Universität Rostock
Lehrstuhl Datenbank- und Informationssysteme

Diplomarbeit

Datenverteilung in Peer-to-Peer Overlay-Netzwerken

Andreas Pohl

geb. am 19. Oktober 1979 in Templin
Matrikelnummer 099202459

Betreuer:	Dr. Holger Meyer, Andre Zeitz
Hochschullehrer:	Prof. Dr. Andreas Heuer
Gutachter:	Prof. Dr. Adelinde M. Uhrmacher

eingereicht am 29. Oktober 2004

Zusammenfassung

In dieser Diplomarbeit wurde ein mobiles Informationssystem (MIS) auf Basis eines dezentralisierten, unstrukturierten Peer-to-Peer Overlay Netzwerkes entwickelt. Es werden Strategien zur Verteilung von Daten im Netzwerk (Replikation) und Probleme bei Änderungsoperationen (Updates) diskutiert. Geeignete Ansätze zur Umsetzung der Datenverteilung und der Updates wurden gewählt und auf das System angepasst. Um die eingeführten Konzepte zu validieren, wurde ein Prototyp eintwickelt.

Schlüsselwörter

Dezentralisiert, unstrukturiert, Peer-to-Peer Overlay Netzwerk, Ad-Hoc, Replikation, Distributed Hash Tables, Updates

Abstract

The subject of this work is a mobile information system (MIS). It is based upon a decentralized, unstructured peer-to-peer overlay network. This work discusses several strategies for distributing data through the network (replication). New problems arrise with replication, i.e. changing operations (updates) on data objects. Suitable approaches for the implementation of the replication and the updates were chosen and adapted to fit into the information system. A prototype was realized to validate the introduced concepts.

Keywords

decentralized, unstructured, peer-to-peer overlay network, ad-hoc, replication, distributed hash tables, updates

CR-Klassifikation

C.2.1 Network communications
C.2.4 Distributed applications
E.2 Hash-table representations
H.2.4 Distributed systems
H.3.2 Information Storage
H.3.4 Information networks

Inhaltsverzeichnis

Kapitel 1

Einleitung

Peer-to-Peer-Netzwerke erfreuen sich immer größerer Beliebtheit. Ständig werden neue Anwendungen auf dieser Basis entwickelt. Das Internet bietet einen großen Vorrat an Ressourcen, sowohl in Form von Speicherplatz, als auch an Rechenleistung. Es ist möglich, mit Hilfe von Peer-to-Peer-Netzwerken solche Ressourcen zu teilen, genauso wie sie zu vereinen. Datenmessungen sagen voraus, dass das Datentransfervolumen solcher Anwendungen stark steigt und ein sehr großer Teil des Internetverkehrs dadurch ausgemacht werden wird [LCC+02].

Peer-to-Peer bedeutet Kommunikation unter Gleichen. Das heißt, solche Netzwerke bilden einen Gegensatz zum heute aktuellen Client/Server-Prinzip, bei dem Server Dienste anbieten, die von Clients genutzt werden. Ein Beispiel dafür sind Webserver und die auf Nutzerseite installierten Browser. In das Adressfeld des Browsers wird eine URL eingegeben, dieser verbindet sich mit dem Webserver und bezieht die Webseite, um sie darzustellen. In einem Peer-to-Peer Netzwerk ist prinzipiell jeder Knoten gleich und nimmt die Rolle eines Servers, eines Clients und eines Routers an, der Nachrichten weiterleitet. Dafür wird ein Overlay-Netzwerk gebildet, das über der bestehenden Infrastruktur eines Netzwerkes, zum Beispiel des Internets, aufgebaut wird.

Man unterscheidet drei Arten von Peer-to-Peer Netzwerken [LCC+02]:

- **Zentralisiert:** Es gibt zentrale Server, die die Kommunikation steuern. Beispielsweise bei Napster [Nap] wurden die Verzeichnisinhalte der Knoten auf solchen Servern gespeichert. Anfragen von Knoten wurden von diesen Maschinen bearbeitet und entsprechende Ergebnisknoten mit gewünschtem Inhalt zurückgeliefert.

- **Strukturiert, dezentralisiert:** In solchen Systemen existieren keine zentralen Server mehr, jedoch herrscht eine bestimmte vordefinierte Netzwerkstruktur vor. In dieser Struktur werden Dateien an bestimmten Orten platziert, um Anfragen besser bearbeiten zu können. Solche Systeme sind beispielsweise Distributed Hash Tables (DHT) wie Tapestry [ZKJ01], Pastry [RD01], Chord [SMK+01] oder CAN [RFH+01].

- **Unstrukturiert, dezentralisiert:** In diesen Systemen gibt es auch keine zentralen Server. Hinzu kommt, dass die vordefinierte Netzwerkstruktur wegfällt und es keine Regelung über die Dateiplatzierungen mehr gibt. Dadurch werden die Suchmechanismen meist über Flooding[1] umgesetzt, so dass eine sehr hohe Netzlast entsteht. Ein Beispiel für ein unstrukturiert dezentralisiertes Peer-to-Peer-Netzwerk ist Freenet [CSWH01].

Zentralisierte Systeme haben zwar den Vorteil, dass Anfragen durch zentrale Indexe leichter realisiert werden können, jedoch gerade die zentralen Server bilden den Schwachpunkt dieser Netzwerke. Zum einen können dadurch schnell Engpässe entstehen, wenn die Anzahl der Anfragen auf einem Server steigt, und zum anderen funktioniert das Netzwerk ohne die Server nicht. Somit gibt es zentrale Angriffspunkte, um das gesamte Netzwerk einzuschränken. Ein weiterer Punkt ist die Vertraulichkeit der Daten. Dritte können leicht von wenigen Stellen aus das gesamte Netzwerk überwachen.

Dezentralisierte Netzwerke haben diese Probleme nicht. Sie sind sehr viel robuster gegen Ausfälle und Überwachung, da es keine zentralen Instanzen gibt. Wenn wenige Knoten ausfallen, bleibt das Netzwerk weiterhin funktionsfähig. In unstrukturierten Systemen entsteht das Problem des Overheads der Suche. Im Gegensatz dazu stehen die strukturierten Netzwerke. DHT-Systeme (dt.: verteilte Hashtabellen) bieten eine hashtabellenähnliche Funktionalität an, bei der Objekte auf Knoten abgebildet werden. Das System liefert auf die Anfrage nach einem Objekt den Knoten, der es speichert.

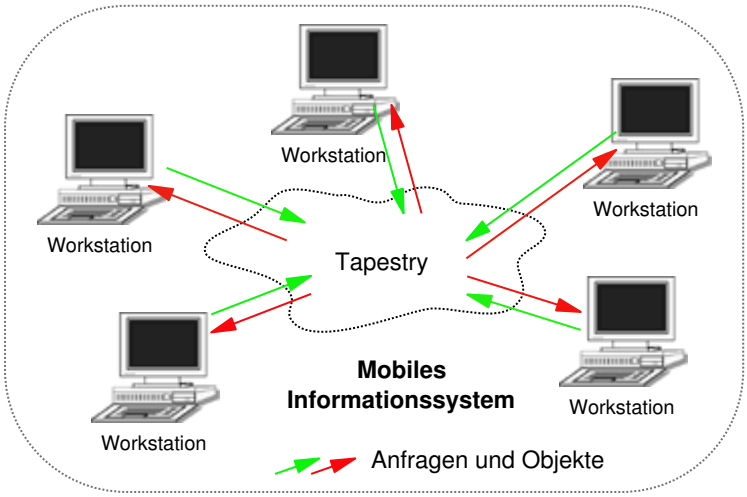

Abbildung 1.1: Die abstrakte Darstellung des mobilen Informationssystems, das in dieser Arbeit entwickelt wurde.

[1] Eine Anfrage wird an alle Nachbarn weitergeleitet.

Im Zuge dieser Diplomarbeit wird ein mobiles Informationssystem (MIS) entwickelt (siehe Abbildung 1.1 für eine abstrakte Darstellung), das auf einer dezentralisierten, strukturierten Infrastruktur aufbaut. Als Lokalisierungs- und Routing-System liegt Tapestry zu Grunde. Das System kann seine Anwendung unter anderem auf Konferenzen, Messen oder Workshops finden, wo sich Wissenschaftler, Techniker oder interessierte Menschen zusammenfinden, um neue Technologien auszutauschen. Die meisten Wissenschaftler reisen mit Notebooks an, auf denen sich oft themenrelevante Dokumente befinden. Um diese Dokumente auszutauschen, können die Rechner über das mobile Informationssystem verbunden werden. Eine zentralisierte Infrastruktur kam für dieses System nicht in Frage, da das Netzwerk spontan aufgebaut wird und keine Server notwendig sein sollen. Da die Rechner meist über WLAN verbunden sein werden, muss mit geringen Bandbreiten gerechnet werden. Daher wurde kein unstrukturierter Ansatz gewählt, denn der Suchaufwand ist dabei sehr hoch.

Das zentrale Interesse der Arbeit richtet sich auf die Verteilung der Daten im Netzwerk durch Replikation. Damit soll in erster Linie die Verfügbarkeit der Daten verbessert werden. Das System legt dafür selbständig Kopien (Replikate) von Objekten auf verschiedenen Knoten im Netzwerk ab. Fällt ein Knoten aus oder ist er überlastet, kann das Objekt von einem der anderen Knoten geladen werden. Replikation erschwert jedoch Änderungsoperationen auf den im Netzwerk befindlichen Daten, da ein Update auf einem Objekt auch auf alle Replikate übertragen werden muss. Je mehr Replikate eines Objektes existieren, desto schwieriger wird eine Änderungsoperation. Es werden in der Arbeit Strategien zur Verteilung der Daten und zum Durchführen von Updates vorgestellt und verglichen.

Die Arbeit ist folgendermaßen aufgebaut: Im Kapitel 2 werden verschiedene Lokalisierungs- und Routing-Systeme vorgestellt. Auf Datenverteilungsmechanismen und Replikation geht Kapitel 3 ein. Der konzeptionelle Aufbau und die Funktionsweise des Systems werden in Kapitel 4 beschrieben. Kapitel 5 stellt die konkrete Realisierung und Arbeitsweise des Prototypen vor. Ein Ausblick auf Erweiterungs- und Verbesserungsmöglichkeiten gibt das Kapitel 6 der Arbeit.

Kapitel 2

Lokalisierung und Routing

In diesem Kapitel werden Lokalisierungs- und Routing-Systeme (kurz Lookup-Service) vorgestellt. Das Tapestry-System dient als Grundlage für diese Diplomarbeit. Alle Systeme sind DHT-Systeme und bieten damit eine hashtabellenähnliche Funktionalität, mit der Objekte auf Knoten im Netzwerk abgebildet werden. Weiterhin bilden alle DHTs dezentralisierte, strukturierte Overlay- oder Ad-hoc-Netzwerke. Es gibt somit keine zentralen Serverknoten, denn alle Knoten im Netzwerk haben die gleichen Rollen.

2.1 Tapestry

Tapestry [ZKJ01] bietet die Möglichkeit, Nachrichten an Objekte oder Dienste mit einem ortsunabhängigen Namen zu senden. Diese Nachrichten werden zu einem Knoten geleitet, der eine Kopie (Replikat) des Objektes speichert und dem Sender am dichtesten[1] ist. Auf den Replikationsbegriff geht Kapitel 3 näher ein. Tapestry basiert auf dem Plaxton-Schema, das 1997 in [PRR97] vorgestellt wurde. Jeder Rechner bzw. Knoten im Netzwerk kann in dieser Struktur die Rolle eines Servers, der Datenobjekte speichert, eines Routers, der Nachrichten weiterleitet oder eines Clients, der Anfragen stellt, annehmen. Alle Objekte und Knoten im Netz bekommen einen Identifikator (ID) in Form einer zufälligen Bitkette bestimmter Länge (in Tapestry 160bit). Diese Identifikatoren sind unabhängig von der Position im Netz. Die Berechnung erfolgt in der Regel durch Hashing-Algorithmen, wie beispielsweise SHA-1 [Rob95].

Für das Routing wird in jedem Knoten eine Routing-Tabelle (Nachbartabelle) gehalten, die die Nachbarn des jeweiligen Knotens speichert. Mit Hilfe dieser Tabellen wird eine Nachricht inkrementell Zahl für Zahl zur Ziel-ID geroutet (siehe Abbildung 2.1). Hierzu wird ein so genanntes Präfix-Routing verwendet. Ein Knoten leitet eine Nachricht an einen Knoten weiter, dessen Knoten-ID das

[1] Dicht bezieht sich hier und im Rest der Arbeit auf die Entfernung im Netzwerk, die aus den Latenzen zwischen Computern berechnet wird.

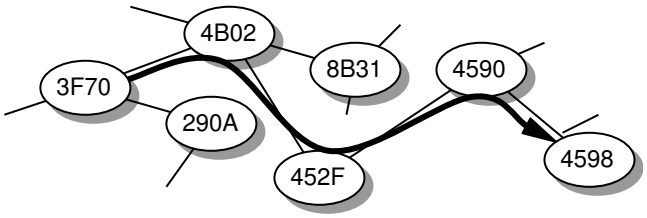

Abbildung 2.1: Routing in Tapestry: Eine Nachricht wird von Knoten 3F70 zu 4598 gesendet. Das inkrementelle Routing verläuft wie folgt: 4*** ⇒ 45** ⇒ 459* ⇒ 4598, wobei die Sterne Platzhalter sind.

gleiche Präfix hat, wie die Objekt-ID. Das Präfix muss eine Zahl länger sein, als das Präfix für den aktuellen Knoten.

Um ein auf Knoten S gespeichertes Objekt O im Netz einzufügen bzw. zu publizieren, sendet S eine Publikationsnachricht an den „Wurzelknoten“ von O. Der Wurzelknoten eines Objektes ist ein eindeutiger Knoten im Netz, der die Wurzel des eingebetteten Baumes bildet[2]. Er speichert die Adresse vom Server S, auf dem das Objekt O gefunden werden kann. Die Speicherung erfolgt in Form eines so genannten *Pointer-Mappings* $ID(O) \rightarrow ID(S)$. Auf jedem weiteren Knoten, der auf dem Pfad von S zur Wurzel liegt, wird das gleiche Pointer-Mapping gespeichert. Sollten mehrere Server das Objekt O speichern, wird in Plaxton nur das Pointer-Mapping mit der kleinsten Netzwerklatenz des aktuellen Knotens zu S gespeichert, während in Tapestry alle Pointer-Mappings sortiert nach der Netzwerklatenz des aktuellen Knotens zu S gespeichert werden [ZHS+03]. Abbildung 2.2 zeigt einen Ausschnitt eines Tapestry-Netzwerkes.

Stellt ein Client eine Anfrage, sendet er eine Nachricht an ein Objekt O, welche über den Wurzelknoten von O geroutet wird. Sobald die Nachricht auf dem Weg zur Wurzel einen Knoten erreicht, der ein Pointer-Mapping mit der gesuchten Objekt-ID enthält, wird sie direkt zum Server, der O speichert, weiter geleitet. Andernfalls wird sie einen Knoten weiter in Richtung der Wurzel geleitet. Erreicht die Nachricht die Wurzel, ist garantiert, dass dort die Adresse des Servers enthalten ist, der O speichert.

2.2 Pastry

Pastry ist Tapestry und Chord (siehe unten) sehr ähnlich. Auch hier werden Knoten und Objekte mit zufällig bestimmten Identifikatoren (Knoten-ID, Objekt-ID) versehen, die ortsunabhängig sind. Sie können auch mit Hilfe von

[2]Ein solcher Baum existiert für jedes publizierte Objekt im Netzwerk. Die Knoten, die ein Objekt speichern, bilden die Blätter eines Baumes.

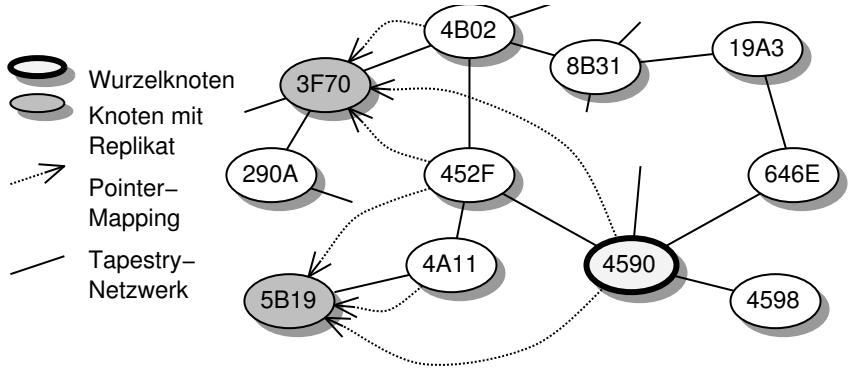

Abbildung 2.2: Pointer-Mappings in einem Tapestry-Netzwerk

Hash-Funktionen wie SHA-1 [Rob95] über der IP Adresse oder des Public-Keys des Knotens bestimmt werden und sind 128bit lang.

Das Routing erfolgt ebenso wie in Tapestry. Eine Nachricht wird zu dem Knoten geleitet, dessen Knoten-ID numerisch am dichtesten an der Objekt-ID liegt. Hierzu wird das in 2.1 beschriebene Präfix-Routing verwendet. Neben der Routing-Tabelle verwaltet ein Pastry-Knoten eine Nachbarschaftsmenge M und eine Blattmenge L. In M werden die $|M|$ dichtesten (Netzwerkentfernung) Knoten gespeichert. Diese Menge dient hauptsächlich der Speicherung von Lokalitätseigenschaften, zum Beispiel für den Aufbau der Routing-Tabelle. Die Blattmenge L enthält $|L|/2$ Knoten, die numerisch im Namensraum vor einem Knoten stehen und $|L|/2$ Knoten, die im Namensraum folgen. Diese Menge wird im Routing verwendet. Bevor eine Nachricht über das Präfix-Routing weitergeleitet wird, überprüft ein Knoten, ob sich die Ziel-ID in seiner Blattmenge befindet.

Pastry beschränkt sich nur auf das Routing selbst. Das Einfügen oder Löschen von Objekten in das Netzwerk bleibt der aufsetzenden Anwendung überlassen.

2.3 Bamboo

Bamboo ist eine freie Implementierung der Pastry-Protokolle, die an einigen Stellen Optimierungen vorgenommen hat. Es ist auf das häufige Ein- bzw. Austreten von Knoten in das Netzwerk spezialisiert [RGR+03]. Viele Systeme neigen dazu, unter solchen Bedingungen durch den Mehraufwand an Nachrichten zusammenzubrechen.

Jeder Knoten n hat eine Blattmenge, in der $2k$ Knoten enthalten sind, die n im Namensraum direkt folgen bzw. vorausgehen. Periodisch sendet ein Knoten seine Blattmenge zu einem zufällig ausgewählten Nachbarn (push). Im Gegen-

satz zu Pastry antwortet der Nachbar mit seiner eigenen Blattmenge (pull). Dadurch wird das System robuster, da neue Knoten bzw. verschwundene Knoten schneller von allen anderen bemerkt werden. In Abbildung 2.3 wird eine Situation dargestellt, in der das Rücksenden der Blattmengen notwendig ist. Während der Ankunft der Knoten B und D ist C kurzzeitig nicht verfügbar, so dass er bei seiner Wiederkehr immer noch A und E als Nachbarn in der Blattmenge hat. Außerdem haben B und D sich selbst als Nachbarn. Ohne Rücksenden der Blattmengen erfährt C nie von B und D und behält immer eine falsche Blattmenge.

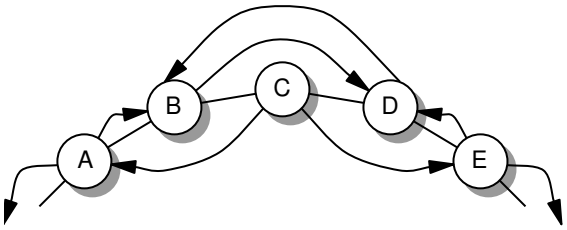

Abbildung 2.3: Blattmengen in Bamboo: $k = 1$. Das Senden und Rücksenden ist notwendig, da die Blattmenge von C sonst falsch bleibt.

Durch das Trennen von Netzwerkein- bzw. austritten vom Synchronisieren der Nachbarschaftsinformationen entsteht beim Eintritt vieler Knoten sehr viel weniger Netzwerkverkehr, als wenn jeder Knoten allen anderen seine Ankunft mitteilen würde.

2.4 Chord

Der Schwerpunkt in Chord [SMK+01] liegt auf einem verteilten Suchprotokoll, durch das eine hashtabellenähnliche Funktionalität angeboten wird, in dem Schlüssel auf Knoten abgebildet werden. Jeder Knoten und jeder Schlüssel im Netz bekommt einen m-bit Identifikator, der durch eine Hash-Funktion wie SHA-1 [Rob95] berechnet wird. Die Identifikatoren werden in einem Kreis modulo 2^m angeordnet. Ein Schlüssel k wird dem ersten Knoten zugeordnet, der den gleichen Identifikator wie k hat oder im Identifikatorraum (im Uhrzeigersinn) folgt. In Abbildung 2.4 ist ein Identifikatorenkreis mit neun Knoten dargestellt.

Jeder Knoten verwaltet eine Routing-Tabelle mit maximal m Zeigern auf Knoten, die im Namensraum folgen. Der $i$te Eintrag von Knoten n enthält den ersten Knoten, der n im Identifikatorkreis nach 2^{i-1} folgt ($n + 2^0$, $n + 2^1$, ..., $n+2^{m-1}$). Stellt ein Client eine Anfrage an einen Knoten n, überprüft dieser, ob der übergebene Schlüssel k zwischen dem Knoten und seinem Nachfolger liegt

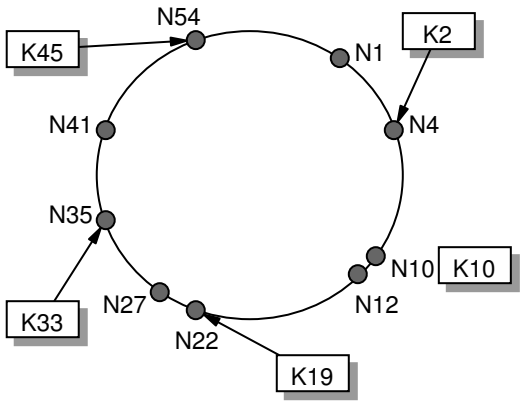

Abbildung 2.4: Ein Identifikatorenkreis mit 9 Knoten und 5 Schlüsseln

($k \in (n, n.nachfolger]$). Ist das der Fall, wird $n.nachfolger$ zurückgegeben. Andernfalls wird die Anfrage an den ersten Knoten aus der Routing-Tabelle von n weitergeleitet, der einen kleineren Identifikator als k hat.

Der Hauptunterschied zwischen Chord und Tapestry ist, dass in Chord kein Zusammenhang zwischen der Entfernung im Identifikatorkreis und der eigentlichen Entfernung im Netzwerk besteht. Dadurch entsteht die Möglichkeit, dass im Overlay-Netzwerk Routen physikalisch sehr lang sind.

2.5 Content-Addressable Network

Ein Content-Addressable Network [RFH+01] (CAN, dt. inhaltsadressierbares Netzwerk) bietet wie die anderen Systeme eine hashtabellenähnliche Funktionalität in einer verteilten Infrastruktur an. Der Kern ist ein virtueller d-dimensionaler kartesischer Koordinatenraum auf einem d-Ring. Der Raum ist so unterteilt, dass jedem Knoten im Netz ein Bereich zugeteilt ist. In diesem virtuellen Koordinatenraum werden Schlüssel-Wert-Paare (k, v) gespeichert. Um einen Wert v zu einem Schlüssel k zu speichern, wird k mittels einer uniformen Hash-Funktion auf einen Punkt p im Koordinatenraum abgebildet. Das Schlüssel-Wert-Paar (k, v) kann dann auf dem zugeordneten Knoten gespeichert werden. Beim Zugriff auf den Wert v eines Schlüssels k, kann ein Knoten mit derselben Hash-Funktion über k den Punkt p berechnen, der dem speichernden Knoten zugeordnet ist. Ein Beispiel für ein CAN-Netzwerk zeigt Abbildung 2.5.

Alle Knoten in einem CAN haben eine Routing-Tabelle, in der die Nachbarknoten und ihre Bereiche verzeichnet sind. Die Bereiche von Nachbarknoten im Koordinatenraum grenzen an den Bereich des Knotens. Eine CAN Nachricht

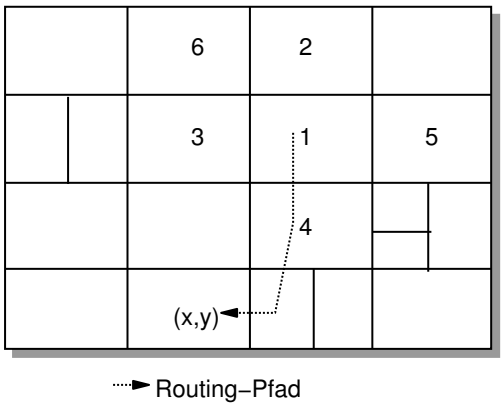

Abbildung 2.5: Ein Content-Addressable Network: Ein 2-d-Raum mit einem Routing-Pfad von Knoten 1 zum Punkt (x, y).

enthält die Zielkoordinaten, so dass ein Knoten eine Nachricht an den Nachbarn weiterleitet, dessen Koordinaten am dichtesten an den Zielkoordinaten liegen. Dieses Vorgehen wird solange wiederholt, bis die Nachricht ihr Ziel erreicht.

2.6 Lokalisierung und Routing im mobilen Informationssystem

Als Grundlage des Peer-to-Peer-Netzwerksystems dieser Arbeit wurde Tapestry gewählt. Die folgenden Gründe sind dafür ausschlaggebend:

- Die Publikation und Lokalisierung von Objekten wurde umgesetzt.
- Eine gut getestete und mehrfach eingesetzte Implementierung in Java existiert als Freeware.
- Es sind gute Dokumentationen zur API und zur Verwendung vorhanden.
- Das zu entwickelnde Replikationssystem (in den folgenden Kapiteln beschrieben) kann sehr gut mit Tapestry umgesetzt werden.

In den folgenden Kapiteln werden verschiedene Strategien und Systeme zur Replikation von Daten in verteilten Systemen sowie das Konzept und die Realisierung des Prototypen im Rahmen dieser Arbeit erläutert.

Kapitel 3

Replikation

Der Begriff Replikation taucht in verschiedenen Bereichen der Wissenschaft auf, beispielsweise der Genetik, verteilten Datenbanken oder Peer-to-Peer-Netzwerken. Im Allgemeinen bedeutet er Vervielfältigung oder Verteilung. In dieser Arbeit wird von folgender Definition für Netzwerke ausgegangen.

Definition 3.0.1 (Replikation) *Die Replikation einer Datei oder eines Objektes in einem Netzwerk ist die Vervielfältigung der Datei auf mehrere Knoten des Netzwerkes. Der Grad der Replikation ist die Gesamtanzahl aller Replikate einer Datei im Netzwerk.*

Es gibt verschiedene Gründe für die Replikation von Objekten in Peer-to-Peer-Netzwerken. In erster Linie soll die Qualität des Dienstes (engl.: Quality of Service, QoS) verbessert bzw. optimiert werden [OSS03a, OSS03b, OSS03c, CKK02]. Dazu zählt beispielsweise die Verfügbarkeit von Objekten, der Durchsatz im Netzwerk beim Laden eines Objektes, die Aktualität eines Objektes. Tritt ein Knoten in ein Netzwerk ein und publiziert lokal gespeicherte Objekte, sollen diese nach Möglichkeit auch nach seinem Austritt, der unter anderem durch Netzfehler ausgelöst werden kann, verfügbar sein. Die Anforderungen an die Replikation sind anwendungsspezifisch. Verteilte Dateisysteme wie OceanStore [BCE+99], CFS [DKK+01] oder FarSite [ABC+02] haben einen hohen Durchsatz beim Download von Daten und hohe Verfügbarkeiten als Anforderung. Verteilte Datenbanken versuchen, niedrige Zugriffszeiten und eine hohe Fehlertoleranz zu erreichen [BD96]. Eine Anwendung, die beispielsweise XML-Dokumente speichert, wird Anfragen verteilen, um niedrige Antwortzeiten zu erreichen, statt die Downloadgeschwindigkeiten zu optimieren. File-Sharing Systeme versuchen den größtmöglichen Inhalt zu bieten, d.h. so viele Dokumente wie möglich im Netzwerk zu halten. Gleichzeitig soll ein hoher Durchsatz erreicht werden. Andere Systeme können versuchen, die Verfügbarkeit ganzer Objektgruppen zu erhöhen, in dem zusammenhängende Objekte zusammen gespeichert werden [BDET00].

Durch Replikation werden jedoch Änderungsoperationen (Updates) auf Objekten schnell sehr aufwendig, da bei einer Änderung eines Objektes auch alle Replikate aktualisiert werden müssen. Wird beispielsweise eine 1-Kopien-Äquivalenz [BD96] gefordert, wird der Aufwand weiter gesteigert. Dabei soll sich das System verhalten, als wäre nur eine einzige Kopie eines Objektes im Netzwerk, d.h. alle Replikate müssen konsistent zueinander sein. Die Gefahr von Inkonsistenzen steigt somit durch Replikation. In [BD96] wird ein Zielkonflikt der Replikationskontrolle beschrieben. Einerseits soll die Verfügbarkeit, Zugriffszeit oder der Durchsatz erhöht werden, andererseits soll der Aufwand bei Updates gering sein, und die Daten sollen konsistent bleiben (siehe Abbildung 3.1). Befinden sich viele Replikate im Netzwerk, verringert sich die Antwortzeit

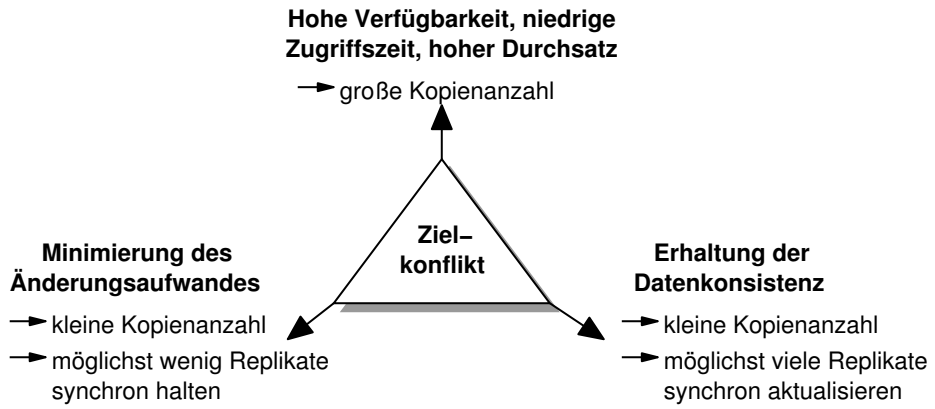

Abbildung 3.1: Der Zielkonflikt der Replikationskontrolle (aus [BD96])

bei Lesezugriffen und erhöht sich bei Schreibzugriffen.

Es gibt eine Reihe von Kriterien, die in dieser Arbeit aufgestellt und nach denen Systeme und Algorithmen zur Replikation untersucht werden. Folgendes Szenario ist vorstellbar: Auf einer Konferenz sind die Teilnehmer per Notebook an ein Netzwerk angeschlossen und können auf einen Dokumentenserver zugreifen. Ein Großteil aller Zugriffe auf die Dokumente des Servers sind lesender Art, hauptsächlich vor und nach den Vorträgen. Während der Vorträge sinken diese Zugriffe, und die Vortragsfolien werden häufig geändert, so dass sich die Anzahl der Schreibzugriffe erhöht. Nach den Vorträgen gehen die Schreibzugriffe auf ein Minimum zurück und die Lesezugriffe pegeln sich auf einem gewissen Level ein.

Die vorhandene Struktur kann durch das mobile Informationssystem ersetzt werden. Dadurch fällt der Dokumentenserver weg und alle Dokumente sind auf den Computern im Netzwerk gespeichert.

3.1 Kriterien für Replikationssysteme

Im Folgenden werden Kriterien aufgestellt, die für Replikationssysteme relevant sind. Die Algorithmen aus Abschnitt 3.3 werden an Hand dieser Kriterien betrachtet.

(1) Replikationsverhalten

(2) Wissensbasis

(3) Qualitätsbedingungen

(4) Updates

(5) Konsistenz

(6) Replikationsdaten

(7) Granularität

3.1.1 Replikationsverhalten

Ändert das Replikationssystem bei Änderungen des Nutzerverhaltens sein eigenes Verhalten? Wie im Szenario beschrieben, können zu verschiedenen Tageszeiten mehr Lesezugriffe oder mehr Schreibzugriffe auftreten. Haben Replikationssysteme ein statisches, dynamisches oder adaptives Replikationsverhalten? Statisch bedeutet fest, das heißt keine änderbaren Parameter beeinflussen die Replikation. Dynamische Verfahren hingegen haben Parameter, die die Replikation beeinflussen und die veränderbar sind. Die Dynamik besteht darin, diese Parameter an Hand des Nutzerverhaltens anzupassen. Bei einem adaptiven Verfahren werden nicht nur Parameter geändert, sondern die gesamte Vorgehensweise der Replikation unter bestimmten Voraussetzungen. Wird von einem Replikationsmodell mit Parametern ausgegangen, bedeutet statisch, dass die Parameter fest sind, dynamisch, dass sie geändert werden und adaptiv, dass sich das Replikationsmodell selbst ändert.

3.1.2 Wissensbasis

Welches Wissen ist für die Verteilung notwendig: Kein Wissen, lokales oder globales Wissen? Reicht lokales Wissen aus, beispielsweise wenn ein Knoten nur Informationen über sich und seine Nachbarn besitzt, oder ist globales Wissen nötig, bei dem ein Knoten die gesamte Netzwerkstruktur und das gesamte Replikationsschema kennen muss?

3.1.3 Qualitätsbedingungen

Gibt es Qualitätsbedingungen (QoS)? Folgende Qualitätsbedingungen sind üblich:

- Mindestverfügbarkeit eines Objektes
- Maximallatenz beim Zugriff auf ein Objekt
- Mindestdurchsatz beim Download eines Objektes
- Fehlertoleranz, eine gewisse Anzahl an Knoten kann ausfallen ohne Datenverlust im Netzwerk
- Mindestqualität eines Objektes (zum Beispiel ein Musiktitel bei einer Musik-Tauschbörse)

Sind solche Bedingungen definierbar?

3.1.4 Updates

Wie werden Updates realisiert? Wie werden Änderungen an einem Objekt auf all seine Replikate übertragen? Sind Updates überhaupt möglich? Wird eine Versionierung verwendet, bei der bei einem Update eine neue Version eines Objektes erzeugt wird? Ein Nutzer könnte in diesem Fall bei der Suche nach einem Dokument eine Mindestversion als Qualitätsbedingung angeben, so dass eine gewisse Aktualität des Dokumentes gewährleistet ist. Oder es existiert immer nur die neuste Version eines Objektes im Netzwerk (Ein-Kopien-Serialisierbarkeit). Dadurch stellt sich die Frage nach der Konsistenz.

3.1.5 Konsistenz

Konsistenz bedeutet Widerspruchsfreiheit. Es wird auch von semantischer Integrität gesprochen. Das bedeutet beispielsweise für eine Datenbank, dass Daten logisch richtig und alle Konsistenzbedingungen erfüllt sind. Die tatsächlichen Gegebenheiten müssen mit der Abbildung in Form der Daten übereinstimmen.

Gibt es Konsistenz- oder Integritätsbedingungen? Ähnlich wie bei konkurrierenden Transaktionen in Datenbanksystemen, die auf gleiche Objekte zugreifen können, können in einem Peer-to-Peer-Netzwerk mehrere Knoten zugleich versuchen, ein Objekt zu aktualisieren. Wie werden nach einem Update die Replikate aktualisiert? Müssen alle Replikate sofort aktualisiert werden oder nach und nach? Lässt man veraltete Versionen von Replikaten möglicherweise einfach verfallen? Wann ist ein Netzwerk in einem konsistenten Zustand?

3.1.6 Replikationsdaten

Was wird in dem System repliziert? Werden die Objekte oder nur Metadaten zu den Objekten verteilt? Metadaten sind Informationen über die Objekte selbst, wie zum Beispiel der Autor eines Artikels, der Verlag eines Buches oder der Interpret eines Musiktitels. Sie können auch statistische Informationen enthalten, wie beispielsweise die Anzahl aller Zugriffe, die Zugriffsrate, die Hitrate, die Fehlrate, die Anzahl der Anfragen und Updates oder den Grad der Verteilung.

Die Frage nach den Replikationsdaten stellt sich unabhängig vom verwendeten Algorithmus zur Verteilung von Objekten. Diese Frage stellt sich bei konkreten Systemen und wird für das mobile Informationssystem dieser Arbeit in Kapitel 4 genauer betrachtet. Systeme wie OceanStore [BCE+99], FarSite [ABC+02], CFS [DKK+01], Freenet [CSWH01] und andere replizieren nur die Objekte selbst, während hier sowohl Metadaten als auch die Objekte repliziert werden.

3.1.7 Granularität

Welche Granulate gibt es? In Datenbanksystemen wird üblicherweise auf Mengen von Objekten zugegriffen. Wie wird der Zugriff in Peer-to-Peer-Netzwerken realisiert: Per Menge, direkt auf ein Objekt oder auf Teile des Objektes?

Auf die Frage der Granularität wird im folgenden Abschnitt eingegangen. Das mobile Informationssystem repliziert ganze Objekte. Somit kann nur auf ganze Objekte, nicht auf Mengen oder Teile davon zugegriffen werden. Die anderen Kriterien werden in Abschnitt 3.3 im Zusammenhang mit den betrachteten Algorithmen behandelt.

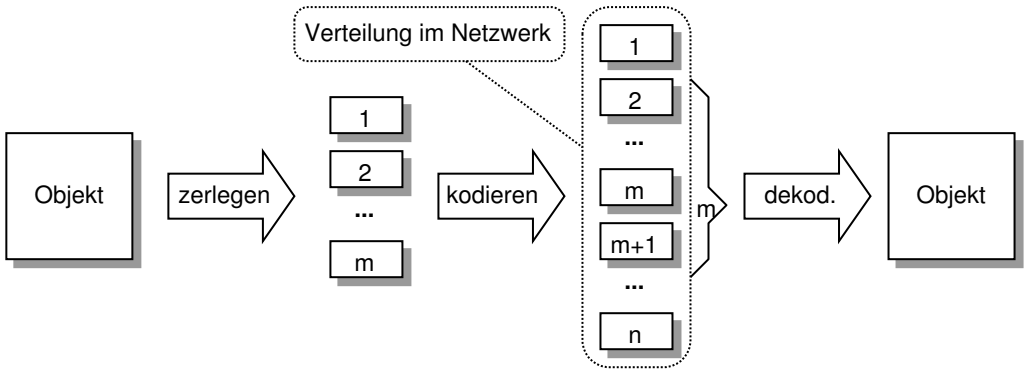

Abbildung 3.2: Erasure Codes

3.2 Granularität in Peer-to-Peer-Netzwerken

Grundlegend gibt es zwei verschiedene Möglichkeiten, Datenobjekte in einem Peer-to-Peer-Netzwerk zu verteilen:

- objektweise und
- blockweise.

Beim objektweisen Vorgehen werden ganze Datenobjekte repliziert, während diese bei der blockweisen Verteilung in einzelne Blöcke zerlegt und nachfolgend verteilt werden.

Eine Technik des blockweisen Verteilens und der redundanten Speicherung sind die Erasure Codes [WK]. Abbildung 3.2 stellt die Vorgehensweise dar. Man zerlegt ein Objekt in m Teile und kodiert sie in n Teile, wobei $n > m$ ist. Danach werden die erzeugten n Fragmente im Netzwerk verteilt. Nun kann das Objekt aus beliebigen m Fragmenten zusammengesetzt werden.

	objektweise	blockweise
XML	■	□
MP3	□	■
Video	□	■
Bilder	■	■
PDF	■	■

Tabelle 3.1: Verteilung verschiedener Objekttypen

Die Tabelle 3.1 stellt einige bekannte Objekttypen dar. XML-Dokumente werden am sinnvollsten objektweise repliziert, wenn beispielsweise eine Volltextsuche auf diesen Objekten durchgeführt werden soll. Bei einer blockweisen Verteilung, bei der ein Knoten normalerweise nur einen Teil des Dokumentes speichert, wird eine Volltextsuche sehr aufwendig, da erst das komplette Objekt zusammengesetzt werden muss. Speichert ein Knoten hingegen das ganze Objekt, können solche Suchanfragen leicht umgesetzt werden.

Bei PDF-Dokumenten bieten sich beide Verfahren an. Auf der einen Seite handelt es sich um Binärdaten, die nicht für eine gewöhnliche Volltextsuche geeignet sind. Auf der anderen Seite gibt es Konverter, die PDF-Dokumente zu ASCII-Dokumenten umwandeln und somit eine Volltextsuche ermöglichen können.

Für die meisten Multimediadaten hingegen ist das blockweise Verteilen besser geeignet, da große Objekte länger brauchen um repliziert zu werden. Bei solchen Daten lässt sich durch das Zerlegen der Objekte ein hoher Grad an Parallelität beim Verteilen als auch beim Laden eines Objektes aus dem Netzwerk erreichen. Verschiedene Teile des Objektes können von verschiedenen Knoten

zugleich geladen werden. Außerdem lassen sich die verfügbaren Speicherplatzressourcen besser balancieren. Ein weiterer Vorteil blockweiser Replikation ist die einfache Umsetzung von Cache-Speichern, da nicht mit verschieden großen Objekten, sondern mit einheitlichen Blöcken gearbeitet werden kann. Bei Bilddaten sind beide Formen der Verteilung vorstellbar. Da es sich meist um kleine Objekte handelt, kann auch eine objektweise Replikation eingesetzt werden.

Wenn eine Anwendung ganze Objekte benötigt, besteht ein Nachteil dieser Verteilungsart. Mindestens ein Knoten pro Block (oder bei Erasure Codes m von n Knoten, die Blöcke speichern) muss mit dem Netzwerk verbunden sein, um ein Objekt aus dem Netz zu laden [BMSV02]. Soll zum Beispiel ein PDF-Dokument oder ein Bild dargestellt werden, muss die gesamte Datei heruntergeladen werden.

Das OceanStore System [BCE+99] setzt blockweise Replikation ein. Bei dem System dieser Arbeit handelt es sich um ein mobiles System. Die Anzahl der Knoten wird häufig variieren. Die Objekte im Netzwerk sind hauptsächlich wissenschaftliche oder technische Arbeiten oder Vortragsfolien und werden meist klein sein. Deshalb ist die objektweise Replikation hier besser geeignet. In einem Umfeld, in dem Knoten häufig ein- und austreten, sinkt die Wahrscheinlichkeit, dass alle für einen Objektdownload benötigten Knoten gleichzeitig online sind. Solch eine Situation kommt der gleich, dass das Objekt gar nicht im Netzwerk vorhanden ist. Es sollen später Anfragen und Volltextsuchen auf Dokumenten im Netzwerk durchgeführt werden. Diese sind dann unmöglich.

3.3 Replikationsalgorithmen

Dieser Abschnitt handelt von Algorithmen und Strategien, mit deren Hilfe die Verteilung von Objekten, das heißt das Erzeugen von Replikaten, in einem Netzwerk vorgenommen werden kann. Es stellen sich immer die Fragen, welche Objekte zu replizieren sind und auf welchen Knoten die Replikate gespeichert werden sollen. Da sich die Replikationsbedingungen dynamisch ändern, muss das Replikationsschema eines Objektes dynamisch angepasst werden.

In [OSS03a] wurde ein Replikationsmodell unter drei Aspekten entwickelt:

- Welche Objekte sollen repliziert werden?
- Wie viele Replikate sollen angelegt werden?
- Wo sollen die Replikate gespeichert werden? (Replica Placement Problem [OSS03c])

Das Modell besteht aus zwei Phasen: *Proactive* und *On-demand*. Die Proactive-Phase läuft vor der Anfrage an ein Objekt bei der Initialisierung eines Knotens ab. Die On-demand-Phase hingegen läuft zur Laufzeit ab. In der Proactive-Phase wird von jedem Objekt eine feste Anzahl von Replikaten angelegt. In

der On-demand-Phase werden Anzahl und Orte der Replikate für ein Objekt dynamisch bestimmt. Es wird von einem dynamischen Replikationsschema oder dynamischen Replaktionsmanagement [RIF02, WJH97] gesprochen. Ein Replikationsschema für ein Objekt ist die Menge an Knoten, die ein Replikat des Objektes speichert. Verschiedene Objekte können verschiedene Verfügbarkeitsanforderungen haben, die in vielen Systemen aus der Popularität bestimmt werden. Je populärer ein Objekt wird, das heißt je mehr Anfragen auf ein Objekt gestellt werden, desto mehr Replikate werden erzeugt. Das Replikationsverhalten wird dynamisch durch die Popularität beeinflusst. Zur Bestimmung der Orte, das heißt der Knoten zur Speicherung der replizierten Objekte, werden je nach Replikationsalgorithmus verschiedene Parameter, wie Kapazitäten (Speicherplatz, Bandbreite der Anbindung usw.) und Uptime der Knoten bestimmt und verwendet.

In den nächsten Abschnitten werden verschiedene Algorithmen vorgestellt, die meist in der On-demand-Phase eingesetzt werden können.

3.3.1 Heuristische Algorithmen

In [OSS03a] werden fünf heuristische Algorithmen zur Erzeugung eines Replikates auf einem Knoten vorgestellt.

1. Random: Ein Knoten wird zufällig ausgewählt.

2. HighlyUpFirst (HUF): Der Knoten mit der höchsten Uptime wird ausgewählt.

3. HighlyAvailableFirst (HAF): Nach der Berechnung der Verfügbarkeiten wird der Knoten mit der höchsten Verfügbarkeit ausgewählt. Die Verfügbarkeit (engl.: availability) wird als steuerbarer, überwachbarer Parameter des QoS definiert (Konzept des QoA (***Q****uality* ***o****f* ***A****vailibility*) [OSS01]).

4. Combined: In der Kombination aus HighlyUpFirst und HighlyAvailableFirst werden die Durchschnittswerte der Uptime und Verfügbarkeit für alle Knoten berechnet. Danach werden die Knoten ausgewählt, die beide Werte über dem Durchschnitt haben. Zwischen diesen wird dann erst der Uptime, dann der Durchschnittswert verglichen.

5. Local: Ein Knoten erzeugt oder ersetzt ein Replikat zur Laufzeit in seinem lokalen Speicher. Beim Ersetzen wird entweder der LRU (***L****ast* ***R****ecently* ***U****sed*) oder der MFU (***M****ost* ***F****requently* ***U****sed*) Algorithmus verwendet.

Der Local-Algorithmus kann mit den ersten vier Algorithmen kombiniert werden, ähnlich wie in den Top-K LRU/MFR-Algorithmen, die im folgenden Abschnitt beschrieben werden. Wird beispielsweise durch HighlyUpFirst ein Knoten zur Speicherung eines Replikates bestimmt, führt er den Local-Algorithmus zur Erzeugung des Replikates aus.

Die Algorithmen HighlyUpFirst, HighlyAvailableFirst und Combined benötigen globales Wissen über die Netzwerkstruktur, um die entsprechenden Knoten zur Speicherung der Replikate zu finden. In einem Netzwerk mit häufigen Ein- und Austritten von Knoten und sich dadurch ständig ändernder Struktur oder sehr großen Netzwerken mit Millionen von Knoten ist in der Regel kein solches globales Wissen vorhanden. Beim Random-Algorithmus und beim Local-Algorithmus wird nur lokales Wissen benötigt.

Das Replikationsschema ändert sich mit dem Nutzerverhalten. Je populärer ein Objekt wird, desto mehr Replikate werden erzeugt. Dabei werden nur Lesezugriffe betrachtet.

Für Updates wurden keine Vorkehrungen getroffen und Qualitätsbedingungen für Objekte sind nicht definierbar. Allerdings haben Qualitätsbedingungen, wie Latenz oder Kapazitäten, Einfluss auf die Platzierung der Replikate.

3.3.2 Top-K LRU/MFR

In [KRT02] werden der Top-K LRU und der Top-K MFR-Algorithmus vorgestellt. Beide hängen von der Popularität eines Objektes ab. Es gibt für jedes Objekt o eine Liste von Knoten $i_1, ..., i_K$, die als Speicherorte für o in Frage kommen. Diese Liste muss von dem darunter liegenden Lookup-Service, wie Tapestry [ZKJ01] oder Chord [SMK+01] bestimmt werden. Die Replikation eines Objektes erfolgt bei einer Anfrage auf dem Knoten, bei dem die Anfrage eingeht. Beim Top-K LRU-Algorithmus (Algorithmus 1, Abbildung 3.3) fragt ein

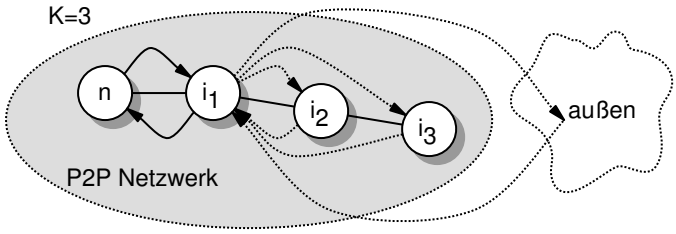

Abbildung 3.3: Der Top-K LRU-Algorithmus

Knoten n bei i_1 nach dem Objekt o. Wenn i_1 es gespeichert hat, erhält n das Objekt von i_1. Hat i_1 o nicht gespeichert, fragt i_1 die restlichen $K-1$ Knoten nach o. Wenn einer der Knoten o speichert, bezieht i_1 das Objekt, speichert ein Replikat in seinem lokalen Speicher und überträgt es an n. Andernfalls lädt i_1 o von einer externen Quelle, speichert auch ein Replikat oder gibt einen Fehler zurück, wenn es nicht verfügbar ist. Im Falle von Speicherproblemen wird der LRU-Algorithmus verwendet, um ein oder mehrere andere replizierte Objekte zu verdrängen. Es kann passieren, dass über dieses Verfahren ein Objekt auf einem Knoten gespeichert ist, aber bei einer Anfrage nicht gefunden wird, weil K

zu klein gewählt wurde, jedoch wird in [KRT02] gezeigt, dass die Wahrscheinlichkeit bei der richtigen Wahl von K gering ist.

1: n ermittelt i_1, den ersten möglichen Speicherort für o, mittels des zu Grunde liegenden Lookup-Services
2: n fragt i_1 nach o
3: **if** i_1 hat o nicht **then**
4: i_1 bestimmt $i_2, ..., i_K$ und fragt jeden der $K-1$ Knoten nach o
5: **if** einer der Knoten $i_2, ..., i_K$ hat o **then**
6: i_1 bekommt o von diesem Knoten
7: **else**
8: i_1 bekommt o von außen
9: **end if**
10: **if** der lokale Speicher von i_1 ist zu voll für o **then**
11: i_1 verdrängt Objekte nach dem LRU Prinzip, bis genug Platz für o vorhanden ist
12: **end if**
13: i_1 speichert eine Kopie von o in seinem lokalen Speicher
14: **end if**
15: i_1 übermittelt o an n

Algorithmus 1: Top-K LRU

Der Algorithmus bietet keine Updatestrategien. Mittels der durch den Lookup-Service gelieferten Listen von Knoten sind zwar einige mögliche Speicherorte für Replikate bekannt, jedoch sind es erstens nicht alle und zweitens besteht keine Sicherheit, dass ein Knoten ein Replikat speichert.

Das Replikationsverhalten ist wie bei den vorangegangenen Algorithmen abhängig vom Nutzerverhalten. Steigen die Lesezugriffe auf ein Objekt an, erweitert der Algorithmus das Replikationsschema. Schreibzugriffe werden nicht betrachtet.

Das benötigte Wissen ist annähernd lokal. Es wird nur die Liste der Speicherkandidaten für ein Objekt vom Lookup-Service benötigt.

Qualitätsbedingungen lassen sich bei Top-K LRU nicht definieren. Die Anzahl der Replikate je nach Popularität eines Objektes ist fest mit dem Algorithmus verankert. Die Wahl eines Wertes für K beeinflusst die Möglichkeit, ein repliziertes Objekt im Netzwerk zu finden, ohne es von außerhalb laden zu müssen. Dadurch kann die Antwortzeit gesenkt werden.

Mit dem Top-K MFR-Algorithmus (Algorithmus 2, Abbildung 3.4) kann eine bessere Antwortzeit des Systems erzielt werden. Das wurde durch Versuche in [KRT02] gezeigt. Bei diesem Verfahren speichert jeder Knoten i eine Tabelle über alle Objekte o, zu denen Anfragen eingegangen sind, und Schätzungen der Anfrageraten $\lambda_o(i)$. Im einfachsten Fall ist $\lambda_o(i)$ die Anzahl aller Anfragen auf Objekt o, die i erhalten hat, dividiert durch die Uptime von i. Jeder Knoten

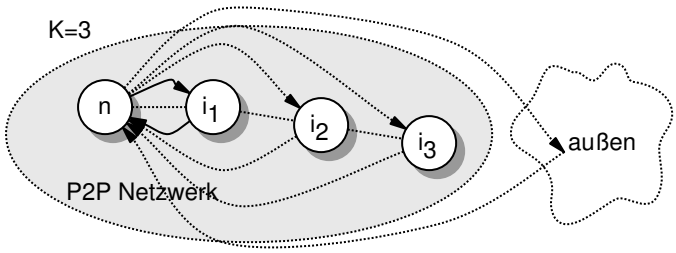

Abbildung 3.4: Der Top-K MFR-Algorithmus

i speichert die Objekte o mit der höchsten Anfragerate $\lambda_o(i)$, wobei so viele Objekte wie möglich gespeichert werden. Wenn eine Anfrage auf o bei i eingeht, wird $\lambda_o(i)$ aktualisiert. Muss i nach dem MFR-Algorithmus das Objekt o speichern, wird es von einer externen Quelle geladen und möglicherweise werden andere Objekte nach MFR verdrängt, wenn es nicht vorhanden ist. Will ein Knoten n ein Objekt o beziehen, stellt er sequentiell Anfragen an die Knoten i_1 bis i_K und stoppt, wenn sich das Objekt im lokalen Speicher eines der Knoten befindet. Dabei aktualisiert jeder Knoten sein $\lambda_o(i)$. Sollte n o nach K Schritten nicht erhalten haben, wird o von einer externen Quelle geladen, aber nicht im lokalen Speicher abgelegt.

```
1: k = 1
2: while k ≤ K und n hat o noch nicht bekommen do
3:     n bestimmt i_k mit Hilfe des Lookup-Services
4:     n fragt i_k nach o
5:     i_k aktualisiert λ_o(i_k)
6:     if i_k hat o then
7:         i_k überträgt o an n
8:     else if i_k hat o nicht, sollte es aber nach MFR haben then
9:         i_k bekommt o von außen
10:        if der lokale Speicher von i_k ist zu voll für o then
11:            i_k verdrängt Objekte o' mit niedrigen λ_o'(i_k)-Werten nach dem
                   MFR Prinzip, bis genug Platz für o vorhanden ist
12:        end if
13:        i_k speichert eine Kopie von o in seinem lokalen Speicher
14:        i_k überträgt o an n
15:    end if
16:    k = k + 1
17: end while
```

Algorithmus 2: Top-K MFR

Top-K MFR hat bis auf die bessere Geschwindigkeit die gleichen Eigenschaften wie Top-K LRU. Es werden nur Leseoperationen betrachtet, keine Schreiboperationen, um das Replikationsschema eines Objektes anzupassen. Die Wissensbasis ist die Liste möglicher Knoten für die Speicherung der Replikate eines Objektes. Updates werden nicht realisiert und Qualitätsbedingungen sind nicht definierbar. Eine sehr kleine Wahl von K kann allerdings die Anzahl der Replikate verringern und die Antwortzeit bei Objektanfragen negativ beeinflussen.

3.3.3 ADR

Ein weiterer Algorithmus namens ADR (***A****daptive* ***D****ata* ***R****eplication*) wird in [WJH97] vorgestellt. Dieser Algorithmus ändert das Replikationsschema in Abhängigkeit von den Lese- und Schreibzugriffen (Read-Write-Pattern) auf ein Objekt. Je nach Situation wird die Strategie angepasst (adaptives Verhalten), so dass sich das Schema sowohl vergrößern als auch verkleinern kann. Es wird versucht den Kommunikationsaufwand zu senken, um die Geschwindigkeit des Systems zu steigern. Da ein lokaler Lesezugriff schneller und billiger ist als ein Zugriff auf einen anderen Knoten im Netzwerk, sind viele Replikate in einem Netzwerk mit vielen Lesezugriffen von Vorteil. Bei einem Schreibzugriff müssen auch alle Replikate geschrieben werden. In einem Netzwerk mit vielen Schreiboperationen bedeuten viele Replikate deshalb einen hohen Kommunikationsaufwand. Der ADR-Algorithmus erhöht bei Lesezugriffen die Anzahl der Knoten im Replikationsschema, während diese bei Schreibzugriffen gesenkt wird. Es wird dabei kein gesamtes globales Wissen über die Netzwerkstruktur oder ein Replikationsschema vorausgesetzt. Das bedeutet, ein Knoten benötigt nur Informationen über seine Nachbarn. Der Algorithmus arbeitet nur auf Baumnetzwerken. Daher dürfen keine Kreise im Netzwerk enthalten sein. Bei einem Lesezugriff wird von dem nächsten verfügbaren Replikat gelesen. Bei einem Schreibzugriff werden alle Replikate aktualisiert. Ein Replikationsschema wird periodisch nach einem bestimmten Zeitraum von einigen Knoten geändert. Zur Bestimmung, welche Änderungen durchgeführt werden sollen, werden drei Tests verwendet: Der Expansionstest, der Kontraktionstest und der Schalttest. Der ADR-Algorithmus ist in Algorithmus 3 dargestellt.

Der *Expansionstest* wird auf jedem $\overline{R}$-Nachbarknoten n ausgeführt, wobei für jeden Nachbarn i von n überprüft wird, ob i mit in das Replikationsschema aufgenommen wird. Ein $\overline{R}$-Nachbarknoten ist ein Knoten, der im Replikationsschema R enthalten ist und einen oder mehrere Nachbarn hat, die nicht in R sind. Dabei werden die Leseanfragen auf das entsprechende Objekt von i nach n mit den Schreiboperationen verglichen, die n von allen Knoten (außer i) erhalten hat. Sind in der letzten Zeitperiode mehr Leseanfragen aufgetreten, wird i in das Schema aufgenommen und das Objekt zu i gesendet.

Der *Kontraktionstest* wird auf jedem Knoten n ausgeführt, der am Rand des Replikationsschemas liegt und damit ein R-Randknoten ist. Das heißt, er

```
 1: if n ist ein R̄-Nachbarknoten und kein R-Randknoten then
 2:     n führt den Expansionstest aus
 3: else if n ist ein R-Randknoten und kein R̄-Nachbarknoten then
 4:     n führt den Kontraktionstest aus
 5: else if n ist ein R̄-Nachbarknoten und ein R-Randknoten then
 6:     n führt den Expansionstest aus
 7:     if der Expansionstest schlägt fehl then
 8:         n führt den Kontraktionstest aus
 9:     end if
10: else if R = {n}, d.h. n hat keine Nachbarn, die zu R gehören then
11:     n führt den Expansionstest aus
12:     if der Expansionstest schlägt fehl then
13:         n führt den Schalttest aus
14:     end if
15: end if
```

Algorithmus 3: ADR

ist ein Blatt des Subgraphen des Schemas. Der Test entscheidet, ob n das Replikationsschema verlässt. Bei diesem Test werden die Schreiboperationen auf dem entsprechenden Objekt, die n von einem Nachbarn i während der letzten Zeitperiode erhalten hat, mit allen Leseanfragen (außer denen von i) verglichen. Überwiegen die Schreiboperationen, verlässt n das Schema. Ein R-Randknoten kann gleichzeitig auch ein $\overline{R}$-Nachbarknoten sein. In diesem Fall wird erst der Expansionstest und bei Fehlschlagen der Kontraktionstest durchgeführt.

Hat ein Knoten n aus R keine Nachbarn, die zu R gehören ($R = \{n\}$), führt er als erstes den Kontraktionstest durch und beim Fehlschlagen des Testes den *Schalttest*. Dieser überprüft, ob einer seiner Nachbarn besser geeignet ist, das Objekt o zu speichern. Wenn ja, wird o an denjenigen übertragen und gelöscht. Es wird die Anzahl aller Anfragen auf das Objekt o, die n von einem Nachbarn i erhalten hat, mit den restlichen Anfragen verglichen. Sind mehr Anfragen auf o von i als andere Anfragen eingegangen, war der Test erfolgreich und das Objekt wird zu i übertragen und gelöscht.

Ein Knoten braucht kein globales Wissen zur Ausführung des ADR-Algorithmus. Er muss seine Nachbarn kennen, muss wissen ob er zu einem Replikationsschema R gehört und ob seine Nachbarn zu R gehören. Im Gegensatz zu Top-K LRU und MFR beeinflusst das komplette Nutzerverhalten das Replikationsschema, sowohl die Schreib- als auch die Lesezugriffe. ADR realisiert Schreibzugriffe folgendermaßen: Ein Knoten aus R, der das Objekt o aktualisieren möchte, schickt eine Schreibanfrage an alle Nachbarn, die auch in R sind. Die Nachbarn leiten die Anfrage an all ihre Nachbarn aus R weiter. Dieser Vorgang wird wiederholt, bis alle Replikate aktualisiert sind. Diese Vorgehensweise wird auch *read-one-write-all* (dt.: eins lesen, alle schreiben) genannt. In Abschnitt 3.3.5

wird auf Updateoperationen näher eingegangen. Auch weniger strenge Anforderungen von Anwendungen an die Synchronisation der Schreiboperationen sind in Kombination mit ADR realisierbar. Je nach den Konsistenzanforderungen der Anwendung lässt sich ein geeignetes Verfahren zur Aktualisierung der Replikate einsetzen.

Qualitätsbedingungen lassen sich nur in Anwendungen definieren, die ADR verwenden, nicht aber im Algorithmus selbst.

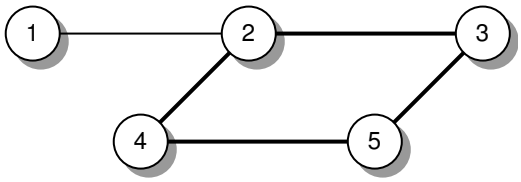

Abbildung 3.5: Ein Netzwerk mit einem Kreis

Der ADR-Algorithmus arbeitet nur auf Baumnetzwerken. Das bedeutet, das Netzwerk darf keine Kreise enthalten. Peer-to-Peer-Netzwerke enthalten allerdings in der Regel Kreise. Ein Beispiel für ein Netzwerk, das einen Kreis enthält, ist in Abbildung 3.5 dargestellt. Möglicherweise lässt sich der Algorithmus so modifizieren, dass er auf Netzwerken arbeitet, die Kreise enthalten. Das ist jedoch nicht Bestandteil dieser Arbeit.

3.3.4 D-Tree

In [CKK02] wurde ein System namens Dissemination Tree (D-Tree, dt.: Verteilungsbaum) entwickelt. In diesem System befinden sich alle Replikate in Multicast-Bäumen, über die Updates verteilt werden. Die Wurzeln der Bäume sind die Primärreplikate (engl.: primary replicas), bzw. die Knoten, die die Primärreplikate speichern. Das OceanStore System [BCE+99] verwendet D-Trees in seinem Replikationssystem. Das D-Tree-System wurde für Web-Inhalte entwickelt. Abbildung 3.6 stellt ein solches System dar. Server speichern Objekte und bilden ein Peer-to-Peer-Netzwerk untereinander. Clients fragen Server nach Objekten. Diese haben Latenzanforderungen, das heißt geringe Antwortzeiten der Server sollen gewährleistet sein und Server haben Kapazitätsgrenzen, wie Speicherplatz, Prozessorauslastung und Bandbreite. Die Anzahl der Replikate soll in diesem System minimiert werden, wobei sowohl die Latenzanforderungen erfüllt als auch die Kapazitätsgrenzen eingehalten werden sollen. In den Multicast-Bäumen sind die Clients die Blätter. Es werden zwei Algorithmen zur dynamischen Platzierung von Replikaten beschrieben: NaivePlacement (dt.: naive Platzierung) und SmartPlacement (dt.: schlaue Platzierung). Die verwendeten Variablen sind in Tabelle 3.2 dargestellt. Durch periodische „Refresh“-

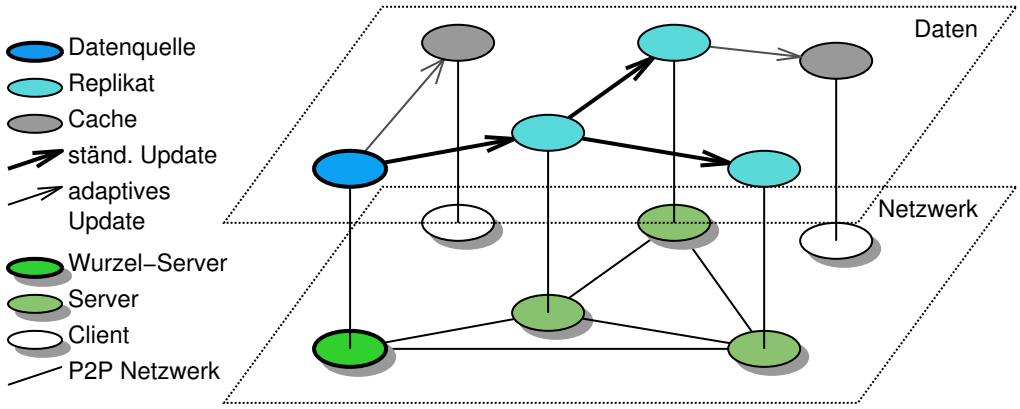

Abbildung 3.6: Ein Dissemination Tree System

Nachrichten kennt ein Vaterknoten die aktuellen Kapazitäten seiner Kinder.

Name	Bedeutung
d_c	die Latenzanforderung des Clients c
l_s	die Kapazitätsgrenze des Servers s
lc_s	die aktuelle Kapazitätsauslastung (CPU, Netzwerk) von s
$rc_s = l_s - lc_s$	die Restkapazität von s
$dist_{overlay}(c, s)$	die Entfernung zwischen c und s im Overlay-Netzwerk
$dist_{IP}(c, s)$	die IP-Entfernung zwischen c und s

Tabelle 3.2: Die Variablen der D-Tree-Algorithmen.

Beim NaivePlacement-Algorithmus (Algorithmus 4) sendet ein Client c eine Anfrage nach einem Objekt o. Die Anfrage routet der Lookup-Service zum dichtesten Server s, der ein Replikat von o speichert. Lassen die Kapazitäten von s es zu ($rc_s > 0$) und ist die Latenzanforderung von c erfüllt ($dist_{overlay}(c, s) \leq d_c$ oder $dist_{IP}(c, s) \leq d_c$), wird s Vater von c im D-Tree. Andernfalls wird auf dem zu c dichtesten Server s', der sowohl die Latenzanforderung von c erfüllt als auch seine Kapazitätsgrenze einhält, ein Replikat erzeugt und s' wird Vater von c im D-Tree. Der Server s zieht also nur sich selbst als Vater in Betracht und erzeugt ein neues Replikat, wenn eine Anforderung nicht erfüllt ist. Es kann allerdings durchaus ein Server existieren, der die Anforderungen erfüllt, so dass kein weiteres Replikat erzeugt werden muss. Auf Grund des kleinen Suchbereichs dieses Algorithmus wird dieser möglicherweise nicht gefunden.

Der SmartPlacement-Algorithmus (Algorithmus 5) versucht den besten Vater für einen anfragenden Client c zu finden. Dabei wird in einer größeren Menge

```
1: procedure NaivePlacement(c, o)
2:     c sendet eine Anfrage nach o an s, dabei werden von allen Servern s' auf
           dem Pfad zu s die IP-Adressen, rc_s' und dist_overlay(c, s') „aufgesam-
           melt“
3:     if rc_s > 0 then
4:         if dist_overlay(c, s) ≤ d_c then
5:             s wird Vater von c; exit
6:         else
7:             s bestimmt dist_IP(s, c)                          ▷ durch anpingen
8:             if dist_IP ≤ d_c then
9:                 s wird Vater von c; exit
10:            end if
11:        end if
12:    end if
13:    for all s' auf dem Pfad zu s, vom dichtesten zu c do
14:        if rc_s' > 0 und dist_overlay(c, s') ≤ d_c then
15:            s sendet ein Replikat an s'; s' wird Vater von c; exit
16:        end if
17:    end for
18:    for all s' auf dem Pfad zu s, vom dichtesten zu c, mit rc_s' > 0 do
19:        s' bestimmt dist_IP(s', c)                            ▷ durch anpingen
20:    end for
21:    c wählt s' mit kleinstem dist_IP(s', c) ≤ d_c
22:    s sendet ein Replikat an s'; s' wird Vater von c; exit
23: end procedure
```

Algorithmus 4: NaivePlacement

von Servern gesucht. Die Menge besteht aus dem Server selbst, seinen Kindern, seinem Vater und seinen Geschwistern. c wählt den Server mit der größten Restkapazität. Sollte keiner den Latenzanforderungen von c und den eigenen Kapazitätsgrenzen genügen, wird wie im NaivePlacement-Algorithmus durch s versucht, ein Replikat auf einem Server s' auf dem Pfad zwischen c und s zu erzeugen. In diesem Fall wird das Replikat allerdings so weit wie möglich weg von c platziert.

Der SmartPlacement-Algorithmus erzeugt mehr Verkehr bei einer Anfrage nach einem Objekt und dem darauf folgenden Eintritt eines Knotens in den D-Tree dieses Objektes, da durch die Suche nach einem Vaterknoten mehr Nachrichten versendet werden als beim NaivePlacement-Algorithmus. Auf der anderen Seite wird im Gegensatz zum NaivePlacement-Algorithmus ein Baum mit weniger Replikaten aufgebaut.

```
1:  procedure SmartPlacement(c, o)
2:      c sendet eine Anfrage nach o an s
3:      s sendet die IP von c an seinen Vater p und seine Kinder sc, wenn
            rc_sc > 0
4:      p leitet die Anfrage an die Geschwister von s sb, wenn rc_sb > 0
5:      s, sb, sc und p senden ihr rc zu c, wenn rc > 0
6:      if c bekommt eine Antwort then
7:          c wählt t als Vater mit dem größten rc_t und dist_overlay(c, t) ≤ d_c;
                exit
8:      else
9:          c sendet eine Nachricht zu s, dabei werden von allen Servern s' auf
                dem Pfad zu s die IP-Adressen, rc_s' und dist_overlay(c, s') „aufge-
                sammelt“
10:         for all s' auf dem Pfad zu s, vom dichtesten zu s do
11:             if rc_s' > 0 und dist_overlay(c, s') ≤ d_c then
12:                 s sendet ein Replikat an s'; s' wird Vater von c; exit
13:             end if
14:         end for
15:         for all s' auf dem Pfad zu s, vom dichtesten zu c, mit rc_s' > 0 do
16:             s' bestimmt dist_IP(s', c)                    ▷ durch anpingen
17:         end for
18:         c wählt s' mit größten dist_IP(s', c) ≤ d_c
19:         s sendet ein Replikat an s'; s' wird Vater von c; exit
20:     end if
21: end procedure
```

Algorithmus 5: SmartPlacement

Bei beiden Algorithmen lassen sich Qualitätsbedingungen, wie Latenz und Kapazitäten, definieren. In [CKK02] wird davon ausgegangen, dass sinnvolle

Werte für die Latenzanforderungen der Clients definiert werden, da das System sonst nicht funktioniert. Wählt man beispielsweise die Latenzanforderungen zu klein, können Clients nicht in die D-Trees eintreten.

Das Replikationsverhalten wird durch das Nutzerverhalten beeinflusst. Je mehr Anfragen also an Objekte gestellt werden, desto mehr Replikate werden erzeugt. Da das D-Tree System für Web-Inhalte entwickelt wurde, beeinflussen nur Leseanfragen das Replikationsschema, welches besonders durch SmartPlacement so klein wie möglich gehalten wird. Dadurch werden Änderungsoperationen nicht unnötig teuer. Updates sind mit Hilfe der D-Trees sehr einfach realisierbar, da alle Knoten des Schemas durch den Baum miteinander verbunden sind.

Die Wissensbasis, mit der die Algorithmen arbeiten, ist lokal. Wie bei ADR müssen nur Informationen über die Nachbarn bekannt sein. Zusätzlich speichert ein Knoten Vater und Kinder im D-Tree. Es muss nicht der gesamte D-Tree bekannt sein.

3.3.5 Update Protokolle

Ein Problem bei der Replikation ist das Ändern von Objekten. Solche Änderungen oder Updates müssen auf dem Objekt und auch allen Replikaten durchgeführt werden. Je nach Konsistenzanforderung gibt es verschiedene Verfahren, bei denen die Objekte unterschiedlich schnell aktualisiert werden.

In [GHOS96] werden zwei Formen der Replikation beschrieben, die Eager-Replikation (dt.: eifrig) und die Lazy-Replikation (dt.: träge). Bei der Eager-Replikation werden alle Replikate in einer atomaren Transaktion aktualisiert und bleiben somit exakt synchronisiert. Dadurch sinkt die Geschwindigkeit eines Updates, da zusätzliche Updates auf den Replikaten ausgeführt und zusätzliche Nachrichten versendet werden müssen. Durch Sperren wird Fehlern, wie zum Beispiel das gleichzeitige Updaten mit verschiedenen Änderungen auf verschiedenen Replikaten eines Objektes, vorgebeugt.

Die Lazy-Replikation hingegen führt die Updates auf den Replikaten asynchron nach dem Abschluss des Updates auf einem Objekt durch. Dadurch sind nicht immer alle Replikate gleich und es kann veraltete Versionen im Netzwerk geben. In einer Umgebung, in der Knoten oft nicht verbunden sind, eignet sich jedoch nur die Lazy-Replikation.

Weitere Strategien werden in [BD96] dargelegt. Die Verfahren variieren in der Anzahl der Replikate, die bei einer Updateoperation synchron aktualisiert werden, von einer bis alle Kopien. Es wird zwischen absolutistischen Verfahren und Abstimmungsverfahren unterschieden. Bei absolutistischen Verfahren wird die Synchronisation über eine ausgewählte Kopie (Primärkopie, engl.: primary copy) realisiert. Auch in [GHOS96] wird von einer Primärkopie gesprochen. Updates dürfen nur mit Zustimmung dieser Kopie ausgeführt werden. Bei den Abstimmungsverfahren erfolgt die Synchronisation über eine Abstimmung der

Kopien. Dazu erhält jede eine gewisse Stimmenanzahl. Ein Zugriff auf ein Objekt kann nur mit einer bestimmten Anzahl an Stimmen (Quorum) durchgeführt werden. Eine weitere Möglichkeit ist, zuzulassen, dass jeder Knoten mit einem Replikat ein Update ausführen darf (Gruppenreplikation). Ein Update wird so an alle Knoten mit einem Replikat gesendet und dort ausgeführt. Gleichzeitige unterschiedliche Updates auf einem Objekt müssen vom System erkannt und aufgelöst werden.

Alle Updateprotokolle sind mit dem ADR-Algorithmus und dem D-Tree-System kombinierbar.

3.4 Fazit

Alle betrachteten Algorithmen sind prinzipiell Caching-Algorithmen. Bei einem Zugriff auf ein Objekt wird das Objekt für den nächsten Zugriff in der Nähe des zugreifenden Knotens gespeichert. Dadurch verbessert sich die Zugriffszeit bei einem Folgezugriff und das System wird entlastet, da der Netzwerkverkehr eingeschränkt wird. Der Unterschied zwischen einem replizierten Objekt und einem einfach zwischengespeicherten Objekt (Cache) ist die Aktualisierung. Ein Replikat muss im Gegensatz zu Objekten im Cache zu anderen Replikaten synchronisiert werden. Wird ein Objekt aktualisiert, müssen die Änderungen auf alle Replikate übertragen werden. Im Cache gelagerte Objekte können je nach Verfahren von Zeit zu Zeit aktualisiert werden, oder gar nicht. Man kann Objekten im Cache eine Verfallszeit zuweisen. Diese wird periodisch überprüft und alle verfallenen Objekte gelöscht, oder sie können erneut geladen werden. In Tabelle 3.3 sind die betrachteten Algorithmen vergleichend nach den in Abschnitt 3.1 genannten Kriterien dargestellt. Sowohl der ADR-Algorithmus als auch das D-Tree System können Updates unterstützen. Damit unterscheiden sich die Algorithmen von Caching-Algorithmen. Da der ADR-Algorithmus nur auf Baumnetzwerken arbeiten kann, sind D-Trees die beste Wahl für das Replikationssystem in dem mobilen Informationssystem dieser Arbeit.

	Replikations-verhalten	QoS definierbar	Updates	Wissensbasis
Random	dynamisch nach Lesezugriffen	nein	nein	lokal
HUF, HAF, Combined	dynamisch nach Lesezugriffen	nein	nein	global
Top-K LRU/ MFR	dynamisch nach Lesezugriffen	nein	nein	lokal
ADR	adaptiv nach Lese- und Schreibzugriffen	nein	ja	lokal
D-Tree	dynamisch nach Lesezugriffen	ja	ja	lokal

Tabelle 3.3: Kriterien der Replikationsalgorithmen

Kapitel 4

Entwurf des mobilen Informationssystems

In diesem Kapitel wird das entwickelte mobile Informationssystem vorgestellt. Es wurde ein Konzept zum Aufbau und der Funktionsweise entwickelt. Die folgenden Abschnitte gehen auf die Zielsetzung und das Konzept genauer ein.

4.1 Ziel

Es gibt verschiedene Ansätze und Ziele von Peer-to-Peer Overlay Netzwerken. Systeme wie OceanStore [BCE+99], Intermemory [GY98] oder CFS [DKK+01] bieten die Funktionalität eines Dateisystems an. Einen anderen Ansatz bilden die weit verbreiteten File-Sharing Anwendungen, wie Napster [Nap], Freenet [CSWH01], KaZaA [KaZ] oder Gnutella [Gnu].

Im Zuge dieser Arbeit wurde ein mobiles Informationssystem entwickelt, das seinen Einsatz beispielsweise auf Konferenzen oder Messen finden kann. Auf solchen Veranstaltungen finden sich Wissenschaftler und interessierte Menschen zusammen, um Neuerungen auf verschiedenen Gebieten der Forschung vorzustellen bzw. sich über diese zu informieren. Heutzutage, nach dem Einzug der Mobilcomputer, reist ein Großteil der Teilnehmer mit ihren Notebooks an. Auf vielen Rechnern befinden sich für die Teilnehmer relevante Informationen, wie zum Beispiel wissenschaftliche Berichte oder Vorträge im XML- oder PDF-Format. Es sind auch Videos von TV-Berichten oder wissenschaftliche Abbildungen vorstellbar. Um diese Informationen austauschen zu können, müssen die Notebooks der Teilnehmer vernetzt werden. In solch einem Szenario bietet sich ein dezentralisiertes, strukturiertes Peer-to-Peer-Overlay-Netzwerk oder Ad-Hoc-Netzwerk an. Es existiert im Voraus kein Wissen über die Anzahl der Rechner im Netzwerk. Es gibt keine Server, daher macht ein zentraler Ansatz keinen Sinn. Das Netzwerk kann aus zwei Computern von sich gegenüber sitzenden Teilnehmern oder aus 1000 Computern bestehen. Die im Netzwerk gespeicherten Informationen sind vorher nicht bekannt. Zur Laufzeit des

Systems sollen diese Informationen angefragt werden können, um relevante Informationen zu finden. Da das Netzwerk spontan aufgebaut wird, das heißt ein Teilnehmer eröffnet ein Netzwerk und beliebige weitere Teilnehmer können diesem beitreten, muss mit vielen Netzein- und Austritten gerechnet werden. Tritt ein Computer dem Netzwerk bei und verlässt es nach kurzer Zeit wieder, sollen die auf ihm gespeicherten Informationen nach Möglichkeit im Netzwerk bleiben, bzw. soll bekannt sein, dass sie existieren. Ein Problem dabei ist die geringe Bandbreite, mit der ein Computer an das Netzwerk angeschlossen ist, da solche Netzwerke aus Mobilitätsgründen mit hoher Wahrscheinlichkeit per WLAN aufgebaut werden. Daher ist ein unstrukturierter Ansatz, wie bei Freenet, auf Grund des hohen Netzaufkommens bei Anfragen ungeeignet. Systeme wie OceanStore setzen auf sehr gut angebundene Server, eines so genannten *Inner Rings*, der die Daten speichert und bei Anfragen bereit stellt. Solche Server sollen hier nicht notwendig sein.

Als Zielsetzung hat das System eine hohe Verfügbarkeit der Informationen sowohl bei Netzwerkfehlern als auch bei sehr vielen Anfragen auf Objekte. Außerdem soll das System serverlos laufen und selbständig aufgebaut werden können.

4.1.1 Verfügbarkeit der Informationen

Eine hohe Verfügbarkeit der Daten und Informationen im Netzwerk kann durch Replikation erreicht werden. Es gibt verschiedene Umstände, die zu Einschränkungen der Verfügbarkeit von Objekten führen können. Zum einen kann ein Objekt sehr populär werden, d.h. es erlangt eine sehr hohe Beliebtheit und wird dementsprechend von sehr vielen Knoten im Netzwerk angefragt. Das Objekt wird zu einem so genannten HotSpot-Objekt [Rat02]. Im Szenario (siehe Kapitel 3) kann solch eine Situation beispielsweise direkt nach einem Vortrag eintreten. Die Folien des Vortragenden befinden sich nur auf dessen Computer und viele Hörer wollen diese zugleich herunterladen. Dadurch kann der Knoten, der das Objekt speichert, schnell überlastet werden, so dass viele Knoten das Objekt nicht beziehen können. Ein anderes Problem besteht, wenn der Knoten, der ein Objekt speichert, das Netzwerk verlässt. Existiert nur die eine Kopie des Objektes im Netzwerk, ist es nach dem Austritt nicht mehr verfügbar. Durch Replikation wird ein Objekt auf verschiedene Knoten verteilt. Dadurch wird bei vielen Anfragen kein Knoten überlastet, und wenn ein Knoten aus dem Netzwerk austritt, bleibt das Objekt weiterhin vorhanden. Ein anderer Grund für Replikation ist die Zugriffsgeschwindigkeit. Existieren mehrere Replikate eines Objektes im Netzwerk, kann der Ladevorgang von mehreren Knoten zugleich erfolgen und damit der Durchsatz und die Zugriffsgeschwindigkeit erhöht werden. Außerdem können Objekte in die Nähe anfragender Knoten platziert werden, so dass Folgeanfragen durch den verringerten Netzwerkverkehr schneller beantwortet werden.

In Kapitel 3 wurden verschiedene Replikationsstrategien vorgestellt. Objekte können beim Eintritt eines Knotens in das Netzwerk auf eine feste Anzahl von Knoten verteilt werden. Zur Laufzeit wird die Anzahl der Replikate je nach Popularität eines Objektes angepasst. Es stehen verschiedene Algorithmen zur Bestimmung der Anzahl der Replikate sowie der Auswahl der Knoten zur Speicherung zur Verfügung.

In dem hier konzipierten mobilen Informationssystem gibt es zwei entscheidende Probleme. Zum einen muss mit vielen Ein- und Austritten in bzw. aus dem Netzwerk gerechnet werden, und zum anderen ist die Bandbreite der Anbindung der Knoten an das Netzwerk gering. Dadurch kann eine feste Replikation jedes Objektes beim Netzeintritt (Proactive-Phase) das gesamte Netzwerk schnell überlasten, gerade bei größeren Objekten. Daher werden in diesem System Metainformationen zu jedem Objekt beim Eintritt verteilt. Diese Informationen werden in Form von Metaobjekten gespeichert, die an andere Knoten übertragen werden können. Zum einen sind diese Objekte klein, da sie nicht das Dokument selbst enthalten, und zum anderen liefert eine Anfrage einen Treffer, auch nach Austritt des speichernden Knotens. Zwar ist keine Volltextsuche möglich, beispielsweise auf XML-Dokumenten, allerdings kann in den Metadaten eine externe URL (***U****niform* ***R****esource* ***L****ocator*) gespeichert werden, von der das Objekt verfügbar ist. Steigt zur Laufzeit die Nachfrage nach einem Objekt, kann statt der Metadaten das Objekt repliziert werden. Abschnitt 4.2.3 beschreibt die verwendete Replikationsstrategie genauer.

4.1.2 Automatischer Netzwerkaufbau

Ein automatischer Aufbau des Netzwerkes bedeutet, dass Computer in einem WLAN automatisch ein Ad-hoc-Netzwerk untereinander aufbauen. Dazu muss die auf einem Rechner laufende Anwendung zuerst überprüfen, ob ein Netzwerk besteht, in das der Rechner eintreten kann. Sollte kein Netzwerk gefunden werden, wird ein neues eröffnet. Dieser Mechanismus kann mit Hilfe der Broadcasting-Technik[1] realisiert werden. Als Startpunkt wird das Subnetz eines zu Grunde liegenden IP-Netzwerkes verwendet, in dem sich ein Computer befindet. Beispielsweise, kann ein Accesspoint per DHCP die WLAN-Karte eines Notebooks konfigurieren, so dass eine Netzwerkverbindung zum IP-Netzwerk hergestellt wird. Über Broadcast-Nachrichten, die an alle Computer eines Subnetzes weitergeleitet werden, kann nach einem bestehenden Ad-hoc-Netzwerk des Informationssystems gesucht werden. Bekommt ein Computer keine Antwort, können im Voraus festgelegte Einstiegspunkte in anderen erreichbaren Netzwerken verwendet werden. Das können Computer sein, die bekanntermaßen sehr häufig online sind. Wird über Broadcasting oder die festgelegten Einstiegspunkte ein Computer gefunden, der mit dem Peer-to-Peer-Netzwerk verbunden

[1] Eine einzelne Nachricht an eine Broadcast-Adresse wird an alle Computer in dem Subnetz weitergeleitet.

ist, wird dieser als Gateway zum Netzwerkeintritt verwendet. Andernfalls wird ein neues Netzwerk gestartet.

4.2 Beschreibung des Systems

In diesem Abschnitt wird das mobile Informationssystem in seinem konzeptionellen Aufbau und seiner Funktionsweise beschrieben. Dafür werden die einzelnen Systemmodule vorgestellt.

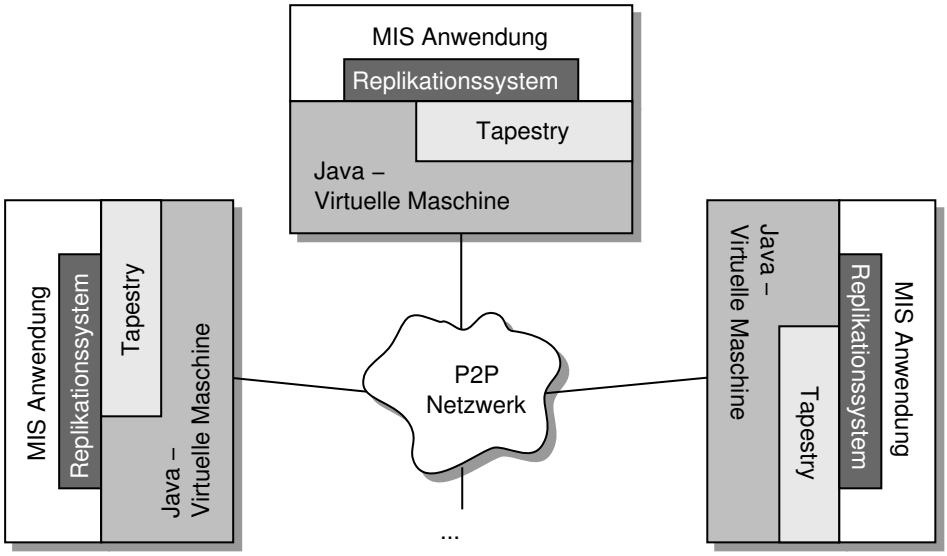

Abbildung 4.1: Der abstrakte Aufbau des Mobilen Informationssystems (MIS)

4.2.1 Modularer Aufbau des Systems

Das in dieser Diplomarbeit entwickelte mobile Informationssystem basiert auf dem Lookup-Service Tapestry [ZKJ01], dessen Funktionsweise in Abschnitt 2.1 beschrieben wurde. Die Anwendung selbst ist modular aufgebaut und wurde wie Tapestry in Java implementiert. Weitergehende Informationen zu Tapestry und seiner Verwendung werden in Kapitel 5 gegeben. Mit Hilfe von Tapestry wird ein Peer-to-Peer-Overlay-Netzwerk aufgebaut, wobei auf jedem Computer im Netzwerk die gleiche Anwendung läuft. Abbildung 4.1 stellt den modularen Aufbau der Anwendung in Zusammenhang mit einem Netzwerk dar. Sie besteht aus verschiedenen Subsystemen (siehe Abbildung 4.2), die über Nachrichten miteinander kommunizieren. Durch diese Struktur ist die Anwendung sehr leicht

erweiterbar. Um beispielsweise Bereichssuchanfragen zu realisieren, kann ein entsprechendes Modul entwickelt werden, das diese Funktionalität ermöglicht.

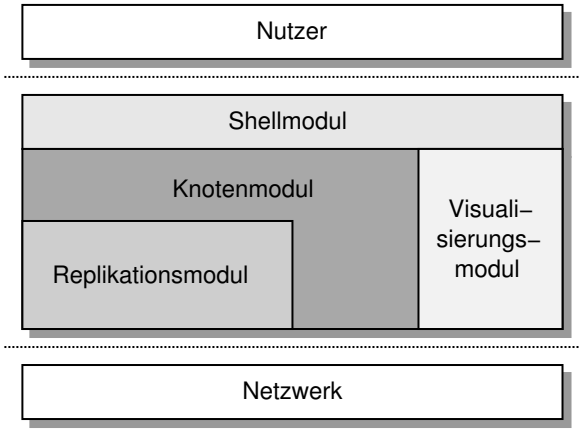

Abbildung 4.2: Die Module der Anwendung

Das Herzstück ist das Knotenmodul. Es verwaltet die lokalen Daten, übernimmt die Publikation der Objekte in das Netzwerk und bearbeitet Anfragen. Der Hauptteil der Kommunikation mit anderen Knoten wird hier gesteuert.

Das Shellsubsystem stellt eine textuelle Benutzerschnittstelle bereit. Damit kann die Anwendung gesteuert werden. Es können zum Beispiel Objekte lokalisiert oder Downloads gestartet werden. Für alle Operationen stehen Befehle bereit.

Das Visualisierungsmodul dient der grafischen Darstellung des gesamten Netzwerkes. Durch dieses Modul publiziert jeder Knoten ein Visualisierungsobjekt. Bei einer Anfrage eines Knotens an dieses Objekt antworten alle Knoten und senden ein Liste ihrer Nachbarn an den Anfragesteller. Mit diesen Daten generiert das Modul einen Graphen in Form von Colossus-Konfigurationsdateien. Colossus [Sch04] ist ein Datamining-Werkzeug zur Darstellung von Graphen, das parallel zu dieser Arbeit an der Universität Rostock entwickelt wurde. Es verwendet 3D-Technologien, wodurch eine sehr anschauliche und übersichtliche Visualisierung ermöglicht wird.

Das Replikationssubsystem steuert die Datenverteilung im Netzwerk. Es bestimmt, welche Objekte wie oft repliziert werden und wo die Replikate gespeichert werden. Außerdem sorgt es dafür, dass Updates ausgeführt werden können und Replikate nach Änderungen synchronisiert werden. In Abschnitt 4.2.3 wird die Replikation genauer beschrieben.

4.2.2 Funktionsweise

Jeder Computer hat ein zu Grunde liegendes lokales Dateisystem. Daraus werden verschiedene Verzeichnisse und Dateien publiziert, so dass aus globaler Sicht eine Verzeichnisstruktur entsteht. Ein Knoten bildet seine lokale Struktur auf die globale ab (siehe Abbildung 4.3). Jeder Computer hat Zugriff auf die glo-

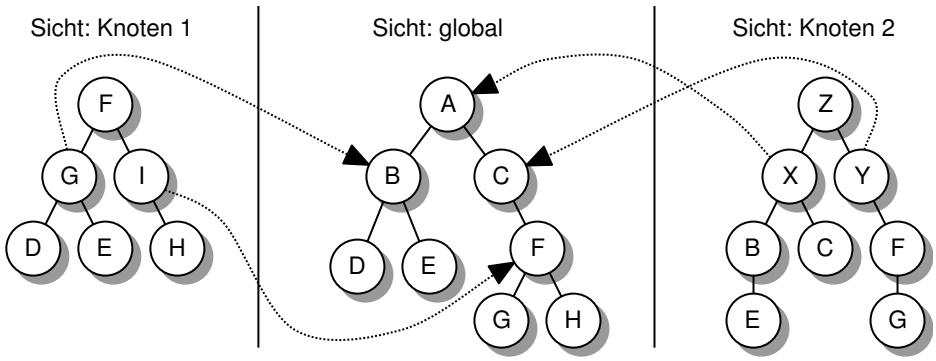

Abbildung 4.3: Lokale Dateistrukturen werden auf eine globale Dateistruktur abgebildet.

bale Struktur und kann Objekte herunterladen. Für jedes Objekt muss dazu ein Computer lokalisiert werden, der ein Replikat speichert.

Es werden XML- und PDF-Dokumente sowie reine Textdateien unterstützt. Zu jedem Dokument existieren Metadaten. In diesen Daten sind neben dem Dokumentnamen und der Größe ein Timestamp der letzten Änderung, eine Versionsnummer, eine externe URL und Zusatzinformationen enthalten. Die URL dient dazu, das Dokument möglicherweise von einer externen Quelle außerhalb des Peer-to-Peer-Netzwerkes zu laden, wenn es nicht verfügbar ist. In den Zusatzinformationen können Autor, Schlüsselwörter oder andere Daten bezüglich des Dokumentes gespeichert werden.

Die im Netzwerk publizierten Dokumente können gesucht und heruntergeladen werden. Es ist eine Navigation in Form von Browsing durch die globale Verzeichnisstruktur möglich. Verzeichnisinhalte können dabei aufgelistet werden, so dass darüber die Dateinamen von Objekten bestimmt werden können, die für den Download notwendig sind. Das System bestimmt daraus den dichtesten Knoten, der ein gesuchtes Objekt speichert, um es herunterzuladen. Bereichsanfragen oder Volltextsuchanfragen werden im Prototyp nicht unterstützt.

Das System legt automatisch Replikate einzelner Objekte im Netzwerk an und verteilt diese auf verschiedene Knoten. Dadurch soll die Verfügbarkeit des Inhaltes erhöht werden. In den nächsten Abschnitten wird die dafür verwendete Replikationsstrategie genauer vorgestellt.

4.2.3 Replikation

Das im Rahmen dieser Diplomarbeit entwickelte Replikationssystem basiert auf einem Zweiphasenmodell, das auf dem Modell aus [OSS03a] basiert. Der wesentliche Unterschied ist, dass in der ersten Phase - der Eintrittsphase - wie oben beschrieben statt den Objekten Metainformationen in Form von Metaobjekten verteilt, während in der zweiten Phase - der Laufzeitphase - die Objekte selbst repliziert werden.

Die Objekte werden bei der Replikation in Bäumen, so genannten Dissemination Trees (D-Tree), organisiert. Die eigentliche Idee dahinter stammt aus einem Web-System [CKK02]. Clients greifen hier auf Server zu, die durch ein Peer-to-Peer-Netzwerk miteinander verbunden sind. Das System wurde in Abschnitt 3.3.4 beschrieben. Die D-Trees werden in dieser Arbeit wie folgt verändert und eingesetzt:

Es gibt keine Unterscheidung zwischen Server- und Clientmaschinen. Die Rollen der Rechner im Netzwerk sind identisch. Jeder der Knoten kann sowohl Server als auch Client sein. Dadurch sind auch die Clientmaschinen im Peer-to-Peer-Netzwerk enthalten.

Auch in diesem System hat jeder Knoten einen Zwischenspeicher (Cache). Fragt ein Knoten in der Clientrolle ein Objekt an, wird je nach seinen Latenzanforderungen durch den SmartPlacement-Algorithmus (siehe Abschnitt 3.3.4) ein Replikat auf dem Pfad zwischen Client und Server erzeugt. Hat der anfragende Knoten das Objekt erhalten, wird eine Kopie in seinem Zwischenspeicher angelegt. Damit tritt er nicht aktiv in den D-Tree ein, sondern passiv als Blatt. Das bedeutet, das Objekt wird nicht als Replikat publiziert und es wird nicht durch den D-Tree aktualisiert. Der Knoten kann einen Verfallszeitraum definieren, nachdem ein Objekt neu geladen bzw. gelöscht werden muss.

Es wäre auch möglich bei jedem Download ein Replikat zu erzeugen. Jedoch soll die Anzahl gering gehalten werden, um Updates nicht zu aufwendig werden zu lassen. Aus demselben Grund wird statt dem NaivePlacement-Algorithmus der SmartPlacement-Algorithmus verwendet.

4.2.4 Ein- und Austritte bei der Replikation

Netzwerkeintritt

Tritt ein Knoten n in ein bestehendes Netzwerk ein, publiziert er alle lokalen Objekte o. Vorher wird für jedes Objekt überprüft, ob es bereits im Netzwerk vorhanden ist. Wenn es nicht vorhanden ist, wird es publiziert und der Knoten wird zum Wurzelknoten des D-Trees für das entsprechende Objekt o. Andernfalls, wenn es bereits im Netz vorhanden ist, wird eine Anfrage an den dichtesten Knoten i gesendet, der ein Replikat speichert, in den D-Tree einzutreten. Dadurch nimmt i n als Kind auf und wird Vater von n.

Die Verteilung der Metaobjekte erfolgt nur, wenn das Objekt nicht im Netzwerk existiert. Die Anzahl ist definierbar. Die Auswahl der Knoten erfolgt zufällig aus der Liste der Nachbarn. Jeder Knoten tritt als Kind in den D-Tree ein und erhält ein Metaobjekt, das die gleiche Objekt-ID wie das Objekt selbst hat. Dadurch können die Knoten mit Metaobjekten auch Anfragen auf das Objekt erhalten. Im Falle einer Download-Anfrage wird das Objekt vom Vater geladen und danach gesendet. Ist das Objekt nicht im Netzwerk vorhanden, da der publizierende Knoten es verlassen hat, bevor ein Replikat auf einem anderen Knoten erzeugt werden konnte, wird versucht das Objekt von außerhalb des Peer-to-Peer-Netzwerkes zu laden. Ist keine URL in den Metadaten angegeben oder besteht keine Verbindung in andere Netzwerke, wird eine Fehlermeldung zurückgegeben.

Netzwerkaustritt

Verlässt ein Knoten n das Netzwerk, müsste er seinem Vater p die Adressen seiner Kindknoten i senden, so dass p alle i als Kinder aufnimmt. Weiterhin müsste n allen Kindern i die Adresse seines Vaters p senden, so dass alle i p als neuen Vater eintragen können. Da die D-Trees, wie im folgenden Abschnitt beschrieben, erweitert wurden, sind diese Informationen bereits vorhanden. Dadurch braucht sich ein Knoten prinzipiell überhaupt nicht abmelden. Damit aber keine unnötigen Verzögerungen entstehen, meldet ein Knoten sich bei seinem Vater ab.

4.2.5 Fehlerbehandlung bei der Replikation

In regelmäßigen Abständen überprüft ein Knoten, ob seine Kinder und sein Vater noch im Netzwerk sind. Dazu sendet ein Vater an seine Kinder Ping-Nachrichten. Kommt in einer gewissen Zeitspanne keine Antwortnachricht, wird der Vorgang wiederholt. Nach beispielsweise drei Versuchen gibt der Vater auf und entfernt das entsprechende Kind. Bekommt ein Kindknoten eine bestimmte Zeit keine Ping-Nachricht vom Vaterknoten, entfernt dieser den Vaterknoten. Zu solchen Vorfällen kann es zum Beispiel durch Netzfehler, wie Störungen des WLANs, kommen. Auch Softwarefehler, wie Betriebssystemabstürze, können die Ursache sein. Meldet sich ein Knoten nicht ordnungsgemäß bei seinen Kindknoten und seinem Vaterknoten ab, wird der D-Tree gespalten. In den ursprünglichen D-Trees hat ein Knoten, der seinen Vater verliert, keinen Anhaltspunkt, einen Ersatz zu finden. Ein Vaterknoten, der ein Kind verliert, hat auch kein Wissen über mögliche Kindeskinder. Abbildung 4.4 zeigt einen solchen Fehlerfall. Knoten 3 verliert aus irgendeinem unvorhersehbaren Grund die Verbindung zum Netzwerk. Knoten 1 kennt weder 4 noch 5 und umgekehrt.

Um den Baum nach einem Fehlerfall wieder „zusammenzusetzen“, wird in diesem System der Wissensbereich eines Knotens um die Austrittsinformationen

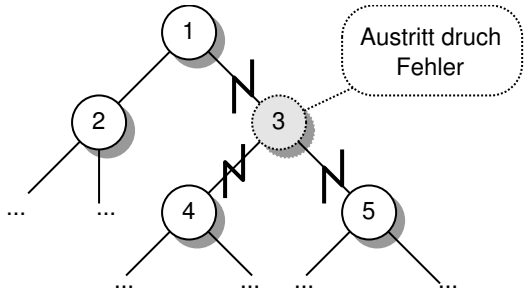

Abbildung 4.4: Die Aufspaltung eines D-Tree durch einen Fehler

eines Kindes, seines Großvater und seiner Geschwister erweitert. Ein Vaterknoten kennt neben seinen Kindern die Kindeskinder und ein Kindknoten kennt neben seinem Vater den Großvater und die Geschwister. Der Austausch der Informationen wird beim Eintritt eines Knotens in den D-Tree und bei Änderungen mit Hilfe der Ping-Nachrichten vorgenommen. Die Übermittlung der Informationen beim Austritt eines Knotens wird dadurch überflüssig. Tritt ein Knoten in einen D-Tree ein, teilt der Vater ihm seinen Großvater und seine Geschwister mit. Hat der eintretende Knoten bereits Kinder, werden diese an den Vater übermittelt [2]. Kommt beispielsweise ein neuer Geschwisterknoten hinzu, wird diese Information in die nächste Ping-Nachricht eingebettet. Bekommt ein Kindknoten einen eigenen Kindknoten dazu oder verliert einen, wird diese Information in die nächste Antwortnachricht eingebettet. Im Beispiel aus Abbildung 4.4 würde Knoten 1 dadurch die Knoten 4 und 5 kennen und diese als neue Kinder aufnehmen.

Wenn der Wurzelknoten ausfällt, müssen sich die Geschwister untereinander einigen, wer die neue Wurzel wird. Dazu sendet jeder Knoten seinen Geschwistern seine Uptime. Der am längsten verbundene Knoten wird die neue Wurzel.

Treten mehrere zusammenhängende Knoten gleichzeitig aus, beispielsweise 3 und 4, reichen auch die Zusatzinformationen nicht aus, um den D-Tree nach Knoten 4 an Knoten 1 zu binden. Der Wurzelknoten r des Baumes nach 4 muss versuchen einen neuen Vater zu finden. Dazu könnte r eine Lokalisierungsnachricht nach dem zum D-Tree gehörenden Objekt senden, bei der nicht nur der dichteste Knoten antwortet, sondern alle Knoten, die ein Replikat speichern. Von allen antwortenden Knoten wählt r den dichtesten Knoten aus, der nicht in seinem D-Tree enthalten ist, und sendet eine Beitrittsanfrage an diesen. Dazu müsste r allerdings seinen kompletten D-Tree kennen. Besser wäre es, eine Nachricht zu senden, auf die nur die Wurzel eines D-Trees antwortet. Bekommt r eine Antwort, wird eine Beitrittsanfrage an die Wurzel des anderen Baumes gesendet. Im folgenden Kapitel wird detailliert auf die Reaktionen des Systems

[2]Dieser Fall kann bei einem Fehler auftreten, wenn zwei D-Trees gekoppelt werden sollen.

auf Fehlersituationen eingegangen.

Da der Austausch der Informationen über Ein- und Austritte von Knoten in und aus dem D-Tree in diesem System über die Ping-Nachrichten realisiert wird, läuft dieser abgekoppelt von den Ein- und Austritten ab. Dadurch wird das System bei vielen gleichzeitigen Änderungen der Baumstruktur nicht überlastet. Andernfalls müssten bei jedem Eintritt bzw. Austritt eines Knotens aus dem D-Tree viele zusätzliche Nachrichten versendet werden, die gerade bei geringen Bandbreiten stark von Nachteil wären.

4.2.6 Änderungsoperationen

Bei der Replikation entsteht das bereits in Abschnitt 3.3.5 beschriebene Problem der Updates. Nach der Änderung eines Objektes müssen auch alle Replikate aktualisiert werden.

In dem mobilen Informationssystem werden die Updates über eine Primärkopie synchronisiert. Dabei wird die Lazy-Replikation verwendet, um die Antwortzeit des Systems niedrig zu halten. Ein Knoten, der ein neues Objekt in das Netzwerk einführt, behält so die volle Kontrolle darüber. Soll das Objekt beispielsweise nur lesbar sein, werden keine Updates zugelassen.

Da der erste publizierende Knoten automatisch die Wurzel des D-Trees dieses Objektes wird, befindet sich jede Primärkopie eines Objektes auf dem Wurzelknoten. Verlässt die Wurzel das Netzwerk, übernimmt der neue Wurzelknoten die Attribute der Primärkopie und die alte Wurzel verliert die Kontrolle.

Will ein Knoten ein Objekt aktualisieren, fragt er bei der Wurzel des D-Trees nach. Dazu wird die Versionsnummer des Objektes über den Vaterknoten bis zur Wurzel durchgereicht. Nur wenn die Version aktuell ist und ein schreibender Zugriff erlaubt ist, sendet die Wurzel eine Genehmigung. In diesem Fall wird das Update an die Wurzel übertragen und ausgeführt.

Die Aktualisierung der Replikate erfolgt verzögert. Periodisch überprüft ein Knoten die Versionsnummer seines Replikates, indem diese mit der des Vaters verglichen wird. Sollte sein Replikat veraltet sein, wird es aktualisiert. Die periodische Überprüfung wird auch mit Hilfe der Ping-Nachrichten durchgeführt, da eine Ping-Nachricht die Versionsnummer des Objektes enthält.

4.2.7 Aufwand bei der Replikation

Durch den Einsatz eines Replikationsmechanismusses mittels D-Trees werden beim Publizieren, Anfragen und Aktualisieren zusätzliche Nachrichten versendet. Außerdem entsteht durch die Erzeugung zusätzlicher Objektkopien ein höherer Speicheraufwand.

Das Versenden einer Nachricht bezieht sich immer auf Sender und Empfänger und nicht auf die Zwischenknoten, über die die Nachricht geroutet wird. Unter der Voraussetzung intakter Routing-Tabellen der Knoten, benötigt eine

Nachricht maximal $log(N)$ Schritte bis zum Ziel, wobei N die Größe des Namensraumes der IDs ist [ZKJ01]. Aus diesem Grund haben alle Operationen einen Aufwand von $O(log(N))$.

Zusätzlich zur normalen Verwaltung der lokalen Objekte müssen zu jedem Objekt Informationen über die Position des Knotens im D-Tree gespeichert werden. Darin enthalten sind die Adressen aller Kinder und Kindeskinder, der Geschwisterknoten, des Vaters und des Großvaters. Eine Adresse besteht aus einer IP-Adresse und einem Port. Außerdem werden periodische Ping-/Pong-Nachrichten zwischen den Knoten eines D-Trees zu seiner Aufrechterhaltung und zur Verbreitung von Updates versendet. In diesen Nachrichten sind die Adressdaten und die Versionsnummer des Objektes enthalten.

Publikation von Objekten

Es gibt zwei Fälle, die bei der Publikation eines Objektes o eintreten können. Entweder o existiert bereits im Netzwerk, da es bereits von einem anderen Knoten publiziert wurde oder es wird das erste Mal publiziert. Existiert es, muss der Knoten in den bestehenden D-Tree eintreten. Dazu werden 2 Nachrichten benötigt. Existiert das Objekt nicht, müssen Metainformationen zu einer festgelegten Anzahl von Nachbarn versendet werden. Dafür werden zwei Nachrichten pro Nachbar benötigt. Bei einem Standardwert von 3 sind das in der Regel 6 Nachrichten.

Für die eigentliche Publikation wird eine Publikations-Nachricht benötigt und 2 weitere zur Bestimmung der Existenz des Objektes. Mit Standardeinstellungen werden daher 4-8 Nachrichten versendet, um ein Objekt im Netzwerk zu publizieren.

Anfragen von Objekten

Bei einer Anfrage eines Objektes o wird je nach Latenz des speichernden Knotens s zum anfragenden Knoten n ein Replikat erzeugt. Dazu wird mit dem SmartPlacement-Algorithmus ein Knoten bestimmt, auf dem ein Replikat angelegt werden soll. Zusätzlich zur Anfragenachricht wird jedem Geschwisterknoten von s, dem Vater und den Kindern mit je einer Nachricht die IP von n mitgeteilt. Je mehr Replikate eines Objektes existieren, desto mehr Knoten befinden sich im D-Tree und desto mehr Nachrichten müssen versendet werden. Die Knoten senden n ihre Kapazitäten. Erfüllt keiner der Knoten die Latenzanforderungen oder hat Restkapazitäten, muss ein Replikat von o erzeugt werden. Dazu wird ein Knoten zwischen n und s gewählt. Bis zur Publikation des neuen Replikates müssen 3 Nachrichten versendet werden und eine weitere zur Übermittlung von o nach n.

Änderungsoperationen auf Objekten

Will ein Knoten n ein Update auf einem Objekt o ausführen und ist er nicht der Wurzelknoten des zugehörigen D-Trees, muss n mit diesem in Kontakt treten. Zur Ausführung der eigentlichen Änderungsoperation werden 3 Nachrichten benötigt. Die Verbreitung des Updates auf die restlichen Replikate erfolgt mit Hilfe der periodischen Ping-Nachrichten. Ein weiterer Aufwand entsteht erst, wenn ein neues Objekt von einem Vater zu einem Kind übertragen wird. Dafür werden 2 Nachrichten benötigt, wovon eine das komplette Objekt enthält und demnach meist größer als übliche Nachrichten ist.

Kapitel 5

Realisierung des mobilen Informationssystems

In diesem Kapitel werden Details der Implementierung und genauen Arbeitsweise des Prototypen beschrieben. Die Software ist in Java entwickelt. Auf jedem Computer des mobilen Informationssystems läuft das gleiche Programm und alle benötigen einen Netzwerkzugang. Dadurch wird ein Peer-to-Peer-Overlay-Netzwerk aufgebaut. Als zu Grunde liegendes System zur Lokalisierung von Objekten und zum Routing von Nachrichten wird Tapestry verwendet.

5.1 Tapestry

Die Funktionsweise des Lookup-Service Tapestry wurde bereits in Abschnitt 2.1 erläutert. Tapestry ist in Java implementiert, wobei ein ereignisorientiertes Programmiermodell verwendet wird. Das Modell basiert auf der asynchronen I/O Bibliothek SEDA/Sandstorm [WCB01, Wel].

Die Anwendung teilt sich in verschiedene Threads auf, die Stages genannt werden. Jede Stage wird als Klasse implementiert. Alle Threads einer Anwendung laufen in einer einzigen JVM (***J**ava **V**irtual **M**achine*), wobei die Kommunikation zwischen den Stages in und außerhalb einer JVM über Nachrichten und Ereignisse läuft. Nach einigen Initialisierungen tritt jede Stage in eine Ereignisschleife ein. Mit Hilfe eines Verteilers in jeder JVM werden Nachrichten und Ereignisse an Stages weitergeleitet. Dazu muss sich jede Stage beim Verteiler anmelden und die Ereignistypen übergeben, die erhalten werden sollen.

In einer Konfigurationsdatei wird in XML-ähnlicher Syntax angegeben, welche Stages zu einer Anwendung gehören. Diese Datei liest SEDA und instanziert alle Threads mit den entsprechenden Initialisierungsparametern. Ein Beispiel für eine Stage, die in der Klasse *MeineAnwendung.Stages.MeineStage* implementiert ist und an die ein Parameter *param* mit dem Wert 5 übergeben wird, kann folgendermaßen aussehen:

```
<MeineStage>
  class MeineAnwendung.Stages.MeineStage
  <initargs>
    param 5
  </initargs>
</MeineStage>
```

Zu Tapestry gehören eine Reihe von Stages. Um eine Tapestry-Anwendung laufen zu lassen, die ein Netzwerk aufbaut, sind folgende Stages nötig und müssen in der Konfigurationsdatei spezifiziert werden:

- **Network:** Die Network Stage ist für die Kommunikation über das Netzwerk zuständig. Es werden Nachrichten über das Netzwerk versendet und empfangen.

- **Router:** Das Herzstück eines Tapestry-Knotens ist die Router Stage. Von dieser wird die gesamte Nachrichten- und Ereignisverarbeitung übernommen. Es werden Nachrichten anderer Knoten und interne Ereignisse in einer JVM akzeptiert. Das Publizieren und Lokalisieren von Objekten im Netzwerk und das Routing von Nachrichten wird hier verwirklicht.

- **TClient:** Diese Stage ist für den Aufbau und die Verwaltung der Routing-Tabelle eines statischen Knotens zuständig. Es wird zwischen statischen und dynamischen Knoten unterschieden. Statische Knoten dienen dem Start eines neuen Netzwerkes, während dynamische Knoten in ein existierendes Netzwerk eintreten.

  In einer statischen Konfiguration wird ein so genannter Vereinigungsknoten (*Federation*) benötigt, um den Aufbau des Netzwerkes zu synchronisieren. Dieser Knoten wird für eine bestimmte Anzahl von statischen Knoten konfiguriert. Jeder statische Knoten muss den Vereinigungsknoten kennen und sendet beim Start eine „Helo“-Nachricht. Nachdem sich alle erwarteten Knoten beim Vereinigungsknoten gemeldet haben, sendet er an alle eine Liste aller statischen Knoten. Diese pingen sich gegenseitig an um ihre Routing-Tabellen aufzubauen. Nach Abschluss sendet jeder Knoten eine Nachricht an den Vereinigungsknoten. Hat dieser von allen eine Nachricht erhalten, sendet er an alle eine Startnachricht, so dass diese mit der normalen Arbeit beginnen können.

- **DTClient:** Die Routing-Tabelle eines dynamischen Knotens wird von der DTClient Stage verwaltet. Diese sorgt für einen Eintritt in ein bestehendes Netzwerk über einen Gateway. Ein Gateway ist ein Knoten eines bestehenden Tapestry-Netzwerkes, den ein eintretender Computer kennen muss.

- **Patchwork:** Mit Hilfe der Patchwork Stage werden Daten zu anderen Knoten bestimmt. Beispielsweise können die Latenz, die Bandbreite und der Paketverlust gemessen werden.

Zur Kommunikation stellt Tapestry eine API bereit, in der eine Reihe von abstrakten Nachrichtenklassen enthalten sind. Benötigte Nachrichten müssen für eine Anwendung implementiert werden. Häufig verwendete Nachrichten sind beispielsweise

- die **TapestryPublish-Nachricht**, um Objekte im Netzwerk zu publizieren,
- die **TapestryUnpublish-Nachricht**, um Objekte aus dem Netzwerk zu entfernen und
- die **TapestryLocate-Nachricht**, um einen Knoten zu finden, der ein Objekt speichert.

Nachdem eine Stage initialisiert wurde, sendet SEDA das StagesInitialized-Ereignis. Danach beginnt ein Knoten, sich ins Netzwerk zu integrieren. Nach dem Aufbau der Routing-Tabelle und dem Abschluss des Integrationsprozesses sendet die TClient/DTClient-Stage die TapestryReady-Nachricht. Nach dem Erhalt können die Anwendungs-Threads beginnen.

5.2 Komponenten des Systems

In Abschnitt 4.2.1 wurden die Komponenten der entwickelten Anwendung bereits vorgestellt. In diesem Abschnitt wird auf die Details eingegangen.

5.2.1 Knotenmodul

Der eigentliche Knoten ist in der Klasse *Node* implementiert. Nach der Initialisierung und dem Erhalten der TapestryReady-Nachricht publiziert die Stage alle Objekte, die lokal gespeichert sind. Dazu wird eine Abbildungsdatei gelesen, die die Abbildung lokaler Verzeichnisse auf globale Verzeichnisse enthält. Eine Abbildung sieht folgendermaßen aus:

```
/lokaler/pfad/A -> /globaler/Pfad/B
```

Ein lokales Verzeichnis A wird auf ein globales Verzeichnis B abgebildet. Dadurch erfolgt die Publikation aller Dateien und Verzeichnisse aus A.

Es werden zwei verschiedene Objekttypen publiziert, Dateiobjekte und Verzeichnisobjekte. Dateien liegen in Verzeichnissen, deshalb ist jedes Dateiobjekt mit einem bestimmten Verzeichnisobjekt assoziiert. Da eine Verzeichnishierarchie vorhanden ist, ist jedem Verzeichnisobjekt ein Vaterverzeichnis zugeordnet.

Dadurch entsteht eine zweischichtige Architektur, die in Abbildung 5.1 grafisch dargestellt ist. Die Strukturschicht wird durch die Verzeichnisobjekte gebildet, die Zeiger auf weitere Verzeichnisobjekte sowie auf Dateiobjekte haben, durch die die Datenschicht gebildet wird.

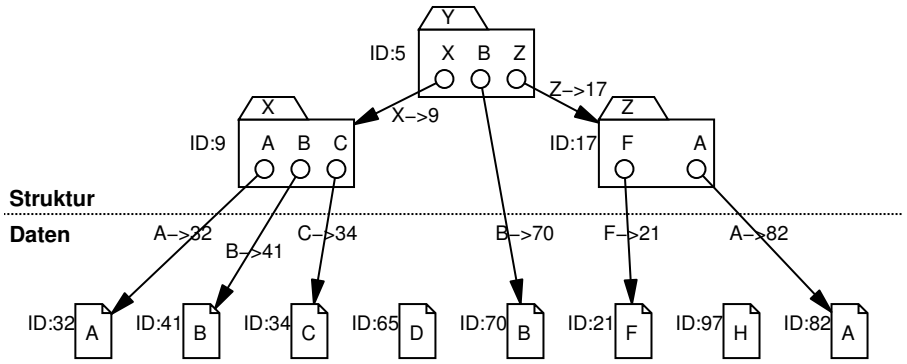

Abbildung 5.1: Die Schichten der Daten

Zur Publikation berechnet die Stage für jedes Objekt per SHA-1 eine eindeutige ID (GUID, ***G**lobal **U**nique **Id**entifier*) und erzeugt ein Metaobjekt. Dieses wird per ObjectPublish-Nachricht an das Replikationsmodul gesendet.

Die Stage reagiert auf die ObjectLocate-Nachricht und die ListDir-Nachricht[1], die in gleichnamigen Klassen implementiert sind. Ein Knoten erhält eine ObjectLocate-Nachricht, wenn er ein bestimmtes Objekt publiziert hat. Als Reaktion wird eine ObjectFound-Nachricht an den Absender geschickt. Eine ObjectLocate-Nachricht kann sowohl für ein Dateiobjekt als auch für ein Verzeichnisobjekt eingehen. Anders verhält es sich bei der ListDir-Nachricht. Diese kann nur für Verzeichnisobjekte eingehen. Erhält ein Knoten diese Nachricht, wird eine Liste von Datei- und Verzeichnisobjekten, die in dem angefragten Verzeichnis enthalten sind, an den Absender übertragen. Die Liste wird in Form von ObjectFound-Nachrichten versendet, wobei für jedes Element eine Nachricht geschickt wird.

Eine ObjectLocate-Nachricht wird zu dem vom Absender dichtesten Knoten geroutet, der ein Replikat des gesuchten Objektes enthält. Dieses Verhalten ist für die ListDir-Nachricht nicht wünschenswert, da in der globalen Verzeichnisstruktur mehrere Knoten ein lokales Verzeichnis auf ein und dasselbe globale Verzeichnis abbilden können. Will ein Knoten den Inhalt solch eines globalen Verzeichnisses auslesen, würde er nur den Inhalt des dichtesten Knotens, der auf das globale Verzeichnis abbildet, erhalten. Aus diesem Grund muss eine

[1]Diese Nachricht wird durch die Shell-Stage nach einem *ls*-Kommando (siehe 5.2.2) versendet.

ListDir-Nachricht an alle Knoten weitergeleitet werden, die ein entsprechendes Verzeichnisobjekt speichern.

Ein anderes Problem tritt bei der Vergabe der Identifikatoren für die Objekte auf. Jedes Objekt muss eine eindeutige ID haben, die per SHA-1 bestimmt wird. Werden die Namen der Objekte wie Verzeichnisnamen oder Dateinamen verwendet, kann es keine zwei gleichnamigen Objekte, wie in folgendem Beispiel, geben.

```
/usr/A
/usr/local/A
```

Dadurch ist es nicht möglich, Dateiobjekte verschiedenen Inhaltes mit gleichem Namen in verschiedenen Verzeichnissen anzulegen. Für Verzeichnisobjekte kann an dieser Stelle der absolute Pfad zur Berechnung verwendet werden, jedoch ändert sich dann die ID, wenn ein Verzeichnis verschoben wird. In jedem Fall ändert sich die ID eines Objektes bei seiner Umbenennung. Um diesem Verhalten vorzubeugen, müssen andere Informationen zur Berechnung der Identifikatoren verwendet werden. Im Prototyp werden für Dateiobjekte die MD5-Summen gebildet, während für Verzeichnisobjekte der Erzeugungszeitstempel verwendet wird. Die Verwendung von Zeitstempeln birgt eine erhöhte Gefahr von Kollisionen, ist jedoch für die prototypische Implementierung ausreichend. In späteren Versionen kann dieses Detail verbessert werden.

Durch die Verwendung des Inhaltes zur ID-Berechnung bei Dateiobjekten, ändert sich eine Objekt-ID mit einer Änderungsoperation. Im Szenario kann der Fall eintreten, dass ein Autor ein Dokument im Netzwerk publiziert und nach der Erzeugung einiger Replikate das Netzwerk verlässt. Danach nimmt er im unverbundenen Zustand Änderungen am Dokument vor und verbindet sich wieder. Er würde nun die neue Version mit neuer ID publizieren, während die alten Replikate nicht aktualisiert werden. Um eine Synchronisation der Replikate vornehmen zu können, muss die alte ID des Objektes bekannt sein, mit der ein Eintritt in den D-Tree des Objektes stattfinden kann. Dazu wird die Abbildungsdatei erweitert, so dass alle bei der letzten Verbindung freigegebenen Objekte mit den entsprechenden Identifikatoren gespeichert werden.

Ein Objekt kann allerdings auch im Netzwerk aktualisiert werden, während ein Knoten mit einer Kopie offline ist. Um einen Wiedereintritt in den D-Tree und die Aktualisierung des Objektes zu sichern, muss die alte ID eines Objektes im Netzwerk bleiben, auch wenn das Objekt selbst auf dem Knoten aktualisiert wurde. Ein Knoten speichert dazu die lineare Versionsfolge in einer Historie ab. Geht eine Anfrage mit einer alten ID ein, wird die neue Version des Objektes übertragen. Nach dem Wiedereintritt eines Knotens mit einer alten Objektversion wird das Objekt aktualisiert (siehe 5.2.4).

Nachdem die Publikation aller lokalen Dateien und Verzeichnisse abgeschlossen ist, wird eine NodeReady-Nachricht an Stages der gleichen JVM versendet.

5.2.2 Shellmodul

Die textuelle Benutzerschnittstelle wurde in der Klasse *Shell* implementiert. Nach der Initialisierung wartet die Stage auf die NodeReady-Nachricht vom Knotenmodul.

Die Klasse besitzt eine eingebettete Klasse namens ShellReader. Diese Klasse implementiert einen Thread, der die Benutzereingaben von einer Konsole liest.

Nachdem die NodeReady-Nachricht eingegangen ist, wird der ShellReader-Thread gestartet. Einem Benutzer stehen dann verschiedene Befehle, ähnlich wie bei einer Unix-Shell, zur Verfügung. Der ShellReader liest die Befehle und führt den entsprechenden Befehls-Handler aus. Handelt es sich um Aufgaben, die andere Knoten betreffen, werden Nachrichten per Tapestry versendet. Die Shell-Stage nimmt entsprechende Antwortnachrichten vom Netzwerk entgegen, wertet diese aus und macht sinnvolle Nutzerausgaben.

Der einfachste Befehl an einer Shell ist das *locate*-Kommando. Der Nutzer übergibt einen gesuchten Datei- oder Verzeichnisnamen, aus dem der Identifikator per MD5 und SHA-1 berechnet wird. Mit dieser ID wird eine ObjectLocate-Nachricht versendet. Danach wartet die Stage auf die ObjectFound-Nachricht oder die TapestryLocateFailure-Nachricht. Die ObjectFound-Nachricht enthält den dichtesten Knoten, der das gesuchte Objekt speichert. Wird eine Tapestry-LocateFailure-Nachricht empfangen, existiert das Objekt nicht im Netzwerk.

Durch das *ls*-Kommando wird eine ListDir-Nachricht versendet. Das Kommando dient der Auflistung des Verzeichnisinhaltes eines globalen Verzeichnisses. Dazu übergibt der Nutzer auch hier den gesuchten Namen. Allerdings muss es sich um ein Verzeichnis handeln. Die Nachricht erreicht jeden Knoten mit dem gesuchten Verzeichnis. Das Knotenmodul sendet daraufhin für jedes enthaltene Objekt eine ObjectFound-Nachricht. Aus allen eingegangenen ObjectFound-Nachrichten ergibt sich der gesamte Inhalt eines globalen Verzeichnisses.

Der Download eines Objektes kann durch den *get*-Befehl eingeleitet werden. Als Parameter muss der Name des gesuchten Dateiobjektes und der Knoten n, der es speichert, angegeben werden. Wie bei den anderen Kommandos wird aus dem Namen der Identifikator berechnet und eine ObjectGet-Nachricht an den Knoten n gesendet. Durch den *locate*-Befehl kann n bestimmt werden. Die darauf folgende Übertragung des Objektes wird in Abschnitt 5.2.4 beschrieben.

Heruntergeladene Objekte werden immer in einem speziell dafür vorgesehenen Verzeichnis abgelegt. Dieses Verzeichnis wird in der Konfigurationsdatei spezifiziert.

5.2.3 Visualisierungsmodul

Mit Hilfe des Visualisierungsmoduls lässt sich ein Netzwerk grafisch darstellen. Die Stage ist in der Klasse *Visualization* implementiert. Es wird ein 3-dimensionaler Graph mittels des Colossus-Werkzeuges [Sch04] erzeugt.

Nach den Initialisierungen und dem Erhalt der NodeReady-Nachricht publiziert die Stage ein Visualisierungsobjekt. Dabei handelt es sich um ein spezielles Verzeichnisobjekt mit dem Namen „visualize“. Ähnlich wie bei dem *ls*-Kommando wird zur Visualisierung eine Visualize-Nachricht abgeschickt. Die Nachricht erreicht jeden Knoten, der ein Visualisierungsobjekt publiziert hat. Das Visualisierungsmodul jedes Knotens sendet daraufhin die Liste aller Nachbarn an den Absender der Visualisierungsnachricht. Die Stage des Absenders generiert aus den Nachbarlisten eine Knoten- und eine Kantendatei, die mit Colossus dargestellt wird.

5.2.4 Replikationsmodul

Die Datenverteilung im Peer-to-Peer-Netzwerk ist in dem Replikationsmodul implementiert. Die zu Grunde liegende Technologie namens Dissemination Tree wurde bereits in Abschnitt 3.3.4 beschrieben. Der Großteil der Funktionalität steckt in der Klasse *Replication*.

Publikation von Objekten

Um ein Objekt zu publizieren, sendet das Knotenmodul eine ObjectPublish-Nachricht an das Replikationsmodul (siehe Abbildung 5.2). In dieser Nachricht sind die kompletten Metainformationen zu dem Objekt enthalten. Es wird damit durch Versenden einer ObjectLocate-Nachricht überprüft, ob das Objekt bereits im Netzwerk vorhanden ist. Im einfachsten Fall, wenn eine TapestryLocate-Failure-Nachricht empfangen wird, ist es nicht vorhanden. Dann wird das Objekt mit Hilfe einer TapestryPublish-Nachricht publiziert. Dadurch wird der

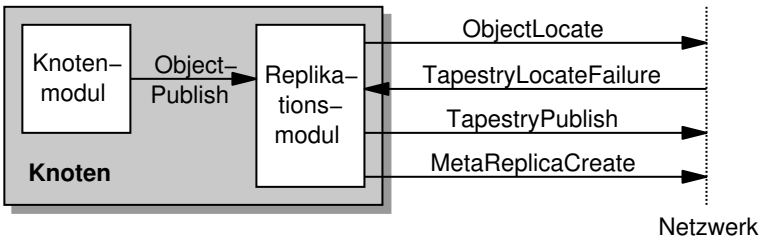

Abbildung 5.2: Die Publikation eines Objektes

Knoten n zur Wurzel des D-Trees des Objektes. Außerdem wird eine in der Konfigurationsdatei festgelegte Anzahl von Metaobjekten an eine zufällige Auswahl von Nachbarn verteilt. Dazu wird an jeden Knoten der Auswahl eine MetaReplicaCreate-Nachricht mit enthaltenem Metaobjekt gesendet. Die Knoten treten damit als Kinder in den D-Tree ein und publizieren ihre Metaobjekte

als Replikate. Erst bei einer Anfrage auf ein Objekt eines der Knoten wird es vom Vater n geladen.

Sollte das Objekt bereits im Netzwerk existieren, muss n in den bestehenden D-Tree des Objektes eintreten. Durch die erhaltene ObjectFound-Nachricht ist der dichteste Knoten i mit einem Replikat bekannt. An diesen wird eine DTJoinRequest-Nachricht mit der entsprechenden Objekt-ID gesendet, um in den D-Tree einzutreten. i bestätigt die Anfrage mit einer DTJoinAccess-Nachricht, nimmt n als Kind auf und wird Vater von n.

Download von Objekten

Stellt ein Knoten n eine Anfrage auf Objekt o, sendet er eine ObjectGet-Nachricht an Knoten i. Das Replikationsmodul von i empfängt diese Nachricht. Ist nur das Metaobjekt von o vorhanden, muss o zuerst vom Vater p geladen werden. Durch die Anwendung des SmartPlacement-Algorithmus wird möglicherweise ein Replikat zwischen n und i erzeugt[2]. Erfüllt i die Latenzanforderung von n, wird kein Replikat erzeugt, sondern das Objekt einfach an n übertragen (siehe nächster Unterabschnitt).

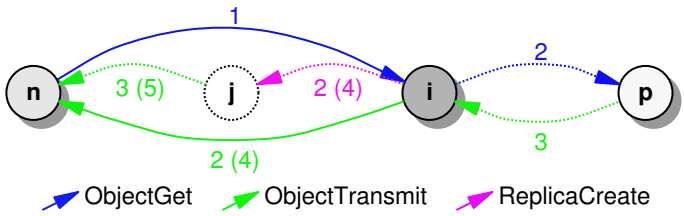

Abbildung 5.3: Der Download eines Objektes von Knoten i nach n. Möglicherweise wird es vorher vom Vater p übertragen und ein Replikat auf j erzeugt.

Wenn ein Replikat erzeugt werden soll, sendet i eine ReplicaCreate-Nachricht mit o an einen Knoten j (nach SmartPlacement). j tritt damit in den D-Tree ein und publiziert sein Replikat im Netzwerk. i wird Vater von j und j wird Kind von i. Danach überträgt j das Objekt o an n und n wird ein Blatt im D-Tree. Das bedeutet, es hat o in seinem lokalen Speicher zwischengespeichert (Cache), hat aber kein Replikat. Wenn das Objekt nach Ablauf der Verfallszeit gelöscht wird, tritt n wieder aus dem D-Tree aus. Dazu sendet n eine DTLeave-Nachricht an j, wodurch n aus der Liste der Kinder von j gelöscht wird.

Übertragung von Objekten

Die Übertragung eines Objektes erfolgt mit Hilfe der Transportnachricht ObjectTransmit. Das Objekt wird dazu in die Nachricht eingebettet und per Ta-

[2]Beim Download des Objektes vom Vater, wird kein zusätzliches Replikat erzeugt.

pestry an den Zielknoten übermittelt. Wird das System zu einem späteren Zeitpunkt um andere Datentypen wie beispielsweise Video erweitert, sollte der Objekttransport auf anderem Wege realisiert werden. Dazu kann ein zusätzlicher TCP-Kanal genutzt werden. Für die eher kleinen Objekte, mit denen das derzeitige System arbeitet, funktioniert ein Objekttransport per ObjectTransmit-Nachricht gut.

Sollten die Objekte doch zu groß für eine Nachricht werden, können sie jeweils in n Teile zerlegt werden und mit n Nachrichten versendet werden.

Ping-Pong-Nachrichten

Zur Erhaltung eines D-Trees sendet ein Knoten n Ping-Nachrichten an alle Kinder. Diese antworten mit Pong-Nachrichten. Antwortet ein Kind i nach einem Timeout nicht, wird der Versuch ein weiteres Mal wiederholt. Bei erneutem Fehlschlagen wird i aus der Liste der Kinder von n entfernt.

Damit der D-Tree in dieser Situation nicht „auseinanderbricht“ und die Struktur robuster gegen Ausfälle wird, werden in den Ping-Pong-Nachrichten zusätzliche Informationen eingebettet. Wie bereits in Abschnitt 4.2.3 beschrieben, kennt jeder Knoten die Kinder seiner Kinder, seine Geschwister und seinen Großvater. In eine Ping-Nachricht sind also die Adresse des Großvaters und der Geschwister eines Kindes enthalten, während eine Pong-Nachricht die Adressen der Kinder eines Kindes enthält. Dadurch kann im Falle eines Netzwerkfehlers eines einzelnen Knoten die Aufrechterhaltung des D-Trees gesichert werden.

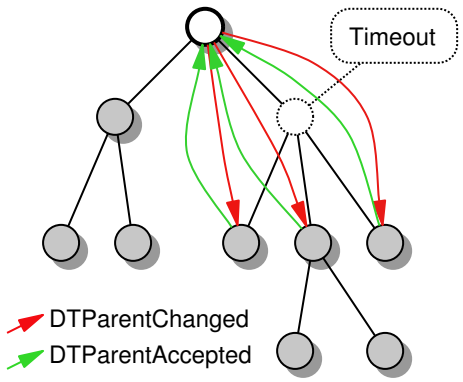

Abbildung 5.4: Das Zusammensetzen eines D-Trees nach dem Timeout eines Knotens.

Im obigen Fall, wenn n nach zwei Versuchen keine Pong-Nachricht von i erhalten hat, werden die Kinder von i als Kinder von n aufgenommen und n wird neuer Vater der Kinder. Dazu sendet n an jedes Kind eine DTParentChanged-Nachricht, in der der neue Großvater und die neuen Geschwister enthalten

sind. Die Kinder antworten mit DTParentAccepted-Nachrichten, die ihre eigenen Kindknoten enthalten (siehe Abbildung 5.4). Erhält ein Knoten lange Zeit keine Ping-Nachricht von seinem Vater, wartet er auf die DTParentChanged-Nachricht seines Großvaters. Hat er keinen Großvater, das heißt sein Vater war die Wurzel, sendet er an alle Geschwister eine UptimeInfo-Nachricht. Der Knoten n mit der höchsten Uptime wird die neue Wurzel und fügt alle Geschwister zu seiner Kindliste hinzu. Die anderen Geschwister hingegen ersetzen den Vatereintrag durch n.

Austritt aus dem D-Tree

Bei einem kontrollierten Austritt eines Knotens n aus einem D-Tree wählt n den Knoten i seiner Kinder mit der kleinsten Latenz aus, um seinen Platz im Baum einzunehmen. Dazu sendet er i eine DTBecomeParent-Nachricht. i sendet daraufhin all seinen Geschwistern eine DTParentChanged-Nachricht. Um den Vaterknoten p von n zu benachrichtigen, sendet n eine DTLeave-Nachricht, die die Adresse von i enthält. Dadurch kann p n durch i ersetzen. Die Umstrukturierung des D-Trees ist in Abbildung 5.5 dargestellt.

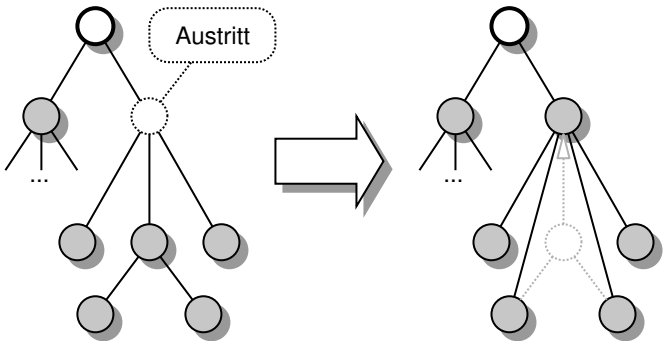

Abbildung 5.5: Ein Austritt aus einem D-Tree

Updates

Updates werden, wie in Abschnitt 4.2.3 beschrieben, über Primärkopien realisiert. Das bedeutet, nur der Knoten, der die Primärkopie eines Objektes besitzt, darf aktiv Updates durchführen.

Will ein Nutzer ein Dokument ändern und es im Netzwerk aktualisieren, muss beim Replikationsmodul eine interne ObjectUpdate-Nachricht mit dem geänderten Objekt eingehen. An dieser Stelle bieten sich mehrere Möglichkeiten an. Einerseits kann, wie in dem System dieser Arbeit, das komplett neue Objekt an das Replikationsmodul übergeben werden oder es werden nur die

Änderungen zwischen beiden Objekten übertragen. Dafür kann bei Textdaten das Diff-Werkzeug[3] und bei Binärdaten Rsync [Rsy] verwendet werden.

Ist der Knoten die Wurzel des D-Trees und ist das Objekt schreibbar, wird die alte Version des Objektes durch die neue ersetzt. Der Versionszähler wird danach um eins inkrementiert und die neue GUID des Objektes in seine Historie eingefügt.

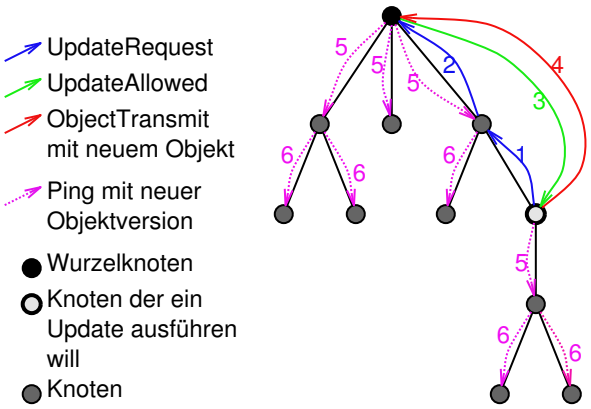

Abbildung 5.6: Der Nachrichtenversand bei einer Updateoperation im mobilen Informationssystem

Wenn der Knoten nicht der Wurzelknoten ist und somit nicht die Primärkopie des Objektes besitzt, muss er eine Updateanfrage an die Wurzel des D-Trees schicken. Ein Beispiel einer Updateoperation ist in Abbildung 5.6 dargestellt. Mit Hilfe einer UpdateRequest-Nachricht, die von Vater zu Vater bis zum Wurzelknoten weitergereicht wird, gelangt die Anfrage nach der Änderung des Objektes zum Besitzer der Primärkopie. Die Nachricht enthält dabei nur die ID und die gespeicherte Versionsnummer des Objektes. Der Wurzelknoten antwortet dem anfragenden Knoten mit einer UpdateAllowed-Nachricht, falls das Objekt schreibbar ist und die gesendete Version mit der der Primärkopie übereinstimmt. Dann wird das neue Objekt an die Wurzel übertragen. Beide Knoten ersetzen die alte mit der neuen Version und erhöhen den Versionszähler um eins.

Sollte eines der beiden Kriterien nicht erfüllt sein, wird eine UpdateRejected-Nachricht gesendet, in der eine Fehlerbeschreibung enthalten ist. In diesem Fall löst das Replikationsmodul des Knotens, der das Update ausführen will, ein UpdateFailed-Event mit Fehlerbericht aus, so dass die Stage, die das Update durchführen wollte, benachrichtigt wird. Bei einem Versionskonflikt kann das Objekt aktualisiert und das Update wiederholt werden. Ist ein Schreibzugriff untersagt, besteht keine Möglichkeit das Objekt zu aktualisieren.

[3] *diff* gehört bei den meisten Unix-Derivaten zum Standardbefehlssatz.

Die Aktualisierung der restlichen Replikate erfolgt mit Hilfe der Ping-Nachrichten. Neben den Knoteninformationen enthält eine Ping-Nachricht die Versionsnummer des Objektes. Dadurch erfährt ein Kindknoten bei der nächsten Ping-Nachricht, dass sein Objekt veraltet ist. Um die Antwortzeit des Netzwerkes klein zu halten, wird das neue Objekt nicht sofort vom Vater übertragen. Es wird nur die neue Version gespeichert und mit der folgenden Ping-Nachricht an die eigenen Kinder weitergeleitet. Die Aktualisierung wird bis zur nächsten Anfrage verzögert. Dadurch kann bei dem ersten Zugriff auf ein Objekt nach einem Update, ähnlich wie beim ersten Zugriff eines Knoten auf ein Objekt, das Latenzkriterium verletzt werden. Die andere Variante ist, Objekte sofort zu aktualisieren. Dadurch entsteht bei Updates von populären Objekten jedoch schnell viel Netzwerklast, wodurch die gesamte Antwortzeit des Systems erhöht werden kann.

5.3 Aufbau des Netzwerkes

Um ein Netzwerk mit anderen Computern aufzubauen, benötigt ein Computer eine konfigurierte Netzwerkkarte mit einem Netzwerkzugang. Die Konfiguration der Netzwerkkarte kann im Falle von WLAN beispielsweise per DHCP über den Accesspoint erfolgen. Ein selbständiger Netzwerkaufbau kann nur zwischen Computern im gleichen Subnetz stattfinden. Dazu wird eine Broadcast-Nachricht gesendet, die von Knoten im Tapestry-Netzwerk beantwortet wird. Der erste antwortende Computer wird zum Gateway für den eintretenden Knoten. Dazu wird eine SEDA-Konfigurationsdatei für einen dynamischen Knoten generiert und SEDA gestartet. Es initialisiert die entsprechenden Stages, und der Computer tritt als weiterer Knoten in das mobile Informationssystem ein.

Erhält ein Computer keine Antwortnachricht, existiert in diesem Subnetz kein Tapestry-Netzwerk. Ein Nutzer hat die Möglichkeit, Gateways in anderen erreichbaren Netzwerken anzugeben, über die eine Verbindung in das Informationssystem erfolgen kann. Andernfalls wird ein neues Overlay-Netzwerk gestartet. Dazu wird eine Konfigurationsdatei generiert, in der der Knoten statisch und zugleich der Vereinigungsknoten ist. Dadurch wird ein Netzwerk mit einem Knoten gestartet, in das dynamische Knoten - wie oben beschrieben - eintreten können.

Kapitel 6

Schlussbetrachtung

In diesem Kapitel wird eine Zusammenfassung der Diplomarbeit gegeben. Außerdem werden Vorschläge und Ideen zur Weiterentwicklung des entwickelten Systems gegeben.

6.1 Zusammenfassung

Ziel dieser Diplomarbeit ist die Entwicklung eines mobilen Informationssystems (MIS) auf Basis eines Peer-to-Peer-Overlay-Netzwerkes. Die prototypische Implementierung des Systems erfolgt in Java. Es bietet die Möglichkeit Dokumente im XML-, PDF- und Textformat auszutauschen. Dazu wird mit Hilfe von Tapestry [ZKJ01] ein strukturiertes, dezentralisiertes Overlay-Netzwerk aufgebaut, in dem jeder Knoten Dokumente freigeben, lokalisieren und herunterladen kann. Um Dokumente im Netzwerk verfügbar zu machen, muss ein Knoten diese publizieren. Danach sind sie für andere Knoten sichtbar und können heruntergeladen werden. Dokumente können auch nach ihrer Publikation verändert werden (Updates). Jedes Dokument im Netzwerk hat eine Versionsnummer, die durch ein Update erhöht wird.

Tritt ein Knoten, der ein Dokument publiziert hat, aus dem Netzwerk aus, verschwindet normalerweise auch das Dokument aus dem Netzwerk. Ein Austritt kann neben einer kontrollierten Aktion des Nutzers auch durch Fehler im Netzwerk auftreten. Gerade in einem mobilen Umfeld, wo mit häufigen Ein- und Austritten in und aus dem Netzwerk zu rechnen ist, sind die Informationen im System relativ instabil. Um dieses Verhalten zu unterbinden, werden Dokumente mehrfach auf verschiedenen Knoten gespeichert. Dadurch bleiben sie auch nach dem Austritt eines Knotens im Netzwerk vorhanden.

Schwerpunkt der Entwicklung ist das Replikationssystem zur Verteilung der Daten in dem Peer-to-Peer-Netzwerk. Das System legt entsprechend dem Nutzerverhalten selbständig Kopien (Replikate) einzelner Dokumente an, und verteilt diese sinnvoll auf andere Knoten im Netzwerk. Dadurch wird auf der einen Seite die Verfügbarkeit der Dokumente erhöht, jedoch entstehen auf der an-

deren Seite durch die Mehrfachspeicherung Probleme. Der Aufwand von Änderungsoperationen wird durch die Replikation stark gesteigert. Wo sonst nur ein einzelnes Dokument aktualisiert werden musste, müssen jetzt zusätzlich alle Replikate verändert werden.

Um Änderungen durchführen zu können, organisiert das System die angelegten Replikate in einer verteilten Baumstruktur namens Dissemination Tree (D-Tree) [CKK02]. Die Synchronisation eines Updates, das heißt der Abgleich aller Replikate, wird per Primärkopie realisiert. Das bedeutet, Änderungsoperationen dürfen nur auf einer ausgewählten Kopie ausgeführt werden. Alle anderen Replikate beziehen die Änderung von dieser Kopie. Da der publizierende Knoten die Wurzel des D-Trees darstellt, ist die Primärkopie darauf gespeichert. Ein Update kann nur von diesem Knoten ausgeführt werden, so dass andere Knoten Änderungsoperationen beantragen müssen.

6.2 Ausblick

Momentan unterstützt das Informationssystem Lokalisierungs- und Verzeichnisanfragen. Lokalisierungsanfragen liefern einen Knoten, auf dem sich ein bestimmtes Objekt befindet und Verzeichnisanfragen listen die Objekte eines bestimmten Verzeichnisses auf. Da das Ziel dieser Arbeit die Verteilung der Daten ist, werden keine Suchanfragen unterstützt, so dass das System in dieser Richtung erweitert werden kann. Durch Suchanfragen soll der Nutzer in der Lage sein, Inhalte zu spezifizieren, die Dokumente enthalten sollen. Das kann der Autor eines Buches sein oder das Erscheinungsjahr eines Artikels. Außerdem sind Volltextsuchen vorstellbar. Wie in Kapitel 3 bereits beschrieben wird, kann auch im Volltext von PDF-Dokumenten gesucht werden, nachdem diese per Konverter in Textdaten umgewandelt wurden. Auch Bereichsanfragen - wie zum Beispiel die Anfrage nach allen veröffentlichten Artikeln eines bestimmten Autors zwischen den Jahren 1999 bis 2002 - können realisiert werden.

Das Visualisierungsmodul kann um einige Features erweitert werden. Derzeit unterstützt es nur die Darstellung des Netzwerkes. Zur besseren Analyse können Pfade von Nachrichten und die Verbreitung von Replikaten dargestellt werden.

Das zur Darstellung verwendete Colossos-Werkzeug stellt das Netzwerk in der im Prototyp eingesetzten Version statisch dar. Das bedeutet, während der Darstellung sind keine Änderungen, wie Ein- bzw. Austritte von Knoten, sichtbar. Eine Verbesserung ist eine dynamische Visualisierung, die auf Netzwerkänderungen in der Darstellung reagiert, wie es beispielsweise das Minitasking-Werkzeug [Sch] für Gnutella-Netzwerke kann. Durch solch ein Verfahren können auch Anfragen und die damit verbundene Ausbreitung von Nachrichten beobachtet werden.

Eine weitere Verbesserung des Systems liegt im Bereich der Netzwerkstruktur. Es ist denkbar, verschiedene Netzbereiche zu definieren und das Browsing

zu erweitern, so dass es das *Level of Detail* unterstützt. Dadurch können bestimmte Bereiche abstrahiert und in andere hineingezoomt werden. Wird diese Technik mit Suchanfragen kombiniert, werden Anfragen in bestimmten Bereichen des Netzwerkes möglich. Außerdem ist das Colossos-Werkzeug für solche Darstellungen vorbereitet, so dass auch die Visualisierung dadurch verbessert werden kann.

In dieser Version des Prototyps besitzen die Daten keinerlei Struktur. Gerade bei XML ist eine Unterstützung der Struktur der Dokumente von Nutzen, da eine sinnvolle Verteilung und Strukturanfragen ermöglicht werden können.

Auch dynamische Strukturierungsmöglichkeiten sind in dem System denkbar. Beispielsweise kann das System um dynamische Verzeichnisse erweitert werden. Das sind Verzeichnisse, die durch eine Anfrage definiert werden, ähnlich wie Sichten in Datenbanksystemen. Es wird dadurch zwischen statischen und dynamischen Verzeichnissen unterschieden.

Planbarkeit von Ereignissen stellt eine Erweiterungsmöglichkeit des Replikationssystems dar. Im Szenario (vgl. Kapitel 3) zum Beispiel steigen die Zugriffe auf die Vortragsfolien während und besonders nach einem Vortrag an. Es ist also sinnvoll, einen zu einer bestimmten Zeit stattfindenden Vortrag als Ereignis zu definieren und vorher die Anzahl der Replikate zu erhöhen, um den steigenden Anfragen gerecht zu werden.

Abbildungsverzeichnis

Liste der Algorithmen

Tabellenverzeichnis

Abkürzungsverzeichnis

ADR	***A**daptive **D**ata **R**eplication*
CAN	***C**ontent-**A**ddressable **N**etwork*
DHT	***D**istributed **H**ash **T**ables*
GUID	***G**lobal **U**nique **Id**entifier*
HAF	***H**ighly **A**vailable **F**irst*
HUF	***H**ighly **U**p **F**irst*
JVM	***J**ava **V**irtual **M**achine*
LRU	***L**ast **R**ecently **U**sed*
MFU	***M**ost **F**requently **U**sed*
MIS	***M**obiles **I**nformations**s**ystem*
QoA	***Q**uality **o**f **A**vailibility*
URL	***U**niform **R**esource **L**ocator*

Literaturverzeichnis

[ABC+02] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.

[BCE+99] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. Oceanstore: An extremely wide area storage system. Technical Report UCB/CSD-00-1102, 1999.

[BD96] Thomas Beuter and Peter Dadam. Prinzipien der replikationskontrolle in verteilten datenbanksystemen. Technical report, Universität Ulm, Sebtember 1996.

[BDET00] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43. ACM Press, 2000.

[BMSV02] R. Bhagwan, D. Moore, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer storage, 2002.

[CKK02] Yan Chen, Randy H. Katz, and John D. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Peer-to-Peer Systems: First International Workshop, IPTPS 2002*, pages 306–318, Cambridge, MA, USA, March 2002.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46, 2001.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. pages 173 – 182, 1996.

[Gnu] Gnutella. http://www.gnutelliums.com/.

[GY98] A. Goldberg and Peter N. Yianilos. Towards an archival intermemory. In *Proceedings of IEEE Advances in Digital Libraries, ADL 98*, pages 147–156, Santa Barbara, CA, 1998. IEEE Computer Society.

[KaZ] KaZaA. http://www.kazaa.com.

[KRT02] Jussi Kangasharju, Keith W. Ross, and David A. Turner. Optimal content replication in p2p communities, 2002.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[Nap] Napster. http://www.napster.com.

[OSS01] Giwon On, Jens Schmitt, and Ralf Steinmetz. The Quality of Availability: Tackling the Replica Placement Problem. Technical Report TR-KOM-2001-11, Darmstadt University of Technology, November 2001.

[OSS03a] Giwon On, Jens Schmitt, and Ralf Steinmetz. The effectiveness of realistic replication strategies on quality of availability for peer-to-peer systems. In *Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P 03)*, pages 57 – 65. Darmstadt University of Technology, IEEE, September 2003.

[OSS03b] Giwon On, Jens Schmitt, and Ralf Steinmetz. QoS-Controlled Dynamic Replication in Peer-to-Peer Systems. *Praxis der Informationsverarbeitung und Kommunikation*, 03(2):96–101, April 2003.

[OSS03c] Giwon On, Jens Schmitt, and Ralf Steinmetz. Quality of availability: Replica placement for widely distributed systems. In K. Jeffay, I. Stoica, and K. Wehrle, editors, *Quality of Service - IWQoS 2003: 11th International Workshop, Berkeley, CA, USA, June 2-4, 2003. Proceedings*, volume 2707 / 2003, pages 325 – 344. Darmstadt University of Technology, Springer-Verlag Heidelberg, August 2003.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[Rat02] S. Ratnasamy. Routing algorithms for dhts: Some open questions, 2002.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[RGR+03] Sean Rhea, Dennis Geels, Timothy Roscoe, , and John Kubiatowicz. Handling churn in a dht. Technical Report UCB/CSD-03-1299, University of California, Berkeley and Intel Research, Berkeley, dec 2003.

[RIF02] Kavitha Ranganathan, Adriana Iamnitchi, and Ian Foster. Improving data availability through dynamic model-driven replication in large peer-to-peer communities, 2002.

[Rob95] Matthew J. B. Robshaw. Md2, md4, md5, sha and other hash functions. Technical Report TR-101, RSA Laboratories, 1995. version 4.0.

[Rsy] Rsync. http://samba.anu.edu.au/rsync/.

[Sch] Schoenerwissen. Minitasking. http://www.minitasking.com/old/.

[Sch04] Hans-Jörg Schulz. Visuelles Data Mining komplexer Strukturen. Master's thesis, Universität Rostock, 2004.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In Roch Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, August 27–31 2001. ACM Press.

[WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[Wel] Matt Welsh. Seda. http://seda.sourceforge.net/.

[WJH97] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

[WK] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison.

[ZHS+03] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 29. Oktober 2004

Thesen

1. Peer-to-Peer-Netzwerke haben heute einen großen Stellenwert im Bereich verteilter Systeme eingenommen. Die am weitesten verbreitete Anwendung dieser Technologie ist die Datenspeicherung.

2. Sie bilden einen Gegensatz zum üblicherweise eingesetzten Client-/Server-Prinzip und können es in vielen Bereichen ersetzen.

3. Dezentralisierte, strukturierte Netzwerke bilden einen guten Ansatz für ein mobiles Informationssystem, da sie robuster als zentralisierte Systeme sind und Anfragen weniger Netzlast erzeugen als bei unstrukturierten Peer-to-Peer-Netzwerken.

4. Durch Änderungen der Popularität von Datenobjekten, Netzwerkfehlern und häufigen Ein- und Austritte aus dem Netzwerk, die besonders in einem mobilen Umfeld vorzufinden sind, wird die Verfügbarkeit der Inhalte verschlechtert.

5. Die Replikation von Daten, beeinflusst vom Nutzerverhalten, und die sinnvolle Verteilung der Objektkopien führt zu einer starken Verbesserung der Verfügbarkeit.

6. Der Aufwand für Änderungsoperationen auf den Datenobjekten im Peer-to-Peer-Netzwerk wird durch die Replikation enorm erhöht, da zusätzlich zu den Originalobjekten alle Replikate synchronisiert werden müssen.

7. Ein Replikationssystem muss die Anzahl der Replikate minimieren, um Änderungsoperationen weiterhin effizient durchführen zu können.

8. Um Änderungen auf alle Replikate übertragen zu können, ist eine verteilte Datenstruktur notwendig, die eine Verbindung zwischen den einzelnen Objektkopien herstellt.

9. Die Aktualisierung aller Replikate eines Objektes in einem atomaren Schritt erhöht die Antwortzeit des Systems drastisch. In einem mobilen System ist daher eine asynchrone Durchführung besser geeignet, die nach Abschluss der Änderungsoperation stattfindet.