Computergestützte Gruppenarbeit

7. Konsistenz

Dr. Jürgen Vogel

European Media Laboratory (EML)
Heidelberg

SS 2005

Inhalt der Vorlesung

1. Einführung
2. Grundlagen von CSCW
3. Gruppenprozesse
4. Benutzerschnittstelle
5. Zugriffsrechte und Sitzungskontrolle
6. Architektur
7. Konsistenz
8. Undo von Operationen
9. Visualisierung semantischer Konflikte
10. Late-Join
11. Netzwerk-Protokolle
12. Entwicklung von Groupware
13. Ausgewählte Groupware

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren

Replizierte Datenhaltung

Wiederholung: Groupware mit replizierter Datenhaltung

- jede Anwendungsinstanz verwaltet einen bestimmten Teil des Anwendungszustands
- Zustandsänderungen (Operationen) werden vom Urheber zu allen betroffenen Instanzen übermittelt und führen dort zur Aktualisierung des Zustands

Zentrale Herausforderungen

- **Synchronisation**: der Zustand eines bestimmten Objekts soll bei allen Anwendungsinstanzen identisch sein
  - eine Operation führt bei allen Objektkopien zu einem identischen Zustand
  - grundlegende Voraussetzung für gemeinsame Gruppenarbeit
  - auch von WYSIWIS gefordert
- **Replikations-Transparenz**: dem Benutzer soll (zu einem gewissen Grad) verborgen bleiben, dass er auf einer lokalen Kopie arbeitet

Synchronisations-Mechanismen

Konsistenzerhaltung ("Consistency Control")

- stellt sicher, dass Operationen bei allen Instanzen zu einem identischen Zustand führen
- ist kritisch bei gleichzeitigen Änderungen

Initialisierung von Anwendungsinstanzen ("Late-Join")

- Übergabe des aktuellen Zustands an Anwendungsinstanzen, die neu in eine bestehende Sitzung eintreten

Anmerkung: Synchronisations-Verfahren sind allgemein in verteilten Systemen erforderlich

Synchronisations-Arten

1) Eng gekoppelte Synchronisation

- jede Operation wird sofort propagiert
- minimale Notification Time
- direkte Zusammenarbeit zwischen Benutzern
- häufig bei synchroner Groupware

2) Lose gekoppelte Synchronisation

- Operationen werden gebündelt übertragen
- vorübergehende unbeeinflusste Arbeit einzelner Benutzer
- entspricht dem Zusammenführen ("Mergen") unterschiedlicher Objekt-Versionen
- häufig bei asynchroner Groupware

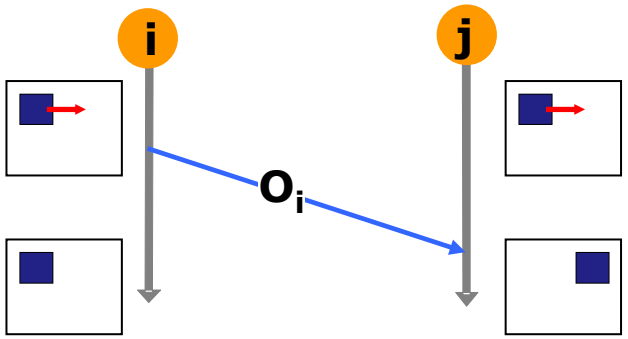
Zunächst betrachten wir ausschließlich (1)

Konsistenzerhaltung (1)

Gefährdung der Konsistenz

1) Szenario: synchrone kontinuierliche Anwendung

- Operation O_i unterliegt einer gewissen Netzverzögerung, so dass Notification Time > Response Time ≥ 0
- Ausführung von O_i bei j führt zu verschiedenen Zuständen (= Inkonsistenz)



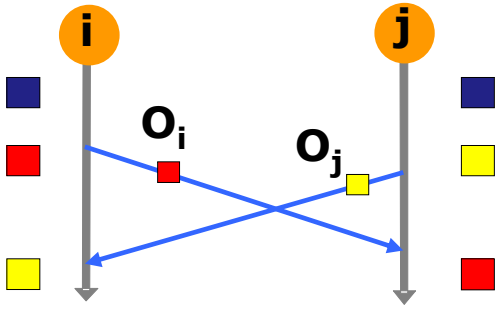
O_i = Stop

➡ Ausführungszeitpunkt einer Operation ist entscheidend

Konsistenzerhaltung (2)

2) Szenario: synchrone diskrete Anwendung

- Benutzer i und j ändern die Farbe eines Objekts fast zeitgleich mit O_i bzw. O_j
- die Netzverzögerung führt zu einer vertauschten Reihenfolge bei der Ausführung (i: O_i O_j, j: O_j O_i) und somit zu verschiedenen Objektzuständen (= Inkonsistenz)



➡ Reihenfolge bei der Ausführung von Operationen ist entscheidend

➡ Anwendung benötigt Mechanismen zur Konsistenzerhaltung

Abgrenzung zu Datenbanken (1)

Simultaner Zugriff mehrerer Benutzer auf ein Objekt wird auch in Datenbanken (DB) ermöglicht.

Eignung von DB für Groupware

- DB verfolgen Isolation, d.h. ein (fast) zeitgleicher Zugriff soll den Benutzern verborgen bleiben
  - Sperrverfahren
  - Änderungen erst nach Transaktionsende sichtbar
  - keine Benachrichtigung über Änderungen

  → behindert echtes kooperatives Arbeiten
- DB sind auf kurze Transaktionen ausgelegt, während in Groupware häufig lange an einem Objekt gearbeitet wird
  → lange Transaktionen kritisch bzgl. Sperren und Rücksetzen
- zentrale DB führen zu hoher Response (und Notification) Time
  → Nachteile zentraler Systeme

Abgrenzung zu Datenbanken (2)

- verteilte DB nutzen meist 2-Phase-Commit (2PC)
  → 2PC ist komplex und zeitaufwendig
- Groupware verwendet häufig komplexe (multimediale) Objekte
  → erfordert spezielle Unterstützung in einer DB

➡ DB und DB-Verfahren sind für viele CSCW-Systeme ungeeignet

➡ für Groupware werden meist spezielle Verfahren eingesetzt

➡ hoher Entwicklungs-Aufwand

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
  - diskrete Anwendungen
  - kontinuierliche Anwendungen
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren

Diskrete Anwendungen

- diskrete Anwendungen: Zustandsänderung durch Benutzeraktion
- Zustandsänderung O_k
  - als Event oder Delta-State: Update des aktuellen Zustands S
  - als State: ersetze den aktuellen Zustand S
- bei Ausführung einer Menge $\{O_k\}$ auf S_i einer Instanz i bestimmt die Ausführungsreihenfolge den neuen Zustand S'_i

➡ *Ordnung* von Operationen

- z.T. sind Operationen voneinander abhängig, z.B.:
  - O_1 erzeugt ein Rechteck
  - O_2 ändert die Farbe des Rechtecks
  - dann muss O_2 immer nach O_1 ausgeführt werden

- der Einfachheit halber betrachten wir im Folgenden o.B.d.A. einen unpartitionierten Anwendungszustand

Kausale Ordnung (1)

Definition

Seien $O_{i,a}$ und $O_{j,b}$ zwei Operationen der Instanzen i und j, dann gilt $O_{i,a} \rightarrow O_{j,b}$, wenn

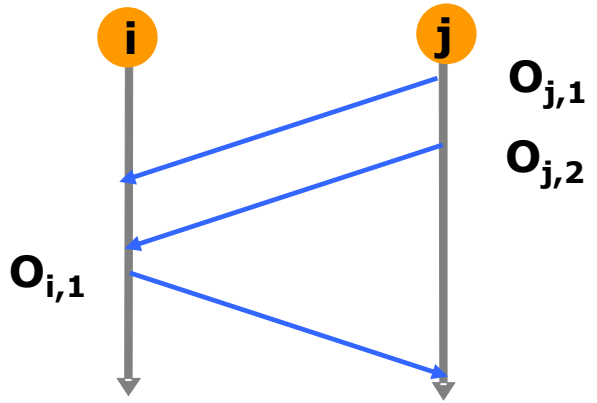
(1) i = j und $O_{i,a}$ wurde vor $O_{i,b}$ erzeugt oder

(2) $i \neq j$ und $O_{i,a}$ wurde ausgeführt, bevor $O_{j,b}$ erzeugt wurde, oder

(3) $\exists\ O_{k,c}$ mit $O_{i,a} \rightarrow O_{k,c}$ und $O_{k,c} \rightarrow O_{j,b}$

Wenn $O_{i,a} \rightarrow O_{j,b}$ heisst $O_{j,b}$ auch **abhängig** von $O_{i,a}$

Beispiel

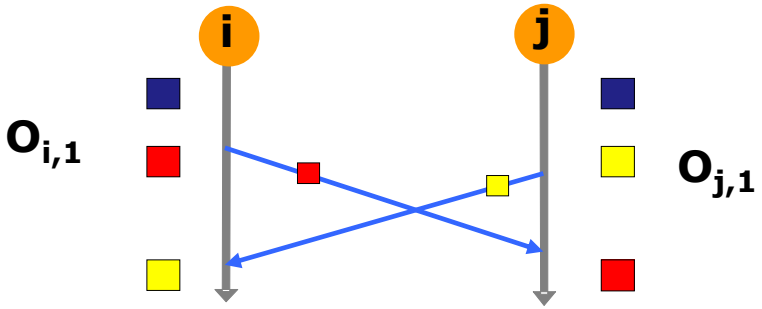


$O_{j,1} \rightarrow O_{j,2}$ und $O_{j,2} \rightarrow O_{i,1}$

Kausale Ordnung (2)

- wenn weder $O_{i,a} \rightarrow O_{j,b}$ noch $O_{j,b} \rightarrow O_{i,a}$, dann sind $O_{i,a}$ und $O_{i,b}$ **nebenläufig** ("concurrent"): $O_{i,a} \parallel O_{j,b}$
- wenn zwei nebenläufige Operationen $O_{i,a}$ und $O_{i,b}$ dieselben Attribute eines Objekts betreffen, dann heißen sie **konfliktär** ("conflicting"): $O_{i,a} \otimes O_{j,b}$

Beispiel



$O_{j,1} \parallel O_{i,1}$ und $O_{j,1} \otimes O_{i,1}$

Kausalität

Definition

Eine Anwendung garantiert **Kausalität** ("Causality"), wenn $\forall$ $O_{i,a}$, $O_{j,b}$ mit $O_{i,a} \rightarrow O_{j,b}$ bei allen Instanzen $O_{i,a}$ vor $O_{j,b}$ ausgeführt wird.

- Kausalität definiert eine partielle Ordnung für abhängige Operationen
- für Konsistenz ist eine Ordnung auf allen Operationen notwendig

Konvergenz

Definition

Ausgehend von einem identischen Initialzustand S^0 garantiert eine Anwendung **Konvergenz** ("Convergence"), wenn gilt $S_i = S_j \; \forall \; i,j$, nachdem alle Instanzen dieselbe Menge Operationen $\{O_k\}$ ausgeführt haben.

- Konvergenz betrifft den Anwendungszustand nach der Ausführung einer bestimmten Menge an Operationen
- (1) Konvergenz erlaubt Abweichungen bei den einzelnen Instanzen, solange noch nicht alle Operationen empfangen oder ausgeführt wurden
- (2) Konvergenz erfordert nicht (!), dass alle Operationen bei allen Instanzen in der selben Reihenfolge ausgeführt werden müssen
- daher sind wir auch daran interessiert, ob der Zustand *korrekt* ist

Korrektheit

Sei P eine virtuelle "perfekte" Instanz, die alle Operationen $O_{i,a}$ in der Reihenfolge ihrer Erzeugung ausführt.
Bei gleichzeitiger Erzeugung von $O_{i,a}$ und $O_{j,b}$ verwendet P ein zusätzliches Kriterium ("Tie-Breaker")).

- P berechnet den Zustand einer nicht verteilten Anwendung

Definition

Ausgehend von einem identischen Initialzustand S^0 garantiert eine Anwendung **Korrektheit** ("Correctness"), wenn gilt $S_i = S_P \; \forall \; i$, nachdem alle Instanzen dieselben Operationen ausgeführt haben.

- Korrektheit gilt wie Konvergenz nur für Instanzen, die alle erforderlichen Operationen besitzen
- Korrektheit $\Rightarrow$ Konvergenz

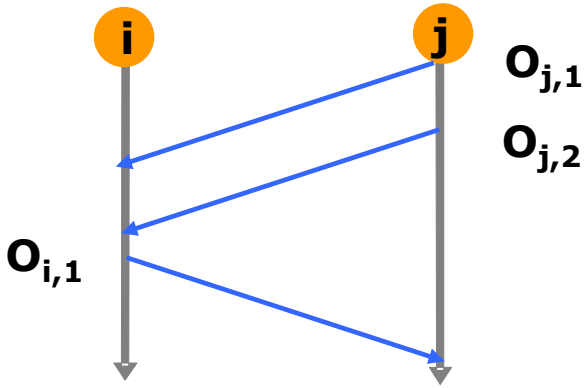
- Überprüfung von Kausalität, Konvergenz und Korrektheit?

Zustandsvektoren

Zustandsvektor ("State Vector")

- für jede Instanz i: Tupel (i,SN_i)
- sei n die Anzahl der Instanzen und $SN \in \mathbb{N}_0$ die Sequenznummer von i, dann gilt: $SV := \langle(1,SN_1),(2,SN_2),\ldots,(n,SN_n)\rangle$
- Definition: $SV[i] := SN_i$
- für jede Operation O_i von i inkrementiere SV[i] (beginnend bei 0)
- jeder Operation O_i ist der inkrementierte SV zugewiesen
- der SV eines Zustands S_i gibt die auf ihm ausgeführten Operationen wieder

Beispiel



$SV_{O_{j,1}} = \langle(i,0),(j,1)\rangle$

$SV_{O_{j,2}} = \langle(i,0),(j,2)\rangle$

$SV_{O_{i,1}} = \langle(i,1),(j,2)\rangle$

Überprüfen von Kausalität mit SV (1)

Sei SV_{O_i} der Zustandsvektor von O_i und SV_j der Zustandsvektor von Instanz j, wenn diese O_i empfängt.

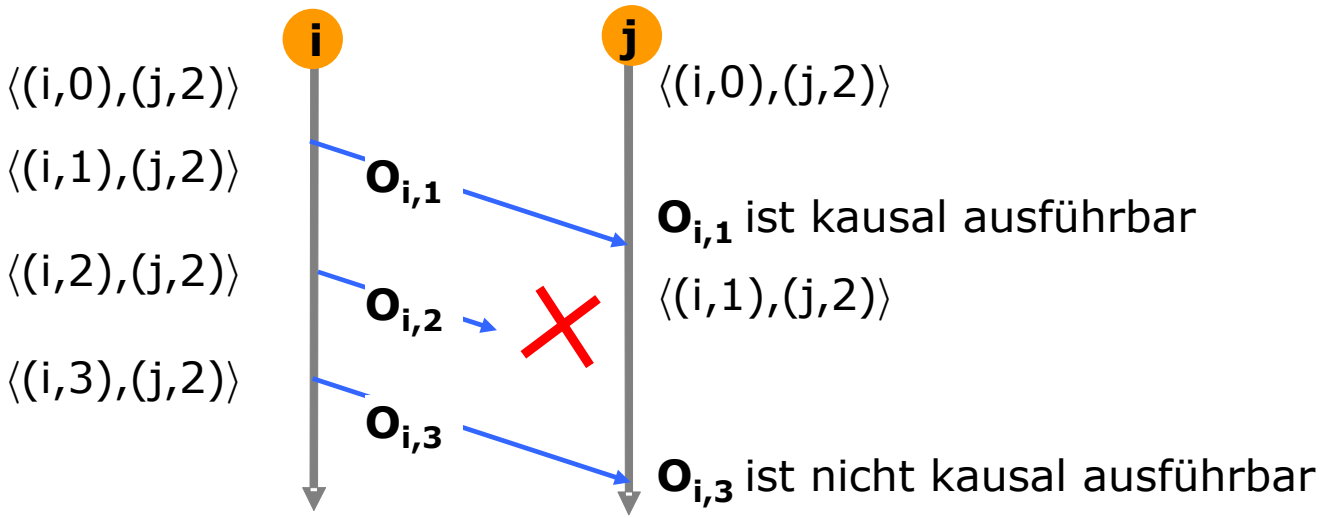
Dann kann O_i von j ausgeführt werden, wenn

(1) $SV_{O_i}[i] = SV_j[i] + 1$ und

(2) $SV_{O_i}[k] \leq SV_j[k] \; \forall \; k \neq i$

In diesem Fall nennt man O_i **kausal ausführbar** ("causally ready")

Beispiel



Überprüfen von Kausalität mit SV (2)

- bevor O_i von j ausgeführt werden kann, wurden alle Operationen, von denen O_i abhängt, von j empfangen und ausgeführt
- ist O_i nicht kausal ausführbar, muss sie gepuffert werden, d.h. die Notification Time erhöht sich
- alle lokalen Operationen sind per Definitionem bei ihrem Urheber kausal ausführbar, d.h. können mit theoretischer Response Time von 0 ausgeführt werden

Überprüfen von Korrektheit mit SV

Für die Überprüfung von Konvergenz und Korrektheit wird eine Ordnung auf allen Operationen benötigt.

Definition

Seien O_i und O_j zwei Operationen der Instanzen i und j, SV_{O_i} und SV_{O_j} die dazugehörigen Zustandsvektoren und $sum(SV) := \sum_k SV[k]$.

Dann ist $O_i < O_j$ wenn

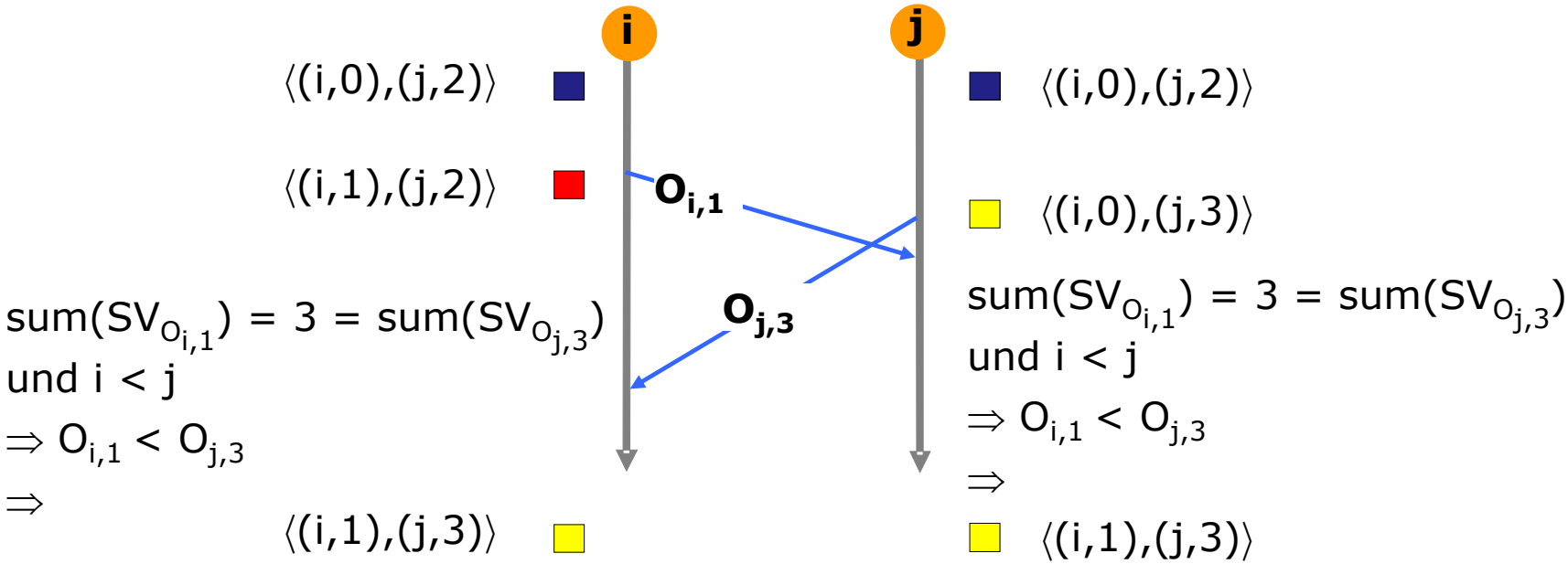
(1) $sum(SV_{O_i}) < sum(SV_{O_j})$ oder

(2) $sum(SV_{O_i}) = sum(SV_{O_j})$ und $i < j$.

Korrektheit liegt dann vor, wenn alle Instanzen ein Menge von Operationen so ausführen, dass derselbe Zustand erreicht wird, der durch Ausführung dieser Operationen nach obiger Ordnung bestimmt wird.

Anmerkung: Im zweiten Fall sind auch andere Tie-Breaker denkbar.

Beispiel


i
j
⟨(i,0),(j,2)⟩
⟨(i,1),(j,2)⟩
$O_{i,1}$
$O_{j,3}$
⟨(i,0),(j,2)⟩
⟨(i,0),(j,3)⟩
$sum(SV_{O_{i,1}}) = 3 = sum(SV_{O_{j,3}})$
und i < j
$\Rightarrow O_{i,1} < O_{j,3}$
$\Rightarrow$
$sum(SV_{O_{i,1}}) = 3 = sum(SV_{O_{j,3}})$
und i < j
$\Rightarrow O_{i,1} < O_{j,3}$
$\Rightarrow$
⟨(i,1),(j,3)⟩
⟨(i,1),(j,3)⟩

Bemerkungen zur globalen Ordnung

Eine Folge von Operationen, die nach < geordnet ist, genügt auch der Kausalität:

Es gilt $O_i \rightarrow O_j \Rightarrow O_i < O_j$.

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
  - diskrete Anwendungen
  - kontinuierliche Anwendungen
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren

Kontinuierliche Anwendungen

Zustandsänderung durch

- (1) Benutzeraktion: Übertragung als Operation
- (2) Fortschreiten der Zeit: automatische lokale Berechnung

➡ bei Ausführung einer Menge $\{O_k\}$ auf S_i einer Instanz i bestimmen die Ausführungsreihenfolge und die Ausführungszeitpunkte den neuen Zustand S'_i

- der Einfachheit halber betrachten wir im Folgenden o.B.d.A. einen unpartitionierten Anwendungszustand

Zeit in kontinuierlichen Anwendungen

Voraussetzung: jede Instanz besitzt eine Uhr mit hinreichender Genauigkeit und Synchronität

Synchronisation von Uhren

- Network Time Protocol (NTP)
- GPS-Uhren
- ➡ vollständige Synchronität ist praktisch nicht zu erreichen (Ungenauigkeiten in Hardware oder Betriebssystem

Verwendung des Begriffs "Zeit"

- Gegeben sei eine physische Uhr. Dann bezeichnen wir mit Zeit einen bestimmten Wert dieser Uhr.
- Aufgrund von Ungenauigkeiten kann es sein, dass nicht alle Uhren der einzelnen Anwendungsinstanzen diesen Wert gleichzeitig erreichen.

Zeitliche Ordnung

Notationen

- $S_{i,t}$: Zustand der Instanz i zum Zeitpunkt t
- O_{i,t^o,t^*}: Operation der Instanz i, die zum Zeitpunkt t^o erzeugt und zum Zeitpunkt t^* ausgeführt wird
- zur Vereinfachung nehmen wir an
  - die Auflösung der Uhr reicht aus, die gleichzeitige Erzeugung zweier Operationen zu verhindern (sonst: Tie-Breaker)
  - zunächst: $t^o = t^*$
- H: Operations-Historie = Menge aller Operationen in einer Sitzung

Definition

Seien O_{i,t^o_i,t^*_i} und O_{j,t^o_j,t^*_j} zwei Operationen, dann gilt

$O_{i,t^o_i,t^*_i} < O_{j,t^o_j,t^*_j}$, wenn (1) $t^*_i < t^*_j$ oder (2) $t^*_i = t^*_j$ und $i < j$

Konsistenz

Definition

Eine kontinuierliche Anwendung garantiert **Konsistenz** ("Consistency"), wenn zu jedem Zeitpunkt t bei allen Instanzen i und j, die alle Operationen $O_{k,t^o,t^*} \in H$ mit Ausführungszeitpunkt $t^* \leq t$ empfangen haben, die Zustände $S_{i,t}$ und $S_{j,t}$ identisch sind.

Konsistenz

- betrifft den Zustand nach der Ausführung eines bestimmten Teils von H
- ➡ erlaubt Abweichungen bei den einzelnen Instanzen, solange noch nicht alle Operationen empfangen oder ausgeführt wurden
- ➡ erfordert nicht (!), dass alle Operationen bei allen Instanzen in der selben Reihenfolge und zum selben Zeitpunkt t^* ausgeführt werden müssen
- ist unabhängig von der Synchronität der Uhren: gleiche Zustände bei einem bestimmten Wert der gemeinsamen Uhr, unabhängig davon, wann dieser erreicht wird

Korrektheit

Sei P eine virtuelle "perfekte" Instanz, die alle Operationen $O_{i,t^o,t^*} \in H$ zum Zeitpunkt t^* ausführt und somit die zeitliche Ordnung einhält.

- P berechnet den Zustand einer nicht verteilten Anwendung

Definition

Eine Anwendung garantiert **Korrektheit** ("Correctness"), wenn zu jedem Zeitpunkt t für alle Instanzen i, die alle Operationen $O_{k,t^o,t^*} \in H$ mit Ausführungszeitpunkt $t^* \leq t$ empfangen haben, gilt $S_{i,t} = S_{P,t}$.

- Korrektheit ist wie Konsistenz unabhängig von der Uhren-Synchronität und gilt nur für Instanzen mit den notwendigen Operationen
- legt keine bestimmte Reihenfolge oder Ausführungszeitpunkte fest (der Zustand muss nur so sein als ob!)

Kausalität

Kausalität: Ausführungsreihenfolge abhängiger Operationen

- wichtig auch für kontinuierliche Anwendungen, z.B. um zu verhindern, dass eine Operation ausgeführt wird, bevor das Zielobjekt erzeugt wurde
- aber: wenn $O_i \rightarrow O_j$ und k empfängt O_j zuerst, müsste O_j bis zum Empfang von O_i verzögert werden

➡ potentiell temporäre Inkonsistenzen

➡ u.U. sollte eine kontinuierliche Anwendung Kausalität daher (für bestimmte Operationen) ignorieren

Zusammenfassung

- diskrete Anwendungen: Kausalität, Konvergenz und Korrektheit
- kontinuierliche Anwendungen: Konsistenz und Korrektheit
- Konsistenzkriterien sagen nichts über die Semantik von Operationen aus, d.h. ein Zustand kann aus der Sicht des Benutzers unlogisch sein, obwohl ihn das System für korrekt befindet

Temporäre Inkonsistenzen

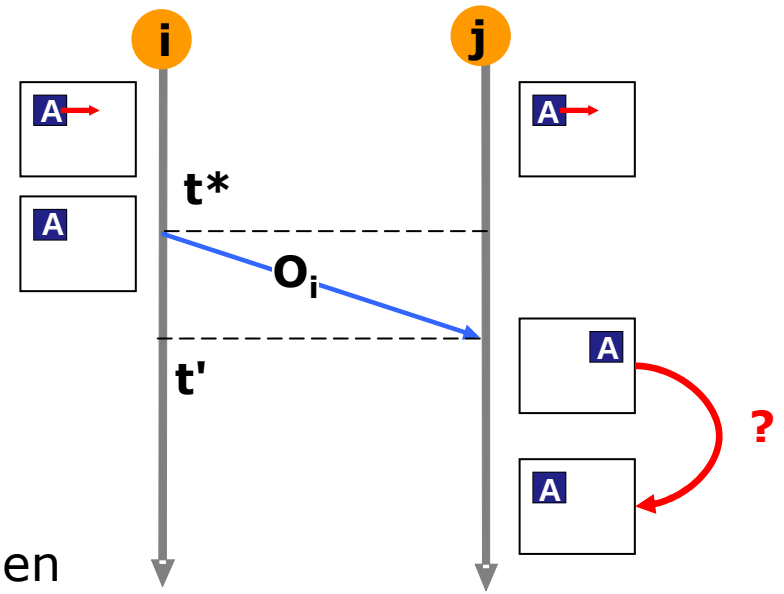
- Konvergenz, Konsistenz und Korrektheit erheben keine Forderungen an Situationen, in denen eine Instanz noch nicht über alle notwendigen Operationen verfügt
- ➡ **temporäre Inkonsistenzen** ("Artefakte") sind möglich

Beispiel

- sei O_{i,t^o,t^*} eine Stopp-Operation für ein bewegtes Objekt A
- falls j O_{i,t^o,t^*} nach t^* empfängt, ist A auf einer falschen Position
- ➡ bis t' ist S_j korrekt
- ➡ bei t' muss j die Position von A berichtigen, um Korrektheit herzustellen

  (1) abrupte Änderung

  (2) interpolierte graduelle Änderung



Sekundäre Inkonsistenzen

- während einer temporären Inkonsistenz sieht der Benutzer einen falschen Zustand
- das System bemerkt die temporäre Inkonsistenz erst beim Empfang der betroffenen Operation
- der Benutzer kann aber weiterhin Aktionen ausführen

➡ Operationen auf inkorrektem Zustand

➡ **sekundäre Inkonsistenzen** (= semantisch)

Beispiel

- Zug-Simulation mit zwei Instanzen i und j
- Zug fährt auf eine Weiche zu, i stellt die Weiche mit O_i
- j empfängt O_i erst nachdem der Zug die Weiche passiert hat

➡ j's Zug ist auf falscher Position

- wenn sich j darüber wundert, dass der Zug auf eine Nebenstrecke fährt, zieht er die Bremse

➡ wäre O_i rechtzeitig angekommen, hätte j dies nicht getan

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
  - Soft State vs. Hard State
  - pessimistische vs. optimistische Konsistenzerhaltung
- Ausgewählte Verfahren

Soft State (1)

Soft State-Verfahren

- eine Instanz versendet periodisch den aktuellen Anwendungs-zustand als State an alle anderen Instanzen (=Ankündigung), sofern dies noch nicht von einer anderen Instanz gemacht wurde
- geänderte oder neue Objekte werden implizit durch diese Ankündigungen übermittelt
- (Ankündigungs-)Intervall = Zeitspanne zwischen zwei auf-einanderfolgenden States
- jede Instanz merkt sich den Zeitpunkt der letzten Ankündigung
- empfängt eine Instanz für ein bestimmtes Objekt für mehrere Intervalle keine Ankündigung, wird dieses gelöscht ("Timeout")
- alle Ankündigungen werden unzuverlässig übertragen; Paket-verluste werden durch nachfolgende Ankündigungen repariert

➡ lose gekoppelte Synchronisation

Das Soft State-Verfahren wird oft in Protokollen verteilter Systeme eingesetzt, z.B. in RTP, RSVP, SAP und RIP.

Soft State (2)

Vorteile

+ alle möglichen Fehler und Inkonsistenzen werden durch die Ankündigungen implizit behoben
  - Konsistenzerhaltung
  - Initialisierung von Instanzen
  - Behandlung von Paketverlusten
+ sehr robust
+ geringe Komplexität
+ einfach zu implementieren

Nachteile

- hohe Notification Time (abhängig von Intervall und Verlustrate)
- nicht vorhersehbare Notification Time
- häufig temporäre Inkonsistenzen
- wiederholter Paketverlust kann zu falschen Timeouts führen
- hohe Datenrate durch periodische Übertragung des Zustands

Hard State (1)

Hard State-Verfahren

- einmalige, explizite und sofortige Benachrichtigung über neue, geänderte und gelöschte Objekte
- Übertragung von Lösch- und Änderungsoperationen als Event (Cue), neue Objekte als States
- zuverlässige Übertragung aller Operationen
- ➡ enge (oder lose) gekoppelte Synchronisation

- wird häufig auf der Anwendungsebene von verteilten Systemen verwendet, z.B. in Groupware

Hard State (2)

Vorteile

+ geringe und besser vorhersehbare Notification Time
+ geringere Wahrscheinlichkeit für temporäre Inkonsistenzen
+ effiziente Datenübertragung und geringe Datenrate

Nachteile

- hohe Komplexität, die Anwendung benötigt
  - Konsistenzerhaltungs-Mechanismen zur Behandlung verspäteter oder falsch sortierter Operationen
  - Late-Join-Verfahren
  - zuverlässiges Transportprotokoll
- aufwendige Implementierung und Fehlersuche

Wenn nicht anders angegeben, verwenden die nachfolgend vorgestellten Konsistenzerhaltungs-Mechanismen Hard State.

Inhalt

- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
  - Soft State vs. Hard State
  - pessimistische vs. optimistische Konsistenzerhaltung
- Ausgewählte Verfahren

Pessimistische Konsistenzerhaltung

- explizite Kontrolle des Objektzugriffs
- gleichzeitige Modifikation eines Objekts durch verschiedene Benutzer wird verhindert (z.B. durch Floor Control mit mutually-exclusive Politik)
- implizite Einhaltung der definierten Konsistenzkriterien

Bewertung

+ keine temporären Inkonsistenzen
- kann zu eingeschränkter Responsiveness führen
- verhindert manche Formen der Kooperation (z.B. gemeinsames Aufheben eines Gegenstands in einer virtuellen Welt)

Optimistische Konsistenzerhaltung

- jeder Benutzer darf zu jeder Zeit alle Objekte lesen und ändern
- gleichzeitige konfliktäre Änderungen sind möglich
- mögliche (temporäre) Inkonsistenzen
- explizite Maßnahmen zur Einhaltung der Konsistenzkriterien

Bewertung

+ unterstützt alle Kooperationsformen
+ hohe Responsiveness
+ gut geeignet für Szenarien mit kurzer Notification Time und geringer Wahrscheinlichkeit für echten parallelen Zugriff
- temporäre (primäre) Inkonsistenzen
- sekundäre Inkonsistenzen

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
- 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.)
- Verwendung von Schreib- und Lese-Rechten auf Objekten
- mehrstufiges Verfahren: (1) Rechteerwerb, (2) Aktion
- Erwerb eines Rechts durch Abstimmen
  - Abstimmungs-Anfrage an alle Instanzen
  - Votum = # zustimmende Antworten
  - Quorum = untere Grenze für erfolgreiches Votum
- Schreiben/Lesen nur bei erfolgreicher Abstimmung
- Lesen
  - Votum = # höchste SN der erreichbaren Instanzen
  - Quorum = erforderliche # Instanzen mit höchster SN
  - Beispiel: n = 4, $SN_1 = 5$, $SN_2 = 6$, $SN_3 = 6$ und Anfrage von 4
    - → 2 und 3 besitzen den aktuellen Zustand
    - → Mehrheit erreicht, d.h. positives Votum
    - → Lesezugriff von 4 liefert Objektzustand mit SN = 6

Abstimm-Verfahren (2)

- Schreibzugriff von Instanz j
  - Übergabe der Zustandsänderung als State an alle erreichbaren Instanzen i (mit inkrementierter SN_j)
  - Vergleich SN_j mit lokaler SN_i: positive Antwort, wenn $SN_j > SN_i$
  - Votum = # positive Antworten
  - Quorum = erforderliche # positiver Antworten
  - bei erfolgreicher Abstimmung, Übernahme des States durch alle erreichbaren Instanzen
  - Beispiel: n = 4, $SN_1 = 5$, $SN_2 = 6$, $SN_3 = 6$, $SN_4 = 6$
    - → Schreibzugriff von 4 mit $SN_4 = 7$
    - → 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 $SN_4 = 6$
    - → maximal eine positive Antwort von 1
    - → Zugriff abgelehnt

Abstimm-Verfahren (3)

- 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): $Quorum_{read} = 1$, $Quorum_{write} = n$
- ➡ 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
+ gut geeignet für asynchrone Anwendungen
+ robust bzgl. Ausfall von Instanzen und Netzwerkfehlern
- temporäre Inkonsistenzen sind zugelassen
- Kausalität und Korrektheit
- Kodierung aller Zustandsänderungen als State

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

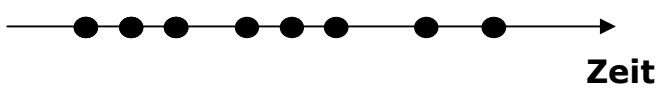
Serialisierung

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

- optimistisches Verfahren für diskrete Anwendungen

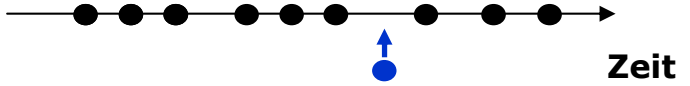
Voraussetzung

- 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

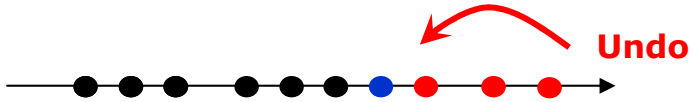


Serialisierungs-Algorithmus

i empfängt Operation O_j in falscher Reihenfolge

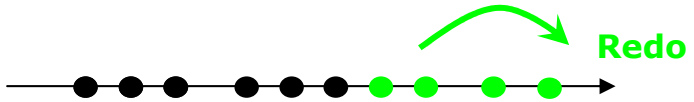


1. mache alle Operationen $O_k \in H_i$ mit $O_k > O_j$ rückgängig



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 ausführbare 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

Intentions-Erhaltung

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 **Intentions-Erhaltung** ("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" und j: S_1 = "ABE"
- kombinierter intentions-erhaltender Zustand: S_1 = "A12BE"
- Ergebnis mit Serialisierung $O_i\ O_j \rightarrow S_1$ = "A1CDE" bzw. $O_j\ O_i \rightarrow S_1$ = "A12BE"

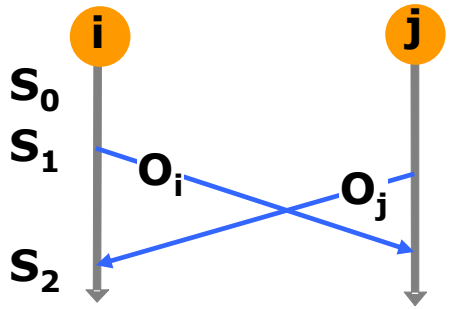
Operations-Transformation (1)

Idee: führe alle Operationen sofort aus (in beliebiger Reihenfolge), so dass Intentions-Erhaltung 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

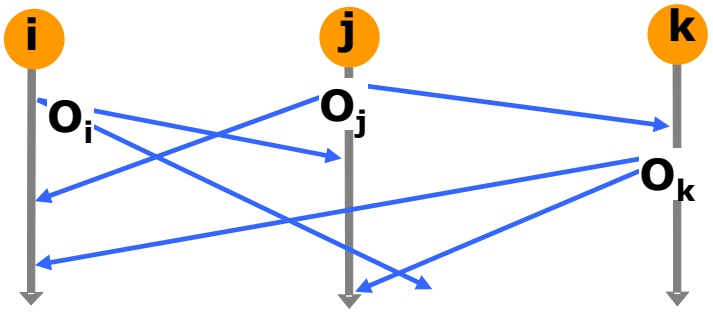
→ transformiere O_j so, dass die Änderung durch O_i berücksichtigt wird: O_j' = "lösche von Index 4 bis 5" → S_2 = "A12BE"

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** $O \mapsto O'$, so dass $O_j'(O_i(S)) \equiv O_i'(O_j(S))$.

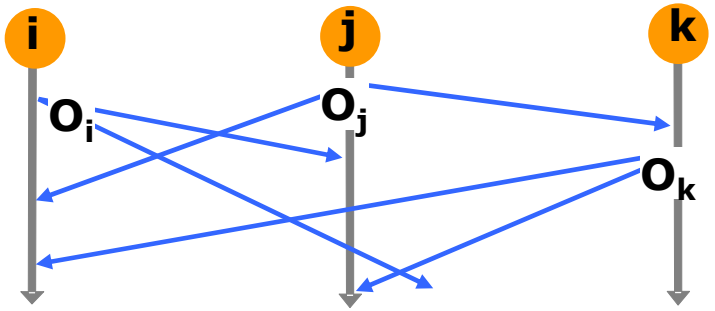
Betrachtung der Anfangszustände

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



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 Anfangszustände wegen $O_j \rightarrow O_k$

➡ finde **Exklusions-Transformation** $O_k \mapsto O_k'$, 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 Intentions-Erhaltung
- Intentions-Erhaltung 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

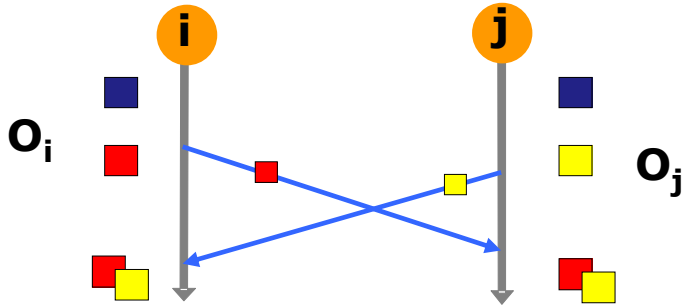
+ Konvergenz und Korrektheit
+ Intentionserhaltung
+ Kausalität durch vorgeschaltete Zustandsvektor-Analyse
+ autonome Ausführung (→ ausschließlich lokales Wissen)
+ sofortige Ausführung lokaler Operationen (→ keine Verlängerung der Response Time)
- komplexe Transformationsfunktionen
- Speicherbedarf 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 Objekt-IDs
  - 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
   - $MCGS_1 = \{\{O_1\}\}$
   - $MCGS_2 = \{\{O_1\},\{O_2\}\}$
   - $MCGS_3 = \{\{O_1,O_4\},\{O_2,O_4\}\}$
   - $\{\{O_1,O_4\},\{O_4,O_3\},\{O_2,O_4\},\{O_4,O_3\}\}$

     $\rightarrow MCGS_4 = \{\{O_1,O_4\},\{O_2,O_4\},\{O_3,O_4\}\}$

Bewertung von Objekt-Duplikation

+ Konvergenz
+ Kausalität durch vorgeschaltete Zustandsvektor-Analyse
+ Intentions-Erhaltung: 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 mehrer 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

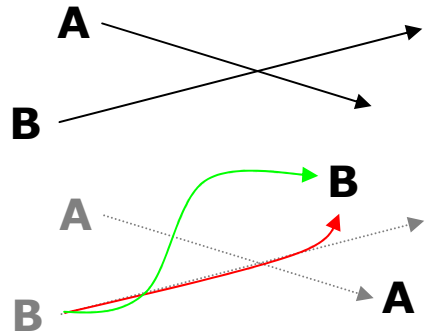
- seien i und j zwei Instanzen, die zum Zeitpunkt t hinreichend viele State Updates für ein Objekt empfangen haben, so dass alle Operationen mit $t^* \leq t$ bekannt sind
- dann ist der Zustand von i und j gleich (da von derselben Kontrollinstanz berechnet)

➡ 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

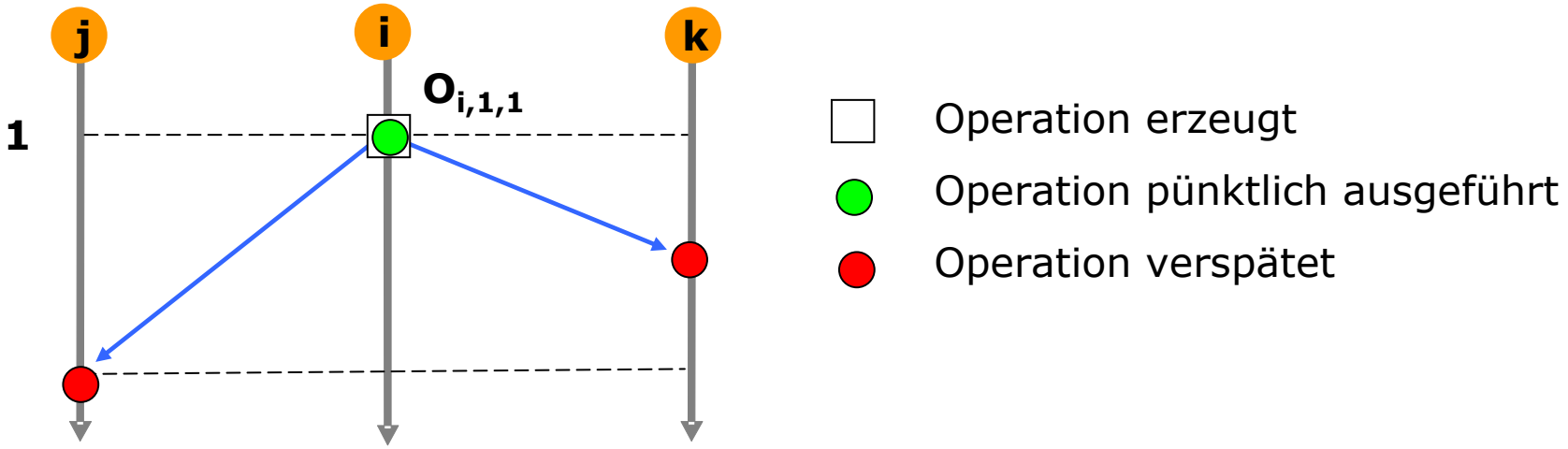
- Einführung in die Synchronisation replizierter Daten
- Konsistenzkriterien
- Klassifikation von Konsistenzerhaltungs-Verfahren
- Ausgewählte Verfahren
  - Sperren
  - Abstimmen
  - Serialisierung
  - Operations-Transformation
  - Objekt-Duplikation
  - Dead Reckoning
  - Local Lag
  - Timewarp
  -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

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

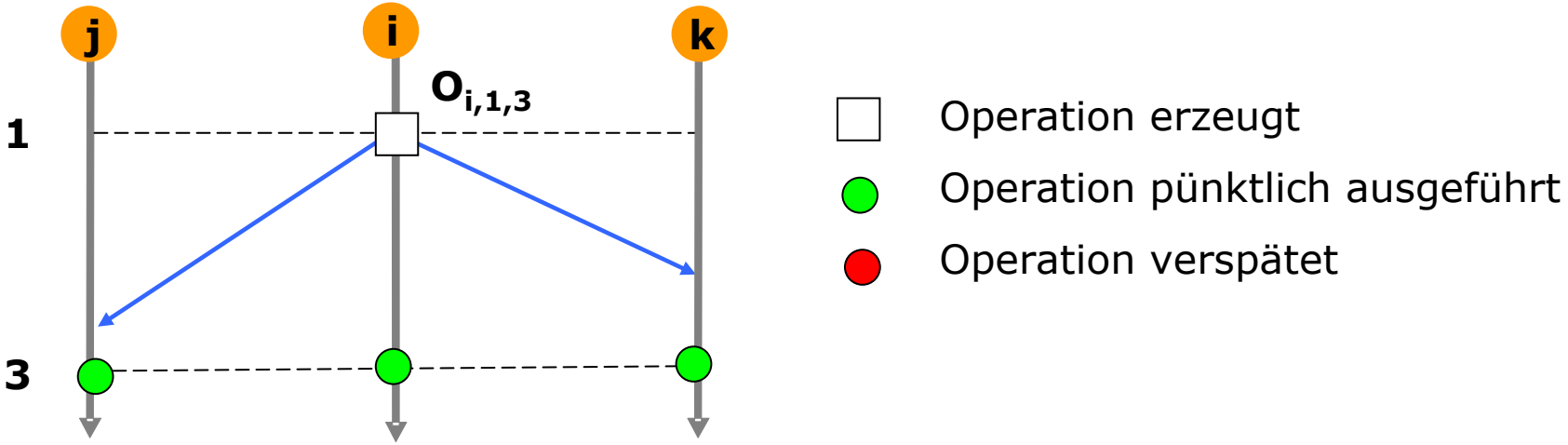
Falls $I_j \leq 0$ → keine temporäre Inkonsistenz

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

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

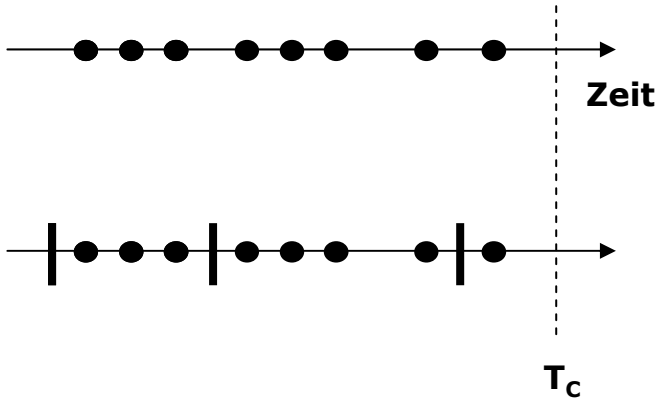
Timewarp

Ziel: Serialisierung von Operationen

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

Voraussetzung

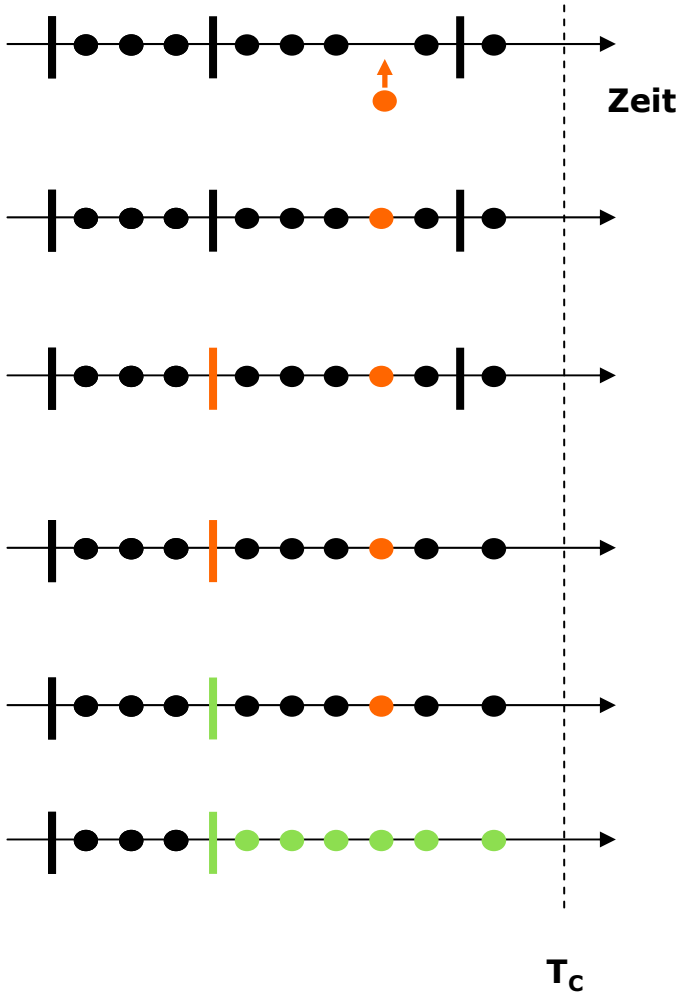
- jede Instanz i speichert eine lokale Operations-Historie H_i, die nach einer bestimmten Ordnung sortiert ist (z.B. Zustands-vektoren oder Ausführungszeit)
- H_i enthält alle lokalen und alle empfangenen Operationen
- zusätzlich speichert i periodisch den aktuellen Zustand S_{i,T_C} ("State Snapshot")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

Verbesserung des Timewarp-Algorithmus:

- Rechenaufwand
  - nachgeführte Zustände
  - rundenbasierter Timewarp
  - Timewarp mit Operations-Filter
- Speicherbedarf
  - Beschränkung der Historie

Nachgeführte Zustände

Timewarp: lösche alte Zustände $S_{i,t'}$ (Schritt 3)

- H_i enthält weniger Zustände
- zukünftige Timewarps erfordern größere Rücksprünge und $H_{i,t}$ beinhaltet mehr Operationen

Idee: ersetze $S_{i,t'}$ während des Timewarps (Schritt 5) durch einen aktualisierten Zustand

- Anzahl von Zuständen in H_i bleibt konstant
- geringerer Mehraufwand durch Speichern von Zuständen

Rundenbasierter Timewarp (1)

Timewarp: jede Operation mit $t^* < T_C$ löst einen Timewarp aus

- Timewarp verbraucht gewisse Rechenzeit
- ➡ im ungünstigsten Fall selbstverstärkender Effekt

Idee: rundenbasierter Timewarp

- Anwendung sammelt alle Operationen innerhalb einer Zeitspanne T (=Runde)
- die Operation mit der kleinsten Ausführungszeit bestimmt den Startzustand $S_{i,t}$ und den auszuführenden Teil der Historie $H_{i,t}$
- $S_{i,t}$ ist entweder der in der letzten Runde berechnete Zustand oder ein älterer
- ➡ pro Runde T wird höchstens ein Timewarp ausgeführt, unabhängig von der Anzahl der verspäteten Operationen
- ➡ Reduktion der Komplexität auf $O(n^2)$
- ➡ verhindert Folge-Timewarps

Rundenbasierter Timewarp (2)

- gut geeignet für kontinuierliche Anwendungen
  - periodisches Update des angezeigten Zustands
  - z.B. Spiele mit x FPS
  - wähle T entsprechend der Update-Frequenz (Rechenzeit pro Timewarp berücksichtigen)
  - ist T klein genug, merkt der Benutzer nichts
- diskrete Anwendungen
  - falls in T keine Operation anfällt, bleibt der Zustand gleich

Timewarp mit Operations-Filter (1)

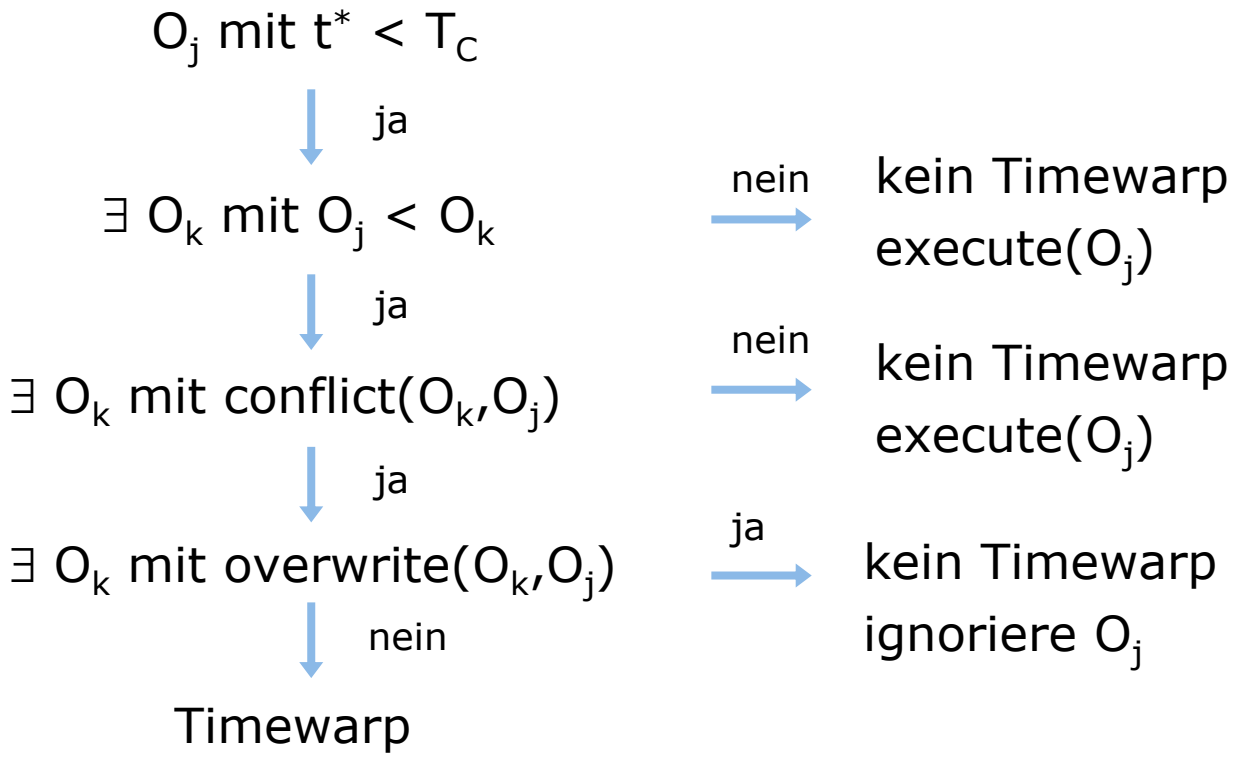
Idee: entscheide mit Hilfe von Anwendungslogik, ob eine verspätete Operation O_j einen Timewarp erfordert

1. ignoriere O_j, wenn der Effekt von O_j auf den aktuellen Zustand nicht sichtbar ist
2. O_j kann ohne Timewarp ausgeführt werden, wenn die Ordnung nicht verletzt wird oder keine semantischen Konflikte entstehen

Anwendungslogik

- conflict(O_i,O_j)
  - Wiederholung: O_i und O_j sind konfliktär, wenn sie dieselben Attribute verändern
  - z.B. O_i = "ändere Farbe", O_j = "ändere Farbe"
- overwrite(O_i,O_j)
  - Ausführen von O_j und O_i erzeugt denselben Zustand wie Ausführen von O_i allein
  - z.B. O_j = "ändere Farbe" und O_i = "lösche"

Timewarp mit Operations-Filter (2)

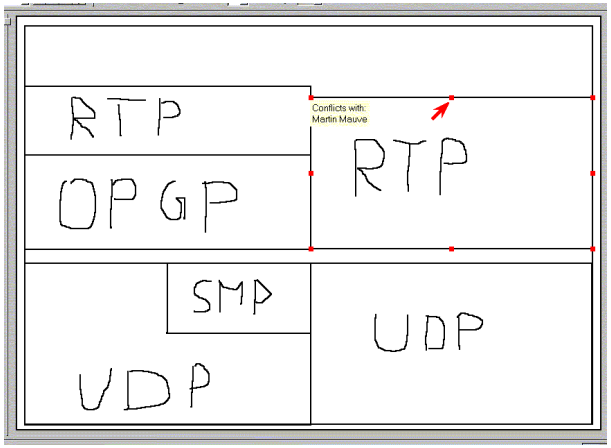


Timewarp mit Operations-Filter (3)

- diskrete Anwendungen
  - conflict und overwrite meist einfach zu definieren
  - gibt es viele verschiedene Operationen, ist eine paarweise Überprüfung aufwendig → statt dessen Kategorisierung
- kontinuierliche Anwendungen
  - falls O_j zu spät eintrifft und ausgeführt werden muss, ist immer ein Timewarp erforderlich
  - O_j zu ignorieren ist möglich, die Überprüfung aber meist schwierig (z.B. O_k löscht das betreffende Objekt)
    → Vorsicht: Seiteneffekte bei bewegten Objekten

Ergebnisse für ein Shared Whiteboard

- Setup für 2 Teilnehmer: Worst Case-Szenario mit Verspätung aller Operationen



#	verspätete Operationen	keine späteren Operationen → ausführen ohne TW	kein Konflikt → ausführen ohne TW	überschrieben → ignorieren	# Timewarps
1	687	310	323	47	7
2	594	246	314	31	2
3	515	193	295	25	2

Beschränkung der Operations-Historie (1)

Ziel: Speicherbedarf für Operations-Historie begrenzen

1) Ersetze Operations-Sequenzen

- ersetze lange Operations-Sequenzen durch semantisch gleichwertige Operation
- z.B. mit Events punktweise erzeugte Freihandlinie → Zustand
- Cues lösen keinen Timewarp aus und werden nicht gespeichert

Beschränkung der Operations-Historie (2)

2) Lösche sicher ausgeführte Operationen

- lösche Operationen, die von allen Instanzen ausgeführt wurden

➡ werden für mögliche Timewarps nicht benötigt

- implizite Bestätigung von Operationen mit Zustandsvektoren
  - Instanz i empfängt O_j von Instanz j mit Vektor SV_{O_j}
  - $SV_{O_j}[k]$ ist die SN der letzten Operation, die j von k empfangen hat

  → j hat alle älteren Operationen von k empfangen

  → falls O_j kausal ausführbar ist: temporäre Inkonsistenzen durch Operationen von j, die nebenläufig zu Operationen von k mit $SN_k \leq SV_{O_j}[k]$ sind, können nicht auftreten

  → aus der Sicht von j kann i alle Operationen von k mit $SN_k \leq SV_{O_j}[k]$ aus der Historie entfernen

  → hat i Bestätigungen von allen Instanzen: lösche Operationen

Beschränkung der Operations-Historie (3)

- explizite Bestätigung von Operationen: periodische Status-Nachrichten mit den aktuellen Zustandsvektoren
  → schnellere Bestätigung aber Kommunikations-Overhead

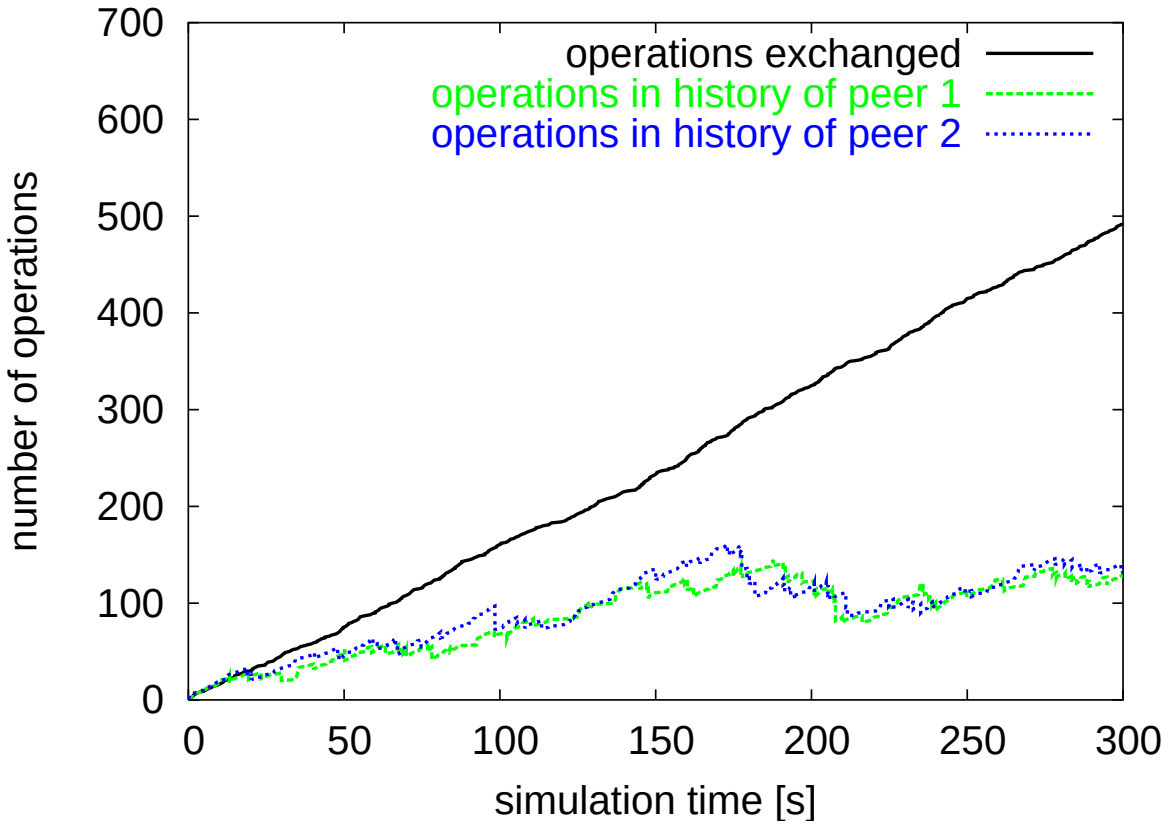
3) Verwende anderen Konsistenzerhaltungs-Mechanismus

- beschränke die von der Historie abgedeckte Zeitspanne T_H:
  lösche O_{j,t^o,t^*} mit $t^* < T_C - T_H$

➡ temporäre Inkonsistenzen durch empfangene Operationen O_{j,t^o,t^*_x} mit $t_x^* < T_C - T_H$ können nicht durch Timewarp behoben werden
  → alternativer Konsistenzerhaltungs-Mechanismus erforderlich
  → Trade-Off T_H: durch Timewarp abgedeckte Zeitspanne vs. Speicherbedarf (z.B. 180s beim mlb)

➡ die Historie beginnt nicht am Anfang einer Sitzung
  → Anwendung muss sicherstellen, dass immer ein Zustand $S_{i,t}$ mit $t = T_C - T_H$ in der Historie enthalten ist

Beispiel-Ergebnis für Lösch-Algorithmus



Zusammenfassung

Klassifikation

- optimistisches Verfahren für alle Anwendungen

Implementierung

- Operations-Historie
  - sortiert nach t^*
  - Einfügen lokaler oder empfangener Operationen
  - Einfügen von State Snapshots
  - Ersetzen von Operations-Sequenzen
  - Löschen alter Operationen
- Empfang verspäteter Operation
  - ist Timewarp erforderlich → Filter
  - Ausführung Timewarp: Startzustand $S_{i,t}$ und $H_{i,t}$
  - Update alter Zustände

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)

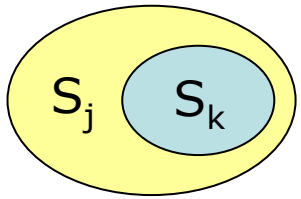
- Zustände sollten Meta-Informationen enthalten (z.B. Zustands-vektoren) → schnelle Überprüfung
- Korrektheit per iterativer Zustandsübertragung oder iterativer Zustandsanfrage
- Instanz i sollte bis zum erfolgreichen Abschluss der Anfrage lokale Benutzeraktionen verhindern (→ Vermeidung sekundärer Inkonsistenzen)

1. Iterative Zustandsübertragung

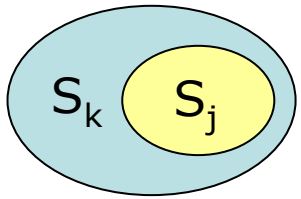
Jede Instanz k vergleicht einen empfangen Zustand S_j mit S_k

1. S_j und S_k enthalten dieselben Operationen
   → NOP

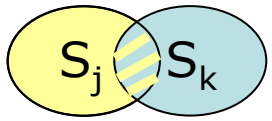
2. S_j enthält alle Operationen von S_k plus einige mehr
   → k hat einige Operationen verpasst und sollte S_j übernehmen



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