

Reaktive Programmierung
Vorlesung 11 vom 28.06.2022
Eventual Consistency

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

- ▶ Einführung
- ▶ Monaden und Monadentransformer
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren: Grundlagen & Implementierung
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ **Eventual Consistency**
- ▶ CRDTs
- ▶ Robustheit, Entwurfsmuster und Theorie der Nebenläufigkeit, Abschluss
- ▶ Reaktive Programmierung in der Praxis

Heute

- ▶ Konsistenzeigenschaften
- ▶ Eventual Consistency
- ▶ Operational Transformation
 - ▶ *Das Geheimnis von Google Docs und co.*

Was ist eigentlich Konsistenz?

- ▶ Konsistenz = **Widerspruchsfreiheit**
- ▶ Logische Konsistenz:
 - ▶ Eine Formelmengemenge Γ ist konsistent im Kalkül wenn: $\exists A. \neg(\Gamma \vdash A)$
 - ▶ Ein Kalkül ist konsistent wenn $\exists A. \neg(\emptyset \vdash A)$
- ▶ In einem verteilten (Speicher)system:
 - ▶ Redundante (verteilte) Daten
 - ▶ **Globale** Widerspruchsfreiheit?

Strikte Konsistenz

Strikte Konsistenz

- ▶ Daten sind zu jedem Zeitpunkt global konsistent.
- ▶ Eine Leseoperation in einem beliebigen Knoten gibt den Wert der letzten globalen Schreiboperation zurück.
- ▶ Beispiel: Lokaler Speicherzugriff
- ▶ In verteilten Systemen **nicht praktikabel**.

Eventual Consistency

Eventual Consistency

Wenn **längere Zeit** keine Änderungen oder Fehler stattfinden konvergieren die Daten an jedem Knoten zu einem gemeinsamen Wert.

- ▶ Beispiel: DNS
- ▶ **Informelle** Anforderung:
 - ▶ Abfragen können beliebige Werte zurückgeben bevor die Knoten konvergieren.
 - ▶ Keine Sicherheitseigenschaft!

Zwischen Strikter und Eventual Consistency

- ▶ Strikte Konsistenz ist zu **stark**, Eventual Consistency häufig zu **schwach**.
- ▶ Dazwischen gibt es ein breites Spektrum an Konsistenzeigenschaften.
- ▶ Wir unterscheiden
 - ▶ Data-Centric Consistency
 - ▶ Client-Centric Consistency

Sequentielle Konsistenz

Sequentielle Konsistenz

- ▶ Zustand nach verteilter Programmausführung = Zustand nach einer äquivalenten sequentiellen Ausführung in einem Prozess.
- ▶ Jeder Prozess sieht die selbe Folge von Operationen.
- ▶ Nur so schnell wie das schwächste Glied

Kausale Konsistenz

Kausale Konsistenz

- ▶ Abschwächung von sequentieller Konsistenz mit selbem Ergebnis
- ▶ Jeder Prozess sieht Operationen die **in Abhängigkeit stehen** in der selben Reihenfolge.

Client-Centric Consistency

- ▶ Monotonic Read
- ▶ Monotonic Write
- ▶ Read your Writes
- ▶ Writes follow Reads

Vektor-Uhren

- ▶ Um Operationen zu ordnen benötigen wir Timestamps
 - ▶ Kausalität muss erhalten bleiben
 - ▶ Timestamps müssen eine totale Ordnung haben
- ▶ Datum und Uhrzeit ungeeignet
- ▶ Lösung: Vektor-Uhren
 - ▶ Jeder Knoten hat einen Zähler, der bei Operationen hochgesetzt wird
 - ▶ Zusätzlich merkt sich jeder Knoten den aktuellsten Zählerwert, den er bei den anderen Knoten beobachtet hat.

Strong Eventual Consistency

- ▶ **Strong Eventual Consistency** garantiert:
 - ▶ wenn zwei Knoten die **gleiche (ungeordnete) Menge** von Operationen empfangen haben, befinden sie sich im **gleichen Zustand**.
- ▶ Beispiel: Versionskontrollsystem *git*
 - ▶ Wenn jeder Nutzer seine lokalen Änderungen an alle verteilt hat, dann haben alle Nutzer die gleiche Sicht auf den *head*.

Monotonie

- ▶ Strong Eventual Consistency kann einfach erreicht werden:
 - ▶ Nach jedem empfangenen Update alle Daten zurücksetzen.
- ▶ Für sinnvolle Anwendungen brauchen wir eine weitere Garantie:

Monotonie

Ein verteiltes System ist monoton, wenn der Effekt jeder Operation erhalten bleibt (keine Rollbacks).

Beispiel: Texteditor

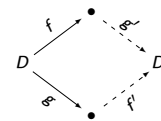
- ▶ Szenario: Webinterface mit Texteditor
- ▶ Mehrere Nutzer können den Text verändern und sollen **immer die neuest mögliche Version** sehen.
- ▶ Siehe Google Docs, Etherpad und co.
- ▶ Der Effekt jeder Operation soll erhalten bleiben
- ▶ Es soll niemals Konflikte geben

Naive Methoden

- ▶ Ownership
 - ▶ Vor Änderungen: Lock-Anfrage an Server
 - ▶ Nur ein Nutzer kann gleichzeitig das Dokument ändern
 - ▶ Nachteile: Verzögerungen, Änderungen nur mit Netzverbindung
- ▶ Three-Way-Merge
 - ▶ Server führt nebenläufige Änderungen auf Grundlage eines **gemeinsamen Ursprungs** zusammen.
 - ▶ Requirement: *the chickens must stop moving so we can count them*
 - ▶ Nicht konfliktfrei möglich.

Operational Transformation

- ▶ Idee: Nicht-kommutative Operationen transformieren



- ▶ Für *transform* muss gelten:

$$\text{transform } f \ g = \langle f', g' \rangle \implies g' \circ f = f' \circ g \quad (1)$$

$$\text{applyOp } (g \circ f) \ D = \text{applyOp } g \ (\text{applyOp } f \ D) \quad (2)$$

Operationen für Text

Operationen bestehen aus **drei** Arten von Aktionen:

- ▶ **Retain**— Buchstaben beibehalten
- ▶ **Delete**— Buchstaben löschen
- ▶ **Insert c**— Buchstaben c einfügen

Eine **Operation** ist eine Sequenz von Aktionen

- ▶ Operationen sind **partiell**.

Ein **Beispiel**:

Eingabe: R P P 2
 Ausgabe: R P 2 2
 Aktionen: Retain,
 Delete,
 Retain,
 Insert 2,
 Retain.

Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$p = [Delete, Insert X, Retain]$
 $q = [Retain, Insert Y, Delete]$
 $compose\ p\ q = [Delete, Insert X, Insert Y, Delete]$

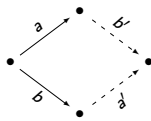
- ▶ $compose$ ist partiell.

- ▶ **Äquivalenz** von Operationen:

$compose\ p\ q \cong [Delete, Delete, Insert X, Insert Y]$

Operationen Transformieren

- ▶ Transformation



- ▶ Beispiel:

$a = [Insert\ X, Retain, Delete]$
 $b = [Delete, Retain, Insert\ Y]$
 $transform\ a\ b = ([Insert\ X, Delete, Retain], [Retain, Delete, Insert\ Y])$

Operationen Verteilen

- ▶ Wir haben die Funktion *transform* die zwei nicht-kommutativen Operationen *a* und *b* zu kommutierenden Gegenständen *a'* und *b'* transformiert.

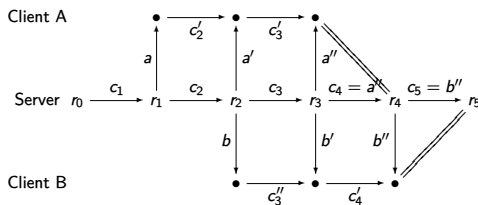
- ▶ Was machen wir jetzt damit?

- ▶ Kontrollalgorithmus nötig

Der Server

- ▶ Zweck:

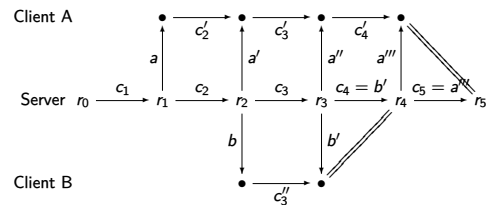
- ▶ Nebenläufige Operationen sequenzialisieren
- ▶ Transformierte Operationen verteilen



Der Server

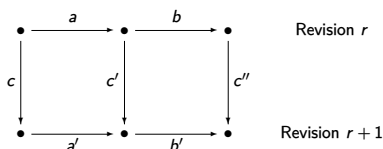
- ▶ Zweck:

- ▶ Nebenläufige Operationen sequenzialisieren
- ▶ Transformierte Operationen verteilen



Der Client

- ▶ Zweck: Operationen puffern während eine Bestätigung aussteht



Zusammenfassung

- ▶ Strikte Konsistenz in verteilten Systemen nicht erreichbar
- ▶ Eventual Consistency
 - ▶ Wenn **längere Zeit** keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im **gleichen Zustand**.
- ▶ Strong Eventual Consistency
 - ▶ Wenn zwei Knoten die **gleiche Menge** Updates beobachten befinden sie sich im **gleichen Zustand**.
- ▶ Operational Transformation
 - ▶ Strong Eventual Consistency auch ohne kommutative Operationen
- ▶ Nächste Woche: Kommutative Replizierte Datentypen (CRDTs)