Fehlertolerante, konsistente Replikation mit Hilfe des PAXOS Algorithmus

Michael Meier
simimeie@stud.uni-erlangen.de

Kurzzusammenfassung

Dieses Dokument soll einen Ueberblick ueber die Funktionsweisse des PAXOS Algorithmus geben, sowie kurz erlaeutern wie dieser Algorithmus zur Fehlertoleranten, konsistenten Replikation von Festplatten eingesetzt werden kann.

1 Einfuehrung

1.1 Zweck

Das Herzstueck des PAXOS Algorithmus ist ein Einigungsprotokoll, das eine beliebige Anzahl von nicht-byzantinischen Fehlern tolerieren kann. Die Konsistenz der Daten bleibt dabei immer gewaehrleistet.

1.2 Geschichte

Der PAXOS Algorithmus wurde 1989 von Leslie Lamport entwickelt. Lamport sah sich damals die Versuche an, ein fehlertolerantes Dateisystem mit Namen ECHO zu entwickeln, und fand den dort verwendeten Algorithmus zu aufwaendig. Die Entwickler versuchten, alle moeglichen Fehlerbedingungen die auftreten konnten speziell zu behandeln. Lamport versuchte daraufhin einen allgemeinen Algorithmus zu entwickeln, und das gelang ihm mit PAXOS. Um den doch recht trockenen Stoff etwas aufzulockern, entwarf er eine Geschichte drumherum: Er beschrieb den Algorithmus als Erfindung, die auf der Insel Paxos Anfang des letzten Jahrtausends gemacht wurde, weil es auf der Insel nur Teilzeit-Politiker gab, die auch mitten waehrend einer Sitzung das Parlament verlassen konnten. Zudem haette das Parlamentsgebaeude so eine miese Akkustik gehabt, dass Kommunikation nur ueber Boten moeglich waere. Dies waere nun zufaellig bei Ausgrabungen entdeckt worden. Er trieb das sogar so weit, dass er einen griechischen Dialekt entwarf, und Vorlesungen zu dem Thema im Indiana-Jones-Outfit gab. Das ganze erwies sich jedoch als kompletter Schuss in den Ofen: Fast niemand erkannte die Bedeutung des Algorithmus, und die Leute die sich ueberhaupt die Muehe machten seine Erlaeuterungen zu lesen, wurden so von den griechischen Buchstaben und eingestreuten Anekdoten abgelenkt dass sie den Algorithmus nicht verstanden. Wohl auch aus diesem Grund wurde sein 1990 eingereichter Artikel erst 1998 veroeffentlicht.

2 Der Algorithmus

2.1 Voraussetzungen

Der Paxos-Algorithmus ist (zumindest wenn er nicht in griechischen Buchstaben beschrieben wird) sehr einfach, daher müssen eine Vielzahl von Bedingungen erfüllt sein, damit er fehlerfrei funktioniert.

- Es gibt keine byzantinischen Fehler
- Das Fehlverhalten der beteiligten Prozesse muss Fail-Stop sein, d.h. sie sind entweder da und voll funktionsfähig oder nicht da.
- Die Prozesse vertrauen einander blind, das heisst ein einziger Prozessor der fehlerhafte Nachrichten sendet, ja sogar eine einzige bei der Uebertragung verfaelschte Nachricht kann das System zusammenbrechen lassen.
- Jeder Prozess muss alle anderen beteiligten Prozesse kennen. Das macht natürlich das dynamische Hinzufügen neuer Prozesse unmöglich.
- Jeder Prozess muss direkt mit jedem anderen Prozess kommunizieren können, und auch mit sich selbst.
- Die Verbindungen duerfen zwar ausfallen, aber wenn eine Nachricht übermittelt wird, dann darf diese nicht verfälscht sein. Nachrichten duerfen also verloren gehen, verdoppelt werden, verzögert werden, vertauscht werden, aber niemals verfälscht werden.
- Wiederanlauf ist erlaubt, d.h. ein ausgefallener Prozess kann zu einem späteren Zeitpunkt wiederkommen.
- Die Prozesse benötigen einen nichtflüchtigen Speicher, auf dem sie Werte ablegen können. Das ist nötig, damit sie nach einem Wiederanlauf nicht Nachrichten senden, die zu Nachrichten die sie vorher versandt haben im Widerspruch stehen.
- Es muss auf irgendeine Weisse festgelegt sein, wie eine Mehrheit definiert ist. Die trivialste Möglichkeit dafür wäre natürlich, einfach die zahlenmässige Mehrheit dafür herzunehmen. Verbessern lässt sich das z.B. indem man besonders stabilen Prozessen mehr Gewicht einräumt. Schon im (fiktiven!) alten Paxos wurde beobachtet, dass fette Parlamentsmitglieder weniger beweglich waren, und sich deshalb öfter im Parlament aufhielten. Daher definierte man dort die Mehrheit als Mehrheit des Körpergewichts aller Parlamentsmitglieder.
- Die Prozesse muessen ueber irgendeine Moeglichkeit verfuegen, das Verstreichen von Zeit festzustellen (z.B. damit sie nach einem Timeout einen neuen Versuch starten koennen)
- Zudem muss, damit jemals ein Fortschritt erreicht werden kann, eine Mehrheit der Prozesse miteinander kommunizieren können. Wenn beispielsweisse das Netzwerk in zwei Teile zerfaellt, so dass in jedem Teil die Hälfte der Prozesse sitzt, können natürlich keine Einigungen mehr erzielt werden, weil niemals die erforderliche Mehrheit erreicht werden kann. Die Konsistenz bleibt während einer solchen Teilung jedoch erhalten, und nachdem die Störung beseitigt ist, läuft das System weiter.

- Fuer die Konsistenz ist es zwar auch nicht noetig, dass die beteiligten Prozesse ueberhaupt irgendetwas tun, schliesslich bleiben auch wenn quasi alles ausfaellt die Daten konsistent. Jedoch macht das wenig Sinn, daher sollte man festlegen dass alle beteiligten so schnell wie moeglich auf empfangene Nachrichten reagieren.

2.2 Funktionsweise

Es gibt 2 Varianten von Paxos: Das Synod-Protcol (Basic Paxos) und das Parliamentary-Protocol (Multi Paxos). Lamport beschreibt dabei das Parliamentary Protocol als Weiterentwicklung des Synod Protocol, die entstand weil das Parlament im Gegensatz zur Synode (Priesterversammlung) nicht nur alle neunzehn Jahre einen symbolischen Leitsatz beschloss, sondern eben permanent Gesetze erliess.

Wir haben also eine Gruppe von Prozessen die Werte vorschlagen können. Jeder Prozess kann Werte vorschlagen, und das Ziel des Algorithmus ist es, dass sich die Prozesse auf einen der vorgeschlagenen Werte einigen. Ausserdem muss sichergestellt werden, dass niemals ein Prozess glaubt, es hätte eine Einigung auf einen Wert gegeben, wenn das gar nicht der Fall ist. Wenn jedoch ein Wert gewählt wurde, sollen alle Prozesse davon erfahren.
Paxos verwendet dazu ein 3 Phasen Protokoll.

1. Phase:

Zuerst schickt der Prozess der einen Vorschlag machen will (Proposer genannt) an eine Menge von Prozessen (auch an sich selbst, denn auch er kann natürlich seinen eigenen Vorschlag annehmen) ein PREPARE-Request. Diese Anfrage ist mit einer Nummer versehen, die eindeutig sein muss. Um dies sicherzustellen, teilt man jedem Prozess einen eigenen Nummernbereich zu, aus dem er seine Nummern auswählt. Da die Anzahl der beteiligten Prozesse ja sowieso bekannt ist, bezeichnen wir sie einmal mit N, ist es leicht jedem Prozess einen eigenen Nummernbereich zuzuteilen: Prozess i (mit 0 <= i < N) darf die Nummern i, i+N, i+2N, i+3N ... verwenden. Der Prozess merkt sich in seinem nichtflüchtigen Speicher, welche Nummer er zuletzt verwendet hat, und verwendet dann die nächstgrössere aus seinem Vorrat.

Jeder Prozess merkt sich ausserdem (im nichtflüchtigen Speicher!) die Nummer des letzten PREPARE Requests das er akzeptiert hat. Wenn er jetzt ein PREPARE Request erhält, so vergleicht er die Nummer dieses Requests mit dieser gespeicherten Nummer.

- Ist die neue Nummer grösser, so antwortet er mit einem Versprechen, nie wieder ein PREPARE Request mit einer niedrigeren Nummer anzunehmen. Falls er zuvor schon einen Vorschlag angenommen hatte, so schickt er ausserdem den Vorschlag mit der höchsten Nummer den er angenommen hat zurück.
- anderenfalls ignoriert er die Anfrage.

2. Phase:

Wenn der Proposer von einer Mehrheit der Prozesse Antworten auf sein PREPARE Request erhalten hat, schickt er ein ACCEPT Request an diese Prozesse. Der Request enthält wieder eine Nummer, und den Vorschlag mit der höchsten Nummer, den er aus den Antworten auf sein PREPARE

Request ermittelt hat. Enthielt keine der Antworten auf das PREPARE Request einen Vorschlag (weil keiner der anderen Prozesse schon einen Vorschlag angenommen hatte), so kann er einen beliebigen eigenen Vorschlag senden.

Die Prozesse die die ACCEPT Requests erhalten reagieren wieder abhängig davon, ob die Nummer der Anfrage kleiner ist als die Nummer des zuletzt von ihnen angenommenen PREPARE Requests.

- Ist die Nummer der Anfrage kleiner, so wird sie ignoriert
- anderenfalls wird die Anfrage akzeptiert (und eine entsprechende Antwort gesendet).

Natürlich wäre es für die Performance besser, wenn dem Proposer mitgeteilt würde, dass sein PREPARE bzw. ACCEPT Request abgelehnt wurde, so dass dieser seine Anfrage abbrechen koennte anstatt bis zum Timeout zu warten, für das korrekte Funktionieren des Algorithmus ist es aber nicht erforderlich.

Erhält der Proposer von einer Mehrheit der Prozesse eine positive Antwort auf sein ACCEPT Request, dann ist der Vorschlag angenommen, die Einigung komplett.

3. Phase:

Nun stellt sich noch das Problem, wie die anderen Prozesse davon erfahren. Die einfachste Möglichkeit ist, einfach den Proposer das Ergebnis verteilen zu lassen. Diese hat natürlich den Nachteil, dass bei einem Ausfall des Proposers in einem ungünstigen Augenblick die Nachricht dass es eine Einigung gegeben hat bis zu seiner Rückkehr nicht verbreitet wird. Auch dies ist natürlich verbesserungsfähig, aber für das korrekte Funktionieren des Algorithmus nicht notwendig.

Eine andere Moeglichkeit waere, jeden Prozessor wenn er einen Wert akzeptiert, dies an alle anderen Prozessoren senden zu lassen. Der offensichtliche Nachteil dieser Methode ist die schiere Masse der dadurch erzeugten Nachrichten.

Generell wichtig ist jedoch, dass alle Daten die der Prozess nicht vergessen darf, auf dem nichtflüchtigen Speicher gesichert werden BEVOR die Daten dann wirklich über das Netz gesendet werden. Es würde den Algorithmus natürlich zerstoeren, wenn z.B. ein Prozess sein Versprechen senden würde, kein PREPARE mit einer Nummer < 10 mehr anzunehmen, dann ausfaellt bevor er das auf seinem Nichtfluechtigen Speicher gesichert hat, und daher nach einem Wiederanlauf wieder bei einer niedrigeren Zahl beginnen wuerde.

Zudem muss es irgendwann eine Einigung auf einen Anführer geben, der der alleinige Proposer ist. Gibt es mehrere, so können diese sich gegenseitig ausbremsen, so dass niemals eine Einigung zustandekommt. Die Konsistenz bleibt dabei zwar immer erhalten, es gibt jedoch moeglicherweisse keinen Fortschritt mehr.

3 Replikation von Daten mit Hilfe von PAXOS

3.1 Petal

Bei Petal handelt es sich um ein System von ueber ein (moeglichst schnelles) Netzwerk miteinander verbundenen Servern, die gemeinsam eine Menge physikalischer Disks verwalten. Die physikalischen Platten koennen dabei (serveruebergreifend) aehnlich wie bei einem RAID-Array zu

einer grossen virtuellen Platte zusammengelegt werden, und/oder zur Erhoehung der Ausfallsicherheit/Lesegeschwindigkeit gespiegelt werden. Die Clients (in Form eines Device-Treibers, der den Clientsystemen die virtuellen Platten zur Verfuegung stellt) verteilen ihre Zugriffe auf die beteiligten Server. Die Kommunikation laeuft ueber UDP und ist fuer das Sysyem transparent.
Das System kann sogar geographisch verteilt werden, um die Ausfallsicherheit zu erhoehen, allerdings steigen damit natuerlich die Uebertragungszeiten, und die Performance sinkt. Im LAN soll die Performance mit einer lokalen Platte vergleichbar sein.
Ein “Global State Modul” verwaltet die Informationen ueber beteiligte Server und vorhandene Festplatten mit Hilfe eines Algorithmus auf Basis von Paxos.

3.2 Disk-Paxos

Disk-Paxos ist eine Variante von Paxos. Er dient zur Implementierung eines fehlertoleranten Systems von Prozessoren und Laufwerken.

Dazu wird das System als deterministischer endlicher Automat modelliert, der eine Folge von Befehlen abarbeitet. Damit ist das Problem darauf reduziert, dass die Prozesse sich darauf einigen muessen, welche Befehle in welcher Reihenfolge abgearbeitet werden, und das leistet eben PAXOS. Jeder Prozessor startet mit einem Anfangswert input[p] und alle Prozessoren geben den gleichen Wert aus, naemlich irgendeinen der Anfangswerte.

Jedem Prozessor wird ein Block auf jeder der Platten zugeteilt. Jeder Prozessor fuehrt eine Reihe von Nummerierten Abstimmungen durch. Der Algorithmus hat zwei Phasen: In der ersten Phase wird in Wert v ausgewaehlt, in der zweiten Phase wird versucht diesen Wert zu commiten. In jeder Phase bricht der Prozessor seine Abstimmung ab, wenn er erfaehrt dass ein anderer Prozessor eine Abstimmung begonnen hat. In diesem Fall kann er dann eine neue hoehere Abstimmungsnummer auswaehlen und es erneut versuchen. Wenn der Prozessor Phase 2 ohne abzubrechen vollendet hat, dann ist der Wert v commited, und der Prozessor kann ihn ausgeben. Wenn ein Prozessor erfolgreich einen Wert commited, wird er auf seinem disk block ablegen dass der Wert commited wurde, und diese Tatsache allen anderen Prozessoren mitteilen. Wenn ein Prozessor erfaehrt dass ein Wert commited wurde, wird er seine Abstimmung abbrechen und den Wert einfach ausgeben.
Um den Algorithmus auszufuehren, verwaltet ein Prozessor p eine Datenstruktur dblock[p], die die folgenden drei Komponenten enthaelt:

mbal	die momentane Abstimmungsnummer
bal	Die groesste Abstimmungsnummer fuer die p Phase 2 erreicht hat
inp	Der Wert den p in Abstimmungsnummer bal versucht hat zu commiten

Anfangs ist bal 0, und inp auf einen speziellen Wert gesetzt, der “keinWert” bedeutet.

disk[d][p] ist der Block auf Disk d auf die der Prozessor p dblock[p] schreibt. Wir nehmen an dass Lesen und Schreiben eines Blocks atomare Operationen sind.

In jeder Phase versucht ein Prozessor zuerst dblock[p] auf disk[d][p] zu schreiben, und dann disk[d][q] fuer alle anderen Prozesse q zu lesen. Er bricht seine Abstimmung ab, falls er irgendeinen Block findet mit einem hoeheren Wert fuer mbal als sein eigener. Die Phase ist abgeschlossen wenn er eine Mehrzahl von Disks geschrieben und gelesen hat, ohne einen hoeheren Wert zu finden. Wenn Phase 1 abgeschlossen ist, waehlt p einen neuen Wert fuer dblock[p].inp aus, setzt dblock[p].bal auf dblock[p].mbal und beginnt Phase 2. Wenn Phase 2 abgeschlossen ist, ist

dblock[p].inp commited. Den Wert zu Beginn der Phase 2 waehlt der Prozessor aus, indem er alle in Phase 1 gelesenen Diskblocks anschaut. Enthalten alle “keinWert”, so setzt er dblock[p].inp auf seinen eigenen Anfangswert input[p]. Andernfalls setzt er dblock[p].inp auf den Wert aus den gelesenen Bloecken, der den hoechsten Wert fuer bal hatte.
Nur die letzte Phase muss fuer jeden Befehl erneut ausgefuehrt werden.
Im Normalfall gibt es nur einen Anfuehrer, die anderen Prozessoren warten nur auf das Ergebnis.

Literaturverzeichnis

LLHP. Erklärungen auf Leslie Lamports Homepage
http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos

LPTP. Leslie Lamport: The part time parliament
ACM Transactions on Computer Systems 16, 2, Seite 133-169, Mai 1998
Bereits 1989 geschrieben und 1990 eingereicht, aber erst 1998 wirklich veröffentlicht)

LPS. Leslie Lamport: Paxos made simple, 2001
http://research.microsoft.com/users/lamport/pubs/pubs.html#paxos-simple

GLDP. Eli Gafny, Leslie Lamport: Disk Paxos
Distributed Computing: 14th International Conference, DISC 2000

PETAL. Edward K.Lee and Chandramohan Thekkath: Petal - Distributed virtual disks.
In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pages 84 - 92, ACM Press, New York, October 1996