Willkommen bei Verteilte Systeme!

Von Datenbanken
über Webdienste
bis zu p2p und Sensornetzen.

Heute: **Replikation, CALM und CRDTs**.
Versprich nur, was du halten kannst.

Ziele

- Ihr kennt verschiedene Arten der Replikation.
- Ihr versteht, dass Replikation zu Inkonsistenzen führen kann.
- Ihr kennt das CALM Theorem.
- Ihr versteht, dass Koordination vermieden werden kann und dies zu einfacheren Systemen führt.

Ablauf heute

- Replikation
- Was ist Availability?
- Welche Konsistenzmodelle gibt es?
- 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.[1]

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

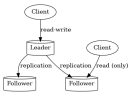
[1] Das machen wir bei Disy: Synchronisierte Caches

Übersicht Replikation

3 Arten von Replikation werden unterschieden:

- Single Leader
- Multi Leader
- Leaderless

Single Leader



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

Hierarchie ähnlich zu NTP.

Multi Leader Replication

Nachteile Single Leader

- Leader nicht erreichbar ⇒ keine Änderungen
- Einzelner Leader → Flaschenhals

Anwendungen

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

Nachteil Multi Leader

- Lösung von Schreibkonflikten nötig

Leaderless Replication

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

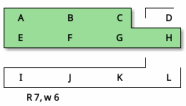
Quorum

- 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: $r + w > n$

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

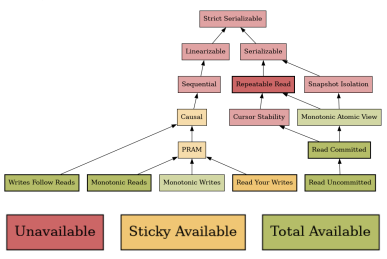
Sticky Available - Beispiel

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

Unavailable

System ist nicht verfügbar bei Netzwerkpartitionen.

Consistency



Consistency und Availability - Bewertung

- Spektrum möglicher Consistency und Availability.
- Verschiedene Teile eines Systems können verschiedene Anforderungen haben.
- Informierte Entscheidungen treffen!
- Unsere Anwendung muss nicht in jedem Fall 100% konsistent sein.
  - Manchmal reicht eine Entschuldigung.
  - Aber angreifbar! (s. ACIDRain Paper (Warszawski and Bailis, 2017))
- Können wir uns auf Angaben von Herstellern verlassen?

Zusammenfassung

- 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 and Alvaro, 2019)

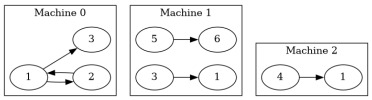
CALM: Essential vs. Accidental Coordination

- 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?

Essenziell vs. Versehentlich

Monotonicity: Distributed Deadlock Detection

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

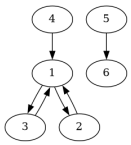


Ein Zyklus im Graph entspricht einem Deadlock.

Welche Deadlocks enthält das System?

Distributed Deadlock Detection - Berechnung

Jede Maschine übermittelt ihre Kanten.

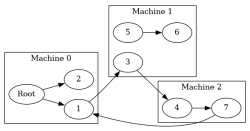


Können zusätzliche Kanten Deadlocks auflösen / entfernen?
–> Nein. Zusätzliche Kanten führen nur zu evtl. zusätzlichen Deadlocks. –> monoton!

CALM: Distributed Garbage Collection

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

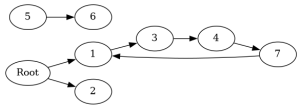
Garbage Collection: finde Objekte, die nicht von `Root` aus erreichbar sind.



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.

–> Monotone Programme aus monotonen Operationen aufbauen.

Zusammenfassung

- Essenzielle vs. versehentliche Koordination
- Motonon: Vermeidet *„für alle“* und *„es gibt kein“*

Koordination minimieren.

CRDTs

Datenstrukturen, die weniger Garantien brauchen.

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

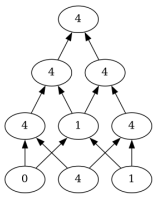
Wir unterscheiden:

- 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 (alle Beteiligten kennen).

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.remo
def contains(element):
    return element in state.added and not element in state
```


Einschränkung 2P-Set?

- einmal hinzugefügtes Element kann nie wieder hinzugefügt werden.

Praxis: 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

Verwaltung des Shopping Carts ohne Koordination.

Brauchen Koordination, sobald wir den Einkauf tätigen.

–> müssen sicherstellen, dass alle Änderungen vorher gesehen wurden.

–> Koordination im System reduzieren, nicht komplett verhindern.

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 (z.B. Schnappschuss — „Keyframe“).
  - Garbage Collection benötigt wiederum Koordination.
- CRDTs werden in Verteilten Systemen eingesetzt: Riak, Redis, Dynamo
- Ein nur auf CRDTs aufbauendere Algorithmus ist Koordinationsfrei

Quellen

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

Ich wünsche Ihnen unkoordinierten Erfolg!

Verweise I

Adya, A. and Liskov, B. (1999). Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis: http://pmg.csail.mit.edu/papers/adya-phd.pdf.

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.

Brown, A. F. and Wilson, G. (2011). The architecture of open source applications. PDF from http://vgc.poly.edu/~juliana/pub/vistrails-aosa.pdf.

Hellerstein, J. M. and Alvaro, P. (2019). Keeping CALM: when distributed consistency is easy. *CoRR*, abs/1901.01930.

Verweise II

Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011). A comprehensive study of convergent and commutative replicated data types. PDF from https://hal.inria.fr/inria-00555588/document or https://dsf.berkeley.edu/cs286/papers/crdt-tr2011.pdf.

Verweise III

Warszawski, T. and Bailis, P. (2017). Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 5–20, New York, NY, USA. Association for Computing Machinery.