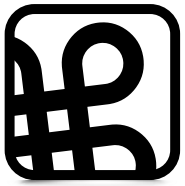


Macoun

Effizientes Offline-First-Sync: Ein Überblick

Daniel Alm (@daniel_a_a)

Sync:


New Task
Past 30 Days
No Filter
Search
Overview
Review
Details
Reports
Total time
150h
30h 2m per week
Most active weekdays
S M T W T F S
Most active hours
12 AM 6 AM 12 PM 6 PM
Time per Project
per
Week
33h 5m
28h 5m
35h 5m
37h 25m
16h 31m
08/05 – 08/11
08/12 – 08/18
08/19 – 08/25
08/26 – 09/01
09/02 – 09/04
Productivity score
69 %
Keep it up
Most productive weekdays
S M T W T F S
Most productive hours
12 AM 6 AM 12 PM 6 PM
Applications
Xcode 32h 45m
Safari 31h 6m
Sketch 20h 13m
Slack 15h 45m
Mail 14h 0m
Pages 12h 40m
QuickTime Player 10h 0m
Messages 5h 15m
Keynote 3h 7m
Projects & Tasks
Client: Acme Inc. 43h 20m
(App usage without a task) 34h 20m
Research for Secret Project 3h 0m
Meeting with Angela 2h 0m
Meeting with Tim 2h 0m
Meeting with Jony & Craig 1h 0m
Design iPhone Y 1h 0m
Client: MegaCorp 35h 30m
Communication 26h 0m
Media 22h 57m

Timing Sync:
Eckdaten nach zwei Monaten

- **4000** Nutzer
- **5-20** Sync-Requests/Sekunde
- **36 GB** Datenbank
- **25,000,000** Tabellenzeilen in der Datenbank
- **400,000,000** synchronisierte Objekte
- **Eine** Beschwerde über Datenverlust

Ablauf

1. Vorüberlegungen
2. 2 ½ mögliche Ansätze
3. Praktische Umsetzung

Vorweg:

Guter Sync ist viel Arbeit

Anforderungen

- Resilienz
- Konsistenz
- Effizienz
- Wartbarkeit
- Offline-First

Vorüberlegungen

- Welche Datenstrukturen habe ich?
- Welche davon muss ich überhaupt synchronisieren?
- Welche Operationen auf diesen Strukturen muss ich abbilden?
- Welche Konflikte sind möglich? Wie löse ich die? Und wo/wann?
- Gibt es einen zentralen Server? Welche Aufgabe kommt diesem zu?

Beispielmodell: Projekt

```
class Project {
  let id: Int64
  let title: String
  let children: [Project]
}

let topLevelProjects: [Project]  // Zum Speichern der Reihenfolge
```

Beispielmodell: Task

```
struct Task {
  let id: Int64
  let start: Date
  let end: Date
  let title: String
  let project: Project?
}
```

Mögliche Operationen: Project

- Erstellen
- Titel ändern
- Verschieben (anderes Elternprojekt, andere Listenposition)
- Löschen (was ist mit Unterprojekten (!?))

Mögliche Operationen: Task

- Erstellen
- Zeitraum ändern
- Titel ändern
- Projekt ändern (Achtung: Cross-Entity-Referenz!)
- Löschen

Die Ansätze

#1: Timestamps

- Repliziere Client-Datenmodell auf dem Server, mit Timestamps für jedes Objekt/jede Eigenschaft

Timestamps: Project

```
struct Project {
  let id: Int64
  let title: String
  let parentID: Int64?
  let position: UInt32

  let createdAt: Date
  let lastTitleChange: Date?
  let lastParentOrPositionChange: Date?
  let deletedAt: Date?
}
```

Timestamps: Task

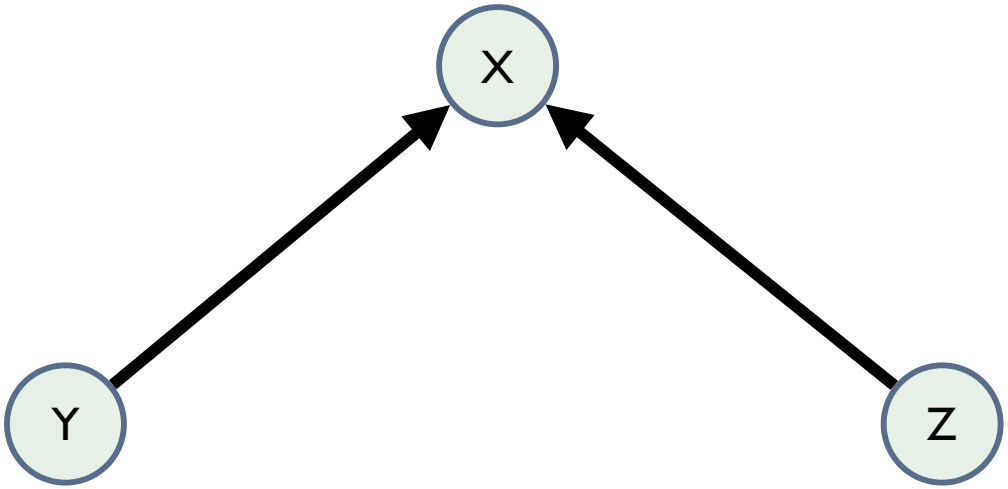
```
struct Task {
  let id: Int64
  let start: Date
  let end: Date
  let title: String
  let projectID: Int64?

  let createdAt: Date
  let lastDateRangeChange: Date?
  let lastTitleChange: Date?
  let lastProjectChange: Date?
  let deletedAt: Date?
}
```

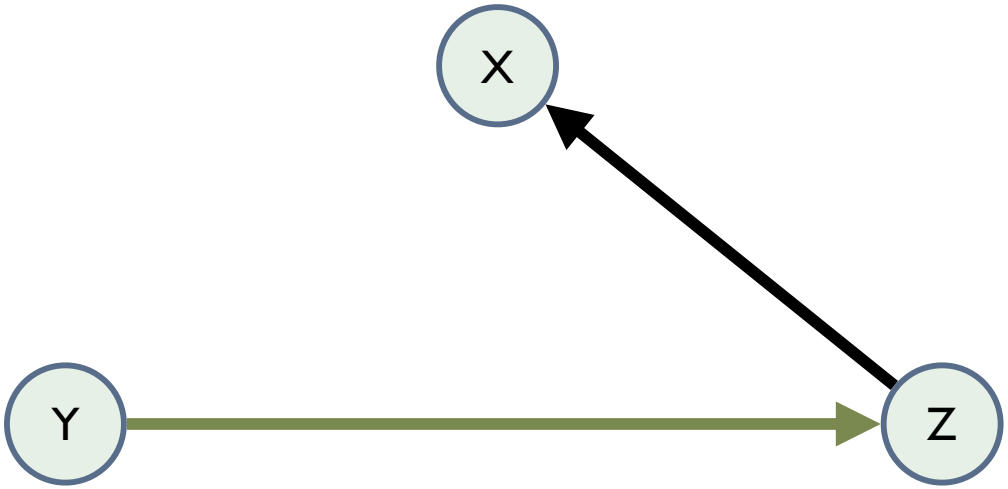
Timestamps: Probleme

- “Tombstones” notwendig, um Lösch-Operationen abzubilden
- Skaliert schlecht
- Viel Komplexität im Server
- Integration (“Merge”) bestehender Daten schwierig
- Konflikte

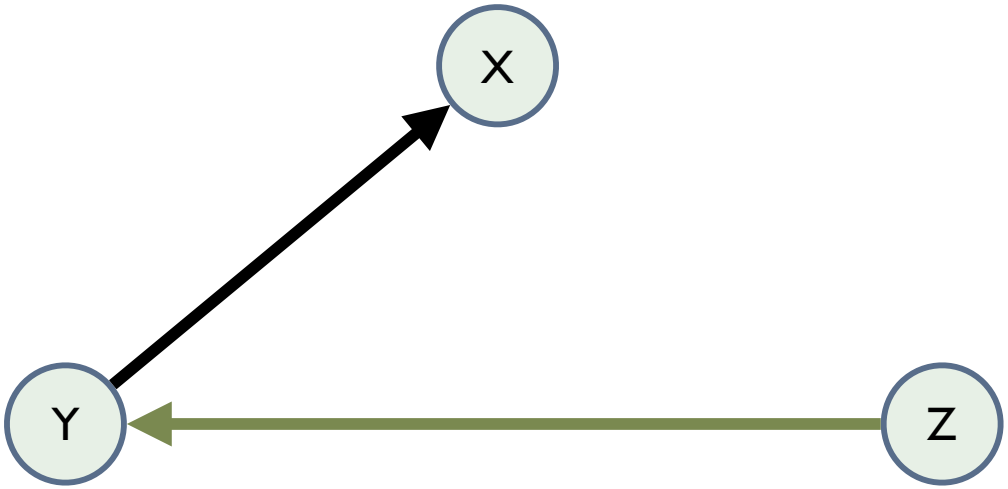
Beispiel: Konflikt



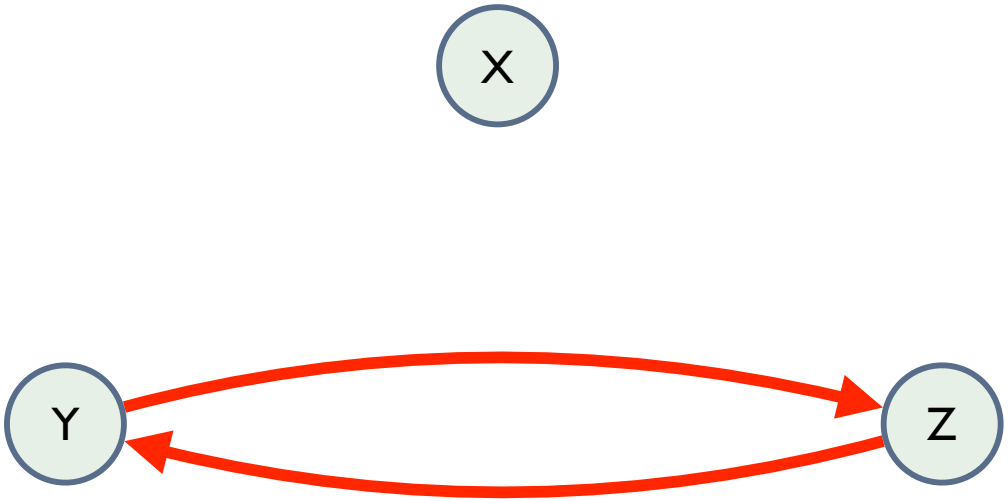
Beispiel: Konflikt (Gerät A)



Beispiel: Konflikt (Gerät B)



Beispiel: Konflikt (nach Sync)



#2: Operational Transformation

- Erfasse semantische Beschreibungen aller Änderungen
- Transformiere aufeinander folgende "gleichzeitige" Änderungen so, dass sie äquivalent sind: $A + T_A(B) = B + T_B(A)$
- Vorteil: potentiell sehr gute "Intentionstreue"
- Nachteil: ?

#3: Operation Log

- Semantische Beschreibung jeder Änderung
- In der *gleichen* Reihenfolge abgespielt, erhält man überall den selben Zustand
- Reihenfolge der Änderungen: z.B. Timestamp

Operation Log: Project

```
enum ProjectOperation {
  case create(id: Int64, title: String, parentID: Int64?, pos: Int)
  case setTitle(id: Int64, title: String)
  case moveToParent(id: Int64, parentID: Int64?, pos: Int)
  case delete(id: Int64)
}
```

Operation Log: Task

```
enum TaskOperation {
  case create(id: Int64, startDate: Date, endDate: Date, title: String,
projectID: Int64?)
  case setDateRange(id: Int64, startDate: Date, endDate: Date)
  case setTitle(id: Int64, title: String)
  case setProject(id: Int64, projectID: Int64?)
  case delete(id: Int64)
}
```

Und jetzt?

- Wir haben das Problem “reduziert” auf:
  - Erfassung (inkl. Serialisierung)
  - Synchronisation
  - Wiedergabe (inkl. Deserialisierung) der Operationen

Erfassung

- Alle Änderungen werden von einer zentralen Stelle aus ausgeführt
  - Alternative/Ergänzung: Übergebe jeder mutierenden Methode ein Logger-Objekt, dem wir die zu erfassenden Änderungen mitteilen
- Serialisierung: Protocol Buffers

Synchronisation der Operationen

- **CRDT: “Conflict-Free Replicated Data Type”**
- Beispiel: “Grow-only Set”
  - Kennt nur eine Operation: Objekt hinzufügen
  - Einfachste Synchronisation: gebe alle bekannten Werte weiter
  - Erklärung: https://bit.ly/crdt_app

Synchronisation: Upload

- Client: “Hier sind alle Operationen, von denen ich noch nicht sicher weiß, dass sie hochgeladen (und angekommen!) sind”
- Server: “Alles klar, die hochgeladenen Operationen haben jetzt IDs 130, 140”
- Client speichert die IDs für die hochgeladenen Operationen

Synchronisation: Download

- Client: “Bitte gib mir alle Operationen ab Stand ID 123, abgesehen von den folgenden: 130, 140”
- Server: “Hier sind Operationen 127, 135, 142. Du bist jetzt auf Stand ID 150”
- Client: speichert neue Operationen und merkt sich “Stand ID 150”

Wiedergabe

- Gruppiere Operationen nach Objekt-Typ, sortiere nach Timestamp
- Spiele für jeden Objekt-Typ dessen Operationen nacheinander ab
- Mögliche Konflikte:
  - Einzelne Eigenschaften: “Last Edit Wins”
  - Hierarchie-Bedingungen: überspringe Operationen, die einen ungültigen Zustand erzeugen würden

Praktische Umsetzung

Optimierung: Cachen des aktuellen Zustands

- Wiedergabe des *gesamten* Logs nach jedem Sync wäre zu viel Arbeit
- Lösung: speichere aktuellen Stand im normalen Datenmodell, wende nur die neuen Änderungen auf diesen Stand an

Problem: Out-of-Order-Änderungen

- Beginne bei Stand **x**
- Wende Operation A an. Neuer Stand: (**x** + A)
- Synchronisation mit Server ergibt Operation B, die vor A stattfand
- Können Stand ((**x** + A) + B) berechnen, benötigen aber (**x** + B + A)
- Problem: wie kommen wir von (**x** + A) zurück zu (**x** + B + A)?

Lösung: “Idempotente” Operationen

- Einfach, wenn (**x** + B + A) = ((**x** + A) + B + A) *für **alle** möglichen Operations-Sequenzen A, B*
- Normalerweise der Fall bei “Last Edit Wins”
- Gegenbeispiel: Hierarchien, da dort die Operationen nicht auf einzelne Felder beschränkt sind

Integration bestehender Daten

- Wie synchronisieren wir Daten, die schon vor dem Sync vorhanden waren?
- Lösung: Erstelle ein Diff von aktuellem Sync-Datensatz und neuem Client, hänge alle Objekte im Diff als “Add”-Operationen ans Operation Log an
- Achtung: hier keine Fehler machen!

Optimierung: Kompression

- Viele ähnliche Einträge

  ⇒ Kompression kann erhebliche Vorteile bringen

- Insbesondere bei der Integration bestehender Daten
- Weniger effizient für "neue", im laufenden Betrieb erstellte Daten
  - Einträge können auch nachträglich zusammengefasst werden, ist aber kompliziert

Problem: Cross-Entity-Konflikte

- Device A erstellt Task in Projekt X
- Device B löscht Projekt X
- Lösung: Nach dem Anwenden (aber vor Abschluss des Syncs!) auf Konflikte prüfen

Problem: Lookup Tables

- Beispiel: Lookup Table zur Deduplizierung von Task-Titeln
- IDs von identischen Einträgen unterscheiden sich von Gerät zu Gerät
- Lösung: Lookup-Table nicht synchronisieren, Daten im Operation Log denormalisieren

Änderungen am Datenmodell

- Lösung: veraltete Clients syncen entweder gar nicht mehr, oder erkennen inkompatible Log-Einträge an deren Versionsnummer

Server-Stack

- Swift/Vapor
- SwiftGRPC hinter Google Cloud Endpoints zur Kommunikation
- Docker-Container via Kubernetes Engine auf Google Cloud
- Datenbank: PostgreSQL (Google Cloud SQL)
- Authentifizierung: zufällige Refresh Tokens + JWT

- Alternative: CloudKit

Timeline: Timing Sync

- Januar: Refactoring/Zentralisieren aller DB-Änderungen
- Februar: Erweitern dieser Klasse um Logging
- Februar/März: SwiftGRPC überarbeitet
- März: Hochladen/Anwenden von Sync, mit Fake-Server getestet
- April: Sync-Server in Vapor/SwiftGRPC
- Mai + Juni: Deployment, erste Beta, Feintuning
- Anfang Juli: Release

Vorteile

- Sehr robust
- Log besteht nur aus BLOBs

  ⇒ effizient

  ⇒ einfacher Server-Code

  ⇒ wartbar

  ⇒ Kompression + Verschlüsselung einfach umzusetzen
- Relativ gut komplett zu testen

Nachteile

- Gelöschte Daten und alte Änderungen belegen immer noch Platz
- Benötigt lokalen Cache, da kein „Random Access“ möglich
  - Nicht tragbar, wenn bei jedem Sync alles neu abgespielt wird
- Team-Funktionen schwer umsetzbar
- Cross-Entity-Konflikte müssen individuell gelöst werden
- Benötigt einen zentralen Server

Zusammenfassung

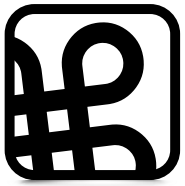
- Jede Sync-Problemstellung ist anders
  - Vorher angestellte Überlegungen sparen euch später viel Zeit
- Vorsicht bei Feld-basierter Synchronisation
- Versucht möglichst viel Nutzerintention zu erhalten

Fragen?

via Twitter: @daniel_a_a

(Vorträge bewerten bitte nicht vergessen!)

Vielen Dank



Macoun

Konflikt: überlappende Tätigkeiten und App-Aktivitäten

- Immer nur eine Tätigkeit zu jeder Zeit (dürfen nicht überlappen)
  - App-Aktivitäten zählen nicht, wenn Tätigkeit zur gleichen Zeit
- Bisher:
  - Bestehende Tätigkeit gelöscht und ersetzt
  - App-Aktivitäten haben Eigenschaft *taskID*

Konflikt: überlappende Tätigkeiten und App-Aktivitäten

- Für Sync:
  - Auf unterschiedlichen Geräten gleichzeitig erstelle Tätigkeiten dürfen überlappen (passiert extrem selten)
  - *taskID* wird zur Laufzeit “berechnet”

Beispiel-Server: Upload

```
func uploadOperationLog(session: SyncUploadOperationLogSession) throws
-> Status? {
  while true {
    guard let message = try session.receive() else { return .ok }
    var newEntry = message.operationLogEntry
    newEntry.metadata.serverTimestamp = Date().timeIntervalSince1970
    newEntry.metadata.serverID = operationLogEntries.count
    operationLogEntries.append(newEntry)

    var response = Timing_UploadOperationLogResponse()
    response.receivedLocalID = newEntry.metadata.localID
    response.resultingServerID = newEntry.metadata.serverID
    response.resultingServerTimestamp =
```


QPS
by method (sum)
1 hr interval (rate)
20/s
15/s
10/s
5/s
0
Aug 19
Aug 26
Sep 02
Sep 09
Errors
1
by response code (sum)
1 hr interval (rate)
2.5/s
2.0/s
1.5/s
1.0/s
0.5/s
0
Aug 19
Aug 26
Sep 02
Sep 09
Encrypt/Decrypt Calls
1 hr interval (rate)
12/s
10/s
8/s
6/s
4/s
2/s
0
Aug 19
Aug 26
Sep 02
Sep 09
Cloud SQL Database - Bytes used
1 hr interval (mean)
40GB
36GB
32GB
28GB
24GB
20GB
Aug 19
Aug 26
Sep 02
Sep 09