Peer-to-Peer
Replikatverwaltung in
BubbleStorm

Bachelorarbeit von
Paul Bächler

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

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
## Inhaltsverzeichnis

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.

I. Einleitung


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


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-


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 ausgefeilte 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 heterogene verteilte Rechenleistung sowie Totalausfälle, die erst spät bemerkt werden können. Das verdeutlicht, dass gut erforschte Algorithmen und Strategien aus (lokal) verteilten Systemen zwar prinzipiell verwendet werden können, jedoch unter den genannten Umständen genau evaluiert und entsprechend angepasst oder sogar verworfen werden müssen.


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


### 2.2.2 Veränderliche und versionierte Replikate


---

Paul Bächer
II. Verwandte Arbeiten

2.2.3 Bisherige Ansätze in Peer-to-Peer Umgebungen


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 Gesamtverkehrs, ist die „Square-Root“-Verteilung, für die zwei verschiedene Varianten vorgestellt werden.


3. Eine zweite Variante vermeidet solche Verdichtungen, indem nur die Pfadlänge k ermittelt und anschließend auf k zufällige Peers repliziert wird.

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, detailliertere Beschreibungen, eine Simulation sowie analytische Ergebnisse finden sich in [12; 10].


Dadurch, dass Anfragen bei jedem Peer lokal verarbeitet werden und mit hoher Wahrcheinlichkeit auf dem gesamten Datensatz operieren, sind sehr vielfältige und komplexe Suchanfragen möglich. Zudem ist die Gestalt solcher Abfragen vollständig der Anwendung, die auf BubbleStorm 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)


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 einmal 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. Auch hierbei bleibt der Zyklus erhalten und die Grade anderer Knoten konstant.

3.1.2 Client-Modus


3.2 Bubbles und BubbleCast

Benutznachrichten 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.


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 parametriert werden können und somit unterschiedliche Größen aufweisen können.

3.3 Messung


\[ D_i := \sum_{v \in V} \deg(v)^i \quad i \in \{0, 1, 2\} \]

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*

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

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 Wiederbeistritt 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


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.

\(^1\)
IV. Ansatz

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


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.
IV. Ansatz

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

Der tatsächliche Replikationsfaktor sei mit $\hat{p}$ 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 $\hat{p}$ verloren. Für eine variierende Anzahl an tatsächlichen Replikaten und dem dazugehörigen Replikationsfaktor $\hat{p}$ lassen sich dann die nachfolgenden Fälle unterscheiden.

(i) $p = \hat{p}$ Der tatsächliche Replikationsfaktor entspricht dem angestrebten Replikationsfaktor. Da Replikation und Verlust gleich wahrscheinlich sind, bleibt die Replikatanzahl konstant.

(ii) $p > \hat{p}$ Es sind zu wenige Replikate im Netzwerk vorhanden. Da Replikation wahrscheinlicher als Verlust ist, steigt die Replikatanzahl.

(iii) $p < \hat{p}$ 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 $p \neq \hat{p}$ 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 $\hat{p}$ 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 $\hat{p}$ besitzen, genau $y$ Knoten tatsächlich über das Replikat verfügen ist
IV. Ansatz

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

mit \( y = 1 \) folgt

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

Frage man nun \( m := 1/p \) Knoten ab, so erhält man mit der Wahrscheinlichkeit

\[ \frac{\tilde{p}}{p} (1 - \tilde{p})^{1/p-1} \] (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

\[ \frac{p}{(1 - p)^{1/p-1}} \] (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 \( K_i \) als auch \( J \) die Replikatmenge filtern. Dabei wird die zu taschende 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 Replicationswahrscheinlichkeit. 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 Replicationswahrscheinlichkeit 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 Wiederbeimitt 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 Replicationswahrscheinlichkeit mit der Differenz des alten und neuen Replikationsfaktors gewichtet. Ist der neue Replikationsfaktor kleiner als der alte, so werden persistente 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 Grades $d$ 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 Replicationswahrscheinlichkeit $\rho$ sowie für das Löschen überflüssig gewordener lokaler Replikate im Falle eines niedrigeren Replikationsfaktors nach dem Wiederbeimitt.
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 Netzwerkaustritts, \( t_0 = 0 \) 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 \texttt{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,

\[
f := \frac{1}{(1 - p_t)^{1/p_t - 1}} \cdot \frac{d D_0}{D_1}
\]

\(^3\)Der von uns gewählte Ansatz wird in Kapitel 5 beschrieben.
sowie

\[
\begin{align*}
  k_0 & := \min(1, fp_t) \\
  k_1 & := \min(1, f(p_t - q_t))
\end{align*}
\]

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 \(a \cdot r_{64} \mod 2^{64} < k\) sendet \(Z_i\) den vollständigen Replikatidentifikator \((r,t)\) an \(Z_i\). Die Bezeichnung dieser Nachricht ist \text{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 lokalen 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 (\text{ACQUISITION\_REQUEST}).

4. \(Z_i\) sendet die von \(J\) angeforderten Replikatdaten \(\delta_{r,t}\) (\text{ACQUISITION\_DATA}).

Falls einer der Protokollschriften 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 \((q_t - p_t) \frac{dp_a}{T_h}\) 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.


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.

Diese Aufteilung wurde gewählt, um in den restlichen BubbleStorm-Kern so minimal wie möglich einzugreifen, damit diese Implementierung austauschbar und vom Replicationssystem 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

Abbildung 5.1: Architekturübersicht des Replicationssubsystems.
sein kann.

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

<table>
<thead>
<tr>
<th>Größe</th>
<th>Bedeutung</th>
<th>Implementierung</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>Replikat-Name</td>
<td>( n )-Bit Ganzzahl (BigInteger).</td>
</tr>
<tr>
<td>( t )</td>
<td>Replikat-Typ</td>
<td>32-Bit Ganzzahl (int).</td>
</tr>
<tr>
<td>( \tau_{r,t} )</td>
<td>Publikationszeitpunkt</td>
<td>32-Bit Ganzzahl (int).</td>
</tr>
<tr>
<td>( \delta_{r,t} )</td>
<td>Replikat-Daten</td>
<td>Nutzdaten, beliebige Länge, byte[]</td>
</tr>
</tbody>
</table>

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 Zᵢ 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.


5.1.2.1 Phase 1 – Zielsuche

Beim 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.

Zumächst 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-
V. Implementierung


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 konkreter 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

2Konkret ist $l$ in dieser Implementierung eine Ganzzahl, die jeweils um eins dekrementiert wird.
das Konzept der Benutzerverbindungen an. Prinziell 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 ressourcen-
sparend, 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 Schablonenmuster
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 anfallenen
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 Protokollschriften in Zusammenhang gebracht werden.
Die Zustände ändern sich jeweils, sobald Daten komplett gesendet oder empfangen
wurden.

Die vier verschiedenen Nachrichten wurden im Paket bubblestorm.replicat-
ion.messages implementiert. Sie stellen jeweils Methoden zur Serialisierung und
Abbildung 5.4: Zustände der beiden Kommunikationspartner im Zusammenhang mit den einzelnen Nachrichten des Protokolls.


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 minimalen Aufwand publizieren zu können. Der von uns gewählte Weg besteht darin, einen regulären BubbleCast zusätzlich mit den Replikationen 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 abgewogen 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.


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 initiaRepliacaAcquisition(), mit welcher der Replikationsalgorithmus gestartet wird. Typischerweise wird sie nach dem Netzwerkbeintritt 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


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.

```java
public class MyTypeHandler implements IBSTypeHandler {
    [...]
    public boolean replicationEnabled() {
        return true;
    }
    [...]
}
```

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.

```java
public class MyTypeMessage implements IBubbleCastMessageType {
    private Replica id;
    [...]
    public Replica getReplicaID() {
        return id;
    }
    [...]
}
```

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

Listing 5.3: Versenden (Publikation) eines Replikats.

```java
IRouterService router;
[...]
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.

```java
IRouterService router;
...
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.

```java
public class MyTypeHandler implements IBSTypeHandler {
  ...
  void processMessage(..., byte[] payload, Replica id,
                        ...) {
    // save replica data payload for id.
  }
  ...
}
```
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 Netzwerkstruktur 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:
VI. Evaluation

- 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 Netzwerkverkehrs-Volumina und Raten sind jeweils als untere Schranken zu verstehen.


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.

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.


<table>
<thead>
<tr>
<th>Knotengrad</th>
<th>4</th>
<th>6</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wahrscheinlichkeit</td>
<td>.55</td>
<td>.3</td>
<td>.1</td>
<td>.05</td>
</tr>
</tbody>
</table>

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.
VI. Evaluation

(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.


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 1300 ms. 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 $p_0$ 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-Intensität 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
VI. Evaluation

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


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-Intensität 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
VI. Evaluation

(a) Durchschnittliche Replikatanzahl über alle Dokumente

(b) Netzwerkverkehr

Abbildung 6.3: Zunahme-Szenario: Replikatanzahl und Netzwerkverkehr
VI. Evaluation

(a) Durchschnittliche Replikanzahl über alle Dokumente

(b) Netzwerkverkehr

Abbildung 6.4: Abnahme-Szenario: Replikanzahl 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 gefolgt werden, dass der Algorithmus empfindlich für solche Ereignisse ist bzw. die Wahl der Knotenverteilung für diese Netzwerkgröße zu drastisch war².

²Die Replikationswahrscheinlichkeit für einen Knoten mit dem Grad 20 wird um den Faktor \( \frac{20}{6} \approx 3 \) skaliert.
Abbildung 6.6: Heterogenes Szenario: Netzwerkverkehr und Replikatverteilung

Die Darstellung des Netzwerkverkehrs in Abbildung 6.6(a) weist wie erwartet eine höhere Varianz im Replicationsdialog 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

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

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


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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Nachrichten-Typ, konstant 0x5E000090</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>IPv4 Quelladresse, 32 Bit binärkodiert, Big EndIan</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>IPv4 Port, 16 Bit binärkodiert, Big Endian</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>Größe der Bubble, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>Start der Bubble, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>Ende der Bubble, 32 Bit Ganzzahl</td>
</tr>
</tbody>
</table>

Tabelle A.1: Format der ReplicaAcquisition-Nachricht.
I. Formatbeschreibungen

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Nachrichten-Typ, konstant 0xE000089</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>IPv4 Quelladresse, 32 Bit binärkodiert, Big Endian</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>IPv4 Port, 16 Bit binärkodiert, Big Endian</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>Time-to-Live, 8 Bit Ganzzahl</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Bubble-Typ, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Zeitstempel, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>8</td>
<td>k</td>
<td>Replikat-Name, 8k Bit Ganzzahl</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Bubble-Typ, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>4</td>
<td>k</td>
<td>Replikat-Name, 8k Bit Ganzzahl</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Letzter Netzwerkaustritt, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>Zufallszahl, 64 Bit Ganzzahl</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>Angestrebter Knotengrad, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Transaktionsidentifikator, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>Anzahl ($n$) Replikationsfaktoren, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>24 + 8i</td>
<td>4</td>
<td>Replikat-Typ $i &lt; n$, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>28 + 8i</td>
<td>4</td>
<td>Repl.-Faktor $i &lt; n$, 32 Bit IEEE 754 Gleitkommazahl</td>
</tr>
</tbody>
</table>

Tabelle A.5: Format der AcquisitionHello-Nachricht.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Anzahl ($n$) Replikat-Ident.</td>
</tr>
<tr>
<td>$4 + i(4 + k)$</td>
<td>$4 + k$</td>
<td>Replikat-Ident. (siehe Tabelle A.4), $i &lt; n$</td>
</tr>
</tbody>
</table>

Tabelle A.6: Format der AcquisitionList- und AcquisitionRequest-Nachricht.
### I. Formatbeschreibungen

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Anzahl ( n ) Replikate</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>Datenelemente, siehe Tabelle A.8</td>
</tr>
</tbody>
</table>

Tabelle A.7: Format der AcquisitionData-Nachricht.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( k + 8 )</td>
<td>Replikat-Ident. mit Zeitstempel (siehe Tabelle A.3)</td>
</tr>
<tr>
<td>( k + 8 )</td>
<td>4</td>
<td>Datenlänge ( l ), 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>( k + 12 )</td>
<td>( l )</td>
<td>Binärdaten des Replikats</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Länge</th>
<th>Beschreibung</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Nachrichten-Typ, konstant ( 0x5E000082 )</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Gesamtlänge der Nachricht, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>IPv4 Quelladresse, 32 Bit binär kodiert, Big Endian</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>IPv4 Port, 16 Bit binär kodiert, Big Endian</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>Größe der Bubble, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>Start der Bubble, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>Ende der Bubble, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
<td>Bubble-Typ, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>Präsenzindikator für Repl.-Daten. Hier: 1</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>Ursprünglicher Replikationszeitpunkt, 32 Bit Ganzzahl</td>
</tr>
<tr>
<td>35</td>
<td>( k )</td>
<td>Replikat-Name, ( k ) beliebig aber fest</td>
</tr>
</tbody>
</table>

Literaturverzeichnis


