die Schnittstellen `AppSocket` und `AppSocketReceiver` zur Verfügung. Darüberhinaus existiert eine Klasse `AppSocketImpl`, die `AppSocket` implementiert und von der Anwendung genutzt werden kann, um konkrete Sockets zu repräsentieren. Abbildung 4.4 zeigt ein Klassendiagramm, welches die angesprochenen Schnittstellen inklusive ihrer Methodendefinitionen repräsentiert.

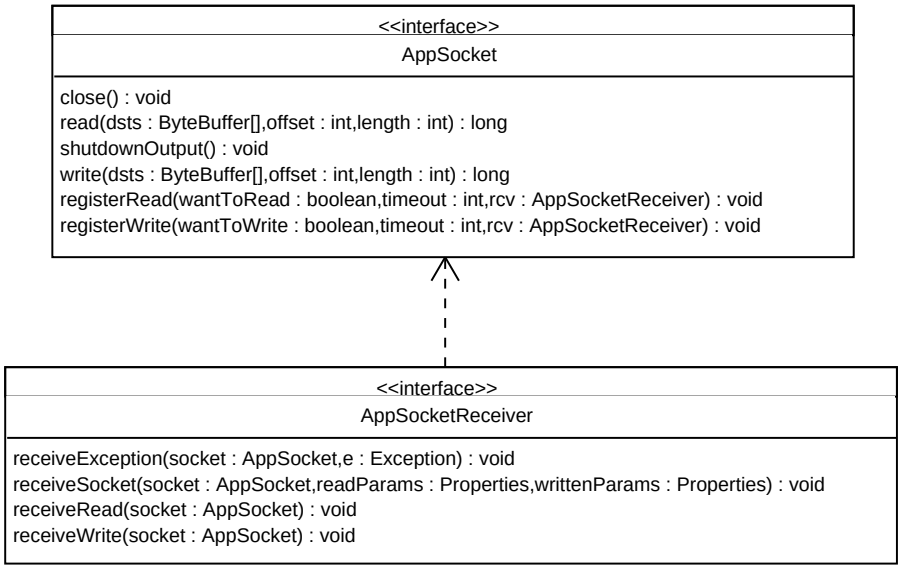


Abbildung 4.4: Schnittstellen zur Verwendung von Benutzerverbindungen

Erzeugen von Benutzerverbindungen

Die Schnittstelle `IRouterService` stellt die Methode `connect(InetAddress addr, int port, String connectionType, AppSocketReceiver rcv)` zur Verfügung, über die eine Applikation die Möglichkeit hat, eine Benutzerverbindung aufzubauen. Die Adresse des zu adressierenden Hosts wird durch die Angabe von `InetAddress addr` und `int port` eindeutig spezifiziert. Damit eine Benutzerverbindung von einer BubbleCast-Nachricht innerhalb der Router-Funktionalität unterschieden werden kann, ist es nötig, einen String-Wert mitzusenden, der die Verbindung als Benutzerverbindung identifiziert, aber prinzipiell beliebig gewählt werden kann. Sobald die Benutzerverbindung hergestellt wurde, erfolgt ein Callback an die Methode `AppSocketReceiver.receiveSocket(AppSocket socket, Properties readParams, Properties writtenParams)`.

```
1 IRouterService router = ...;
2 router.connect(ip, port, "CustomConnectionType", new CustomAppSocketReceiver());
```

Listing 4.5: Verbindung zu einem Host herstellen

Benutzerverbindungen akzeptieren

Bevor eine Benutzerverbindung hergestellt werden kann, muss ein Host, zu dem eine Verbindung aufgebaut werden soll, eine konkrete Ausprägung der Schnittstelle `AppSocketReceiver` an seinem `RouterService` registrieren. Durch diesen Vorgang kennt der `RouterService` den Handler für Benutzerverbindungen eines speziellen Typs und kann, sofern eine solche Verbindung hergestellt wurde, den zugehörigen Socket dorthin weiterreichen.

```
IRouterService router = ...;
router.register("CustomConnectionType", new CustomAppSocketReceiver());
```

Listing 4.6: Akzeptieren eingehender Benutzerverbindung bestimmten Typs

Schreib- und Leseoperationen auf einer Benutzerverbindung

Damit eine Applikation auf einem bereits hergestellten Socket Schreib- oder Leseoperationen durchführen kann, muss sie sich an diesem Socket für einen solchen Vorgang registrieren. Dies geschieht, indem die Methode `AppSocket.registerRead` respektive `AppSocket.registerWrite` mit den entsprechenden Argumenten aufgerufen wird. Ein Callback über die Methode `AppSocketReceiver.receiveRead` bzw. `AppSocketReceiver.receiveWrite` informiert die Applikation darüber, dass die Operation nun durchgeführt werden kann. Innerhalb dieser Methoden ist der zugehörige `AppSocket` bekannt, so dass nun mittels `AppSocket.read` oder `AppSocket.write` die entsprechende Operation ausgeführt werden kann. Sofern nicht alle Daten gelesen oder geschrieben worden sind, muss sich die Applikation erneut an dem Socket registrieren, um die restlichen Daten verarbeiten zu können.

Die Listings 4.7 und 4.8 zeigen, wie eine beispielhafte Umsetzung von `AppSocketReceiver.receiveWrite` und `AppSocketReceiver.receiveRead` aussehen könnte.

```
public void receiveWrite(AppSocket socket) {
  try {
    long written;
    do {
      written = socket.write(new ByteBuffer[] { buffer }, 0, 1);
    } while (written != 0);
  } catch (IOException e) {
    e.printStackTrace();
  }

  /* there is still more data to be written, so we have to
   * re-register for writing; otherwise, the buffer has been
   * sent completely */
  if (buffer.remaining() != 0) {
    socket.registerWrite(true, 0, this);
  } else {
    socket.registerWrite(false, 0, this);
    socket.shutdownOutput();
  }
}
```

Listing 4.7: Beispielhafte Implementierung von `receiveWrite`

```
public void receiveRead(AppSocket socket) {
  int hashCode = socket.hashCode();

  long read;
  ByteBuffer buffer = alreadyReadData.get(hashCode);

  try {
```

```
8       do {
9         read = socket.read(new ByteBuffer[] { buffer }, 0, 1);
10
11        if (buffer.remaining() == 0) {
12          ByteBuffer b = ByteBuffer.allocate(buffer.capacity() * 2);
13          b.put(buffer);
14          buffer = b;
15        }
16      } while (read != 0);
17    } catch (IOException e) {
18      e.printStackTrace();
19    }
20
21    alreadyReadData.put(hashCode, buffer);
22
23    /* transform the read data into a string and check if the
24     * message has been fully received */
25    StringBuffer sb = new StringBuffer();
26
27    buffer.flip();
28    while(buffer.hasRemaining()) {
29      sb.append((char)buffer.get());
30    }
31
32    if (isMessageComplete(sb.toString())) {
33      alreadyReadData.remove(hashCode);
34      socket.registerRead(false, 0, this);
35      processMessage(socket, sb.toString());
36      socket.shutdownOutput();
37    } else {
38      socket.registerRead(true, 0, this);
39    }
40  }
```

Listing 4.8: Beispielhafte Implementierung von `receiveRead`

Kapitel 5

Anforderungen

> Kein Ding entsteht oder ereignet sich aufs Geratewohl, sondern alles in begründeter Weise und durch Notwendigkeit.
>
> *Leukipp*

Bevor eine Erhebung der Anforderungen konkretisiert werden kann, soll zunächst Einblick darüber gegeben werden, was ein Wiki typischerweise charakterisiert. Unter einem Wiki versteht man eine Ansammlung diverser Artikel, die üblicherweise im HTML-Format präsentiert werden und von jedem Benutzer eingesehen werden können. Die Nutzung von HTML ermöglicht es, Artikel mit ähnlichem Bezug durch Querverweise in Form von Hyperlinks miteinander zu verknüpfen. Ein Wiki kann als eine Ausprägung von Content-Management-Systemen verstanden werden, mit dem Unterschied, dass es explizit der sozialen Kommunikation und Kollaboration dient. Die Benutzer eines Wikis repräsentieren eine aktive Gemeinschaft, da jeder Beteiligte Änderungen in Echtzeit an bereits gespeicherten Artikeln vornehmen oder neue, noch nicht existente, Artikel erstellen kann.

Dieses Kapitel widmet sich der Anforderungserhebung für das zu implementierende Peer-to-Peer Wiki mit Fokus auf grundsätzlichen Aspekten wie Funktionsumfang und Benutzbarkeit der Anwendung.

5.1 Anforderungserhebung

Das Peer-to-Peer Wiki wird in der Programmiersprache Java (Version 6.0) implementiert und nutzt zur Implementierung der grafischen Benutzeroberfläche die Java Foundation Classes (Swing).

Basisfunktionalitäten

Die Basisfunktionalitäten eines Wikis liegen in der Beschaffung, Darstellung und Erzeugung von Artikeln. Die Software soll die Erstellung neuer Artikel über einen komfortablen Editor unterstützen. Das Format eines Artikels soll sich an der Syntax der Wiki Markup Language (WikiML) orientieren, die unter Anderem auch von der Software MediaWiki[7] benutzt wird,

[7] MediaWiki ist die Wiki-Software, die von Wikipedia verwendet wird.

um Artikel zu speichern. Der integrierte Editor soll den Zugang zu dieser Syntax durch eine Werkzeugleiste, die Shortcuts zu diversen Befehlsstrukturen der WikiML bietet, unterstützen. Artikel sollen mitsamt den zugehörigen, beschreibenden Daten unter Verwendung der Extensible Markup Language (XML) persistent gehalten werden können.

Bereits eingestellte Artikel müssen über die Software angezeigt werden können. Die Darstellung von HTML-behafteten Inhalten wird grundsätzlich von Swing-Komponenten unterstützt. Das Präsentationsformat der Artikel kann daher problemlos über HTML erfolgen. Da ein Artikel mit Hilfe der WikiML-Syntax verfasst wird, ist vor der Darstellung eine Transformation von WikiML-Strukturen nach HTML erforderlich.

Ein Artikel, der bereits eingestellt worden ist, muss für jeden Benutzer veränderbar sein. Änderungen sollen ausschließlich über den integrierten Editor der Software durchgeführt werden.

Die Applikation soll die Suche nach einem bestimmten Artikel unterstützen. Die grafische Schnittstelle für diese Funktionalität soll sich dabei an dem Vorbild aktueller Browser orientieren und dem Benutzer Eingabemöglichkeiten bereitstellen, um Artikel anhand ihres Titels adressieren zu können. Diese Adressierung entspricht semantisch einem exakten Matching des eingegebenen Textes auf die Artikeltitel der gespeicherten Artikel. Sofern ein korrespondierender Artikel nicht verfügbar ist, soll der Benutzer die Möglichkeit bekommen, einen neuen Artikel unter diesem Titel über den integrierten Editor anzulegen.

Versionierung

Die Software soll die Möglichkeiten einer rudimentären Versionierung nutzen, um eine lückenlose Änderungshistorie für jeden Artikel bereitstellen zu können. Die Integration dieser Anforderung mündet in die Verfügbarkeit wesentlicher Operationen, die nachfolgend beschrieben werden.

Die Versionierung setzt voraus, dass die Anwendung dem Benutzer die Möglichkeit bietet, mehrere Versionen eines Artikels abzuspeichern. Sobald ein Benutzer einen Artikel editiert hat und seine Änderungen speichern möchte, soll der bisher aktuelle Inhalt als Änderungsinformation (Diff) archiviert und die neue Version publiziert werden.

Die Versionsgeschichte eines Artikels soll für jeden Benutzer auf Anfrage einsehbar sein. Das Programm soll hierzu eine tabellarische Übersicht liefern, die eine nach Änderungszeitpunkt sortierte Liste aller verfügbaren Versionen zu dem entsprechenden Artikel anzeigt. Über diese Ansicht soll es des Weiteren möglich sein, vorhergehende Versionen auszuwählen und rekonstruieren zu lassen. Die Auswahl von zwei differierenden Versionen soll dem Benutzer die Möglichkeit eröffnen, Änderungen zwischen diesen beiden Versionen visuell darzustellen, wobei geänderte, gelöschte oder hinzugefügte Textabschnitte hierbei farblich hervorgehoben werden sollen.

Sofern zwei Benutzer zur gleichen Zeit Änderungen an einem Artikel vornehmen, besteht die Gefahr, dass konfligierende Versionszustände in das System eingespielt werden. Die Software muss diese Zustände erkennen und soll versuchen, den Konflikt automatisch aufzulösen. Schlägt dies fehl, so muss der Benutzer, der zuletzt seine Änderungen eingespielt hat, die Gelegenheit erhalten, den Konflikt manuell aufzulösen. In jedem Fall muss eine Benachrichtigung des Benutzers durch die Software erfolgen.

Zusätzliche Features

Die Integration einer Volltextsuche soll die Benutzbarkeit der Applikation aufwerten. Die Volltextsuche soll einen Suchtext konsumieren und eine Liste von möglichen Treffern, geordnet nach einem vorgeschlagenen Maß für die Relevanz der spezifischen Resultate, zurückliefern. Der Benutzer hat nachfolgend die Wahl, konkrete Artikel aus der Menge der Resultate anzeigen zu lassen.

Die Software soll es dem Benutzer ermöglichen, Artikel durch die Integration von medialen Inhalten auszugestalten. Im Kontext dieser Arbeit soll sich die Unterstützung auf die pixelbasierten Grafikformate JPEG und PNG beschränken. Der Benutzer kann über einen Auswahldialog Bilddateien dieser Formate selektieren und in das Netzwerk einpflegen, so dass die Grafik später im Artikel verwendet werden kann. Die Speicherung der Bilddateien soll Base64-kodiert erfolgen.

Kapitel 6

Versionsmanagement

> Das Problem zu kennen ist wichtiger, als die Lösung zu finden, denn die genaue Darstellung des Problems führt automatisch zur richtigen Lösung.
>
> *Albert Einstein*

Das Versionsmanagement beschreibt eine Vorgehensweise zur Archivierung diverser Dokumentstände nebst zugehöriger Änderungsprotokollierung. Versionsmanagementsysteme setzen dieses Paradigma praktisch um und werden typischerweise dort eingesetzt, wo eine lückenlose Dokumentation eines Entwicklungsprozesses nötig ist. Dies ist beispielsweise der Fall bei umfangreichen Projekten in der Softwareentwicklung, an denen mehrere Personen beteiligt sind. Jeder Beteiligte hat Zugriff auf das sogenannte *Repository*, welches den gesamten Datenbestand repräsentiert und in den meisten Fällen zentral von einem Server verwaltet wird. Mehrere Benutzer können parallel auf diesem Datenbestand arbeiten und Änderungen durchführen, die nach erfolgreicher Einpflege protokolliert und mit zusätzlichen Kommentaren des Ausführenden belegt werden können. Dies ermöglicht die nötige Dokumentation, um Änderungen verfolgen zu können und erlaubt es ferner durch die Speicherung alter Versionen vorhergehende Dateistände wiederherzustellen. Komplexe Mechanismen stellen dabei sicher, dass der Datenbestand von einem konsistenten Zustand wieder in einen konsistenten Zustand überführt wird.

In diesem Kapitel soll beschrieben werden, was die typischen Aufgaben eines solchen Systems sind und welche Unterscheidungsmerkmale sich feststellen lassen. Eine kurze Beschreibung bestehender Implementierungen gibt Auskunft darüber, wie diese Systeme die genannten Anforderungen erfüllen. Ferner befasst sich dieses Kapitel mit den grundsätzlichen Parallelisierungsstrategien und gibt nach diesem anfänglichen Exkurs Einblick darüber, wie eine Versionsverwaltung in einem Wiki mit verteiltem Datenbestand realisiert werden kann.

Es ist dabei zu beachten, dass ein vollständiges und ausgereiftes Versionierungssystem sehr komplex ist und aufgrund des Umfangs nicht in dieser Weise Einzug in die Arbeit erhalten kann. Die Ausgestaltung der Versionierungsanteile beschränkt sich daher auf Kernkonzepte, die im Laufe des Kapitels konkretisiert werden sollen.

6.1 Grundsätzliche Unterschiede

Die auffälligste Unterscheidung zwischen Versionsmanagementsystemen besteht in der Art der Datenhaltung. Hier gibt es grundsätzlich zwei verschiedene Möglichkeiten: Die Speicherung des Datenbestandes auf einem zentralen Server, der für jeden Benutzer mit entsprechender Berechtigung zugänglich ist, oder aber die partielle Speicherung des Datenbestandes auf den Endgeräten der Benutzer. Partiell bedeutet in diesem Kontext, dass jedes Endgerät eine Teilmenge des gesamten Datenbestandes verwaltet. Diese Teilmengen können disjunkt sein, müssen es jedoch nicht. Beide Ansätze nehmen gravierend darauf Einfluss, wie ein Versionsmanagementsystem funktioniert.

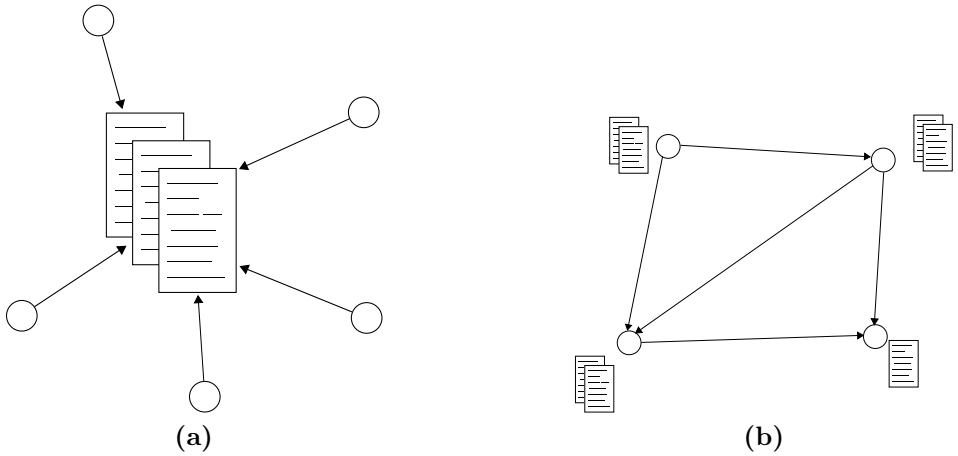


Abbildung 6.1: Skizze für (a) zentrale und (b) verteilte Datenhaltung

Die Vor- und Nachteile beider Ansätze stehen konträr zueinander, wie folgende Auflistung zeigt:

Zentral	**Verteilt**
– Single point of failure	+ *Kein* single point of failue
– Teuer, da zusäztliche Serverhardware nötig ist	+ Kostengünstig, da vorhandene Hardware genutzt wird
• Erfordert explizites Branching	• Bringt Branching implizit mit
+ Zentrale Haltung ermöglicht bessere Kontrolle über den Datenbestand	– Dezentrale Haltung bietet nur geringfügige Kontrolle über den Datenbestand

6.2 Aufgaben eines Versionsmanagementsystems

Die Konzeption einer Versionsverwaltung setzt das Wissen über die typischen Aufgaben eines derartigen Systems voraus. Zusammengefasst lassen sich fünf wesentliche Aspekte herausstellen, die sich eng an [3] anlehnen.

6.2.1 Protokollierung der Änderungen

Über das Mitführen von Änderungsinformationen zwischen zwei Dateiversionen wurde bereits einführend gesprochen. Die Motivation liegt darin, dass der Benutzer zu einem gegebenen Zeitpunkt eine komplette Änderungshistorie aus den Metadaten[8] des *Repository* gewinnen kann.

Dies führt zu einer Reihe von Diskussionspunkten bezüglich der internen Struktur von Versionsinformationen. Für eine einfache Änderungshistorie ist für jede Version einer Datei zunächst eine Versionsnummer zu vergeben. Zu einer Versionsnummer gehören ferner das Datum der Einpflege sowie der Name des Benutzers, der diese Aktion durchgeführt hat. Optionale Kommentare, die einer Änderung angefügt werden können, werden von den meisten Implementierungen unterstüzt und erhöhen die Transparenz der Änderungen für andere Benutzer.

6.2.2 Wiederherstellung alter Versionen

Das alleinige Abfragen der Metadaten zu einer Version ist oft nicht das, was der Benutzer möchte. In der praktischen Arbeit mit Versionsmanagementsystemen lassen sich viele Situationen finden, in denen explizit ältere Versionsstände rekonstruiert werden müssen, um eine vollständige Sicht auf die Evolution des Dokumentes gewinnen zu können. Dies fördert ebenso die Transparenz, da für jeden Benutzer ersichtlich ist, was von Version zu Version geändert worden ist und wer diese Änderungen vollzogen hat. Das Versionsmanagementsystem muss hierbei gewährleisten, dass eine bereits eingepflegte Version jederzeit aus dem Datenbestand gewonnen werden kann. Es muss ferner gewährleisten, dass diese Aktion den Datenbestand von einem konsistenten Zustand wieder in einen konsistenten Zustand überführt.

6.2.3 Archivierung einzelner Release-Stände

Einige Implementierungen erlauben es, den aktuellen Stand des kompletten Datenbestands, oder einem Teil davon, in einen separaten Datenbestand auszugliedern. Dieser Stand wird dabei festgehalten und existiert unabhängig von dem ursprünglichen Datenbestand, auf den weitere Änderungen erfolgen können.

[8] An dieser Stelle sei vermerkt, dass der Begriff *Metadaten* in der Fachwelt verschiedene Ausprägungen kennt. Der intuitive Gebrauch dieses Begriffs meint beschreibende Daten, die Informationen über andere Daten (meist einem Zusammenschluss von Daten oder Objekten) enthalten. In diesem Sinne wäre beispielsweise die Angabe eines Autors zu einem Artikel ein Metadatum. Dagegen beschreiben Metadaten im Bezug auf Datenbankmanagementsysteme jene Daten, die Aussagen über die Beschaffenheit der strukturellen Merkmale einer Datenbank liefern. Im Kontext dieser Arbeit wird dieser Begriff aus Sicht der intuitiven Gebrauchsweise benutzt.

Dieses Vorgehen findet besonders bei der Softwareentwicklung Verwendung, da der Datenbestand für ein bestimmtes Release unabhängig von dem Entwicklungszweig ausgelagert und festgehalten werden kann.

6.2.4 Koordinierung eines gemeinsamen Zugriffs

Aus dem Zugriff von mehreren Benutzern auf den gleichen Datenbestand dürfen keine Inkonsistenzen resultieren. Dies ist beispielsweise dann der Fall, wenn zwei Benutzer an der gleichen Datei arbeiten und unabhängig voneinander Änderungen in das System einpflegen, ohne vorher etwaige Aktualisierungen dieser Datei bezogen zu haben. Das Versionsmanagementsystem muss sicherstellen, dass derartige Konflikte nicht auftreten oder behoben werden können.

6.2.5 Vorantreiben mehrerer Zweige

Grundsätzlich kann die gesamte Historie zu einem Dokument als Baumstruktur angesehen werden. Einige Systeme ermöglichen es, dass zu einem bestimmten Versionsstand mehrere nachfolgende Versionsstände existieren können. Dieses Konzept wird durch sogenannte *branches*, sprich Abzweigungen von dem Hauptentwicklungsstrang, realisiert. *Branches* existieren dabei unabhängig voneinander und können ebenso unabhängig weiterentwickelt werden.

6.3 Bestehende Versionsmanagementsysteme

Im Zuge der Recherche wurden zwei bestehende Implementierungen im Detail betrachtet, die unterschiedliche Topologien nutzen. Der nachfolgende Abschnitt soll einen Überblick über die Eigenheiten dieser Systeme geben und kurz deren Arbeitsweise erläutern.

6.3.1 Concurrent Versions System

Das Concurrent Versions System (CVS) basiert auf den Versionsmanagementsystemen RCS und SCCS und ermöglicht erstmals den Zugriff auf ein zentral verwaltetes Repository über das Netzwerk. Die Arbeitsweise unterscheidet sich daher grundlegend von den Vorgängerversionen. Typischerweise holt sich der Benutzer lokale Kopien der aktuellen Versionen vom zentralen CVS-Server. Dabei werden Metadaten angelegt, die den letzten lokalen Versionsstand beschreiben. Anzumerken ist, dass eine Kopie als vollständig neue Datei von CVS angesehen wird. Anders als bei RCS ist es daher nicht nötig, für die Bearbeitung einer Datei diese explizit zu sperren. Der Benutzer kann ohne Weiteres lokale Änderungen durchführen und diese zu gegebenem Zeitpunkt in das Repository einpflegen. Problematisch ist es, wenn mehrere Benutzer zur gleichen Zeit die selbe Datei verändern und ihre lokalen Versionsstand einpflegen wollen. Dies kann Versionskonflikte induzieren, für die CVS eine sogenannte Merge-Funktionalität bereitstellt, um diese Konflikte zu lösen und erneut eine einheitliche und konsistente Sicht auf den Versionsstand des Projektes herzustellen. CVS verwendet Delta-Kodierung, um die Änderungshistorie einer Datei platzsparend zu speichern. Prinzipiell sieht es CVS vor, nur textuelle Dateien zu speichern. Für

die Versionierung von binären Inhalten ist dieses System nicht ausgelegt. Das Nachfolgersystem Subversion (SVN) verspricht Lösungen für dieses und einer Reihe weiterer Probleme.

6.3.2 Git

Im Gegensatz zu CVS basiert Git auf einer vollständig verteilten Architektur. Es existiert kein zentraler Server, der das Repository verwaltet, sondern jeder Benutzer bekommt eine lokale Kopie der kompletten Änderungshistorie und pflegt diese Kopie eigenverantwortlich. Die meisten Benutzeraktionen können dadurch lokal ausgeführt werden und kommen ohne Zugriff auf das Netzwerk aus. Die Besonderheit bei Git ist, dass nicht zwischen lokalen und entfernten Entwicklungszweigen unterschieden wird. Die Grundvoraussetzungen hierfür werden durch eine Reihe von Werkzeugen gebildet, die eine effiziente Handhabung von nicht-linearen Entwicklungssträngen ermöglichen. Ferner sorgt das Git Protokoll für einen flexiblen und effizienten Datenaustausch zwischen den Repositories, was nicht zuletzt den Vorteil mitbringt, dass Benutzer selbst entscheiden können, von welchem Autor sie den aktuellen (lokalen) Versionsstand beziehen möchten. Git unterstützt zudem eine authentische Sicht auf die komplette Änderungshistorie, bedingt durch die Speicherungsstrategie des Datenbestandes. Der Name einer neu eingepflegten Version basiert auf der vollständigen Historie bis zu dieser Version. Zusätzlich ist es möglich, Versionen mit GPG digital zu signieren.

6.4 Parallelisierungsstrategien

Es wurde bereits beschrieben, dass die parallele Bearbeitung eines Dokumentes zu einem inkonsistenten Zustand führen kann. Um dieses Problem zu beheben, existieren zwei differierende Ansätze [3], die nachfolgend erläutert werden sollen.

6.4.1 Lock Modify Write

Lock Modify Write stellt die traditionelle Vorgehensweise dar und ist in der Literatur ebenfalls unter der Bezeichnung *Lock Modify Unlock* bekannt. Es basiert auf der pessimistischen Grundannahme, dass die gleichzeitige Bearbeitung einer Datei grundsätzlich zu Problemen führt und daher wenig erstrebenswert für die praxistaugliche Arbeit ist. Aus diesem Grund müssen vor einer Änderung an einer Datei die Schreibzugriffsrechte durch einen sogenannten *Lock* explizit erworben werden. Dies sperrt die Datei vor den möglichen Zugriffen anderer Benutzer solange, bis der ausführende Benutzer seine Änderungen eingespielt und die Datei explizit wieder entsperrt hat. Diese Strategie findet Anwendung in den Versionsmanagementsystemen RCS und Visual Source Safe [10] von Microsoft.

6.4.2 Copy Modify Merge

Die Strategie *Copy Modify Merge* steht konträr zu *Lock Modify Write* da sie eine grundsätzlich optimistische Annahme verfolgt und das gleichzeitige Ändern von einer Datei aus dem Datenbestand erlaubt. Nachdem Änderungen vollzogen wurden, muss die Einpflege (engl. *commit*)

explizit durchgeführt werden. Das Versionsmanagementsystem erkennt hierbei, ob es konfligierende Zustände gibt und versucht diese gegebenenfalls aufzulösen. Schlägt dieser Versuch fehl, so wird der Benutzer, der den Konflikt verursacht hat, benachrichtigt und muss die Auflösung manuell herbeiführen. Diese Vorgehensweise erleichtert vor allem die Arbeit für räumlich getrennte Benutzer, da ein deutlich flüssigeres Arbeiten ermöglicht wird. Diese Strategie wird unter Anderem von den Versionsmanagementsystemen CVS und Subversion umgesetzt.

6.5 Realisierung

Nach diesem kurzen Exkurs über die Grundlagen von Versionsmanagementsystemen soll im Folgenden eine Anforderungsanalyse und Spezifikation für ein derartiges System innerhalb einer verteilten Anwendung erstellt werden. Die Anforderungsanalyse betrachtet hier die bereits erwähnten typischen Aufgaben eines Versionsmanagementsystems und konkretisiert diese in Bezug auf die zu entwickelnde Anwendung. Abschließende Betrachtungen gehen auf interne Details wie Objektrepräsentation, Speicherung und Konfliktbewältigung ein.

6.5.1 Anforderungen

Protokollierung von Änderungen

Die Protokollierung von Änderungen erscheint durchaus sinnvoll in dieser Umgebung und kann exakt in der beschriebenen Art und Weise implementiert werden. Im Wesentlichen ist hierfür lediglich eine adäquate Speicherung der Metadaten zu einer Version entscheidend.

Diese Metadaten erlauben uns den Zugriff auf eine eindeutige und lückenlose Änderungshistorie. An dieser Stelle tritt ein grundsätzliches Problem mit verteilten Datenbeständen auf. Sofern der anfordernde Benutzer diese Informationen nicht vollständig lokal abrufen kann, müssen die entsprechenden Daten über das Netzwerk angefordert werden. Selbst wenn man davon ausgeht, dass der komplette Datenbestand konstant im Netzwerk gehalten wird, gibt es keine Garantie dafür, dass zu einem gegebenen Zeitpunkt diese Information zugänglich ist (beispielsweise aufgrund von Überlastung des Netzverkehrs).

Wiederherstellung alter Versionen

Die Anforderung, ältere Versionen nach Bedarf rekonstruieren zu können erscheint ebenso sinnvoll. Für ein verteiltes Wiki ist es jedoch nicht relevant, dass alte Versionsstände erneut geändert und abgespeichert werden können, denn das hätte einerseits nur einen geringen praktischen Nutzen, andererseits würde es Aktualisierungskaskaden über das Netzwerk induzieren, was alleine schon aus Gründen der Performanz nicht tragbar ist.

Archivierung einzelner Release-Stände

Wie bereits dargestellt, ist die Archivierung von Release-Ständen von primärer Relevanz in der Software-Entwicklung. Für ein Peer-to-Peer-basiertes Wiki hat diese Anforderung keinen besonderen Nutzwert, da sich Einträge sukzessive entwickeln und zunächst immer der aktuellste Versionsstand eines Eintrags präsentiert werden soll.

Koordinierung eines gemeinsamen Zugriffs

Durch die entsprechende Parallelisierungsstrategie - siehe Abschnitt 6.5.2 - ist dies ohne weitere Probleme möglich.

Vorantreiben mehrerer Zweige

Das gleichzeitige Vorantreiben mehrerer Entwicklungsstränge bringt der verteilte Charakter einer Peer-to-Peer basierten Applikation bereits implizit mit. Das Problem hierbei ist, dass ohne ein explizites Anlegen von separaten Zweigen viele Versionen mit identischer Versionsnummer koexistieren können, was im allgemeinen Fall nicht der Situation entspricht, die in einer solchen Umgebung wünschenswert ist. Um diesem Problem entgegenzuwirken, soll das Versionsmanagementsystem Verfahren bereitstellen, um Konflikte automatisch zu erkennen und lösen zu können. Die Möglichkeit besteht, dass durch die automatische Bereinigung der Versionskonflikt zwar gelöst werden konnte, aber die resultierende Version nun nicht mehr widerspruchsfrei ist. Eine widerspruchsfreie Datenhaltung erfordert daher die Beteiligung des Nutzers, der über derartige Vorgänge informiert werden muss, um automatisch durchgeführte Änderungen an den Einträgen einsehen und gegebenenfalls korrigieren zu können. Diese Vorgehensweise erlaubt eine einheitliche Behandlung von Versionskonflikten, auch wenn sie in ihren Grundsätzen konträr zu dem Verzweigungsgedanken steht.

6.5.2 Parallelisierungsstrategie

Die Grundannahme für die Versionierung in einem Peer-to-Peer basierten Wiki ist, dass der Datenbestand auf die Endgeräte einzelner Benutzer (verschiedener *Peers*) verteilt wird. Sofern Teile des Datenbestandes auf mehreren Peers repliziert werden - also ein und dieselbe Version auf n Peers vorhanden ist -, würde die Strategie *Lock Modify Write* erfordern, dass Benutzer die Dateien der zugehörigen Dokumente auf diesen n entfernten Peers sperren können. Eine konkrete Umsetzung dieses Vorgehens belastet das Netzwerk zusätzlich mit *Lock*- und *Unlock*-Nachrichten und ist aus Gründen der Performanz inakzeptabel. Das Prinzip des *Copy Modify Merge* erscheint in einer verteilten Umgebung durchaus sinnvoller und nicht zuletzt benutzbarer, da Änderungen direkt durchgeführt und nachfolgend im Netzwerk verteilt werden können. Im Zuge der Konsistenzwahrung erfordert es jedoch einen Algorithmus, der konfligierende Zustände zwischen Versionen lösen kann. Der zusätzliche Aufwand zur Realisierung eines solchen Algorithmus wird wohlwissend zu Gunsten der Benutzbarkeit des resultierenden Systems in Kauf genommen.

6.5.3 Format der Datenbasis

Dieser Abschnitt diskutiert die Konzeption der Datenbasis und geht darauf ein, welche Informationen nötig sind, um einen Eintrag eindeutig zu identifizieren und wie mehrere Versionen zu einem Eintrag platzsparend archiviert werden können.

6.5.3.1 Metadaten

Für das Versionsmanagementsystem sind Dokumentversionen, gleich ob textuelle oder mediale Inhalte vorliegen, generische Objekte die eindeutig identifiziert werden können. Man unterscheidet dabei nicht nur Eintrag von Eintrag, sondern auch Versionen eines Eintrags untereinander. Die eindeutige Identifikation eines solchen Objekts kann dabei durch das 4-Tupel

$$Version := (Titel, Basisversion, Versionsnummer, Hash)$$

erfolgen. Die Speicherung eines Hash-Wertes über den Inhalt einer Version ermöglicht es, unterschiedliche Einträge trotz identischer Basisversion identifizieren zu können. Um detailliertere Informationen zu einem Versionsstand liefern zu können, sollen die Attribute *Autor* und *Datum* zusätzlich gespeichert werden.

Die Semantik der verwendeten Attribute ist dabei:

Titel	Bezeichnet den Titel des Dokuments.
Basisversion	Repräsentiert die Versionsnummer des Eintrags, auf welchem die aktuelle Version basiert.
Versionsnummer	Repräsentiert die Versionsnummer dieser Eintragung.
Hash	MD5-Hashwert, der über den Inhalt dieser Version gebildet wird. Der Hashwert wird unter Anderem dazu genutzt, Versionen mit identischer Basisversion aber unterschiedlichem Inhalt zu identifizieren.
Autor	Bezeichnet den Namen des Benutzers, der diese Version eingespielt hat.
Datum	Bezeichnet das Datum der Einpflege.

6.5.3.2 Delta-Kodierung

Grundsätzlich können mehrere Versionen zu einem Eintrag in der verteilten Datenbasis existieren. Eine naive Speicherungsstrategie würde zu jeder dieser Versionen den vollen Umfang des Inhalts speichern. Diese Strategie ist jedoch nicht sonderlich effizient, da sie bei wachsendem Datenbestand enormen Speicherplatz benötigt.

Die Änderungen von Version zu Version sind bei textuellen Daten jedoch oft marginal. Mitunter werden oftmals lediglich orthographische Schwächen beseitigt oder anderweitige, geringfügige Korrekturen vorgenommen. Dies führt zu der Idee, dass Speicherungsverfahren in der Form zu optimieren, dass lediglich die Änderungen zwischen der alten und neuen Version gespeichert werden.

Dieses Verfahren, auch bekannt als *Delta-Kodierung* oder *Differenzspeicherung*, gewährleistet eine platzsparende Speicherung für Dokumente, die in mehreren Versionen vorliegen. Grundsätzlich existieren zwei Möglichkeiten, dieses Verfahren zu implementieren.

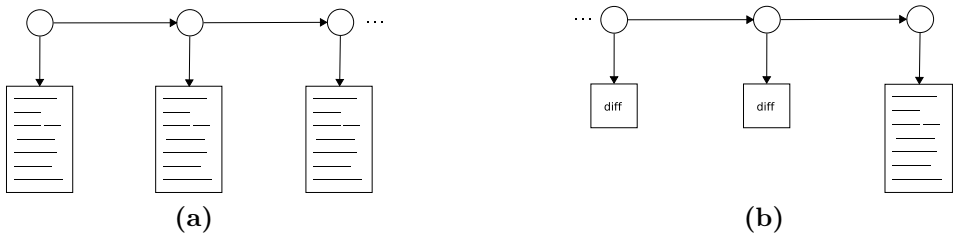


Abbildung 6.2: Skizze für Speicherung in (a) Volldarstellung und mittels (b) Delta-Kodierung

(i) Die ursprüngliche Version zu einem Dokument wird in Volldarstellung gespeichert, künftige Versionen nur als Änderungsinformationen.

(ii) Die aktuellste Version zu einem Dokument wird in Volldarstellung gespeichert, frühe Versionen nur als Änderungsinformationen.

Dies erfordert einen Mechanismus, der sukzessive Änderungsinformationen auf eine Version in Volldarstellung anwenden kann, um den Inhalt einer gesuchten Version zu rekonstruieren. Ein derartiger Mechanismus kann durch die Integration von *diff*- und *patch*-Funktionalitäten realisiert werden.

Mögliche Integration in ein Wiki

Das Konzept eines Wikis ist es, stets die aktuellste Version eines Dokuments zu präsentieren. Vorangegangene Versionen werden generell auf expliziten Wunsch des Benutzers erzeugt und dargestellt. Aus Performanzgründen ist daher die Implementierungsvariante (ii) vorzuziehen, da die aktuellste Repräsentation in Volldarstellung vorliegt und nicht aufwendig rekonstruiert werden muss.

Wichtig ist dabei, dass Änderungsinformationen nicht verteilt werden, sondern lediglich in den lokalen Datenbasen der Peers vorliegen. Es soll jedoch möglich sein, konkrete Änderungsinformationen über das Netzwerk anfragen zu können, um dem lokalen Datenbestand der Peers eine konsistente und vollständige Sicht auf die Versionsgeschichte der Dokuments zu ermöglichen.

6.5.4 Konsistenzwahrung

Es wurde bereits dargelegt, dass die konsistente Haltung des Datenbestandes eine Aufgabe des Versionsmanagementsystems ist. Dieser Abschnitt erläutert, in welcher Art und Weise Benutzer Änderungen einpflegen können und geht ferner auf Situationen ein, die Konflikte zwischen Versionen eines Dokuments hervorrufen. Abschließend wird ein Algorithmus zur Lösung von konfligierenden Zuständen betrachtet. Der vorgestellte Algorithmus ist gleichzeitig die Basis für die spätere Realisierung dieser Funktionalität in der zu entwickelnden Wiki Applikation.

6.5.4.1 Speichervorgang

In einem auf *Copy Modify Merge*-basierten Versionsmanagementsystem ist es üblich, dass während des Einpflegevorgangs überprüft wird, ob sich bereits eine aktuellere Version des Dokuments im Datenbestand befindet. Dieser Fall tritt ein, wenn zwei Benutzer die gleiche Version eines Dokuments beziehen, unabhängig voneinander Änderungen daran vornehmen und nacheinander ihre neuen Versionen einpflegen möchten. Sofern eine neue Version vorliegt, muss das Versionsmanagementsystem den Benutzer darüber informieren und ihm die Möglichkeit geben, seine Änderungen mit denen des neuen Inhalts abzugleichen, um diese unter einer neuen Version erneut einpflegen zu können. Liefert die Abfrage keine anderen, neuen Versionen mit identischer Basisversion, so können die Änderungen eingepflegt werden. Damit diese Änderungen letztlich unter einer neuen Version erscheinen können, müssen Werte für die Metadaten [9] *Basisversion* und *Versionsnummer* angepasst werden.

$$Basisversion_{neu} = Versionsnummer_{alt}$$

$$Versionsnummer_{neu} = \lfloor Versionsnummer_{alt} + int(rand() \cdot c + 1) \rfloor$$

c ist dabei ein konstanter Anteil, der die gefundene Zufallszahl auf eine Zahl aus dem Interval $[1; c]$ transformiert.

6.5.4.2 Entstehung von Versionskonflikten

Betrachtet man den Speichervorgang als Funktion der Zeit, so lässt sich festhalten, dass Δt Zeiteinheiten (dies entspricht dem *timeout* der Nachricht) benötigt werden, bevor die Ergebnisse der Abfrage ausgewertet werden können. Betrachten wir nun folgendes Szenario, in dem zwei Benutzer 1 und 2 die gleiche Version eines Artikels beziehen. Beide Benutzer nehmen Änderungen an X vor und möchten diese nun zu den Zeitpunkten t_1 (Benutzer 1) und t_2 (Benutzer 2) speichern. Daraus ergeben sich für jeden separaten Speichervorgang zwei Zeitintervalle $A = [t_1; t_1 + \Delta t]$ und $B = [t_2; t_2 + \Delta t]$. Da Änderungen erst dann eingepflegt werden können, wenn die notwendige Abfrage während des Speichervorgangs ein Resultat geliefert hat, sind zeitlich parallele Speichervorgänge über dem gleichen Dokument X innerhalb eines Zeitintervalls nicht sichtbar. Das heißt, wenn sich die Zeitintervalle A und B überschneiden oder gar zusammenfallen, werden parallele Änderungen aufgrund annähernd zeitgleichen Speicherns nicht bemerkt.

Dieser Umstand führt dazu, dass mehrere Versionen mit identischer Basisversion, aber unterschiedlichem Inhalt, im Datenbestand existieren können. Es liegt in der Verantwortung des Versionsmanagementsystems, diese Konflikte zu erkennen und aufzulösen.

6.5.4.3 Behebung von Versionskonflikten

Versionskonflikte können relativ einfach erkannt werden, wenn eine erneute Abfrage nach neuen Versionen durchgeführt wird, mit dem Ziel, verschiedene Versionen mit identischer Basisversion

[9] Vgl. Abschnitt 6.5.3.1

zu finden. Diese Abfrage kann nur dann korrekte Ergebnisse liefern, wenn eine konsistente Sicht auf den Datenbestand möglich ist. Die Frage ist nun, wann dieser Zustand wieder hergestellt ist?

In Abschnitt 6.5.4.2 wurde gezeigt, dass konfligierende Versionen nur dann zustande kommen, wenn zwei Benutzer annähernd gleichzeitig separate Änderungen an einer gemeinsamen Basisversion in den Datenbestand einpflegen möchten. Wenn für die Abfrage-Intervalle gilt, dass $A = [t_0, t_1]$ und $B = [t_1, t_2]$ ist, dann markiert der Zeitpunkt t_1 die letztmögliche Überschneidung. Da sich beide Intervalle in jeweils Δt Zeiteinheiten aufteilen, lässt sich ein konsistenter Zeitpunkt direkt aus der größtmöglichen gesamten Zeitspanne, die durch beide Intervalle abgedeckt wird, finden:

$$t_c \geq t + 2\Delta t$$

Das bedeutet nichts anderes, als dass zum Zeitpunkt t_c die Sicht auf den Datenbestand für einen beliebigen Benutzer konsistent ist. Führt man nun die Abfrage erneut durch und erhält mehrere Versionen mit identischer Basisversion, aber unterschiedlicher Prüfsumme, so muss der vorliegende Versionskonflikt aufgelöst werden.

In Abschnitt 6.5.1 wurde bereits erwähnt, dass das Versionsmanagementsystem diese Funktion weitestgehend selbst erledigen kann, jedoch besteht die Notwendigkeit, das Resultat des automatischen Resolvierens durch den Benutzer prüfen zu lassen. Die Verantwortung für diese Aufgabe wird dem Benutzer zuteil, dessen eingepflegte Version den höheren Wert für den zugehörigen *Hashwert* aufweist.

Abschließend soll der Algorithmus, der den Vorgang der Konfliktbereinigung durchführt, schematisch illustriert und grob beschrieben werden. Die Grundvoraussetzung für diesen Algorithmus ist, dass zwei Versionen X und Y vorliegen, die folgende Eigenschaften aufweisen:

(i) X und Y haben eine gemeinsame Basisversion Z

(ii) Die Prüfsummen von X und Y unterscheiden sich

Die beteiligten Benutzer stellen fest, dass es konfligierende Versionen gibt. Zunächst muss geprüft werden, welcher Benutzer nun für das korrekte Zusammenführen der konfligierenden Versionen verantwortlich ist. Der Benutzer, der nicht verantwortlich ist, beendet an dieser Stelle den Algorithmus. Ausgehend von einer gemeinsamen Basisversion Z werden die Differenzen zu den eingepflegten Versionen X und Y untersucht (Diff über (Z,X) und (Z,Y)). Basierend auf den gewonnenen Informationen wird eine gemeinsame neue Version erzeugt, deren Struktur sich folgendermaßen zusammensetzt:

- Wurde eine Zeile in keiner bzw. maximal einer Nachfolgerversion (X oder Y) verändert, so kann die betreffende Zeile direkt in den zusammengeführten Text übernommen werden.
- Wurde eine Zeile in beiden Nachfolgerversionen (X und Y) verändert, so werden beide geänderten Zeilen übernommen, aber als konfligierend im Text markiert, da der Algorithmus diesen Konflikt nicht lösen kann.

Nachdem dieser Vorgang beendet ist, muss der Benutzer den resultierenden, zusammengeführten Text überprüfen, da eventuell Konflikte nicht gelöst werden konnten oder die Übernahme

von veränderten Zeilen semantische Unstimmigkeiten in die zusammengeführte Version bringen würden. Sofern der Benutzer das Resultat editiert oder bestätigt hat, kann auf Basis des zusammengeführten Textes eine neue Version gebildet werden, deren Versionsnummer und Basisversion nach Abschnitt 6.5.4.1 angepasst werden müssen.

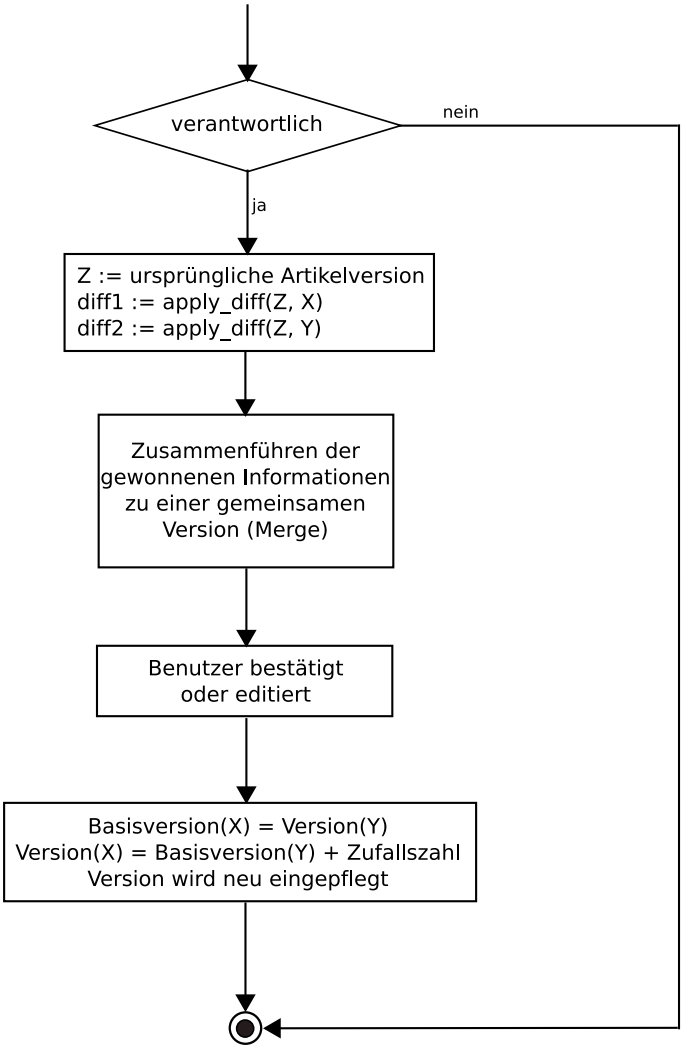


Abbildung 6.3: Ablauf des Konfliktlösens

Kapitel 7

Anwendungsprotokoll

> Hüte dich vor dem, der nur ein Buch besitzt.
>
> *Thomas von Aquin*

Sämtliche Knoten in einem Peer-to-Peer Netzwerk sind diesem System unter einem gemeinsamen Leitmotiv beigetreten. Um kollaborativ an Aufgaben im Kontext eines solchen Systems arbeiten zu können, müssen Peers untereinander kommunizieren und Daten austauschen. Die Kommunikation findet dabei üblicherweise in Form von kleinen Datenpaketen statt, die Nachrichten[10] genannt werden. Typischerweise besteht für jeden Anwendungsfall, der dezentral bearbeitet wird, die Notwendigkeit einer separaten Nachricht. Nachrichten kodieren die entsprechenden Informationen maschinenlesbar, so dass zwei Peers direkt miteinander kommunizieren können und die Partizipation des Benutzers nicht erforderlich ist. Dieses maschinenlesbare Format muss exakt spezifiziert werden, damit gewährleistet ist, dass jeder Peer Nachrichten korrekt versenden, aber auch empfangen und auswerten kann. Die Spezifikation der Gesamtheit aller Nachrichten für eine dezentralisierte Anwendung wird als Anwendungsprotokoll verstanden. Das Anwendungsprotokoll liefert die exakte Beschreibung, *wie* Nachrichten verschickt werden, aber auch *wann* die Notwendigkeit für eine entsprechende Nachricht gegeben ist.

Gegenstand dieses Kapitels ist es, zunächst die Anforderungen an die Kommunikation für das Peer-to-Peer Wiki aufzustellen. Im Folgenden werden darauf basierend die Protokolleinheiten des Anwendungsprotokolls erarbeitet.

7.1 Anforderungen

Die Anforderungen für das Anwendungsprotokoll leiten sich aus der Anforderungsanalyse ab, die bereits in Kapitel 5 durchgeführt worden ist. Konkret werden die nachfolgenden Anwendungsfälle unterschieden:

- Artikel in Volldarstellung finden
- Artikel in Volldarstellung beziehen
- Suchanfragen ausführen
- Alle Versionen zu einem Artikel finden

[10] Eine Nachricht ist eine Protokolleinheit.

- Eine Version eines Artikels beziehen
- Einen Artikel veröffentlichen oder aktualisieren

Die interne, maschinenlesbare Repräsentation der Nachrichten erfolgt über XML. Zugehörige DTD-Spezifikationen zu den einzelnen Nachrichten können dem Anhang A entnommen werden.

In Kapitel 4 wurde gezeigt, dass prinzipiell zwei verschiedene Bubble-Klassen existieren. Abbildung 7.1 stellt jene Nachrichten des Applikationsprotokolls dar, die den BubbleCast als Kommunikationsprimitive nutzen und veranschaulicht die Zuordnung zu der entsprechenden Bubble-Klasse. Sämtliche BubbleCast-Nachrichten werden mit einem Certainty-Wert $c = 3$ parametrisiert, was einer Trefferwahrscheinlichkeit von $\approx 99.99\%$ entspricht [16]. Für den Balance-Faktor muss die Anforderung eingehalten werden, dass das Produkt der Balance-Werte für Abfrage- und Daten-Bubble 1 entspricht[11]. Da zu diesem Zeitpunkt keine simulativen oder praktischen Ergebnisse vorliegen, die konkrete Werte für Balance-Faktoren unter bestimmten Anwendungsszenarien vorschlagen und eine eigene Evaluation außerhalb des Fokus dieser Arbeit liegt, erfolgt die Parametrisierung mit $b = 1$ sowohl für BubbleCast-Nachrichten der Bubble-Klasse A als auch der Bubble-Klasse B.

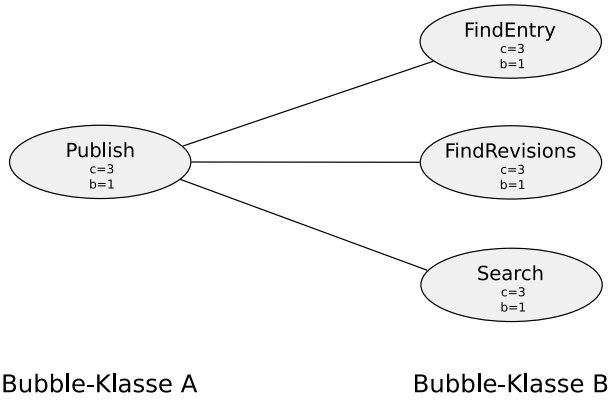


Abbildung 7.1: Einordnung und Parametrisierung der BubbleCast-Nachrichten

7.2 Protokolldefinition

Für die in Abschnitt 7.1 angeführten Anwendungsfälle sollen nun Protokolleinheiten spezifiziert werden. Typischerweise erfolgt die Bearbeitung eines Anwendungsfalls nach dem *request & response* Modell. Das heißt, zu jeder gestellten Anfrage, existiert eine Nachricht, die diese Anfrage gegebenenfalls beantworten kann. Antworten müssen auf Seiten des anfragenden Peer jedoch der ursprünglichen Anfrage zugeordnet werden können. Zu diesem Zweck führt jede Protokolleinheit ein zusätzliches Feld `MessageID` mit. Dieser Identifikator kann prinzipiell beliebig vergeben werden, muss jedoch bis über den Timeout der Anfrage eindeutig bleiben. Peers, die eine Anfrage beantworten, beziehen sich in ihrer Antwort explizit auf die erhaltene `MessageID`. Eine Ausnahme bildet die Veröffentlichung und Aktualisierung eines Artikels. Da die zugehörige

[11] Vgl. Abschnitt 4.1

Protokolleinheit aus Performanzgründen nicht mit einer Nachricht im Falle des Erfolgs quittiert werden muss, ist es auch nicht notwendig, eine konkrete `MessageID` zu vergeben. Nachrichten, die per Unicast gesendet werden, enthalten die zusätzlichen Felder `Address` und `Port`, welche die Herkunft der Nachricht belegen.

Die Beschreibung der Protokolleinheiten geht darauf ein, welchen Adressierungstyp die Nachricht (BubbleCast oder Unicast) verfolgt, welche Parameter dem Inhalt inbegriffen sind und in welchen Situationen eine Antwort auf die jeweilige Nachricht zurück gesendet werden muss.

7.2.1 Artikel in Volldarstellung finden

FindEntry

Adressierungstyp : BubbleCast

Argumente : Titel

Möchte der Benutzer den Inhalt eines bestimmten Artikels angezeigt bekommen, so ist es notwendig, dass sich das Wiki zunächst einen Überblick verschafft, welche Versionen zu dem gewünschten Artikel im Netzwerk vorhanden sind. Die Anwendung verschickt die Nachricht `FindEntry` und gibt als Parameter den Titel des Artikels an.

FindEntryResponse

Adressierungstyp : Unicast

Argumente : Titel, Basisversion, Versionsnummer und Prüfsumme

Die erwartete Antwort auf `FindEntry` ist `FindEntryResponse`. Diese Nachricht soll von einem Peer verschickt werden, wenn sie zuvor eine `FindEntry`-Nachricht erhalten hat und die Anfrage beantworten kann. Die Parameter von `FindEntryResponse` enthalten wiederum den Titel des Artikels, zusätzlich aber weitere Informationen bezüglich der Basisversion, Versionsnummer und Prüfsumme (eine vollständige Beschreibung des Artikels anhand all seiner Metadaten ist nicht notwendig). Da der Adressierungstyp von `FindEntry` der BubbleCast ist, werden auf Seiten des anfragenden Peer mehrere Antworten erwartet. Der anfragende Peer hat nach Eingang dieser Antworten die Möglichkeit, die gewünschte (in den meisten Fällen aktuellste) Version daraus zu filtern.

7.2.2 Artikel in Volldarstellung beziehen

GetEntry

Adressierungstyp : Unicast

Argumente : Titel

Über die Nachricht `FindEntry` kann eine Menge von Metadaten zu einem gewünschten Artikel gewonnen werden. Die Anwendung filtert aus dieser Menge die Artikeldaten heraus, die der aktuellsten Fassung entsprechen. Diese Version muss nun explizit per `GetEntry`-Anfrage bezogen werden. Die Anfrage benötigt lediglich den Titel des Eintrags und wird direkt an den anbietenden Peer gesendet.

GetEntryResponse

Adressierungstyp : Unicast

Argumente : Artikel in Volldarstellung und zugehörige Metadaten

Der anbietende Peer beschafft die aktuellste Version dieses Artikels in Volldarstellung aus seinem lokalen Datenbestand und schickt die entsprechenden Daten mittels `GetEntryResponse` zurück an den anfragenden Peer.

7.2.3 Suchanfragen ausführen

Search

Adressierungstyp : BubbleCast

Argumente : Suchtext

Durch die `Search`-Nachricht können komplexe Suchanfragen per BubbleCast in einem BubbleStorm-Netzwerk verteilt werden. Die Suchanfrage für das Wiki beinhaltet dazu lediglich einen Suchtext, der sowohl für die Titel- als auch Volltextsuche herangezogen werden kann.

SearchResponse

Adressierungstyp : Unicast

Argumente : Menge von Resultaten, wobei jedes Resultat folgende Attribute beinhaltet: Titel, Autor, Relevanz und kurzer Textausschnitt, indem der Suchtext vorkommt

Sofern ein Peer eine `Search`-Nachricht empfängt, prüft er den darin enthaltenen Suchtext gegen seinen lokalen Datenbestand. Liefert diese Prüfung Daten, so muss die Nachricht mit `SearchResponse` beantwortet werden. `SearchResponse` kann mehrere Resultate kapseln. Jedes Resultat beinhaltet Angaben zur Relevanz des Treffers, zur Basisversion und Versionsnummer des Treffers und liefert über einen kurzen Textauschnitt Auskunft darüber, in welchem Bezug die Suchbegriffe innerhalb des Textes vorkommen.

7.2.4 Alle Versionen zu einem Artikel finden

FindRevisions

Adressierungstyp : BubbleCast

Argumente : Titel

Um die Versionshistorie zu einem Artikel lückenlos anzeigen zu können, müssen alle im Netzwerk verfügbaren Versionen zu einem Artikel gelistet werden. Unter Umständen ist dieser Vorgang redundant, nämlich genau dann, wenn sich bereits sämtliche Versionen im lokalen Repository des anfragenden Peer befinden. In der Regel ist es jedoch so, dass keine vollständige Versionshistorie lokal vorliegt. Die Anwendung bedient sich der Nachricht `FindRevisions` und beinhaltet als Argument den Titel zu dem korrespondierenden Artikel.

FindRevisionsResponse

Adressierungstyp : Unicast

Argumente : Titel, Menge von Versionsinformationen, jeweils mit Basisversion, Versionsnummer, Autor, Datum und der Information, ob diese Version als Änderungsinformation oder in Volldarstellung vorliegt

Der Peer, der eine `FindRevisions`-Nachricht erhält, prüft in seinem lokalen Datenbestand, ob sich dort Versionen zu dem entsprechenden Artikel befinden. Ist dies nicht der Fall, so ist eine Benachrichtigung nicht erforderlich. Im Erfolgsfall wird an den anfragenden Peer `FindRevisionsResponse` gesendet. Diese Nachricht aggregiert eine oder mehrere Versionen, die den Anforderungen der Anfrage genügen und anhand einer Teilmenge ihrer Metadaten repräsentiert werden.

7.2.5 Eine Version eines Artikels beziehen

GetRevision

Adressierungstyp : Unicast

Argumente : Titel und Versionsnummer

Die Anwendung bekommt über `FindRevisionsResponse` von verschiedenen Peers eine Ansammlung aller im Netzwerk existierenden Versionen zu einem bestimmten Artikel. Der Benutzer hat nun über die vollständige Versionshistorie die Möglichkeit, eine Version zur Ansicht rekonstruieren zu lassen. Sofern die Versionshistorie in Delta-Kodierung gar nicht oder nur lückenhaft vorhanden ist, müssen einzelne Versionen[12] über das Netzwerk bezogen werden. Die Anwendung nutzt hierzu die `GetRevision`-Nachricht, die explizit eine nicht vorhandene Versi-

[12] Es ist vom konkreten Anwendungsfall abhängig, ob diese Versionen in Form einer Änderungsinformation oder in Volldarstellung bezogen werden müssen.

on von einem Peer, von dem sie zuvor eine `FindRevisionsResponse`-Nachricht erhalten hat, bezieht.

GetRevisionResponse

Adressierungstyp : Unicast

Argumente : Versionsinhalt und Metadaten zu dieser Version

Eine empfangene `GetRevision`-Nachricht wird grundsätzlich mit `GetRevisionResponse` beantwortet. Die Antwort beinhaltet die vollständigen Versionsinformationen.

7.2.6 Einen Artikel veröffentlichen oder aktualisieren

Publish

Adressierungstyp : BubbleCast

Argumente : Versionsinhalt in Volldarstellung und Metadaten zu dieser Version

Neu angelegte oder geänderte Artikel müssen in ihrer aktuellen Fassung im Netzwerk publiziert werden. Der Versionsinhalt wird immer in Volldarstellung übermittelt. Die Nachricht beinhaltet des Weiteren sämtliche Metadaten, die zu der neuen Version gespeichert worden sind. Ein Peer, der eine `Publish`-Nachricht empfängt, gliedert die neuen Versionsinformationen in sein lokales Repository ein. Das induziert den Diff zu der letzten Version, sofern diese lokal vorhanden ist. Eine `Publish`-Nachricht muss nicht mit einer Antwort quittiert werden.

Kapitel 8

Design und Implementierung

> Alles ist, wie es ist, weil es so geworden ist.
>
> *D'Arcy Wentworth Thompson*

Dieses Kapitel beschäftigt sich mit dem Design und der Implementierung des Peer-to-Peer-basierten Wikis. Um die Modularisierung der Applikation zu fördern, wurden einzelne Klassen funktionalen Komponenten zugeordnet. Diese Komponenten existieren unabhängig voneinander und kommunizieren über wohldefinierte Schnittstellen untereinander. Obwohl die Applikation ein Projekt kleinerer Größenordnung ist, macht diese strikte Trennung der Verantwortlichkeiten durchaus Sinn, da einzelne Komponenten dadurch flexibel - weil austauschbar - gehalten und bereits sehr früh getestet werden können.

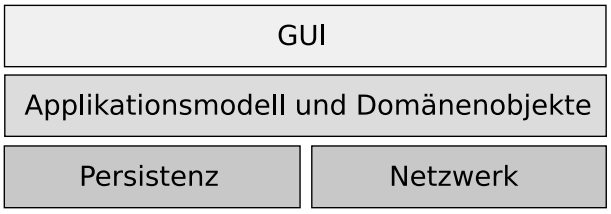


Abbildung 8.1: Komponenten der Applikation (Ebenen)

Für die Kommunikation der Komponenten untereinander zeichnet sich die Klasse `WikiController` verantwortlich. Die Klasse agiert als zentraler Koordinator für Anfragen, die von einer Komponente an eine andere, beteiligte Komponente weitergeleitet werden müssen. Dazu kapselt `WikiController` konkrete Implementierungen unter den statischen Typen der entsprechenden Schnittstelle einer jeden Komponente. Konkret sind dies: `ModelInterface`, `NetworkClient-Interface`, `PersistenceInterface`.

Im Zuge dieses Kapitels soll jede Komponente kurz beschrieben werden. Die Ausführungen beziehen sich dabei auf eine Top-Level-Sicht der einzelnen, beteiligten Packages. Das heißt, es wird beschrieben, welche Packages und Klassen an der Funktionsweise beteiligt sind, aber nicht, wie diese konkret implementiert oder welche Design Patterns berücksichtig worden sind. Implementierungsdetails können der dem Programm beiligenden Javadoc-Dokumentation entnommen werden. Die in diesem Kapitel gezeigten Klassendiagramme verstecken aus Gründen

der Übersicht Methoden und Attribute. Die ausführlichen Varianten dieser Diagramme finden sich in Anhang B.

8.1 Technische Details

Nachfolgende Tabelle gibt einen interessanten Einblick über Metriken, die den Projektumfang beschreiben.

Metrik	**Wert**
Anzahl Codezeilen	11496
Anzahl Packages	31
Anzahl Klassen	149

Um die Applikation zu starten, muss mindestens ein JRE[13] der Version 6.0 auf dem System installiert sein. Des Weiteren muss beachtet werden, dass sich das Projekt des BubbleStorm Prototypen im Classpath befindet. Das Wiki kann über die Klasse `wiki.frontend.WikiUi` gestartet werden.

8.2 Verwendete Bibliotheken

Dieser Abschnitt soll einen Überblick über die externen Bibliotheken geben, die innerhalb des Projekts genutzt werden. Eine hinreichende Überprüfung der Funktionsweise dieser Bibliotheken wird als gegeben vorrausgesetzt.

Bliki Wikipedia Engine Der Zweck der Bliki Wikipedia Engine[14] ist es, vorhandene Daten im Format der Wiki Markup Language in eine benutzerfreundliche Darstellungsform zu transformieren. Das Projekt unterstützt die Umwandlung nach HTML und eignet sich daher hervorragend für die Präsentation der Artikel im Wiki.

Apache Commons Apache Commons[15] ist Teil des Apache Software Foundation-Projektes und repräsentiert eine Sammlung diverser Bibliotheken, die wiederverwendbare Java-Komponenten bereitstellen. Jede dieser Komponenten ist so organisiert, dass sie ohne Schwierigkeiten in verschiedenen Projektumgebungen eingesetzt werden kann. Zu den Komponenten zählen beispielsweise Hilfsfunktionalitäten für die Verarbeitung von Zeichenketten, aber auch Bibliotheken, die Kodierungen in alternative Darstellungsformen unterstützen. Das Wiki nutzt die Komponente `Codec`, um Base64-Kodierungen für Bilddateien durchzuführen, sowie die Komponente `Lang`, um grundsätzliche Methoden zu nutzen, die auf Zeichenketten operieren.

java-diff Das Projekt java-diff[16] setzt das Problem der *longest common subsequence* in der Programmiersprache Java um und liefert die Differenzen zwischen zwei übergebenen Texten.

[13] Java Runtime Environment
[14] http://www.iterating.com/products/Bliki-Wikipedia-Engine
[15] http://commons.apache.org/
[16] http://www.incava.org/projects/java/java-diff/

Es wird innerhalb des Wiki dazu genutzt, um Änderungsinformationen zwischen einer Version und seiner Nachfolgerversion zu erzeugen.

jPatchLib jPatchLib[17] setzt einen Patch-Algorithmus nach der Vorlage des GNU Patch in Java um. Mit Hilfe dieser Bibliothek können die Informationen, die aus einem Diff zwischen zwei konsekutiven Versionen gewonnen werden, dazu genutzt werden, um die ursprüngliche Version wiederherzustellen.

Lucene Lucene[18] ist eine Open-Source-Bibliothek, die ebenfalls als Teil des Apache Software Foundation-Projekts entwickelt worden ist. Durch die Unterstützung von Lucene lassen sich komplexe und sehr effiziente Volltextsuchen über beliebige Textinhalte in eigenen Projekten implementieren. Lucene besteht aus zwei wesentlichen Bestandteilen: Eine Komponente, die für die Indizierung von textuellen Inhalten verantwortlich ist und eine Komponente, die Abfragen verarbeitet und den vorhandenen Index nach Übereinstimmungen durchsucht. Die Abfragekomponente liefert zusätzlich zu den Resultaten ein Maß für die Relevanz eines Artikels in Abhängigkeit des Suchtextes. Für Lucene existieren eine Reihe von Erweiterungen, zu denen unter anderem das Syntax-Highlighting einer Suchtext-Umgebung für ein Resultat zählt.

Lucene eignet sich im Besonderen für die Realisierung des Peer-to-Peer Wikis, da durch die unterliegende Netzwerkstruktur von BubbleStorm beliebig komplexe Anfragen an den lokalen Datenbestand jeder Peer möglich sind. Die Verwendung von Lucene soll aufzeigen, in welchem Maße Applikationen, die auf BubbleStorm aufsetzen, von den Eigenschaften dieses Peer-to-Peer Systems profitieren.

8.3 Paketübersicht

wiki.aggregator Dieses Package beinhaltet sogenannte Aggregator-Klassen, die sich für die Anbindung des Applikationsprotokolls an die GUI oder Simulation verantwortlich zeichnen. Ein Aggregator delegiert lokale Anfragen an die Netzwerkkomponente der Applikation und wartet auf Resultate, die weiter ausgewertet und an die aufrufende Komponente (GUI oder Simulation) weitergereicht werden.

wiki.command Dieses Package beinhaltet alle Klassen, die an der Umsetzung des *Command*-Entwurfsmusters beteiligt sind.

wiki.configuration Die Klassen innerhalb dieses Packages sind für die Verwaltung der Konfiguration einer Wiki-Instanz verantwortlich.

wiki.frontend Enthaltene Klassen sind für die Realisierung der Benutzeroberfläche zuständig. Eine informelle Beschreibung der Benutzeroberfläche findet sich in Abschnitt 8.4.

wiki.model Enthaltene Klassen repräsentieren das Applikationsmodell sowie Domänenobjekte. Eine Übersicht über die Funktionsweise und Struktur dieses Packages liefert Abschnitt 8.5.

wiki.network Die Klassen innerhalb dieses Packages werden dazu genutzt, um BubbleStorm als unterliegendes Netzwerksystem in die Wiki-Applikation zu integrieren und das Applikationsprotokoll umzusetzen. Eine Übersicht über die Funktionsweise und Struktur dieses Packages liefert Abschnitt 8.7.

[17] http://developer.gauner.org/jpatchlib/
[18] http://lucene.apache.org/

wiki.persistence Enthaltene Klassen sind für das Schreiben und Laden von Domänenobjekten zuständig. Ein weiterer Bestandteil dieses Packages ist die Integration der Lucene Bibliothek. Eine Übersicht über die Funktionsweise und Struktur dieses Packages liefert Abschnitt 8.6.

wiki.simulation Dieses Package stellt die Simulationsumgebung für das Wiki bereit. Zur Simulationsumgebung zählen ebenfalls Klassen, die zur Auswertung gewonnener Resultate genutzt werden können. Fener enthält es einen Konsolen-Client, der dazu genutzt werden kann, um rudimentäre Protokolltests mit beliebig vielen lokalen Wiki-Instanzen durchzuführen.

wiki.util Dieses Package beinhaltet Hilfsklassen, die so allgemein sind, dass sie an verschiedenen Stellen des Projekts genutzt werden.

wiki.validation Dieses Package beinhaltet rudimentäre Tests einzelner Klassen und Komponenten des Projekts.

8.4 Benutzeroberfläche

Dieser Abschnitt widmet sich der grafischen Oberfläche der Applikation und soll einen informellen Überblick über die wesentlichen Komponenten liefern. An der Realisierung dieser Komponenten ist eine Vielzahl von Klassen beteiligt, die in erster Linie die Benutzbarkeit der Anwendung verbessern und nur wenig bis keine Applikationslogik kapseln, weswegen an dieser Stelle auf eine Erläuterung der entsprechenden Packages und Klassen verzichtet wird. Sämtliche referenzierten Abbildungen zu Oberflächenkomponenten finden sich im Anhang dieser Arbeit.

8.4.1 Hauptfenster

Das Layout der Applikation unterteilt sich grundsätzlich in drei voneinander unabhängige Bereiche. Über Menü und Werkzeugleiste wird es dem Benutzer ermöglicht, grundsätzliche Funktionen der Anwendung auszuführen. Der zentrale Bereich der Anwendung ist für die visuelle Repräsentation der Artikel bestimmt.

8.4.1.1 Werkzeugleiste

Die Werkzeugleiste stellt dem Benutzer eine Reihe von Funktionen zur Verfügung (Abbildung C.1). Diese sind im Einzelnen:

(1) Öffnet eine neue Registerkarte zur Suche.

(2) Öffnet den Dialog zum Hochladen von Bilddateien.

(3) Über diese Eingabeleiste können Artikel über ihren Titel adressiert werden. Existiert zu der Eingabe ein Artikel, so wird dieser in einer neuen Registerkarte angezeigt. Anderenfalls hat der Benutzer die Möglichkeit, einen Artikel unter diesem Titel anzulegen.

8.4.1.2 Register

Artikelansicht und Suche werden im Wesentlichen durch das Bedienelement *Register* ermöglicht. Sowohl die Artikelverwaltung als auch die Durchführung einer Suche werden als separate Registerkarten innerhalb dieser Struktur interpretiert. Dies bringt den entscheidenden Vorteil mit sich, dass der Benutzer komfortabel zwischen mehreren, gleichzeitig geöffneten Registerkarten wechseln kann.

Artikelverwaltung

Die funktionalen Anforderungen an die Artikelverwaltung werden durch die Artikelansicht, der Möglichkeit zum Editieren des Artikels und der Darstellung der Versionshistorie gebildet. Um den Zugriff auf diese Funktionalitäten für den Benutzer möglichst einfach und intuitiv zu halten, nutzt die Artikelverwaltung ebenfalls die Möglichkeiten einer Register-Darstellung, jedoch mit konstanten Registerkarten. Jede Registerkarte bildet dabei die visuelle Repräsentation einer funktionalen Anforderung ab.

Artikelansicht: Die Ansicht eines Artikel (Abbildung C.2) unterscheidet sich gemäß der internen Repräsentation. Liegen textuelle Inhalte vor, so erfolgt die Ausgabe grundsätzlich im HTML-Format. Sofern mediale Inhalte vorliegen, werden diese in ihrer entsprechenden Repräsentation innerhalb dieser Registerkarte dargestellt. Textuelle Inhalte können Referenzen auf andere Artikel enthalten. Ein Klick auf einen Link öffnet den referenzierten Artikel in einer separaten Registerkarte. Eine Sonderform der Referenz ist die Einbettung von Bilddateien in den laufenden Text. Referenzierte Bilddateien werden an entsprechender Stelle innerhalb des Textes dargestellt und verweisen bei Mausklick auf eine neue, separate Registerkarte, die den Artikel zu der Bilddatei öffnet. Die untere Leiste in der Artikelansicht gibt Auskunft darüber, wann der Artikel zuletzt editiert worden ist und welcher Benutzer die letzten Änderungen vollzogen hat.

Integrierter Editor: Der integrierte Editor (Abbildung C.3) wird dem Benutzer nur dann angeboten, wenn es sich bei dem Inhalt des Artikels um textuelle Daten handelt. Eine Änderung an medialen Inhalten wird insofern unterstützt, als das der Benutzer die Möglichkeit hat, unter einer neuen Version neuen Inhalt in Volldarstellung in das Netzwerk einzupflegen.

Die Oberfläche des Editors präsentiert drei wesentliche Bereiche:

- Eine Werkzeugleiste, die verschiedene Buttons anbietet, um vorgefertigte Code-Strukturen der Wiki Markup Language in den Editierbereich einzufügen.
- Ein Editierbereich, der die Inhalte eines Artikeltextes in Wiki Markup Language beinhaltet und Manipulationen durch den Benutzer erlaubt.
- Eine Button-Leiste mit der Möglichkeit, eine Vorschau zum aktuellen Artikeltext aufzurufen (Preview), Bilddaten in das Netzwerk einzupflegen (Upload) und den aktuellen Artikeltext als neue Version zu speichern (Save).

Versionshistorie: Die Versionshistorie zu einem Artikel (Abbildung C.4) wird durch eine tabellarische Übersicht repräsentiert, die absteigend nach dem Änderungszeitpunkt der einzelnen Versionen sortiert ist. Die Tabelle liefert darüber hinaus einige weitere, für den Benutzer interessante Daten zu der Version: Name des Autors und ob diese Version in Volldarstellung (Full version) oder Delta-Kodierung (Patch available) verfügbar ist. Bei

Auswahl eines Versionseintrages aus der Versionshistorie kann der Benutzer diese Version rekonstruieren und in der Artikelansicht anzeigen lassen. Selektiert der Benutzer zwei Versionseinträge, so hat er die Möglichkeit einen visuellen Diff über den ausgewählten Versionen zu erzeugen.

Suche

Abbildung C.5 zeigt die Repräsentation der Suchfunktion. Die Oberfläche unterteilt sich in zwei wesentliche Bereiche. Die obere Hälfte stellt dem Benutzer ein Eingabefeld zur Verfügung, um Suchanfragen zu formulieren. Es ist nicht nötig, eine Unterscheidung zwischen Volltext- und Artikelsuche zu treffen, da die integrierte Suchmaschine beide Fälle simultan behandeln kann. Die untere Hälfte ist für die Ausgabe der Trefferliste reserviert. Zu einem Treffer werden weitere Informationen dargestellt, wie beispielsweise ein Maß für die Relevanz des Treffers in Anbetracht des Suchtextes. Darüberhinaus gibt ein Treffer, sofern möglich, Auskunft darüber, in welcher textuellen Umgebung der Suchtext im Artikel aufgetreten ist. Der Benutzer hat die Möglichkeit, per Mausklick auf einen Treffer den korrespondierenden Artikel zu laden und in einer separaten Registerkarte anzeigen zu lassen.

8.4.2 Dialoge

8.4.2.1 Konfiguration der Anwendung

Abbildung C.6 zeigt den Konfigurationsdialog. Über diesen Dialog hat der Benutzer die Möglichkeit, verschiedene Einstellungen für die Anwendung vorzunehmen.

8.4.2.2 Herstellen von Netzwerkverbindungen

Abbildung C.7 zeigt den Verbindungsdialog. Dieser Dialog wird zum Programmstart geöffnet. Damit das Wiki benutzbar ist, muss über diesen Dialog eine Verbindung aufgebaut werden, anderenfalls (Wahl von *Cancel*) terminiert die Programmausführung. Der Benutzer hat die Möglichkeit, sich über die Angabe eines lokalen oder entfernten Hostcaches zu verbinden oder aber eine beteiligte Peer direkt über ihren Hostnamen und ihren Port anzusprechen, um einem vorhandenen Netzwerk beizutreten. Sofern die Eingaben des Benutzers nicht korrekt sind, öffnet sich ein Hinweisfenster mit entsprechender Hilfestellung. Sollte eine Verbindung trotz korrekter Eingaben nicht zustande kommen, so wird durch den unterliegenden Netzwerk-Layer ein eigenes BubbleStorm-Netzwerk erzeugt, dem andere Wiki-Instanzen beitreten können.

8.4.2.3 Vorschaufenster für den Editor

Das Vorschaufenster (Abbildung C.8) ist Bestandteil des Editors. Es bietet die selbe Funktionalität wie die Artikelansicht, ist jedoch aus Gründen der Übersicht in einen separaten Dialog ausgelagert.

8.4.2.4 Bilder-Upload

Mediale Inhalte in Form von Bilddateien kann der Benutzer mit Hilfe des Dialogs in Abbildung C.9 in das Netzwerk einpflegen. Der Benutzer gibt den Pfad zur Datei entweder manuell oder mittels Datei-Auswahl-Dialog an und vergibt für die dahinterstehende Bilddatei einen logischen Bezeichner, der als Titel innerhalb des Wikis genutzt wird. Das Vorschaufenster rechts im Dialog gibt dem Benutzer die Möglichkeit, vor dem endgültigen Einpflegen die Bilddaten in skalierter Form einzusehen.

8.4.2.5 Visueller Diff

Dieser Dialog (Abbildung C.10) umfasst zwei voneinander getrennte Textbereiche, in denen jeweils eine zu prüfende Artikelversion dargestellt wird. Ein Diff beider Versionen arbeitet auf einzelnen Textabschnitten und gibt Auskunft darüber, ob ein Abschnitt verändert, hinzugefügt oder gelöscht worden ist. Diese Information ist farblich kodiert, so dass direkt einsehbar ist, welche Anteile des Artikels verwendet worden sind und in welcher Form diese Veränderung erfolgt ist.

Nachfolgende Auflistung zeigt die Farbkodierung für differierende Textabschnitte:

- *Grün*: Der markierte Abschnitt wurde in der neuen Version hinzugefügt.
- *Rot*: Der markierte Abschnitt ist in der neuen Version nicht mehr vorhanden.
- *Gelb*: Der markierte Abschnitt wurde verändert.

8.5 Applikationsmodell

Das Applikationsmodell zeichnet sich für die interne Repräsentation und Verwaltung der Artikeldaten verantwortlich. Die Schnittstelle `ModelInterface` abstrahiert von konkreten Implementierungen für diese Anforderungen und spezifiziert Methoden, die grundlegende Operationen auf den aggregierten Artikeldaten erlauben. Eine konkrete Implementierung dieser Schnittstelle wird durch die Klasse `WikiModel` realisiert. `WikiModel` aggregiert zunächst alle lokal gespeicherten Artikel. Im Verlauf der Sitzung werden alle über das Netzwerk bezogenen Artikel ebenfalls in `WikiModel` aggregiert, aber nicht lokal gespeichert. Die Aggregation der Artikeldaten erfolgt über eine Listenstruktur, die Objekte des Typs `WikiEntry` sammelt. `WikiEntry` kapselt wiederum eine Listenstruktur über alle vorhandenen Versionseinträge, die unter dem Typ `VersionInfo` erscheinen. Die Klasse stellt des Weiteren einige grundlegende Methoden bereit, um die vorgehaltenen Versionen zu verwalten und intern verarbeiten zu können. Grundsätzlich lassen sich für einen Artikel invariante und variante Attribute unterscheiden. Invariante Attribute sind konstant über alle Versionen eines Artikels und werden somit innerhalb der Klasse `WikiEntry` vorgehalten. Zu diesen Attributen zählen:

- Der konkrete Typ des Artikeleintrags (entweder textuell oder medial), repräsentiert durch die Enumerator-Klasse `EntryType`.
- Der Titel des Artikels, repräsentiert durch einen `String`-Wert.

Invariante Attribute sind mit jeder gespeicherten Version zu einem Artikel veränderlich und werden daher in der Klasse `VersionInfo` vorgehalten. Zu diesen Attributen zählen:

- Der Autor dieser Version.
- Die Basisversion und Versionsnummer der gespeicherten Version.
- Eine MD5-Prüfsumme über den konkreten Inhalt.
- Der konkrete Inhalt der Version, der unter der Schnittstelle `WikiContent` erscheint, deren konkrete Implementierungen textuellen (`TextContent`) bzw. medialen (derzeit nur `ImageContent`) Inhalt kapseln.
- Das Datum der Einpflege dieser Version.
- Die Darstellung der Version (Diff oder Volltext), repräsentiert durch die Enumerator-Klasse `ContentType`.

Ein zusätzliches `boolean`-Flag in `VersionInfo` gibt an, ob diese Version während einer Sitzung über das Netzwerk bezogen wurde. Dieses Flag *kann* ferner dazu genutzt werden, replizierte Versionen von Versionen zu unterscheiden, die nicht unter Replikation stehen. Die Schnittstellen `WikiEntryObserver` und `WikiEntryObservable` erlauben es realisierenden Klassen, Observer-Funktionalitäten für Domänenobjekte zu nutzen. Die Klasse `HistoryItem` dient zur Kapselung von Daten, die für die Versionshistorie relevant sind.

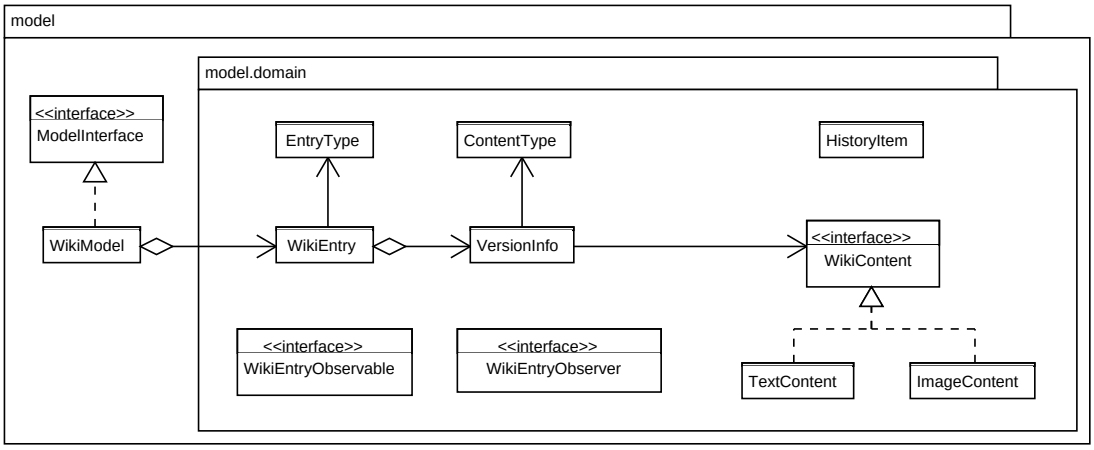


Abbildung 8.2: Paketübersicht Modell

8.6 Persistenz

Die Aufgabe dieser Ebene ist es, die Daten der Domänenobjekte persistent abzulegen, so dass sie zu einer späteren Sitzung erneut geladen werden können. Die Schnittstelle `PersistenceInterface` beschreibt hierfür Methodendeklarationen, die minimalen Anforderungen an den Umgang mit gespeicherten Domänenobjekten genügen. Sämtliche auftretenden Ausnahmen dieses Pakets werden über `PersistenceException` geworfen und behandelt. Eine einfache, datei-orientierte

Realisierung der Schnittstellte `PersistenceInterface` wird über die Klasse `SimpleDiskStorage` bereitgestellt. An dieser Stelle sind andere Realisierungen, die beispielsweise die Persistenz über eine Datenbankschicht ermöglichen, denkbar und hinsichtlich großen Datenvolumen sicherlich auch sinnvoll. Die Speicherung von Domänenobjekten erfolgt im XML-Format. Hierbei ist zu unterscheiden, ob der Artikel in Volldarstellung oder als Änderungsinformation vorliegt. Für die konkrete Speicherung zeichnet sich die Methode `processSaveEvent` verantwortlich. Die Methode nutzt diverse Compiler (`DiffXMLCompiler`, `EntryXMLCompiler`), um zu speichernde Domänenobjekte nach XML zu transformieren. Eine Formatbeschreibung befindet sich im Anhang A. Die Methode `loadEntries` ist für das Laden gespeicherter Artikeldaten aus einem gegebenen Verzeichnis verantwortlich (vgl. lokales `String`-Attribut `entryPath`). Dazu nutzt die Methode die interne Klasse `FileListProcessor`, die eine Dateiliste für Artikel in Volldarstellung und eine Dateiliste für Änderungsinformationen konsumiert. Diese Dateilisten stellen eine flache Hierarchie des Datenbestandes dar, der im Folgenden in eine baumartige Struktur für jeden Artikel transformiert werden soll. XML-Parser der korrespondierenden Klassen (`DiffParser`, `EntryParser`) sorgen dafür, dass die entsprechenden Daten aus den gespeicherten XML-Dokumenten extrahiert werden. `FileListProcessor` implementiert die Schnittstelle `ParserCallback`, welches die Methode `parserFinished` spezifiziert. Diese Methode wird dann aufgerufen, wenn ein XML-Parser ein gegebenes XML-Dokument vollständig geparst und die Versionsinformationen in ein Objekt des Typs `VersionInfo` (vgl. Abschnitt 8.5) transformiert hat.

Die Integration der Volltextsuche erfolgt über die Lucene API. Die Klasse `LuceneIntegration` implementiert hierbei sämtliche Zugriffsmethoden auf diese Bibliothek. Konkret werden folgenden Funktionalitäten realisiert:

(i) Indizierung der aktuellsten Version eines lokal vorhandenen Artikels.

(ii) Entfernen eines bereits indizierten Artikels aus dem lokalen Index.

(iii) Durchführung der Volltextsuche.

Die Realisierung von Funktionalität (i) benötigt eine Dokumentenklasse `WikiDocument`, die sich verantwortlich dafür zeigt, ein gegebenes Objekt des Typs `WikiEntry` für den Lucene Indexer aufzubereiten. Hierbei werden diejenigen Attribute des Artikels extrahiert, die in den lokalen Lucene-Index als indizierte Felder übernommen werden sollen. Diese Attribute sind im Konkreten:

- Der Inhalt der letzten Artikelversion (essentiell für die Volltextsuche).
- Der Titel des Artikels.
- Das Einpflegedatum der letzten Artikelversion.
- Einen MD5-Hashwert über den Artikelinhalt.
- Der konkrete Typ des Artikeleintrags.
- Der Autor der letzten Artikelversion.

Die Speicherung zusätzlicher Attribute, die nicht für die Volltextsuche relevant sind, erscheint zunächst redundant, ist aber durchaus sinnvoll und gewollt. Eine Suchanfrage kann somit direkt weitere Attribute zurückliefern, ohne dass über den Weg des Applikationsmodells diese Daten umständlich erfragt werden müssten. Die Speicherung des MD5-Hashwerts ist essentiell für die Realisierung von Funktionalität (ii). Über diesen Hashwert ist es möglich, alle Einträge im Lucene-Index zu dem korrespondierenden Artikel zu lokalisieren, um diese bei Bedarf entfernen

zu können. Artikeldaten werden dann aus dem Lucene-Index entfernt, wenn eine neue Version lokal gespeichert wurde und in den Index übernommen werden soll. Der Lucene-Index indiziert somit stets die aktuellste lokal gespeicherte Version. Die Implementierung von Funktionalität (iii) sucht nach einem übergebenen Suchtext in den indizierten Feldern für die Attribute Titel und Inhalt. Die Suchergebnisse werden im Folgenden für die Anzeige aufbereitet. Über eine Erweiterung der Lucene API[19] lässt sich ferner ein kurzer Textausschnitt gewinnen, der die textuelle Umgebung zeigt, in welcher der Suchtext auftritt. Die Anteile des Suchtextes können dabei farbig hervorgehoben werden. Nach der Aufbereitung wird eine Client-Klasse, welche die Schnittstelle `QueryCallbackReceiver` realisiert, über die gefundenen Suchergebnisse via Callback informiert.

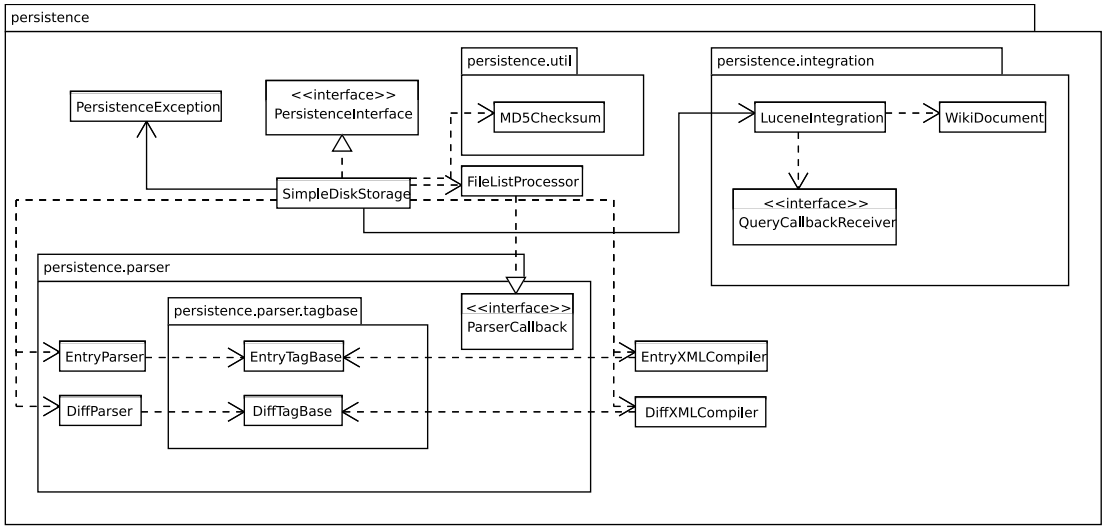


Abbildung 8.3: Paketübersicht Persistenz

8.7 Netzwerk

Die Integration von BubbleStorm als zugrundeliegendes Peer-to-Peer System und Realisierung des Applikationsprotokoll ist Aufgabe dieser Komponente. Die Umsetzung dieser Aspekte findet maßgeblich innerhalb der Klasse `WikiServent` statt. Diese Klasse koordiniert jedweden Zugriff auf die Netzwerkfunktionen von BubbleStorm und läuft als eigenständiger Thread. Die Sichtbarkeit von `WikiServent` kann für andere, benutzende Klassen durch die Schnittstellen `NetworkClientInterface` und `NetworkServerInterface` reduziert werden. `NetworkClientInterface` wird beispielsweise innerhalb des Projekts von diversen Klassen genutzt, um Anfragen per BubbleCast oder Benutzerverbindung zu versenden (Client-Verhalten). Klassen, die sich dieser Funktionalität bedienen, realisieren typischerweise die Schnittstelle `ResponseReceiver`. Diese Schnittstelle deklariert die Methode `responseReceived(Object o)`, die dann aufgerufen wird, wenn Ergebnisse einer zuvor gestellten Anfrage vorhanden sind und verarbeitet wurden. Die

[19] Diese Erweiterung wurde über die *sandbox* des Lucene-Projekts bezogen.

Schnittstelle `NetworkServerInterface` hingegen ermöglicht denjenigen Klassen, die ankommende Verbindungen oder empfangene Nachrichten behandeln, eine Antwort zu erzeugen und an den anfragenden Peer zurückzusenden (Server-Verhalten).

Damit die Applikation die Netzwerkfunktionen von BubbleStorm nutzen kann, besteht eine Abhängigkeit zwischen `WikiServent` und einer Instanz von `RouterService`, die unter dem statischen Typ der Schnittstellte `IRouterService` genutzt wird. Über die Registrierung von Handler-Klassen an der Instanz von `RouterService` können eingehende BubbleCast-Nachrichten und Benutzerverbindungen an die entsprechenden, applikationsspezifischen Handler weitergeleitet und verarbeitet werden. Dies geschieht in der privaten Methode `registerMessages`. Sind diese Vorkehrungen abgeschlossen, kann die Applikation nun einem BubbleStorm-Netzwerk als voll funktionstüchtiger Netzwerkknoten beitreten. Über die Methode `quit` kann `WikiServent` jederzeit terminiert werden.

Grundsätzlich unterscheidet sich die Vorgehensweise für das Versenden von Nachrichten, die über einen BubbleCast in das Netzwerk verteilt werden, von denjenigen, die über eine Benutzerverbindung direkt zwischen zwei partizipierenden Hosts übertragen werden.

BubbleCast-Nachrichten werden über das von BubbleStorm induzierte Overlay-Netzwerk geroutet. Die Schnittstellte hierfür liefert wiederum `RouterService` über die Methode `bubblecast-(IBubbleCastMessageType)`. Protokolleinheiten, die den BubbleCast als Kommunikationsprimitive nutzen, sind foglich als realisierende Klassen der Schnittstelle `IBubbleCastMessageType` implementiert. Eine konkrete Ausprägung dieser Schnittstelle konsumiert die zu versendende Nachricht als Parameter über den Konstruktor. `WikiServent` stellt für jeden BubbleCast-Nachrichtentyp eine öffentliche Methode bereit (bspw. `bubblecastFindEntry`), die eine Instanz der korrespondierenden Nachrichtenklasse instanziert und diese an den Routing-Mechanismus von BubbleStorm delegiert.

Die Übertragung einer Nachricht mittels Benutzerverbindung erfolgt durch die Methode `Router-Service.connect` unter der Angabe der Hostadresse, des Hostports und eines `AppSocket-Receiver`. Eine konkrete Realisierung von `AppSocketReceiver`, die für das Schreiben auf einen erhaltenen Socket genutzt werden kann, wird über die Klasse `DirectResponseConnector` bereitgestellt. Diese Klasse transformiert die zu sendende Nachricht in einen `ByteBuffer` und schreibt die entsprechenden Daten als Bytestorm auf den Socket, sobald die Berechtigung zum Schreiben seitens BubbleStorm zugeteilt worden ist.

Um zu sendende Anfragen oder Antworten in das richtige XML-Format der Protokolleinheiten zu transformieren, wird die Klasse `MessageCompiler` genutzt. Diese Klasse stellt für jede Protokolleinheit eine statische Methode bereit, die entsprechende Parameter konsumiert (beispielsweise eine `WikiEntry`-Instanz für eine Publish-Nachricht) und ein `String`-Objekt liefert, welches der Formatbeschreibung dieser Protokolleinheit genügt.

Bis auf Nachrichten des Typs Publish werden zu jeder gestellten Anfrage potenzielle Antworten erwartet. Zu jeder Anfrage wird ein Identifikator `messageId` vergeben, welcher für den anfragenden Peer eindeutig sein muss. Erhaltene Antworten beziehen sich explizit auf diesen Identifikator, was eine Zuordnung von Antworten zu einer zuvor gestellten Anfrage ermöglicht. Da auf eine gesendete BubbleCast-Nachricht viele Antworten erwartet werden, aggregiert `WikiServent` alle erhaltenen Nachrichten, für die eine Zuordnung zu einer zuvor gestellten Anfrage über diesen Identifikator gefunden werden kann in einem zugehörigen `ResponseHandler`. Diese abstrakte Basisklasse ist die Grundlage für spezifische Handler. Die Basisklasse `ResponseHandler` verwaltet eine Liste von `ResponseWrapper`-Objekten, wobei jede Instanz von `ResponseWrapper` einer

vollständig geparsten Nachricht entspricht, zu der ferner die Hostadresse und der Hostport des Senders gespeichert werden. Nach einer gewissen Wartezeit[20] (*timeout*) initiiert der Handler die Bearbeitung der kollektierten Nachrichten. Die Klasse `ResponseHandler` erhält über erbende Klassen eine Referenz auf einen `ResponseReceiver`, der per Callback über die Ergebnisse der Verarbeitung informiert wird.

Die Behandlung eingehender BubbleCast-Nachrichten wird vollständig durch den Routing-Mechanismus von BubbleStorm erledigt und an die entsprechenden Handler delegiert. Diese Handler transformieren das übergebene `byte`-Array in eine `String`-Repräsentation, um die erhaltene Nachricht parsen zu können. Die Verarbeitung findet direkt in den Handlern statt und nutzt Komponenten höherer Ebenen (Domänenobjekte, Modell), um die Anfrage zu bearbeiten. Antworten auf einen BubbleCast werden grundsätzlich mit einem Unicast beantwortet, der in BubbleStorm über eine Benutzerverbindung realisiert wird und über die Methode `NetworkServerInterface.sendMessageUC` initiiert werden kann.

Eingehende Benutzerverbindungen erfordern eine weitere Indirektionsebene, da ankommende Nachrichten über eine Socket-Verbindung zunächst explizit gelesen werden müssen. Diese Funktionalität wird durch die innere Klasse `ServentObserver` realisiert (innerhalb von `WikiServent`), welche die Schnittstelle `AppSocketReceiver` implementiert. `ServentObserver` kümmert sich um das korrekte Auslesen von dem empfangenen Socket und übernimmt anschließend das Parsen der erhaltenen Nachricht sowie die Zuordnung zu einem zum ursprünglichen Anfragezeitpunkt erzeugten `ResponseHandler`.

Bevor eine erhaltene Nachricht, gleichwohl ob durch einen BubbleCast oder eine Benutzerverbindung empfangen, weiterverarbeitet werden kann, muss sie zunächst geparst werden. Diese Funktion übernimmt die Klasse `MessageParser`, die sich eines SAX-Parsers bedient, um entsprechende Daten aus dem der Nachricht zugrundeliegenden XML-Format zu extrahieren. Für `MessageParser` existieren konkrete Spezialisierungen für jeden Nachrichtentyp, welche die extrahierten Daten (die zunächst nur als `String`-Objekte vorliegen) über getter-Methoden unter dem korrekten Typ des Datums zurückliefern.

[20] Diese Wartezeiten können sich für verschiedene Nachrichtentypen unterscheiden.

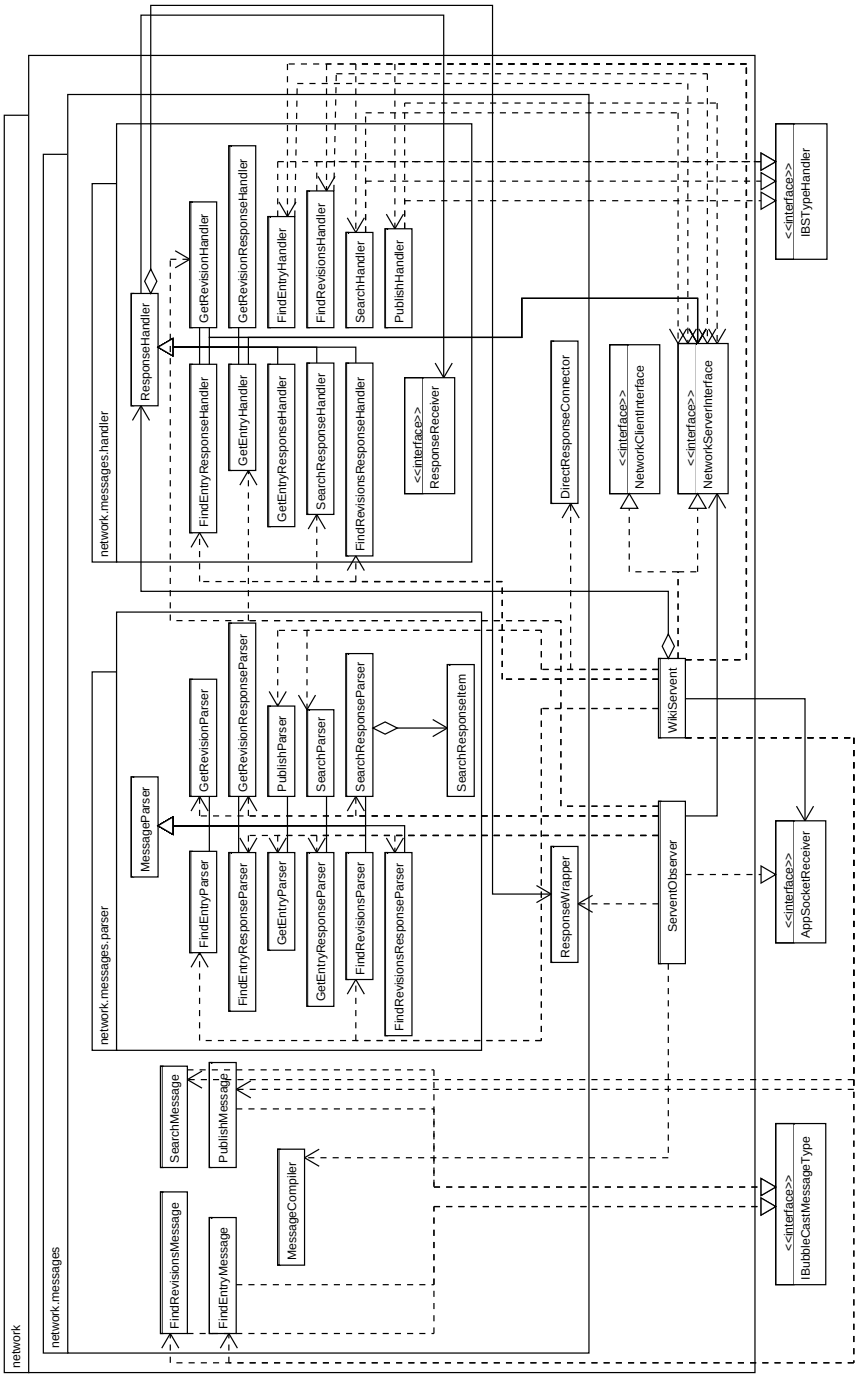


Abbildung 8.4: Paketübersicht Netzwerk

Kapitel 9

Evaluation

> Das letzte Ziel aller wissenschaftlichen Erkenntnis besteht darin, das größtmögliche Tatsachengebiet aus der kleinstmöglichen Anzahl von Axiomen und Hypothesen zu erhellen.
>
> *Albert Einstein*

Die in Kapitel vorgestellte Referenzimplementierung soll innerhalb dieses Kapitels als Basis für eine Evaluation herangezogen werden. Die Evaluation beschränkt sich hierbei auf zwei Szenarien, die in einem lokalen Netzwerk mit differierender Anzahl von beteiligten Wiki-Knoten durchgeführt werden sollen. Der Verzicht auf ein verteiltes Szenario soll kurz begründet werden. Eine verteilte Umgebung erfordert einen wesentlich höheren Aufwand für die Umsetzung, was zunächst den Rahmen dieser Bachelorarbeit deutlich sprengen würde. Des Weiteren muss in Frage gestellt werden, ob der zusätzliche Mehraufwand gerechtfertigt wäre, wenn sich die relevanten Umgebungsparameter nicht oder nur geringfügig ändern. Zweifelsohne wäre die Latenz höher, jedoch nicht in einem realistischen Maß, da hierfür eine global-verteilte Umgebung nötig wäre. Die Begrenzung der Bandbreite sowie der Nachrichtenverlust bei realen Verbindungen würden sich ebenfalls nur auf die Latenz auswirken.

Die durchgeführten Szenarien unterscheiden sich hinsichtlich der Dynamik der Netzwerkknoten. In einem lokal-statischen Szenario ist die Anzahl der Netzwerkknoten konstant über die Dauer der Simulation. Diese Knoten verlassen das Netzwerk nicht und neue Knoten treten dem Netzwerk nicht bei (kein Churn). In einem lokal-dynamischen Szenario ist eine anfängliche Zahl von Netzwerkknoten gegeben. Zu gewissen Zeitpunkten verlässt ein Knoten das Netzwerk, während ein neuer Knoten dem Netzwerk beitritt (Churn).

9.1 Aufbau der Simulation

Das Paket `wiki.simulation` stellt die nötigen Klassen bereit, um automatisierte Testläufe durchzuführen. Einstiegspunkt ist die Klasse `Simulation`, die für die Durchführung verantwortlich ist. Eine Simulation arbeitet hierbei in verschiedenen Schritten. Zunächst wird die gewünschte Anzahl an Netzwerkknoten erzeugt. Damit das hierbei entstehende Netzwerk nicht in lokale Cluster zerfällt, wartet die Simulation eine kurze Zeitperiode lang zwischen konsekutiven Netzwerkbeitritten. Jeder hinzugefügte Netzwerkknoten korrespondiert mit einem simu-

lierten Benutzer (Klasse `SimulatedUser`), der für die konkrete Ausführung von Operationen zuständig ist. Ein simulierter Benutzer führt Buch darüber, ob eine von ihm gestartete Operation erfolgreich abgeschlossen wurde oder nicht. Nachdem das Netzwerk konstruiert wurde, sorgt die Simulation für eine initiale Verteilung von zufällig gewählten Artikeln. Damit auch dieser Schritt automatisiert werden kann, ist eine Datenbasis für Artikel erforderlich, aus der ein simulierter Benutzer randomisiert Artikel holen und im Netzwerk publizieren kann. Hierfür wurden Dumps der Datenbanken von Wikipedia und Wiktionary herangezogen[21], aus denen ein `ArticleParser` die relevanten Informationen ausliest. Der Titel eines bereits publizierten Artikels wird für alle Benutzer zugänglich gemacht, so dass später folgende Operationen sich an bereits vorhandenen Artikeln orientieren können. Nach der Verteilung von Artikeln startet der eigentliche Simulationslauf in der Klasse `Simulation`. Die Simulation läuft über eine gewünschte Zeit, währenddessen nach einem gewissen Intervall wiederholend Ereignisse auftreten, die behandelt werden müssen. Der Zeitpunkt für den Eintritt eines Ereignisses wird dabei zufällig bestimmt, muss aber innerhalb einer anpassbaren unteren und oberen Grenze liegen. Ein eintretendes Ereignis startet die Suche nach einem zufälligen simulierten Benutzer, der eine zufällig gewählte zulässige Operation durchführt. Zulässige Operationen sind hierbei:

- Der Bezug eines bereits publizierten Artikels (*GetArticle*).
- Der Bezug der Versionshistorie zu einem bereits publizierten Artikel (*FetchHistory*).
- Die Rekonstruktion einer alten Version zu einem bereits publizierten Artikel (*FetchFormerVersion*).
- Das Suchen nach einem Artikeltitel oder per Volltext (*KeywordSearch*).
- Das Publizieren eines noch nicht veröffentlichten Artikels (*PublishArticle*).

Um die Komplexität der simulierten Benutzer gering zu halten, werden über die Operation *PublishArticle* ausschließlich noch nicht veröffentlichte Artikel publiziert. Wird die Simulationszeit überschritten, so ist der Simulationslauf beendet. Die Simulation erzeugt bei jedem Lauf zwei Logdateien, die im Folgenden durch die Klasse `AnalyseLog` ausgewertet werden:

- `serventlog.csv`: Die Klasse WikiServent schreibt Daten bezüglich des gesamten ein- und ausgehenden Netzwerkverkehrs in diese Datei
- `userlog.csv`: Jeder simulierte Benutzer schreibt seine durchgeführten Operationen nebst Erfolg bzw. Misserfolg und unter Angabe der Latenz der Operation in diese Datei.

`AnalyseLog` liest die Daten die über die Logdateien bereitgestellt werden und generiert Tabellen sowie Kommando- und Datendateien für Gnuplot. Bevor die konkreten Szenarien dargestellt und ausgewertet werden, muss man noch die Frage nach dem Protokollieren des Erfolgs von zulässigen Operationen auflösen. Jede Operation bedient sich eines für die Simulation spezifischen Aggregators, der das Applikationsprotokoll nutzt, um die korrespondierende Operation zu erfüllen. Schlägt dieser Aggregator fehl, so meldet er über einen Callback an den simulierten Benutzer den Misserfolg der Operation, bzw. den Erfolg bei geglückter Operation. Die entsprechenden Aggregatoren finden sich im Paket `wiki.aggregator` (Klassen mit dem Präfix `Sim`). Der *certainty*-Faktor für BubbleCast-Nachrichten ist - ebenso wie im GUI-gestützten Wiki - mit dem Wert 3 parametrisiert.

[21] `http://dumps.wikimedia.org`

9.2 Ressourcenverbrauch

Die Messung des Ressourcenverbrauchs liefert einen ersten Anhaltspunkt, wie performant die Applikation arbeitet. Es muss natürlich betont werden, dass diese Werte aufgrund des Speichermanagements der Java Virtual Machine nicht als exakt zu betrachten sind. Sie liefern lediglich qualitative Angaben über den Verbrauch der Anwendung.

Um den Speicherverbrauch eines einzelnen Wikis zu messen, wurde ein Netzwerk bestehend aus einem einzigen Knoten erzeugt. Nach einer initialen Verteilung von 250 Artikel wurde der Simulationslauf mit der Dauer einer Stunde gestartet. Gemessen wurde der Speicherverbrauch mit dem Programm JConsole vor und nach der Simulation. Die nachfolgende Tabelle zeigt die Resultate.

Initial	Heap-Speicherverbrauch	ca. 3,3 MB
	Sonstiger Speicherverbrauch	ca. 20 MB
Nach Simulation	Heap-Speicherverbrauch	ca. 11 MB
	Sonstiger Speicherverbrauch	ca. 23 MB

Nachfolgend soll der Speicherverbrauch als Funktion der Knotenanzahl dargestellt werden. Dazu wurde eine Simulation mit lokal-statischem Szenario gestartet, in der sukzessive Netzwerkknoten beitreten, bis ein Limit von 200 aktiven Knoten erreicht wurde. Die Messdaten wurden zu verschiedenen Zeitpunkten während des Netzwerkaufbaus mittels JConsole ermittelt.

Anzahl Knoten	**Heap-Speicherverbrauch**	**Sonstiger Speicherverbrauch**
0	ca. 3,2 MB	ca. 19,6 MB
25	ca. 7,0 MB	ca. 20,9 MB
50	ca. 10,9 MB	ca. 20,8 MB
100	ca. 19,3 MB	ca. 21,0 MB
150	ca. 28,5 MB	ca. 21,1 MB
200	ca. 40,9 MB	ca. 21,1 MB

Die hier gezeigten Werte zeigen einen ungefähr linearen Anstieg des Heap-Speicherverbrauchs in Abhängigkeit von der Anzahl Knoten. Nach der Erzeugung des Netzwerks liegt der durchschnittliche Heap-Speicherverbrauch bei ca. 204,5. KB pro aktivem Netzwerkknoten. Nach anschließender Simulation über eine Stunde, bei der insgesamt 431 Operationen durchgeführt und 31475 Nachrichten gesendet und empfangen wurden, betrug der Heap-Speicherverbrauch insgesamt (für alle Knoten) ca. 217 MB, während der sonstige Speicherverbrauch der Simulation bei ca. 26,2 MB lag.

9.3 Lokal-statisches Szenario

Zur Simulation lokal-statischer Szenarien wurden Konfigurationen mit 50 und 300 aktiven Netzwerkknoten betrachtet. Die simulierten Szenarien liefen jeweils über eine Stunde. Die nachfolgenden Tabellen schlüsseln den hierbei induzierten Netzwerkverkehr nach Nachrichtentyp auf und liefern Angaben zu mittlerem Wert für Nachrichtengröße sowie gesamtem Netzwerkverkehr.

Netzwerkverkehr nach Nachrichtentypen für Simulation mit 50 Knoten

Nachrichtentyp	Gesendet			Empfangen		
	Anzahl	Größe (Byte)	∅ (Byte)	Anzahl	Größe (Byte)	∅ (Byte)
FindEntry	61	3738	61	1325	88771	67
FindEntryResponse	540	128963	239	532	126909	239
GetEntry	82	7611	93	82	7611	93
GetEntryResponse	82	567018	6915	80	495126	6189
FindRevisions	129	8260	64	2652	183712	69
FindRevisionsResponse	1059	415053	392	1048	409702	391
GetRevision	85	9105	107	85	9105	107
GetRevisionResponse	85	511309	6015	83	462490	5572
Search	59	3606	61	1226	80173	65
SearchResponse	1226	1642744	1340	1077	1323264	1229
Publish	157	867267	5524	3671	21095036	5746
Total	3565	4164674	1168	11861	24281899	2047

Netzwerkverkehr nach Nachrichtentypen für Simulation mit 300 Knoten

Nachrichtentyp	Gesendet			Empfangen		
	Anzahl	Größe (Byte)	∅ (Byte)	Anzahl	Größe (Byte)	∅ (Byte)
FindEntry	62	3658	59	3700	239377	65
FindEntryResponse	847	200644	237	748	176811	236
GetEntry	79	7141	90	79	7141	90
GetEntryResponse	79	540303	6839	78	538199	6900
FindRevisions	124	7964	64	7587	532516	70
FindRevisionsResponse	1586	551781	348	1573	547729	348
GetRevision	84	8968	107	84	8968	107
GetRevisionResponse	84	518798	6176	83	483508	5825
Search	59	3553	60	3507	232171	66
SearchResponse	3444	2323447	675	2844	1814754	638
Publish	141	842335	5974	8577	53478373	6235
Total	6589	5008592	760	28860	58059547	2011

Bezüglich des Anteils an dem induzierten Datenvolumen heben sich in beiden Simulationsläufen die Nachrichtentypen Publish und SearchResponse deutlich aufgrund ihrer mittleren Nachrichtengröße und Anzahl gegenüber den anderen Nachrichten ab. Für die Publish-Nachrichten liegt das einerseits an der initialen Verteilung von Artikeln, andererseits auch daran, dass hierbei der komplette Artikel inklusive all seiner Metadaten übertragen wird. Ein ähnliches Verhalten zeigen auch die durchschnittlichen Größen der Nachrichtentypen GetEntryResponse und GetRevisionResponse: Hier werden ebenfalls Artikel mit all ihren zusätzlichen Attributen übertragen. Für SearchResponse begründet sich der hohe Anteil am Datenvolumen dadurch, dass für jeden Treffer die textuelle Umgebung des Suchwortes aus dem Artikel extrahiert und in HTML eingebettet wird. Gegenüber einer simplen Trefferanzeige erhöhen diese zusätzlichen Layout- und Strukturierungsdaten den Umfang der SearchResponse-Nachrichten signifikant. Die Häufigkeit einzelner Nachrichtentypen lässt sich durch die Arbeitsweise der Aggregatoren erklären. So wird beispielsweise eine FindRevisions-Nachricht zum Abrufen der Versionshistorie, aber auch für das explizite Anfragen nach einer älteren Version eines Artikels genutzt. Die stark erhöhte Anzahl gesende-

ter Antworten auf eine BubbleCast-Nachricht lässt sich dadurch erklären, dass ein gesendeter BubbleCast von mehreren Peers empfangen und bei erfolgreicher Auswertung auch beantwortet wird. Tatsächlich korrespondiert der Quotient $\frac{Anzahl_{empfangen}}{Anzahl_{gesendet}}$ bei BubbleCast-Nachrichten zu der jeweiligen Bubble-Größe bei den durchgeführten Simulationen, wie die nachfolgende Tabelle verdeutlicht. Die exakte Berechnung der Bubble-Größe bezieht sich hierbei auf die Angaben in [16].

Verhältnis gesendeter und empfangener BubbleCasts

Nachrichtentyp	Simulation mit 50 Knoten	Simulation mit 300 Knoten
FindEntry	21,72	59,68
FindRevisions	20,56	61,19
Search	20,78	59,44
Publish	23,38	60,83
Exakt	25,98	63,64

Die nachfolgende Tabelle zeigt in Abhängigkeit des Nachrichtentyps, wieviele Antwortnachrichten auf einen BubbleCast im Durchschnitt gesendet werden. Der Nachrichtentyp Publish wird an dieser Stelle nicht betrachtet, da eine Antwort in diesem Fall nicht erfolgt. Die Durchschnittswerte wurden auf die nächstkleinere Ganzzahl gerundet.

Durchschnittliche Anzahl von Antworten auf einen BubbleCast

Nachrichtentyp	Simulation mit 50 Knoten	Simulation mit 300 Knoten
FindEntry	8 (532/61)	12 (742/62)
FindRevisions	8 (1048/129)	12 (1573/124)
Search	18 (1077/59)	48 (2844/59)

Auffallend ist, dass im Mittel wesentlich mehr Search-Anfragen beantwortet werden können. Das liegt daran, dass eine Anfrage per FindEntry respektive FindRevisions auf einem eindeutigen Artikeltitel beruht, während das Suchwort bei einer Search-Anfrage potenziell in verschiedenen Artikeln auftreten kann. In diesem Kontext muss gesondert erwähnt werden, dass die Artikelbasis für die durchgeführten Simulationen durch einen Dump der englischen Wiktionary gestellt wurde. Die Artikel befassen sich mit Wörtern aus der englischen Sprache nebst Erklärung und Querverweisen, so dass ein Vorkommen einzelner Artikeltitel, womöglich in einem beschreibenden Kontext innerhalb eines anderen Artikels, sehr wahrscheinlich ist.

Während der Simulation wurden die folgenden Latenzzeiten gemessen. Diese Zeiten sind selbstverständlich spezifisch für eine Operation, da jeder zu einer Operation korrespondierende Aggregator eine unterschiedliche Anzahl von Nachrichtentypen nutzt und gegebenenfalls mehrere Male auf Antworten warten muss. Die anschließende Tabelle gibt darüberhinaus Auskunft über die Erfolgsrate einzelner Operationen.

Latenzzeiten und Erfolgsrate einzelner Operationen

Operation	Latenz in ms (Mittelwert)	Latenz in ms (Standard-abweichung)	Erfolgsrate
50 Knoten			
GetArticle	3336	2882	93,44% (57/4)
FetchHistory	2395	3314	89,55% (60/7)
FetchFormerVersion	3345	3740	88,24% (75/10)
KeywordSearch	4429	2353	88,14% (52/7)
PublishArticle	328	440	100% (157/0)
Erfolgsrate (Total)			93,47% (401/28)
300 Knoten			
GetArticle	3820	3480	87,10% (54/8)
FetchHistory	1214	1025	100% (55/0)
FetchFormerVersion	3347	2756	92,31% (72/6)
KeywordSearch	4361	2490	83,05% (49/10)
PublishArticle	332	986	100% (141/0)
Erfolgsrate (Total)			93,92% (371/24)

Die wenigen Fehlerfälle resultieren vermutlich nicht aus dem probabilistischen Charakter von BubbleStorm, sondern aus der Tatsache, dass der BubbleStorm Prototyp während der Simulation in manchen Fällen den Aufbau einer Benutzerverbindung mit einer `BadHandshakeException` quittierte. In diesen Fällen wurden daher keine Daten an die Aggregatoren geliefert, weshalb diese folgerichtig eine fehlerhaft durchgeführte Operation meldeten. Die hohe Varianz der einzelnen Operationen lässt sich ebenfalls dadurch begründen. Die Aggregatoren melden den Erfolg, wenn vor Ablauf eines Timeouts ein Ergebnis vorliegt. Folglich wird bei allen fehlgeschlagenen Operationen die Zeit bis zum Timeout gewartet, was die Standardabweichung natürlich hinzu höheren Werten manipuliert. Abbildung 9.1 zeigt weitere Diagramme, die den gesamten Netzwerkverkehr für empfangene und gesendete Nachrichten sowie die bereits erwähnten Erfolgsraten und die Verteilung der durchgeführten Operationen pro beteiligtem Netzwerkknoten für die durchgeführten Simulationsszenarien beschreiben.

9.4 Lokal-dynamisches Szenario

Simuliert wurden drei verschiedene Konfigurationen, die sich lediglich bezüglich der Churn-Rate voneinander unterscheiden. Die restlichen Umgebungsparameter (Knotenzahl usw.) sind gegenüber dem lokal-statischen Szenario mit 300 aktiven Netzwerkknoten identisch, so dass ein direkter Vergleich mit einer Konfiguration ohne Churn möglich ist. Churn wurde in den durchgeführten Simulationen so festgelegt, dass nach jeweils t Sekunden ein Peer das Netzwerk verlässt, während ein neuer Peer dem Netzwerk beitritt. Die nachfolgende Tabelle schlüsselt den induzierten Netzwerkverkehr gruppiert nach Churn-Abstand in Sekunden auf.

Netzwerkverkehr nach Churn-Rate für Simulation mit 300 Knoten

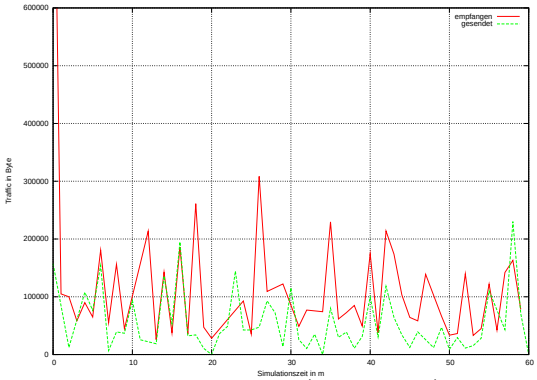
Nachrichtentyp	Churn-Abstand 90s		Churn-Abstand 60s		Churn-Abstand 30s	
	Anzahl	Größe (Byte)	Anzahl	Größe (Byte)	Anzahl	Größe (Byte)
FindEntry	3667	237791	3068	201678	2999	192344
FindEntryResponse	1503	356697	1473	352296	1256	296175
GetEntry	163	14857	139	12879	136	12424
GetEntryResponse	162	940571	136	855623	133	712956
FindRevisions	7782	551023	8515	587435	6334	448832
FindRevisionsResponse	3459	1189645	3615	1251240	2503	876111
GetRevision	181	19323	203	21484	163	17893
GetRevisionResponse	180	914569	203	1113261	160	1154511
Search	3787	253871	3245	220003	3940	259531
SearchResponse	6772	3897878	5442	4216663	6687	4256394
Publish	7149	48622685	6495	44453566	6803	46850535
Total	34805	569989910	32534	53286128	31114	55077706
Durchschnitt	1283 Byte/Nachricht		1328 Byte/Nachricht		1408 Byte/Nachricht	

Die anschließende Tabelle gibt darüberhinaus Auskunft über die Erfolgsrate einzelner Operationen sowie deren Latenzzeiten.

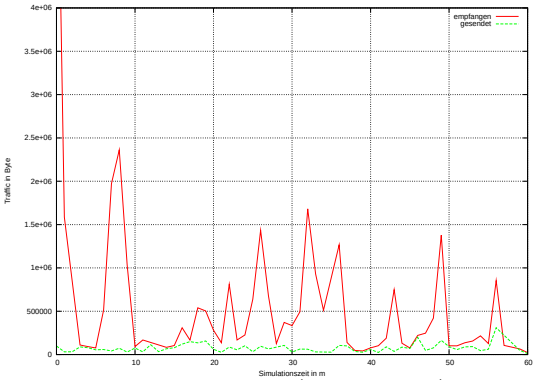
Latenzzeiten und Erfolgsrate einzelner Operationen

Operation	Latenz in ms (Mittelwert)	Latenz in ms (Standard-abweichung)	Erfolgsrate
Churn-Abstand 90s			
GetArticle	3305	2748	90,16% (55/6)
FetchHistory	1949	2784	89,83% (53/6)
FetchFormerVersion	2968	2545	94,94% (75/4)
KeywordSearch	4568	2236	75,81% (47/15)
PublishArticle	123	244	100% (112/0)
Erfolgsrate (Total)			91,69% (342/31)
Churn-Abstand 60s			
GetArticle	3063	2797	91,67% (44/4)
FetchHistory	2001	2777	89,86% (62/7)
FetchFormerVersion	2970	2574	95,24% (80/4)
KeywordSearch	4799	2635	78% (39/11)
PublishArticle	126	314	100% (107/0)
Erfolgsrate (Total)			92,74% (332/26)
Churn-Abstand 30s			
GetArticle	3771	3086	85,42% (41/7)
FetchHistory	1326	1687	100% (48/0)
FetchFormerVersion	3494	3249	91,04% (61/6)
KeywordSearch	4904	2512	78,57% (55/15)
PublishArticle	155	324	100% (105/0)
Erfolgsrate (Total)			91,72% (310/28)

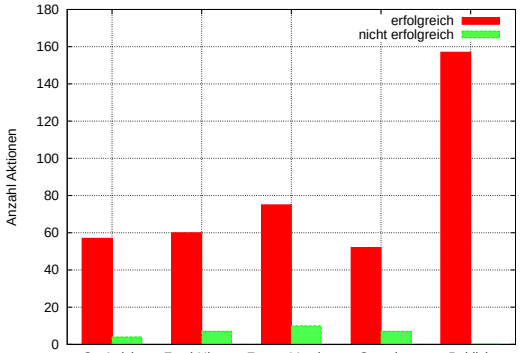
Bei geringerem Churn-Abstand scheint sich die Anzahl der durchgeführten Operationen zu verringern, was sich ebenfalls durch eine geringere Anzahl gesendeter und empfangener Nachrichten bemerkbar macht. Die Gründe sind allerdings eher bei zufallsbasierten Fluktuationen zu suchen, als bei der Leistungsfähigkeit des Systems. In diesem Kontext ist die Zunahme der durchschnittlichen Nachrichtengröße bei geringerem Churn-Abstand ebenfalls auffallend. Dies lässt sich jedoch auf die niedrigere Gesamtzahl transferierter Nachrichten in Kombination mit einer erhöhten, zufallsbedingten Anzahl von durchgeführten PublishArticle-Operationen zurückführen, welche einen großen Anteil an dem erzeugten Datenvolumen haben. Ein direkter Zusammenhang zwischen Churn-Rate und dem Erfolg von Operationen konnte aufgrund der Probleme, die bereits in Abschnitt 9.3 geschildert wurden, in den durchgeführten Szenarien nur schwer festgestellt werden. Die Anzahl fehlgeschlagener Operationen, die bedingt durch geworfene Ausnahmen im Kern des BubbleStorm Prototypen verursacht wurden, lassen sich nur schwer gegenüber den Auswirkungen des Churns abgrenzen. Jedoch kann festgehalten werden, dass die Erfolgsrate in lokal-statischen wie auch lokal-dynamischen Szenarien ungefähr gleich bleibt, so dass durch Churn bedingte fehlgeschlagene Operationen das Gesamtergebnis in den hier betrachteten Konfigurationen nur marginal zu beeinträchtigen scheint. Bei hinreichend langer Ausführungszeit der Simulation - und damit entsprechend vielen Churn-Aktivitäten - ist anzunehmen, dass sich die Erfolgsrate spürbar senkt, da die Wiki-Applikation keine Anbindung an einen Replikationsmechanismus besitzt, der für eine aktive Stabilisierung der Anzahl von Artikeln sorgen könnte.



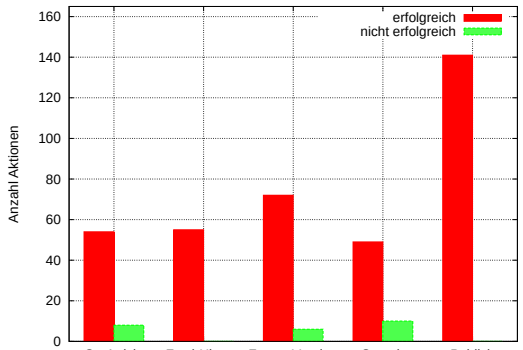
Netzwerkverkehr (50 Knoten)



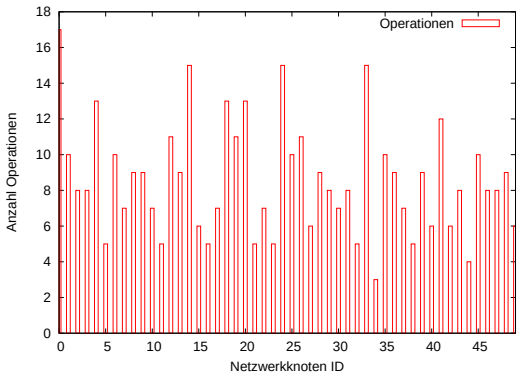
Netzwerkverkehr (300 Knoten)



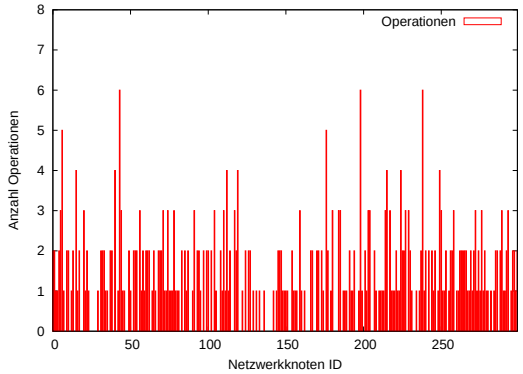
Erfolgsrate nach Operation (50 Knoten)



Erfolgsrate nach Operation (300 Knoten)



Verteilung Operationen/Benutzer (50 Knoten)



Verteilung Operationen/Benutzer (300 Knoten)

Abbildung 9.1: Metriken für die lokal-statischen Simulationsszenarien

Kapitel 10

Diskussion

> Die Schöpfung ist niemals vollendet.
> Sie hat zwar einmal angefangen, aber
> sie wird niemals aufhören.
>
> *Immanuel Kant*

Zunächst ist eine erfreuliche Erkenntnis festzustellen: Die Evaluation zeigt, dass das im Zuge dieser Arbeit entwickelte Peer-to-Peer Wiki korrekt arbeitet. Die Messergebnisse deuten jedoch auch darauf hin, dass durchaus noch Potenzial besteht, den BubbleStorm Prototypen zu verbessern. Die stellenweise noch auftretenden Ausnahmen verhindern leider, dass die Software eine 100%ige Erfolgsrate in lokal-statischen Umgebunden erzielt. Die Unterstützung für mediale Inhalte wurde rudimentär umgesetzt, so dass Grafiken in Artikel eingebettet und im Netzwerk verteilt werden können. Dieses Feature ist jedoch mit Vorsicht zu genießen: Grafiken tendieren im Vergleich zu Textdateien dazu, wesentlich mehr Speicherplatz zu benötigen. Die höhere Anforderung an das dadurch induzierte Datenvolumen wirkt sich selbstverständlich auch auf die Netzlast in einem BubbleStorm-Netzwerk aus. Da BubbleStorm nicht dafür optimiert ist, umfangreiche Datenvolumen zu verteilen, ist von der Nutzung großer Bilddateien abzusehen.

Die Versionierung wurde komplett umgesetzt und arbeitet von Seiten der Applikation fehlerfrei. Jedoch muss darauf hingewiesen werden, dass die verwendete Bibliothek jPatchLib in sehr seltenen Fällen keine korrekten Resultate erzielt. Dies wurde leider erst in einem sehr weit fortgeschrittenen Stadium der Arbeit festgestellt, so dass die Nutzung einer anderen Bibliothek nicht mehr umgesetzt werden konnte.

Konfligierende Versionsstände könnten durch eine Erweiterung des BubbleStorm Prototypen um einen publish/subscribe-Ansatz präemptiv verhindert werden. Die grundlegende Idee soll an dieser Stelle grob skizziert werden. Ein Peer A, der eine Änderung an einem bestimmten Artikel vornimmt, sendet zuvor eine Nachricht per BubbleCast über das Netzwerk und registriert sich somit bei allen Peers, die diese Nachricht empfangen. Die Registrierung zeigt hierbei an, dass der entsprechende Artikel gerade geändert wird. Sofern einer dieser Peers, bei denen sich Peer A registriert hat, eine Publish-Nachricht empfängt, die eine neue Version des sich in Bearbeitung befindlichen Artikels beinhaltet, besteht die Möglichkeit, Peer A über die neue Version zu informieren. Peer A erhält dadurch Kenntnis über zwischenzeitliche Updates und kann diese direkt in die aktuelle Änderung einbeziehen, was konfligierende Versionsstände grundsätzlich verhindert. Dieses Verfahren sollte sich über die Implementierung eines neuen BubbleCast-Nachrichtentyps nebst zugehöriger Behandlungsroutine bereits für den aktuellen BubbleStorm Prototypen umsetzen lassen. Die Umsetzung liegt jedoch außerhalb des Fokus dieser Bachelorarbeit und soll daher als Idee verstanden werden, die Thema künftiger Arbeiten sein könnte.

Unabhängig von einer präemptiven Behandlung von Versionskonflikten, könnte eine grundlegende Versionierungsstrategie von der Applikation entkoppelt werden. Grundlegende Ansätze und Ideen liefern die Arbeiten, die in Kapitel 3 diskutiert worden sind und Update-Strategien mit dem Replikationsmanagement koppeln.

Die Integration der Volltextsuche zeigt die bereits im einleitenden Teil dieser Arbeit angepriesenen Stärken des BubbleStorm-Systems. An dieser Stelle wäre eine Erweiterung des Wikis hinzu der Unterstützung von semantischen Abfragen denkbar. Dadurch könnten Artikeldaten über mehrere Artikel hinweg über logische Beziehungen miteinander verknüpft werden, was umfangreiche Abfragemöglichkeiten erlauben würde. Derartige Bestrebungen gibt es bereits bei anderen Wiki-Systemen, wie beispielsweise der Erweiterung *Semantic MediaWiki*[11] für die MediaWiki-Software der WikiMedia Foundation Inc.

Die Programmierschnittstelle des BubbleStorm Prototypen ist relativ einfach zu verwenden und benötigt keine Vorkenntnisse hinsichtlich Netzwerk-spezifischer Behandlung wie z. B. das Lesen von Sockets. Dies gilt jedoch nur für die Nutzung der grundlegenden Netzwerkfunktionen des BubbleStorm-Systems wie beispielsweise dem Versenden von BubbleCast-Nachrichten. Um das Konzept der Benutzerverbindung innerhalb einer eigenen Anwendung nutzen zu können, ist eine zusätzliche Indirektionsebene nötig, deren Realisierung Kenntnisse im Umgang mit Sockets voraussetzt. Eine derartige Realisierung kann in ihren Grundzügen jedoch generisch erfolgen und somit von der Applikation abstrahiert werden, so dass sich - wie bei der Behandlung von BubbleCasts - nur Unterschiede hinsichtlich der den Nachrichtentypen zugehörigenen Handler ergeben. Im Zuge weiterer Arbeiten an dem BubbleStorm Prototypen könnte daher die Nutzung der Benutzerverbindungen dahingehend simplifiziert werden, dass diese Handler lediglich an dem Routing-Mechanismus von BubbleStorm registriert werden müssen.

Die Nutzung des in Kapitel 4 beschriebenen BubbleStorm-eigenen Replikationsmechanismus konnte im Zuge dieser Arbeit leider nicht erfolgen, da die Implementierungsarbeiten an dem Replikationsalgorithmus in etwa mit dem Ende dieser Arbeit geschlossen wurden.

Anhang A

Formatbeschreibungen

Speicherformate

```
<?xml version="1.0" ?>
<!DOCTYPE WikiEntry [
    <!ELEMENT WikiEntry (BaseVersion, Version, Author, Type, Time, Checksum, Data)>
    <!ATTLIST WikiEntry Title CDATA #REQUIRED>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Version (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT Type (#PCDATA)>
    <!ELEMENT Time (#PCDATA)>
    <!ELEMENT Checksum (#PCDATA)>
    <!ELEMENT Data (#PCDATA)>
]>
```

Listing A.1: Speicherformat für einen Artikel in Volldarstellung (DTD)

```
<?xml version="1.0" ?>
<!DOCTYPE VersionInfo [
    <!ELEMENT VersionInfo (Title, BaseVersion, Version, Author, Type, Time, Checksum,
        Diff)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Version (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT Type (#PCDATA)>
    <!ELEMENT Time (#PCDATA)>
    <!ELEMENT Checksum (#PCDATA)>
    <!ELEMENT Diff (#PCDATA)>
]>
```

Listing A.2: Speicherformat für einen Artikel in Delta-Kodierung (DTD)

Protokolleinheiten

```
<?xml version="1.0" ?>
<!DOCTYPE FindEntry [
    <!ELEMENT FindEntry EMPTY>
```

```
    <!ATTLIST FindEntry MessageID CDATA #REQUIRED
                        Title CDATA #REQUIRED>
]>
```

Listing A.3: Formatbeschreibung `FindEntry`

```
<?xml version="1.0" ?>
<!DOCTYPE FindEntryResponse [
    <!ELEMENT FindEntryResponse (BaseVersion, Version, Checksum)>
    <!ATTLIST FindEntryResponse Address CDATA #REQUIRED
                                Port CDATA #REQUIRED
                                MessageID CDATA #REQUIRED
                                Title CDATA #REQUIRED>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Version (#PCDATA)>
    <!ELEMENT Checksum (#PCDATA)>
]>
```

Listing A.4: Formatbeschreibung `FindEntryResponse`

```
<?xml version="1.0" ?>
<!DOCTYPE GetEntry [
    <!ELEMENT GetEntry EMPTY>
    <!ATTLIST GetEntry Address CDATA #REQUIRED
                       Port CDATA #REQUIRED
                       MessageID CDATA #REQUIRED
                       Title CDATA #REQUIRED>
]>
```

Listing A.5: Formatbeschreibung `GetEntry`

```
<?xml version="1.0" ?>
<!DOCTYPE GetEntryResponse [
    <!ELEMENT GetEntryResponse (BaseVersion, Version, Author, EntryType, Date, Checksum,
        Data)>
    <!ATTLIST GetEntryResponse Address CDATA #REQUIRED
                               Port CDATA #REQUIRED
                               MessageID CDATA #REQUIRED
                               Title CDATA #REQUIRED>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Version (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT EntryType (#PCDATA)>
    <!ELEMENT Date (#PCDATA)>
    <!ELEMENT Checksum (#PCDATA)>
    <!ELEMENT Data (#PCDATA)>
]>
```

Listing A.6: Formatbeschreibung `GetEntryResponse`

```
<?xml version="1.0" ?>
<!DOCTYPE Search [
    <!ELEMENT Search EMPTY>
    <!ATTLIST Search MessageID CDATA #REQUIRED
```

```
                    Keywords CDATA #REQUIRED>
]>
```

Listing A.7: Formatbeschreibung `Search`

```
<?xml version="1.0" ?>
<!DOCTYPE SearchResponse [
    <!ELEMENT SearchResponse ((Item)+)>
    <!ATTLIST SearchResponse Address CDATA #REQUIRED
                             Port CDATA #REQUIRED
                             MessageID CDATA #REQUIRED>
    <!ELEMENT Item (Title, Author, Relevancy, Surrounding)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT Relevancy (#PCDATA)>
    <!ELEMENT Surrounding (#PCDATA)>
]>
```

Listing A.8: Formatbeschreibung `SearchResponse`

```
<?xml version="1.0" ?>
<!DOCTYPE FindRevisions [
    <!ELEMENT FindRevisions EMPTY>
    <!ATTLIST FindRevisions MessageID CDATA #REQUIRED
                            Title CDATA #REQUIRED>
]>
```

Listing A.9: Formatbeschreibung `FindRevisions`

```
<?xml version="1.0" ?>
<!DOCTYPE FindRevisionsResponse [
    <!ELEMENT FindRevisionsResponse ((Item)+)>
    <!ATTLIST FindRevisionsResponse Address CDATA #REQUIRED
                                    Port CDATA #REQUIRED
                                    MessageID CDATA #REQUIRED
                                    Title CDATA #REQUIRED>
    <!ELEMENT Item (BaseVersion, Version, Author, Date, ContentType)>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Version (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT Date (#PCDATA)>
    <!ELEMENT ContentType (#PCDATA)>
]>
```

Listing A.10: Formatbeschreibung `FindRevisionsResponse`

```
<?xml version="1.0" ?>
<!DOCTYPE GetRevision [
    <!ELEMENT GetRevision EMPTY>
    <!ATTLIST GetRevision Address CDATA #REQUIRED
                          Port CDATA #REQUIRED
                          MessageID CDATA #REQUIRED
                          Title CDATA #REQUIRED
                          Version CDATA #REQUIRED>
```

```
]>
```

Listing A.11: Formatbeschreibung `GetRevision`

```
<?xml version="1.0" ?>
<!DOCTYPE GetRevisionResponse [
    <!ELEMENT GetRevisionResponse (BaseVersion, Author, ContentType, Date, Checksum, Data
        )>
    <!ATTLIST GetRevisionResponse Address CDATA #REQUIRED
                                  Port CDATA #REQUIRED
                                  MessageID CDATA #REQUIRED
                                  Title CDATA #REQUIRED
                                  Version CDATA #REQUIRED>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT ContentType (#PCDATA)>
    <!ELEMENT Date (#PCDATA)>
    <!ELEMENT Checksum (#PCDATA)>
    <!ELEMENT Data (#PCDATA)>
]>
```

Listing A.12: Formatbeschreibung `GetRevisionResponse`

```
<?xml version="1.0" ?>
<!DOCTYPE Publish [
    <!ELEMENT Publish (Title, BaseVersion, Version, Author, EntryType, ContentType, Date,
        Checksum, Data)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT BaseVersion (#PCDATA)>
    <!ELEMENT Version (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT EntryType (#PCDATA)>
    <!ELEMENT ContentType (#PCDATA)>
    <!ELEMENT Date (#PCDATA)>
    <!ELEMENT Checksum (#PCDATA)>
    <!ELEMENT Data (#PCDATA)>
]>
```

Listing A.13: Formatbeschreibung `Publish`

Anhang B

UML-Diagramme

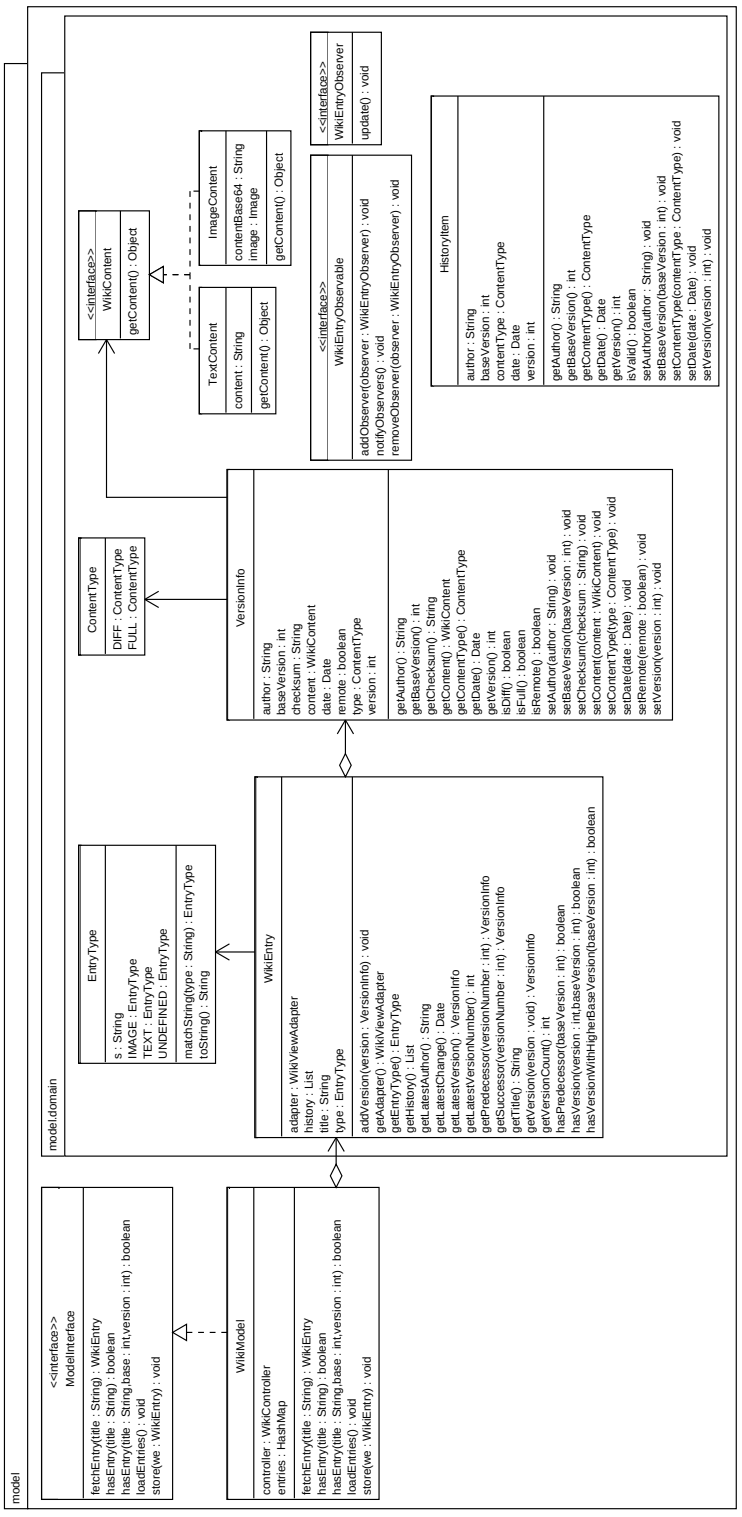


Abbildung B.1: Paketübersicht Modell (Detail)

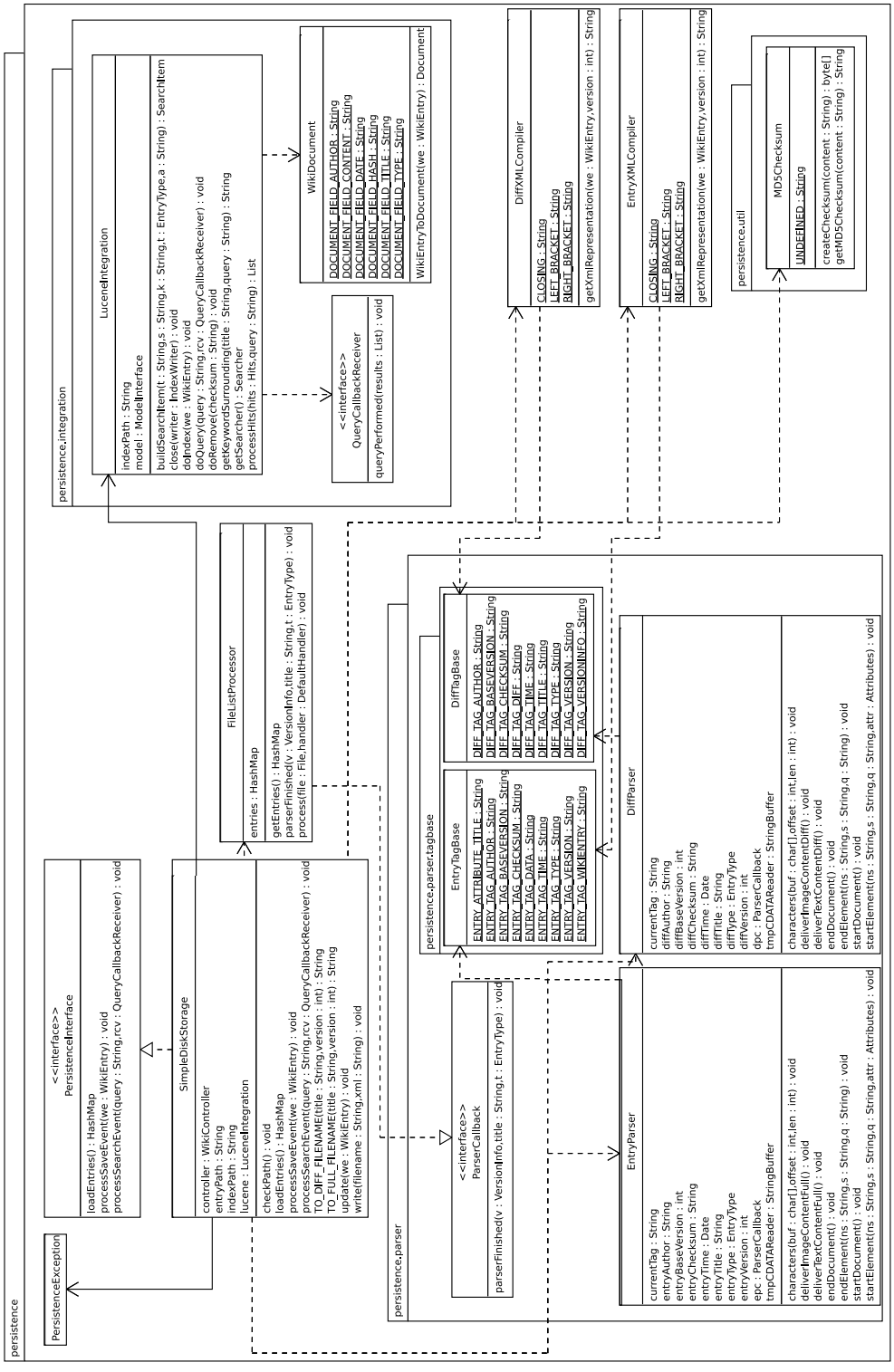


Abbildung B.2: Paketübersicht Persistenz (Detail)

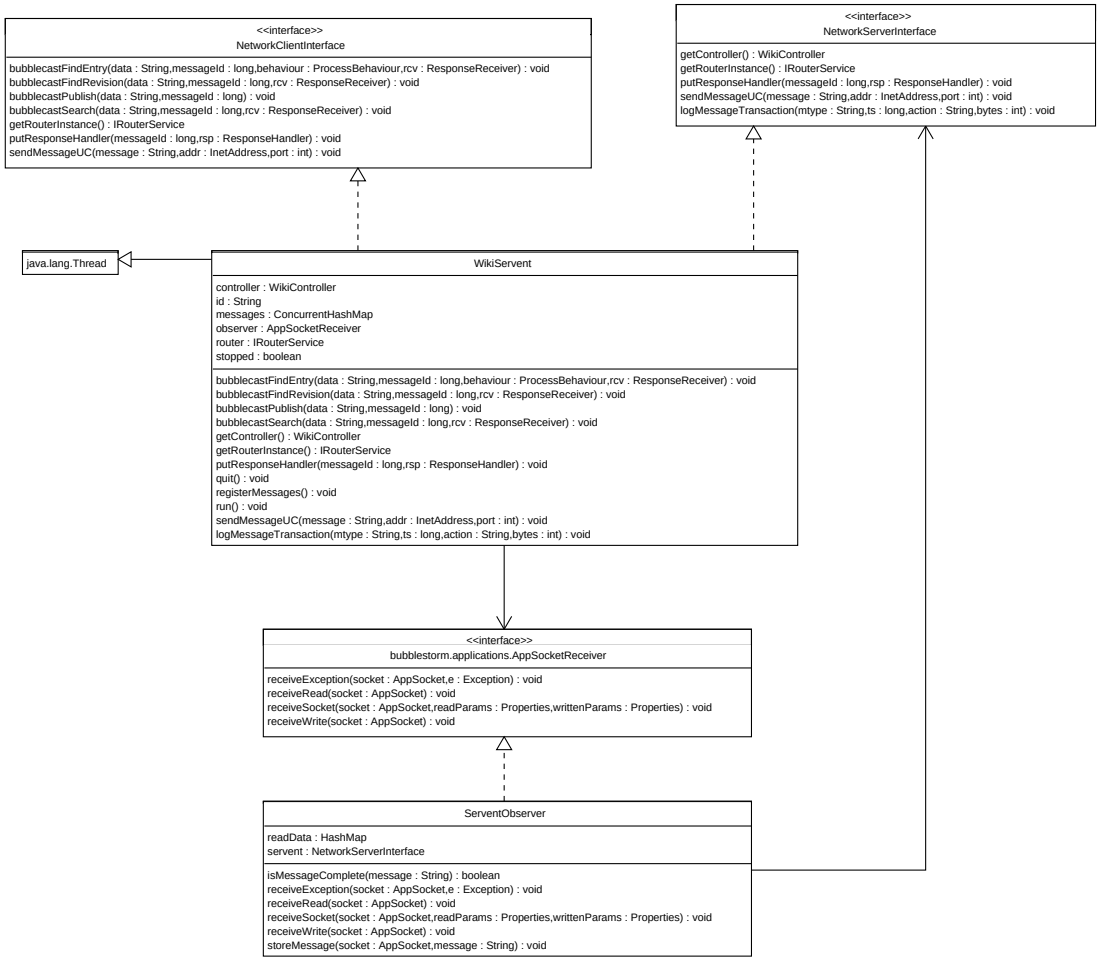


Abbildung B.3: Klassendiagramm zu `network`

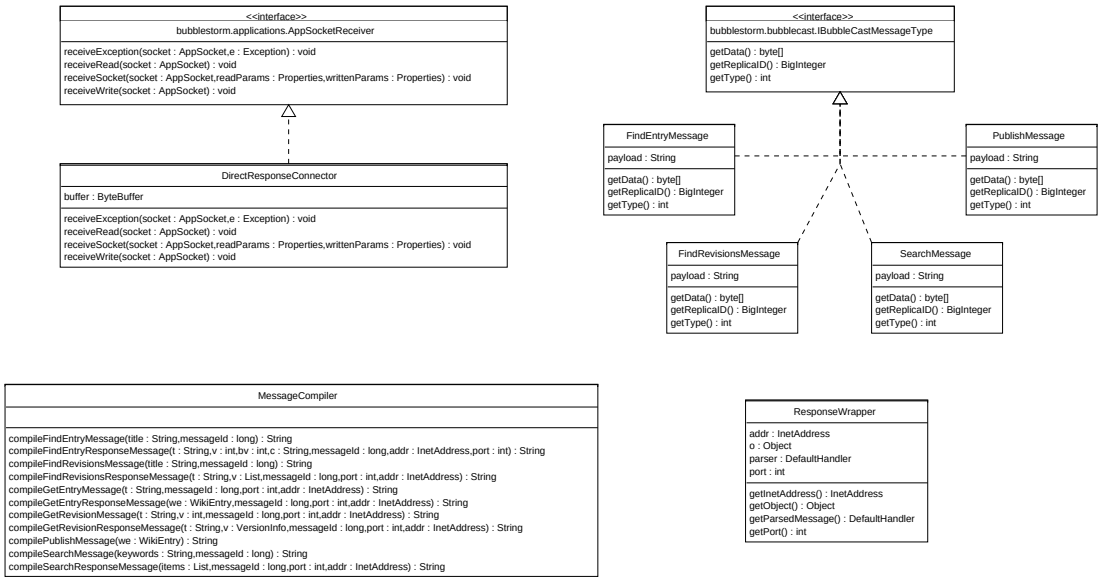


Abbildung B.4: Klassendiagramm zu `network.messages`

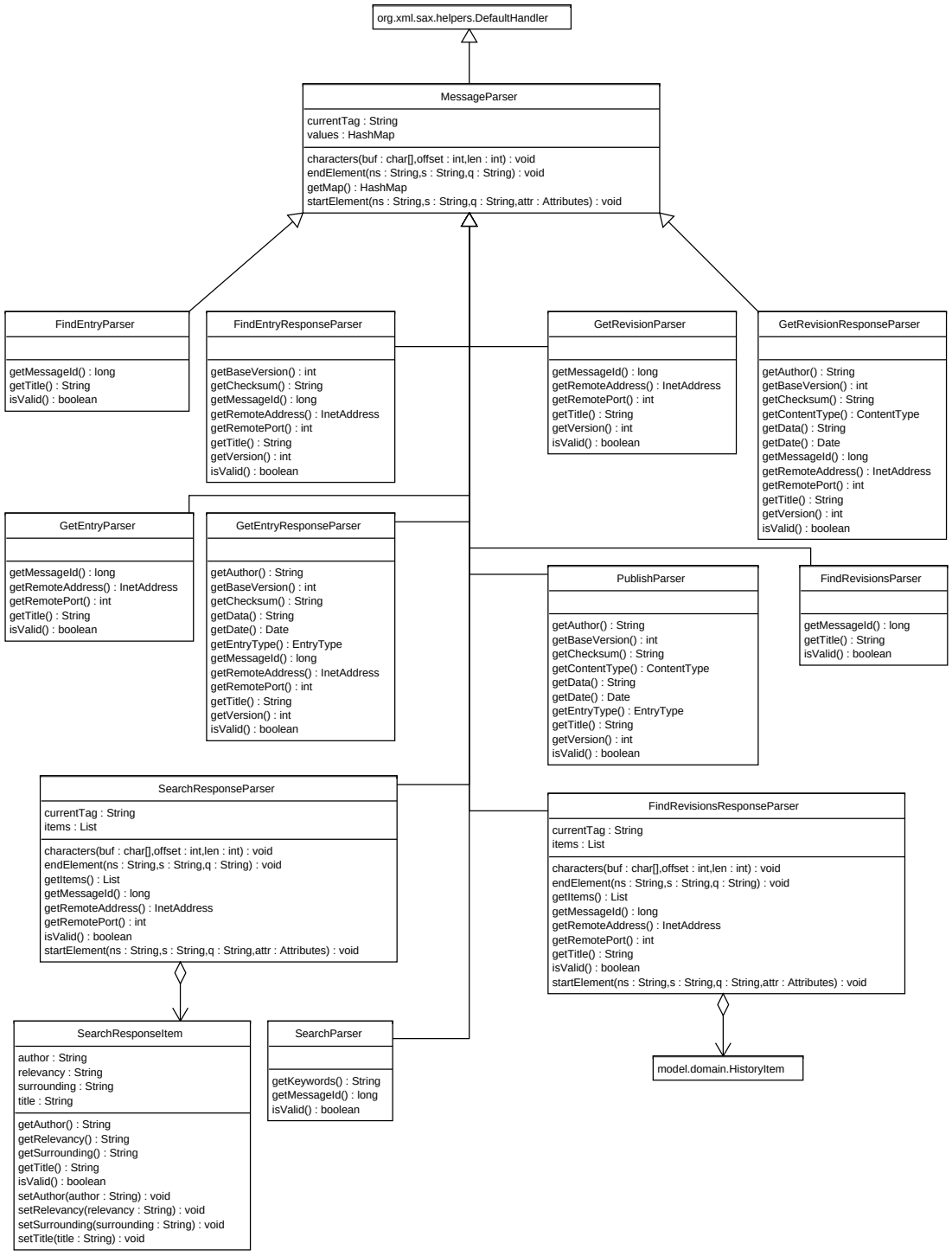


Abbildung B.5: Klassendiagramm zu `network.messages.parser`

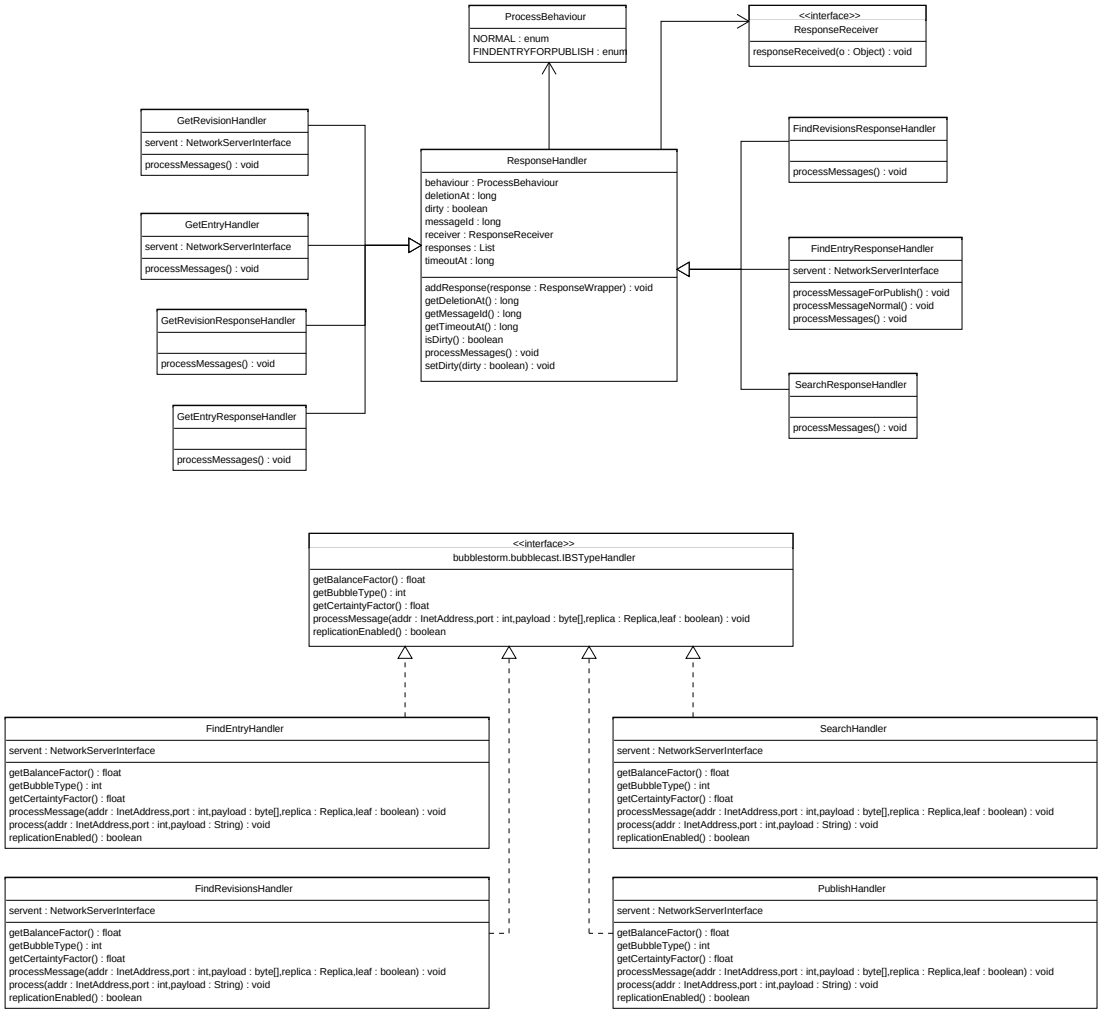


Abbildung B.6: Klassendiagramm zu `network.messages.handler`

Anhang C

Bildschirmausschnitte

Abbildung C.1: Werkzeugleiste

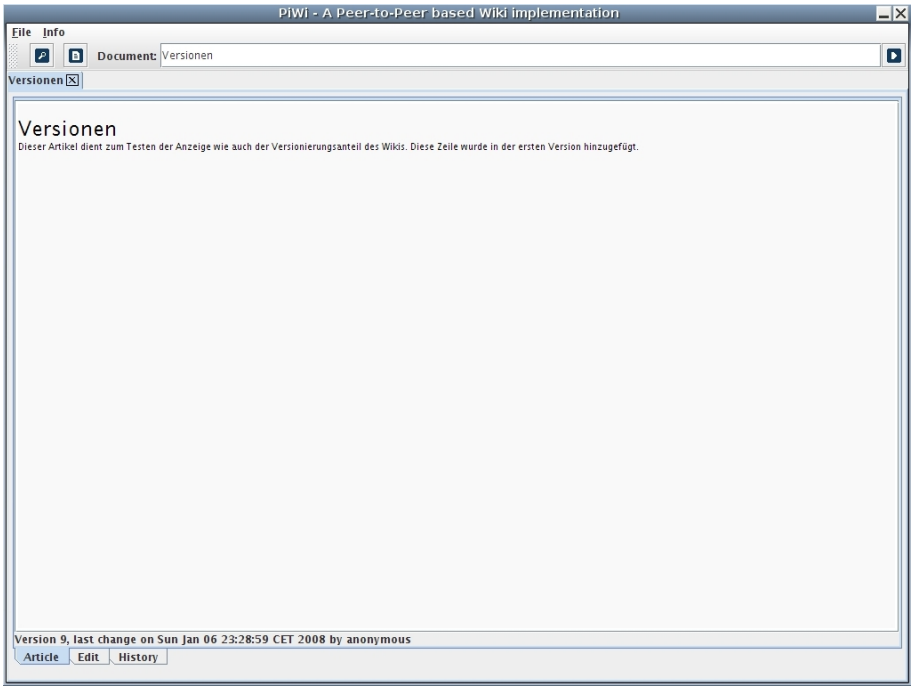


Abbildung C.2: Artikelansicht

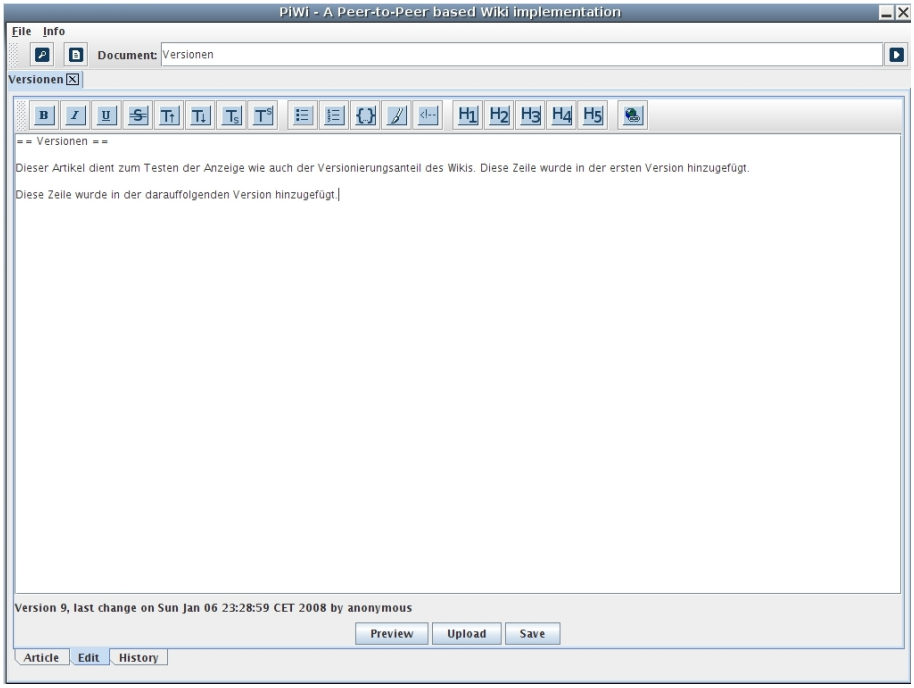


Abbildung C.3: Integrierter Editor

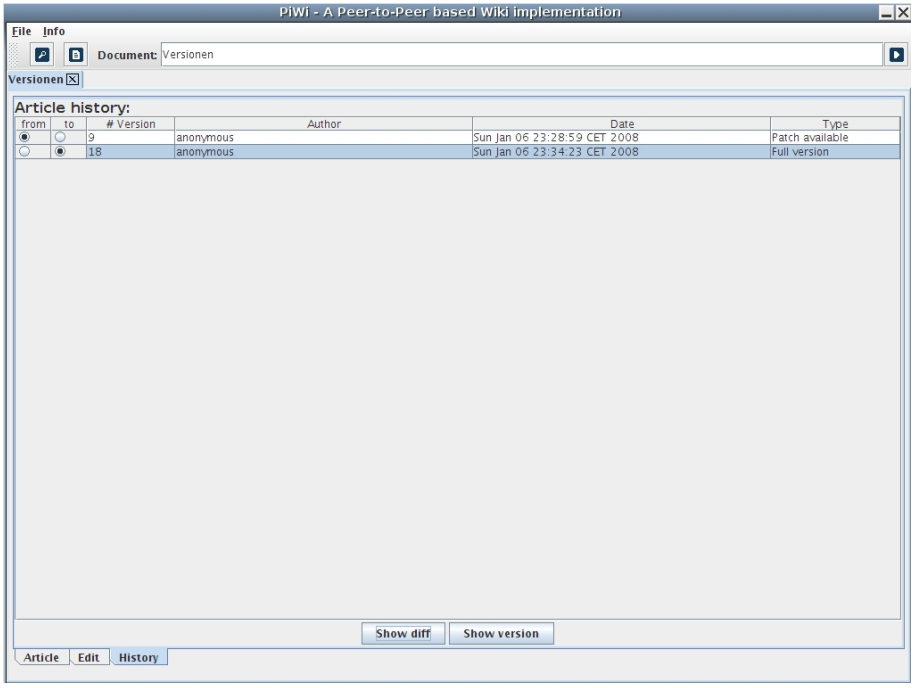


Abbildung C.4: Anzeigen der Versionshistorie

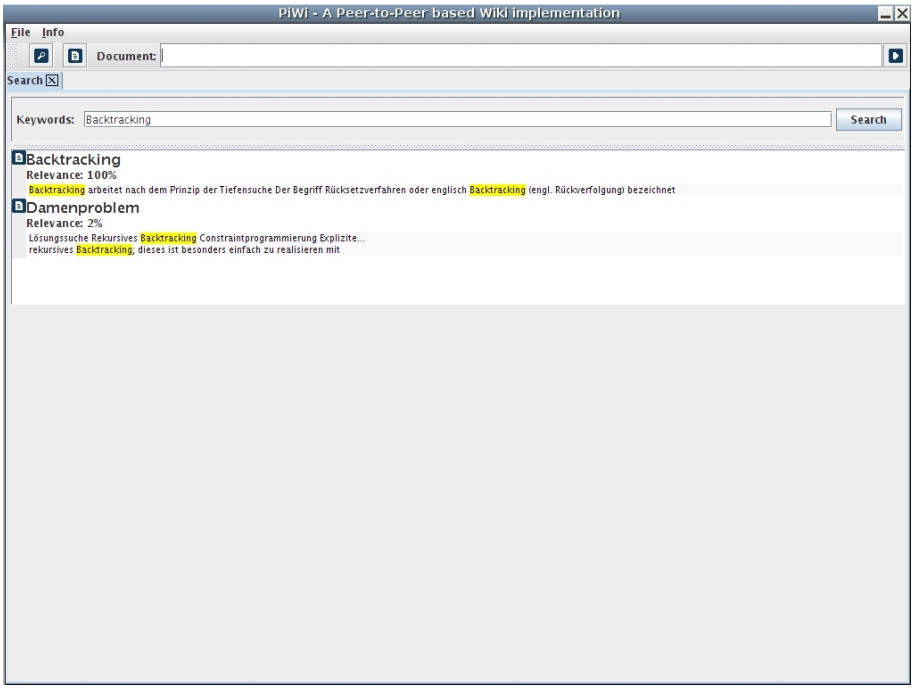


Abbildung C.5: Artikel suchen

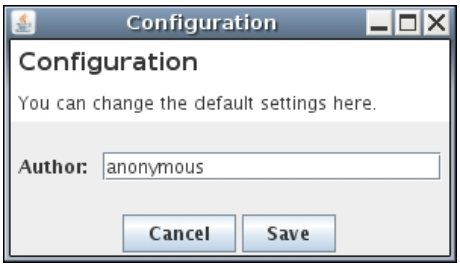


Abbildung C.6: Konfigurationsdialog

Abbildung C.7: Verbindung herstellen

Abbildung C.8: Vorschaufenster für editierten Text

Abbildung C.9: Hochladen von Bilddateien

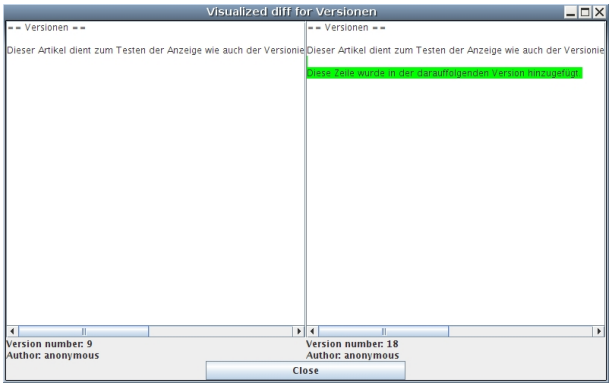


Abbildung C.10: Visueller Diff zwischen zwei Versionen

Literaturverzeichnis

[1] P. Baecher. Peer-to-Peer Replica Maintenance in BubbleStorm, 2008.

[2] R. Bahmani and D. Schulz. Peer-to-Peer Infrastructures (Lab Report from Caffee). Technical report, Databases and Distributed Systems, Department of Computer Science, Darmstadt University of Technology, 2007.

[3] M. Bar and K. Fogel. *Open Source Development with CVS*. Paraglyph Press, Inc., Scottsdale, Arizona, 3rd edition, 2003.

[4] Petr Baudis. Git - Fast Version Control System. http://git.or.cz, Abruf: 8. Oktober 2007.

[5] M. Breyer, A. Eigenstetter, and C. Wirth. Peer-to-Peer Infrastructures (Lab Report from Creative Wizards). Technical report, Databases and Distributed Systems, Department of Computer Science, Darmstadt University of Technology, 2007.

[6] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS2003*, 2003.

[7] C. Gross, T. Philipp, and M. Lehn. Peer-to-Peer Infrastructures (Lab Report from PCAD). Technical report, Databases and Distributed Systems, Department of Computer Science, Darmstadt University of Technology, 2007.

[8] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. 2001.

[9] J. Marinacci and C. Adamson. *Swing Hacks*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st edition, June 2005.

[10] Microsoft Visual SourceSafe. Glossar. http://msdn2.microsoft.com/de-de/library/0b11a65t(VS.80).aspx, Abruf: 28. Dezember 2007.

[11] Ontoworld.org. Semantic MediaWiki. http://ontoworld.org, Abruf: 6. Januar 2008.

[12] Derek Robert Price. CVS - Open Source Version Control. http://www.nongnu.org/cvs/, Abruf: 8. Oktober 2007.

[13] J. Ritter. Why Gnutella Can't Scale. No, Really. http://www.darkridge.com/ jpr5/doc/gnutella.html, Abruf: 20. Dezember 2007.

[14] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability. http://www.cs.cmu.edu/ kunwadee/research/p2p/gnutella.html, Abruf: 20. Dezember 2007.

[15] M. Swoboda. Implementation of a Prototype for the BubbleStorm Peer-to-Peer Network. Master's thesis, TU Darmstadt, 2008.

[16] W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann. BubbleStorm: resilient, probabilistic and exhaustive Peer-to-Peer search. In *Proceedings of the 2007 ACM SIGCOMM Conference*, pages 49–60. ACM Press, 2007.

[17] W. W. Terpstra, C. Leng, and A. P. Buchmann. BubbleStorm: analysis of probabilistic exhaustive search in a heterogeneous Peer-to-Peer system. Technical Report TUD-CS-2007-2, TU Darmstadt, May 2007.

[18] Z. Wang, S. K. Das, M. Kumar, and H. Shen. Update Propagation through Replica Chain in Decentralized and Unstructured P2P Systems.