

Reaktive Programmierung
Vorlesung 14 vom 21.06.17: Eventual Consistency

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

Fahrplan

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

Heute

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

Was ist eigentlich Konsistenz?

- ▶ Konsistenz = **Widerspruchsfreiheit**
- ▶ In der Logik:
 - ▶ Eine Formelmenge Γ ist konsistent wenn: $\exists A. \neg(\Gamma \vdash A)$
- ▶ In einem verteilten 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.
- ▶ In echten verteilten Systemen **nicht implementierbar**.

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.

Eventual Consistency

Eventual Consistency

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

- ▶ Beispiel: DNS

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 eingchecked 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 **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}

$$\text{query}(P) = P$$

$$\text{update}(P, +, m) = P + m$$

$$\text{merge}(P_1, P_2) = \max(P_1, P_2)$$

- ▶ 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}

$$\text{query}((P, N)) = \text{query}(P) - \text{query}(N)$$

$$\text{update}((P, N), +, m) = (\text{update}(P, +, m), N)$$

$$\text{update}((P, N), -, m) = (P, \text{update}(N, +, m))$$

$$\text{merge}((P_1, N_1), (P_2, N_2)) = (\text{merge}(P_1, P_2), \text{merge}(N_1, N_2))$$

CvRDT: Mengen

- ▶ Ein weiterer einfacher CRDT:
 - ▶ Zustand: $P \in \mathcal{P}(A)$, Datentyp: $\mathcal{P}(A)$

$$\text{query}(P) = P$$

$$\text{update}(P, +, a) = P \cup \{a\}$$

$$\text{merge}(P_1, P_2) = P_1 \cup P_2$$

- ▶ 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: $\mathcal{P}(A)$

$$\mathit{query}((P, N)) = \mathit{query}(P) \setminus \mathit{query}(N)$$

$$\mathit{update}((P, N), +, m) = (\mathit{update}(P, +, m), N)$$

$$\mathit{update}((P, N), -, m) = (P, \mathit{update}(N, +, m))$$

$$\mathit{merge}((P_1, N_1), (P_2, N_2)) = (\mathit{merge}(P_1, P_2), \mathit{merge}(N_1, N_2))$$

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**

CmRDT: Zähler

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

CmRDT: Last-Writer-Wins-Register

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

Vektor-Uhren

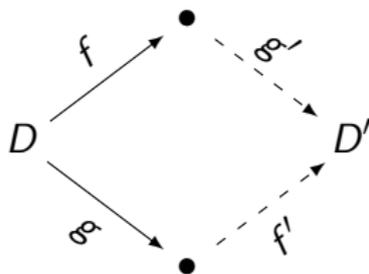
- ▶ Im LWW Register benötigen wir Timestamps
 - ▶ Kausalität muss erhalten bleiben
 - ▶ Timestamps müssen eine total Ordnung haben
- ▶ Datum und Uhrzeit ungeeignet
- ▶ Lösung: Vektor-Uhren
 - ▶ Jeder Knoten hat einen Zähler, der bei Operationen hochgesetzt wird
 - ▶ Zusätzlich merkt sich jeder Knoten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

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?

Operational Transformation

- ▶ Idee: Nicht-kommutative Operationen transformieren



- ▶ Für *transform* muss gelten:

$$\begin{aligned} \text{transform } f \ g = \langle f', g' \rangle &\implies g' \circ f = f' \circ g \\ \text{applyOp } (g \circ f) \ D &= \text{applyOp } g \ (\text{applyOp } f \ D) \end{aligned}$$

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 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:

Aktionen:

Operationen für Text

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

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

Eine **Operation** ist eine Sequenz von Aktionen

Eingabe: 1 P 7
Ausgabe: R
Aktionen: *Retain*,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen: Ein Beispiel:

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

Eine **Operation** ist eine Sequenz von Aktionen

Eingabe: P 7
Ausgabe: R
Aktionen: *Retain*,
Delete,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen: Ein Beispiel:

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

Eine **Operation** ist eine Sequenz von Aktionen

Eingabe: 7
Ausgabe: R P
Aktionen: *Retain*,
Delete,
Retain,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen: Ein Beispiel:

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

Eine **Operation** ist eine Sequenz von Aktionen

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

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:

Ausgabe: R P 1 7

Aktionen: *Retain*,
Delete,
Retain,
Insert 1,
Retain.

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen: Ein Beispiel:

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

Eine **Operation** ist eine Sequenz von Aktionen

- ▶ Operationen sind **partiell**.

Eingabe:

Ausgabe: R P 1 7

Aktionen: *Retain,*
Delete,
Retain,
Insert 1,
Retain.

Operationen Komponieren

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

- ▶ Keine einfache Konkatination!

- ▶ Beispiel:

$$p = [Delete, Insert X, Retain]$$

$$q = [Retain, Insert Y, Delete]$$

$$compose\ p\ q =$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

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

Operationen Komponieren

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

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [\textit{Insert X}, \textit{Retain}]$$

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

$$\textit{compose } p \ q = [\textit{Delete},$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

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

Operationen Komponieren

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

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [\textit{Retain}]$$

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

$$\textit{compose } p \ q = [\textit{Delete}, \textit{Insert X},$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

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

Operationen Komponieren

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

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [\textit{Retain}]$$

$$q = [\textit{Delete}]$$

$$\textit{compose } p \ q = [\textit{Delete}, \textit{Insert } X, \textit{Insert } Y,$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

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

Operationen Komponieren

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

- ▶ Keine einfache Konkatination!

- ▶ Beispiel:

$$p = []$$

$$q = []$$

$$\text{compose } p \ q = [\textit{Delete}, \textit{Insert X}, \textit{Insert Y}, \textit{Delete}]$$

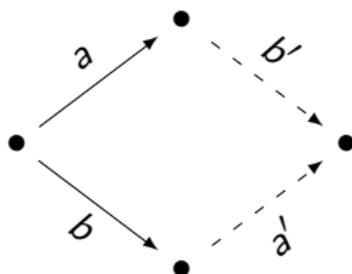
- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

$$\text{compose } p \ q \cong [\textit{Delete}, \textit{Delete}, \textit{Insert X}, \textit{Insert Y}]$$

Operationen Transformieren

► Transformation



► Beispiel:

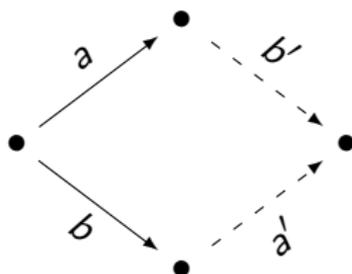
$a = [\textit{Insert X}, \textit{Retain}, \textit{Delete}]$

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

$\textit{transform a b} = ([$
 $\quad , [$
 $\quad]$
 $\quad)$

Operationen Transformieren

► Transformation



► Beispiel:

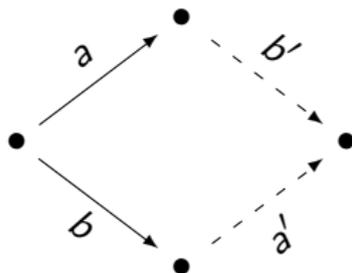
$a = [Retain, Delete]$

$b = [Delete, Retain, Insert Y]$

$transform\ a\ b = ([Insert\ X,$
 $\quad\quad\quad , [Retain,$
 $\quad\quad\quad)$

Operationen Transformieren

► Transformation



► Beispiel:

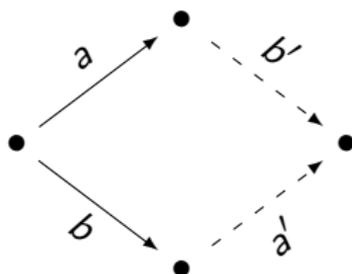
$a = [Delete]$

$b = [Retain, Insert Y]$

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

Operationen Transformieren

- ▶ Transformation



- ▶ Beispiel:

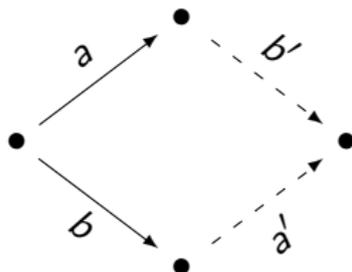
$a = []$

$b = [Insert\ Y]$

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

Operationen Transformieren

► Transformation



► Beispiel:

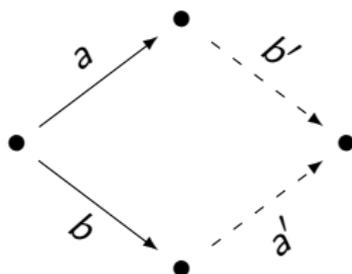
$$a = []$$

$$b = []$$

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

Operationen Transformieren

► Transformation



► Beispiel:

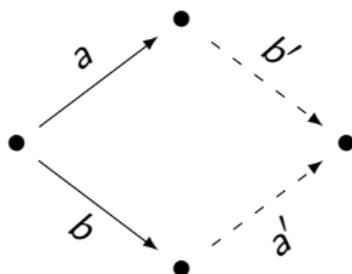
$$a = []$$

$$b = []$$

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

Operationen Transformieren

► Transformation



► Beispiel:

$a = [\textit{Insert X}, \textit{Retain}, \textit{Delete}]$

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

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

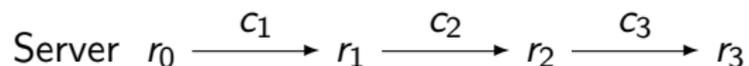
Operationen Verteilen

- ▶ 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

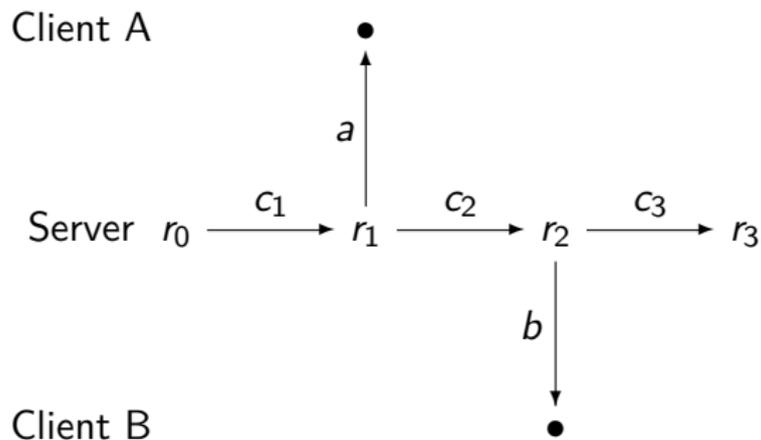
Client A



Client B

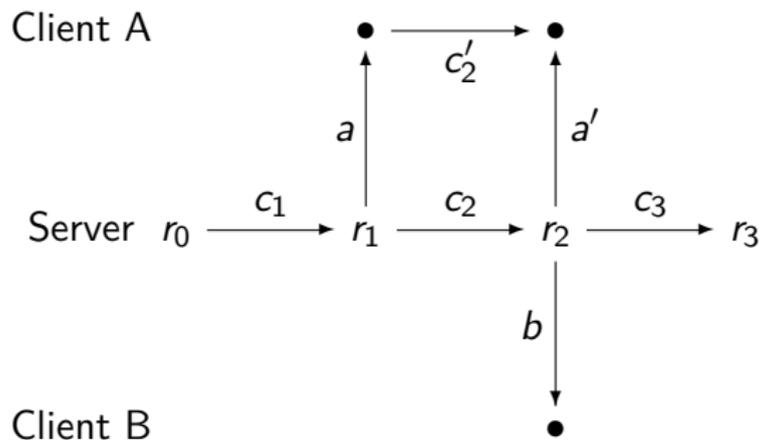
Der Server

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



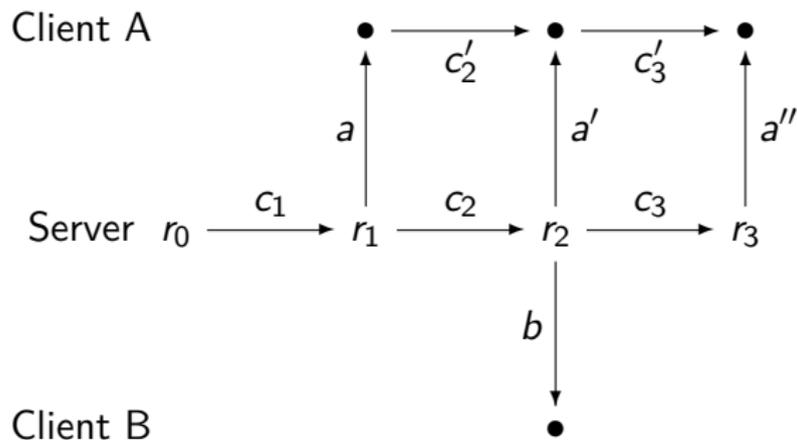
Der Server

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



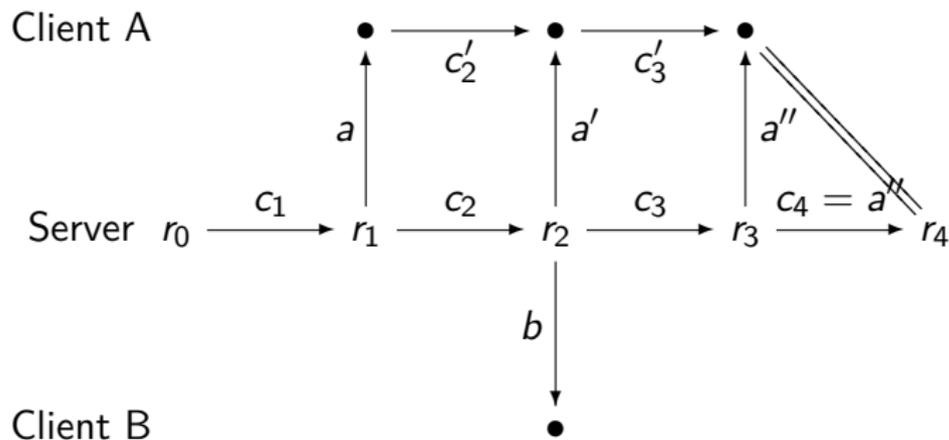
Der Server

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



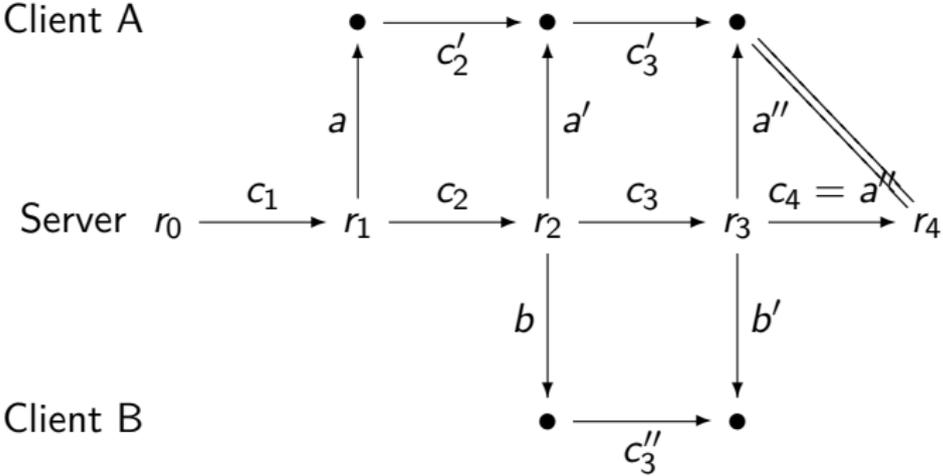
Der Server

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



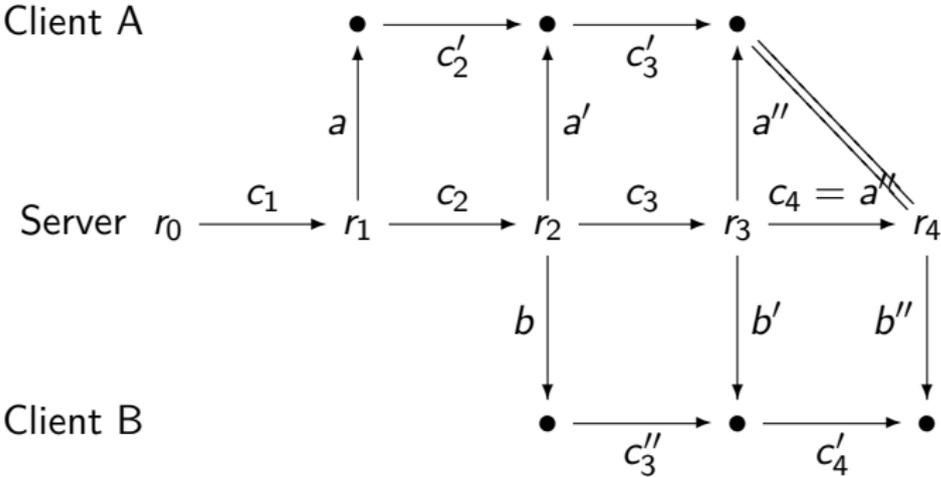
Der Server

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



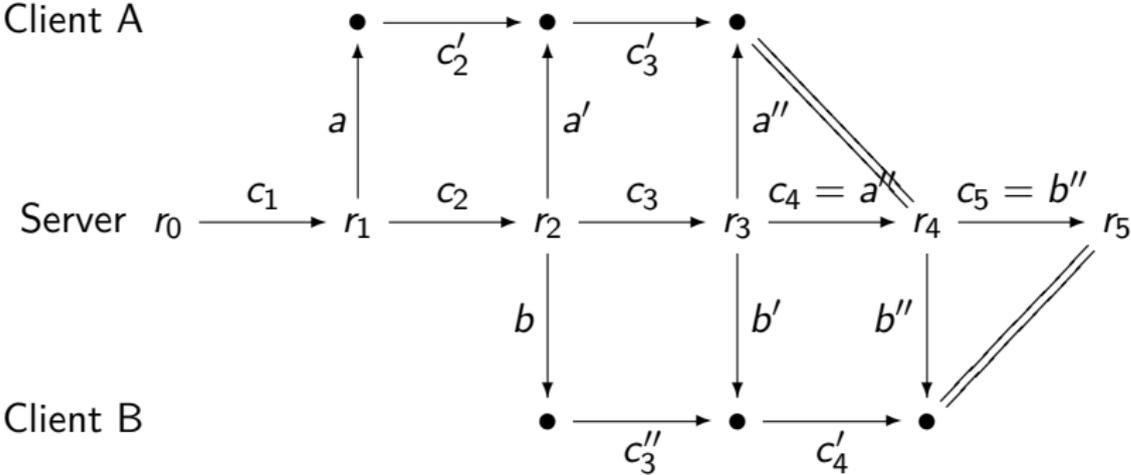
Der Server

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



Der Server

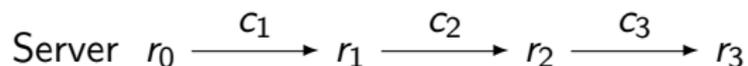
- ▶ Zweck:
 - ▶ Nebenläufige Operationen sequentialisieren
 - ▶ Transformierte Operationen verteilen



Der Server

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

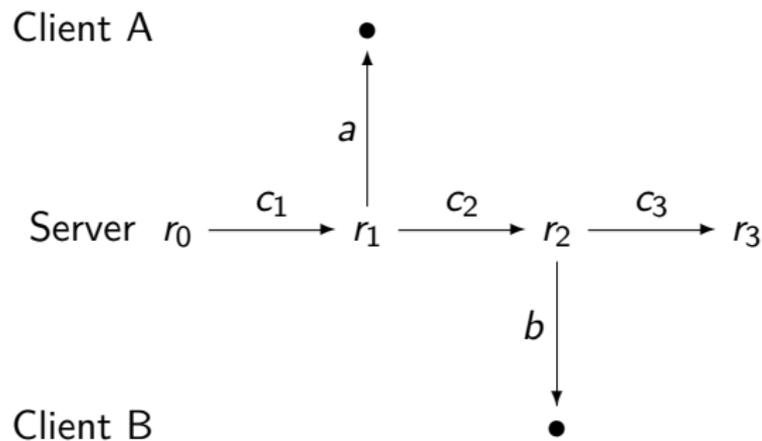
Client A



Client B

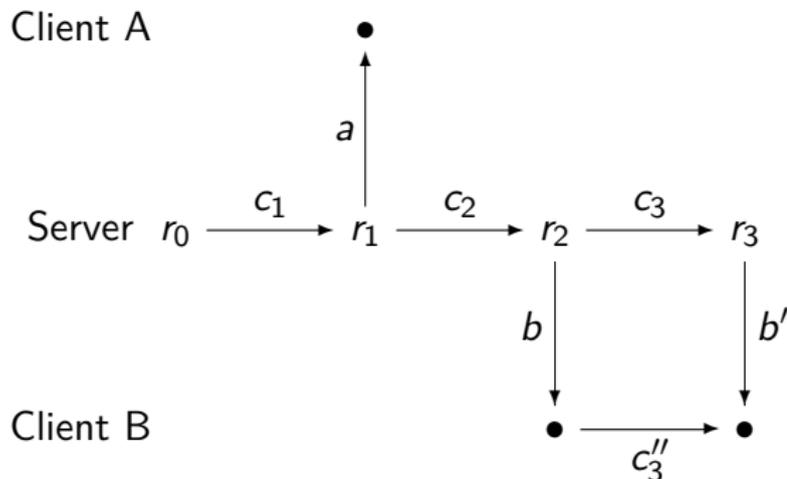
Der Server

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



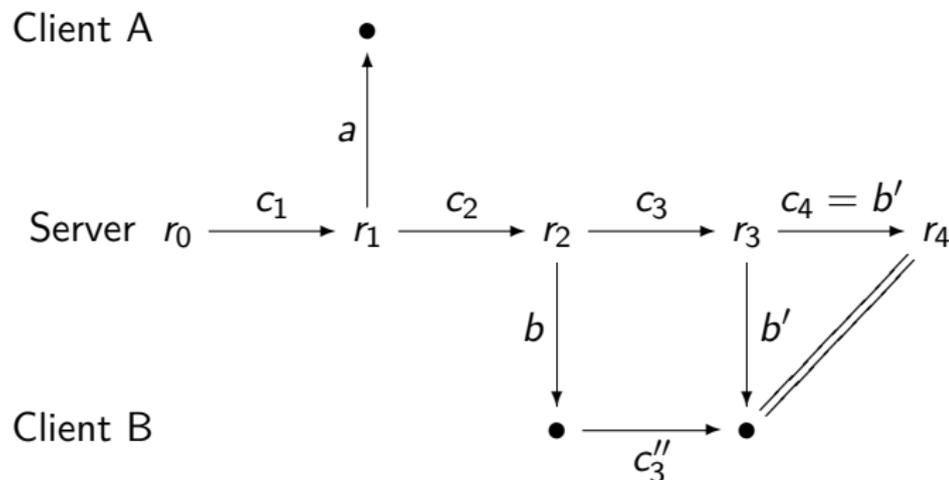
Der Server

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



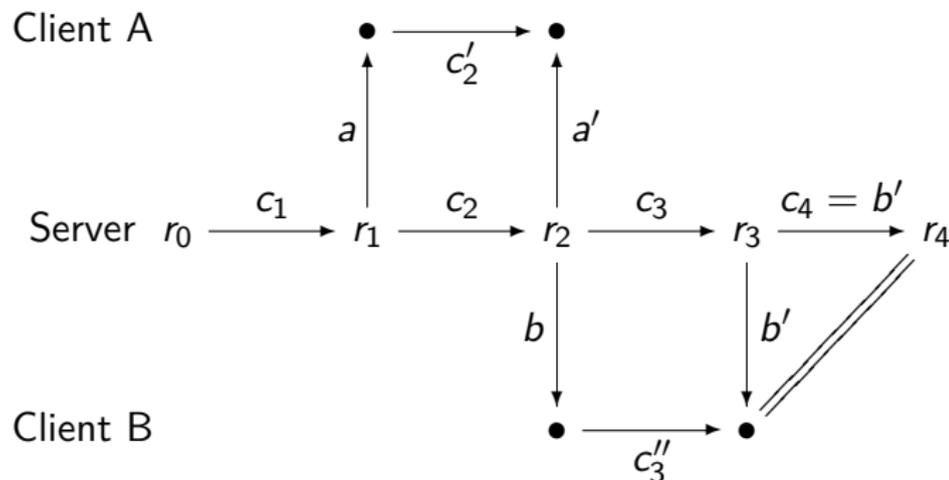
Der Server

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



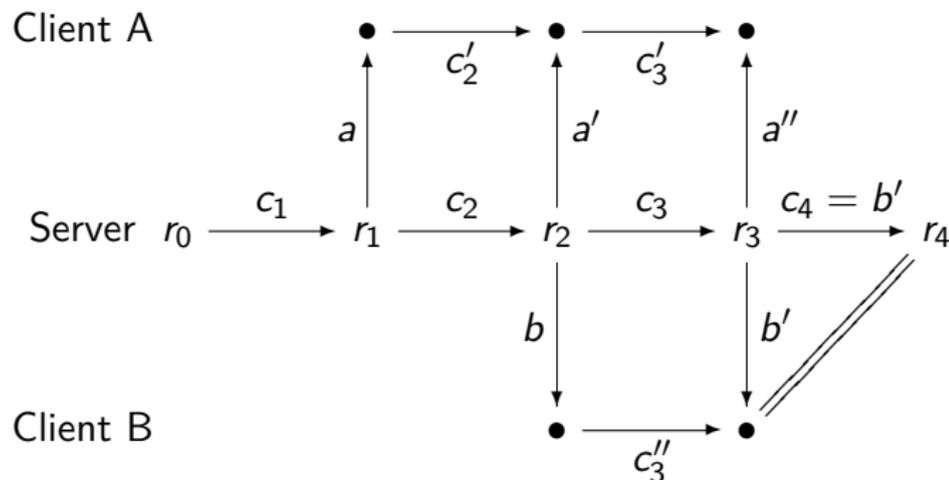
Der Server

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



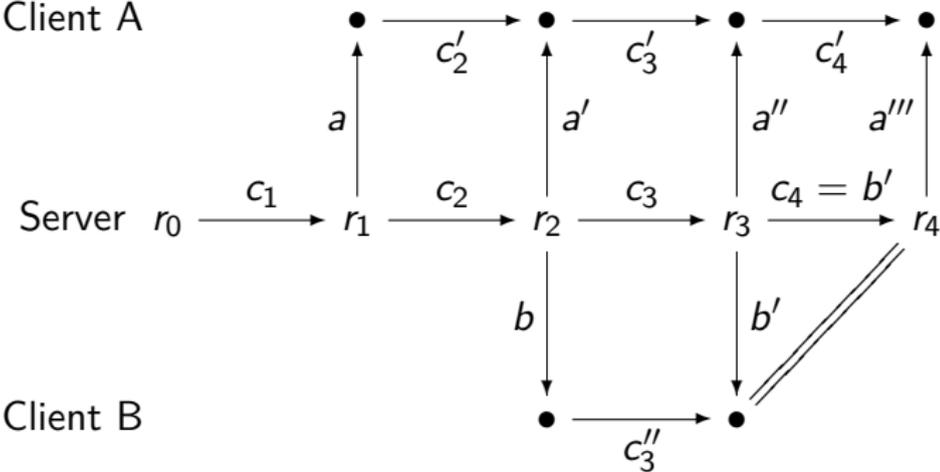
Der Server

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



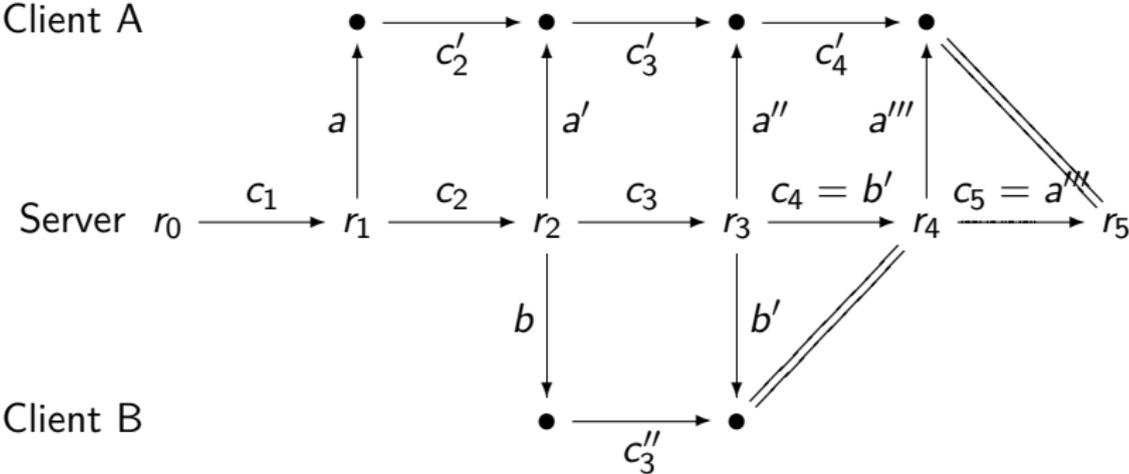
Der Server

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



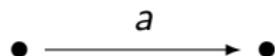
Der Server

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



Der Client

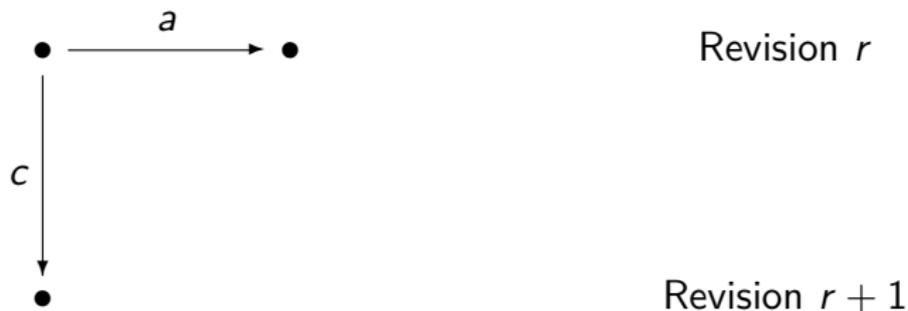
- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



Revision *r*

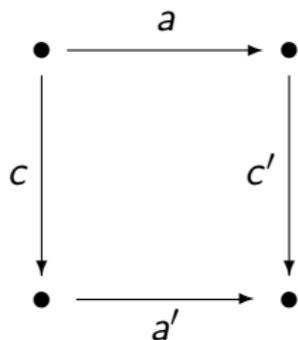
Der Client

- ▶ Zweck: Operationen puffern während eine Bestätigung aussteht



Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht

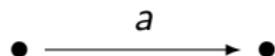


Revision r

Revision $r + 1$

Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



Revision *r*

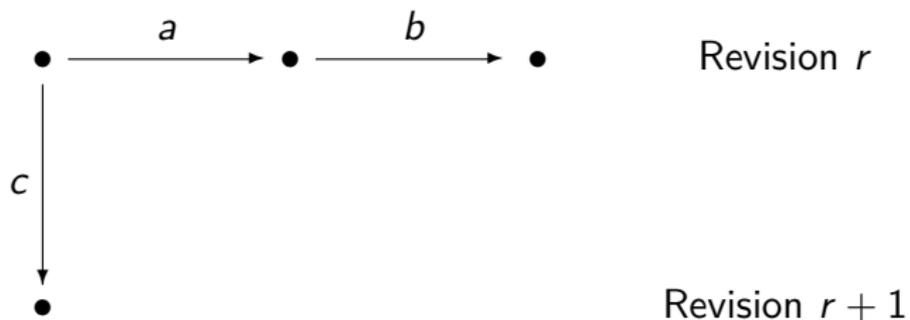
Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



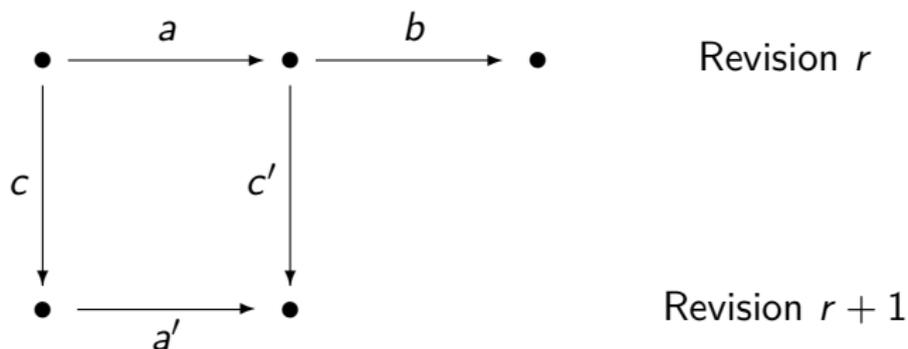
Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



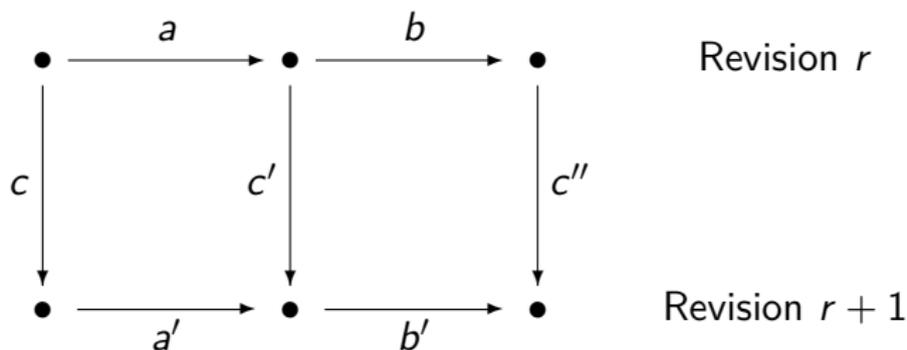
Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



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