Zuverlässige Koordinierung in Cloud-Systemen

Florian Heisig
Friedrich-Alexander-Universität Erlangen-Nürnberg
florian.heisig@informatik.stud.uni-erlangen.de

ABSTRACT

Eine wichtige Gemeinsamkeit aller Cloud-Systeme ist der Gedanke verteilte Berechnungen auf günstiger Commodity-Hardware in großer Stückzahl durchzuführen. Außerdem soll die Cloud nach dem Infrastructure as a Service-Prinzip dem Nutzer immer genau die benötigte Menge an Ressourcen zur Verfügung stellen. Dies bedeutet für das Cloud-System, dass es durch Hardware-Ausfälle und die flexiblen Anpassungen ständig einer starken Dynamik unterliegt.

Damit die vielen parallelen Prozesse der Cloud korrekt zusammenarbeiten, müssen ihre Abläufe koordiniert werden. Dies muss auch unter den beschriebenen Rahmenbedingungen zuverlässig und fehlerfrei erfolgen.

Statt den Aufwand einer Eigenentwicklung zu betreiben, bietet es sich an, fertige, externe Komponenten für diese Aufgabe zu integrieren. Eine verbreitete Lösung ist *Apache ZooKeeper*, das eine zuverlässige Koordinierung ermöglichen soll. ZooKeeper steigert durch die Verwendung eines Server-Verbunds, welcher die replizierten Daten konsistent verwaltet, die Skalierbarkeit, Performanz und Fehlertoleranz des Dienstes. Zugriffe erfolgen atomar und das System garantiert die Aktualität und Persistenz der Daten auf allen Servern. Für den Benutzer präsentiert sich ZooKeeper mit einer einfachen Schnittstelle und einem Datenmodell, das einem Dateisystem nachempfunden ist. Neben Koordinierungsfunktionen wie Shared Locks und Barrieren ist ZooKeeper auch für Leader Election, die Verwaltung von Gruppen und als Namensdienst verwendbar. Zusätzlich zur Funktionsweise aus Benutzersicht und den grundlegenden technischen Konzepten soll die Anwendbarkeit von ZooKeeper in der Cloud aufgezeigt werden.

1. EINFÜHRUNG

Das Cloud Computing ist ein aktuelles Thema und wird zur Zeit sehr kontrovers diskutiert. Strittig ist bereits wie es überhaupt zu definieren ist. Bisher existiert keine einheitliche Definition, doch zur gemeinsamen Basis aller Definitionsversuche gehört – neben Virtualisierung, verteiltem Rechnen und dem Utility-Grid – der Strukturwandel innerhalb der Rechenzentren [13].

Dahinter steht die bewusste Abkehr vom Gedanken ein Rechenzentrum aus wenigen, teuren Hochleistungsrechnern aufzubauen. Mit dem Cloud Computing schlägt man die genau entgegengesetzte Richtung ein und vernetzt in großer Stückzahl billige Commodity-Hardware. Dabei nimmt man Ausfälle der Hardware bewusst in Kauf und muss die notwendige Zuverlässigkeit durch fehlertolerante Software herstellen, welche Ausfälle erkennt und die Netzwerkknoten geeignet ersetzt.

Koordinierung bezeichnet in diesem Zusammenhang eine Abstimmung der Aktivitäten verschiedener Knoten aufeinander. Dies kann die Einigung auf eine gemeinsame Sicht der Realität sein, wie zum Beispiel bei der Verwaltung von Gruppen, bei der eine gemeinsame Sicht auf die Mitglieder existieren muss. Häufig wird auch ein Anführer benötigt um Eindeutigkeit – beispielsweise bei der Vergabe von IDs – herzustellen. Produzieren mehrere Knoten Daten, welche anschließend von einem anderen (Konsument) weiter verarbeitet werden sollen, so muss der Konsument über neu verfügbare Daten benachrichtigt werden. Nutzen mehrere Knoten eine gemeinsame Ressource parallel, so müssen sie sich darauf einigen, wer die Ressource zuerst nutzen darf.

Koordinierung kommt hierbei zum Einsatz, um den Zugriff auf gemeinsame Ressourcen zu synchronisieren. Häufig definiert man dazu einen kritischen Abschnitt, dessen Betreten an bestimmte Bedingungen geknüpft ist. Der kritische Abschnitt wird durch ein Lock geschützt, welches beispielsweise garantiert, dass nur ein Knoten gleichzeitig Zutritt erhält.

In rein lokalen Anwendungen lässt sich dieses nicht-triviale Problem noch relativ effizient lösen, weil alle beteiligten Prozesse unmittelbar auf das lokale Lock zugreifen können. Bei verteilten Anwendungen muss hingegen auch die Koordinierung verteilt gelöst werden. Für das Locking im verteilten Fall gibt es verschiedene Algorithmen, doch alle erfordern eine Kommunikation über das Netzwerk. Im Allgemeinen ist jedes Betreten und Verlassen des kritischen Abschnitts mit mindestens einem Nachrichtenaustausch verbunden. Dadurch verlangsamt sich die Koordinierung und das Datenvolumen erhöht sich.

In Cloud-Systemen verschlimmert sich die Lage durch die große Anzahl paralleler Prozesse zusätzlich. Um zu verhindern, dass in diesem Umfeld Koordinierung zum Flaschenhals wird, müssen Algorithmen zum Einsatz kommen, die extrem effizient arbeiten und sparsam im Nachrichtenaustausch sind.

Vor dem Hintergrund der enormen Menge von Commodity-Hardware in Cloud-Systemen müssen auch andere Maßstäbe an die Zuverlässigkeit der Koordinierung angelegt werden. Die Cloud unterliegt durch Ausfälle und externe Anpassungen einer starken Dynamik. Ein Dienst zur Koordinierung der Knoten muss daher nicht nur gut skalieren, sondern auch mit der Dynamik der Cloud schritthalten können. Im Bereich der Fehlertoleranz ist es wichtig, dass der Dienst trotz Ausfällen zuverlässig arbeitet und das System in einem konsistenten Zustand hält.

Geläufiges Vorgehen in vielen Projekten ist es, die benötigten Funktionen zur Koordinierung selbst zu entwickeln und direkt in die konkrete Anwendung einzubetten. Dies hat den Vorteil, dass nur die benötigten Funktionen integriert werden und der Code schlank bleibt.

Dieses Vorgehen hat aber auch eine Menge von Nachteilen. Das hat damit zu tun, wie in der Praxis viele Projekte ablaufen. Zeit ist meist nur begrenzt vorhanden und die Entwicklung eines Koordinierungsdienstes stellt eine Aufgabe dar, die sich im Allgemeinen nicht mit dem Kernproblem der zu entwickelnden Anwendung beschäftigt. Als Folge davon werden nur wenig Ressourcen in die Entwicklung der Koordinierung gesteckt. Deshalb entstehen hier häufig keine optimalen Lösungen bezüglich Performanz, Robustheit und Fehlertoleranz [16]. Einige Lösungen sind sogar schlichtweg fehlerhaft, denn Koordinierungsdienste sind nicht trivial und anfällig für Verklemmungen und Wettlaufsituationen.

Hierbei muss man bedenken, dass die Koordinierung oft sehr grundlegend für die Anwendung ist. Ein Fehler in der Koordinierung kann zum Ausfall der kompletten, darauf aufbauenden, Anwendung führen. Daher liegt es nahe auf eine ausgereifte Komponente zurückzugreifen, welche alle benötigten Funktionen bietet, anstatt immer wieder in Eigenentwicklungen zu investieren. Als existierende Lösungen ist hier insbesondere *ZooKeeper* [15, 16] zu nennen, das sich als Standard-Lösung in diesem Bereich etablieren will. Im Folgenden soll nun der Aufbau und die Verwendung von ZooKeeper beleuchtet werden. Der erste Abschnitt beschäftigt sich mit der grundlegenden Architektur von ZooKeeper und seiner Funktionsweise. Neben der API werden wichtige Konzepte aus Benutzersicht genauer erläutert. Abschnitt 3 stellt anschließend die Verwendung von ZooKeeper für die Koordinierung im Cloud-Computing-Umfeld vor.

2. ZOOKEEPER

Apache ZooKeeper ist ein Unterprojekt von Hadoop [3], einem Open-Source-Framework für verteilte Anwendungen. Gestartet wurde das Projekt ZooKeeper bei Yahoo Research [20] und wird immer noch hauptsächlich von diesen Entwicklern getragen [8]. ZooKeeper ist einen generischer Koordinierungsdienst, der alle gängigen Koordinierungsprobleme in verteilten Systemen behandeln soll. Auf diese Weise soll er sämtlichen Anwendungen zur Verfügung stehen und ihnen die Entwicklung einer Eigenlösung ersparen. Zusätzlich zur Koordinierung kann ZooKeeper beliebige Metadaten in einer Datenbank verwalten. Seine vielfältigen Einsatzbereiche umfassen daher neben der eigentlichen Koordinierung unter anderem auch noch verwandte Dienste wie Konfiguration, Leader Election, die Verwaltung von Gruppen und den Einsatz als Namensdienst.

ZooKeeper wurde mit dem Ziel entwickelt einen hoch verfügbaren, skalierbaren Dienst anzubieten, der einfach zu handhaben ist. Er soll im Gegensatz zu unsauberen Eigenentwicklungen robust sein und eine gute Performanz bieten. Generell wurde ZooKeeper auf Workloads ausgerichtet, in denen das Lesen – gegenüber dem Schreiben – die dominante Operation ist [15]. Dies ist dadurch zu erklären, dass Daten meistens von einem Knoten geschrieben und anschließend von vielen anderen gelesen werden.

Im Kontrast zu einem verteilten Dateisystem ist ZooKeeper nicht primär auf Datenhaltung ausgelegt. Die in ZooKeeper enthaltene Datenbank dient in erster Linie zur Aufbewahrung von Metainformationen von geringer Größe (wenige Kilobytes). Die Daten werden beispielsweise für die Koordinierung verwendet oder enthalten Konfigurationen. Um größere Datenmengen mit ZooKeeper zu verwalten, bietet es sich an, die eigentlichen Daten in einem verteilten Dateisystem zu speichern und deren Metainformationen, wie den Speicherort, in ZooKeeper abzulegen.

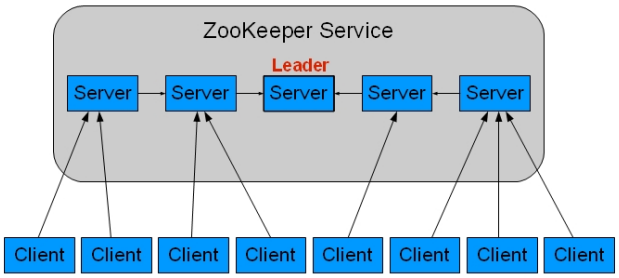

Abbildung 1: Architektur von ZooKeeper

2.1 Architektur

ZooKeeper [7] arbeitet nach dem Client-Server-Prinzip und ist daher in Server- und Client-Seite aufteilbar. Auf der Server-Seite besteht er aus dem eigentlichen Koordinierungsdienst, welcher die Hauptarbeit leistet. Dieser Teil läuft in Java und beantwortet die Anfragen der Clients. Auf der Client-Seite stehen Bindings für Java, C, Perl, Python und REST zur Verfügung, über die die Clients Verbindung zum Koordinierungsdienst aufbauen.

Die Aufgabe von ZooKeeper ist die Verwaltung von Metainformationen und kleinen Datenmengen. Diese Daten liegen auf der Server-Seite und können dort aufgrund der geringen Größe im Arbeitsspeicher gehalten werden. Durch die – im Vergleich zur Festplatte – kurzen Zugriffszeiten auf den Arbeitsspeicher kann damit eine geringere Latenz und ein höherer Durchsatz erzielt werden.

Der ZooKeeper-Dienst (siehe Abbildung 1) wird von mehreren Rechnern bereitgestellt, auf denen jeweils ein Replikat von ZooKeeper läuft. Dieses sogenannte *Ensemble* dient der Fehlertoleranz und erhöht die Verfügbarkeit des Dienstes. Jeder Rechner verfügt über ein vollständiges Abbild der Datenstruktur mit den verwalteten Informationen. Zusätzlich wird persistent ein Transaktionsprotokoll geführt, und es werden regelmäßig Zustandssicherungen erzeugt. Im Transaktionsprotokoll werden alle Änderungen vermerkt, bevor sie in der Datenbank von ZooKeeper erscheinen. Dies stellt sicher, dass keine Daten verloren gehen und ein Server wieder anlaufen kann.

Alle Server-Prozesse kennen einander und halten ihre Datenbasis über einen Konsensalgorithmus untereinander konsistent. Dieser Konsens funktioniert nur solange die Mehrheit aller Server-Prozesse verfügbar ist. Fallen mehr als die Hälfte aller Server aus, kann ZooKeeper dies nicht mehr tolerieren und stellt seinen Dienst ein.

Unter den Servern des Ensembles gibt es immer genau einen Anführer (Leader), der beim Start gewählt wird. Der Anführer garantiert die eindeutige Ordnung aller Transaktionen, indem er fortlaufende Sequenznummern für jede Änderung vergibt. Aus diesem Grund laufen alle Schreiboperationen über den Leader. Leseoperationen können von jedem beliebigen Server beantwortet werden, da alle über ein Replikat der Datenbasis verfügen. Bei Ausfall des Anführers kann

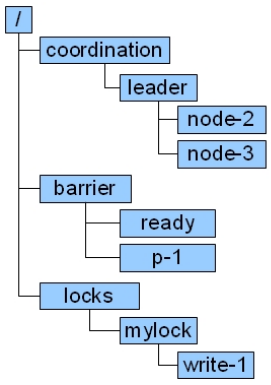

Abbildung 2: Beispiel für das hierarchisches Datenmodell von ZooKeeper

unter den verbliebenen Rechnern ein neuer Leader gewählt werden.

Bei geschickter Zuordnung der Server zu den Clients trägt die replizierte Struktur des ZooKeeper-Dienstes zur Lastverteilung und Skalierbarkeit bei. Eine einfache Möglichkeit besteht darin jedem Server gleich viele Clients zuzuordnen. Komplexere Strategien berücksichtigen bei der Zuordnung auch das durchschnittliche Anfragevolumen der Clients.

Ohne Lastverteilungsstrategie können sich Clients mit einem beliebigen Server des Ensembles verbinden. Sollte die Verbindung einmal abbrechen, kann sich der Client mit einem anderen Server verbinden, wobei die Sitzung des Clients erhalten bleibt. Über die TCP-Verbindung zum Server kann der Client nun Anfragen senden und Antworten oder Benachrichtigungen empfangen. Leseoperationen werden immer direkt vom verbundenen Server beantwortet. Schreiboperationen hingegen werden zuerst an den Leader weitergeleitet. Der Leader sorgt anschließend dafür, dass die Änderung auf den Servern des Ensembles konsistent vollzogen werden. Sobald die Mehrheit des Ensembles die Änderung persistent gemacht hat, erhält der Client über den Server, mit dem er verbunden ist, eine Antwort des Leaders.

2.2 Datenmodell

Die von ZooKeeper verwalteten Daten werden in eine hierarchische Datenstruktur (siehe Beispiel in 2) eingeordnet, die der eines Standarddateisystems ähnelt. Zu jedem Knoten der Struktur existiert ein absoluter Pfad, der aus mehreren durch Slash („/") getrennten Komponenten besteht.

Zu einem Dateisystem gibt es allerdings einige wichtige Unterschiede: Erstens gibt es keine Unterscheidung zwischen Verzeichnis und Datei. Alle Knoten können sowohl Daten speichern als auch Unterknoten enthalten. Zweitens werden Daten immer komplett gelesen oder geschrieben. Auf jede Leseanfrage werden immer alle im Knoten gespeicherten Daten am Stück zurückgeliefert und beim Schreiben werden die Daten eines Knotens immer komplett ersetzt. Es gibt somit weder partielle Lese- und Schreiboperationen, noch strombasierte Zugriffe. Dazu besteht im Allgemeinen auch keine Veranlassung, da ZooKeeper nur kleine Datenmengen verwaltet. Das Lesen und Schreiben kann auf diese Weise als atomarer Vorgang gestaltet werden. Zusammen mit der strengen Ordnung aller Transaktionen ist es durch diese Einschränkung leichter, Konsistenz zu garantieren.

Die Knoten sind darauf ausgelegt wenige Kilobytes an Daten zu speichern. Außerdem werden für jeden Knoten noch einige Metainformationen abgelegt. Es gibt Versionsnummern für alle Datenänderungen (siehe Abschnitt 2.5.3), sowie Zeitstempel für das Caching beim Client. Zusätzlich verfügt jeder Knoten über eine Zugriffskontrollliste (ACL) mit der der Zugriff für bestimmte Nutzer eingeschränkt werden kann.

2.3 Garantien

Um auch in komplexen Fällen eine präzise Synchronisation zu ermöglichen, garantiert ZooKeeper die Einhaltung gewisser Eigenschaften. Die erste wichtige Eigenschaft ist die *sequentielle Konsistenz*, die besagt, dass alle Anfragen eines Clients sequentiell ausgeführt werden. Die Schreibzugriffe unterliegen durch Verwendung des Leaders sogar einer global definierten Ordnung. Dadurch werden alle Schreiboperationen eines Clients in der Reihenfolge des Sendens ausgeführt.

Zweitens garantiert ZooKeeper mit dem *Single System Image* die Konsistenz der Replikate. Ein Client erhält, egal mit welchem Server er verbunden ist, die gleiche Sicht auf den Dienst. Es wird jedoch nicht garantiert, dass zwei verschieden Clients zu jeder Zeit auch die gleiche Sicht auf die Daten haben.

Lese- und Schreibzugriffe sind in ZooKeeper atomar. Daher ist eine Operation entweder erfolgreich oder schlägt komplett fehl. Im Falle eines Fehlers bleiben die Daten aufgrund der *Atomarität* unverändert und es gibt keine Teilergebnisse.

Zu ZooKeepers Garantien gehört auch die *Verlässlichkeit*. Wenn Daten erfolgreich verändert wurden, bleibt diese Änderung im System persistent erhalten, solange die Daten nicht erneut verändert werden.

Des Weiteren garantiert ZooKeeper innerhalb gewisser Zeitschranken die *Aktualität* der Daten, sowie die zeitnahe Benachrichtigung über Änderungen bei der Verwendung von Watches (siehe Abschnitt 2.5.2).

2.4 Schnittstelle

Anwendungen müssen zur Verwendung von ZooKeeper eine Client-Bibliothek einbinden und über diese eine Verbindung zum Server herstellen. Die Library wickelt die gesamte Kommunikation mit dem ZooKeeper-Dienst ab. Sie verschickt die Anfragen und nimmt Benachrichtigungen vom Server entgegen. Die ZooKeeper-API besteht aus wenigen einfachen Funktionen:

- `String create(path, data, acl, flags)`
  Erzeugt einen Knoten in der ZooKeeper-Datenbank unter dem angegebenen Pfad. Über den Methodenparameter `flags` kann der Knoten als Ephemeral Node (siehe Abschnitt 2.5.1) oder Sequence Node (siehe Abschnitt 2.5.4) definiert werden. Rückgabewert ist der verwendete Dateiname.

- `void delete(path, expectedVersion)`
  Löscht einen Knoten, wenn die Versionsnummer passt.

- `Stat setData(path, data, expectedVersion)`
  Ersetzt die Daten eines Knotens bei passender Version und gibt Statistikdaten (Versionsnummern, Zeitstempel, etc.) des Knotens zurück.

- `(data, Stat) getData(path, watch)`
  Liest Daten (`data`) sowie Statistikinformationen (`Stat`) von einem Knoten. Mit dem optionalen Watch (siehe Abschnitt 2.5.2) kann der Knoten auf Änderungen überwacht werden.

- `Stat exists(path, watch)`
  Prüft die Existenz eines Pfads. Optional kann darauf ein Watch gesetzt werden.

- `String[] getChildren(path, watch)`
  Abfrage der Unterknoten, die als Liste zurückgegeben werden. Ein Watch ist optional.

- `void sync(path)`
  Blockiert bis alle Daten propagiert wurden und der Server synchron mit dem Leader ist.

Aufbauend auf diesen Primitiven können von den Anwendungen komplexere Dienste implementiert werden.

2.5 Wichtige Konzepte

ZooKeeper bietet dem Benutzer einige besondere Konzepte, die es ihm ermöglichen, auch komplizierte Koordinierungsfunktionen effizient und einfach umzusetzen. Die wesentlichen Konzepte werden in diesem Abschnitt kurz vorgestellt. Im Abschnitt 3 wird darauf eingegangen, wie sich hiermit in der Praxis auftretende Koordierungsprobleme lösen lassen.

2.5.1 Vergängliche Knoten (Ephemeral Nodes)

Ephemeral Nodes sind eine spezielle Ausprägung der Knoten im Datenmodell von ZooKeeper. Sie existieren per Definition nur so lange wie die Sitzung, die sie erzeugt hat. Das bedeutet, dass sie mit ihrem Erzeuger „sterben".

Diese speziellen Knoten können jederzeit von Nutzern erzeugt werden und verhalten sich während ihrer „Lebenszeit" wie alle anderen Knoten. Sobald jedoch die Sitzung, in der sie erzeugt wurden, endet, werden die Ephemeral Nodes automatisch gelöscht.

Sie eignen sich beispielsweise zur Überwachung des Status von Teilnehmern einer Gruppe. Beim Beitritt wird von jedem Gruppenteilnehmer in einem gemeinsamen Pfad ein Ephemeral Node erzeugt. Dieser Pfad gibt dann Aufschluss über alle Teilnehmer der Gruppe. Fällt nun ein Teilnehmer aus oder beendet die Verbindung, so wird automatisch sein Knoten gelöscht und die Sicht auf die Gruppe aktualisiert. Für eine umfangreichere Verwaltung der Gruppe ergänzt man die Knoten noch um zusätzliche Informationen.

Das Konzept der vergänglichen Knoten erspart es dem Nutzer die Existenz der anderen Clients periodisch durch eigenes Nachfragen überprüfen zu müssen. Außerdem entfallen Aufräumarbeiten bei den Daten, die ein Knoten hinterlegt hat.

2.5.2 Wächter (Watches)

Watches sind eine Umsetzung des Observer-Entwurfsmusters in ZooKeeper. Ein Client kann für jeden beliebigen Knoten ein Watch setzen und ihn damit überwachen. Erfährt ein überwachter Knoten in der Folge eine Änderung, so wird dem Client eine Nachricht zugestellt. Watches werden immer nur einmalig ausgelöst und müssen bei Bedarf neu registriert werden. Es können auch nicht-existente Knoten überwacht werden. Der Client wird dann bei Erzeugung des jeweiligen Knotens benachrichtigt. Durch das Zusammenspiel von Watches und Ephemeral Nodes lassen sich diverse Koordinierungsfunktionen effizient umsetzen, da auf aktives Warten verzichtet werden kann.

2.5.3 Bedingte Änderungen (Conditional Updates)

Um Änderungen an Knoten nachvollziehbar zu machen, verfügt jeder Knoten in ZooKeeper über Versionsnummern, die bei einigen API-Funktionen zurückgegeben werden. Über die Versionsnummern kann ein Client feststellen, ob die Daten zwischenzeitlich von parallelen Clients geändert wurden.

Da ZooKeeper Daten immer komplett ausliest und das Konzept Dateien zu öffnen nicht unterstützt, sind Änderungen an den Daten immer mit einem gewissen Risiko verbunden, wie folgendes einfaches Beispiel zeigt: Ein Client möchte einen Zähler inkrementieren. In ZooKeeper sind dies drei getrennte Schritte, nämlich den Wert aus dem Knoten lesen, ihn inkrementieren und anschließend das Ergebnis zurückschreiben. Daher kann es passieren, dass ein anderer Client während dieses Vorgangs den Zähler inkrementiert. Die Folge ist ein falscher Zählerstand.

ZooKeeper umgeht dieses Problem durch bedingte Änderungen. Die Funktionen `setData` und `delete` der API (siehe Abschnitt 2.4) benötigen als Parameter für das Schreiben die erwartete Version des Knotens (`expectedVersion`). Stimmt die erwartete Version nicht mit der vorhandenen überein, schlägt die Operation fehl. Der Client kann sich nun die neuere Version besorgen.

2.5.4 Generierte Knotennamen (Sequence Nodes)

ZooKeeper ermöglicht es, generierte Knotennamen zu verwenden. Dazu wird an den eigentlichen Namen automatisch eine Identifikationsnummer nach dem Muster „name-x" angehängt, wobei x ein monoton steigender Zähler ist. Auf diese Weise können – relativ zu einem Elternknoten – eindeutige Namen erzeugt werden.

3. VERWENDUNG VON ZOOKEEPER

ZooKeeper bietet in den Grundfunktionen schon alle notwendigen Mittel für die Verwendung als Namensdienst oder Konfigurationsdatenbank. Namen und Konfigurationen sind schlichte Metadaten, die einfach in der Datenbank von ZooKeeper abgelegt werden können. Hierbei kommt zum Tragen, dass ZooKeeper Daten ähnlich einem Dateisystem ablegt.

Komplexere Funktionen lassen sich mittels der Konzepte aus Abschnitt 2.5 effizient implementieren. Für einige klassische Koordinierungsfunktionen existieren bereits sogenannte *Rezepte* [9], die als Konventionen dienen können und zeigen, wie die jeweiligen Funktionen am Besten umgesetzt werden. Insbesondere sollen damit Probleme wie aktives Warten (Polling) und Herden-Effekte[1] vermieden werden, um die Last gering zu halten und die Skalierbarkeit zu erhöhen. Die unnötigen Anfragen durch das Polling oder Herden-Effekte erhöhen die Nachrichtenkomplexität des Algorithmus und somit Latenz und Netzwerklast.

[1]Herden-Effekte bezeichnen hier das gleichzeitige Aufwachen vieler Prozess durch ein Ereignis, wodurch hohe Lastspitzen entstehen.

Einige der Rezepte werden in den folgenden Abschnitten vorgestellt und sollen ein Gefühl dafür vermitteln, wie Koordinierung mit ZooKeeper umgesetzt wird. Zusätzlich wird anhand von prominenten Systemen aus dem Bereich Cloud Computing die praktische Anwendbarkeit belegt.

ZooKeeper wird häufig für die Verwaltung einer großen Anzahl von Servern, wie es im Cloud-Computing-Umfeld der Regelfall ist, eingesetzt. Dabei werden Informationen über die zum Cluster gehörigen Server gespeichert und ihre Anwesenheit überwacht. Die Fähigkeiten für diese Verwaltung von Gruppen bringt ZooKeeper „out of the box" mit. Insbesondere kommt hier das Konzept der Ephemeral Nodes zum Einsatz. Auf der Gruppenverwaltung aufbauend kann dann beispielsweise Arbeit an die Rechner im Cluster delegiert werden. Dazu wird die Gruppenverwaltung gemeinsam mit Leader Election (siehe Abschnitt 3.1) eingesetzt, um einen Anführer für den Cluster zu wählen. Der Anführer verteilt dann die Arbeit und übernimmt weitere zentrale Aufgaben für den Cluster. Neben diesen Grundfunktionen kommen auch diverse Synchronisierungs- und Koordinierungs-Primitiven, wie Shared Locks (siehe Abschnitt 3.2) und Barrieren (siehe Abschnitt 3.3) zur Anwendung.

3.1 Leader Election

Eine wiederkehrende Aufgabe in Cloud-Computing-Systemen ist die Wahl eines Anführers aus einer Menge von Knoten. Dies kommt immer dann vor, wenn z. B. ein zentraler Organisator oder ein ID-Generator benötigt wird. Mit Hilfe eines Leaders können eindeutige Entscheidungen getroffen werden. Er dient somit der Ausübung zentraler Kontrolle.

3.1.1 Rezept

Mit dem folgenden Rezept wird die Wahl eines Anführers und die Neuwahl bei Ausfall gezeigt. Benötigt wird dafür ein gemeinsames, allen Clients bekanntes Verzeichnis in der ZooKeeper-Datenbank (z. B. `/coordination/leader`). Unter diesem Pfad legen alle Clients einen Knoten an, der sie bei der Wahl repräsentiert. Folgenden Ablauf muss jeder Client ausführen:

1. Erzeuge einen neuen Knoten unter dem Pfad `/coordination/leader` mit dem Namen „node-". Setze dabei die Ephemeral und Sequence Flags. Damit existiert für den Client ein vergänglicher Knoten mit eindeutigem Namen (durch das Anhängen der Sequenznummer).

2. Frage nun mit `getChildren()` die anderen Unterknoten von `/coordination/leader` ab und suche den Unterknoten mit der nächstkleineren Sequenznummer.

3. Wenn der eigene Knoten die kleinste Sequenznummer hat, ist er Leader. Ansonsten setze ein Watch auf den Knoten mit der nächstkleineren Sequenznummer.

Durch die Verwendung eines Watch erfolgt bei Ausfall des Leaders eine automatische Benachrichtigung. Insbesondere ist kein Polling notwendig. Da alle erzeugten Knoten Ephemeral Nodes sind, werden sie bei Ausfall automatisch gelöscht. Dies löst bei maximal einem anderen Client ein Watch aus, der darauf reagieren kann. Durch das Beobachten der nächst kleineren Sequenznummer wird pro Ausfall immer nur ein Watch ausgelöst, und es tritt kein Herden-Effekt auf, bei dem alle Prozesse erwachen und einen neuen Leader suchen.

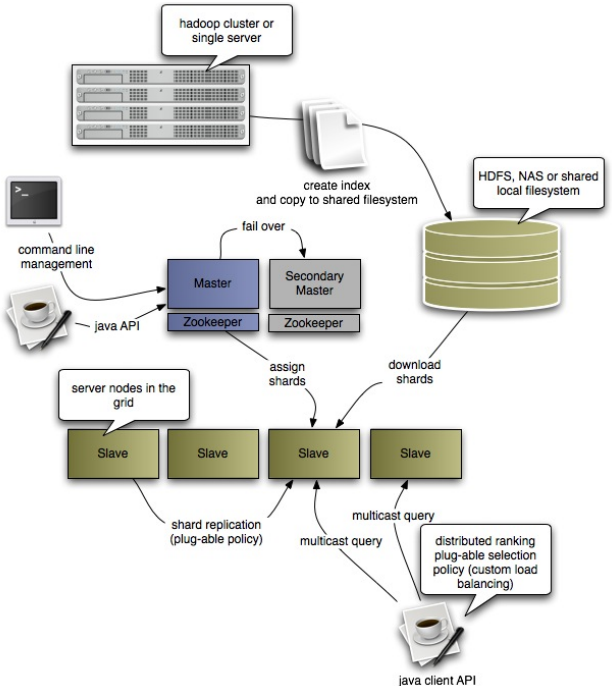

Abbildung 3: Anwendungsfall von ZooKeeper in Katta: Die Verwaltung einer Gruppe von Slave-Rechnern (aus [1])

Wird das Watch ausgelöst, führt der Client die Schritte zwei und drei erneut durch. Er wird entweder der neue Anführer oder setzt wieder ein Watch auf seinen neuen Vorgänger. Auf diese Weise wird ein ausgefallener Leader immer sofort ersetzt.

3.1.2 Anwendungsfälle in der Cloud

Katta [2] ist eine Lösung zur verteilten Speicherung sehr großer Datenbank-Indizes, wie den Schlüssel-Wert-Paaren von Hadoop MapReduce [12] und den Indizes der Volltextsuchmaschine Lucene [4]. Zur Speicherung werden die Indizes in *Shards* unterteilt, die repliziert auf mehreren Servern (Slaves) abgelegt werden. Sharding bezeichnet eine Technik zur horizontalen Partitionierung von Datenbanken – also die Verteilung von verschiedenen Datenbankzeilen auf verschiedene Tabellen (Beispiel: eine Datenbank mit Kunden wird nach Postleitzahlen aufgeteilt). Durch die Aufteilung der riesigen Datenmenge findet diese stückweise auf der Commodity-Hardware der Cloud Platz. Des Weiteren trägt die Verteilung zur Fehlertoleranz und Skalierbarkeit bei. Um die Verteilung der Shards und Replikate auf die Slaves vorzunehmen und ihre Position zu verwalten, wird ein zentraler Master benötigt, welcher als Koordinator fungiert. Die Abbildung 3 zeigt, wie ZooKeeper innerhalb von Katta eingesetzt wird, um den Master zu wählen. Fällt der Master aus, wird er mittels ZooKeeper ersetzt. Außerdem verwendet auch der Master selbst ZooKeeper für die Verwaltung der Slaves und die Verwaltung der Index-Shards. So kann er beispielsweise Informationen über die aktiven Slaves und die zugeteilten Shards in ZooKeeper speichern.

Genauso wie Katta verwendet Apache Solr [5,10], eine auf Lucene basierende Volltextsuchmaschine für Unternehmen,

ZooKeeper für die Wahl eines Masters zur Verwaltung von Shards und Replikaten. Bei Apache Hadoop [3] kommt ein mittels ZooKeeper gewählter Master-Prozess zum Einsatz, um MapReduce-Jobs auszuführen. Der Master-Prozess vergibt dabei Arbeit an die Worker-Prozesse und überwacht die Ausführung. Auch Apache Cassandra [6, 17], ein von Facebook entwickeltes, verteiltes Open-Source-Datenbanksystem zur Verwaltung großer Datenmengen, nutzt ZooKeeper zur Leader Election. Bei Cassandra ist der Leader dafür verantwortlich, den Knoten im Cluster Wertebereiche und Replikate zuzuweisen. Die Metadaten darüber werden ebenfalls in ZooKeeper abgelegt.

3.2 Shared Locks

Locks stellen Berechtigungen zur Nutzung einer Ressource dar. Sie beschränken die Zahl der Clients, die gleichzeitig Zugriff haben. Während bei einem klassischen Lock (gegenseitiger Ausschluss) immer maximal ein Client das Lock halten kann, sind bei einem Shared Lock mehrere Clients erlaubt, solange sie nur lesend auf die Ressource zugreifen. Leseoperationen werden als unkritisch angesehen und können parallel ausgeführt werden. Soll hingegen schreibend zugegriffen werden, müssen auch hier alle anderen Clients warten.

3.2.1 Rezept

Zur Verwendung von Locks muss zuerst ein Lock-Verzeichnis definiert werden (z. B. `/locks/mylock`). Das Lock-Verzeichnis nimmt anschließend die gesamten Informationen über das Lock und die Clients, die es verwenden wollen, auf. Um lesenden Zugriff zu erhalten befolgt der Client diesen Ablauf:

1. Erzeuge mit `create()` einen Unterknoten im Verzeichnis `/locks/mylock` mit dem Namen „read-" sowie Ephemeral und Sequence Flag. Die zugeteilte Sequenznummer dient dem späteren Vergleich und ist mit dem Ziehen einer Nummer in einer Warteschlange vergleichbar.
2. Frage mit `getChildren()` alle vorhandenen Unterknoten in `/locks/mylock` ab.
   (a) Wenn kein Unterknoten mit kleinerer Sequenznummer und „write-" im Namen existiert, ist das Lock erteilt und der Ablauf beendet.
   (b) Andernfalls überwache den „write-"-Knoten mit der nächstkleineren Sequenznummer mit Hilfe eines Watches. Rufe dazu `exists()` an dem Knoten auf und setze das Watch-Flag.
      i. Wenn `exists()` *false* zurückgibt, springe zurück zu Schritt 2.
      ii. Ansonsten warte auf die Auslösung des Watch und springe anschließend zu Schritt 2.

Für schreibenden Zugriff gilt dieser Ablauf:

1. Erzeuge mit `create()` einen Unterknoten im Verzeichnis `/locks/mylock` mit dem Namen „write-" sowie Ephemeral und Sequence Flag.
2. Frage mit `getChildren()` alle vorhandenen Unterknoten in `/locks/mylock` ab.
   (a) Wenn kein Unterknoten mit kleinerer Sequenznummer existiert, ist das Lock erteilt und es kann abgebrochen werden.
   (b) Ansonsten überwache den Knoten mit der nächstkleineren Sequenznummer über ein Watch. Rufe dazu `exists()` an dem Knoten auf und setzt das Watch-Flag.
      i. Wenn `exists()` *false* zurückgibt, springe zurück zu Schritt 2.
      ii. Sonst warte auf die Auslösung des Watch und springe anschließend zu Schritt 2.

Man kann erkennen, dass für Leseoperationen alle vorhergehenden Schreiboperationen abgewartet werden müssen. Für Schreiboperationen müssen sowohl Lese- als auch Schreiboperationen abgewartet werden. Die Freigabe eines Locks erfolgt in beiden Fällen durch Löschen des in Schritt 1 angelegten Knotens.

Durch kleine Änderungen lassen sich aus Shared Locks auch Revocable Shared Locks machen. Um den Halter eines Locks zu bitten, dieses abzugeben, können Clients in den Knoten des Halters die Nachricht „unlock" schreiben. Der Halter kann nun, gegebenenfalls nach Ausführung einer Abbruchprozedur, das Lock freigeben.

3.2.2 Anwendungsfälle in der Cloud

Locks sind sehr grundlegend für die Synchronisation in massiv-parallelen Cloud-Systemen. Immer wieder greifen mehrere Prozesse auf gemeinsame Daten/Ressourcen zu. In solch einem kritischen Abschnitt darf immer nur ein Prozess den exklusiven Zugriff erhalten. Die dafür notwendige Koordinierung ist insbesondere in Cloud-Systemen nicht trivial.

Bei Yahoo wird ZooKeeper benutzt, um das Locking in diversen Diensten zu ermöglichen. Die Entwickler in den Projekten können ZooKeeper einfach als Locking-Dienst einbinden, was sie von der Arbeit befreit, eine eigene Lösung zu implementieren. Daneben wird ZooKeeper bei Yahoo u. a. auch für die Verwaltung von Gruppen und Leader Election eingesetzt [10].

3.3 Doppelte Barrieren

Eine Barriere ist eine Methode zur Synchronisierung einer Gruppe von parallel arbeitenden Prozessen/Threads. An einer Barriere müssen alle Prozesse so lange warten, bis auch alle anderen Prozesse die Barriere erreicht haben. Eine Barriere ist also ein Punkt, an dem sich alle Prozesse in ihrer Abarbeitung synchronisieren und dann gemeinsam in den weiteren Ablauf starten.

Eingesetzt werden Barrieren vornehmlich bei iterativen Verfahren, in denen die nächste Runde auf den Ergebnissen der vorherigen aufbaut. Dazu müssen alle Berechnungen der vorherigen Runde beendet sein. Ähnlich ist es bei komplexen Abläufen, die aus mehreren Phasen bestehen. Mit Barrieren kann der Übergang in die nächste Phase synchronisiert werden. Eine doppelte Barriere ist eine Spezialform der Barriere, die einen bestimmten Bereich definiert, den alle Prozesse gemeinsam betreten und auch gemeinsam wieder verlassen.

3.3.1 Rezept

Angenommen die Gruppe der zu synchronisierenden Prozesse hat x Mitglieder. Des Weiteren sei der Pfad des Barriere Knotens innerhalb von ZooKeeper `/barrier`. Der Ablauf für die Clients besteht aus einer Eintrittsprozedur und einer Austrittsprozedur. Folgender Ablauf dient dem Eintritt in die doppelte Barriere:

1. Setze ein Watch auf den Knoten `/barrier/ready` mittels `exists()`. Das Erscheinen dieses Knotens dient als Signal, dass die erste Barriere passiert werden darf.
2. Erstelle einen Knoten `/barrier/p-` als Ephemeral Node mit Sequenznummer. Hiermit meldet sich der Prozess bei der Barriere an.
3. Frage die Liste der Unterknoten von `/barrier` ab:
   (a) Sind insgesamt x Prozesse bei der Barriere registriert, so erstelle den Knoten `/barrier/ready` und passiere die erste Barriere.
   (b) Sind weniger als x Prozesse registriert, so warte auf die Nachricht, dass der `/barrier/ready` Knoten erzeugt wurde und passiere anschließend die Barriere.

Der `ready`-Knoten dient als Zeichen dafür, dass alle Prozesse die Barriere erreicht haben. Er wird vom letzten eintreffenden Prozess erzeugt. Als Folge davon werden über das Watch alle anderen Prozesse aufgeweckt und können gemeinsam die Barriere passieren. Anschließend durchlaufen sie den Bereich innerhalb der Barriere und führen dabei den dort vorhandenen Code aus. Der hier auftretende Herden-Effekt ist gewünscht. Für den anschließenden Austritt aus der Barriere wird folgender Ablauf genutzt:

1. Frage die Liste der Unterknoten von `/barrier` ab:
   (a) Wenn keine Prozessknoten vorhanden sind, verlasse die Barriere.
   (b) Wenn nur noch der eigene Prozessknoten übrig ist, lösche ihn und verlasse die Barriere.
   (c) Wenn der eigene Prozessknoten die kleinste Nummer hat, überwache den Knoten mit der höchsten Nummer über ein Watch. Gehe zu Schritt 1, wenn er verschwindet.
   (d) Ansonsten: Lösche den eigenen Knoten und überwache den Knoten mit der kleinsten Nummer über ein Watch. Gehe zu Schritt 1, wenn er verschwindet.

Innerhalb der Barriere kann es jederzeit zu Ausfällen der Prozesse kommen. Mit dem Prozess verschwindet auch der Ephemeral Node in ZooKeeper, mit dem sich der Prozess registriert hatte. Die übrig gebliebenen Prozesse sollen die Barriere dann weiterhin verlassen können. Deshalb ist das Verfahren mit dem `ready`-Knoten für den Austritt aus der Barriere nicht geeignet. Stattdessen löschen die Prozesse am Ende der Barriere ihren Knoten und warten solange, bis alle Knoten gelöscht wurden. Ausgefallene Knoten werden automatisch gelöscht.

Damit allerdings nach jeden Löschvorgang nicht alle Prozesse aufwachen und prüfen müssen, ob alle Knoten gelöscht wurden, dient der Knoten mit der niedrigsten Nummer als Anhaltspunkt. Alle Prozesse überwachen nach dem Löschen des eigenen Knotens den niedrigsten. Nur der niedrigste Knoten löscht sich nicht sofort, sondern überwacht den mit der höchsten Nummer.

Nach und nach erreichen schließlich alle Prozesse die Barriere und löschen ihren Knoten. Wenn der höchste Knoten gelöscht wird, wacht der Prozess mit dem niedrigsten Knoten auf und prüft die Zahl der registrierten Prozesse. Ist nur noch er selbst verblieben, kann er den Knoten löschen, woraufhin alle Prozesse erwachen und die Barriere verlassen. Haben hingegen einige Prozesse das Ende der Barriere noch nicht erreicht, setzt er wieder ein Watch auf den jetzt höchsten Knoten. Damit wacht immer maximal ein Prozess zur Überprüfung auf und es ist trotzdem sichergestellt, dass alle Prozesse die Barriere gemeinsam verlassen.

Für den Fall, dass zwischenzeitlich der niedrigste Knoten ausfällt und damit gelöscht wird, suchen sich die anderen Prozesse den niedrigsten Knoten unter den verbliebenen als Anhaltspunkt.

3.3.2 Anwendungsfälle in der Cloud

Ein Bestandteil des Hadoop-Frameworks ist unter anderem das MapReduce-Framework [12, 18]. MapReduce dient der parallelen Verarbeitung großer Datenmengen auf Rechner-Clustern. Die mit dem Framework modellierten Rechenjobs laufen in festgelegten Phasen ab. Die Map-Operation erzeugt aus den Eingangsdaten Schlüssel-Wert-Paare und speichert diese als Zwischenergebnis ab. In der folgenden Sortierphase werden diese Zwischenergebnisse sortiert, um anschließend mit der Reduce-Operation bearbeitet zu werden. Die Reduce-Operation darf allerdings nicht starten, bevor die Map- und Sortierphase auf allen Rechnern beendet ist. Um diese Eintritte in die nächste Verarbeitungsphase zu synchronisieren, werden von ZooKeeper koordinierte Barrieren verwendet.

Hadoop kommt beispielsweise bei Yahoo zum Einsatz, um die Daten für die firmeneigene Internet-Suchmaschine zu produzieren und deren Index zu berechnen [14, 19]. Die verwendeten Cluster führen auf über 10.000 Prozessorkernen verteilte Rechenjobs durch, die alle miteinander abgestimmt werden müssen.

4. FAZIT

ZooKeeper ist angetreten, um eine Art Standard-Lösung für die Koordinierung in Cloud-Computing-Systemen zu werden. Die Beispiele aus Abschnitt 3 zeigen, dass ZooKeeper mittlerweile in vielen Projekten aus dem Cloud-Computing-Umfeld eingesetzt wird. Dies lässt darauf schließen, dass die Implementierung einen Reifegrad erreicht hat, der es erlaubt ZooKeeper produktiv einzusetzen. Der Status als Unterprojekt von Apache Hadoop kommt ZooKeeper hier zu gute und unterstützt seine Verbreitung. Neben ZooKeeper ist auch der Chubby Lock Service [11] sehr verbreitet, der allerdings nur intern bei Google Anwendung findet.

Es ist aber anzumerken, dass ZooKeeper immer noch großteils von Yahoo getragen wird. Sowohl die aktiven Entwickler als auch die Anwender von ZooKeeper haben meist Verbindungen zu Yahoo. Deshalb beschränken sich die Informationen, die man über ZooKeeper findet, in erster Linie auf Veröffentlichungen seitens der Entwickler. Unabhängige Untersuchungen und Evaluationen des Systems sind nicht bekannt.

ZooKeeper ermöglicht es, robuste, zuverlässige Koordinierungsfunktionen in eigene Systeme zu integrieren ohne den Aufwand der Eigenimplementierung und den damit verbundenen Problemen zu betreiben. Der Aufwand für die Einarbeitung in ZooKeeper sowie die Einrichtung des Dienstes sollten aber nicht außer Acht gelassen werden. Dies ist eine einmalige Investition, wie sie bei allen externen Komponenten notwendig ist. Doch durch das Verstecken der Komplexität hinter einer einfachen API bleibt ZooKeeper für die Einbindung in eigene Projekte weiter attraktiv.

Funktionen, die ZooKeeper nicht direkt mitbringt, lassen sich anhand von Rezepten (siehe Abschnitt 3) und der eingängigen Dateisystemstruktur einfach implementieren. Dies hat unter anderem den Vorteil, dass ZooKeeper nicht auf wenige geplante Funktionen beschränkt ist, sondern die Programmierer eigene Konstrukte umsetzen können. Das erleichtert auch die Anpassung und Optimierung hinsichtlich des Einsatzszenarios.

Die Skalierbarkeit von ZooKeeper wird in erster Linie durch die Größe des Arbeitsspeichers und den Master-Server begrenzt. Die Arbeitsspeichergröße beschränkt die Menge an Daten, die mit ZooKeeper verwaltet werden können. Jeder ZooKeeper-Rechner hält ein Replikat der kompletten Datenbank im Speicher um die Performanz zu erhöhen. Gleichzeitig begrenzt dies aber auch die Datenmenge. Weil ZooKeeper jedoch für die Verwaltung von Daten im Kilobyte-Bereich konzipiert wurde, kann er trotz begrenztem Arbeitsspeicher eine große Anzahl von Knoten verwalten.

Der Master-Server gibt die Grenze für den maximalen Durchsatz des Systems vor, da alle Schreiboperationen über ihn geleitet werden. Bei hoher Last kann er zum Flaschenhals für das System werden.

Insgesamt lässt sich die Koordinierung in Cloud-Computing-Systemen durch ZooKeeper stark vereinfachen. Ob ZooKeeper sich weiter durchsetzen wird, hängt stark davon ab, wieviel Zuspruch er bei Projekten außerhalb von Apache bzw. Yahoo findet. Letztendlich ist die Frage, ob sich die Annahmen, die bei der Entwicklung von ZooKeeper getroffen wurden, mit der Realität der Projekte decken. ZooKeeper muss nicht nur die vielfältigen funktionalen Anforderungen erfüllen, sondern auch nichtfunktionalen Anforderungen wie Performanz und Fehlertoleranz ausreichend begegnen können.

5. LITERATUR

[1] 101tec Inc. Katta – About. `http://katta.sourceforge.net/about`. [letzter Zugriff: 22.06.2010].

[2] 101tec Inc. Katta – Distributed Lucene. `http://katta.sourceforge.net`. [letzter Zugriff: 22.06.2010].

[3] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org`. [letzter Zugriff: 28.05.2010].

[4] Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/java/docs/`. [letzter Zugriff: 22.06.2010].

[5] Apache Software Foundation. Apache Solr. `http://lucene.apache.org/solr/`. [letzter Zugriff: 22.06.2010].

[6] Apache Software Foundation. The Apache Cassandra Project. `http://cassandra.apache.org/`. [letzter Zugriff: 22.06.2010].

[7] Apache Software Foundation. Zookeeper 3.3 Documentation. `http://hadoop.apache.org/zookeeper/docs/r3.3.0/`. [letzter Zugriff: 13.04.2010].

[8] Apache Software Foundation. ZooKeeper Credits. `http://hadoop.apache.org/zookeeper/credits.html`. [letzter Zugriff: 28.05.2010].

[9] Apache Software Foundation. ZooKeeper Recipes and Solutions. `http://hadoop.apache.org/zookeeper/docs/r3.3.0/recipes.html`. [letzter Zugriff: 15.06.2010].

[10] Apache Software Foundation. Zookeeper/PoweredBy. `http://wiki.apache.org/hadoop/ZooKeeper/PoweredBy`. [letzter Zugriff: 22.06.2010].

[11] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–113, 2004.

[13] T. Distler and T. Hönig. Ausgewählte Kapitel der Systemsoftware: Cloud Computing. `http://www4.informatik.uni-erlangen.de//Lehre/SS10/HS_AKSS/slides/00_Einfuehrung.pdf`, 2010.

[14] Flavio Junqueira. ZooKeeper: Because building distributed systems is a zoo. Talk at UPC (Universidad Politécnica de Catalunya) `http://wiki.apache.org/hadoop/ZooKeeper/ZooKeeperPresentations`. [letzter Zugriff: 22.06.2010].

[15] F. Junqueira, M. Konar, A. Kornev, and B. Reed. ZooKeeper - Hadoop Summit. `http://research.yahoo.com/files/zookeeper-HadoopSummit.pdf`. [letzter Zugriff: 22.06.2010].

[16] F. P. Junqueira and B. C. Reed. The life and times of a zookeeper. In *Proceedings of the 21th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 46–46, 2009.

[17] A. Lakshman and P. Malik. Cassandra – A Decentralized Structured Storage System. In *Proceedings of the 3rd ACM International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, pages 35–40, 2009.

[18] A. Singer. Programmierunterstützung im Kontext von Cloud Computing. `http://www4.informatik.uni-erlangen.de//Lehre/SS10/HS_AKSS/papers/04_Ausarbeitung_Alexander_Singer.pdf`, 2010.

[19] Wikipedia. Hadoop at Yahoo! `http://en.wikipedia.org/wiki/Hadoop#Hadoop_at_Yahoo.21`. [letzter Zugriff: 22.06.2010].

[20] Yahoo Inc. Yahoo Research - Zookeeper. `http://research.yahoo.com/project/1849`. [letzter Zugriff: 28.05.2010].