

Computergestützte Gruppenarbeit

7. Konsistenz

Dr. Jürgen Vogel

*European Media Laboratory (EML)
Heidelberg*

SS 2006

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

1) Konsistenzerhaltung ("Consistency Control")

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

2) Initialisierung von Anwendungsinstanzen ("Late-Join")

- Übergabe des aktuellen Zustands an Instanzen, 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

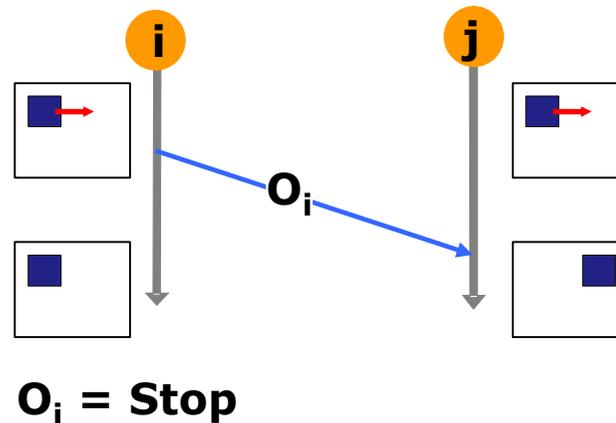
Zunächst betrachten wir ausschließlich (1)

Konsistenzzerhaltung (1)

Gefährdung der Konsistenz

1) Szenario: synchrone kontinuierliche Anwendung

- Operation O_i unterliegt einer gewissen Netzverzögerung, so dass $\text{Notification Time} > \text{Response Time} \geq 0$
- Ausführung von O_i bei j führt zu verschiedenen Zuständen
→ **Inkonsistenz**

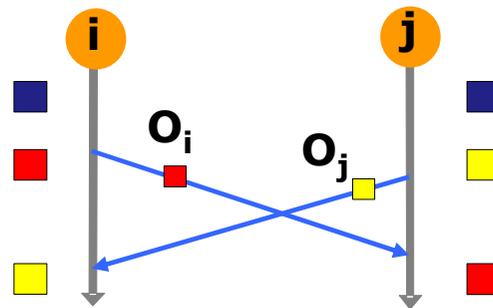


➔ Ausführungszeitpunkt einer Operation ist entscheidend

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

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)

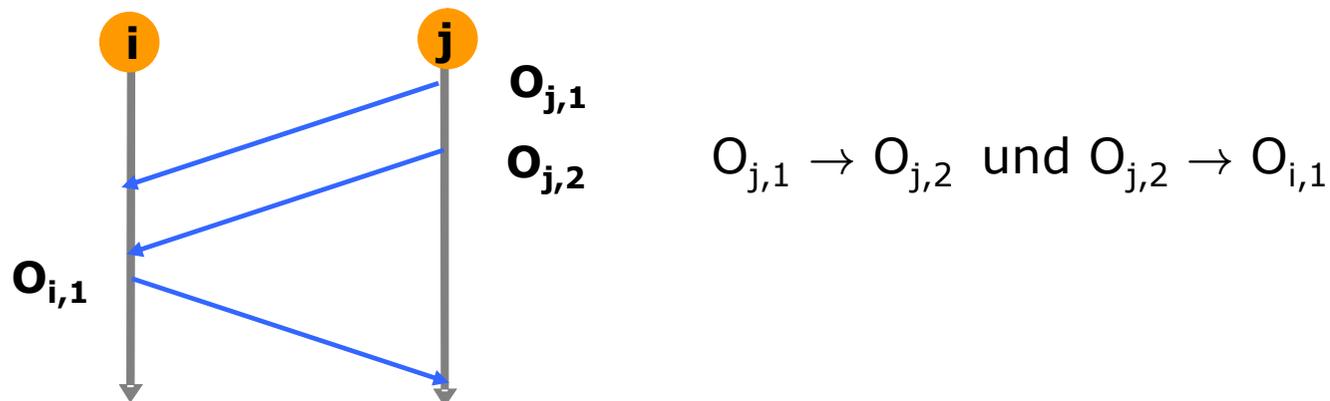
Definition

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 $O_{i,a} \rightarrow O_{j,b}$ bezeichnet man $O_{j,b}$ auch als **abhängig** von $O_{i,a}$

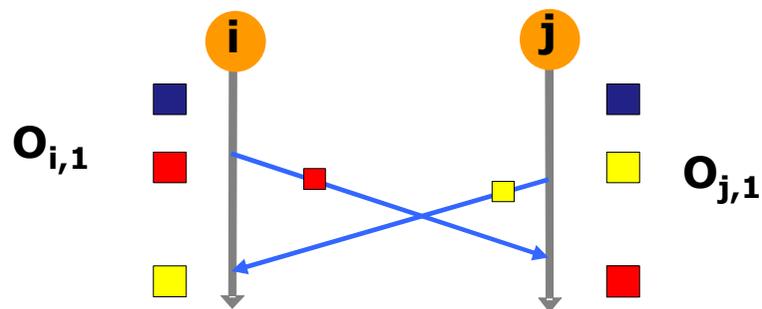
Beispiel



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 $O_{j,b}$ **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"): $O_{i,a} \quad O_{i,b}$

Beispiel



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

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 erfordert 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
- ➔ (3) Konvergenz erfordert nicht (!), dass alle Operationen zuverlässig übertragen werden müssen
- ➔ daher sind wir auch daran interessiert, ob der Zustand *korrekt* ist

Korrektheit

Sei P eine virtuelle "perfekte" Instanz, die alle Operationen $O_{i,a}$ einer Sitzung in der eindeutigen Reihenfolge ihrer Erzeugung ausführt.

➔ 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 i dieselben Operationen ausgeführt haben.

➔ Korrektheit gilt wie Konvergenz nur für Instanzen, die alle erforderlichen Operationen besitzen

➔ Korrektheit \Rightarrow Konvergenz

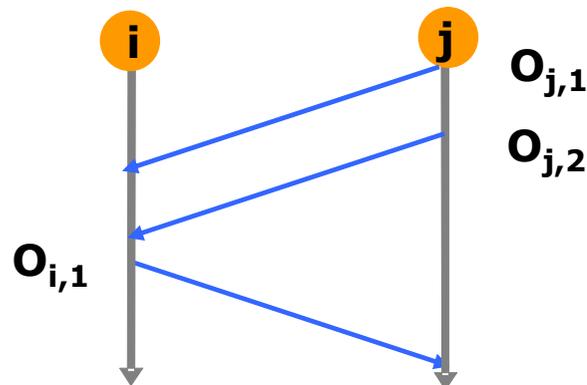
➔ Überprüfung von Kausalität, Konvergenz und Korrektheit?

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: $SV := \langle (1, SN_1), (2, SN_2), \dots, (n, SN_n) \rangle$
- 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 Zustandsvektor SV_{O_i} zugewiesen
- jedem Zustand S_i einer Instanz i ist ein Zustandsvektor SV_i zugewiesen, der die auf ihm ausgeführten Operationen repräsentiert

Beispiel



$$SV_i = SV_j = \langle (i, 0), (j, 0) \rangle$$

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

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

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

$$= SV_i = SV_j$$

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.

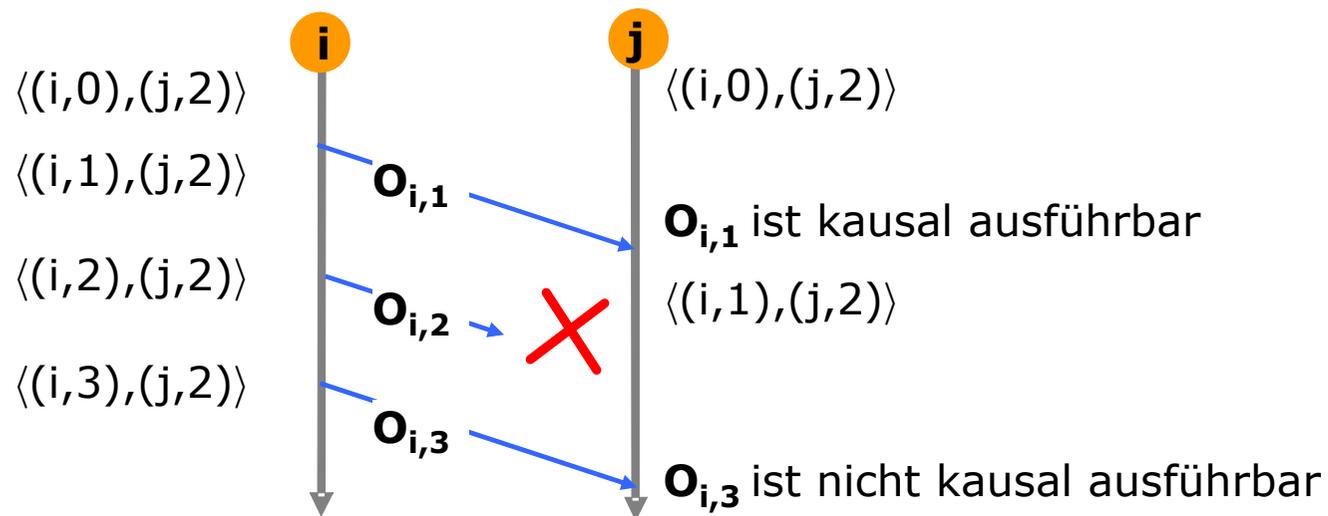
Dann kann O_i von j ausgeführt werden, wenn

(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 Konvergenz mit SV

Für die Überprüfung von Konvergenz 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 $\text{sum}(SV) := \sum_k SV[k]$.

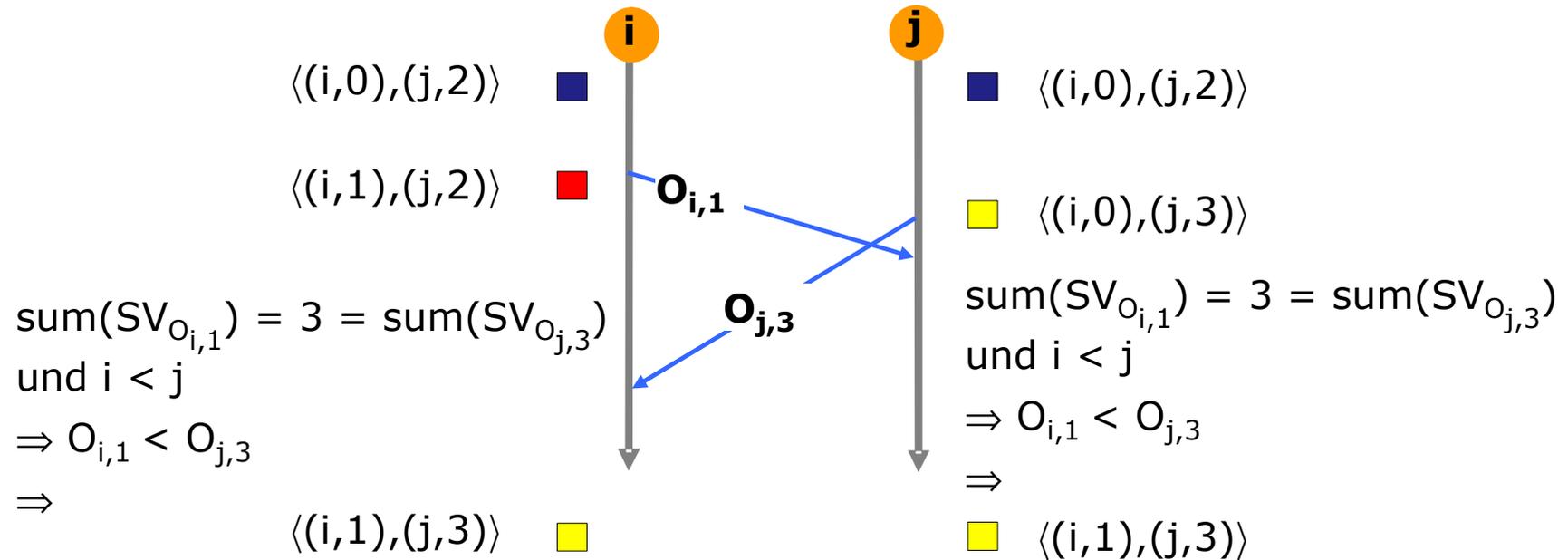
Dann ist $O_i < O_j$ wenn

- (1) $\text{sum}(SV_{O_i}) < \text{sum}(SV_{O_j})$ oder
- (2) $\text{sum}(SV_{O_i}) = \text{sum}(SV_{O_j})$ und $i < j$.

Konvergenz 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



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^*} : Operation der Instanz i , die zum Zeitpunkt t^* auszuführen ist
zur Vereinfachung nehmen wir an, dass die Auflösung der Uhr ausreicht, gleichzeitige Ausführungszeitpunkte zweier Operationen zu verhindern (sonst: Tie-Breaker)
- H : Operations-Historie = Menge aller Operationen in einer Sitzung

Definition

Seien O_{i,t_i^*} und O_{j,t_j^*} zwei Operationen, dann gilt
 $O_{i,t_i^*} < O_{j,t_j^*}$ wenn $t_i^* < t_j^*$

Konsistenz

Definition

Eine kontinuierliche Anwendung garantiert **Konsistenz** ("Consistency"), wenn zu jedem Zeitpunkt t bei allen Instanzen i und j , die alle Operationen $O_{k,t^*} \in H$ 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 $O_{i,t^*} \in H$ 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 $O_{k,t^*} \in H$ 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 bei kontinuierlicher Groupware

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 Folgefehler (temporäre Inkonsistenzen)
- ➔ u.U. sollte eine kontinuierliche Anwendung Kausalität daher (für bestimmte Operationen) ignorieren

Zusammenfassung

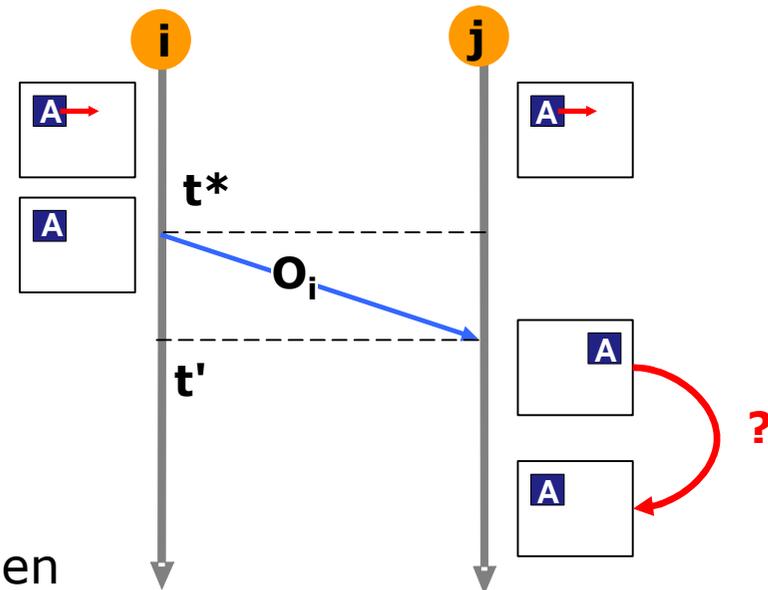
- diskrete Anwendungen: Kausalität, Konvergenz und Korrektheit
- kontinuierliche Anwendungen: Konsistenz und Korrektheit
- (syntaktische) Konsistenzkriterien sagen nichts über die Semantik von Operationen aus, d.h. ein Zustand kann aus der Sicht des Benutzers unlogisch sein, obwohl ihn das System für 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^*} eine Stopp-Operation für ein bewegtes Objekt A
- falls j O_{i,t^*} nach t^* empfängt, ist A auf einer falschen Position
- ➔ bis t' ist S_j korrekt
- ➔ bei t' muss j die Position von A berücksichtigen, um Korrektheit herzustellen
 - (1) abrupte Änderung
 - (2) interpolierte graduelle Änderung



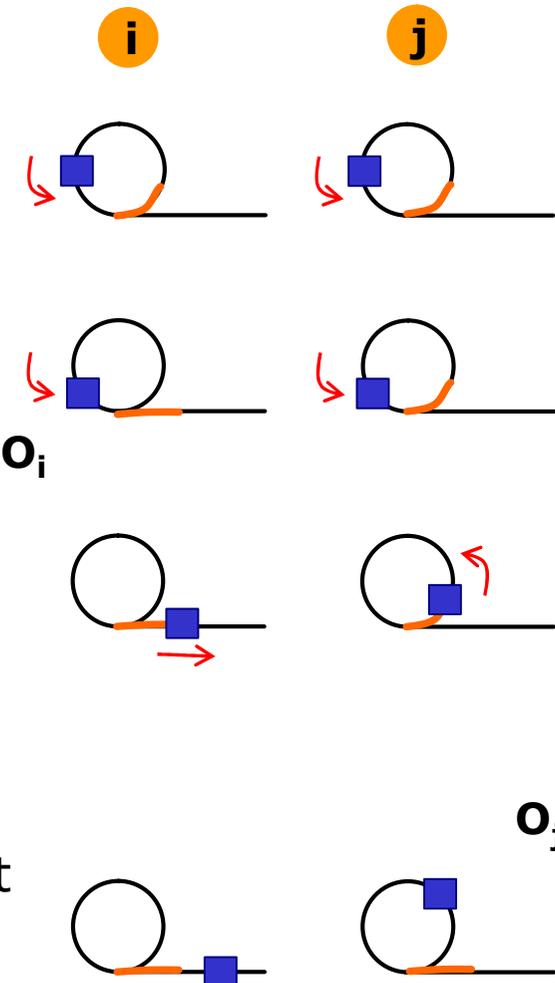
Sekundäre Inkonsistenzen (1)

- 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)

Sekundäre Inkonsistenzen (2)

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 nicht auf die Nebenstrecke fährt, zieht er die Bremse mit O_j
- ➔ 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 Anwendungszustand 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 aufeinanderfolgenden 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; Paketverluste 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 und 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
- 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 $SN_4 = 7$

→ 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 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, $\text{Quorum} = \sum \text{Gewicht positive Antworten} / \text{Gesamtgewicht}$
 - Write All Read Any (WARA): $\text{Quorum}_{\text{write}} = n$, $\text{Quorum}_{\text{read}} = 1$
- 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

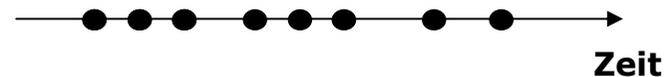
Serialisierung

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

- optimistisches Verfahren für diskrete Anwendungen

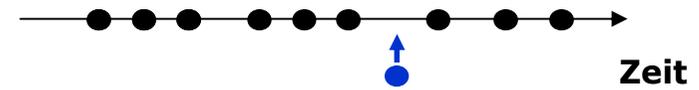
Voraussetzung

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

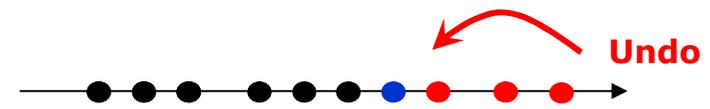


Serialisierungs-Algorithmus

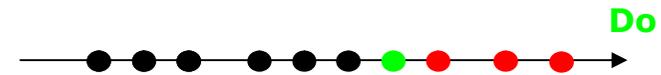
i empfängt Operation O_j in falscher Reihenfolge



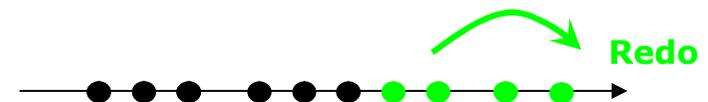
1. mache alle Operationen $O_k \in H_i$ mit $O_k > O_j$ rückgängig



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) Zustandsänderung

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^0 der Erzeugungszeitpunkt von O_{i,t^0,t^*} .

Definition

Die **Intention** einer Operation O_{i,t^0,t^*} ist der Effekt, der durch die Ausführung von O_{i,t^0,t^*} auf dem von Instanz i zum Zeitpunkt t^0 angezeigten Zustand erzielt wird.

Eine Anwendung gewährleistet **Intentionserhaltung** ("Intention Preservation"), wenn die Intention aller O_{i,t^0,t^*} bei allen Instanzen gewahrt bleibt und nebenläufige Operationen nicht konkurrieren.

Beispiel

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

Operations-Transformation (1)

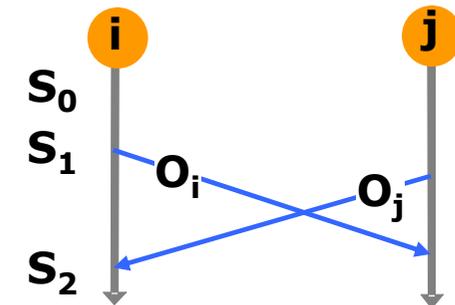
Idee: führe alle Operationen sofort aus (in beliebiger Reihenfolge), so dass die Intentionserhaltung gewährleistet wird

- 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 $S_0 = \text{"ABCDE"}$, $O_i = \text{"füge '12' bei Index 1 ein"}$, $O_j = \text{"lösche von Index 2 bis 3"}$.
Betrachte Instanz i :

- $O_i \rightarrow S_1 = \text{"A12BCDE"}$
- 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: $O_j' = \text{"lösche von Index 4 bis 5"}$ → $S_2 = \text{"A12BE"}$



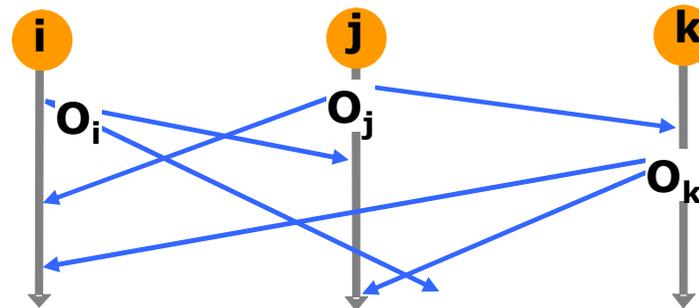
Operations-Transformation (2)

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:

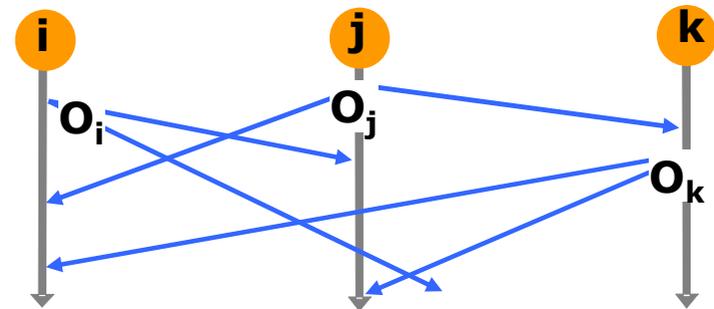


- $O_i \parallel O_k$ aber wegen $O_j \rightarrow O_k$ sind die Ausgangszustände von i und k ungleich
- ➔ IT berechnet falsche Operationen O_i' und O_k'

Operations-Transformation (3)

Beispiel

- $S = \text{"ABCDE"}$, $O_i = \text{"füge '12' bei Index 1 ein"}$, $O_j = \text{"füge '23' bei Index 0 ein"}$, und $O_k = \text{"füge '45' bei Index 2 ein"}$



- intentionserhaltender Zustand: "2345A12BCDE"
- k : $O_j(S) = \text{"23ABCDE"}$, $O_k(O_j(S)) = \text{"2345ABCDE"}$
- i : $O_i(S) = \text{"A12BCDE"}$, $O_j \parallel O_i$, $O_j' \circ O_i(S) = \text{"23A12BCDE"}$
 - Empfang von $O_k \parallel O_i \rightarrow O_k' = \text{"füge '45' bei Index 4 ein"}$
 - $O_k' \circ O_j' \circ O_i(S) = \text{"23A1452BCDE"}$
- ➔ 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 $O_k' \mapsto O_k''$: $O_k'' \circ O_j' \circ O_i(S) = \text{"2345A12BCDE"}$
- ➔ hier: ET O_k gegen O_j : $O_k' = \text{"füge '45' bei Index 0 ein"}$ und IT O_k' gegen O_i : $O_k'' = \text{"füge '45' bei Index 2 ein"}$

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

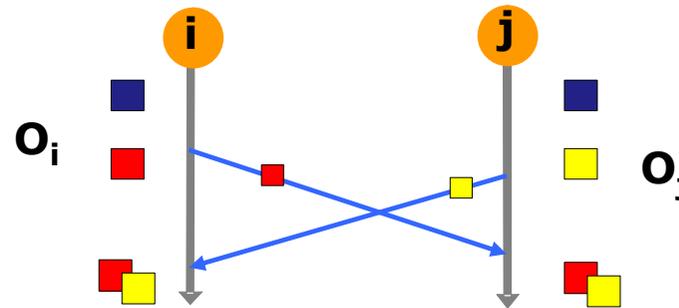
- + Konvergenz und Korrektheit
- + Intentionserhaltung
- + Kausalität durch vorgeschaltete Zustandsvektor-Analyse
- + autonome Ausführung (→ ausschließlich lokales Wissen)
- + sofortige Ausführung lokaler Operationen (→ keine Verlängerung der Response Time)
- komplexe Transformationsfunktionen IT und 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 $O_1 \prec O_2$
- 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: $\forall O_i, O_j \in CG \neg (O_i \quad O_j)$

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

$$CGS = \{CG_1, CG_2, \dots, CG_n\} \text{ mit}$$

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

$$(2) \forall O_i, O_j \in GO: \text{wenn } \neg (O_i \quad O_j) \exists CG_i \in CGS \text{ mit } O_i, O_j \in CG_i$$

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 $O_i \sim O_j$.

Beispiel 2: $GO = \{O_1, O_2, O_3, O_4\}$ mit $O_1 \sim O_2 \rightarrow \{O_1, O_3, O_4\}$ 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, \dots, O_n
- MOVIC erzeugt eine Folge $MCGS_1, MCGS_2, \dots, MCGS_n$
- $MCGS_i$ 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** $\forall O_j \in CG_x \neg(O_i \rightarrow O_j)$ **THEN** $CG_x += \{O_i\}$
 - iii. **ELSEIF** $\forall O_j \in CG_x O_i \rightarrow O_j$ **THEN** $C--$
 - iv. **ELSE**
 - $CG_n = \{O \mid (O \in CG_x) \wedge \neg(O \rightarrow O_i)\}$
 - $CG_y = CG_n + \{O_i\}$
 - $MCGS_i += \{CG_y\}$
 - $MCSG_i += \{CG_x\}$
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 $O_1 \succ O_2, O_1 \succ O_3$ und $O_2 \succ O_3$

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^0_i,t^*_i} empfangen, wenn sie einen State empfangen hat, der den Effekt von O_{i,t^0_i,t^*_i} enthält

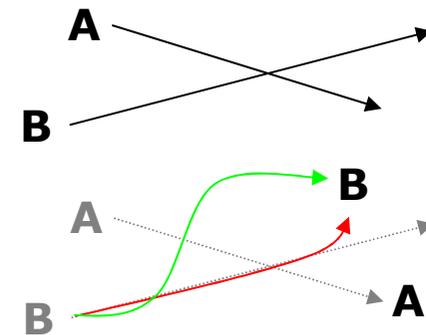
Konsistenz

- 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

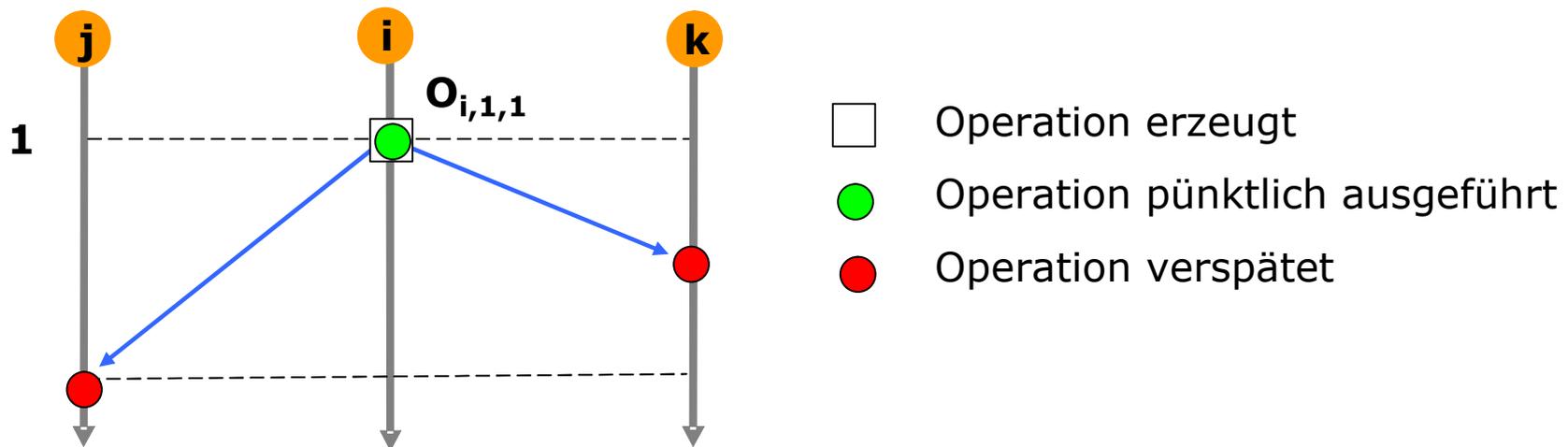
Local Lag – Motivation (1)

Beobachtung

Inkonsistenzen werden häufig durch die Netzwerkverzögerung verursacht: Empfang von Operationen O_{i,t^0,t^*_i}

- in unterschiedlicher Reihenfolge
- nach der geplanten Ausführungszeit

Beispiel für kontinuierliche Anwendung mit $t^0 = t^* = 1$



→ 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

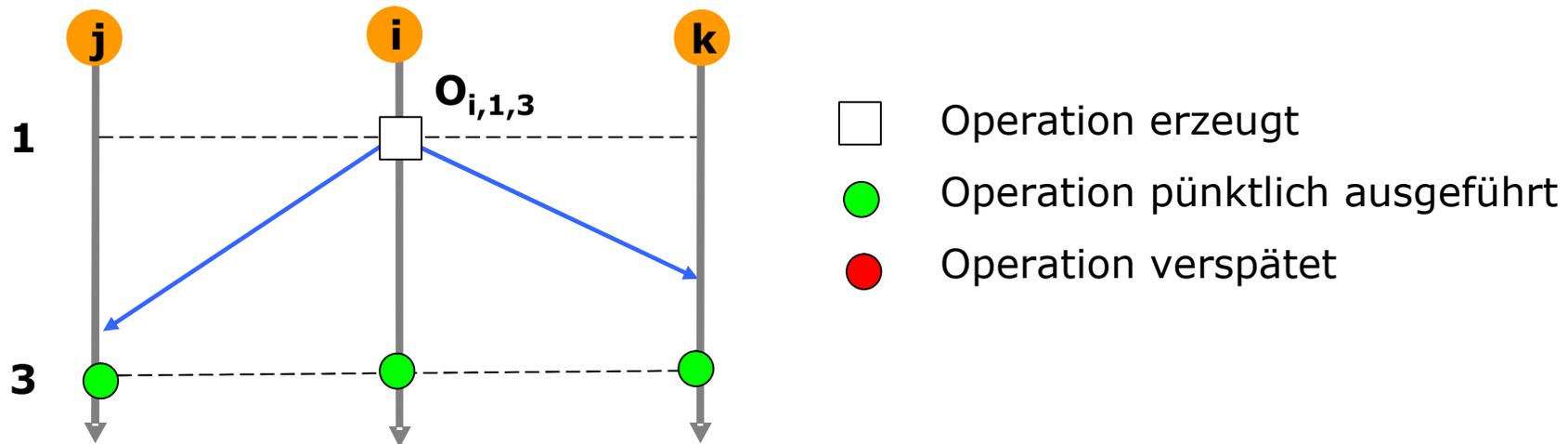
Idee: Verfahren zur Vermeidung von temporären Inkonsistenzen

Local Lag (1)

Idee: verhindere temporäre Inkonsistenzen durch $t^* > t^0$

- nutze die Zeitspanne $t^* - t^0$, um O_{i,t^0,t^*} 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 $t^0 = 1$ und $t^* = 3$



Local Lag (2)

- zu früh empfangene Operationen werden bis zum Erreichen von t^* gepuffert
- die Zeitspanne $t^* - t^0$ 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^0,t^*} :

$$I_j(O_{i,t^0,t^*}) = d(i,j) - (T_i^* - T_j^*) - (t^* - t^0)$$

- $d(i,j)$ = 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
- $t^* - t^0$ = Zeitspanne zwischen Erzeugung und Ausführung von O

Falls $I_j \leq 0 \rightarrow$ keine temporäre Inkonsistenz

Einfluss der Uhren-Abweichung

$$I_j(O_{i,t^0,t^*_i}) = d(i,j) - (T^*_i - T^*_j) - (t^* - t^0)$$

- falls $T^*_i \gg T^*_j$, würde $d(i,j)$ kompensiert, ohne dass Local Lag erforderlich ist (d.h. $t^* = t^0$)
- funktioniert nur in die Richtung $i \rightarrow j$
- in der Richtung $j \rightarrow i$ 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 $t^* - t^0$ 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: $t^* - t^0 \geq d(i,j) - (T_i^* - T_j^*)$ 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 $l_{\min} < t^* - t^0 < r_{\max}$
- wenn $l_{\min} > r_{\max} \rightarrow$ echter Trade-Off, evtl. Auflösung durch Evaluation

Implementierung

- Warteschlange für alle Operationen O_{i,t^0_i,t^*_i} , sortiert nach t^*
- führe O_{i,t^0_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

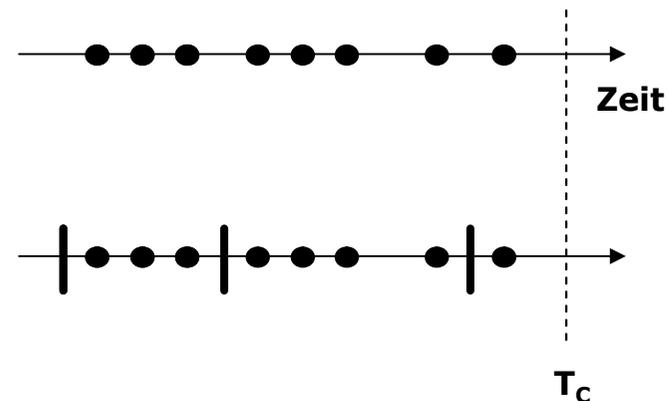
Timewarp

Ziel: Serialisierung von Operationen

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

Voraussetzung

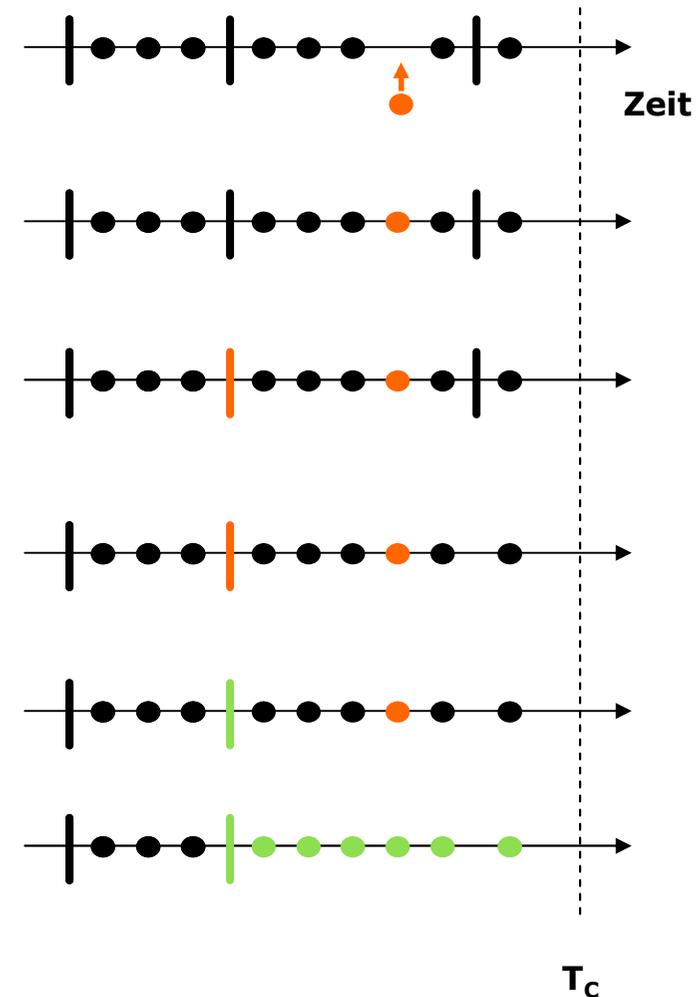
- jede Instanz i speichert eine lokale Operations-Historie H_i , die nach einer bestimmten Ordnung sortiert ist (z.B. Zustandsvektoren oder Ausführungszeit)
- H_i enthält alle lokalen und alle empfangenen Operationen
- zusätzlich speichert i periodisch den aktuellen Zustand S_{i,T_C} ("State Snapshot")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



Timewarp-Algorithmus

i empfängt eine Operation O_{j,t^0,t^*_x} mit $t^*_x < T_C$

1. füge O_{j,t^0,t^*_x} an der richtigen Stelle in H_i ein
2. bestimme den ersten Zustand $S_{i,t} \in H_i$ mit $S_{i,t} < O_{j,t^0,t^*_x}$
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 $H_{i,t} = \{O_{j,t^0,t^*_x} > S_{i,t} \text{ mit } t^* \leq T_C\}$ (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) Zustandsänderung

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 (\sim Server)
 - 2) dynamische Auswahl, z.B. per *Exponential Feedback Raise* (EFR)
 - sende Anfrage an alle Instanzen
 - jeder Kandidat i ($i = 1, \dots, N$) zieht eine Zufallszahl $x \in [0, 1]$
 - wenn $x < 1/N \rightarrow$ 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 \rightarrow lösche Feedback Timer
 - Kandidaten = alle Instanzen ohne (bewusste) Inkonsistenz
- ➔ Idealfall: eine Antwort
- ➔ Timer bedeutet zusätzliche Wartezeit

Konsistenz des Antwort-Zustands

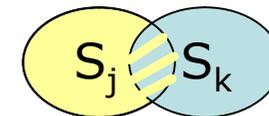
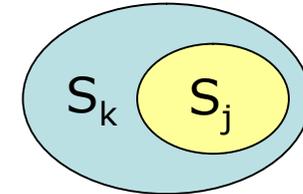
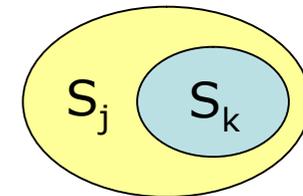
Problem: Kandidat j kann die Korrektheit / Vollständigkeit des eigenen Zustands S_j nicht garantieren (z.B. wenn verspätete Operation unterwegs)

- Zustände sollten Meta-Informationen enthalten (z.B. Zustandsvektoren) → 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