Student Workbook

Telekooperation 5

Inhalt

Teil 1: Grundlagen
Teil 2: Klassifizierung
Teil 3: Bedarfsanalyse
Teil 4: Bausteine eines CSCW-Systems
Teil 5: Vorgehensmodell zur Umsetzung
Teil 6: CSCW Architekturen
Teil 7: Verteilte Softwaresysteme
Teil 8: Prozesse und Synchronisation
Teil 9: Transaktionen und Nebenläufigkeit



FACHHOCHSCHULE HAGENBERG

Unterlagen zur Lehrveranstaltung

Stefan Probst
© SSW Consulting
Jormannsdorf 99 • A-7431 Bad-Tatzmannsdorf
Telefon +43 699 / 81 50 90 66
E-Mail: stefan.probst@fh-hagenberg.at

Einleitung

Das vorliegende Dokument dient als Arbeitsunterlage für Teilnehmer der Lehrveranstaltung „Telekooperation". Das Werk einschließlich aller Teile ist urheberrechtlich geschützt. Die komplette oder nur teilweise Verbreitung und Verwendung dieses Dokuments, in welcher Art auch immer, bedarf der ausdrücklichen Zustimmung des Urhebers.

Grundlegende Literatur

Als Grundlage zur Erstellung dieses Werkes wurden ausschließlich folgende Literaturquellen herangezogen und gegebenenfalls zitiert:

- G. Schwabe, N. Streitz, R. Unland (2001) *CSCW-Kompendium*. ISBN 3-540-67552-3, Springer, 2001.
- Wikipedia.de – Die freie Enyklopädie im Netz (www.wikipedia.de)
- M. Schmees, *Einführung in CSCW und CSCL*, Universität Oldenburg, 2001.
- A. Tanenbaum, M. van Stehen (2003) *Verteilte Systeme – Grundlagen und Paradigmen*. ISBN 3-827-37057-4, Pearson Studium, 2003.

Verwendung dieser Arbeitsunterlage

SYMBOL-LEGENDE	
ⓘ	Informationen
!	Wichtige Hinweise
🖋	Beispiele
?	Verständnisfragen

In dieser Arbeitsunterlage werden Sie immer wieder auf verschiedene Symbole und Schreibweisen stoßen.

Spezielle Ausdrücke und Begriffe werden durch einfache Anführungszeichen (‚') gekennzeichnet. Kommandos werden innerhalb doppelter Anführungszeichen („") erwähnt. Variable Teile innerhalb eines Kommandos sind *kursiv* gedruckt, genauso wie betonte Wörter. Am Seitenrand werden Sie immer wieder Symbole finden, die auf spezielle Informationen hindeuten.

Inhaltsverzeichnis

KAPITEL 1

Grundlagen von CSCW Systemen 1-1
Telekooperation 1-2
CSCW 1-2
CSCL 1-3
Eckdaten von CSCW 1-3
Vorteile von CSCW 1-4
Herausforderungen von CSCW 1-5
CSCW als Überbegriff 1-6
Workflow Management Systeme 1-6
Groupware Systeme 1-7
Grundlegende Begriffe 1-8
Kommunikation 1-10
Äußerer Kontext 1-11
Phasen der Kommunikation 1-13
Computervermittelte Kommunikation 1-13
Kooperation und Koordination 1-14
CSCW-Themengebiete 1-16
Verständnis für Gruppenarbeit 1-17
Gruppentheorie 1-19
Verhältnis Individuum / Gruppe 1-20
Interaktionsstruktur 1-23
Der Meinungsbildungsprozess 1-24
Verhältnis zwischen Gruppen 1-25
Arbeitsmethoden und –techniken 1-25
Entwicklung von Werkzeugen 1-26
Beurteilung und Adaptierung 1-26
Der Einführungsprozess 1-27

KAPITEL 2

Klassifikation von CSCW Systemen 2-1
Klassifikationskriterien 2-2
Raum und Zeit 2-3
Art und Grad der Unterstützung 2-3
Kommunikationsunterstützung 2-4
Koordinationsunterstützung 2-4
Kooperationsunterstützung 2-5
Systemklassen 2-5
Aufbau con CSCW Systemen 2-6
Zentrale Architektur 2-6
Replizierte Architektur 2-7
Hybride Architektur 2-7
Klassifizierung CSCL 2-8

KAPITEL 3

Bedarfsanalyse 3-1
Durchführung 3-2
Aufgabenanalyse 3-4
Arbeitsprozessanalyse 3-5
Interaktionsanalyse 3-6
Soziale Kooperationsstruktur 3-6
Arbeitsmittelanalyse 3-7
Aneignungsanalyse 3-8
Arbeitsraumanalyse 3-9
Informationsspeicher 3-10
Durchführung der Analyse 3-11

KAPITEL 4

Bausteine von CSCW-Systemen 4-1
Design 4-2
Grundprinzipien 4-2
Kommunikationsorientierte synchron 4-3
Textbasierte Konferenzsysteme 4-4
Audiovermittelte Kommunikation 4-5
Videovermittelte Kommunikation 4-6
Anwendung 4-7
Kommunikationsorientierte asynchron 4-7
E-Mail Systeme 4-8
Computerkonferenzsysteme 4-9
Newsgroups 4-10
Koordinationswerkzeuge 4-11
Terminkoordination 4-11
Zentrale Kalendersysteme 4-12
Dezentrale Kalendersysteme 4-13
Vorgangssteuerung 4-13
Projektmanagementwerkzeuge 4-14
Materialkoordination 4-14
Computerunterstützte Sitzungen 4-15
Kollaborationsorientierte synchrone 4-16
Kooperationsunterstützung 4-17
Koordinationsunterstützung 4-18
Kommunikationsunterstützung 4-18
Kollaborationsorientierte asynchrone 4-19
Gemeinsamer Arbeitsbereich 4-20

KAPITEL 5
Vorgehensmodell zur Umsetzung 5-1
CSCW-Design 5-2
Problematik beim Design 5-3
Design 5-4
Prozessmodell 5-4
Phasen des Prozessmodells 5-5
Iterationen im Prozessmodell 5-5
Teamstruktur 5-6
Anwender 5-7
Umsetzung 5-8
Schulung 5-9
KAPITEL 6
CSCW Architekturen 6-1
Architekturformen 6-2
Groupwareplattformen 6-3
Synchrone Softwarearchitekturen 6-4
Herausforderungen 6-4
Referenzmodelle 6-5
Zentrale Architektur 6-6
Replizierte Architektur 6-6
Hybride Architektur 6-7
Zusammenfassung 6-7
KAPITEL 7
Verteilte Softwaresysteme 7-1
Definition Verteiltes System 7-2
Eigenschaften 7-2
Software Architekturen 7-3
Middleware 7-5
Ziele von Middleware 7-5
Transparenz 7-6
Offenheit 7-8
Schnittstellendefinition 7-9
Beispiel ISO/OSI Referenzmodell 7-9
Skalierbarkeit 7-10
Skalierbarkeit - Dimensionen 7-11
Skalierungstechniken 7-13
Betrachtungsebenen von VSS 7-16
Hardware Konzepte 7-16
Software Konzepte 7-17
Distributed Operating Systems 7-18
Netzwerkbetriebssysteme 7-20
Middleware 7-21
KAPITEL 8
Prozesse und Synchronisation 8-1
Aufgaben 8-2
Prozessarten 8-2
Software Agenten 8-3
Synchronisierung 8-4
Uhr-Synchronisierung 8-4
Zeitstempel 8-7
Globaler Status 8-7
Koordination zwischen Prozessen 8-8
Wahl-Algorithmen 8-8
Wechselseitiger Ausschluss 8-9
Zentralisierter Algorithmus 8-9
Verteilter Algorithmus 8-10
Token-Ring Algorithmus 8-11
Vergleich der Algorithmen 8-12
KAPITEL 9
Transaktionen und Nebenläufigkeit 9-1
Transaktionen 9-2
Elementare Funktionen 9-3
Eigenschaften (ACID) 9-3
Probleme im Mehrbenutzerbetrieb 9-4
Klassifizierung von Transaktionen 9-8
Nebenläufigkeitssteuerung 9-10
Serliaisierbarkeit 9-10
Nebenläufigkeitskontrolle 9-12
Sperren 9-13
Zeitmarkenverfahren 9-15

Kapitel

8

Prozesse und Synchronisation

Bei verteilten Systemen (z.B. CSCW-System) arbeiten meist mehrere Prozesse gleichzeitig mit gemeinsam genutzten Daten. Damit keine Konflikte entstehen und die Konsistenz der Daten gewährleistet werden kann, müssen die Prozesse und deren Tätigkeit synchronisiert werden.

Prozesse in verteilten Systemen greifen häufig gleichzeitig auf gemeinsam genutzte Daten zu. Tatsächlich kann jedoch immer nur ein Prozess Daten ändern bzw. eine Ressource nutzen. Es ist insofern nötig, die verschiedenen Prozesse entsprechend zu koordinieren um die Benutzbarkeit und Konsistenz der Daten und Ressourcen gewährleisten zu können. Dieser Koordinationsprozess wird auch Synchronisation genannt. Die Synchronisation selbst muss allerdings wiederum koordiniert werden. Dabei müssen gewisse Eigenheiten von verteilten Systemen berücksichtigt werden. Dieses Kapitel erläutert diese Eigenheiten und die damit verbundenen Probleme und zeigt Algorithmen zur Synchronisation von Prozessen.

Lehrinhalte und -ziele

Am Ende des Kapitels sollten Sie die Probleme, die bei der Synchronisation von Prozessen auftreten kennen und erläutern können.

Darüber hinaus sollten Sie Algorithmen und Vermeidungstechniken für die Synchronisations-probleme und deren Problematiken im Einsatz kennen.

Aufgaben

Motivation / Inhalt

- Verschiedene Prozessarten
- Prozesse kommunizieren miteinander
  - Stellen Daten und Ressourcen zur Verfügung
    - Gegenseitiger Ausschluss (Konflikte vermeiden)
  - Abgleich der Prozesse → Synchronisierung
- Probleme der Synchronisierung
  - Keine globale Uhrzeit
  - Wer koordiniert Synchronisierung?
  - Wie werden Zugriffskonflikte vermieden?
  - Transaktionen
  - Konsistenz

Verteilte Systeme werden auf mehrere Prozesse aufgeteilt. Dabei kommen unterschiedliche Prozesstypen zum Einsatz, die untereinander Kommunizieren. Bei dieser Kommunikation werden Daten und Ressourcen zur Verfügung gestellt, die dann innerhalb des Systems gemeinsam genutzt werden können. Damit diese Kooperation möglich ist, müssen etwaige Konflikte vermieden werden. Eine solche Vermeidung von Konflikten ist nur durch den Abgleich der Prozesse möglich. Dieser Koordinationsprozess wird auch *Synchronisierung* genannt. Die Synchronisierung ist in verteilten Systemen viel schwieriger als bei abgeschlossenen Einzelsystemen. Zum Beispiel steht in verteilten Systemen keine globale Uhrzeit zur Verfügung. Außerdem muss bei gleichberechtigten Teilnehmern eine Einigung erzielt werden, wer die Synchronisierung koordiniert. Zugriffskonflikte müssen vermieden werden. Transaktionen erstrecken sich über mehrere Systeme und können zu unterschiedlichen Zeitpunkten fehlschlagen.

Prozessarten

Prozessarten

- Clients implementieren üblicherweise Benutzerschnittstellen
  - Verbirgt Kommunikation mit Server
  - Manchmal bereits sinnvolle Teilaufgaben am Client gelöst
- Server
  - Iterative Entwicklung (starr)
  - Nebenläufig
    - Ein / mehrere Dienst(e)
    - Zustandslos / Zustandsbehaftet

In verteilten Systemen kommen vorwiegend unterschiedliche Prozessarten zum Einsatz.

Die Aufgabe von Clients ist es, die Verbindung zwischen einem menschlichen Benutzer und einem entfernten Server herzustellen. Zudem können häufig sinnvolle Teilaufgaben bereits am Client gelöst werden um z.B. den Server zu entlasten.

Die Implementierung von Servern kann unterschiedlich erfolgen. Iterative Server verarbeiten Client-Anforderungen selbst und geben eine Antwort an den anfordernden Client zurück. Nebenläufige Server verarbeiten die Anforderung nicht selbst sondern delegieren diese an einen separaten Thread oder einen anderen Prozess. Die Art und Weise der Implementierung hat hohe Auswirkungen auf die Skalierbarkeit und das Antwortzeitverhalten der Server.

Lab 8.1



Manchmal ist es sinnvoll, dass der Client bereits Aufgaben löst. Dazu muss der Client die Applikation kennen um mit dem Server kommunizieren zu können. Allerdings können sich Serverversionen häufig ändern und es insofern nötig sein, dass sich die Kommunikationsschnittstelle ändert.

Überlegen Sie, welche Möglichkeiten es gibt, dafür zu sorgen, dass Clients sich dynamisch an diesen Lebenszyklus von Servern anpassen können.

Software Agenten

Agenten

- Prozess, der als autonome Einheit arbeitet, aber in der Lage ist mit anderen Agenten zusammenzuarbeiten
  - Interaktion mit Hilfe eines Kommunikationsprotokolls auf Applikationsebene (Agent Communication Language)
    - Unterscheidung Zweck und Inhalt der Nachricht
  - Agenten haben Fähigkeit, selbst zu handeln
    - Ergreifen Initiative, wenn erforderlich
    - Kann auf Änderungen seiner Umgebung reagieren
    - Kann mit anderen (verteilten) Agenten kommunizieren

Eine weitere Prozessart stellen Software-Agenten dar. Dabei handelt es sich um autonome Einheiten, die in der Lage sind, eine Aufgabe in Zusammenarbeit mit anderen, möglicherweise entfernten Agenten auszuführen. Damit diese Zusammenarbeit möglich ist, einigen sich die Agenten auf ein Kommunikationsprotokoll auf der Applikationsebene, die so genannte *Agent Communication Language*. Diese Sprache trennt streng zwischen dem Zweck und dem Inhalt einer Nachricht. Eine Nachricht kann nur eine begrenzte Anzahl von Zwecken haben. Beispielsweise kann die Aufgabe einer Nachricht sein, den Empfänger aufzufordern, einen bestimmten Dienst bereitzustellen. Analog dazu kann eine Nachricht die Aufgabe haben, auf eine zuvor gesendete Anforderungsnachricht zu reagieren. Nachrichten können auch zur Koordinierung bzw. zum Informationsaustausch zwischen Agenten, Softwaresystemen oder Kommunikationspartnern heran gezogen werden.

Synchronisierung

Synchronisierung

- Synchronisierung sorgt dafür, dass zur richtigen Zeit das Richtige passiert
  - Zeit: Keine globale Uhr
  - Globaler Status: Kein gemeinsamer Speicher Schwer festzustellen, welchen aktuellen Status ein System hat
  - Koordination zwischen Prozessen
- Gegenseitiger Ausschluss (nicht gleichzeitig dieselbe Ressource benutzbar)
- Transaktionen

Die Synchronisierung der Prozesse hat die Aufgabe, sicherzustellen dass zur richtigen Zeit das richtige passiert. Gerade der Zeitaspekt birgt jedoch bei verteilten Systemen enorme Probleme in sich, da keine globale Uhr vorausgesetzt werden kann. Zudem gibt es keinen globalen Status, da kein gemeinsamer sondern ein verteilter Speicher zur Verfügung steht. Dies macht die Koordination zwischen Prozessen, z.B. die Durchführung von Transaktionen oder die Vermeidung von gegenseitigen Ausschlüssen, sehr schwierig.

Uhr-Synchronisierung

Uhr-Synchronisierung

- Es gibt keine globale Zeit
  - Manchmal muss Einigkeit über Zeit herrschen (→ siehe Zeitstempel bei Transaktionen)
- Beispiel: System sammelt aktuelle Daten und stellt diese im Web bereit
  - Was ist aktuell?
  - Was passiert wenn Uhr in anderem System *nach* oder *vor* geht?
- Synchronisierung der Uhrzeit ist ein nicht triviales Problem!

Eines der Hauptprobleme bei der Synchronisierung liegt darin, dass es keine globale Zeit gibt. Viele Mechanismen erwarten aber Zeitpunkte (z.B. Zeitstempel bei Transaktionen, Timeouts). Wenn Systeme unterschiedliche Auffassungen bezüglich der Zeitpunkte haben, kann es zu großen Problemen kommen die mitunter Inkonsistenzen oder Fehlverhalten hervorrufen können. Auf der anderen Seite ist die Erzielung einer Stimmigkeit bezüglich der Zeit ein nicht triviales Problem.

Uhr-Synchronisierung

- Physische Uhren
  - Quarz-Kristalle schwingen und setzen in Intervallen Interrupts ab
  - Interrupts bewirken das Erhöhen eines Zählers → Uhrzeit wird erhöht
  - Problematik: Quarze schwingen innerhalb einer Toleranzgrenze (minimale Abweichung)
    - Über lange Zeit werden sich Uhrzeiten unterscheiden

Uhr-Synchronisierung

- Unterschiedliche Zeitmessungen
  - Sonnentage: Zeit zwischen zwei Zenitdurchgängen der Sonne
    - → Umlaufzeit der Erdrotation ist nicht konstant (Gezeitenreibung, atmosphärischer Widerstand)
    - Durchschnitt wird berechnet
  - Atomuhr: Anzahl der Übergänge des Cäsium 133-Atoms
    - Sehr konstant
    - Problem: Ist um 3 msek kleiner als Sonnentag → Mittag immer früher
  - Universal Coordinated Time (UTC)
    - Schaltsekunden wenn Unterschied Atom / Sonnentag > 800 ms

Eine Problematik der Uhrsynchronisierung liegt in den unterschiedlichen Zeitmessverfahren. Das Problem liegt in der Ungenauigkeit der verwendeten Zeitmessverfahren, da diese entweder nicht genau 100% korrekt sind (z.B. 3 msek Unterschied bei Atomuhren) oder auf einem berechneten Durchschnittswert basieren (Sonnentage). Bei der universell koordinierten Zeit werden zusätzlich Schaltsekunden in Abhängigkeit von äußeren Faktoren (Unterschied Atom / Sonnentag ist größer 800 ms) eingeführt.

Uhr-Synchronisierung

- Algorithmen:
  - Jede Maschine hat Timer, der h-mal einen Interrupt veranlasst.
  - Wenn Timer abgelaufen: Software-Uhr + 1
  - Synchronisationszeitpunkt: δ / 2ρ
    - δ: Zugelassene Abweichung der Uhren
    - ρ: Maximale Abweichung des Timers vom Optimalwert

Algorithmen zur Uhrzeitsynchronisation nehmen an, dass jede beteiligte Maschine über einen Timer verfügt, der Sekunde eine bestimmte Anzahl von Unterbrechungen (*Interrupts*) veranlasst. Nachdem der Timer abgelaufen ist, addiert die Unterbrechungs-anforderungsroutine den Wert 1 zu einer Software-Uhr, die die Anzahl der Ticks (Interrupts) verwaltet, die seit einem festgelegten Zeitpunkt in der Vergangenheit vergangen sind. Da die Anzahl nicht genau pro Sekunde festgelegt werden kann, verfügen alle Algorithmen über einen zulässigen Abweichungswert der Uhren. Das bedeutet, dass der Algorithmus für die Festlegung der Synchronisationszeitpunkte davon ausgehen muss, dass die Abweichung der Interrupts genau gegenläufig erfolgt.

Uhr-Synchronisierung

- Konkrete Algorithmen unterscheiden sich darin, wie Synchronisierung erfolgt
  - Zeit-Server: Teilnehmer passen ihre Uhrzeit an die des Zeitservers an
  - Berkeley-Algorithmus: Zeit-Server ermittelt Zeit der anderen Maschinen
    - Berechnet Durchschnittswert
    - Weist anderen Maschinen Durchschnittswert zu

Zeitstempel

Zeitstempel

- Wird Uhrzeit immer benötigt?
- Bei nebenläufigen Ereignissen kann nichts ausgesagt werden, wann Ereignisse aufgetreten sind oder welches Ereignis zuerst aufgetreten ist
- Für viele Anwendungen reicht eine Einigung über die Reihenfolge von Ereignissen → Einführung von Zeitstempel
  - Wenn a vor b, dann Zeitstempel(a) < Zeitstempel(b)
  - Es ist egal, ob Zeitstempel gleich der Uhrzeit ist
  - Es ist egal, ob Zeitstempel wirklich eine Sekunde lang dauert
  - Zeitstempel können nur wachsen, nie vermindert werden! → Synchronisation

Für viele Zwecke ist es ausreichend, dass sich alle Maschinen über dieselbe Zeit einig sind. Es ist jedoch nicht erforderlich, dass diese Zeit auch mit der realen Zeit übereinstimmt. Lediglich die temporale Reihenfolge von Aktionen muss von allen Maschinen richtig interpretiert werden können. Zeitstempel sind fiktive Zeiteinheiten, die mit realen Uhrzeiten nichts zu tun haben müssen. Wichtig ist, dass Zeitstempel nur wachsen können, niemals vermindert werden können. Durch den Wert der Zeitstempel ist also die Reihenfolge der damit verbundenen Ereignisse festgelegt. Eine Synchronisation zwischen mehreren Maschinen muss also so aussehen, dass nur die Addition als erlaubte Operation zur Verfügung steht. Somit kann die Synchronisierung nur darin bestehen, dass die Maschine mit dem kleinsten Zeitstempel solange eine Addition des Zeitstempels durchführt, bis dieser mit den anderen Maschinen übereinstimmt.

Globaler Status

Globaler Status

- Globaler Status besteht aus den lokalen Status jedes Prozesses, zusammen mit den Nachrichten, die gerade übertragen werden
- Notwendig um aktuellen Zustand des Systems zu kennen
  - Sind lokale Berechnungen unterbrochen und keine weiteren Nachrichten vorhanden, kann kein weiterer Fortschritt mehr gemacht werden
    - Deadlock
    - Verteilte Berechnung korrekt beendet
- Zustand muss konsistent sein
  - Alle lokalen Systeme erfassen ihren Zustand und synchronisieren sich

Der globale Status eines verteilen Systems besteht aus dem lokalen Status jedes Prozesses, zusammen mit den Nachrichten, die gerade übertragen werden (d.h. die gesendet aber noch nicht empfangen oder ausgewertet wurden). Die Kenntnis über den globalen Status ist aus vielen Gründen nötig. Ist beispielsweise bekannt, dass lokale Berechnungen unterbrochen wurden und sich keine weiteren Nachrichten mehr im Übertragungsstatus befinden, hat das System offenbar einen Status angenommen, in dem kein weiterer Fortschritt gemacht werden kann. Ohne Kenntnis des globalen Status könnte nun keine Aussage darüber gemacht werden, ob sich das System in einem Fehlerzustand befindet (z.B. Deadlock) oder in einem Erfolgszustand (z.B. verteilte Berechnung wurde korrekt beendet).

Eine verteilte Momentaufnahme des globalen Status muss allerdings konsistent sein. Das bedeutet, dass alle lokalen Systeme sich bei der Erfassung ihres Zustands synchronisieren.

Koordination zwischen Prozessen

Koordination zwischen Prozessen

- Synchronisierung und Koordinierung muss von jemanden angefordert, verwaltet, ausgewertet werden
- Alle Prozesse sind genau gleich
  - Ein Prozess muss die Rolle des Koordinators übernehmen
  - Wahl-Algorithmen stellen sicher, dass alle Prozesse sich auf einen neuen Koordinator einigen
  - Unterschiedliche Ansätze
    - Beispiel: Bully-Algorithmus

Die Synchronisierung und Koordinierung von Prozessen muss jedoch von jemanden angefordert, verwaltet und ausgewertet werden. Da meistens davon ausgegangen wird, dass alle Prozesse genau gleich sind, muss ein Prozess die Rolle des Koordinators übernehmen. Prinzipiell sollte jeder Prozess diese Funktion übernehmen können, welcher Prozess die Rolle tatsächlich übernimmt wird durch einen Wahlalgorithmus bestimmt.

Wahl-Algorithmen

Wahl-Algorithmen

- Bully-Algorithmus
  - Prozesse verfügen über Prozessnummer, die allen bekannt ist
    - Prozess *P* sendet bei Wahl eine Nachricht an alle Prozesse mit höherer Nummer
    - Wenn keiner reagiert, wird *P* zum neuen Koordinator (Annahme: andere Prozesse sind abgestürzt)
    - Wenn ein höher gelegener Prozess antwortet, wird dieser zum neuen Koordinator
    - Gewinner teilt seinen Sieg allen mit!
  - Wahl wird ausgelöst wenn
    - Aktueller Koordinator nicht mehr reagiert
    - Wenn neuer Prozess gestartet wird

Wie wir anhand dieses Algorithmus sehen, wird besonders auf Ausfallssicherheit Wert gelegt. Beim Absturz des aktuellen Koordinators würden alle anderen Prozesse vergeblich auf dessen Anweisungen warten und ziellos weiterarbeiten. Da allerdings die Prozesse regelmäßig den Koordinator kontaktieren und feststellen, wenn dieser keine Antworten mehr sendet, können abgestürzte Koordinator-prozesse erkannt werden. In einem solchen Fall wird ein neuer Koordinator durch das Veranlassen einer neuen Wahl bestimmt.

Ein Problem das jedoch alle Wahl-Algorithmen haben ist, dass sie über alle teilnehmenden Prozesse bescheid wissen müssen, da der gewählte Prozess seinen Sieg allen anderen Teilnehmern mitteilt. Für die Verwaltung dieser Teilnehmerliste gibt es jedoch unterschiedliche (zentrale und dezentrale) Ansätze, wobei natürlich dezentrale um einiges schwieriger zu realisieren sind als zentrale.

Wechselseitiger Ausschluss

Wechselseitiger Ausschluss

- Realisieren kritische Bereiche, die zu einem bestimmten Zeitpunkt nur von einem Prozess durchlaufen werden dürfen
  - Nur ein Client kann zu einem Zeitpunkt den Drucker benutzen
  - Gemeinsam genutzte Datenstruktur darf zu einem Zeitpunkt nur von einem Prozess benutzt werden
- Drei Ansätze
  - Zentralisierter Algorithmus
  - Verteilter Algorithmus
  - Token-Ring Algorithmus

Um Konflikte auf gemeinsam genutzte Ressourcen zu vermeiden, werden Systeme oftmals mithilfe kritischer Bereiche programmiert. Hier wird sichergestellt, dass sich zu einem bestimmten Zeitpunkt immer nur ein Prozess im kritischen Bereich befindet. Das Ein- und Austreten der Prozesse in den kritischen Bereich muss allerdings koordiniert werden. Auch hier gibt es unterschiedliche Ansätze (zentral, verteilt, Token-Ring) um diese Anforderung zu realisieren.

Zentralisierter Algorithmus

Zentralisierter Algorithmus

- Ein Prozess wird als Koordinator gewählt (z.B. Bully-Algorithmus)
- Ablauf eines Prozesses, der in kritischen Bereich eintreten möchte:
  - Sendet eine Anforderungsnachricht an Koordinator
    - Teilt mit in welchen Bereich er eintreten möchte
  - Koordinator managt Bereich
    - Erteilt Berechtigung wenn sich kein anderer Prozess im kritischen Bereich befindet
    - Verweigert Berechtigung wenn anderer Prozess im kritischen Bereich
    - Sendet keine Antwort und lässt anfragenden Prozess warten
  - Das Verlassen wird dem Koordinator mitgeteilt
    - Kann nächsten Prozess Berechtigung geben

Es hängt von der konkreten Implementierung ab, ob ein zweiter Prozess der in einen gesperrten kritischen Bereich eintreten möchte eine Fehlernachricht oder gar keine Nachricht bekommt. Eine Fehlernachricht würde eine Abweisung bedeuten, welche wieder vom Programmierer behandelt werden müsste. Gar keine Nachricht würde bedeuten, dass der anfragende Prozess so lange blockiert wird, bis der kritische Bereich frei wird. Dann würde der Koordinator den blockierten Prozessen der Reihe nach die Erlaubnis für das Eintreten in den kritischen Bereich erteilen.

Es ist leicht erkennbar, dass der Algorithmus wechselseitigen Ausschluss garantiert: Der Koordinator erteilt nur jeweils einem Prozess gleichzeitig Eintritt in einen kritischen Bereich. Außerdem ist der Algorithmus fair, weil die Anforderungen in der Reihenfolge erledigt werden, in der sie empfangen werden. Kein Prozess wartet endlos. Das Schema ist auch einfach zu implementieren und es sind nur drei Nachrichten pro Verwendung eines kritischen

Bereichs erforderlich (Anforderung, Erteilung, Freigabe).

Der zentralisierte Ansatz hat jedoch auch Nachteile. Der Koordinator ist ein einziger Ausfallspunkt, wenn er also abstürzt, stürzt möglicherweise das gesamte System ab. Wenn Prozesse normalerweise blockieren, nachdem sie eine Anforderung gestellt haben, können sie nicht zwischen einem toten Koordinator und einer verweigerten Berechtigung unterscheiden, weil in beiden Fällen keine Nachricht zurückgeschickt wird. Darüber hinaus kann ein einziger Koordinator in einem sehr großen System einen Leistungsengpass darstellen.

Verteilter Algorithmus

Verteilter Algorithmus

- Bedingung: Ereignisse im System müssen vollständig sortiert vorliegen
  - Eindeutig, welches Ereignis zuerst aufgetreten ist
  - Zeit-Synchronisation oder Zeitstempel
- Funktionsweise
  - Algorithmus sendet Nachricht an alle Teilnehmer (auch sich selbst)
    - Namen des Bereichs, eigene Prozessnummer, aktuelle Zeit(einheit)

Verteilter Algorithmus

- Funktionsweise (Fortsetzung)
  - Prozesse werten Anforderung aus
    - Nicht im kritischen Bereich: Antworte mit OK
    - Im kritischen Bereich: Keine Antwort, Anforderung in Warteschlange
    - Will selbst eintreten: Vergleicht Zeitstempel der Anforderung mit Zeitstempel eigener Anforderung
      - Niedrigste Anfragezeitpunkt gewinnt
      - Anforderung kleiner als eigene: Antworte mit OK
      - Sonst: keine Antwort, Anforderung in Warteschlange
  - Anfragender Prozess wartet auf *alle* Antworten
    - Nur dann, wenn alle zustimmen
  - Nach Verlassen sendet er OK an alle Prozesse seiner Warteschlange (andere wartende Prozesse)

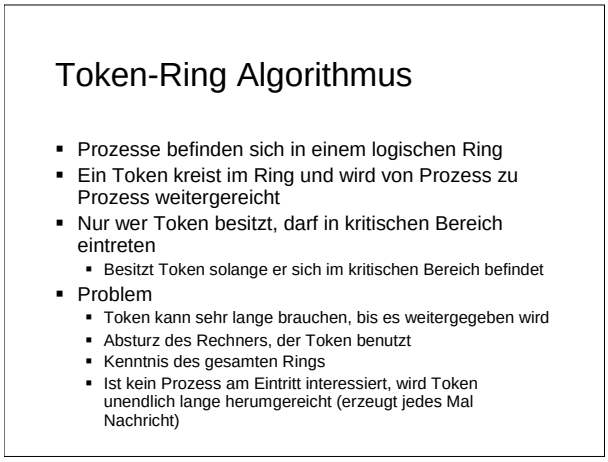
Verteilte Algorithmen versuchen, die Problematik des zentralen, einzigen Ausfallpunkts zu vermeiden. Damit verteilte Algorithmen funktionieren, müssen die Ereignisse im System vollständig sortiert vorliegen. Das bedeutet, dass jederzeit eindeutig festgestellt werden kann, welches Ereignis zuerst aufgetreten ist. Dazu müssen natürlich die beteiligten Kommunikationspartner über synchronisierte Uhren verfügen.

Beispiel: Prozess 0 sendet jedem eine Anforderung mit dem Zeitstempel 8, während gleichzeitig Prozess 2 jedem eine Anforderung mit dem Zeitstempel 12 sendet. Prozess 1 ist nicht daran interessiert, in den kritischen Bereich einzutreten, deshalb sendet er beiden Sendern eine OK-Nachricht. Prozess 0 und Prozess 2 erkennen beide den Konflikt und vergleichen die Zeitstempel. Prozess 2 stellt fest, dass er verloren hat, deshalb erteilt er Prozess 0 die Berechtigung, indem er eine OK-Nachricht sendet. Prozess 0 stellt jetzt die Anforderung von Prozess 2 für die spätere Verarbeitung in die Warteschlange und tritt in den kritischen Bereich ein. Nachdem er fertig ist, entfernt er

die Anforderung von Prozess 2 aus seiner Warteschlange und sendet eine OK-Nachricht an Prozess 2 und erlaubt diesem somit in seinen kritischen Bereich einzutreten. Der Algorithmus funktioniert, weil im Falle eines Konflikts der kleinste Zeitstempel gewinnt und sich alle über die Reihenfolge der Zeitstempel einig sind.

Leider wurde der einzelne Ausfallspunkt beim zentralen Algorithmus jetzt durch n Ausfallspunkte ersetzt. Das heißt die Situation hat sich erheblich verschlechtert. Zudem sind jetzt auch viel mehr Nachrichten für die Koordination nötig.

Token-Ring Algorithmus



Beim Token-Ring Algorithmus ist durch den Besitz (oder dem Fehlen) eines Tokens definiert, ob ein Prozess in den kritischen Bereich eintreten darf. Zu einem bestimmten Zeitpunkt kann immer nur ein Prozess das Token (die Berechtigung zum Eintritt in einen kritischen Bereich) besitzen. Das Token bleibt solange im Besitz des Prozesses, bis der kritische Bereich wieder verlassen ist.

Der Algorithmus ist sehr einfach zu implementieren. Allerdings steht auch dieser Algorithmus vor dem Problem, dass der zentrale kritische Ausfallpunkt durch n mögliche Ausfallspunkte ersetzt wurde, da bei Absturz eines Rechners der im Besitz des Tokens ist, das Token verloren geht. Zudem entstehen durch die Weiterreichung des Tokens sehr viele und zum Teil unnötige Nachrichten.

Vergleich der Algorithmen

Vergleich

Algorithmus	Nachrichten	Ausfallspunkt	Probleme
Zentralisiert	3	1	Koordinator Absturz
Verteilt	2(n-1)	n	Absturz eines beliebigen Prozesses
Token Ring	1 bis ∞	n	Verlorenes Token, Prozessabsturz

Jeder Algorithmus leidet unter Abstürzen
Allerdings: Verteilter Ansatz ist noch schlechter als zentraler Ansatz!

Die Beobachtung die aus dem Vergleich der drei Ansätze gemacht werden kann, ist sehr lehrreich. Der zentralisierte Algorithmus ist der einfachste und gleichzeitig der effizienteste, da nur drei Nachrichten erforderlich sind um in einen kritischen Bereich zu treten. Allerdings leiden alle drei Algorithmen sehr unter Abstürzen. Es müssten spezielle Maßnahmen sowie zusätzliche Komplexität eingeführt werden um zu Vermeiden das ein Absturz eines Teilnehmers den Absturz des gesamten Systems bewirkt. Insofern ist es interessant zu beobachten, dass die verteilten Algorithmen noch sensibler gegenüber Abstürzen sind als die zentralen.

Verständnisfragen



Versuchen Sie folgende Verständnisfragen zu beantworten.

1.) Welche Probleme gilt es bei der Synchronisierung von Prozessen zu berücksichtigen?

2.) Wozu ist es notwendig, den globalen Status eines Systems zu kennen?

3.) Wozu dienen kritische Bereiche und wie können wechselseitige Ausschlüsse vermieden werden?

4.) Was ist die (paradoxe) Eigenheit bei Algorithmen zur Vermeidung wechselseitiger Ausschlüsse?

5.) Bei welchen Tätigkeiten ist es notwendig, Prozesse miteinander zu synchronisieren?

Kapitel

9

Transaktionen und Nebenläufigkeit

Transaktionen regeln den Zugriff gemeinsam genutzter Daten. Bei solchen Zugriffen kann es zu Konflikten kommen die entsprechend behandelt werden müssen. Insbesondere sorgt der Transaktionsmechanismus dafür, dass das System auch bei Abbrüchen in einem konsistenten Zustand bleibt.

Eine der wichtigsten Aufgaben von Systemen, die kollaborativ Zugriff auf Daten ermöglichen ist die Gewährleistung der Konsistenz der Datenbasis. Transaktionen regeln den Zugriff auf mehrere Datenelemente innerhalb einer einzigen atomaren Operation, die entweder als Gesamtes erfolgreich abgeschlossen werden kann oder fehlschlägt. Bei Fehlschlägen sorgen Transaktionen gleichzeitig dafür, dass das System in einen konsistenten Zustand (dem zustand vor der Transaktion) zurückversetzt wird. Bei verteilten Systemen kommt allerdings die Problematik hinzu, dass die Daten selbst häufig dezentral gehalten werden und insofern die Transaktionen und dezentralen Datenbanken koordiniert werden müssen.

Lehrinhalte und -ziele

Sie sollten am Ende des Kapitels verstehen, wozu Transaktionen in verteilten Systemen benötigt werden und welche Probleme durch den Mehrbenutzerbetrieb auf verteilten Datenbanken entstehen können. Diese Erkenntnis soll Ihnen später helfen, Konsistenzbedingungen in verteilten Systemen (z.B. CSCW-Systemen) zu definieren bzw. gewährleisten zu können.

Transaktionen

Transaktionen

- Wechselseitiger Ausschluss
  - Gleichzeitige Benutzung einer gemeinsam genutzten Ressource durch höchstens einen Prozess
- Transaktionen
  - Schützen gemeinsam genutzte Daten
  - Zugriff auf mehrere Datenelemente innerhalb einer atomaren Operation
  - Schutz vor logischen Fehlern die durch gleichzeitigen Zugriff mehrerer Benutzer entstehen können
  - Bei Abbruch oder Fehler wird Ursprungszustand wiederhergestellt
  - Verteiltes System: Prozesse müssen synchronisiert werden!

Algorithmen für den wechselseitigen Ausschluss stellen sicher, dass eine gemeinsam genutzte Ressource höchstens durch einen Prozess gleichzeitig genutzt wird.

Transaktionen schützen ebenfalls gemeinsam genutzte Ressourcen vor dem gleichzeitigen Zugriff nebenläufiger Prozesse. Zusätzlich erlauben sie einem Prozess, innerhalb einer einzigen atomaren Operation auf mehrere Datenelemente zuzugreifen um diese zu verändern. Wird der Prozess während der Transaktion unterbrochen, so wird der ursprüngliche Zustand wiederhergestellt. Da bei verteilten Systemen häufig mehrere Prozesse auf die gleichen Daten zugreifen, sind Transaktionen ein zentrales Konzept in solchen Systemen. Im Gegensatz zu Einzelplatz-systemen müssen die Transaktionen jedoch in einem verteilten System synchronisiert werden.

Transaktionen

- Computermodell ist ähnlich zu realen Verhandlungen
  - Prozess kündigt an, dass Transaktion mit einem oder mehreren Prozessen beginnen will
  - Teilnehmer haben „Verhandlungszeit“ für
    - Optionen
    - Entitäten
    - Operationen
  - Initiator kündigt an, dass bisher erledigte Arbeit festgeschrieben werden soll (Abschluss)
  - Wenn Teilnehmer sich weigert, wird Zustand vor Transaktion wiederhergestellt
    - Alles oder Nichts

Während der „Verhandlungszeit“ ist es wahrscheinlich, dass die Teilnehmer bereits Daten und Entitäten erzeugen, ändern oder löschen. Alle diese Tätigkeiten müssen im Falle eines Abbruchs rückgängig gemacht werden. Diese „Alles oder Nichts-Eigenschaft“ erleichtert die Aufgabe des Systemherstellers.

Elementare Funktionen

Elementare Funktionen

Funktion	*Beschreibung*
`BEGIN_TRANSACTION`	Markiert den Start einer Transaktion
`END_TRANSACTION`	Transaktion abschließen und Festschreiben versuchen
`ABORT_TRANSACTION`	Transaktion abbrechen und alten Wert wiederherstellen
`READ`	Daten lesen
`WRITE`	Daten schreiben

Beispiel: Flugbuchung

- BEGIN_TRANSACTION
  - Res. Wien → Frankfurt ✔
  - Res. Frankfurt → Chicago ✔
  - Res. Chicago → Las Vegas ✔
- END_TRANSACTION

- BEGIN_TRANSACTION
  - Res. Wien → Frankfurt ✔
  - Res. Frankfurt → Chicago ✔
  - Res. Chicago → Las Vegas ✘
- ABORT_TRANSACTION

- Alles oder Nichts-Eigenschaft
  - Entweder alle Transaktionen durchführbar oder keine Transaktion
  - Bspl: Zurücknehmen der bereits gebuchten Flüge wenn Chicago nach Las Vegas nicht gebucht werden kann

Eigenschaften (ACID)

Eigenschaften (ACID)

- ***A**tomicity* (Atomarität): Transaktion wird entweder ganz oder gar nicht ausgeführt
- ***C**onsistency* (Konsistenz): System ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand
- ***I**solation* (Isolation): Nebenläufige Transaktionen beeinflussen sich nicht gegenseitig (Benutzer hat Eindruck als würde er mit Daten alleine arbeiten)
- ***D**urability* (Dauerhaftigkeit): Nach erfolgreichem Abschluss der Transaktion werden Änderungen / Ergebnis dauerhaft abgespeichert

Die erste Schlüsseleigenschaft von Transaktionen ist, dass sie atomar sind. Diese Eigenschaft stellt sicher, dass jede Transaktion entweder vollständig ausgefüllt wird, oder überhaupt nicht, und wenn sie ausgeführt wird, wird sie innerhalb einer einzigen, unteilbaren, unmittelbaren Operation ausgeführt. Während eine Transaktion ausgeführt wird sehen andere Prozesse die Zwischenzustände nicht (Isolation).

Die zweite Eigenschaft besagt, dass sie konsistent sind. Das bedeutet, wenn das System bestimmte Invarianten hat, die immer eingehalten werden müssen, dann müssen diese, wenn sie vor der Transaktion gelten, auch nach ihr gelten. Als Beispiel für eine solche Invariante ist das Gesetz der Gelderhaltung in einer Bank. Nach einer internen Übertragung muss der Geldbetrag in der Bank der gleiche sein wie vor der Übertragung, auch wenn das Geld jetzt auf einem anderen Konto ist. Während der Transaktion könnte diese Invariante allerdings verletzt werden (wenn Geld von Konto A

abgebucht und noch nicht auf Konto B eingelangt).

Die dritte Eigenschaft besagt, dass Transaktionen isoliert oder serialisierbar sind. Das bedeutet, wenn zwei oder mehr Transaktionen gleichzeitig ausgeführt werden, sieht für sie und alle anderen Prozesse das Endergebnis so aus, als wären alle Transaktionen sequentiell in einer bestimmten Reihenfolge ausgeführt worden.

Die vierte Eigenschaft besagt, dass Transaktionen dauerhaft sind. Ein Fehler nach dem Festschreiben kann die Ergebnisse nicht rückgängig machen oder dafür sorgen, dass sie verloren gehen.

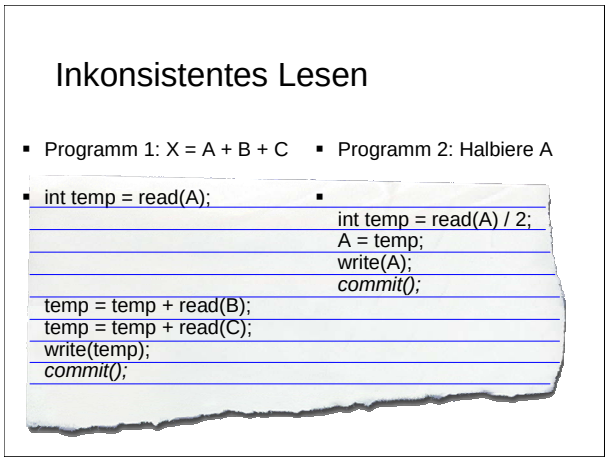
Probleme im Mehrbenutzerbetrieb

Isolation:
Probleme Mehrbenutzerbetrieb

- Inkonsistentes Lesen (*Nonrepeatable Read*)
- Lesen inkonsistenter Zustände
- Abhängigkeiten von nicht freigegebenen Daten (*Dirty Read*)
- Phantom Problem
- Verlorengegangenes Ändern (*Lost Update*)
- Integritätsverletzung durch Mehrbenutzer-Anomalie

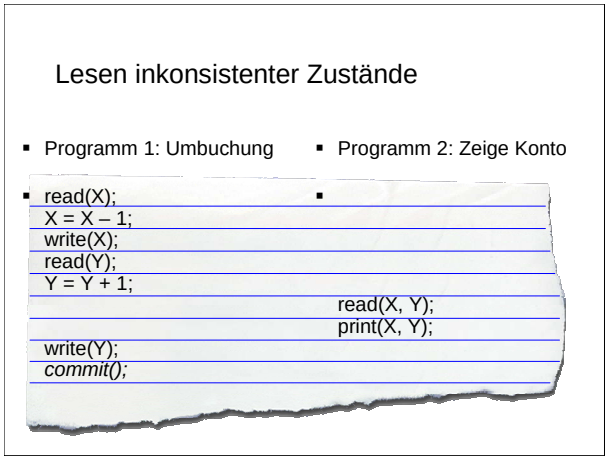
Beim Mehrbenutzerbetrieb können durch den gleichzeitigen Zugriff mehrerer Prozesse oder Benutzer Konflikte entstehen. Den ACID-Kriterien (speziell der Komponente *Isolation*) entsprechend muss diese Komplexität jedoch vor dem Benutzer verborgen bleiben. Tatsächlich soll dem Benutzer glaubhaft gemacht werden, er ist der einzige der das System benutzt. Je nach Stärke der verwendeten Mechanismen kann es insofern zu Problemen kommen, wenn gleichzeitig mehrere Benutzer auf die gemeinsamen Daten zugreifen.

Inkonsistentes Lesen



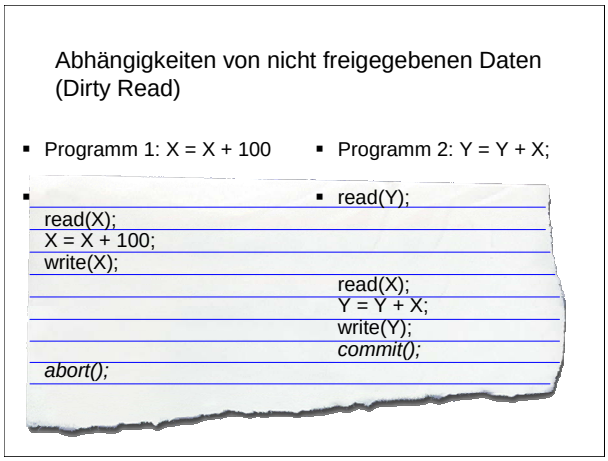
Inkonsistente Ergebnisse, die durch wiederholte Lesevorgänge hervorgerufen wurden. Ein nicht wiederholbarer Lesevorgang (inkonsistentes Lesen) findet statt, wenn eine einzelne Datenzeile in einer Transaktion mehrmals gelesen und die Zeile zwischen den Lesevorgängen von einer anderen Transaktion aktualisiert wird. Da bei den wiederholten Lesevorgängen der ersten Transaktion unterschiedliche Daten abgerufen werden, konnten die Ergebnisse in dieser Transaktion nicht wiederholt werden.

Lesen inkonsistenter Zustände



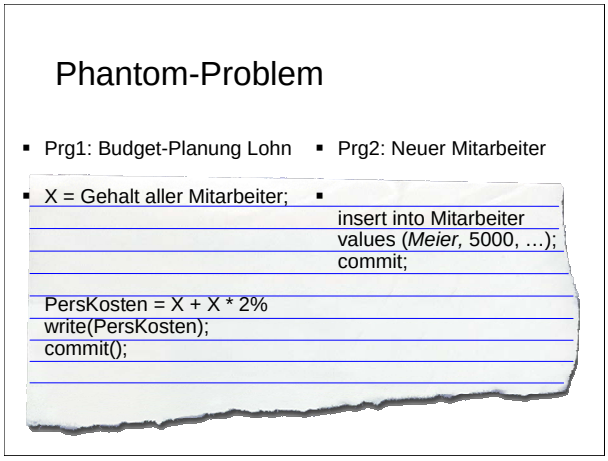
Ein Lesevorgang bei dem Daten abgerufen wurden, für die kein Commit ausgeführt wurde. Dieser Lesevorgang findet statt, wenn Daten geändert wurden, aber noch nicht geschrieben wurden. Eine zweite Transaktion liest noch alte Daten (in unserem Beispiel Y, X ist ja aktuell!). Da die zweite Transaktion sowohl aktuelle Daten (X) als auch nicht aktuelle Daten (Y) liest, können Invarianten im System verletzt sein. In unserem Beispiel fehlt auf Konto X eine Geldeinheit, die gerade auf Y überwiesen wird, Programm 2 kann dies aber noch nicht sehen. Für Programm 2 sieht es so aus, als wäre die Geldeinheit verloren gegangen.

Dirty Read



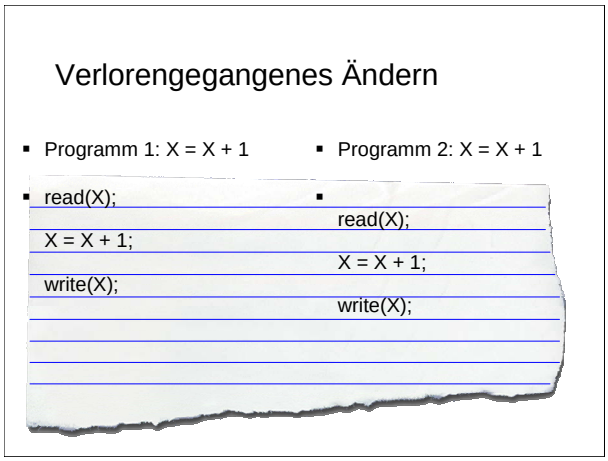
Lesen von nicht freigegebenen Daten: Ein Lesevorgang, bei dem Daten abgerufen werden, für die kein Commit ausgeführt wurde. Dieser Lesevorgang findet statt, wenn Daten durch eine Transaktion geändert werden und die geänderten Daten von einer zweiten Transaktion für weitere Berechnungen herangezogen werden. Bricht die erste Transaktion ab, so wird natürlich auch die darin getätigte Berechnung rückgängig gemacht. Allerdings bedeutet dass, das Transaktion 2 mit „falschen" Daten gerechnet hat, die Berechnung kann nicht nachvollzogen werden!

Phantom-Problem



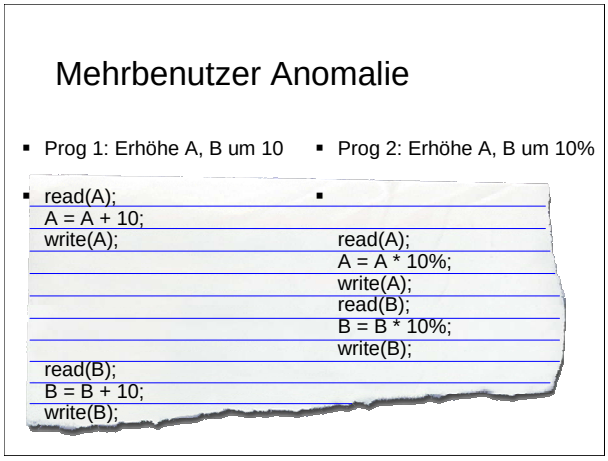
Ein Lesevorgang, bei dem eine Transaktion eine Zeile abzurufen versucht, die zu Beginn der Transaktion nicht existiert, aber von einer zweiten Transaktion eingefügt wird, bevor die erste Transaktion abgeschlossen ist. Wenn die erste Transaktion erneut nach der Zeile sucht, wird die plötzlich aufgetauchte Zeile gefunden. Berechnete Werte in Transaktion 1 stimmen nicht mit den tatsächlichen Daten überein, eine erneute Berechnung würde ein anderes Ergebnis produzieren.

Verlorengegangenes Ändern



Zwei Transaktionen lesen denselben Wert und ändern diesen. Da Transaktion 2 den Wert vor der Änderung von Transaktion 1 liest, bekommt diese die Änderung nicht mit. Beim Speichern des Wertes überschreiben sich die Transaktionen, die letzte Transaktion gewinnt.

Mehrbenutzer Anomalie



In unserem Beispiel behindern sich die beiden Programme gegenseitig, kein Programm löst wirklich seine Aufgabe. Programm 1 liest den Wert B, nachdem dieser von Programm 2 verändert wurde (also hat B einen anderen Wert als Programm 1 ursprünglich annimmt). Programm 2 hat zwar eine Pflicht erfüllt, allerdings wird der Wert B durch Programm 1 so verändert, dass sich der Wert nach Beendigung nicht um 10% erhöht hat!

Problemübersicht

Problemübersicht

- Lost Update
  - Änderungen gehen verloren
- Dirty Read
  - Ungültige Daten werden gelesen
- Nonrepeatable Read
  - Transaktionen stören sich gegenseitig, Aktionen können nicht nachvollzogen werden
- Nebenläufigkeitssteuerung
  - Muss garantieren, dass mehrere Transaktionen gleichzeitig ausgeführt werden können, während sie ständig isoliert bleiben.

Klassifizierung von Transaktionen

Klassifizierung von Transaktionen

- Flache Transaktion
  - Genügt allen ACID Eigenschaften
  - Einfachste Transaktionstyp
  - Einschränkungen
    - Erlauben nicht das festschreiben von Teilergebnissen
      - Bspl.: Flug: Buchung 1 + 2 okay, nur für 3 muss Ersatz gefunden werden
    - Problematik von verteilten Daten
      - Management der Teildaten

Flache Transaktionen

In Datenbanksystemen werden Transaktionen grundsätzlich als eine Folge von Operationen betrachtet, die den ACID-Eigenschaften genügen. Dieser Transaktionstyp wird als *flache Transaktion* bezeichnet und ist der einfachste und am häufigsten benutzte Typ. Dieser Typ hat jedoch einige Einschränkungen. Die größte Einschränkung flacher Transaktionen ist das Fehlen des Festschreibens von Teilergebnissen. Insofern ist es schwierig, die Transaktion über mehrere Knotenpunkte, die jeweils nur Teildaten verwalten, zu verteilen.

Klassifizierung von Transaktionen

- Verschachtelte Transaktionen
  - Besteht aus mehreren untergeordneten Transaktionen (Baumstruktur)
  - Bricht Vatertransaktion ab, müssen alle untergeordneten Transaktionen ebenfalls abgebrochen werden
  - Verteiltes System
    - Verschachtelung über mehrere Maschinen
  - Nachteile
    - Verwaltungsaufwand für Teile („privates Universum")

Verschachtelte Transaktionen

Verschachtelte Transaktionen sind wichtig in verteilten Systemen weil sie eine natürliche Möglichkeit bieten, Transaktionen über mehrere Maschinen zu verteilen. Verschachtelte Transaktionen verfolgen jedoch im Allgemeinen eine logische Verteilung der Arbeit der ursprünglichen Transaktion. Beispielsweise kann die Transaktion, mit der die drei verschiedenen Flüge reserviert wurden, logisch in drei untergeordnete Transaktionen zerlegt werden. Jede dieser untergeordneten Transaktion kann separat verwaltet werden und ist unabhängig von den beiden anderen.

Klassifizierung von Transaktionen

- Verteilte Transaktionen
  - Flache Transaktion, die über mehrere Maschinen verteilt ist
  - Unteilbare Transaktion auf verteilten Daten
  - Transaktionsteile kümmern sich um lokale Daten, sind aber in keiner Hierarchie!
  - Problem: Verteilte Algorithmen für Sperren und Abschließen (Festschreiben)

Verteilte Transaktionen

Flache Transaktionen, die über mehrere Maschinen verteilt sind, werden auch als verteilte Transaktionen bezeichnet. Eine verschachtelte Transaktion ist eine Transaktion, die logisch ein eine Hierarchie aus untergeordneten Transaktionen zerlegt ist. Im Gegensatz dazu ist eine verteilte Transaktion logisch eine flache, unteilbare Transaktion, die auf verteilten Daten arbeitet.

Das größte Problem bei verteilten Transaktionen ist, dass man separate verteilte Algorithmen braucht, um Daten zu sperren und die gesamte Transaktion festzuschreiben.

Nebenläufigkeitssteuerung

Nebenläufigkeitssteuerung

- Mehrere, gleichzeitig ablaufende Transaktionen, die gleiche Daten nutzen
- Oberstes Ziel: Isolation und Konsistenz gewährleisten!
  - Zugriff auf Datenelemente in bestimmter Reihenfolge
  - Endergebnis ist dasselbe, als wären alle Transaktionen sequenziell ausgeführt worden
- Scheduler: Erstellt Ausführungspläne
  - Korrekte Kontrolle der Nebenläufigkeit

Die Eigenschaften von Konsistenz und Isolation werden grundsätzlich realisiert, indem die Ausführung nebenläufiger Transaktionen korrekt kontrolliert wird. Die Konsistenz erzielt man, indem man Transaktionen Zugriff auf Datenelementen in einer bestimmten Reihenfolge erteilt, wobei das Endergebnis dasselbe ist, als währen alle Transaktionen sequenziell ausgeführt worden. Die Steuerung und Erstellung der dafür notwendigen Ausführungspläne übernimmt eine Komponente, die als *Scheduler* bezeichnet wird.

Serialisierbarkeit

Serialisierbarkeit

- Nebenläufigkeitssteuerung garantiert das korrekte Ausführen mehrerer gleichzeitiger Transaktionen
  - Unterschiedliche Ausführungsvarianten / Reihenfolge
- Eine verschränkte Ausführung mehrerer Transaktionen heißt serialisierbar, wenn ihr Effekt identisch zum Effekt einer (beliebig gewählten) seriellen Ausführung dieser Transaktionen ist

Die wichtigste Aufgabe für Nebenläufigkeitssteuerung ist die Gewährleistung, dass mehrere Transaktionen gleichzeitig ausgeführt werden können, während sie ständig isoliert bleiben. Das bedeutet, das Endergebnis sollte dasselbe sein, als wären die Transaktionen nacheinander in einer bestimmten Reihenfolge ausgeführt worden.

Serialisierbarkeit

- Transaktion 1: x = 0; x = x + 1;
- Transaktion 2: x = 0; x = x + 2;
- Transaktion 3: x = 0; x = x + 3;

- Erlaubt:
- x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3; seriell
- x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3; erlaubt
- Nicht erlaubt:
- x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3; falsch
- …

Im obigen Beispiel werden drei Transaktionen, die gleichzeitig von drei separaten Prozessen ausgeführt werden. Würden sie sequentiell ausgeführt, wäre der endgültige Wert von x gleich 1, 2 oder 3, abhängig von der Reihenfolge der Transaktionen. Es sind nun verschiedene Varianten im Ausführplan möglich, der Wert darf allerdings nur 1, 2 oder 3 annehmen.

Plan 1 ist tatsächlich serialisiert, die Transaktionen werden streng sequentiell ausgeführt, sodass diese den Bedingungen der Serialisierbarkeit genügen.

Plan 2 ist nicht serialisiert, aber trotzdem erlaubt, weil die Ergebnisse für x einen gültigen Wert darstellen, der auch durch eine rein sequentielle Abarbeitung auftreten könnte.

Plan 3 ist nicht erlaubt, weil x den Wert 5 annehmen würde, welcher durch keine der Transaktionen erreicht werden könnte.

Erlaubt man dem System die Freiheit, eine beliebige Reihenfolge der Operationen zu wählen, vorausgesetzt es entsteht die richtige Antwort, brauchen die Programmierer keinen eigenen wechselseitigen Ausschluss mehr zu realisieren.

Nebenläufigkeitskontrolle

Nebenläufigkeitskontrolle

- Wichtig ist zu verstehen, was berechnet werden soll
  - Bspl.: Wert von x soll geändert werden
  - Transaktion ist Folge von Schreib-/Leseoperationen
- Konflikt erzeugende Operationen müssen korrekt eingeplant werden

	$read_i(x)$	$write_i(x)$
$read_j(x)$	✔	✘
$write_j(x)$	✘	✘

Schreib-/Lesekonflikt Schreib-/Schreibkonflikt

Das Konzept hinter Nebenläufigkeitskontrolle ist es Konflikt erzeugende Operationen zu erkennen und korrekt einzuplanen. Zwei Operationen erzeugen einen Konflikt, wenn sie mit demselben Datenelement arbeiten und mindestens eine von ihnen eine Schreiboperation ist. Bei einem Lese-/ Schreibkonflikt ist genau eine der Operationen eine Schreiboperation. Bei einem Schreib-/ Schreibkonflikt liegen zwei Schreiboperationen vor.

Es gilt zu beachten, dass es keine Rolle spielt, ob die Konflikt erzeugenden Operationen aus derselben Transaktion stammen oder ob sie aus unterschiedlichen Transaktionen kommen. Zudem gilt, dass zwei Leseoperationen nie einen Konflikt erzeugen.

Nebenläufigkeitskontrolle

- Algorithmen synchronisieren Schreib-/Leseoperationen
  - Wechselseitiger Ausschluss durch Sperren
  - Explizite Reihenfolge der Operationen
- Pessimistische Ansätze
  - Operationen vor Ausführung synchronisiert
- Optimistische Ansätze
  - Ausführung, bei Konflikt wird eine oder mehrere Transaktionen abgebrochen

Algorithmen für die Nebenläufigkeitskontrolle können im Allgemeinen klassifiziert werden, indem man betrachtet wie Lese- und Schreiboperationen synchronisiert werden. Die Synchronisierung kann entweder durch einen wechselseitigen Ausschlussmechanismus für gemeinsam genutzte Daten (Sperren) realisiert werden oder durch die explizite Festlegung einer Reihenfolge für die Operationen unter Verwendung von Zeitstempeln. Pessimistische Ansätze synchronisieren Operationen vor deren Ausführung während optimistische Ansätze die Ausführung bei Auftreten eines Konflikts abbrechen. Welcher Ansatz gewählt wird hängt von der Aufgabenstellung und der Auftrittswahrscheinlichkeit von Konflikten ab. Generell gilt jedoch, dass pessimistische Ansätze nur bei kurzlebigen Transaktionen eingesetzt werden während optimistische Ansätze bei langfristigen Transaktionen eingesetzt werden.

Sperren

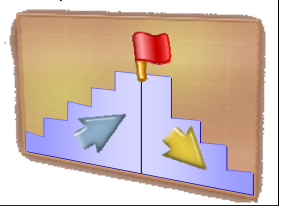
Sperren

- Scheduler sperrt Datenelement
- Wird Element nicht mehr benötigt, wird Sperre aufgehoben
- Scheduler verwaltet Sperren, sodass nur gültige Ausführungspläne entstehen

Eine Möglichkeit der Vermeidung von Konflikten ist das Setzen von Sperren auf Datenelementen. Dabei sperrt der Scheduler Datenelemente und macht diese so für andere Transaktionen nicht mehr zugänglich. Erst wenn die Sperre aufgehoben wurde (nachdem die Transaktion zu Ende ist), können die Datenelemente von einer anderen Transaktion verwendet werden.

Zwei Phasen Sperre

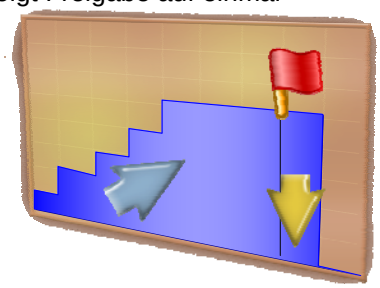
- Scheduler prüft, ob Operation in Konflikt mit einer bestehenden Sperre ist
  - Ja: Operation wird verzögert
  - Nein: Scheduler sperrt Objekt
- Scheduler gibt Sperre frei, wenn Operation erfolgreich durchgeführt
- Nach ersten Freigabe einer Sperre, können nur mehr weitere Freigaben erfolgen, keine neuen Sperren!
- → Zwei Phasen
  - Sperrphase
  - Freigabephase



Bei der Zwei-Phasen-Sperre fordert der Scheduler während der ansteigenden Phase zuerst alle Sperren an, die er benötigt, und gibt sie dann während der absteigenden Phase wieder frei.

Zwei Phasen Sperre

- Oftmals erfolgt Freigabe auf einmal (commit)



In vielen Systemen tritt die absteigende Phase erst dann auf, wenn die ausführende Transaktion beendet ist und alle Daten festgeschrieben oder abgebrochen wurden. Somit erfolgt die Freigabe in einem Schritt. Dies hat zwei große Vorteile:

1.) Die Transaktion liest immer einen Wert, der von einer festgeschriebenen Transaktion geschrieben wurde.

2.) Alle Sperranforderungen und Freigaben können vom System verarbeitet werden, ohne dass die Transaktion sie bemerkt. Die Freigabe erfolgt bei einem COMMIT oder ABORT und muss nicht extra von der Transaktion angefordert werden.

Zwei Phasen Sperre

- Verteiltes 2PL
  - Daten über mehrere Maschinen repliziert
  - Scheduler kümmert sich um
    - Management von Sperren
    - Weitergabe der Operation an lokalen Datenmanager
  - 2PL wird auf mehreren Maschinen koordiniert ausgeführt

Bei der Umsetzung des Zwei Phasen Sperrprotokolls in einem verteilten System wird davon ausgegangen, dass die Daten über mehrere Maschinen verteilt sind. Auf den einzelnen Knotenpunkten gibt es Transaktionsmanager, die mit einem zentralen Sperrmanager kommunizieren und ihm mitteilen, welche Sperren benötigt werden. Wurde eine Sperre erteilt, kommuniziert der Transaktionsmanager nachfolgend direkt mit den lokalen Datenmanagern. Nachdem die Operation abgeschlossen ist, gibt der Transaktionsmanager die Sperre an den Sperrmanager zurück.

Zeitmarkenverfahren

Zeitmarkenverfahren

- Zeitmarken sind eindeutig und werden fortlaufend vergeben
- Höhe der Zeitstempel bestimmen Reihenfolge
  - Ältere Transaktion muss vor jüngerer Transaktion lesen / schreiben
  - Verursacht ältere Transaktion einen Konflikt, wird diese abgebrochen

Ein völlig anderer Ansatz ist das Zeitmarkenverfahren. Hier werden eindeutige und fortlaufende Zeitstempel den einzelnen Transaktionsoperationen zugewiesen. Die Höhe der Zeitstempel bestimmen dann in einem Konfliktfall, welche Operation gültig ist und welche abgebrochen werden muss.

Vorteilhaft ist die einfache Realisierung in verteilten Systemen, wobei natürlich dann die Aspekte der Zeitsynchronisierung ins Spiel kommen.

Nachteil ist, dass lang andauernde Transaktionen mit hoher Wahrscheinlichkeit ihre letzten Schritte sehr spät durchführen und insofern oftmals zu spät kommen werden und insofern abgebrochen werden.

Zeitmarkenverfahren

T1	T2	T3	A		B		C	
jüngste	älteste		mrs	mws	mrs	mws	mrs	mws
ts = 200	ts = 150	ts = 175	0	0	0	0	0	0
read B			0	0	200	0	0	0
	read A		150	0	200	0	0	0
		read C	150	0	200	0	175	0
write B			150	0	200	200	175	0
write A			150	200	200	200	175	0
	write C ✗		150	200	200	200	175	*150* ✗
	abort		150	200	200	200	175	0

Verständnisfragen



Versuchen Sie folgende Verständnisfragen zu beantworten.

1.) Wozu dienen Transaktionen?

2.) Welche Probleme können im Mehrbenutzerbetrieb entstehen?

3.) Wie können Transaktionen klassifiziert werden?

4.) Was bedeutet Serialisierbarkeit?