Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Wieso, Weshalb, Warum?

1.2 Zielsetzung dieser Arbeit

These: Mittels Saga-Pattern implementierte lang andauernde Transaktionen (LLT) ermöglichen (k)eine ausfallsichere Konsistenz in Microservicesystemen.
Leitfragen: Wie kann mithilfe von BASE-Eigenschaft und Softstate ausfallsichere Konsistenz hergestellt werden?
(Wie) kann der 2 Phasencommit in eine Saga-Architektur integriert werden?

1.3 Aufbau dieser Arbeit

- Kapitel 1: Theoretische Grundlagen
- Kapitel 2: Methodik
  - Entwurf und Implementierung eines reinen Saga-Systems
  - Abänderung der Implementierung durch
    - Saga-Werkzeuge (Checkpoints, Recovery Mechanismen)
    - 2 Phasencommit
  - Bewertung der Systeme
    - Wie Fehleranfällig ist das System?
      - Welche Single-Point-of-Failures gibt es?
      - Welche Recovery-Mechanismen gibt es?
      - Welche Datenanomalien sind die Folge eines Ausfalls?
    - Wie komplex ist das System?
    - Wie hoch ist der Netzwerkverkehr? (Anzahl der benötigten Requests)
      - Wie performant ist der Happy-Path?
      - Wie performant ist der Sad-Path?
  - Erläuterung des Mess- und Bewertungsvorgangs
- Kapitel 3: Ergebnisse aus Kapitel 2

- Kapitel 4: Diskussion - Bewerten der Ergebnisse in Bezug auf die These und der Leitfragen; Beantwortung der These und der Leitfragen

2 Grundlagen

2.1 Grundlagen

Um das in dieser Arbeit betrachtete Saga-Pattern zu verstehen, sollen zuerst einige Grundlagen erläutert werden. Besonders die im Titel der Arbeit enthaltenen Begriffe *System* und *Konsistenz* sollen in diesem Abschnitt erläutert werden.

2.1.1 System allgemein

Ein System beschreibt einen abgegrenzten Bereich der objektiven Realität. Außerhalb dieses Bereichs liegt die Umgebung, die somit nicht zum System gehört. Zwischen des Systems und seiner Umgebung befindet sich der Systemrand.

2.1.2 System in der Softwareentwicklung

In der Softwareentwicklung besteht ein System aus einer Menge miteinander interagierenden Softwarekomponenten. Diese Komponenten arbeiten an einem gemeinsamen Ziel. Neben der Software und deren Quellcode gehören auch Nutzerhandbücher, Tests, Bestandteile für die Instandhaltung sowie Spezifikationen und Konzepte zum System.

2.1.3 Zustand von Systemen

Ein Softwaresystem befindet sich zu jedem Zeitpunkt in einem Zustand. Der Wechsel eines Zustands ist die Folge von Nutzerinteraktionen und festgelegten Routinen. Damit das System reibungslos funktionieren kann, darf es nur zwischen gültigen Zuständen wechseln.

2.1.4 Sichten auf Systeme

Softwaresysteme können unter verschiedenen Sichten betrachtet werden. TODO
- mathematische Sicht: Endlicher Automat - Endliche Menge von Zuständen - Endliches Alphabet - Übergangsfunktion von Zustand x Alphabet -> Zustand - Startzustand - endliche Menge von Endzuständen

2.1.5 Transaktionen

Ein Übergang von einem Zustand in den nächsten Zustand im Modell des Zustandsautomats ist eine einzige Operation. Häufig äußert sich eine solche Operation jedoch in einer Folge von auszuführenden Operationen. Eine solche Folge von Operationen wird als Transaktion bezeichnet. Teil einer Transaktion kann beispielsweise eine oder mehrere

Datenbankoperationen oder der Aufruf einer externen Schnittstelle sein. Transaktionen können auf verschiedenen Ebenen betrachtet werden. Auf Datenbankebene werden Transaktionen anders umgesetzt als auf Ebene der Geschäftslogik.

2.1.6 ACID Eigenschaft von Transaktionen

ACID ist ein Akronym, welches häufig verwendet wird, um Eigenschaften auszudrücken, die eine Transaktion auf Datenbankebene aufweisen sollte. Die Buchstaben stehen für *A*tomicity (Atomarität), *C*onsistency (Konsistenz), *I*solation (Isolation) und *D*urability (Dauerhaftigkeit).
Atomarität beschreibt hier das Alles-Oder-Nichts-Prinzip. Eine Transaktion, also eine Folge von Datenbankoperationen, muss entweder alle Operationen ausführen oder in einem Fehlerfall alle Änderungen auf den ursprünglichen Zustand zurückrollen. Als illustratives Beispiel soll das Szenario einer Überweisung dienen. Eine Überweisung besteht aus dem Erhöhen des Kontostands des Debitors und dem Verringern des Kontostands des Kreditors. Eine solche Transaktion kann fehlschlagen. Dabei ist sicherzustellen, dass die initialen Kontostände wiederhergestellt werden. Wird die Atomarität nicht gewährleistet, könnte sich die Gesamtsumme des im System enthaltenen Systems verändern, Geld geht also verloren oder Geld wird aus dem Nichts generiert. Das ist natürlich nicht akzeptabel.
Konsistenz lässt sich mit widerspruchsfrei übersetzen. War das System vor Ausführung der Transaktion widerspruchsfrei, also konsistent, dann ist es das nach Ausführung der Transaktion ebenso. Um dies zu gewährleisten, sind verschiedene Mechanismen im Datenbanksystem im Einsatz. Dies werden im folgenden Abschnitt beschrieben.
Isolation stellt sicher, dass sich parallel in Ausführung befindliche Transaktionen nicht gegenseitig beeinflussen können. Das bedeutet, dass eine Transaktion Teile der Datenbasis sperren können muss. Eine solche Sperre führt jedoch zu blockierendem Verhalten und schränkt somit die nebenläufige Benutzung der Datenbank ein.
Dauerhaftigkeit von Transaktionen muss gewährleistet sein, damit bei einem Ausfall Transaktionsdaten nicht verloren gehen. Um Dauerhaftigkeit zu erreichen, wird ein Transaktionslog gefährt, welches im Ausfall verwendet werden kann, um die verlorenen Transaktionen nachzuspielen.

2.1.7 Konsistenz auf Datenbankebene

Auf Datenbankebene wird die Konsistenz durch drei Mechanismen sichergestellt: Bereichsintegrität, Entitätsintegrität, Referentielle Integrität und logische Konsistenz.
Die *Bereichsintegrität* stellt lediglich sicher, dass die Werte einer Spalte in einem definierten Wertebereich befinden.
Die *Entitätsintegrität* beschreibt die Eindeutigkeit eines Datensatzes und somit Fähigkeit, diesen Datensatz mittels Primärschlüssel eindeutig identifizieren zu können.
Die *Referentielle Integrität* gewährleistet, dass Relationen zwischen Datensätzen nicht auf nicht existierende Datensätze zeigen können. Ist ein Fremdschlüssel gesetzt, so muss der referenzierte Datensatz existieren.
Die in der Datenbank enthaltenen Daten drücken einen Sachverhalt aus. Die semantische Bedeutung dieser Daten kann nicht durch das Datenbanksystem sichergestellt werden. Es

liegt in der Verantwortung des Entwicklers, allen Nutzerinteraktionen mit dem System die korrekten Transaktionen zuzuordnen. Ein System ist *logisch konsistent*, wenn kein logisch inkonsistenter Zustand erreichbar ist.

2.2 Methoden für die Sicherstellung von Konsistenz

Um einen konsistenten Zustand zu gewährleisten gibt es verschiedene Werkzeuge. Dazu gehören vor allem die Unterstützung von lokalen Transaktionen der Datenbank und der 2 Phasencommit. In diesem Abschnitt werden die (...) beschrieben.

2.2.1 Transaktionen auf Datenbankebene

Eine lokale Transaktion einer Anwendung wird oft durch eine Nacheinanderausführung von SQL-Statements realisiert. Zu Beginn wird dem DBMS signalisiert, dass die folgenden Operationen als atomare Transaktion zu interpretieren sind. Analog dazu gibt es ein Signal, welches das Ende einer solchen Folge markiert, den *Commit*. Tritt vor Erreichen des Commits ein Fehler auf, wird ein *Rollback* ausgeführt. Dieser Rollback überführt die Daten in den Zustand vor Beginn der Transaktion. Die Datenbank - Transaktion - Begin - Commit - Rollback - Aufruf aus der Anwendung

2.2.2 Transaktionen auf Anwendungsebene

- Operation ist ein Funktionsaufruf - Verkettung ist mittels try-catch Block behandelbar - Verantwortung des Entwicklers, dass alle Operationen erfolgreich sind - Verantwortung des Entwicklers, Fehler zu behandeln - Abwicklung der Transaktion per DBMS Transaktionen sinnvoll

2.2.3 Grenzen der Konsistenz in Verteilten Systemen

- Transaktion kann Aktion beinhalten, die eine Abhängigkeit aufruft (zB Aufruf einer externen Http-Schnittstelle) - Zentrales Problem: Wie stelle ich sicher, dass ein Aufruf angekommen ist? ..., dass der Aufruf erfolgreich war? Wie gehe ich vor, wenn eine Aktion einer Transaktion nicht geklappt hat? - Folge von abhängigen Operationen (= Substitution: Ergebnis eines Aufrufs ist Argument der nächsten Aufrufs)

2.2.4 2 Phasencommit

- 2 Phasen Commit als verteilte Umsetzung des Transaktionsvorgehens - zentraler Koordinator - Vorbereitungsphase: Alle teilhabende Akteure der Transaktion geben dem Koordinator die Bestätigung, dass die Operation ausgeführt werden kann. Damit sie dieses Versprechen halten können, beinhalten die Vorbereitungen oft Blockierungen auf Datenbankseite. - Commitphase: Koordinator gibt den Teilhabenden das Signal, ihre Operation auszuführen. Bei Erfolg werden auch alle Blockierungen aufgehoben. Die Transaktion wird als erfolgreich markiert. - Optionale Rollbackphase: Falls ein Teilhabender einen Fehler zurückgibt, werden alle ausgeführten Änderungen zurückgenommen (Rollback). Danach

wird die Transaktion als abgebrochen markiert. Alle Blockierungen müssen zurückgenommen werden. - Nachteile: sehr hohe Chattines, sehr langsam, blockierend, geringer Throughput, komplexe Implementierung

2.2.5 Saga-Pattern

- Fehlerbehandlungsstrategie für monolithische und verteilte Systeme - Auflösen der Atomarität der Transaktionen in einzelne lokale Transaktionen T - Definieren von Kompensationsaktionen C - jedes T hat ein C - sequentielle Ausführung der Ts - schlägt ein T fehl, kann ein entsprechendes C ausgeführt werden

2.2.6 Vergleich von 2 Phasencommit und Sagas

- In welcher Ebene wird der Fehler erkannt - Wird der Fehler zugelassen - In welcher Ebene wird der Fehler behoben - Gleichzeitig(atomare Operationen durch 2PC gleichzeitig auslösen) vs sequentiell (Atomarität wird ersetzt durch sequentielle Operationen, die einzeln behandelt werden können)

3 Saga Pattern

3.1 Allgemeine Probleme verteilter Transaktionen bei lang bestehenden Transaktionen

- Long Lived Transactions Zentrales Problem: Locks in der Datenbank

3.2 Funktionsweise

- Saga ist erfolgreich, wenn alle Ts erfolgreich ausgeführt werden -> Übergang von: Ausgangszustand -> Ausführung T1 -> Ausführung T2 -> Endzustand - Saga ist fehlgeschlagen, wenn ein T fehlschlägt - Ausführung der Cs - Systemzustand ist danach immer noch konsistent - Backward Recovery: Alle Ts, die ausgeführt wurden, werden durch Ausführung des entsprechenden Cs gerollbackt - Forward Recovery: Einführung von Save Points zwischen den Ausführungen (zB T1-T5, Checkpoint nach T2 und Checkpoint nach T3) - Fehler in T3 führt zu Rollback bis letztem Save Point S1 (entspricht Ausführung von C3, Zustand nach Ausführung nach T1 und T2) - Wiederaufnehmen der Saga: Ausführung von T3 - T5 - Kompletter Rollback, falls es nicht geht (Backward Recovery)

3.2.1 Ts und Cs

3.2.2 Forward Recovery

3.2.3 Backward Recovery

3.2.4 Saga Execution Component

3.2.5 Transaktionslog

3.3 Implementierungsstrategie

3.3.1 Orchestration

3.3.2 Choreographie

3.4 Abwägung - wann 2PC und wann Saga

3.5 Abwägung - Wann Orchestration und wann Choreographie

4 Design

4.1 Design Choreografie

4.2 Design Orchestration

5 Implementierung

Abbildungsverzeichnis