Hauptseminar DOOS SS 2002

12. Chord und CFS: Ein kooperatives, verteiltes Dateisystem

Alexander von Gernler
<Alexander.Gernler@informatik.stud.uni-erlangen.de>

15.07.2002

Kurzzusammenfassung

Chord ist ein verteilter Lookup-Service, auf dessen Basis Peer-to-Peer Anwendungen realisiert werden können. Spezielles Kennzeichen des Algorithmus ist ein Lookup-Aufwand, der nur logarithmisch in der maximalen Anzahl der teilnehmenden Knoten wächst.

Die Ausarbeitung[1] stellt das Chord-Projekt vor und präsentiert als realisierte Anwendung das verteilte Dateisystem CFS, das von den Stärken des zugrundeliegen den Chord profitiert.

1 Einleitung

1.1 Motivation

Eine alte Internetlegende (vgl. [NetpXX]) besagt, dass das Internet ehedem ins Leben gerufen wurde, um den USA ein Kommunikationsnetzwerk zu bieten, das auch einem Atomschlag widerstehen kann, weil es dezentral arbeitet.

Tatsächlich aber (vgl. [FrEs02]) hat sich längst eine Struktur herausgebildet, bei der der durchschnittliche Internetbenutzer als Konsument auftritt, während er seine gesamte Information von einigen wenigen Content Providern bezieht, die mittels großer Rechenanlagen in der Lage sind, den vielen zentralisierten Anfragen nachzukommen.

Chord beschreitet dagegen einen Weg, der tatsächlich ein verteiltes Dateisystem realisiert, das im obigen Sinne dezentral und ausfallsicher ist.

Vorteile: Das Chord Projekt verfolgt ein Systemmodell mit folgenden Eigenschaften:

- **Lastverteilung:** Durch die Verwendung einer Hashfunktion wird die Last statistisch gleich verteilt auf alle teilnehmenden Server.
- **Dezentraler Betrieb:** Keiner der teilnehmenden Knoten nimmt spezielle Aufgaben wahr, sondern alle Knoten sind gleichberechtigt und leicht ersetzbar.
- **Skalierbarkeit:** Die Kosten für einen Lookup wachsen logarithmisch in der Zahl der teilnehmenden Knoten, somit ist das System sehr gut skalierbar.

[1] `http://www4.informatik.uni-erlangen.de/Lehre/SS02/HS_DOOS/pdf/handout-sialgern.pdf` (Original) bzw. `http://www.rommel.stw.uni-erlangen.de:/~grunk/werke/chord.ps` (ständig aktualisiert)

- **Verfügbarkeit:** Durch ständige Aktualisierung der Sicht jedes Knotens auf das System ist der Ausfall einzelner Knoten tolerierbar, ohne dass die Fähigkeit zum Lookup beeinträchtigt werden würde.
- **Flexible Namensgebung:** Wegen des verwendeten Hashings ist das Konzept der Namensgebung in den darüberliegenden Schichten für die Funktionsweise von Chord völlig irrelevant. Der Namensraum von Chord ist eindimensional und nicht baumartig organisiert. Dies bietet den Anwendungen höherer Schichten große Freiheit in ihrer Namensvergabe.

Nachteile: Das Projekt leidet auch unter einigen Schwächen. Insbesondere folgende sind zu bemerken:

- **Keine Anonymität:** Die Papers des MIT (vgl. [Dabe01], [DBK+01], [DKK+01], [SMK+01]), die Chord und CFS vorstellen, betonen auffällig oft, dass Anonymität kein Designhintergedanke beim Erstellen von Chord gewesen sei, und dass sich Anonymität als zusätzliche Schicht noch implementieren lassen kann. Hiermit beschäftige ich mich in Abschnitt 4.1.

  Insbesondere bietet Chord nach den Kriterien, die in [Haus02] genannt werden, weder Herausgeber-, Leser-, Server- noch Dokumentenanonymität.
- **Kein Schutz gegen böswillige Teilnehmer:** In Kapitel 4.3 werden Konzepte für mögliche DoS-Attacken gegen das System vorgestellt.

1.2 Voraussetzungen

Weil sie von Chord bzw. CFS verwendet werden, ist ein grundlegendes Verständnis von Hashfunktionen und Kryptographie nicht unwichtig.

Ich gebe im Anhang A einen Überblick über die wichtigsten Grundbegriffe der beiden Themen, sowie Referenzen zu weiterführender Literatur.

2 Chord

2.1 Konsistente Hashfunktion

Jeder Knoten im Chord-System besitzt eine eindeutige ID n, die m Bit breit ist. Sie ergibt sich aus dem Hashwert über die IP-Adresse und einen zusätzlichen Parameter, den sog. virtuellen Knoten-Index. Dieser Index soll im Abschnitt 3.1.4 noch näher erläutert werden.

Wir benötigen also eine Hashfunktion h mit

$$h : \Sigma^* \to \Sigma^m \quad m \in \mathbb{N}$$

wobei Σ^m der Raum ist, in dem alle IDs des Systems liegen.

Konkrete Implementation: Für Chord wird die kryptographische Hashfunktion SHA-1 (vgl. [Schn96]) eingesetzt, die auf 160 Bit hasht. Das bedeutet, dass das System maximal 2^{160} Knoten besitzen kann. Dies ist nach heutigen Gesichtspunkten mehr als ausreichend. Wir konkretisieren

$$h : \Sigma^* \to \Sigma^{160}; \quad \Sigma = \{0, 1\} \quad \text{(Binäralphabet)}$$

2.2 Ringförmige Anordnung der Knoten

Die Knoten sind, wie Abbildung 2.1 zeigt, in einem logischen Ring in der Reihenfolge ihrer ID angeordnet.

Hier fällt auf, dass ortsnahe Knoten (z.B. selbes IP-Subnetz oder gleicher Host, aber anderer Knoten-Index) durch Anwendung der Hashfunktion sehr gleichmäßig im Ring (vgl. Anforderungen an kryptographische Hashfunktionen, A.1) verteilt werden und praktisch nicht an aufeinanderfolgenden Stellen aufzufinden sein werden.

Diese Erkenntnis kann anhand von Tabelle 2.1 in der Praxis schnell verifiziert werden.

In diesem Ring wird modular gerechnet, d. h. bei einem Überlauf in der ID beginnt ein nächster Durchlauf der Ringstruktur.

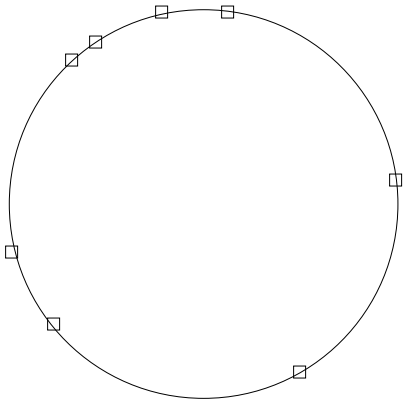

Abbildung 2.1: Eine zufällige Konstellation von Chord-Teilnehmern im virtuellen Ring

IPv4-Adresse	**Hashwert nach SHA-1 (hexadezimal)**
192.168.1.2	47b45a2710f977f44be86430c652195584de7d6d
10.1.1.2	5d24799200fda73b5270bf5a5baa4d666557b433
131.188.3.4	6ee3de708483d071c4f2eab0de1276b5398f5f80
10.1.1.1	81e0fb1230266b5f338346513d6adf506decda08
192.168.1.1	88dc1c83e5e3da8094423005d240b470c61e9a11
192.168.1.3	910734f0fd2aaf43dd5cc11b0377f8787115ddf9
172.16.1.2	cfc69d767feee8113eacf81102d181fb3d39bba0
172.16.1.1	da168c3e486b5e9618ef7aaa4b6cd14bb3217ae4

Tabelle 2.1: Beispiele für Hashwerte nach IP-Adressen, sortiert nach ihrer Reihenfolge im Chord-Ring

2.3 Gemeinsamer Adressraum von Schlüsseln (Datenblöcken) und Knoten

Auch Datenblöcke, die per DHash (vgl. Abschnitt 3.1) im System abgelegt werden sollen, erhalten durch die Hashfunktion eine eindeutige ID, den Schlüssel, der im selben Adressraum liegt wie die Knoten-IDs.

Für die Speicherung von Schlüsseln (Daten) im Ring gilt dann folgende Regel:

Verantwortlich für einen Datenblock x mit dem Schlüssel $n = h(x)$ ist der nächste aktive Nachfolger von n im Chord-Ring, `successor(n)`.

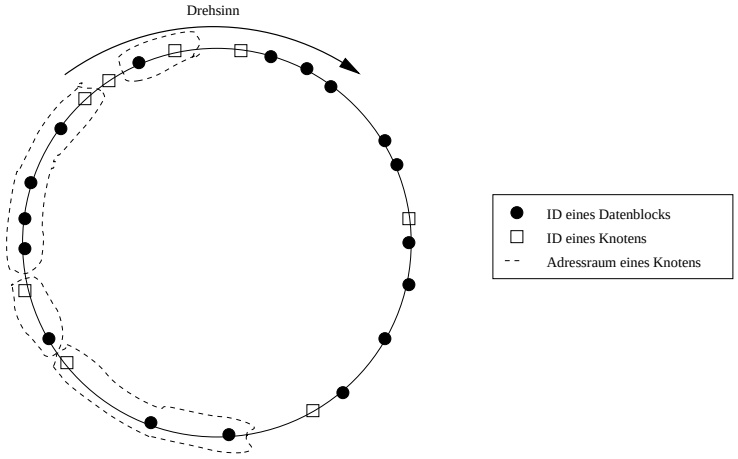

Abbildung 2.2: Die Chord-Teilnehmerknoten und Daten, für die sie zuständig sind. Nur teilweis gekennzeichnete Adressräume.

2.4 Der Lookup Algorithmus

2.4.1 Datenstrukturen jedes Knotens

Es werden eigene Daten pro Knoten gehalten die die Sicht des jeweiligen Knoten auf das System wiedergeben:

Nachfolgerliste	*Unbedingt Nötig* für das Funktionieren des Algorithmus. Beinhaltet eine Liste der r unmittelbaren, aktiven Nachfolgerknoten auf dem Ring. Vergleiche hierzu Tabelle 2.2, S. 5
Finger Table	*Optional* für das Verfahren, bringt aber einen erheblichen Gewinn an Geschwindigkeit und ermöglicht erst die logarithmische Anzahl von Schritten.
Zeiger auf den Vorgänger	*Optional*. Der Vorgänger könnte zwar auch durch eine iterierte Folge von Lookups jedes Mal aufs Neue bestimmt werden, aber dies ist für die Praxis zu aufwendig.

Die Größe r der Nachfolgerliste stellt Redundanz bereit und kann deshalb Feineinstellungen unterworfen werden. In der Literatur wird allerdings empfohlen, mit einem $r = 2 \log_2 N$ zu arbeiten. N stellt hierbei die maximale Anzahl der erwarteten Teilnehmer dar.

Es leuchtet ein, dass alle r Knoten gleichzeitig ausfallen müssen, um dem Gesamtsystem Schaden zuzufügen, denn nur dann ist eine Unterbrechung des Ring-Lookups möglich.

2.4.2 Durchführung von Lookups: Linearer Aufwand

Im naiven Ansatz wird auf der Suche nach dem Schlüssel n für einen Datenblock nun so lange von Knoten zu Knoten gesprungen, bis die ID des nachfolgenden Knotens gerade größer ist

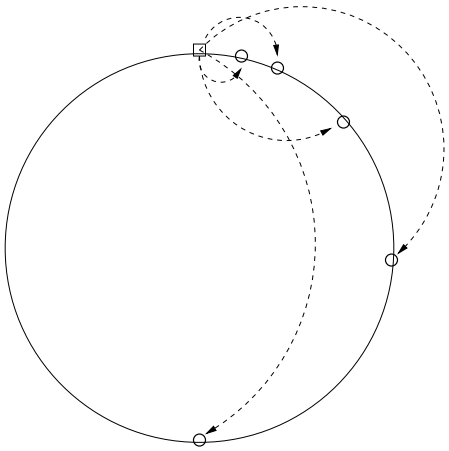

Abbildung 2.3: Ein Knoten und seine im Abstand von Zweierpotenzen entfernten Sprungziele, ab denen Nachfolgerknoten in einen bestimmten Eintrag in der Finger table gelistet werden.

Nr.	**ID**	**IP**
1	5d24799200fda73b5270bf5a5baa4d666557b433	10.1.1.2
2	6ee3de708483d071c4f2eab0de1276b5398f5f80	131.188.3.4
3	81e0fb1230266b5f338346513d6adf506decda08	10.1.1.1
4	88dc1c83e5e3da8094423005d240b470c61e9a11	192.168.1.1
5	910734f0fd2aaf43dd5cc11b0377f8787115ddf9	192.168.1.3
6	cfc69d767feee8113eacf81102d181fb3d39bba0	172.16.1.2

Tabelle 2.2: Beispiel für eine Nachfolgerliste des Knotens mit der IP 192.168.1.2, wenn die Maximalanzahl der Knoten $N = 8$ und somit die Länge der Nachfolgerliste $r = \log_2 N = 6$. Die Bereichsangabe, die das Intervall kennzeichnet, das der jeweilige Finger-Eintrag abdeckt, wurde hier aus Platzgründen weggelassen.

als n. Der so gefundene Knoten ist somit (vgl. Regel aus Kapitel 2.3) zuständig für den Datenblock.

Bei diesem Verfahren wird nur die Nachfolgerliste (Tabelle 2.2) benötigt. Das Verfahren ist korrekt und kann immer als Fallback benutzt werden, wenn die viel schnellere Methode gerade nicht funktioniert, weil z. B. die Finger Table nicht auf dem aktuellsten Stand ist (vgl. Abschnitt 2.4.3).

Da dieses Verfahren aufgrund der Ringstruktur aber einen mittleren Aufwand von $N/2$ Schritten benötigt, ist es für große Anzahlen von Knoten ineffizient.

2.4.3 Durchführung von Lookups: Logarithmischer Aufwand

Die Einführung der Finger Table, die sich nicht mehr in linearen Schritten, sondern in Schritten von Zweierpotenzen vom eigenen Knoten entfernt, beschleunigt den Suchvorgang nun. Der i-te Eintrag enthält jeweils den nächsten aktiven Nachfolger im Abstand von 2^{i-1} zum eigenen Knoten (vgl. Abbildung 2.3, 5). In unserem Fall hat die Finger Table also $m = \log_2 2^{160} = 160$ Einträge.

Interessanterweise stimmt der erste Eintrag der Finger Table mit dem direkten Nachfolger, `successor(n)`, überein.

Die Struktur, die der Finger Table zugrundeliegt, garantiert also, dass vom aktuellen zum gesuchten Knoten in jedem Schritt mehr als die Hälfte der Entfernung zurückgelegt wird. Hieraus folgt unmittelbar der in der Knotenzahl logarithmische Suchaufwand.

Der Algorithmus ist in Abbildung 2.4 beschrieben.

```
// ask node n to find id's successor
n.find_successor(id) {
    n' = find_predecessor(id);
    return n'.successor;
}

// ask node n to find id's predecessor
n.find_predecessor(id) {
    n' = n;
    while (id not in [n', n'.successor])
        n' = n'.closest_preceding_finger(id);
    return n';
}

// return closest finger preceding id
n.closest_preceding_finger(id) {
    for i=m downto 1 {
        if (finger[i].node in [n,id])
            return finger[i].node;
    }
    return n;
}
```

Abbildung 2.4: Der Lookup-Algorithmus als Pseudocode

2.4.4 Beispiele für Chord-Lookups

Anhand von Abbildung 2.5 möchte ich kurz auf das entstehen von Lookup-Pfaden nach dem Algorithmus aus Abbildung 2.4 eingehen.

`id1:` Es wird zu dem Eintrag in der Finger table gesprungen, der gerade noch kleiner ist als die gesuchte `id1`. Gleichzeitig stellt sich heraus, dass dieser Knoten und sein direkter Nachfolger auf dem Ring die gesuchte ID eingrenzen. Zuständig für die ID ist also der Nachfolger.

`id2:` Der Sprung über die Finger table approximiert zwar den Ort der ID schon sehr gut, aber es liegt noch ein weiterer Knoten zwischen der ID und dem aktuellen. Deshalb wird noch ein Sprung über die Finger table gemacht (Eintrag 1 stimmt mit dem direkten Nachfolger überein), so dass nun ein Knoten gefunden ist, der mit seinem Nachfolger die ID eingrenzt. Auch hier ist wieder der Nachfolger zuständig für die ID.

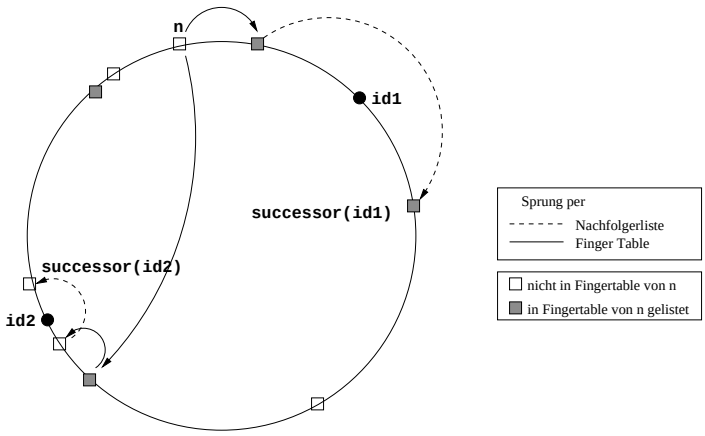

Abbildung 2.5: Zwei Beispiele für Lookups vom Knoten n aus

2.5 Flexibilität: Join und Leave

Es ist möglich, dem System während der Laufzeit dynamisch Knoten hinzuzufügen bzw. zu entfernen.

Chord baut daraufhin schrittweise seine Routingtabellen um, so dass die Änderungen in der Knotenkonstellation berücksichtigt werden.

Wichtig ist hierbei, das folgende zwei Bedingungen (in der Literatur als Invarianten bezeichnet) im Chord-Ring immer aufrecht erhalten werden:

1. Jeder Knoten kennt seine Nachfolger
2. Für einen Schlüssel k ist der Knoten successor(k) verantwortlich

2.5.1 Join

Ein hinzukommender Knoten n muss für einen erfolgreichen Join nacheinander folgende Dinge erledigen:

1. Die eigenen Datenstrukturen von n initialisieren. Es werden also die Finger table, die Nachfolgerliste und ein Zeiger auf den Vorgänger mit sinnvollen Werten gefüllt.

2. Die Vorgängerknoten von n darauf aufmerksam machen, dass n jetzt in den Ring aufgenommen wird. Diese tragen daraufhin n in ihre Datenstrukturen ein.

3. Die nächsthöhere Softwareschicht darüber informieren, dass nun ein anderer Knoten für die Datenblöcke zwischen dem Vorgänger von n und dem Knoten n selbst zuständig ist, nämlich n.

Ist kein weiterer Knoten bekannt, handelt es sich bei n folglich um den einzigen Knoten, und alle Zeiger in seiner Finger- und Nachfolgertabelle zeigen auf ihn selbst. Dies wäre aber trivial.

Für einen Join gehen wir immer von dem nichttrivialen Fall aus, dass dem Knoten n bereits ein Knoten n' bekannt ist, der selbst Mitglied im Chord Ring ist.

(Das Problem, immer einen ersten Kontakt selbst kennen zu müssen, ist bei Peer-to-Peer-Netzen weit verbreitet.)

n wird nun zum Update seiner internen Datenstrukturen sehr viele Informationen von Knoten n' bekommen, da n' eine vollwertige Sicht auf den Ring besitzt.

Für die Finger Table sind die m Positionen auf dem Ring durch einfaches Ausrechnen der Abstände in Abhängigkeit der eigenen ID bekannt. Alles, was noch zu tun ist, ist den Knoten n' nach dem jeweiligen Nachfolger für die errechnete Position in der Finger Table zu fragen.

2.5.2 Leave

Für ein Leave ist gar kein spezielles Vorgehen notwendig. Die Tatsache, dass Chord mit dem plötzlichen Ausfall eines Knotens bestens zurechtkommt, legt nahe, dass auch keine gesonderte Abmeldung aus dem System erforderlich ist.

2.5.3 Stabilisation

Die Möglichkeit, dass Knoten dem Ring zu beliebigen Zeitpunkten hinzutreten oder ihn verlassen können, bedeutet eine erhebliche Fluktuation, die durch ständige Aktualisierung der knoteneigenen Datenstrukturen kompensiert werden muss, soll der Ring nicht plötzlich unbrauchbar werden.

Deshalb werden folgende Stabilisationsmaßnahmen (vgl. Algorithmus in Abbildung 2.6) durchgeführt:

Der Zeiger auf den Vorgänger wird nach dem Hinzukommen eines neuen Knotens zunächst leergelassen. Dies stellt kein Problem dar, weil der Algorithmus diesen Zeiger nicht unbedingt benötigt. Stellen andere Knoten fest, dass sie den Knoten n als Nachfolger haben, rufen sie dessen `notify()`-Funktion auf. Hierdurch wird der Zeiger sinnvoll gesetzt.

Der Zeiger auf den Nachfolger wird periodisch geprüft und gesetzt durch den regelmäßigen Aufruf von `stabilize()`.

Die Finger Tabelle wird auch periodisch aktualisiert, allerdings aus Effizienzgründen jeweils nur ein zufällig ausgewählter Eintrag (vgl. hierzu `fix_fingers()`). Einen guten Zufallszahlengenerator vorausgesetzt, ist dieses Verfahren nicht nur effizient, sondern auch fair, weil gleichverteilt.

```
n.join(n') {
predecessor = NULL;
successor = n'.find_successor(n);
}

// periodically verify n's immediate successor,
// and tell the successor about n
n.stabilize() {
        x = successor.predecessor;
        if (x in [n, successor]) {
                successor = x;
        }
        successor.notify(n);
}

// n' thinks it might be our predecessor
n.notify(n') {
        if ((predecessor == NULL) || n' in [predecessor, n])
                predecessor = n';
}

// periodically refresh finger table entries
n.fix_fingers() {
        i = random(FINGERTABLESIZE);
        finger[i].node = find_successor(finger[i].start);
}
```

Abbildung 2.6: Der Stabilisations-Algorithmus von Chord in Pseudocode.

3 Praktische Anwendung: CFS

Basierend auf den Eigenschaften des Chord-Ringes und als sogenannter „proof of concept“ wird nun das verteilte, redundante und fehlertolerante Dateisystem CFS als Anwendung von Chord vorgestellt.

CFS ist nicht etwa Fiktion, sondern bereits implementiert, und arbeitet laut Aussage der Autoren auf freien Unices wie Linux (alte Versionen mit Kernel-Patch für SFS, vgl. [SFS]), Open- oder FreeBSD.

Die Designhintergedanken des Filesystems erläutere ich in Abschnitt 3.2.2 näher.

Das Dateisystem CFS besteht aus drei Schichten:

CFS	Interpretiert die Blöcke als Teile von Dateien oder Verwaltungsstrukturen des Dateisystems. Stellt eine Dateisystemfunktionalität nach oben bereit.
DHash	Speichert zuverlässig unstrukturierte Datenblöcke und bietet deren Abruf nach oben an.
Chord	Verwaltet die Routingtabellen, um bestimmte Schlüssel im System wiederzufinden. Bedient sich dabei einer speziellen Lookup-Strategie, die bei N Knoten nur $log_2 N$ Lookups benötigt.

3.1 DHash

3.1.1 Aufgabe

Die Chord-Schicht stellt die Möglichkeit bereit, anhand des Keys (der sich aus dem Hash eines Datenblocks ergibt) den zuständigen Knoten zu finden, der den Block speichert. Allerdings bleibt die Aufgabe der persistenten und redundanten Speicherung der Datenblöcke, sowie ein Caching derselben der DHash-Schicht überlassen, denn vom Filesystem wird schlicht die Bereitstellung der Blöcke ohne weitere Interpretation verlangt.

DHash stellt dafür folgende Schnittstelle bereit:

Aufruf	**Beschreibung**
`put_h(block)`	Berechnet den Hashwert des übergebenen Blocks `block` und speichert ihn an der errechneten Adresse auf dem Knoten, der laut Chord dafür zuständig ist.
`put_s(block, pubkey)`	Wird für root-Blöcke benutzt (vgl. Abschnitt 3.2.2). Der gegebene Block *muss* mit dem Private Key signiert sein, der mit dem übergebenen Public-Key `pubkey` korrespondiert. Der Block wird dann unter dem Hashwert des Public- Keys gespeichert.
`get(key)`	Holt den zu dem Hashwert `key` passenden Block vom Server und liefert ihn an den Aufrufer zurück.

3.1.2 Redundanz durch Replikation

Jeder Block, der gespeichert wird, wird von DHash nicht nur an der zuständigen Knoten, sondern auch noch an ihren k Nachfolgern abgelegt, wobei $k \leq r$ je nach Sicherheitsbedürfnis zu wählen ist, mit r als Länge der Nachfolgerliste. Es leuchtet ein, dass eine redundante Speicherung über die in der Nachfolgerliste bekannten Knoten hinaus absolut sinnlos ist. Die Daten würden zwar dann vorliegen, könnten aber nicht zugegriffen werden, da der Chord-Ring schon zerbrochen ist, wenn die r Nachfolger gleichzeitig ausfallen.

Bei Ausfall eines Knotens ist durch das Chord-Verfahren ohnehin schon gegeben, dass dann der Nachfolger des ausgefallenen Knotens für die Daten zuständig ist. Also sind keine weiteren Schritte erforderlich, und die gewünschten Daten sind bei einem Ausfall des Vorgängers bereits auf dem richtigen Knoten vorrätig.

Wir erinnern uns, dass durch die Hash-Technik praktisch garantiert ist, dass die k Nachfolger örtlich sehr gut getrennt von der ursprünglichen Speicherung aufbewahrt werden. Damit wird der Fall abgefangen, dass durch die Unerreichbarkeit eines kompletten Subnetzes auch ein zusammenhängender Teil des Chord-Rings ausfällt, und somit der gesamte Ring unbrauchbar wird.

3.1.3 Caching entlang des Lookup-Pfads

Durch das logarithmische Springen bis zum Ziel überschneiden sich die Zugriffspfade mit hoher Wahrscheinlichkeit nicht erst beim Zielknoten selbst, sondern berühren vorher auch häufig einen oder mehrere gleiche Knoten auf dem Weg zum eigentlich verantwortlichen Knoten. (vgl. Abbildung 3.7).

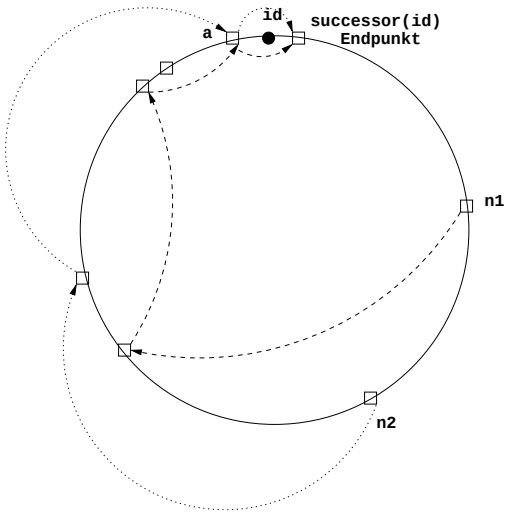

Abbildung 3.7: Überschneidung der Zugriffspfade von $n1$ und $n2$ bereits bei a

Dadurch macht es Sinn, nach einem erfolgten Zugriff auch auf allen Knoten, die im Lookup-Prozess involviert waren, den angeforderten Datenblock zwischenzuspeichern

Der Block-Cache pro Host wird hierbei im LRU[2]-Verfahren verwaltet und ist komplett unabhängig von der Replikation. Die Cachegröße ist ein Parameter, der dann im Betrieb den Feineinstellungen unterworfen sein wird.

3.1.4 Lastverteilung

Um verschieden gut ausgestattete Rechner (wir erinnern uns an den Peer-to-Peer-Gedanken) im Netz ihren Kapazitäten entsprechend auszulasten und auch auf aktuelle Lastsituation einzugehen, existiert die Möglichkeit, jedem Rechner durch die Verwendung mehrerer virtueller

[2]Least Recently Used, vgl. etwa [Hauc01]

Server zusätzliche IDs auf dem Chord-Ring zu erteilen. Es können auch dynamisch virtuelle Server nachgestartet oder wieder beendet werden.

Da die Hashfunktion wie im Anhang A.1 erwähnt, der Gleichverteilung genügt, wird jeder aktive Server einen in etwa gleich großen Anteil an zu speichernden Datenblöcken auf dem Chord-Ring bekommen. Mehrere virtuelle Server zu betreiben, heißt also, mehr Daten speichern zu müssen.

Mit diesem Verfahren kann die Last zwischen hardwareseitig unterschiedlich gut ausgerüsteten Rechnern statisch oder dynamisch (noch nicht implementiert) nivelliert werden.

Anmerkung: Die Verwaltung mehrer virtueller Server zieht einen zusätzlichen Programmieraufwand nach sich, um von der gegebenen Lokalität der virtuellen Knoten auch wirklich zu profitieren. Andernfalls würden Server auf der selben Maschine ihr seperates Teilwissen über das Netz nicht gegenseitig nutzen können, und es würde sehr viel Overhead entstehen. Darauf soll hier aber nicht weiter eingegangen werden.

3.1.5 Persistenz

Daten auf dem Chord-Ring haben nur eine begrenzte Lebensdauer, da der DHash-Layer nicht selbst entscheiden kann, welche Daten noch benötigt werden, und welche schon veraltet sind. Die Lebensdauer kann beim Speichern in DHash angegeben und auch periodisch verlängert werden.

Das Löschen von Datenblöcken ist also nur ein Nicht-Verlängern der Lease-Time des jeweiligen Blocks.

Dieses Vorgehen ist sehr bequem und entspricht genauso wie der problemlose Leave einzelner Knoten (vgl. Abschnitt 2.5.2) der Grundphilosophie hinter Chord.

3.2 Dateisystemschicht: CFS

3.2.1 Aufgabe

CFS stellt ein Filesystem mit hierarchischer Struktur, Verzeichnissen und Dateien bereit. Durch DHash wird bereits die persistente und zuverlässige Speicherung einfacher Datenblöcke, zugreifbar durch einen Schlüssel, den Hashwert, ermöglicht.

3.2.2 Konzept

In einem weltweiten CFS-System ist es vorgesehen, viele verschiedene Filesysteme bereitzustellen. Die Filesysteme sind hierbei read-only, nur der Autor kann Veränderungen vornehmen.

Dem Benutzer bleibt dann selbst überlassen, aus welchem Angebot er wählt. Sichergestellt ist, dass er darauf vertrauen kann, das das Filesystem wirklich von demjenigen Autor stammt, dessen kryptographischen Public-Key er kennt.

Es wird generell zwischen *Datenblöcken* und *Verwaltungsstruktur* unterschieden. Das Konzept erinnert an Log-structured Filesysteme (vgl. hierzu [Hauc01]), die auch mit Verzeigerung zwischen Verwaltungsstruktur und Datenblöcken arbeiten.

Datenblöcke	Aus der Gesamtheit der Datenblöcke für einen Eintrag ergibt sich die gespeicherte Datei. Die Datenblöcke werden unter der ID abgelegt, die ihrem Hashwert entspricht
Verwaltungs-struktur	Filesystem-Metadaten. Ein normaler Verwaltungsstruktur-Block benennt ein oder mehrere Files, indem er die dazugehörigen Datenblöcke anhand ihrer ID aufzählt. Bekanntlicherweise ergibt sich hieraus die Datei. Analog dazu ergibt sich ein Verzeichnis aus der Aufzählung seiner Bestandteile
Root-Block	Die Besonderheit an diesem Block (der sonst als normales Verzeichnis erscheint) ist, dass er nicht unter seinem Hashwert gespeichert wird, sondern unter dem Hash über den Public Key des asymmetrischen kryptographischen Schlüssels, mit dessen Private Key er signiert wurde.

Durch die Signatur auf dem Root-Block wird die Authentizität des Filesystems sichergestellt, da nur der Autor selbst in der Lage ist, den Block zu signieren. Mithilfe der Public-Key-Kryptographie kann sich auch jeder Teilnehmer selbst überzeugen, dass es sich um das echte Filesystem handelt. Auch ein Einfügen oder Verändern von Dateien unterhalb des Root-Blocks ist nicht möglich, da sich dadurch ja der Hashwert der Verzeichnisknoten im Baum oder der Datenblöcke selbst ändern würde (vgl. Abbildung 3.8). Dadurch wieder würden die veränderten Daten irgendwo im System zu liegen kommen, wo nicht auf sie referenziert wird.

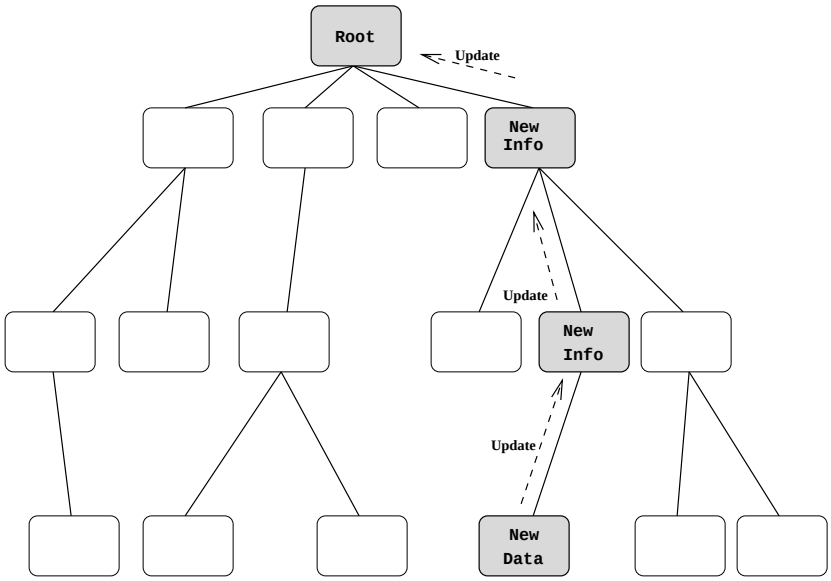

Abbildung 3.8: Die Änderung eines einzigen Datenblocks zieht rekursiv die Änderung von Filesystem-Metadaten bis schliesslich zum Root-Block nach sich.

4 Ausblick

CFS ist zwar schon implementiert, seine Verwendung aber noch eher theoretischer Natur. Vor allem vermisst der Benutzer noch die Möglichkeit, im Ring bestimmte Objekte (Daten)

zu suchen, wofür aus Performancegründen auch ein Indexierungsmechanismus auf Filesystemebene nötig wäre.

Desweiteren sind mir beim Durchlesen der Quellen mehrere Gedanken gekommen, die ich abschliessend kurz zur Diskussion stellen will.

4.1 Einführung der Anonymität zerstört Performance

Die Autoren weisen in den Arbeiten, die sich mit Chord beschäftigen, ausdrücklich darauf hin, dass Chord nicht im Hinblick auf Anonymität, sondern Performance konzipiert wurde. Interessant wäre zwar die Implementation einer Anonymisierenden Schicht nach dem Prinzip von Mixmaster-Kaskaden und ähnlichen Mechanismen, um Sender- und Empfängeranonymität zu gewährleisten. Doch leider ist zu erwarten, dass sich in diesem Fall Anonymität und Performance gegenseitig ausschliessen, schliesslich könnte man nach einer Einführung der anonymisierenden Schicht nicht mehr auf direktem Weg von logarithmischen Lookup-Zeiten profitieren.

Dass Sicherheit und Bequemlichkeit einander oft genug diametral gegenüber stehen, ist eine grundlegende Erkenntnis, zu der ich auf das hervorragende Werk von Bruce Schneier [Schn00] verweisen darf.

4.2 IP-basierte Quota kein wirksamer Schutz

CFS sieht vor, den verbrauchten Speicherplatz pro publizierender IP-Adresse zu begrenzen. Dieses Vorgehen ist nicht unproblematisch: Es erfordert ein Überprüfen der IP-Adresse des publizierenden Teilnehmers. Abgesehen davon, dass falsche IP-Adressen von einem Teilnehmer, der ein Gateway in einem benachbarten Subnetz innehat, massenweise glaubhaft genug simuliert werden können, zerstört die IP-basierte Quota auch eine etwaige, wenn noch vorhandene Senderanonymität.

4.3 Attacke auf Chord: Aufbau eines Subringes

Böswillige Teilnehmer können bereits durch falsche Auskünfte über ihren direkten Nachfolger den Chord-Ring aufbrechen, schliesslich ist der böswillige Teilnehmer nicht ausgefallen und seiner Auskunft wird im Protokoll Glauben geschenkt, ohne die anderen $r - 1$ Nachfolgerknoten noch zu konsultieren.

Hierdurch kann ein einziger Teilnehmer durch den Aufbau eines separaten Subrings einen normal arbeitenden Chord-Ring sabotieren.

Diese Attacke ist als „Partitionierung“ in [SiMo02] zu finden.

4.4 Chord zum Selbermachen

Natürlich kann man Chord auch selbst testen - eine Implementation existiert bereits. Hilfe beim Einrichten bietet hierbei das Chord-HOWTO (vgl. Link auf [ChorXX]).

Zum Einrichten wird allerdings das SFS-Paket (vgl. [SFS]) benötigt, das ebenfalls am MIT entwickelt wurde.

A Theoretische Grundlagen

A.1 Einweg-Hashfunktionen

Definition: Unter dem Begriff Hashfunktionen versteht man Funktionen h, die einen beliebig langen String über einem Alphabet Σ auf einen begrenzten String über dem selben Alphabet abbilden:

$$h: \quad \Sigma^* \to \Sigma^n; \quad n \in \mathbb{N}$$

Kollisionen: Es ist sofort ersichtlich, dass eine Abbildung von einem potentiell unendlich mächtigen Raum in einen begrenzten Raum nicht injektiv sein kann. Der geneigte Beobachter erwartet also zu Recht Kollisionen folgender Form:

$$\exists (x, x') \in \Sigma^* \times \Sigma^*; \quad x \neq x': \quad h(x) = h(x')$$

Ohne weiteren Beweis füge ich noch folgende wichtige Anforderungen an kryptographisch brauchbare Hashfunktionen hinzu. Dem interessierten Leser sei [Gern01] und [Buch99] empfohlen.

Einwegberechenbarkeit: Für gegebenes $y = h(x) \in \Sigma^n$ ist es praktisch unmöglich, *irgendein* $x' \in \Sigma^*$ (auch $x = x'$ erlaubt) zu berechnen, so dass $y = h(x')$. Mit anderen Worten: Es ist kein effizienter Algorithmus zur Berechnung von Urbildern bekannt.

Kollisionsresistenz: Für ein *beliebiges* $x \in \Sigma^*$ ist es praktisch unmöglich, irgendeine Kollision (x, x') zu finden.

Effiziente Berechenbarkeit: Um die Funktion auch praktisch einsetzen zu können, muss sie mit annehmbarem Aufwand berechnet werden können. Diese Einschätzung ist abhängig von der derzeit verwendeten Rechnergeneration und dem Einsatzgebiet der Hashfunktion.

Gleichverteiltheit: Die Wahrscheinlichkeit, durch eine zufällig gewählte Eingabe x auf einem bestimmten Hashwert $h(x)$ zu kommen ist für jeden Hashwert $h(x)$ gleich hoch.

A.2 Public-Key Kryptographie

Ein kurzer Abriss: Gegeben sei das Kryptosystem $\mathbf{K} = (\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ mit

Klartextraum $\mathcal{P}$	Menge aller Klartexte
Chiffretextraum $\mathcal{C}$	Menge aller verschlüsselten Texte
Schlüsselraum $\mathcal{K}$	Menge aller Schlüssel
Verschlüsselungsfunktionen $\mathcal{E}$	Alle Funktionen, die vom Klartext- in den Chiffretextraum abbilden
Entschlüsselungsfunktionen $\mathcal{D}$	Alle Funktionen, die vom Chiffretext- in den Klartextraum abbilden

Asymmetrische Kryptographie funktioniert nun so:

$$\forall e \in \mathcal{K} : \exists d \in \mathcal{K}: \quad \forall p \in \mathcal{P}: \quad D_d(E_e(p)) = p$$

In Worten: Man verschlüsselt den Klartext p zunächst mit dem Schlüssel e. Anschliessend gibt es eine Entschlüsselung vermöge d (das aber nicht notwendigerweise aus der Kenntnis von e folgen muss), so dass man aus dem Chiffretext wieder den Klartext p gewinnen kann.

Literatur

[Buch99] Buchmann, Johannes. *Einführung in die Kryptographie*. Springer, 1999. ISBN 3-4540-66059-3

[ChorXX] The Chord Project. `http://pdos.lcs.mit.edu/chord`

[Dabe01] Dabek, Frank. *A Cooperative File System, Master's Thesis*. Massachusetts Institute of Technology, 2001

[DBK+01] Dabek, F., Brunskill, E., Kaashoek, M. Frans, Karger, D., Morris, R., Stoica, I., Balakrishnan, H.. *Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service*. MIT Laboratory for Computer Science.

[DKK+01] Dabek, F., Kaashoek, Frans M., Karger, D., Morris, R., Stoica, I. *Wide-area cooperative storage with CFS*. MIT Laboratory for Computer Science.

[FrEs02] Freude, A., Espenschied, D. *Insert_Coin: Verborgene Mechanismen und Machtstrukturen im freisten Medium von allen.* `http://www.online-demonstration.org/insert_coin/`

[Gern01] von Gernler, A. *Einweg-Hashfunktionen in der Kryptographie*. Ferienakademie 2001. `http://www.rommel.stw.uni-erlangen.de/~grunk/werke/hash.ps`

[Hauc01] Hauck, Franz. *Systemprogrammierung I*. Vorlesung im WS 2000/2001 an der FAU Erlangen-Nürnberg. `http://www4.informatik.uni-erlangen.de/Lehre/WS00/V_SP1/`

[Haus02] Hausner, Christian. *Freehaven und Publius: Anonyme und zensurresistente Verbreitung von Informationen*. Hauptseminar DOOS an der FAU Erlangen-Nürnberg. `http://www4.informatik.uni-erlangen.de/Lehre/SS02/HS_DOOS/pdf/handout-sichhaus.pdf`

[NetpXX] Die Geschichte des Internets. `http://www.netplanet.org/geschichte/history.html`.

[Schn96] Schneier, Bruce. *Applied Cryptography*. Wiley and Sons, 1996, ISBN 0-471-11709-9

[Schn00] Schneier, Bruce. *Secrets & Lies - Digital Security in a Networked World*

[SFS] Self-certifying Filesystem. `http://www.fs.net.`

[SMK+01] Stoica, I., Morris R., Karger D. et al. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* Massachusetts Institute of Technology, 2001

[SiMo02] Sit, E., Morris, R. *Security Considerations for Peer-to-Peer Distributed Hash Tables*. Massachusetts Institute of Technology, 2002.

Version des Skripts (RCS)

`$Id: chord.tex,v 1.14 2002/07/06 12:19:41 grunk Exp $`