Verteilte Systeme

5. Synchronisation

Sommersemester 2011

Institut für Betriebssysteme und Rechnerverbund

TU Braunschweig

Dr. Christian Werner
– Bundesamt für Strahlenschutz –

Überblick

- Zeit in Verteilten Systemen
  - Uhrensynchronisation
    - Christian's Algorithm
    - Der Berkeley-Algorithmus
    - Network Time Protocol (NTP)
  - Logische Uhren (Lamport)
    - Happened-Before-Beziehung
    - Logische Uhren stellen
    - Anwendungen
  - Globale Zustände und deren Anwendung
    - Distributed Snapshot
    - Der Begriff des Cut
    - Der Algorithmus von Lamport und Chandy
- Auswahlalgorithmen
  - Bully-Algorithmus
  - Ring-Algorithmus
- Gegenseitiger Ausschluss
  - Zentraler Algorithmus
  - Verteilter Algorithmus
  - Token-Ring-Algorithmus
- Transaktionen in Verteilten Systemen
  - Flache Transaktionen
  - Geschachtelte Transaktionen
  - Verteilte Transaktionen

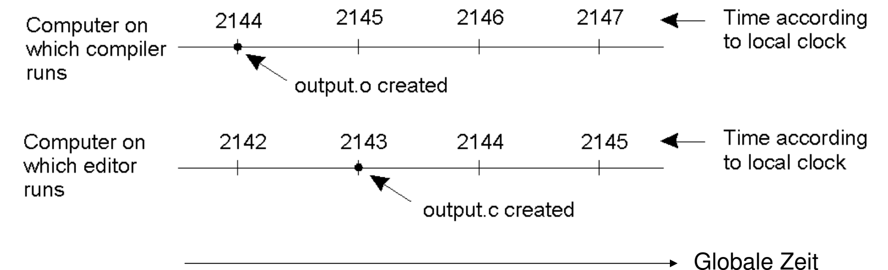
INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete

Zeit in Verteilten Systemen

- Sehr wichtiges weiteres Problem: wie koordinieren und synchronisieren Prozesse sich, z.B.
  - Beim Zugriff auf gemeinsam verwendete Ressourcen
  - Bei der Feststellung, welcher Prozess ein bestimmtes Ereignis zuerst ausgelöst hat (verteilte Online-Auktionen)
- Für viele dieser Algorithmen ist ein gemeinsames Verständnis der „Zeit" in allen beteiligten Knoten notwendig.
- Dies ist in einem zentralisierten System kein Problem, da es dort nur eine Zeitquelle gibt.
- In verteilten Systemen hat jedoch jeder Knoten seine eigene Zeitquelle und damit u.U. eine andere Uhrzeit.
- Problem?

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete

Beispiel: Verteilte SW-Entwicklung



- Die Quelldatei ist augenscheinlich älter als die Zieldatei. Ergebnis: Sie wird bei einem erneuten Make *nicht* neu übersetzt.

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete

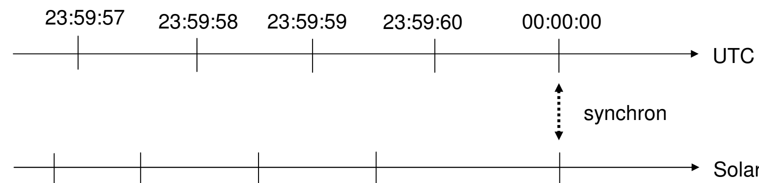
5-5 Zeit

- Jeden Tag gegen Mittag erreicht die Sonne ihren höchsten Punkt am Himmel.
- Die Zeitspanne zwischen zwei aufeinanderfolgenden Ereignissen dieses Typs heißt Tag (genauer gesagt: Sonnentag).
- Eine *Sonnensekunde* ist 1/86400 dieser Spanne.
- Die Zeitmessung findet heute auch mit Hilfe von Atomuhren statt: eine Sonnensekunde ist die Zeit, die ein Cäsium-133-Atom benötigt, um 9.192.631.770 mal zu schwingen.
- Am 1.1.1958 entsprach diese Atomsekunde genau einer Sonnensekunde. Danach gibt es jedoch Abweichungen aufgrund von astronomischen Effekten.

5-6 TAI und UTC

- Diese „Temps Atomique International“ (TAI) stimmt daher nicht 100% mit der Sonnenzeit überein, weshalb man eine Korrektur anwendet.
- Wenn die beiden Zeiten mehr als 800 ms auseinander liegen, dann wird eine Schaltsekunde („leap second“) eingeführt bzw. gelöscht.
- Diese neue Zeit heißt UTC – Universal Coordinated Time.
- Es gibt weltweit eine ganze Reihe von UTC-Sendern, für die man inzwischen recht preisgünstige Empfänger erhalten kann.

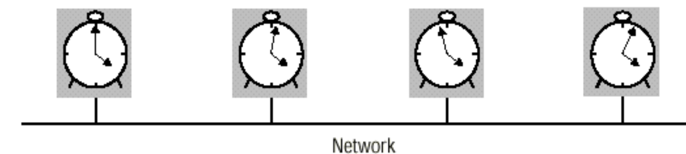
5-7 Schaltsekunden



- Wenn die beiden Zeiten nicht mehr „synchron genug“ laufen, werden sie durch eine Schaltsekunde angepasst.
- Meist wird die Zeit Ende oder Mitte des Jahres angepasst.
- Weitere Infos: http://www.ptb.de/de/wegweiser/infoszurzeit/_index.html

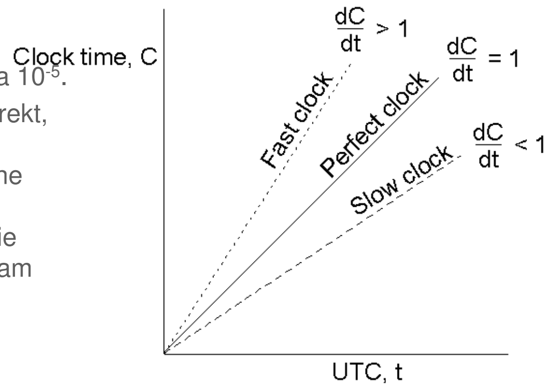
5-8 Zeit in verteilten Systemen

- Computer haben eine lokale Uhr, die H mal pro Sekunde einen Interrupt auslöst. Die Interrupts werden gezählt und messen die Zeit.
- Problem: die Uhren unterschiedlicher Computer zeigen unterschiedliche Zeiten an!
- Problem 1: unterschiedliche Startzeiten, kann relativ leicht gelöst werden
- Problem 2: unterschiedliche Laufzeiten!



Genauigkeit von Uhren

- Der Wert der Uhr von Maschine p zum UTC-Zeitpunkt t is $C_p(t)$.
- Chips haben eine Genauigkeit von etwa 10^{-5}.
- Eine Uhr arbeitet korrekt, wenn sie die vom Hersteller angegebene maximale Driftrate ρ einhält, auch wenn sie dann etwas zu langsam oder zu schnell ist.

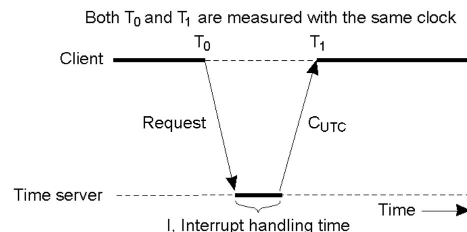


Uhrensynchronisation

- Folge: zu einem Zeitpunkt Δt nach der Synchronisation zweier Uhren können die beiden Uhren maximal 2ρΔt auseinander liegen.
- Will man sicherstellen, dass zwei Uhren niemals mehr als δ auseinander liegen, muss man die Uhren mindestens alle **δ/2ρ** Sekunden synchronisieren.
- Heute hat nicht jeder Rechner einen UTC-Empfänger, so dass man keine vollständige externe Synchronisation durchführen kann.
- Vielmehr gibt es eine Reihe von Algorithmen, die auf der Verwendung weniger Time-Server beruhen.

Der Algorithmus von Christian (1989)

- Es wird die Existenz eines UTC-Empfängers im System angenommen, der dann als *Time-Server* fungiert.
- Jede andere Maschine sendet mind. alle δ/2ρ Sekunden ein Time Request an den Server, der so schnell wie möglich mit der aktuellen UTC antwortet.



Algorithmus von Christian

- Nun könnte man die Uhr auf die erhaltene UTC-Zeit setzen.
- Erstes ABER:
  - Zeit darf niemals rückwärts laufen, so dass ein Zeitpunkt zweimal auftauchen kann! Wenn der Anfragende eine schnelle Uhr hat, dann kann UTC u.U. stark hinterher laufen. Ein Setzen auf den UTC-Wert wäre dann falsch.
- Lösung: die Uhren werden nicht auf einmal angepasst, sondern graduell, so lange, bis die richtige Zeit erreicht ist (ein Clock-Tick wäre nicht mehr z.B. 10ms, sondern nur noch 9ms)

5-13 Algorithmus von Christian

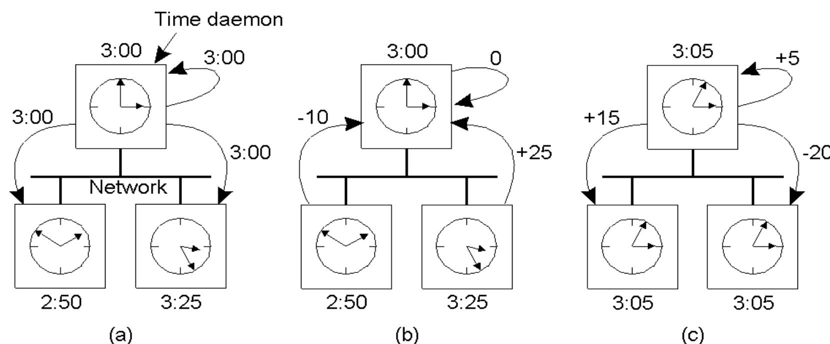
- Zweites ABER:
  - Die Signallaufzeit für die Nachrichten ist größer als 0. Wenn die Antwort kommt, ist sie praktisch schon veraltet.
- Lösung von Christian: versuche die Signallaufzeit zu messen
- Beste Abschätzung, wenn sonstige Informationen fehlen: $(T_1-T_0)/2$
- Wenn die ungefähre Zeit I bekannt ist, die der Time-Server zur Abarbeitung des Requests benötigt: $(T_1-T_0-I)/2$
- Verbesserung des Ergebnisses: messe diesen Wert häufig und nehme den Durchschnitt, aber lasse „Ausreißer“ aus der Wertung

5-14 Der Berkeley-Algorithmus

- Entwickelt für eine Gruppe von Berkeley-UNIX-Rechnern ebenfalls 1989
- Sinnvoll, wenn kein UTC-Empänger zur Verfügung steht.
- Ein Rechner ist der Koordinator (=Time-Server)
- Dieser Server ist im Gegensatz zu dem Server von Christian *aktiv*. Er fragt bei allen anderen Rechnern nach der aktuellen Zeit und bildet einen Durchschnittswert.
- Dieser wird dann als neuer aktueller Uhrenwert an alle anderen gegeben.

5-15 Der Berkeley-Algorithmus

- Alle anderen sollten dann ihre Uhren entsprechend langsamer oder schneller laufen lassen, bis alle Uhren in etwa mit dem Durchschnittswert

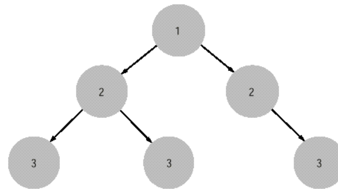


5-16 Network Time Protocol NTP

- Entwickelt in den 80er Jahren, inzwischen ein Internet RFC (mit diversen Verbesserungen)
- Ziele:
  - Clients sollen sich möglichst genau mit UTC synchronisieren können, trotz stark schwankender Übertragungsverzögerungen im Netz
  - Bereitstellung eines zuverlässigen Dienstes mittels Redundanz
  - Clients sollen in der Lage sein, sich oft zu synchronisieren, Skalierbarkeit wird damit ein Thema

5-17 Funktionsweise von NTP

- Der NTP-Dienst wird von einem Netzwerk von Servern erbracht.
- Die „primary server“ sind direkt mit der UTC-Quelle verbunden.
- Die „secondary server“ synchronisieren sich mit den „primary servers“ etc.



Note: Arrows denote synchronization control, numbers denote strata.

5-18 Funktionsweise von NTP

- Das Server-Netzwerk ist rekonfigurierbar, um auf Fehler entsprechend reagieren zu können.
- Die Server tauschen häufig Nachrichten aus, um Netzwerkverzögerungen und Uhrungenauigkeiten zu messen.
- Basierend auf diesen Messungen lässt sich die Qualität eines Servers abschätzen und Clients können die beste Quelle wählen.

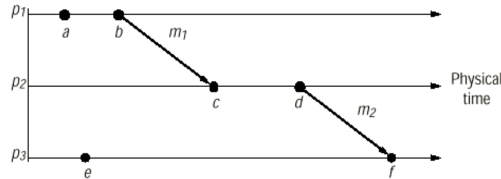
5-19 Logical Time

- Es ist generell unmöglich, physikalische Uhren in einem verteilten System absolut zu synchronisieren.
- Damit ist es auch nicht möglich, basierend auf diesem System die Reihenfolge zweier beliebiger Ereignisse zu bestimmen.
- Für einige Anwendungen benötigt man jedoch genau diese Information, dafür aber keinen Bezug zur realen Zeit.
- Die Lösung dafür: logische Zeit (logical time)
  - Wenn zwei Ereignisse im selben Prozess stattfinden, dann fanden sie in der Reihenfolge ihrer Beobachtung statt.
  - Wenn eine Nachricht zwischen zwei Prozessen ausgetauscht wird, dann ist das Sendereignis immer vor dem Empfangsereignis.

5-20 Die *Happened-Before*-Beziehung

- Aus diesen Beobachtungen machte Lamport die Happened-Before-Relation „ $\rightarrow$“oder auch die „relation of causal ordering“:
  - Wenn in einem Prozess p_i gilt: $a \rightarrow_i b$, dann gilt allgemein für das System: $a \rightarrow b$
  - Für jede Nachricht m gilt: send(m) $\rightarrow$ receive(m), wobei send(m) das Sendeereignis im sendenden Prozess und receive(m) das Empfangsereignis im empfangenden Prozess darstellt
  - Wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, dann gilt auch $a \rightarrow c$.
- Ereignisse, die nicht in dieser Beziehung stehen, werden als *nebenläufig* bezeichnet.

5-21 Beispiel



- $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow d$, $d \rightarrow f$, und also $a \rightarrow f$, aber nicht $a \rightarrow e$
- Wenn $a \rightarrow b$ gilt, wird eine kausale oder möglicherweise kausale Beziehung angenommen.

5-22 Praktische Umsetzung

- Jeder Prozess P_i hat nun eine logische Uhr, die beim Auftreten eines Ereignisses a abgelesen wird und den Wert $C_i(a)$ liefert.
- Dieser Wert muss so angepasst werden, dass er als $C(a)$ eindeutig im ganzen verteilten System ist.
- Ein Algorithmus, der die logischen Uhren entsprechend richtig stellt, muss folgendes umsetzen:
  Wenn $a \rightarrow b$, dann $C(a) < C(b)$.

5-23 Umsetzung von Happens-Before

- Prozesse wenden deshalb den folgenden Algorithmus an, um ihre Uhren richtig zu stellen:
  - C_i wird vor jedem neuen Ereignis in P um eins erhöht: $C_i := C_i + 1$. Der neue Wert ist der „Timestamp“ des Ereignisses.
  - Wenn ein Prozess P_i eine Nachricht m sendet, dann sendet er den Wert $t = C_i$ mit.
  - Bei Erhalt von (m,t) berechnet P_j den neuen Wert $C_j := \max(C_j, t)$ und wendet dann die erste Regel an, um den Timestamp für receive(m) festzulegen.

5-24 Lamports Timestamps

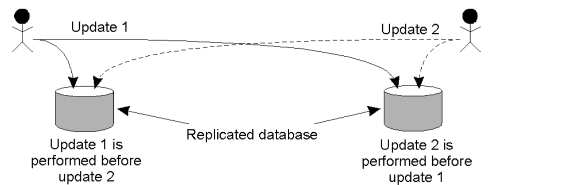
0		0		0
6	A	8		10
12		16		20
18		24	B	30
24		32		40
30		40		50
36		48	C	60
42		56		70
48	D	64		80
54		72		90
60		80		100

0		0		0
6	A	8		10
12		16		20
18		24	B	30
24		32		40
30		40		50
36		48	C	60
42		61		70
48	D	69		80
70		77		90
76		85		100

a) Drei Prozesse mit unterschiedlichen Clock Rates.
b) Lamports Algorithmus löst das Problem.

Beispiel: Replizierte Datenbank

- Zwei Kopien der selben Datenbank
- Kunde überweist $100 auf sein $1000-Konto.
- Gleichzeitig überweist ein Bankangestellter 1% Zinsen auf den aktuellen Kontostand.
- Die Operationen werden per Multicast an beide DBS geschickt und kommen dort in unterschiedlicher Reihenfolge an, Ergebnis: Inkonsistenz ($1111 vs $1110)

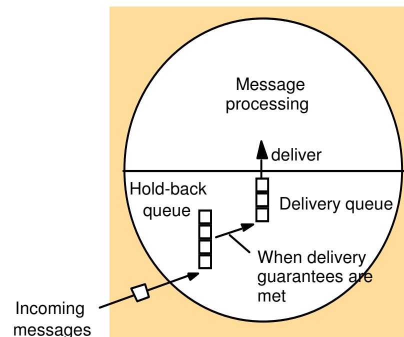


Lösung: Totally-Ordered Multicast

- Anforderung bei solchen Problemen: alle Kopien müssen immer im gleichen Zustand sein
- Lösung dafür: totally-ordered multicast, d.h., alle Nachrichten werden bei allen Empfängern in derselben Reihenfolge ausgeliefert
- Annahmen:
  - Reihenfolgeerhaltung
  - Nachrichten gehen nicht verloren
- Implementierung??

Totally-Ordered Multicast

- Jeder Prozess hat eine lokale Warteschlange für eingehende Nachrichten, geordnet nach den Timestamps, die Hold-Back Queue. Elemente hierin dürfen nicht ausgeliefert werden.
- Außerdem hat jeder Prozess eine Delivery-Queue. Elemente hierin werden entsprechend ihrer Reihefolge an die Anwendung ausgeliefert.

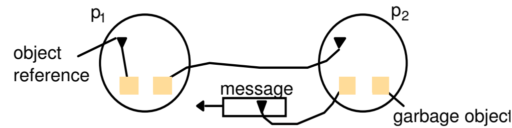


Globale Systemzustände

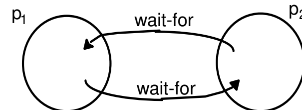
- Es gibt eine Reihe von Gelegenheiten, bei denen man gern über den Gesamtzustand des verteilten Systems Bescheid wüsste
- Der Gesamtzustand des Systems besteht aus
  - Den lokalen Zuständen der Einzelkomponenten (der Prozesse) und
  - Allen Nachrichten, die sich zur Zeit in der Übertragung befinden.
- Diesen Zustand exakt zur selben Zeit bestimmen zu können ist so unmöglich wie die exakte Synchronisation von Uhren – es lässt sich kein globaler Zeitpunkt festlegen, an dem alle Prozesse ihre Zustände festhalten sollen.

5-29 Anwendungen des globalen Zustands

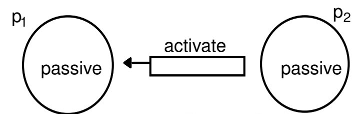
a. Garbage collection



b. Deadlock



c. Termination

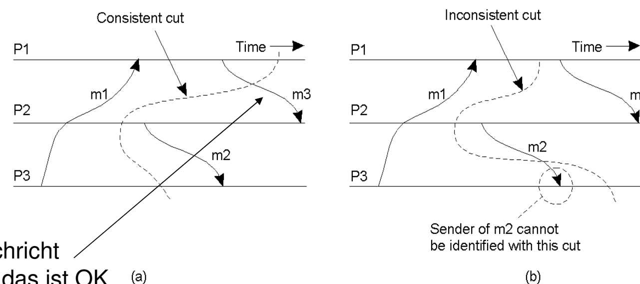


5-30 Distributed Snapshot

- Wie kann man nun den globalen Zustand eines verteilten Systems ermitteln?
- Lösung von Chandy und Lamport (1985): Distributed Snapshot:
  - ermittle einen Zustand, in dem das System möglicherweise war,
  - der aber auf jeden Fall konsistent ist
- Konsistenz bedeutet insbesondere: wenn festgehalten wurde, dass Prozess P eine Nachricht m von einem Prozess Q empfangen hat, dann muss auch festgehalten sein, dass Q diese Nachricht geschickt hat. Sonst kann das System nicht in diesem Zustand gewesen sein.

5-31 Consistent und Inconsistent Cut

- Definition der Konsistenz über den sog. „cut“, der für jeden Prozess das letzte aufgezeichnete Ereignis angibt.



m3 ist Nachricht in Transit; das ist OK

5-32 Formale Definition des Cut

- Gegeben sei ein System $\wp$ von N Prozessen p_i (i=1,...,N).
- Betrachtet man nun den globalen Zustand $S=(s_1,...,s_N)$ des Systems, dann ist die Frage, welche globalen Zustände möglich sind.
- In jedem Prozess findet eine Serie von Ereignissen statt, womit jeder Prozess mittels der Geschichte seiner Ereignisse charakterisiert werden kann:
  $history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, ...>$
- Jeder endliche Präfix der Geschichte eines Prozesses wird bezeichnet mit
  $h_i^k = <e_i^0, e_i^1, e_i^2, ..., e_i^k>$

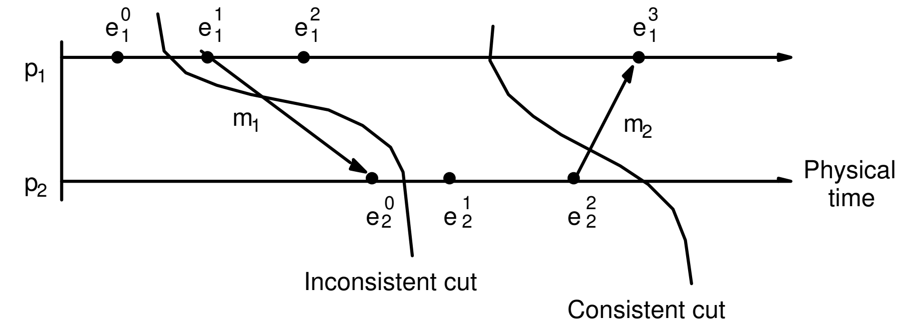
Formale Definition des Cut

- Ein Cut ist damit definiert wie folgt

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N}$$

- Der Zustand s_i aus dem globalen Zustand ist dann genau derjenige von p_i, der durch das Ausführen des letzten Ereignisses im Cut erreicht wird, also von $e_i^{c_i}$
- Die Menge $\{e_i^{c_i} : i = 1,2,\ldots N\}$ wird als Frontier des Cuts bezeichnet.

Beispiel



Frontier: $< e_1^0, e_2^0 >$ $< e_1^2, e_2^2 >$

Definition des konsistenten Cut

- Ein Cut ist dann konsistent, wenn er für jedes Ereignis, das er enthält, auch alle Ereignisse enthält, die zu diesem Ereignis in der Happened-Before-Relation (s. Zeit in verteilten Systemen) stehen:

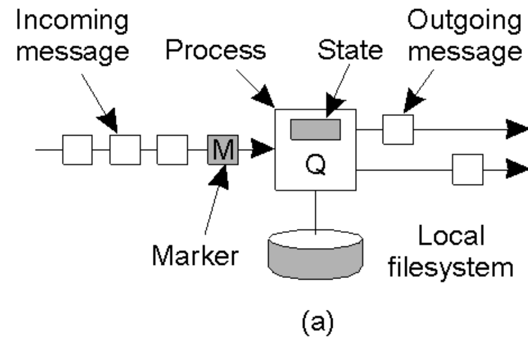
$$\forall e \in C, f \rightarrow e \Rightarrow f \in C$$

- Ein globaler Zustand ist konsistent, wenn er mit einem konsistenten Cut korrespondiert.

Lamport/Chandy-Algorithmus

- Prozesse sind mittels Punkt-zu-Punkt-Kanälen verbunden.
- Ein oder mehrere Prozesse starten den Algorithmus zur Feststellung eines Systemzustands, so dass gleichzeitig immer mehrere Snapshots erstellt werden können.
- Das System läuft unterdessen ungehindert weiter.
- Die Prozesse verständigen sich über Markierungsnachrichten über die Notwendigkeit der Speicherung eines Systemzustands.

Prozessmodell für den Algorithmus



Der Algorithmus

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel *c*:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of *c* as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of *c* as the set of messages it has received over *c* since it saved its state.

end if

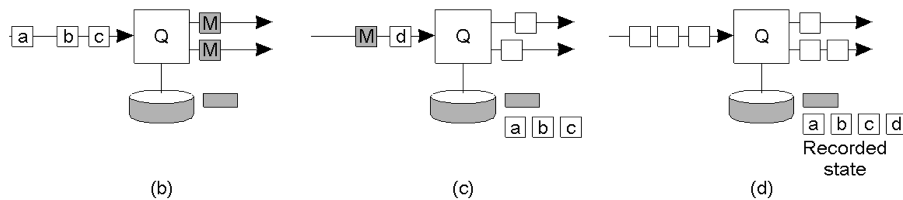
Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel *c*:

p_i sends one marker message over *c*

(before it sends any other message over *c*).

Ablauf des Algorithmus



b) Prozess Q erhält zum ersten Mal einen Marker und hält seinen lokalen Zustand fest
c) Q hält alle ankommenden Nachrichten fest
d) Q erhält einen Marker auf seinem Eingangskanal und stoppt die Aufzeichnung für diesen Kanal

Ende des Algorithmus

- Wenn Q einen Marker auf allen Eingangskanälen erhalten und verarbeitet hat, ist für diesen Prozess der Algorithmus beendet.
- Q sendet dann seinen lokalen Zustand sowie die aufgezeichneten Nachrichten für alle Eingangskanäle an den initiierenden Prozess.
- Dieser wertet schließlich das Ergebnis entsprechend aus, analysiert also z.B. bestimmte Zustandsprädikate.
- Man kann beweisen, dass dieser Algorithmus immer einen konsistenten Cut erzeugt.

5-41 Zeit in Verteilten Systemen: Zusammenfassung

- Die „gleiche" Zeit in allen Rechnern eines Systems kann nicht erreicht werden.
- Synchronisationsalgorithmen für die physikalische Zeit funktionieren recht gut (NTP).
- In vielen Anwendungen genügt Wissen über die Ordnung von Ereignissen ohne quantitative Zeitangaben → Verwendung logischer Zeit.
- Es ist unmöglich, einen globalen Systemzustand „gleichzeitig" aufzuzeichnen.
- Der Algorithmus von Lamport und Chandy macht einen „Distributed Snapshot".
- Dieser Snapshot hat möglicherweise so nie genau als Systemzustand stattgefunden, aber er ist konsistent.

5-42 Auswahlalgorithmen

- In vielen verteilten Algorithmen benötigt man einen Prozess, der eine irgendwie geartete besondere Rolle spielt, z.B. als Koordinator, Initiator oder Monitor.
- Die Aufgabe eines Auswahlalgorithmus ist es, einen Prozess unter vielen gleichartigen zu bestimmen, der diese Rolle übernimmt.
- Wichtigstes Ziel: am Ende der Wahl sind sich alle darüber einig, wer der neue Koordinator ist.

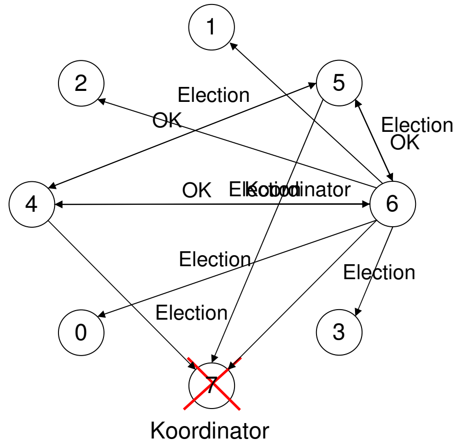
5-43 Auswahlalgorithmen

- Vorgehen:
  - Jeder Prozess hat eine Nummer, die allen anderen Prozessen bekannt ist.
  - Kein Prozess weiß, welcher andere Prozess gerade funktioniert oder nicht läuft.
  - Alle Algorithmen wählen den Prozess mit der höchsten Nummer aus. Der Weg kann sehr unterschiedlich sein.
- Bekannte Algorithmen:
  - Bully-Algorithmus
  - Ring-Algorithmus

5-44 Der Bully-Algorithmus

- Wenn ein Prozess P feststellt, dass der augenblickliche Koordinator nicht mehr reagiert, startet er den Auswahlprozess:
  - P schickt eine ELECTION-Nachricht an alle Prozesse mit höherer Nummer
  - Bekommt er keine Antwort, ist er der neue Koordinator.
  - Bekommt er eine Antwort, ist seine Aufgabe erledigt. Der Antwortende übernimmt seine Arbeit.

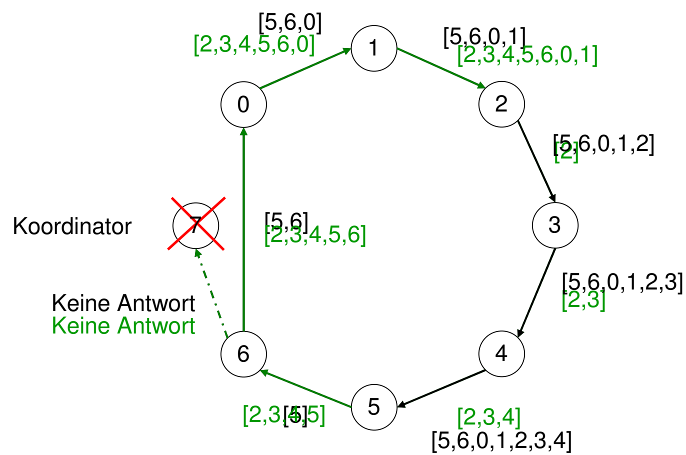
Beispiel: Bully-Algorithmus



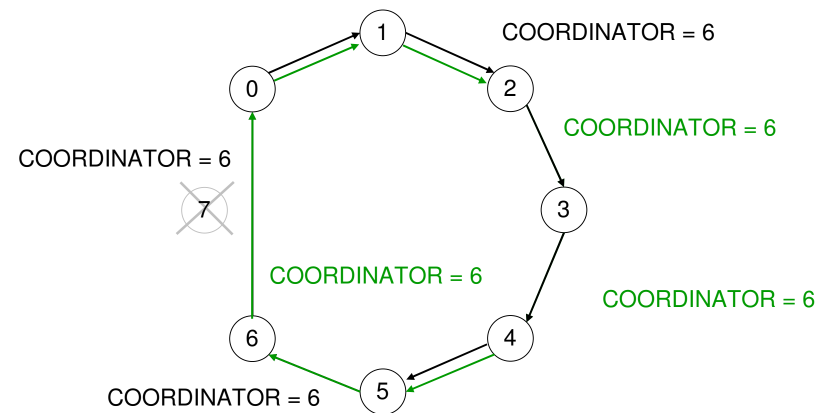
Ein Ring-Algorithmus

- Für diesen Algorithmus sind die Prozesse logisch in Form eines Rings organisiert, d.h., jeder Prozess besitzt einen Vorgänger und einen Nachfolger entsprechend aufsteigender Prozessnummern.
- Wenn ein Prozess feststellt, dass der Koordinator nicht mehr funktioniert, sendet er eine ELECTION-Nachricht auf den Ring, in die er sich als ersten einträgt.
- Jeder weitere aktive Prozess fügt sich selbst in die Liste ein.
- Wenn die Nachricht wieder beim Initiator eintrifft, wandelt er diese in eine KOORDINATOR-Nachricht um, die den neuen Koordinator und die aktiven Mitglieder enthält.

Beispiel: Ring-Algorithmus



Beispiel: Ring-Algorithmus



5-49 Auswahlalgorithmen: Zusammenfassung

- Bully- und Ring-Algorithmus lösen das eingangs beschriebene Auswahlproblem in Verteilten Systemen.

5-50 Gegenseitiger Ausschluss

- Wenn sich zwei oder mehrere Prozesse beim Zugriff auf gemeinsame Daten koordinieren müssen, um die Konsistenz der Daten zu erhalten, geschieht dies am einfachsten über das Konzept der *kritischen Region*.
- Jeweils nur ein Prozess darf in einer kritischen Region aktiv sein, d.h., es wird gegenseitiger Ausschluss (mutual exclusion) erreicht.
- In Ein-Prozessor-Systemen werden Semaphore oder Monitore verwendet, um gegenseitigen Ausschluss zu implementieren (s. Vorlesung Betriebssysteme und Netze).

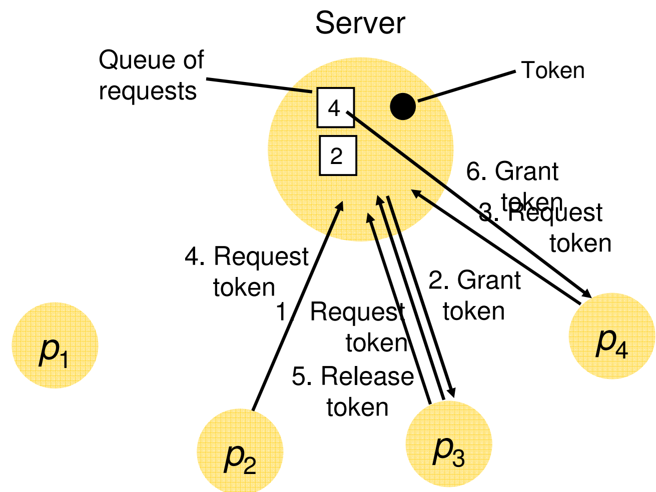
5-51 Mutual Exclusion in verteilten Systemen

- Wie kann man Mutual Exclusion in verteilten Systemen umsetzen?
- Wir betrachten drei Algorithmen:
  - Ein zentralisierter Algorithmus
  - Ein verteilter Algorithmus
  - Ein Token-Ring-Algorithmus

5-52 Ein zentralisierter Algorithmus

- Einer der Prozesse wird zum Koordinator für eine kritische Region bestimmt.
- Alle anderen müssen sich nun zuerst an den Koordinator wenden, bevor sie die entsprechende Region betreten.
- Wenn die kritische Region frei ist, erhält der Prozess das OK vom Server. Nach Abarbeitung der Aufgaben gibt der Prozess dieses Token zurück.
- Ist die Region nicht frei, wird der anfragende Prozess in eine Warteschlange aufgenommen. Er erhält erst das Token, wenn alle Prozesse vor ihm bedient wurden.

Beispiel: Mutual-Exclusion-Server



Eigenschaften des Algorithmus

- Mutual Exclusion wird erreicht: es ist immer nur ein Prozess im kritischen Bereich, da der Server immer nur ein Token vergibt
- Fair: Tokens werden in der Reihenfolge der Anfrage vergeben
- Einfach zu implementieren
- Nur 3 Nachrichten pro Zugang zur kritischen Region
- Koordinator ist single point of failure, d.h., Problem, wenn der Koordinator zusammenbricht
- Prozesse, die nach einem Request blockieren, können nicht zwischen einem „toten" Koordinator und einer langen Warteschlange unterscheiden.
- Performance Bottleneck in großen Systemen

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Ein verteilter Algorithmus

- Der folgende Algorithmus besitzt keinen ausgewiesenen Koordinator.
- Alle Prozesse verständigen sich über Multicast-Nachrichten.
- Jeder Prozess besitzt eine logische Uhr (s. Kapitel zu Zeit in VS).
- Wenn ein Prozess eine kritische Region betreten will, sendet er ein Request an alle anderen Prozesse.
- Erst wenn alle Prozesse ihr OK gegeben haben, kann der Prozess die kritische Region betreten.

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

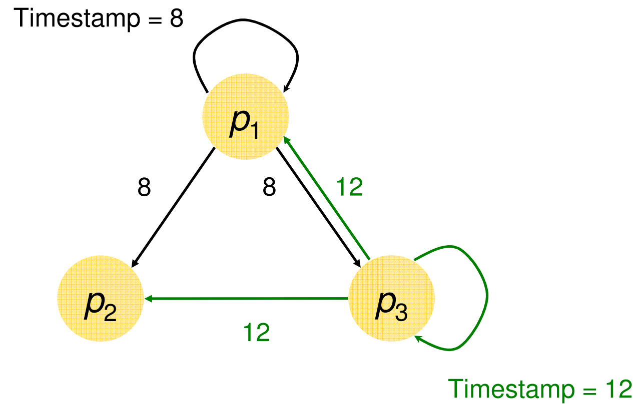
Der Algorithmus (Ricart and Agrawala, 1981)

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = $(N - 1)$);
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

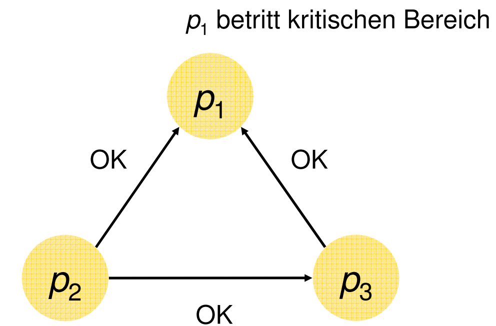
INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Beispiel: Verteilter Algorithmus (1)



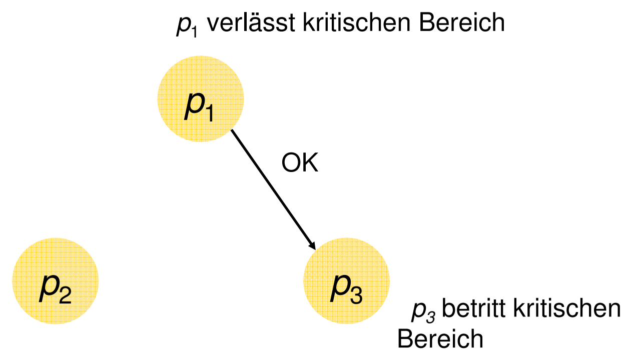
Prozess p_1 und p_3 wollen gleichzeitig in den kritischen Bereich.

Beispiel: Verteilter Algorithmus (2)



Prozess p_1 hat den niedrigeren Timestamp und gewinnt.

Beispiel: Verteilter Algorithmus (3)



Wenn Prozess p_1 fertig ist, gibt er den kritischen Bereich frei und sendet ebenfalls ein OK an p_3.

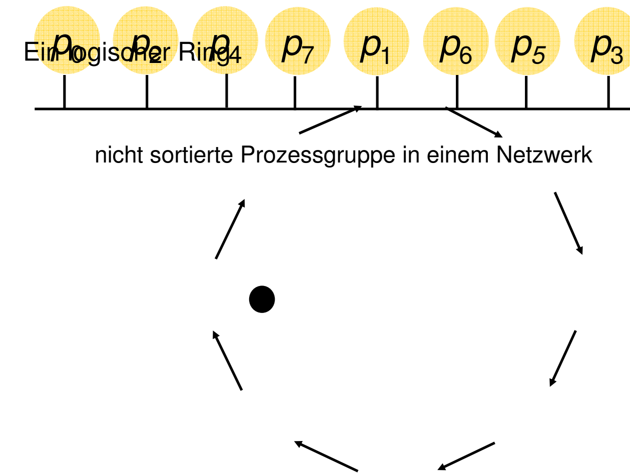
Eigenschaften des Algorithmus

- Der single-point-of-failure wurde ersetzt durch n points-of-failure. Wenn ein Prozess nicht mehr arbeitet, funktioniert das System nicht mehr.
- Dieses Problem könnte durch explizite Verwendung eines Request-Reply-Protokolls ersetzt werden (jede Nachricht wird sofort bestätigt). Wenn keine Bestätigung kommt, ist der Prozess nicht mehr aktiv.
- Jeder Prozess muss immer bei der Entscheidung mitwirken, obwohl er evtl. gar kein Interesse an der kritischen Region hat.
- Verbesserung: eine einfache Mehrheit genügt
- Insgesamt ist der Algorithmus langsamer, komplizierter, teurer und weniger robust, aber, wie Tanenbaum sagt, :“Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

Ein Token-Ring-Algorithmus

- Die Prozesse in einem lokalen Netz werden in einer logischen Ringstruktur entsprechend der Prozessnummern organisiert.
- Ein Token kreist; wer das Token besitzt, darf in den kritischen Bereich, allerdings nur einmal.

Ein Token-Ring-Algorithmus



Eigenschaften des Algorithmus

- Korrektheit ist ebenfalls leicht zu sehen. Nur ein Prozess hat das Token zur selben Zeit.
- Kein Prozess wird ausgehungert, da die Reihenfolge durch den Ring bestimmt ist.
- Maximal muss ein Prozess warten, bis alle anderen Prozesse einmal im kritischen Bereich waren.
- Verlorene Token erfordern Neugenerierung durch Koordinator.
- Verlust eines Tokens ist schwer zu erkennen, da es sich auch um einen sehr langen Aufenthalt in einem kritischen Bereich handeln kann.
- Tote Prozesse müssen erkannt werden.

Vergleich

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

5-65 Gegenseitiger Ausschluss: Zusammenfassung

- Gegenseitiger Ausschluss im verteilten System ist schwieriger zu erreichen als in einem Ein-Prozessor-System.
- Es existieren verschiedene Algorithmen mit unterschiedlicher Bedeutung für die Praxis.
- Vollkommene Verteilung bringt hier viele Nachteile mit sich.

5-66 Transaktionen

- Transaktionen und gegenseitiger Ausschluss hängen eng miteinander zusammen
- Ziel: eine bestimmte Ressource soll nur von einem Client zur selben Zeit benutzt werden
- Ganz generell haben Transaktionen dasselbe Ziel: Schutz von Ressourcen vor gleichzeitigem Zugriff
- Transaktionen gehen jedoch noch wesentlich weiter:
  - Es ist möglich, auf mehrere Ressourcen in einer einzigen atomaren Operation zuzugreifen.
  - Diverse Arten von Fehlern können abgefangen werden, so dass Prozesse in den Zustand vor Beginn der Ausführung einer Transaktion zurückgesetzt werden.

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

5-67 Transaktionen

- Transaktionen bestehen aus einer Folge von Operationen (Anfragen an Server), für die bestimmte Eigenschaften gelten – die **ACID**-Eigenschaften.
- Beispiel: Flugbuchung
  - Ein Kunde will von Hamburg nach Dallas fliegen; leider gibt es keinen Nonstop-Flug, sondern man muss zweimal umsteigen.
  - Alle drei Flüge müssen einzeln gebucht werden.
  - Der Kunde hat natürlich nur Interesse an einer Buchung, wenn alle drei Flüge buchbar sind.
  - → Implementierung durch eine Transaktion

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

5-68 Die ACID-Eigenschaften

- ACID ist ein von Härder und Reuter (zwei Deutsche, immerhin) vorgeschlagenes Acronym.
- Bedeutung:
  - **A**tomicity: alle Operationen der Transaktion oder keine
  - **C**onsistency: eine Transaktion überführt das System von einem konsistenten Zustand in den anderen
  - **I**solation: jede Transaktion muss von der Ausführung anderer Transaktionen unabhängig bleiben
  - **D**urability: wenn eine Transaktion abgeschlossen ist, müssen die Ergebnisse auf permanentem Speicher gesichert werden

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Transaktions-Primitive

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

- Mit diesen Primitiven lassen sich alle für die weiteren Untersuchungen interessanten Operationen einer Transaktion unabhängig von der spezifischen Anwendung ausdrücken.

Beispiel Flugbuchung

- Erfolgreiche Transaktion:

BEGIN_TRANSACTION
reserve HH -> FRA
reserve FRA -> JFK
reserve JFK -> DAL
END_TRANSACTION

- Alle Flüge gebucht (COMMIT)

- Abgebrochene Transaktion, da JFK -> DAL ausgebucht ist:

BEGIN_TRANSACTION
reserve HH -> FRA
reserve FRA -> JFK
reserve JFK DAL
ABORT_TRANSACTION

- **Keine** Flüge gebucht

Klassifizierung von Transaktionen

- Traditionell wurden und werden immer noch am häufigsten so genannte **„flache" Transaktionen** verwendet (flat transactions).
- Probleme:
  - Flache Transaktionen sind sehr strikt und erlauben kein Commit von Teilergebnissen.
  - Sie sind möglicherweise nicht sehr effizient, da alle Operationen sequentiell ausgeführt werden.
  - Daten sind möglicherweise verteilt, so dass mehrere Rechner beteiligt werden müssen.
- Lösungen:
  - Geschachtelte Transaktionen
  - Verteilte Transaktionen

Geschachtelte Transaktionen

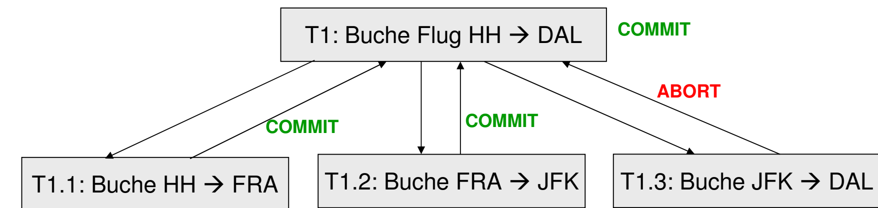
- Geschachtelte Transaktionen erweitern das bisherige (flache) Transaktionsmodell, indem sie gestatten, dass Transaktionen aus anderen Transaktionen zusammengesetzt sind.
- Die Transaktion auf der höchsten Ebene wird als *top-level transaction* bezeichnet, die anderen als *subtransactions*.
- Die Verwendung geschachtelter Transaktionen bringt ein neues Problem mit sich: *was passiert, wenn eine Teiltransaktion abgebrochen wird?*

5-73 Vorteile geschachtelter Transaktionen

- Zusätzliche Nebenläufigkeit:
  - Subtransactions auf der selben Hierarchieebene können nebenläufig ausgeführt werden
  - Bankenbeispiel: die Operation branchTotal() muss für sämtliche Konten die Methode getBalance() aufrufen. Man könnte jeden dieser Aufrufe als Untertransaktion starten
- Unabhängiges Commit oder Abort:
  - Dadurch werden Transaktionen potentiell robuster (hängt von der Anwendung ab)
  - Die Elterntransaktion muss jeweils entscheiden, welche Folge ein Abort der Untertransaktion haben soll.

5-74 Beispiel für teilweise Commit

- Betrachte das Beispiel von weiter vorn.
- Angenommen, der Flug JFK → DAL ist nicht buchbar.
- Dann kann es trotzdem sinnvoll sein, die Flüge HH → FRA und FRA → JFK zu buchen, da dies vielleicht schon schwierig war
- Dies ist im geschachtelten Modell möglich.



5-75 Regeln für das Commitment geschachtelter Transaktionen

- Eine Transaktion darf nur abgeschlossen werden, wenn ihre Untertransaktionen abgeschlossen sind.
- Wenn eine Untertransaktion abschließt, entscheidet sie unabhängig, entweder provisorisch zu comitten oder endgültig abzubrechen.
- Wenn eine Elterntransaktion abbricht, werden auch alle Subtransaktionen abgebrochen.
- Wenn eine Subtransaktion abbricht, entscheidet die Elterntransaktion, was weiter geschieht.
- Wenn eine Elterntransaktion commited ist, dann können alle provisorisch committeten Untertransaktionen ebenfalls committet werden.

5-76 Verteilte Transaktionen

- Geschachtelte Transaktionen entstehen durch **logische Partitionierung** größerer Transaktionen.
- Die Subtransaktionen werden unter Umständen auf unterschiedlichen Rechnern ausgeführt.
- Die Subtransaktionen sind auf der Blattebene logisch **nicht mehr teilbare Einheiten** (Beispiel: Reservierung eines einzelnen Fluges).
- Trotzdem können diese auf mehreren Rechnern arbeiten.
- Man spricht dann von **verteilten Transaktionen**.

5-77

Concurrency Control

- Um die Performance zu steigern, werden Transaktionen möglichst nebenläufig ausgeführt.
- Dabei können Konflikte entstehen, wenn zwei Transaktionen auf dasselbe Datum zugreifen wollen
  - READ-WRITE: eine Transaktion will ein Datum schreiben, das eine andere gerade liest
  - WRITE-WRITE: zwei Transaktionen wollen dasselbe Datum schreiben
  - READ-READ ist kein Konflikt!
- Concurrency Control verhindert Inkonsistenzen.

5-78

Beispiel: Das “lost update”-Problem

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

5-79

Locking

- Älteste und am weitesten verbreitete Form der Concurrency Control
- Einfachste Variante: exklusive Locks
  - Wenn ein Prozess Zugriff auf eine Ressource (ein Datenobjekt) benötigt, bittet er den Scheduler (über den Transaction Manager) um ein exklusives Lock
  - Wenn er es erhalten hat und seine Arbeit anschließend beendet hat, gibt er das Lock wieder zurück
- Vergabe der Locks in einer Weise, dass nur korrekte Schedules entstehen
- Beispiel nächste Seite: U muss auf T warten

INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

5-80

Two-Phase Locking

- Bekannter Algorithmus, von dem bewiesen ist, dass er korrekte Schedules erstellt, wenn sich alle Transaktionen daran halten.
- Ablauf:
  - Bei Erhalt einer Operation op(T,x): Prüfung ob Konflikt mit anderen Operationen, für die schon ein Lock vergeben ist. Falls ja, wird op(T,x) verzögert, falls nein, bekommt T das Lock für x.
  - Der Scheduler gibt niemals ein Lock für x ab, außer der Data Manager bestätigt die Ausführung der Operation.
  - Wenn der Scheduler erst einmal ein Lock für T abgegeben hat, wird er niemals ein neues Lock für irgendein Datum für T reservieren. Jeder Versuch von T in dieser Hinsicht bricht T ab.

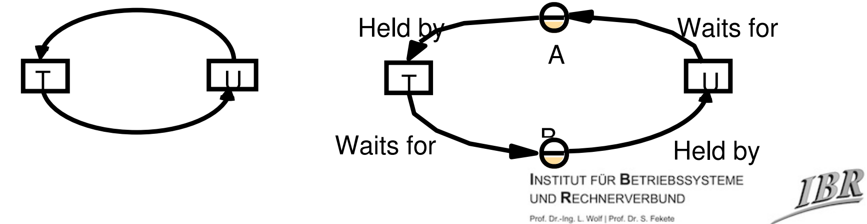
INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Deadlocks

- Ein Deadlock ist ein Zustand, in dem jedes Mitglied einer Gruppe von Transaktionen darauf wartet, dass ein anderes Mitglied ein Lock freigibt.
- Je feiner die Granularität bei der Concurrency Control, desto geringer die Gefahr von Deadlocks.
- Frage: kann man Deadlocks erkennen bzw. verhindern?

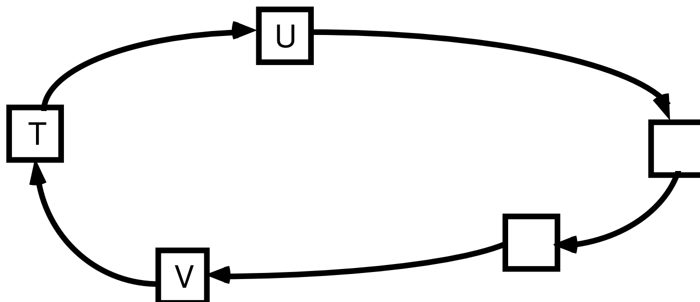
Beispiel für ein Deadlock und Wait-For-Graph

Transaction *T*		**Transaction *U***	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
•••		•••	
•••		•••	
		•••	



Allgemein: Zyklus im Wait-For-Graph

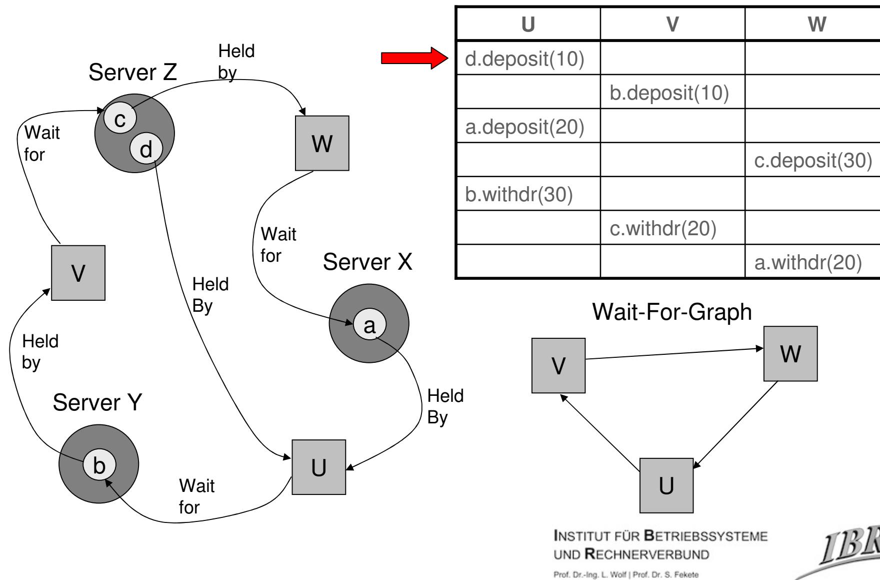
- Es gilt: wenn im Wait-for-Graph ein Zyklus existiert, dann ist das System in einem Deadlock



Distributed Deadlocks

- In verteilten Transaktionen wird auch das Problem der Deadlocks noch einmal eine Stufe schwieriger → verteilte Deadlocks können entstehen.
- Verteilte Deadlocks können u.U. nicht am lokalen Wait-For-Graphen erkannt werden.
- Vielmehr muss ein globaler Graph untersucht und dieser dann auf Zyklen untersucht werden.

Beispiel für verteiltes Deadlock



U	V	W
d.deposit(10)		
	b.deposit(10)	
a.deposit(20)		
		c.deposit(30)
b.withdr(30)		
	c.withdr(20)	
		a.withdr(20)

Konstruktion des globalen Graphen

- Der globale Wait-For-Graph wird aus den lokalen Graphen konstruiert.
- Einfachste Lösung:
  1. Der zentrale Koordinator sammelt die lokalen Graphen und konstruiert den Graphen.
  2. Anschließend untersucht er ihn auf Deadlocks.
  3. Er fällt eine Entscheidung und informiert die betreffenden Server über die Notwendigkeit eines Transaktionsabbruchs.
  4. Gehe zu 1
- Nicht immer gut wegen der üblichen Probleme (Bottleneck, single point of failure)

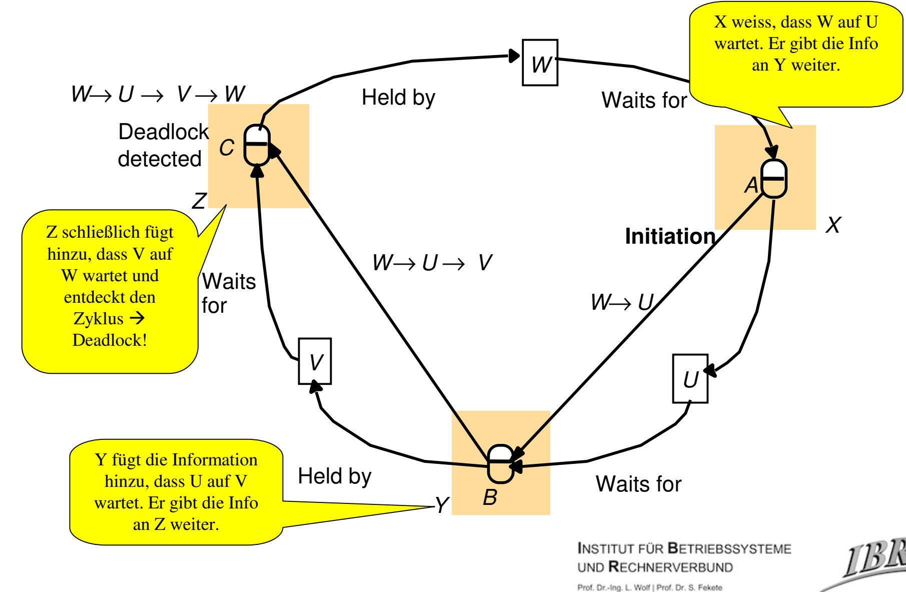
Institut für Betriebssysteme und Rechnerverbund
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Verteilte Ermittlung des Graphen

- Es gibt verteilte Algorithmen zur Ermittlung des globalen Graphen
- Algorithmus „Edge Chasing“
- Idee:
  - Konstruiere den Graphen nicht, aber gebe den einzelnen Servern Informationen über einige seiner Kanten
  - Die Server versuchen Zyklen zu finden, indem sie „probes“ entlang der Pfade des Graphen durch das System schicken.
  - Eine solche Nachricht enthält Informationen über bekannte Pfade des Graphen.

Institut für Betriebssysteme und Rechnerverbund
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Beispiel



Institut für Betriebssysteme und Rechnerverbund
Prof. Dr.-Ing. L. Wolf | Prof. Dr. S. Fekete
IBR

Transaktionen: Zusammenfassung

- Transaktionen sind ein mächtiges Mittel zur Realisierung des nebenläufigen Zugriffs auf Daten.
- Geschachtelte und verteilte Transaktionen erweitern das Modell, benötigen aber auch neue Algorithmen.

Diskussion