TODOs (Vor dem Druck entfernen!)

Ein streng konsistenter Koordinierungsdienst als Basis für verteilte Anwendungen

Masterarbeit

von

Nils Axer

aus

Gräfelfing

vorgelegt an der

Abteilung Betriebssysteme

Prof. Dr. Michael Schöttner

Heinrich-Heine-Universität Düsseldorf

2. Mai 2019

Gutachter:

Prof. Dr. Michael Schöttner

Zweitgutachter:

Prof. Dr. Martin Mauve

Betreuer:

Stefan Nothaas

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

1.2 Anforderungen

1.3 Ziele der Arbeit

Kapitel 2

Theorie

2.1 CAP-Theorem

Um ein System zu entwerfen, das die Anforderungen erfüllt, ist es zunächst wichtig herauszufinden, welche Eigenschaften ein verteiltes System überhaupt theoretisch erfüllen kann. Dabei hilft das CAP-Theorem [5]. Es beweist, dass ein verteilter Speicher von den drei Eigenschaften

- **Konsistenz**: Wird von den Autoren als *Linearizability* (auch *Atomic Consistency*) definiert: Wenn ein Write abgeschlossen ist, müssen alle Reads, die danach beginnen, den Wert des Writes oder eines späteren Writes zurückgeben. Dies ist ein sehr starkes Konsistenzmodell.
- **Verfügbarkeit**: Das System ist zu jedem Zeitpunkt erreichbar und antwortet auf alle Anfragen an nicht abgestürzte Knoten.
- **Partitionstoleranz**: Das System funktioniert weiter falls das Netzwerk partitioniert wird, also ein Teil der Knoten von einem anderen Teil abgeschnitten ist.

höchstens zwei gleichzeitig erfüllen kann.
Konsistenz und Verfügbarkeit kann man als graduelle Eigenschaften ansehen. Im CAP-Theorem wird von einem sehr starken Konsistenzmodell ausgegangen, das sich je nach Anwendung auch lockern lässt. Ebenso kann man die Verfügbarkeit graduell anpassen. Es muss also abgewogen werden, welche dieser Eigenschaften für ein System wichtig sind und welche nicht. Deshalb kann es viele verschiedene System mit verschiedenen Eigenschaften für ganz unterschiedliche Einsatzgebiete geben. Partitionstoleranz ist in der Realtität wichtig, da man diese in typischen Einsatzorten wie Datenzentren nie ausschließen kann. Netzwerkpartitionierungen können z.B. durch Ausfälle von einzelnen Switches oder Routern hervorgerufen werden.
Sehr viele verteilte Systeme sind auf hohe Verfügbarkeit optimiert, da sie z.B. ständig über das Internet erreichbar sein sollen. Darunter sind z.B. die meisten SQL- und NoSQL-Datenbanken und Key-Value-Stores wie DXRAM. SQL-Datenbanken wollen nach dem

ACID-Prinzip möglich starke Konsistenz der Daten, weshalb sie meist die Partitionstoleranz aufgeben. NoSQL-Datenbanken lockern das Konsistenzmodell nach dem BASE-Prinzip und versuchen dadurch die Partitionstoleranz zu erreichen. Einige Systeme verwenden auch ein externes System um wichtige Daten, die auch bei einer Netzwerkpartitionierung noch konsistent vorhanden sein müssen, separat zu speichern. Dies ist ein Anwendungsgebiet für das System, das in dieser Arbeit entworfen werden soll.
Ein weiteres Beispiel für ein System, welches die Konsistenz zugunsten der Verfügbarkeit und der Partitionstoleranz abschwächt, ist das Domain Name System (DNS). Die DNS-Einträge werden auf den Nameservern für eine bestimmte Zeit zwischengespeichert, um Verfügbarkeit und Partitionstoleranz zu erreichen. Dadurch ist es jedoch möglich, dass diese Caches nicht aktuell und somit nicht konsistent zu ihren Ursprüngen sind.
Besonders wichtig für das System, das in dieser Arbeit entworfen werden soll, ist die Konsistenz. Alle Knoten im verteilten System sollten stets die gleiche Sicht auf den Status haben, sodass sie sich koordinieren und auf bestimmte Dinge einigen können. Dafür muss gerade die *Atomic Consistency* erreicht werden, die auch im CAP-Theorem verwendet wird.
Um Konsistenz in einem verteilten System zu erreichen, muss das fundamentale Problem der Einigung auf einen Wert (oder einer Sequenz von Werten) gelöst werden. Wenn sich alle Knoten eines verteilten Systems auf den gleichen Wert geeinigt haben und dieser für alle sichtbar ist, dann ist das System für alle Knoten konsistent. Algorithmen, die dies ermöglichen, nennt man deshalb *Einigungsalgorithmen*.
Da das Problem im Allgemeinen nicht trivial ist, sollen im Folgenden ein paar Algorithmen vorgestellt werden, die das Problem mit vereinfachten Modellen, die meist nicht den realen Bedingungen entsprechen, lösen.

2.1.1 Flooding Consensus

Der Flooding Consensus Algorithmus löst das Einigungsproblem in einem synchronen System und mit einem Fail-Stop-Modell, d.h. Server können abstürzen und senden dann keine Nachrichten mehr. Der Algorithmus läuft in n Runden ab. Jeder Server hat zu Beginn einen Wert und speichert sich die gesamte Liste der empfangenen Werte. In jeder Runde flutet jeder Server das Netzwerk mit seinen bekannten Werten, d.h. er sendet seine Liste an alle teilnehmenden Server. Falls in einer Runde mindestens ein Server beim Senden oder vor dem Senden ausfällt, kann in dieser Runde keine Einigung erreicht werden, da es sein kann, dass die Server unterschiedliche Werte in ihren Listen haben. Um f fehlerhafte Server zu kompensieren, werden also maximal $f+1$ Runden benötigt, bis die Server alle die gleichen Werte in ihren Listen haben. Dann können sie mit einer Funktion, die auf der Liste ausgeführt wird, den Entscheidungswert bestimmen. Dies könnte z.B. die Maximumsfunktion sein. Um festzustellen, wann eine Runde vorbei ist, muss die maximale Nachrichtenlaufzeit und die maximale Bearbeitungszeit bekannt sein. Wenn dies bekannt ist, können die Server in jeder Runde diese Zeit abwarten und es ist sicher, dass ein Server, von dem in dieser Zeit keine Nachricht erhalten wurde, abgestürzt ist. Da man diese Zeit in der Praxis jedoch normalerweise nicht bestimmen kann, ist dieser Algorithmus meistens nicht für einen Einsatz geeignet.

2.1.2 2-Phase-Commit

Der 2-Phase-Commit-Algorithmus löst das Einigungsproblem in einem asynchronen System mit einer perfekten Ausfallerkennung. Hier geschieht die Einigung mithilfe eines Koordinators, der vorher mit einem externen Verfahren bestimmt werden muss, z.B. einem Leader-Election-Algorithmus. Da der Koordinator auch ausfallen kann und ein neuer Koordinator einen inkonsistenten Wert verbreiten könnte, kann dieser nicht einfach alle Werte einsammeln und entscheiden, sondern der Algorithmus muss in zwei Phasen durchgeführt werden:

- **Phase 1**: Der Koordinator sendet eine *propose*-Nachricht an alle Server mit seinem Wert, auf den sich geeinigt werden soll. Die Server akzeptieren den vorgeschlagenen Wert und merken sich diesen. Der Koordinator wartet auf Antworten von allen nicht ausgefallenen Servern (hier wird die Ausfallerkennung benötigt). Sobald er von allen Servern eine Antwort erhalten hat, ist es sicher, dass ein möglicher neuer Koordinator denselben Wert verbreiten wird, falls dieser Koordinator ausfällt. Die Aktivierung eines neuen Koordinators kann ebenfalls nur durch die perfekte Ausfallerkennung erreicht werden, sonst kann es sein, dass es mehrere oder keinen Koordinator gibt.
- **Phase 2**: Der Koordinator sendet eine *decide*-Nachricht an alle Server, die die Entscheidung auf den vorgeschlagenen Wert darstellt. Sobald er von allen Server eine Antwort erhalten hat, haben sich alle Server auf diesen Wert geeinigt.

Der 2-Phase-Commit Algorithmus wird in der Praxis eingesetzt, z..B. in der Oracle Database [1] um verteilte Transaktionen zu ermöglichen. Da es dabei jedoch keine perfekte Ausfallerkennung gibt, wird bei unerfolgreichem Ausführen des Algorithmus, z.B. bei Ausfällen oder Nachrichtenverlusten, die Datenbank blockiert und die Transaktion als „In-Doubt “markiert. Die Datenbank versucht dann, den Fehler automatisch zu beheben, dies kann jedoch nicht in allen Fällen geschehen. Dann muss der Fehler manuell behoben werden.
Da der Algorithmus keine Fehlertoleranz garantieren kann und blockiert, sobald einer der teilnehmenden Server ausfällt, ist er nicht für das System, das in dieser Arbeit entworfen werden soll, geeignet.

2.2 FLP-Theorem

Zum Verständnis von Einigungsalgorithmen ist noch ein weiteres Theorem wichtig: Das FLP-Theorem [4]. Es zeigt, dass es keinen Einigungsalgorithmus in einem **asynchronen System** geben kann, der bei Knotenausfällen korrekt arbeitet und immer terminiert. Ähnlich wie bei CAP-Theorem kann man hier ebenfalls drei Eigenschaften formulieren, die ein Algorithmus nicht gleichzeitig erfüllen kann: Sicherheit, Verfügbarkeit und Fehlertoleranz. Ein Unterschied zum CAP-Theorem ist, dass hier von einem schwächeren Modell ausgegangen wird: Die teilnehmenden Server können zwar abstürzen, das Netzwerk ist jedoch zuverlässig und es kann insbesondere keine Netzwerkpartitionierung geben. Alle

Einigungsalgorithmen, die auf einem asynchronem Modell basieren, sind also nur partiell korrekt. Für die in dieser Arbeit vorgestellten Algorithmen bedeutet dies, dass sie sich auf die Eigenschaften Sicherheit und Fehlertoleranz fokussieren. Dadurch kann man im Allgemeinen nicht beweisen, dass sie auch terminieren. Meist wird durch geschickt gewählte Timeouts oder Ähnliches in der Praxis erreicht, dass die Algorithmen zur meisten Zeit terminieren und zu einem Ergebnis kommen. Tatsächlich gibt es auch Einigungsalgorithmen, die immer terminieren, jedoch dafür die Sicherheit oder die Fehlertoleranz opfern. Beispiele für Algorithmen, bei denen es zugunsten der Terminierung manchmal falsche Ergebnisse geben kann, sind der 3-Phase-Commit-Algorithmus [2] und probabilistische Algorithmen wie Proof-of-Work in Blockchains .

uer erklären

2.3 Crash-Recovery-Modell

Alle in dieser Arbeit vorgestellten Einigungsalgorithmen basieren auf dem *Crash-Recovery-Modell*. Server arbeiten asynchron, können abstürzen und nach einem Neustart wieder am Cluster teilnehmen. Außerdem kann das Netzwerk unzuverlässig sein. Nachrichten können also beliebig viel Zeit von einem Knoten zu einem anderen Knoten benötigen, verloren gehen oder die Reihenfolge ändern. Ziel dabei ist, dass der Einigungsalgorithmus solange eine Einigung erzielen kann, wie eine Mehrheit der Server funktioniert und erreichbar ist. Die teilnehmenden Server dürfen sich jedoch nicht beliebig falsch verhalten, also keine falschen Nachrichten schicken und keine Systemzustände erreichen, die nicht vorgesehen sind. Es gibt auch Einigungsalgorithmen, die gegen solche byzantinischen Fehler geschützt sind [9], jedoch erhöht dies die Komplexität deutlich und es werden im Vergleich zum Crash-Recovery-Modell mehr Server benötigt, um sich vor der gleichen Anzahl an fehlerhaften bzw. ausfallenden Servern zu schützen, was das System insgesamt langsamer macht.

2.4 Einigungsalgorithmen

2.4.1 Paxos

Der bekannteste Einigungsalgorithmus ist Paxos [7,8]. Paxos implementiert eine *Replicated State Machine*.

Grafik zur Replicated State Machine

Bei diesem Modell besitzt jeder Knoten eine Zustandsmaschine, die Befehle ausführt und dadurch den Zustand ändert. Der Einigungsalgorithmus kümmert sich darum, dass die Befehle auf allen Knoten in der gleichen Reihenfolge durchgeführt werden, sodass der Zustand auf allen Knoten konsistent ist.
Die einfache Version von Paxos (*Basic Paxos*) ermöglicht die Einigung auf einen einzelnen Wert. Dabei werden die Teilnehmer in drei Rollen aufgeteilt:

- Die **Acceptors** hören auf vorgeschlagene Werte, speichern diese und akzeptieren sie, falls sie nicht schon einen konfligierenden Wert akzeptiert haben.

- Der **Proposer** schlägt einen Wert vor, auf den sich geeinigt werden soll. Er sendet diese an die Acceptors.
- Die **Learner** erhalten den Wert, auf den sich geeinigt wurde und können entsprechend handeln, z.B. indem sie eine Antwort an einen Client senden.

Die Einigung auf einen Wert erfolgt in zwei Phasen:

1. a) Der Proposer wählt eine Proposal Number n und sendet eine Prepare-Nachricht mit n an die Acceptors.
   b) Die Acceptors akzeptieren n, falls sie noch keine Proposal Number akzeptiert haben, die größer als n ist. Sie antworten mit der höchsten Proposal Number, die sie akzeptiert haben.
2. a) Wenn der Proposer von einer Mehrheit der Acceptors eine Antwort auf seine Prepare-Nachricht mit der Proposal Number n] erhalten hat, sendet er eine Accept-Nachricht an die Acceptors mit n und einem Wert v, auf den sich geeinigt werden soll.
   b) Die Acceptors akzeptieren den vorgeschlagenen Wert v mit der Proposal Number n, falls sie noch nicht auf eine Prepare-Nachricht mit höherer Proposal Number geantwortet haben.

Um sich auf beliebig viele Werte zu einigen, werden beliebig viele Paxos-Instanzen hintereinander ausgeführt (*Multi-Paxos* [3]). Dabei können mehrere Optimierungen vorgenommen werden. Da sich mehrere Proposer gegenseitig behindern würden und das System u.U. dann häufig keine Einigung erzielen könnte, sollte im Normalbetrieb nur ein einzelner Proposer vorhanden sein. Dieser kann durch einen beliebigen Leader-Election-Algorithmus gewählt werden. Falls dieser ausfällt, muss ein neuer Proposer gewählt werden. Es ist jedoch kein Problem, falls es zu einem Zeitpunkt mehrere Leader geben sollte, da Paxos dann garantiert, dass keine unterschiedlichen Werte akzeptiert werden. Dies sollte jedoch nicht die Regel sein. Außerdem kann der Leader die Phase 1 einmalig für beliebig viele Werte durchführen, wodurch sich die Nachrichtenanzahl reduziert.
In Implementierungen wird meist eine Client-Server-Architektur verwendet. Dabei übernehmen die Server alle drei Rollen. Die Clients senden (Lese- und Schreib-)Anfragen an das System und bekommen die Werte zurück, auf die sich geeinigt wurde, ähnlich einer Datenbank.
Paxos wurde bereits in anderen Arbeiten implementiert [3, 12]. Die Autoren berichten jeweils von nicht-trivialen Problemen, auf die sie während der Arbeit an einer Implementierung gestoßen sind. Dafür mussten sie unter anderem Optimierungen vornehmen, die nicht in der initialen Beschreibung des Algorithmus enthalten sind. Außerdem gibt es sehr wenige Implementierungen, insbesondere welche die Open-Source sind.

2.4.2 Zookeeper/Zab

Kerneigenschafte
genau formuliere
eigene Grafik zu
Paxos Randfall e
stellen

Zookeeper implementiert einen Algorithmus namens Zab [6]. Zab ist ebenfalls ein Einigungsalgorithmus, der jedoch leicht andere Garantien bietet. Die Autoren nennen ihr Modell ein *Primary-Backup System*. Es gibt zu jedem Zeitpunkt höchstens einen Primary-Server, der Vorschläge für den nächsten Befehl an die anderen Server im Cluster senden kann. Die anderen Server sind exakte Backup-Replikate des Primary-Servers. Falls der Primary-Server ausfällt, wird eine neuer Primary-Server aus den vorhandenen Backup-Servers gewählt. Die Autoren erklären den Unterschied zu Paxos so: Zab soll garantieren, dass jede Zustandsänderung von allen vorherigen Zustandsänderungen abhängt und somit niemals vor diesen ausgeführt werden darf. Außerdem sollen von einem Proposer mehrere Befehle, hier Transaktionen genannt, einzeln und nebenläufig vorgeschlagen werden können, ohne dass sich ihre Reihenfolge ändern kann. Paxos kann in diesem Randfall jedoch nicht die initiale Reihenfolge garantieren (siehe Abbildung 2.1).

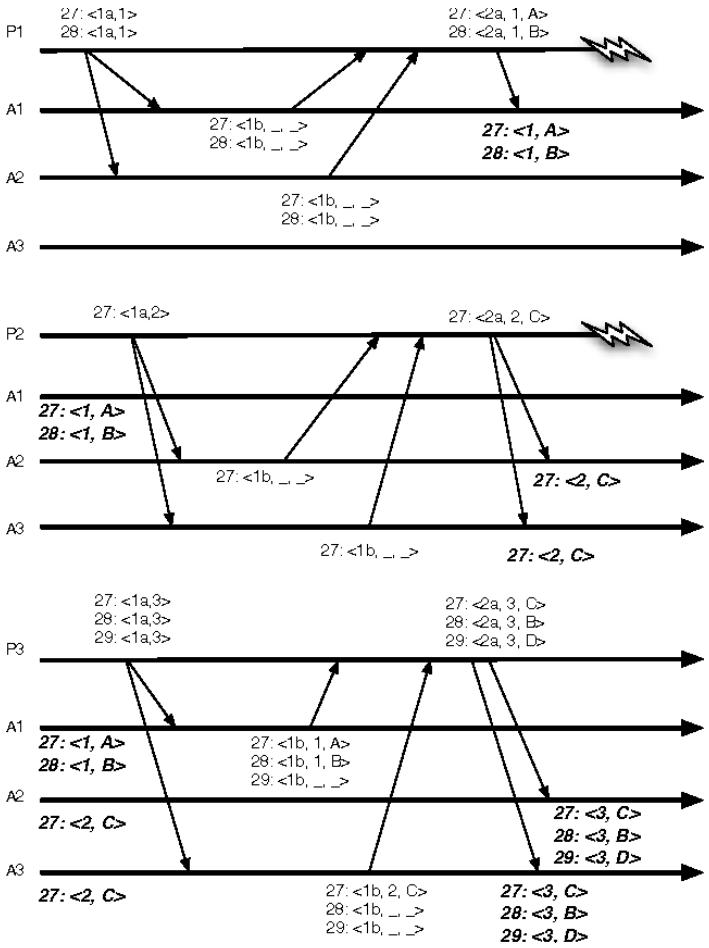

Abbildung 2.1: Randfall von Paxos, bei dem sich die Reihenfolge von Zustandsänderungen ändern kann, nachdem sie bereits in einer anderen Reihenfolge von einen Proposer vorgeschlagen wurden. P1 schlägt die Werte A und B für die Indizes 27 und 28 vor. P3 schlägt jedoch die Werte C und B für die Indizes 27 und 28 vor und bekommt dafür eine Mehrheit, obwohl B eigentlich von A abhängig ist.

Bei Paxos kann dieser Randfall jedoch umgangen werden, indem mehrere Transaktionen eines Proposers in eine einzelne Paxos-Vorschlag verpackt werden und immer nur ein Vorschlag gleichzeitig aktiv sein kann. Dies schadet laut den Autoren jedoch entweder dem Durchsatz oder der Latenz, weshalb sie sich für die Entwicklung von Zab entschieden haben.
Zab teilt den Algorithmus in drei Phasen ein:

1. **Discovery**: Es wird ein Kandidat für einen neuen Leader bestimmt und dieser bekommt die Logs (hier History genannt) aller anderen Server (Follower) zugeschickt. Der Kandidat wird durch einen beliebigen Leader Election Algorithmus bestimmt.

2. **Synchronization**: Der Kandidat schlägt sich selbst als Leader vor und sammelt Stim-

men dafür ein. Außerdem werden die Logs der Follower auf den Stand des Leaders gebracht. Wenn der Kandidat ein Quorum hat, wird er zum Leader.

3. **Broadcast**: Der normale Betrieb. Der Leader schlägt Werte vor und wartet auf Bestätigungen der anderen Server. Falls er die Verbindung zu einer Mehrheit der Server verliert, tritt er als Leader zurück und geht wieder in die Discovery-Phase über.

Durch die Discovery- und Synchronization-Phasen werden die Kerneigenschaften des Systems sichergestellt, z.B. dass der neue Leader keine bereits abgeschlossenen Zustandsänderungen löschen kann und dass alle Server die Befehle in der gleichen Reihenfolge ausführen. Die Broadcast-Phase läuft ähnlich dem 2-Phase-Commit-Protokoll [2] ab.

2.4.3 Raft

Raft Algorithmus detailliert erklären

Raft [10, 11] basiert ebenso wie Paxos auf dem Replicated-State-Machine-Modell basiert. Die Autoren begründen die Entwicklung von Raft damit, dass Paxos schwierig zu verstehen und zu implementieren sei. Deswegen wollten sie einen einfacheren Algorithmus entwickeln, der auf dem selben Modell basiert. Raft geht ebenso wie Paxos von einem Crash-Recovery-Modell und einem unzuverlässigen Netzwerk aus. Im Gegensatz zu Paxos wird das Einigungsproblem bei Raft in drei Teilprobleme aufgeteilt:

- **Leader Election**: Die Wahl eines Leaders ist bei Raft fester Bestandteil des Algorithmus
- **Log Replication**: Das Log mit den Befehlen muss vom Leader auf die anderen Server des Cluster repliziert werden und es muss sich auf eine Reihenfolge geeinigt werden.
- **Safety**: Es muss Folgendes gelten: Alle Befehle müssen von allen Zustandsmaschinen in der gleichen Reihenfolge ausgeführt werden.

Leader Election

Ein Server hat immer einen der States *Follower*, *Candidate* oder *Leader*. Alle Server starten als Follower. Raft teilt die Zeit in *Terms* auf, die fortlaufend nummeriert werden. Jeder Term beginnt mit der Wahl eines Leaders. Jeder Server speichert seinen aktuellen Term und tauscht ihn bei jeder Kommunikation mit den anderen Servern aus. Wenn ein Server eine Anfrage mit einem alten Term bekommt, dann lehnt er diese ab. Wenn er eine Anfrage mit einem höheren Term bekommt, übernimmt er diesen und wird zum Follower. Eine Leader Election wird gestartet, wenn ein Follower eine Zeit lang keinen Heartbeat von einem Leader bekommt. Er wird dann zu einem Candidate und versucht Leader zu werden. Dabei startet er einen neuen Term, indem er seinen gespeicherten Term inkrementiert. Er sendet dann Vote Requests an alle anderen Server. Diese geben dem, Candidate ihre Stimme, falls sie ihre Stimme nicht bereits im selben Term einem anderen Server gegeben haben. Falls der

Candidate von der Mehrheit der Server eine Stimme erhalten hat, wechselt er seinen Status zum Leader. So ist sichergestellt, dass es in einem Term maximal einen Leader geben kann. Es kann jedoch passieren, dass es in einem Term keinen Leader gibt. Falls mehrere Follower im gleichen Term eine Wahl starten (falls der Timer, der die letzte Nachricht von einem Leader überwacht, etwa zur gleichen Zeit ausläuft), kann es sein, dass es keine Mehrheit für einen Kandidaten gibt. Dann beginnen ein oder mehrere Candidates einen neuen Term und starten eine neue Wahl. Damit solche Konflikte selten vorkommen, sollten die Timeouts randomisiert werden, sodass es unwahrscheinlich ist, dass mehrere Server gleichzeitig eine Wahl starten.
Sobald ein Server sich zum Leader ernannt hat, kann er Anfragen von Clients bearbeiten und sendet periodisch Heartbeats an alle Follower, um seine Leadership aufrecht zu erhalten.

Log Replication

Im normalen Betrieb, d.h. ohne Leaderwechsel, nimmt der Leader Client-Anfragen an, schreibt diese ins Log und versucht sie, auf die Follower zu replizieren. Dazu sendet er sogenannte Append Entries Requests mit den Einträgen, die er replizieren möchte, an die Follower. Die Follower fügen diese in ihr eigenes Log ein und senden eine Bestätigung zurück, falls dies erfolgreich durchgeführt wurde. Der Leader merkt sich für jeden Follower einen *Match Index*, der angibt, bis zu welchem Eintrag das Log des Followers dem des Leaders entspricht. Dadurch kann er feststellen, wann ein Eintrag auf eine Mehrheit der Server repliziert wurde. Sobald dies der Fall ist, kann er den Eintrag seiner State Machine übergeben und ausführen. Dann kann eine Antwort an den Client gesendet werden. Mit dem nächsten Append Entries Request oder Heartbeat wird der *Commit Index* an die Follower weitergegeben, sodass diese den Eintrag ebenfalls ausführen können.
Nach einem Leader-Wechsel muss der Leader die Logs der Follower seinem eigenen Log angleichen. Dafür speichert er sich für jeden Follower einen *Next Index*, der angibt, welcher Eintrag als nächstes an diesen Follower gesendet werden soll. Nach der Wahl zum Leader wird dieser für alle Follower auf den Index des letzten Eintrags initialisiert. Der Leader sendet außerdem bei jedem Append Entries Request den Term des letzten Eintrags, der vor den gesendeten Einträgen im Log ist, mit. Follower lehnen ein Append Entries Request ab, falls dieser Term nicht mit dem Term des Eintrags an diesem Index in ihrem Log übereinstimmt. Dann muss der Leader den *Next Index* dieses Followers dekrementieren und es nochmal versuchen. Falls diese Überprüfung beim Follower nicht fehlschlägt und bereits inkonsistente Einträge vorhanden sind, werden alle folgenden Einträge gelöscht, sodass das Log dann bis zum Index des letzten gesendeten Eintrags dem des Leader entspricht.

Safety

Die zentrale Eigenschaft, die gelten muss damit das System korrekt funktioniert, ist: Sobald ein Log-Eintrag von einer State Machine ausgeführt wurde, darf keine andere State Machine auf einem anderen Server einen anderen Befehl für diesen Log-Eintrag ausführen. Dadurch wird offensichtlich sichergestellt, dass alle State Machines die Befehle in der gleichen Reihenfolge bearbeiten. Diese Eigenschaft ist in Raft äquivalent zu: Wenn ein Leader

entscheidet, dass ein Eintrag *committed* ist und diesen der State Machine übergibt, dann muss dieser Eintrag in den Logs aller zukünftigen Leadern vorhanden sein. Dies wird durch zusätzliche Einschränkungen beim Committen und bei der Leader Election erreicht.

- **Leader Election**: Die Candidates senden mit ihren Vote-Requests den Term und Index des letzten Eintrags ihres Logs mit, anhand derer die wählenden Server entscheiden, ob sie dem Candidate ihre Stimme geben oder nicht. Damit Log Einträge, die auf einer Mehrheit der Server repliziert sind, nicht gelöscht werden, gibt der Server dem Candidate keine Stimme, falls sein Log kompletter ist, d.h. falls der Term des letzten Eintrags größer ist oder falls der Term gleich ist und der Index größer ist. Dadurch hat der nächste Leader dann das kompletteste Log unter einer Mehrheit der Server.
- **Commit**: Um einen Eintrag zu committen, muss mindestens ein Eintrag aus dem aktuellen Term des Leaders auf einer Mehrheit der Server repliziert worden sein. Sonst kann es passieren, dass ein Server Leader wird, der die auf einer Mehrheit replizierten Einträge nicht in seinem Log hat, da er nur neuere Einträge in seinem Log hat. Dann würden Einträge gelöscht, die schon committed sind.

rende Abbildung

Kapitel 3

Implementierung

3.1 Ziele

3.2 Server

3.2.1 Threads

3.2.2 Bootstrapping

3.2.3 Persistenz

3.3 Client

3.3.1 Exactly-Once-Semantics

3.3.2 API

3.4 Zukünftige Verbesserungen

Kapitel 4

Evaluation

4.1 Andere Systeme

4.1.1 Zookeeper

4.2 Benchmark

4.3 Ergebnisse

Anhang A

Mein Anhang

Klassendiagramme und weitere Anhänge sind hier einzufügen.

Literaturverzeichnis

[1] Oracle database administrator's guide. https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/distributed-transactions-concepts.html#GUID-8152084F-4760-4B89-A91C-9A84F81C23D1. Aufgerufen am 1.3.2019.

[2] Muhammad Atif. Analysis and verification of two-phase commit three-phase commit protocols. 2009 International Conference on Emerging Technologies, pages 326–331, 2009.

[3] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[4] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM, 32(2):374–382, April 1985.

[5] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51–59, June 2002.

[6] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), pages 245–256, 2011.

[7] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[8] Leslie Lamport. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001.

[9] Leslie Lamport. Byzantizing paxos by refinement. In Proceedings of the 25th International Conference on Distributed Computing, DISC'11, pages 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] D. Ongaro, J.K. Ousterhout, D.F. Mazières, M. Rosenblum, and Stanford University. Computer Science Department. Consensus: Bridging Theory and Practice. 2014.

[11] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[12] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. ACM Comput. Surv., 47(3):42:1–42:36, February 2015.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Axer, Nils

Düsseldorf, 2. Mai 2019