Transaktionsmanagement für Cloud Speicherdienste am Beispiel von CloudTPS und ElasTraS

Martin Junghanns
martin.junghanns@studserv.uni-leipzig.de

7. März 2012

Zusammenfassung

In dieser Seminararbeit werden die Transaktionssysteme CloudTPS und ElasTraS konzeptionell vorgestellt und hinsichtlich ihrer Architektur, der Erfüllung der ACID Anforderungen in einer verteilten Umgebung und ihrem Verhalten im Fehlerfall verglichen. Es handelt sich um zwei Systeme, deren Zielstellung es ist, ACID konforme Transaktionen mit den wesentlichen Eigenschaften einer Cloud Infrastruktur zu vereinen. CloudTPS setzt dabei auf bestehende Cloud Datenbanken wie BigTable oder HBase auf, wohingegen ElasTraS eine eigenständige Datenbank auf Basis eines verteilten, fehlertoleranten Dateisystems darstellt. Es wird sich zeigen, dass aufgrund des CAP-Theorems Einschränkungen in der Verfügbarkeit der Systeme toleriert werden müssen, um Ausfallsicherheit und Konsistenz zu erreichen.

1 Einleitung

Cloud Computing bietet mit Infrastructure-as-a-Service (IaaS) den Anbietern verteilter Anwendungen die Möglichkeit, auf eine theoretisch unbegrenzte Menge an Rechen-, Speicher- und Netzwerkressourcen zuzugreifen. Es müssen keine eigenen Rechenzentren mehr administriert werden, die benötigten Infrastrukturen werden bei einem IaaS Anbieter bedarfsgerecht in Anspruch genommen. Durch die Elastizität der Infrastruktur sind die Kosten für den Nutzer variabel [1]. Skalierbarkeit, Verfügbarkeit und Fehlertoleranz zählen zu den Anforderungen an verteilte Datenbankmanagementsysteme (DBMS) [2]. Sie werden durch die neue Situation um *Elastizität* und *Selbstverwaltung* ergänzt.

Relationale Datenbankmanagementsysteme (RDBMS) wie MySQL[1] oder Oracle[2] können in einer Cloud Umgebung gehostet und verwendet werden. Sie bieten in verteilten Szenarien den gewohnten Funktionsumfang: die Verwendung von SQL als deklarative Anfragesprache, statische Partitionierung zur horizontalen oder vertikalen Verteilung der Daten, komplexe Bereichs- und Verbundanfragen über mehreren Partitionen und Replikation zur Verbesserung der Performance von Lesezugriffen und zur Gewährleistung von Fehlertoleranz. Ein wesentlicher Nachteil von RDBMS ist die statische Partitionierung der Relationen [3],[4]. Beim Einsatz einer Bereichsfragmentierung sind die Grenzen einzelner Bereiche durch das Fragmentierungsattribut fest definiert, weswegen Änderungen der Bereiche mit administrativem Aufwand verbunden sind. Dynamische Partitionierung ist jedoch die Grundvoraussetzung für Skalierbarkeit und Elastizität innerhalb der Cloud Infrastruktur. Darüber hinaus schränken partitionsübergreifende, schreibende Transaktio-

[1] http://www.mysql.com/
[2] http://www.oracle.com/

nen bei gleichzeitiger Einhaltung der ACID Anforderungen die Verfügbarkeit und somit die Skalierbarkeit des Systems ein [5].

Cloud DBMS wie Google BigTable [6], Amazon SimpleDB [7] oder Apache Cassandra [8] basieren auf einem wesentlich einfacheren Datenmodell: Attribut-Wert Paaren. Diese Paare sind die kleinste verwaltbare Einheit im Sinne der Partitionierung. Der Vorteil dieses vereinfachten Datenmodells ist die Fähigkeit zur effizienten, dynamischen Partitionierung der Daten. Darüber hinaus bieten die genannten Systeme verschiedene Formen eines abgeschwächten Konsistenzmodells, der *Eventual Consistency*, an. Dieses Modell führt dazu, dass Änderungen nach Beendigung einer Transaktion nicht zwingend sichtbar sind, sondern erst nach einer nicht-definierten Zeit der Inkonsistenz. Durch diese Auflockerung erreichen Cloud Datenbanken Skalierbarkeit. Aus den genannten Vorteilen ergeben sich jedoch auch Nachteile: Abfragen sind nur unter Angabe der Primärschlüssel möglich, komplexe Bereichs- oder Verbundabfragen können nur innerhalb einer Partition ausgeführt werden. Anwendungen, welche eine konsistente Sicht auf die Daten erfordern, sind bedingt durch die Eventual Consistency praktisch nicht umsetzbar.

Transaktionsmanagementsysteme erweitern Cloud Speicherdienste um die Möglichkeit, ACID konforme Transaktionen über mehreren Datensätzen und Partitionen auszuführen. Mit CloudTPS [5] und ElasTraS [9] werden zwei Vertreter solcher Systeme vorgestellt und verglichen. CloudTPS setzt dabei auf die bestehenden Cloud Datenbanken auf, wohingegen ElasTraS eine eigenständige Datenbank auf Basis eines verteilten, fehlertoleranten Dateisystems innerhalb der Cloud Infrastruktur ist.

In Kapitel 2 werden die theoretischen Grundlagen dieser Seminararbeit dargelegt. In den Kapiteln 3 und 4 werden CloudTPS bzw. ElasTraS vorgestellt und anhand beider Konzepte die Erfüllung der ACID Forderungen in einem verteilten System erläutert. Dabei werden im Rahmen der Vorstellung von ElasTraS strukturelle Vergleiche mit CloudTPS angestellt. Kapitel 5 befasst sich mit den wesentlichen Unterschieden der beiden Systeme und zeigt deren jeweilige Vor- und Nachteile auf. Darüber hinaus dient es der Zusammenfassung der Arbeit.

2 Grundlagen

Im folgenden Abschnitt werden die grundlegenden Begriffe im Bereich Transaktionen und Transaktionsmanagement kurz erläutert.

2.1 Transaktion

Eine der wichtigsten Anforderungen an eine Datenbank ist das Unterstützen gleichzeitiger Zugriffe auf den verwalteten Datenbestand. Damit parallele Lese- und Schreibzugriffe auf die Datenbank ausgeführt werden können, wird jeder Verarbeitungsvorgang innerhalb der Datenbank in einer **Transaktion** gekapselt. Diese Überführung in eine elementare Ausführungseinheit garantiert, dass sich die Datenbank nach Beendigung oder Abbruch einer Transaktion in einem konsistenten Zustand befindet [10].

2.2 ACID

Die ACID-Eigenschaften sind Forderungen, deren Erfüllung dazu führt, dass Transaktionen korrekt und zuverlässig ausgeführt werden. Atomarität (engl. ***A****tomicity*) legt fest,

dass eine Transaktion entweder vollständig oder gar nicht ausgeführt wird. Zwischenergebnisse werden nicht gespeichert. Die Konsistenz (engl. ***C**onsistency*) bezeichnet das Prinzip der Erhaltung eines widerspruchsfreien Zustandes vor und nach einer Transaktion. Auch beim Abbruch einer Transaktion muss dieser Zustand wiederhergestellt werden. Isolation (engl. ***I**solation*) ist die Forderung nach dem Eindruck des alleinigen Zugriffs auf die Datenbank. Der Zugriff wird aus Sicht des Nutzers in einem ‚isolierten' Umfeld ausgeführt. Um Wartezeiten zu vermeiden, sollen Nutzertransaktionen nicht länger als nötig blockiert werden. Schlussendlich beschreibt das Prinzip der Dauerhaftigkeit (engl. ***D**urability*) die Anforderung einer fortwährenden Speicherung nach Abschluss einer Transaktion [11].

2.3 CAP-Theorem

Als eine Folge des stetigen Wachstums des World Wide Web und der damit verbundenen Probleme bzgl. der Skalierbarkeit verteilter Systeme, stellte Brewer in einer Keynote auf dem ACM-Symposium im Jahr 2000 eine These hinsichtlich des Zusammenhangs zwischen drei wesentlichen Kernkonzepten verteilter Systeme auf [12]. Er stellte fest, dass die Forderungen nach Konsistenz (engl. ***C**onsistency*), Verfügbarkeit (engl. ***A**vailability*) und Ausfalltoleranz (engl. ***P**artition Tolerance*) in einem verteilten System nicht gleichzeitig erfüllt werden können. Stattdessen können immer nur zwei der Zielsetzungen zu einem Zeitpunkt erreicht werden. Diese These wurde 2002 formal bewiesen und wird seitdem als Brewer's Theorem oder CAP-Theorem bezeichnet [13].

Konsistenz bedeutet, dass ein verteiltes System nach Beendigung einer Transaktion einen widerspruchsfreien Zustand erreicht haben muss. Eine schreibende Transaktion kann somit erst beendet werden, wenn die Änderung in allen Replikaten des betroffenen Datensatzes durchgeführt wurde. Für die Dauer dieser Operation ist der Datensatz für sämtliche Zugriffe gesperrt.

Verfügbarkeit definiert im Kontext des CAP-Theorems den Zustand eines verteilten Systems, in welchem jederzeit Lese- und Schreiboperationen in akzeptabler Zeit durchgeführt werden können. Welche Zeitspanne hierbei akzeptabel ist, definiert die jeweilige Anwendung.

Ausfalltoleranz eines verteilten Systems ist dann erreicht, wenn der Ausfall einzelner Knoten oder der Kommunikationswege zwischen den Knoten nicht zum Ausfall des gesamten Systems führt.

Durch das Theorem entsteht somit ein Zielkonflikt in der Konzeption verteilter Systeme. Bereits beim Entwurf muss festgelegt werden, welche Forderungen durch das System erfüllt werden müssen.

2.4 BASE

Einen möglichen Umgang mit dem genannten Zielkonflikt stellt das BASE-Paradigma [14] dar (engl. ***B**asically **A**vailable, **S**oft State and **E**ventually Consistent*), welches Skalierbarkeit und Verfügbarkeit der unmittelbaren Konsistenz der Daten vorzieht. Es ist somit eine Alternative zum ACID-Paradigma. In Zusammenhang mit BASE findet der Begriff der *Eventual Consistency* Anwendung [15]. Es handelt sich hierbei um ein Modell, bei welchem die Konsistenz erst nach Ablauf einer nicht definierten Zeitspanne (engl. *inconsistency time*) erreicht wird. Durch diese Auflockerung der Konsistenzforderung wird die Skalierbarkeit eines verteilten Systems unter Gewährleistung von Verfügbarkeit und Aus-

falltoleranz ermöglicht. Eventual Consistency steht dem Gegenentwurf der Strong Consistency gegenüber, die das Konzept der 1-Kopien-Serialisierbarkeit [2] umsetzt.

Die Paradigmen ACID und BASE verhalten sich in der Datenbankwelt antagonistisch. Relationale Datenbanken tendieren eher zu ACID, nicht-relationale Cloud Datenbanken hingegen zu BASE.

2.5 Eventual Consistency

Die von CloudTPS unterstützten Cloud Datenbanksysteme lassen sich anhand der jeweiligen Form der Eventual Consistency klassifizieren. Es werden nur diejenigen Formen betrachtet, die im Rahmen dieser Arbeit Verwendung finden, für weitere Informationen sei auf [15] verwiesen.

Read-your-writes Consistency legt fest, dass durch einen Prozess vorgenommene Änderungen an einem Datensatz in allen anschließenden Lesezugriffen des gleichen Prozesses sichtbar sind.

Monotonic Read Consistency definiert, dass in aufeinanderfolgenden Leseoperationen eines Datensatzes durch einen Prozess keine älteren Versionen des Datensatzes erhalten werden.

3 CloudTPS

Die Entwickler von CloudTPS verfolgen das Ziel, bestehende Cloud Datenbanksysteme um die Fähigkeit zur Ausführung von ACID-Transaktionen zu erweitern. Die Transaktionen beinhalten dabei wahlweise einzelne (engl. *single-item transactions*) oder mehrere Datensätze (engl. *multi-item transactions*). Die Einhaltung der Konsistenz wird auch bei Serverausfall oder Netzwerkpartitionierung garantiert. Es handelt sich um eine zusätzliche Schicht, welche zwischen der Applikation und der Speicherschicht eingesetzt wird.

Die Funktionsweise von CloudTPS ist das Aufteilen der Datenmenge auf lokale Transaktionsmanager (LTMs), die Teile der Daten im Hauptspeicher verwalten und die Forderung nach Konsistenz erfüllen. Durch die Verteilung der Transaktionen wird die Entstehung eines Bottlenecks durch einen zentralen Transaktionsmanager verhindert. Atomarität von Transaktionen wird anhand eines Zwei-Phasen-Commit Protokolls [2] (2PC) gewährleistet, resultierende Änderungen werden periodisch in das jeweilige Speichersystem zurückgeschrieben. Durch Replikation der Daten zwischen den LTMs wird dem Problem der Server- und Netzwerkausfälle begegnet.

CloudTPS wurde explizit für den Einsatz in Webapplikationen entwickelt. Das Konzept zur Umsetzung skalierbarer, effizienter Transaktionen stützt sich auf drei wesentliche Eigenschaften solcher Anwendungen: Transaktionen finden üblicherweise im Rahmen einer Nutzeranfrage statt, was zu einer signifikant kurzen Transaktionsdauer führt (engl. *short-lived transactions*). Darüber hinaus ist die Anzahl Datensätze, welche in eine Transaktion involviert sind, häufig sehr gering und die Datensätze selbst werden üblicherweise anhand ihrer Primärschlüssel identifiziert. Die dritte wesentliche Eigenschaft ist, dass Leseanfragen selbst dann gute Resultate liefern, wenn die Daten noch nicht konsistent bzw. nicht aktuell sind. CloudTPS wurde auf Basis der genannten Anforderungen entwickelt und zielt insbesondere auf den Einsatz in Bezahl- oder Auktionssystemen ab, bei denen Skalierbarkeit unter Einhaltung konsistenter Datenbestände im Vordergrund steht.

3.1 Datenmodell

CloudTPS hat zum Ziel, möglichst portabel bzw. universell auf verschiedenen Cloud Datenbanken einsetzbar zu sein. Diese Anforderung schränkt die Wahl des Datenmodells stark ein, die Entwickler entschieden sich für ein Modell auf Basis von Schlüssel-Wert Paaren (engl. *key-value pairs*). Der Zugriff auf die Daten erfolgt über einfache GET/PUT Anweisungen auf Basis des Primärschlüssels. Die Primärschlüssel aller an einer Transaktion beteiligten Datensätze müssen vor Beginn der Transaktion bekannt sein. Diese Forderung wird durch die o.g. Eigenschaft von Webanwendungen erfüllt.

Ein weiteres Unterscheidungskriterium aktueller Cloud Datenbanken ist die jeweilige Form der Eventual Consistency. Einige Anbieter, wie zum Beispiel BigTable, unterstützen Read-your-writes Consistency auf einzelnen Datensätzen, wohingegen z.B. SimpleDB ausschließlich Eventual Consistency anbietet. CloudTPS implementiert hierfür verschiedene Mechanismen, um sich dem Verhalten des darunterliegenden Speichersystems anpassen zu können.

3.2 Architektur

Die zentrale Komponente von CloudTPS ist das Transaction Processing System (TPS), welches die Logik zur Ausführung von Transaktionen komplett von der jeweiligen Speicherschicht abstrahiert. Abbildung 1 verdeutlicht die Interaktion zwischen den Schichten: Clients wie Browser oder Webservices senden Anfragen (engl. *requests*) an eine Web Applikation. Die Anfragen führen zu Transaktionen, welche an das TPS weitergeleitet werden. Hierbei wird deutlich, dass CloudTPS eine Middleware ist, die zwischen Anwendungs- und Speicherschicht eingesetzt wird und die Kommunikation zwischen den genannten Schichten koordiniert. Das TPS interagiert mit dem Speichersystem dergestalt, dass es Daten auf Anfrage lädt und diese periodisch zurückschreibt (engl. *checkpoint*). Ergebnisse werden vom TPS zur Verfügung gestellt und durch die Webapplikation an den Client gesendet (engl. *response*).

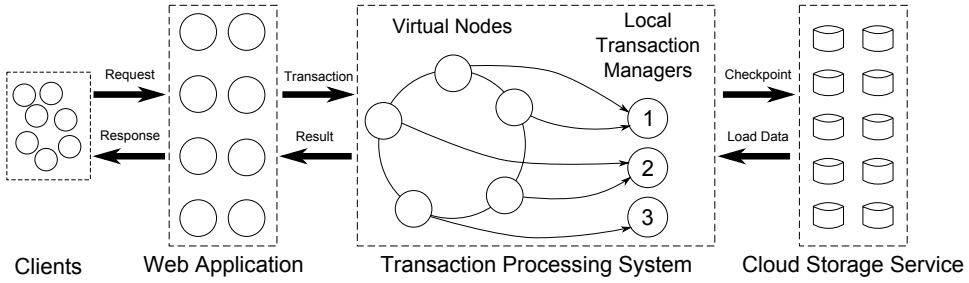

Abbildung 1: Organisation des CloudTPS Systems [5].

Die im Speichersystem hinterlegten Daten werden zunächst in sog. Virtual Nodes aufgeteilt. Virtual Nodes sind paarweise disjunkte Teilmengen der Gesamtdatenmenge und werden den LTMs zugeordnet. Die Zerlegung der Datenmenge erfolgt über ein Consistent Hashing [16] Verfahren, welches sich dadurch auszeichnet, dass eine Größenänderung der Hashtabelle keine komplette Umverteilung der Daten auslöst, dies stellt eine wesentliche Anforderung an ein skalierbares System dar.

Mehrere Virtual Nodes können einem LTM zugeordnet werden, dieser ist für sämtliche Aktionen in Verbindung mit den zugewiesenen Daten verantwortlich. Dazu zählen die Ausführung von Transaktionen auf den verwalteten Daten und deren jeweilige Replikation auf andere LTMs. Die Zuordnung zwischen Virtual Nodes und den LTMs wird an

jedem LTM und in einer Tabelle im Cloud Datenbanksystem hinterlegt. Applikationen und neue LTMs haben somit die Möglichkeit, die aktuelle Systemzugehörigkeit abzufragen. Wie bereits erwähnt, werden Teile der Daten im Hauptspeicher des jeweiligen LTM gehalten. Es ist nicht erforderlich, den kompletten Datenbestand zu laden, sondern nur diejenigen Daten, welche an der aktuellen Transaktion beteiligt sind. Datensätze, deren aktuelle Version aus dem Speichersystem geladen werden kann, werden von den LTMs mittels Ersetzungsstrategie entweder verworfen oder für performante Lesezugriffe weiter im Speicher gehalten. Die Entscheidung, ob ein Datensatz im Speicher gehalten wird, hängt von Faktoren wie Zugriffshäufigkeit und Speicherverbrauch ab.

Die konsequente Ausrichtung auf Webapplikationen wird in der Umsetzung von Transaktionen deutlich: Diese werden nicht erst bei eingehenden Requests initialisiert, sondern stehen den LTMs als Byte-Code nach dem Start des TPS zur Verfügung. Eine Transaktion umfasst eine Menge von Sub-Transaktionen, wobei jede Sub-Transaktion genau einem Datensatz zugeordnet ist. Mittels der gegebenen Primärschlüssel wird die Transaktion initialisiert und ausgeführt.

3.3 Umsetzung der ACID Forderungen

CloudTPS unterstützt verteilte Transaktionen. Nachfolgend wird erläutert, wie dabei die ACID Forderungen erfüllt werden.

3.3.1 Atomarität

CloudTPS implementiert ein Zwei-Phasen-Commit Protokoll zur Sicherstellung der Atomarität. Zunächst wird ermittelt, welche LTMs an einer Transaktion beteiligt sind, anschließend wird die Transaktion an einen der LTMs gesendet, der dadurch zum Koordinator (K-LTM) des gesamten Prozesses wird. Jede Transaktion besitzt ein Statusobjekt, dieses beinhaltet den Zeitstempel der Transaktion, die Zustimmung oder Ablehnung der beteiligten LTMs zum COMMIT und eine Liste der Datenänderungen, die geschrieben werden sollen.

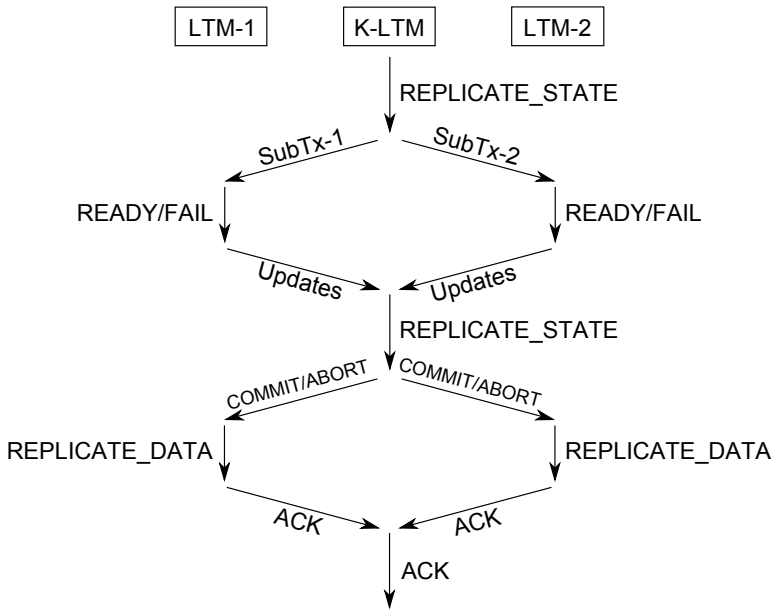

Abbildung 2: Zwei-Phasen-Commit Protokoll in CloudTPS.

Abbildung 2 zeigt den Ablauf am Beispiel von drei LTMs. Der Koordinator nimmt die Transaktion von der Applikation entgegen und koordiniert die Ausführung der Sub-Transaktionen (*SubTx-1*, *SubTx-2*) an den beteiligten LTMs. In Phase 1 wird zunächst der Status am K-LTM an dessen Backup-LTMs repliziert (*REPLICATE_STATE*). Dieser Vorgang wird mit einer SUBMIT Nachricht an die Applikation bestätigt. Anschließend sendet der K-LTM die Transaktion an die beteiligten LTMs (*SubTx-1*, *SubTx-2*), welche zunächst die Transaktion bis zu dem Punkt verarbeiten, an dem zwischen COMMIT und ROLLBACK entschieden wird. An diesem Punkt sendet jeder LTM seine Zustimmung (*READY*) oder Ablehnung (*FAIL*) des globalen COMMIT zum Koordinator. Im Fall einer Zustimmung werden die Änderungen (*Updates*) als REDO [10] Informationen ebenfalls übermittelt. Erhält der K-LTM die Zustimmung zum COMMIT von allen beteiligten LTMs, repliziert dieser erneut den Status der Transaktion inklusive aller Änderungen und ordnet deren Umsetzung an (*COMMIT*). Erhält der K-LTM mindestens eine Ablehnung des COMMIT, wird die Transaktion abgebrochen (*ABORT*). In Phase 2 werden die Änderungen umgesetzt, die beteiligten LTMs replizieren ebenfalls ihre Daten (*REPLICATE_DATA*) und senden nach erfolgreicher Durchführung eine Nachricht an den K-LTM (*ACK*). Dieser sendet nach Beendigung der zweiten Phase eine Nachricht über erfolgreiche Ausführung und das Ergebnis der Transaktion an die Applikation (*ACK*).

Infolge der Replikation sowohl der geänderten Daten, als auch des Status einer Transaktion, ist die atomare Durchführung einer Transaktion sowohl bei Ausfall des K-LTM, als auch eines beteiligten LTMs gewährleistet.

3.3.2 Konsistenz

Die Erfüllung der Forderung nach Konsistenz wird von CloudTPS mit der korrekten Ausführung der Transaktionen verbunden. Wird eine Transaktion auf einer konsistenten Datenbank ausgeführt, befindet sich diese nach Abschluss der Transaktion wiederum in einem konsistenten, eventuell veränderten Zustand. Daraus resultiert, dass die Forderung nach Konsistenz erfüllt ist, solange die Transaktionen korrekt und widerspruchsfrei ausgeführt werden.

Um Strong Consistency innerhalb des verteilten Systems zu gewährleisten, müssen die von einer Transaktion verursachten Änderungen nach deren Beendigung für jede zukünftige Transaktion sichtbar sein. CloudTPS unterstützt mehrere Konsistenzstufen in Abhängigkeit des zugrunde liegenden Cloud Datenbanksystems, eine Unterstützung von Monotonic Read Consistency ist jedoch obligatorisch. Wird zusätzlich Read-your-writes Consistency unterstützt, kann der Datensatz nach dem Rückschreiben aus dem jeweiligen LTM verworfen werden, da jeder folgende Lesezugriff den geänderten Datensatz erhält. Bietet das Speichersystem neben Monotonic Read nur Eventual Consistency an, müssen die Zeitstempel der letzten Version eines Datensatzes in den LTMs hinterlegt werden. Dies ist erforderlich, um Datensätze, welche aus der Speicherschicht geladen wurden, auf ihre Aktualität hin zu prüfen. Sind sie nicht aktuell, wird die laufende Transaktion zur Wahrung der Strong Consistency abgebrochen und verworfen.

3.3.3 Isolation

Eine Transaktion in CloudTPS besteht aus einer Menge Sub-Transaktionen, welche jeweils auf genau einen Datensatz zugreifen. Um Isolation zwischen zwei konkurrierenden Transaktionen zu gewährleisten, müssen demzufolge alle konkurrierenden Sub-Transaktionen serialisiert durchgeführt werden. Der bereits erwähnte Status einer Transaktion beinhaltet

u.a. einen Zeitstempel, welcher den Startzeitpunkt der Transaktion definiert. Mit Hilfe dieses Zeitstempels werden die Zugriffe auf Datensätze dergestalt koordiniert, dass eine Sub-Transaktion eine Änderung an einem Datensatz erst dann durchführen kann, wenn alle älteren Transaktionen ihre Änderungen abgeschlossen haben. Trifft eine Transaktion auf einen Zeitstempel, der größer ist als ihr eigener, wird sie zurückgesetzt und neu gestartet. Diese Situation kann zum Beispiel bei Verzögerungen im Netzwerk auftreten. Das Zwei-Phasen-Protokoll wurde deshalb um einen Status RESTART erweitert, welcher in der ersten Phase von den beteiligten LTMs neben Zustimmung und Ablehnung zum COMMIT an den K-LTM gesendet werden kann.

3.3.4 Dauerhaftigkeit

Um Dauerhaftigkeit gewährleisten zu können, ist es erforderlich, dass alle Änderungen, welche von abgeschlossenen Transaktionen durchgeführt wurden, in die jeweilige Speicherschicht zurückgeschrieben werden. Als Vorbedingung wird allgemein angenommen, dass die verwendete Cloud Datenbank die Forderung nach Dauerhaftigkeit erfüllt.

Solange die Änderungen an den Daten ausschließlich im Hauptspeicher der LTMs umgesetzt wurden, ist die Replikation der LTMs das wesentliche Mittel zur Umsetzung von Dauerhaftigkeit. Aus Gründen der Performanz werden die Daten nicht unmittelbar, sondern periodisch in das Speichersystem übertragen. Sobald die Daten übertragen wurden, obliegt die Verantwortung für die Dauerhaftigkeit der verwendeten Datenbank. Kommt es vor dem Rückschreiben zum Ausfall eines LTM, sind alle Datensätze verloren, die noch nicht zurückgeschrieben wurden. Für die Wiederherstellung des LTM werden dessen Backup-LTMs verwendet. Um unnötige Schreibzugriffe zu vermeiden, wird für jeden Datensatz ein Zeitstempel des Rückschreibens verwaltet, welcher ebenfalls an die Backup-LTMs repliziert wird.

3.4 Fehlertoleranz

Die Zuverlässigkeit des Systems ist im Wesentlichen davon abhängig, dass jedem LTM aktuelle Informationen über den globalen Zustand des TPS zur Verfügung stehen. Dieser Zustand umfasst eine Auflistung aller LTMs und der ihnen zugewiesenen Virtual Nodes. Ohne diese Informationen werden Transaktionen zu falschen LTMs weitergeleitet und schlagen somit fehl. Das TPS muss eine permanente Veränderung der Struktur tolerieren: LTMs werden dem System hinzugefügt, andere fallen aus oder werden manuell deaktiviert. Eine Aktualisierung der Systemzugehörigkeit muss in jedem Fall durchgeführt werden.

CloudTPS nutzt für diesen Prozess ebenfalls ein Zwei-Phasen-Commit Protokoll. In Phase 1 werden zunächst alle LTMs über eine Aktualisierung informiert, diese warten auf die Beendigung aller ihnen zugeteilten Transaktionen und bestätigen mit einem COMMIT ihre Zustimmung zur Aktualisierung. In Phase 2 wird nach Erhalt aller COMMITs die Systemzugehörigkeit aktualisiert und mittels Replikation und Relokation der Daten umgesetzt. Jede Aktualisierung ist mit einem aktuellen Zeitstempel verbunden, welcher von allen LTMs an jede versendete Nachricht angehängt wird. Mit Hilfe dieses Zeitstempels können LTMs feststellen, ob ihre Informationen über den Status des TPS veraltet sind. Ist dies der Fall, verwirft der betroffene LTM seinen aktuellen Status und tritt dem TPS neu bei. Der aktuelle Status wird, wie bereits erwähnt, sowohl in den LTMs, als auch in einer zentralen Tabelle im Cloud Datenbanksystem abgelegt. Parallele Aktualisierungsprozesse, hervorgerufen durch gleichzeitiges Beitreten, Austreten oder Ausfallen von LTMs, werden

durch ein Optimistisches Locking Verfahren behandelt. Es darf immer nur ein Aktualisierungsprozess durchgeführt werden, erhält ein LTM während der Durchführung eine weitere Anfrage, so wird diese mit dem ABORT Befehl verworfen und mit einer zufälligen Wartezeit versehen, um eine fortlaufende Zurückweisung zu vermeiden.

Ein weitere denkbare Situation stellt der Ausfall des Netzwerkes bzw. die Netzwerkpartitionierung dar. Ohne entsprechende Maßnahmen arbeiten alle Partitionen unabhängig voneinander weiter und verletzen somit die Forderung nach Konsistenz. Bezüglich des in Abschnitt 2.3 beschriebenen CAP-Theorems entschieden sich die Entwickler von CloudTPS im Fall einer Netzwerkpartitionierung die Sicherstellung der Konsistenz vor der Verfügbarkeit der Daten zu priorisieren. Im Fall einer Partitionierung kann das System unter zwei Bedingungen Transaktionen weiter durchführen:

1. Eine der Partitionen beinhaltet mehr als 50% der LTMs des aktuellen TPS Status und stellt somit die Mehrheitspartition dar.
2. Die Mehrheitspartition ist in der Lage, alle Datensätze mit Hilfe der Replikate wiederherzustellen.

Die Wahl einer Mehrheitspartition geschieht mit Hilfe des 2PC: In der ersten Phase wird an alle erreichbaren LTMs eine Aktualisierungsnachricht des TPS Status gesendet. Entspricht die Anzahl der eingehenden Zustimmungen der Mehrheit aller LTMs, wird diese Partition als Mehrheitspartition festgelegt. Wird keine Mehrheit festgestellt, werden nicht erreichbare LTMs in die erste Phase weiterer 2PC Iterationen eingebunden, bis die Mehrheit erreicht wird. Solange keine Mehrheitspartition festgelegt ist, weist das TPS alle eingehenden Transaktionen zurück. Wurde eine Mehrheitspartition gebildet, kann diese alle laufenden Transaktionen wiederherstellen und neue Transaktionen entgegennehmen. Minderheitspartitionen wird der Zugriff auf sämtliche Datensätze während der Wiederherstellung verwehrt.

In CloudTPS existieren zu jeder Transaktion und zu jedem Datensatz $N + 1$ Replikate. Dies lässt den parallelen Ausfall von höchstens N Rechnern zu. Einer Mehrheitspartition kann es im schlechtesten Fall an mehr als N Replikaten eines Datensatzes fehlen. Ist dies der Fall, weist das TPS solange eingehende Transaktionen ab, bis alle Datensätze manuell wiederhergestellt wurden.

Während des Wiederherstellungsprozesses kann die Situation eintreten, dass einzelne Partitionen wieder verbunden werden und somit LTMs existieren, welche in mehreren Partitionen vertreten sind. Es muss sichergestellt werden, dass ein LTM nur einer Partition zugeordnet ist. Dies geschieht durch die Festlegung, dass ein LTM nach Beitritt einer Partition keiner weiteren Partition beitreten darf. Dies hat allerdings zur Folge, dass während des Wiederherstellungsprozesses keine erneute Netzwerkpartitionierung toleriert werden kann. Nach Abschluss des Wiederherstellungsprozesses wird der Status des TPS mit dem Zeitstempel der LTMs der Mehrheitspartition aktualisiert und neue LTMs bzw. LTMs der Minderheitspartitionen können dem System wieder beitreten.

4 ElasTraS

ElasTraS (**Elas**tic **Tra**n**S**actional relational database) ist ein Cloud Datenbanksystem, welches im Gegensatz zu CloudTPS keine bestehenden Cloud Datenbanken um ACID-Transaktionen erweitert, sondern eine relationale Datenbank inklusive Transaktionsmanagement auf Basis eines verteilten, fehlertoleranten Dateisystems bietet und auf diesem Weg versucht, die in Abschnitt 1 genannten Defizite von RDBMS auszugleichen. Die Entwickler orientierten sich an den Designentscheidungen bestehender Key-Value Datenbanken und erfüllen dadurch alle Anforderungen, die an ein Cloud Datenbanksystem gestellt werden.

In ElasTraS wird grundsätzlich in zwei Arten der Datenverwaltung unterschieden: Mehrere kleine Datenbanken, welche unterschiedlichen Organisationen zugeordnet sein können (engl. *multi-tenant*) oder eine umfangreiche Datenbank, welche durch einen Mandanten verwaltet wird (engl. *single-tenant*). Die kleinste verwaltbare, logische Einheit in ElasTraS ist die Partition. Eine Partition kann im multi-tenant Szenario eine oder mehrere kleine Datenbanken beinhalten, wohingegen sie im single-tenant Szenario eine Teilmenge einer einzelnen, umfangreichen Datenbank aufnimmt. Auf der Ebene der Partitionen werden Konzepte wie Lastverteilung, Elastizität, Fehlertoleranz und Transaktionen umgesetzt. Skalierbarkeit wird erreicht, weil Änderungsoperationen ausschließlich innerhalb einer Partition stattfinden dürfen. Darüber hinaus bietet ElasTraS die Möglichkeit, sich selbst in einem dynamischen Modus zu verwalten. Dies umfasst das automatische Teilen und Zusammenführen von Partitionen auf Basis definierter Partitionierungsschemata.

4.1 Datenmodell

ElasTraS basiert auf dem relationalen Datenbankmodell: Datenbanken beinhalten Relationen, welche über Fremdschlüsselbeziehungen verknüpft sind. Wie bereits erwähnt, erfolgt die Partitionierung im multi-tenant Szenario auf Ebene der Datenbanken, wobei eine oder mehrere Datenbanken einer Partition zugewiesen werden. Im single-tenant Szenario erfolgt die Aufteilung der Daten auf Grundlage eines partitionierten Datenbankschemas. ElasTraS gibt dabei keine konkrete Herangehensweise vor, empfiehlt jedoch die Partitionierung unter Verwendung einer abhängigen horizontalen Partitionierung [2] des Datenbankschemas. Diese setzt eine baumartige Struktur des Schemas voraus, welches anhand des Primärschlüssels der Wurzelrelation partitioniert wird. Zur Vermeidung partitionsübergreifender Zugriffe werden somit alle abhängigen Fragmente einer Relationen der gleichen Partition zugeordnet.

Es wird deutlich, dass bei der Entwicklung von ElasTraS vergleichbare Designentscheidungen wie bei CloudTPS getroffen wurden, dies liegt daran, dass aufgrund der gleichen Einsatzszenarien ähnliche Anforderungen gestellt werden: Transaktionen in Web- bzw. Cloud Applikationen umfassen nur selten die komplette Datenbank, sondern werden auf einer geringen Anzahl zusammenhängender Datensätze ausgeführt.

4.2 Architektur

Die Architektur von ElasTraS ist in Abbildung 3 dargestellt. Auf der unteren Ebene befindet sich ein verteiltes, fehlertolerantes, append-only[3] Speichersystem (engl. *distributed faul-tolerant storage*) (DFS), welches Strong Consistency für Lese- und Schreibzugriffe,

[3] Append-Only Dateisysteme wurden für Anwendungen konzipiert, in denen häufiger neue Datensätze geschrieben werden, als bestehende Datensätze geändert [6].

Dauerhaftigkeit von Schreibzugriffen und Replikation der Daten zur Verfügung stellt. Dieser Ansatz unterscheidet sich von CloudTPS, bei dem die Speicherschicht eine Cloud Datenbank mit abgeschwächten Konsistenzforderungen ist und CloudTPS selbst die Erfüllung der Strong Consistency sicherstellen muss.

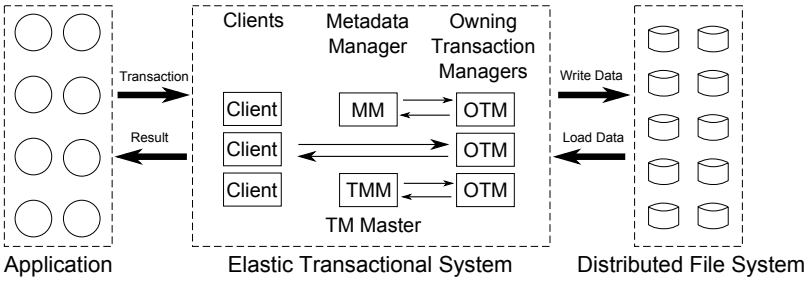

Abbildung 3: Organisation des ElasTraS Systems [9].

Der hohe Aufwand, den Änderungsoperationen im DFS verursachen, macht eine eine Entkopplung der Datenverantwortung zur Erreichung von Skalierbarkeit notwendig: Vergleichbar mit dem Konzept der LTMs in CloudTPS existieren innerhalb von ElasTraS sog. Owning Transaction Manager (OTMs), denen eine oder mehrere Partitionen zugewiesen sind. Diese werden im Hauptspeicher der OTMs gehalten, damit Zugriffe auf das vergleichsweise langsame DFS vermieden werden können. Darüber hinaus garantiert ein OTM die Erfüllung der ACID Forderungen innerhalb jeder einzelnen zugewiesenen Partition und ist für die Wiederherstellung von Partitionen im Fehlerfall verantwortlich. Jeder OTM beinhaltet einen Transaktionsmanager, einen Logmanager und mehrere Proxies zur Kommunikation mit den anderen Systemkomponenten.

Welche Partitionen einem OTM zugewiesen werden, wird durch den Transaction Manager (TM) Master bestimmt. Dessen Aufgaben umfassen darüber hinaus die Neuzuordnung von Partitionen zur Gewährleistung von Lastverteilung und Elastizität, das Erkennen von Ausfällen einzelner OTMs und die Koordination der Maßnahmen zur Wiederherstellung nach einem Ausfall. Es handelt sich um eine zentrale Komponente, welche jedoch zur Vermeidung eines Single Point Of Failure innerhalb des Systems repliziert wird.

Der Meta-Data Manager (MM) verwaltet die systemkritischen Informationen: Die Zuweisung von Partitionen zu OTMs und die aktuellen Lease Informationen der OTMs und des TM Masters. Ein Lease stellt einen zeitlich begrenzten exklusiven Zugriff auf einen Server innerhalb der Infrastruktur, und somit auf eine Menge von Partitionen, dar. Neben der Verwaltung der Leases stellt der MM sicher, dass zu jedem Zeitpunkt ein Lease nur genau einem Knoten zugewiesen ist. Die Verfügbarkeit des MM ist entscheidend für die korrekte Funkionsweise von ElasTraS. Deshalb wird der Zustand des MM mit einer Variante von Paxos [17], einem Strong Consistency Protokoll repliziert, um einen Ausfall des MM zu tolerieren.

Es existiert eine Client Bibliothek, welche es den Applikationen ermöglicht, auf ElasTraS zuzugreifen. Vergleichbar mit dem TPS in CloudTPS gewährleistet die Bibliothek Verteilungstransparenz, um die Komplexität des Auffindens von Partitionen vor der Applikation zu verbergen. Der Client ermittelt mit Hilfe des MM den für eine Anfrage zuständigen OTM und leitet die Anfrage an den entsprechenden Server weiter. Im Fehlerfall werden Anfragen wiederholt bzw. nach einer Neuzuordnung einer Partition an den neuen OTM weitergeleitet. Die Informationen über die Abbildung von Partitionen auf OTMs werden von den Clients zwischengespeichert. Zur Verbesserung der Systemperformanz werden auch Informationen weiterer Partitionen geladen.

Die Referenzimplementierung von ElasTraS nutzt das Apache Hadoop Distributed File System[4] (HDFS) als Speicher. Als Meta-Daten Manager wird Apache ZooKeeper[5] eingesetzt, ein verteilter Koordinationsdienst für verteilte Applikationen. Darüber hinaus wurde ein eigenes Transaktionsmanagementsystem implementiert. Dies geschah aus zwei Gründen: Zum Einen unterstützen nur wenige Open Source Datenbanken eine append-only Semantik des Dateisystems und zum Anderen wird das Konzept der abhängigen horizontalen Partitionierung, welches im Prototyp umgesetzt wurde, ebenfalls nur von wenigen Open Source Datenbanken unterstützt [9]. Die Entwickler orientierten sich an den Konzepten von BigTable und dessen Open Source Implementierung HBase.

4.3 Umsetzung der ACID Forderungen

In ElasTraS erfüllen Transaktionen die ACID Anforderungen innerhalb einer Partition. Nachfolgend werden die eingesetzten Techniken und Protokolle erläutert, welche in der prototypischen Implementierung Anwendung finden.

4.3.1 Atomarität

Im Gegensatz zu CloudTPS, welches die Ausführung von Transaktionen auf mehrere LTMs verteilt, beschränkt sich die Ausführung von Transaktionen in ElasTraS auf eine Partition innerhalb eines OTM. Eine Synchronisation mehrerer OTMs ist nicht vorgesehen und die Implementierung eines Zwei-Phasen-Protokolls somit nicht erforderlich. Der Prototyp von ElasTraS implementiert eine Optimistische Synchronisation [10] zur Gewährleistung der Serialisierbarkeit nebenläufiger Transaktionen.

Im Rahmen dieses Protokolls wird angenommen, dass nebenläufige Transaktionen nicht konkurrieren und somit weder Schreib- noch Lesesperren gesetzt werden müssen. In einer ersten Phase, der Lesephase, führt jede Transaktion ihre Operationen aus, alle Änderungen werden in einem Transaktionspuffer gespeichert. In der anschließenden Validierungsphase werden die Transaktionspuffer auf Konflikte hin untersucht. Besteht ein Konflikt, werden die beteiligten Transaktionen abgebrochen und deren Änderungen rückgängig gemacht. Nach der Validierungsphase folgt die Schreibphase, nach deren Beendigung alle Änderungen global zur Verfügung stehen.

Analog zu CloudTPS werden alle Änderungen zunächst im Hauptspeicher des jeweiligen OTM ausgeführt. Um die Atomarität von Transaktionen auch beim Ausfall eines OTM zu gewährleisten, ist die Nachvollziehbarkeit der Änderungen einer Transaktion sicherzustellen. Bevor die Schreibphase durchgeführt wird und die Änderungen global verfügbar sind, werden alle Änderungen der Transaktion in einem Write Ahead Log (WAL) des OTM gesichert. Das WAL befindet sich im konsistenten DFS, ein erfolgreicher Schreibzugriff stellt somit die Dauerhaftigkeit des Eintrages sicher. Für jede Transaktion wird ein COMMIT Eintrag an das WAL angefügt, nur diese Einträge werden bei der Wiederherstellung eines OTM berücksichtigt.

4.3.2 Konsistenz

Die Forderung nach Konsistenz ist analog zu CloudTPS zum Einen durch die korrekte Ausführung der Transaktionen und zum Anderen dank der Zusicherung der Konsistenz

[4] `http://hadoop.apache.org/hdfs/`
[5] http://zookeeper.apache.org/

innerhalb der Speicherschicht gewährleistet. Da es sich beim DFS um einen verteilten Speicher mit Strong Consistency handelt, sind keine zusätzlichen Maßnahmen zur Unterstützung verschiedener Formen von Eventual Consistency erforderlich. Werden Transaktionen atomar durchgeführt und deren Änderungen erfolgreich in das DFS eingebracht, ist die Forderung nach Konsistenz erfüllt.

4.3.3 Isolation

Durch die bereits in Abschnitt 4.3.1 beschriebene Optimistische Synchronisation paralleler Transaktionen wird gleichzeitig die Forderung nach Isolation erfüllt. Führt die parallele Ausführung in der Validierungsphase zu einem Konflikt, werden alle am Konflikt beteiligten Transaktionen abgebrochen und deren Änderungen rückgängig gemacht.

4.3.4 Dauerhaftigkeit

Änderungen einer Transaktion werden zunächst im Hauptspeicher des OTM in einem Schreibpuffer verwaltet. Analog zu CloudTPS müssen diese Änderungen periodisch in das Speichersystem zurückgeschrieben werden, dieses garantiert die Dauerhaftigkeit von Anfügeoperationen. Durch die append-only Semantik des DFS können Änderungen nicht wie bei RDBMS innerhalb einer Seite erfolgen. ElasTraS verwendet daher ein Datenformat, welches sich an den SSTables[6] (Sorted String Tables) von BigTable orientiert. Datenänderungen werden als neue SSTable in das DFS eingebracht, dies erfolgt asynchron und blockiert damit keine aktuellen Änderungsoperationen innerhalb des OTM. Bei Lesezugriffen müssen demnach die SSTables aus dem DFS mit dem Schreibpuffer innerhalb des OTM zusammengeführt werden. Auch der Schreibpuffer wird als SSTable organisiert, das Zusammenführen kann demzufolge in linearer Zeit erfolgen. Durch das periodische Rückschreiben des Schreibpuffers entstehen viele Fragmente innerhalb des DFS, welche bei einem Lesezugriff im ungünstigsten Fall zusammengeführt werden müssen. ElasTraS führt diese Fragmente periodisch zusammen, um Lesezugriffe zu beschleunigen. Welche Daten aus dem Schreibpuffer des OTM verdrängt werden, wird vergleichbar mit CloudTPS durch ein Least Recently Used (LRU) Verfahren bestimmt.

4.4 Fehlertoleranz

Analog zu CloudTPS ist es auch bei ElasTraS notwendig, jedem TM aktuelle, korrekte Informationen über den globalen Zustand des Systems zur Verfügung zu stellen. Die Verantwortung obliegt in ElasTraS dem Meta-Daten Manager. Es ist zunächst erforderlich, einen kurzen Überblick über die Funktionsweise von ZooKeeper zu geben, bevor die Fehlertoleranz des Gesamtsystems beschrieben werden kann. Abbildung 4 stellt den Aufbau von ZooKeeper in Verbindung mit ElasTraS schematisch dar.

ZooKeeper stellt TMs den Zugriff auf das DFS zur Verfügung, dafür werden sog. znodes verwaltet. Ein znode hält zwei wichtige Meta-Informationen: Den Verweis auf einen Datenbereich innerhalb des DFS und die IP des TM, welcher auf diesen Bereich momentan exklusiven Zugriff hat. Will ein TM exklusiven Zugriff auf einen bestimmten Datenbereich erhalten, so erfordert dies einen Lease des entsprechenden znodes, in welchen die IP eingetragen wird. Ein TM gilt als aktiv, wenn er den Lease auf einen znode mit seiner IP hält. Leases müssen in periodischen Abständen erneuert werden, geschieht dies nicht,

[6] Eine SSTable ist eine unveränderliche Struktur, in der die Datensätze nach Primärschlüsseln sortiert abgelegt sind und schnelle Lookups und Scans unterstützt werden.[6]

können andere TMs den Lease erhalten. Über die IP Adressen in den znodes ist eine bidirektionale Kommunikation zwischen den OTMs und dem TM Master möglich. Eine weitere Funktionalität von ZooKeeper sind Watches. Mit einem Watch ist es möglich, sich über Änderungen des Lease Zustandes eines znode informieren zu lassen. Der TM Master, welcher für die Komposition der OTMs verantwortlich ist, registriert Watches auf allen znodes, welche den OTMs zugeordnet sind. Um den Ausfall des TM Master zu erkennen, registrieren die OTMs einen Watch auf den znode des Masters. Durch die Strong Consistency in Zookeeper werden Änderung an den znodes atomar durchgeführt und sind nach Abschluss der Änderung global sichtbar.

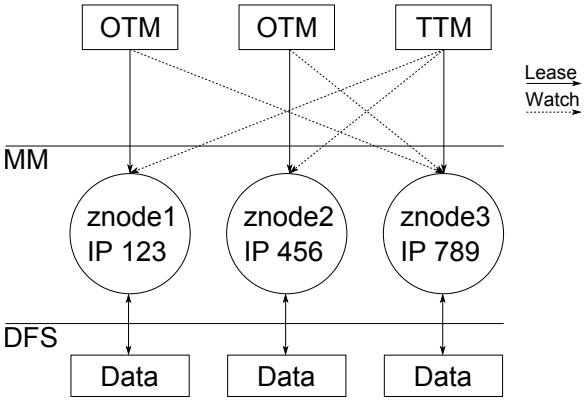

Abbildung 4: Schematischer Aufbau von ElasTraS in Verbindung mit ZooKeeper als Meta-Daten Manager.

Bei Ausfall eines OTM wird der TM Master durch den Watch auf dem znode des OTM benachrichtigt. Der Master löscht zunächst atomar den znode des ausgefallenen OTM. Erfolgt der Ausfall des OTM in Folge einer Netzwerkpartitionierung, wird durch das Löschen sichergestellt, dass der betreffende OTM keine neue Verbindung zum znode herstellen kann, wodurch mehrere unabhängige Partitionen aktiv sein können. Stellt andererseits der OTM den Verlust seines eigenen Leases fest, unterbindet er den Zugriff auf die von ihm verwalteten Partitionen. Somit ist die Toleranz gegenüber Netzwerkpartitionierung unter gleichzeitiger Einschränkung der Verfügbarkeit sichergestellt.

Partitionen, welche durch einen ausgefallenen OTM verwaltet wurden, werden dem TM Master zugewiesen. Dieser hat die Aufgabe, die Partitionen auf die verbliebenen OTMs zu verteilen. Erhält ein OTM Zugriff auf eine inkonsistente Partition, so führt er zunächst die Wiederherstellung anhand des WAL durch, bevor die Partition für Zugriffe wieder zur Verfügung steht. Da nur abgeschlossene Transaktionen in das WAL eingetragen werden, ist lediglich eine REDO [10] Wiederherstellung erforderlich.

Ein Ausfall des TM Master wird ebenfalls durch ZooKeeper behandelt: Wie in Abschnitt 4.2 beschrieben, sind innerhalb des Systems replizierte TM Master Prozesse vorhanden, von denen jedoch nur einer den Lease auf den znode des Masters hält. Kommt es zu einem Ausfall des TM Masters, werden die Replikate durch die Watch Funktion darüber informiert. ZooKeeper stellt sicher, dass nur ein Replikat den Lease auf den znode erhält. Der entsprechende Prozess trägt seine IP in den znode ein, wodurch alle OTMs über die Änderung informiert werden.

5 Fazit

Beide Systeme setzen sich zum Ziel, ACID-Transaktionen unter Einhaltung von Skalierbarkeit, Verfügbarkeit, Elastizität, Selbstverwaltung und Fehlertoleranz zu gewährleisten. Die Architekturen sind dabei ähnlich konzipiert: Daten werden von der Speicherschicht entkoppelt, im Hauptspeicher verteilter Transaktionsmanager verwaltet und periodisch in den Speicher zurückgeschrieben. Ein Client ermöglicht der Anwendung transparenten Zugriff auf das System.

Einer der wesentlichen Unterschiede ist die Wahl der Speicherschicht: CloudTPS setzt auf bestehende Cloud Datenbanken wie HBase oder SimpleDB auf, wohingegen ElasTraS die Erfüllung der Dauerhaftigkeit an ein verteiltes, fehlertolerantes Dateisystem delegiert. Fehlertoleranz durch Replikation wird bei CloudTPS durch die Verteilung von Kopien zwischen den LTMs erreicht, während in ElasTraS das DFS für die Replikation der Daten verantwortlich ist.

Ein weiterer wichtiger Unterschied ist die Partitionierung der Anwendungsdaten. CloudTPS verteilt die Daten auf mehrere LTMs und unterstützt verteilte Transaktionen auf Basis eines Zwei-Phasen-Commit Protokolls. ElasTraS beschränkt Transaktionen auf eine Partition, die Koordination verteilter Transaktionen ist somit nicht erforderlich.

Die Forderung nach Skalierbarkeit ist bei CloudTPS genau dann erfüllt, wenn nur wenige Datensätze innerhalb einer Transaktion benötigt werden. Somit ist CloudTPS auf den Einsatz in entsprechenden Anwendungen beschränkt. ElasTraS ist skalierbar, solange Transaktionen nur innerhalb einer Partition ausgeführt werden. Dies führt ebenfalls zu einer Beschränkung der Einsatzmöglichkeiten, da nicht generell garantiert werden kann, dass eine Partitionierung des Datenbankschemas existiert, bei der partitionsübergreifende Zugriffe vermieden werden. Den Verzicht auf verteilte Transaktionen begründen die Autoren von ElasTraS mit dem zentralen Transaktions Manager des Zwei-Phasen-Commit Protokolls: Dieser stellt ihrer Ansicht nach ein Bottleneck innerhalb des Systems dar. Das dies nicht zwingend der Fall ist, wurde in der Implementierung von CloudTPS gezeigt: Durch Replikation der Transaktionsmanager und dynamischer Festlegung des Koordinator LTM einer Transaktion wird der Transaktionsmanager als Bottleneck ausgeschlossen. Es sei jedoch erwähnt, dass ElasTraS im Gegensatz zu CloudTPS den kompletten Funktionsumfang eines RDBMS innerhalb einer Partition zur Verfügung stellt.

Ein Bottleneck und gleichzeitig Single Point of Failure in CloudTPS ist der zentrale Zeitserver, welcher die korrekte Funktion des Zeitstempelverfahrens zur Einhaltung der Isolation garantiert. Die Autoren diskutieren jedoch einen Ansatz auf Basis von LTM-IDs um diesem Problem zu begegnen und Eindeutigkeit innerhalb des Systems zu gewährleisten.

Beide Systeme unterstützen die Ausführung komplexer, verteilter Leseanfragen. CloudTPS bietet die Möglichkeit entsprechende Anfragen direkt an die Speicherschicht weiterzuleiten. Die gelesenen Daten sind in Folge der Eventual Consistency nicht immer aktuell, es wird jedoch garantiert, dass die gelesenen Daten die gleiche Versionsnummer tragen. In ElasTraS werden verteilte Leseanfragen an die entsprechenden OTMs weitergeleitet und das Ergebnis im Client aggregiert, bevor es an die Anwendung übertragen wird.

Sowohl CloudTPS als auch ElasTraS bieten die Möglichkeit, eine dynamische Datenrelokation bei Änderung der Lastverhältnisse durchzuführen und somit die Forderung nach Elastizität zu erfüllen. Dies ist insbesondere in Cloud Infrastrukturen erforderlich, bei denen die Möglichkeit besteht, bedarfsorientiert Ressourcen in Anspruch zu nehmen.

Weiterhin sei angemerkt, dass beide Systeme zuverlässige Mechanismen zur Fehlererkennung bzw. Fehlerbehebung implementieren. CloudTPS bietet mit dem Konzept der Mehr-

heitspartition ein zuverlässiges Verfahren im Fall einer Netzwerkpartitonierung an, Änderungen an der Systemzugehörigkeit werden dabei durch ein 2PC atomar durchgeführt. ElasTraS delegiert die Replikation der Daten und des WAL an das DFS, Rechner- bzw. Netzwerkausfälle werden durch den Einsatz von ZooKeeper toleriert.

Abschließend lässt sich feststellen, dass beide Systeme hinsichtlich des CAP-Theorems die Forderung nach Konsistenz und Ausfallsicherheit erfüllen, Letztere ist nur durch den Verzicht auf Verfügbarkeit im Fehlerfall zu gewährleisten. In CloudTPS weist das TPS eingehende Transaktionen solange ab, bis eine Mehrheitspartition gebildet wurde. In ElasTraS sind bei Ausfall eines OTM die verwalteten Daten für den Zeitraum der Neuzuweisung durch den TM Master und die eventuelle Wiederherstellung der Daten nicht verfügbar.

In dieser Seminararbeit wurden die Transaktionssysteme CloudTPS und ElasTraS vorgestellt und hinsichtlich ihrer Architektur, der Erfüllung der ACID Anforderungen und ihrem Verhalten im Fehlerfall verglichen. Beide Systeme stellen überzeugende Konzepte zur Vereinigung von Transaktionen und den Eigenschaften moderner Cloud Infrastrukturen dar. Die Konzepte zielen dabei klar auf den Einsatz innerhalb von Webanwendungen ab. Es sei abschließend anzumerken, dass für beide Systeme prototypische Implementierungen zur Verfügung stehen, offiziell werden jedoch keine kommerziellen Systeme genannt, welche eines der beiden Systeme produktiv einsetzen.

Literatur

[1] BITKOM. Cloud Computing - Evolution in der Technik, Revolution im Business. http://www.bitkom.org/files/documents/BITKOM-Leitfaden-CloudComputing_Web.pdf, 2009. zuletzt abgerufen am 11.01.2012.

[2] Erhard Rahm. *Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Addison-Wesley, 1994.

[3] Oracle Corporation. Partitioning with Oracle Database 11g Release 2. http://www.oracle.com/technetwork/database/options/partitioning/twp-partitioning-11gr2-2009-09-130569.pdf, 2010. zuletzt abgerufen am 11.01.2012.

[4] MySQL AB. Guide to MySQL 5.1 Partitioning. http://www.mysql.de/why-mysql/white-papers/mysql_wp_partitioning.php, 2009. zuletzt abgerufen am 11.01.2012.

[5] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for Web applications in the cloud. Technical Report IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, February 2010. http://www.globule.org/publi/CSTWAC_ircs53.html.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[7] Amazon Web Services LLC. Amazon SimpleDB. http://aws.amazon.com/de/simpledb/. zuletzt abgerufen am 11.01.2012.

[8] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[9] Sudipto Das, Shashank Agarwal, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self managing transactional database for the cloud. Technical report, CS, UCSB, 03/2010 2010.

[10] T. Härder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, 2001.

[11] Gunter Saake, Andreas Heuer, and Kai-Uwe Sattler. *Datenbanken: Implementierungstechniken ; [Architekturprinzipien, Dateiorganisation & Zugriffsstrukturen, massiv verteilte Datenverwaltung in P2P-Systemen, Illustration der Konzepte am Beispiel aktueller DBMS-Produkte].* mitp-Verl., Bonn, 2., aktualisierte und erw. aufl. edition, 2005.

[12] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.

[13] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, page 2002, 2002.

[14] Dan Pritchett. Base: An acid alternative. *Queue*, 6:48–55, May 2008.

[15] Werner Vogels. Eventually consistent. *Queue*, 6:14–19, October 2008.

[16] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for

relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[17] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.