OpenDHT-Dienst als gemeinsame DHT-Infrastruktur

Mohannad Alnablsi
(alnablsi@cs.tu-berlin.de)

Seminar „Internet Routing“ ,
Technische Universität Berlin

SS 2009 (Version vom 3. Juli 2009)

Zusammenfassung

Verteilte Hashtabellen (DHTs) haben viele Vorteile, u.a. die Skalierbarkeit und Fehlertoleranz. Man möchte diese Vorteile nutzen, um andere Anwendungen wie Instant Messaging, Broadcast oder Packet-Routing zu realisieren. OpenDHT stellt eine DHT zur Verfügung und bietet eine gemeinsame und einfache Schnittstelle zur Nutzung der DHT an. Diese Arbeit behandelt die Inhalte von der Arbeit ”OpenDHT: A Public DHT Service and Its Uses” [Rhea et al., 2005]. Dabei wird eine Zusammenfassung der wesentlichen Teile präsentiert. In dieser Arbeit werden Aspekte von DHT wie die Grundlagen, die Verwendungen und Schnittstellen erläutert. OpenDHT wird als Instanz dieser Aspekte dargestellt und erklärt. Abschließend wird eine Evaluation von OpenDHT vorgestellt.

1 Einleitung

Verteilte Systeme sind kompliziert. Hier ist mit ”verteiltes System” ein System aus Rechnern gemeint, die miteinander vernetzt sind. Sie erledigen durch Zusammenarbeit die Aufgabe des Systems. Bei dieser Zusammenarbeit ergeben sich die Herausforderungen Organisation, Koordination und Zuverlässigkeit. Nicht zu vergessen ist, dass in der Rechnerwelt Ausfälle von Hardware oder Netzwerken nicht auszuschließen sind. Bei mehreren Rechnern steigt sogar die Wahrscheinlichkeit, dass ein Fehler auftritt, was zum Absturz des ganzen Systems führen kann. Verteilte Systeme haben einen Ansatz zur Bewältigung der beschriebenen Herausforderungen.

Verteilte Hashtabellen sind eine Instanz von verteilten Systemen, die mit den genannten Herausforderungen konfrontiert ist. Die verteilten Hashtabellen haben viele Vorteile, u.a. die Skalierbarkeit und Fehlertoleranz. Man kann verteilte Hashtabellen verwenden zur Speicherung von Werten, zur Abbildung von Schlüsseln auf verteilten Rechnern und zur effizienten Adressierung von Rechnern. Es existieren bereits viele DHTs. Bevor man die DHTs verwendet, ist zunächst eine aufwändige Konfigurierung und Installation erforderlich.

OpenDHT ist eine fertig installierte und konfigurierte DHT, die als Dienst angeboten wurde. Der Dienst OpenDHT ist kostenlos und allen zugänglich. Bei der Gestaltung dieses Dienstes wurde die Allgemeinheit des Dienstes angestrebt. Durch einfach zu verwendende Schnittstellen ist OpenDHT allgemein zugänglich.

In dieser Arbeit wird zuerst auf die Grundlagen und Verwendung von DHT eingegangen. Danach wird OpenDHT und dessen Ausführung der Aufgaben von DHT

erläutert. Die Nutzung einer benutzerseitigen Bibliothek in OpenDHT wird im vierten Kapitel vorgestellt. Anschließend werden Garantien für die Funktion von OpenDHT erklärt. Abschließend wird eine Evaluation des Dienstes vorgestellt sowie die Arbeit zusammengefasst und die Originalarbeit bewertet.

2 Hintergrundinformation

Hash-Funktionen spielen heute eine sehr wichtige Rolle. Eine Hash-Funktion wird hauptsächlich benutzt, um *Hashwerte* mit bestimmten Eigenschaften von anderen Daten zu erzeugen. Diese Hashwerte benutzt man, um Daten in Hash-Tabellen zu speichern. Die Daten sollen einen Schlüssel haben, wonach sie auffindbar sind. Der Schlüssel einer MP3-Datei kann der Dateiname sein. Hier sprecht man von $(Schlüssel, Wert)$ als Paar. Für eine MP3-Datei gibt es das Paar $(Dateiname, Daten\ der\ Datei)$. Die Daten werden in die Hashtabelle nach ihrem Schlüssel gespeichert. Wenn man den Schlüssel hat, kann man die Daten aus der Hashtabelle bekommen.

Die Hashtabelle kann man als ein Array sehen. Die Indezis dieses Arrays sind Hashwerte. Ein Element wird in dieses Array z.B. bei dem Index 43 genau dann gespeichert, wenn der Hashwert seines Schlüssels 43 beträgt.

Das Zugreifen eines Elements in einem Array hat die Aufwandklasse O(1), wenn man den Index dieses Elements kennt. Beim gegebenen Schlüssel kann man den Wert schnell und effizient finden: Es wird zunächst der Hashwert des Schlüssels berechnet. Danach wird das Element, dessen Index mit dem Hashwert identisch ist, aus dem Array gelesen.

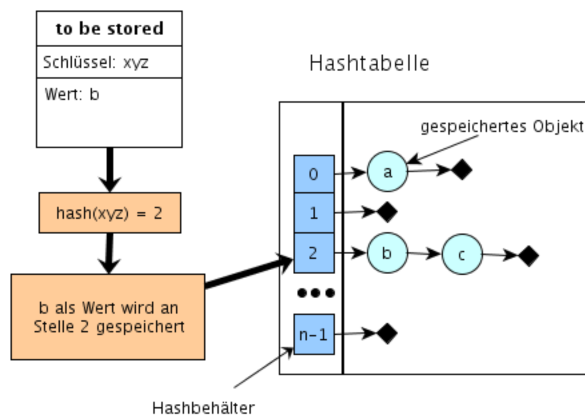

Abbildung 1: Hashtabelle

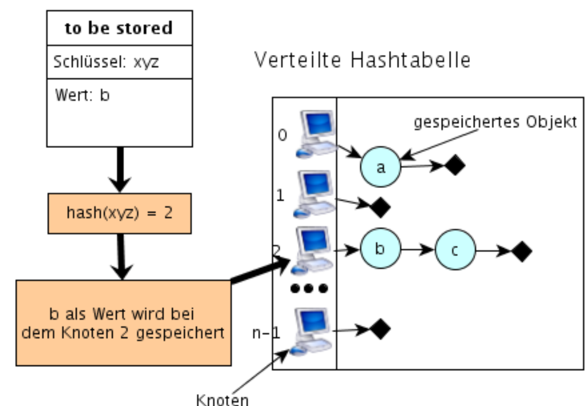

Abbildung 2: verteilte Hashtabelle

In der Abbildung 1 wird eine Hashtabelle dargestellt. Das (Schlüssel, Wert)-Paar (xyz, b) wird in die Hashtabelle gespeichert. Die hashfunktion $hash()$ bildet xyz auf 2 ab. So wird der Wert in das Array an der Stelle 2 gespeichert.

2.1 Verteilte Systeme und DHT

Verteilte Hashtabellen (DHT) sind eine Erweiterung des Prinzips von Hashtabellen auf ein verteiltes Modell. Die Hashtabelle ist auf verschiedene Rechner verteilt. Die Rechner werden anstelle eines zentralen Array benutzt. Man bezeichnet diese Rechner als Knoten der DHT. Die Daten werden auf den Knoten gespeichert. Analog zu der traditionellen Hashtabelle dient der Schlüssel bei der DHT zur Auffindung des Knotens, auf dem das gesuchte Objekt gespeichert wird. Die Abbildung 2 stellt eine verteilte Hashtabelle dar.

(xyz, b) wird in die Hashtabelle gespeichert. Da die hashfunktion $hash()$ xyz auf 2 abbildet, wird b auf den Knoten 2 gespeichert. Bei DHT gibt es verschiedene Herausforderungen. Es stellen sich folgende Fragen: Welche Struktur hat die DHT? Wie kann man die Daten auf die Knoten verteilen? Ist die Anzahl der Knoten konstant? Wie können Knoten eingefügt werden? Was passiert, wenn ein Knoten ausfällt? Wie findet man den gesuchten Knoten?

Es gibt viele Beispiele für DHT. Chord [Stoica et al., 2003], CAN [Ratnasamy et al., 2001] und Pastry [Rowstron and Druschel, 2001] sind einige der Berühmten.
OpenDHT basiert auf der Bamboo-DHT-Implementierung [Rhea et al., 2004, 5. Kap.]. Bamboo ist eine Weiterentwicklung der DHT von Pastry. Es folgt eine kurze Beschreibung der Struktur in Bamboo und des DHT-Routings, wie in [Rhea et al., 2004] beschrieben:

Die Knoten von Bamboo sind in einem Netzwerk. Im Fall des Internets sollen die Knoten öffentliche IP-Adressen besitzen. Jeder Knoten in Bamboo hat einen Zahlenidentifikator aus dem Bereich $[0, 2^{160}[$. Der Identifikator wird zum Beispiel vom SHA-1-Hashwert[1] von der IP-Adresse mit dem Port berechnet. Eine weiter Möglichkeit ist die Benutzung des Hashwerts der öffentlichen Schlüssel vom Knoten. Die Menge der Identifikatoren kann man als ein Ring sehen, worin die Identifikatoren in Uhrzeigersinn aufsteigend sortiert sind. In Abbildung 3 sind die Knoten $\{gelb, grün, blau, rot, orange, braun\}$ Teilnehmer einer DHT. *blau* hat einen größeren Identifikator als *grün*. Der Identifikator von *grün* ist größer als der von *gelb*.

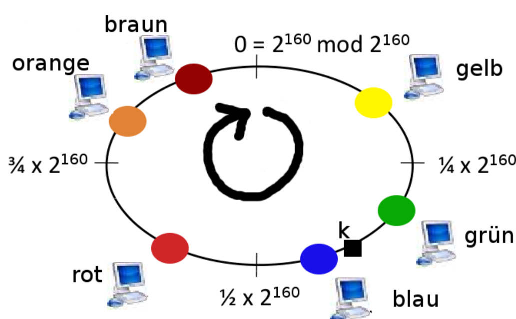

Abbildung 3: Knoten in DHT

Eintrag	0. Spalte	1. Spalte
0	00100101	-
1	-	11100111
2	100001100	-
3	-	10110101
4	-	10101011

Tabelle 1: Routingtabelle für Knoten mit dem Indikator 101001100

Die Knoten *gelb* und *blau* sind die Nachbarknoten von *grün*. Man bezeichnet *gelb* als Vorgänger von *grün* und *blau* als Nachfolger von *grün*. Jeder Knoten p speichert eine gerade Zahl von Nachbarknoten in einer Liste. Diese Liste bezeichnet man als Blätterliste. Es werden immer gleich viele Nachfolger und Vorgänger in der Blätterliste gespeichert. Angenommen, *blau* hätte vier Nachbarn in der Blätterliste. Dann wären sie zwei Vorgänger: *grün* und *gelb*, und zwei Nachfolger: *rot* und *orange*.

Ein Wert mit dem Identifikator k wird genau dann auf den Knoten *blau* gespeichert, wenn der Identifikator von k dem von *blau* näher ist als allen anderen Identifikatoren. Man bezeichnet *blau* als den verantwortlichen Knoten für k in der DHT.

Der Knoten p hat eine weitere Liste von Knoten aus dem DHT-Bereich als *Routingtabelle*. Die Knoten der Routingtabelle werden folgendermassen gewählt: Bei den Knoten des ersten Eintrags der Routingtabelle ist die erste Ziffer der Identifikatoren identisch mit der ersten Ziffer des Identifikators von p. Die zweite Ziffer unterscheidet sich von der zweiten Ziffer bei p. Die Knoten des zweiten Eintrags haben die gleichen ersten zwei Ziffern wie bei p und unterscheiden sich von p in der dritten Ziffer usw. Die Tabelle kann auch weitere Spalten haben. Der Identifikator des Knotens in der ersten Spalte hat die Ziffer 1 für die erste von p-Identifikator unterschiedliche Ziffer.

[1] SHA-1: Secure Hash Algorithm, Hashfunktion mit Hashwerten der Länge 160 Bits.

In der Tabelle 1 wird ein Beispiel für eine Routingtabelle für den Knoten mit dem Identifikator 101001100 dargestellt. Hier geht man davon aus, dass die Identifikatoren binäre Zahlen sind. Die Anzahl der Knoten bei jedem Tabelleneintrag und die Anzahl der Einträge beeinflussen die Effizienz der Erreichbarkeit der Knoten innerhalb der DHT. Man kann sie konfigurieren.

Wenn der Knoten p einen Knoten oder einen Wert in der DHT suchen will, sucht er in seiner Blätterliste, ob der Identifikator des Wertes zwischen p und seinen Nachbarknoten liegt. Wenn ja, wird die Suchanfrage an den Knoten, der dem Wert am nächsten liegt, delegiert. Die Suche endet, wenn der aktuelle Knoten dem Wert am nächsten ist. Falls der Knoten nicht im Bereich der Nachbarknoten liegt, sucht p in der Routingtabelle den Knoten p_r aus, der dem gesuchten Knoten am nächsten ist. p sendet dann an p_r eine Suchanfrage. Wenn p_r den gesuchten Knoten oder Wert besitzt, wird er zurück geliefert. Sonst macht p_r das gleiche wie p und schickt die Suchanfrage weiter. Diesen Suchmechanismus nennt man *routing innerhalb der DHT*. Er ist in der Abbildung 4 (a) dargestellt. Hier ist zu unterscheiden zwischen einer rekursiven Suche und einer iterativen Suche. Die rekursive Suche wurde im vorherigen Absatz beschrieben. Die iterative Suche umfasst die gleichen Knoten wie die rekursive Suche. Statt einen Wert zurück zu liefern, liefern die Knoten jedoch den Knoten aus ihren Routingtabellen zurück, der dem gesuchten Wert am nächsten ist. p sucht dann bei diesem Knoten nach dem Wert weiter. Die Abbildung 4 (b) stellt diesen Vorgang dar. Der Knoten mit dem Identifikator 111... sucht nach dem Wert, der im Bereich der Knoten 011.. liegt. Bei der rekursiven Suche wird die Anfrage erst an 00.. geschickt. 00... schickt die Anfrage weiter an 010..., der anschließend an 011.. weiterleitet. Bei der iterativen Suche schickt 111.. die Anfrage an 00..., 00... antwortet mit dem Identifikator von 010... . Jetzt fragt 111... bei 010... nach dem Wert weiter.

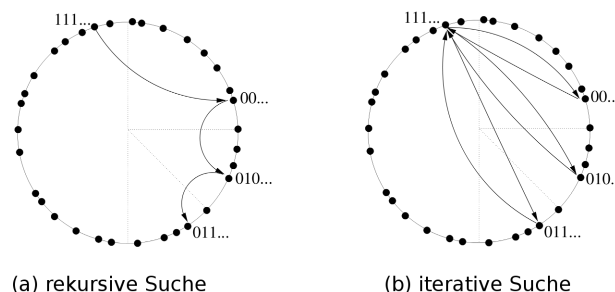

Abbildung 4: Routing in DHT aus [Rhea et al., 2004]

2.2 Verwendung von DHT

"There are three broad classes of interfaces in the DHT literature, and they each occupy very different places on the generality/simplicity spectrum (...)". [Rhea et al., 2005, K. 2.1] Die Autoren erwähnen, dass diese Klassen von Schnittstellen verschiedene Fähigkeiten anbieten. Sie weisen auf die Arbeit [Dabek et al., 2003] hin. Dort wurden Definitionen für P2P-Programmierschnittstellen vorgeschlagen, die eine ähnliche Klasse von Schnittstellen präsentiert[2].

Die verschiedenen Klassen der Schnittstellen von DHT unterscheiden sich bei Erfüllung mancher der folgenden Aufgaben in einer Anwendung.

[2] In [Dabek et al., 2003] wurde eine Schnittstelle vorgeschlagen, mit der man andere DHT-Schnittstellen simuliert.

1. **Storage:** Hier wird die DHT zur Speicherung von Werten verwendet. Diese Aufgabe wird im Zusammenhang mit den Hashtabellen-Operationen $put(Schlussel, Wert)$ und $get(Schlüssel)$ verwendet. Hier werden beide Operationen bei dem Knoten ausgeführt, der für diesen Wert verantwortlich ist. Diese Schnittstelle fungiert nur auf der Wertebene und liefert keine zusätzlichen Informationen oder Funktionalitäten, die den verantwortlichen Knoten oder die Knoten auf dem Weg dahin betreffen. Diese Schnittstelle hat den Vorteil der einfachen Nutzung und den Nachteil der Beschränkung der Nutzung der DHT auf Speicherung.
2. **Lookup** liefert im Gegensatz zu **Storage** den für den *Schlüssel* verantwortlichen Knoten zurück. Die Anwendung kann diesen Knoten zur Durchführung von Aufgaben, wie zum Beispiel Weiterleiten von Netzwerkpaketen, verwenden. Die DHT kann hier zur Suche nach Gateways verwendet werten. Die Aufgaben, die der verantwortliche Knoten durchführt, werden von dem Benutzer spezifiziert. Das heißt, dass die verantwortlichen Knoten beliebige Programme ausführen sollen.
3. **Routing** kann als die Ausdehnung von **Lookup** auf dem ganzen Weg zu dem verantwortlichen Knoten gesehen werden. Hier ist es möglich, Aufgaben sowohl auf dem Endknoten durchzuführen, als auch auf allen Knoten, die auf dem Weg zum Endknoten sind und die zum Routing innerhalb der DHT dienen. Ein Beispiel wäre eine Broadcast-Nachricht an den Endknoten zu schicken. Alle Knoten können diese Nachricht auch empfangen und ihren Zustand aktualisieren.

3 OpenDHT

Die Autoren von [Rhea et al., 2005] haben Bamboo-DHT auf vielen Rechnern in PlanetLab installiert. PlanetLab ist ein Gruppe von verschiedenen Rechnern auf der ganzen Welt [Bavier et al., 2004]. Die Rechner werden von verschiedenen Forschungseinrichtungen zum Testen und zur Evaluation von Projekten für Netzwerke und verteilte Systeme genutzt. OpenDHT ist ein Dienst, der von den Autoren von [Rhea et al., 2005] angeboten wird. OpenDHT benutzt die Bamboo-DHT und läuft auf den Rechner von PlanetLab. In diesem Kapitel wird auf OpenDHT im Detail eingegangen.

3.1 Überblick OpenDHT

OpenDHT wurde als kostenloser öffentlicher Dienst angeboten. Bei dem Design von OpenDHT wurde ein allgemeiner Dienst realisiert, der DHT für die Anwendungen anbietet. OpenDHT bleibt dabei sehr allgemein und nicht anwendungsspezifisch. Die fundamentale Aufgabe von OpenDHT ist die **Storage**. Diese Aufgabe hat den Vorteil der einfachen und allgemeinen Nutzung. Die Aufgabe **Lookup** wird dann auf **Storage** aufgebaut und durch die Benutzung von einer Bibliothek namens "Recursive Distributed Rendezvous" (ReDiR) realisiert. Da die Aufgabe **Lookup** die Ausführung benutzerspezifischer Programme anfordert, können die Benutzer, die **Lookup**-Aufgaben benötigen, die DHT als Treffpunkt für ihre eigene Klient-Knoten verwenden, die einer Anwendung gehören. Auf diesen Knoten werden dann mittels ReDiR die benutzerspezifischen Aufgaben erledigt. Diese Knoten sind nicht mit den Knoten von der Bamboo-DHT von OpenDHT zu verwechseln. Die DHT-Knoten von OpenDHT erfüllen immer die **Storage**-Aufgaben. Dagegen sind die Knoten einer Anwendung Klient-Knoten und erfüllen die **Lookup**-Aufgaben.

3.2 Wie OpenDHT die DHT-Aufgabe "Storage" erfüllt

Die Storage-Aufgabe wird durch die Schnittstellen $put(Schlüssel, Wert)$ und $get(Schlüssel)$ angeboten. Das bietet eine einfache Nutzung für Anwendungen, die Datenspeicherung

erfordern. Solche Anwendungen sind u.a. Dateisysteme. Die Schnittstellen von OpenDHT können u.a. mit XML-RPC über HTTP benutzt werden. So gewährleistet man die einfache Nutzung des Dienstes sogar für Benutzer hinter Firewall oder NAT. In OpenDHT kann man einen beliebigen Schlüssel für den gespeicherten Wert wählen. Um das Problem zu vermeiden, dass die beliebten Schlüssel wie z.B $H(internetrouting)$ reserviert werden, können verschiedene Daten unter dem gleichen Schlüssel gespeichert werden. Ein $get()$ liefert dann alle Werte zurück, die unter diesem Schlüssel gespeichert sind. Der Klient soll den von ihm gesuchten Wert identifizieren. Ein Schlüssel ist in OpenDHT 160 Bits lang. Wenn ein Klient Werte in der DHT speichern will, muss der Klient einen TTL-Parameter beim Aufruf der $put()$-Methode übergeben. Der TTL-Parameter bestimmt die Lebensdauer des gespeicherten Wertes. Die Verwendung solcher TTL-Werte hat den Nachteil, dass der Klient seine Daten erneut speichern und dabei die TTL-Werte intern behalten und überwachen muss. Der Vorteil bei der Verwendung von TTL ist der Verzicht auf die Garbage-Collection. So wird es sichergestellt, dass keine Daten in der DHT bleiben, die von keiner Anwendung und keinem Benutzer gebraucht werden. Eine Art von Authentifizierung ist in OpenDHT implementiert.

Zum Speichern gehört auch das Löschen. Das wird bei OpenDHT durch zwei Möglichkeiten realisiert:

1. Die Werte werden nicht mehr aktualisiert und nach Ablauf ihrer TTL werden sie gelöscht.
2. Es gibt einen simulierten $remove()$-Aufruf.

$remove()$ ist nicht mehr als ein $put()$, dessen Wert einen Hinweis zu dem Schlüssel, den man löschen möchte, gibt. Ein $remove()$ wird zwar von der Benutzerseite als $remove(Schlüssel, TTL)$ aufgerufen. Es wird aber intern als $put(Schlüssel, Spezialwert)$ realisiert. $remove()$ wird dann auf den Knoten gespeichert, der den originalen zu löschenden $put()$ hat. Der verantwortliche Knoten erkennt diesen $Spezialwert$ des $put()$s und entfernt dann den gesuchten Schlüssel. Beim TTL-Wert des $remove()$s ist zu beachten, dass er größer als der TTL-Wert des zu löschenden Schlüssels sein soll. Wenn die Werte eines Knotens repliziert werden kann es vorkommen, dass ein $put()$ durch eine Replikation wieder hergestellt wird, der $remove()$ aufgrund einer abgelaufenen TTL aber nicht. Das kann zur Inkonsistenz innerhalb von OpenDHT führen.

Ein $update()$-Aufruf wird durch einen $remove()$- und danach einen $put()$-Aufruf simuliert. Das kann wiederum zur Inkonsistenz bei den TTL-Werten führen. Wenn verschiedene Benutzer einen Wert gleichzeitig ändern wollen, kommt es zu Inkonsistenz bei der Reihenfolge der Veränderungen. "OpenDHT currently provides only eventual consistency." [Rhea et al., 2005, k. 5.3.3]

3.3 Wie OpenDHT die DHT-Aufgabe "Lookup" erfüllt

In dem Abschnitt 2 Seite 5 wurde die Lookup-Aufgabe erläutert. Die Herausforderung bei Lookup ist nicht das Auffinden eines Knotens in der DHT, sondern, dass dieser Knoten ein vom Benutzer bestimmtes Programm ausführen soll. Der Grundgedanke von OpenDHT ist, einen allgemeinen Dienst anzubieten. Deshalb wird die Aufgabe Lookup außerhalb der DHT von OpenDHT ausgeführt. Das heißt, dass die Knoten, die beim $put()$ und $get()$ benutzt werden, nicht direkt zur Ausführung von Lookup dienen. Die DHT von OpenDHT dient dann nur zur Auffindung des gesuchten Knoten. Der Knoten führt dann das angeforderte Programm aus. Dieser Knoten ist nicht unbedingt einer der Knoten von OpenDHT. Dies kann ein beliebiger Rechner sein. Die Lookup-Knoten sollen sich bei OpenDHT anmelden und sich zur Ausführung benutzerspezifischer Programme bereitstellen. Das wird in OpenDHT mit Hilfe von ReDiR realisiert. OpenDHT wird dann als ein Treffpunkt bzw. *Rendezvous* gesehen. Zu diesem Zweck bietet OpenDHT zur DHT-Aufgabe "Lookup" die folgenden Schnittstellen:

1. $lookup()$ zur Auffindung eines Knotens
2. $join()$ zur Bereitstellung eines Knotens zur Ausführung von Lookup-Aufgaben.

4 ReDiR mit OpenDHT

Die Schnittstelle $lookup()$ liefert einen Knoten zurück. Dieser Knoten soll einen Dienst anbieten, z.B. Weiterleiten von Paketen. So bezeichnet man das Weiterleiten von Paketen als eine Anwendung. Diese Anwendung benutzt die DHT zur Erfüllung von Aufgaben, die nicht unbedingt Aufgaben der DHT im Sinne von Abschnitt 2 Seite 5 sind.

ReDiR ist eine benutzerseitige Bibliothek. ReDiR soll auf allen Knoten installiert werden, die einer Anwendung gehören und einen Dienst mit OpenDHT anbieten.

4.1 Anwendungspezifische Dienste mit ReDiR

Die Knoten einer Anwendung müssen identifiziert, damit sie zu den Diensten der Anwendung beitragen, ohne mit anderen Anwendungen zu kollidieren. Die Anwendungen müssen eine eindeutige Bezeichnung haben. Diese heißt in ReDiR *namespace*. Alle Knoten, die einen Dienst anbieten oder benutzen, sollen diesen *namespace* betreten.

Damit die $lookup()$-Schnittstelle sinnvoll bleibt, soll sie nur die Knoten innerhalb eines *namespace*s und nicht aus der ganzen DHT zurückliefern. Um das zu realisieren, sollen Informationen über den Teilnehmer eines *namespace*s gespeichert werden. Eine naive Lösung wäre ein $put()$ mit dem Schlüssel *namespace* und einer Liste aller Teilnehmer eines *namespace*s als Wert. Der Nachteil dieser Lösung ist der Aufwand der Suche innerhalb dieser Liste sowie der Aufwand der Aktualisierung dieser Liste, solange alle Teilnehmer unter einem Schlüssel gespeichert sind.

Die Knoten eines *namespace*s werden bei ReDiR in einer Baumstruktur gespeichert, nämlich in einem zweidimensionalen Quadtree. Der Baum besteht aus Stufen. Der Wurzelkonten liegt in der Stufe 0, Kinder von dem Wurzelknoten liegen in der Stufe 1. Die Höhe des Baums ist die letzte Stufe + 1. Die Knoten werden innerhalb einer Stufe von links nach rechts abhängig von ihrem Schlüssel aufsteigend indiziert. Der erste Knoten links hat den Index 0.
Jeder Knoten in dem *namespace* wird mit den Koordinaten $(Stufe, Rang)$ eindeutig identifiziert. ReDiR fügt die Knoten dann in die DHT einzeln unter den Schlüssel $H(namespace, Stufe, Rang)$ ein. Beispielsweise wird der Wurzelknoten aus dem *namespace* "internetrouting" unter dem Schlüssel
$H(internetrouting, 0, 0)$=5e6ca0595a09f99f3f95ebc2f974e6dd39f93b0b[3] gespeichert.

In jeder Stufe wird der Schlüsselbereich in so viele Teile geteilt, wie es Knoten in der Stufe gibt. Jeder Knoten ist dann für den Teilbereich seines Ranges verantwortlich. Zum Beispiel ist der Knoten mit dem Index 1 (entspricht Rang 2) für den 2. Teilbereich verantwortlich. Die Abbildung 5 stellt ein Beispiel für den Baum dar. Die Knoten des Baumes sind als Karos mit ihren $(Stufe, Rang)$-Koordinaten dargestellt. Bei den Knoten des Baumes sind verschiedene Knoten des *namespace*s registriert. Im nächsten Abschnitt wird die Registrierung in den *namespace* erklärt.

[3]Hier wurde SHA1-1 als Hashfunktion verwendet.

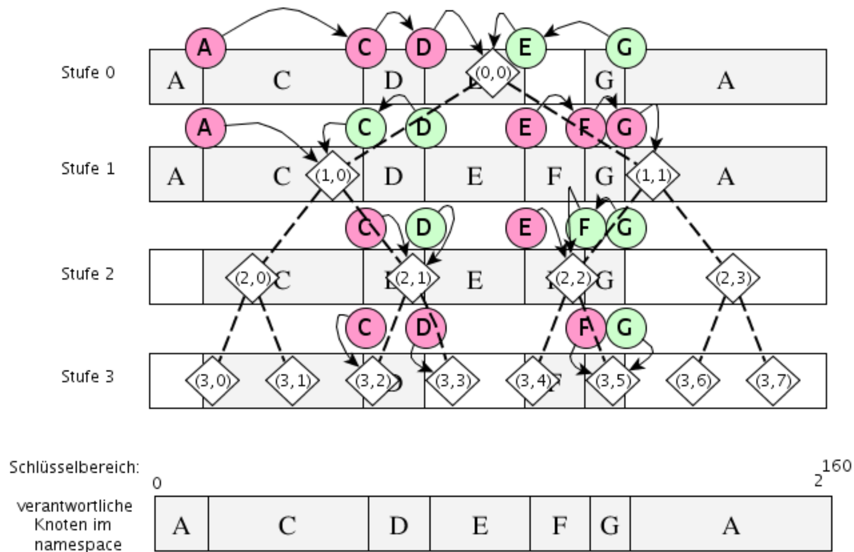

Abbildung 5: Baumstruktur in ReDiR

4.2 Registrierung in *namespace*

namespace umfasst alle Knoten, die einer bestimmten Anwendung dienen und einen Dienst für diese Anwendung anbieten. Hier wird beschrieben, wie neue Knoten in diesem *namespace* eingefügt werden. Sei n der neue Knoten. n ruft ein OpenDHT $get()$ bei dem Knoten k_s auf, der für den Teilbereich von n verantwortlich ist. Dabei muss n eine bestimmte Stufe s wählen, von der die Registrierung startet. Der $get()$-Aufruf liefert eine Liste von dem bei k_s eingetragenen Knoten zurück. n führt bei k_s ein $put()$ aus. Dabei wird n in die Liste der Knoten eingefügt. Falls der Schlüssel von n der kleinste oder der größte Schlüssel ist, wiederholt n den Vorgang bei allen verantwortlichen Knoten k_{s+1} in der niedrigeren Stufe und allen verantwortlichen Knoten k_{s-1} in der höheren Stufe. Bei den höheren Stufen wird der Vorgang entweder dann beendet, wenn der Wurzelknoten erreicht wird, oder dann, wenn der Schlüssel von n nicht der Größte oder der Kleinste in der Liste des verantwortlichen Knotens $k_{s-1,s-2...}$ ist. Bei den niedrigeren Stufen wird der Vorgang wiederholt, bis n der einzige Knoten in der Liste ist. Man kann die Registrierung vom Knoten F in der Abbildung 5 verfolgen. Die Registrierung fängt in der Stufe 2 beim Baumknoten $(2, 2)$ an.

4.3 Lookup in *namespace*

Bei einem Lookup wird innerhalb eines *namespace*s der Knoten gesucht, der für einen Schlüssel verantwortlich ist. Der Schlüssel von dem verantwortlichen Knoten ist größer (mod 2^{160}) als alle Schlüssel in seinem Verantwortungsbereich. Der Verantwortungsbereich eines Knotens ist zwischen seinem Schlüssel und dem Schlüssel des Nachbarknotens. Als Beispiel ist der Verantwortungsbereich des Knotens *grün* aus der Abbildung 3 der Bereich zwischen *grün* und *gelb*. Diesen verantwortlichen Knoten bezeichnet man als Nachfolgerknoten für den Bereich. In der Abbildung 5 ist zum Beispiel der Knoten C verantwortlich für den Bereich, der vor C und zwischen A und C liegt. Der Aufruf hat die Form $lookup(namespace, Schlüssel)$. Die Ausführung findet in den folgenden Schritten statt: Man fängt in einer Stufe s an und sucht den Knoten k_s, der für den Teilbereich des Schlüssels verantwortlich ist.

- Falls es bei k_s keinen Knoten gibt, der Nachfolger des Schlüssels ist, wiederholt man den Vorgang bei einer höheren Stufe $s - 1$.
- Falls es bei k_s Knoten gibt, die für die Bereiche um den Bereich von dem Schlüssel als Nachfolgerknoten dienen, dann wird der Vorgang bei einer niedrigeren Stufe $s + 1$ wiederholt.
- Falls es bei k_s einen Knoten gibt, der für den Schlüssel verantwortlich ist, dann wird die Suche beendet und der Knoten wird zurück geliefert.

5 Garantien beim Speichern

OpenDHT ist ein großer Dienst. Es wurde in der Arbeit [Rhea et al., 2005] diskutiert, welche Strategie zur Vergabe von Speicher an Klienten geeignet ist. Daher werden die Klienten anhand ihrer IP-Adressen eindeutig identifiziert. Klienten hinter NAT werden als ein Klient betrachtet. Eine Strategie, die pro Knoten gestaltet ist, weist Schwierigkeiten auf. Eine Überwachung jedes Klienten bei allen Knoten ist aufgrund der Größe von OpenDHT und der Natur von DHT und aufgrund der **Storage**-Aufgabe nicht realistisch. In OpenDHT wird eine Strategie gewählt, die den Klienten innerhalb einer Speicherplatte *gerecht* behandelt. Ein Vorteil dafür ist die Lastverteilung, denn Klienten, die ihre $put()$s angemessen auf verschiedene Knoten verteilen, werden immer Speicherplatz bekommen. So werden die gleichen Knoten nicht immer mit den alten $put()$s belastet.

Wie man in ReDiR gesehen hat, können verschiedene Klienten einen bestimmten Schlüssel verwenden, um einen *namespace* zu betreten. Der Schlüssel wird bei einem bestimmten Knoten verwaltet und verbraucht dessen Speicherplatz . Eine gerechte Verteilung der Speichervergabe verhindert, dass ein bösartiger Klient alle anderen Klienten blockiert, indem er den Speicherplatz in einem Knoten flutet. Die Autoren von [Rhea et al., 2005] glauben, eine Strategie zur Speichervergabe sei auf der Ebene der Speicherplatten geeignet. Eine globale Strategie ist grundsätzlich nicht ausgeschlossen. Die wird aber in OpenDHT nicht verwendet.

Die Dimensionen Speicher und Zeit

Wie im Abschnitt 3.2 beschrieben, benötigt die $put()$-Schnittstelle nicht nur den Schlüssel und den Wert, sondern auch einen TTL-Wert. Der TTL-Wert legt fest, wie lange die Daten im OpenDHT erhalten bleiben. Alle Knoten in OpenDHT müssen eine minimale Rate an Speichervergabe garantieren.

5.1 Garantieren einer Rate bei der Speichervergabe

Diese Rate wird mit dem folgenden Term definiert:

$$r_{min} = \frac{K}{TTL_{max}}$$

Wobei r_{min}: Rate der Speichervergabe, K: die Kapazität der Speicherplatte und TTL_{max}: der maximale erlaubte TTL-Wert. Um $put()$s von den Klienten zu akzeptieren, hat OpenDHT ein zweidimensionales Problem. Denn es muss nicht nur überprüft werden, ob der Speicherplatz verfügbar ist, sondern auch, ob die Vergabe dieses Speicherplatzes in der Zukunft die minimale Rate der Speichervergabe r_{min} bedrohen kann. Ein Beispiel für dieses Problem ist in der Abbildung 6 dargestellt. In der Abbildung 6-(a) wird ein $put()$ dargestellt, das viel Speicherplatz für kurze Zeit anfordert. In der Abbildung 6-(b) ist ein $put()$ abgebildet, das wenig Speicherplatz für lange Zeit anfordert. Als zweidimensionales Problem haben beide $put()$s den gleichen Verbrauch, denn sie benötigen die gleiche

Fläche in der Abbildung. Allerdings beeinflussen sie die minimale Rate r_{min} unterschiedlich. Die gestrichelte Linie in der Abbildung entspricht der Rate r_{min}. Wir sehen, dass die Akzeptanz des $put()$s in der Abbildung (a) zur Verletzung der Garantie von r_{min} führt. In der Abbildung (b) ist das nicht der Fall. So wird im Fall (a) nicht akzeptiert, in (b) aber doch. Diese Bedingung wird in OpenDHT wahrgenommen.

Formal schreibt man Folgendes: Ein $put()$ wird genau dann akzeptiert, wenn die folgende Bedingung aus [Rhea et al., 2005] erfüllt ist:

$$f(\tau) + x \leq K.$$

Dabei ist x: der angeforderte Speicherplatz, K: die Plattenkapazität, $\tau \in [0, TTL]$ dieses $put()$s und $f(\tau)$ ist eine Funktion, die den benutzten Speicherplatz vom Zeitpunkt des Aufrufs bis zu einem Zeitpunkt τ zurück liefert[4]. Weitere Informationen hierzu finden sich in [Rhea et al., 2005].

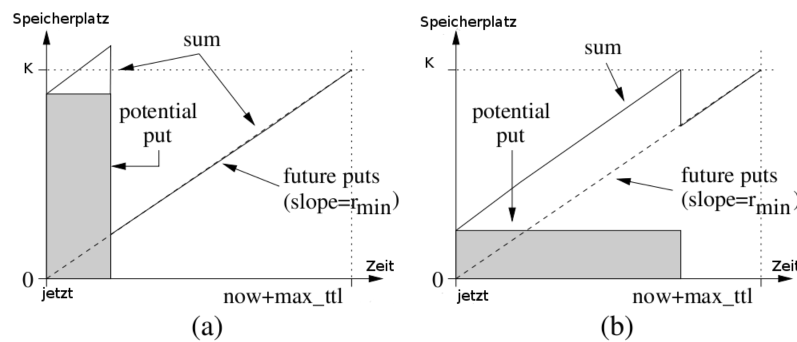

Abbildung 6: Beispiel von $put()$ aus [Rhea et al., 2005]

5.2 Verteilung der Speichervergabe zwischen Klienten

In OpenDHT betrachtet man nicht nur den Speicherplatz als Ressource, sondern auch die Zeitdauer, für die dieser Speicherplatz reserviert bleibt. Daher definiert man eine Einheit für die Ressourcen innerhalb von OpenDHT, die man als *commitment* bezeichnet.
Commitment ist definiert als das Produkt aus dem angeforderten Speicherplatz eines $put()$s und seiner TTL.

Bei der Verteilung der Speicherplatzvergabe (auf verschiedene Klienten) wird nur die Rate des Commitments betrachtet, und zwar nur zum Zeitpunkt der Anforderung. Unabhängig von den vorherigen Commitments werden alle momentanen Anforderungen erfüllt. Das entspricht dem Fair-Queuing-Algorithmus [Greenberg and Madras, 1992].
In OpenDHT hat jeder Knoten eine Warteschlage für jeden Klienten, der bei ihm ein $put()$ aufruft. Das angeforderte $put()$ wird zusammen mit der Zeit seiner Ankunft[5]in die Warteschlange eingefügt, bis sie voll ist.
Der Knoten sucht aus den Warteschlangen den Eintrag heraus, der die früheste Ankunftzeit hat. Dieser $put()$-Eintrag wird auf die Bedingung im Abschnitt 5.1 (Seite 10) überprüft. Falls die Bedingung nicht erfüllt ist, bearbeitet der Knoten keine $put()$s mehr und wartet, bis die $put()$-Anforderung erfüllt wird. $put()s$ mit früheren Ankunftszeiten können andere $put()$s bedrängen. So werden Klienten mit wenigen bzw. kleineren $put()$s schneller bedient.

[4] $f(\tau) = B(t_{jetzt}) - D(t_{jetzt}, t_{jetzt} + \tau) + \tau r_{min}$, wobei $B(t)$ den benutzten Speicherplatz zum Zeitpunkt t zurück liefert. $D(t_1, t_2)$ liefert den freigegebenen Speicherplatz zwischen t_1 und t_2 zurück.

[5] Die Ankunftszeit ist eine virtuelle Zeit ≥ 0. Sie ist das Maximum der Ankunftszeiten aller vorher akzeptierten $put()$s und der Ablaufzeiten dieser $put()$s.

6 Evaluation

Die Autoren von [Rhea et al., 2005] haben die Performanz von OpenDHT evaluiert. Im folgenden Abschnitt werden die Einstellungen und die Ergebnisse präsentiert.

Proof of concept

Die Konfiguration: OpenDHT wurde in PlanetLab [Bavier et al., 2004] auf 170-250 Maschinen installiert. Der Beobachtungszeitraum war von August 2004 bis Februar 2005. Ein Klient hat für diese Zeitdauer $put()$s und $get()$s ausgeführt. Der Klient führte jede Sekunde ein $put()$ aus. Die gespeicherten Daten haben die Größe b Bytes und den TTL-Wert l, wobei $b \in \{32, 64, 128, 256, 512, 1024\}$ und $l \in \{1Stunde, 1Woche, 1Tag\}$.
Die Daten wurden nochmals mit $get()$ aus OpenDHT gelesen und es wurden folgende Werte registriert:
Sind die Daten im Laufe der TTL noch verfügbar, werden die Latenzen des Aufrufs registriert. Sonst wird der Aufruf für eine Stunde wiederholt. Falls die Daten noch nicht zu finden sind, wird ein Fehler registriert.

Die Ergebnisse: Es wurden jeweils neun Millionen $put()$s und $get()$s durchgeführt. Es wurden nur 28 Werte als fehlerhaft oder verloren registriert. OpenDHT hat eine gewisse Stabilität gezeigt. Die Latenzwerte sehen aber nicht so viel versprechend aus.
In der Abbildung 7 sieht man die durchschnittliche Latenz sowie die Latenz von 95% der Aufrufe[6]. Im Allgemeinen liegt die Latenz bei nicht weniger als 1 Sekunde. Sie kann sehr oft um die 7 Sekunden liegen, manchmal auch bei 100 Sekunden. Die 95% Latenzen können als Referenz dafür akzeptiert werden, dass OpenDHT nicht immer schlecht ist. Ob die gleichen Knoten das Problem der langen Latenzen verursachen, oder ob es an der Zeitverteilung liegt, ist in [Rhea et al., 2005] nicht beschrieben.

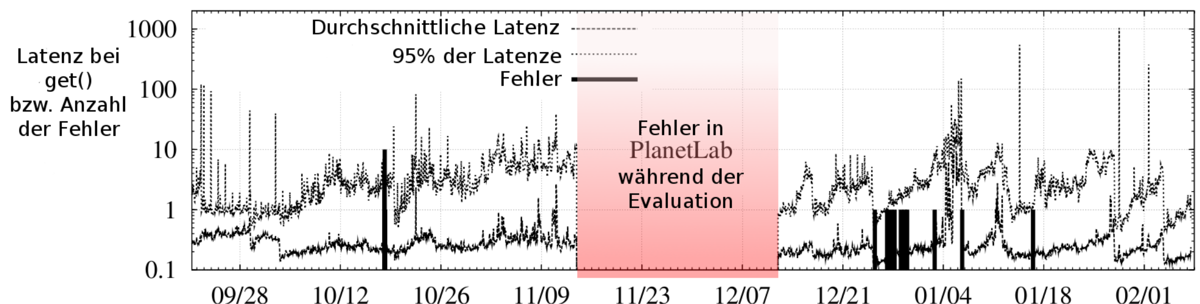

Abbildung 7: Eine Evaluation für 3,5 Monate aus [Rhea et al., 2005]

7 Zusammenfassung

In dieser Arbeit wurden die Aspekte von DHT erläutert. Es wurden die Ansätze von Bamboo kurz beschrieben. Der OpenDHT-Dienst ist eine Installation von Bamboo in PlanetLab. OpenDHT bietet eine DHT und stellt die Schnittstellen $get()$, $put()$ und $lookup()$ zur Verfügung. OpenDHT hat besondere Eigenschaften als eine Infrastruktur:

[6] Es wurde in [Rhea et al., 2005] nicht erwähnt, wie die 95% ausgewählt wurden. Möglicherweise wurden die 5% schlechteste Latenzen immer ausgefiltert.

1. allgemein verwendbar durch entfernte Aufrufe
2. kostenlos im PlanetLab
3. robust und zuverlässig durch die stabile Bamboo.

OpenDHT war in der letzten Zeit berühmt als eine DHT-Infrastruktur und wurde von verschiedenen Anwendungen verwendet. Die Nachteile bei OpenDHT sind die hohen Latenzen und der Ausstieg aus der Praxis seit dem 01.07.2009 [Rhea, 2009].

Bewertung des Originalpapers

In dem Originalpaper [Rhea et al., 2005] wurden die Eigenschaften und Schnittstellen von OpenDHT vorgestellt und gut erklärt. Das Paper präsentiert OpenDHT als offenen gemeinsamen allgemeinen Dienst. OpenDHT sollte am 1.Juli.2009 [Rhea, 2009] aus dem Betrieb genommen werden. Daher wird dieses Paper nicht mehr der Realität entsprechen und nur als "*proof of concept*" für Bamboo dienen.

Literatur

[Bavier et al., 2004] Bavier, A. C., Bowman, M., Chun, B. N., Culler, D. E., Karlin, S., Muir, S., Peterson, L. L., Roscoe, T., Spalink, T., and Wawrzoniak, M. (2004). Operating systems support for planetary-scale network services. In *NSDI*, pages 253–266. USENIX.

[Dabek et al., 2003] Dabek, Zhao, Druschel, Kubiatowicz, and Stoica (2003). Towards a common API for structured peer-to-peer overlays. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 2.

[Greenberg and Madras, 1992] Greenberg, A. G. and Madras, N. (1992). How fair is fair queuing? *J. of the ACM*, 39, 3:568–598.

[Ratnasamy et al., 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A scalable Content-Addressable network. In Guerin, R., editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 161–172, New York. ACM Press.

[Rhea, 2009] Rhea, S. (2009). Notice of my intention to stop maintaining opendht. Online In Internet. `http://opendht.org/`. [Stand 04.06.2009].

[Rhea et al., 2004] Rhea, S. C., Geels, D., Roscoe, T., and Kubiatowicz, J. (2004). Handling churn in a DHT (awarded best paper!). In *USENIX Annual Technical Conference, General Track*, pages 127–140. USENIX.

[Rhea et al., 2005] Rhea, S. C., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. (2005). OpenDHT: a public DHT service and its uses. In Guérin, R., Govindan, R., and Minshall, G., editors, *SIGCOMM*, pages 73–84. ACM.

[Rowstron and Druschel, 2001] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??

[Stoica et al., 2003] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw*, 11(1):17–32.