Verteilte Systeme 2012: Zusammenfassung

Inhaltsverzeichnis

Verteilte Systeme 2012: Zusammenfassung

INHALTSVERZEICHNIS

Verteilte Systeme 2012: Zusammenfassung

INHALTSVERZEICHNIS

Verteilte Systeme 2012: Zusammenfassung

1 Systemarchitekturen nach Flynn

1.1 SISD (Von-Neumann-Architektur)

Single Instruction Single Data
ein Instruktionsstrom und ein Datenstrom
sequentielle Uniprozessorarchitektur, ggf. aber interne Parallelität durch Pipelining oder intelligente I/O-Kanäle

1.2 SIMD

Single Instruction Multiple Data
ein Instruktionsstrom, mehrere Datenströme
Beispiele: Vektorprozessoren, Grafikkarten
Ausblenden einzelner Prozessoren durch Tagging möglich ← Prozessoren können simultan verschiedene Operationen ausführen

1.3 MIMD

Multiple Instruction Multiple Data
Systeme mit

- gemeinsamem Speicher („shared memory“, SMP)
- verteiltem Speicher („distributed memory“, DMS)

Prozessoren werden über Verbindungsnetzwerk gekoppelt, jeder wird über einen unabhängigen Instruktionsstrom gesteuert
Prozessoren arbeiten autonom und haben über Verbindungsnetzwerk Zugriff auf die Daten der anderen Prozessoren

1.4 MISD

Multiple Instruction Single Data
Eher nicht verwendet, als Beispiel aber FPGA

2 SPMD - Single Program, Multiple Data

nicht Teil der Flynn-Taxomonie
Grundprinzip: Alle (MIMD-)Prozessoren arbeiten auf Kopien desselben Programmcodes, aber mit unterschiedlichen Daten und ggf. in unterschiedlichen Modulen.
Vorteile

- leichte Programmentwicklung, -debugging, -wartung
- leichtere Synchronisation als „echte“ MIMD-Programmierung
- gröbere Parallelitätsgranularität als SIMD

3 Verteilte Systeme allgemein

3.1 Definition

Menge miteinander verbundener, autonomer Computer, die dem Nutzer wie ein einzelnes kohärentes System erscheinen.

- „Computer“: Prozessoren/Prozesse
- „autonom“ : jeder Knoten hat private Kontrolle (kein SIMD)
- „miteinander verbunden“ : Informationsaustausch ist möglich

Typen:

- Computer in WANs - Internet, Intranet
- Computer im LAN - Hausnetz einer Universität
- kooperierende Prozesse/Threads - Prozesse und Threads auf einer Maschine

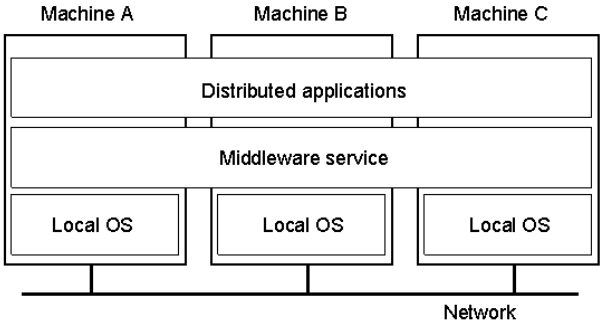
3.2 Vorteile/Motivation

- Informationsaustausch
- Zuverlässigkeit durch Replikation
- Ressourcensharing (Drucker, Festplattenspeicher, Rechenleistung)
- Leistungssteigerung durch Parallelisierung
- Vereinfachung des Systemdesigns durch Entkopplung/Spezialisierung

3.3 Anforderungen

- Transparenz - Verteilung bleibt dem Benutzer verborgen
- Offenheit - Austausch und Erweiterbarkeit (von Komponenten)
- Skalierbarkeit - gleich gute Leistung unabhängig von der Nutzeranzahl
  - Größe - mehr Nutzer und Ressourcen
  - geographische Verteilung
  - administrativ - über Organisationsgrenzen hinweg administrierbar

Realisierung von Transparenz durch Middleware:



4 Parallele Programmierung

4.1 Betriebssystemsicht

- Multicomputerbetriebssystem: erbringt die Systemdienste verteilt und transparent
  - präsentiert dem Nutzer ein kohärentes System
  - hat vollständige Kontrolle über Knoten und deren Ressourcen
  - kann keine heterogenen Systeme verwalten
- Netzwerk-Betriebssystem: erlaubt, Dienste entfernt zu nutzen, bietet aber keine Transparenz
  - verteilte Rechner mit autonomen Betriebssystemen und eigener Ressourcenverwaltung
  - eingebaute Netzwerkfunktionalität
  - skalierbar und offen

Middleware:

- bietet Abstraktionen für Netzwerkprogrammierung ← **bessere Transparenz**
- bietet relativ kompletten Satz von Diensten
- Event Handling und Filtering
- Auffinden von Ressourcen für mobiles Computing
- Unterstützen von Datenströmen

4.2 Parallele Programmierung

Ausgewogenes System: Forderung pro Operation/s:

- 1 Byte Hauptspeicherkapazität
- 100 Byte Plattenspeicher
- 1 Bit I/O-Rate

Leistungssteigerung durch:

- Pipelining: mehr Instruktionen pro Zeit durch parallel arbeitende Funktionale Einheiten
- Superscalar: mehr Instruktionen pro Takt durch duplizierte funktionale Einheiten
- Out-of Order Execution: mehr Instruktionen pro Zeiteinheit durch Vermeidung von Pipelinestauungen
- Multilevel Caches: mehr Instruktionen pro Zeiteinheit durch Vermeidung von Speicherwartezeiten
- SIMD: viele (gleiche) Instruktionen auf einem Datenstrom

4.3 Speedup

$S_N(n) = \frac{T_1(n)}{T_N(n)}$
n = Problemgröße
N = Anzahl Prozessoren/Knoten

absoluter Speedup: T_1 = Zeit des optimalen sequentiellen Algorithmus
relativer Speedup: T_1 = Zeit des paralellen Algorithmus auf einem Prozessor
Normalerweise: $1 \leq S_N(n) \leq N$
Slowdown, falls $S_N(n) < 1$ | zB wenn Mehraufwand für Parallelisierung > Gewinn

4.4 Amdahls Gesetz

N	Anzahl Prozessoren
P	Anteil des parallelisierbaren Programmcodes
1-P	Anteil des sequentiellen Programmcodes

$S_N(n) \leq \frac{1}{(1-P)+\frac{P}{N}} \leq \frac{1}{(1-P)}$
Speedup ist limitiert durch sequentiellen Anteil.
Amdahl-Effekt: Bei fester Prozessoranzahl steigt die Beschleunigung mit wachsender Problemgröße.

4.5 Effizienz

$E_N(n) = \frac{S_N(n)}{N}$
Quotient aus erreichtem und theoretisch maximalem Speedup. $\rightarrow E_N(n) = \frac{T_1(n)}{T_N(n)*N}$

5 Parallele Maschinenmodelle

5.1 PRAM - Parallel Random Access Memory Machine

Modelle:

- exclusive read, exclusive write
  - schwächstes Modell, Speicherverwaltung muss Minimum an Nebenläufigkeit unterstützen
- concurrent read, exclusive write
  - mehrere simultane Lesezugriffe
  - serielle Schreibzugriffe
- concurrent read, concurrent write
  - mächstigstes Modell
  - kann auf EREW simuliert werden

textbf(n,m)-PRAM modelliert Parallelrechner mit n Prozessoren und m Speicherworten, ähnlich einem shared memory MIMD System.

5.2 LogP

L	*latency*	für kleine Nachrichten
o	*overhead*	Aufwand für Senden/Empfangen
g	*gap*	Verzögerung zw. 2 Nachrichten (Bandbreite)
P	*processors*	Anzahl Prozessor-/Speichermodule

- anwendbar für massiv parallele Systeme/Cluster
- ignoriert lange Nachrichten und Sättigung des Kommunikationsmediums

5.3 BSP - Bulk synchronous parallel model

- parallele Berechnungen und Kommunikation im Wechsel
- Synchronisation zw. den Phasen
- keine Gefahr von Deadlocks

6 Parallele Programmiermodelle

6.1 Erzeugen von Parallelität

explizit:

- Threads: fork/join
- Prozesse
- RPCs

implizit:

- Matrixoperationen
- Prolog: parallel AND/OR
- vektorielle Ausdrücke

Kommunikation:
shared memory oder message passing

6.2 Programmspezifikation

Datenparallelität:

- alle Datenelemente werden gleich behandelt
- ein Kontrollfluss
- gut Skalierbar
- passt gut zu SIMD

Kontrollparallelität:

- simultane Ausführung verschiedener Instruktionsströme
- mehrere Kontrollflüsse
- schwer skalierbar
- passt gut zu MIMD

7 Interprozesskommunikation

- Ein **Prozess** ist ein Objekt des Betriebssystems, durch das Anwendungen sicheren Zugriff auf die Ressourcen eines Computers erhalten.
  - Prozesse sind voneinander **isoliert**
- zum Informationsaustausch muss Interprozesskommunikation eingesetzt werden (synchron oder asynchron)
- Nachrichtenübertragung ist fehleranfälliger und schwieriger, oft aber die einzige Möglichkeit

7.1 Eigenschaften von Netzwerken

- Skalierbarkeit
- Zuverlässigkeit
- Sicherheit
- Mobilität - werden mobile Systeme unterstützt
- Quality of Service
- Multicast
- Leistung

7.2 Leistungsparameter eines Kommunikationskanals

- **Latenzzeit**: Verzögerung zwischen dem Zeitpunkt, zu dem ein Prozess beginnt, eine Nachricht zu senden, bis zu dem Zeitpunkt, zu dem der empfangende Prozess beginnt, sie zu empfangen.
- **Bandbreite**: Gesamtmenge der übertragenen Daten per Zeiteinheit.
- **Jitter**: Varianz in der Latenz. Wichtig für Echtzeitanwendungen.

8 Persistenz und Synchronität in Kommunikationssystemen

Eine Kommunikation ist

- **persistent**, wenn Nachrichten bis zur Auslieferung gespeichert werden
- **transient**, wenn Nachrichten gespeichert werden, solange Sender und Empfänger ausgeführt werden

- **synchron**, wenn der Sender blockiert wird, bis die Nachricht beim Empfänger gespeichert oder ausgeliefert wurde.
- **asynchron**, wenn der Sender unmittelbar nach Senden der Nachricht fortgesetzt wird

9 MPI - Message Passing Interface

Parallelprogrammierung mit Nachrichtenaustausch möglich durch:

- direkten Zugriff aufs Netzwerk - effizient, aber unportabel
- eigenständige parallele Programmiersprache - zu aufwändig
- Unterprogrammbibliotheken, zB MPI, RPC, RMI

Ziele von MPI:

- Effizienz durch Parallelität
- Portabilität
- leichte Programmierung

MPI ist Programmierschnittstelle, nicht -sprache. **Grundlegende Konzepte** sind:

- Punkt-zu-Punkt Kommunikation zw. 2 Prozessoreinheiten
- kollektive Operationen
- komplexe Datentypen
- „Gruppen“, „Kontexte“, „Kommunikatoren“ zur koordinierten Kommunikation gleicher PE-Gruppen
- virtuelle Topologien zur effizienten Abbildung der virtuellen auf die reale Prozessortopologie

9.1 Punkt-zu-Punkt-Kommunikationsarten

- standard: synchron oder gepuffert - je nach Implementierung
- synchron: beendet, wenn Nachricht empfangen wurde
- gepuffert: beendet sofort
- ready: beendet sofort

Jeweils blocking und non-blocking.

10 Prozesse und Threads

10.1 Prozesse

Ein **Prozess** ist ein in Ausführung befindliches Programm, welches auf einem Prozessor läuft.
Das Betriebssystem sorgt für **Nebenläufigkeitstransparenz**, sodass sich Prozesse nicht beeinflussen.
Prozess besteht aus:

- mindestens einem Thread
- einer Ausführungseinheit mit eigenem Adressraum, Thread-Synchronisation und Kommunikationsressourcen

10.2 Threads

Ein **Thread** ist Betriebssystemabstraktion einer Aktivität. Threads sind nocht voneinander isoliert und besitzen nur einen Kontext.

- Einrichtung eines Threads 10 bis 20 mal schneller als eines Prozesses
- Wechsel zu Thread im selben Prozess 5 bis 50 mal schneller
- Threads in einem Prozess können Ressourcen gemeinsam Nutzen ohne auf Interprozesskommunikation zurückzugreifen
- Aber: Threads im selben nicht gegeneinander geschützt

11 Kommunikation über gemeinsamen Speicher

Systemmodelle: Distributed Memory vs. Shared Memory
Verteilter Speicher tauscht Daten per Nachrichten aus
Bei Shared Memory wird in gemeinsamen Speicherbereich geschrieben.

11.1 Shared Memory

Speicherzugriff über gemeinsames Medium, zB Bus.
Kommunikation ist implizit und transparent, Hardware aufwändig und nicht beliebig skalierbar.

11.2 Distributed Memory

- explizite Kommunikation zwischen Prozessen, aber schwierig implementierbar und fehleranfällig
- jeder Knoten mit wenigen anderen verbunden → gute skalierbarkeit
- Hardwaretopologie beeinflusst Kommunikationsleistung

11.3 Cache-Kohärenzprotokolle

Cache-Kontroller sorgt für Kohärenz (gleichen Zustand auf allen verteilten Rechnern)

- CPU, Speicher und Cache brauchen für Multiprozessorsystem nicht geändert werden
- Cache-Kontroller hat 2 Seiten: CPU und Bus
- Reihenfolge der Buszugriffe zur Serialisierung

11.3.1 Invalidationsprotokolle

Neu überschriebener Block wird in anderen Caches invalidiert. Block muss aber im Status *exclusive* sein.
Sobald Block *exclusive* ist, kann ohne weitere Transaktionen geschrieben werden.
Ein Block wird durch *read exclusive* belegt, der Bus realisiert die Serialisierung.
Kohärenz wird durch *read* und *read exclusive* vom Bus hergestellt.

11.3.2 Update-Protokolle

Schreiboperationen ändern den Wert aller Caches → **update** Transaktion vonnöten.
Vorteile:

- kürzere Zugriffszeit, Schreibaktionen verursachen keine späteren Cache-Misses
- spart Bandbreite

Nachteile u.a.:

- Konsekutive Schreibaktionen des Prozessors erzeugen mehrere Update-Transaktionen.

11.3.3 MSI Writeback Invalidationsprotokoll

Zustände:

- invalid (i)
- shared (S)
- dirty/modified (M) → nur ein Cache besitzt diesen Block

Prozessorereignisse

- PrRd (read)
- PrWr (write)

Bus-Transaktionen:

- BusRd: Kopie des Blocks ohne Schreibabsicht
- BusRdX: Kopie des Blocks mit Schreibabsicht
- BusWB: schreibt Block in den Speicher

Aktionen

- Status aktualisieren, Bus-Transaktionen durchführen, Block lesen/schreiben

11.3.4 Erweiterung: MESI

Problem mit MSI: read/modify Zyklus erfordert 2 Bustransaktionen, auch wenn der Block nicht als „shared“ in einem anderen Cache liegt.
Lösung: zusätzlicher Zustand **exclusive**

- exclusive (exclusive clean): nur dieser Cache besitzt den Block
- modified (exclusive dirty): nur dieser Cache besitzt den Block, der bereits verändert wurde
- braucht zusätzliches Signal im Bus, das angibt, ob Block shared ist

11.4 UMA, NUMA, ccNUMA

- UMA: Einsockel-Multicore-Systeme → PC, Laptop
- NUMA: Cray T3E (früher), Cray XT5, 6
- ccNUMA: alle Mehrsockel-Multicore-Systeme
- DM: Cluster

12 OpenMP

12.1 Konzepte

- Kommunikation über gemeinsamen Speicher (für Multicore- und SMP-Systeme)
- Fork-Join-Parallelisierung: implizites Multithreading
- Standardisierung
- API bestehend aus Compiler-Direktiven → Übersetzung mit Standard-Compiler möglich

12.2 Konstrukte

- Parallele Region - #pragma omp parallel
- Arbeitsteilung, explizit setzbar durch Scheduletypen - # pragma omp parallel for
- Datenumgebung
- Synchronisation
- Laufzeitbibliothek, Prozessumgebung

Beispiel:

```
double Results[HUGE];
#pragma omp parallel for
for (i=0; i<HUGE; i++) {          // Achtung: Seiteneffekte
        do_huge_comp (Results[i]);   // in do_huge_comp vermeiden
}
```

OpenMP Direktiven gelten für einen Block, zB eine for-Schleife.

12.3 Laufzeitverhalten

- dynamisch (Standard)
  - Anzahl aktiver Threads zwischen parallelen Regionen variiert
  - Setzen der Threads auf maximale Anzahl
- statisch
  - Zahl der Threads ist fest
  - Kontrolle durch Programmierer

12.4 Hybride Programmierung

Nutzung von MPI und OpenMP

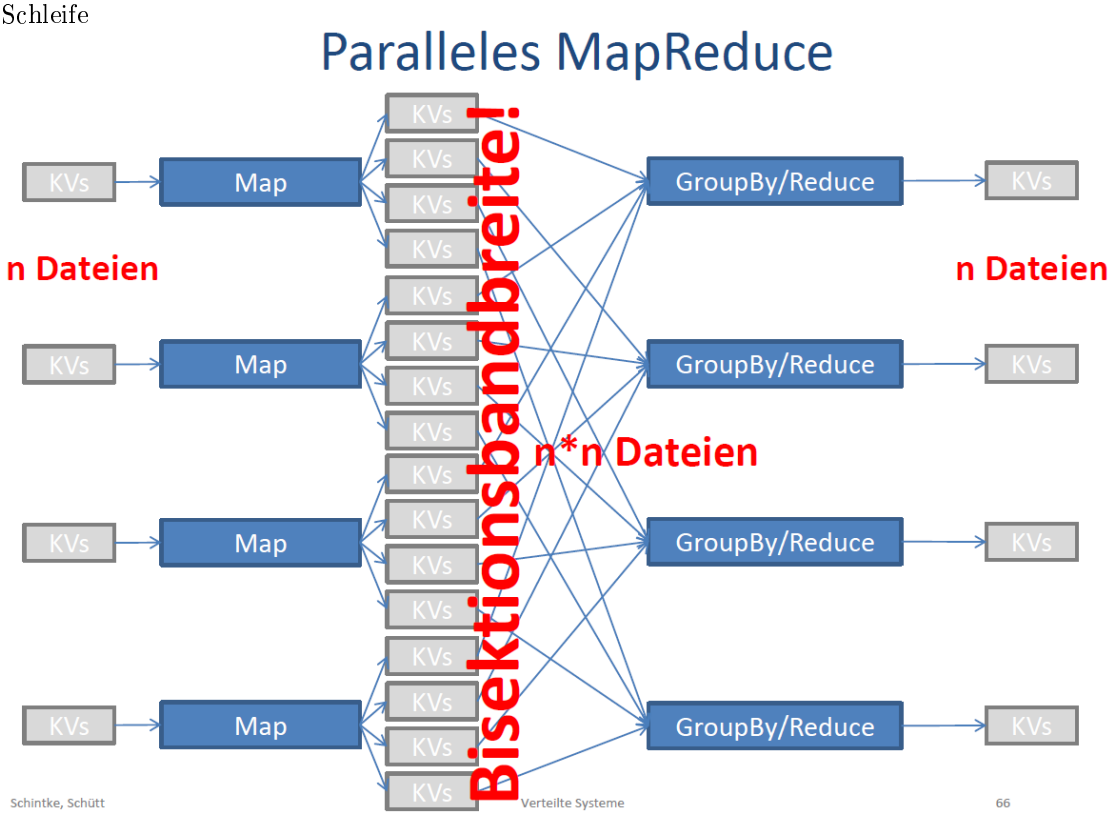
- OpenMP auf Knoten mit gemeinsamem Adressraum
- MPI für knotenübergreifende Kommunikation

Vorteile:

- flexible Anpassung der Anwendung an Zielsystem
- Speichereinsparung durch Nutzung des gemeinsamen Speichers über OpenMP
- numerische Libraries laufen automatisch mit mehreren Threads

13 MapReduce

MapReduce = skalierbares, fehlertolerantes, paralleles oder verteiltes Programmierparadigma
Kann als Realisierung von BSP angesehen werden - siehe auch BSP - Bulk synchronous parallel model
2 Phasen, dazwischen Kommunikation und in der Regel mehrere Iterationen in Schleife



14 Zeit in verteilten Systemen

14.1 Uhren in verteilten Systemen

Wozu?

- Zeitstempel zB für Abhängigkeiten in makefiles, Ticketverfahren usw..

Problem: in verteilten Systemen existiert keine globale Uhrzeit.
Lösung: logische Zeit

14.2 Logische Uhren

14.2.1 Zustände und Ereignisse

Verteiltes System = Menge von N Prozessen p_i mit i = 1,2,...,N
Jeder p_i besitzt einen Zustand s_i abhängig von seinen Variablen.
Kommunikation ausschließlich per Nachrichtenaustausch
Prozesse führen Aktionen aus. Interessant sind nur Aktionen, die den Zustand verändern → Ereignisse.
Ordnungsrelation: e $\rightarrow_i$ e' genau dann wenn e vor e' in p_i stattfindet.
Entspricht der kausalen Reihenfolge des Auftretens der Ereignisse. Auch **happens before Relation** genannt.
Für Prozess p_i mit $p_i : e \rightarrow_i e'$, dann gilt e → e'.
Für jede Nachricht m gilt: send(m) → receive(m).
Falls e → e' und e' → e" gilt auch e → e"

14.2.2 Logische Zeit

Statt Uhren zu synchronisieren werden Ereignisse in logische Reihenfolge gebracht. Nicht alle Ereignisse lassen sich mit „→“ anordnen. Nebenläufige Ereignisse werden mit a||e gekennzeichnet.

14.2.3 Lamports logische Uhren

Eine logische Uhr ist ein monoton ansteigender Softwarezähler. Wert steht nicht in Beziehung zur Uhrzeit.
Jeder Prozess p_i besitzt eine eigene logische Uhr, um Ereignisse mit Zeitstempeln zu versehen.

14.2.4 Vektoruhren

Mit Lamport-Uhren kann aus L(e) < L(e') nicht auf e → e' geschlossen werden. Vektoruhren sind Erweiterungen der Lamportuhren. Eine Vektoruhr V_i im Prozess p_i ist ein Array von N Integern.

- VC1: $V_i[j] = 0$ initial für alle i,j ∈ N
- VC2: Bevor p_i einem Ereignis einen Zeitstempel gibt, setzt er $V_i[i] := V_i[i] + 1$.
- VC3: p_i gibt jeder gesendeten Nachricht den Wert t = V_i mit.
- VC4: Wenn p_i in einer Nachricht einen Zeitstempel empfängt, setzt er $V_i[j] := \max(V_i[j], t[j]), j \in N$

Eine Vektoruhr hat dann die Form V_i = (0,1,4,1) für 4 Prozesse.

14.2.5 Vergleich von Vektorzeitstempeln

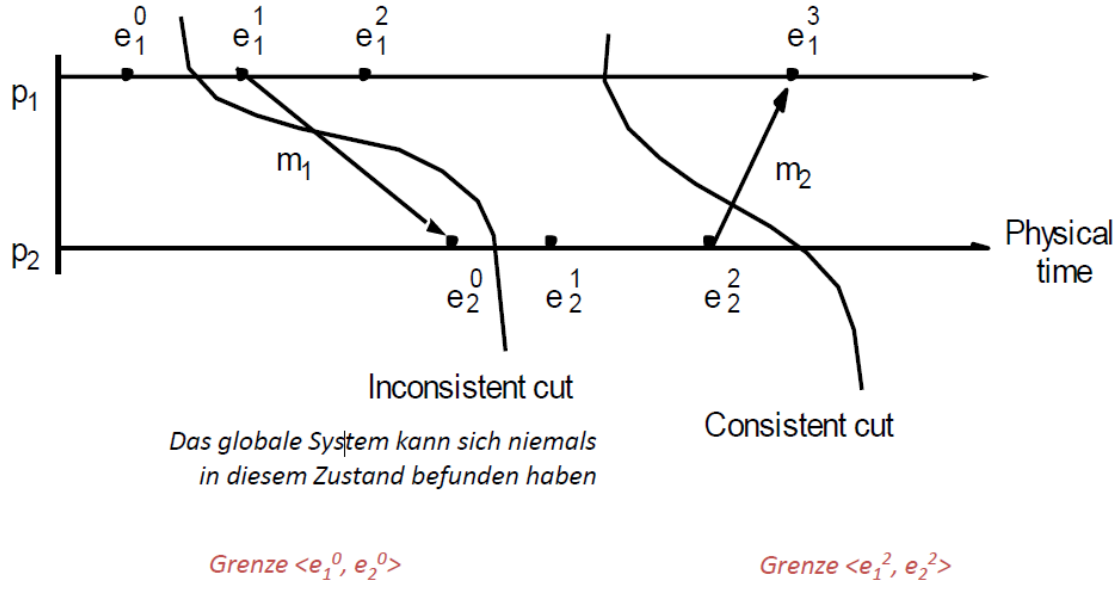
Für zwei Vektoruhren V und V' gilt:

- V = V' genau dann wenn V[i]) V'[i] für i ∈ 1,2,...,N
- V ≤ V' genau dann wenn V[i] ≤ V'[i] für i ∈ 1,2,...,N
- V < V' genau dann wenn V ≤ V' ∧ V != V'
- V || V' genau dann wenn ¬(V < V') ∧ ¬(V' < V) → nebenläufig. Beispiel: (2,1,0) und (0,0,1)

15 Globale Zustände

15.1 Snapshot

Ermittlung des globalen Zustands eines verteilten Systems.
Chandy-Lamport Algorithmus 1985: „Distributed Snapshot Algorithm", ermittle einen Zustand in dem das System möglicherweise war, der **konsistent** ist.
Konsistenz: Wenn festgehalten wurde, dass P eine Nachricht von Q empfangen hat, muss auch festgehalten sein, dass Q diese geschickt hat.



15.2 Definition eines Schnittes

Gegeben:

- System mit N Prozessen p_i

- Menge der globalen Zustände S = (s_1, ..., s_n wird betrachtet. Welche sind möglich?
- Jeder Prozess kann durch die **history** seiner Ereignisse charakterisiert werden:
  history(p_i) = h_i = $<e_i^0, e_i^1, e_i^2, ...>$
- Jedes endliche Präfix der Geschichte eines Prozesses wird bezeichnet mit:
  h_i^k = $<e_i^0, e_i^1, e_i^2, ..., e_i^k>$

Definition: Ein **Schnitt** ist eine Teilmenge der globalen history, d.h.
C = $h_1^{c1} \cup h_2^{c2} \cup ... \cup h_N^{cN}$

Aus Sicht des Prozesses p_i gilt: Der Zustand s_i im globalen Zustand S, der dem Schnitt C entspricht, ist genau derjenige, der von p_i durch das Ausführen des letzten Ereignisses im Schnitt erreicht wird, also $e_i^{c_i}$.
Die gesamte Ereignismenge $\{e_i^{c_i}$: i = 1,...,N$\}$ wird als Grenze (frontier) bezeichnet.
Ein Schnitt ist **konsistent**, wenn er für jedes Ereigniss, das er enthält, auch alle Ereignisse enthält, die zu diesem in der happens-before Relation stehen:
$\forall e \in C, f \rightarrow e \Rightarrow f \in C$
Ein konsistenter globaler Zustand ist ein Zustand, der einem konsistenten Schnitt entspricht. Die Ausführung des verteilten Systems kann als Folge globaler konsistenter Zustände beschrieben werden: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow ....$

15.3 Der verteilte Snapshot-Algorithmus von Chandy und Lamport

Annahmen:

- Zuverlässige Punkt-zu-Punkt-Kommunikation zwischen Prozessen
- Kanäle sind unidirektional und FIFO
- Prozessgraph ist zusammenhängend

Vorgehen

- Einer oder mehrere Prozesse starten den Algorithmus
- Das System läuft unterdessen normal weiter
- Prozesse verständigen sich über Markernachricht über die Notwendigkeit der Speicherung des Systemzustands

```
receive a marker M on channel in;
do {
        if (M is the first received marker) {
```

```
                record internal process state;
                for (each outgoing channel i)
                        send one marker M over channel i; /* before sending any
other message over i */
                for (each incoming channel i except in) {
                        initialize state of c[i] = { };
                        turn on recording of messages arriving over
                                channel i in c[i];
                }
        } else
                        finish recording of channel in;
} until (received a marker on each incoming channel) /* we're done then */
```

16 Speichermodelle und Konsistenz

16.1 Speichermodelle

definieren, welche Ergebnisse Lese- und Schreibtransaktionen zurückliefern, insbesondere konkurrierende.
sind das Interface zwischen Hardware und der Semantik der darauf laufenden Software.

16.2 Konsistenzmodelle

Vertrag zwischen Datenspeicher und darauf zugreifenden Prozessen.
Erwartung: Ergebnis eines Read liefert immer den Wert des letzten Write.
Arten von Konsistenzmodellen:

- Datenzentrierte: Aus sicht des Speichers
- Klientenzentrierte: Aus sicht des Client

16.2.1 Strenge Konsistenz

Alle Schreiboperationen sind für alle Prozesse unmittelbar sichtbar und es existiert eine globale, für alle gleiche absolute Reihenfolge.

16.2.2 Sequentielle Konsistenz

Das Ergebnis der Ausführung ist das selbe,

- als wenn die Lese- und Schreib-Operationen von allen Prozessen auf dem Datenspeicher in irgendeiner sequentiellen Reihenfolge ausgeführt worden wären
- und die Operationen in jedem einzelnen Prozess in der von dem Programm vorgesehenen Reihenfolge ausgeführt werden (sog. Programmreihenfolge).

- alle Prozesse sehen selbe Verzahnung
- über Zeit wird nichts ausgesagt
- Beispiel: (a) ist sequentiell konsistent, (b) nicht.

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)b	R(x)a

(a)

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

16.2.3 Linearisierbarkeit

Ein Datenspeicher ist **linearisierbar** wenn jede Operation einen lose synchronisierten Zeitstempel besitzt und die folgenden drei Bedingungen gelten:

- Das Ergebnis jeder Ausführung ist dasselbe, als wären die Lese- und Schreiboperationen aus Sicht des Speichers von allen Prozessen in irgend einer sequenziellen Reihenfolge ausgeführt worden.
- Wenn $timestamp_{OP1}(\mathrm{x}) < timestamp_{OP2}(\mathrm{x})$, dann muss der Speicher die Operation OP1 vor OP2 sehen.
- Die Operationen in allen Prozessen erscheinen in der vom Programm vorgesehenen Reihenfolge.

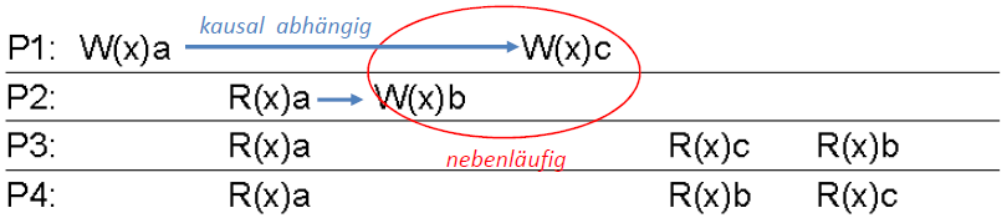
Ein linearisierbarer Datenspeicher ist auch sequentiell konsistent.

16.2.4 Kausale Konsistenz

Ein Speicher ist kausal konsistent wenn alle Schreiboperationen, die potentiell in einem kausalen Verhältnis stehen, von allen Prozessen in derselben Reihenfolge gesehen werden.

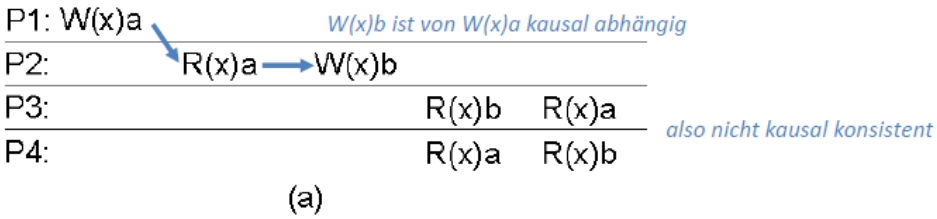
- Nebenläufige Schreiboperationen, die in keinem kausalen Verhältnis stehen, können von verschiedenen Prozessen in unterschiedlicher Reihenfolge gesehen werden.
- Schwächeres Modell als sequentielle Konsistenz.
- Vergleichbar mit Lamports „happens before“-Relation.

- Folgendes Beispiel erfüllt die kausale, nicht jedoch sequentielle Konsistenz



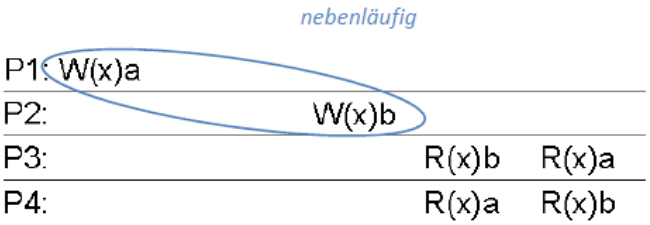
Schreiboperationen können auch über dazwischenliegende Leseoperationen Kausalketten aufbauen:

(a) nicht kausal konsistent:



(a)

(b) kausal konsistent:



(b)

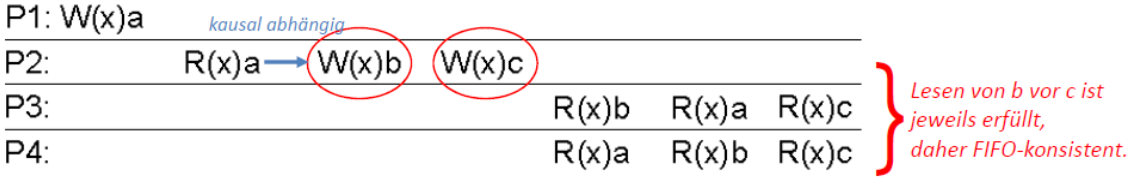
Kausale Konsistenz trifft auf **alle** Abhängigkeiten zu, also auch Kommunikationsoperationen.

16.2.5 FIFO-Konsistenz

Ein Speicher ist **FIFO-konsistent**, wenn

- die innerhalb eines Prozesses ausgeführten Schreiboperationen von allen anderen Prozessen in der Reihenfolge gesehen werden, in der sie ausgeführt werden,
- während Schreiboperationen unterschiedlicher Prozesse von anderen Prozessen in beliebiger Reihenfolge gesehen werden dürfen.
  (Dies ist die Abschwächung gegenüber der Kausalen Konsistenz)

Folgendes Beispiel ist nicht kausal konsistent, aber FIFO-konsistent:

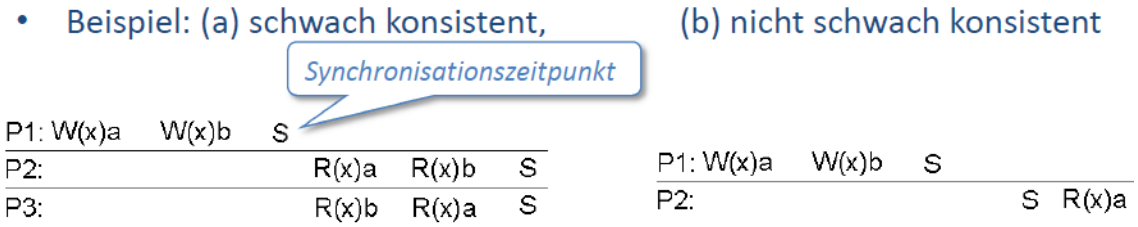


16.3 Konsistenzmodelle in verteilten Transaktionen

16.3.1 Schwache Konsistenz

In Transaktionssystemen ist (von außen betrachtet) die Reihenfolge der Schreiboperationen innerhalb kritischer Bereiche irrelevant; lediglich am Ende müssen die Kopien synchron sein. Dies wird schwache Konsistenz genannt.
Vorgehen: Alle Schreiboperationen werden lokal ausgeführt, anschließend werden die Kopien synchronisiert.

- Schwache Konsistenz ist auf **Gruppen von Operationen** definiert
- Schwache Konsistenz definiert **Synchronisationszeitpunkte**, nicht die Form der Konsistenz. Verteilter Speicher darf vorübergehend inkonsistent sein.



16.3.2 Release-Konsistenz

Gemeinsam benutzte Daten werden beim Verlassen eines kritischen Bereichs synchronisiert. Daten stehen jedem Prozess nach „acquire“ zur Verfügung. Wenn „acquire“ nicht benutzt sind, sind die Daten undefiniert. Beispiel:

16.3.3 Entry-Konsistenz

Gemeinsame Daten eines kritischen Bereichs werden erst direkt vor Eintritt synchronisiert.

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)		
P2:						Acq(Lx)	R(x)a	R(y)NIL
P3:							Acq(Ly)	R(y)b

16.4 Klientenbasierte Konsistenzmodelle

In verteilten Systemen werden Objekte idR öfter gelesen als geschrieben. Konsistenzgarantien werden für Sitzungen benötigt.
Sitzung: Folge logisch zusammengehörender Lese- und Schreibzugriffe einer Anwendung.
Dafür ist die (schwächere) klientenbasierte Konsistenz genügend.

16.4.1 Monotones Lesen

Wenn ein Prozess den Wert eines Datenelements x liest, gibt jede nachfolgende Leseoperation für x in diesem Prozess denselben oder einen aktuelleren Wert zurück.
Nur für einen einzelnen Klienten wird Lesekonsistenz garantiert, nicht jedoch für nebenläufige Zugriffe.

16.4.2 Monotones Schreiben

Die Schreiboperation eines Prozesses am Datenelement x wird abgeschlossen, bevor eine nachfolgende Schreiboperation auf x durch denselben Prozess erfolgen kann.
Es werden mindestens die Kopien synchronisiert, auf denen danach eine neue Schreiboperation beginnt.

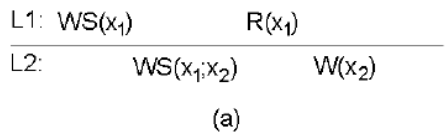
16.4.3 Read-Your-Writes Konsistenz

Die Folge einer Schreiboperation eines Prozesses auf ein Datenelement x wird in nachfolgenden Leseoperationen auf x durch denselben Prozess stets sichtbar sein.
Laufende Schreiboperationen werden abgeschlossen, bevor eine nachfolgende Leseoperation desselben Prozesses ausgeführt wird.

16.4.4 Writes-Follow-Reads-Konsistenz

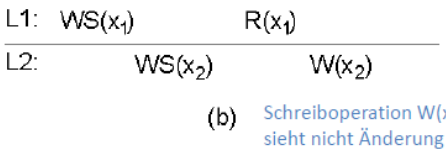
Schreiboperationen desselben Prozesses auf x erfolgen stets auf dem zuletzt gelesenen Wert, egal von welcher Kopie x zuletzt gelesen wurde.

Beispiel: (a) korrekt



(a)

(b) nicht korrekt



(b) Schreiboperation $W(x_2)$ sieht nicht Änderung $WS(x_1)$

17 Erlang

dynamisch getypte Sprache mit schwachen Methoden, um eigene Datentypen zu bauen. Real-Time Garbage Collectin, wobei jeder Prozess seinen eigenen Heap besitzt.

17.1 „Variablen“

- bekommen nur einmal einen Wert zugewiesen
- beginnen mit Großbuchstaben oder Unterstrich
- „_“ steht immer wieder für nicht belegte Variable
- Variablen der Form „_Foo“ stehen für Variablen, deren Wert nicht benötigt wird (Konvention)

17.2 =

Bedeutung wie in der Mathematik, diese Operation verändert keine Werte von bereits belegten Variablen, sondern wertet aus, ob die Ausdrücke links und rechts matchen (kann zu Exception führen).
Zuweisungsoperator ist „:=“

17.3 Funktionen

Der letzte Ausdruck einer Funktion ist ihr Rückgabewert.

```
f() -> 27.

g(1) -> 1;
g(X) -> f() + X.

h(X, Y) ->
    Z = g(Y),
    X + Z.
```

Beispiel simple_math.erl:

```
-module(simple_math).
-export([square/1]).
square(X) ->
      X * X.
cube(X) ->
      square(X) * X.
```

17.4 Atome

sind für sich selbst stehende Identifikatoren. Atome fangen mit Kleinbuchstaben an und können durch einfache Anführungszeichen umrahmt werden.

17.5 Tupel

sind container statischer Länge, die häufig ein Atom als Identifikator an erster Stelle besitzen.
Beispiel:

```
Car = {car,
            {honda, civic},
            {horsepower, 100}}.

{car, Type, Power} = Car.
```

→ Type = {honda, civic} und Power = {horsepower, 100}

17.6 Listen

sind Container variabler Länge, die einfach verkettet sind.
Um Anfang und Rest der Liste zu bekommen, wird **[Head|Tail]** verwendet.
Beispiele

```
List = [1, 2, 3, four, 5.0]
[Head|Tail] = List
[H1,H2|T2] = List
```

17.7 Module

Logisch getrennte Codeblöcke. Mit *modulname:funktion* kann über Modulgrenzen hinweg gearbeitet werden. Alternativ können Module mittels *-import(modul)* importiert werden.

17.8 Anonyme Funktionen: fun(X)

```
Square = fun(X) -> X * X end.
Cube = fun(X) -> Square(X) * X end.
```

17.9 Parallelverarbeitung

leichtgewichtige Prozesse

- Code wird immer innerhalb eines Prozesses ausgeführt
- keine Nutzung von Betriebssystem-Prozessen oder Threads
- intern gescheduled, Prozesse teilen keine Ressourcen (eigener Heap)

Prozesse können mit Namen registriert werden: register(Name, Pid)
→ Name oder Pid eines Prozesses muss bekannt sein, um Nachrichten senden zu können. Nicht-lokale Fehlerbehandlung: Nicht alle Prozesse müssen korrekt sein → bessere Fehlertoleranz
Kein shared Memory zur Vermeidung von:

- kritischen Sektionen
- Deadlocks
- Abhängigkeiten zwischen Prozessen
- fehlender Unterstützung von Verteilung

17.10 Prozesse

Einheit für Parallelverarbeitung, die von der Laufzeitumgebung (nicht dem OS) verwaltet wird → leichtgewichtig und kooperativ.
Verwaltung:

- Pid = spawn(Fun, [Parameters]) zum Starten eines Prozesses
- Pid ! Message zum Senden einer Nachricht
- receive/1 built-in functions zum Empfangen von Nachrichten
- Senden und Empfangen asynchron möglich

17.11 Fehlertoleranz

Prozesse können gekoppelt (linked) werden, um eine Fehlerkette zu definieren. Stirbt ein Prozess, bekommen gekoppelte Prozesse ein exit-Signal. Links werden mittels *link(Pid)* oder *spawn_link(Module, Function, Args)* angelegt und sind bidirektional. Mittels *unlink(Pid)* können Links aufgehoben werden.
Ein Prozess kann explizit das Exit-Signal senden: *exit(Pid, Reason)*. Dafür muss kein Link vorhanden sein.
Stirbt ein Prozess wegen eines Fehlers, bewirkt das Exit-Signal im Standardfall, dass auch die Empfänger beendet werden → Propagierung des Signals.
Die Exit-Signale können explizit behandelt werden: *process_flag(trap_exit, true)*.

18 Gruppenkommunikation

18.1 Abstraktionen

18.1.1 Prozesse

Verteilter Algorithmus besteht aus einer Menge verteilter Prozesse, die mit- und untereinander kommunizieren. Ein Prozess kann als endlicher Automat angesehen werden.
Ein korrekter Prozess bearbeitet eine unendliche Anzahl an Events und stürzt nicht ab.
Prozesse kommunizieren durch den Austausch von Events. Ein verteilter Algorithmus lässt sich als Menge von Event-Handlern beschreiben. Die Events legen den Programmablauf durch ihr Auftreten fest.
Eigenschaften von Prozessen:

- Sicherheit (safety): Nothing bad happens
- Lebendigkeit (liveness): Something good will happen

Ausfallmodelle für Prozesse:
Die Fehlereinheit sind Prozesse → entweder ein Prozess arbeitet, oder er ist komplett abgestürzt.
Fehlerklassen:

1. Absturz (crash)
2. Auslassung (omission)
3. Wiederanlaufen mit altem persistentem Zustand
4. bösartige (byzantinische) Fehler

Crash Stop: Prozess stoppt und macht dann nichts mehr
Crash Recovery: Prozess stoppt, arbeitet später aber vielleicht weiter. Der Zustand kann persistent gespeichert sein.
Korrekter Prozess: Prozess, der nicht ausfällt.

18.1.2 Zeit

Ausfalldetektor: Erkennt, ob ein anderer Prozess abgestürzt ist.
Vorteile:

- Prozess- und Link-Abstraktionen benötigen kein Zeitkonzept
- Verhalten von Ausfällen kann ohne das Zeitkonzept beschrieben werden

Es gibt Ausfalldetektoren für partiell und vollständig synchrone Systeme.
Perfekter Ausfalldetektor:

- Voraussetzungen: synchrones System und PerfectPointToPointLinks pp2p

- Ereignisse: indication: <crash | p_i> um mitzuteilen, dass p_i ausgefallen ist
- Eigenschaften:
  PFD1 (Vollständigkeit, strong completeness): Jeder abgestürzte Prozess wird durch jeden Prozess erkannt.
  PFD2 (Genauigkeit, strong accuracy): Prozess p wird erkannt → p ist ausgefallen.

Korrektheit (Grundidee)

- Vollständigkeit: Wenn ein Prozess abstürzt, sendet er keine Hearbeats mehr. Der Ausfalldetektor erhält dann wegen der pp2p-Links keine Heartbeats mehr und er erkennt P als abgestürzt.
- Genauigkeit: Nur wenn ein Prozess keine Heartbeats mehr schickt, kann er als abgestürzt erkannt werden.

18.1.3 Abstraktion für Kommunikationskanäle

Kommunikationskanal (channel, link)
Punkt-zu-Punkt Kommunikation mit gegebenen Eigenschaften
einheitliche Schnittstelle
Fair-Loss Link

- Nicht alle Nachrichten gehen verloren: ∞ oft senden → ∞ oft ausliefern
- Link erzeugt keine eigenen Nachrichten
- Nachricht endlich oft senden → Nachricht nicht unendlich oft ausliefern

Stubborn-Link

- Nachricht einmal senden → Nachricht unendlich oft ausliefern
- Link erzeugt keine eigenen Nachrichten

Perfect Link

- Jede gesendete Nachricht wird letztendlich ausgeliefert
- Keine Nachricht wird mehrfach ausgeliefert
- Link erzeugt keine eigenen Nachrichten

18.2 Multicasts

senden Nachrichten an eine Gruppe von Prozessen. Wie Broadcast, aber für Gruppen. Sie besitzen eine einheitliche Schnittstelle und können - ähnlich wie Links - aufeinander aufbauen, um Auslieferungsgarantien (auch bzgl. der Reihenfolge) zu erreichen. Geschlossene Gruppen:

- Nur Mitglieder der Gruppe können an die Gruppe senden
- Sender liefert die Nachricht auch an sich selber aus

Offene Gruppen

- Benachrichtigung sowohl innerhalb als auch von außerhalb möglich

18.2.1 Best-Effort Multicast

Sendet allen Prozessen der Gruppe inkl. sich selbst eine Nachricht. Gibt keine Zuverlässigkeitsgarantien, fällt also ein Sender aus, können einige Nachrichten ausgeliefert sein, andere Empfänger bekommen sie nicht.
Eigenschaften:

- BEM1 Gültigkeit: Wenn p_i und p_j korrekt sind, wird jede gesendete Nachricht schließlich ausgeliefert
- BEM2 Keine Duplikate: Keine Nachricht wird mehrfach ausgeliefert
- BEM3 Keine Erzeugung: Keine Nachricht wird ausgeliefert, die nicht gesendet wurde.

18.2.2 Reliable Multicast

Motivation:
bem sichert korrekte Auslieferung zu, solange der Sender nicht ausfällt. Bei Ausfall ist die Situation aber unklar und perfect links garantieren nicht die Auslieferung bei Ausfall des Senders.
rm sichert korrekte Auslieferung an korrekte Prozesse auch bei Ausfall des Senders zu. Die korrekten Prozesse einigen sich über die Menge der auszuliefernden Nachrichten.
Eigenschaften:

- RM1 = BEM1 Gültigkeit: p_i und p_j korrekt $\rightarrow$ jede gesendete Nachricht wird schließlich ausgeliefert
- RM2 = BEM2 Keine Duplikate
- RM3 = BEM3 Keine Erzeugung
- RM4 **Alle korrekten Prozesse oder keiner:** Wird m von einem korrekten Prozess ausgeliefert, liefern alle korrekten Prozesse m aus

18.2.3 Uniform Reliable Multicast (URM)

Motivation:
Bei rm einigen sich nur die korrekten Empfängerprozesse, ob Nachricht bei Ausfall des Senders ausgeliefert werden soll.
Bei urm einigen sich alle Prozesse.
Eigenschaften:

- URM1 = BEM1 Gültigkeit
- URM2 = BEM2 Keine Duplikate
- URM3 = BEM3 Keine Erzeugung
- URM4 **Liefert irgendein Prozess aus, so liefern alle korrekten Prozesse aus:** Wird m von irgendeinem (korrekten oder fehlerhaften) Prozess ausgeliefert, so liefern alle korrekten Prozesse m aus.

18.2.4 Stubborn Multicast

Geht mit Crash-Recovery Prozessen um. Idee: Multicast über Stubborn-Link. Infolgedessen werden Nachrichten unendlich oft ausgeliefert.
→ kein Aufwand für Logging, kein Ausfalldetektor nötig

18.2.5 Logged BEM

Geht ebenfalls mit Crash-Recovery Prozessen um, ggf. mit persistentem Zustand, vermeidet aber doppelte Auslieferung. Der Multicast erfolgt über einen Stubborn-Link, bei Auslieferung wird aber geloggt und vor Auslieferung das log abgefragt.

18.3 Geordnete Multicasts

Basis-Multicasts liefern in beliebiger Reihenfolge aus.

- FIFO-Ordnung: Sendet ein korrekter Prozess erst m und dann m', wird m auch vor m' von jedem korrekten Prozess ausgeliefert.
- Kausale Ordnung: wenn multicast(g,m) → multicast(g,m'), wird m vor m' von jedem korrekten Prozess ausgeliefert.
- Totale Ordnung: Liefert ein korrekter Prozess Nachricht m vor Nachricht m' aus, dann liefert jeder andere korrekte Prozess, der m' ausliefert, zuvor m aus.

18.3.1 FIFO geordneter Multicast

Benutzt einen beliebigen Basis-Multicast, zB bem-, rm- oder urm-multicast.
Jeder Prozess besitzt:

- S_{pg} Zähler der Nachrichten p an Gruppe g
- R_{qg} Zähler der Nachrichten an Gruppe g, die p von q ausgeliefert hat

Algorithmus:

- Will p an g fo-multicasten, fügt er S_{pg} der Nachricht hinzu, multicastet sie und erhöht S_{pg} um 1

- beim Empfang einer Nachricht von q mit Sequenznummer S prüft p, ob S = R_{qg} + 1. Falls ja, wird mittels fo-deliver ausgeliefert
- Der B-multicast stellt sicher, dass Nachrichten schließlich ausgeliefert werden, es sei denn der Sender fällt aus

18.3.2 Kausal geordneter Multicast

Nutzt die Happens Before Relation - aber nur auf Multicastnachrichten - sowie Vektorzeitstempel.
Algorithmus:

- Nach Auslieferung einer Nachricht von p_j aktualisiert p_i seinen Zeitstempel durch Erhöhen des j-en Elements des Vektors
- Verglichen mit der normalen Vektoruhrregel, wo $V_i[j]$:= max($V_i[j]$, t[j]) für j=1, 2, ...N wissen wir in diesem Algorithmus, dass sich nur das j-te Element erhöht.
- Wird rm-multicast statt bem-multicast genutzt, so ist das Protokoll sowohl zuverlässig, als auch kausal geordnet.
- Wird es mit einem Sequenz-Algorithmus gekoppelt erhält man totale und kausale Ordnung.

18.3.3 Total Order Multicast

Ansatz: Anhängen total geordneter Identifikatoren an Multicastnachricht. Jeder empfangene Prozess ordnet die Nachrichten anhand der Identifikatoren ähnlich dem FIFO-Algorithmus. Die Prozesse haben jedoch gruppenspezifische Sequenznummern.
Für den total geordneten Multicast existieren zwei Lösungsansätze mittels Basis-Multicast:

- Nutzung eines globalen Sequenznummerngenerators
- Die Prozesse einigen sich gemeinsam auf eine Sequenznummer für jede Nachricht, zB mittels ISIS-Algorithmus.

19 Konsens

Grundlegendes Problem in verteilten Systemen. Anwendungen unter andererem:

- Berechnung von Schnappschüssen
- Deadlockerkennung
- Broad-/Multicast

- Election
- Mutual Exclusio

19.1 Regulärer Konsens

Grundidee: Prozesse tauschen vorgeschlagene Werte in Runden aus und einigen sich nach N Runden.
Es gibt hierfür zwei verschiedene Algorithmen:

- Flooding: wenige Runden, dafür viele Nachrichten
- Hierarchical Consensus: wenige Nachrichten aber viele Runden

19.1.1 Voraussetzungen

- Prozesse sind eindeutig unterscheidbar (Pid, totale Ordnung)
- Prozesse dürfen jederzeit ausfallen, werden dann aber nicht neu gestartet
- perfekter Ausfalldetektor
- perfekte Kommunikationskanäle - jede Nachricht wird letztendlich ausgeliefert

19.1.2 Ereignisse

- Vorschlag: <cPropose, v> um Wert v vorzuschlagen
- Ergebnis: <cDecide, v> um entschiedenen Wert v zurückzuliefern

19.1.3 Eigenschaften

C1 Gültigkeit	Wenn ein Prozess v entscheidet, so wurde v von einem Prozess vorgeschlagen.
C2 Übereinstimmung	Keine zwei korrekten Prozesse entscheiden unterschiedlich.
C3 Termination	Jeder korrekte Prozess entscheidet letztendlich.
C4 Integrität	Kein Prozess entscheidet mehr als einmal.

19.1.4 Flooding

Start: Der <cPropose, v> ausführende Prozess schlägt den Wert v vor und sendet ihn per Multicast an alle anderen Prozesse.
Runden 1....N:

- jeder Prozess sammelt alle vorgeschlagenen Werte
- in jeder Runde schickt jeder Prozess alle Werte an alle
- werden neue Werte erhalten, werden diese mit der eigenen Wertemenge vereinigt

- Runde wird erst beendet, wenn alle Vorschläge von allen korrekten Prozessen erhalten wurden
  - jede Nachricht enthält die Rundennummer

Ein Prozess darf entscheiden, sobald er von allen anderen korrekten Prozessen einen Vorschlag erhalten hat und in der letzten Runde kein Prozess neu ausgefallen ist, da dieser sonst an einige seine Wertemenge gesendet haben kann, an andere nicht.
Komplexität:

- Bester Fall (keine Ausfälle): 1 Runde
- Schlechtester Fall (N-1 Ausfälle): N Runden
- in jeder Runde N^2 Nachrichten vor Entscheidung und N^2 decided-Nachrichten
- im schlechtesten Fall also N Runden * N^2 Nachrichten = O(N^3)

19.1.5 Reg. Kons. durch hierarchische Entscheidung

Voraussetzung: totale Ordnung auf den Prozessen: $p_1 > p_2 > ... > p_n$
Grundidee: Der korrekte Prozess mit dem höchsten Rang (kleinste ID) entscheidet autonom. Wenn also p_1 nicht abstürzt, müssen alle anderen seinen Wert übernehmen.
Allgemein: In der k-ten Runde entscheidet p_k und broadcastet seine Entscheidung. Alle anderen warten auf Entscheidung von p_k oder auf Meldung über den Ausfall von p_k.
So wird eine schnelle Rückgabe des Ergebnisses ermöglicht, bevor alle Prozesse entschieden haben.
Komplexität: Braucht immer N Runden, je Runde werden N Nachrichten versandt, da immer nur ein Prozess broadcastet.
Verbesserung: Keine Nachrichten an ranghöhere Prozesse senden.

19.2 Uniform Consensus

Keine zwei Prozesse entscheiden unterschiedlich, korrekt oder nicht. Wenn der Sender ausfällt, müssen die korrekten Prozesse später so entscheiden, wie der abgestürzte Prozess.
Grundidee: Alle Prozesse warten N Runden, bevor sie entscheiden. Spätestens nach N Runden wurde der Wert auch im ungünstigsten Fall propagiert.
Regular Consensus entscheidet zu früh, Uniform Consensus wartet. (Insb. wenn der Sender abstürzt.)

19.2.1 Unif. Cons. durch Flooding

Implementationsidee: Auslieferung erst nach N *beb*-Runden (Best Effort Broadcast). Warum N?
Weil beb ggf. nur einen Prozess erreicht. Kommunikationskette über N bebs

sorgt dafür, dass alle Prozesse die Information erhalten. Falls kein Prozess erreicht wird, gibt es keine Senderentscheidung. Nur die korrekten Prozesse, die an allen N Runden teilgenommen haben, entscheiden.
Komplexität: N Runden mit je N^2 Nachrichten = N^3

19.3 Wechselseitiger Ausschluss

Zur gemeinsamen aber nicht gleichzeitigen Nutzung von Ressourcen, realisierbar zB durch Semaphore.
Anforderungen:

1. Safety: zu jedem Zeitpunkt max. 1 Prozess in critical section
2. Liveness: kein deadlock, keine starvation
3. Ordering: Wenn P_i vor P_j um Eintritt in die critical section gebeten hat, betritt er auch zuerst den Bereich.
4. Fehlertoleranz: 1. und 2. müssen auch bei Fehlern erfüllt sein

19.3.1 Maekawa-Algorithmus

Grundidee: Um kritischen Abschnitt zu betreten, müssen nicht alle Prozesse zustimmen. Die Zustimmung einer Teilmenge reicht, solange die Teilmengen überlappend sind. Prozesse haben nur eine Stimme und stimmen für andere Prozesse ab, den kritischen Bereich zu betreten.
Annahmen: Prozesse sind korrekt und wir haben pp2p Links.

Algorithmus:
Jeder Prozess p_i hat eine Wählermenge $V_i \subseteq \{p_1, p_2, ..., p_N\}$

- p_i ist selbst Element von V_i
- Für alle V_i, V_j gilt: Es gibt mindestens ein gemeinsames Element (Überlappung)
- Um fair zu sein, enthält die Menge V_i genau K Elemente. Jeder Prozess p_i ist dann in M Mengen vertreten.

Man kann erreichen, dass K $\approx N^{\frac{1}{2}}$ ist und M=K gilt. Dann werden nur $3N^{\frac{1}{2}}$ Nachrichten für das Betreten des kritischen Abschnitts benötigt.

```
Initialization:
        state := RELEASED;
        voted := FALSE;

For p_i to enter the critical section:
        state := WANTED;
        Multicast request to all processes in V_i;
```

```
        wait until(number of replies received = K);
        state := HELD;

On receipt of a request from p_j at p_i:
        if (state = HELD or voted = TRUE)
        then
                queue request from p_j without replying;
        else
                send reply to p_j;
                voted := TRUE;
        end if

For p_i to exit the critical section
        state := RELEASED;
        Multicast release to all processes in V_i;

On receipt of a release from p_i at p_j
        if (queue of requests is non-empty)
        then
                remove head of queue;
                send reply to p_k;
                voted := TRUE;
        else
                voted := FALSE;
        end if
```

Eigenschaften:

- **Korrektheit:**
  - **Sicherheitseigenschaft:** Es können nicht 2 Prozesse gleichzeitig im kritischen Abschnitt sein, weil $V_i \cap V_j \neq \emptyset$ und die Prozesse, die in V_i und V_j sind, nur eine Stimme abgegeben haben.
  - **Lebendigkeit** und **Reihenfolge:** Ja, wenn Zeitstempel eingeführt wird
- **Fehlertoleranz**: Was passiert, wenn Nachrichten verloren gehen oder Prozesse abstürzen? Maekawa toleriert Prozessabstürze, solange diese nicht zur Wählermenge gehören.
- **Komplexität:** Eintritt in kritischen Abschnitt erfordert $3\sqrt{N}$ Nachrichten.

Maekawa ist nicht Deadlockfrei

Beispiel: 3 Prozesse p_1, p_2, p_3 mit folgenden Wählermengen:
$V_1 = \{p_1, p_2\}$
$V_2 = \{p_2, p_3\}$

$V_3 = \{p_3, p_1\}$
Annahme: Alle drei wollen gleichzeitig in den kritischen Bereich:
p_1 erlaubt $p_2 \rightarrow$ voted = TRUE
p_2 erlaubt $p_3 \rightarrow$ voted = TRUE
p_3 erlaubt $p_1 \rightarrow$ voted = TRUE
Keiner besitzt die Mehrheit, alle blockieren.
Der Algorithmus kann durch Einführung von Lamport-Zeitstempeln deadlockfrei gemacht werden, der den frühsten Kandidaten bevorzugt.

20 Wahlalgorithmen

20.1 Zweck von Wahlalgorithmen

In vielen verteilten Systemen wird ein Prozess mit herausragender Rolle benötigt (Koordinator, Monitor, ...). Aufgabe des Wahlalgorithmus ist die Bestimmung dieses Prozesses unter vielen, sodass sich am Ende alle einig sind, wer gewählt wurde.
Voraussetzungen

- Prozesse sind unterscheidbar
- Jeder Prozess kennt die Prozessnummern aller anderen Prozesse
- Aber: Kein Prozess weiß, welche anderen Prozesse gerade laufen.

Ein Prozess initiiert den Wahlvorgang, darf aber gleichzeitig nicht 2 Vorgänge starten. Allerdings können mehrere Prozesse gleichzeitig eine Wahl initiieren, zB wenn der Leader ausgefallen ist und dies gleichzeitig festgestellt wird.
O.B.d.A.: Der Leader ist der korrekte Prozess mit der höchsten ID.
Anforderungen:

- E1 **Sicherheit:** Jeder Wahlteilnehmer p_i hat am Ende $elected_i = P$, wobei P der korrekte Prozess mit höchster ID ist.
- E2 **Lebendigkeit:** Alle korrekten Wahlteilnehmer p_i haben am Ende $elected_i = P$.

20.2 Ring-Algorithmus von LeLann

Prozesse sind in unidirektionalem logischen Ring angeordnet. Jeder Prozess kennt seinen direkten und seine indirekten Nachfolger.
Nachrichtenarten:
e-Nachricht (election): um neuen Koordinator zu wählen
c-Nachricht (coordinator): um gewählten Koordinator bekanntzugeben

- Start
  - Der Algorithmus wird von beliebigem p_i gestartet

  - p_i sendet e-Nachricht mit seiner Prozessnummer an seinen Nachfolger p_{i+1}
  - Antwortet p_{i+1} nicht, wird angenommen, er ist ausgefallen. Dann wird die e-Nachricht an p_{i+2} gesendet.

- Ablauf
  - e- oder c-Nachrichten enthalten Liste mit Prozessnummern
  - bei Empfang einer e-Nachricht:
    * Ist die eigene Nummer nicht dabei, wird sie hinzugefügt und die Nachricht an den Nachfolger geschickt
    * ist sie dabei, ist die Nachricht einmal um den Ring gelaufen. p_i sendet nun eine c-Nachricht
  - bei Empfang einer c-Nachricht
    * Prozess mit der höchsten Nummer in der Liste ist der neue Koordinator
    * ist die c-Nachricht einmal um den Ring gelaufen, wird sie gelöscht

20.3 Ring-Algorithmus von Chang und Roberts

Wie LeLann, aber die e-Nachricht enthält nur eine Prozessnummer. Alle Prozesse besitzen eine Prozessnummer und der korrekte Prozess mit der höchsten Nummer wird neuer Leader. Die Wahl wird durch eine **election message** gestartet, der Leader wird in einer **elected message** bekannt gegeben.
Erhält ein Prozess eine election-Nachricht mit einer geringeren Nummer als der eigenen, ersetzt er diese durch seine Prozessnummer und sendet die Nachricht weiter.

20.3.1 Korrektheit

- **E1 Sicherheit:** Jeder Teilnehmer p_i hat am Ende $leader = P$, wobei P der korrekte Prozess mit der höchsten ID ist. Da eine erfolgreiche Nachricht einmal um den Ring läuft, werden alle ID's verglichen
- **E2 Liveness:** Alle korrekten Teilnehmer p_i haben am Ende $leader = P$. Falls keine Prozesse ausfallen, läuft die election-Nachricht maximal zweimal um den Ring und der Alg. terminiert nach einer weiteren Runde mit elected-Nachrichten.

20.3.2 Komplexität

- Schlechtester Fall: Vorgänger hat höchste Prozessnummer → N-1 Hops zum Erreichen. Dieser erkennt dann aber noch nicht, dass er Leader ist und sendet die Nachricht erneut um den Ring → weitere N hops. Noch einmal N hops werden für die elected-Nachrichten benötigt. ⇒ 3N-1 Nachrichten

- Bester Fall: Wir sind Leader ⇒ 2N-1 Nachrichten

20.4 Bully-Algorithmus

Findet den aktiven Knoten mit höchster PID, macht ihn zum Leader und teilt dies allen anderen mit. Jeder Prozess kann den Algorithmus starten, wenn er merkt, dass der bisherige Koordinator ausgefallen ist.

20.4.1 Eigenschaften:

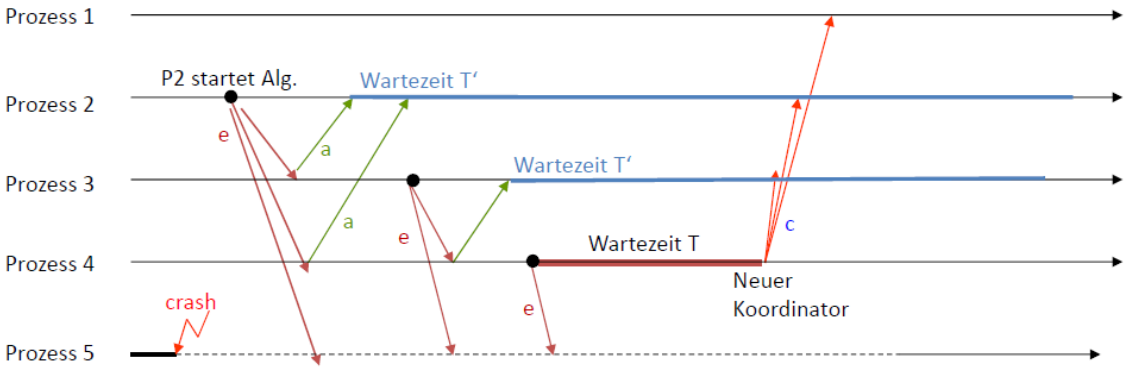
- Prozesse dürfen ausfallen (crash-recovery)
- Jeder Prozess kennt alle anderen und kann Nachrichten an sie senden
- Synchrones System mit Timeouts zum Erkennen von Ausfällen (perfekter Ausfalldetektor)

20.4.2 Nachrichtentypen:

- election e: Leitet Wahl ein
- answer a: Antwort auf e-Nachricht
- coordinator c: Bekanntmachen des Leaders

20.4.3 Funktionsweise

1. Prozess p_i sendet e-Nachricht an alle Prozesse p_j, j > i
   - Antwortet innerhalb Zeit T niemand, ist p_i der neue Leader
   - erhält p_i innerhalb T eine a-Nachricht, wartet er eine längere Zeit T'. Kommt bis dahin keine c-Nachricht, startet er den Algorithmus neu.
2. Erhält p_j eine e-Nachricht von p_i, antwortet er und startet ebenfalls den Algorithmus
3. Erhält p_j eine c-Nachricht von p_i, setzt er *elected* = p_i
4. Ist ein Prozess nach Ausfall wieder aktiv, startet er den Algorithmus
5. der Prozess mit höchster PID ernennt sich selbständig zum Koordinator und schickt c-Nachricht an alle



P2 bemerkt als erster den Ausfall des bisherigen Koordinators P5. P3 und P4 starten den Algorithmus aufgrund ihrer höheren Prozessnummer.

20.4.4 Eigenschaften

- synchroner Algorithmus
- Es kann vorkommen, dass sich 2 Prozesse für den Koordinator halten

20.4.5 Komplexität

- Bester Fall: N-2 Nachrichten: Prozess mit zweithöchster PID erkennt Ausfall des Koordinators, ernennt sich selbst und schickt N-2 c-Nachrichten
- Schlechtester Fall: O(N^2) Nachrichten: Prozess mit kleinster ID schickt allen N-1 höheren Prozessen e-Nachrichten, diese schicken N-2 e-Nachrichten usw...

20.5 Echo-Algorithmus

Für beliebige zusammenhängende, ungerichtete Graphen.
Erzeugt einen Spannbaum mit dem Initiator als Wurzel. Hin- und Rückwelle mit gleichen Tokens.

```
received = 0;
father = NULL;
Initiator:
        forall (q in neighbours) do
                send <token> to q;
        while (received < #neighbours) {
```

```
                receive <token>;
                received = received + 1;
        }
        decide

Participant:
        receive <token> from neighbour q;
        father = q;
received = received + 1;
        forall (q in neighbours, q notEquals father) do
                send <token> to q;
        while (received < #neighbours) do {
                receive <token>;
                received = received + 1;
}
send <token> to father;
```

20.5.1 Eigenschaften

- Spannbaum
  - durch father-Kanten gegeben, über den die Rücknachrichten laufen
  - Spannbaum wird aber in Hinwelle festgelegt
  - Abschluss des Baumaufbaus über Rückwelle indiziert
- Komplexität: Auf jeder Kante werden max. 2 Nachrichten gesendet → O(2E)
- Termination: Jeder Prozess sendet maximal eine Nachricht auf jeder Kante
- Entscheidung: Der Initiator (Wurzel) erhält auf jeder Kante eine Nachricht, sofern alle Prozesse korrekt sind

Beispiel Foliensatz 9 ab Folie 23!

21 Transaktionen

21.1 Definition

Folge von Operationen, die als unteilbare logische Einheit ausgeführt wird. Es werden entweder alle oder keine Operationen ausgeführt und das Ergebnis ist erst am Ende sichtbar.

21.2 Transaktionen in Verteilten Systemen

- Klassische Transaktionen trennen zwischen Datenbasis (DB) und Client/Anwendung. Hierbei liegt der Fokus auf Konsistenz der Daten, nicht auf Verfügbarkeit. Ablauf: konsist. DB → BOT → read/write → EOT → konsist. DB
- Verteilte Systeme legen den Fokus auf Verfügbarkeit. Konsistenz ist oft unklar. Das Transaktionssystem löst zwei zentrale Probleme: **Ausfalltransparanz** (wie oben) und zusätzlich **Nebenläufigkeitstransparenz**

21.3 Fehlermodell

Das Transaktionssystem garantiert immer einen konsistenten Zustand, auch bei nebenläufigem Zugriff, wenn Server oder Clients abstürzen oder Nachrichten verspätet oder gar nicht ausgeliefert werden. Die folgenden Ausfälle werden toleriert:

- Fehlschlagen von Schreiboperationen oder Defekt von Speicher anhand einer Prüfsumme
- Absturz von Server oder Clients, beim Neustart wird der vorige Zustand aus Logs wiederhergestellt
- Nachrichten werden verzögert oder gehen verloren, dürfen aber nicht gefälscht sein
- **Byzantinische (bösartige) Fehler sind nicht erlaubt!**

21.4 ACID

- **Atomicity:** Alle oder keine Operationen werden ausgeführt
- **Consistency:** Transaktion überführt das System von einem konsistenten Zustand in einen anderen
- **Isolation:** Tansaktionen arbeiten unabhängig voneinander
- **Durability:** Nach EOT sind Ergebnisse persistiert

21.5 Nebenläufige Transaktionen

Ziel: Möglichst viel Nebenläufigkeit erlauben und dabei Konflikte korrekt einkalkulieren. Ohne Nebenläufigkeitskontrolle können folgende Probleme auftreten:

- Lost Update: Zwei Transaktionen lesen alten Wert und arbeiten beide darauf
- Inconsistent Retrieval: Transaktionen arbeiten auf inkonsistenten Zwischenwerten

21.5.1 Serielle Äquivalenz

Zwei Operationen stehen zueinander in Konflikt, wenn sie auf die selben Daten zugreifen und mindestens eine Operation schreiben will.
Zwei Transaktionen stehen in Konflikt, wenn es konfligierende Operationen gibt.
Zwei Transaktionen T_1 und T_2 sind **serialisiert** ($T_1 \rightarrow T_2$), falls gilt:
$\forall$ Paare der in Konflikt stehenden Operationen $op_1 \in T_1$ und $op_2 \in T_2$ gilt: $op_1 \rightarrow op_2$.
Die Ausführung der Transaktionen T_1, T_2, ..., T_m ist äquivalent zu ihrer Ausführung in serieller Reihenfolge $T_{i_1} \rightarrow T_{i_2} \rightarrow ... \rightarrow T_{i_m}$
Serielle Äquivalenz löst das Lost Update und Inconsistent Retrievals-Problem.

21.6 Recovery

Wenn eine Transaktion abgebrochen wird (abort), muss sichergestellt werden, dass andere Transaktionen keine Seiteneffekte sehen (Isolations-Eigenschaft).
Probleme:

- Dirty Read: Transaktion A liest Objekt einer Transaktion B, die später abgebrochen wird
- Premature Write: Interaktion zwischen Schreiboperationen auf demselben Objekt durch zwei Transaktionen, von denen eine abbricht.

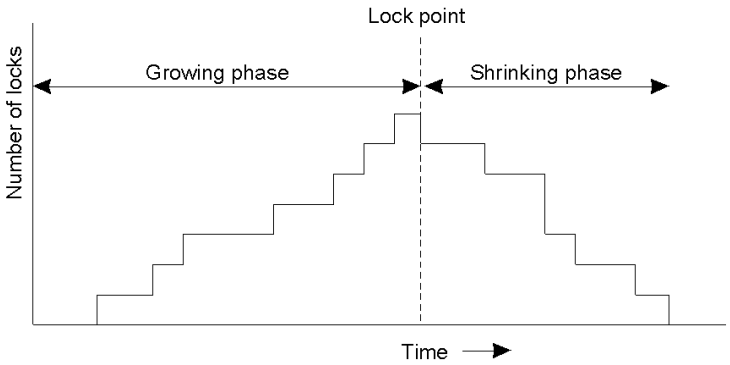
21.6.1 Wiederherstellung nach Abbrüchen

Eine Transaktion, die festschreibt, nachdem sie die Ergebnisse einer noch offenen Transaktion benutzt hat, lässt sich nicht zurückfahren. Für die Wiederherstellbarkeit muss ein Commit solange verzögert werden, bis alle offenen Transaktionen abgeschlossen sind, deren Ergebnisse benutzt wurden.
Problem: kaskadierende Abbrüche!

21.7 Nebenläufigkeitskontrolle

Erstellung eines Ablaufplans für in Konflikt stehende Operationen, der serielle Äquivalenz einhält.
Realisierung: Durch Locks oder Regel: Transaktion darf keine neuen Sperren anfordern, nachdem sie die erste Sperre aufgehoben hat.

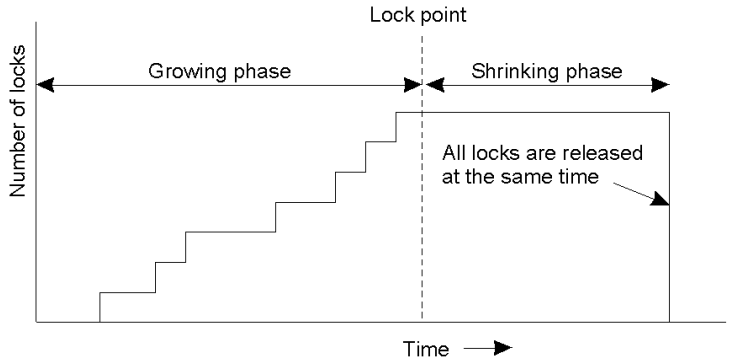
21.7.1 2 Phase Locking



Ablauf:

- Bei Eingang 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.
- Scheduler gibt Lock für x erst ab, wenn Data Manager bestätigt, dass die Operation ausgeführt wurde, für die gelockt wurde.
- Nachdem ein Lock für T aufgehoben wurde, wird kein neues Lock mehr für T erteilt

Problem: Kaskadierende Abbrüche
Lösung: Strong 2PL: Alle Locks werden zur gleichen Zeit aufgehoben:



21.7.2 Granularität

Ein Sperrtyp für alle Daten ist nicht praktikabel. Eine bessere Lösung ist es, viele Lese- aber nur einen Schreibzugriff zuzulassen. Leseoperationen verwenden shared locks, Schreibzugriffe verwenden exclusive locks. Lesesperren können zu

Schreibsperren „promoted“ werden, wenn sie nicht geshared sind.
Regeln:

- Wenn T auf einem Objekt eine **Leseoperation** durchgeführt hat, darf eine nebenläufige Transaktion U darauf nicht schreiben, bis T ein commit oder abort durchgeführt hat.
- Wenn T auf einem Objekt eine **Schreiboperation** durchgeführt hat, darf eine konkurrierende Transaktion U weder lesen noch schreiben, bis T ein commit oder abort durchgeführt hat.

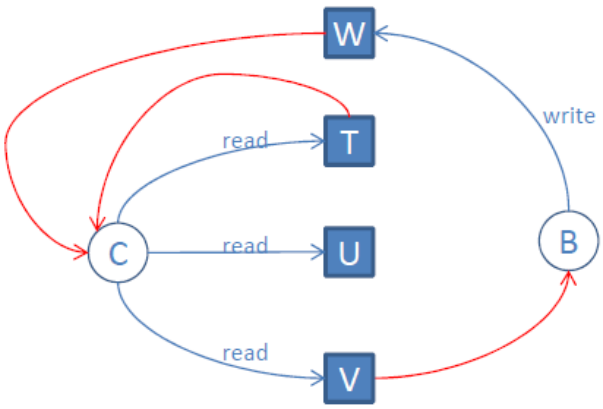
21.8 Deadlocks

Zustand, bei dem jedes Mitglied einer Gruppe darauf wartet, dass ein anderes die Sperre freigibt. Je feingranularer die Nebenläufigkeitskontrolle, desto geringer die Gefahr von Deadlocks.

21.8.1 Wait-for-Graph

beschreibt Wartebeziehungen zwischen Transaktionen. Knoten = Transaktionen, Kanten sind Wartebeziehungen. Eine Kante existiert von T nach U, wenn T darauf wartet, dass U eine Sperre freigibt.
Beispiel:

- T, U und V besitzen Lesesperre für c (blau)
- W besitzt Schreibsperre für b (blau)
- T und W wollen Schreibsperre auf c setzen (rot)
- jede Transaktion wartet nur auf ein Objekt, trotzdem ist V in 2 Zyklen (V-W, V-W-T)
- Lösung: V abbrechen und die Zyklen auflösen



21.8.2 Deadlockerkennung

Sperrmanager analysiert die Wait-for-Graphen. Wenn er einen Zyklus enthält, ist das System in einem Deadlock. Beim Einfügen neuer Kanten wird jeweils auf Zyklen überprüft. Wenn ein Zyklus gefunden wird, wird die Transaktion gesucht, deren Abbruch zur Auflösung der Zyklen führt. Mögliche Kandidaten sind die Transaktion mit den meisten Zyklen oder die, deren Sperre bald abläuft.

21.8.3 Vermeidung von Deadlocks

Die einfachste Lösung ist es, bei Transaktionseintritt einfach alle benötigten Sperren zu erwerben, was jedoch zu restriktiv ist. Besser ist es, die benötigten Objekte in einer definierten Prioritätsreihenfolge zu sperren.

21.9 Optimistische Nebenläufigkeitskontrollen

Annahme: Konflikte treten eher selten auf; Transaktionen arbeiten, als wären sie alleine. Tritt ein Konflikt auf, muss eine Transaktion abgebrochen werden.
3 Phasen:

1. Arbeitsphase: Transaktion besitzt eigene Kopie aller Daten und arbeitet darauf
2. Validationsphase: Nach Abschluss wird überprüft, ob es Konflikte mit anderen Transaktionen gibt/gab. Falls ja, müssen diese aufgelöst werden.
3. Aktualisierungsphase: Wurde die Transaktion validiert, werden die Daten persistiert

21.9.1 Validation

Zu allen überlappenden Transaktionen wird serielle Äquivalenz überprüft - siehe 21.5.1.
Überlappend sind alle Transaktionen, die noch nicht abgeschlossen waren, als die neue Transaktion startete (BOT).
Transaktionen werden in Reihenfolge des Eintritts in Validationsphase nummeriert. Da Validationsphase kurz, kann sie als kritische Sektion implementiert werden. Dies erfordert eine global eindeutige aufsteigende Nummerierung.
Strategien

- Rückwärtsvalidation: Regeln werden mit denjenigen Transaktionen geprüft, die vorher in Validationsphase eintraten
- Vorwärtsvalidation: Regeln werden mit noch aktiven Transaktionen geprüft, die später begonnen haben

21.9.2 Rückwärtsvalidation

Teste ob der eigene Read-Set [1] mit dem Write-Set früherer Transaktionen überlappt. Falls ja: Abbruch. Sehr leicht zu implementieren

21.9.3 Vorwärtsvalidation

Teste, ob der eigene Write-Set mit dem Read-Set paralleler Transaktionen überlappt. Falls ja, gibt es 2 Möglichkeiten:

1. Abbruch der anderen und Fortführung der eigenen Transaktion
2. Verzögerung der Validation, bis die konkurrierenden Transaktionen beendet sind (derweil können aber weitere Transaktionen starten).

Vorteil: Lesetransaktionen haben nie Konflikte.

21.9.4 Zeitstempelverfahren

Alle Transaktionen werden durch Start-Zeitstempel eindeutig geordnet. Anforderungen können so vollständig durch Zeitstempel sortiert werden.
Regeln:

- Die Anforderungen einer Transaktion, ein Objekt zu schreiben, ist nur dann gültig, wenn das Objekt zuletzt von einer früheren Transaktion gelesen oder geschrieben wurde
- Die Anforderung einer T., ein Objekt zu lesen, ist nur dann gültig, wenn das Objekt zuletzt von einer früheren T. geschrieben wurde.

Jede Transaktion T bekommt einen eindeutigen Zeitstempel ts(T). Jede Operation von T besitzt diesen. Jedes Datenobjekt besitzt 2 Zeitstempel: $ts_{RD}(x)$ und $ts_{WR}(x)$. $ts_{RD}(x)$ enthält ts(TI), wobei TI die letzte Transaktion ist, die x gelesen hat. $ts_{WR}(x)$ analog für Schreibzugriff.
Konfliktlösung:

1. Scheduler erhält read(T,x) mit Zeitstempel ts: Wenn ts $< ts_{WR}(x)$ wurde die letzte WR Operation nach dem Start von T durchgeführt $\rightarrow$ T wird abgebrochen. Wenn ts $> ts_{WR}(x)$, darf das Read stattfinden. $ts_{RD}(x)$ wird auf max(ts, $ts_{RD}(x)$) gesetzt.
2. Scheduler erhält write(T,x) mit ts. Ist ts $< ts_{RD}(x)$, wird T abgebrochen, da eine jüngere Transaktion x gelesen hat. Ist ts $> ts_{RD}(x)$, darf der Wert geschrieben werden. Außerdem wird $ts_{WR}(x)$ auf max(ts, $ts_{WR}(x)$ gesetzt.

[1] Read-Set enthält alle Leseoperationen, die eigene Schreiboperationen beeinflusst haben

22 Structured Overlay Networks

22.1 Gossiping

- Pull: Hole die Daten von Peer (am effektivsten, da keine Nachrichten versendet werden, wenn der Peer evtl. schon up to date ist)
- Push: Sende Updates an Peers
- PushPull

22.2 Distributed Hash Tables (DHT)

...ist eine normale Hashtabelle, die verteilt ist. Jeder Knoten bietet eine lookup Operation an, um einen Wert für einen Schlüssel zurückzuliefern. Außerdem hält er routing pointers. Wird ein Wert nicht gefunden, wird auf einen anderen Knoten verwiesen.
DHT's sind sehr gut skalierbar. Die Zeit, um einen Wert zu finden, ist genau wie die Größe der Tabelle **logarithmisch**. Die Tabelle ist selbstverwaltend, jeder Knoten ist dafür verantwortlich, Daten und Routing Pointers aktuell zu halten.

22.2.1 Wofür DHTs?

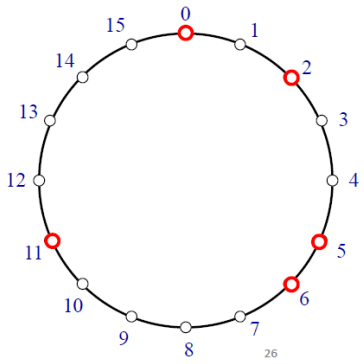
Verteilte Authorisierungssysteme: funktioniert auch, wenn einzelne Knoten angegriffen werden.
Verteiltes Backup
Verteiltes Dateisystem (braucht Replikation, für Schreibzugriffe wird Transaktionsmanagement benötigt)

22.3 Chord

Verwende einen logischen Namensraum - Identifikationsraum - der aus ID's {0,1,2,...,N-1} besteht. Der Identifikationsraum ist ein logischer Ring mod N und jeder Knoten wählt eine zufällige ID durch Hash H.

Example:

- Identifier space **N=16 {0,...,15}**
- Five nodes **a, b, c, d, e**
  - a picks 6
  - b picks 5
  - c picks 0
  - d picks 11
  - e picks 2



Verteilte Systeme 26

Der Nachfolger ist der erste folgende Knoten, wenn man den Ring im Uhrzeigersinn abläuft.

22.3.1 Datenspeicherung

Global bekannte Hashfunktion wird verwendet, um jedem key/value Paar eine ID H(key) zuzuweisen. Jedes item wird beim Nachfolger gespeichert. Jeder Knoten hält außerdem einen Zeiger auf seinen Nachfolger und seinen Vorgänger. Der Nachfolger eines Knotens n ist succ(n+1).

22.3.2 DHT Lookup

Für Schlüssel k, berechne H(k) und folge den Nachfolger-Zeigern, bis k gefunden wurde.
Definitionen:

- (a,b] ist der Teil des Rings im Uhrzeigersinn von a (ohne a selbst) und bis einschließlich b
- n.foo() ist ein RPC von foo() auf n
- n.bar ist ein RPC um den Wert der Variablen bar von n zu holen

Put und Get sind ebenfalls Lookups. Um den Lookup zu beschleunigen, wird eine Routingtabelle („finger table“) verwendet:

1. Finger zeigt auf succ(n+1)
2. Finger zeigt auf succ(n+2)
3. Finger zeigt auf succ(n+4)
4. letzter Finger zeigt auf succ(n+2^{M-1})

22.4 Ring Maintenance

Periodisch werden die Nachfolge- und Vorgängerzeiger aktualisiert, sodass sie auf den nächsten Vorgänger bzw. Nachfolger zeigen. (Ringstabilisierung)
Wenn ein Knoten n neu dazukommt, wird sein Nachfolger durch lookup(n) gefunden und der Zeiger gesetzt. Den Rest übernimmt die Stabilisierung.
Wenn ein Ring mit einem Knoten n neu erstellt wird, ist sein Nachfolger er selbst, der Vorgänger ist leer.
Die Tabelleneinträge der finger table werden periodisch aktualisiert und der Index des nächsten zu reparierenden Finger wird gespeichert (initial 0).

22.5 Fehlerbehandlung

Der Ausfall eines Nachfolgerzeigers bedeutet den Zusammenbruch des Rings.
Lösung: Ein Knoten hält eine Liste an Nachfolgern der Größe r. r = log(N) ist eine sinnvolle Wahl.
Fällt ein Vorgänger aus, wird dieser Zeiger genullt.

23 Transaktionen II: Non-Blocking atomic commit

23.1 Konsens- vs. Commit-Protokolle

Konsensprotokolle entscheiden, wie trotz unzuverlässiger Komponenten eine eindeutige Entscheidung gefällt wird, die am Ende allen bekannt ist, zB durch Mehrheitsentscheidung. Ein Beispiel dafür ist Paxos Consensus.
Commitprotokolle realisieren Transaktionen durch Einsatz von Konsensprotokollen, zB 2PC, 3PC, Paxos Commit (nutzt Paxos Consensus)

23.2 Fehlermodell

Nicht-Byzantinisches Fehlermodell
Erlaubte Fehler:

- Prozesse sind unterschiedlich schnell, können jederzeit abstürzen oder neu starten
- Nachrichten können verzögert oder dupliziert werden oder verloren gehen

Nicht erlaubt ist unerkannt korrumpierter Speicher oder Nachrichten
Annahme: Es liegt nichtflüchtiger Speicher vor

23.3 2PC, 3PC

23.3.1 2PC Algorithmus

Transaktionsmanager, Phase 1:

- empfange req_commit
- erzeuge Transaktionsnummer T und schreibe prepare(T) in Log
- sende vote_request(T) an alle Resource Manager
- warte auf Antworten

Transaktionsmanager, Phase 2:

- falls von allem RMs vote_commit(T) erhalten
  - schreibe global_commit(T) in Log
  - sende global_commit(T) an alle RM
  - führe commit lokal durch und lösche Eintrag in Log
- falls von ≥ 1 RM vote_abort(T) erhalten oder Timeout:
  - schreibe global_abort(T) in Log
  - sende global_abort(T) an alle RM

Resource Manager, Phase 1:

- empfange vote_request(T)
- schreibe vote_abort(T) oder vote_commit(T) in Log
- sende Entscheidung an TM
- falls vote_abort(T) führe lokal abort durch, lösche Eintrag in Log

Resource Manager, Phase 2:

- warte auf global_abort(T) oder global_commit(T) von TM
- schreibe global_abort(T) oder global_commit(T) in Log
- antworte an TM mit ack
- führe abort oder commit lokal durch und lösche Logeintrag

Komplexität: 1 TM und N RMs
Anzahl Operationen: 3N+1 Nachrichten, N+1 Schreiboperationen in Log
Gesamt-Latenzzeit im Erfolgsfall: 4 Nachrichtenverzögerungen + Zeit für 2 Disk-Schreibzugriffe
Eigenschaften

- Atomarität: Alle oder keiner durch Protokoll garantiert
- Konsistentz: garantiert durch Atomarität
- Isolation: 2PC hat keine Nebenläufigkeitsanomalien, da Ergebnisse erst nach global_commit veröffentlicht werden
- Dauerhaftigkeit: Zustand ist stets persistiert
- Verfügbarkeit: **Nein! 2PC blockiert**, wenn TM in Phase 2 abstürzt

23.3.2 3PC

Einbinden einer dritten Phase. TM sendet vote_request() wie in 2PC und schreibt bei positiver Antwort aller RMs *PRECOMMIT* in den Log. Daraufhin sendet der TM *prepare_ commit* an alle RM und wartet auf ready_commit Antworten. Danach wird commit ins Log geschrieben und global_commit an alle RMs gesendet.
Fehlerszenarien:
Bei Wiederanlauf:

- TM blockierte in WAIT: abort
- TM blockierte in PRECOMMIT: dieser oder Ersatz-TM sendet global_commit an alle RM

- RM blockierte im PRECOMMIT: commit
- RM blockierte in READY
  - er hatte vote_commit entschieden
  - aber TM hat evtl. noch nicht entschieden
  - bei Wiederanlauf irgendeinen anderen RM kontaktieren
    - falls dieser in PRECOMMIT → precommit (alle RMs sind in PRECOMMIT)
    - sonst abort

23.3.3 Vergleich 2PC - 3PC

2PC	**3PC**
blockiert, wenn TM ausfällt	nicht-blockierend
braucht bei Erfolg 4 Nachrichtenübertragungen	braucht 6 Nachrichtenübertragungen
	benötigt perfekten Ausfalldetektor
	für global verteilte Systeme 6 Schritte kaum akzeptabel

23.4 Paxos

23.4.1 Problemstellung

Ausfalltolerante Systeme ohne Single-Point-of-Failure. Der Transaktionsmanager muss ausfalltolerant sein und die Entscheidung über Commit/Abort muss dezentral erfolgen. Der Zustand muss dezentral gespeichert sein.
Lösungsidee:

- **Paxos Consensus:** Nutzung eines verteilten, ausfalltoleranten Konsens als Hilfsmittel
- **Paxos Commit:** Transaktionsprotokoll mit Hilfe des Konsensverfahrens

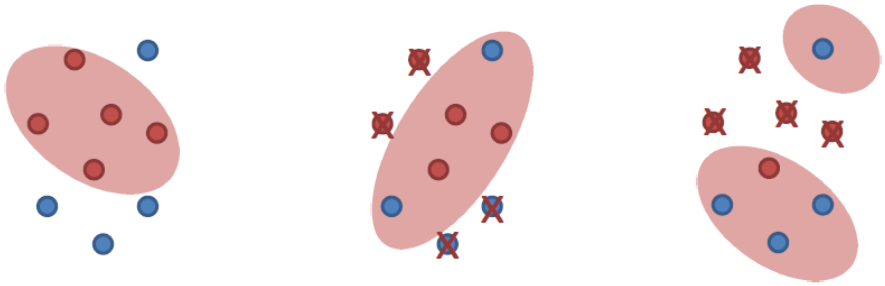
23.4.2 Paxos Consensus

Konsens in einer Gruppe verteilter Prozesse
Eigenschaften:

- Sicherheit:
  - Es wird nur ein Wert gewählt
  - Es wird nur ein Wert gewählt, der zuvor vorgeschlagen wurde
  - Ein Prozess erfährt nur dann dass ein Wert gewählt wurde, wenn das auch passiert ist
- Liveness
  - Wenn hinreichend viele korrekte Prozesse existieren, wird letztendlich ein Wert gewählt

– Falls ein Wert gewählt wurde, wird jeder beteiligte Prozess ihn letztendlich erfahren

Paxos ist sicher, aber nicht notwendigerweise lebendig. Der Algorithmus benötigt Timeouts oder Ausfalldetektoren.
Idee: Mehrheitsentscheidung in einer Gruppe. Wenn die Mehrheit informiert ist, hat in jeder anderen Mehrheitsmenge mindestens ein Mitglied die aktuelle Information. Eine beliebige Minderheit darf ausfallen.



1. Bevor ein Prozess einen eigenen Wert für den Konsens vorschlagen kann, prüft er, ob nicht schon ein Konsens erzielt wurde
2. Er versucht den bisherigen (oder falls möglich seinen eigenen) Wert bei der Mehrheit der Gruppe zu etablieren

Nebenläufigkeiten werden über die Rundennummer kontrolliert - nur Anfragen mit der größten Rundennummer werden beachtet.
2. gelingt nur, wenn nach 1. keine höhere Rundennummer auftauchte.
Voraussetzungen:

- 3 Rollen: Proposer, Acceptor, Learner
- asynchrone Kommunikation
- Crash Recovery (für Acceptors nichtflüchtiger Speicher vonnöten)
- Acceptors sind vorab bekannt inkl. ihrer Anzahl
- Minderheit der Acceptors darf ausfallen

Rollen:

- Proposer: treiben mit eindeutigen Rundennummern das Konsensprotokoll an. Proposer versucht ohne Unterbrechung eines anderen durch 1. und 2. Phase zu kommen, dann kommt es zu einem Konsens
- Acceptor: Bildet verteilten, replizierten Speicher. Sobald die Mehrheit der Acceptors ein Proposal akzeptieren, steht der Konsens fest und es gibt kein Zurück mehr.

- Learner: Sammelt den Konsens ein und verteilt die Entscheidung

Sonderfälle: Proposer verwendet zu kleine Rundennummer: nack/naccepted Nachricht mit aktueller Runde zurückschicken, dann lernt der Proposer die aktuelle Runde.
Zweiter Proposer kommt dazu: Neue Runde r+x wird gestartet, ältere accepts werden nicht bestätigt. Wechselseitige Behinderung ist möglich.
Proposer fällt nach prepare/accept Nachricht aus: Anderer Proposer startet neue Runde r+x.
Komplexität:
Paxus Consensus:
Nachrichten: $\geq$ maj * 4 (+1 fürs decide)
Nachrichtenverzögerungen: ≥ 4
Fast Paxos Consensus
Kleinste Rundennummer r=1 kann auf erste Phase verzichten, da noch kein Konsens vorhanden sein kann.
Nachrichten: $\geq$ maj * 2 (+1 fürs decide)
Nachrichtenverzögerungen: ≥ 2

23.4.3 Commit mit Konsens

Drei Rollen:

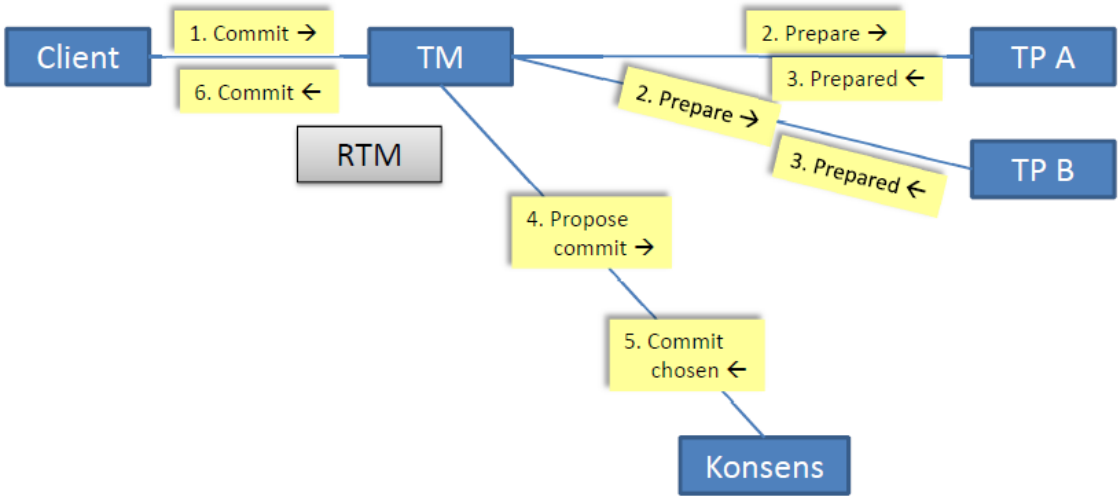
- 1 Transaction Manager
- mehrere Replizierte Transaction Manager
- mehrere Transaction Participants, einer je Ressource, die die zu einer Transaktion gehörenden lokalen Operationen ausführen (bei 2PC und 3PC RMs genannt)

Annahmen: Beim Start sind alle Teilnehmer bekannt und alle TPs sowie eine Mehrheit der RTM sind erreichbar. Byzantinische Fehler sind verboten.

Konsens über die Entscheidung des TM

TM „lernt“ den Wert des Konsens

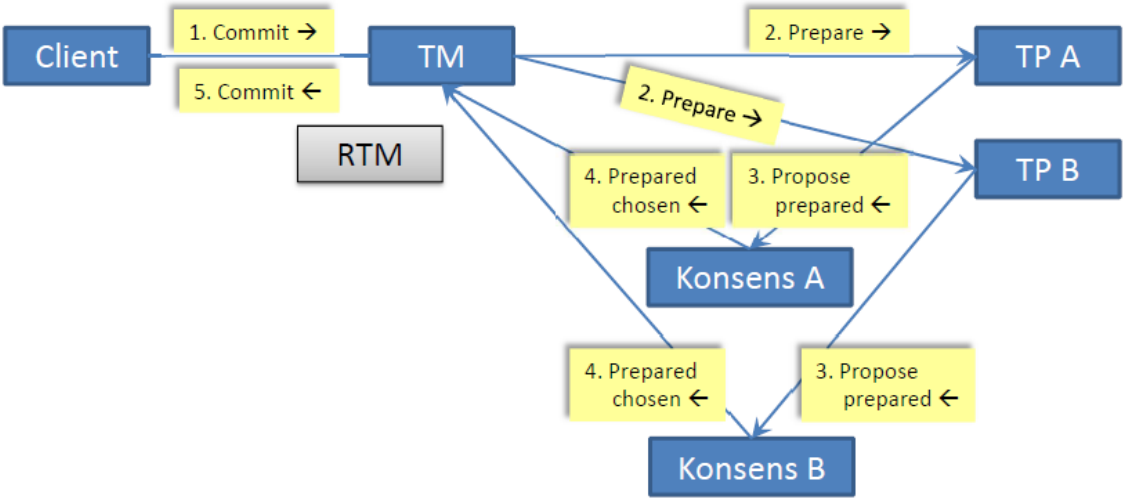
TM ist zustandslos



Konsens über jede TP Entscheidung

TM kombiniert Einzelentscheidungen

TM ist zustandslos



Wozu Paxos Consensus in Paxos Commit?
Nutze die Consensus Box mit ihren 2F+1 Acceptors als replizierte TMs: Es gibt dann 2F RTMs + 1TM. Sobald F+1 RTMs von allen RMs den Status prepared sehen, ist der Status committed.

Commit mit Konsens: Vergleich

Naheliegender Ansatz	**Paxos Commit**
6 Schritte	5 Schritte (mehr Nachrichten)
TM kann Lösung über Konsens lernen	TM kann Einzelentscheidungen lernen und kombinieren
RTM kann Lösung über Konsens lernen oder selbst abort vorschlagen	RTM kann Einzelentscheidungen lernen oder abort vorschlagen und Gesamtentscheidung fällen.

Vergleich 2PC und Paxos Commit:

2PC	**Paxos Commit**
3N+1 Nachrichten	3N + 2F(N+1) + 1 Nachrichten
N+1 stabile writes	N+2F+1 stabile writes
4 Nachrichtenverzögerungen	5 Nachrichtenverzögerungen
2 stabile Schreibverzögerungen	2 stabile Schreibverzögerungen toleriert F Ausfälle!

Wenn F = 0 und TM = Acceptor: 2PC == Paxos