

Reaktive Programmierung  
Vorlesung 14 vom 26.06.19  
Eventual Consistency

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

17.06.21 2019-07-10

1 [31]



## Fahrplan

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

RP SS 2019

2 [31]



## Heute

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

RP SS 2019

3 [31]



## Was ist eigentlich Konsistenz?

- ▶ Konsistenz = **Widerspruchsfreiheit**
- ▶ In der Logik:
  - ▶ Eine Formelmenge  $\Gamma$  ist konsistent wenn:  $\exists A. \neg(\Gamma \vdash A)$
- ▶ In einem verteilten System:
  - ▶ Redundante (verteilte) Daten
  - ▶ **Globale** Widerspruchsfreiheit?

RP SS 2019

4 [31]



## 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.
- ▶ In echten verteilten Systemen **nicht implementierbar**.

RP SS 2019

5 [31]



## Sequenzielle Konsistenz

### Sequenzielle Konsistenz

- ▶ Zustand nach verteilter Programmausführung = Zustand nach einer äquivalenten sequentiellen Ausführung in einem Prozess.
- ▶ Jeder Prozess sieht die selbe Folge von Operationen.

RP SS 2019

6 [31]



## Eventual Consistency

### Eventual Consistency

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

- ▶ Beispiel: DNS

RP SS 2019

7 [31]



## Strong Eventual Consistency

- ▶ Eventual Consistency ist eine **informelle** Anforderung.
  - ▶ Abfragen können beliebige Werte zurückgeben bevor die Knoten konvergieren.
  - ▶ Keine Sicherheit!
- ▶ **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 eingeecheckt hat, dann haben alle Nutzer die gleiche Sicht auf den *head*.

RP SS 2019

8 [31]



## 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 neueste Version** sehen.
- ▶ Siehe Google Docs, Etherpad und co.



## 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*



## Conflict-Free Replicated Data Types

- ▶ Konfliktfreie replizierte Datentypen
- ▶ Garantieren
  - ▶ Strong Eventual Consistency
  - ▶ Monotonie
  - ▶ Konfliktfreiheit
- ▶ Zwei Klassen:
  - ▶ Zustandsbasierte CRDTs
  - ▶ Operationsbasierte CRDTs



## Zustandsbasierte CRDTs

- ▶ Konvergente replizierte Datentypen (CvRDTs)
- ▶ Knoten senden ihren gesamten Zustand an andere Knoten.
- ▶ Nur bestimmte Operationen auf dem Datentypen erlaubt (*update*).
- ▶ Eine **kommutative, assoziative, idempotente** merge-Funktion
  - ▶ Funktioniert gut mit Gossiping-Protokollen
  - ▶ Nachrichtenverlust unkritisch



## CvRDT: Zähler

- ▶ Einfacher CvRDT
  - ▶ Zustand:  $P \in \mathbb{N}$ , Datentyp:  $\mathbb{N}$ 
    - $query(P) = P$
    - $update(P, +, m) = P + m$
    - $merge(P_1, P_2) = \max(P_1, P_2)$
- ▶ Wert kann nur größer werden.



## CvRDT: PN-Zähler

- ▶ Gängiges Konzept bei CRDTs: Komposition
- ▶ Aus zwei Zählern kann ein komplexerer Typ **zusammengesetzt** werden:
  - ▶ Zähler P (Positive) und Zähler N (Negative)
  - ▶ Zustand:  $(P, N) \in \mathbb{N} \times \mathbb{N}$ , Datentyp:  $\mathbb{Z}$ 
    - $query((P, N)) = query(P) - query(N)$
    - $update((P, N), +, m) = (update(P, +, m), N)$
    - $update((P, N), -, m) = (P, update(N, +, m))$
    - $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$



## CvRDT: Mengen

- ▶ Ein weiterer einfacher CRDT:
  - ▶ Zustand:  $P \in \mathcal{P}(A)$ , Datentyp:  $\mathcal{P}(A)$ 
    - $query(P) = P$
    - $update(P, +, a) = P \cup \{a\}$
    - $merge(P_1, P_2) = P_1 \cup P_2$
- ▶ Die Menge kann nur wachsen.



## CvRDT: Zwei-Phasen-Mengen

- ▶ Durch Komposition kann wieder ein komplexerer Typ entstehen.
- ▶ Menge P (Hinzugefügte Elemente) und Menge N (Gelöschte Elemente)
- ▶ Zustand:  $(P, N) \in \mathcal{P}(A) \times \mathcal{P}(A)$ , Datentyp:  $\mathcal{P}(A)$ 
  - $query((P, N)) = query(P) \setminus query(N)$
  - $update((P, N), +, m) = (update(P, +, m), N)$
  - $update((P, N), -, m) = (P, update(N, +, m))$
  - $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

RP SS 2019

17 [31]



## Operationsbasierte CRDTs

- ▶ Kommutative replizierte Datentypen (CmRDTs)
- ▶ Knoten senden nur **Operationen** an andere Knoten
- ▶ *update* unterscheidet zwischen lokalem und externem Effekt.
- ▶ Netzwerkprotokoll wichtig
- ▶ Nachrichtenverlust führt zu Inkonsistenzen
- ▶ Kein *merge* nötig
- ▶ Kann die übertragenen **Datenmengen** erheblich **reduzieren**

RP SS 2019

18 [31]



## CmRDT: Zähler

- ▶ Zustand:  $P \in \mathbb{N}$ , Typ:  $\mathbb{N}$
- ▶  $query(P) = P$
- ▶  $update(+, n)$ 
  - ▶ lokal:  $P := P + n$
  - ▶ extern:  $P := P + n$

RP SS 2019

19 [31]



## CmRDT: Last-Writer-Wins-Register

- ▶ Zustand:  $(x, t) \in X \times timestamp$
- ▶  $query((x, t)) = x$
- ▶  $update(=, x')$ 
  - ▶ lokal:  $(x, t) := (x', now())$
  - ▶ extern: *if*  $t < t'$  *then*  $(x, t) := (x', t')$

RP SS 2019

20 [31]



## Vektor-Uhren

- ▶ Im LWW Register benötigen wir Timestamps
  - ▶ Kausalität muss erhalten bleiben
  - ▶ Timestamps müssen eine total 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.

RP SS 2019

21 [31]



## Operational Transformation

- ▶ Die CRDTs die wir bis jetzt kennengelernt haben sind recht einfach
- ▶ Das Texteditor Beispiel ist damit noch nicht umsetzbar
- ▶ Kommutative Operationen auf einer Sequenz von Buchstaben?
  - ▶ Einfügen möglich (totale Ordnung durch Vektoruhren)
  - ▶ Wie Löschen?

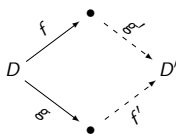
RP SS 2019

22 [31]



## Operational Transformation

- ▶ Idee: Nicht-kommutative Operationen transformieren



- ▶ Für *transform* muss gelten:

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

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

RP SS 2019

23 [31]



## 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

Ein **Beispiel**:

Eingabe: R 1 P 7  
Ausgabe: R P 1 7  
Aktionen: *Retain*,  
*Delete*,  
*Retain*,  
*Insert 1*,  
*Retain*.

- ▶ Operationen sind **partiell**.

RP SS 2019

24 [31]



## Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**
  - ▶ Keine einfache Konkatenation!
- ▶ Beispiel:
 
$$p = [\text{Delete}, \text{Insert X}, \text{Retain}]$$

$$q = [\text{Retain}, \text{Insert Y}, \text{Delete}]$$

$$\text{compose } p \ q = [\text{Delete}, \text{Insert X}, \text{Insert Y}, \text{Delete}]$$
- ▶ *compose* ist partiell.
- ▶ **Äquivalenz** von Operationen:
 
$$\text{compose } p \ q \cong [\text{Delete}, \text{Delete}, \text{Insert X}, \text{Insert Y}]$$

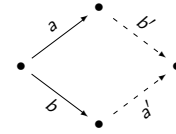
RP SS 2019

25 [31]



## Operationen Transformieren

- ▶ Transformation



- ▶ Beispiel:
 
$$a = [\text{Insert X}, \text{Retain}, \text{Delete}]$$

$$b = [\text{Delete}, \text{Retain}, \text{Insert Y}]$$

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

RP SS 2019

26 [31]



## 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

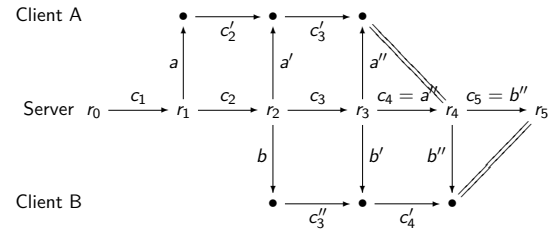
RP SS 2019

27 [31]



## Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



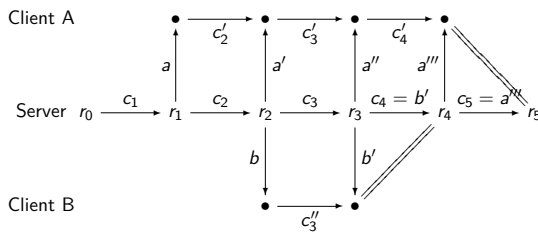
RP SS 2019

28 [31]



## Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



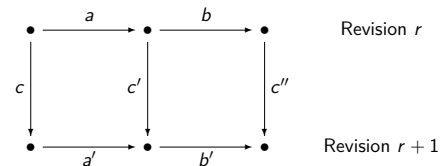
RP SS 2019

29 [31]



## Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



RP SS 2019

30 [31]



## Zusammenfassung

- ▶ Strikte Konsistenz in verteilten Systemen nicht erreichbar
- ▶ Strong Eventual Consistency
  - ▶ Wenn **längere Zeit** keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im **gleichen Zustand**.
  - ▶ Wenn zwei Knoten die **gleiche Menge** Updates beobachten befinden sie sich im **gleichen Zustand**.
- ▶ Conflict-Free replicated Data Types:
  - ▶ Zustandsbasiert: CvRDTs
  - ▶ Operationsbasiert: CmRDTs
- ▶ Operational Transformation
  - ▶ Strong Eventual Consistency auch ohne kommutative Operationen

RP SS 2019

31 [31]

