Reaktive Programmierung
Vorlesung 12 vom 05.07.2022
Konfliktfreie Replizierte Datentypen

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

Rückblick: 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 Zeitpunk 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 eingecheckt 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).

Partielle Ordnungen

- Was bedeutet, dass ein Effekt erhalten bleibt?
- Unsere Daten müssen **größer** werden
- Semantik definierbar

Partielle Ordnung
Eine Partielle Ordnung ist eine Relation die **reflexiv**, **anti-symmetrisch** und **transitiv** ist.

Conflict-Free Replicated Data Types

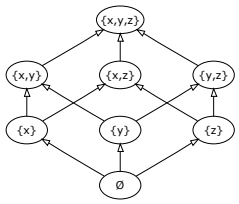
- Konfliktfreie replizierte Datentypen
- Garantieren
  - Strong Eventual Consistency
  - Monotonie
  - Konfliktfreiheit
- Zwei Klassen:
  - Zustandsbasierte CRDTs
  - Operationsbasierte CRDTs

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

Merge

- Die Merge-Funktion ist **kommutativ**, **assoziativ** und **idempotent**
- Die Merge-Funktion bildet einen Verband (Lattice)



CvRDT: Mengen

- Ein 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 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: $\mathcal{P}(A)$

    $query((P, N)) = query(P) \setminus query(N)$

    $update((P, N), +, m) = (update(P, +, m), N)$

    $update((P, N), -, m) = (P, update(N, +, m))$

    $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

CvRDT: Zähler

- Ein weiterer (vermeintlich) 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.
- Aber: Semantik eines Zählers leider nicht eingehalten

CvRDT: Verteilter Zähler

- Jeder Knoten hat seinen Eigenen Zähler
  - Zustand: $DP \in (id \times \mathbb{N})$, Datentyp: $\mathbb{N}$

    $query(DP) = \sum_{(id,P)} query(P)$

    $update(DP, +, m) = DP \cup update(DP_{id}, +, m)$

    $merge(DP^1, DP^2) = \{(id, merge(DP^1_{id}, DP^2_{id}) \mid (id, \_) \in DP^1 \cup DP^2\}$
- Effekt bleibt erhalten!

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: $\mathbb{Z}$

    $query((P, N)) = query(P) - query(N)$

    $update((P, N), +, m) = (update(P, +, m), N)$

    $update((P, N), -, m) = (P, update(N, +, m))$

    $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

CmRDT: Last-Writer-Wins-Register

- Gegeben eine eindeutigen und total geordneten Timestamp T:
- Zustand: $(X, T) \in X \times timestamp$
- $query((X, T)) = X$
- $update((X, T), write, Y) = (Y, T_{now})$
- $merge((X_1, T_1), (X_2, T_2)) = if\ T_1 > T_2\ then\ (X_1, T_{now})\ else\ (X_2, T_{now})$

Vektor-Uhren

- Im LWW Register benötigen wir Timestamps
  - Kausalität muss erhalten bleiben
  - Timestamps müssen eine 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.
- Die Ordnung ist aber partiell!

Operationsbasierte CRDTs

- Kommutative replizierte Datentypen (CmRDTs)
- Knoten senden nur **Operationen** an andere Knoten
- *update* unterscheidete 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: $(x, t) \in X \times timestamp$
- $query((x, t)) = x$
- $update(=, x')$
  - lokal: $(x, t) := (x', now())$
  - extern: $if\ t < t'\ then\ (x, t) := (x', t')$

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