Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

Sperr-Verfahren

Algorithmus

- exklusiver Schreibzugriff auf Objekte durch Vergabe von Sperren
- unterschiedliche Granularität von Sperren
  - → Objekthierarchie
  - → Trade-Off (Verwaltungs- und Kommunikations-)Overhead vs. Benutzbarkeit
- implizites vs. explizites Anfordern von Sperren
- implizites vs. explizites Freigeben von Sperren
- Fehlerbehandlung notwendig, falls die eine Sperre besitzende Instanz abstürzt
- pessimistisches Hard State-Verfahren für diskrete Anwendungen

Bewertung des Sperr-Verfahrens

+ Kausalität, Konvergenz und Korrektheit
- eingeschränkte Zusammenarbeit
- Verwaltungs- und Kommunikations-Overhead
- Wartezeit durch das Anfordern von Sperren
- Deadlocks sind möglich und müssen aufgelöst werden

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

Abstimm-Verfahren (1)

Algorithmus

- jedes Objekt erhält eine Sequenznummer SN ($\mathbb{N}$, Zeitstempel etc.)
- Vergabe von Schreib- und Lese-Rechten auf Objekten
- Erwerb eines Rechts durch Abstimmen
  - Abstimmungs-Anfrage an alle Instanzen
  - Votum = Anzahl der Zustimmungen
  - Quorum = mindestens erforderliche Zustimmungen für erfolgreiches Votum
- Schreiben/Lesen nur bei erfolgreicher Abstimmung

Abstimm-Verfahren (2)

Lesezugriff

- Abstimmung: erfrage SN_i jeder erreichbaren Instanz i
- $SN_{max} = max \{SN_i\}$
- Zustimmung von i, wenn $SN_i = SN_{max}$
- Votum = Anzahl der Instanzen i mit $SN_i = SN_{max}$
- Quorum = erforderliche Anzahl Instanzen mit höchster SN

Beispiel

- n = 4: $SN_1 = 5$, $SN_2 = 6$, $SN_3 = 6$, $SN_4 = 5$
- Quorum := n/2
- Instanz 4 möchte den Zustand lesen: Anfrage an 1,2 und 3

→ $SN_{max} = 6$, Instanzen 2 und 3 besitzen aktuellen Zustand

→ Mehrheit erreicht, d.h. positives Votum

→ Lesezugriff von 4 liefert Objektzustand mit SN = 6

Abstimm-Verfahren (3)

Schreibzugriff von Instanz j

- Abstimmung: erfrage SN_i jeder erreichbaren Instanz i
- Zustimmung von i, wenn $SN_j + 1 > SN_i$
- Votum = Anzahl der Zustimmungen
- Quorum = erforderliche Anzahl der Zustimmungen
- bei erfolgreicher Abstimmung:
  - j verändert den Zustand
  - inkrementiere SN_j
  - aktuellen Zustand an alle i senden

Beispiel: n = 4, $SN_1 = 5$, $SN_2 = 6$, $SN_3 = 6$, $SN_4 = 6$

→ Schreibzugriff von 4 mit $SN_4 + 1 = 7$

→ positive Abstimmung, auch wenn eine Instanz nicht antwortet

→ erfolgreicher Schreibzugriff führt zu $SN_i = 7 \; \forall$ i

Beispiel: n = 4, $SN_1 = 5$, $SN_2 = 6$, $SN_3 = 6$, $SN_4 = 5$

→ Schreibzugriff von 4 mit $SN_4 + 1 = 6$

→ maximal eine Zustimmung von 1

→ Zugriff abgelehnt

Abstimm-Verfahren (4)

- unterschiedliches Quorum für Lesen / Schreiben denkbar
- Verfahren zum Festlegen des Quorums, z.B.
  - einfache Mehrheit: n/2 + 1 für n gerade, (n + 1)/2 für n ungerade
  - gewichtete Mehrheit: jeder Instanz hat bestimmtes Gewicht, Quorum = $\sum$ Gewicht positive Antworten / Gesamtgewicht
  - Write All Read Any (WARA): $Quorum_{write} = n$, $Quorum_{read} = 1$
- Konsistenz = Mehrheit der Instanzen besitzt aktuellen Zustand
- pessimistisches Verfahren für diskrete Anwendungen
- ursprüngliche Verwendung bei verteilten Datei- und DB-Systemen

Bewertung des Abstimm-Verfahrens

+ Konvergenz, Kausalität und Korrektheit
+ gut geeignet für asynchrone Anwendungen
+ robust bzgl. Ausfall von Instanzen und Netzwerkfehlern
- temporäre Inkonsistenzen sind zugelassen
- Kodierung aller Zustandsänderungen als State
- Verzögerung durch Abstimmung

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

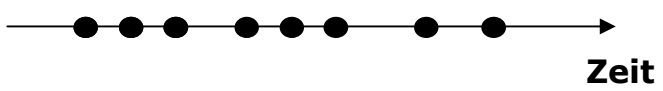
Serialisierung

Ziel: alle Instanzen führen alle Operationen in derselben Reihenfolge wie die virtuelle perfekte Instanz P aus

- optimistisches Verfahren für diskrete Anwendungen

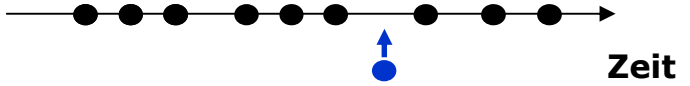
Voraussetzung

- jede Instanz i speichert eine lokale Operations-Historie H_i, die nach einer bestimmten Ordnung sortiert ist (z.B. Zustands-vektoren oder Ausführungszeit)
- H_i enthält alle lokalen und alle empfangenen Operationen

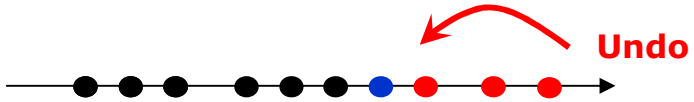


Serialisierungs-Algorithmus

i empfängt Operation O_j in falscher Reihenfolge

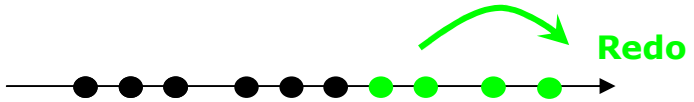


1. mache alle Operationen $O_k \in H_i$ mit $O_k > O_j$ rückgängig



2. führe O_j aus

3. führe alle $O_k \in H_i$ mit $O_k > O_j$ aus



Bewertung von Serialisierung

+ Konvergenz und Korrektheit
+ Kausalität kann vorgeschaltet werden: ausschließliches Einfügen kausal ausführbarer Operationen
+ autonome Ausführung (→ ausschließlich lokales Wissen)
+ sofortige Ausführung lokaler Operationen (→ keine Verlängerung der Response Time)
+ Verwendung der Operations-Historie für andere Funktionen
- erfordert Undo aller Operationen
- Speicherbedarf für Historie
- konfliktäre Operationen überschreiben sich gegenseitig und nur der Effekt der zuletzt ausgeführten bleibt erhalten
- visuelle Artefakte bei (abrupter oder gradueller) Zustands-änderung

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

Intentionserhaltung

Sei t^o der Erzeugungszeitpunkt von O_{i,t^o,t^*}.

Definition

Die **Intention** einer Operation O_{i,t^o,t^*} ist der Effekt, der durch die Ausführung von O_{i,t^o,t^*} auf dem von Instanz i zum Zeitpunk t^o angezeigten Zustand erzielt wird.

Eine Anwendung gewährleistet **Intentionserhaltung** ("Intention Preservation"), wenn die Intention aller O_{i,t^o,t^*} bei allen Instanzen gewahrt bleibt und nebenläufige Operationen nicht konkurrieren.

Beispiel

- sei S_0 = "ABCDE", O_i = "füge '12' bei Index 1 ein", O_j = "lösche von Index 2 bis Index 3" und $O_i \parallel O_j$
- dann ist die Intention von i: S_1 = "A12BCDE" / von j: S_1 = "ABE"
- kombinierter intentionserhaltender Zustand: S_1 = "A12BE"
- Ergebnis mit Serialisierung $O_i\ O_j \rightarrow S_1$ = "A1CDE" bzw. $O_j\ O_i \rightarrow S_1$ = "A12BE"

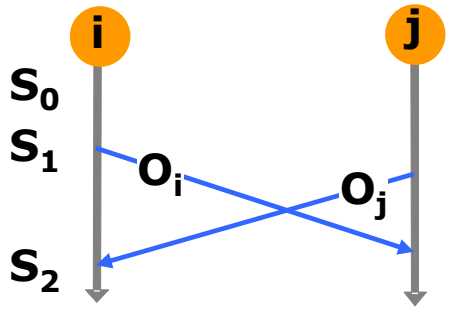
Operations-Transformation (1)

Idee: führe alle Operationen sofort aus (in beliebiger Reihenfolge), so dass die Intentionserhaltung gewährleistet wird

- lokale Operationen: unveränderte Ausführung
- empfangene Operation O_j: berücksichtige, dass sich der Zustand, der zum Zeitpunkt der Erstellung von O_j gültig war, in der Zwischenzeit durch nebenläufige O_i geändert hat

➡ transformiere O_j so, dass zwischenzeitliche Änderungen berücksichtigt werden

Beispiel: sei S_0 = "ABCDE", O_i = "füge '12' bei Index 1 ein", O_j = "lösche von Index 2 bis 3". Betrachte Instanz i:



- O_i → S_1 = "A12BCDE"
- O_j sollte "CD" löschen, deren Indizes haben sich aber durch O_i verschoben

→ transformiere O_j so, dass sie die Änderung durch O_i berücksichtigt: O_j' = "lösche von Index 4 bis 5" → S_2 = "A12BE"

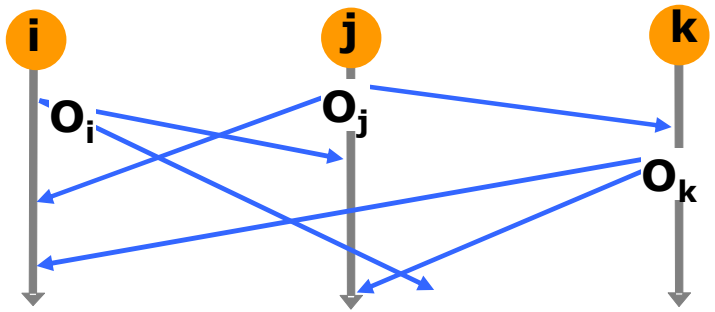
Operations-Transformation (2)

Allgemein

Sei O(S) der Zustand, der sich durch Anwendung von O auf S ergibt. Finde für $O_i \parallel O_j$ **Inklusions-Transformationen** (IT) $O \mapsto O'$, so dass $O_j'(O_i(S)) \equiv O_i'(O_j(S))$.

Betrachtung der Ausgangszustände

- bisher: O_i und O_j beziehen sich auf identische Zustände
- es sind aber auch unterschiedliche Ausgangszustände denkbar:

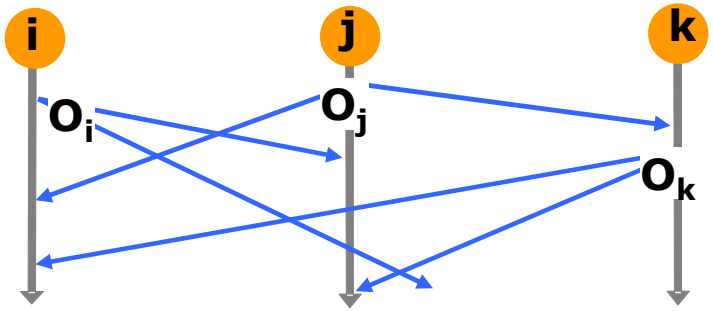


- $O_i \parallel O_k$ aber wegen $O_j \rightarrow O_k$ sind die Ausgangszustände von i und k ungleich

➡ IT berechnet falsche Operationen O_i' und O_k'

Operations-Transformation (3)

Beispiel



- S ="ABCDE", O_i ="füge '12' bei Index 1 ein", O_j ="füge '23' bei Index 0 ein", und O_k "füge '45' bei Index 2 ein"
- intentionserhaltender Zustand: "2345A12BCDE"
- k: $O_j(S)$ = "23ABCDE", $O_k(O_j(S))$ = "2345ABCDE"
- i: $O_i(S)$ = "A12BCDE", $O_j \parallel O_i$, $O_j' \circ O_i(S)$ = "23A12BCDE"
  - Empfang von $O_k \parallel O_i \rightarrow O_k'$ = "füge '45' bei Index 4 ein"
  - $O_k' \circ O_j' \circ O_i(S)$ ="23A1452BCDE"

➡ Index 2 in O_k und Index 1 in O_i beziehen sich auf verschiedene Ausgangszustände wegen $O_j \rightarrow O_k$

➡ finde **Exklusions-Transformation** (ET) $O_k \mapsto O_k'$, so dass bei anschließender Anwendung der Inklusions-Transformation $O_k' \mapsto O_k''$: $O_k'' \circ O_j' \circ O_i(S)$ = "2345A12BCDE"

➡ hier: ET O_k gegen O_j: O_k' = "füge '45' bei Index 0 ein" und IT O_k' gegen O_i: O_k'' = "füge '45' bei Index 2 ein"

Bemerkungen

- der gültige Anfangszustand einer Operation und die Beziehung zwischen Operationen wird i.d.R. durch Zustandsvektoren bestimmt
- die Transformation der Operationen gegeneinander erfordert eine lokale Operations-Historie
- zusammenfassendes Funktionsprinzip: Konsistenzkriterium Intentionserhaltung
- Intentionserhaltung ist tendenziell eher ein semantisches Kriterium, im Gegensatz zu den syntaktischen Kriterien Kausalität, Konvergenz, Konsistenz und Korrektheit
- verschiedene OT-Algorithmen: GOT, dOPT, adOPTed, …
- optimistisches Verfahren für Texteditoren mit relativen Operationen (→ diskret)

Bewertung von Operations-Transformation

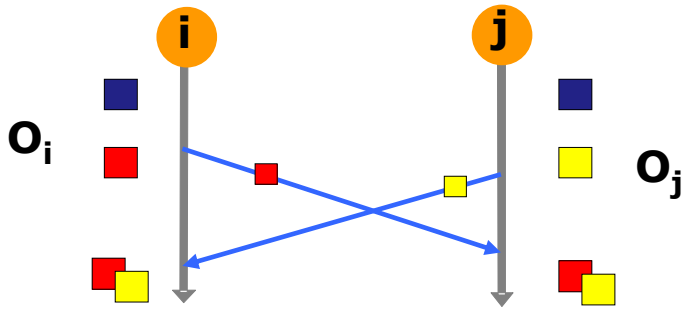
+ Konvergenz und Korrektheit
+ Intentionserhaltung
+ Kausalität durch vorgeschaltete Zustandsvektor-Analyse
+ autonome Ausführung (→ ausschließlich lokales Wissen)
+ sofortige Ausführung lokaler Operationen (→ keine Verlängerung der Response Time)
- komplexe Transformationsfunktionen IT und ET
- Speicherbedarf für Historie
- Intentionserhaltung funktioniert nicht für absolute Operationen

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

Objekt-Duplikation

Idee: nebenläufige konfliktäre Operationen führen zur Erzeugung von unterschiedlichen Versionen (Duplikaten) des betroffenen Objekts



- der Effekt jeder Operation bleibt erhalten (→ vgl. Serialisierung: nur der Effekt der letzten konfliktären Operation bleibt erhalten)
- die Auflösung von Konflikten bleibt den Benutzern überlassen

- Konflikte lassen sich z.B. durch Zustandsvektoren feststellen
- Versionen sind als zusammenhängend markiert
- Versions-Management durch die Benutzer (Speichern, Löschen …)
- optimistisches Verfahren für diskrete Anwendungen

Grundlegender Algorithmus

Ziel: Erzeuge eine minimale Anzahl von Objektversionen.

Idee: Gegeben sei eine Folge von Operationen. Dann bestimme Teilfolgen so, dass

1. alle Operationen in einer Teilfolge untereinander nicht konfliktär (= kompatibel) sind
2. jede Teilfolge die maximale Menge kompatibler Operationen enthält

Erzeuge für jede Teilfolge eine Objektversion durch Ausführen der enthaltenen Operationen.

Beispiel 1

- O_1, O_2, O_3 mit $O_1 \parallel O_2 \parallel O_3$ und $O_1 \otimes O_2$
- Teilfolgen $\{O_1,O_3\}$ und $\{O_2,O_3\}$

Kompatible Gruppen

Definition
Gegeben sei eine Gruppe (Folge) von Operationen GO. Dann nennt man eine Untergruppe von GO **kompatible Gruppe** ("Compatible Group") CG, wenn sie ausschließlich paarweise kompatible Operationen enthält: $\forall\ O_i,O_j \in CG\ \neg\ (O_i \otimes O_j)$

Beispiel 1: $\{O_1,O_3\}$ und $\{O_2,O_3\}$

Definition
Gegeben sei GO. Dann ist die **kompatible Gruppen-Menge** ("Compatible Group Set") CGS gegeben durch

$CGS = \{CG_1, CG_2, \ldots, CG_n\}$ mit

(1) $\forall\ O \in GO\ \exists\ CG_i \in CGS$ mit $O \in CG_i$

(2) $\forall\ O_i,O_j \in GO$: wenn $\neg\ (O_i \otimes O_j)\ \exists\ CG_i \in CGS$ mit $O_i, O_j \in CG_i$

Beispiel 1: $CGS = \{\{O_1,O_3\},\{O_2,O_3\}\}$

Maximale Kompatible Gruppen-Menge

Definition

CG_i ist eine **maximale kompatible Gruppe** ("Maximum Compatible Group") MCG, wenn $\forall\ O_i \in GO$ mit $O_i \notin CG_i\ \exists\ O_j \in CG_i$ mit $O_i \otimes O_j$.

Beispiel 2: $GO = \{O_1,O_2,O_3,O_4\}$ mit $O_1 \otimes O_2 \rightarrow \{O_1,O_3,O_4\}$ ist MCG

Definition

Ein CGS ist eine **maximale CGS** MCGS, wenn

(1) $\forall\ CG_i \in CGS$: CG_i ist MCG

(2) alle MCGs in GO sind in MCGS

Es kann gezeigt werden, dass für jede GO genau eine MCGS existiert.

Beispiel 2: $\{\{O_1,O_3,O_4\},\{O_2,O_3,O_4\}\}$ ist MCGS

MOVIC Algorithmus (1)

Erzeugung von Objekt-Versionen

Sei M die MCGS für GO. Dann erzeuge für jede $CG_i \in M$ eine Objektversion durch Ausführung aller $O_i \in CG_i$.

Gesucht: verteilter Algorithmus zur Erzeugung der MCGS

MOVIC – Multiple Object Versions Incremental Creation

- gegeben sei eine Folge $O_1, O_2, \ldots, O_n$
- MOVIC erzeugt eine Folge $MCGS_1, MCGS_2, \ldots, MCGS_n$
- $MCGS_i$ ist die MCGS für O_1 bis O_i
- $MCGS_i$ wird aus $MCGS_{i-1}$ und O_i erzeugt

MOVIC Algorithmus (2)

1. $MCGS_i = \{\}$, $C = |MCGS_{i-1}|$
2. **WHILE** $MCGS_{i-1} \neq \{\}$
   i. entferne CG_x aus $MCGS_{i-1}$
   ii. **IF** $\forall O_j \in CG_x \neg(O_i \otimes O_j)$ **THEN** CG_x += $\{O_i\}$
   iii. **ELSEIF** $\forall O_j \in CG_x$ $O_i \otimes O_j$ **THEN** C--
   iv. **ELSE**
      - $CG_n = \{O \mid (O \in CG_x) \wedge \neg(O \otimes O_i)\}$
      - $CG_y = CG_n + \{O_i\}$
      - $MCGS_i$ += $\{CG_y\}$
      - $MCSG_i$ += $\{CG_x\}$
3. **IF** C = 0 **THEN**
   i. $CG_n = \{O_i\}$
   ii. $MCGS_i$ += $\{CG_n\}$
4. $\forall CG_n$: **IF** $\exists CG_z \in MCGS_i$ mit $CG_n \subseteq CG_z$ **THEN** $MCGS_i$ -= CG_n

MOVIC Algorithmus (3)

- es kann gezeigt werden, dass
  - MOVIC die MCGS für GO konstruiert
  - die Operationen in beliebiger Reihenfolge ausführbar sind
- benötigt werden weitere Algorithmen zur
  - Vergabe von IDs für die verschiedenen Objektversionen
  - graphischen Darstellung überlappender Objekte

Beispiel

O_1,O_2,O_3,O_4 mit $O_1 \otimes O_2$, $O_1 \otimes O_3$ und $O_2 \otimes O_3$

1. Reihenfolge O_1,O_2,O_3,O_4
   - $MCGS_1 = \{\{O_1\}\}$
   - $MCGS_2 = \{\{O_1\},\{O_2\}\}$
   - $MCGS_3 = \{\{O_1\},\{O_2\},\{O_3\}\}$
   - $MCGS_4 = \{\{O_1,O_4\},\{O_2,O_4\},\{O_3,O_4\}\}$
2. Reihenfolge O_1,O_2,O_4,O_3
   - $MCGS_1 = \{\{O_1\}\}$
   - $MCGS_2 = \{\{O_1\},\{O_2\}\}$
   - $MCGS_3 = \{\{O_1,O_4\},\{O_2,O_4\}\}$
   - $\{\{O_1,O_4\},\{O_4,O_3\},\{O_2,O_4\},\{O_4,O_3\}\}$
     → $MCGS_4 = \{\{O_1,O_4\},\{O_2,O_4\},\{O_3,O_4\}\}$

Bewertung von Objekt-Duplikation

+ Konvergenz
+ Kausalität durch vorgeschaltete Zustandsvektor-Analyse
+ Intentionserhaltung: erlaubt unterschiedliche Sichtweisen / kein gegenseitiges Überschreiben von Operationen
+ Konflikte werden explizit sichtbar
+ autonome Ausführung (→ ausschließlich lokales Wissen)
+ sofortige Ausführung lokaler Operationen (→ keine Verlängerung der Response Time)
- Korrektheit
- Erzeugung neuer Duplikate nicht immer intuitiv für den Benutzer
- Handhabung bei vielen Versionen eines Objekts (insbesondere bei iterativer Erzeugung multipler Versionen)
- Speicherbedarf für Historie

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

Dead Reckoning (1)

Dead Reckoning-Algorithmus

- Zustandsvorhersage ("Dead Reckoning"): Zustandsänderungen durch den Fortschritt der Zeit werden von jeder Instanz lokal berechnet, z.B. die Route eines Flugzeugs
- jedes Objekt wird von einer bestimmten Instanz k kontrolliert, z.B. von der Instanz des Flugzeugpiloten
- Zustandsänderungen durch Benutzeraktionen dürfen nur von k vorgenommen werden
- signifikante (d.h. ab einem bestimmten Grenzwert) nicht-vorhersehbare Zustandsänderungen werden von k als State Update propagiert

➡ signifikante Zustandsänderungen werden nur von k entdeckt und propagiert, z.B. eine Kollision zweier Flugzeuge bleibt von nicht-Kontrollinstanzen unbemerkt

- States werden unzuverlässig übertragen
- Fehlerabsicherung durch periodische State Updates → Soft State

Dead Reckoning (2)

- Erweiterungen
  - Kooperation mehrerer Benutzer auf einem Objekt: Operationen werden an k gesendet und dort serialisiert und propagiert
    → erhöhte Response Time
  - im Fehlerfall Kontrollübergabe an andere Instanz möglich
    → erfordert Auswahlverfahren
- pessimistisches Verfahren für synchrone kontinuierliche Anwendungen, z.B. für massive Distributed Virtual Environments (DVEs) und militärische Simulationen

Untersuchung der Konsistenzkriterien (1)

Definition: Instanz j hat eine Operation O_{i,t^o_i,t^*_i} empfangen, wenn sie einen State empfangen hat, der den Effekt von O_{i,t^o_i,t^*_i} enthält

Konsistenz

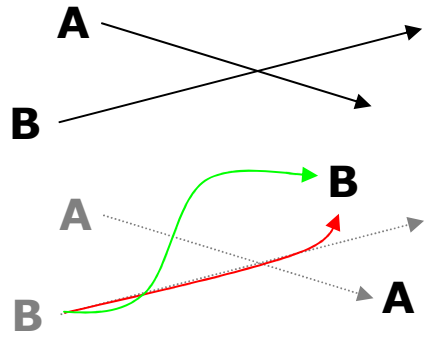
- seien i und j zwei Instanzen, die zum Zeitpunkt t hinreichend viele State Updates für ein Objekt empfangen haben, so dass alle Operationen mit $t^* \leq t$ bekannt sind
- dann ist der Zustand von i und j gleich (da von derselben Kontrollinstanz berechnet)

➡ Konsistenz wird eingehalten

Untersuchung der Konsistenzkriterien (2)

Korrektheit

- unzuverlässige Übertragung ganzer Zustände
- ➡ Verlust einzelner Benutzeraktionen ist nicht feststellbar
- Beispiel
  - i und j kontrollieren zwei Flugzeuge A und B auf Kollisionskurs
  - nach einer gewissen Zeit erhält i ein State Update für B, das B auf eine Position weg von A verschieben würde



  - falls State Updates verloren wurden, kann i nicht feststellen, ob eine Kollision stattgefunden hat oder B ausgewichen ist
  - i berechnet ggf. einen inkorrekten Zustand
- die virtuelle Instanz P empfängt dagegen alle Updates zuverlässig
- ➡ Korrektheit wird nicht eingehalten

Untersuchung der Konsistenzkriterien (3)

- weiterer Grund für inkorrekten Zustand: wenn die Kontrollinstanz ein State Update zu spät empfängt, kann es beim eigenen Update nicht mehr berücksichtigt werden
- die virtuelle Instanz P empfängt dagegen alle Updates rechtzeitig

Kausalität

- wird wegen der möglichen Paketverluste nicht eingehalten

Bewertung von Dead Reckoning

+ Konsistenz
+ geringe Komplexität: O(n)
  - abhängige Operationen und Wechselwirkungen zwischen Objekten müssen werden nur von der jeweiligen Kontrollinstanz festgestellt
  - daher gut geeignet für kontinuierliche Anwendungen mit hoher Benutzeranzahl und großer Anzahl an Objekten
- Kausalität und Korrektheit
- temporäre Inkonsistenzen sind wahrscheinlich
- Kodierung aller Zustandsänderungen als State
- eingeschränkte Kooperation bei reinem Dead Reckoning
- Nachteile zentralisierter Verfahren

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

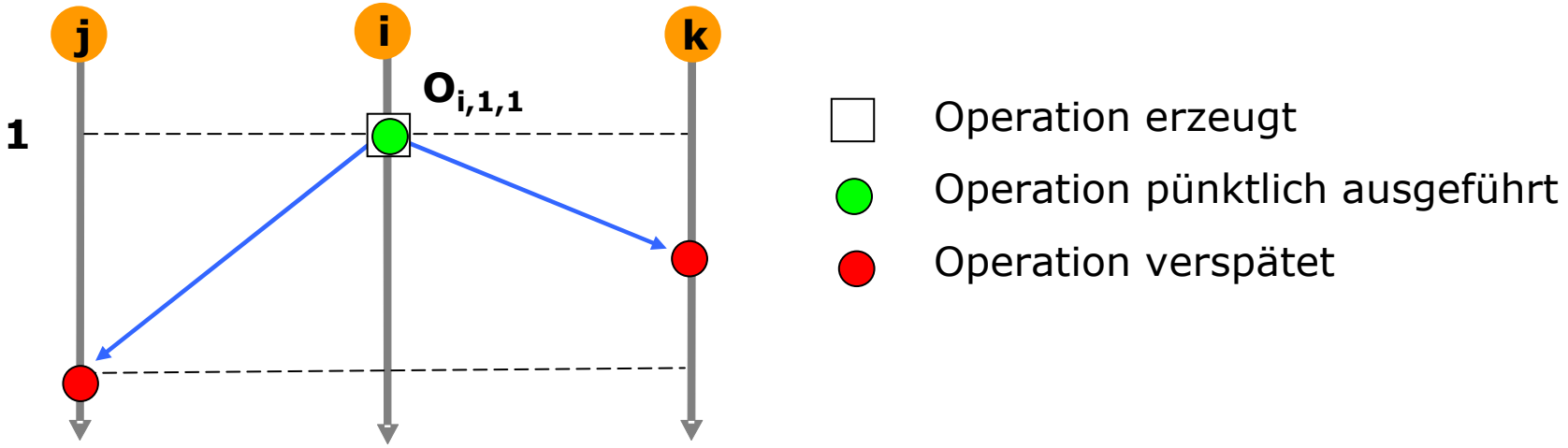
Local Lag – Motivation (1)

Beobachtung

Inkonsistenzen werden häufig durch die Netzwerkverzögerung verursacht: Empfang von Operationen O_{i,t^o_i,t^*_i}

- → in unterschiedlicher Reihenfolge
- → nach der geplanten Ausführungszeit

Beispiel für kontinuierliche Anwendung mit $t^o = t^* = 1$



➡ temporäre Inkonsistenz bei j und k: $S_t \neq S_{t^*}$

Local Lag – Motivation (2)

- Mechanismen zur Herstellung von Konsistenz / Korrektheit erforderlich mit den folgenden Nebenwirkungen
  1. temporäre Inkonsistenzen sind sichtbar und führen oft zu sekundären Inkonsistenzen
  2. Zustand muss korrigiert werden (→ Rechenaufwand)
  3. Anzeige des korrigierten Zustands führt zu Artefakten

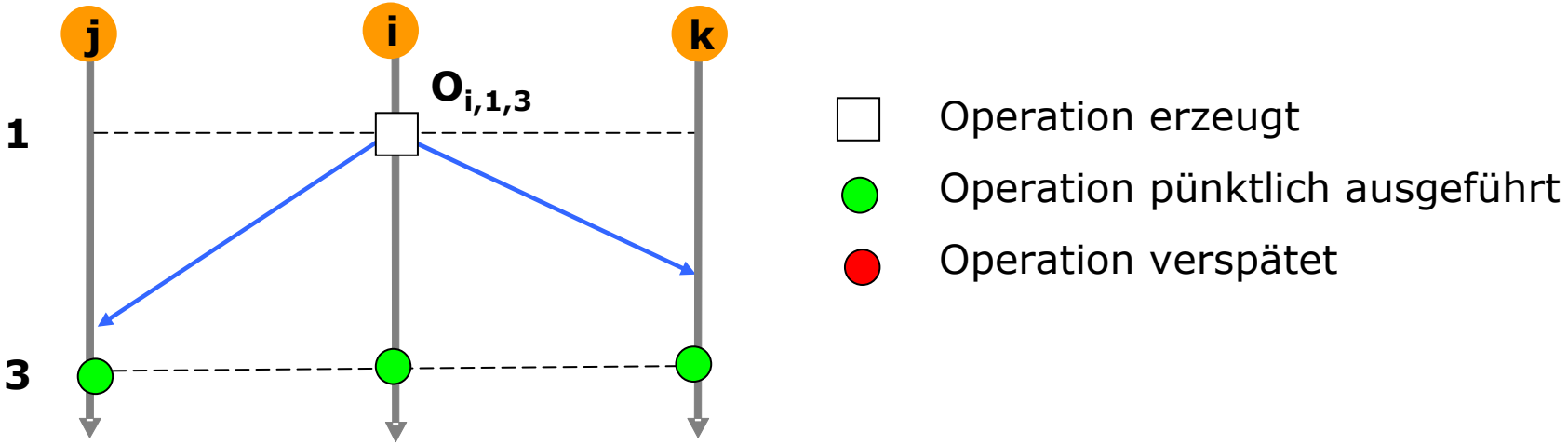
Idee: Verfahren zur Vermeidung von temporären Inkonsistenzen

Local Lag (1)

Idee: verhindere temporäre Inkonsistenzen durch $t^* > t^o$

- nutze die Zeitspanne $t^* - t^o$, um O_{i,t^o_i,t^*_i} an alle Instanzen zu übertragen
- im Idealfall ist die Übertragung vor Erreichen von t^* abgeschlossen, so dass alle Instanzen O zur gleichen Zeit t^* ausführen können

Beispiel für kontinuierliche Anwendung mit $t^o = 1$ und $t^* = 3$



Local Lag (2)

- zu früh empfangene Operationen werden bis zum Erreichen von t^* gepuffert
- die Zeitspanne $t^* - t^o$ nennt man **Local Lag**
- optimistisches Verfahren für (kontinuierliche und diskrete) synchrone Anwendungen

Untersuchung der Konsistenzkriterien

- falls alle Operationen vor ihrem Ausführungszeitpunkt empfangen werden, werden diese in zeitlicher Ordnung ausgeführt

➡ Einhaltung von Kausalität, Konvergenz, Konsistenz und Korrektheit

Temporäre Inkonsistenzen

Dauer I einer temporären Inkonsistenz für j und O_{i,t^o_i,t^*_i}:

$$I_j(O_{i,t^o_i,t^*_i}) = d(i,j) - (T^*_i - T^*_j) - (t^* - t^o)$$

- d(i,j) = Netzverzögerung zwischen i und j
- T^*_k = Wert einer gemeinsamen Uhr zu dem Zeitpunkt, an dem die physische Uhr von k den Wert t^* erreicht
- $T^*_i - T^*_j$ = Abweichung der physischen Uhren von i und j
- $t^* - t^o$ = Zeitspanne zwischen Erzeugung und Ausführung von O

Falls $I_j \leq 0$ → keine temporäre Inkonsistenz

Einfluss der Uhren-Abweichung

$$I_j(O_{i,t^o_i,t^*_i}) = d(i,j) - (T^*_i - T^*_j) - (t^* - t^o)$$

- falls $T^*_i >> T^*_j$, würde d(i,j) kompensiert, ohne dass Local Lag erforderlich ist (d.h. $t^* = t^o$)
- funktioniert nur in die Richtung i → j
- in der Richtung j → i ist die Verspätung dann umso größer

Auswahl eines Local Lag-Wertes (1)

Trade-Off

- hoher Wert wünschenswert, um die Wahrscheinlichkeit für temporäre Inkonsistenzen zu minimieren
- aber: hoher Wert für $t^* - t^o$ bedeutet hohe Response Time für den lokalen Benutzer, was ab einem gewissen Wert störend wirkt

➡ Kompromiss erforderlich

1) Wünschenswerter Wert für Local Lag l_{min}

- Ziel: $t^* - t^o \geq d(i,j) - (T^*_i - T^*_j)$ für möglichst viele O, i und j

➡ wähle max {d(i,j)} (z.B. 5ms im LAN, 40ms Kontinent, 150ms weltweit)

- zusätzlich maximale Uhrenabweichung (z.B. 10ms Linux, 50ms Windows)

➡ Local Lag wäre nur in Ausnahmesituationen nicht ausreichend (z.B. Paketverlust, Jitter)

Auswahl eines Local Lag-Wertes (2)

2) Höchste akzeptable Response Time r_{max}

- hängt vom Benutzer und der Anwendung ab
- sollte individuell durch Evaluation festgestellt werden
- für schnelle interaktive Anwendungen 50-100 ms
- im Idealfall höher als der minimale Wert aus Schritt 1

3) Auswahl des Local Lag-Wertes

- wenn $l_{min} < r_{max}$, setze $l_{min} < t^* - t^o < r_{max}$
- wenn $l_{min} > r_{max}$ → echter Trade-Off, evtl. Auflösung durch Evaluation

Implementierung

- Warteschlange für alle Operationen O_{i,t^o_i,t^*_i} , sortiert nach t^*
- führe O_{i,t^o_i,t^*_i} aus wenn t^* erreicht ist (und die Operation kausal ausführbar ist)
- ➡ neues Implementations-Paradigma für lokale Operationen
- traditionelle Vorgehensweise:
  1. führe Benutzeraktion aus,
  2. zeige neuen Zustand an und
  3. erzeuge und versende die entsprechende Operation
- mit Local Lag:
  1. erzeuge und versende Operation,
  2. sortiere Operation zusammen mit den empfangenen in die Warteschlange und
  3. führe Operation aus und zeige den neuen Zustand, sobald ihre Ausführungszeit erreicht wird

Bewertung von Local Lag

+ Einhaltung von Konvergenz, Konsistenz, Korrektheit und Kausalität im optimalen Fall
+ verhindert in der Praxis den größten Teil der temporären Inkonsistenzen bzw. verringert deren Dauer
+ geringer Aufwand zur Laufzeit
+ Fairness durch Angleichung von Response und Notification Time
- Wahl eines geeigneten Local Lag-Wertes
- Verlängerung der Response Time

Local Lag ist nicht ausreichend und muss mit anderen Konsistenz-erhaltungs-Verfahren kombiniert werden.

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

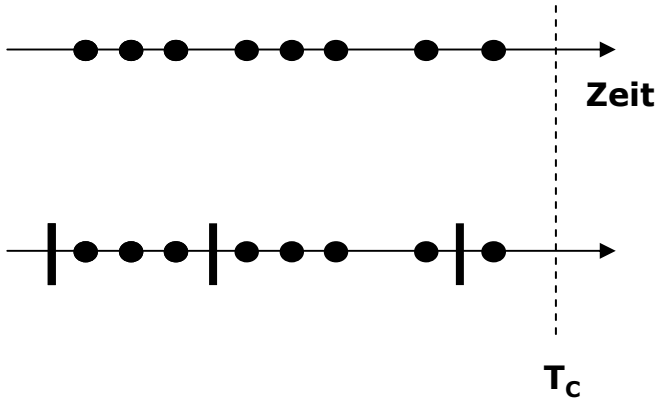
Timewarp

Ziel: Serialisierung von Operationen

- alle Instanzen führen alle Operationen in derselben Reihenfolge (und zum richtigen Zeitpunkt) aus → P

Voraussetzung

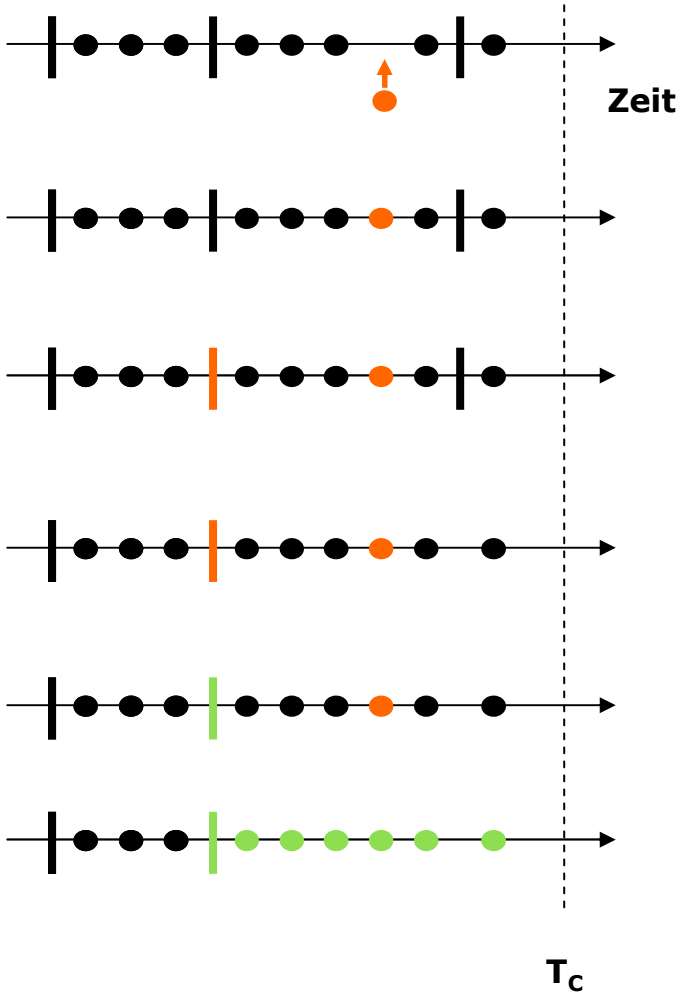
- jede Instanz i speichert eine lokale Operations-Historie H_i, die nach einer bestimmten Ordnung sortiert ist (z.B. Zustands-vektoren oder Ausführungszeit)
- H_i enthält alle lokalen und alle empfangenen Operationen
- zusätzlich speichert i periodisch den aktuellen Zustand S_{i,T_C} ("State Snapshot") in H_i (T_C = aktuelle Zeit)



Timewarp-Algorithmus

i empfängt eine Operation O_{j,t^o,t^*_x} mit $t^*_x < T_C$

1. füge O_{j,t^o,t^*_x} an der richtigen Stelle in H_i ein
2. bestimme den ersten Zustand $S_{i,t} \in H_i$ mit $S_{i,t} < O_{j,t^o,t^*_x}$
3. lösche alle (potentiell inkonsistenten) Zustände $S_{i,t'} > S_{i,t}$
4. setze den Zustand von i auf $S_{i,t}$
5. führe alle Operationen $H_{i,t} = \{O_{j,t^o,t^*} > S_{i,t}$ mit $t^* \leq T_C\}$ (im schnellen Vorlauf) aus
6. zeige neuen Zustand an



Bewertung von Timewarp (1)

+ Korrektheit (inkl. Konvergenz / Konsistenz)
+ alle diskreten und kontinuierlichen Anwendungen
+ autonome Ausführung (→ ausschließlich lokales Wissen)
+ sofortige Ausführung lokaler Operationen (→ keine Verlängerung der Response Time)
+ Verwendung der Operations-Historie für andere Funktionen
- Komplexität: $O(n^3)$ im Worst Case
  - sei n die Anzahl der Instanzen
  - jede Instanz erzeugt während einer bestimmten Zeitspanne eine beschränkte Anzahl von Operationen → n Operationen
  - um Abhängigkeiten zwischen den Objekten zu berücksichtigen, müssen alle Operationen untereinander verglichen werden → n^2 Vergleiche
  - pro Zeitspanne werden n Operationen empfangen, d.h. n Timewarps → n^3

Bewertung von Timewarp (2)

- Speicherbedarf für H_i, insbesondere für State Snapshots
  - ➡ Trade-Off bei Snapshot-Frequenz: Operationen pro Timewarp vs. Speicherverbrauch
- Implementierung des schnellen Ausführens von Operationen
- visuelle Artefakte bei (abrupter oder gradueller) Zustands-änderung

Verbesserungen von Timewarp: Übung

Kausalität

- kausale Ordnung kann mit SV hergestellt werden
- ggf. führt die notwendige Verzögerung einer Operation zu einem Timewarp

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  - Zustandsanfragen

Zustandsanfragen

Idee: Instanz i fordert externen Zustands zur Reparatur von temporären Inkonsistenzen an

- (1) Welche Instanz antwortet auf eine Zustandsanfrage?
- (2) Wie stellt man die Konsistenz des Antwort-Zustands sicher?

Anmerkung

Zusätzlicher Typ im Datenmodell: State, Event, Delta-State, Cue und **Query**

Auswahlverfahren ("Feedback Raise")

Problem: viele Antwortkandidaten aufgrund der replizierten Datenhaltung

1) Vorauswahl der antwortenden Instanz (~ Server)

2) dynamische Auswahl, z.B. per *Exponential Feedback Raise* (EFR)

- sende Anfrage an alle Instanzen
- jeder Kandidat i (i = 1,..,N) zieht eine Zufallszahl x $\in$ [0,1]
  - wenn x < 1/N → sende sofort Zustand an alle Instanzen
  - sonst stelle Feedback Timer: t = $T_{max}(1 + \log_N x)$ mit T_{max} maximale Wartezeit
- läuft der Timer aus, sende Zustand an alle Instanzen
- Empfang Zustand → lösche Feedback Timer
- Kandidaten = alle Instanzen ohne (bewusste) Inkonsistenz

➡ Idealfall: eine Antwort

➡ Timer bedeutet zusätzliche Wartezeit

Konsistenz des Antwort-Zustands

Problem: Kandidat j kann die Korrektheit / Vollständigkeit des eigenen Zustands S_j nicht garantieren (z.B. wenn verspätete Operation unterwegs)

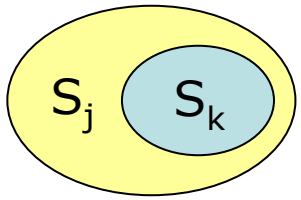
- Zustände sollten Meta-Informationen enthalten (z.B. Zustands-vektoren) → schnelle Überprüfung
- Korrektheit per iterativer Zustandsübertragung oder iterativer Zustandsanfrage
- Instanz i sollte bis zum erfolgreichen Abschluss der Anfrage lokale Benutzeraktionen verhindern (→ Vermeidung sekundärer Inkonsistenzen)

1. Iterative Zustandsübertragung

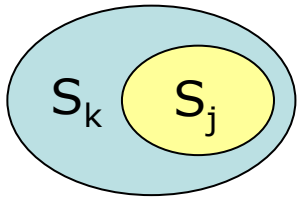
Jede Instanz k vergleicht einen empfangen Zustand S_j mit S_k

1. S_j und S_k enthalten dieselben Operationen
   → NOP

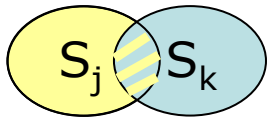
2. S_j enthält alle Operationen von S_k plus einige mehr
   → k hat einige Operationen verpasst und sollte S_j übernehmen



3. S_k enthält alle Operationen von S_j plus einige mehr
   → j hat einen inkorrekten / unvollständigen Zustand versendet
   → k versendet S_k zur Korrektur (→ Feedback Raise)



4. S_j und S_k enthalten unterschiedliche Operationsmengen
   → j und k sind temporär inkonsistent
   → warten auf korrekten Zustand S_l
   → oder Übernahme des besseren Zustands



2. Iterative Zustandsanfrage

Anfragende Instanz i überprüft Konsistenz von S_j

- vergleiche S_j mit verfügbaren Meta-Informationen (SV)
  - empfangene Operationen
  - periodische Sitzungsnachrichten
- bei entdeckter Inkonsistenz / Unvollständigkeit
  → wiederhole Zustandsanfrage
- dauert tendenziell länger als iterative Zustandsübertragung

Bewertung von Zustandsanfragen

Klassifikation

- optimistisches Verfahren für alle Anwendungen

Bewertung

+ Reparatur von Inkonsistenzen in Ausnahmesituationen, z.B. partitioniertem Netzwerk
- benötigt u.U. mehrere Iterationen
- potentiell unvollständiger Zustand als Ergebnis

Local Lag, Timewarp und Zustandsanfragen

Kombination der Verfahren

1. Local Lag: verhindert die meisten temporären Inkonsistenzen
2. Timewarp: behebt Inkonsistenzen auf Basis der lokalen Operations-Historie
3. Zustandsanfragen: falls lokale Reparatur unmöglich

Zusammenfassung

Replizierte Datenhaltung erfordert Konsistenzerhaltung

- Konsistenzkriterien: Kausalität, Konvergenz, Konsistenz, Korrektheit und Intentions-Erhaltung
- Ordnungen: kausale und globale Ordnung mit Zustandsvektoren, zeitliche Ordnung
- Inkonsistenzen: temporär und sekundär
- Hard State- und Soft State-Mechanismen
- optimistische und pessimistische Verfahren: Sperren, Abstimmen, Serialisierung, Operations-Transformation, Objekt-Duplikation, Dead Reckoning, Local Lag, Timewarp und Zustandsanfragen

➡ DAS optimale Verfahren gibt es nicht

➡ anwendungsspezifische Lösung

Literaturhinweise (1)

- Allgemein und Abstimm-Verfahren
  U.M. Borghoff, J.H. Schlichter, Computer-Supported Cooperative Work – Introduction to Distributed Applications, Springer Verlag, Berlin, Heidelberg, New York, 2000, Kapitel 4 und 5
- Zustandsvektoren
  Lamport, L. *Time, Clocks, and the Ordering of Events in a Distributed System*. In: Communications of the ACM, Vol. 21, No. 7, pages 558–565, 1978
- Allgemein und Operations-Transformation
  Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. *Achieving Convergence, Causality Preservation and Intention Preservation in Real-Time Cooperative Editing Systems*. In: ACM Transactions on Computer-Human Interaction, Vol. 5, No. 1, pages 63–108, 1998
- Objekt-Duplikation
  Sun, C. and Chen, D. *Consistency Maintenance in Real-Time Collaborative Editing Systems*. In: ACM Transactions on Computer-Human Interaction, Vol. 9, No. 1, pages 1–41, 2002.

Literaturhinweise (2)

- Dead Reckoning
  Srinivasan, S. *Efficient Data Consistency in HLA/DIS++*. In: Proc. ACM WSC, Coronado, CA, USA, pages 946–951, December 1996.
- Allgemein, Local Lag und Timewarp
  M. Mauve, Distributed Interactive Media, PhD Thesis, University of Mannheim, 2000
- Allgemein, Local Lag, Timewarp und Zustandsanfragen
  J. Vogel, Consistency Algorithms and Protocols for Distributed Interactive Applications, PhD Thesis, University of Mannheim, 2004
- Allgemein, Local Lag und Timewarp
  Mauve, M., Vogel, J., Hilt, V., and Effelsberg, W. *Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications*. In: IEEE Transactions on Multimedia, Vol. 6, No. 1, pages 45–57, 2004