

# Verteilte Systeme

Prof. Dr. Martin Becke

CaDS - HAW Hamburg

Version 0.9

# Inhalt

- 1 Koordination : Konsens, Synchronisation und Fehler
  - Koordination
  - Konsens
  - Synchronisation
  - Logische Uhren

# Koordination

Prozess mit verschiedenen Zielen

- ▶ Kooperation
- ▶ Informationsaustausch
- ▶ Konsistentes Verhalten
- ▶ Kohärentes Verhalten

# Koordination

## Anforderungen

- ▶ Einigung (Konsensbildung)
- ▶ Synchronisation (Globale Ordnung)
- ▶ Fehlerhandling und Recovery

# Konsens

## Idee

- ▶ Gleiche Sicht auf die Daten
- ▶ Selbst Licht zu langsam (Rechnung im Script)
- ▶ Nicht endgültig lösbar

# CAP Theorem

## Idee

- ▶ Konsistenz (C)
- ▶ Verfügbarkeit (A)
- ▶ Partitionstoleranz (P)
- ▶ Herausforderung : C vs A

# Konsensmodelle

## Ideen Umsetzung

- ▶ Tunable Consistency
- ▶ Quorum-Systeme
- ▶ CRDTs
- ▶ Hybride Ansätze

# Konsensmodelle

## Ideen Definition

- ▶ Keine feste Definition, eher eine Orientierung
- ▶ Datenzentrische Konsistenzmodelle
- ▶ Client-zentrische Konsistenzmodelle
- ▶ Alternative Konsistenzmodelle



# Konsensmodelle

## Datenzentrische Konsistenzmodelle

- ▶ Strong Consistency (Traum)
- ▶ Linearizability Consistency (Realisitisch ?)
- ▶ Sequential Consistency
- ▶ Causal Consistency
- ▶ Eventual Consistency

# Datenzentrische Konsistenzmodelle

## Atomare Konsistenz

- ▶ Gleich Linearizability Consistency, manchmal als Strong Consistency interpretiert
- ▶ Protokolle : Paxos, Raft oder Zab
- ▶ Grundidee : Definition eines Zyklus
- ▶ Beispiel der Anwendung ist Finanztransaktionssystemen

# Datenzentrische Konsistenzmodelle

## Sequentielle Konsistenz

- ▶ Operationen in der gleichen Reihenfolge
- ▶ Synchronisation der Knoten kann verzögert erfolgen
- ▶ Ausgewogene Mischung aus Konsistenz und Leistung
- ▶ Keine Kausalität

# Datenzentrische Konsistenzmodelle

## Kausale Konsistenz

- ▶ Kausalen Beziehungen zwischen Operationen
- ▶ Nur kausale Beziehung bei Operationen mit Abhängigkeit
- ▶ Beispiel Tweet mit Kommentaren und alternativer Tweet
- ▶ Kausaler Konsistenz basiert in Implementierung gerne auf logische Uhren

# Datenzentrische Konsistenzmodelle

## Eventual Consistency

- ▶ Daten vorübergehend inkonsistent (Nicht als eventuell übersetzen)
- ▶ Fokus auf höherer Verfügbarkeit (C)
- ▶ Beispielhaft umgesetzt in DynamoDB
- ▶ Immer höhere Verbreitung

# Client-zentrische Konsistenzmodelle

## Monotonic Reads

- ▶ Liest niemals ältere Daten, die er zuvor gelesen hat
- ▶ Liest gleiche oder neuere Daten unabhängig seiner Position
- ▶ Beispielhaft Zeitstempel oder Versionsnummer bei Read

# Client-zentrische Konsistenzmodelle

## Monotonic Writes

- ▶ Schreibvorgänge in der Reihenfolge des Clients
- ▶ Beispielhaft : Zeitstempeln oder Versionsnummern bei Write
- ▶ Beispiel Banktransaktionen

# Client-zentrische Konsistenzmodelle

Read your Writes

- ▶ Benutzer sieht immer seine Änderungen
- ▶ Beispielhaft eistempeln oder Versionsnummern bei Wrtie
- ▶ Beispiel Statusaktualisierungen bei sozialen Netzen



# Client-zentrische Konsistenzmodelle

Writes follow Reads

- ▶ Kausale Konsistenz und das „Writes follow Reads“-Prinzip sind eng verbunden
- ▶ Schreiboperationen, die auf Leseoperationen basieren, sind in logischer konsistenter Reihenfolge
- ▶ Beispiel ist Cassandra
- ▶ Google Docs verwendet das Operational Transformation (OT) Framework

# Konsistenzmodell

## Alternativen

- ▶ Abweichung ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability)
- ▶ BASE (Basically Available, Soft state, Eventually consistent)
- ▶ PACELC (Partition-tolerance, Availability, Consistency, Eventual consistency, Latency, and Consistency trade-offs)

# Konsistenzmodell

## Alternativen

- ▶ Continuous Consistency
  - ▶ CONIT-Modell
- ▶ Fork Consistency
- ▶ Multidimensional Consistency
- ▶ Adaptable Consistency
- ▶ View Consistency

# Konsistenssichten

## Write Write Conflicts

- ▶ Option 1 : Sperren und Transaktionen
- ▶ Option 2 : Optimistische Nebenläufigkeitskontrolle
- ▶ Option 3 : Versionskontrolle und Zusammenführungsstrategien

# Synchronisation

## Globaler Zustand

- ▶ Herausforderungen der Koordination
- ▶ Bestimmung von Ordnung
- ▶ Bestimmung von Autorität
- ▶ NIEMALS ZURÜCK STELLEN!

# Synchronisation

## Protokolle

- ▶ Remote-write protocols
  - ▶ Auch konfliktfrei möglich ; z.B Journal
- ▶ Local-write protocols

# Synchronisation

## Lösungsstrategien

- ▶ Physikalische Uhren
- ▶ Logische Uhren
- ▶ Locking

# Synchronisation

## Physikalische Uhren

- ▶ Drift Problem
- ▶ Präzise und akkurat
- ▶ Basis häufig Universal Time Coordinated (UTC)



# Physikalische Uhren

## Globale Zeitstempel

- ▶ Network Time Protocol (NTP)
- ▶ Precision Time Protocol (PTP) (HW)
- ▶ Cristian's Algorithm
- ▶ Berkeley Algorithm

# Physikalische Uhren

## Network Time Protocol (NTP)

$$D = (t_4 - t_1) - (t_3 - t_2)$$

$$\theta = \frac{(t_2 - t_1) + (t_3 - t_4)}{2}$$

# Synchronisation

happens-before

- ▶  $e_1$  und  $e_2$  sind Ereignisse im selben Prozess, und  $e_1$  tritt vor  $e_2$  auf.
- ▶  $e_1$  ist das Senden einer Nachricht, und  $e_2$  ist das Empfangen dieser Nachricht.

# Synchronisation

## Logische Uhren

- ▶ Für jeden Prozess  $P_i$  wird die logische Uhr  $C_i$  bei jedem Ereignis, das innerhalb von  $P_i$  auftritt, inkrementiert :  
 $C_i := C_i + 1$ .
- ▶ Bei jedem Nachrichtenaustausch zwischen zwei Prozessen  $P_i$  und  $P_j$  (wobei  $P_i$  die Nachricht sendet und  $P_j$  sie empfängt) gelten die folgenden Regeln :
  - ▶ Die Uhr des sendenden Prozesses  $P_i$  wird vor dem Senden der Nachricht inkrementiert :  $C_i := C_i + 1$ .
  - ▶ Die Uhr des empfangenden Prozesses  $P_j$  wird auf das Maximum der eigenen Uhr und des empfangenen Zeitstempels (plus eins) gesetzt :  $C_j := \max(C_j, C_i + 1)$ .

# Synchronisation

## Lamport Clock

- Jeder Prozess  $P_j$  im verteilten System führt eine logische Uhr  $C_j$  mit. Jedes Mal, wenn ein Prozess eine Transaktion initiiert, wird die logische Uhr  $C_j$  inkrementiert, und der Transaktion  $T_i$  wird der Zeitstempel  $C_j(T_i)$  zugeordnet.

# Synchronisation

## Lamport Clock

- Wenn ein Prozess  $P_j$  eine Transaktion  $T_i$  an einen anderen Prozess  $P_k$  sendet, wird die logische Uhr  $C_j$  inkrementiert und zusammen mit der Transaktion an  $P_k$  gesendet. Bei Empfang der Transaktion wird die logische Uhr  $C_k$  des empfangenden Prozesses auf das Maximum von  $C_k$  und dem empfangenen Zeitstempel (plus eins) gesetzt :  
$$C_k := \max(C_k, C_j(T_i) + 1).$$

# Synchronisation

## Lamport Clock

- Bei der Ausführung von Transaktionen werden die zugeordneten Zeitstempel verwendet, um die relative Ordnung der Transaktionen zu bestimmen und mögliche Konflikte aufzulösen. Zum Beispiel kann ein Konflikt in Form eines schreibgeschützten oder schreibgeschützten Zugriffs auf dieselben Daten auftreten. In solchen Fällen kann der Zeitstempel verwendet werden, um die Transaktionen konsistent zu ordnen, indem die Transaktion mit dem kleineren Zeitstempel vor der Transaktion mit dem größeren Zeitstempel ausgeführt wird.

# Synchronisation

## Vector Clock

- ▶ Jeder Prozess  $P_i$  im verteilten System führt eine Vektoruhr  $V_i$  mit der Länge  $n$  (Anzahl der Prozesse im System) und initialisiert alle Komponenten auf Null.
- ▶ Bei jedem internen Ereignis, das innerhalb von  $P_i$  auftritt, wird die Komponente  $V_i[i]$  der Vektoruhr inkrementiert :  
 $V_i[i] := V_i[i] + 1$ .
- ▶ Bei jedem Nachrichtenaustausch zwischen zwei Prozessen  $P_i$  und  $P_j$  (wobei  $P_i$  die Nachricht sendet und  $P_j$  sie empfängt) gelten die folgenden Regeln :
  - ▶ Die Komponente  $V_i[i]$  der sendenden Vektoruhr wird vor dem Senden der Nachricht inkrementiert :  $V_i[i] := V_i[i] + 1$ .
  - ▶ Die Vektoruhr des empfangenden Prozesses  $P_j$  wird auf das Elementweises Maximum der eigenen Vektoruhr und der empfangenen Vektoruhr gesetzt :  $V_j[k] := \max(V_j[k], V_i[k])$ , für  $k = 1, 2, \dots, n$ .



# Vector Clock

## Ordnung

- ▶  $E_1 \rightarrow E_2$  (happens-before) : Wenn für alle  $k$ ,  $V_1[k] \leq V_2[k]$  und für mindestens ein  $k$ ,  $V_1[k] < V_2[k]$  gilt.
- ▶  $E_1 \parallel E_2$  (konkurrierend) : Wenn für mindestens ein  $k$ ,  $V_1[k] < V_2[k]$  und für mindestens ein  $k$ ,  $V_2[k] < V_1[k]$  gilt.

# Ordnung

## Allgemein

- ▶ Teilweise Ordnung (oder partielle Ordnung)
- ▶ Totale Ordnung
- ▶ Kausale Ordnung

# Synchronisation

## Conflict-Free Replicated Data Types

- ▶ Verteilte Datenstruktur
- ▶ Replikationen von Daten (Algorithmisch)
- ▶ Beispiele : Grow-only Counter (G-Counter) oder 2P-Set (Zwei-Phasen-Set)

# Synchronisation

## Locking

- ▶ Ressourcen sperren
- ▶ *Shared Locks* und *Exclusive Locks*
- ▶ Zentrale (Koordinator) oder De-Zentral (Algorithmisch)
- ▶ Probleme von Deadlocks

# Synchronisation

## Philosophenproblem

- ▶ Deadlocks : Wenn jeder Philosoph gleichzeitig eine Gabel aufnimmt und auf die andere wartet, entsteht ein Deadlock. Kein Philosoph kann mit dem Essen fortfahren, da jeder auf die Freigabe der anderen Gabel wartet.
- ▶ Verhungern (Starvation) : Ein Philosoph könnte verhungern, wenn er immer wieder von anderen Philosophen überstimmt wird, die die Gabeln ergreifen, bevor er sie erreichen kann. In solchen Fällen kommt der betroffene Philosoph möglicherweise nie zum Essen.
- ▶ Leistungsprobleme : Die Leistung des Systems kann beeinträchtigt werden, wenn Philosophen zu lange auf den Zugriff auf Gabeln warten müssen oder wenn ineffiziente Synchronisationsstrategien eingesetzt werden.