Key-Value-Stores
Am Beispiel von Scalaris

Natanael Arndt

arndtn@gmail.com

15. April 2012

Inhaltsverzeichnis

Keywords Key/Value Store, Scalaris, P2P

1 Einführung

Seit einigen Jahren haben special purpose Datenbanken unter dem Begriff NoSQL große Bedeutung erlangt. NoSQL Datenbanken sind vor allem für ihre liberalen Konzepte bekannte, etwa Schemafreiheit, einfache Replikation, simple Programmierschnittstellen, Eventual Consistency und die BASE Eigenschaften als Alternative zu ACID. Außerdem sind sie in der Lage sehr große Datenmengen zu verarbeiten[1]. Die ersten NoSQL Datenbanken sind zwar bereits im Jahr 1979 entstanden [EFH+11, Seite 1] aber erst mit dem zunehmenden Einsatz von Cloud Computing sind sie aufgrund der eben genannten Eigenschaften vermehrt in unser Blickfeld gerückt.

NoSQL Datenbanken können in die vier Kern-Kategorien Wide Column Stores, Document Stores, Graphdatenbanken und Key-Value-Stores eingeteilt werden. Es gibt aber auch weitere Kategorien, wie Objektdatenbanken, XML-Datenbanken, Grid-Datenbanken und viele andere. [EFH+11, Seite 6] In dieser Arbeit wird besonders auf Key-Values-Stores eingegangen.

Schon im Namen grenzen sich NoSQL Datenbanken von herkömmlichen relationalen Datenbanken ab. Diese Unterschiede werden besonders deutlich im Umgang mit der Konsistenz. Aber auch alt eingesessene Hersteller folgen diesem neuen Trend, wie z. B. Oracle mit der „Oracle NoSQL Database“[2]. Einige verstehen NoSQL auch nicht als vollkommene Abgrenzung bisheriger Technologien und interpretieren den Namen als „not only SQL“ [EFH+11, Seite XIII].

1.1 Key-Value-Stores

Key-Value- oder auch Tuple-Stores stellen eine einfache Struktur aus Schlüssel-Wert-Paaren (vergleichbar mit assoziativen Arrays) zu Verfügung. Die Schlüssel können dabei in Namensräume oder separate Datenbanken aufgeteilt werden – dies nutzt zum Beispiel das weiter unten vorgestellte Scalaris für die verteilte Speicherung. Die Werte können nicht nur einfache Zeichenketten sondern oft auch komplexere Datentypen enthalten [EFH+11, Seite 7]. Dabei wird kein Schema definiert, sondern der Datentyp kann frei gewählt werden. Im Gegensatz zu den anderen NoSQL-Kernsystemen, Document-Stores und Wide-Column-Stores, wird bei Key-Value-Stores lediglich ein Index über die Schlüssel erstellt. Durch diese simple Struktur und den eingeschränkten Funktionsumfang ist eine sehr effiziente Verarbeitung der Daten möglich. Jedoch schränkt dies auch die mögliche Komplexität von Anfragen ein, so dass Entwickler auf den Funktionsumfang der Programmierschnittstellen angewiesen sind. [EFH+11, Seite 7]

[1] Von `http://nosql-database.org/` - Stand 15. April 2012

[2] Von `http://www.oracle.com/technetwork/products/nosqldb/overview/index.html` - Stand 15. April 2012

In dieser Arbeit wird zur Demonstration von Key-Value-Stores das Beispiel Scalaris herangezogen und detailliert beschrieben. Scalaris ist ein verteilter Key-Value-Store der im Gegensatz zu anderen ebenfalls verteilten Systeme, wie Amazons Dynamo bzw. SimpleDB eine *Strong Consistency* bereitstellt und damit alle ACID Eigenschaften erfüllt [SSR08a].

1.2 CRUD Operationen statt SQL

Für NoSQL Datenbanken hat sich noch keine allgemeine Anfragesprache, wie SQL für RDBMS (Relational Database Management Systems), durchgesetzt. Um dennoch die grundlegenden Operationen durchführen zu können haben sich die vier CRUD-Operationen etabliert, die von allen Key-Value-Stores unterstützt werden. CRUD steht für **C**reate, **R**ead, **U**pdate und **D**elete. Eine Gegenüberstellung der vier Operationen zu den entsprechenden Operationen in SQL wird in Tabelle 1 dargestellt.

Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

Tabelle 1: Gegenüberstellung von CRUD mit SQL

Diese Operationen werden in den NoSQL Systemen meistens in Form eines *Application Programming Interface* (API), einer HTTP-Schnittstelle oder einer eigenen Anfragesprache zur Verfügung gestellt. Die Operation *Create* wird verwendet, um neue Datensätze anzulegen. Diese Datensätze können mit *Read* gelesen, mit *Update* geändert werden und schließlich mit *Delete* gelöscht werden.

Scalaris bietet APIs für Erland, Java, Python und Ruby, sowie eine generische JSON Schnittstelle über HTTP an. Die Create-, Read- und Update-Operationen sind auf dem Transaction Layer (siehe Kapitel 2) implementiert. Die Create- und Update-Operationen werden durch die Methode `write(Key, Value)` zur Verfügung gestellt. Die Read-Operation wird durch `read(Key)` implementiert. Durch Angabe eines *Transaction Logs* können die beiden Methoden `write(TLog, Key, Value)` und `read(TLog, Key)` kombiniert werden, um komplexe atomare Transaktionen durchzuführen. Um Einträge bzw. Schlüssel zu löschen muss die Methode `delete(Key)` oder `delete(Key, Timeout)` auf dem *Replication Layer* aufgerufen werden, sie kann allerdings keinen Erfolg garantieren, da zum Löschen alle Replikate (eine Mehrheit genügt nicht) verfügbar sein müssen. Eine vollständige API-Beschreibung ist im „Scalaris: Users and Developers Guide“ [sca12, Seite 17] verfügbar.

1.3 BASE als Alternative zu den ACID Eigenschaften

Bei NoSQL Datenbanken stehen meist hohe Verfügbarkeit und Verteilbarkeit, woraus eine höhere Wahrscheinlichkeit für Partitionierung folgt, im Vordergrund. Aufgrund des CAP-Theorems von Brewer muss die Bevorzugung der beiden Eigenschaften (Verfügbarkeit und Partitionstolleranz) zulasten der Konsistenz geschehen [Bre00]. Dies erfordert einen anderen Umgang mit Konsistenz als im Bereich der relationalen Datenbanken (RDBMS) üblich. Als Alternative zu den in RDBMS verwendeten ACID Eigenschaften (atomicity, consistency, isolation, durability) schlägt daher Brewer die BASE Eigenschaften, **B**asically **A**vailable **S**oft-state **E**ventual consistency, vor [Bre00]. Dabei wird die Verfügbarkeit der Konsistenz untergeordnet [EFH+11]. Daraus resultiert eine schwächere Konsistenz, BASE verfolgt also einen optimistischeren Konsistenzansatz als ACID.

Strong Consistency Gilbert und Lynch definieren *atomic* oder auch *linearizable consistency* so, dass alle Operationen durch eine lineare Ordnung (Totalordnung) geordnet sind. Auf diese Weise erscheinen sie für eine Anwendung, als wären sie auf einem einzigen Knoten ausgeführt worden (Verteiltransparentz) [GL02]. Dies entspricht der *Strict Consistency* wie sie in Scalaris implementiert ist: „Any read operation has to return the result of the latest write operation on the same data item.“ [sca12, Seite 7]

Weaker/Eventual Consistency Diese Strong-Consistency-Bedingung wird für „Weaker Consistency Conditions“ aufgeweicht, indem nur noch eine Halbordnung t-*Connected* der Lese- und Schreib-Operationen vorausgesetzt wird. Die Operationen sind unter folgenden Bedingungen t-*Connected*:

1. Sofern alle Nachrichten fehlerfrei übertragen werden sind alle Operationen atomar (*atomic*) konsistent.

2. Ist eine fehlerfreie Nachrichtenübertragung nicht möglich sind die Operationen gemäß der Halbordnung P aus Definition 1 geordnet.

Auf diese Weise wird eine Zeitobergrenze festgelegt nach der die Konsistenz hergestellt ist.

Definition 1 *Es gibt einen zentralen Knoten an den alle anderen Knoten Anfragen stellen. Die Lese- und Schreib-Operationen sind sortiert.*

Wird ein Knoten angefragt, stellt er eine Anfrage an den zentralen Knoten, bekommt er nach einer gewissen Zeit keine Antwort, beantwortet er die an ihn gestellte Anfrage auf Basis der lokal vorhandenen, vorher ausgeführten, Operationen. Das Ergebnis ist dann mit den lokal ausgeführten Operationen konsistent.

Nach einem Zeit-Intervall δ in dem keine Nachrichten verloren gehen, sind alle Operationen a, die vor δ ausgeführt wurden, in P Vorgänger der Operationen b, die nach δ ausgeführt wurden (also $a <_P b$). [GL02]

Viele NoSQL Datenbanken (z. B. Dynamo/SimpleDB und Riak) garantieren nur eine Eventual Consistency, Scalaris hingegen ermöglicht eine strenge Konsistenz (*strict consistency* oder *strong consistency*) bei gleichzeitiger Toleranz gegenüber Partitionierung wodurch die Verfügbarkeit den anderen Eigenschaften zum Opfer fällt, da Transaktionen immer nur in der Mehrheitspartition mit mindestens $\lceil (r-1)/2 \rceil$ Knoten ausgeführt werden können. [sca12, Seite 7]

2 Überblick über Scalaris

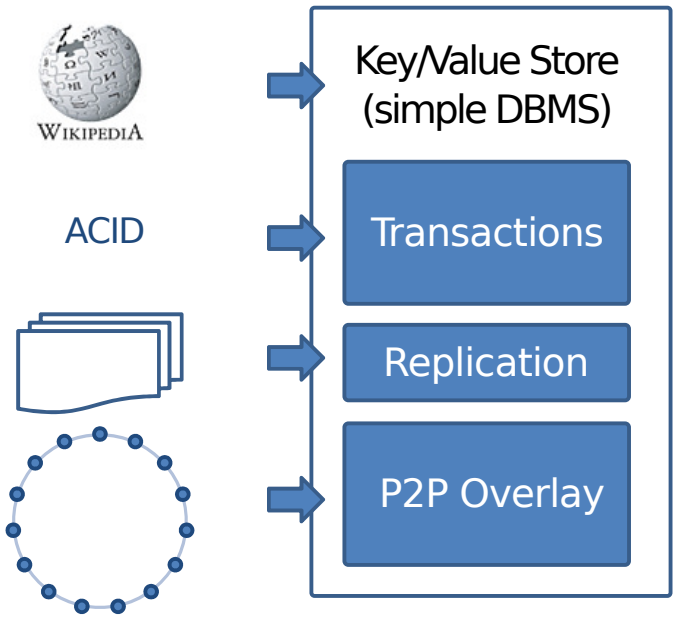

Abbildung 1: Aufbau von Scalaris [SSR08b]

Scalaris ist aus drei Schichten aufgebaut P2P Overlay mit Chord$^{\#}$, Replikationsschicht und Transaktionsschicht. Das Schichtenmodell wird in Abbildung 1 dargestellt. Die oberste Schicht repräsentiert Anwendungen, die Scalaris nutzen, um ihre Daten abzulegen, hier beispielhaft eine Wikipedia Implementierung.

Das P2P Overlay bildet eine verteilte Hash-Tabelle (distributed hash table, DHT) mittels Chord$^{\#}$ in der die Einträge alphabetisch nach ihren Schlüsseln sortiert abgelegt werden. Die Replikationsschicht repliziert alle Einträge (Schlüssel-Wert-Paare) auf mehrere Knoten, um Ausfallsicherheit zu gewährleisten. In der Transaktionsschicht werden durch ein Paxos-Consensus-Protokoll die ACID-Eigenschaften sichergestellt.

3 P2P Overlay mit Chord$^{\#}$

Auf unterster Ebene verwendet Scalaris Chord$^{\#}$ als Peer-To-Peer-Overlay zur Speicherung der Schlüssel-Wert-Paare und um sie effizient abzurufen [SSR08a]. Chord$^{\#}$ ist eine verteilte Hash-Tabelle, auf die mit einem logarithmischen Aufwand zugegriffen werden kann [SSR06]. Die Schlüssel werden alphabetisch sortiert abgelegt, was Range-Queries ermöglicht. Im Gegensatz zu Chord wird bei Chord$^{\#}$ aber nicht auf dem Schlüsselraum (*key space*) sondern auf den Knoten (*node space*) geroutet [SSR06, SSR08a]. Dies stellt eine logarithmische Komplexität des Routings sicher, welche Schütt et. al. [SSR06] bewiesen haben.

Die Knoten sind ähnlich wie in anderen DHTs in einer logischen Ringstruktur angeordnet [SSR06]. Die Routingtabelle eines Knotens besteht aus Fingern welche auf andere Knoten, in logarithmisch wachsendem Abstand, zeigen. Die Routingtabelle wird nach Formel 1[3] rekursiv erstellt.

$$finger_i = \begin{cases} nachfolger & : i = 0 \\ finger_{i-1}.finger_{i-1} & : i \neq 0 \end{cases} \tag{1}$$

Dabei setzt jeder Knoten seinen Nachfolger an Position $i = 0$. Um jeden weiteren Finger (Stelle i) zu berechnen fragt er den Knoten der bei ihm an Position $i - 1$ steht nach dessen Eintrag an seiner Position $i - 1$. Auf diese Weise entsteht eine Routingtabelle mit logarithmisch wachsendem Abstand der Finger [SSR06, SSR08a].

[3] Der Operator $x.y$ fragt von Knoten x den Eintrag an Stelle y in seiner Routingtabelle ab

4 Replikations- und Transaktionsschicht

Um tolerant gegenüber Knotenausfällen zu sein, bietet Scalaris die Möglichkeit die Daten auf mehrere (r) Knoten zu verteilen. Lese- und Schreiboperationen werden dabei jeweils auf der Mehrheit der Knoten ausgeführt, um den Ausfall von bis zu $\lfloor (r-1)/2 \rfloor$ Knoten zu tolerieren [SSR08a]. Demzufolge ist Scalaris bei einer Partitionierung nur noch im größeren Teil verfügbar, sofern er über die Hälfte der Knoten enthält. Andernfalls ist die Partition nicht mehr entscheidungsfähig. Bei Ausfällen oder gewollter Abschaltung einzelner Knoten kann das System neu konfiguriert werden und r angepasst werden. [SRHS10]

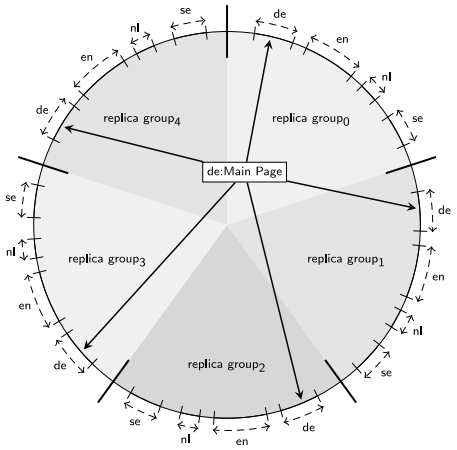

Abbildung 2: Darstellung der Verteilung eines Schlüssels auf mehrere Replikate, welche auf unterschiedlichen Knoten liegen [SRHS10]

Um die Schlüssel auf mehrere Knoten zu verteilen verwendet Scalaris eine symmetrische Replikation, wie sie von Ghodsi et. al. [GAH07] beschrieben wird. Dazu werden den Schlüsseln numerische Präfixe gegeben anhand derer sie zu einem Replikat zugeordnet werden können [SSR08a]. Da in Scalaris jedem Knoten ein Start-Schlüssel zugewiesen wird, kann so die Verteilung der Replikate klar geregelt werden, um z. B. eventuellen geographischen Anforderungen zu entsprechen. Die Verteilung eines Schlüssels „de:Main Page“ auf fünf Replikate wird in Abbildung 2 schematisch dargestellt.

Zur Transaktionsverwaltung setzt Scalaris ein optimistisches Verfahren ein. Dabei werden die Änderungen zuerst in einer Lese-Phase (*read-phase*) lokal vorgenommen. Anschließend werden diese Änderungen an die zuständigen Knoten übermittelt, die ebenfalls Replikate dieses Datensatzes halten. Hat eine Mehrheit von Knoten diesen Datensatz bereits für Änderungen gesperrt oder wurde er zwischenzeitlich geändert muss die lokale Änderung wieder rückgängig gemacht werden. Andernfalls wird die Änderung mithilfe eines Paxos-Protokolls (siehe unten) an die anderen Replikate übermittelt. Je-

dem Eintrag (Schlüssel-Wert-Paar) wird eine Versionsnummer zugeordnet. Zuerst wird in einem Durchlauf des Paxos-Consensus-Protokolls (siehe Abschnitt 4.1) die aktuell höchste Versionsnummer „gelesen“. Anschließend wird in einem Paxos-Commit die um eins erhöhte Versionsnummer geschrieben. Es müssen genauso wie bei einem Majority Voting immer $\lceil (r-1)/2 \rceil$ Knoten abgefragt werden, um die aktuelle Version zu lesen. [SRHS10]

4.1 Das Paxos-Protokoll

Um fehlertolerante Transaktionen auf den Replikaten zu ermöglichen setzt Scalaris ein angepasstes Paxos-Consensus-Protokoll ein [SSR08a]. Im Gegensatz zu einem 2-Phasen-Commit-Protokoll blockiert es nicht beim Ausfall einzelner Knoten. Das Paxos-Consensus-Protokoll ist vergleichbar mit einem 3-Phasen-Commit-Protokoll nutzt aber mehrere Acceptors, um nicht auf einen einzelnen Transaction Manager angewiesen zu sein [SSR08a],[EFH+11, Seite 47].

In einem Paxos-Consensus-Protokoll gibt es folgende Rollen, die unter den Teilnehmern aufgeteilt werden:

Proposer Gibt einen Wert vor, der von den anderen Teilnehmern bestätigt werden soll

Acceptor Nimmt den Wert an und überprüft ob es sich um den aktuell gültigen Wert handelt

Learner Sammelt die Bestätigung von einer Mehrheit der *Acceptors* ein und sendet die endgültige Entscheidung an alle Teilnehmer weiter

Diese Rollen werden von den folgenden Teilnehmern übernommen:

Transaction Manager (TM) Ist ein *Acceptor*, der ebenfalls die koordinierende Aufgabe des *Learners* erfüllt

Replicated Transaction Manager (RTM) Weitere *Acceptor*, die über die Gültigkeit des Wertes entscheiden. Bei einem Ausfall des Transaction Managers übernimmt einer der Replicated Transaction Manager dessen Rolle.

Transaction Participant (TP) Bringt als *Proposer* einen Wert ein und erhält am Ende das Ergebnis der Entscheidung

Ablauf des Konsens-Protokolls

Ein *Proposer* startet das Konsens-Protokoll indem er einen Wert, der bestätigt werden soll, zusammen mit einer Rundennummer r an die *Acceptors* sendet. Die *Acceptors* reagieren auf die Anfrage des *Proposers* indem sie den letzten Wert, den sie kennen und über den ein Konsens erreicht wurde, zurückgeben. Sobald der *Proposer* die Antworten von einer Mehrheit der *Acceptors* erhalten hat ermittelt er daraus den neusten Wert und gibt diesen wieder an alle *Acceptors* zurück. Die *Acceptors* senden diesen Wert als *accepted* an den *Learner*. Der *Learner* wartet bis er von einer Mehrheit die *accepted*-Meldung bekommt und gibt den Wert dann als Entscheidung an alle bekannt. (Vgl. Algorithm 1-3 in [SRHS10].) Das Konsens-Protokoll kann in zwei Phasen unterteilt werden. In der ersten Phase werden die nötigen Informationen gesammelt, um zu entscheiden ob der Wert bestätigt werden kann. In der zweiten Phase wird der Konsens allen interessierten Teilnehmern mitgeteilt.

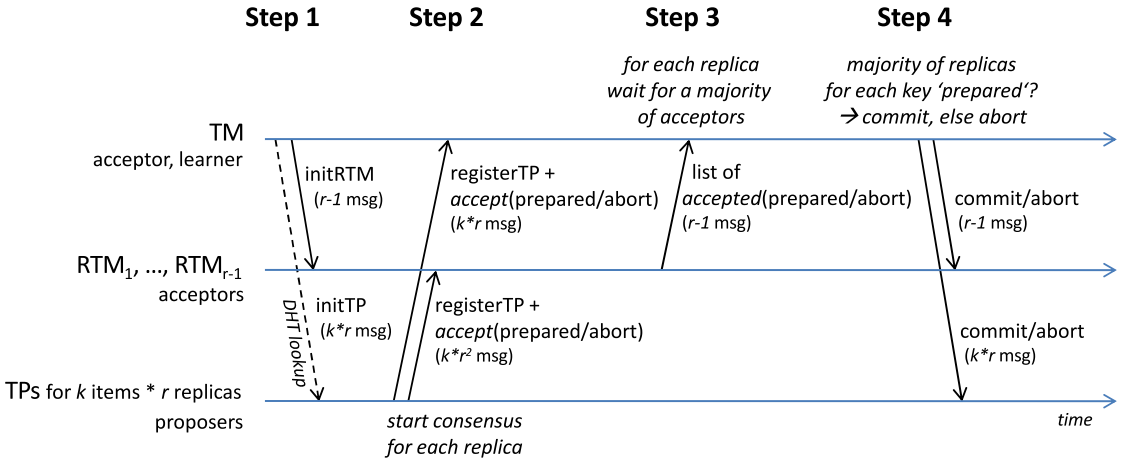

Abbildung 3: Ablauf des Scalaris Commit-Prozesses [SRHS10]

Ablauf des Commit-Protokolls

Ein abgewandelter Paxos-Commit nach Gray und Lamport [GL06] läuft wie folgt ab: Der *Transaction Manager* fragt die *Transaction Participants*, die eine anstehende Transaktion eintragen möchten, ob sie bereit sind. Diese starten daraufhin einen neuen Paxos-Consensus indem sie ihre Antwort, *accept(prepared)* oder *accept(abort)*, an alle *Acceptors* senden. Bei diesem Durchlauf wird allerdings die erste Phase (die Lese-Phase) ausgelassen, man spricht dann auch von einem *Fast Paxos Consensus*. Hat der *Transaction Participant* sich für *prepared* entschieden sperrt er den entsprechenden Datensatz. Die *Acceptors* senden ihre *accepted* Nachrichten an den *Transaction Manager/Learner*, wel-

cher die Antworten zusammenfasst und an alle *Transaction Participants* zur Ausführung gibt. Dieser Vorgang wird in Abbildung 3 dargestellt. [GL06, SRHS10]

5 Implementation

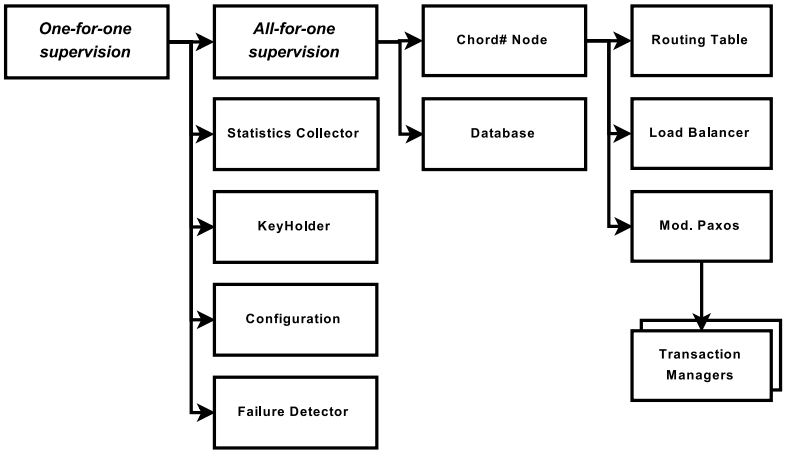

Abbildung 4: Prozessübersicht von Scalaris [SSR08a]

Scalaris, Chord$^{\#}$ und der Paxos-Algorithmus wurden in Erlang unter Verwendung des Actor-Models implementiert. Dabei gibt es *Actors* und *Messages*, die zwischen *Actors* ausgetauscht werden. Dies ermöglicht eine einfache Umsetzung der theoretischen Struktur in den Code.

Die einzelnen Prozesse sind in einer Baumstruktur, wie sie in Abbildung 4 dargestellt wird, angeordnet. Ein Supervisor-Knoten kann dabei bei Gelegenheit seine untergeordneten Knoten beenden, wodurch die Konsistenz sichergestellt wird. Der One-for-one-Supervisor startet seine direkt untergeordneten Prozesse im Fehlerfall jeweils einzeln neu, da sie keine für die Konsistenz relevanten Aufgaben haben. Der All-for-one-Supervisor hingegen beendet beim Ausfall eines Unterprozesses alle anderen Unterprozesse, um sie gemeinsam neuzustarten. Dies ist besonders für die Datenbank und den Chord$^{\#}$-Prozess wichtig, um die Konsistenz zu wahren. Diese Struktur ist in jedem Knoten vorhanden. [SSR08a]

Transaktionen

Scalaris unterstützt Transaktionen, um mehrere Operationen atomar ausführen zu können. Transaktionen werden in zwei Phasen ausgeführt, einer Lese- und einer Commit-

Phase. Für die Lesephase wird ein λ-Ausdruck, der alle Operationen enthält, verwendet. Dieser wird an die Transaction-API weitergegeben, wo er ausgeführt wird. Sind weiterhin alle ACID Eigenschaften erfüllt, so wird die Commit-Phase ausgeführt. Transaktionen werden asynchron ausgeführt, um den ausführenden Prozess nicht zu blockieren. Bei der Ausführung des Commits besteht die Möglichkeit über eine `SuccessFun` und eine `FailureFun` Einfluss auf den Rückgabewert der Transaktion zu nehmen. [SSR08a]

Parallel lesen und schreiben

Um eine höhere Erreichbarkeit zu erzielen können Lese- und Schreib-Operationen parallel ausgeführt werden, ohne dabei die Atomarität und Konsistenz der Daten zu gefährden. Dabei greifen Lese-Operationen noch auf den alten Stand der Daten zu, bevor eine Schreib-Operation vollständig ausgeführt wird. Dies wird durch eine Schleife (`asyncLoop`) erreicht, die alle Anfragen entgegennimmt. Lese-Operationen werden sofort ausgeführt und das Ergebnis zurückgegeben. Schreib-Operationen hingegen werden an eine andere Schleife (`syncLoop`) weitergegeben, um dort sukzessive ausgeführt zu werden. Ist eine Schreib-Operationen beendet wird über eine Nachricht `updatestate` der neue Zustand an die `asyncLoop` übermittelt. [SSR08a]

6 Evaluation

Zur Evaluation der Leistungsfähigkeit von Scalaris wurde eine Kopie der Wikipedia, unter Verwendung von Scalaris als Datenbank, implementiert. Der Wiki-Text wurde mit plog4u gerendert und durch einen jetty Webserver ausgeliefert. Durch einen Load-Generator (siege) wurden zufällige Seiten von einem Load-Balancer (haproxy) abgerufen, der die Anfragen gleichmäßig auf 16 Server verteilt hat. Mit diesem Aufbau konnten 2500 Transaktionen pro Sekunde durch lediglich 16 Server bearbeitet werden. Im Gegensatz dazu bearbeitet die „echte“ Wikipedia 45.000 Anfragen pro Sekunde, wovon aber nur 2000 Anfragen pro Sekunde auf das Backend mit etwa 200 Servern durchgereicht werden.

Atomicity, Consistency, Isolation und Durability

Die **Atomicity** wird durch die Verwendung von *Transactionlogs* sichergestellt. Eine Konsistenz im Sinne von Fremdschlüsselbeziehungen gibt es in Scalaris nicht. Jedoch wird die **Konsistenz** zwischen den einzelnen Replikaten, ebenso wie die **Isolation** der einzelnen Datensätze, durch die Verwendung das Paxos-Protokolls und das *Locking* der Datensätze

gewährleistet. Die **Durability** wird durch die Replikation auf mehrere Knoten sichergestellt, eine dauerhafte Speicherung ist aber z. B. durch die Verwendung von DETS oder Mnesia möglich.

7 Zusammenfassung

Diese Arbeit gibt eine Einführung in NoSQL Datenbanken und insbesondere Key-Value-Stores. Dabei werden grundlegende Konzepte wie CRUD und BASE vorgestellt und Ansätze von Weak- bzw. Eventual-Consitency einer Strong Consistency gegenübergestellt. Als Beispiel eines Key-Value-Stores wird Scalaris näher betrachtet und es wird auf dessen Besonderheiten gegenüber anderen Datenbanken dieser Kategorie eingegangen.

Nach einem kleinen Überblick über den Aufbau von Scalaris wird die Routingstruktur der verteilten Hash-Tabelle (DHT) Chord$^{\#}$ vorgestellt und die Funktionsweise des in Scalaris verwendeten angepassten Paxos-Consensus-Protokolls näher erläutert. Durch die Verwendung des Paxos-Concensus-Protokolls und eines Paxos Commit-Verfahrens ist Scalaris immer nur erreichbar, solange eine Mehrheit, bestehend aus mindestens $\lceil (r-1)/2 \rceil$ Knoten, erreichbar ist. Dies ist zum Beispiel im Fall einer Partitionierung in den Minderheits-Partitionen (Partitionen mit weniger als $\lceil (r-1)/2 \rceil$ Knoten) nicht möglich. Im Gegensatz zu anderen NoSQL Datenbanken ist Scalaris daher nicht immer erreichbar. Dies geschieht aber zugunsten einer Strong Consistency und der übrigen ACID Eigenschaften.

Später wird die Implementierung von Scalaris gezeigt und Konzepte zur Transaktionsausführung und zum parallelen Lesen und Schreiben vorgestellt. Zum Schluss wird die Leistung von Scalaris als Datanbank hinter einer großen Webanwendung gezeigt und es werden die einzelnen Aspekte der ACID Eigenschaften und ihre Umsetzung in Scalaris dargestellt.

Literatur

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Principles of Distributed Computing*, PODC '00, Juli 2000. (Keynote).

[EFH+11] Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer und Markus Brückner. *NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Hanser, 2011.

[GAH07] Ali Ghodsi, Luc Alima und Seif Haridi. Symmetric Replication for Structured

Peer-to-Peer Systems. In Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin und Aris Ouksel, Hrsg., *Databases, Information Systems, and Peer-to-Peer Computing*, Jgg. 4125 of *Lecture Notes in Computer Science*, Seiten 74–85. Springer, 2007.

[GL02] Seth Gilbert und Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web. *SIGACT News*, 33:51–59, Juni 2002.

[GL06] Jim Gray und Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Marz 2006.

[sca12] *Scalaris: Users and Developers Guide*. Number 0.4.0+svn. Zuse Institute Berlin and onScale solutions., 2012.

[SRHS10] Florian Schintke, Alexander Reinefeld, Seif Haridi und Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. In *CCGRID*, Seiten 448 – 454, 2010.

[SSR06] Thorsten Schütt, Florian Schintke und Alexander Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. In *GP2PC'06*, Mai 2006.

[SSR08a] Thorsten Schütt, Florian Schintke und Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, Victoria, British Columbia, Canada, 2008.

[SSR08b] Thorsten Schütt, Florian Schintke und Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. In *Seventh ACM SIGPLAN Erlang Workshop*, 2008. `http://ftp.sunet.se/pub/lang/erlang/workshop/2008/`.