Technische Universität Darmstadt
Fachbereich Informatik

# Peer-to-Peer Replikatverwaltung in BubbleStorm

Bachelorarbeit von
Paul Bächer

Betreuer: Dipl.-Inform. Christof Leng

Eingereicht bei: Prof. Alejandro P. Buchmann, Ph. D.
Fachgebiet Datenbanken und Verteilte Systeme
Fachbereich Informatik
TU Darmstadt
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, März 2008

Zusammenfassung

Zielgerichtete Replikation in Peer-to-Peer Netzwerken wurde in der Vergangenheit vorrangig für strukturierte Architekturen erforscht. Die dort gewonnenen Erkenntnisse lassen sich jedoch nicht ohne weiteres auf unstrukturierte Netzwerke wie BubbleStorm übertragen. Diese Arbeit stellt daher einen probabilistischen Replikationsalgorithmus für BubbleStorm vor, der vollständig dezentral und besitzerlos arbeitet. Im Vordergrund steht dabei die aktive Stabilisierung einer angestrebten Replikatanzahl unter ständig wechselnder Netzwerkzusammensetzung. Wie BubbleStorm selbst ist der Algorithmus hierbei in der Lage, sich die vorherrschende Leistungsheterogenität im Netzwerk zunutze zu machen. Eine Referenzimplementation im BubbleStorm-Prototyp stellt die Grundlage für die Evaluation dar und bereitet die praktische Verwendung des Algorithmus für konkrete Anwendungsszenarien vor.

# Inhaltsverzeichnis


1 Einleitung 1

2 Verwandte Arbeiten 3
- 2.1 Einordnung von Peer-to-Peer Systemen . . . . . 3
- 2.2 Replikation . . . . . 5
  - 2.2.1 Verteilung und Stabilisierung . . . . . 5
  - 2.2.2 Veränderliche und versionierte Replikate . . . . . 6
  - 2.2.3 Bisherige Ansätze in Peer-to-Peer Umgebungen . . . . . 7

3 BubbleStorm 9
- 3.1 Topologie . . . . . 10
  - 3.1.1 Netzwerkbeitritt und Austritt . . . . . 11
  - 3.1.2 Client-Modus . . . . . 11
- 3.2 Bubbles und BubbleCast . . . . . 11
- 3.3 Messung . . . . . 13

4 Ansatz 14
- 4.1 Anforderungen . . . . . 14
- 4.2 Modell . . . . . 15
  - 4.2.1 Replikate . . . . . 15
  - 4.2.2 Replikationsfaktor . . . . . 15
- 4.3 Algorithmus . . . . . 16
  - 4.3.1 Idee . . . . . 16
  - 4.3.2 Einfacher Ansatz . . . . . 18
  - 4.3.3 Erweiterte Version . . . . . 22
    - 4.3.3.1 Protokoll- und Algorithmusbeschreibung . . . . . 23

5 Implementierung 25
- 5.1 Design . . . . . 25

5.1.1 Datenrepräsentation . . . . . 26
5.1.2 Replikationsprotokoll . . . . . 27
5.1.2.1 Phase 1 – Zielsuche . . . . . 28
5.1.2.2 Phase 2 – Replikationsdialog . . . . . 29
5.1.3 Publikation und Replikation . . . . . 31
5.1.4 Konfiguration . . . . . 33
5.2 Anwenderschnittstelle . . . . . 33
5.2.1 Anwendungsbeispiel . . . . . 34
5.2.1.1 Publikation . . . . . 34
5.2.1.2 Replikation . . . . . 36

6 Evaluation 37
6.1 Evaluationsaufbau und -umgebung . . . . . 37
6.1.1 Metriken . . . . . 37
6.1.2 Szenarien . . . . . 38
6.2 Evaluationsergebnisse . . . . . 39
6.2.1 Birth/Death (Churn) . . . . . 39
6.2.2 Zunahme-Szenario . . . . . 41
6.2.3 Abnahme-Szenario . . . . . 42
6.2.4 Heterogenes Szenario . . . . . 45
6.2.5 Fazit . . . . . 46

7 Zusammenfassung und Ausblick 47
7.1 Kritik . . . . . 47
7.2 Ausblick . . . . . 48

A Formatbeschreibungen 49

# Abbildungsverzeichnis


3.1 Darstellungen eines BubbleStorm-Netzwerks. . . . . . . . . . . . . . 10
3.2 Balls-in-Bins Veranschaulichung . . . . . . . . . . . . . . . . . . . . 12

4.1 Abschwächung der Replikationswahrscheinlichkeit $\rho$. . . . . . . . . . 19
4.2 Darstellung des Replikationsalgorithmus. . . . . . . . . . . . . . . . 21

5.1 Architekturübersicht des Replikationssubsystems. . . . . . . . . . . 26
5.2 Zeitlicher Ablauf des Replikationsprotokolls. . . . . . . . . . . . . . 28
5.3 Protokollablauf in der zweiten Phase. . . . . . . . . . . . . . . . . . 30
5.4 Zustände der Kommunikationspartner. . . . . . . . . . . . . . . . . . 31
5.5 Zusammenhang zwischen Publikation und Replikation. . . . . . . . 32

6.1 Birth/Death-Szenario: Replikatanzahl und Netzwerkverkehr . . . 40
6.2 Birth/Death-Szenario: Dauer und Replikatverteilung . . . . . . . . 42
6.3 Zunahme-Szenario: Replikatanzahl und Netzwerkverkehr . . . . . 43
6.4 Abnahme-Szenario: Replikatanzahl und Netzwerkverkehr . . . . . 44
6.5 Heterogenes Szenario: Replikatanzahl . . . . . . . . . . . . . . . . . 45
6.6 Heterogenes Szenario: Netzwerkverkehr und Replikatverteilung . . 46

# Listings


5.1 Ein Bubble-Typ wird zur Replikation markiert. . . . . . . . . . . 35
5.2 Eine Nachricht mit gesetztem Replikationsidentifikator. . . . . . . 35
5.3 Versenden (Publikation) eines Replikats. . . . . . . . . . . . . . . 35
5.4 Starten des Replikationsmechanismus. . . . . . . . . . . . . . . . . . 36
5.5 Verarbeitung publizierter Replikate. . . . . . . . . . . . . . . . . . . 36

# Kapitel 1

# Einleitung

Seit einigen Jahren wird ein signifikanter Teil des Datenverkehrs im Internet durch dezentrale Peer-to-Peer Systeme induziert. Durch das Einsparen zentraler Ressourcen und durch Infrastrukturen, die mit der Anzahl der Benutzer skalieren, können sehr große Netzwerke kostengünstig aufgebaut und betrieben werden. Dabei taucht allerdings eine ganze Reihe von Problemen auf, die zwar zum Teil schon von lokal verteilten Systemen bekannt sind, jedoch für die Größenordnung und das topologische Ausmaß eines modernen Peer-to-Peer Netzwerks neu evaluiert werden müssen.

Eines dieser Probleme ist die Verfügbarkeit von Daten, die in einem solchen Netzwerk erstellt, verarbeitet oder einfach nur gespeichert werden sollen. Da Peers das Netzwerk zu beliebigen Zeitpunkten betreten und verlassen können, ist die Verfügbarkeit eines einzelnen Peers zunächst – relativ zu zentralen Systemen – gering. Das lässt sich dadurch ausgleichen, dass Kopien von Daten (Replikate) auf mehrere Peers verteilt werden und somit die Gesamtverfügbarkeit erhöht wird. Nach einer initialen Verteilung muss sichergestellt werden, dass ständige Änderungen an der Netzwerkzusammensetzung nicht zum Verlust solcher Replikate führen. In vielen Peer-to-Peer Systemen wie beispielsweise Tauschbörsen ist dieser Prozess relativ unkoordiniert; die Kopien sind meist proportional zur Beliebtheit verteilt, weil Peers die heruntergeladenen Daten kurzzeitig selbst bereitstellen. Für bestimmte Anwendungen ist eine derart naive Strategie jedoch nicht wünschenswert, da der Speicherplatz einzelner Knoten limitiert ist und die Gefahr des Verlorengehens für weniger beliebte Daten besteht. Zudem erfordern einige Anwendungen eine ganz spezifische Anzahl von Kopien, worauf sich darauf aufbauende Algorithmen und Konzepte verlassen müssen.

Das Ziel dieser Arbeit ist die Entwicklung eines Algorithmus zur Verwaltung von Replikaten und die Implementierung dessen in der BubbleStorm Peer-to-Peer Architektur. Zu einer gegebenen Netzwerkgröße existiert eine angestrebte Anzahl von Replikaten, die erreicht werden soll. Ausgehend von einer Initialverteilung besteht die Herausforderung darin, diese Anzahl unter Einfluss von Veränderungen an der Netzwerkgröße zu erreichen. Dabei steht aufgrund des Designs von BubbleStorm keine zentrale Instanz zur Verfügung und es gibt keinen designierten Besitzer von Daten. Der Algorithmus muss sich also vollständig dezentral um die aktive Stabilisierung der Replikatanzahl kümmern. Diese Aufgabe ist von der initialen Publikation von Replikaten zu unterscheiden, welche von BubbleStorm bzw. darauf aufbauenden Anwendungen gelöst wird.

Diese Arbeit ist wie folgt gegliedert: Kapitel 2 befasst sich mit dem aktuellen Stand der Forschung auf diesem und eng verwandten Gebieten sowie der grundlegenden Einordnung von Peer-to-Peer Systemen. In Kapitel 3 wird das BubbleStorm Peer-to-Peer System vorgestellt, welches die Grundlage für die Implementierung dieser Arbeit darstellt. Eine anschließende Betrachtung des vorgeschlagenen Algorithmus im Detail und dessen Implementierung in BubbleStorm erfolgt in Kapitel 4 und 5. Die Evaluierung dieser Implementierung wird in Kapitel 6 präsentiert, Kapitel 7 bietet eine Zusammenfassung und einen Ausblick auf zukünftige Weiterentwicklungen.

# Kapitel 2

# Verwandte Arbeiten

Sowohl Peer-to-Peer Systeme als auch Replikation sind für sich genommen gut erforschte Gebiete. In Abschnitt 2.2.3 werden einige Arbeiten diskutiert, die sich mit beiden Themen im Kontext beschäftigen, d.h. Replikation in Peer-to-Peer Netzen.

## 2.1 Einordnung von Peer-to-Peer Systemen

Auf höchster Ebene können Peer-to-Peer Systeme zunächst in hybride und echte Systeme unterteilt werden. Ist zur vollständigen Funktion eines Netzwerks eine zentrale Instanz notwendig, so spricht man von einem *hybriden* System. Diese Notwendigkeit kann sich z.B. daraus ergeben, dass ein zentraler Server einen Suchindex über alle Peers bereithält oder ein Loginserver zwecks Abrechnungszwecken erforderlich ist. Ein populäres Beispiel ist die ehemalige Napster-Tauschbörse, die für die Suchfunktion einen zentralen Server benötigte.

*Echte* Peer-to-Peer Systeme sind völlig unabhängig von einer zentralen Instanz und können unter Angabe eines bekannten Peers dem Netzwerk beitreten und alle bereitgestellten Funktionen nutzen. Solche Architekturen sind wiederum unterteilt in *strukturierte* und *unstrukturierte* Systeme.

In *strukturierten* Systemen wird das Layout des Netzwerks durch festgelegte Vorschriften konstruiert, d.h. Knoten sind nach bestimmten Regeln mit anderen Knoten verbunden. Das gilt auch für die Daten in solchen Netzwerken, in den meisten Fällen ist ein Knoten für eine deterministisch ausgewählte Menge an Daten ver-

antwortlich. Aufgrund dieser Struktur ist es einfach, ein bestimmtes Datum zu finden, falls ein entsprechender Identifikator gegeben ist. Die meisten Implementierungen nutzen hierfür eine Tabellensemantik, in der einem Schlüssel genau ein Datum zugeordnet wird. Durch deterministische Routing-Protokolle wird meist eine sehr kurze (qualitativ logarithmisch lange) Route zum gewünschten Datum gefunden. Als Beispiele seien hier CAN [8], Tapestry [14] und Chord [9] genannt. Als Nachteil ist die forcierte Abfragesemantik durch die vorgegebene Struktur zu sehen, sodass komplexere Abfragen wie Range Queries oder Stichwortsuchen zunächst auf das zugrunde liegende System abgebildet werden müssen. Dies erfordert häufig Indizes und damit weitere Indirektionsebenen, die sich negativ auf die Performanz des Systems auswirken. Dennoch hat sich die Forschung in jüngster Zeit vor allem auf strukturierte Systeme konzentriert, da eine solche Semantik für sehr viele Anwendungen ausreichend oder sogar ideal ist.

*Unstrukturierte* Systeme erlauben beliebige Verbindungen zwischen einzelnen Knoten. Durch dieses ungeordnete Layout des Netzwerkes ist es schwierig, ein bestimmtes Datum, von dem ein Identifikator bekannt ist, zu lokalisieren. Der in der Praxis verfolgte Ansatz ist hierbei in der Regel das Netzwerk zu fluten. Dieses Verfahren skaliert offensichtlich aufgrund der Menge des induzierten Datenverkehrs nicht gut. Jedoch ist es bei solchen Systemen von Vorteil, dass keine bestimmte Abfragesemantik existiert. Das Verarbeiten einer Abfrage kann bei einzelnen Knoten fast beliebig komplex ausfallen und operiert theoretisch auf der kompletten Datenmenge im Netzwerk.

Gelegentlich werden neben echten und hybriden Architekturen noch Supernode-Systeme erwähnt. Dabei gibt es zwei Klassen von Peers, leistungsfähige und gut angebundene Supernodes sowie reguläre Peers. Supernodes übernehmen stets mehr Verantwortung und sind untereinander gut verbunden, während reguläre Peers nur zu genau einer Supernode konnektiert sind. Im Rahmen dieser Arbeit werden solche Systeme den echten Peer-to-Peer Systemen untergeordnet, da eine bestimmte Supernode nicht als Single Point of Failure für das gesamte Netzwerk zu verstehen ist. Des Weiteren kann ein Supernode-System ebenfalls in strukturiert und unstruktuiert unterteilt werden, wodurch sich die Sicht als Spezialfall eines echten Peer-to-Peer Systems anbietet.

## 2.2 Replikation

Überall dort, wo Daten gespeichert werden, besteht die Gefahr eines Verlustes. Angefangen bei lokalen Systemen über Cluster und lokale Netzwerke bis hin zu massiv verteilten Architekturen wie Peer-to-Peer Netzen behilft man sich hierbei durch das Anlegen von redundanten Kopien, also Replikaten. Die Ausfallwahrscheinlichkeiten multiplizieren sich, wodurch sich die Gesamtverfügbarkeit erhöht, da nur eine Kopie tatsächlich notwendig ist. Während dies bei lokalen Systemen beispielsweise durch mehrere Datenträger erreicht werden kann, benötigt ein lokal verteiltes Netzwerk bereits ausgefeiltere Methoden: Zeit muss synchron gehalten werden, Netzwerkverbindungen können ausfallen und beteiligte Komponenten können sich entgegen des Protokolls verhalten. Der Schritt zu Peer-to-Peer Netzen bringt weitere Einschränkungen mit sich wie z.B. schlechte Netzwerkverbindungen, extrem niedrige Verfügbarkeit, schwache und heterogen verteilte Rechenleistung sowie Totalausfälle, die erst spät bemerkt werden können. Das verdeutlicht, dass gut erforschte Algorithmen und Strategien aus (lokal) verteilten System zwar prinzipiell verwendet werden können, jedoch unter den genannten Umständen genau evaluiert und entsprechend angepasst oder sogar verworfen werden müssen.

Neben der erhöhten Verfügbarkeit lässt sich mit Replikation auch das Problem von „Hot-Spots“ abschwächen. Diese entstehen, wenn sehr beliebte Inhalte auf nur ein oder wenige Peers verteilt werden und das Peer-to-Peer Netzwerk somit zu einer zentralen Struktur entartet. Die so erzeugte Last ist für einzelne Peers kaum zu verarbeiten, da sie sehr wahrscheinlich nur über einen winzigen Bruchteil der kombinierten Bandbreite und Rechenleistung der Interessenten verfügen.

### 2.2.1 Verteilung und Stabilisierung

Im Kontext Replikation in Peer-to-Peer Netzwerken können die beiden folgenden Operationen unterschieden werden.

**Verteilung/Publikation** Sobald ein Peer ein Datum erzeugt, das im Netzwerk verfügbar gemacht werden soll, muss dieses auf eine Anzahl von anderen Peers verteilt werden. Zur Verteilung werden die unterschiedlichsten Ansätze

verwendet. In unstrukturierten Netzen kommen häufig probabilistische Algorithmen zum Einsatz, bei denen die Replikate auf eine zufällige Teilmenge von Knoten verteilt werden. In jedem Fall besteht bei dieser Operation eine Herausforderung in der Ermittlung der notwendigen Anzahl an Kopien. Diese ist zumeist abhängig von der Netzwerkgröße und der gewünschten Erreichbarkeitswahrscheinlichkeit der so verteilten Daten.

**Stabilisierung** Würde das Netzwerk nach der Verteilung in der gleichen Zusammensetzung weiterbestehen, so wäre kein zusätzlicher Eingriff notwendig. Diese Annahme trifft jedoch offensichtlich nicht zu, da ständig Peers hinzukommen oder das Netzwerk verlassen. Zur Stabilisierung muss daher zeit- oder ereignisgesteuert überprüft werden, ob Replikate noch in der intendierten Anzahl vorhanden sind. Falls nicht, müssen entsprechend neue Kopien verteilt oder überflüssige Kopien gelöscht werden.

### 2.2.2 Veränderliche und versionierte Replikate

Je nach Einsatzgebiet kann es notwendig sein, dass sich Replikate nachträglich verändern lassen und somit versioniert sind. In der Literatur spricht man hierbei von „mutable“ (veränderlichen) Replikaten. Die einfachste Form der Replikation erlaubt nur unveränderliche, also statische Daten, was für eine große Anzahl von Anwendungen ausreichend ist. Dabei werden einige sehr kritische Probleme, die bei veränderlichen Replikaten auftreten, umgangen wie z.B. das Aktualisieren aller vorhandener Kopien. Diese Aufgabe wird dadurch erschwert, dass replizierende Peers während einer Änderung nicht im Netzwerk sein können und nachträglich benachrichtigt und aktualisiert werden müssen. Des Weiteren müssen Konflikte und gleichzeitige Änderungen aufgelöst werden, eine solche Behandlung hängt meist stark von der Semantik der Daten ab.

Dieses Problem lässt sich zwar prinzipiell direkt innerhalb des Peer-to-Peer Systems lösen, jedoch geht damit eine große Steigerung der Komplexität des Systems einher. Da nicht jede darauf aufbauende Anwendung diese Art der Replikation benötigt, resultiert daraus in solchen Fällen möglicherweise ein Overhead. Je nach Design der darunter liegenden Peer-to-Peer Schicht ist es jedoch möglich, eine solche Funktionalität durch die Anwendung bereitzustellen.

### 2.2.3 Bisherige Ansätze in Peer-to-Peer Umgebungen

In [6] wird ein Peer-to-Peer Replikationsmechanismus für strukturierte Netzwerke vorgestellt. Dieser arbeitet transparent auf einer verteilten Hashtabelle (Distributed Hash Table – DHT) und kann probabilistische Verfügbarkeitsgarantien liefern. Objekte sind hier unter einem Identifikator in einer DHT gespeichert. Statt zu einem gegebenen Objekt eine Liste von Replikat-Identifikatoren explizit zu speichern, wird mittels einer Hashfunktion eine unendliche Folge von Replikaidentifikatoren aus der Objekt-ID generiert. Dadurch wird eine Indirektionsschicht eingespart und jedem Peer sind durch alleinige Kenntnis der Objekt-ID alle möglichen Orte für Replikate des Objekts bekannt. Dadurch, dass Positionen einzelner Replikate ermittelt werden können, kann die Gesamtverfügbarkeit an Peers sowie die Verfügbarkeit einzelner Replikate empirisch ermittelt werden. Durch eine Abfrage einer kleinen repräsentativen Menge an Peers und anschließender statistischer Betrachtungen lassen sich relativ präzise Aussagen über die tatsächlichen Werte treffen. Dadurch ist es dem Algorithmus möglich, die Anzahl der Replikate derart zu wählen, dass eine untere Schranke für die Verfügbarkeitswahrscheinlichkeit eines Objekts eingehalten wird.

Mit Replikation und Suche in unstrukturierten Netzwerken beschäftigt sich [7]. Hier findet Replikation jedoch nur als Reaktion auf eine erfolgreiche Suchanfrage statt. Wird ein unbeliebtes Objekt für eine längere Zeit nicht gesucht, besteht somit die Gefahr, dass das Objekt und alle Replikate dessen verloren gehen. Es werden drei verschiedene Mechanismen evaluiert.

1. Die besitzerbasierte Replikation speichert eine Kopie des gefundenen Datums auf dem suchenden Knoten, was zu einer Replikatverteilung proportional zur Beliebtheit führt.

Von den Autoren favorisiert, weil optimal hinsichtlich des Gesamtnetzwerkverkehrs, ist die „Square-Root“-Verteilung, für die zwei verschiedene Varianten vorgestellt werden.

2. Die erste Variante repliziert entlang des Pfades zwischen suchendem Peer und dem Peer, welches das gewünschte Objekt besitzt. Dabei kommt es jedoch zu starken topologischen Verdichtungen in der Verteilung.
3. Eine zweite Variante vermeidet solche Verdichtungen, indem nur die Pfadlänge $k$ ermittelt und anschließend auf $k$ zufällige Peers repliziert wird.

[2] stellt einen autonom arbeitenden Algorithmus vor, der ein globales Verzeichnis benutzt. Als Grundlage wird hier PlanetP [3] verwendet, welches einen Publish/-Subscribe Mechanismus implementiert. Die Autoren geben jedoch an, dass sich diese Architektur auch auf eine verteilte Hashtabelle übertragen lässt, wenn periodische Abfragen in Kauf genommen werden. Jedes Peer ist hier für eine Menge an Dateifragmente zuständig; Peers und deren verwaltete Fragmente sowie geschätzte Verfügbarkeiten werden in einem globalen Verzeichnis hinterlegt. Periodisch werden Daten mit geringer Verfügbarkeit an zufällige andere Peers weitergegeben, wobei Peers mit hoher Verfügbarkeit bevorzugt werden. Dadurch ist eine sehr geringe Anzahl an Replikaten notwendig, um hohe Gesamtverfügbarkeit zu garantieren. In einer ähnlichen Arbeit [1] wird ebenfalls PlanetP verwendet, die Anforderung an die Replikation ist hierbei, dass mindestens eine Kopie im gesamten Netzwerk vorhanden ist.

In [13; 4] werden primär Updatestrategien für Replikate vorgestellt, d.h. hier steht die Wahrung der Konsistenz im Vordergrund. Dazu müssen Replikate durch neue Versionen ersetzbar sein und ggf. alte Versionen aus dem Netzwerk entfernt werden. Diese Problemstellung liegt außerhalb des Rahmens dieser Arbeit und wird daher hier nicht genauer betrachtet.

# Kapitel 3

# BubbleStorm

Das im Zuge dieser Arbeit verwendete Peer-to-Peer System BubbleStorm ist ein neuartiger Ansatz, der beliebige Suchanfragen erlaubt. Im Gegensatz zum allgemeinen Trend der Forschung in jüngster Zeit setzt BubbleStorm auf eine unstrukturierte Architektur. Der grobe Aufbau und die Funktionsweise von BubbleStorm sollen in diesem Kapitel kurz dargestellt werden, detaillierte Beschreibungen, eine Simulation sowie analytische Ergebnisse finden sich in [12; 10].

Um Abfragen zu beantworten, verteilt BubbleStorm sowohl die Abfrage an sich als auch Daten auf zufällige Peers im Netzwerk. Erhält ein Peer eine Abfrage, die auf ein lokal vorhandenes Datum passt, so kann die Abfrage beantwortet werden. Die Wahrscheinlichkeit, dass eine solche Übereinstimmung gefunden wird ist überraschend hoch. Die Grundlage hierfür liefert das Geburtstagsparadoxon: Wählt man eine zufällige Gruppe von Personen, so ist die Wahrscheinlichkeit, dass zwei Personen am gleichen Tag Geburtstag haben (ohne Jahrgang), bereits für kleine Gruppen sehr hoch. Bei einer Gruppengröße von 41 Personen liegt sie beispielsweise bei etwa 90%.

Dadurch, dass Anfragen bei jedem Peer lokal verarbeitet werden und mit hoher Wahrscheinlichkeit auf dem gesamten Datensatz operieren, sind sehr vielfältige und komplexe Suchanfragen möglich. Zudem ist die Gestalt solcher Abfragen vollständig der Anwendung, die auf BubbleStrom aufbaut, überlassen. Dazu zählen beispielsweise Bereichsabfragen, unscharfe Suche sowie Volltextsuche ohne zusätzlich verteilten Index.

Zur Zeit existiert ein Prototyp einer BubbleStorm-Implementierung in Java, der die Grundlage für die Implementierung und Evaluation dieser Arbeit darstellt.

## 3.1 Topologie

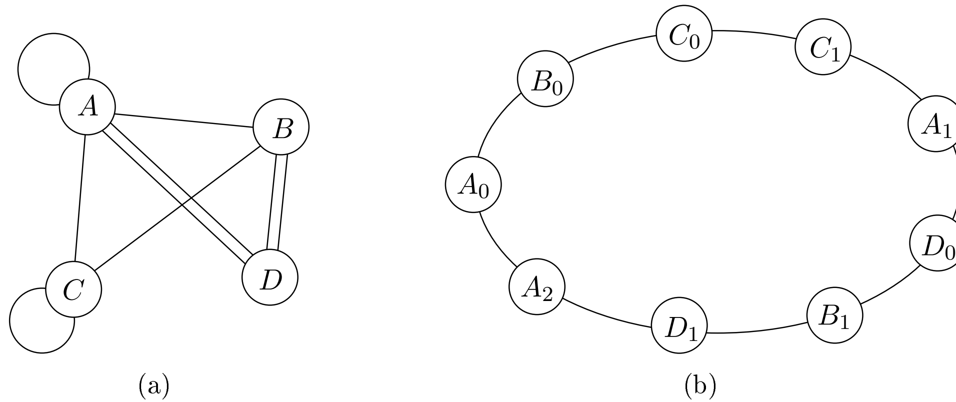



Abbildung 3.1: Ein BubbleStorm-Netzwerk dargestellt als Mehrfachkantengraph (a) und Zyklus (b)

Das BubbleStorm Netzwerk bildet einen zufälligen Multigraph, d.h. Mehrfachkanten sind zulässig. Dabei strebt jeder Knoten eine gerade Anzahl von Kanten (Verbindungen) an. Diese Anzahl kann vom Knoten selbst gewählt werden, typischerweise proportional zur Leistungsfähigkeit. Das erlaubt es BubbleStorm eine heterogene Leistungsverteilung optimal auszunutzen. Diese Topologie kann als Mehrfachkantengraph $G = (V, E)$ modelliert werden, wobei $V$ die Menge der tatsächlichen Peers sind und die Multimenge $E$ Verbindungen zwischen Peers darstellt.

Da jeder Knoten eine gerade Anzahl an Verbindungen wählt, existiert ein Eulerkreis auf diesem Netzwerk, d.h. ein Zyklus, der jede Kante im Graph genau ein mal besucht. Dieser Zyklus lässt sich wiederum als Graph auffassen, wobei die Knoten hier als *Positionen* eines physikalischen Knotens bezeichnet seien. Jeder Knoten $v \in V$ im Multigraph bildet somit auf eine Menge von Positionen im Zyklus ab. Für die nun folgenden Betrachtungen ist die Sicht auf das Netzwerk als Zyklus geeigneter, daher wird diese verwendet.

### 3.1.1 Netzwerkbeitritt und Austritt

Das Netzwerk wird durch den ersten Knoten erzeugt, der eine Anzahl von Kanten zu sich selbst und somit den Zyklus erstellt. Tritt ein Knoten dem Netzwerk bei, so integriert er sich an mehreren zufälligen Stellen im Zyklus. Dazu wird eine zufällige Kante $(a, b)$ ausgewählt und der neue Knoten $c$ positioniert sich zwischen diese, womit $(a, b)$ durch $(a, c), (c, b)$ ersetzt wird. Dieser Prozess wird solange wiederholt, bis der gewünschte Knotengrad erreicht ist. Es ist zu beachten, dass der Zyklus dabei erhalten bleibt und der Grad anderer Knoten sich dadurch nicht verändert.

Beim Verlassen des Netzwerks macht ein Knoten $c$ seine Änderungen am Graph rückgängig, indem er seine erzeugten Kanten $(a', c), (c, b')$ wieder durch $(a', b')$ ersetzt[1]. Auch hierbei bleibt der Zyklus erhalten und die Grade anderer Knoten konstant.

### 3.1.2 Client-Modus

In BubbleStorm werden zwei Arten von Knoten unterschieden. Neben den regulären *Peers* existiert der *Client*-Modus für leistungsschwache Knoten, bei dem nur eine Verbindung zu einem (stärkeren) Peer gehalten wird und keine Topologienachrichten verarbeitet werden. Tritt ein Knoten dem Netzwerk bei, so befindet er sich zunächst immer im Client-Modus. Erst wenn der Beitrittsvorgang beendet ist, wird er gegebenenfalls zum Peer. Falls nicht explizit gegensätzlich erwähnt, sind mit „Knoten“ in dieser Arbeit immer vollwertige Peers gemeint.

## 3.2 Bubbles und BubbleCast

Benutzernachrichten innerhalb von BubbleStorm werden per *BubbleCast* versendet. Ein BubbleCast adressiert eine kleine zufällige Teilmenge des Netzwerks, die im Folgenden *Bubble* genannt wird. Diese Bubble kann auch inkrementell adressiert werden, um die Anfrage schrittweise auszuweiten. Es werden sowohl Daten

[1] Es muss nicht notwendigerweise $a = a'$ bzw. $b = b'$ gelten, da sich die Nachbarknoten durch Betritt/Austritt verändert haben können.

als auch Anfragen als BubbleCast verschickt; sobald ein Peer beide Nachrichten erhält, kann es die Anfrage beantworten. Die Wahrscheinlichkeit, dass dieser Fall eintritt lässt sich mit Hilfe des *Balls-in-Bins*-Problems aus der Kombinatorik berechnen. Dabei handelt es sich um eine generalisierte Variante des eingangs erwähnten Geburtstagsparadoxons, bei der anschaulich zwei verschiedenfarbige Mengen von Bällen zufällig auf eine Menge von Eimern verteilt werden. Landen in einem Eimer zwei Bälle unterschiedlicher Farben, so kann die Anfrage im Sinne von BubbleStorm beantwortet werden. Die Bubblegrößen sind jeweils anpassbar, sodass unter Inkaufnahme stärkerer Ressourcennutzung eine nahezu beliebige hohe Beantwortungswahrscheinlichkeit einer Anfrage erzielt werden kann. Zwei komplementäre Bubbles (Daten und Abfrage) müssen dabei keineswegs die gleiche Größe aufweisen: Sind Abfragen deutlich häufiger als neue Daten, so bietet es sich an, die Datenbubble größer und die Abfragebubble kleiner zu dimensionieren, um den Datenverkehr aus globaler Netzwerksicht zu minimieren.

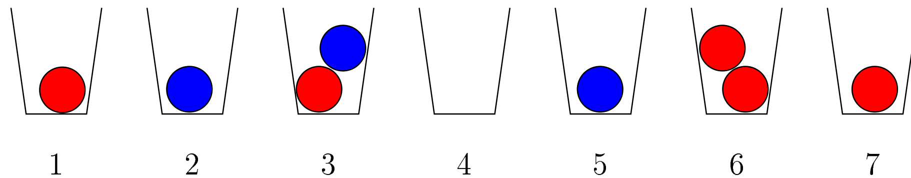



Abbildung 3.2: Balls-in-Bins Veranschaulichung. Nach der zufälligen Verteilung der Bälle befinden sich im dritten Eimer beide Farben. Übertragen auf BubbleStorm stellen die beiden Farben jeweils Bubbles dar, die sich in diesem Beispiel überschneiden. Somit kann die Abfrage zu einem Datum beantwortet werden.

Die Ausbreitung der Nachricht wird durch eine Anzahl an zu erforschenden Knoten strikt beschränkt. Zu diesem Zweck ist die (restliche) Bubblegröße in der Nachricht vermerkt. Im Gegensatz zu einem typischen Flutungsmechanismus, der an alle ausgehenden Verbindungen weiterleitet, wird die verbleibende Bubblegröße an höchstens $m$ Knoten weitergeleitet, wobei jeder Knoten einen gleich großen Anteil der Restgröße erhält. Die Nachricht induziert somit eine baumartige Verbreitungsstruktur, wodurch jeder Knoten nur eine konstante Anzahl an Weiterleitungen vornehmen muss.

Bubblegrößen sind abhängig von der Netzwerkzusammensetzung sowie von freien Variablen, die der Benutzer wählen kann. Je nach Aufgabe und Funktion einer

Bubble sind gegebenenfalls unterschiedliche Bubblegrößen notwendig. Daher kann der Benutzer verschiedene Bubble-Typen definieren, die jeweils unabhängig durch freie Variablen parametrisiert werden können und somit unterschiedliche Größen aufweisen können.

## 3.3 Messung

Um einige wichtige Größen wie z.B. Bubblegrößen oder Replikationsfaktor zu berechnen, ist die Kenntnis der Gesamt-Netzwerkgröße und die grobe Beschaffenheit des Netzes erforderlich. BubbleStorm verwendet dazu eine Erweiterung [11] des in [5] vorgestellten Algorithmus. Die gemessenen Werte sind dabei

$$D_i := \sum_{v \in V} \deg(v)^i \qquad i \in \{0, 1, 2\}$$

wobei $V$ die Menge der Knoten im Netzwerk ist und $\deg(k)$ der Grad des Knotens $k$. Ist $n$ die Gesamtnetzwerkgröße, so gilt offensichtlich $n = D_0$. Die dazu notwendigen Nachrichten werden periodischen Ping-Nachrichten angefügt und stellen somit nur einen minimalen Overhead dar. Der Algorithmus arbeitet rundenbasiert, die aktuelle Messrunde sei mit $\tau$ bezeichnet.

Aus diesen Werten errechnet sich wiederum der *Match Threshold*

$$T := \frac{D_1^2}{D_2 - 2D_1}$$

welcher eine erste Annäherung an die Gesamtnetzwerkgröße $n$ darstellt, jedoch auch Knotengradheterogenität (und somit Leistungsheterogenität) berücksichtigt [10]. Mit zunehmender Heterogenität sinkt $T$, da die Leistung eines Knotens quadratisch zu seinem Knotengrad ist.

# Kapitel 4

# Ansatz

## 4.1 Anforderungen

An dieser Stelle seien noch einmal kompakt wiederholend die Kernanforderungen an den Algorithmus genannt. Diese sind:

**Stabilisierung** Der Algorithmus soll eine gegebene Replikatanzahl, die sich als Funktion diverser Faktoren verändert, erreichen.

**Dezentral** Es soll kein (verteiltes) Verzeichnis geben, das Metainformationen (Ort, Version, etc.) zu Replikaten enthält.

**Besitzerlos** Für ein gegebenes Datum gibt es keinen designierten Besitzer, der primär oder ausschließlich für die Replikation dieses Datums verantwortlich ist.

**Persistenz** Ein Knoten kann zeitweise das Netzwerk verlassen und beim Wiederbeitritt vorher gespeicherte Replikate einbringen.

**Heterogenität** Leistungsunterschiede zwischen einzelnen Knoten sollen berücksichtigt werden, indem jeder Knoten Arbeit proportional zu seiner Leistung verrichtet.

Auch wenn die Persistenz-Anforderung zunächst trivial erscheint, muss streng kontrolliert werden, welche Replikate von einer früheren Sitzung weiterverwendet werden können. Wird dies nicht geprüft, so besteht die Gefahr der Überbevölkerung eines Replikats, da sich die gewünschte Replikatanzahl zwischenzeitig geändert haben könnte.

## 4.2 Modell

Das hier vorgestellte Modell ergänzt – soweit möglich – das in [12; 10] verwendete Modell. Eine Kenntnis dessen ist jedoch zum Verständnis nicht erforderlich, vielmehr teilen sich die Modelle einen Namensraum.

Ein beitretender Knoten, der aktiv den Replikationsmechanismus startet, wird stets $J$ genannt. Andere Knoten, die mit $J$ im Zuge der Replikation kommunizieren sind mit $Z_i$ bezeichnet, was unmittelbar eine $1 : n$-Beziehung impliziert.

### 4.2.1 Replikate

Ein Replikat wird über das Tupel $(r, t)$ identifiziert, wobei $r \neq 0$ ein eindeutiger Replikat-Identifikator und $t \neq 0$ ein Replikat-Typ ist. Die Gruppierung nach Typ ist notwendig, da das angestrebte Replikationsausmaß im Folgenden durch den Typ bestimmt wird. Somit können verschiedene Arten von Daten unterschiedlich stark repliziert werden.

Jeder Replikat-Typ hat somit seinen eigenen Identifikator-Raum für Replikate[1], der Typ entspricht stets dem Bubble-Typ in 3.2. Zu jedem Replikat existiert außerdem ein Zeitpunkt $\tau_{r,t}$, zu dem dieses Replikat publiziert wurde. Die eigentlichen Daten, die zu einem Replikat gehören, seien mit $\delta_{r,t}$ bezeichnet.

### 4.2.2 Replikationsfaktor

Der Replikationsfaktor, der für jeden Replikat-Typ $t$ individuell gewählt wird, gibt an, wie groß der Anteil der Knoten im Netzwerk ist, der ein Replikat besitzt. In BubbleStorm wird dieser Anteil zunächst durch die Publikation von Daten innerhalb einer Bubble festgelegt. Das bedeutet, dass die Replikatanzahl zum Zeitpunkt der Replikation genau der Bubblegröße $s_t$ entspricht. Diese Größe wird als Grundlage verwendet, um den angestrebten Replikationsfaktor $p_t$ zu errechnen. Um die Leistungsheterogenität im Netzwerk zu berücksichtigen, kommt dabei

[1] Eine Implementierung kann $t$ auch als Teil von $r$ realisieren, wodurch der Identifikator atomar ist.

nicht die Netzwerkgröße $n$ zum Einsatz, sondern der *Match Threshold* $T$, d.h.

$$p_t = \frac{s_t}{T}$$

Die angestrebte Anzahl an Replikaten ist somit $h_t := p_t n$.

Falls ein Replikat-Typ $t$ für die persistente Speicherung vorgesehen ist, wird der zum Netzwerkaustritt gültige Replikationsfaktor $q_t$ sowie der Zeitpunkt des Austritts $\tau_0$ mitgesichert.

## 4.3 Algorithmus

### 4.3.1 Idee

In einer idealisierten Umgebung ließe sich der Replikationsfaktor wie folgt stabilisieren. Ändert sich die Netzwerkstruktur (Knoten stößt hinzu oder verlässt das Netzwerk), so fragt ein beliebiger Knoten alle anderen Knoten nach allen Replikaten. Dadurch ist praktisch eine komplette Netzwerkkenntnis gegeben. Stimmt der Replikationsfaktor nicht mehr, wird entsprechend durch Löschen oder Verteilen reagiert. Dieses Verfahren ist jedoch aus mehreren Gründen in der Praxis nicht umsetzbar. Zunächst ist es schwierig zu erkennen, ob ein Knoten das Netzwerk verlassen hat. Er könnte nur vorübergehen blockiert sein, an einer anderen Stelle dem Netzwerk beigetreten sein und schließlich darf nur genau ein Knoten das Wegfallen bemerken. Noch viel gravierender ist jedoch das Anfordern aller Replikate im Netzwerk: Der Erwartungswert dieses Datenvolumens ist die Summe der Daten multipliziert mit der Replikatanzahl. Für einen einzelnen Knoten ist diese Menge offensichtlich nicht zu bewältigen. Des Weiteren ist eine komplette Netzwerkkenntnis in den meisten Systemen, so auch BubbleStorm, aufgrund der massiven Verteilung nicht möglich.

Im Folgenden wird daher ein probabilistischer Algorithmus vorgeschlagen, der sich *Churn* zunutze macht, um die Replikatanzahl zu stabilisieren. Churn bezeichnet den Prozess des permanenten Verlassens und Hinzustoßens von Knoten im Netzwerk. Durch das Verlassen eines Knotens können Replikate verloren gehen, die dann von einem anderen Knoten beim Beitritt repliziert werden könnten.

Für Ersteres lassen sich selbstverständlich nur Aussagen probabilistischer Qualität treffen, was sich dementsprechend auf Replikationsstrategie beim Beitritt auswirken muss.

Der tatsächliche Replikationsfaktor sei mit $\tilde{p}$ und der angestrebten Replikationsfaktor mit $p$ bezeichnet. Liegt gleichmäßiger Churn vor, d.h. die Netzwerkgröße ändert sich nicht, so bietet sich zur Stabilisierung der Replikatanzahl folgendes Verfahren an. Tritt ein Knoten dem Netzwerk bei, so repliziert er das Datum mit der Wahrscheinlichkeit $p$. Verlässt ein Knoten das Netzwerk, so geht ein Replikat mit der Wahrscheinlichkeit $\tilde{p}$ verloren. Für eine variierende Anzahl an tatsächlichen Replikaten und dem dazugehörigen Replikationsfaktor $\tilde{p}$ lassen sich dann die nachfolgenden Fälle unterscheiden.

| | | |
|---|---|---|
| (i) | $p = \tilde{p}$ | Der tatsächliche Replikationsfaktor entspricht dem angestrebten Replikationsfaktor. Da Replikation und Verlust gleich wahrscheinlich sind, bliebt die Replikatanzahl konstant. |
| (ii) | $p > \tilde{p}$ | Es sind zu wenige Replikate im Netzwerk vorhanden. Da Replikation wahrscheinlicher als Verlust ist, steigt die Replikatanzahl. |
| (iii) | $p < \tilde{p}$ | Es sind zu viele Replikate im Netzwerk vorhanden. Da Verlust wahrscheinlicher als Replikation ist, sinkt die Replikatanzahl. |

Folglich wird die tatsächliche Replikatanzahl in den Fällen $p \neq \tilde{p}$ in Richtung $p$ korrigiert und ist somit konvergent. Dieser Mechanismus lässt sich durch eine dynamische Anpassung der Replikationswahrscheinlichkeit erweitern. Sind beispielsweise viel mehr Replikate im Netzwerk als notwendig, so wäre eine Replikationswahrscheinlichkeit kleiner als $p$ geschickt, um die Konvergenzgeschwindigkeit zu erhöhen. Dazu ist allerdings die Messung des tatsächlichen Replikationsfaktors $\tilde{p}$ notwendig, um zu wissen, wie stark die Abweichung zu $p$ ist.

Eine solche Messung kann implizit durch die Abfrage anderer Knoten und anschließender Auswertung erfolgen. Die Wahrscheinlichkeit, dass bei einer Abfrage von $m$ Knoten, die jeweils das Replikat mit einer Wahrscheinlichkeit von $\tilde{p}$ besitzen, genau $y$ Knoten tatsächlich über das Replikat verfügen ist

$$\Pr[Y = y] = \binom{m}{y} \tilde{p}^y (1 - \tilde{p})^{m-y},$$

mit $y = 1$ folgt

$$\Pr[Y = 1] = m\tilde{p}(1 - \tilde{p})^{m-1}.$$

Fragt man nun $m := 1/p$ Knoten ab, so erhält man mit der Wahrscheinlichkeit

$$\frac{\tilde{p}}{p}(1 - \tilde{p})^{1/p-1} \tag{4.1}$$

genau ein Replikat. Ist das tatsächlich der Fall, d.h. das Replikat wurde genau ein Mal empfangen, so wird das Replikat mit der Wahrscheinlichkeit

$$\frac{p}{(1 - p)^{1/p-1}} \tag{4.2}$$

repliziert. Das bedeutet für die kombinierte Replikationswahrscheinlichkeit $\rho$ aus (4.1) und (4.2)

$$\rho := \frac{\tilde{p}}{p}(1 - \tilde{p})^{1/p-1} \cdot \frac{p}{(1 - p)^{1/p-1}}.$$

Der aufmerksame Leser stellt fest, dass $\rho = p$ für $p = \tilde{p}$. Für den Fall dass $p \neq \tilde{p}$ wird $\rho$ abgeschwächt (siehe Abbildung 4.1).

### 4.3.2 Einfacher Ansatz

Basierend auf den Resultaten im vorherigen Abschnitt lässt sich jetzt eine erste einfache Version des Replikationsprotokolls für BubbleStorm formulieren, die zunächst nur einen Replikat-Typ unterstützt und die Persistenz- sowie Heterogenitätanforderung außen vor lässt. Hierbei soll die Kernidee anhand einer komplexitätsreduzierten Variante des Algorithmus veranschaulicht werden. In der ersten

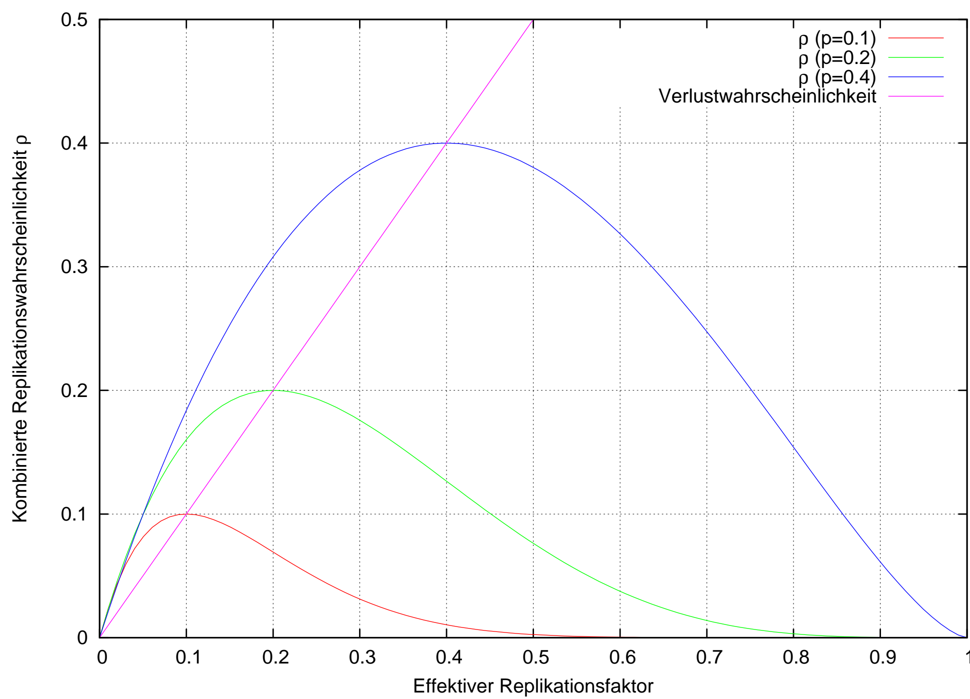



Abbildung 4.1: Abschwächung der Replikationswahrscheinlichkeit $\rho$. Die Abbildung zeigt die kombinierte Replikationswahrscheinlichkeit $\rho$ als Funktion des effektiven Replikationsfaktors $\tilde{p}$ für drei verschiedene angestrebte Replikationsfaktoren $p$ (0.1, 0.2 und 0.4). Hier ist deutlich zu erkennen, dass für den stabilen Zustand $\tilde{p} = p$ unmittelbar $\rho = p$ folgt.

Phase des Protokolls sucht der beitretende Knoten ($J$) eine zufällig gewählte Anzahl an Zielknoten ($Z_i$), wobei jeder Knoten im Netzwerk mit der gleichen Wahrscheinlichkeit ausgewählt wird. Insgesamt werden $m = 1/p$ Knoten kontaktiert. Zwischen $J$ und einem Knoten $Z_i$ läuft nun die zweite Phase des Protokolls ab, die nachfolgend dargestellt ist.

1. $J$ sendet eine uniform gewählte Zufallszahl $o \in [0, 1]$ an $Z_i$.
2. $Z_i$ entscheidet für jeden lokal verfügbaren Replikat-Identifikator $r$, ob dieser gesendet werden soll (Selektion). Dazu testet $Z_i$, ob $f(o, r) < p/((1-p)^{1/p-1})$ gilt. Replikat-Identifikatoren, die dieser Bedingung genügen, werden zu $J$ gesendet. Dieser Schritt entspricht der Realisierung der Wahrscheinlichkeit (4.2).

   Die Funktion $f$ ist jedem Knoten bekannt und dient dazu eine neue, gleichverteilte Zufallszahl aus $o$ und dem Replikat-Identifikator $r$ zu erzeugen. Dadurch wird ein Identifikator $r$ entweder von allen Knoten gesendet (sofern $r$ vorhanden ist) oder von keinem Knoten.
3. Alle Replikate, die $J$ genau ein mal erhalten hat, sollen repliziert werden. Zu diesem Zweck sendet $J$ die entsprechenden Replikat-Identifikatoren an einen Knoten $Z_i$, der das Replikat gespeichert hat und fordert die dazugehörigen Daten an.
4. $Z_i$ sendet die Replikat-Daten $\delta_r$ an $J$ zurück.

Dieses Protokoll zeigt die Kernidee des Algorithmus. Das Ziel, mit der Wahrscheinlichkeit $\rho$ zu replizieren, wurde dadurch erreicht, dass sowohl der Knoten $K_i$ also auch $J$ die Replikatmenge filtern. Dabei wird die zu tauschende Datenmenge stark reduziert. Die Wahrscheinlichkeit, dass beide Ereignisse (Filterung bei $K_i$ und Filterung bei $J$) gleichzeitig eintreten, ist gerade $\rho$ (4.3.1) und entspricht somit der gewünschten Replikationswahrscheinlichkeit. Die zufällige Auswahl von Zielknoten erfolgt dabei über einen BubbleCast an dessen äußeren Knoten *Random Walks*[2] der Länge $l$ gestartet wird.

Die verbleibenden Abschnitte in diesem Kapitel befassen sich nun mit der Erweiterung des Algorithmus um mehrere Replikat-Typen (d.h. unterschiedliche Replikationsfaktoren) und Persistenz.

[2] „Zufallsbewegung“ bzw. „Irrfahrt“: Die Nachricht wird zufallsbasiert weitergeleitet

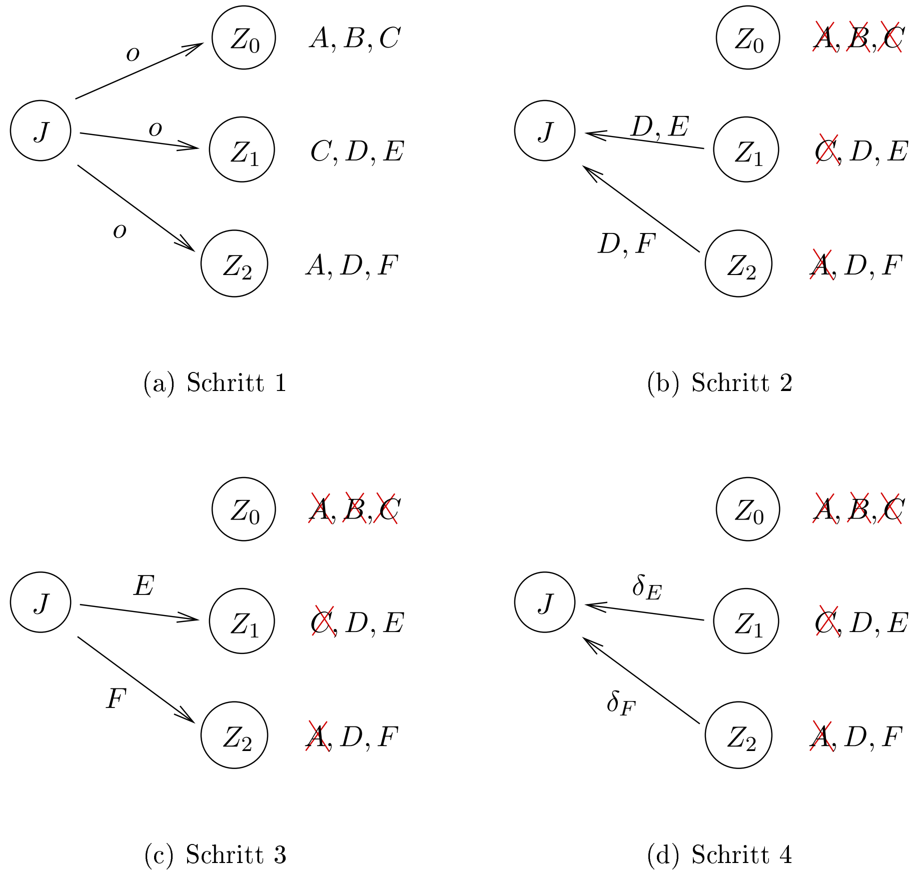



Abbildung 4.2: Darstellung des einfachen Replikationsalgorithmus. Zunächst wird die Zufallszahl $o$ verteilt (a), dann die lokal vorhandenen Replikate als Funktion von $o$ gefiltert und zurückgeschickt (b). Alle einmal erhaltenen Replikate werden dann von den jeweiligen Knoten angefordert ((c) und (d)).

### 4.3.3 Erweiterte Version

Um mehrere Replikationsfaktoren und somit mehrere Replikat-Typen zu unterstützen, ist eine Anpassung des Protokolls notwendig. Da die Anzahl der abgefragten Zielknoten von diesem Faktor abhängt und direkt in die Replikationswahrscheinlichkeit eingeht, muss das Protokoll entsprechend modifiziert werden. Statt $1/p$ Knoten abzufragen, werden $m = \max(1/p_t)$ über alle Replikat-Typ $t$ kontaktiert. Innerhalb dieser Knotenmenge wird für jeden Replikat-Typ $t$ eine Teilmenge bestimmt, deren Größe genau $1/p_t$ entspricht. Diese Auswahl wird dadurch getroffen, dass jedem Zielknoten genau eine Zahl $j \in \{0, 1, \ldots, m-1\}$ zugewiesen wird und einen Replikat-Typ $t$ nur verarbeitet, falls $j < 1/p_t$.

Für die Persistenzanforderung ist es zunächst notwendig, dass Knoten beim Verlassen des Netzwerks den Zeitpunkt $\tau_0$ sowie alle bekannten Replikationsfaktoren $p_t$ als $q_t$ speichern. Diese Daten werden beim Wiederbeitritt von den Zielknoten benötigt. Replikate, die während der Abwesenheit neu erzeugt wurden, können regulär behandelt werden. Existierte ein Replikat jedoch schon zum Austrittszeitpunkt, so wird die Replikationswahrscheinlichkeit mit der Differenz des alten und neuen Replikationsfaktors gewichtet. Ist der neue Replikationsfaktor kleiner als der alte, so werden persistent eingebrachte Replikate mit der Differenzwahrscheinlichkeit gelöscht.

Leistungsheterogenität wird in BubbleStorm durch variierende Knotengrade im Overlaynetzwerk abgebildet. Je leistungsfähiger ein Knoten ist, desto höher wird der Knotengrad gewählt, um somit verhältnismäßig mehr Anfragen bearbeiten zu können. Da der eigene Knotengrad $d$ stets bekannt ist und durch die Messwerte (3.3) $D_0$ und $D_1$ auch der durchschnittliche Knotengrad im Netzwerk, liefert das Verhältnis des eigenen Grads $d$ zum durchschnittlichen Grad, also

$$d \cdot \frac{D_0}{D_1},$$

einen zuverlässigen Gewichtungsfaktor für Heterogenität. Dieser Faktor findet an zwei Stellen Verwendung: Einerseits für die Replikationswahrscheinlichkeit $\rho$ sowie für das Löschen überflüssig gewordener lokaler Replikate im Falle eines niedrigeren Replikationsfaktors nach dem Wiederbeitritt.

#### 4.3.3.1 Protokoll- und Algorithmusbeschreibung

Basierend auf den Erweiterungen im vorherigen Abschnitt formulieren lässt sich jetzt die finale Version des Replikationsalgorithmus formulieren. Ein beitretender Knoten, der den Replikationsalgorithmus ausführt, verbindet zunächst zu $m = \max(1/p_t)$ Knoten. Um nicht einer selbstinduzierten DDoS-Attacke zu unterliegen, werden diese Knoten nicht gleichzeitig kontaktiert, sondern qualitativ sequenziell in kleineren Gruppen. Die genaue Strategie hier ist der Implementierung überlassen[3], jedoch muss prinzipiell davon ausgegangen werden, dass sich der gesamten Prozess über einen längeren Zeitraum erstreckt und dies entsprechend beachtet werden muss.

Anschließend läuft zwischen einem Knoten $Z_i$ $(1 \leq i \leq m)$ und dem beitretenden Knoten $J$ das folgende Protokoll ab.

1. Der Knoten $J$ startet eine Transaktion mit einem zufälligen Identifikator $j \in \{0, 1, \ldots, m-1\}$, d.h. dieser Identifikator wird nicht nochmals vergeben, es sei denn die Transaktion schlägt fehl. Danach werden folgende Daten $J \rightarrow Z_i$ übertragen.

   - $t_0$ Zeitpunkt des letzten Netzwerkaustritt, $t_0 = 0$ falls $J$ vorher noch nie im Netzwerk war.
   - $o$ Uniform gewählte Zufallszahl $o \in \{0, 1, \ldots, 2^{64} - 1\}$.
   - $d$ Den von $J$ angestrebten Knotengrad.
   - $j$ Transaktionsidentifikator, siehe oben.
   - $q_t$ Zum Zeitpunkt $t_0$ gültige Replikationsfaktoren für alle bekannten Replikat-Typen $t$. Falls $J$ noch nie im Netzwerk war, setze $q_t = 0$ für alle $t$.

   Die Bezeichnung dieser Nachricht ist `ACQUISITION_HELLO`.

2. Bei $Z_i$ werden nun alle Replikat-Typen $t$ verarbeitet, für die gilt $j < p_t^{-1}$. Dazu wird $f$ mit 4.2 und den Erweiterungen für Heterogenität und Persistenz gebildet,

$$f := \frac{1}{(1 - p_t)^{1/p_t - 1}} \cdot \frac{dD_0}{D_1}$$

[3] Der von uns gewählte Ansatz wird in Kapitel 5 beschrieben.

sowie

$$\begin{aligned} k_0 &:= \min(1, fp_t) \\ k_1 &:= \min(1, f(p_t - q_t)) \end{aligned}$$

Für jeden Replikat-Identifikator $r$ des Typs $t$ setzt $Z_i$

$$k := \begin{cases} k_0 \cdot 2^{64} & \text{falls } \tau_{r,t} < t_0 \\ k_1 \cdot 2^{64} & \text{sonst} \end{cases}$$

Die unteren 64 Bit des Replikatidentifikators seien mit $r_{64}$ bezeichnet. Falls nun $o \cdot r_{64} \mod 2^{64} < k$ sendet $Z_i$ den vollständigen Replikatidentifikator $(r, t)$ an $Z_i$. Die Bezeichnung dieser Nachricht ist `ACQUISITION_LIST`.

3. $J$ verwaltet zwei Listen von Replikatidentifikatoren. In der Empfangsliste $R$ befinden sich alle Replikatidentifikatoren, die gegebenenfalls von einer früheren Sitzung vorhanden sind. Eine weitere Liste $I$ enthält ignorierte Identifikatoren, die leer initialisiert wird. Für jeden empfangenen Identifikator aus dem zweiten Schritt, der sich nicht in der Ignorierliste $I$ befindet, wird nun folgendes Verfahren angewendet (falls sich der Identifikator in der Ignorierliste befindet, wird er ignoriert).
   a) Falls $(r, t) \in R$ verschiebe den Identifikator in $I$ und lösche den Identifikator aus der Liste der lokal vorhandenen Replikate. Die mit dem Identifikator assoziierten Daten $\delta_{r,t}$ werden ebenfalls gelöscht.
   b) Falls $(r, t) \notin R$ füge den Identifikator $R$ hinzu und fordere die assoziierten Daten $\delta_{r,t}$ von $Z_i$ an (`ACQUISITION_REQUEST`).
4. $Z_i$ sendet die von $J$ angeforderten Replikatdaten $\delta_{r,t}$ (`ACQUISITION_DATA`).

Falls einer der Protokollschritte einen Fehler verursacht (wie etwa eine Zeitüberschreitung auf Netzwerkebene) wird die Transaktion abgebrochen und der Transaktionsidentifikator $j$ kann wieder verwendet werden.

Sobald alle Transaktionen mit Schritt 4 abgeschlossen sind, ist noch der Fall $q_t > p_t$ zu handhaben, d.h. der neue Replikationsfaktor ist kleiner als er in einer früheren Sitzung war. Da $J$ nun möglicherweise zu viele Replikate hat, werden die Replikate aus früheren Sitzungen mit Wahrscheinlichkeit $(q_t - p_t)\frac{dD_0}{D_1}$ gelöscht. Anschließend ist der Algorithmus beendet und der Knoten ist vollständig dem Netzwerk beigetreten.

# Kapitel 5

# Implementierung

In diesem Kapitel werden die Implementationsdetails des Replikationsprotokolls präsentiert. Die Grundlage für die Implementierung stellt ein BubbleStorm-Prototyp in Java 1.6 dar, der alle grundlegenden BubbleStorm-Eigenschaften beinhaltet. Das Design dieses Prototyps ist darauf ausgelegt, mehrere Knoteninstanzen innerhalb eines Programms zu erzeugen und ist transparent gegenüber tatsächlicher Verteilung der Knoten auf unterschiedlichen Systemen. Die Netzwerkeingabe- und -ausgabe nutzt die Java-NIO-Schnittstelle, die unter anderem asynchrone (nicht blockierende) Sockets und Multiplexing[1] bereitstellt. Dadurch kann eine Knoteninstanz auch mit vielen Verbindungen theoretisch noch sehr gut skalieren.

Die im Rahmen dieser Arbeit entstandenen Quelltexte befolgen die für Java üblichen Formatierungs- und Bezeichnerkonventionen. Alle Klassen, Pakete und Methoden sind mit JavaDoc dokumentiert.

Per typographischer Konvention sind Klassen- und Methodennamen sowie alle weiteren Code-Elemente innerhalb dieses Kapitels in `Festbreitenschrift` gesetzt.

## 5.1 Design

Die Implementierung des Replikationssubsystem setzt auf mehreren Ebenen an. Zunächst wurde der BubbleStorm-Kern um die notwendigen Protokollnachrichten

[1] Unter *Multiplexing* versteht man in diesem Zusammenhang die gleichzeitige Überwachung von $n$ Sockets auf Ereignisse.

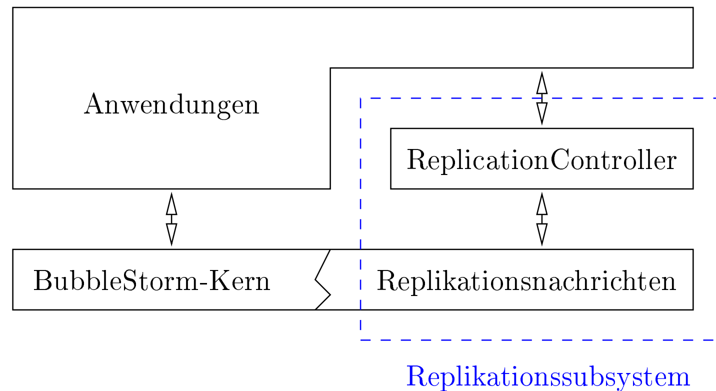



Abbildung 5.1: Architekturübersicht des Replikationssubsystems.

erweitert; dazu gehören jene Nachrichten, die über die BubbleStorm-Topologie geroutet werden. Auf einer Ebene darüber befindet sich der Kern des Replikationssystems, der eine Anwenderschnittstelle bereitstellt und mit dem darunter liegenden BubbleStorm-Kern kommuniziert. Ebenfalls auf dieser Ebene wird das Replikationsprotokoll zwischen zwei Knoten abgewickelt.

Diese Aufteilung wurde gewählt, um in den restlichen BubbleStorm-Kern so minimal wie möglich einzugreifen, damit diese Implementierung austauschbar und vom Replikationssystem entkoppelt ist. Ein weiterer Grund dafür war die Tatsache, dass der BubbleStorm-Kern zum Zeitpunkt dieser Implementierung noch selbst in Entwicklung war und somit potentiellen Änderungen unterlag.

Im Paket `bubblestorm.replication` befinden sich alle replikationsspezifischen Klassen, die nicht unmittelbar im Kern von BubbleStorm arbeiten.

### 5.1.1 Datenrepräsentation

Die Implementierung trennt Replikat-Metadaten, d.h. den Identifikator $(r, t)$ sowie den Erstellungszeitpunkt $\tau_{r,t}$, strikt von den eigentlichen Nutzdaten $\delta_{r,t}$. Eine solche Trennung ist aus folgenden beiden Gründen sinnvoll:

1. Die Anwendung kann selbst bestimmen wie Daten repräsentiert und gespeichert werden, was sowohl dem Design als auch der Performanz zuträglich

sein kann.

2. Verschiedene Anwendungen haben verschiedene Anforderungen an die Speicherung und Haltung von Daten, eine generalisierte Methode kann nicht alle Bedürfnisse abdecken.

| Größe | Bedeutung | Implementierung |
|---|---|---|
| $r$ | Replikat-Name | $n$-Bit Ganzzahl (`BigInteger`). |
| $t$ | Replikat-Typ | 32-Bit Ganzzahl (`int`). |
| $\tau_{r,t}$ | Publikationszeitpunkt | 32-Bit Ganzzahl (`int`). |
| $\delta_{r,t}$ | Replikat-Daten | Nutzdaten, beliebige Länge, `byte[]` |

Tabelle 5.1: Abbildung der Modellgrößen in der Implementierung.

Um die Daten so zu speichern, dass der Replikationsmechanismus auf diese zugreifen kann, muss lediglich das Interface `ReplicaDataProvider` implementiert werden. Die zentrale Methode `getData()` dient dann als Callback und wird aufgerufen, wenn die entsprechenden Daten zur Replikation benötigt werden.

Tabelle 5.1 zeigt, wie die im Modell eingeführten Variablenbezeichner in der Implementierung abgebildet wurden.

$(r, t)$ und $\tau_{r,t}$ sind in der Klasse `Replica` zusammengefasst. Die lokal vorhandenen Instanzen dieser Klasse und deren Replikationszeitpunkte werden von `ReplicaStorage` verwaltet.

Alle persistenten Daten werden von der Klasse `Persistency` in binärer Form geladen und gespeichert.

### 5.1.2 Replikationsprotokoll

In diesem Abschnitt wird die konkrete Implementierung des Replikationsprotokolls als Teil der Stabilisierung beschrieben. Darunter ist die Kommunikation zwischen einem dem Netzwerk beitretendem Knoten $J$ und mehreren Zielknoten $Z_i$ zu verstehen. Grundsätzlich teilt sich dieser Vorgang in zwei Phasen auf; in der ersten Phase werden Zielknoten gesucht, mit denen der Replikationsdialog gestartet wird. Dieser Dialog stellt dann die zweite Phase dar. Abbildung 5.2 zeigt die

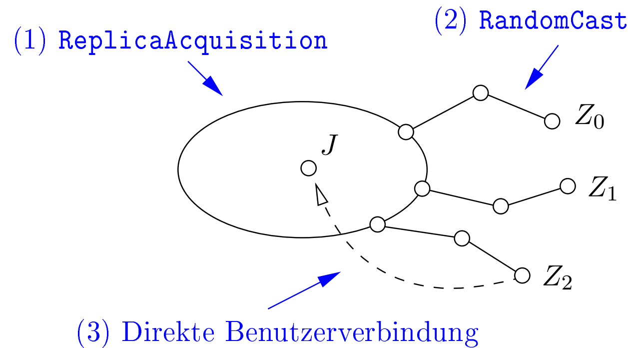



Abbildung 5.2: Zeitlicher Ablauf des Replikationsprotokolls: Ausgehend von $J$ leiten die äußeren Knoten eines BubbleCasts (1) die Nachrichten per Random Walk (2) an $Z_i$ weiter. Diese starten dann das Replikationsprotokoll mit $J$ durch eine Benutzerverbindung (3).

beiden Phasen im Überblick, in den nächsten beiden folgenden Unterabschnitten werden die Phasen jeweils ausführlich dokumentiert.

Der gesamte Prozess findet asynchron statt und wird durch den `ReplicationController` gesteuert. Dieser verwaltet sowohl die Transaktionsidentifikatoren für alle Direktverbindungen als auch die Ignorier- und Empfangsliste für Replikate. Hierbei kommen Java-Features zur Synchronisation wie beispielsweise `Locks` für Nebenläufigkeit zum Einsatz.

#### 5.1.2.1 Phase 1 – Zielsuche

Bei der Zielsuche wurde in Kapitel 4 von der genauen Vorgehensweise abstrahiert und davon ausgegangen, dass $m$ zufällige Knoten adressiert werden können. Um diese Anforderung in die Praxis umzusetzen, werden bei der Zielsuche zwei Topologienachrichten miteinander kombiniert. Dabei wird die Quelladresse jeweils weitergereicht, um somit in der zweiten Phase eine direkte Verbindung zu ermöglichen.

Zunächst wird die neu entworfene Nachricht `ReplicaAcquisition` versendet. Diese Nachricht ähnelt in Aufbau und Verhalten einer `BubbleCast`-Nachricht, d.h. sie verbreitet sich nach dem gleichen Algorithmus, kommt aber mit einigen Fel-

dern weniger aus und ist somit kleiner. Der Zweck hierbei ist es, die Knoten am Rand einer Bubble zu finden und anzusprechen, also mit möglichst großem Abstand zum beitretenden Knoten $J$. Wie bei einer `BubbleCast`-Nachricht ist es möglich, die Bubble in kleinen Schritten zu erforschen und somit inkrementell eine gewünschte Anzahl an Randknoten zu erreichen. Somit kann sichergestellt werden, dass $J$ nicht einer selbstinduzierten DDoS-Attacke unterliegt, weil zu viele Knoten adressiert wurden und mit der zweiten Phase starten. Die Größe der Bubble wird so gewählt, dass mindestens $2m$ Randknoten adressiert werden können. Damit wird sichergestellt, dass im Fall von verloren gegangenen Nachrichten noch genug Raum für weitere `ReplicaAcquisition`-Nachrichten innerhalb der Bubble ist. Die – nicht ganz triviale – Berechnung für die entsprechenden Start- und Endpositionen dieser partiellen BubbleCasts ist in `ReplicationController` implementiert.

Die Randknoten einer solchen Bubble senden dann eine Nachricht des Typs `RandomCast`. Dieser realisiert einen `Random Walk` der Länge $l$, wobei die Nachricht mit einer Lebensdauer $l$ ausgestattet ist, die an jedem Knoten reduziert[2] wird. Anschließend wird die Nachricht an einen zufälligen Nachbarknoten weitergeleitet. Sobald die Lebensdauer der Nachricht an einem Knoten unterschritten ist, initiiert dieser Knoten die zweite Phase des Protokolls, die im nächsten Abschnitt erläutert ist.

#### 5.1.2.2 Phase 2 – Replikationsdialog

In dieser Phase kontaktiert ein ausgewählter Zielknoten $Z_i$ den beitretenden Knoten $J$. Zwischen diesen Knoten wird dann konkret der (deterministische) Protokollablauf aus 4.3.3.1 befolgt, der in Abbildung 5.3 nochmals kurz in technischer Ausprägung visualisiert ist.

Im Gegensatz zu den vorherigen Nachrichten wird diese Kommunikation nicht über das Overlay-Topologienetzwerk abgewickelt, sondern mittels einer Direktverbindung im Sinne einer dedizierten TCP-Verbindung. Dadurch wird das Overlaynetzwerk nicht unnötig mit Daten überschwemmt, die ohnehin nur von den beiden Endknoten verarbeitet werden. Der BubbleStorm-Prototyp bietet hierzu

[2] Konkret ist $l$ in dieser Implementierung eine Ganzzahl, die jeweils um eins dekrementiert wird.

Abbildung 5.3: Protokollablauf in der zweiten Phase.

das Konzept der *Benutzerverbindungen* an. Prinzipiell handelt es sich dabei um eine Schnittstelle, die asynchrone (nicht blockierende) Ein- und Ausgabe auf einer TCP-Verbindung ermöglicht. Diese Art der Kommunikation ist sehr ressourcensparend, da sie keinen zusätzlichen Thread benötigt, jedoch müssen Anwendungen um dieses Konzept herum entworfen werden.

Die Kommunikation wird auf Code-Ebene durch die gemeinsame Oberklasse `AcquisitionConnection` ermöglicht. Die Klasse nutzt das Schablonenmethode Entwurfsmuster, und definiert die zentralen (abstrakten) Methoden `writeComplete()` und `checkReadComplete()`. Darüber wird der implementierenden Klasse mitgeteilt, dass alle Daten versendet oder neue empfangen wurden. Zu diesen Implementierern gehören die Klassen

`AcquisitionJoinConnection` welche den Replikationsdialog auf Seite von $J$, also dem beitretenden Knoten, repräsentiert und übernimmt alle dort anfallenden Arbeiten sowie

`AcquisitionTargetConnection` die den Replikationsdialog aus Zielknotensicht ($Z_i$) implementiert.

Aufgrund der asynchronen Kommunikation speichern beide Seiten Zustände, die in Abbildung 5.4 mit den Protokollschritten in Zusammenhang gebracht werden. Die Zustände ändern sich jeweils, sobald Daten komplett gesendet oder empfangen wurden.

Die vier verschiedenen Nachrichten wurden im Paket `bubblestorm.replication.messages` implementiert. Sie stellen jeweils Methoden zur Serialisierung und

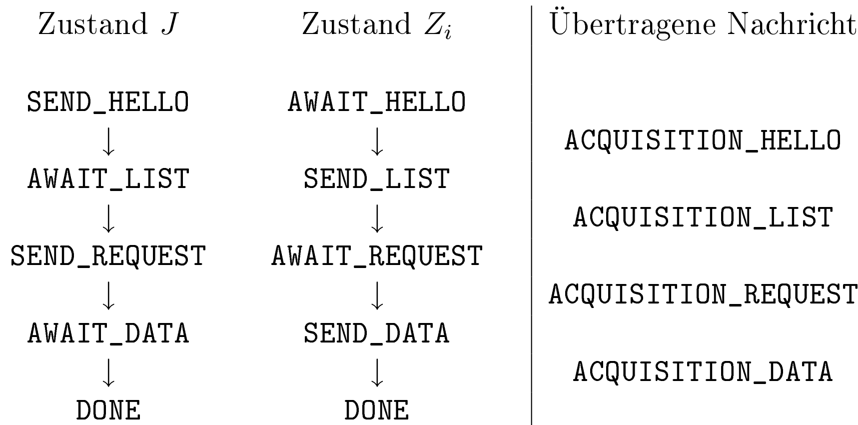



Abbildung 5.4: Zustände der beiden Kommunikationspartner im Zusammenhang mit den einzelnen Nachrichten des Protokolls.

Deserialisierung bereit, um einfach über eine Netzwerkverbindung versendbar zu sein. Die Deserialisierung kann jeweilig auch mit unvollständigen Daten aufgerufen, da dem Aufrufer nicht bekannt ist, ob die Nachricht komplett oder partiell vorliegt. Falls die vorhandenen Daten nicht für eine Deserialisierung ausreichen, wird dies durch das Werfen einer entsprechenden `MessageIncompleteException` signalisiert. Für diese Nachrichten wurden außerdem jeweils Unit-Tests programmiert, die auf korrekte Serialisierung und Deserialisierung prüfen, da dieser Mechanismus besonders häufig zu fehleranfälligen Implementierungen neigt.

### 5.1.3 Publikation und Replikation

Auch wenn die Publikation von Replikaten nicht unmittelbar Teil dieser Arbeit ist, wurden entsprechende Voraussetzungen geschaffen, um Replikate mit minimalem Aufwand publizieren zu können. Der von uns gewählte Weg besteht darin, einen regulären BubbleCast zusätzlich mit den Replikatinformationen zu versehen. Das hier zu konkretisierende Implementierungsdetail ist die Erweiterung von der Klasse `BubbleCast`, die eine BubbleCast-Nachricht auf Code-Ebene realisiert. Aufgrund der Eigenschaft als grundlegende Nachricht für die Kommunikation innerhalb der Topologie sollte jede Änderung, die zu einer größeren Nachrichtenlänge führt, vorsichtig abgewägt werden. Insgesamt mussten der Nachricht zwei neue Felder hinzugefügt werden, deren Länge jeweils 4 und $n$ Byte (hier $n = 16$)

beträgt. Da es sich hierbei um eine signifikante Größenänderung handelt und beide Felder nicht immer sinnvoll gesetzt werden können, wurde zunächst ein Byte hinzugefügt, das die Präsenz der beiden neuen Felder anzeigt. Demnach ist eine reguläre BubbleStorm-Nachricht nur ein Byte größer.

Bei den Feldern handelt es sich um den Replikatname $r$ sowie den ursprünglichen Replikationszeitpunkt $\tau_{r,t}$. Der Replikat-Typ $t$ entspricht dem Bubble-Typ und ist bereits in der BubbleCast-Nachricht enthalten; er musste folglich nicht hinzugefügt werden.

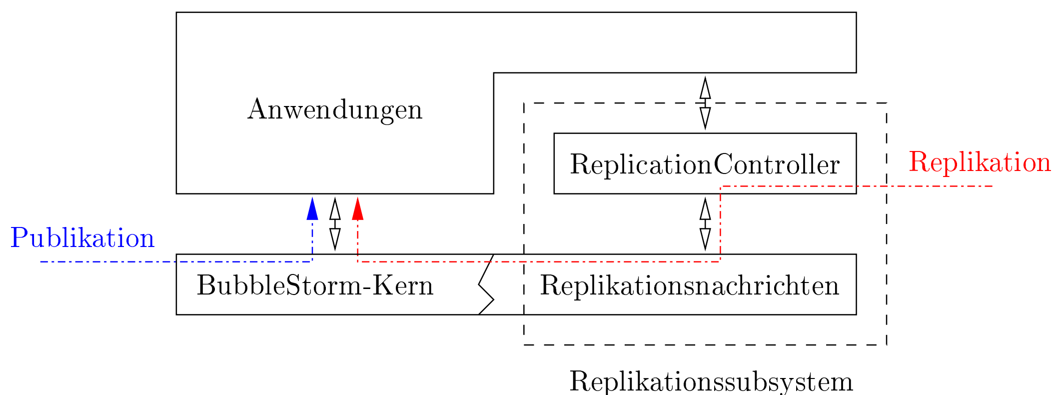



Abbildung 5.5: Zusammenhang zwischen Publikation und Replikation.

Ein Replikat kann nun publiziert werden, indem eine `BubbleCast`-Nachricht mit gesetztem Replikat-Identifikator verschickt wird. Empfänger der Nachricht nehmen das Replikat dann in ihre lokale Replikatliste auf. Hat ein beitretender Knoten $J$ das Replikationsprotokoll erfolgreich beendet, so werden alle neu hinzugewonnenen Replikate ebenfalls über den BubbleStorm-Kern als lokal erzeugte `BubbleCast`-Nachricht empfangen. Dadurch ist der darunter liegende Prozess für den Anwender transparent, es gibt nur *eine* wohldefinierte Schnittstelle, über die Replikate empfangen werden können. Dieser Sachverhalt ist in Abbildung 5.5 graphisch dargestellt.

Alle weiteren Details im Kontext der Publikation beziehen sich auf die Anwenderschnittstelle und sind im gleichnamigen Abschnitt auf Seite 33 zu finden.

### 5.1.4 Konfiguration

In dieser Implementierung wurden die in 4 nicht weiter definierten Konstanten und Verhaltensweisen wie folgt gewählt. Diese Werte sind zur Kompilierzeit beliebig in der Klasse `Constants` anpassbar, die mit ‡ (†) markierten Einträge müssen (sollten) jedoch für alle Knoten in einem Netzwerk gleich gewählt werden.

`replicaIdentifierBytes` ‡ Gibt die Größe des Replikat-Identifikator-Raums in Bytes an.

`randomWalkLength` † Lebensdauer von `RandomCast`-Nachrichten.

`acquisitionBatchSize` Anzahl der gleichzeitig zu kontaktierenden Knoten in Protokollphase 1 (Zielsuche).

`acquisitionBatchInterval` Pause zwischen zwei Anfragen in Protokollphase 1 (Zielsuche).

`enablePersistency` Wahrheitswert, der die Persistenzfunktionen an- oder abschaltet.

`replicaStoragePath` Pfad zu einem Verzeichnis, in dem persistent gespeicherte Replikat-Identifikatoren abgelegt werden.

`replicaConnectionIdentifier` ‡ Identifikator der Benutzerverbindung für das Replikationsprotokoll.

`initialReadBufferSize` Startwert für den Lese-Puffer bei Verbindungen des Replikationsprotokolls.

## 5.2 Anwenderschnittstelle

Die für den Anwender relevante Schnittstelle ist eine Obermenge der regulären BubbleStorm-Schnittstelle. Konkret sind folgende Interfaces bzw. Klassen von Bedeutung:

`Replica` Diese Klasse speichert den Replikat-Identifikator $(r, t)$ und den ursprünglichen Publikationszeitpunkt $\tau_{r,t}$. Der Replikat-Name $r$ wird mit einem `BigInteger` kodiert.

`ReplicaDataProvider` Implementierungen dieses Interfaces müssen die Nutzdaten zu Replikat-Identifikatoren liefern, sobald der Replikationsmechanismus sie anfordert. Diese Notwendigkeit ergibt sich daraus, dass die Replikationsschicht zunächst nur mit Identifikatoren arbeitet. Eine Test-Implementierung wird durch `DummyDataProvider` zur Verfügung gestellt.

`IRouterService` Das BubbleStorm-Hauptinterface zur Knotenkontrolle besitzt die neue Methode `initiateReplicaAcquisition()`, mit welcher der Replikationsalgorithmus gestartet wird. Typischerweise wird sie nach dem Netzwerkbeitritt aufgerufen.

`IBSTypeHandler` Dieses (aus BubbleStorm bekannte) Interface besitzt eine neue Methode `replicationEnabled()`, deren Implementierung zurückgeben sollte, ob der entsprechende Bubble-Typ unter Replikationskontrolle gesetzt werden soll.

`IBubbleCastMessageType` Ebenfalls aus BubbleStorm bekannt sollte eine Implementierung dieses Interfaces den zur Nachricht gehörenden Replikationsidentifikator mit `getReplicaID()` zurückgeben. Falls ein Bubble-Typ nicht zur Replikation vorgesehen ist, kann `null` zurückgegeben werden.

Zur Konfiguration bzw. zum Experimentieren ist weiterhin die Klasse `Constants` zu erwähnen, in der die in Abschnitt 5.1.4 genannten Einstellungen vorgenommen werden können.

### 5.2.1 Anwendungsbeispiel

Dieser Abschnitt konkretisiert ein minimales Anwendungsbeispiel, das den Replikationsmechanismus verwendet. Die notwendigen Schritte lassen sich in Publikation und Replikation aufteilen. Während es bei der Publikation darum geht, aktiv ein neues zu replizierendes Datum erstmalig im Netzwerk zu verbreiten, sind bei der Replikation nur passive Behandlungen der zu replizierten Daten notwendig.

#### 5.2.1.1 Publikation

Zunächst muss ein Bubble-Typ-Handler unter Replikationskontrolle gesetzt werden, damit der Replikationsmechanismus entsprechende Typen zur Verbreitung

berücksichtigen kann.

Listing 5.1: Ein Bubble-Typ wird zur Replikation markiert.

```
1 public class MyTypeHandler implements IBSTypeHandler {
2   [...]
3   public boolean replicationEnabled() {
4     return true;
5   }
6   [...]
7 }
```

Nachrichten, die über diese Bubble geschickt und repliziert werden sollen, müssen einen Identifikator zurückgeben. Der Identifikator direkt kann als Instanzvariable innerhalb des Typs gespeichert werden.

Listing 5.2: Eine Nachricht mit gesetztem Replikationsidentifikator.

```
1 public class MyTypeMessage implements IBubbleCastMessageType
      {
2   private Replica id;
3   [...]
4   public Replica getReplicaID() {
5     return id;
6   }
7   [...]
8 }
```

Um jetzt Nachrichten zu publizieren, wird die oben erstellte Nachricht nun als BubbleCast versendet.

Listing 5.3: Versenden (Publikation) eines Replikats.

```
1 IRouterService router;
2 [...]
3 router.bubblecast(new MyTypeMessage(id, data);
```

#### 5.2.1.2 Replikation

Um Replikate zu erhalten, ruft ein Knoten nach dem vollständigen Netzwerkbeitritt auf Topologie-Ebene den Replikationsalgorithmus auf.

Listing 5.4: Starten des Replikationsmechanismus.

```
1 IRouterService router;
2 [...]
3 router.initiateReplicaAcquisition(new DummyDataProvider());
```

Die so gewonnenen Replikate werden dann über den Bubble-Handler aus Listing 5.1 empfangen. Alle empfangenen Replikate befinden sich automatisch ebenfalls unter Replikationskontrolle, der Anwender muss nur noch die dazugehörigen Daten speichern.

Listing 5.5: Verarbeitung publizierter Replikate.

```
1 public class MyTypeHandler implements IBSTypeHandler {
2   [...]
3   void processMessage([...], byte[] payload, Replica id,
       [...]) {
4     // save replica data payload for id.
5   }
6   [...]
7 }
```

# Kapitel 6

# Evaluation

In diesem Kapitel soll der Replikationsmechanismus praktisch erprobt und evaluiert werden. Als Grundlage dient dazu die in Kapitel 5 vorgestellte Implementierung.

## 6.1 Evaluationsaufbau und -umgebung

Das Design des Prototyps erlaubt es, mehrere Knoten innerhalb einer Programminstanz zu starten. Jedem Knoten wird dabei ein Port zugewiesen, über den der gesamte Netzwerkverkehr abgewickelt wird. Diese Architektur macht die Netzwerkschicht vollständig transparent, d.h. der Anwender kann nicht zwischen einem tatsächlich verteilten Netzwerk und einem rein lokalen Netzwerk unterscheiden. Dadurch ist eine adäquate Evaluation bereits vollständig lokal möglich, was sich die hier präsentierte Evaluation zunutze macht.

In den nächsten zwei Abschnitten werden Metriken und Szenarien beschrieben, die als Dimensionen der Evaluation verstanden werden können. Dabei soll jedes Szenario simuliert und alle Metriken erhoben werden.

### 6.1.1 Metriken

Metriken meint die verschiedenen zu sammelnden Evaluationsdaten. Diese werden als Teil der Evaluation direkt oder indirekt ausgegeben; insbesondere werden folgende Daten erfasst:

- Durchschnittliche Anzahl an Replikaten über alle Dokumente und Standardabweichung davon als Funktion der Zeit.
- Dokumente/Replikate Verteilung zu ausgewählten Zeitpunkten.
- Netzwerkverkehr und -belastung als Funktion der Zeit.
  - Durch `ReplicaAcquisition`: Summe der Nachrichtengrößen.
  - Durch `RandomCast`: Summe der Nachrichtengrößen.
  - Durch den Replikationsdialog: Summe des Gesamtverkehrs.
- Dauer, bis ein beitretender Knoten das Replikationsprotokoll erfolgreich beendet hat.
- Anzahl der Knoten im Netzwerk, sofern nicht konstant.
- Stichprobenartige Erhebung von CPU und Speicherauslastung.

### 6.1.2 Szenarien

Die verschiedenen Szenarien unterscheiden sich durch das globale Verhalten des Netzwerks, insbesondere die Knotenzusammensetzung und deren Dynamik. Vor dem spezifischen Szenario wird zuerst ein Netzwerk aufgebaut, bis die initial gewünschte Größe erreicht wird. Sobald sich dieses Netzwerk stabilisiert hat, d.h. alle Knoten über korrekte Messwerte verfügen, werden Replikate für 500 verschiedene Dokumente in korrekter Anzahl verteilt als wären sie publiziert worden. Dieser Zustand ist die Ausgangssituation für alle Szenarien. Zu beachten ist, dass bei dieser Evaluation Replikate ohne Nutzdaten verwendet wurden, d.h. die Netzwerkverkehr-Volumina und Raten sind jeweils als untere Schranken zu verstehen.

Beim Netzwerkbetritt führt ein beitretender Knoten jeweils den Replikationsmechanismus aus. Es werden 500 Knoten simuliert, alle 5 Sekunden ändert sich die Netzwerkzusammensetzung gemäß des entsprechenden Szenarios. Soweit nicht anderweitig vermerkt, ist der Knotengrad über alle Knoten konstant.

**Birth/Death (Churn)** In jedem Schritt verlässt ein zufälliger Knoten das Netzwerk und ein anderer tritt bei. Die Laufzeit dieses Szenarios ist so zu wählen, dass sich eine erkennbare Konvergenz einstellt.

**Zunahme** Das Netzwerk startet mit 100 statt 500 Knoten. Die Netzwerkgröße wird pro Schritt um eins erhöht, indem ein neuer Knoten beitritt. Mit einer Wahrscheinlichkeit von 50% tritt ein weiterer Knoten bei und ein anderer verlässt das Netzwerk. Hat das Netzwerk eine Größe von 500 Knoten erreicht, folgt anschließend eine Birth/Death-Phase.

**Abnahme** Analog zum Zunahme-Szenario, die Netzwerkgröße verringert sich um eins. Mit einer Wahrscheinlichkeit von 50% verlässt ein weiterer Knoten das Netzwerk und ein anderer tritt bei. Sobald sich die Größe auf 100 Knoten verringert hat, folgt eine Birth/Death-Phase.

**Birth/Death heterogen** Dieses Szenario entspricht zunächst dem Birth/Death-Szenario. Beitretende Knoten haben jedoch nicht alle den gleichen Knotengrad, sondern weisen die folgende Verteilung auf.

| Knotengrad | 4 | 6 | 10 | 20 |
|---|---|---|---|---|
| Wahrscheinlichkeit | .55 | .3 | .1 | .05 |

Der durchschnittliche Knotengrad dieser Verteilung entspricht den konstanten Knotengraden des Anfangsnetzwerks ($\bar{d} = d = 6$).

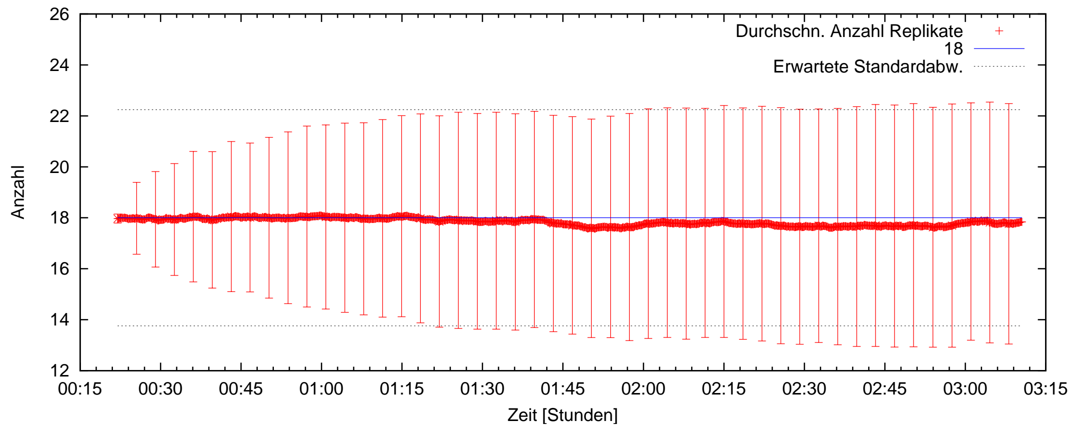
## 6.2 Evaluationsergebnisse

Die hier präsentierten Evaluationsergebnisse wurden mit handelsüblicher Hardware[1] gewonnen. CPU- und Speicherauslastung lagen dabei selten über 25% (CPU) respektive 10% (Speicher). Pro Szenario wurden Protokolldateien von 200 bis 500 MiB erzeugt, aus denen dann mittels eines Python-Skripts die relevanten Daten extrahiert und aggregiert wurden.

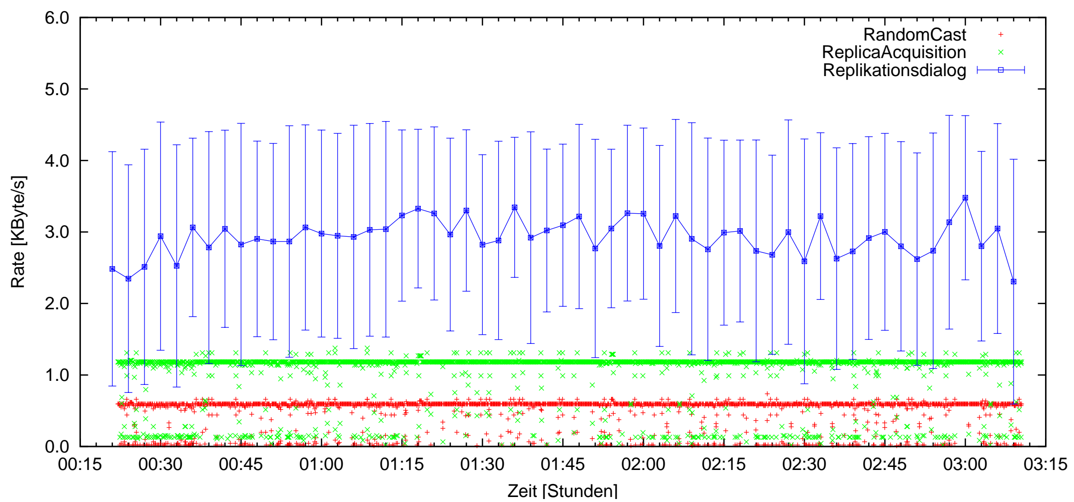
### 6.2.1 Birth/Death (Churn)

Abbildung 6.1(a) zeigt die durchschnittliche Replikatanzahl mit Standardabweichung. Die Soll-Anzahl liegt in diesem Szenario bei 18 Replikaten, die im Durchschnitt stabil gehalten wird. Da die Replikate vor Beginn des Szenarios in exakt korrekter Anzahl verteilt werden, liegt die Standardabweichung zunächst bei 0

[1] Das verwendete Testsystem verfügt über eine 2.4 Ghz Quadcore-CPU mit 4 GiB Speicher.




(a) Durchschnittliche Replikatanzahl über alle Dokumente




(b) Netzwerkverkehr

Abbildung 6.1: Birth/Death-Szenario: Replikatanzahl und Netzwerkverkehr

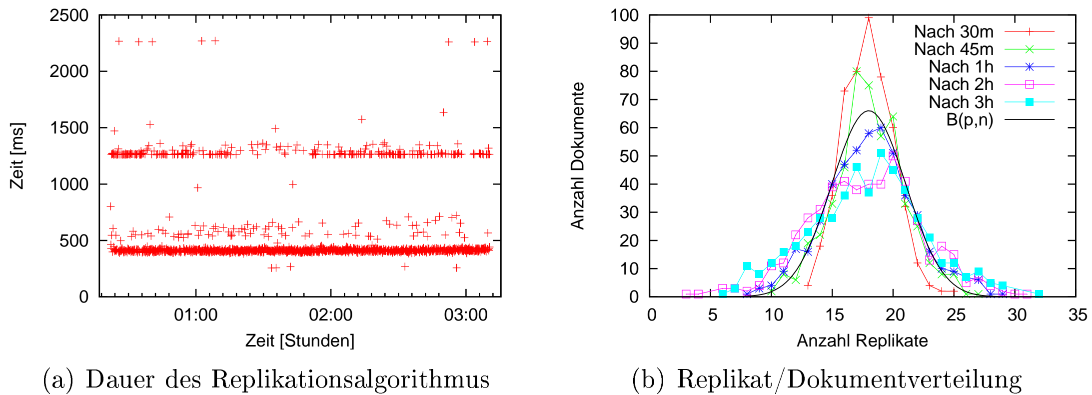
und ist nach ca. einer Stunde weitgehend konvergent. Eine andere Perspektive auf diese Replikatverteilung bietet Abbildung 6.2(b): Hier sind die Replikatverteilungen zu fünf verschiedenen Zeitpunkten dargestellt. Auch hier lässt sich nach einer Stunde eine erste Annäherung in Form einer Binomialverteilung ausmachen, die aufgrund der statistischen Varianz in Verbindung mit der relativ geringen Replikatanzahl jedoch nicht 100%ig erreicht wird.

Bei der Darstellung des Netzwerkverkehr (Abbildung 6.1(b)) ist zwischen den drei verschiedenen Typen von Nachrichten zu unterscheiden. ReplicaAcquisition- und RandomCast-Nachrichten sind stets gleich groß und werden in gleichem Umfang versendet. Daraus resultiert eine größtenteils konstante Datenrate von 600 B/s bzw. 1.2 KiB/s. Der Replikationsdialog hingegen weist mehr Varianz auf. Grund dafür ist die unterschiedliche Anzahl der Replikat-Identifikatoren, die in diesem Protokoll ausgetauscht werden. Durchschnittlich treten hierbei Raten von etwa 3 KiB/s auf.

Abbildung 6.2(a) zeigt schließlich die Zeiten, die Knoten für die Abwicklung des gesamten Replikationsvorgangs benötigen. Auffällig sind dabei extreme Verdichtungen bei etwa 400 ms und 1300ms. Diese sind auf die Herangehensweise der Knotensuche bei der Replikation zurückzuführen. Während viele Knoten bereits nach dem Adressieren des ersten Teils einer Bubble genügend Zielknoten gefunden haben, benötigen andere zwei oder sogar drei Schübe. Diese Unterschiede treten vor allem dann auf, wenn ein Knoten teilweise mehrfach adressiert wurde, was bei einem relativen kleinen Netzwerk von 500 Knoten mit einem probabilistischen Algorithmus nicht vermeidbar ist.

### 6.2.2 Zunahme-Szenario

Die Soll-Replikatanzahl $pn$ in diesem Szenario liegt initial bei 8 und nach der Zunahme-Phase bei 18 Replikaten. Während der Zunahme innerhalb der ersten Stunde ist in Abbildung 6.3(a) deutlich erkennbar, dass die Replikatanzahl zunächst stark ansteigt und schließlich ihren Höhepunkt bei durchschnittlich 24 Replikaten erreicht. Aufgrund der geringen Churn-Itensität während des Anstiegs ist die Verlustwahrscheinlichkeit effektiv herabgesetzt und stellt kein hinreichendes Gegengewicht zur Replikation dar. Die Folge ist eine stärkere Replikation als beabsichtigt. Sobald die Zunahme-Phase beendet ist, setzt der Birth/Death-Prozess




(a) Dauer des Replikationsalgorithmus

(b) Replikat/Dokumentverteilung

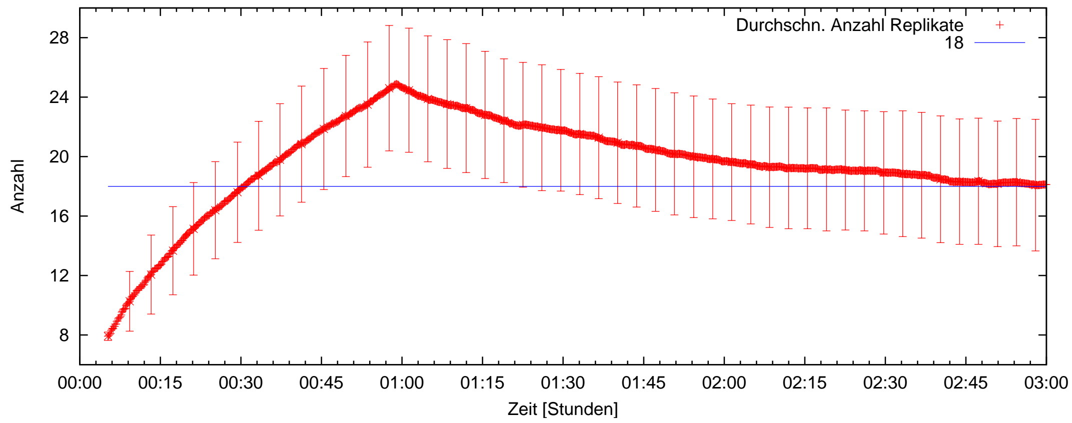
Abbildung 6.2: Birth/Death-Szenario: Dauer und Replikatverteilung

ein, wodurch sich die Replikatanzahl wieder auf die gewünschte Anzahl von 18 stabilisiert.

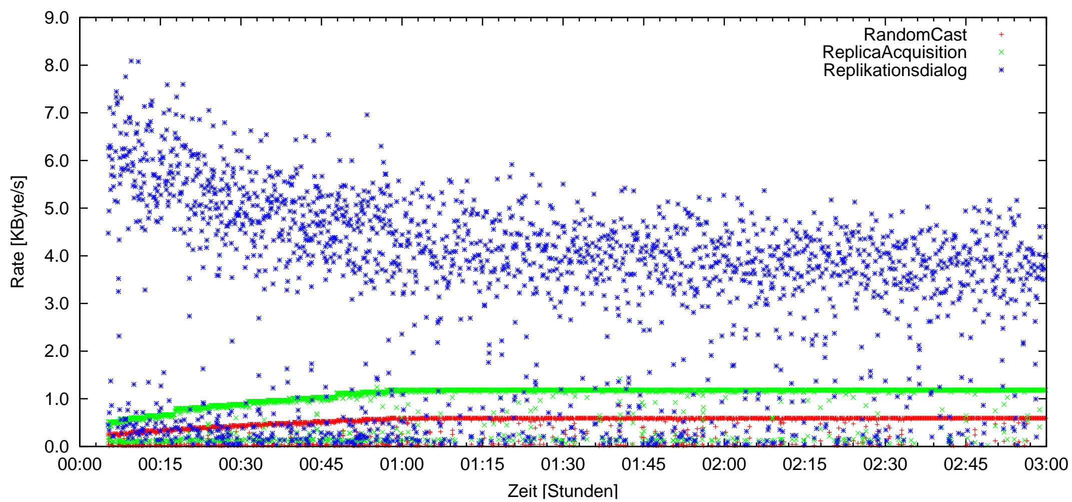
Ein Blick auf den Netzwerkverkehr (Abbildung 6.3(b)) zeigt für Replica-Acqusition- und RandomCast-Nachrichten einen erwartungsgemäß proportionalen Anstieg der Raten. Je mehr Knoten im Netzwerk, desto mehr Knoten werden auch durch den Replikationsalgorithmus adressiert. Der Replikationsdialog hingegen zeigt anfangs höhere Raten auf. Durch den erhöhten Replikationsbedarf während der Zunahme müssen mehr Replikat-Identifikatoren ausgetauscht werden. Nach einer Stunde pendeln sich die Raten dann auf jene Werte ein, die auch im regulären Birth/Death-Fall zu erkennen sind.

### 6.2.3 Abnahme-Szenario

Bei diesem Szenario (Abbildung 6.4) sinkt die Replikatanzahl proportional zur Netzwerkgröße (d.h. linear), was wegen der geringen Churn-Itensität nicht durch Replikation ausgeglichen werden kann. Da die gewünschte Replikatanzahl als Funktion der Netzwerkgröße nicht linear ist, fällt die Replikatanzahl unter den angestrebten Wert von 8. In der anschließenden Birth/Death-Phase wird dieser jedoch nicht mehr mit der Genauigkeit des vorherigen Szenarios erreicht. Im Gegensatz zu allen anderen Szenarien sind hier auch Dokumente verloren gegangen, d.h. es befindet sich weniger als ein Replikat für ein Dokument im Netzwerk. Dies

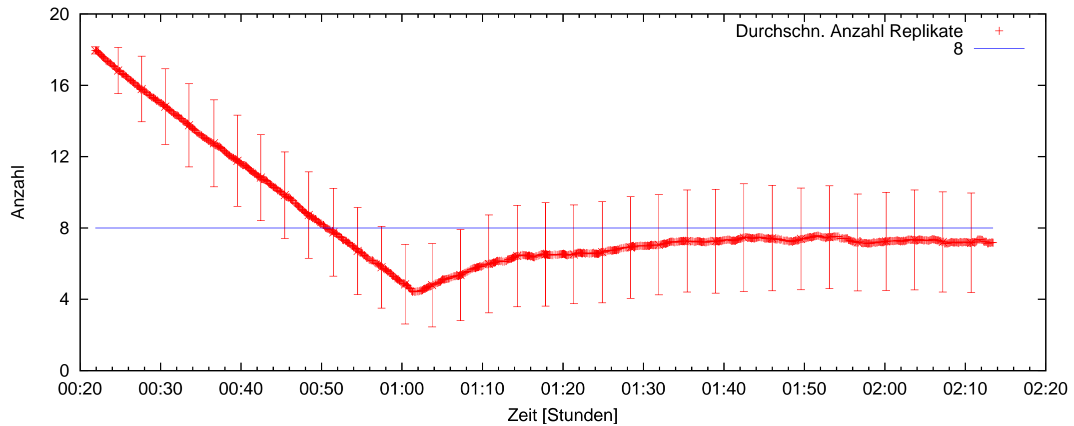



(a) Durchschnittliche Replikatanzahl über alle Dokumente

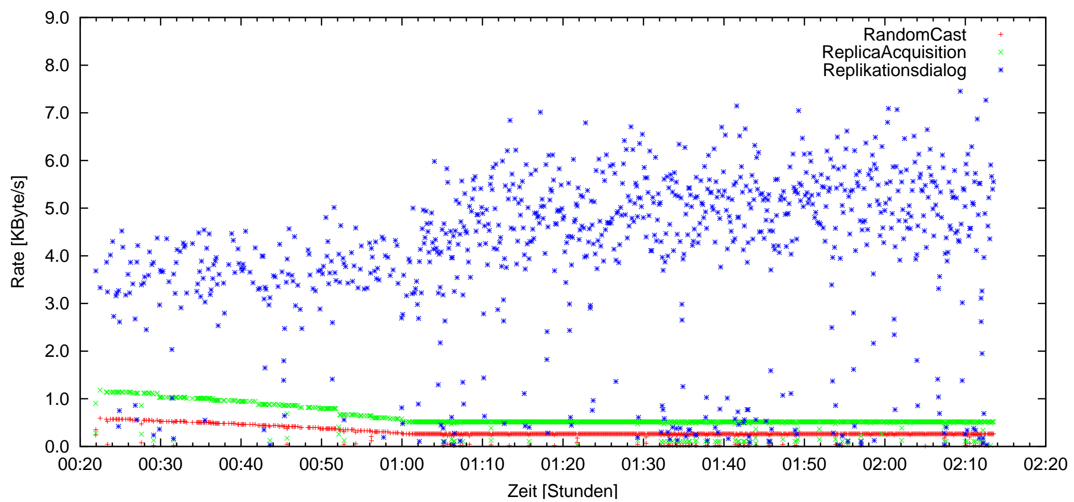



(b) Netzwerkverkehr

Abbildung 6.3: Zunahme-Szenario: Replikatanzahl und Netzwerkverkehr




(a) Durchschnittliche Replikatanzahl über alle Dokumente




(b) Netzwerkverkehr

Abbildung 6.4: Abnahme-Szenario: Replikatanzahl und Netzwerkverkehr

lässt sich durch die relativ kleine Anzahl von 100 Knoten erklären, bei der sich die statistische Varianz des Vorgangs bemerkbar macht. Im Fall des Dokumentverlusts ist außerdem zu beachten, dass die durchschnittliche Anzahl kurzzeitig bei nur 4 Replikaten lag, was den Effekt noch weiter verstärkt.

### 6.2.4 Heterogenes Szenario

Das initiale Netzwerk in diesem Szenario entspricht dem des regulären Birth/Death-Szenarios, alle Knoten haben also den gleichen Knotengrad. Für beitretende Knoten wird jedoch ein Grad gemäß der Verteilung aus Abschnitt 6.1.2 gewählt. Die durchschnittliche Replikatanzahl ist in Abbildung 6.5 visualisiert. Hier ist bemerkenswert, dass die Anzahl etwa eine Stunde lang sehr stabil bei 18 Replikaten liegt, aber dann deutlich zunimmt (20% Abweichung).

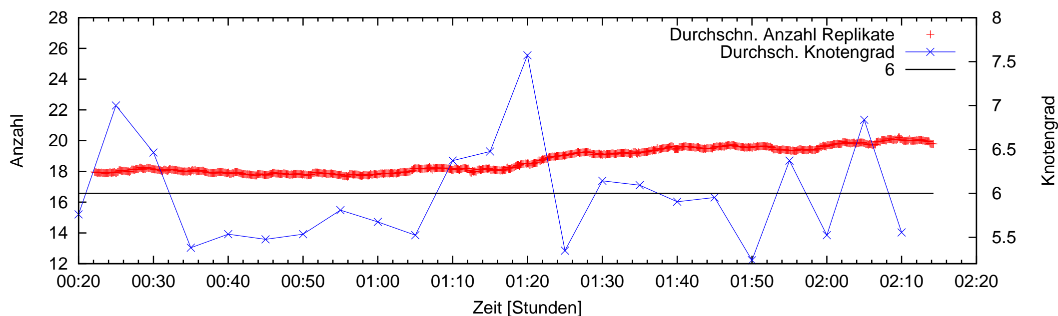
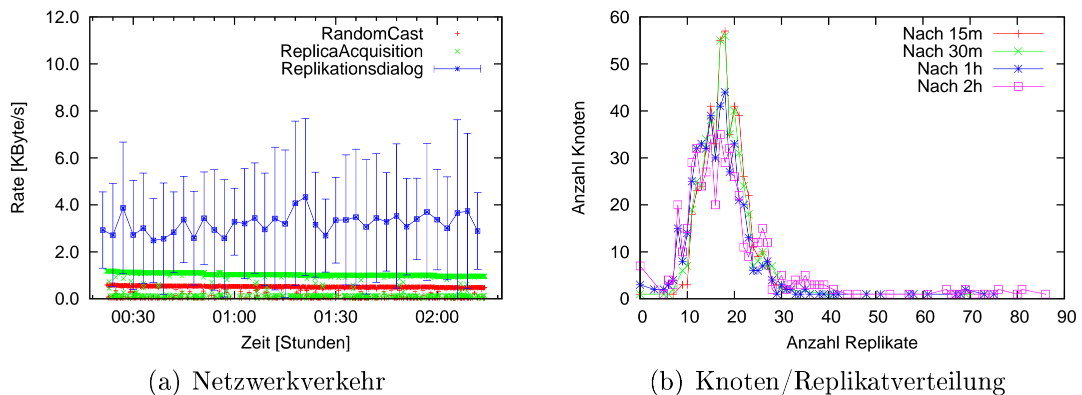



Abbildung 6.5: Heterogenes Szenario: Replikatanzahl

Bei der Suche nach der Ursache für dieses Verhalten war insbesondere der durchschnittliche Knotengrad beitretender Knoten auffällig, der ebenfalls in Abbildung 6.5 dargestellt ist. Dieser sollte bei 6 liegen, wird jedoch bei 01:20, 01:55 sowie 02:05 deutlich überschritten. Der Beitritt mehrerer Knoten mit Grad 20 führt hier zu massiver Replikation und sorgt somit für einen Anstieg der durchschnittlichen Replikatanzahl. Daraus kann gefolgert werden, dass der Algorithmus empfindlich für solche Ereignisse ist bzw. die Wahl der Knotenverteilung für diese Netzwerkgröße zu drastisch war[2].

[2] Die Replikationswahrscheinlichkeit für einen Knoten mit dem Grad 20 wird um den Faktor $\frac{20}{6} \approx 3$ skaliert.




(a) Netzwerkverkehr

(b) Knoten/Replikatverteilung

Abbildung 6.6: Heterogenes Szenario: Netzwerkverkehr und Replikatverteilung

Die Darstellung des Netzwerkverkehrs in Abbildung 6.6(a) weist wie erwartet eine höhere Varianz im Replikationsdialog auf, da durch die unterschiedlichen Knotengrade unterschiedlich viele Replikat-Identifikatoren ausgetauscht werden. Die ReplicaAcquisition- und RandomCast-Nachrichten sind dagegen weiterhin relativ konstant. Abbildung 6.6(b) zeigt schließlich Verteilungen über die Anzahl der Replikate pro Knoten, worin sich die Auswirkungen der unterschiedlichen Knotengrade wiederfinden.

### 6.2.5 Fazit

Zusammenfassend ist in allen Szenarien zu erkennen, dass die angestrebte Stabilisierung der Replikatanzahl erreicht wird. Während es in Phasen der überwiegenden Zu- oder Abnahme zu temporären Abweichungen kommt, können diese durch Perioden stabiler Netzwerkgröße und vorherrschendem Churn ausgeglichen werden. Die leichte Überbevölkerung im heterogenen Fall ist unter Zuhilfenahme von weiteren Protokolldaten erklärbar und ermöglicht somit die Berücksichtigung für zukünftige Verbesserungen.

# Kapitel 7

# Zusammenfassung und Ausblick

In dieser Arbeit wurde ein probabilistischer Replikationsmechanismus für BubbleStorm als unstrukturiertes Peer-to-Peer Netzwerk vorgestellt. Die Ziele waren dabei, dass der Algorithmus besitzerlos und dezentral arbeitet und dabei die (Leistungs-)Heterogenität des Netzwerks berücksichtigt. Die dazugehörige Implementierung in einem BubbleStorm-Prototyp stellt die praktische Realisierung dieses Mechanismus dar, die schließlich erfolgreich einer grundlegenden Evaluation unterzogen wurde.

Elemente, die aus verschiedensten Gründen keinen Einzug in diese Arbeit erhalten haben oder sollten, sind in den nächsten beiden Abschnitten aufgezählt.

## 7.1 Kritik

Beim Protokolldesign bzw. den einzelnen Nachrichten wurden manche Felder sehr großzügig bemessen. Für ein produktives Szenario wäre es sinnvoll, einige Felder zu verkleinern. Beispielsweise ist ein Knotengrad der Größenordnung $2^{31} - 1$ sehr weit von einer vernünftigen Abschätzung entfernt. Des weiteren wurde die Replikatpublikation direkt in BubbleCast-Nachrichten integriert, was zu einem Overhead von etwa 3% hinsichtlich der Größe führt, auch wenn das Replikationssystem keine Verwendung findet. Eine prinzipielle Design-Alternative besteht in einem seperaten Nachrichten-Typ anstelle der Erweiterung der regulären BubbleCast-Nachricht, die den Overhead vermeidet. In einer realistischen Umgebung ist dieser jedoch vernachlässigbar, da die Nutzdaten ohnehin meist wesentlich größer als die Header-Felder sind.

Bei der Evaluation mit einem heterogenen Netzwerk zeigte sich der Replikationsalgorithmus empfindlich gegenüber starken Knotengradschwankungen. Hier wäre eine andere Form der Beeinflussung der Replikationswahrscheinlichkeit durch den Knotengrad zu erforschen, wie beispielsweise eine exponentielle Abschwächung statt einem linearen Ansatz.

## 7.2 Ausblick

Der BubbleStorm-Prototyp unterstützt beim Routen von Topologie-Nachrichten das Priorisieren von Nachrichten. Dieses Feature wurde im Zuge der Implementierung des Replikationssystems ignoriert, wodurch die dafür relevanten Nachrichten immer und sofort gesendet werden. Dieses Verhalten sabotiert offensichtlich die vorhandene Nachrichtenpriorisierung, jedoch würde eine Implementierung vor allem aufgrund der theoretischen und analytischen Überlegungen dazu den Rahmen dieser Arbeit sprengen.

In einer realistischen Einsatzumgebung besteht zweifelsohne der Wunsch, dass ein gegebenes Replikat ab einem bestimmten Zeitpunkt nicht mehr benötigt wird und gelöscht werden kann. Da der Algorithmus per Design jedoch keinen Besitzer für ein Replikat kennt und die replizierenden Peers weitgehend unbekannt sind, erfordert eine solche Funktionalität weitere Überlegungen.

Schließlich ist die Flexibilität des Replikationsalgorithmus zu erwähnen. Obwohl dieser speziell für BubbleStorm entworfen wurde, könnte er prinzipiell überall dort Verwendung finden, wo eine Messung der Netzwerkgröße möglich ist. Eine Integration in andere Systeme ist daher nicht ausgeschlossen.

# Anhang A

# Formatbeschreibungen

Falls nicht explizit anders erwähnt, gelten folgende Spezifikationen.

- Ganzzahlen sind im Big-Endian-Format.
- Offsets und Längen sind in Byte angegeben.
- Mit $k$ wird die Länge des Replikat-Identifikators bezeichnet. Diese Belegung ist im gesamten Netzwerk fest aber beliebig.

## Nachrichten

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Nachrichten-Typ, konstant `0x5E000090` |
| 4 | 4 | IPv4 Quelladresse, 32 Bit binärkodiert, Big Endian |
| 8 | 2 | IPv4 Port, 16 Bit binärkodiert, Big Endian |
| 10 | 4 | Größe der Bubble, 32 Bit Ganzzahl |
| 14 | 4 | Start der Bubble, 32 Bit Ganzzahl |
| 18 | 4 | Ende der Bubble, 32 Bit Ganzzahl |

Tabelle A.1: Format der `ReplicaAcquisition`-Nachricht.

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Nachrichten-Typ, konstant `0x5E000089` |
| 4 | 4 | IPv4 Quelladresse, 32 Bit binärkodiert, Big Endian |
| 8 | 2 | IPv4 Port, 16 Bit binärkodiert, Big Endian |
| 10 | 1 | Time-to-Live, 8 Bit Ganzzahl |

Tabelle A.2: Format der `RandomCast`-Nachricht.

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Bubble-Typ, 32 Bit Ganzzahl |
| 4 | 4 | Zeitstempel, 32 Bit Ganzzahl |
| 8 | $k$ | Replikat-Name, $8k$ Bit Ganzzahl |

Tabelle A.3: `Replica` mit Zeitstempel ($k$ beliebig aber fest).

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Bubble-Typ, 32 Bit Ganzzahl |
| 4 | $k$ | Replikat-Name, $8k$ Bit Ganzzahl |

Tabelle A.4: `Replica` ohne Zeitstempel ($k$ beliebig aber fest).

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Letzter Netzwerkaustritt, 32 Bit Ganzzahl |
| 4 | 8 | Zufallszahl, 64 Bit Ganzzahl |
| 12 | 4 | Angestrebter Knotengrad, 32 Bit Ganzzahl |
| 16 | 4 | Transaktionsidentifikator, 32 Bit Ganzzahl |
| 20 | 4 | Anzahl ($n$) Replikationsfaktoren, 32 Bit Ganzzahl |
| $24 + 8i$ | 4 | Replikat-Typ $i < n$, 32 Bit Ganzzahl |
| $28 + 8i$ | 4 | Repl.-Faktor $i < n$, 32 Bit IEEE 754 Gleitkommazahl |

Tabelle A.5: Format der `AcquisitionHello`-Nachricht

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Anzahl ($n$) Replikat-Ident. |
| $4 + i(4 + k)$ | $4 + k$ | Replikat-Ident. (siehe Tabelle A.4), $i < n$ |

Tabelle A.6: Format der `AcquisitionList`- und `AcquisitionRequest`-Nachricht.

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Anzahl ($n$) Replikate |
| 4 | * | Datenelemente, siehe Tabelle A.8 |

Tabelle A.7: Format der `AcquisitionData`-Nachricht.

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | $k+8$ | Replikat-Ident. mit Zeitstempel (siehe Tabelle A.3) |
| $k+8$ | 4 | Datenlänge $l$, 32 Bit Ganzzahl |
| $k+12$ | $l$ | Binärdaten des Replikats |

Tabelle A.8: Teilformat der `AcquisitionData`-Nachricht, siehe Tabelle A.7.

| Offset | Länge | Beschreibung |
|---|---|---|
| 0 | 4 | Nachrichten-Typ, konstant `0x5E000082` |
| 4 | 4 | Gesamtlänge der Nachricht, 32 Bit Ganzzahl |
| 8 | 4 | IPv4 Quelladresse, 32 Bit binärkodiert, Big Endian |
| 12 | 2 | IPv4 Port, 16 Bit binärkodiert, Big Endian |
| 14 | 4 | Größe der Bubble, 32 Bit Ganzzahl |
| 18 | 4 | Start der Bubble, 32 Bit Ganzzahl |
| 22 | 4 | Ende der Bubble, 32 Bit Ganzzahl |
| 26 | 4 | Bubble-Typ, 32 Bit Ganzzahl |
| 30 | 1 | Präsenzindikator für Repl.-Daten. Hier: 1 |
| 31 | 4 | Ursprünglicher Replikationszeitpunkt, 32 Bit Ganzzahl |
| 35 | $k$ | Replikat-Name, $k$ beliebig aber fest |

Tabelle A.9: Format der `BubbleCast`-Nachricht mit Replikat-Informationen.

# Literaturverzeichnis


[1] F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping and Random Replication to Support Reliable Peer-to-Peer Content Search and Retrieval. Technical Report DCS-TR-494, Department of Computer Science, Rutgers University, July 2002.

[2] F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Autonomous Replication for High Availability in Unstructured P2P Systems. In *The 22nd IEEE Symposium on Reliable Distributed Systems (SRDS-22)*. IEEE Press, Oct. 2003.

[3] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Infrastructure support for P2P information sharing. Technical Report DCS-TR-465, Department of Computer Science, Rutgers University, Nov. 2001.

[4] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *In the proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS2003 (to appear)*, 2003.

[5] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *FOCS*, pages 482–491, 2003.

[6] P. Knežević. *High Data Availability and Consistency for Distributed Hash Tables Deployed in Dynamic Peer-to-peer Communities.* PhD thesis, Technische Universität Darmstadt, 2007.

[7] C. LV, P. CAO, E. COHEN, K. LI, and S. SHENKER. Search and replication in unstructured peer-to-peer networks, 2001.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the*

*2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[10] 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, New York, NY, USA, Aug. 2007. ACM Press.

[11] W. W. Terpstra, C. Leng, and A. P. Buchmann. Brief announcement: Practical summation via gossip. In *Twenty-Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2007)*, pages 390–391, New York, NY, USA, Aug. 2007. ACM Press.

[12] 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.

[13] Z. Wang, S. K. Das, M. Kumar, and H. Shen. Update propagation through replica chain in decentralized and unstructured p2p systems.

[14] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment, 2003.