Willkommen bei Verteilte Systeme!

Von Datenbanken
über Webdienste
bis zu p2p und Sensornetzen.

Heute: **Datenbanken, CALM und CRDTs**.

Wiederholung

- Probleme mit Uhren
- Synchronisation von Uhren
- Logische Zeit
  - Lamport
  - Vektor
- Mutex
- Wahlalgorithmen

Lernziele heute

- Ihr kennt verschiedene Arten der Replikation.
- Ihr erkennt, dass Replikation zu Inkonsistenzen führen kann.
- Ihr kennt die Aussage des CAP Theorems und misstraut Aussagen wie "DB xy ist CA".
- Ihr kennt die Tradeoffs zwischen Konsistenz und Verfügbarkeit.
- Ihr kennt das CALM Theorem.
- Ihr erkennt, dass Koordination vermieden werden kann und dies zu einfacheren Systemen führt.

Ablauf heute

- Replikation
- CAP Theorem
- Was ist Availability?
- Welche Konsistenzmodelle gibt es?
- Wie können wir prüfen, ob ein verteiltes System ein Konsistenzmodell korrekt umsetzt?
- Lässt sich Koordination vermeiden?

Replikation

Speichern von Kopien auf mehreren Maschinen, die über Netzwerk verbunden sind.

Gründe für Replikation:

- Geographische Skalierung: Daten eines Nutzers näher am Nutzer –> Verringerung der Latenz
- Anwendung funktioniert trotz ausgefallenen Knoten.
- Größenmäßige Skalierung: Mehr Nutzer können die Anwendung gleichzeitig verwenden.

Annahme: Gesamter Datensatz passt auf eine Maschine –> Keine Partitionierung (Sharding)

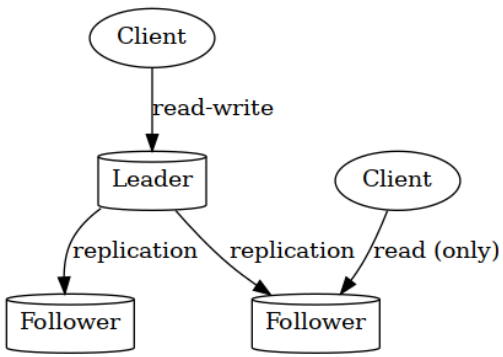
Übersicht Replikation

3 Arten von Replikation werden unterschieden:

- Single Leader
- Multi Leader
- Leaderless

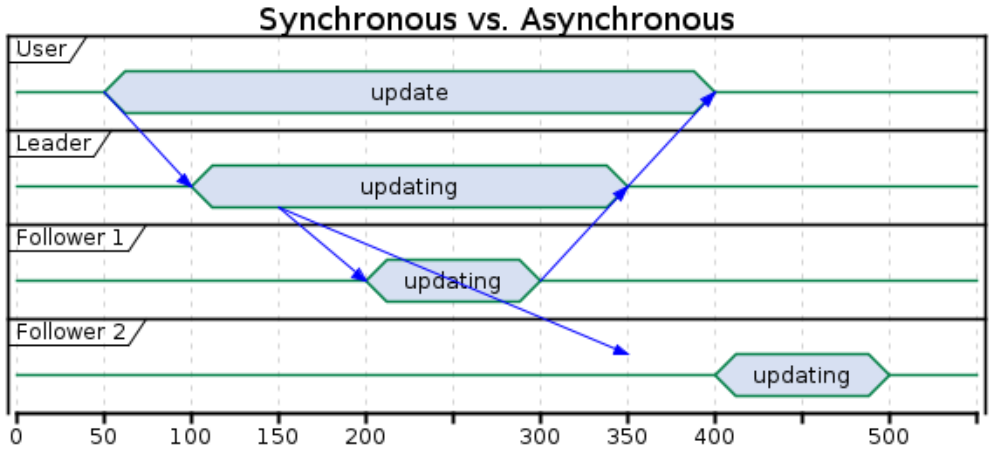
Replikation kann synchron oder asynchron erfolgen.

Single Leader



- Replika: Knoten, der eine Kopie speichert
- Leader: Eine Replika
- Schreiben: Anfrage an Leader
- Leader schreibt lokal
- Sendet geänderte Daten an alle anderen Replikas (Follower)
- Follower speichern die Änderungen lokal
- Lesen auch von Followern

Synchronous vs. Asynchronous



- Replikation zu Follower 1 ist synchron.
- Replikation zu Follower 2 is asynchron.

Was sind die Vor- und Nachteile von (a)synchroner Replikation?

Synchronous vs. Asynchronous 2

Vorteil synchroner Replikation

- Bestätigte writes wurden bereits repliziert.
- Absturz des Leader ohne Datenverlust

Nachteil synchroner Replikation

- Nicht verfügbar bei Absturz von Followern

Kombination synchroner und asynchron üblich

Beispiel:

- 1 synchroner Follower
- Absturz: Neuen synchronen Follower wählen

Replication Logs: Statement Based

Wie werden Änderungen an den Daten übertragen?

Beispiel: SQL

- Leader überträgt Statements (`INSERT`, `UPDATE`, `DELETE`).
- Follower führen Statements aus
- Was tun wenn Statement `NOW` enthält?
- `NOW` könnte vor Übermittlung mit Wert des Leader ersetzt werden.

Weitere Probleme: Autoincrement, Trigger usw.

Replication Logs: Write Ahead Log (WAL)

- SQL Datenbanken verwalten WAL.
- An WAL wird nur angefügt (append-only)
- Einträge in WAL werden von den Followern angewandt

Problem: Struktur des WAL üblicherweise Implementierungsdetail

- Verschiedene Versionen einer DB gleichzeitig?
- Schließt Zero-Downtime-Updates aus
- Logische Logs oft von der Storage Engine entkoppelt

Lag

Zeitverlust

Replikation braucht Zeit: Inkonsistenzen

⇒ Consistency (ab Folie 30)

Multi Leader Replication

Nachteile Single Leader

- Leader nicht erreichbar $\Rightarrow$ keine Änderungen
- Einzelner Leader $\rightarrow$ Flaschenhals

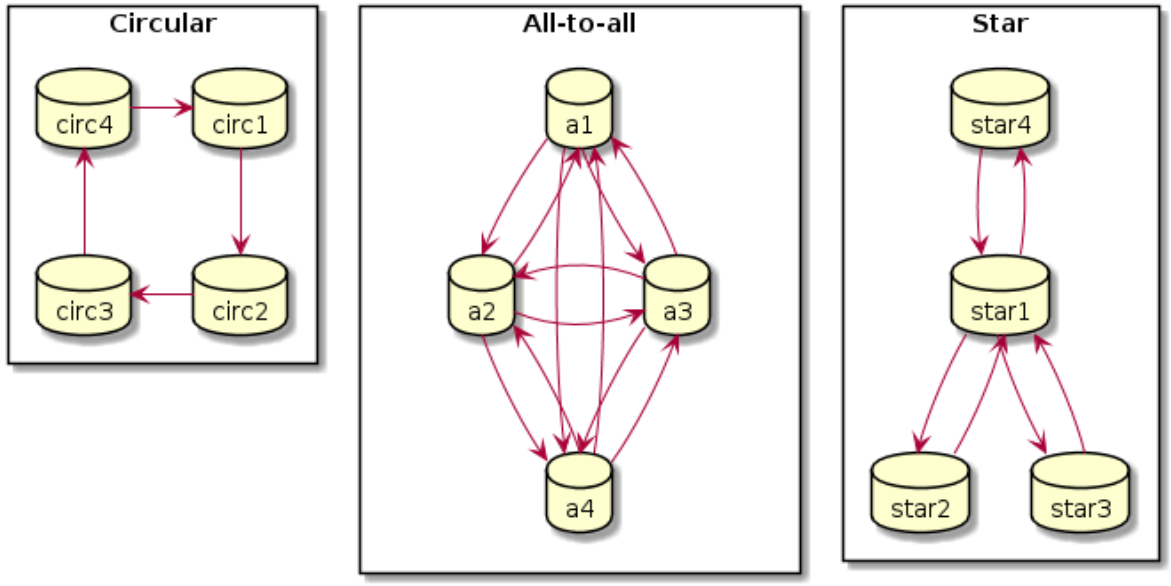
Anwendungen

- Progressive Apps: Offline arbeiten
- Kollaborative Apps: Etherpad, Google Docs etc.

Nachteil Multi Leader

- Lösung von Schreibkonflikten nötig

Topologien



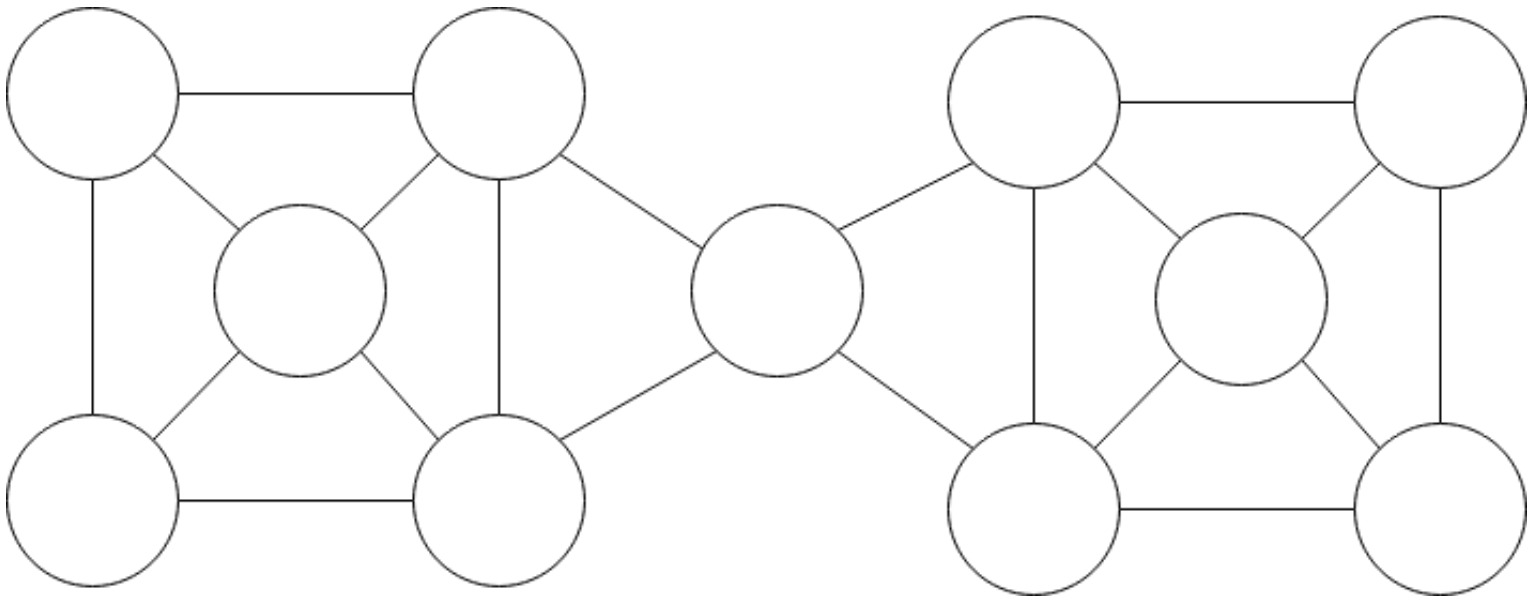
Kennzahlen

Durchmesser Der längste kürzeste Pfad.

Bisektionsbandbreite Kanten zu löschen zum Zerlegen in zwei ähnlich große Teile

Übung Toplogien

Bestimme Durchmesser, Bisektionsbandbreite und Knoten/Kantenkonnektivität für folgende Toplogie:



Leaderless Replication

- Verbreitet durch Amazons Dynamo DB
- Auch Riak, Cassandra, Voldemort
- Writes auf jedem Knoten
- Meist „Quorum“ Reads und Writes.

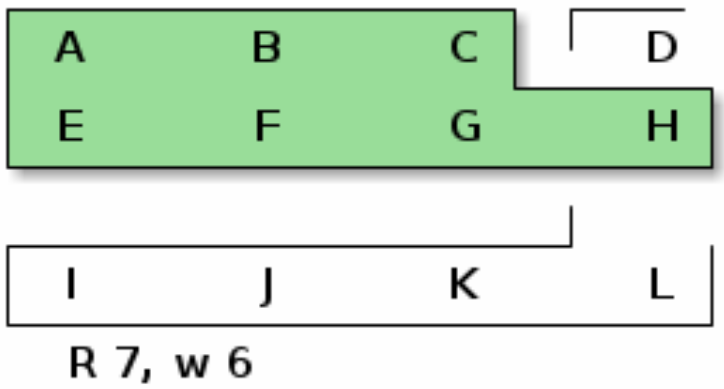
Quorum

- Sende jeden write und read an n Knoten
- **write** ist erfolgreich wenn w Knoten ihn bestätigen
- **read** ist erfolgreich wenn r Knoten ihn bestätigen

Quorum Bedingung: $w + r > n$:

- garantiert Überlapp zwischen w-Knoten und r-Knoten
- $w < n$ kann bei ausgefallenen Knoten schreiben
- $r < n$ kann bei ausgefallenen Knoten lesen
- $w > \frac{n}{2}$ kann write-write Konflikte vermeiden

Quorum: Write-Write-Konflikte vermeiden



- Wenn $w \leq \frac{n}{2}$ können 2 Nutzer widersprüchliche Daten schreiben.
- Beim Lesen erkennbar, da $r > n - w$
- write-write Konflikt oder stale data

Zusammenfassung Replikation

- Single, Multi, Leaderless
- (a)synchrone Replikation
- Inkonsistenzen möglich
- Quorum Bedingung: $r + w > n$

CAP (Consistency, Availability, Partition Tolerance)

A shared-data system can have at most two of the three following properties: consistency, availability, and tolerance to network partitions.

CAPs Consistency (Konsistenz)

CAP meint mit Consistency Linearizability.

- Jede Operation atomar
- Jeder Prozess sieht die gleiche Ordnung von Operationen.
- Diese Ordnung entspricht der realen Ordnung von Operationen.

CAPs Availability (Verfügbarkeit)

- Jede Anfrage an einen verfügbaren Knoten liefert eine Antwort
- Auch bei Netzwerkpartitionen.

CAPs Partition (Teilung)

Eine Netzwerkpartition teilt das System in 2 oder mehr Seiten. Die einzelnen Seiten können nicht miteinander kommunizieren.

Kann man entscheiden, dass keine Netzwerkpartitionen auftreten?

CAP - Bewertung

- Linearizability **OR** Total Availability.
- CAP Theorem wird oft im Marketingsprech verwendet, um Datenbanken zu charakterisieren.
- Konsistenzmodelle neben Linearizability, verschiedene Grade von Verfügbarkeit

Welche Kombinationen sind möglich?

Availability

- Total Available / High Available
- Sticky Available
- Unavailable

Literatur: Highly Available Transactions: Virtues and Limitations Bailis et al. (2013).

Total Available / High Available

- Antwort erhält, wer **einen** korrekten (nicht versagenden) Server kontaktieren kann
- Auch bei Netzwerkpartitionen zwischen Servern

Sticky Available

- Antwort erhält, wer einen Server kontaktieren kann, der den gesamten, **dem Nutzer bekannten Zustand** beinhaltet
- Auch bei Netzwerkpartitionen zwischen Servern

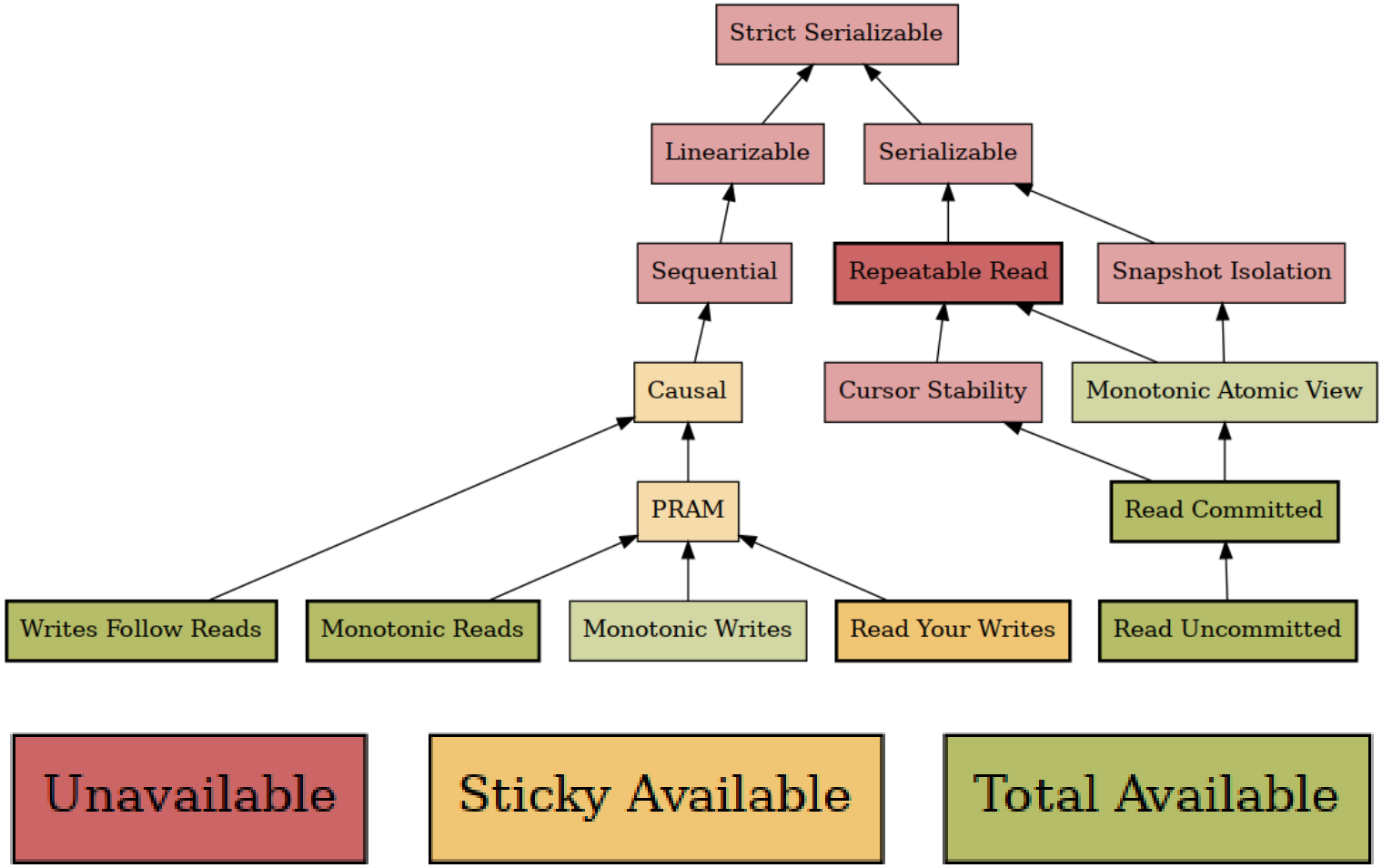
Sticky Available - Beispiel

- Daten auf mehrere Server repliziert
- Jede Replika enthält alle Daten
- Nutzer kontaktiert immer denselben Server
- $\Rightarrow$ Sticky Available

Unavailable

System ist nicht verfügbar bei Netzwerkpartitionen.

Consistency



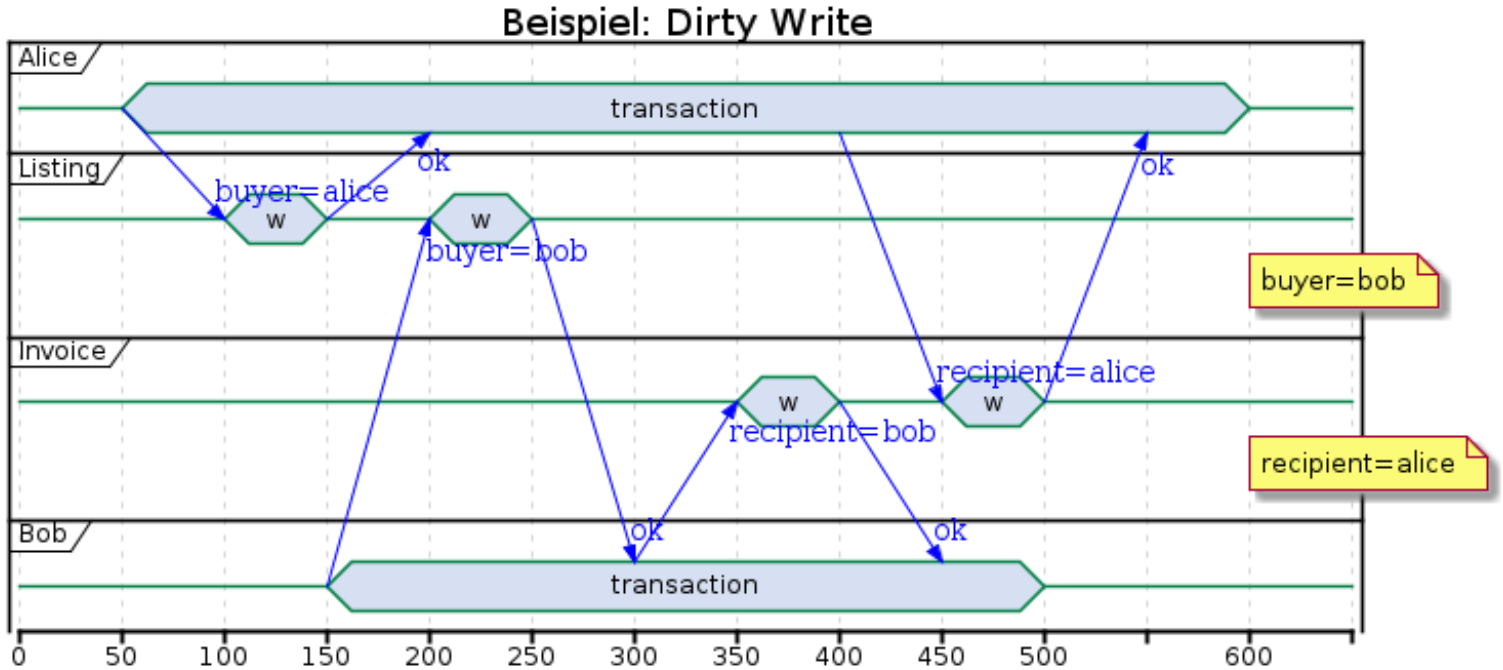
Isolation

Welche Regeln gelten wenn mehrere Transaktionen gleichzeitig stattfinden?

Read Uncommitted

Verhindert dirty writes:

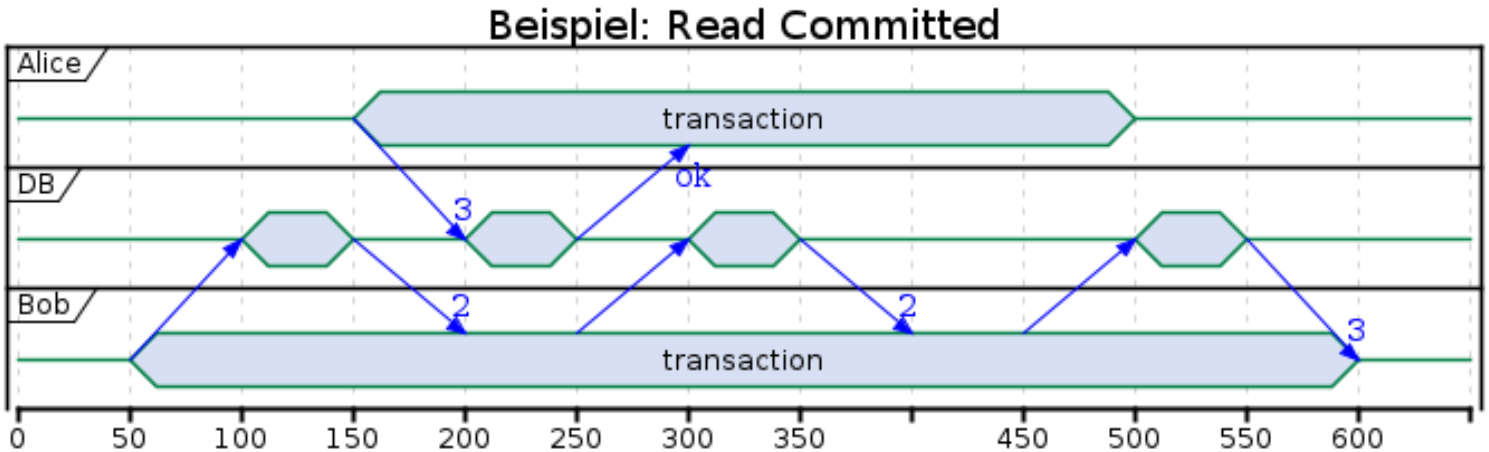
Beim Schreiben in eine DB werden nur Daten überschrieben, die bereits committed wurden.



Read Committed

Verhindert zusätzlich dirty reads:

Beim Lesen einer DB werden nur Werte gelesen, die bereits committed wurden.



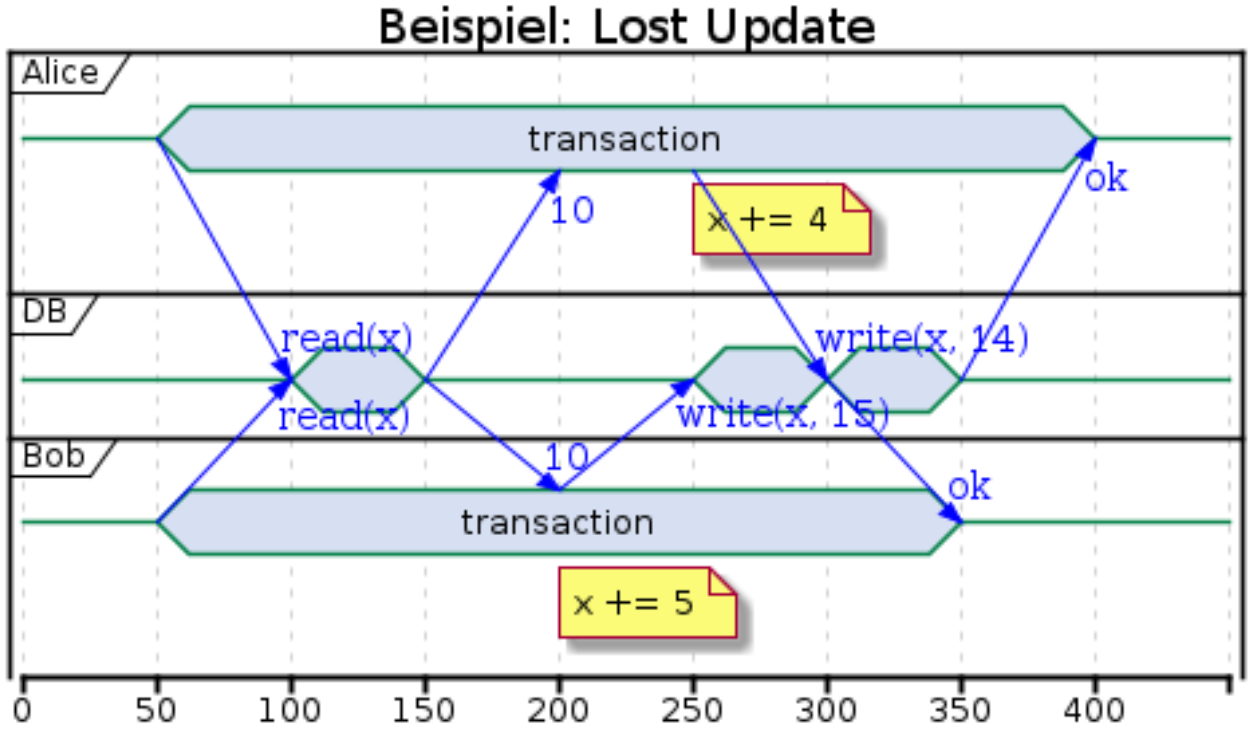
Repeatable Read

Mehrere Definitionen:

- ANSI SQL Standard: Falls eine Transaktion ein Datum mehrfach liest, wird immer derselbe Wert zurückgegeben.
- Literatur: Verbiete zusätzlich Lost Updates.
- Postgres, MySQL: Snapshot Isolation.

Literatur: Weak Consistency: a generalized theory and optimistic implementations for distributed transactions; Adya and Liskov (1999)

Repeatable Read - Lost Update



Repeatable Read - Availability

ANSI:

- High Availability möglich: Clientseitiger Cache für gelesene Werte.

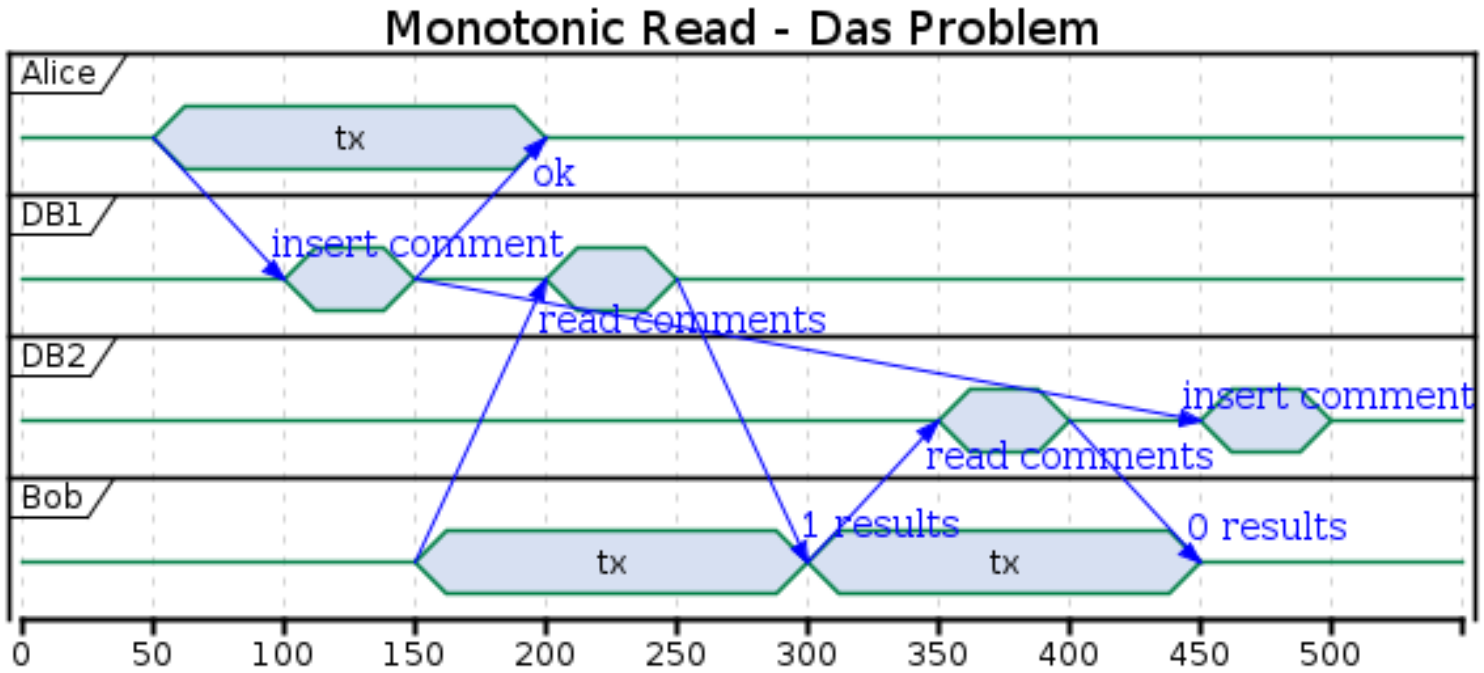
Literatur:

- Unavailable: Verhinderung von Lost Updates benötigt Koordination.

Sessions

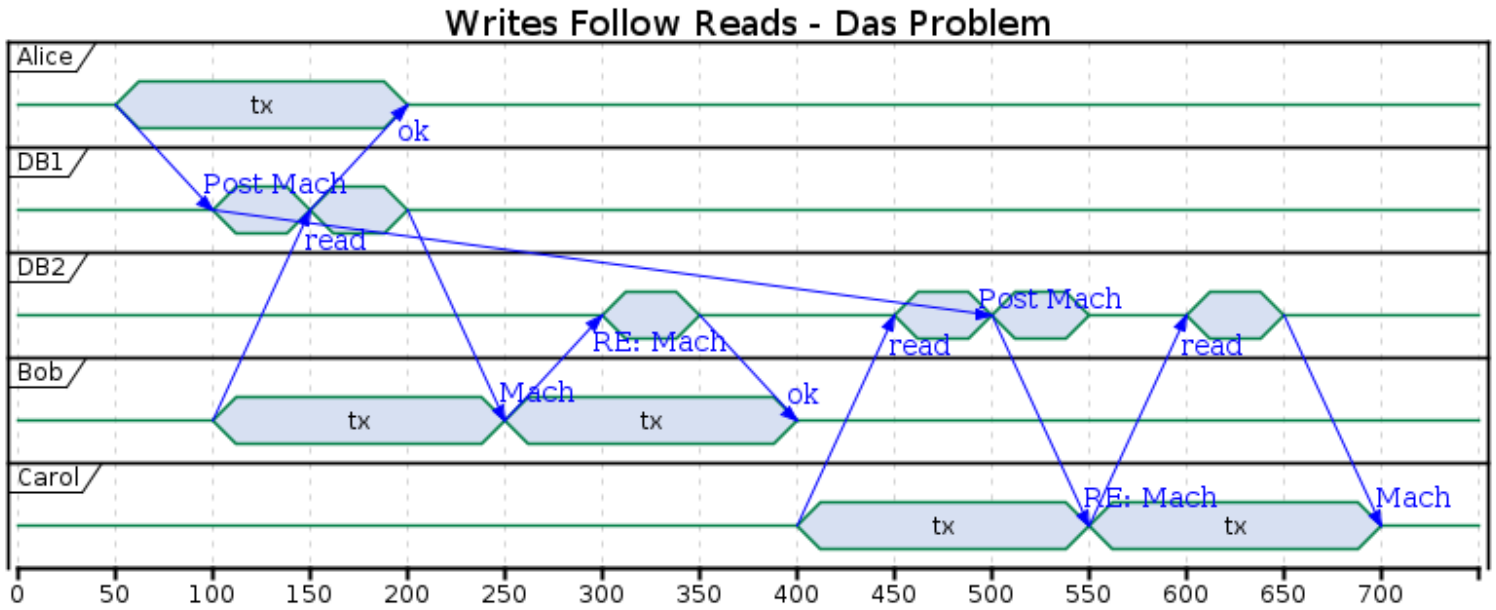
Welche Regeln gelten transaktionsübergreifend innerhalb einer Session?

Monotonic Reads/Writes



Monotonic Reads Sobald ein Wert von einem Client gelesen wurde, wird dieser oder ein späterer Wert bei folgenden reads gelesen.

Writes Follow Reads



Writes Follow Reads 2

Das Problem: Carol sieht die Antwort (Re: Mach) vor dem eigentlichen Post (Mach).

Anforderung:

- Eine Session sieht einen Effekt einer Transaktion T_1 und führt anschließend T_2 aus.
- Dann darf eine andere Session den Effekt von T_2 nur sehen, wenn sie auch T_1 sieht.

$\rightarrow$ happens-before

Monotonic Reads/Writes, Writes Follow Reads - Availability

- Alle 3 Garantien für Sessions können mit High Availability implementiert werden.
- Das System muss sicherstellen, dass writes nur sichtbar werden wenn alle Abhängigkeiten eines writes auf allen Replikas vorhanden sind.
- Beispiel Bulletin Board aus Vorlesung Koordination.

Read Your Own Writes

Wenn ein Client ein Datum liest nachdem er es geupdatet hat, muß read den geupdateten Wert oder einen späteren Wert liefern.

2 Transaktionen:

- T_1:$w_x(1)$
- T_2:r_x

Wenn T_1 auf einer Seite einer Netzwerkpartition ausgeführt wird und T_2 auf der anderen Seite, kann der aktuelle Wert nicht gelesen werden.

Sticky Availability behebt dieses Problem.

Consistency und Availability - Bewertung

- Wir haben ein Spektrum an möglicher Consistency und Availability.
- Verschiedene Teile eines Systems können verschiedene Anforderungen haben.
- Modelle helfen uns informierte Entscheidungen zu treffen.
- Unsere Anwendung muss nicht in jedem Fall 100% konsistent sein.
  - Manchmal reicht eine Entschuldigung.
  - Aber dies kann von Angreifern ausgenutzt werden (s. ACIDRain Paper).
- Können wir uns auf die Angaben von Datenbankherstellern verlassen?

Zusammenfassung

- Replikation kann zu Inkonsistenzen führen
- Konsistenzmodelle definieren Garantien
- Konsistenzmodelle häufig unterschiedlich definiert
- Entscheiden welche Garantien unser System benötigt
- Verteilte Systeme sind kompliziert und haben Bugs

→ Können wir die Komplexität von Koordination vermeiden?

CALM Theorem

> *Consistency as Logical Monotonicity (CALM). A program has a consistent, coordination-free distributed implementation if and only if it is monotonic.*

Paper: Keeping CALM: when distributed consistency is easy; Hellerstein, Alvaro; 2019

CALM: Essential vs. Accidental Coordination

- Ähnlich zu Komplexität (No Silver Bullet; Brooks; 1987) kann Koordination in essentielle und versehentliche Koordination unterteilt werden.
- Essentielle Koordination:
  - Ist nötig um bestimmte Garantien geben zu können.
- Versehentliche Koordination:
  - Kann mit einem anderen Design vermieden werden.
- –> Welche Probleme können konsistent, ohne Koordination, verteilt gelöst werden und welche nicht?

CALM: Monotonicity

Formale Definition:

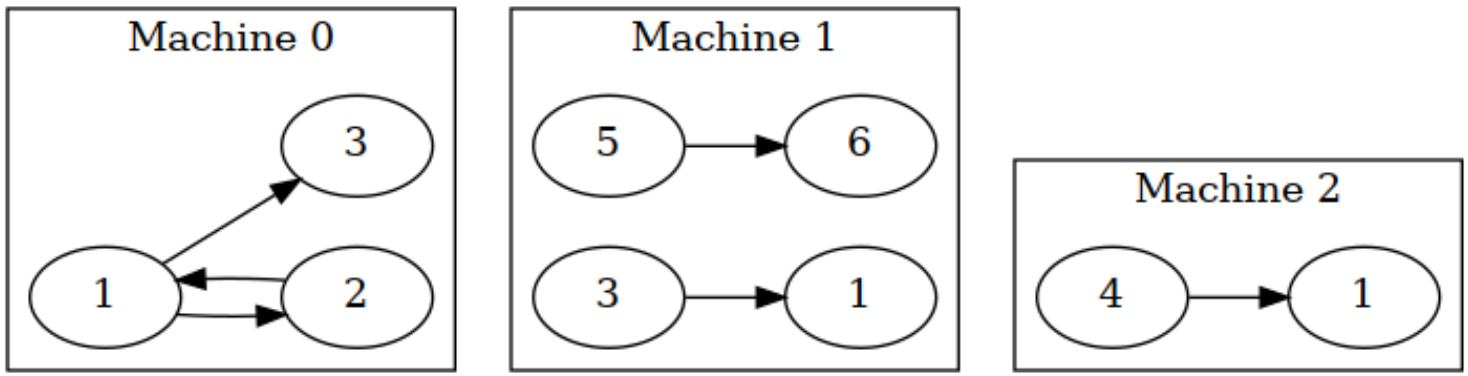
Ein Programm P ist monoton, wenn für alle input sets S, T gilt: wenn $S \subseteq T$ dann $P(S) \subseteq P(T)$

Beispiel: Set addition

- S = add(a), add(b)
- T = add(a), add(b), add(c)
- $S \subseteq T$ = true
- P(S) = {a, b}
- P(T) = {a, b, c}
- $P(S) \subseteq P(T)$ = true

Monotonicity: Distributed Deadlock Detection

Waits-for-graph: Kante i –> j bedeutet, dass Transaktion i auf ein Lock wartet, das von Transaktion j gehalten wird.

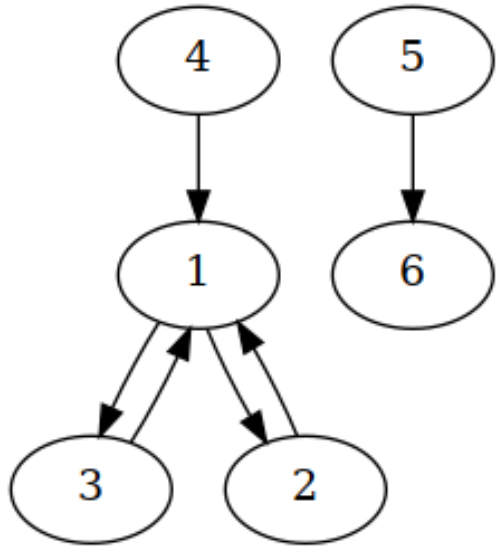


Ein Zyklus im Graph entspricht einem Deadlock.

Welche Deadlocks enthält das System?

Distributed Deadlock Detection - Berechnung

Jede Maschine übermittelt ihre Kanten.

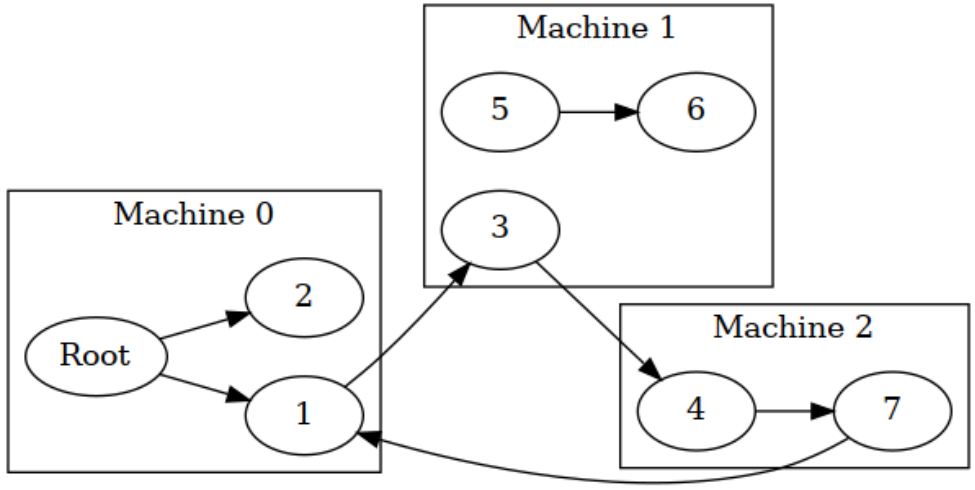


Können zusätzliche Kanten dazu führen Deadlocks auflösen?
–> Nein. Zusätzliche Kanten führen nur zu evtl. zusätzlichen Deadlocks. –> monoton!

CALM: Distributed Garbage Collection

Objektgraph: Kante entspricht Referenz von einem Objekt zu einem anderen.

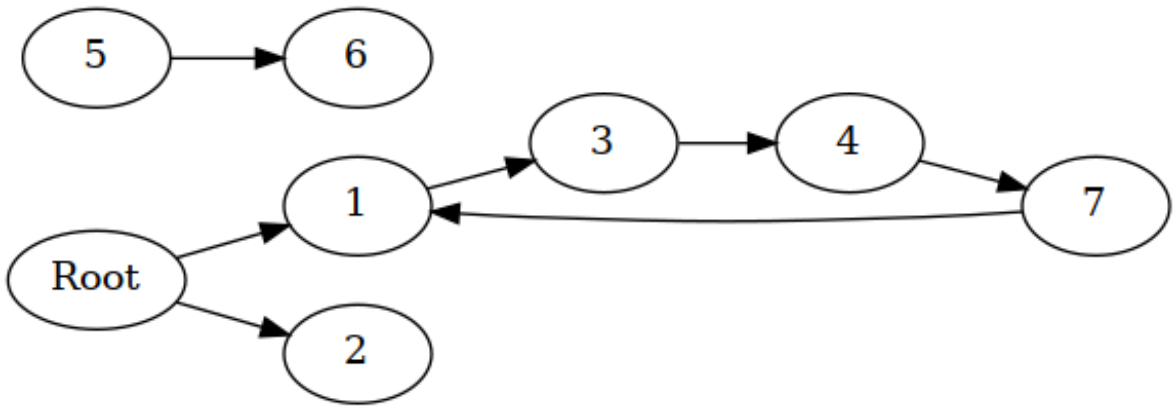
Garbage Collection: finde Objekte, die nicht von `Root` aus erreichbar sind.



Welche Objekte können entfernt werden?

Distributed Garbage Collection - Berechnung

Jede Maschine übermittelt ihre Kanten.



Können weitere Kanten dazu führen unser Ergebnis ändern?

–> Ja. Eine Kante von 1 zu 5 würde dazu führen, dass 5 und 6 nicht collected werden können –> nicht monoton! –> Wir benötigen Koordination

Deadlocks vs. Garbage Collection

- Beide Probleme werden ähnlich gelöst.
- Deadlocks benötigt im Gegensatz zu Garbage Collection keine Koordination.

Was ist der Unterschied?

- Bei Deadlocks fragen wir, ob ein Zyklus existiert.
- Bei Garbage Collection fragen wir, ob **kein** Pfad existiert.

Die 2te Frage kann nur beantwortet werden, wenn wir alle Kanten gesehen haben.

–> Solange wir $\forall$ und $\nexists$ verbieten bleiben wir monoton.

Composability

Wenn die Funktionen f und g monoton sind, ist auch $f(g(x))$ monoton.

–> Wir können monotone Programme aus monotonen Operationen konstruieren.

Beispiel: Shopping Cart

Können wir den Shopping Cart einer Webanwendung monoton gestalten?

Idee: Wir modellieren den Inhalt des Shopping Cart als Set.

Das Hinzufügen eines Artikels ist damit monoton und wir benötigen keine Koordination.

Aber wie können wir einen Artikel aus dem Shopping Cart entfernen?

Idee: Wir verwalten ein Add-Set und ein Remove-Set.

Shopping Cart: Bewertung

Die Verwaltung des Shopping Carts kann ohne Koordination erfolgen.

Allerdings benötigen wir Koordination, sobald wir den Einkauf tätigen.

–> Wir müssen sicherstellen, dass alle Änderungen des Shopping Carts gesehen wurden.

–> Aber wir konnten die nötige Koordination im System reduzieren.

CRDTs

Paper: A Comprehensive study of Convergent and Commutative Replicated Data Types; Shapiro et al. (2011)

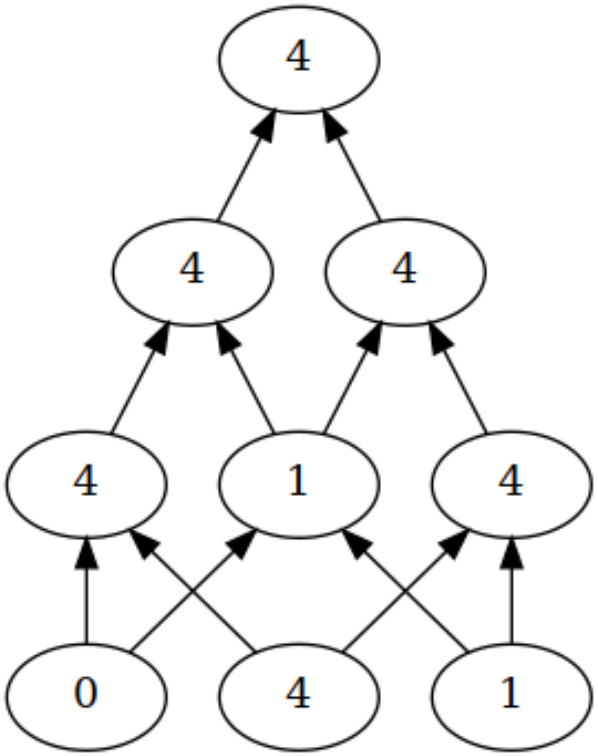
Wir unterscheiden:

- Operation Based Replication: CmRDT
- State Based Replication: CvRDT

State Based Replication

- Wir updaten den State eines Objekts lokal in einer Replika.
- Der geupdatete State wird an alle anderen Replikas übermittelt und dort mit dem jeweiligen lokalen State gemerged.
- Wenn State + Merge assoziativ, kommmutativ und idempotent sind, wird keine Koordination benötigt.

Beispiel: Integer, max



- assoziativ: max(max(1,2),3) = max(1,max(2,3))
- kommutativ: max(1,2) = max(2,1)
- idempotent: max(1,2) = max(max(1,2),2)

Operation Based Replication

- Das System repliziert Operationen anstatt State.
- Benötigt einen zuverlässigen broadcast channel, der die vom CRDT bestimmte Ordnung berücksichtigt (Beispiel: kausal).
- Operationen die nach der Ordnung gleichzeitig (concurrent) stattfinden, müssen kommutativ sein.

Beispiel:

- +7, -5 = -5, +7

Operation Based vs. State Based

- State Based:
  - Simpler, da keine Ordnung der Nachrichten benötigt wird und jede Änderung lokal betrachtet werden kann.
  - Nachrichten müssen irgendwann ankommen, aber Reihenfolge ist egal.
  - Überträgt immer gesamten State.
  - Keine Gruppenzugehörigkeit nötig.
- Operation Based:
  - Komplexer, sämtliche Nachrichten betrachtet.
  - Überträgt nur die Operationen. („diff“)
  - Benötigt Gruppenzugehörigkeit.

Counter 1

Operation Based ist trivial.

Versuch State Based:

```
state = 0
def increment():
    state = state + 1

def merge(other_state):
    state = max(state, other_state)

def value():
    return state
```

Entspricht die Implementierung einem CRDT?

Counter 1?

Counter 1 ist kein CRDT. Wenn wir 2 Counter haben und jeden einmal inkrementieren, wird der Wert auf 1 statt 2 konvergieren.

Neuer Versuch.

Counter 2

```
state = [0 ,0] # Annahme 2 Counter im System
def increment():
    my_id = get_id()
    state[my_id] = state[my_id] + 1
def merge(other_state):
    for i in len(state):
        state[i] = max(state[i], other_state[i])
def value():
    return sum(state)
```

CRDT: G-Counter = Grow-only Counter
Allerdings State Based CRDT + Gruppenzugehörigkeit
→ Wie *Vector Clocks*.

G-Set (Grow only)

```
state = Set()
def add(element):
    state.add(element)

def merge(other_state):
    state.union(other_state)

def contains(element):
    return element in state
```

2P Set (2 Phase)

```
state = {added: Set(), removed: Set()}
def add(element):
    state.added.add(element)
def remove(element):
    state.removed.add(element)
def merge(other_state):
    state = {added: state.added.union(other_state.added),
             removed: state.removed.union(other_state.remov
def contains(element):
    return element in state.added and not element in state.
```

Einschränkung 2P-Set?

- einmal hinzugefügtes Element kann nie wieder hinzugefügt werden.

Counter - Übung

Implementiere einen Counter, der `decrement` bietet.

Weitere CRDTs

- PN-Set: Counter für jedes Element, Wert des Coutner entscheidet über Set-Zugehörigkeit.
- 2P2P Graph: Je ein 2P Set für Knoten und Kanten.
- Verschiedene Implementierungen von Registern.
- Datentypen für kollaborative Textbearbeitung.

→ Conflict-free_replicated_data_type#Known_CRDTs

Zusammenfassung

- CRDTs können genutzt werden, um Koordination zu vermeiden oder zumindest einzuschränken.
- CRDTs benötigen eine Form der Garbage Collection, um performant zu bleiben.
  - Garbage Collection benötigt wiederum Koordination.
- CRDTs werden in Verteilten Systemen eingesetzt: Riak, Redis, Dynamo

Quellen

- Highly Available Transactions: Virtues and Limitations; Bailis, Davidson, Fekete, Ghodsi, Hellerstein, Stoica; 2013
- Weak Consistency: a generalized theory and optimistic implementations for distributed transactions; Adya; 1999
- Designing Data-Intensive Application: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems; Martin Kleppmann; 2017
- Keeping CALM: when distributed consistency is easy; Hellerstein, Alvaro; 2019
- A Comprehensive study of Convergent and Commutative Replicated Data Types; Shapiro et al.; 2011
- Architecture of Open Source Applications; Brown, Wilson; 2014

Ich wünsche Ihnen unkoordinierten Erfolg!

Verweise I

Adya, A. and Liskov, B. (1999). Weak consistency: A generalized theory and optimistic implementations for distributed transactions.

Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2013). Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011). A comprehensive study of convergent and commutative replicated data types.

Bilder: