Vorlesungsmitschrift
Verteilte Systeme
SS 2007 FH-Aachen

Bei Prof. Dr.-Ing. Oßmann

Getext von Paul Bütow

Letzte Aktualisierung: 28. Februar 2008

Inhaltsverzeichnis

1 Diverses

1.1 Ein bischen Blabla...

Ich garantiere nicht, dass der Inhalt dieses Dokuments weder korrekt noch vollständig ist. *Kursiv geschriebene Kommentare habe ich selbst hinzugefügt.* Bei Fehlern bitte per E-Mail (`skript@paul.buetow.org`) an mich wenden!

1.1.1 Anmerkung zu den Grafiken

Die Grafiken sind etwas pixelig/unscharf. Ich bitte dies zu entschuldigen. Sie erfüllen jedoch ihren Zweck.

1.1.2 Download URL

Das aktuelle Dokument mitsamt Sourcen liegt unter:

```
ftp://ftp.buetow.org/pub/studium/FHAC_VS-SS07/
```

1.1.3 Technische Informationen

Falls es interessiert: Dieses Dokument wurde erstellt mithilfe von:

- LaTeX, rubber, make
- Dia (http://www.gnome.org/projects/dia/)
- Vim (http://www.vim.org)
- FreeBSD (http://www.FreeBSD.org)

2 Vorlesung

2.1 Überblick

- Grundeigenschaften
- Systemmodelle
- Netzwerke, Interprozesskommunikation
- Verteilte Objekte, entfernte Aufrufe
- Verteilte Dateisysteme
- Namensdienste
- Zeit und globale Zustände
- Koordination und Nebenläufigkeitskontrolle
- Transaktionen
- Replikationen

2.2 Literatur

- Coulouris, Dollimore, Kindberg “Verteilte Systeme”
- Tanenbaum, van Steen “Verteilte Systeme”

2.3 Grundeigenschaften

Grundeingenschaften verteilter Systeme: Zusammenarbeit mehrerer Komponenten auf vernetzten Computern. Koordination und Kommunikation erfolgt durch den Austausch von Nachrichten.

Daraus ergibt sich die Nebenläufigkeit der Komponenten. Keine globale Uhr. Unabhängige Ausfälle von Komponenten.

2.4 Beispiele

- Tauschbörsen
- Sichere Speicherung auf verteilten Servern
- Mobiles und Allgegenwärtiges
- Rechnen

2.5 Motivation

Motivation für den Aufbau verteilter Systeme: Gemeinsame Nutzung von Ressourcen.

Was ist der Unterschied zur Betriebssystem-Motivation?

- Heterogenität der Komponenten
- Offenheit (weltweit akzeptierte Standards)
- Sicherheit bei Teilausfällen (Fehler, Angriffe)
- Skalierbarkeit (Systeme wachsen)

2.6 Beispiele für gemeinsam genutzte Ressourcen

- Hardware: Drucker, Festplatten
- Daten: Dateien
- Dienste z.B. Suchmaschinen

2.7 Client Server Modell

In verteilten Systemem verwendet man oft das "Client Server Modell". Ein Server bietet einen Dienst (Service) an ("passiv"). Ein Client fragt bei einem Server nach einen Dienst ("request for service").

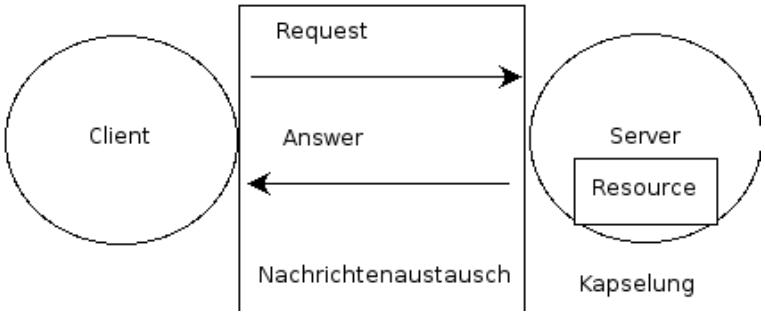
Die Dienste bergrenzen den Ressourcenzugriff auf eine wohldefinierte Operationsmenge. Implementation von Servern erfolgt üblicher Weise durch einen Prozess auf einem vernetzten Computer.

1. Client: Ruft eine Operation auf dem Server auf

2. Server: Schickt Antwort
3. Client: Erhält die Antwort

→ “Eine Interaktion”, “remote invocation”

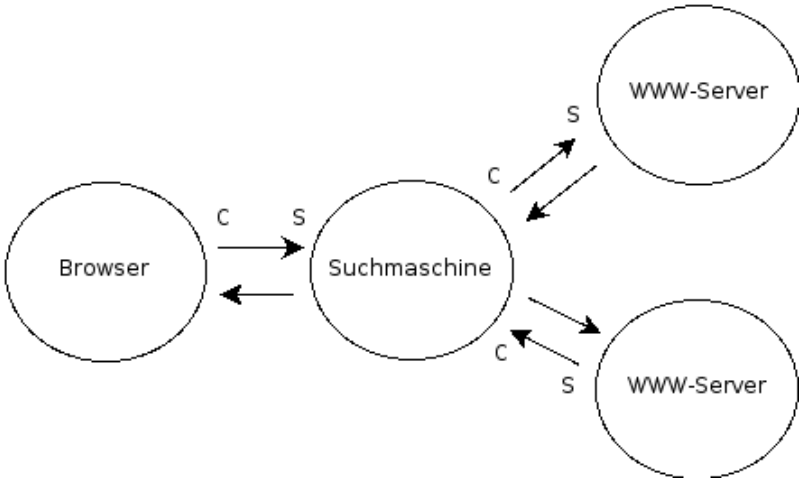
Remote heißt nicht “geometrisch entfernt”. Besser: Es wird ein Dienst erbracht, der in einem anderen Adressraum bereitliegt.



Frage

Gibt es in Java einen Adressraum? Antwort: Ja (Siehe NullPointerException).

Ein Prozess kann gleichzeitig Client und Server sein. Ähnliche Beziehung: Funktions-



bzw. Methodenaufruf. Server werden “ständig” ausgeführt. Clients reden mit der überge-

ordneten Applikation. Das Client-Server Modell beschreibt viele verteilte Systeme (nicht alle).

2.8 Ziele beim Design eines verteilten Systems

Berücksichtigt werden:

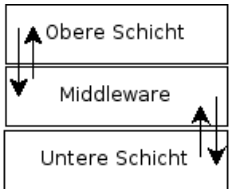
- Heterogenität
- Offenheit
- Skalierbarkeit
- Sicherheit (gegen Angriffe)
- Fehlertoleranz
- Nebenläufigkeit
- Transparenz

2.8.1 Heterogenität

Heterogenität in den Bereichen

- Netzwerk
- Computerhardware
- Betriebssysteme
- Programmiersprachen

Wesentliches Hilfsmittel zur Implementation heterogener Systeme: Einführung einer Middleware. Middleware ist eine Softwareschicht, die eine Programmierabstraktion zur Verfügung stellt, und die Heterogenität des darunterliegenden Systems verbirgt.



Beispiele: Java, RMI, CORBA, SQL, RPC. (Bereitstellung einer einheitlichen Programmierschnittstelle).

2.8.2 Offenheit

Ziel Offenheit (Um später Erweiterungen vornehmen zu koennen). Z.B. Hinzufügung neuer Dienste für unterschiedliche Client-Programme.

- Ziel wird erreicht durch Offenlegung der Spezifikation aller Softwareschnittstellen
- Wird unterstützt durch Standardisierung der Schnittstellen.

Beispiele für Offenheit:

- Internet-Protokolle
- RFC (Request For Comment)
  `http://www.ietf.org`

2.8.3 Skalierbarkeit

Die Leistungsfähigkeit von Systemen soll durch Hinzufügen von Komponenten steigen. Steigender Bedarf an Dienstleistungen muß mit beschränkten Kosten befriedigt werden.

Ist ein Linux-Cluster ein skalierbares System? Nur schwer skalierbar (wg. Neuverkabelung bei Hinzunahme neuer Rechner).

Wachstum bringt oft (relativen) Verlust an Leistung. Beispiel: Finden eines Objekts unter n (Skalierungsgröße) Objekten.

- Bei linearem Suchen: Aufwand $\sim n$
- Bei binärem Suchen: Aufwand $\sim log(n)$

Angestrebt wird ein Aufwandswachstum $\approx log(n)$ (n "Größe des Systems").

Weiteres Problem: Manche Ressourcen erschöpfen sich! Beispiel: IP-Adressen. Leistungsengpässe sollten durch Dezentralisierung vermieden werden!

Anfangs: Ein zentraler DNS-Server, der "alles" in einer Datei wusste.

2.8.4 Behandlung von Fehlern

Erkennen von Fehlern:

- Prüfsummen ("leicht möglich")
- Absturz eines Servers schwer zu erkennen

Maskierung von Fehlern:

- Mehrfaches Übertragen von Nachrichten
- Mehrfache Speicherung von Daten

Frage: Greifen diese Maßnahmen garantiert? Nein. Absolute Garantien kann man nie erwarten. Im Normalfall kann man die Sicherheit jedoch ausreichend "hochschrauben".

"Tolerieren von Fehlern" ist eine weitere Vorgehensweise.

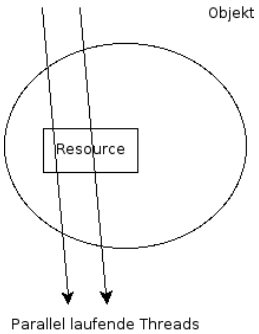
- Weiterleitung von Fehlern an den Benutzer (übergeordnete Schicht). Beispiele:
  1. Browser findet Seite nicht
  2. Java: Exception
  3. C: Nichts (nur einen Crash)
- Oft kann die Schicht, die den Fehler feststellt, nicht entscheiden, wie weiter vorzugehen ist.

Zur Maskierung von Fehlern verwendet man oft redundante Komponenten. Beispiele:

- Mehrere Routen im Netzwerk
- DNS: Jede Tabelle auf mehreren Servern
- Replizierte Datenbanken (Techniken später im Detail)

2.8.5 Nebenläufigkeit

"Gleichzeitige" Benutzung von einer Ressource durch mehrere Benutzer. Ohne Nebenläufigkeit hätten wir die sequentielle Benutzung von Ressourcen. → Schlechter Durchsatz. Im



Allgemeinen lassen Server viele Clients gleichzeitig zu. Ressource sei als Objekt gekapselt

(Siehe Bild).

In vielen Fällen ist die Synchronisation notwendig um inkonsistente Ergebnisse zu vermeiden. Objekte, die gemeinsam genutzte Ressourcen verwalten, müssen in einer nebenläufigen Umgebung korrekt arbeiten.

2.8.6 Transparenz

Transparenz bedeutet, dass der Benutzer gewisse Details nicht berücksichtigen muß, die die Implementation effizient machen.

- **Zugriffstransparenz:** Lokale und entferne Zugriffe auf Objekte erfolgen mit den gleichen Methoden. Wenn man Sockets verwendet, kann man Nachrichten lokal und entfernt gleichartig verschicken.
- **Orts- und Positionstransparenz:** Man kann auf Ressourcen zugreifen, ohne ihren Ort zu kennen. Man kann die Ressourcen von verschiedensten Orten aus benutzen.
- **Nebenläufigkeitstransparenz:** Gemeinsame "gleichzeitige" Nutzung ohne Beeinträchtigung.
- **Replikationstransparenz:** Man arbeitet mit Repliken (Kopien) ohne Beeinträchtigung.
- **Fehlertransparenz:** Partielle (teilweise) Ausfälle beeinträchtigen den Benutzer nicht.
- **Leistungs- und Skalierungstransparenz:** Das System kann neu konfiguriert oder erweitert werden, um höhere Leistungen zu erzielen.

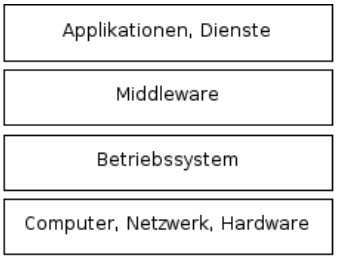
2.9 Systemmodelle

Modelle heben strukturierte Eigenschaften von Systemen hervor und ermöglichen eine einheitliche Sichtweise auf verschiedenartige Realisationen.

- **Architektonisches Modell:** Platzierung von Komponenten. Verbindungen und Beziehungen zwischen Komponenten.
- **Interaktionsmodell:** Zusammenspiel zwischen Teilen beim Nachrichtentausch.
- **Fehlermodell:** Beschreibung von Fehlermöglichkeiten der Prozesse und Kommunikationskanäle. Festlegung was man als "korrekt" bzw. "zuverlässig" interpretiert.
- **Sicherheitsmodell:** Beschreibung von Gefahren durch Angriffe und Abwehrmaßnahmen.

2.9.1 Beispiele für architektonische Modelle

- Client - Server
- Peer to Peer (gleichrangige Prozesse)
- Schichten-Architektur



Schichten-Architektur

Die Middleware stellt zur Verfügung:

- Entfernte Methodenaufrufe
- Kommunikation in Gruppen
- Benachrichtigung über Ereignisse
- Replikation von Daten
- Namensdienste (z.B. Matr.-Nr. in einer HS)
- Sicherheitskonzepte
- Transaktionen

Die Übertragung von Aufgaben an die Middleware vereinfacht den Entwurf und die Implementation von verteilten Systemen.

Aber es gibt Funktionalitäten, die vollständig und zuverlässig nur bei Kenntnis der Applikation implementiert werden können. Solche Funktionen kann man nicht vollständig in Middleware verstecken.

Beispiel: Ist TCP/IP die geeigente Middleware um Sicherheit beim Transport von E-Mails zu gewährleisten?

→ TCP/IP ist dafür nicht geeignet, weil z.B. keine geeigneten Maßnahmen gibt, um längere Ausfälle von Servern zu maskieren!

2.10 Interaktionsmodelle

- Wer kommuniziert mit wem, wann, warum, ... ?
- Wie beeinflusst das Timing von Nachrichten die Kommunikation und den Ablauf?
- Was ist der "Zustand" von einem verteilten Algorithmus?

2.10.1 Synchrones verteiltes System

- Für die Ausführungszeit zu jedem Schritt eines Prozesses gibt es bekannte obere und untere Schranken.
- Jede Nachricht wird innerhalb einer begrenzten Zeit (fehlerfrei) empfangen.
- Die Abweichungsgeschwindigkeit von lokalen Uhren im Bezug wahren Zeit ist beschränkt. (*Uhren gehen eigentlich nie zu 100 Prozent gleich. Die Abweichungsgeschwindigkeit ist das, wie schnell Uhren "auseinanderwandern"*)

Das sind sehr harte Forderungen! Von normalen Systemen sind diese nicht zu erfüllen. *Dieses Modell ist eigentlich ein Wunschmodell.* (Wahrscheinliche Werte, Mittelwerte usw. sind oft leicht zu erfüllen).

2.10.2 Asynchrones verteiltes System

- Keine Begrenzung für Prozessausführung
- Unbegrenzte Nachrichtenverzögerung
- Uhrabweichungen

Schon das "Einigungsproblem" zwischen 2 Teilnehmern im asynchronen System ist nicht lösbar.

2.11 Zeit und globale Zustände

Für Ereignisse gilt:

- Man will wissen, ob ein Ereignis vor oder nach einem anderen Ereignis stattfand (*Wir denken bisher mit vor und nach an die Zeit, später werden wir auch an was Anderes denken*).
- Man will wissen, ob man alle Informationen hat, die zu einem bestimmten Ereignis geführt haben.

- Man will wissen, ob ein System (aus mehreren Komponenten) sich (zu einem Zeitpunkt) in einem Zustand befunden hat.

Ansatz 1: Versuche Uhren zu bauen

N Prozesse (evtl. auf verschiedenen Rechnern)

Die echte wahre Zeit sei t.

Hardwareuhr: $H_i(t) \quad i = 1, ..., N$

Softwareuhr: $C_i(t) = \alpha * H_i(t) + \beta$

Ziele:

1. $C_i(t) \approx t$

   $C_i(t) - t$ heißt Uhrabweichung von Uhr i zum Zeitpunkt t.

2. $C_i(t) \approx C_j(t)$

   $C_i(t) - C_j(t)$ heißt Synchronisationsfehler zwischen den Uhren i und j zum Zeitpunkt t.

Uhren werden "synchronisiert". $S(t)$ sei eine Referenzuhr (z.B. UTC).

1. Externe Synchronisierung über einem Zeitintervall gilt für alle $t \in I$:

   $|S(t) - C_i(t)| < D$ für alle $i = 1, ..., N$

   So heissen die Uhren C_i mit $i = 1, ..., N$ extern synchronisiert mit Genauigkeit D. (D ist fest für alle i, t). Uhrabweichung in I ist kleiner als D.

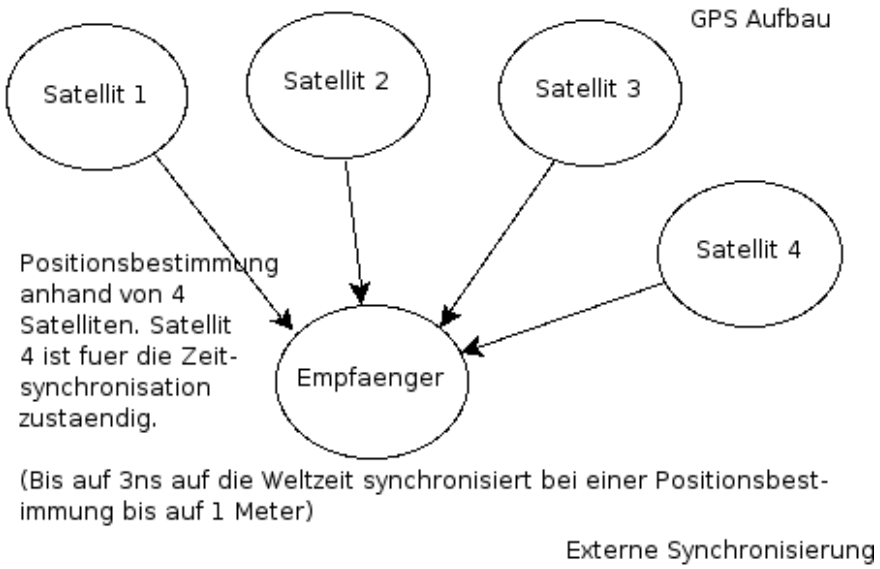
   5 Sekunden Abweichungen pro Tag: $\frac{5s}{24*3600s} = 57 * 10^{-6} = 57ppm$

   Bei billigen Uhren die man im Laden bekommt kann man leicht $10ppm$ *erreichen. Atomuhren liegen bei einem Faktor von ca.* 10^{-15}.

2. Interne Synchronisierung git für alle $t \in I$ und $i, j \in 1, ..., N$ dass

   $|C_i(t) - C_j(t)| < D$

   so heissen die Uhren intern synchronisiert mit Genauigkeit D. (Uhren stimmen untereiander bis auf D überein.



Ist ein System extern D-Synchronisiert, ist es intern garantiert $2D$-synchronisiert. Wenn ein System intern D-Synchronisiert ist, kann das zugehörige externe D beliebig groß sein.

2.11.1 Einige Korrektheitsbegriffe für Uhren

Eine Uhr die still steht, geht 2 mal am Tag richtig. Eine Uhr die 1 Minute vor geht, geht immer falsch?!? Es ist nicht einfach die Korrektheit einer Uhr zu definieren.

1. Eine Hardwareuhr heißt korrekt, wenn ihre Abweichgeschwindigkeit (drift-rate) durch eine Schranke $\rho > 0$ begrenzt ist.

   D.h., der Fehler beim Messen von Intervallen zwischen Echtzeiten t und t' ist begrenzt durch:

   $$(1-\rho)(t'-t) \leq H(t') - H(t) \leq (t'-t)(1+\rho)$$

   $$\Leftrightarrow 1-\rho \leq \frac{H(t')-H(t)}{t'-t} \leq 1+\rho$$

   $$\rightarrow -\rho \leq \frac{H(t')-H(t)}{t'-t} - 1 \leq \rho \text{ (Gangabweichungsgeschwindigkeit)}$$

   Erster Korrektheitsbegriff: Gangabweichungsgeschwindigkeit beschränkt.

Frage: Darf eine in dem Sinne korrekte Uhr "springen"? Antwort: nein! *Differenzierbare Funktionen müssen stetig sein.* Derartige Uhren dürfte man nicht synchronisieren ("stellen"). → Dieser Korrektheitsbegriff ist oft zu stark.

2. Ein weiterer Korrektheitsbegriff: Eine Hardwareuhr heißt korrekt, wenn sie "monoton" ist.

   $$t < t' \rightarrow H(t) < H(t')$$

   → Das ist oft ein zu schwacher Korrektheitsbegriff.

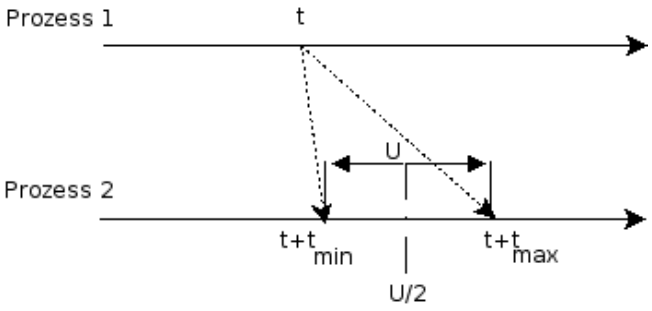
3. Eine Hardwareuhr heißt korrekt, wenn sie monoton ist und die Gangabweichung zwischen Synchronisationspunkten begrenzt ist.

2.12 Methoden zur Synchronisation

2.12.1 Interne Synchronisation in einem synchronem System

Annahme: Die Transportzeit einer Nachricht (vom Senden zum Empfangen) liegt garantiert zwischen t_{min} und t_{max}. D.h. $t_{min} \leq t_{tr} \leq t_{max}$. Die anderen Zeiten seien vernachlässigibar. Nenne $t_{max} - t_{min}$ die "Unsicherheit".

Prozess P_1 senden seine Zeit t in einer Nachricht zu Prozess P_2.



P_2 setzt beim Empfang seine Zeit auf:

$t_2 = t + \frac{1}{2}(t_{max} + t_{min}) \rightarrow$ Synchronisationsfehler $< \frac{U}{2}$

Wenn man so N Uhren intern synchronisiert, ist der Fehler $\leq U(1 - \frac{1}{N})$

In asynchronen Systemen können keine Garantien gegeben werden.

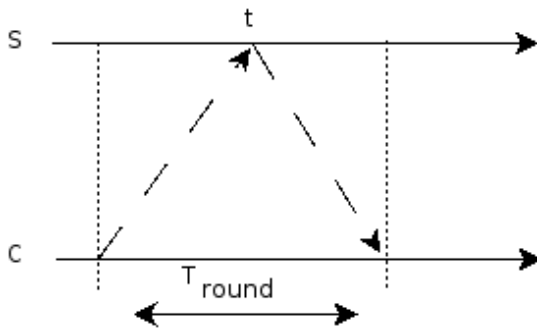
Beobachtung: Die Round-Trip Zeit (RTT) in realen Systemen ist oft relativ kurz.

2.12.2 Christians Methode zur externen Synchronisation

- Es handelt sich um einen "probabilistischen Algorithmus".
- Mit Wahrscheinlichkeit $> p_0$ ist eine Synchronisation möglich.
- Typische Roundtrip-Zeiten sind $1, ..., 10ms$
- Abweichungsgeschwindigkeit der lokalen Uhr die zur Roundtrip-Zeimessung benutzt wird, ist vernachlässigbar klein.
- Der Prozess, der mit Round-Trip den Zeitserver nach der Zeit gefragt hat, setzt seine Zeit auf t (Zeitpunkt des Servers) $+\frac{T_{round}}{2}$ sofern man keine zusätzlichen Informationen hat.

Christians Methode: Benutze die Round-Trip Zeit, um die Transportzeit von Nachrichten für den Einzelfall zu "kennen".

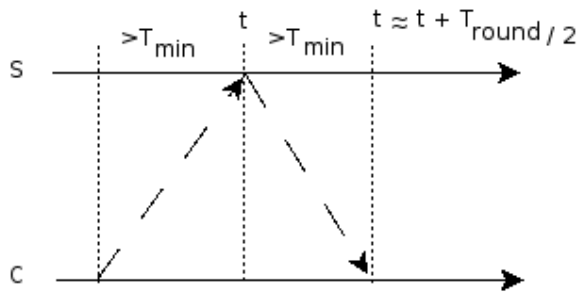
Der Prozess p setzt seine eigene Zeit auf $t + \frac{T_{round}}{2}$. Mit $t :=$ Zeitstempel des Servers. Annahme: Es ist etwas mehr bekannt, nämlich eine untere Schranke T_{min} für den Nach-



richtentransport ($\rightarrow T_{round} \leq T_{min}$). Client setzt seine Uhr wie vorher auf $t + \frac{T_{round}}{2}$. Die Genauigkeit ist $\pm(\frac{T_{round}}{2} - T_{min})$.

Synchronisiere nur, wenn die beobachte Round-Trip-Zeit klein genug ist. Vorsicht: Beim Synchronisieren sind ggf. Korrektheitsbedinungen (Monotonie o.Ä...) zu beachten!

- Schutz gegen Ausfall: Einrichtung mehrerer Server.



- Erreichen kleinerer Round-Trip-Zeit: Benutze "nahe" Server.
- Problem: Betrügerische Server: Kryptologische Methoden (Authentifizierung). Abgleich mehrerer Servern.

2.12.3 Berkeley Algorithmus zur internen Synchronisierung

- Auswahll eines Koordinierungsmasters.
- Der Master fragt die Slaves ab. Diese senden ihre eigene Uhrzeiten zurück.
- Der Master schätzt die lokalen Zeiten der Slaves durch Beobachtung der Roundtrip-Zeit.
- Der Master bildet (durch gewichtete Mittelung) eine "möglichst genaue" Referenzzeit.
- Der Master schickt allen Slaves "Korrekturwerte" die angeben, wie sie jeweils aus ihrer lokalen Zeit die Referenzzeit berechnen können.

Experiment: 15 Computer kann man in üblichen LANs relativ leicht auf ca. 25μs synchronisieren. (Bei ca 10ms Roundtrip)

Keine Garantien!

Problem (weiterhin): Uhren ermöglichen es nicht immer, die Frage zu beantworten, ob ein Ereignis a vor einem Ereignis b stattgefunden hat.

Einführung eines anderen Konzepts:

2.13 Logische Zeit und logische Uhren

Beobachtung:

1. Innerhalb eines Prozesses ist es leicht zu entscheiden, was "früher" oder "später" ist.
2. Prozesse treten nur mit Nachrichten in Kontakt. (Keine Back-Channels)

Modellvorstellung: Ein Prozess ist eine wohldefinierte Abfolge von Ereignissen.

Typische Ereignisse: Senden einer Nachricht, Empfangen einer Nachricht, Berechnung eines Ergebnisses, Änderung eines Attributes eines Objekts, Speichern einer Datei.

Idee (Lamport) definiere eine "geschehen vor" Relation. Relation zwischen zwei Ereignissen.

Schreibweise "$\rightarrow$"

Zuerst für Ereignisse in einem Prozess i:

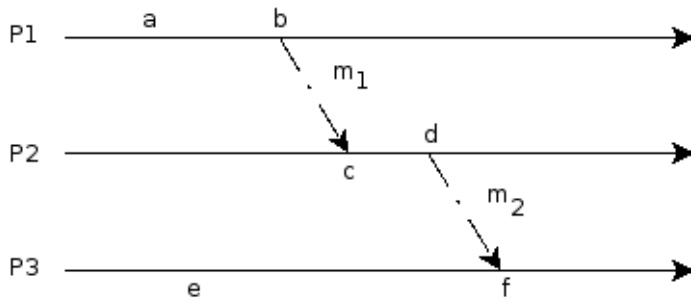
$l_1 \rightarrow_i l_2$ l_1 und l_2 im selben Prozess, dann ist die Reihenfolge klar.

Die durch $\rightarrow_i$ sortierte Folge der Ereignisse in einem Prozess nennt man eine "history". Klar ist: Eine Nachricht kann erst empfangen werden, nachdem sie gesendet wurde.

Definiere die "geschehen-vor" Relation $\rightarrow$ nun wie folgt:

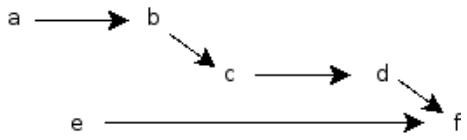
1. Falls es einen Prozess i gibt, so dass $l \rightarrow_i e'$ dann gillt $l \rightarrow e'$
2. Für jede Nachricht m gilt send(m)$\rightarrow$receive(m)
3. Es gilt $e \rightarrow e'$ und $e' \rightarrow e''$ dann gilt $e \rightarrow e''$

Beispiel:

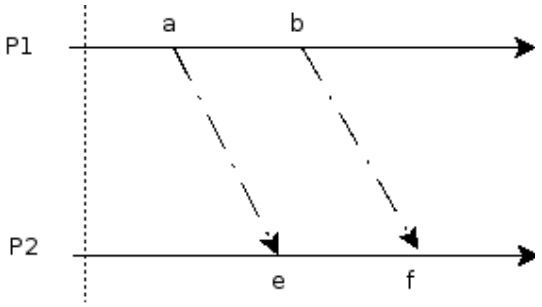


Ergibt:

Wegen 1.: $a \to b$ $c \to d$ $e \to f$
Wegen 2.: $b \to c$ $d \to f$



Frage: Gilt $e \to c$?? Kann es sein, dass für zwei Ereignisse e und e' sowohl $e \to e'$ als auch $e' \to e$ gilt? Antwort: Nein! Hier gibt es zwei Begründungen für $a \to f$.



Es gilt weder $a \to e$ noch $e \to a$ so bezeichnet man a und e als nebenläufig. Schreibweise: $a||e$

Die "geschehen vor" Relation stellt einen möglichen Datenfluss dar. Fragen der Art: "Hat Ereignis e_i das Ereignis e_j möglicherweise beeinflusst?" sollen beantwortet werden.

Wie stellt man "→" auf einem Rechner fest?

Erster Ansatz:

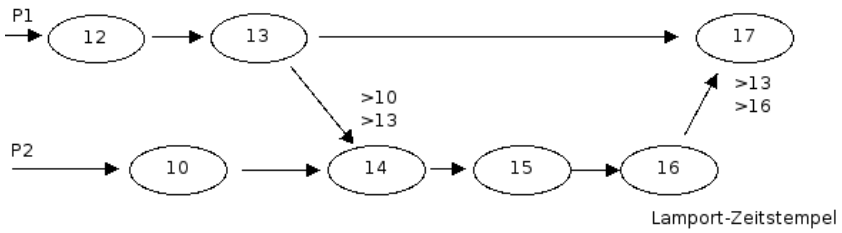
2.13.1 Logische Uhr von Lamport

- Jeder Prozess i hat seine eigene logische Uhr L_i. L_i wächst monoton.
- Die eigene Zeit wird Nachrichten als Zeitstempel mitgegeben.

Wie wird L_i jeweils aktualisiert? L_i wird vor (bei) einem Ereignis in P_i erhöt.

- Sendet Prozess P_i eine Nachricht m, so wird der Nachricht der Zeitstempel $t = L_i$ mitgegeben.
- Beim Empfang einer Nachricht berechnet Prozess P_j das Wort $L_j = max(L_j, t)+1$ und weist diesen Stempel dem Empfangsereignis zu.
- Die L_i (Lamport-Zeitstempel) werden mit 0 initialisiert.

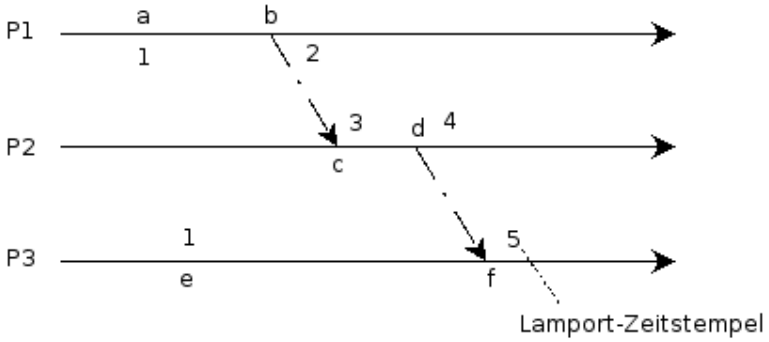
Diese Definition wurde Wörtlich aus dem Buch abgeschrieben, nicht ganz Widerspruchsfrei. Daher wirds im Folgenden nochmal Veranschaulicht dargestellt.



Anschaulich: Das Empfangsereignis bekommt einen Zeitstempel der sowohl höher ist als der Stempel des Sendeereignis als auch der Zeitstempel, der im eigenem Prozess vor ihm liegt.

Frage: Was kann man aus Lamport-Zeitstempeln ableiten? Beispiel:

- Wenn $e \rightarrow e'$ dann gilt $L(e) < L(e')$
- Es gilt nicht: $L(e) < L(e') \Rightarrow e \rightarrow e'$ (Ein Beispiel: $L(b) > L(e)$ aber $e||b$)



- Man kann aus $L(e) < L(e')$ nur folgern $e' \not\rightarrow e$

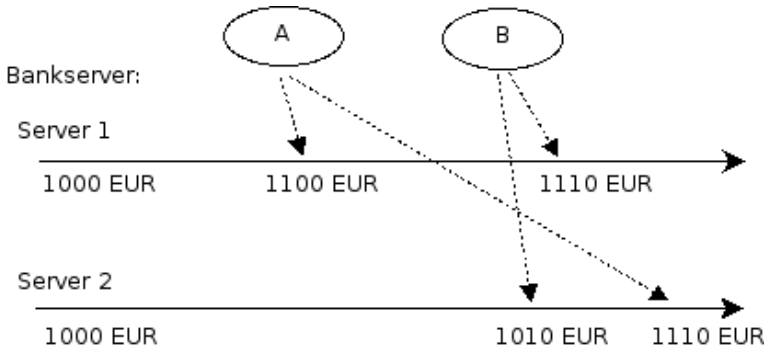
Manchmal gibt es Probleme, bei welchen sich alle Teilnehmer über eine Reihenfolge einig sein müssen, wobei egal ist, welche Reihenfolge es ist.

Beispiel: Eine Bank führt Konten mehrfach damit man gegen Abstürze einzelner Server gesichert ist. Transaktionen (Überweisungen) werden an alle Server geschickt. Für den Kunden ist die Ausführungsreihenfolge von Transaktionen egal.

Korrektheitsforderung:

- Jede einzele Transaktion muß korrekt gesendet werden.
- Alle Server müssen zu dem gleichen Ergebnis kommen. D.h. die Server müssen sich auf eine (beliebige) Reihenfolge einigen.

Wenn die Server sich nicht einigen könnte folgendes passieren:



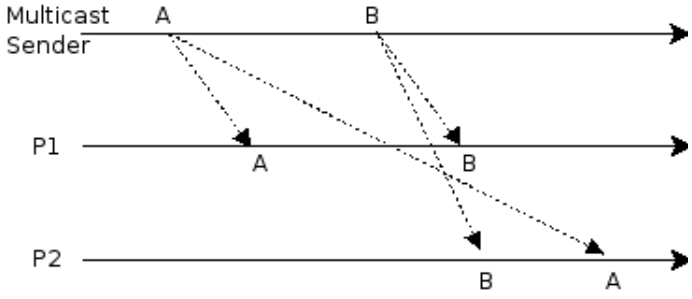
- Kunde hat 1000 EUR

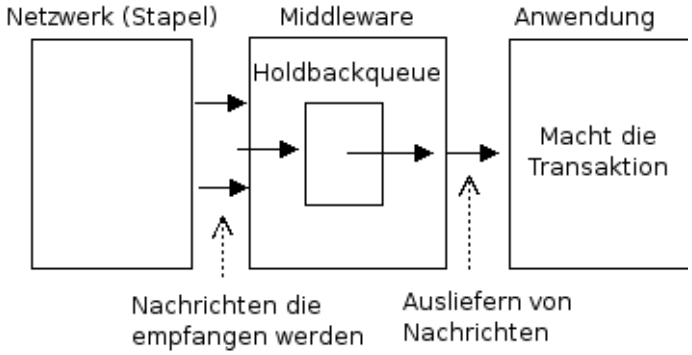
- Transaktion A: Einzahlung 100 €
- Transaktion B: Verzinsung um 1%

Dieses Probem kann mit Lamport-Zeitstempeln gelöst werden, wie im Folgendem gezeigt wird:

Grundidee der Lösung: Alle Server müssen die Transaktionen in der gleichen Reihenfolge ausführen. Angewandtes Konzept ist ein "vollständig geordneter Multicast" (Totally Ordered Multicast).

Unterscheide von jetzt an konzeptionell zwischen dem Empfang von Nachrichten und





der Auslierferung von Nachrichten. Bei einem "vollstängig geordneten Multicast" sorgt die Middleware dafür, dass alle Anwendungen die Nachrichten in der gleichen Reihenfolge ausgeliefert bekommen.

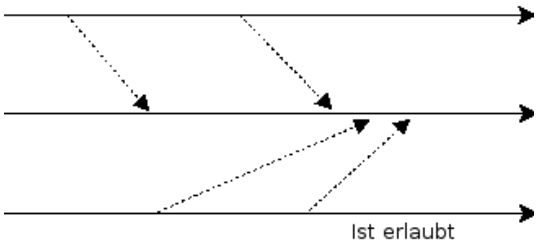
1. Teilschritt zur Realisation eines vollst. geordn. Multicasts mit Hilfe von Lamport Zeitstempeln: Erweitere die Lamport-Zeitstempel um die Rechner-Nummer

(Prozess-ID o.Ä.) als Nachkommastelle!

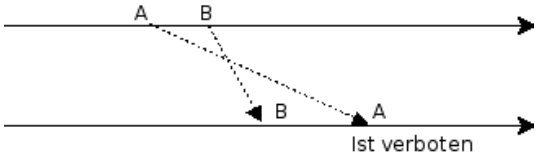
⇒ Es gibt keine gleichen Zeitstempel mehr! D.h. für zwei verschiedene Ereignisse e_1 und e_2 gilt jetzt entweder $L(e_1) < L(e_2)$ oder $L(e_2) < L(e_1)$.

2. Teilschritt: Implementiere damit den "Totally Ordered Multicast" wie folgt:

   Jede Nachricht bekommt den Zeitstempel des Senders und wird auch an den Absender gesandt. Empfangene Nachrichten werden in eine lokale Warteschlange eingefügt die nach Zeitstempeln geordnet wird.



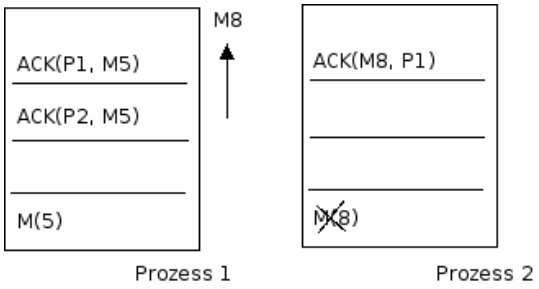
   Nun stelle eine weitere Forderung auf: Die Nachrichten die ein Sender abschickt, kommen bei allen Empfängern in der Reihenfolge des Absendens an! Diese For-



   derung ist entscheidend, aber relativ leicht zu erfüllen (z.B. durch Nummerierung von Paketen oder verbindungsorientiertem Protokoll).

   Der Empfänger einer Nachricht multicastet eine Bestätigungsnachricht (ACK Acknowledge) an alle anderen Prozesse. (Nachricht bekommt Zeitstempel und wird an den eigenen Absender gesandt!) Nun sammeln sich in den Warteschlangen die Originalnachrichten (die vollst. *geordnet* ausgeliefert werden sollten) und die zugehörigen Bestätigungen.

   Eine Nachricht wird aus der Hold-Back Queue ausgeliefert, wenn sie an der Spitze steht und man von allen Prozessen die zugehörige ACK-Meldung bekommen hat. Damit werden Nachrichten erst ausgeliefert, wenn klar ist, dass alle anderen Pro-

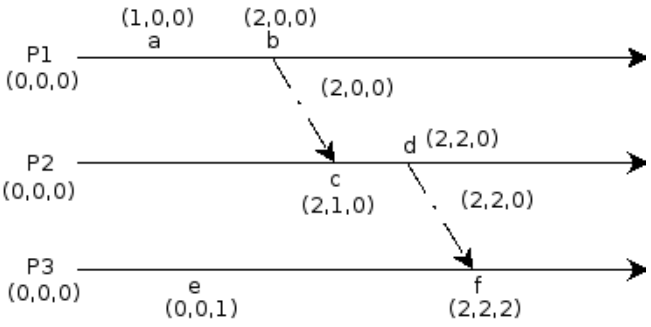


zesse diese Nachricht auch empfangen haben.
Vorsicht: Jede Nachricht die "gemulticastet" werden soll hat bei N Teilnehmern N^2 ACK-Nachrichten zur Folge.

Die Lamport-Zeitstempel waren nicht geeignet, die "$\rightarrow$"-Relation auf einfache Weise zu überprüfen. Deswegen: Erweiterung des Konzepts.

2.13.2 Vektor-Zeitstempel

- N-Prozesse, jeder Prozess I führt seine eigene Vektor-Uhr V_i. Jede Vektor-Uhr V_i ist ein Vektor mit N Einträgen $V_i[j]$.
- Idee: $V_i[j] \quad$ (mit $i \neq j$) ist die Anzahl von Ereignissen in P_j, von welchen eventuell ein Datenfluss zu P_i erfolgen konnte.
- $V_i[i]$ ist die Anzahl der Ereignisse, welchen P_i selbst Zeitstempel zugewiesen hat.
- Initialisierung: $V_i[j] = 0 \quad$ für alle $i, j = 1..N$.



Zuweisung eines Zeitstempels zu einem Ereignis:

1. Eigenzeit erhöhen $V_i[i] := V_i[i] + 1$
2. Bei einem Sendeereignis schickt man den eigenen Zeitstempel in der Nachricht mit, d.h. die korrekte Vektoruhr.
3. Beim Empfang: Nach Erhöhen der Eigenzeit (vgl. 1) Bildung des Maximums aus empfangenem Zeitstempel und eigener Vektoruhr (elementweise) ergibt neue Vektoruhrzeit.

Vergleich von Vektorzeitstempeln

- $v = v'$ iff $v[j] = v'[j]$ ist für alle j
- $v \leq v'$ iff $v[j] \leq v'[j]$ ist für alle j
- $v < v'$ iff $v[j] \leq v'[j]$ und $v \neq v'$ ist für alle j

Man kann zeigen: $e \rightarrow e' \Rightarrow v(e) < v(e')$ *Das ging auch schon bei Lamport-Zeitstempeln.* und $v(e) < v(e') \Rightarrow e \rightarrow e'$. *Das ging bei den Lamport-Zeitstempeln noch nicht.*

Man kann erkennen, ob Ereignisse nebenläufig sind: $x||y$ wenn weder $x \leq y$ ist noch $y \leq x$ ist.

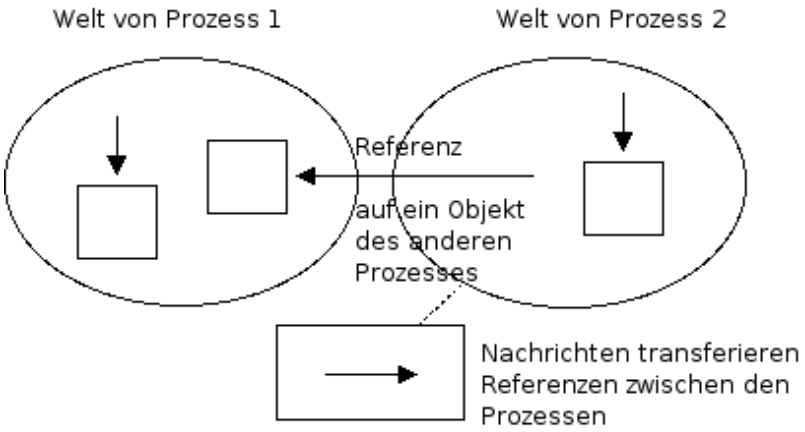
D.h. Vektorzeitstempel erlauben es, die "$\rightarrow$"-Relation auszuwerten. **Nachteil:** Hoher Aufwand (Speicher und Netzlast) bei grösserem N. **Aber:** Es gibt kein Verfahren mit "kleinerem Aufwand" um durch Vergleich logischer Uhren Nebenläufigkeit festzustellen.

2.14 Globale Zustände

Aufgabe: Man muß feststellen können ob bestimmte Situationen vorliegen.

Beispiele:

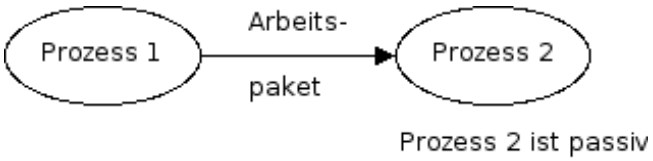
- Ist ein Objekt überflüssig? (Garbage Collection)
- Ist ein verteilter Algorithmus fertig?
- Ist ein Deadlock vorhanden?
- Ist garantiert, dass Ereignis A "vor" Ereignis B auftritt?
- Kann es möglicherweise passieren, dass zwei Benutzer eine Datei gleichzeitig benutzen?



Zur verteilten Garbage Collection: Objekte werden durch Referenzen benutzbar. Referenzen, die momentan transferiert werden, müssen berücksichtig werden.

Das Erkennen, ob ein Algorithmus in einem verteilten System abgeschlossen ist, kann z.B. schwer sein, wenn folgendes gemacht wird:

Prozesse warten (passiv) auf Aktivierung (d.h. Zusendung eines Arbeitspaketes). Während



das Paket unterwegs ist, sind alle Prozesse "passiv", aber der Algorithmus noch nicht abgeschlossen.

Problem: Weil es keine globale Zeit gibt, ist man nicht in der Lage eine Momentaufnahme (Snapshot) des Systems herzustellen. Deswegen kann man nicht feststellen ob zu einer Zeit t eine Situation vorliegt.

Frage: kann man aus lokalen Zuständen, die zu unterschiedlichen Zeiten aufgenommen wurden, einen sinnvollen globalen Zustand zusammensetzen?

Einfache aber schlechte **Antwort:** Manchmal.

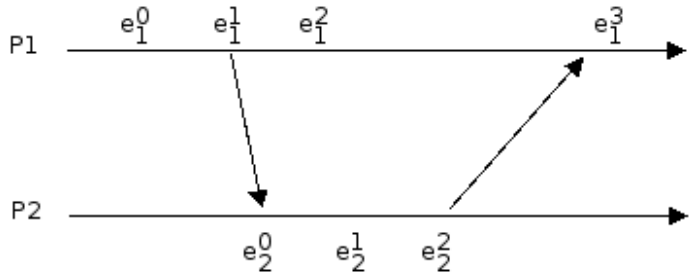
Antworten auf Fragen der Art:

- Hätte das System in dem Zustand X sein können?
- Ist das System garantiert nie im Zustand X gewesen?

sind teilweise möglich.

2.14.1 Konzepte zur "Formalisierung"

Idee des "Schnittes" eines Systems



Die "History" von P_1 ist $e_1^0 \quad e_1^1 \quad e_1^2 \quad e_1^3 \quad ...$

Die "History" von P_2 ist $e_2^0 \quad e_2^1 \quad e_2^2 \quad e_2^3 \quad ...$

$$\begin{array}{cccc|} e_1^0 & e_1^1 & ... & e_1^k \\ e_2^0 & e_2^1 & ... & e_2^j \end{array}$$

e_1^k und e_2^j bilden die Front eines Schnittes. Die Front einschliesslich alle vorherigen Zustände sind "im Schnitt enthalten"

Ein möglicher Schnitt des obigen Systems wäre:

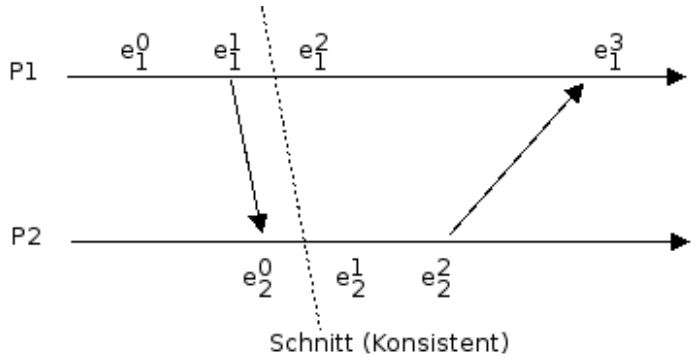
$$\begin{array}{ccc|} & & e_1^0 \\ e_2^0 & e_2^1 & e_2^2 \end{array}$$

Ein Schnitt ist konsistent, wenn für jedes Ereignis des in ihm enthalten ist, auch alle Ereignisse enthalten sind, die im Sinne der "geschehen vor"-Relation "vor ihm" liegen.

Der Schnitt *von oben* ist inkonsistent, weil z.B. $e_1^1 \rightarrow e_2^0$ und e_2^0 ist im Schnitt und e_1^1 ist nicht im Schnitt. Ein konsistenter globaler Zustand wird definiert als konsistenter Schnitt. Der Zustand eines Systems entspricht nie einen inkonsistenten Schnitt.

Die konsistenten Schnitte ensprechen alle möglichen Situationen. Ein verteiltes System bewergt sich durch konsistente globale Zustände.

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow ...$



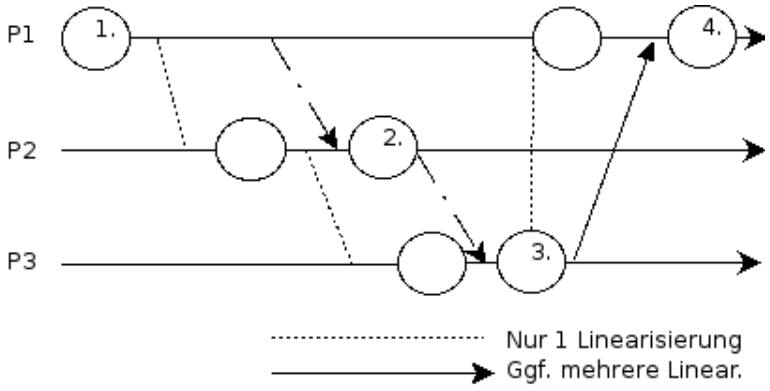
Ereignisse "→" in einzelnen Prozessen: "Senden", "Empfangen", "inneres Ereignis".

Ein "RUN" ist eine Anordnung aller Ereignisse in einem System die konsistent ist mit jeder lokalen history. Für das obige System ist

$$e_1^0 \quad e_1^1 \quad e_1^2 \quad e_1^3 \quad e_2^0 \quad e_2^1 \quad e_2^2$$

ein RUN.

Ist ein RUN auch konsistent mit der globalen "geschehen vor"-Relation, nennt man ihn eine Linearisierung des Systems. Es kann mehrere Linearisierungen geben!



2.14.2 Auswertung von Prädikaten in globalen Zuständen

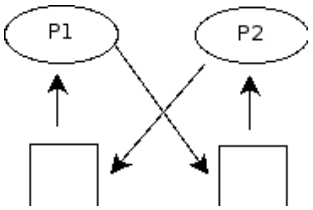
Interessante Prädikate:

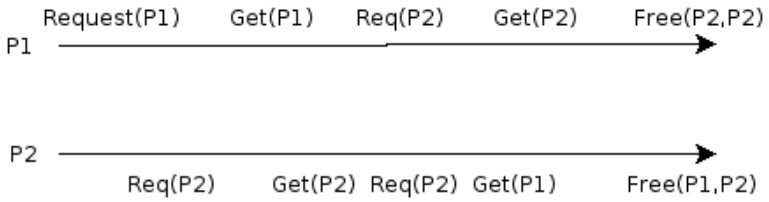
- System ist in einem Deadlock
- Objekt ist “Garbage”
- Prozess ist terminiert
- Ein Attribut eines Objektes hat einen bestimmten Wert

Für viele Prädikate ist die Auswertung schwer. Deswegen definiert man “neue Prädikate” mit Hilfe der gewünschten Prädikate.

φ ist ein Prädikat das in einem globalen Zustand ausgewertet werden kann.

Möglicherweise φ iff Es gibt einen konsistenten globalen Zustand S, der eine Linearisierung der Hostorie H durchläuft, so dass $\varphi(S) = true$. Definitiv φ iff Für alle





Linearisierungen L von H gibt es einen konsistenten globalen Zustand S (der kann von der Linearisierung abhängen) den L durchläuft mit $\varphi(S) = true$.

Vorsicht: $\overline{moeglicherweise} \quad \varphi \rightarrow definitiv \quad \overline{\varphi}$

aber

aus definitiv $\overline{\varphi} \not\rightarrow \overline{moeglicherweise \quad \varphi}$

Oft interessieren einen “stabile Prädikate”. Ein Prädikat heißt stabil, wenn daraus, dass es einen Zustand wahr ist folgt, dass es in möglichen Folgezustände wahr ist.

Anschaulich:

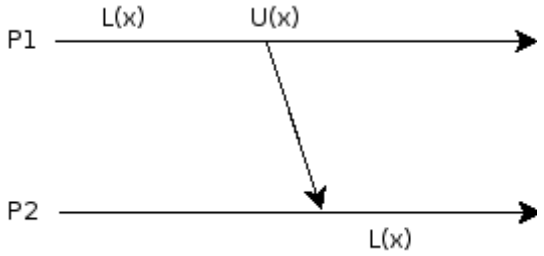
Möglicherweise φ: Möglicherweise hat sich das System so entwickelt, dass zu irgendeinem Zeitpunkt einmal φ galt.

Definitiv φ: Das System hat sich definitiv so entwickelt, dass auf jeden Fall in irgendeinem Zustand φ galt.

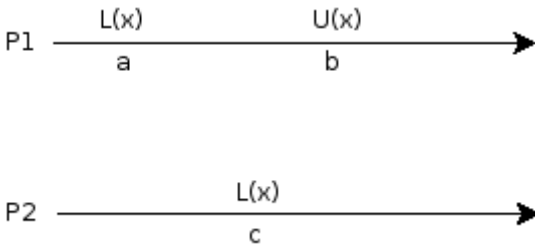
Beispiel:

- Objekte können von Prozessen "gelockt" werden
- $L(x)$ Lock auf ein Objekt haben
- $U(x)$ Unlock (Lock sperrt) zurückgeben

Prädikat φ "es liegt ein Konflikt vor". D.h. wenn ein Objekt zweimal gelockt wurde, liegt ein Konflikt vor.

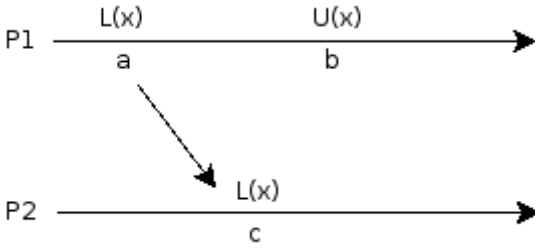


- definitiv (Konflikt) nein
- möglicherweise (Konflikt) nein
- definitiv (kein Konflikt) ja
- möglicherweise (kein Konflikt) ja

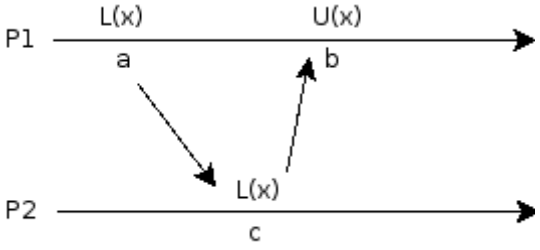


Alle Linearisierungen des Systems: *abc*, *acb*, *cab*.

- definitiv (Konflikt) nein *abc*
- möglicherweise (Konflikt) ja *acb*
- definitiv (kein Konflikt) ja am Anfang
- möglicherweise (kein Konflikt) ja am Anfang



Reicht noch nicht um "definitiv (Konflikt)" wahr zu machen (denn es gibt noch die Linearisierung *abc* die konfliktfrei ist)



- definitiv (Konflikt) ja
- möglicherweise (Konflikt) ja

- definitiv (kein Konflikt) nein
- möglicherweise (kein Konflikt) nein

2.15 Koordination und Übereinstimmung

Grundprobleme:

- Zugriff auf gemeinsam genutzte Ressourcen muß koordiniert werden.
- In vielen Situationen muß Einigkeit hergestellt werden (z.B. wer der Master ist, ob eine Transaktion gültig ist, ...).

Algorithmmus für wechselseitigen Ausschluss:

N Prozesse $P_i \quad i = 1...N \quad$ greifen mit dem folgenden Protokoll auf eine Ressource zu:

- enter(): Eintritt in den kritischen Bereich (ggf. blockiert)
- exit(): Kritischen Bereich verlassen

Anforderungen an einem Algorithmus zur Realisierung des wechselseitigen Ausschlusses:

1. Sicherheit: Höchstens ein Prozess ist zu einer Zeit in einem kritischen Bereich.
2. Liveness: Anforderungen in dem kritischen Bereich einzutreten sind irgendwann erfolgreich.
3. Reihenfolgeforderungen: Reihenfolge des Eintretens darf nicht gegen die "$\rightarrow$"-Relation zwischen Anforderungen verstossen. *Diese Anforderung ist schwer zu erfüllen, daher wird diese vorerst weggelassen.*

Bewertung und Messung der Leistungsfähigkeit von Algorithmen in verteilten Systemen:

- "Verbrauchte Bandbreite" (Anzahl der Nachrichten pro enter/exit-Operationen)
- "Client-Verzögerung" bei jeder enter/exit-Operation.
- "Systemdurchsatz": Wieviele Vorgänge erreicht man pro Zeiteinheit.

Beispiel 1

...für ein Verfahren zur Sicherung des "wechselseitigen Ausschlusses".

Algorithmus mit zentralem "Token"-Server. Ein Zentralserver vergibt "ein Token" an einen Client, der Zugriff auf die Ressource haben will. Der Client, der das Token besitzt, darf die Ressource benutzen. Nach der Benutzung gibt der Client das Token an den Zentralserver zurück.

- Annahme: Der Nachrichtenaustausch ist fehlerfrei. Wechselseitiger Ausschluss wird durch die Einmaligkeit des Tokens gewährleistet.
- Sortiere die eingehenden Anfragen nach Tokens in FIFO-Ordnung. Dann bekommt jeder Prozess der anfragt irgendwann den Zugriff weil in der Warteschlange nur endlich viele Prozesse vor ihm stehen und jeder von diesen das Token nach Benutzung der Ressource zurückgibt.
- Dabei gehen wir davon aus, dass Prozesse nie abstürzen.
- Aufwand zum Eintritt in den kritischen Bereich: 2 Nachrichten für den Eintritt (Anfrage und Übermittlung des Tokens).
- Beim Verlassen: Eine Nachricht: Rückgabe des Tokens.
- Wenn der Eintritt unmittelbar erfolgreich ist, dauert das nur eine Round-Trip Zeit.
- Nachteil des Verfahrens ist die starke Abhängigkeit vom Zentralserver der eventuell durch viele Anfragen überlastet wird.

Alternative ohne zentralen Server: Token-Ring

- Jeder Teilnehmer, der das Token besitzt, aber nicht benötigt, schickt es weiter. Das Token wird von allen "in die gleiche Richtung" weitergeschickt, damit jeder das Token irgendwann erhält.
- Vorsicht: Nachteil ist, dass ständig Nachrichten versandt werden, auch wenn niemand in den kritischen Bereich will.
- Verzögerung: Im schlimmsten Fall sind bei N Teilnehmern N Nachrichtenzeiten notwendig.
- Man könnte (z.B. mit einem Multicast) alle Teilnehmer "gleichzeitig" nach dem Token fragen. Im erfolgreichen Fall ist das schnell!

- Ausbauen der Idee ⇒ Verfahren von Ricart - Agruala

  Das Verfahren ist relativ schnell und es lassen sich zusätzlich Reihnfolgengarantien erbringen, aber man braucht Multicasts. ⇒ Hohes Nachrichtenaufkommen

- Anderes Konzept: Wahl-Algorithmus von Maekawa. Idee:

  1. Jeder Prozess P_i erhält eine Wählermenge V_i und zwar, wenn alle in V_i enthaltenen Prozesse zustimmen, bekommt P_i das Zutrittsrecht.
  2. Jeder Prozess darf nur eine Stimme abgeben.

  Wie sehen die Wählermengen aus:

  $V_i = \{P_1, ..., P_N\}$

  Und für alle $i, j = 1...N$ gilt:

  1. $P_i \in V_i$
  2. $V_j \cap V_i \neq \{\}$ Je zwei Wählermengen enthalten mindestens ein gemeinsames Element!!!
  3. $|V_i| \approx k$ d.h. alle Wählermengen haben (ungefähr) die gleiche Grösse "Fäirness"
  4. Jeder Prozess P_j ist in M der Wählermengen enthalten. "Gleich-Wichtigkeit aller Prozesse.

  Frage: Warum ist wechselseitiger Ausschluss gesichert?

  Annahme: Zwei Prozesse P_a und P_b haben beide das Zugriffsrecht erhalten.

  ⇒ P_a und P_b haben die jeweils notwendigen Stimmen aus V_a und V_b erhalten.

  ⇒ (Wegen 2) ein Prozess (der in V_a und V_b enthalten ist) hat 2 Stimmen abgegeben ⇒ Widerspruch, *wechselseitiger Ausschluss ist gesichert.*

  Maekawa: Es gilt $k \approx \sqrt{N} \quad k \approx M$

  Berechnung der optimalen Wählermengen ist sehr schwer!

  Es geht einfach: $|V_i| \approx 2\sqrt{N}$

  Annahme N ist eine Quadratzahl z.B. $N = 9$.

1	2	3
4	5	6
7	8	9

$V_1 = \{1, 2, 3, 4, 7\}$
$V_2 = \{1, 2, 3, 5, 8\}$
$V_3 = \{1, 2, 3, 6, 9\}$
$V_4 = \{1, 4, 5, 6, 7\}$
$V_5 = \{2, 4, 5, 6, 8\}$
$V_6 = \{3, 4, 5, 6, 9\}$
$V_7 = \{1, 4, 7, 8, 9\}$
$V_8 = \{2, 5, 7, 8, 9\}$
$V_9 = \{3, 6, 7, 8, 9\}$

Jede Wählermenge enthält $2n - 1$ Elemente $= 2\sqrt{N} - 1$ Elemente.

Jedes Element ist in $2n - 1$ Wählermengen enthalten.

Durch die Konstruktion ist sichergestellt, dass zwei beliebige Wählermengen mindestens 1 gemeinsames Element haben (Schnittpunkte von Zeilen oder Spalten).

Der Algorithmus muß noch "Deadlock-frei" gemacht werden, damit er wirklich anwendbar ist, das geht!

Weil die Wählermengen im Vergleich zu N (Teilnehmerzahl) klein sind, ist das Verfahren relativ schnell. Nachrichtenaufwand $2\sqrt{N}$ (falls $k = M = \sqrt{N}$) nämlicn $\sqrt{N}$ mal Anfrage nach Zustimmung.

Was für Fehler sind problemantisch:

- Verlust von Nachrichten wird nicht toleriert.
- Wenn einzelne Prozesse abstürzen, sind nicht sofort alle betroffen. Manchmal entsteht durch einen Einzelausfall noch kein Gesamtausfall.

Modifikationen um Fehler zu entdecken und zu tolerieren sind oft möglich. Dazu braucht man in einem verteilten System die Möglichkeit, Fehler zu entdecken.

2.16 Fehlerdetektor

Ein Fehlerdetektor ist ein Dienst, der Abfragen beantwortet ob ein bestimmter Prozess fehlgeschlagen bzw. abgestürzt ist! Er liefert die Antwort unverdächtig (unsuspected)

oder verdächtigt (suspected).

Vergleiche Brandmelder:

- Ein Prozess wird als verdächtig eingestellt obwohl er korrekt arbeitet. Das darf ab und zu passieren.
- Ein Prozess arbeitet nicht korrekt, wird er als unverdächtig eingestuft. Das sollte nach Möglichkeit nicht passieren.
- Man kann "Time-Outs" verwenden, um Prozesse als "fehlgeschlagen" zu identifizieren.

Ein weiteres Koordinationsproblem: Wahl genau eines Prozesses. (z.B. um die Rolle eines Koordinators zu übernehmen). Alle Prozesse sind gleichberechtigt.

Üblicher Ablauf:

1. Ein Prozess (oder mehrere) initiiert eine Wahl (Grund könnte z.B. sein, dass ein Teilnehmer festgestellt hat, dass ein zentraler Server, der bisher Koordinator war, abgestürzt ist.)
2. Wahl selbst
3. Verteilung des Wahlergebnisses

Ziel: Fehlertoleranz gegen Absturz von Prozessen. Bei erfolgreicher Wahl: Einigkeit darüber, wer Koordinator ist. Einigkeit darüber, ob die Wahl erfolgreich war.

2.16.1 Teilproblem

Sichere Multicast-Kommunikation (*Beim Verteilen eines Wahlergebnisses soll der Verteiler z.B. selbst nicht abstürzen*).

- Anwendung: Gruppenkommunikation und Koordination.
- Ziel: Auslieferungsgarantie. (*Das ist keine Empfangsgarantie*)

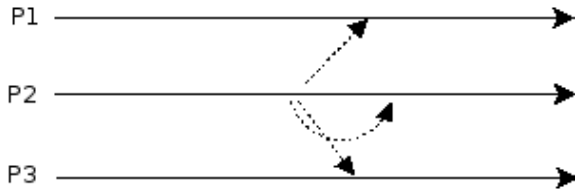
UDP/IP Multicasts erbringen üblicherweise keinerlei Garantie. Sie bieten aber oft die Möglichkeit, Nachrichten relativ schnell an relativ viele Empfänger weiterzuleiten.

Vorteil von Multicasts: Sendende Prozess macht nur eine Send-Operation, die als atomar gilt. Normalerweise kann man Multicasts effizienter realisieren als das Nacheinandersenden mit 1:1 Kommunikation.

Nun versuche ein Verefahren für einen zuverlässigen Multicast anzugeben.

- Voraussetzung: Zu jedem Prozess hat jeder andere Prozess eine zuverlässige 1:1 Verbindung.
- Operationen:
  - **multicast**(g,m) mit
    g := Gruppe in welcher der Multicast stattfindet
    m := Nachricht
  - **deliver**(m) Auslieferung einer Nachricht an einem empfangenden Prozess

    (Vorsicht: Auslieferung erfolgt nicht sofort beim Empfang sondern eventuell später, eventuell gar nicht.)
- Ein Prozess multicasted auch an sich selbstm. D.h. die Multicast-Nachricht wird (im erfolgreichen Fall) auch an den Initiator des Multicasts ausgeliefert.



Grundlegender (Basic Multicast)

wird wie folgt implementiert:

B-multicast(g,m):
  für jeden Prozess $p \in g$

Bei receive(m) in p :
  B-deliver(m) in p

Problem: Es kann sein, dass der Sender abstürzt nachdem einige Send-Operationen schon durchgeführt wurden, und einige noch nicht. Deswegen ist der Basic-Multicast kein zuverlässiger Multicast. Es ist nicht sichergestellt, dass alle Teilnehmer in der Gruppe entweder m ausliefern, oder keiner.

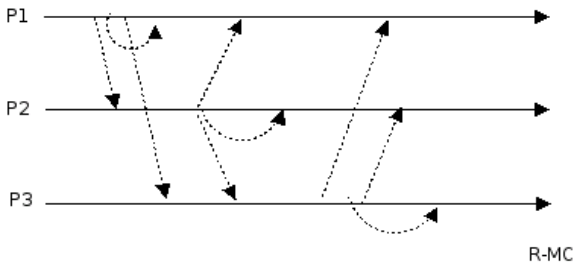
"Zuverlässiger Multicast'

(Reliable-Multicast).

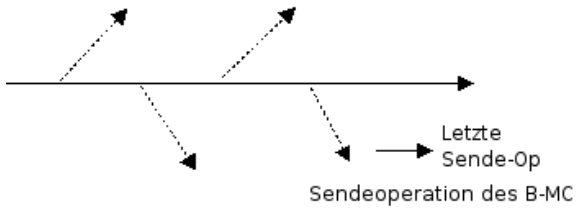
Voraussetzung:

- Integrität: Die ausgelieferte Nachricht ist gleich der Gesendeten. Keine Nachricht wird doppelt ausgeliefert.
- Liveness: Jede Nachricht erreicht ihr Zeil, sofern das Ziel *bzw. der Zielprozess* nicht abgestürzt ist.
- Einigung: Wenn ein Prozess die Nachricht ausliefert, liefern alle korrekten Prozesse die Nachricht irgendwann aus (korrekt heißt hier: Nicht abgestürzt.).
- Einigungsbedingung: "Alles oder Nichts" (*Hier besser formuliert: Alle oder Keiner*)
- Fehlertoleranz: Einzelne Empfänger oder Sender dürfen abstürzen.

Verfahren/Algorithmus siehe Aushändigung Jeder Prozess liefert erst dann aus, wenn er



selbst den B-multicast erfoglreich hinter sich gebracht hat. Der B-Multicast verwendet sichere 1:1 Kanäle. Wenn ein Prozess das Ende seines B-Multicasts erlebt, kann er sicher



sein, dass alle Prozesse die noch weiterleben seine Nachricht noch bekommen haben und diese irgendwann ausliefern, sofern sie nicht vorher abstürzen.

Mit Hilfe eines verlässlichen Multicasts lassen sich viele Koordinationsprobleme Lösen.

Ein weiteres Beispielproblem: Herstellen von Konsens

- Ein oder mehrere Prozesse schlagen Werte vor.
- Die Prozesse einigen sich auf einen dieser Werte.
- Konsens soll selbst dann erzielt werden, wenn Fehler vorkommen.

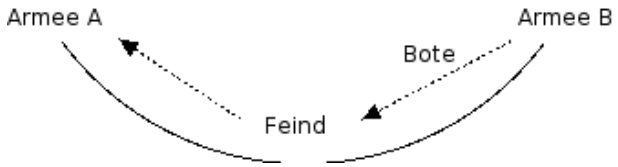
Folgendes Fehlermodell:

- f von N Prozessen arbeiten fehlerhaft (Sind z.B. fehlerhaft programmiert oder betrügen)
- Der Nachrichtenaustausch ist fehlerfrei.
- Das System sei synchron.

Für $N \geq 3f + 1$ ist das Konsensproblem lösbar! Man braucht mindestens $f + 1$ Nachrichtenrunden.

Ein Beispiel für ein "unlösbares Konsensproblem":

- Zwei (befreundete) Armeen müssen sich auf "Angriff" oder "Rückzug" einigen.



- Fehlermodell: Nachrichten können verloren gehen (Der Bote muß durch die feindlichen Linien)
- Annahme: Es gibt ein Protokoll, dass mit M Nachrichten Konsenz herstellt.

  Weil die Nachricht M verloren gehen kann, das Protokoll aber sicherstellt, dass selbst dann Einigung erzielt wird, ist Nachricht M nicht nötig.

  $\Rightarrow$ Es gibt ein Konsensprotokoll mit $M - 1$ Nachrichten

  $\Rightarrow$ Es gibt ein Konsensprotokoll mit 0 Nachrichten

  Das ist ein Widerspruch!

2.17 Verteilte Objekte und entfernter Aufruf

2.17.1 Teilproblem 1: Datentransport zwischen Client und Server

Daten müssen zwischen Client und Server transportiert werden.

- **Daten in Objekten:** Menge verknüpfter Objekte. Attribute von Objekten im Speicher
- **Nachrichten:** Folge von Bytes

Notwendig ist die Umwandlung von "Objekten" in eine "flache Darstellung" (z.B. in eine Folge von Bytes). Verschiedene Rechner/Systeme stellen schon die elementaren Datentypen verschieden im Speicher dar. Die Darstellung im Speicher ist eine zur Übertragung ungeeignete Form.

Generelle Methoden zur Übertragung:

1. Umwandlung in ein externes Format. Umwandlung in dieses Format beim Senden und Empfangen.
2. Werte werden im Format des Senders übertragen. Empfänger wandelt die Daten ggf. um.

Oft müssen mehrere Werte gemeinsam übertragen werden. Das Zusammenfassen zu einer Nachricht ist dann effektiver als die Übertragung in einzelnen Nachrichten. Diese Technik nennt man "Marshalling" (das ist: Die Umsetzung von strukturierten Datentypen in ein externes Format).

Standardisierte Verfahren:

1. Objektserialisierung in Java
2. Datendarstellung in CORBA für Argumente und Ergebnisse entfernter Methodenaufrufe.

   CORBA: **C**ommon **O**bject **R**equest **B**roker **A**rchitecture.

Marshalling wird von der Middleware ausgeführt. Üblicherweise Umsetzung in Binär- oder Oktettdarstellung. (Evtl. XML wandelt in "lesbares ASCII" um).

In CORBA hat man die folgenden Datentypen standardisiert: 15 elemtare Datentypen:

Typ	Größe
short	16 Bit
long	32 Bit
float	32 Bit
double	64 Bit
char	
boolean	
octet	
any	

Zusammengesetzte Typen:

Typ	Beschreibung
Sequence	Länge + Elemente
String	Länge + Elemente
Array	Elemente in der gegebenen Reihenfolge.
	Anzahl der Elemente muß fest und bekannt sein.
Struct	Komponenten hintereinander

In CORBA: Ausrichtung von Teilen auf 32-Bit-Grenzen. Typen werden nicht mit übertragen, weil Sender und Empfänger sich bei jeder Nachricht einig sind, welche Daten ausgetauscht werden. Die Spezifikation von Datentypen für den Austausch erfolgt in der **I**nterface **D**efinition **L**anguage (IDL).

Beispiel:

```
struct Person {
    string name;
    string place;
    long year;
};
```

Man verwendet einen Schnittstellencompiler. Dieser erzeugt aus den Beschreibungen der Datentypen in der IDL die Operationen für Marshalling und Unmarshalling.

Achtung: Auch Verweise (Referenzen) auf Objekte treten als Daten auf. D.h. für entfernte Objekte muß es "eindeutige Bezeichner" geben. Referenzen müssen unabhängig von Adressräumen und Implementationen weitergegeben werden können.

Java Objektserialisierung: Klasseninformationen, Methoden usw. können ggf. mit übertragen werden oder müssen dem Empfänger zur Verfügung stehen. Aufbau ist re-

kursiv.

Zwei Sichtweisen aufs Programmieren:

1. Objektorientiertes Programmiermodell

   Um dieses Modell auf verteilte Systeme zu erweitern verwendet man üblicher Weise so etwas wie **R**emote **M**ethod **I**nvocation (RMI) *(Achtung: 2 Arten von RMI, einmal das was Java macht und einmal das Konzept an sich selbst).*

   Sichtweise: "Methoden ändern Attribute"

2. Ereignisorientiertes Programmiermodell

   Objekte erhalten Nachrichten über die Ereignisse von entfernten Objekten. Das "System" selbst soll Ereignisse herstellen können und dann Benachrichtigungen erzeugen.

   Mit so einem Konzept kann man dann ereignisorientiert programmieren. Man will auch in verteilten Systemen ereignisbasiert programmieren können.

Für beide Fälle benötigt man eine Schnittstellendefinition, in welcher nicht nur die Datentypen festgelegt sind, sondern auch die "Parameter" und "Ergebnisse" *(In Anführungszeichen weil von verschiedenen Programmiersprachen diese Begriffe anders verwendet werden)* und die Methodenaufrufe selbst.

Methoden werden durch ihre Signatur festgelegt. Die Schnittstelle muß Implementationsunabhängig festgelegt werden (Ausser man bleibt in einer Sprache). Dafür wird wieder die IDL verwendet.

Beispiel:

```
struct Person {
    string name;
    string place;
    long year;
};

interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p);
    void getPerson(in string name, out Person p);
```

```
    long number();
};
```

Parameter werden als “Input” oder “Output” deklariert, damit klar ist, wann Werte zwischen Aufrufer und Aufrufenden zu transportieren sind. Zeiger (als Referenz in Adressraum) dürfen nicht als Parameter verwendet werden. Das ist sinnvoll, weil Adressen nur in einem Adressraum interpretierbar sind.

Man kann üblicherweise nicht direkt auf Attribute von entfernten Objekten zugreifen, sondern nur mit GET- oder SET-Methoden.

Man will Referenzen auf Objekte übergeben können. Idee: Verwende global eindeutige Bezeichner für Objekte.

32 Bit	32 Bit	32 Bit	32 Bit	
Internetadresse	Portnummer	Zeit	Objektnummer	Schnittstelle des entfernten Objekts

Aufrufsemantik

- Vielleicht
- Mindestens einmal
- Höchstens Einmal
- JAVA und CORBA: Hächstens einmal
- SUN-RPC: Mindestens einmal

2.17.2 Begriff der “idempotenten Opertation”

Eine Operation heißt idempotent wenn die zweimalige Ausführung dasselbe bewirkt, wie die einmalige Ausführung.

- Implementierung von RMI: Welche Teilprobleme werden gelöst?
- Entferntes Referenzmodul: Umsetzung lokaler Referenzen in entfernte Referenzen mittels einer Tabelle entfernter Objekte.
- Proxy: Spiegelt eine “lokale Kopie”des entfernten Objekts vor.
- Das Proxy-Objekt verbirgt die Objekt-Referenz-Umsetzung.
- Das Marshalling und Un-Marshalling: Senden und Empfangen von Nachrichten

- SKELETON: Implementiert die Methoden der entfernten Schnittstelle

Die SKELETON-Methoden werden üblicherweise nur einmal pro Klasse implementiert, und ein DISPATCHER verteilt die entfernten Aufrufe an die richtigen Objekte der Klasse.

Wie werden die Klassen für PROXY und SKELETON erzeugt? Dafür gibt es Schnittstellencompiler!!!

Entfernte Aufrufe werden üblicherweise von eigenen Threads abgearbeitet, damit die Benutzung der entfernten Schnittstelle nicht zu zu starker Beeinträchtigung beim Besitzer des benutzten Objektes führt.

Implementierung von entfernten Objekten müssen THREAD-SAFE sein. Ein entferntes Objekt steht nur in einem Prozess zur Benutzung zur Verfügung. Damit nicht ständig für alle entfernten Objekte die zugehörigen Prozesse laufen müssen, erlaubt man "passive Objekte" die von einem Aktivator gestartet werden.

Persistente Objekte: Die Existenz ist gesichert über die Beendigung von Prozessen hinweg. In CORBA gibt es den Persistant Object Service. Der benutzt Festplatten für die Zwischenspeicherung. In verteilten Systemen muß man Objekte mit verteiltem Garbage-Collection aufräumen.

Ereignisse und Benachrichtigungen

Ziel: Objekt reagiert auf Änderungen in einem andern Objekt.

Die Situation ist durch folgende Eigenschaften oft gekennzeichnet: Benachrichtigungen sind typischerweise asynchron.

Man weiß nie, wann welches Ereignis stattfind bzw. wann es gemeldet wird. Heterogenität (Einsatz verschiedenster Systeme) ist erwünscht.

Modellvorstellung: Publish-Subscribe Modell

- Publisher: In Objekten geschehen Ereignisse. Die Objekte geben bekannt, dass sie über diese Ereignisse benachrichtigen können.
- Subscriber: Objekte geben bekannt, dass sie über bestimmte Ereignisse benachrichtig werden wollen.

Ereignisse haben Attribute

- Namen des Erzeugerobjekts
- Operation die das Ereignis auslöste
- Parameter solcher Operationen
- Zeit

Konzeptidee:

- Trenne den Publisher und den Subscriber um getrennte Entwicklung zu ermöglichen.
- Begrenze dadurch auch die Arbeit, die die Subscriber für den Publisher bedeuten.
- Richte einen Ereignisdienst ein mit Beobachterobjekten.
- Bei Benachrichtigungen kann man verschiedene Garantieanforderungen stellen.
- Realisierung mit IP-Multicast.
- Keinerlei Auslieferungs- oder Reihenfolgegarantie.
- Beim RELIABLE-Multicast wäre eine Alles- oder Nichts-Semantik realisiert.
- Eventuell will man Echtzeitanforderungen erfüllen.
- In synchronen Systemen gibt es Multicast-Protokolle mit Echtzeitgarantie.
- Zuverlässigkeit und Reihenfolgegarantie. Damit kann man Garantien in ereignisbasieren System erbringen.

Beobachter haben verschiedenste Aufgaben:

- “Weitergabe”, eventuell mit Garantie
- “Filterung”: Nur die “interessanten” Nachrichten werden weitergeleitet
- “Mustererkennug”: Kombinationen von Ereignissen führen zur Benachrichtigung
- “Mailbox”: Abholdienst, falls der Subscriber selten aktiv ist.

2.17.3 Das verteilte Objektmodell

- **Entfernte Objektreferenz:** Objekte können Methoden eines entfernten Objektes aufrufen, wenn sie eine Objektreferenz auf dieses Objekt besitzen.
- **Entfernte Schnittstelle:** Jedes entfernte Objekt hat eine entfernte Schnittstelle, die angibt, welche seiner Methoden aufgerufen werden können.

(Siehe Aushändigungen)

Entfernte Objektreferenzen:

- Identifikation, die innerhalb eines verteilten Systems ein entferntes Objekt eindeutig identifiziert.
- **Achtung:** Man will Objekte eventuell verschieben, ohne dass man die Objektreferenzen ändert.
- **Achtung:** Man muß davon ausgehen, dass entfernte Methodenaufrufe fehlschlagen (z.B. Netzwerkausfall, Serverausfall). Deswegen sollten Aufrufer (Clients) in der Lage sein, Exceptions zu verarbeiten die im Fehlerfall auftreten.

Probleme beim entfernten Methodenaufruf:

- Anforderungsnachrichten können verloren gehen.

  A) Wiederhole Anforderungsnachrichten

- Anforderungen an den Server gehen mehrfach ein.

  B) Filter Duplikate aus

- Ergebnisnachrichten gehen verloren.

  C) Speichere die Folge der Ergebnisnachrichten und wiederhole ggf. verlorene Nachrichten, ohne dass die eigentliche Operation auf dem Server ein zweites Mal gemacht wird.

Aufrufsemantik:

- Vielleicht: Weder A noch B noch C
- Mindestens einmal: A B nicht C nicht

  (Es wird schon auf eine Ergebnisnachricht gewartet, zumindest eine Zeitlang)

- Höchstens einmal: A und B und C

  Es wird davon ausgegangen, dass der Server abstürzen kann. Daher kann ein entfernter Methodenaufruf keine "garantiert genau einmal" Forderung erfüllen.

2.18 Verteilte Dateisysteme

Verteilte Dateisysteme bilden die Grundlagen für viele andere verteilte Dienste.

Anforderungen:

- **Zugriffstransparanz:** Client Programme müssen nicht darauf Rücksicht nehmen, dass sich die benutzten Dateien in einem verteilten Dateisystem befinden. Sie benutzen die gleichen Aufrufe.
- **Orts- und Mobilitätstransparenz:** Der Ort, an welchem Daten physikalisch liegen, darf sich ändern, ohne dass Benutzer darauf Rücksicht nehmen müssen. Die Benutzer dürfen ihren Ort auch ändern.
- **Skalierungstransparenz:** Inkrementelles Wachstum erlaubt effektive Behandlung größerer Datenmengen, großer Benutzermengen, u.s.w.

Nebenläufige Benutzung von Dateien soll möglich sein. Eventuell will man sogar nebenläufige Änderungen von Dateien. (Oft wird kein globales hartes Sperrkonzept eingesetzt, weil die Performance dann zu stark leidet!).

- **Replikation** oder teilweise Replikation soll möglich sein (Caching).
- **Heterogenität** von Hardware und Betriebssystemen soll unterstützt werden.
- **Fehlertoleranz**, d.h. z.B. Verlust von Nachrichten oder Absturz von Prozessen soll toleriert werden.

  Verwende z.B. idempotente Dateioperationen. Verwende "zustandslose" Server. Die Idee dabei ist, dass ein Server sich nach Absturz neu starten läßt, und sich danach so verhält als sei er nicht abgestürzt. Realisierung mit LOG-Dateien und COMMIT-Records.

- **Konsistenz:** Eine Forderung könnte sein: Ein-Kopie-Aktualisierungssemantik: "Alle Clients sehen das, was sie sehen würden, wenn es die Datei nur einmal gäbe."

  Problem: Änderungen von Dateibereichen müssen in allen Repliken gemacht werden.

- **Sicherheit und Zugriffskontrolle:** Authentifizierung, Signaturen, Verschlüsselung
- **Effizienz:** Es sollen die gleichen Dienste erbracht werden, wie in einem klassischen Dateisystem. Man möchte ein hohes Maß an Leistung (Durchsatz).

Verteilte Dateisysteme werden oft mit Hilfe eines "Flachen Dateidienstes" implementiert. Verwendung einer einfachen Schnittstelle mit wenigen der definierten Operationen.

- Der flache Dateidienst verwendet UFID (Unique File Identifier) innerhalb des verteilten Dateisystems.
- Die Operationen (READ, WRITE, CREATE, DELETE, GETATTRIB Und SETATTRIB) werden mit Hilfe von Remote-Procedure-Calls implementiert.
- Um Dateinamen verweden zu können wird zum flachen Dateidienst ein Verzeichnisdienst hinzugefügt. Dieser realisiert die Umsetzung von Namen $\leftrightarrow$ UFID.
- Das Client-Modul stellt den Applikationen die übliche Benutzung (API) zur Verfügung.

Zu den Operationen mit Ausnahme von CREATE sind die Operationen des flachen Dateidienstes idenpotent spezifiziert. Die Verwendung der "Mindestens Einmal" Semantik für die Implementation der RPC ist hinreichend.

Selbst CREATE ist nicht kritisch. Jeder Aufruf erzeugt zwar eine neue "Datei" (UFID) aber die Eindeutigkeit ist gewährleistet. Eventuell muß der Server manchmal ungenutzte UFIDs aufräumen.

Server können leicht zustandslos implementiert werden. Ein Server heißt zustandslos, wenn er nach Absturz problemlos wieder gestartet werden kann.

Vorsicht: Das klassische UNIX-Dateisystem entspricht diesem modell in wesentlichen Punkten nicht!

- "OPEN" und "CLOSE" in UNIX erfordern einen Zustand.
- Bei "READ" und "WRITE" Operationen gibt es eine File-Position, die einen Zustand entspricht.
- READ und WRITE Operationen in UNIX sind nicht idenpotent.

Zugriffskontrolle erfolgt üblicherweise auf dem Server. Sicherung gegen gefälschte Identifikationen und gefälschte UFIDs ist nötig. Das Ergebnis einer Zugriffsüberprüfung wird dem Client mitgeteilt. Das Ergebnis wird nicht beim Server gespeichert weil sonst die Zustandslosigkeit schwer zu realisieren wäre.

Eine Methode wäre:

- Bei der Umwandlung Name → UFID wird an den Client zurückgegeben, was der Aufrufer alles mit der Datei darf.

Die Schnittstelle zum Verzeichnisdienst:

- Alle Methoden sind idenpotent spezifiziert.
- Um hierarchische Dateisysteme abzubilden, benutzt man mehrfach die Umsetzung Name → UFID.
  - abc/xyz/java.com
  - "abc" → UFID(abc)
  - Dann ist UFID(abc), suche nach xyz
  - Dann ist UFID(abc/xyz), suche nach java.com

2.18.1 Eine Beispimplementation (NFS)

...für ein verteiltes Dateisystem: SUN-NFS (Siehe ausgehändigtes Blatt)

Ein mögliches NFS-Szenario

- T READ soll heißen in der Datei wird das Zeichen an der ersten Position gelesen. Der Aufruf erfolgt zur Zeit T.
- T WRITE(C) Ein Client ruft zur Zeit T die Schreiboperation auf, die Zeichen C an die erste Position der Datei schreibt.
- OPEN und CLOSE teilt dem lokalen Dateisystem das übliche mit

Blöcke im Client-Cache werden nur beim COMMIT bzw. hier "CLOSE" zum Server übertragen. Verwende $t = 5s$ als Aktualisierungsschranke.

(C, t_m) auf dem Server heißt: Dateiinhalt ist C und wurde zur Zeit t_m auf dem Server das letzte Mal geändert.

2.18.2 Ein weiteres verteiltes Dateisystem (AFS)

AFS: Andrew-File-System

- Gesamte Dateien werden vom Server zum Client übertragen und dort gecached. Der Cache im Client ist permanent, d.h. er überlebt den Absturz oder Neustart eines Client-Computers. Die Idee ist, dass so vielleicht viele Dateien benutzt werden können, ohne dass weitere Netzlast entsteht.

- Bei "Open" wird festgestellt, ob man mit einer lokalen Kopie arbeiten kann.
- Wie wird die Aktualität von Kopien gewährleistet?? Wie nahe kommt man an die Ein-Kopie-Aktualisierungssemantik?
- Bei der Übertragung einer Kopie vom Server zum Client stellt der Server dem Client ein Callback-promise-Token zur Verfügung. Der Server "verspricht" dem Client, ihn über Aktualisierungen zu informieren.
- Zustand des promise-Tokens auf dem Client
  - "VALID" bei Zustellung
  - "CANCELLED" wird gesetzt, wenn der Server über eine Aktualisierung berichtet

Weiteres siehe Ausgehändigtem. Falls bei "OPEN" das Token den Wert "Cancelled" hat, wird eine neue Kopie geholt. Bei Neustart des Clients muß er für alle Dateien im Cache prüfen, ob er CALLBACKs verpaßt hat. D.h. die Callback-promises müssen aktualisiert werden. Das schützt gegen den Verlust von Callback-Nachrichten. Auf diese Weise kann die Ein-Kopie-Aktualisierungssemantik nicht garantiert werden. Aber man versucht, sich ihr zu nähern bei gleichzeitig sehr niedriger Netzlast (Netzlast nur bei OPEN und CLOSE*, viel niedriger als bei NFS*).

Erstaunlicherweise verhält sich das AFS in der Praxis relativ gut. Warum ist das so?

- Weil viele Dateien relativ klein sind, kann der Client tatsächlich ganze Dateien cachen.
- Die meisten Dateien werden nur gelesen und sehr sehr selten geändert (Performance besser als NFS wegen geringer Netzlast).
- Dateien die sowohl viel gelesen als auch geschrieben werden, werden oft nur von einem Benutzer benutzt (Performance besser als NFS wegen geringer Netzlast).
- Dateien werden oft gehäuft hintereinander von einem Benutzer (auf einem Client) benutzt.

Problem: Datenbanken passen überhauptnicht in dieses Bild. Bei Datenbanken sind Korrektheit und nebenläufige Benutzung unverzichtbar.

2.19 Transaktionen und Nebenläufigkeitskontrolle

Verwende folgende Sichtweise:

- Ein Server stellt mehrere Objekte bereit.
- Eine Transaktion wird von einem Client angefordert (spezifiziert) und beeinflußt im allgemeinen mehrere Objekte.

Vom Server soll die folgende Garantie erbracht werden:

- Die gesamte Transaktion wird durchgeführt...
- ...oder die Transaktion hat keine Wirkung (und es wird dem Client mitgeteilt, dass die Transaktion fehlschlug)

Transaktionen sollen nebenläufig ausgeführt werden! Damit Abstürze des Servers toleriert werden, werden alle Objekte presistent (dauerhaft) gespeichert. Clients fordern, dass eine Folge von Schritten, die eine Transaktion bilden, atomar ausgefürt werden. Strikte Hintereinanderausführung ist zu ineffektiv: Man möchte die Teilschritte verzahnen.

Isolationsbedingung: Jede Transaktion ist so durchzuführen, dass sie durch andere, nebenläufig ablaufende Transaktionen, nicht gestört wird. Zwischeneffekt einer Transaktion sind unsichtbar für alle anderen Transaktionen.

Es gibt zwei Probleme bei nebenläufiger Ausführung *(siehe Aushändigungen: Lost Update, Problem der inkonsistenten Abrufe).*

Üblicherweise fordert man von einem Transaktionssystem die "serielle Äquivalenz" wenn man mehrere Transaktionen ausführt. Das Ergebnis einer nebenläufigen verzahnten Ausführung wird als korrekt betrachtet, falls es eine Hintereinanderausführung der Transaktion gibt, die zum selben Ergebnis führt.

Beispiel:

Schritt	**Transaktion T**	**Schritt**	**Transaktion U**
T1	x = read(i)	U1	y = read(j)
T2	write(i,10)	U2	write(j,30)
T3	write(j,20)	U3	z = read(i)

Am Anfang sei der Wert von i = 1 und j = 2

Reihenfolge T dann U:

Schritt	Transaktion	Wert
T1	x = read(i)	x = 1
T2	write(i, 10)	i = 10
T3	write(j, 20)	j = 20
U1	y = read(j)	y = 20
U2	write(j,30)	j = 30
U3	z = read(i)	z = 10

Gesamtwirkung: i = 10, j = 30, x = 1, y = 20, z = 10

Reihenfolge U dann T:

Schritt	Transaktion	Wert
U1	y = read(j)	y = 2
U2	write(j,30)	j = 30
U3	z = read(i)	z = 1
T1	x = read(i)	x = 1
T2	write(i, 10)	i = 10
T3	write(j, 20)	j = 20

Gesamtwirkung: i = 10, j = 20, x = 1, y = 2, z = 1

Eine nicht seriell äquivalentes Interleaving:

Schritt	Transaktion T	Schritt	Transaktion U	Wert
T1	x = read(i)			x = 1
T2	write(i, 10)			i = 10
		U1	y = read(j)	y = 2
		U2	write(j,30)	j = 30
T3	write(j, 20)			j = 20
		U3	z = read(i)	z = 10

Gesamtwirkung: i = 1, j = 20, x = 1, y = 2, z = 10

Wie kann man sicherstellen, dass die Verzahnung, die man wählt, seriell äquivalent ist? Idee: Konflikterzeugende Operation.

Ein Operationspaar erzeugt einen Konflikt $\Leftrightarrow$ Die kombinierte Wirkung hängt von der Reihenfolge ab.

Beispiele:

- read(a); read(a) kein Konflikt
- read(a); write(a,5) Konflikt
- write(a,5); write(a,10) Konflikt

Wenn man Transaktionen so ausführt, dass alle konflikterzeugenden Operationspaare in der gleichen Reihenfolge ausgeführt werden (d.h. wenn man einmal zuerst die Operation von Transaktion U gemacht hat, dann bei Konflikt immer erst die Operation von U) dann ist die Ausführung seriell äquivalent.

Wie sorgt man dafür, dass nicht gegen Konfliktregeln verstoßen wird:

- Kontrolliere den Zugriff auf Objekte durch Sperren! (LOCKs)

Vorsicht: Man darf Sperren nicht zu früh freigeben. Eine korrekte Vorgehensweise ist das 2-Phasen-Sperren.

- Sammelphase: Sperren anfordern
- Freigabephase: Sperren freigeben

Probleme bei Sperren:

- Deadlocks
- Hocher Verwaltungsaufwand
- Geringer Durchsatz

Deadlocks sind hier nicht so problemantisch, weil das Transaktionssystem die Möglichkeit hat Transaktionen abzubrechen!

Idee: Mache bereits eingetretene Wirkung rückgängig. Aber, der Abburch von Transaktionen kann zu zusätzlichen Problemen führen: *Siehe Aushändigung "Dirty-Read".*

- Sperren müssen gehalten werden, bis "COMMIT" oder "ABORT" feststeht.
- Anderenfalls kann das Rückgängigmachen, da dabei "WRITE-Operationen" vorkommen, zu Problemen führen.

Da das strikte Vermeiden von Konflikten kann die Performance stark herabsetzt. Man kann eine andere Vorgehensweise verwenden: "Optimistische Vorgehensweise". Führe die Transaktionen an Versuchsszenarien der Objekte aus. Danach prüfte auf Konfliktfreiheit und schreibe erst dann fest (Festschreiben entspricht COMMIT)

Bisher lagen alle Objekte, die von einer Transaktion betroffen wurden, auf einem Server. *Bei den verteilten Transaktionen sieht dies anders aus.*

2.19.1 Verteilte Transaktionen

Die von einer Transaktion betroffenen Objekte werden von mehreren Servern verwaltet. Eventuell kann eine Transaktion auch Unter-Transaktionen starten *(siehe Aushändigung "Verteilte Transaktionen").*

Problem:

- Die Atomarität einer Transaktion (alles oder nichts) muß gesichert bleiben.
- Server können abstürzen!
- Nachrichten können verloren gehen!
- Wichtigster Punkt: Es muß Einigkeit hergestellt werden, ob
  - festschreiben...
  - ...oder abbrechen

Lösungsansatz: Atomare Commit Protokollei mit Hilfe eines Koordinators. (Vorher eventuell Wahl eines Koordinators)

Aufgaben des Koordinators:

- Beim Start einer Transaktion
  - Client teilt dem Koordinator mit, dass er eine Transaktion startet.
  - Der Koordinator gibt dem Client eine TID (Transaktionsidentifikation) zurück und startet die Transaktionsverwaltung. Die TID ist eindeutig im verteilten System.
- Während der Ausführung
  - Jeder Server, der ein Objekt verwaltet, das von einer Transaktion betroffen ist, meldet dem Koordinator, dass er an der Transaktion beteiligt ist ("JOIN"-Operation).

Koordinator fürt für jede Transaktion eine liste aller Teilnehmer. Jeder Teilnehmer hat eine Referenz auf den Koordinator. Alle diese Listen sind permanent (absturzgesichert) gespeichert!

Wenn die Transaktion beendet werden soll:

- Client fordert mit "CLOSE-TRANSACTION" den Koordinator auf, das Commit-Protokoll auszuführen.

- Eventuell gibt es schon vorher irgendeinen Teilnehmer der "ABORT-TRANSACTION" vom Koordinator anfordert.

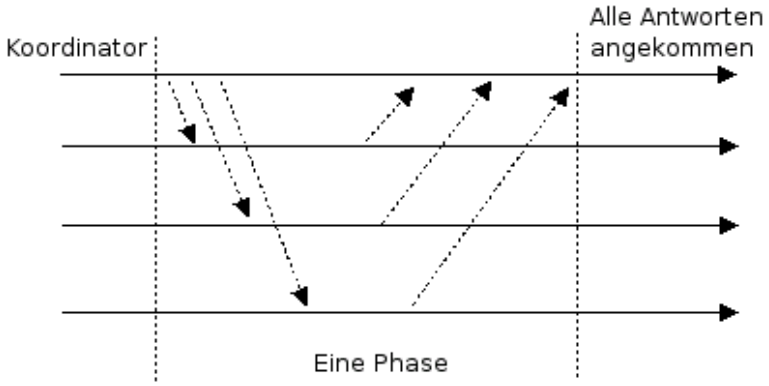
(Siehe Aushändigung: "Eine verteilte Banking-Transaktion").

2.19.2 Commit-Protokolle

Ein-Phasen Commit-Protokoll

Ein einfaches (unzureichendes) Ein-Phasen Commit-Protokoll:

- Koordinator entscheidet und fordert alle Teilnehmer auf, die Transaktion festzuschreiben oder abzubrechen.
- Anforderung wiederholen, bis eine Bestätigung von allen Teilnehmern erhalten ist. Problem: Wenn ein Server nicht festschreiben kann, kann er bei diesem Protokoll keinen Abbruch veranlassen! Gründe dafür können sein: Die lokale Nebenläufigkeitskontrolle hat einen Deadlock festgestellt.



Beim optimistischen Verfahren kann es sein, dass konflikterzeugende Operationen den Abbruch als Konfliktlösung verlangen. Auch das Erkennen eines Serverabsturzes kann nicht verwendet werden.

Abhilfe: Zwei-Phasen Commit-Protokoll

Zwei-Phasen Commit-Protokoll

Anforderungen:

- Wenn ein Teilnehmer Abbruch verlangt, muß abgebrochen werden.

- Wenn Konsens über Festschreiben besteht, müssen alle festschreiben.
- Wenn Server abstürzen, soll das die Atomarität nicht gefährden.

Phasen:

1. **Phase (Abstimmungsphase):** Unter den Teilnehmern wird per Abstimmung festgestellt, ob festgeschrieben oder abgebrochen werden soll.
   - Teilproblem: Stimmt ein Teilnehmer für "festschreiben", muß er in der Lage sein festzuschreiben. Er darf noch nicht festgeschrieben haben.
   - Vorgehensweise: Bevor ein Teilnehmer mit "festschreiben" antwortet, sichert er seine Stimme im permanenten Speicher zusammen mit den festzuschreibenden Werten.

     Die Transaktion hat bei diesem Teilnehmer den Zustand "vorbereitet zum Festschreiben" ("PREPARED").
2. **Phase (Mitteilungsphase):** Der Koordinator teilt allen Teilnehmern das Ergebnis der Abstimmung mit. Die Teilnehmer führen es aus *(festschreiben oder abbrechen)*.

Fehlermodell:

- Verlust von Nachrichten
- Absturz von Servern
- Server können neugestartet werden und ihre Aufgabe mithilfe des permanenten Speichers fortführen
- Keine Zeitschranken

Weiteres siehe Aushändigungen. Das Protokoll hat ein Problem, wenn Teilnehmer, die im unsicheren Zustand sind, den Koordinator nicht erreichen. Eventuell kann man anderen Teilnehmer fragen. Falls ein Teilnehmer die Entscheidung des Koordinators kennt, darf er sie anderen mitteilen.

2.19.3 Bemerkungen zur Nebenläufigkeitsktontrolle bei Verteilten Systemen

Jeder Server fürt für seine Objekte lokal eine Nebenläufigkeitsktontrolle durch (d.h. lokal erreicht man serielle Äquivalenz).

$\alpha : a, B \qquad \beta : b, B$

Das reicht nicht aus, um global serielle Äquivalenz zu sichern.

Eine Möglichkeit: Benutzung von Sperren im gesamten verteilten System. Vorsicht: Jetzt ist es viel schwieriger Deadlocks festzustellen, weil man nicht so leicht eine Momentaufnahme (Snapshot) machen kann um die "Situation zu einem Zeitpunkt" einzufangen.

Es kann zu "Phantom Deadlocks" kommen, das sind Deadlocks, die in Wirklichkeit nicht da sind.

Die Probleme führen dazu, dass man ungern globale Sperren verwendet. Besser: Global optimistische Vorgehensweise. Ein Koordinator ist für die Feststellung und Lösung von Konflikten verantwortlich.

2.20 Replikation

- Aufgabe: Hohe Verfügbarkeit von Daten und Leistungsverbesserung.
- Ansatz: Daten werden auf mehreren Servern gehalten und zur Verfügung gestellt.
- Forderung: Replikationstransparenz. Die Clients sollen nicht erkennen, dass mehrere physikalische Kopien existieren. Jedes Objekt gibt es als "logisches Objekt" nur einmal.

Wenn man Dienste repliziert, muß die spezifizierte Korrektheit für den reguliertem Dienst weiter eingehalten werden. (Was immer das heißen mag!)

Systemmodell asynchrones System:

- Nur Prozessabstürze werden berücksichtigt.
- Keine Netwerkunterbrechungen und kein Nachrichtenverlust.

Replikenmenge (RM) verwalten eine einzelne Replik. Die RM führen alle Operationen "wiederherstellbar" aus. D.h. die Repliken überstehen Abstürze.

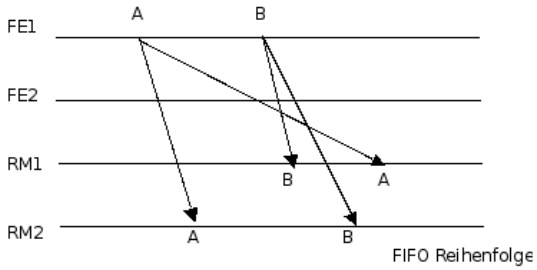
Anforderungen von Clients werden über ein Frontend (FE) an die RM geleitet. Dabei sind die FE zusamemen mit den RM dafür verantwortlich, Replikationstransparenz zu gewährleisten.

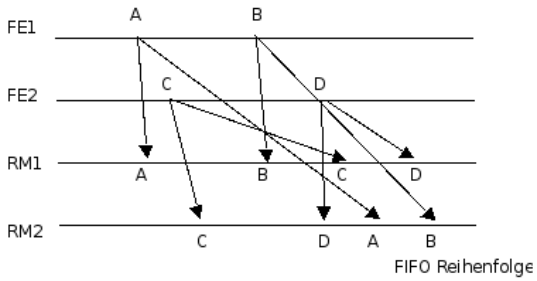
Eine Replikation wird in 5 Phasen unterteilt:

1. Das FE sendet die Anforderung an ein oder alle RM.

2. Koordination: Die RM koordinieren die Reihenfolge, in der die Anforderungen ausgeführt werden (oder sie einigen sich auf Abbruch).

   An die Reihenfolgen stellt man oft gewisse Anforderungen. Eine Forderung ist z.B. die Forderung, dass FIFO Reihenfolge eingehalten wird. Genauer heißt das: Wenn ein Frontend Anforderung r vor r' absetzt, verarbeitet jeder korrekte RM r vor r' sofern er r' verarbeitet. Vorsicht: Die FIFO-Garantie sichert noch keine





   Ein-Kopie-Aktualisierungs-Semantik.

3. Die RM führen die Anforderungen aus (ggf. versuchsweise).

4. Einigung: Die RM einigen sich darüber, welche Anforderungen festgeschrieben werden. (Und es wird ggf. festgeschrieben).

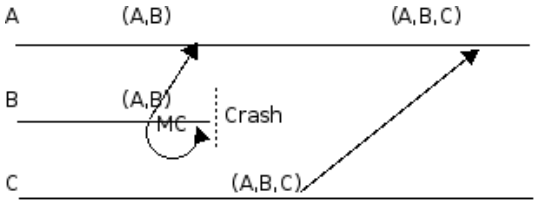
5. Antwort: Ein oder mehrere RM antworten dem Frontend!

Eventuell wird dem FE schon früher mitgeteilt, ob seine Anforderung ausgeführt wird oder nicht.

Damit alle RM wissen, wer bei Koordinationsvorgängen beteiligt ist, wird ein Gruppenmitgliedsschaftdienst eingeführt! Beitrit und Austritt von Mitgliedern und Gruppenkommunikation werden damit realisiert.

Eine Aufgabe des Gruppenmitgliedsschaftdienstes ist es “Gruppenansichten” zu verwalten und zu verteilen.

Gruppenansicht (VIEW) ist eine Liste der aktuellen Gruppenmitglieder. Identifikation von Mitgliedern durch eine global eindeutig Prozessdienstidentifikation. Beim Aus- oder Eintritt von Mitgliedern wird eine neue Ansicht erzeugt. *Weiteres siehe Aushändigung.*



2.20.1 2 Arten Daten bzw. Dienste zu replizieren

1. Art: Passive (Primär/Backup)-Replikation. (Siehe Aushändigungen).
2. Art: Namensdienste / Verzeichnisdienste

2.21 Namensdienste

Namen verweisen auf unterschiedlichste Ressourcen (Computer, entfernte Objekte, Dateien, Dienste...). Namen erleichtern Nutzung und Kommunikation. Beispiel: URL für WEB-Zugriffe.

Namensdienste setzen dann den Namen in die Objektreferenz um. Beispiel: DNS

- Domain-Name → Attribute des Hosts, z.B. IP, Host-Typ

CORBA:

1. Naming Service
   - Name eines entfernten Objektes → Objektverweis
2. Trading Service
   - Name eines entfernten Objektes → Objektverweis + Attribute

Namen sind oft spezifisch für Dienste.

- Dateinamen im Dateisystem
- Prozess-ID im Betriebssystem
- Benutzername auf einem Rechner
- *E-Mail Adressen*
- *usw...*

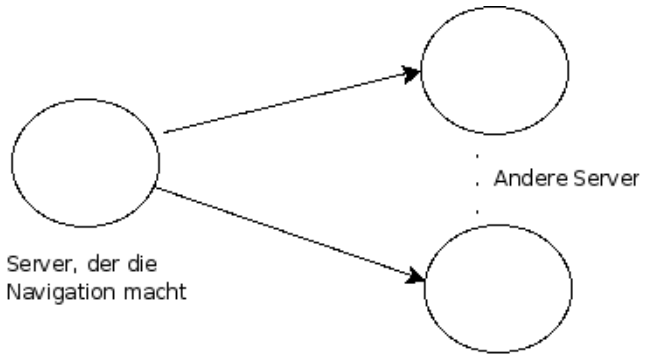
Die Namen werden nur im Kontext dieses Dienstes verwendet.

Aufgaben und Anforderungen an Namensdienste:

- Verwaltung mehrerer Kontexte
- Auflösung von Namen, d.h. die Umsetzung Name → Attribute
- Erstellen und Löschen von Bindungen Name ↔ Attribute
- Verwaltung von Namen über Systemgrenzen hinweg
- Verwaltung sehr großer Mengen von Namen
- Der Dienst soll Änderungen der Implementation und Komponenten über lange Zeit überstehen
- Der Dienst muß höchst verfügbar sein
- Lokale Fehler lassen nicht den gesamten Dienst ausfallen

Suche von Namen auf mehreren Servern heißt Navigation.

- Iterative Navigation
  - Ein Server kontaktiert mehrere Server hintereinander um den Namen zu finden.
- Rekursive Navigation
  - Server, die nach Namen gefragt werden, kontaktieren ihrerseits aktiv weitere Server.



Iterative Navigation

2.22 Verzeichnisdienste (Directory Service, Yellow Pages)

Gesucht wird nach einer Dienstleistung. Es ist "egal", wer diesen Dienst anbietet.

- "Drucke ein Dokument auf einem nahen Drucker".
- "Gib mir eine Wettervorhersage"

Verzeichnisdienst:

- Gib mir alle (einen, mehrere) Namen mit den folgenden Attributen.

3 Übungen

3.1 Übung 1

3.1.1 Aufgabe 1

Aufgabenstellung

Geben Sie fünf Typen für Hardware-Ressourcen und fünf Typen für Daten- oder Software-Ressourcen an, die sinnvoll gemeinsam genutzt werden können. Geben sie Beispiele für die gemeinsame Nutzung, wie sie in der Praxis in verteilten Systemen auftritt.

Lösung

Gemeinsam genutzte Ressourcen:

1. Netzwerk (Hardware)
   - Gute Skalierbarkeit
   - Relativ gute Positionstransparenz
   - Gute Implementationstransparenz
2. Speicher (Arbeitsspeicher / RAM)
   - Gemeinsame Nutzung durch Prozesse
3. Festplattenspeicher / Dateisystem
   - Gut skalierbar
   - Leicht benutzbar durch offene Schnittstelle (z.B. FTP, ...)
   - Mitunter werden verschiedene Dateiformate verwendet!
4. Rechenleistung
   - Seti-Projekt, leicht skalierbar
   - Wettervorhersage, schwer saklierbar im Sinne von VS
5. CAMPUS - HIS-QUIZ

- Prüfungsanmeldung
- Statuslisten
- Raumbelegungen
- Skalierbarkeit ist nicht so wichtig *da die Anzahl der Studierenden sehr unwahrscheinlich sehr stark ansteigen wird*
- Hohe Sicherheitsanforderungen

3.1.2 Aufgabe 2

Aufgabenstellung

Wie können die Uhren in zwei Computern, die über ein lokales Netzwerk verbunden sind, ohne Benutzung einer externen Uhr synchronisiert werden? Welche Faktoren schränken die Genauigkeit ein? Welche Methoden zur Verbreitung der Zeit kennen Sie? Wie genau sind Sie?

3.1.3 Aufgabe 3

Aufgabenstellung

Die Kerntechnologien des WWW für die Anzeige von Informationen sind HTML, URLs und HTTP. Erläutern Sie jeweils ihre Funktionen und welches Ziel durch ihren Einsatz erreicht wird. Sind die Technologien als allgemeine Grundlage von Client/Server-Programmierung ausreichend und geeignet?

Lösung

- HTML (**H**yper**t**ext **M**arkup **L**anguage)
  - Grammatik, Syntax, Semantik
  - Standardisierung
  - Offenheit
  - Leichte Verwendbarkeit
  - Implementationsunabhängigkeit
  - Erweiterbarkeit
- URL (**U**niform **R**esource **L**ocator)
  - Bsp.: http://www.webmail.fh-aachen.de/ossmann/arbk/pra1.zip
    1. Domain Name Servie: Macht aus dem Hostnamen eine IP-Nummer
    2. *Anfrage des Servers auf den jew. Port*

3. *Anfrage des Servers auf das jew. Unterverzeichnis*

- Eindeutige Bezeichnung in verteilten Systemen
- Orts. bzw. Positionstransparenz
- Einigkeit über die Interpretation von Zeichen
- Bereitstellung eines Namensraums

- HTTP (**H**yper**t**ext **T**ransfer **P**rotocol)
  - Offenheit
  - Standardisierung
  - Implementationsunabhängig
  - HTTP ist üblicherweise nicht gut geeigent beliebige Client-Server Anwendungen zu realisieren

3.1.4 Aufgabe 4

Aufgabenstellung

Geben Sie Beispiele von URLs an, und geben sie an, was die einzelnen Teile bedeuten. Inwiefern wird mit URLs das Ziel der Positionstransparenz erreicht? Wozu dient der DNS im Internet. Auch in Programmiersprachen werden Namen verwendet, wozu? Gibt es Parallellen zu URLs?

Lösung

"Bischen" miterledigt in Aufgabe 3 meint der Prof.

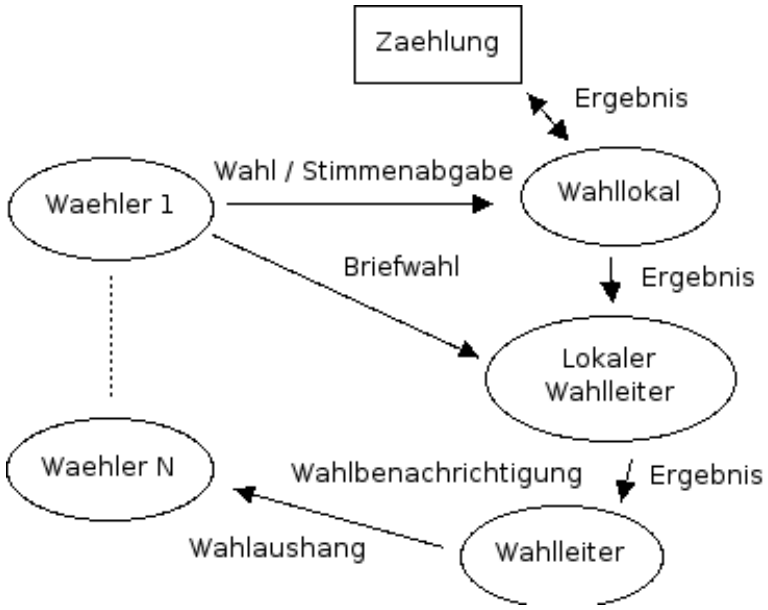
3.1.5 Aufgabe 5

Aufgabenstellung

Die Durchführung einer Wahl in einem demokratischen Staat kann als Operation in einem verteiltem System aufgefasst werden. Üblicherweise gibt es einen Wahlleiter, der die Wahl organisiert, durchführt und Ergebnisse feststellt. Welche Arten von Fehlern werden normalerweise berücksichtigt (Fehlermodell)? Welche Arten von Angriffen (Sicherheitsmodell) werden normalerweise berücksichtigt? Versuchen Sie eine Interaktionsmodell zu erstellen, das die beteiligten Komponenten und ihre Beziehungen sichtbar macht.

Lösung

Siehe Grafik.



3.1.6 Aufgabe 6

Aufgabenstellung

Es soll ein Dienst "INFO" realisiert werden. INFO verwaltet eine potentiell sehr grosse Menge von Informationen als gemeinsam nutzbare Ressource, auf welche Benutzer durch Angabe eines Schlüssels (Zeichenkette) zugreifen können. Diskutieren Sie, wie Sie die Namensgebung (d.h. Schlüsselbezeichnunge) organisieren, damit ein geringer Leistungsverlust entsteht, wenn die Anzahl der Ressourcen (Informationne), die verwaltet werden muß, steigt. Schlagen Sie vor, wie Sie INFO implementieren, damit Leistungsengpässe vermieden werden, wenn die Anzahl der Benutzer sehr gross ist.

Lösung

Bsp.: www.gpsies.com, Website die alle Routen für Rad-, Jogger- und Wanderwege. Ohne Bilder würde der Dienst weniger Speicher benötigen als mit Bilder. Für die Streken müssen jedoch alle Wegpunkte vorhanden sein, da ein Weg nie "nur" geradeaus führt. 1MB Text würde ca 80 Seiten Text entsprechen. Anzahl der Wegpunkte ist hier ca. auf 150 pro Weg beschränkt. Das wären dann ca. 8 Byte an Daten pro Wegstrecke. 32 Bit für eine Erdkoordinate entspricht eine Genauigkeit auf 10cm.

- Rundtouren haben Namen als Schlüssel
- Jede Rundtour besteht aus < 5000 Wegpunkten a 20 Bytes, d.h. 100KBytes/Tour

- Derzeit < 10000 Touren für Deutschland
- D.h. zur Zeit ca 1GByte Speicher nötig
- Wenn alle Betrachter von Touren nur 1 Server zur Verfügung haben und die Anzahl der Betrachter wächst, wird der Server zum Engpass!
  1. Ausweg: Mehrere Server die den gesamten Datenbestand anbieten. (Ist aus Backup-Gründen sowieso gut!)
- Weiteres Wachstum der Betrachterzahl
- Wenn alle Server am gleichen Ort stehen wird das Netzwerk zum Engpass
  1. Netzwerk besser machen
  2. Bringe die Daten näher zum Benutzer durch Einrichten lokaler Server
- Erweiterung: Jede Tour kann mit Fotos und Videos erweitert werden. Jede Tour benötigt eventuell 20 MByte.
  - Idee: Jeder Server speichert nur die Touren, die "in der Nähe" liegen.
  - Vorsicht: Es wäre schlecht, wenn jede Tour nur auf einem Server gespeichert wäre. D.h. nun sollte die gesamte Information mehrfach verteilt speichern.
  - Vorsicht: Es ist jetzt viel schwerer, eine konsistente Ansicht des Systems herzustellen
- Eventuell vergeben Benutzer gleiche Namen für verschiedene Touren (z.B. "Rund um den See")
- Durch Erweiterung (z.B. Benutzername, Regionsname, Ländername, Kontinentname, ...) kann man wieder eindeutige Namen herstellen
- Intern verwende eindeutige Nummerierung o.Ä.

3.1.7 Aufgabe 7

Aufgabenstellung

In der Programmiersprache C benutzten Programme (Prozese) einfache sequentielle Dateien, die das Betriebssystem per FAT32 verwaltet. Welchen Service erbring das Dateisystem. Welche Operationsmenge wird realisiert? Wird Implementationstransparenz erreicht? Kann man die Benutzung als Client-Server Anwendung auffassen? Muß ein Benutzer wissen, wo seine Daten auf der Festplatte stehen? Können in diesem Beispiel Ressourcen von mehreren genutzt werden?

Lösung

Welche Dienstleistung erbringt ein Dateisystem? Welche Operationen werden durch Dateisystem erbracht?

1. Verwaltung von "Dateien" die durch Dateinamen bezeichnet werden
2. Schreiboperationen
3. Leseoperationen
4. Open: Dateiname → Dateihandle
5. Close: Dateihandle
6. Verzeichnisdienst (*Macht aus einem Dateinamen eine Referenz*)
7. Gegenüber einer Reihe von Varianten (z.B. Festplattengrösse, FAT16, FAT32, Clusterung, Fragmentierung, ...) ist Implementationstransparenz erreicht
8. Man kann nicht beliebig leicht zwischen Betriebssystem wechseln
9. Bei USB-Datenträgern hat man das Ziel der Implementationstransparenz stärker berücksichtigt als bei Festplatten (*daher wird bei den Sticks auch meistens ein FAT-Dateisystem verwendet, da dies relativ einfach aufgebaut ist und von den meisten Betriebssystemen gelesen werden kann*).
10. Positonstransparenz ist erreicht. Der Benutzer muß nichts über den physikalischen Ort der Dateien wissen.
11. Grundsätzlich ist vorgesehen, dass mehrere Prozesse die gleiche Datei nebenläufig benutzen.

3.1.8 Aufgabe 8

Aufgabenstellung

Ein Dienst wird von mehreren Servern implementiert. Geben sie Gründe dafür an, warum es sinnvoll sein kann, Ressourcen zwischen den Servern zu übertragen. Ist es zufriedenstellend, wenn Clients ihre Anfrage nach einem Dienst durch Multicast-Nachrichten stellen? Lässt sich so Mobilitätstransparenz erreichen?

Lösung

Beispiel: Benutzer hat "Web-Space". Der Anbieter bringt diese Daten sinvollerweise auf einen Server der "nahe" beim Benutzer ist. Wenn der Benutzer sich bewegt (Wohnortwechsel AC → NYC) sollten sich die Daten mitbewegen.

Wenn man nicht weiss, wo eine Ressource liegt, könnte man eine Multicast-Nachricht an die Gruppe der möglichen Orte nach der Ressource fragen.

- Wenn alle Benutzer immer mit einem Multicast nach einer Ressource fragen erzeugt das sehr viel unnötigen Verkehr im Netz.
- Viele Server werden erfolglos gefragt

Richte einen Verzeichnis- oder Namensdienst ein.

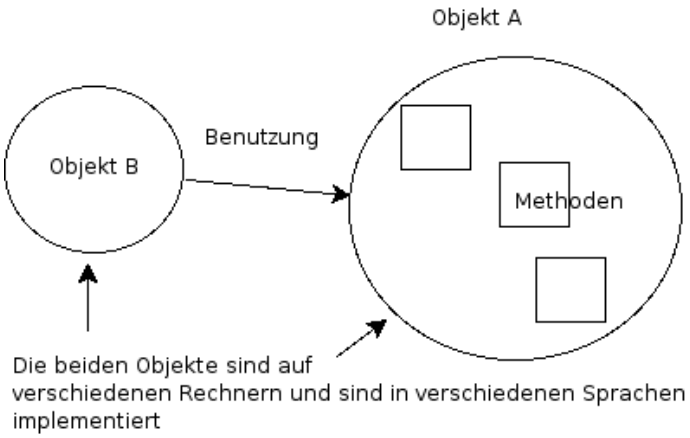
3.1.9 Aufgabe 9

Aufgabenstellung

Ein Server-Programm, das in einer bestimmten Programmiersprache erstellt wurde, stellt die Implementierung eines Objekts bereit, welches für den Zugriff durch Clients vorgesehen ist, die in anderen Programmiersprachen geschrieben wurden. Geben Sie Probleme an, die durch die Heterogenität verursacht werden. Wie kann man Sie lösen?

Lösung

Grund-Datentypen: integer, char, byte, float, double, ... ⇒ Konvertierung zwischen den Sprachen muß geregelt werden.



- Oft muß Einigkeit über die Darstellung im Speicher erzielt werden (Little-Endian, ..., Zeichensätze, ...).
- Es kann sein, dass eine Sprache, Maschine, usw. andere zusammengesetze Datentypen hat (array, struct, enum).
- Evtl. muß man zusammengesetzte Datentypen nachbilden oder zwischen Typen konvertieren.
- Methodenaufruf:
  1. Parameterübergabemechanismen
  2. Namensräume
  3. Wie werden Objektreferenzen dargestellt?

3.2 Übung 2

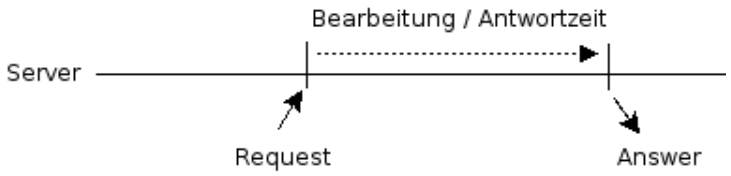
3.2.1 Aufgabe 1

Aufgabenstellung

Welche Faktoren beeinflussen die Antwortzeiten von Applikationen, die auf von einem Server verwaltete gemeinsam genutzte Daten zugreifen. Beschreiben Sie möglichst Lösungsanstätze um zu kleinen Antwortzeiten zu gelangen. Wie sinnvoll sind diese?

Lösung

Einflussfaktoren:



- Anzahl der User (Clients).

  ⇒ Durch Erhöhung der Anzahl der Server kann man versuchen die Zahl der Clients pro Server zu erniedrigen. Wenn zwischen den Servern nicht zu viel Koordination erforderlich ist, wird das die Antwortzeit erniedrigen.

- Werden im mittel mehrere Clients nebenläufig bedient, steigt für Jeden die Antworteit.
- Komplexität des Problems. Schwere Probleme erfordern lange Rechenzeit.

  ⇒ Bessere Algorithmen, bessere Implementation, bessere CPU, besserer Rechner.
- Andere Prozesse auf dem gleichen Rechner welche CPU-Zeit brauchen.

  ⇒ Verteilung der verschiedenen Dienste auf verschiedene Rechner bzw. CPUs.
- Schlechtes Scheduling, schlechte Performance des Betriebssystems

  ⇒ Betriebssystem anpassen, anders konfigurieren, anderes Betriebssystem.

3.2.2 Aufgabe 2

Aufgabenstellung

Geben Sie Beispiele für Fehler in Hardware und Software an, die durch den Aufbau von Redundanz vermieden werden können. Ist der Einsatz im Einzelnen sinnvoll?

Lösung

Hardware:

- Netzteil fällt aus

  ⇒ Mehrfache Netzteile, ist technisch aufwändig, geht aber. Ist für normale Benutzer (PC-Bereich) zu teuer aber für hochverfügbare Anwendungen realisiert.
- Festplatten-Crash

  ⇒ RAID-Systeme a.Ä. schützen vor dem Einzelausfall. Man muß eigentlich prüfen, ob durch das Hinzufügen wirklich die Ausfallrate sinkt!

Software:

- Benutze “Exceptions” um Fehler abzunfangen, die ihre Software selbst feststellt. Der Programmierer bringt alle Exceptions die ihm bekannt sind!
  - Eventuell führen nicht berücksichtigte Exceptions zum Prozessabbruch.
  - Kann eventuell abgefangen und durch Redundanz maskiert werden (*z.B. gleiches Programm auf anderem Betriebssystem und/oder in einer anderen Programmiersprache*)

- Fehlerhafter Algorithmus
  - Doppelte Implementation bringt da nichts.
  - Eventuell kann man leicht feststellen, dass ein Ergebnis fehlerhaft ist (z.B. beim Sortieren).
  - Verschiedene Algorithmen + verschiedene Implementationen + verschiedene Betriebssysteme + ..., könnten helfen. Aber das ist extrem aufwändig.

3.2.3 Aufgabe 3

Aufgabenstellung

Betrachten Sie einen einfachen Server, der Client-Anforderungen ausführt, ohne auf anderen Server zuzugreifen. Erklären Sie, warum es im Allgemeinen nicht möglich ist, eine Begrenzung für die Zeit zu setzen, die es dauern darf, bis ein solcher Server auf eine Client-Anforderung reagiert. Was wäre erforderlich, damit der Server Anforderungen in einer begrenzten Zeit ausführen kann? Ist dies eine praktikable Option?

Lösung

- Scheduling ist unvorhersagbar
- Seitenfehler oder allgemein Verfügbarkeit von Ressourcen ist schwer vorhersagbar
- Netzwerkaspekte sind schwer vorhersagbar
- Caching-Effekte

Im Normalfall kann man diese Effekte nicht so gut kontrollieren, dass man Garantien abgeben kann. Viele Effekte kann der Anwendungsprogrammierer selbst nicht beeinflussen.

3.2.4 Aufgabe 4

Aufgabenstellung

Geben Sie für jeden der Faktoren, die zu der Zeit beitragen, die es dauern, eine Nachricht zwischen zwei Prozessen über einen Kommunikationskanal zu übertragen, an, welche Massnahmen erforderlich sind, um einen Grenzwert für seinen Beitrag zur Gesamtzeit vorzugeben. Warum werden diese Massnahmen in den aktuellen allgemeinen verteilten Systemen nicht angewendet?

Lösung

Haben wir schon genug zu gesagt meint der Prof

3.2.5 Aufgabe 5

Aufgabenstellung

Angenommen, ein grundlegender Lesevorgang von einer Festplatte liest manchmal Werte, die sich von den geschriebenen unterscheiden. Schlagen Sie vor, wie dieser Fehlertyp erkannt bzw. maskiert werden kann um einen weniger schwerwiegenden Fehlertyp zu erzeugen.

Lösung

1. Idee: Speichere zusätzliche Prüfinformationen (Paritätsbits, Prüfsummen, ...)
2. Idee: Mehrfaches Lesen ⇒ Mehrheitsentscheid
   - Lese 10 mal hintereinander
   - Restfehlerwahrscheinlichkeit ganz grob $(P_E)^N$ mit $P_E :=$ Einzelfehler. *Für keine Restfehlerwahrscheinlichkeit:* $n \rightarrow$ *unendlich.*
   - Wähle N so, dass die Fehlerrate ausreichend niedrig ist, d.h. dass z.B. andere Fehler häufiger oder schwerwiegender sind.
   - Falls (z.B. bei kleinem N) kein vernünftiger Mehrheitsentscheid möglich ist, sollte man dem Benutzer den Fehler mitteilen.

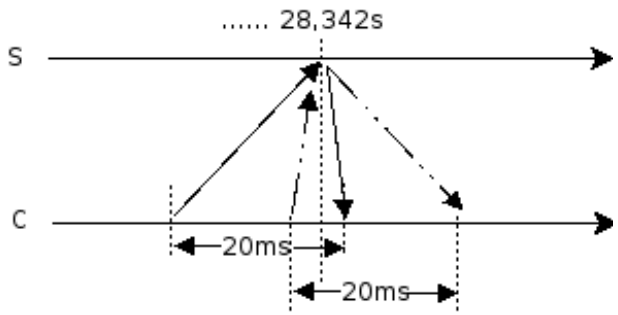
3.2.6 Aufgabe 6

Aufgabenstellung

Ein Client versucht, eine Synchronisierung mit einem Zeit-Server durchzuführen. Er zeichnet in der nachfolgend gezeigten Tabelle die Round-Tip-Zeiten und die Zeitstempel auf, die vom Server zurückgegebe wurden.

Round-Trip (ms)	**Zeitstempel (h:min:sec)**
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

Welche dieser Zeiten sollte er verwenden um seine Uhr zu setzen. Auf welche Zeit sollte er sie setzen? Schätzen Sie die Genauigkeit der Einstellung. Ändern sich die Antworten, wenn bekannt ist, dass die Zeit zwischen Senden und Empfangen einer Nachricht im System mindestens 8ms beträgt?

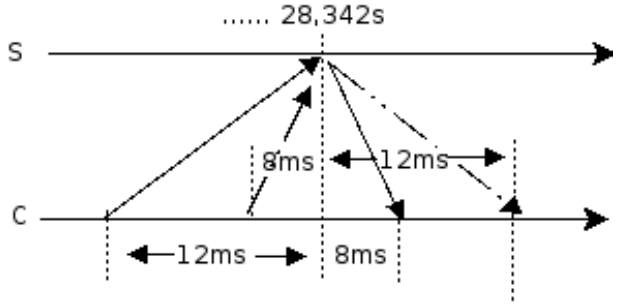


Lösung

Nehme die kleinste Round-Trip-Zeit weil damit die Ungenauigkeit minimal wird. (Bild). Empfangene Zeitstempel: $\frac{RTT}{2} = 10$ms. D.h. hier setzt der Client seine Zeit auf $28,342$ms $+10$ms $= 28,352$s. Genauigkeit kann um 10ms abweichen.

Wenn zusätzlich bekannt ist, dass eine Nachricht mindestens 8ms braucht, nimmt man nach wie vor zur Synchronisation das Paar mit der kleinsten Round-Trip Zeit.

Diagramm mit den Extremfällen bei 20ms RTT und mindestens 8ms Nachrichtenverzögerung. Setze eigene Uhr auf $28,342s + \frac{20ms}{2} = 28,352s$



Jetzt ist die Garantiegenauigkeit $\pm 2ms$

$2ms = \frac{1}{2}(20ms - 2 * 8ms)$

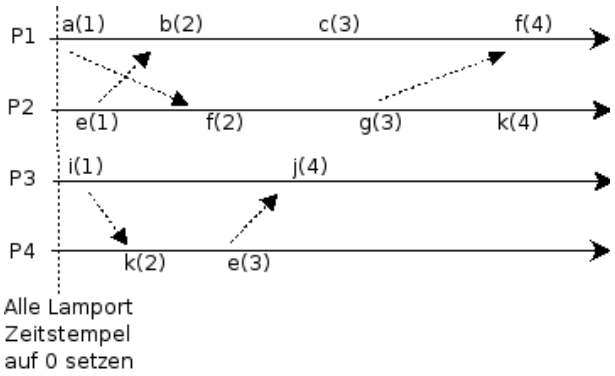
3.2.7 Aufgabe 7

Aufgabenstellung

Im folgenden Diagramm ist für 4 Prozesse die zeitliche Abfolge von Ereginissen und Nachrichten (Senden/Empfangen) angegeben. Ordnen Sie den Ereignissen Lamport-Zeitstempel zu. Gibt es nebenläufige Ereignisse? (Bild).

Lösung

Alle Ereignisse von P_1 und P_2 sind nebenläufig zu allen Ereignissen aus P_3 und P_4 weil zwischen diesen Paaren keine Kommunikation stattfindet. Ereignise die den gleichen



Lamport-Zeitstempel haben sind nebenläufig.

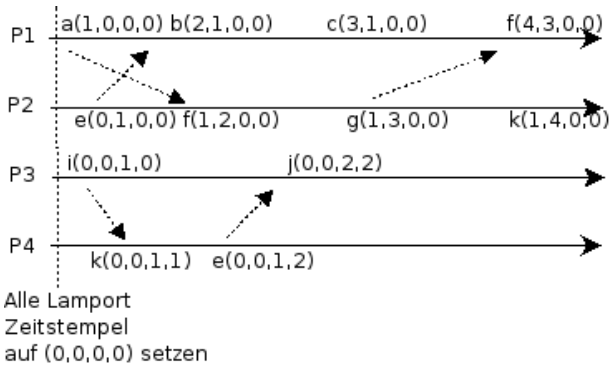
3.2.8 Aufgabe 8

Aufgabenstellung

Zur Zuordnung von Zeistempeln sollen nun Vektoruhren verwenden werden. Ordnen Sie den Ereignissen aus Aufg. 7 Vektor-Zeitstempel zu.

Lösung

Siehe Grafik.

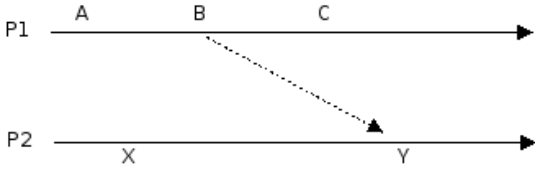


3.3 Übung 3

3.3.1 Aufgabe 1

Aufgabenstellung

Gegeben ist der folgende Ablauf von zwei Prozessen P_1 und P_2 mit insgesamt 5 Ereignissen. Aufgrund von Vektorzeitstempeln können Sie die globale Relation $\rightarrow$ zwischen Ereignissen auswerten. Sie verfügen aber nicht über die Kenntnis der zeitlichen Reihenfolge von Ereignissen.



1. Geben Sie die Historien history(P_1) und history(P_2) an!
2. Geben Sie einen RUN des Systems an der keine LINEARISIERUNG des Systems ist! Geben Sie einen konsistenten und einen inkonsistenten Schnitt des Systems an. Geben Sie die Linearisierung an, die durch die zeitliche Reihenfolge festgelegt ist! Skizzieren Sie die zugehörigen Schnitte!
3. Geben Sie alle Linearisierungen des Systems an! Zeichnen Sie zu den Linearisierungen jeweils zugehörige zeitliche Abläufe in einem Diagramm ähnlich dem oben angegebenen!

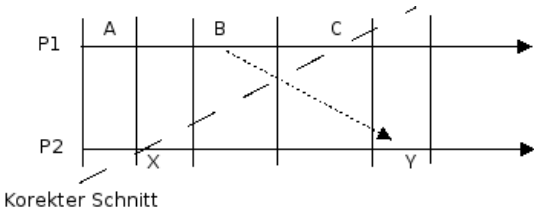
Lösung

1. History $(P_1) = ABC$
   History $(P_1) = XY$

2. $CBAYX$ ist kein RUN und keine Linearisierung.

   $AXBCY$ ist ein RUN *Weil ABC ist in der richtigen Reihenfolge und auch XY ist in der richtigen Reihenfolge.*

   $XYABC$ ist ein RUN aber keine Linearisierung, weil (z.B.) gegen $B \rightarrow Y$ verstossen wird.



3. Liste aller Linearisierungen

   $ABXZY$
   $ABXCY$
   $ABXYC$
   $AXBCY$
   $AXBYC$
   $XABCY$
   $XABYC$

3.3.2 Aufgabe 2

Aufgabenstellung

In dieser Aufgab wird ein verteiltes System betrachtet, welches aus zwei Prozessen P_1 und P_2 besteht. Beide Prozesse verwenden eine Kopie der Datei x. Jede einzelne Kopie hat einen Zeitstempel. Die Kopie von Prozess P_1 hat Zeitstempel a, die Kopie von P_2 hat Zeitstempel b. Immer wenn ein Prozess an der Datei Änderungen (z.B. mit dem Editor) vornimmt, wird der Zeitstempel um 1 erhöt.

Ein Prozess kann einem anderen die Datei inklusive Zeitstempel in einer Nachricht senden. Dann hat nach dem Empfang die Kopie des Empfängers den Zeitstempel der Kopie des Absenders.

Am Anfang haben die beiden Prozesse die gleiche Version der Datei und beide Kopien haben den Zeistempel $a = b = 4710$.

Die Systemadministrator hat die folgende Regel aufgestellt: Benutzer P_2 darf die Datei selbst nicht ändern. Er darf immer nur Kopien benutzen, die er von P_1 erhalten hat. Es ist aber nicht klar, ob Benutzer P_2 sich immer an diese Regel hält.

1. Man überlege sich, dass, wenn die Regel eingehalten wird, der Zeitstempel b nie grösser sein kann, als der Zeitstempel a. Ein interessantes Prädikat in diesem System ist also das Prädikat $b > a$.

2. Der folgende Ablauf hat stattgefunden. Dabei bezeichnen E das Editieren der lokalen Dateikopie, S das Senden der lokalen Dateikopie und R das Empfangen der lokalen Dateikopie. Man gebe jeweils die Zeitstempel an, die die Dateikopien erhalten!

3. a) Gibt es einen Zeitpunkt, zu dem $b > a$ wahr ist?
   b) Gilt "möglicherweise $b > a$?
   c) Gilt "definitiv $b > a$?

   Schöpft also der Systemadministrator in den vorliegenden Situationen Verdacht? Kann er anhand seiner Kenntnisse beweisen, dass P_2 unerlauberweise die Datei geändert hat?

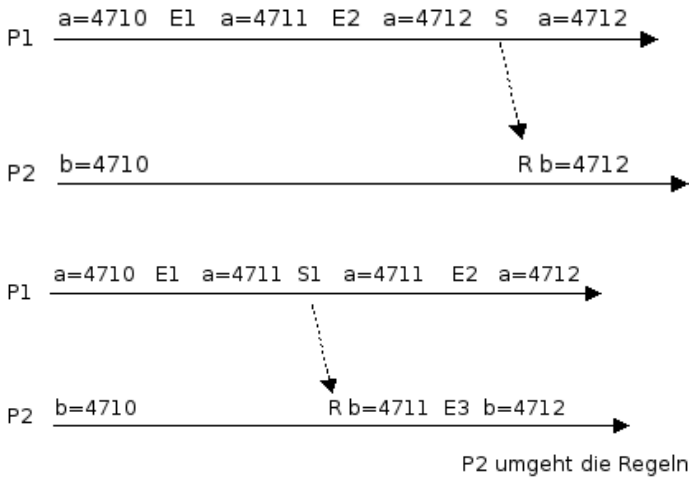
4. Wie sieht ein Verlauf aus, in welchem "möglicherweise $b > a$" nicht gilt? Sie dürfen z.B. zusätzliche Nachrichten versenden.

5. Wie sieht ein Verlauf aus, in welchem "definitiv $b > a$" gilt, also ein Ablauf in welchem der Systemadministrator den Nachweis des Verstosses gegen die Regel erbringen kann. Sie dürfen z.B. zusätzliche Nachrichten einzeichnen.

Lösung

1. Wird P_2 nur durch Empfang neuer Versionen von P_1 den Wert von b erhöt, kann nie $b > a$ sein (sofern sich P_2 an die Regeln hält).

2. *Was ist jedoch wenn P_2 sich nicht an die Regeln hält? Kann man das feststellen?* Auswertung der "$\rightarrow$"-Relation kann der Systemadministrator mit Vektorzeitstempeln durchführen.



3. a) Nach E_3 und vor E_2 ist $a = 4711$ und $b = 4712$

   b) "möglicherweise ist $b > a$'

   Im folgender Linearisierung

   $E_1 S_1 R_1 E_3 | E_2$

   gibt es einen globalen Zustand (der mit dem angedeuteten Schnitt aassoziert ist) in welchem $a = 4711 \quad b = 4712$ ist, d.h. $b > a$. Es ist also möglicherweise $b > a$.

   c) *Frage anders ausgedrückt:* Gibt es in jeder Linearisierung einen globalen Zustand (d.h. Schnitt) in welchen $b > a$?

   Um zu zeigen, dass (definitiv...) nicht gilt, reicht es also, eine Linearisierung zu finden bei welcher in keinem Zustand $b > a$ ist.

	E_1	S_1	R_1	E_2	E_3
$a = 4710$	4711	4711	4711	4712	4712
$b = 4710$	4710	4710	4711	4711	4712

3.3.3 Aufgabe 3

Aufgabenstellung

In einem verteilten System wollen zwei Prozesse hintereinander auf einen gemeinsamen Drucker drucken. Durch einen "Token" soll der Zugriff geregelt werden. Nur der Prozess,

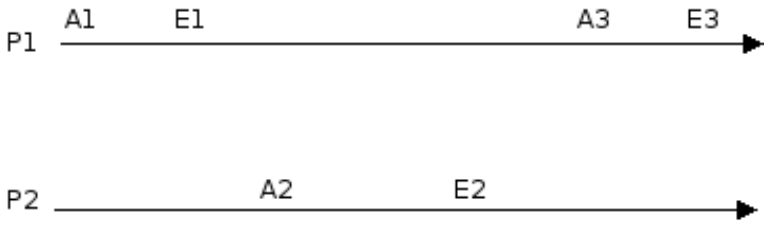
der das “Token” hat, darf drucken. Ein Prozess darf durch Nachrichten an den anderen Prozess versuchen, das Token zu bekommen. Er erhält das Token ebenfalls durch Nachrichtenversand. Veranschaulichen Sie sich, welche Arten von Abläufen mit Ereignissen sich prinzipiell ergeben. Wie kann man erreichen, dass

möglicherweise(“P_1 druckt” und “P_2 druckt”)

nicht wahr ist! Gehen sie davon aus, dass Nachrichten fehlerfrei gesendet und empfangen werden.

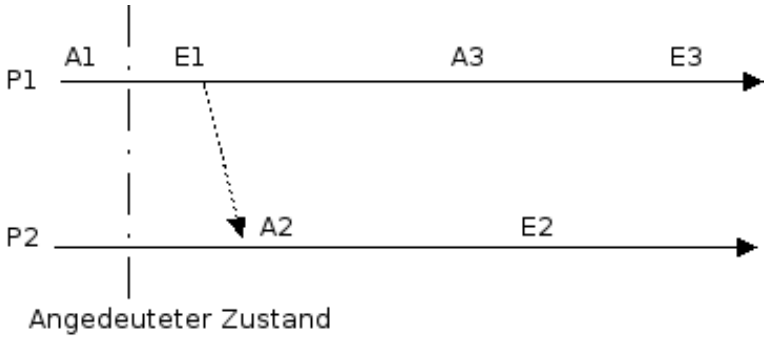
Lösung

Ereignisse: A Anfang des Druckens, E Ende des Druckens. Man möchte nicht, dass sich



Druckerbelegungen überlappen. D.h. man will wechselseitigen Ausschluss.

Wenn zwei Prozesse in einem globalen Zustand (d.h. konsistenten Schnitt) jeweils A gemacht haben, und noch keiner E, dann sei das Prädikat “Konflikt” wahr. In einem



so programmierten System ist das Prädikat definitiv ($\overline{Konfkikt}$) wahr. Denn in jeder Linearisierung gibt es den angedeuteten Zustand $A_1|...$ in welchem $\overline{Konflikt}$ wahr ist.

Deswegen ist das Prädikat "definitiv ($\overline{Konflikt}$)" für diese Aufgabenstellung nicht hilfreich.

Das Prädikat möglicherweise (Konflikt) ist besser. Wenn möglicherweise (Konflikt) unwahr ist, heißt das, dass es in keiner Linearisierung irgendeinen globalen Zustand gibt, in welchem ein Konfliktkt besteht. Das ist das, was man will.

Man regele Zugriff durch ein Token. Ein Prozess darf A nur ausführen , wenn er das Token besitzt. Solange ein Prozess "zwischen A und E ist, darf er das Token nicht versenden. Wir gehen P_1 am Anfang des Tokens. Durch die Regeln ist jetzt die Konfliktfreiheit sichergestellt.

3.3.4 Aufgabe 4

Aufgabenstellung

Es sei φ ein Prädikat in einem verteilten System. Beweisen Sie, dass aus definitiv (nicht φ) nicht folgt nicht (möglicherweise(φ)).

Lösung

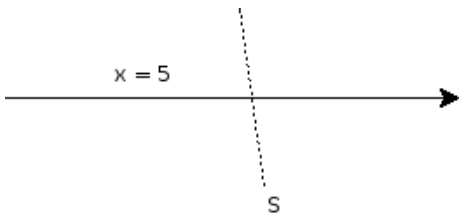
Zu zeigen ist: Aus definitiv($\overline{\varphi}$) folgt nicht $\overline{moeglicherweise \quad \varphi}$.

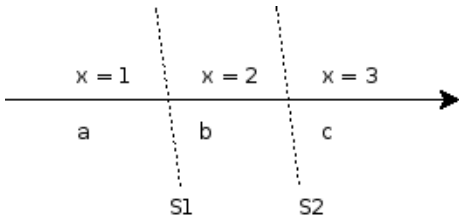
Es reicht, ein Beispiel bei welchen definitiv $\overline{\varphi}$ gilt und möglicherweise φ.

- definitiv $\overline{\varphi}$: In jeder Linearisierung gibt es jeweils einen globalen Zustand, so dass φ unwahr ist.
- möglicherweise φ: Es gibt eine Linearisierung, so dass φ wahr ist.

Prädikate werden in globalen Zuständen (Schritten) ausgewertet. Ihr Wahrheitswert kann sich ändern.

Betrachte Zuweisungen an Variable als Ereignisse. Im Schnitt S hat x den Wert 5,





d.h. $x == 5$ ist "true". Prädikat φ: $x = 1$. Im zweiten Bild wird gezeigt:

S_1 zeigt, dass 1 gilt
S_2 zeigt, dass 2 gilt

3.4 Übung 4

3.4.1 Aufgabe 1

Aufgabenstellung

Der Algorithmus von Maekawa zum wechselseitigen Ausschluss soll für $N = 16$ Prozesse implementiert werden. Es soll die in der Vorlesung behandelte einfache Methode zur Bestimmung der Wählermengen angewandt werden. Die Prozesse seien von 1..16 durchnummeriert.

1. Geben Sie für die Prozesse $1, 2, 5, 6, 8$ die Wählermengen an!
2. In wievielen Wählermengen ist ein Prozess jeweils enthalten?
3. Eine Fordernung an die Wählermengen ist, dass je zwei Wählermengen mindestens ein gemeinsames Element haben müssen. Ist diese Forderung auch erfüllt, wenn man die Wählermengenbestimming z.B. für $N = 12 = 3 * 4$ mit einer rechteckigen $3*4$ Matrix anstelle mit einer quadratischen Matrix durchführt. Kann man stattdessen auch eine 1x12 Matrix nehmen?

Lösung

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$V_1 = \{1, 2, 3, 4, 5, 9, 13\}$
$V_2 = \{1, 2, 3, 4, 6, 10, 14\}$

$V_5 = \{1, 5, 6, 7, 8, 9, 13\}$
$V_6 = \{2, 4, 5, 7, 8, 10, 14\}$
$V_8 = \{4, 5, 6, 7, 8, 12, 16\}$

Jede Wählermenge hat $7 = 2 * 4 - 1 = 2 * \sqrt{16} - 1 = 2 * \sqrt{N} - 1$ Elemente.

Jeder Teilnehmer ist in $7 = \sqrt{N} - 1$ Mengen enthalten.

1	2	3	4
5	6	7	8
9	10	11	12

$V_1 = \{2, 5, 6, 7, 8, 10\}$

Man darf es auch mit einer rechteckigen Matrix machen: Jeder kommt gleichviel in jeder Wählermenge vor.

1	2	3	4	5	6	7	8	9	10	11	12

$V_1 = \{1, 2, .., 12\}$

Man darf es auch mit einer extrem rechteckigen Matrix machen. Jeder kommt gleichviel in jeder Wählermenge vor.

Erweiterung: Was macht man für $N = 13$

1	2	3	4
5	6	7	8
9	10	11	12
13			

Wenn man nun die "Zeilen-Spalten"-Konstruktion macht, ist die Frage, ob man noch Wählermengen erhält, die wechselseitigen Ausschluss sichern.

$V_1 = \{1, 2, 3, 4, 5, 9, 13\}$ hat 7 Elemente
$V_2 = \{1, 2, 3, 4, 6, 10\}$ hat 6 Elemente
$V_{13} = \{1, 6, 9, 13\}$ hat 4 Elemente

Die Wählermengen sind nicht mehr alle gleich gross und Teilnehmer sind verschieden oft in Wählermengen enthalten.

Durch Eintragen "von oben" ist gesichert, dass 1 gemeinsames Element bleibt.

Wenn ein Teilnehmer abstürzt, muß man neue Wählermengen bestimmen. Frage: Darf man den abgestürzten Teilnehmer einfach aus allen Wählermengen rausnehmen?

1	2	3	4
5	6	7	8
9	10	11	12
13			

Wenn man Teilnehmer 13 rausnehmen würde, dann gäbe es kein Problem. Falls 5 abstürzt und entfernt wird, ist $V_7 = \{3, \not{5}, 6, 7, 8, 11\}$ und $V_{13} = \{1, \not{5}, 9, 13\}$. D.h. V_7 und V_{13} haben kein gemeinsames Element mehr. Die Vorgehensweise, abgestürzte Teilnehmer in allen Wählermengen zu streichen ist unzulässig.

3.4.2 Aufgabe 2

Aufgabenstellung

Das Durchführen einer Zentral-Abiturprüfung kann als Operation in einem verteitel System betrachtet werden, wobei die Aufgaben per MULTICAST in die Teilnehmer versandt werden.

1. Unterscheidet man bei diesem Problem zwischen “Empfang” und “Auslieferung” der Nachrichten? Wann wird ausgeliefert?

2. Wie wird bei dieser Aufgabe das Problem gelöst, dass dadurch entsteht, dass es in einem verteilten System keine globale Zeit gibt?

3. Liegt einer solchen Prfüfungsopteration eher ein “synchrones” oder ein “asynchrones” Systemmodell zugrunde? Wie wird mit ”Nachrichtenverlust” umgegangen?

4. Finden Sie andere Beispiele dafür, dass zwischen “Empfang” und “Auslieferung” einer Nachricht unterschieden wird!

Lösung

1. Die Aufgaben werden per Multicast an die Teilnehmer der Prüfung verteilt. Im Normalfall treffen die Aufgaben “beim Lehrer” ein, werden aber an die Teilnehmer noch nicht ausgeliefert. Lehrer ist stellvertretend für die Middleware.

2. Man verhindert die Kommunikation der Teilnehmer von dem Zeitpunkt an, bei dem die erste Aufgabe ausgeliefert wird. Weil es keine globale Zeit gibt, stoppt man die Kommunikation schon etwas vorher. Dabei setzt man implizit voraus, dass das System synchron ist. Die Aufgaben werden in der Realität einen gewissen Zeitraum vor der Auslieferung versandt um sicherzustellen, dass sie rechtzeitig da

sind. Darauf kann man sich nur in einem synchronen System verlassen. Man will, dass entweder alle die gleichen Aufgaben lösen, oder keiner irgendeine.

3. Beim Design gehen alle davon aus, dass das System sich synchron verhält. Im Normalfall verhält sich die Realität für ein Zentralabitur auch "synchron genug".

3.4.3 Aufgabe 3

Aufgabenstellung

In der Vorlesung wurde ein Algorithmus zum zuverlässigem Multicast vorgestellt.

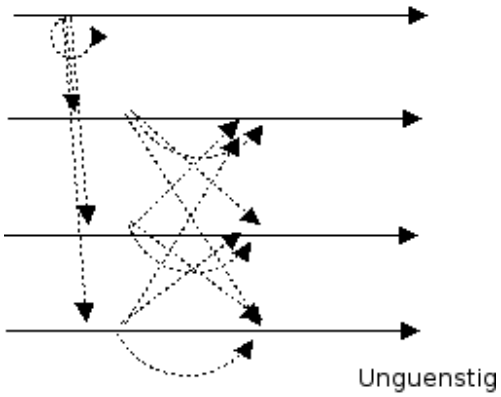
1. Wieviele Nachrichten werden dabei über 1:1 Kanäle versandt, wenn N Prozesse teilnehmen und keine Prozesse abstürzen? Ist die Anzahl davon abhängig, in welcher Reihenfolge Nachrichten bei Prozessen eintreffen?

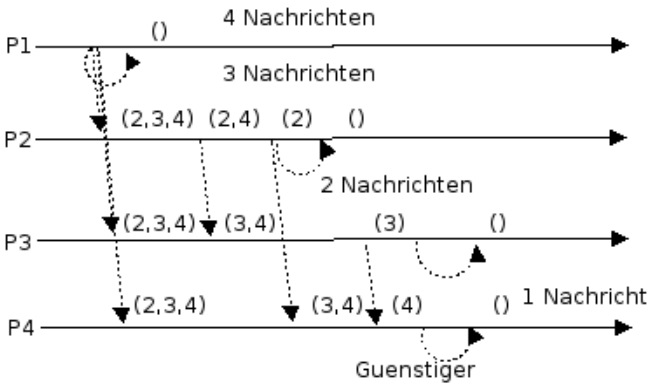
2. Sie modifizieren den Algorithmus dergestalt, dass ein Prozess nur noch an die Prozesse multicastet, von welchen er selbst noch keine Nachricht erhalten hat. Überlegen Sie sich für den Fall $N = 4$ wie solch ein Ablauf dann aussehen kann. Kann man sagen, mit wievielen nachrichten man in deinem günstigen Fall auskommt?

Lösung

1. Jeder Prozess (Teilnehmer) schickt irgendwann im Rahmen seines Basic-Multicasts an jeden Teilnehmer eine Nachricht. D.h. insgesamt werden bei N Teilnehmern N^2 Nachrichten verschickt.

2. Im Schlimmsten Fall senden alle Teilnehmer nach dem Empfang der ersten Nachricht so schnell, dass sie keine weitere Nachricht empfangen, bevor sie mit dem Senden fertig sind. Dann $N + (N-1)^2$ Nachrichten. Im optimalen Fall sind ver-





mutlich $1 + 2 + 3 + ... + N = N + (N - 1) + ... + 1$ Nachrichten erforderlich $= \frac{N*(N+1)}{2} \approx \frac{N^2}{2}$. Ist die gleiche Grössenordnung aber immerhin Reduktion um $\frac{1}{2}$.

3.4.4 Aufgabe 4

Aufgabenstellung

Die "ELECTION" Schnittstelle stellt zwei entfernte Methoden bereit:

- VOTE: Diese Methode verarbeitet zwei Parameter, mit denen ein Client den Namen eines Kandidaten (String) und die Nummer eines Wählers (32-Bit Integer) übergibt.
- RESULT: Diese Methode verarbeitet zwei Parameter, über die der Server dem Client den Namen des Kandidaten und die Anzahl der Stimmen für diesen Kandidaten mitteilt.

Welche Parameter dieser Methoden sind jeweils Eingabe-, welche Ausgabe-Parameter? Erklären Sie Marshalling und Un-Marshalling an diesen Parametern.

Lösung

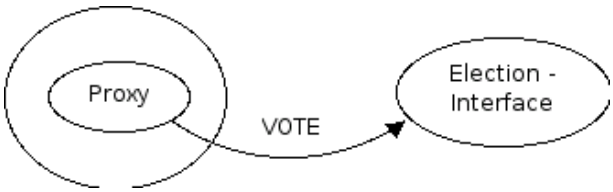
- `VOTE` wird beim Wählen aufgerufen
- `RESULT` wird nach der Wahl aufgerufen

`VOTE(name: STRING; Vnumber: int32)` Beides sind "in-Parameter".

Darstellung in einer Nachricht

- Methoden-ID: int32, Big-Endian 4 Bytes

- `STRING`:
  1. Variante: Länge 1 Byte gefolgt von 16-Bit Zeichen, und zwar so viele, wie die Länge angibt.
  2. Variante: Folge von Zeichen (z.B. Unicode) gefolgt von einem Terminierungszeichen.



Eventuell baut man "Rückabeparameter" ein, um aufgetretene Fehler identifizieren zu können. Der Proxy könnte evtl. liefern:

- 0: Ich habe eine positive Antwort vom Election-Interface bekommen
- -1: Time-Out, keine Antwort in vorgegebener Zeit
- ...

Das wäre als "out-Parameter" zu betrachten und in der IDL so anzugeben.

RESULT: Man gibt einen Kandidatennamen an und erhält wieviele Stimmen er in der Wahl erhalten hat.

```
RESULT(in Cname: STRING; out Cvotes: int32)
```

Marshalling wie oben. Eventuell zusätzlich: Detektierte Fehler weiterreichen.

3.4.5 Aufgabe 5

Aufgabenstellung

Der ELECTION Dienst muß sicherstellen, dass eine Wahl aufgezeichnet wird, wenn ein Wähler annimmt, dass er seine Wahl abgegeben hat. Beschreiben Sie, welche Auswirkung eine Vielleicht-Aufrufsemantik auf den ELECTION-Dienst hat. Ist eine Mindestens-Einmal Semantik für den ELECTION-Dienst ausreichend oder ist eine Höchstens-Einmal Semantik empfohlen.

Lösung

Bei einer "Vielleicht" Aufruf Semantik werden Stimmen, deren Nachrichten verloren gehen, nicht gezählt. Da nicht auf Antworten gewartet wird, wird auch nicht wiederholt. Da Nachrichtenverlust (z.B. bei UDP-Paketen) vorkommt, ist diese Korektheitsforderung zu schwach.

Variante: Mindestens einmal:

- Es werden so lange `VOTE`-Nachrichten geschickt, bis eine Antwort antrifft.

Bei dieser Variante filtert der Empfänger mehrfach eintreffende `VOTE`-Aufrufe nicht und `VOTE` wird deswegen eventuell mehrfach aufgerufen und Stimmen werden dann mehrfach gezählt.

In diesem Sinne sollte man eine "Höchstens einmal"-Semantik implementieren.

3.4.6 Aufgabe 6

Aufgabenstellung

Skizzieren Sie eine Implementierung des ELECTION-Dienstes, die sicherstellt, dass Ihre Aufzeichnungen konsistent bleiben, wenn durch mehrere Clients nebenläufig zugregriffen wird.

Lösung

- Versuche `VOTE` so zu implementieren, dass mehrfache Stimmenabgabe nur einmal gezählt wird.
- Implementiere `VOTE` idempotent!

Um herauszubekommen, wer schon Stimmen abgegeben hat, wird eine Hash-Tabelle implementiert mit dem Wählernamen als Schlüssel. (`WÄHLER_TABELLE`).

`WÄHLER_TABELLE.ISIN(WÄHLER)` liefert `TRUE` genau dann wenn der Wähler gewählt hat.

Weitere Datenstruktur zählt für jeden Kandidaten die Stimme. (`VOTES[KANDIDAT]`)

Wählertabelle ist am Anfang leer. `VOTES` hat am Anfang für alle Kandidaten den Wert 0.

Implementation:

```
if (! WÄHLER_TABELLE.ISIN(WÄHLER)) then {
```

```
        WÄHLER_TABELLE.PUTIN(WÄHLER, CANDIDATE);
        VOTES[CANDIDATE]++;
}
```

Bei einer Wahl soll diese Methode nebenläufig benutzt werden können, und für jede eintreffende `VOTE`-Nachricht wird dann ein `THREAD` erzeugt, welcher `VOTE` ausführt.

Vorsicht!!! Die oben angegebene Implementation ist nicht Thread-safe! Die Implementation sollte trotzdem effizient sein. Synchronisation ist nötig.

3.4.7 Aufgabe 7

Aufgabenstellung

Ein Client nimmt entfernte Prozeduraufrufe auf einem Server vor. Der Client benötigt 5ms, um die Argumente für eine Anforderung zu berechnen, und der Server benötigt 10ms, um eine Anforderung zu bearbeiten. Die lokale Betriebssystemverarbeitungszeit für jede Sende- oder Empfangsoperation beträgt 0.5ms. Die Netzwerkzeit für jede Nachricht beträgt 3ms. Das Marshalling und das Un-Marshalling benötigen 0.5ms pro Nachricht. Berechnen Sie die Zeit, die der Client benötigt, um zwei Anforderungen zu erzeugen und zurückzukerhen:

1. Wenn er single threaded ist
2. Wenn er zwei Threads hat, die Anforderungen auf einem einzigen Prozessor nebenläufig ausführen können

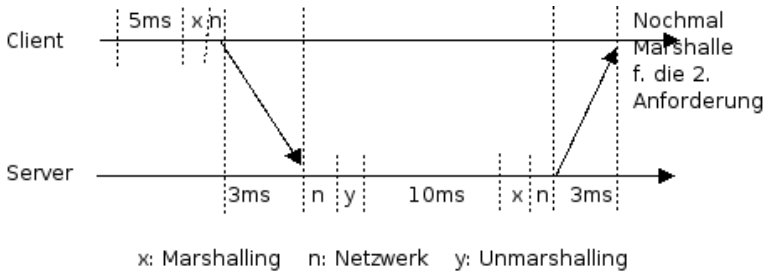
Die Zeiten für Kontextumschaltungen können Sie ignorieren

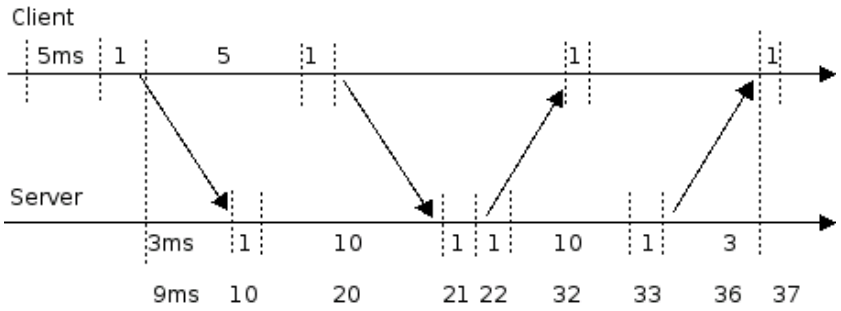
Lösung

Gesamtzeiten der Clientanforderungen:

1. Anforderung: Single-Threaded: $(4 + 6 + 5 + 10)\text{ms} = 25\text{ms}$
2. Anforderung: 50ms

Mit 2 Threads 27ms, das ist spürbar besser.





3.5 Übung 5

3.5.1 Aufgabe 1

Aufgabenstellung

Warum gibt es keine OPEN- und CLOSE-Operationen in unsere Schnittstelle zum flachen Dateidienst bzw. Verzeichnisdienst. Was sind die Unterschiede zwischen den LOOKUP-Operationen unseres Verzeichnisdienstes und dem OPEN von UNIX?

Lösung

Die Operationne im flachen Dateidienst sollten so sein, daß sie einfach mit "zustandslosen Servern" zu implementieren sind. Dabei sind beispielsweise idempotente Operationen hilfreich. In diesem Sinne passen die üblichen "OPEN" und "CLOSE" Operationen nicht zum flachen Dateidienst. Mit "OPEN" werden üblicherweise Zeiger auf aktuelle Positionen und Puffer für Inhalte angelegt.

OPEN im Vergleich zum LOOKUP:

- LOOKUP ist eine Operation im Verzeichnisdienst: Name → UFID und ist idenpotent

- "OPEN" verschmicht Verzeichnisdienst und Dateidienst. Einerseits bekommt man eine Dateireferenz ("handle"), anderseits wird die Verwaltungsstruktur (Zeiger, Puffer, ...) initialisiert (Zustand).

3.5.2 Aufgabe 2

Aufgabenstellung

In einem verteilten System werden mehrere Dateisysteme benutzt. Warum sollten UFIDs eindeutig über all diese Systeme sein. Wie wird die Eindeutigkeit sichergestellt?

Lösung

In einem einzelnen Dateisystem sind Namen eindeutig! Anderenfalls könnte man Dateien nicht über den Namem referenzieren. Eine einzelne Datei kann durchaus mehrere Namen haben. Will man mehrere Dateisysteme in ein verteiltes Dateisystem integrieren, so müssen die neuen Namen wieder eindeutig sein. Die "Schreibweise" von Namen muß dabei harmonisiert werden. Oft werden neue Dateisysteme eingebunden indem man eine Baumstruktur verwendet. Dabei bekommen die alten Dateinamen quasi Prefixe die von der Position des Einbaus ins bestehende Dateisystem abhängen.

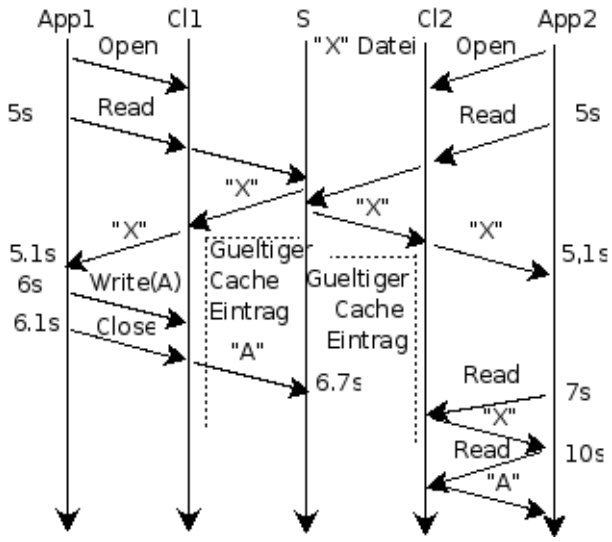
3.5.3 Aufgabe 3

Aufgabenstellung

In welchem Ausmaß weicht SUN-NFS von der Ein-Kopie-Aktualisierungssemantik ab? Konstruieren Sie ein Szenario, in dem zwei Prozesse auf Benutzerebene, die eine Datei gemeinsam nutzen, auf einen einzelnen UNIX-Host korrekt arbeiten, aber Inkonsistenzen aufweisen, wenn sie auf unterschiedlichen Hosts ausgeführt werden.

Lösung

SUN-NFS garantiert nicht die Ein-Kopie-Aktualisierungssemantik! Notation wie in der Vorlesung: READ liest das einzige Zeichen einer Datei. WRITE schreibt das einzige. 5s Aktualisierungsinterval und Gültigkeit von Cache-Einträgen zu prüfen. In diesem Ablauf hat das NFS bereits gegen die Ein-Kopie-Aktualisierungssemantik verstoßen, aber die Applikationen stellen diesen Verstoß noch nicht fest, weil sie relativ harmlose Operationen machen und die zeitlichen Abläufe die Entdeckung erschweren. Versuche zwei Programme so zu schreiben, daß mit hoher Treffsicherheit der Verstoß gegen die Ein-Kopie-Aktualisierungssemantik bemerkt wird. Idee: Programm A schreibt in die Datei und Programm B soll "garantiert danach" lesen, und etwas falsches lesen. Damit das klappt, muß B schon auf seinem Client vorher eine Kopie im Cache haben.

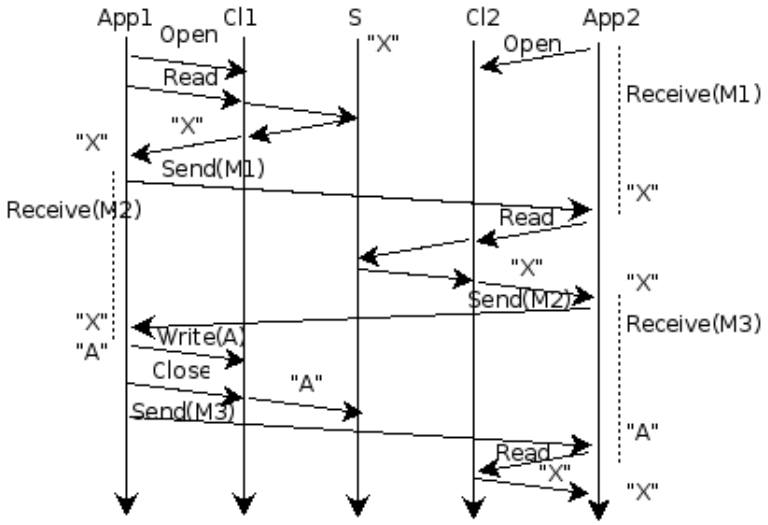


Annahme: Die Zeitverzögerung durch Nachrichtenaustausch und Scheduling sind, im Vergleich zum Aktualisierungsintervall (5s), vernachlässigbar klein (ms Bereich).

Damit wirklich was schief geht, synchronisieren sich die Prozesse durch Nachrichten-Austausch. Programmiere wie folgt:

Prozess A	**Prozess B**
Open	Open
C = Read	
Send(M1)	
	Receive(M1)
	C = Read
	Send(M2)
Receive(M2)	
Write(A)	
Close	
Send(M3)	
	Receive(M3)
	C = Read

Auf einem System, in welchem A und B auf dem gleichen Client laufen, liefert der zweite READ von B auf jeden Fall “A”, weil es nur eine Client-Cache-Kopie gibt. Ablauf wenn A und B auf verschiedenen Clients laufen: Dateiinhalt am Anfang: “X”. Durch den zusätzlichen Nachrichtenversand werden “Geschehen-Vor”-Beziehungen erzwungen.



Dadurch wird das Ergebnis des 2. Reads von B vorhersagbar und ein Verstoß gegen die Ein-Kopie-Aktualisierungssemantik nachweisbar.

3.5.4 Aufgabe 4

Aufgabenstellung

Wie geht das ANDREW-FILE-SYSTEM (AFS) damit um, dass CALLBACK-Nachrichten verloren gehen?

Lösung

Die “Callbacks” sind Benachrichtigungen vom (File-) Server zum Client, auf welchem das AFS die Datei cached. Da der Client beim Wiederstart nach einem Absturz seine Callback Promises überprüft und erneuert, ist der Verlust von Callbacks kein grosses Problem.

3.5.5 Aufgabe 5

Aufgabenstellung

Welche Ziele werden durch Caching auf Client- bzw. Server-Seite in einem verteilten Dateisystem erreicht?

Lösung

- Client-Caching: Weniger Netzlast, dadurch geringere Antwortzeit für den Client.
- Server-Caching: Weniger Schreib/Lese-Vorgänge auf der Festplatte, dadurch schnelleres Dateisystem. Eventuell geringere Antwortzeit des Servers.

3.6 Übung 6

3.6.1 Aufgabe 1

Aufgabenstellung

Ein Server verwaltet die Objekte $a_1, a_2, ..., a_n$. Der Server stellt zwei Operationen für seine Clients bereit:

- **read(i)** gibt den Wert von a_i zurück
- **write(i,wert)** weist a_i den angegebenen Wert zu

Die Transaktion T und U sind wie folgt definiert:

- T: x=read(j); y=read(i), write(j,44); write(i,33);
- U: x=read(k); y=write(i,55), read(j); write(k,66);

Geben Sie eine serielle äquivalente Verzahnung ("Interleavings") der Operationen der Transaktionen T und U an. Geben Sie eine nicht seriell äquivalente Verzahnung der operationen der beiden Transaktionen an!

Lösung

x und y seien Variablen, die beide Transaktionen gemeinsam benutzen.

T	**U**
x=read(j)	x=read(k)
y=read(i)	write(i,55)
write(j,44)	y=read(j)
write(i,33)	write(k,66)

(Statt a_j schreibe "j"). Ausgangswerte: $i = 1, j = 2, k = 3$.

Was ist die Gesamtwirkung für die Reihenfolge T, U und U, T?

T,U	Ergebnis
x=read(j)	x=2
y=read(i)	y=1
write(j,44)	j=44
write(i,33)	i=33
x=read(k)	x=3
write(i,55)	i=55
y=read(j)	y=44
write(k,66)	k=66

U,T	Ergebnis
x=read(k)	x=3
write(i,55)	i=55
y=read(j)	y=2
write(k,66)	k=66
x=read(j)	x=2
y=read(i)	y=55
write(j,44)	j=44
write(i,33)	i=33

Versuche ein echtes Interleaving, welches seriell äquivalent ist anzugeben.

U1	x=read(k)	x=3
T1	x=read(j)	x=2
U2		i=55
U3		y=2
U4		k=66
T2		y=55
T3		j=44
T4		i=33

ist seriell äquivalent zu U, T. Vertausche $U1, T1$, dann ergibt sich ein nicht seriell äquivalentes Interleaving.

3.6.2 Aufgabe 2

Aufgabenstellung

Erklären Sie, weshalb es bei der seriellen Äquivalenz erfoderlich ist, dass eine Transaktion keine weiteren Sperren setzen darf, nachdem sie eine Sperre für ein Objekt freigegeben hat.

Beispiel: Ein Server stellt wie in Aufg. 1 read und write zur Verfügung. Die Transaktionen T und U wie folgt definiert:

- T: x=read(i); write(j,44);
- U: write(i,55); write(j,66);

Beschreiben Sie die Verzahnung der Transaktionen T und U, in denen die Sperren für aufgehoben werden, sodass die Verzahnung nicht seriell äquivalent ist.

Lösung

- Keine Sperren, d.h. keine Nebenläufigkeitskontrolle ⇒ Geht nicht.
- Sperren für jedes Objekt: Frühzeitiges aufheben von Sperren. D.h. Aufheben wenn die Transaktion das Objekt nicht mehr braucht ⇒ das geht immernoch schief.
- Zwei-Phasen-Sperren: Aber gebe Sperren frei, bevor "COMMIT oder ABORT" feststeht ⇒ Das geht immer noch schief (durch die Wiederherstellung).
- Striktes Zwei-Phasen-Sperren ⇒ Funktioniert

Bis hier jedoch noch keine verteilten Transaktionen. Verteilte Transaktionen: Keine globale Nebenläufigkeitskontrolle ⇒ Geht evtl. schief.

3.7 Übung 7

3.7.1 Aufgabe 3

Aufgabenstellung

Nennen Sie ein Beispiel für die Verzahnung (Interleaving) von zwei Transaktionen, die seriell äquivalent auf jedem Server, aber nicht global seriell äquivalent ist.

Lösung

Vergleiche Übung 6 Aufgabe 2. Transaktionen:

- T: x=read(i); write(j,44);
- U: write(i,55); write(j,66);

T wird zerlegt in:

- T_I: x=read(i)

- T_J: write(j,44)

U wird zerlegt in:

- U_I: write(i,55)
- U_J: write(j,66)

Folgende Server müssen folgende Operationen ausführen:

- Server I hat T_I und U_I auszuführen. Lokal seriell äquivalent.
- Server J hat T_J und U_J auszuführen. Lokal seriell äquivalent.
- Server I macht zuerst I_I (x=read(i)) dann U_I (write(i,55))
- Server J macht zuerst U_J (write(j,66)) dann T_I (write(i,44))

Server	**I**	**Server**	**J**
	x=1		
	i=2		j=3
T_I	x=2	U_J	j=66
U_I	i=55	T_J	j=44

⇒ x=2, i=55, j=44 ⇒ Das Gesamtergebnis ist nicht seriell äquivalent ⇒ Eine globale Nebenläufigkeitskontrolle ist nötig.

3.8 Übung 8

3.8.1 Aufgabe 1

Aufgabenstellung

Drei Computer stellen einen replizierten Dienst bereit. Die Hersteller behaupten, dass jeder Computer eine mittlere Laufzeit zwischen Fehlern von 5 Tagen hat; ein Ausfall wird normalerweise innerhalb von 4 Stunden repariert. Welche Verfügbarkeit hat der replizierte Dienst? Wie ändert sich die Verfügbarkeit in Abhängigkeit von der Anzahl der Computer?

Lösung

Verfügbarkeit des einzelnen Computers:

- Zeit, die er korrekt läuft / Gesamte betrachtete Zeit

  = (5 Tage - 4h) / 5 Tage

$= \frac{5*24h-4h}{5*24h} = 0,96^-$

$= 96{,}6..\%$

Die Fehlerwahrscheinlichkeit p_1 für einen Einzelrechner beträgt also $\frac{4h}{5*24h} = 0,03^-$. Der replizierte Dienst wird als verfügbar betrachtet, wenn mindestens in Rechner läuft. D.h. er ist nicht verfügbar, wenn alle Rechner ausfallen.

- Fehlerwahrscheinlichkeit bei 3 Rechnern ist also $p_3 = p_1^3 = 0,000037...$
- Verfügbarkeit: $v_3 = 1 - p_3 = 0,999962...$

Das entspräche einer mittleren Ausfallzeit von $p_3 * 1$ Jahr $\approx 18,9$ Minuten / Jahr.

Allgemein: $v_n = 1 - p_1^N$

3.8.2 Aufgabe 2

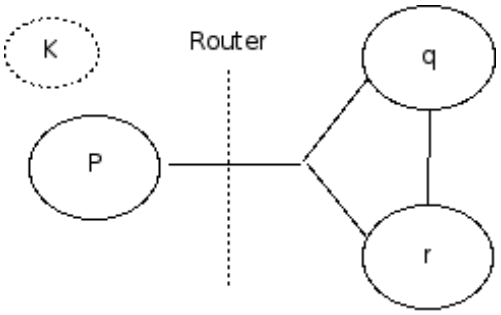
Aufgabenstellung

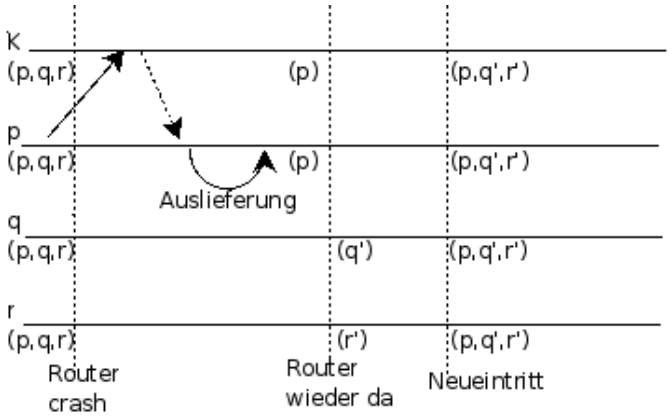
Ein Router, der Prozess p von zwei anderen, q und r, abtrennt, fällt aus, unmittelbar nachdem p das Multicasting von Nachricht m initiiert hat. Erklären Sie, was als Nächstes passiert, wenn das Gruppenkommunikationssystem ansichtssynchron ist?

Lösung

Annahme:

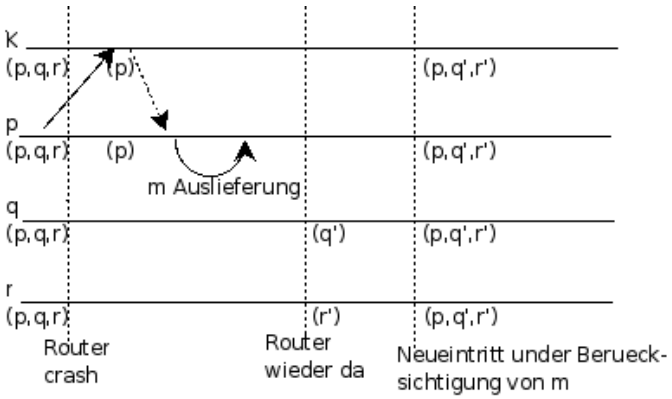
1. Fall: Ein Koordinator realisiert das Kommunikationssystem und er liegt in der

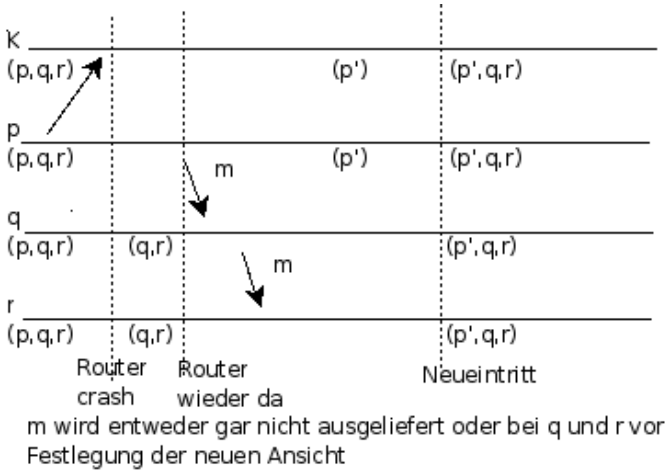




Nähe von p.

- p teilt dem Koordinator mit, er möchte m multicasten. Vorher ist nichts schiefgegangen ⇒ p q und r haben also Ansicht (p, q, r) ausgeliefert.


K
(p,q,r)
(p')
(p,q',r')
p
(p,q,r)
(p)
(p,q',r')
m Auslieferung
q
(p,q,r)
(q')
(p,q',r')
r
(p,q,r)
(r')
(p,q',r')
Router crash
Router wieder da
Neueintritt under Berueck-sichtigung von m


K
(p,q,r)
(p')
(p',q,r)
p
(p,q,r)
(p')
(p',q,r)
m
q
(p,q,r)
(q,r)
(p',q,r)
m
r
(p,q,r)
(q,r)
(p',q,r)
Router crash
Router wieder da
Neueintritt
m wird entweder gar nicht ausgeliefert oder bei q und r vor Festlegung der neuen Ansicht

2. Fall: Koordinator auf gleicher Seite wie q und r und k erlebt das Initiieren des Multicasts noch.

3.8.3 Aufgabe 3

Aufgabenstellung

Zur Reihenfolge der Aktualisierung in replizierten Diensten gibt es die folgenden Möglichkeiten:

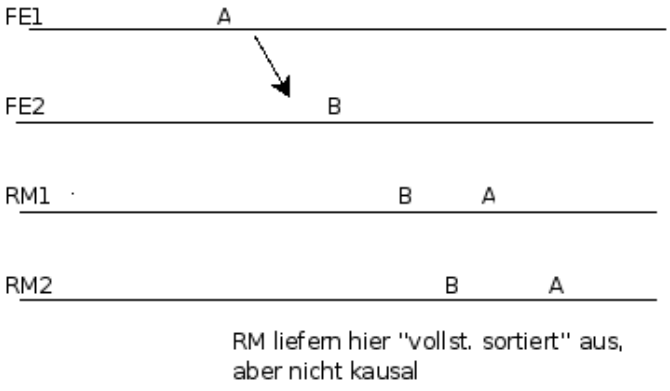
- FIFO-Reihenfolge: Wenn ein Frontend die Anforderung r und danach die Anforderung s absetzt, wird jeder korrekte Repliken-Manager, der s verarbeitet, zuvor r verarbeiten.
- Kausale Reihenfolge: Wenn die Anforderung r vor der Anforderung s abgesetzt wurde, wird jeder korrekte Repliken-Manager, der s verarbeitet, zuvor r verarbeiten.
- Vollständige Reihenfolge: Wenn ein korrekter Repliken-Manager r von der Anforderung von s verarbeitet, wird jeder korrete Repliken-Manager, der s verarbeitet, zuvor r verarbeiten.

Machen Sie sich klar, dass es sich tatsächlich um verschiedene Reihenfolgen handeln kann.

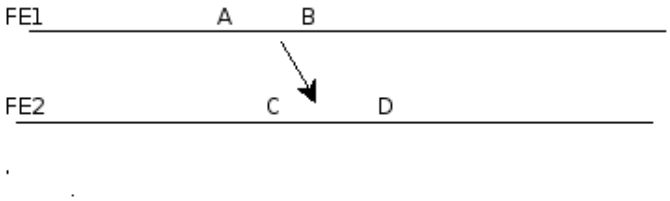
Lösung

Bei FIFO sind nur die “geschehen vor” Relationen der einzelnen FE ↔ RM Paare zu erfüllen, bei “kausal” alle, deswegen erfüllt ein System, welches die kausale Reihenfolge erfüllt, auch die FIFO Reihenfolge.

Aus “kausal” folgt nicht “vollständig”. Beispiel: Weil A und B nebenläufig sind, dürfen



RM1 und RM2 verschiedene Reihenfolgen wählen ohne gegen Kausalität zu verstoßen.



Folgt aus “volständig” kausal? RM1: C D A B, RM2: A B C D, erfüllt FIFO, ist aber nicht vollständig sortiert, ist nicht kausal weil RM1 mit D A dagegen verstößt.