Einfuehrung in Distributed Hash Tables

Filip Martinovský
FB Informatik
FH Zittau/Görlitz
safimart@stud.hs-zigr.de

Philipp Wagner
FB Informatik
FH Zittau/Görlitz
siphwagn@stud.hs-zigr.de

Abstract

Distributed Hash Tables (DHT) stellen eine einfache Möglichkeit dar, hochverfügbare und verteilte Systeme zu konstruieren. Seit den viel beachteten initialen Veröffentlichungen zu konsistenten Hashverfahren [KLL+97] und einer Arbeit zu verteilten Hashtabellen [SMK+01] hat auch die Industrie den Wert der Technologie erkannt und beispielsweise in Amazons Dynamo [HJK+07] umgesetzt.

Dieser Beleg ist eine Einführung in die Theorie der verteilten Hashtabellen. Die Umsetzung von Chord [SMK+01], einem Protokoll für verteilte Hashtabellen, erfolgt in Erlang[1].

1 Motivation

Unbestritten sind viele Unternehmen in der heutigen Zeit existentiell abhängig von ihrem Datenbestand. Diese Daten müssen zu jedem Zeitpunkt verfügbar und für korrekte Ergebnisse möglichst konsistent sein.

Neben Verfügbarkeit und Konsistenz ist darüber hinaus die Latenzzeit eine wichtige Komponente in Geschäftsmodellen. Googles VP Marissa Mayer erklärte beispielsweise, daß eine Steigerung der Latenz um eine halbe Sekunde einen Einbruch des Suchvolumens von 20% bedeutet. Bei Amazon ist die Steigerung der Ladezeit von 100 *ms* für einen Rückgang der Verkäufe um 1% verantwortlich. [KL07]

In der Entwicklung verteilter Systeme ist es notwendig Architekturmuster offenzulegen, so daß das Verhalten hinsichtlich Konsistenz, Verfügbarkeit und Latenzverhalten modelliert und untersucht werden kann.

Distributed Hash Tables sind eine Basis für dezentrale, verteilte Systeme. Sie finden breite Anwendung in Peer-to-Peer Technologien, wie beispielsweise BitTorrent oder auch in anderen verteilten Speicherlösungen: das Backend von Amazon, *Amazon Dynamo* [HJK+07], basiert auf einem verteilten Hashverfahren.

1.1 Brewer's CAP Theorem

Bei der Konsktruktion von verteilten Systemen gibt es beschränkende Eigenschaften. In einer Keynote Speech postulierte Eric A. Brewer die Vermutung, daß unter den 3 Eigenschaften (Abbildung 1): (Starke) Konsistenz, Verfügbarkeit und Partition Tolerance bei einem verteilten System nur 2 gleichzeitig erfüllbar sind. [Bre00] Die Vermutung erhielt den Namen *CAP*, gemäß ihrer Eigenschaften:

Consistency stellt die Frage ob ein System nach einer Operation in einem konsistenten Zustand übertritt. In verteilten Datenbanksystemen muss dabei zwischen strong und weak consistency [2] unterschieden werden.

strong consistency Hierbei haben alle Klienten des Systems die gleiche Sicht auf die Daten, welche durch Two-Phase-Commit Algorithmen auch bei Dateiänderungen gewährleistet werden kann. Das beste Beispiel für strong consistency ist vermutlich das in Relationalen Datenbanksystemen verwendete *ACID* (**A**tomicity **C**onsistency **A**vailability **D**urability). Berichte aus der Industrie und Forschung legen jedoch nahe, daß ACID in verteilten Umgebungen in einer schlechten Verfügbarkeit resultiert. [HJK+07] [FGC+97]

weak consistency/eventual consistency Bei schwacher Konsistenz kann es passieren, daß nicht alle Benutzer des Systems die gleiche Sicht auf die Daten haben. Ein lesender Klient sieht also nicht zwangsläufig die letzten Schreibzugriffe.

Schwache Konsistenz lässt sich am besten an einem Beispiel erklären: Angenommen man hat ein System mit N Nodes, davon sind W Nodes bei jedem Schreibzugriff und R Nodes bei jedem lesenden Zugriff beteiligt. *Starke Konsistenz* lässt sich nur erreichen wenn $R+W>N$ ist, was bedeutet dass sich die Menge R und W überschneiden. Der lesende Zugriff hat jederzeit die Möglichkeit die letzte schreibende Änderung im System zu entdecken.[3]

Ein RDBMS mit synchroner Replikation ist beispielsweise: $N=2, W=2, R=1$, stark konsistent.

Ein System tritt in Eventuelle Konsistenz über, wenn $R+W<N$ ist. Das bedeutet, die Mengen R und W müssen sich nicht mehr überschneiden. Lesende Zugriffe müssen zwangläufig nicht mehr garantiert die letzten Schreibzugriffe im System sehen, es kann Inkonsistent sein und ist eventuell noch konsistent. Die Änderungen an Datensätzen werden durch Gossip Protokolle durch die beteiligten Instanzen propagiert.[HJK+07] [GBL+03]

Ein RDBMS mit asynchroner Replikation, beispielsweise $N=2, W=1, R=1$ ist nicht konsistent.

Availability gibt eine Aussage darüber ob das System die Fähigkeit besitzt trotz des Ausfalls von Nodes vollständig zu operieren.

Partition Tolerance bezieht sich auf die Fähigkeit eines Systems im Falle einer Partitionierung des Netzwerks zu arbeiten. Par-

[1] http://www.erlang.org

[2] starker und schwacher Konsistenz

[3] Beispiel von: http://practicalcloudcomputing.com/post/282603471/eventual-consistency-for-techies übernommen.

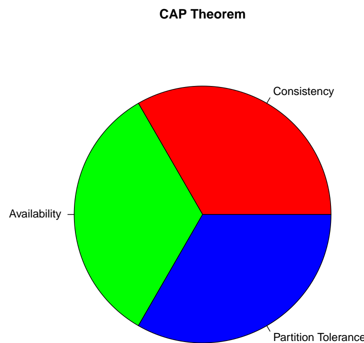


Abb. 1: CAP Theorem

titionen können sich beispielsweise durch getrennte Kommunikationsleitungen ausdrücken. Gilbert und Lynch definierten Partition Tolerance [GL02] als:

> No set of failures less than total network failure is allowed to cause the system to respond incorrectly.

Das CAP Theorem ist real und in der Welt der verteilten Systeme muss eventuelle Konsistenz zwangsläufig hingenommen werden. Denn angenommen die Anzahl an Nodes ist $N > 1000$ und $R = 1$, so ist es (besonders unter konkurrierendem lesendem/schreibenden Zugriff) äußerst unreal und ineffizient eine synchrone Replikation mit $W = N$ zu setzen um eine starke Konsistenz der Daten zu gewährleisten.

Opfert man jeweils eine der 3 Eigenschaften, ergeben sich die Fälle:

Consistency + Availability bei geclusterten Datenbanklösungen, klassischen RDBMS.

- Two-Phase Commit

Consistency + Partition-Tolerance bei verteilten Datenbanken.

- Systemweite Blockierung möglich
- Pessimistische Locking-Strategie

Partition-Tolerance + Availability bei DNS und Web-Caches.

- Optimistische Update Strategie
- Auflösen von Konflikten

1.2 Anwendungsbeispiele

Google BigTable ist *Consistency + Availability*. Google legt den Fokus auf starke Konsistenz und hohe Verfügbarkeit, denn eine Replikation der Daten erfolgt nicht auf Ebene der Datenbank sondern wird vom GFS (**G**oogle **F**ile **S**ystem) übernommen.

Amazons Dynamo ist *Availability + Partition Tolerance* und erfüllt die Merkmale der Verfügbarkeit und der Toleranz gegenüber Ausfällen. Dynamo verwendet modifizierte Distributed Hash Tables, die Gegenstand dieses Belegs sind und erreicht so Hochverfügbarkeit und Toleranz gegenüber Netzwerkfehlern.

Warum das Google File System keine Distributed Hash Tables verwendet, wird in [HGL03] erläutert. Das Google File System hat durch die zentrale Verwaltung und den Master Node sehr genaues Wissen über seine Nodes und kann so, ohne unnötige und das Netzwerk belastende Anfragen zu stellen, sehr spezielle Annahmen für Berechnungsschritte[4] machen um beispielsweise die Lokalität von Maschinen auszunutzen: *"[...] it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage."*.

2 Theorie

2.1 Hashing

Hash-Funktionen (aus dem Englischen *"to hash"* = zerhacken), im Deutschen auch als Streuwertfunktionen bekannt, stellen eine Abbildung von einer größeren Quellmenge in einen kleineren Zielraum (auch Schlüsselraum genannt) dar. Sie sind in vielen Bereichen der Informatik relevant, denn oft ist es eine Notwendigkeit Daten in kurzer Zeit in bestehende Verzeichnisse einzufügen oder auszulesen. Beispiele sind:

- Assoziative Arrays
- Hashindexe in Datenbanken
- Symboltabellen von Compilern

Darüber hinaus eignen sich kryptographische Hash-Funktionen ausserordentlich dafür, die Integrität von Daten zu sichern (Listing 1) und sind prädestiniert für Aufgaben wie Datenverschlüsselung (One-Way-Hash) oder Prüfsummenberechnung.

Eine ausführliche Einführung in Hashfunktionen kann an dieser Stelle nicht gegeben werden und die Thematik kann einer reichhaltigen Literatur entnommen werden, wie dem hervorragenden *"Art of Computer Programming, Volume 3: Sorting and Searching "* [Knu98] oder *"Numerical recipes: the art of scientific computing"* [wil07], .

Listing 1: md5 und sha1

```
philipp@banana:~/svn/dht-svn$ md5sum dht.tex
19a658e2233d444a1d78943bb8c765c2  dht.tex
philipp@banana:~/svn/dht-svn$ sha1sum dht.tex
41db64329bf8944ecb91725fabc1cfd198a12ba9  dht.tex
```

Heutzutage wird angenommen, daß kryptographische Hashfunktionen, wie *Message Digest 5 (MD5)* und *Secure Hash Algorithm 1 (SHA1)*, Daten möglichst gleichverteilen und *gute* Hashfunktionen sind.

Gute Hashfunktionen haben als Kriterien:

- **Datenreduktion**
- **Zufälligkeit**
- **Eindeutigkeit**
- **Effizienz**

Sichere Hashfunktionen sind darüber hinaus:

- **Kollisionsfrei** Hashwerte sollten eine geringe Wahrscheinlichkeit für Kollisionen aufweisen und möglichst eine Gleichverteilung der Zielmenge.
- **Unumkehrbar**

2.2 Consistent Hashing

Nicht jeder Inhalt im Internet wird gleichermassen angefordert. Durch populäre Downloads, oder oft angeforderte Inhalte, ist es

[4] MapReduce

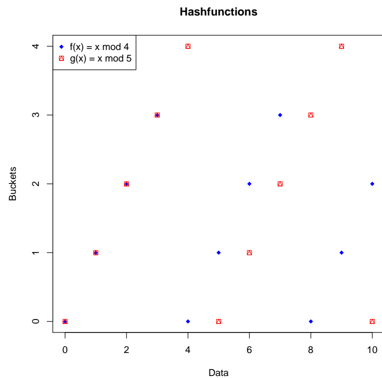


Abb. 2: Hash-Funktion: Zuordnung Daten zu Schlüssel.

notwendig Last zu verteilen und Caches einzurichten, so daß Inhalte aus dem Hauptspeicher geladen werden.[5] Ein intuitiver Ansatz die Last zu verteilen, wäre es URL[6] Adressen durch eine einfache Hashfunktion auf die Server zu abzubilden.

Audioscrobbler verwendete für seine Clients beispielsweise folgenden Hash[7]:

```
server = serverlist[hash(key) modulo serverlist.length];
```

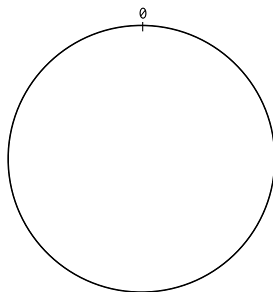
Das Problem daran ist: sobald ein Server hinzukommt oder ausfällt werden (im schlimmsten Fall alle) URL Adressen einem anderen Server zugewiesen. Der Inhalt im Cache wird hinfällig, siehe Abbildung 2, und Datenbank und Festplatte müssen belastet werden um alle Inhalte wieder in den Cache zu laden.

Konsistentes Hashing löst das Problem dieser wechselnden Sichten. Vorgeschlagen wurde es von einer Forschergruppe am Massachusetts Institute of Technology und in der Publikation *"Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web"* [KLL+97] auf ein theoretisches Fundament gestellt.

Die Idee von konsistentem Hashing lässt sich so beschreiben, daß sowohl für den Node[8] als auch für die Resource Hashfunktionen existieren:

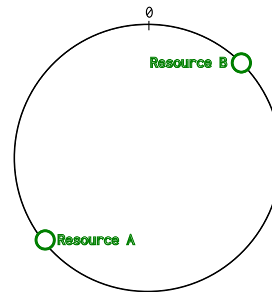
- Node: $h_{Node} : \{1, \ldots, n\} \rightarrow \{0, \ldots, 2^m - 1\}$
- Key: $h_{Resource} : \{1, \ldots, k\} \rightarrow \{0, \ldots, 2^m - 1\}$

Bei konsistentem Hashing stellt man sich den Schlüsselraum am besten als Ring vor:

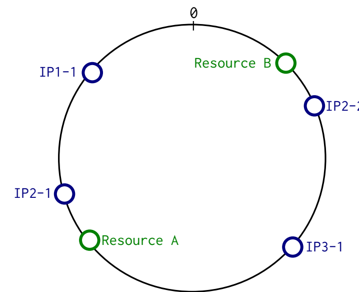


[5] Siehe memcache.
[6] Uniform Resource Locator
[7] http://www.audioscrobbler.net/development/ketama
[8] Server, Teilnehmer

Die Ressourcen werden in den Schlüsselraum gehashed, durch Anwendung kryptographischer Hashfunktionen wie MD5 (RFC1321[9]) oder SHA-1, da diese möglichst kollsionsfrei sind und so die Wahrscheinlichkeit von doppelten Einträgen minimiert wird.



Abschliessend werden die Nodes in den Schlüsselraum des gleichen Ringes gehashed. Es ist notwedig ein eindeutiges Merkmal zu hashen, beispielsweise die IP Adresse des Servers.



Der verantwortliche Node für eine Resource ist nun jener Node, welcher im Uhrzeigersinn ausgehend von der Resource als Erster getroffen wird. Ist dieser Server ausgefallen, so wird zum nächsten Server im Ring gegangen. Somit müssen im Falle eines Ausfalls nicht die Caches für alle Server geladen werden, sondern nur die Resourcen für die der Node nun verantwortlich ist.

3 Chord

Chord [SMK+01] ist ein Protokoll zur Implementierung eines verteilten Lookup Service und die konsequente Umsetzung von konsistentem Hashing. Dabei ist adressiert das Modell folgende Probleme:

- **Lastverteilung:** Durch die Gleichverteilung der (meist kryptographischen) Hashfunktion wird eine Lastverteilung gewährleistet. Die Teilnehmer sind so im Idealfall für eine gleiche Anzahl an Ressourcen im Ring zuständig.
- **Dezentralisierung:** Es werden bei Chord keine zentralen Server benötigt. Der Ring ist ein lose Kopplung gleichberechtigter Instanzen.
- **Skalierbarkeit:** In der Komplexitätsanalyse wird ersichtlich, daß eine Suche im Chord-Protokoll in $O(\log n)$ liegt und somit auch Netze mit vielen Teilnehmern möglich sind.
- **Verfügbarkeit:** Chord reorganisiert das Netz, wenn Nodes hinzukommen oder den Ring spontan verlassen. Eine periodisch ausgeführte Stabilisierungsoperation ist dafür verantwortlich, daß Änderungen im Ring von den Knoten propagiert werden.
- **Flexible Namensgebung:** durch die Hashfunktion ist der Schlüsselraum flach. Die Hashfunktionen macht keine Annahme über die Namensgebung der Schlüssel und die jeweilige Applikation kann dies selbst bestimmen.

[9] http://www.ietf.org/rfc/rfc1321.txt

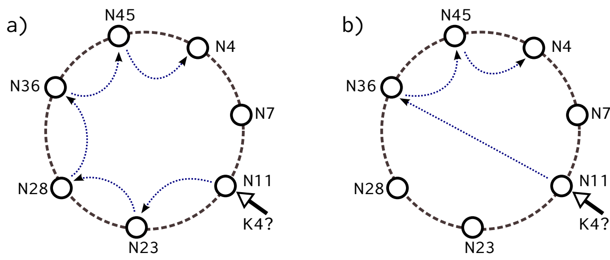


Abb. 3: Lookup

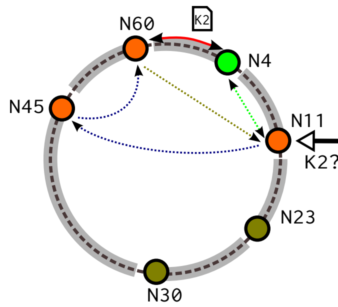


Abb. 4: Suche nach Schlüssel 2 in einem Ring der Größte 2^6.

3.1 Lookup

3.1.1 *Lookup ohne Finger-Informationen*

Hat ein Knoten keine zusätzlichen Routinginformationen, kennt er nur seinen Vorgänger und Nachfolger. Eine Suche nach einem Schlüssel würde eine sequentielle Suche werden, wie in Abbildung 3 beschrieben. Die Komplexität würde ein Aufwand von $O(n)$ bedeuten, und dies ist in großen Netzen nicht hinnehmbar.

3.1.2 *Lookup mit Finger-Informationen*

Um das System also zum Skalieren zu bringen, muss das Ziel entweder eine konstante Suchzeit $O(1)$ oder ein logarithmischer Aufwand $O(\log n)$ sein. Eine konstante Suchzeit kann erreicht werden, indem jeder Node den gesamten Ring kennt und Anfragen direkt an den verantwortlichen Node weitergeleitet werden. Dies bedeutet jedoch einen hohen Verwaltungsaufwand und stellt eine Speicherplatzanforderung für den Client dar. Chord ist deshalb für eine Suche in logarithmischer Zeit vorgesehen. Es werden nur m Einträge in der Routingtabelle, auch Fingertabelle genannt, benötigt und m ist die Bitlänge der Hashfunktion.

Der i-te Eintrag in der Fingertable ist der Knoten, der den 2^{i-1}-ten Eintrag im Ring verwaltet. Es ist ersichtlich, daß der erste Eintrag der Tabelle $k+2^0$ der Nachfolger des Knotens k ist. Das Ziel der Fingertable ist mit jedem Suchschritt den Suchraum innerhalb des Rings zu halbieren.

Der Lookup wird an einem konkreten Beispiel beschrieben: in Abbildung 4 wird in einem Ring der Größe 2^6 bei Knoten 11 nach dem Schlüssel 2 gesucht, die Fingertables der Knoten sind mit Listing 1 gegeben.

Die Suche läuft folgendermassen:

1. Node 11 wählt Node 45 aus seiner Fingertable, denn dieser ist der größte Node der Fingertable der vor Key 2 liegt.
2. Node 45 wählt Node 60 aus seiner Fingertable, denn Node 60 ist der Node mit dem höchsten Hashwert vor Key 2.
3. Node 60 hat als Nachfolger Node 4. Da Node 4 schon einen höheren Hashwert als 2 hat, ist Node 4 für Key 2 verantwortlich und eine Benachrichtigung des suchenden Knotens

Index	k + 2^{i-1} modulo 2^m	Knoten
Knoten k = 11		
1	12	23
4	19	23
5	27	30
6	43	45
Knoten k = 45		
1	46	60
4	53	60
5	61	4
6	13	23
Knoten k = 60		
1	61	4
...	...	...

Tab. 1: Fingertable für Knoten k=11, k=45, k=60

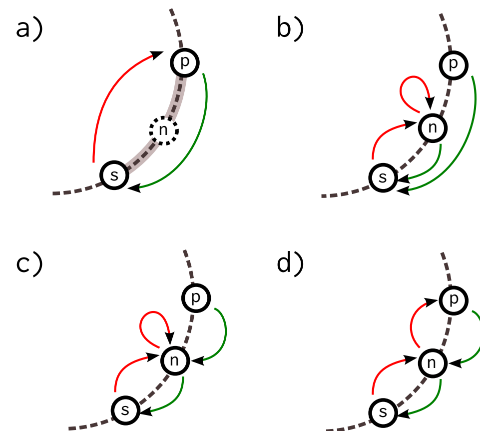


Abb. 5: Einfügen eines Knotens

erfolgt.

4. Datenaustausch zwischen Node 4 und Node 11.

Es sei an dieser Stelle auf den Beweis verzichtet, daß die Suchzeit in Chord in wirklich in $O(\log n)$ liegt. Denn die Beweisidee in [SMK+01] ist offensichtlich: durch Abbildung 4 und den Routinginformationen in Listing 1 ist einzusehen, daß sich mit jedem Schritt der Suchraum mit hoher Wahrscheinlichkeit halbiert. Die Suche nach einem Wert wird letztendlich zu einer binären Suche mit bekanntermaßen logarithmischen Aufwand.

3.2 Operationen

3.2.1 *Einfügen eines Knotens*

Soll ein neuer Knoten n hizugefügt werden so muss diesem bereits ein Knoten x aus dem Ring bekannt sein. Ausgehend von x kann der zu n zugehörige Nachfolger s im Ring gesucht werden, gemäß der thematisierten Lookup Strategie.[10], in Abbildung 5a dargestellt.

n setzt seinen Nachfolger-Zeiger auf s und informiert ihn über die Änderung. Ist n der Vorgänger von s im Ring (d.h. seine ID ist größer als die des urspünglichen Vorgängers p) setzt s seinen Vorgänger-Zeiger auf n, siehe Abbildung 5b.

[10] Die Position ist dabei abhängig von der gewählten Veschlüsselung aus MAC-Adresse, IP-Adresse... und Hash-Funktion

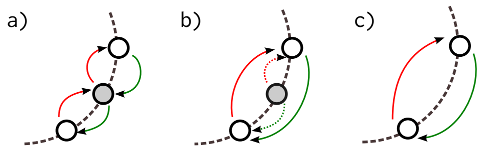


Abb. 6: Löschen eines Knotens

Bei der periodischen Ausführung der Stabilisierungsfunktion wird p feststellen, dass n der Vorgänger von s ist und aktualisiert dementsprechend seinen Nachfolger-Zeiger auf n (Abbildung 5c) und informiert seinen neuen Nachfolger, welcher seinen Vorgänger-Zeiger auf p setzen wird. Damit ist der Ring wieder in einem stabilen Zustand (Abbildung 5d) übergetreten und alle Routinginformationen sind konsistent.

3.2.2 Löschen eines Knotens

Für das Löschen eines Knotens ist es nicht unbedingt erfordelich eine Funktion zur Verfügung zu stellen, da sich der Ring beim Löschen eines Knotens periodisch stabilisiert. Es ist jedoch aus Sicht der Datensicherung und der Performanz wünschenswert, denn verlässt ein Knoten freiwillig den Ring[11] kann er die Stabilisierung autonom durchführen und deutlich beschleunigen.

Die erforderte Funktionalität ist einfach: der Node der den Ring verlassen möchte aktualisiert die Zeiger seines Vorgängers und Nachfolgers so, daß beide aufeinander zeigen, Abbildung 6b. Gleichzeitig wechselt der Node in einen Zustand in dem Lookups gleich an den Nachfolger weitergeleitet werden. Dadurch werden die Ergebnisse der gerade laufenden Stabilisierungsfunktionen so manipuliert, dass sich der Ring nach dem Löschen des Knotens in einem stabilen Zustand befindet (Abbildung 6c).

3.2.3 Stabilisierungsphase

Da die Topologie dezentral ist gibt es keine zentrale Instanz die die Konsistenz der Routinginformationen gewährleistet. Jeder Teilnehmer im Ring ist also dafür verantwortlich seine Routinginformationen auf dem neuesten Stand zu halten.

Knoten können dem Ring jederzeit verlassen oder beitreten, die Routinginformationen der einzelnen Knoten würden somit ungültig. Die für die Distributed Hash Table grundlegende Lookup Operation wäre nicht mehr möglich.

Die in [SMK+01] als Stabilisierung bezeichnete Aktualisierung der Routinginformationen wird bei jedem Knoten im Ring periodisch ausgeführt. Die Stabilisierung erfolgt in 3 Schritten:

1. **Nachfolger prüfen**
   Jeder Node x prüft, ob er noch der Vorgänger seines Nachfolgers s ist. Ist dies nicht der Fall, ist ein neuer Node n zwischen beiden aufgetaucht und x setzt seinen Nachfolger-Zeiger auf n, gleichzeitig wird der Vorgänger-Zeiger von n auf x gesetzt.

2. **Vorgänger prüfen**
   Eine Prüfung der Vorgänger-Zeiger ist notwendig um über den Ausfall eines Nodes notifiziert zu werden. Ist der Vorgänger-Zeiger nicht existent, so setzt er den Vorgänger-Zeiger auf sich selbst. [12]

   Sollte ein Knoten in der nächsten Stabilisierungsphase bemerken, daß sein Nachfolger ausgefallen ist, so wird der nächste aktive Knoten in der Fingertable als neuer Nachfolger gesetzt und die Zeiger aktualisiert.

3. **Fingertable aktualisieren**
   Nach den ersten beiden Stabilisierungsschritten kann davon ausgegangen werden, daß der Nachfolger des Knotens existiert und eine Anfrage gestartet werden kann. Um die Fingertable für den Node n aufzubauen, ist es notwendig die Nodes zu finden die für den $n+2^{i-1}$-ten Schlüssel zuständig sind, $i \in \{1,\ldots,m\}$.

4 Implementierung

Eine Umsetzung von Chord erfolgte in der Programmiersprache Erlang[13]. Die Softwareauswahl wurde für Erlang entschieden, denn mit dieser Sprache lassen sich prototypische Implementierungen verteilter Systeme effizient durchführen. Eine sehr gute und humorvolle Einführung ist für den interessierten Leser mit dem freien Buch [TH10] gegeben. Die folgende Einführung wurde mit Erlaubnis der Autoren aus [MWH10] übernommen.

4.1 Erlang

Erlang ist eine funktionale Programmiersprache die von Ericssons Computer Science Laboratory in den späten 1980er Jahren entwickelt wurde. Die Sprache, ähnlich wie Java von einer VM interpretiert, bringt ein Framework (*OTP*[14]) für die Entwicklung von parallelen, verteilten und fehlertoleranten Systemen mit. 1998 wurde die Sprache und die VM von Ericsson als *Open Source* zur Verfügung gestellt. Die Charakteristika von Erlang sind:

- Funktionen höherer Ordnung
- Prozesse und Message-Passing
- leichte Skalierbarkeit
- Soft Real-Time Fähigkeit

Variablen

In Erlang beginnt ein Name einer Variable immer mit einem Großbuchstaben. Werte werden mit Hilfe des Zuweisungsoperators = an Variablen gebunden. Es ist zu beachten, dass in Erlang *single assignment* gilt, das heisst eine Variable kann nur einmal an einen Wert gebunden werden:

```
1> Q = 4683.
4683
2> A = 30484.
30484
3> Q = A.
** exception error: no match of right
hand side value 30484
```

Atoms

Atoms sind durch Namen fest definierte IDs. Die Idee kommt aus der Programmiersprache *Prolog*. Atomare Datentypen müssen in Erlang immer mit einen Kleinbuchstaben beginnen oder werden in quotes geschrieben:

```
> atom1.
atom1
> 'Atom2'.
'Atom2'
> test@web.de.
'test@web.de'
```

Boolescher Datentyp

In Erlang existiert kein expliziter Datentyp für Wahrheitswerte. Wahrheitswerte werden durch die Atome `true` und `false` dargestellt.

[11] Also nicht durch einen Systemabsturz o.ä..

[12] Bemerkung: In der Fachliteratur wird der Zeiger auf *NULL* gesetzt, es hat aber keine Auswirkungen ihn auf sich selbst zu setzen.

[13] http://www.erlang.org

[14] Open Telecommunication Platform

Zahlen

Integer

Erlang bietet die Möglichkeit Integer mit verschiedenen Basen zu verwenden. Diese sind im Format `BASIS#WERT` einzugeben:

```
> -10.
-10
> 2#101010101010110.
21846
> 16#CAFEBABE.
3405691582
```

Float

Gleitkommazahlen werden gemäß dem 64-bit Format des IEEE754-1985 Standards abgespeichert (11 bit Exponent, 52 bit Mantisse):

```
> 1.2E10 - 1.2E-10.
1.2e10
> 1.231.
1.231
```

Tupel

Tupel dienen zur Speicherung einer festen Anzahl von Elementen die meist in Beziehung zueinander stehen. Ein Tupel kann unterschiedliche Datentypen enthalten und wird durch geschweifte Klammern gekennzeichnet:

```
> { 'Map', 16#BABE }.
{ 'Map', 47806 }
> tuple_size({ 'Map', 16#BABE }).
2
```

Um die Arbeit mit Tuplen zu vereinfachen gibt es in Erlang die Funktionen `element`, `tuple_size` und `setelement`.

Listen

Listen werden wie Tupel für das Abspeichern von Daten genutzt. Im Unterschied zu Tupeln haben Listen eine variable Anzahl an Elementen. Listen werden durch eckige Klammern `[]` gekennzeichnet.

```
> [ 'Map', 16#BABE ].
[ 'Map', 47806 ]
> length([ 'Map', 16#BABE ]).
2
> [ test1 | [ test2 | [] ] ].
[ test1, test2 ]
```

Pattern Matching

Erlang nutzt bei der Bindung von Variablen *Pattern-Matching* und ermöglicht es so Werte aus komplexen Datenstrukturen zu extrahieren:

```
> {person, Name, en} = {person, 'Thomas', de}.
** exception error: no match of
right hand side value {person,'Thomas',de}
>  {person, Name, de} = {person, 'Thomas', de}.
{person,'Thomas',de}
> Name.
'Thomas'
```

Es gibt auch spezielle Operatoren für das Pattern-Matching bei Listen:

```
> [Hd | Tl] = [test1, test2, test3].
> Hd.
test1
erl> Tl.
[test2, test3]
```

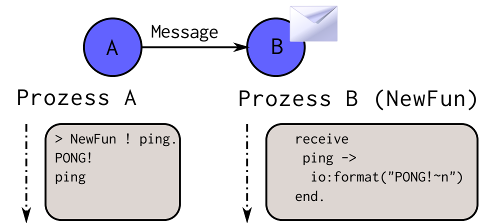


Abb. 7: Message Passing in Erlang

Funktionen

Funktionen können in Erlang ausschliesslich in Modulen definiert werden und können erst nachdem sie kompiliert wurden geladen werden. Die meisten rekursiven Funktionen nutzen das Pattern-Matching:

```
mymap(Fun,[]) ->
 [];
mymap(Fun,[Hd|Tl]) ->
 [Fun(Hd) | mymap(Fun,Tl)].
```

Dieses kurze Programm definiert die *map* Funktion. Als Funktion höherer Ordnung ruft *map* die Funktion `Fun` (eine anonyme Funktion) mit jedem Element in der Liste auf und liefert anschließend eine Ergebnisliste gleicher Kardinalität.

Nebenläufigkeit in Erlang

Die große Stärke von Erlang ist die Unterstützung von Nebenläufigkeit. Jede Funktion kann mittels der Funktion `spawn` aufgerufen werden und als ein neuer Prozess in der Erlang VM fungieren. Die Erlang VM verwaltet die Prozesse unabhängig vom Betriebssystem und verteilt selbstständig die Erlang VM Threads auf Betriebssystem-Prozesse.[15] Die Erlangprozesse sind sehr leichtgewichtig, da sie innerhalb der Erlang VM lediglich Funktionen sind und so der Overhead durch Kontextwechsel und Process Control Blocks entfällt. Der Rückgabewert von `spawn` ist die *Pid*[16] des neu erstellten Prozesses.

Message Passing

Weil ein Aufruf der Funktion `spawn` asynchroner Natur ist wird ein realisiert Erlang die Kommunikationen zwischen den Prozessen mit Hilfe des Message Passing. Nachrichten, Erlang Ausdrücke, werden an Prozesse mit dem `!` Operator gesendet. Die Nachricht wird in der Mailbox des empfangenden Prozesses gespeichert und mittels Pattern-Matching innerhalb der `receive` Klausel ausgelesen.

```
> NewFun = spawn(fun()-> receive
> ping -> io:format("PONG!~n")
> end end ).
<0.48.0>
> NewFun ! ping.
PONG!
ping
```

In diesem Beispiel wird ein Prozess erstellt, der innerhalb seiner *receive* Klausel auf die Nachricht `ping` wartet und bei Erfolg die Nachricht `PONG!` herausgibt und beendet. Dies ist dargestellt in Abbildung 7.

[15] http://www.defmacro.org/ramblings/concurrency.html

[16] Process identifier in der Erlang VM.

4.2 Chord

4.2.1 Aufbau

Das System wurde in 3 Schichten aufgeteilt:

Basis Schicht enthält Distanz- und Hashing-Funktionen die als Grundbausteine für den Chord-Algorithmus notwendig sind.

Chord Schicht ist der Kern der Implementierung. Diese Schicht setzt die Funktionen für Node-Join, Lookup und Stabilisierungsfunktion um, sowie Hilfsfunktionen für die Kommunikation zwischen Nodes.

Daten Schicht implementiert, aufbauend auf den unteren Schichten, die Funktionen für das Abspeichern der Daten und deren Lookup. Intern ist eine einfache Backup-Strategie umgesetzt indem ein Schlüssel zweifach redundant an einen Node und dessen Nachfolger verteilt wird, damit Daten nach Absturz eines Nodes dem Ring erhalten bleiben.

Die Nodes in dem Chord-Ring werden als Erlang-Prozesse implementiert. Um die Absturzgefahr dieser Prozesse zu minimieren, kommunizieren diese mittels Funktionen in getrenten Prozessen.

4.2.2 Stabilisierung

Die Stabilisierungsfunktion ist eine der wichtigsten Funktionen für den Chord-Ring. Ohne Stabilisierung wäre nach Absturz eines Nodes kein lookup mehr möglich und noch wichtiger: es könnte gar kein Chord-Ring aufgebaut werden.

Die Stabilisierungsfunktion wird periodisch aufgerufen und ruft die Funktion `node_updater` auf, diese erledigt die eigentliche Arbeit der Stabilisierungsfunktion:

- Neue Nodes in den Ring aufnehmen
- Nodes ohne Rückmeldung aus den Routingtabellen zu entfernen

Beim Erstellen eines Nodes wird gleichzeitig ein Stabilisierungprozess gestartet und mit dem Prozess des jeweiligen Nodes verlinkt. Tritt in einem der beiden Prozesse ein Fehler auf, werden beide Prozesse beendet.

Listing 2: Periodische Stabilisierung

```
node_updater(NodePid) ->
  receive
  after
    5000 ->
      updater( NodePid ),
      node_updater( NodePid )
  end.
```

Mit den Funktionen `node_id`, `node_sc` und `node_pr` kann man die Informationen über den Node, den Nachfolger-Node oder den Vorgänger des Nodes abrufen. Der Rückgabewert dieser Funktionen ist ein 3-Tupel `{Num, Id, Pid}`:

Num ist die Enfernung des Nodes vom aktuellem Node (interne Information)

Id der Hashwert, bestimmt die Position des Nodes in dem Chord-Ring

Pid die Pid des Nodes unter Erlang also die Addresse des Nodes

Die `updater` Methode überprüft zuerst den Nachfolger des Nodes, durch Anwendung der Funktion `node_sc( NodePid )`. Anschliessend wird der Vorgänger (`node_pr`) des Nodes mit `NextPid` überprüft. Sind die Routinginformationen noch konsistent, d.h. die Zeiger-Informationen sind aktuell, wird die Fingertable auf Basis des Nachfolgers gebaut. Ist jedoch ein neuer Node hinzugekommen, wird die Fingertable auf Basis des neuen Nodes aufgebaut. Greift die letzte Regel ist ein Fehler bei der Kommunikation aufgetreten und der Node `NextPid` wird aus der Fingertable entfernt und ein neuer Nachfolger ermittelt.

Listing 3: Stabilisierung

```
updater(NodePid) ->
  NodeId = node_id( NodePid ),
  {_, _NextId, NextPid} = node_sc( NodePid ),
  case node_pr( NextPid ) of
    { _, NodeId,  NodePid } -> ReqNode = NextPid;
    { _, _NewId,  NewPid } -> ReqNode = NewPid;
    {error, timeout }       -> ReqNode = NodePid
  end,
  try finger_table(NodeId, NodePid, ReqNode) of
    FingerTable ->
      NewFingerTable = [ {0, NodeId + 1, ReqNode} |
           FingerTable],
      NodePid ! #msg{id=set_finger, from=self(), data=
           NewFingerTable}
  catch
    error:badarith ->
      FingerTable = [ {I, idistance(NodeId, math:pow(2, I
           )), ReqNode } || I <- lists:seq(1, 3, 1)],
      NewFingerTable = [ {0, NodeId + 1, ReqNode} |
           FingerTable],
      NodePid ! #msg{id=set_finger, from=self(), data=
           NewFingerTable};
    _:_  ->
      NewFingerTable = [ {I, idistance(NodeId, math:pow
           (2, I)), NodePid } || I <- lists:seq(0, 3, 1)]
           ,
      NodePid ! #msg{id=set_finger, from=self(), data=
           NewFingerTable}
  end,
  SucNodePid = element(3, node_sc(NodePid)),
  SucNodePid ! #msg{id=set_pre, from=self(), data={NodeId
       , NodePid}},
  ok.
```

Die wichtigste Routine bei konsistenten Hashverfahren ist die Bestimmung des für einen Schlüsselbereich zuständigen Node.

Listing 4: Bestimmung des für Schlüssel zuständigen Node

```
findNode(HashKey, NodePid) ->
  NodeId = node_id( NodePid ),
  { _, PrevId, _PrevPid} = node_pr( NodePid ),
  Found = distance(HashKey, NodeId) =< distance(HashKey,
       PrevId),
  case Found of
    true  -> NodePid;
    false ->
      {_Pos, _NextId, NextPid } = node_fs(HashKey,
           NodePid),
      findNode(HashKey, NextPid)
  end.
```

4.2.3 Node Join

Um dem Chord-Ring beizutreten muss dem neuen Node bereits ein Teilnehmer des Ringes bekannt sein. Die Methode `join` in Listing 5 bekommt also die Adresse des neuen Nodes: `ChordNodePid`. Anschliessend wird der Hashwert `NewNodeId` des Nodes aus seiner Pid `NewNodePid` bestimmt und die Adresse `NextNodePid` des Nachfolgers im Ring gesucht.

Nachdem diese Informationen bekannt sind, kann die Fingertable des Knotens neu aufgebaut werden. Gemäß der in Abbildung 4 gezeigten Strategie wird die bekannte Pid des Nachfolgers angefragt und die für NewNodeId $+2^i$ verantwortlichen Knoten gesucht.

Mit `NewNodePid !#msg{id=set_finger, from=self(), data= NewFingerTable}` wird die Fingertable des beitretenden Nodes gesetzt. Abschliessend wird der Vorgängerzeiger auf den, mit `node_pr( NextNodePid )` gefundenen, Vorgängerknoten gesetzt (der Vorgänger des neuen Knotens ist, wie in der Theorie erwähnt, der Vorgänger des Nachfolgers) und der Vorgängerzeiger des Nachfolgers gesetzt.

Listing 5: Beitritt eines Nodes

```
join(NewNodePid, ChordNodePid) ->
```

```
  NewNodeId = node_id(NewNodePid),
  NextNodePid = findNode(NewNodeId, ChordNodePid),

  FingerTable = [ {X+1, idistance(NewNodeId, math:pow(2,X
       )), findNode(idistance(NewNodeId, math:pow(2,X)),
       NextNodePid)} || X <- lists:seq(0,3,1)],
  NewFingerTable = [{0, NewNodeId + 1, NextNodePid} |
       FingerTable],

  {_, PreId, PrePid} = node_pr( NextNodePid ),

  NewNodePid ! #msg{id=set_finger, from=self(), data=
       NewFingerTable},
  NewNodePid ! #msg{id=set_pre, from=self(), data={PreId,
        PrePid}},
  NextNodePid ! #msg{id=set_pre, from=self(), data={
       NewNodeId, NewNodePid}},
  ok.
```

4.3 Store

Im Chord-Ring werden *(Schlüssel,Wert)*-Paare gespeichert und der Befehl zum Speichern kann an die Adresse eines beliebigen, bekannten Node `NodePid` gestellt werden. Dieser bestimmt den für den Schlüssel `Key` verantwortlichen Node `StoreId` und dessen Nachfolger `SucceId`.

Anschliessend wird der Befehl zum Speichern des *(Schlüssel,Wert)*-Paar an die Adressen `StorePid`, `SuccePid` der beiden Nodes gesandt oder bei keiner Verbindung nach 2 Sekunden mit einem Fehler quittiert.

```
node_store(Key, Value, NodePid) ->
  { _, StoreId, StorePid } = find_node(Key, NodePid),
  { _, SucceId, SuccePid } = node_successor(StorePid),

  io:format("Store␣Key:~32.16.0b\t␣NodeId:~32.16.0b\n",[
       Key, StoreId]),

  StorePid ! #msg{id=store_key, from=self(), data={Key,
       Value}},
  receive
        #msg{id=store_key, from=StorePid, data=ok} -> ok
  after
    2000 -> error
  end,

  io:format("␣BackupId:~32.16.0b\n",[SucceId]),

  SuccePid ! #msg{id=store_key, from=self(), data={Key,
       Value}},
  receive
        #msg{id=store_key, from=SuccePid, data=ok} -> ok
  after
    2000 -> {error, timeout}
  end.
```

4.4 Beispiel

4.4.1 Erstellen der Nodes

Zuerst werden die Nodes *Ananas*, *Banana*, *Citronella* und *Dattel* erstellt:

```
Ananas = node_create("Ananas"),
Banana = node_create("Banana"),
Citronella = node_create("Citronella"),
Dattel = node_create("Dattel"),
```

Es werden im Beispiel die Namen der Nodes gehashed, damit für das Beispiel deutlich wird auf welche Nodes die Ressourcen verteilt werden.

```
Node <Ananas>
NodeID: 3e2
NodePID:<0.40.0>

Node <Banana>
NodeID:e6f
NodePID:<0.41.0>

Node <Citronella>
NodeID:8af
NodePID:<0.42.0>

Node <Dattel>
NodeID:eb5
NodePID:<0.43.0>
```

Die NodeID ist ein *MD5* Hash (128bit Länge) und stellt die Position im Chord-Ring dar. Für das Beispiel werden nur die ersten 3 Zeichen verwendet, da diese *für dieses Beispiel* eindeutig sind und es nachvollziehbarer machen.

4.4.2 Beitritt der Nodes

Jeder Node kann die Basis für einen Chord-Ring sein. Im folgenden möchte *Ananas* dem Chord-Ring beitreten, der bisher lediglich aus *Banana* besteht. Anschliessend treten weitere Nodes dem Ring bei und erweitern diesen.

```
join_node(Ananas, Banana),
join_node(Citronella, Banana),
join_node(Dattel, Citronella),
```

Bevor die Stabilisierungsroutinen zur Aktualisierung der Routinginformationen ausgeführt wurden stellt sich der Ring dar als:

```
JoinNode
NodeID:3e2 (Ananas)
SuccessorId:e6f (Banana)

JoinNode
NodeID:8af (Citronella)
SuccessorId:e6f (Banana)

JoinNode NodeID:eb5 (Dattel)
SuccessorId:e6f (Banana)
```

Nachdem die Stabilisierung abgeschlossen ist, stellt ist der Ring dar als:

```
NodeID:eb5 (Dattel)
Successor:3e2 (Ananas)

NodeID:3e2 (Ananas)
SuccessorID:8af (Citronella)

NodeID:8af (Citronella)
SuccessorId:e6f (Banana)

NodeID:e6f (Banana)
SuccessorId:eb5 (Dattel)

(Dattel -> Ananas -> Citronella -> Banana -> Dattel)
```

4.4.3 put

Nachdem der Ring in einen stabilen Zustand übergetreten ist, können (Schlüssel, Wert) Paare im Chord-Ring abgelegt werden. Der put Befehl `node_store` zum Speichern des Paares nimmt als Eingabe: Schlüssel, Wert und Node. Der Befehl zum speichern kann an einen beliebigen Node gesendet werden, welcher die Daten weiterverteilt.

```
Key1 = hash("lipsum"),
Key2 = hash("nes"),
node_store(Key1, "Lorem␣ipsum␣dolor␣sit␣amet", Ananas),
node_store(Key2, "NES␣controller␣to␣USB␣gamepad", Banana)
     ,
```

Chord ist für dynamische Systeme konzipiert, in dem ein Teilnehmer zu jeder Zeit dem Ring beitreten oder den Ring verlassen kann. Das bedeutet, daß Daten redundant im Ring verteilt sein müssen, im Beispielprogramm werden die Daten beim verantwortlichen Node und Nachfolger abgelegt.

```
Store Key:804
NodeId:8af (Citronella)
BackupId:e6f (Banana)

Store Key:1d7
NodeId:3e2 (Ananas)
BackupId:8af (Citronella)
```

4.4.4 get

Anfragen für einen Schlüssel können an einem beliebigen Node gestellt werden und die Daten werden im Ring mit der bereits beschriebenen Lookup Strategie gesucht.

```
print_lookup(Key1, Citronella),
print_lookup(Key2, Citronella),
```

Es antworten die Nodes, die sich für den Schlüssel verantwortlich zeigen:

```
Node <Citronella> NodePid:<0.42.0> KeyId:804
Value: Lorem ipsum dolor sit amet

Node <Ananas> NodePid:<0.40.0> KeyId:1d7
Value: NES controller to USB gamepad
```

4.4.5 Node Leave

Verlässt ein Node den Ring, im folgenden wird ein Erlang Prozess hart terminiert[17], zeigt sich der Backup Node verantwortlich für Anfragen. Ein Node tritt aus dem Ring aus:

```
erlang:exit(Ananas, brutal_kill)
```

und ein erneuter Lookup:

```
print_lookup(Key1, Dattel),
print_lookup(Key2, Dattel),
```

zeigt, daß sich nun *Citronella* für die Schlüssel verantwortlich zeigt:

```
Node <Citronella> NodePid:<0.42.0> KeyId:804
 Value: Lorem ipsum dolor sit amet

Node <Citronella> NodePid:<0.42.0> KeyId:1d7
 Value: NES controller to USB gamepad
```

5 Fazit

Es erfolgte eine Einführung in Distributed Hash Tables und das in [SMK+01] vorgestellte Protokoll *Chord* wurde hinsichtlich seiner Komplexität untersucht und in Erlang implementiert.

A Literatur

[Bre00] BREWER, Eric A.: *Towards Robust Distributed Systems*. www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, 2000. – Keynote Speech at ACM Symposium on the Principles of Distributed Computing

[FGC+97] FOX, Armando ; GRIBBLE, Steven D. ; CHAWATHE, Yatin ; BREWER, Eric A. ; GAUTHIER, Paul: Cluster-Based Scalable Network Services, 1997, S. 78–91

[GBL+03] GUPTA, Indranil ; BIRMAN, Ken ; LINGA, Prakash ; DEMERS, Al ; RENESSE, Robbert van: Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In: *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03*, 2003

[GL02] GILBERT, Seth ; LYNCH, Nancy: Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. In: *In ACM SIGACT News*, 2002, S. 2002

[HGL03] HOWARD, Sanjay G. ; GOBIOFF, Howard ; LEUNG, Shun tak: *The Google File System*. 2003

[HJK+07] HASTORUN, Deniz ; JAMPANI, Madan ; KAKULAPATI, Gunavardhan ; PILCHIN, Alex ; SIVASUBRAMANIAN, Swaminathan ; VOSSHALL, Peter ; VOGELS, Werner: Dynamo: amazon's highly available key-value store. In: *In Proc. SOSP*, 2007, S. 205–220

[KL07] KOHAVI, Ron ; LONGBOTHAM, Roger: Online Experiments: Lessons Learned. In: *IEEE Computer* 40 (2007)

[KLL+97] KARGER, David ; LEHMAN, Eric ; LEIGHTON, Tom ; LEVINE, Mathhew ; LEWIN, Daniel ; PANIGRAHY, Rina: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: *In ACM Symposium on Theory of Computing*, 1997, S. 654–663

[Knu98] KNUTH, Donald E.: *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. 2. Addison-Wesley Professional, 1998 http://www.worldcat.org/isbn/0201896850. – ISBN 0201896850

[MWH10] MARTINOVSKÝ, Filip ; WAGNER, Philipp ; HEINZE, Stefan: *Konzept und Implementierung eines MapReduce Framework in Erlang*. 2010. – FH Zittau-Goerlitz, Beleg

[SMK+01] STOICA, Ion ; MORRIS, Robert ; KARGER, David ; KAASHOEK, M. F. ; BALAKRISHNAN, Hari: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, 2001

[TH10] TROTTIER-HEBERT, Frederic: *Learn You Some Erlang: For Great Good*. http://www.learnyousomeerlang.com. Version: 2010. – Freies Onlinebuch. Lizenziert unter CreativeCommons 3.

[wil07] *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3. Cambridge University Press, 2007. – ISBN 0521880688

[17] Es tut auch nicht weh.