Verteilte Betriebssysteme

12. Kapitel
Transaktionen und Replikationen

Matthias Werner
Professur Betriebssysteme

12.1 Einführung

- In Kapitel 11 nur **einzelne** verteilte Lese- und Schreibaktion
- Problem stellt sich für **Sequenzen** z.T. noch verschärft dar
- Allgemeines Konzept: **Transaktionen** ➡ wohlbekannt von Datenbanken
  - In verteilten Systemen auch außerhalb klassischer Datenbanken genutzt, siehe z. B. Plurix und Rainbow-OS (➡ Kapitel 3)
- Datenbank dient hier als Modell
- Wiederholung:

Transaktion

Eine Transaktion T_i ist eine Folge von Schreib- oder Leseoperationen (w_i oder r_i), abgeschlossen durch entweder ein Commit (c_i) oder ein Abort (a_i).

Wiederholung

- ACID-Forderungen:
  - **Atomicity**: Alles oder nichts
  - **Consistency**: Vor- und Nachzustand ist konsistent.
  - **Isolation**: Zwischenergebnisse (von aussen) nicht sichtbar
  - **Durability**: Commit wird nicht zurückgenommen.
- **Konflikt**: Zwei Operationen aus unterschiedlichen Transaktionen auf das gleiche Datum, wobei mindestens eine eine Schreiboperation ist
- **Plan**: Partiell geordnete Menge aller Transaktionsoperationen einer Menge von Transaktionen $T_1, T_2, \ldots, T_n$, bei der lokalen Reihenfolgen erhalten bleiben und Konflikte geordnet werden

Wiederholung (Forts.)

- **Eigenschaften von Plänen**
  - Ein Plan S heißt **rücksetzbar** (*recoverable*, $S \in RC$), wenn jede Transaktion erst dann bestätigt wird, wenn alle Transaktionen, von denen sie gelesen hat, bereits bestätigt sind oder abgebrochen wurden
  - Ein Plan S **vermeidet einen kaskadierenden Abbruch** (*avoiding cascading abort*, $S \in ACA$), wenn keine Transaktion aus S unbestätigte Daten liest
  - Ein Plan S heißt **strikt** ($S \in ST$), wenn von keiner Transaktion aus S unbestätigte Daten gelesen oder überschrieben werden

Wiederholung (Forts.)

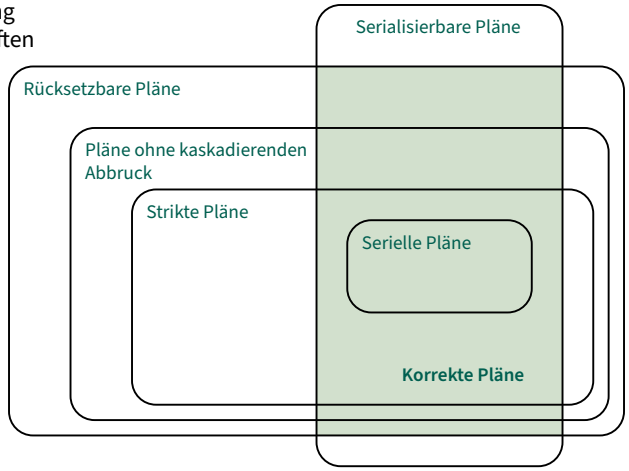
Bestätigte Projektion

Die bestätigte Projektion $C(S)$ eines Plans S ist die Teilmenge von S, die entsteht, wenn man aus S alle Operationen entfernt, die nicht zu in S bestätigten Transaktionen gehören

- **Eigenschaften von Plänen**
  - Ein Plan S heißt **seriell**, wenn für jedes Paar von Transaktionen alle Operationen der einen vor jeder Operation der anderen ausgeführt werden
  - Zwei Pläne S und S' heißen **äquivalent**, wenn sie dieselben Ausgabewerte liefern und denselben Datenzustand zurücklassen.
  - Ein Plan S heißt **serialisierbar** ($S \in SR$), wenn seine bestätigte Projektion $C(S)$ zu einem seriellen Plan äquivalent ist
  - Ein Plan S heißt **korrekt**, wenn er *serialisierbar* und *rücksetzbar* ist
- Zur Realisierung von korrekten (seriellisierbaren und rücksetzbaren) Ausführungen wird für Transaktionen das Zweiphasen-Sperrprotokoll genutzt (*two phase lock*, 2PL)

Wiederholung (Forts.)

- Zusammenhang von Eigenschaften



Wiederholung (Forts.)

- **Zweiphasen-Sperrprotokoll**
  1. Eine Transaktion muss jedes Datenelement vor dem ersten Zugriff mit einer dem Zugriff entsprechenden Sperre (Lesesperre/Schreibesperre) belegen
  2. Kein Datenelement darf mit unverträglichen Sperren belegt werden
     - Lesesperre ➡ keine Schreibsperre
     - Schreibsperre ➡ keine weitere Sperre
  3. Eine Transaktion darf nach der ersten Freigabe einer Sperre keine weitere Sperre setzen
  4. Am Ende der Transaktion müssen alle von ihr gehaltenen Sperren freigegeben sein
- **Striktes 2PL**
  - Alle jemals erworbenen Sperren werden bis zum Ende der Transaktion gehalten
- **Konservatives 2PL**:
  - Alle Sperren werden zu Beginn gesetzt

12.2 Deadlock

Verteiltes 2PL

- In einem verteilten System gibt es entweder
  - einen **zentralen** Transaktionsmanager (TM) ➡ Flaschenhals, Single Point of Failure, auch lokale Transaktionen müssen global gemacht werden
  - eine pro Knoten ➡ Modellannahme im Folgenden
- **Verteiltes 2PL**
  - Zugriffswünsche werden an den Knoten gesendet, der das Datum verwaltet
  - lokaler Scheduler kann **nicht** feststellen, ob eine Transaktion an einem anderen Ort schon Sperren freigegeben hat ➡ Verletzung der Zweiphasen-Regel
  - Lösung: **striktes** Sperren ➡ Sperren werden bis Commit gehalten

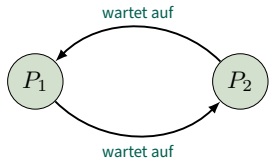
  ➡ **Deadlock** möglich

Wiederholung: Deadlock

- Deadlocks (Verklemmungen) sind zyklische Wartebedingungen
- Sie können auftreten, wenn das Systemdesign folgende Bedingungen erfüllt:
  1. Betriebsmittel werden exklusiv genutzt
  2. Prozesse besitzen schon Betriebsmittel, während sie auf andere warten
  3. Verdrängung findet nicht statt

Zyklischer Wartegraph

P_1	P_2
`lock(x)`	
	`lock(y)`
`lock(y)`	
	`lock(x)`



- Verklemmungen sind gekennzeichnet durch eine **zyklische Wartesituation**
- Zur Darstellung kann man den sogenannten **Wartegraphen** verwenden
- Eine Verklemmung liegt vor, wenn der Graph Zyklen besitzt

Ansatz: Zentrale Zyklenentdeckung

- Ein globaler **Verklemmungsentdecker** (*global deadlock detector*, *GDD*) existiert zentral an einem Knoten
  - Sammelt periodisch lokale Wartegraphen und vereinigt sie zu einem globalen Wartegraphen
  - Prüft den Wartegraphen auf Zyklen
  - Wählt bei Verklemmung ein „Opfer“ (abzubrechenden Prozess) aus
  - ➡ Wegen der nicht ganz aktuellen Information über die globale Wartesituation können sogenannte **Phantom-Verklemmungen** auftreten, d.h. Verklemmungen, die gar nicht existieren

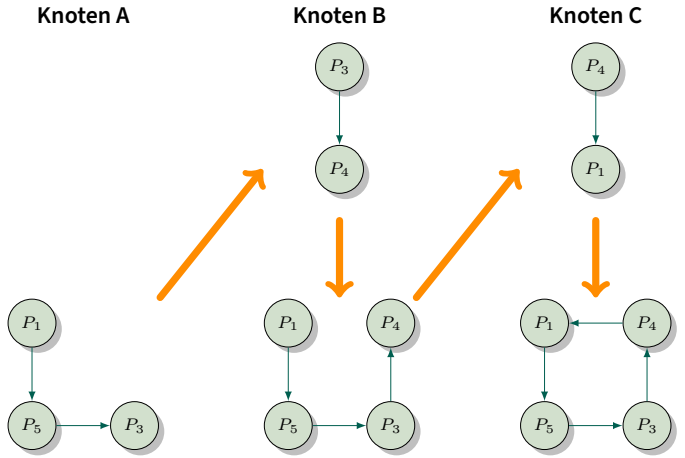
Verteilte Zyklenentdeckung (Path Pushing)

- Das Zusammensetzen des globalen Wartegraphen kann auch **dezentral** erfolgen
- Jeder Knoten unterhält seinen lokalen Wartegraphen als Teil des globalen Wartegraphen
- Jeder Knoten kennt:
  - den Besitzer einer Sperre auf ein lokales Datum,
  - die Bewerber auf eine Sperre auf ein lokales Datum
  - die lokalen Prozesse und ihre noch nicht abgeschlossenen kritischen Datenoperationen und deren Ausführungsorte

Verteilte Zyklenentdeckung (Path Pushing) (Forts.)

- Bei jedem Einfügen einer Kante wird eine Zyklenprüfung durchgeführt
  - Wird ein Zyklus gefunden, so liegt eine lokale Verklemmung vor
  - Wird kein Zyklus gefunden, so könnte eine verteilte Verklemmung vorliegen, weshalb alle gefundenen Pfade als **deadlock detection messages** (DDM) an betroffene Knoten geschickt werden
  - Beim Empfang einer DDM fügt der Knoten den Pfad in den lokalen Graphen ein und führt wiederum eine Zyklenprüfung durch
- Nach endlich vielen Schritten wird eine Verklemmung erkannt oder der Algorithmus hält an

Beispiel



Diskussion

Die Effizienz des Algorithmus kann gesteigert werden

- Jeder Knoten schickt nur Pfade $T_i, \ldots, T_j$, für die $i < j$ gilt
  ➡ nur die Hälfte der Pfade werden verschickt
- Jeder Knoten schickt seine DDM zu dem Ort, an dem weitere Information über die **letzte** Transaktion T_j des Pfads gefunden werden kann
  - T_j ist lokale Transaktion ➡ da der Transaktionsmanager entfernte Operationen verschickt und auf Rückmeldung wartet, ist der Ort einer Blockierung von T_j bekannt
  - T_j ist Besitzer einer lokalen Sperre ➡ Heimatort von T_j ist bekannt
  - T hat eine lokale Sperre angefordert ➡ Heimatort von T_j ist bekannt
- Es werden immer noch **Phantom-Verklemmungen** erkannt

Chandy-Misra-Haas-Algorithmus

- Der **Chandy-Misra-Haas-Algorithmus** kommt mit **weniger** Nachrichten aus und produziert auch keine Phantom-Verklemmungen
  - Jeder Knoten hat einen **Koordinator**, der die lokalen Prozesse überwacht
  - Muss ein Prozess P_i auf einen anderen nichtlokalen Prozess P_j warten, so schickt der Koordinator eine **Test-Nachricht** (P_i, P_i, P_j) (*probe message*) an den Koordinator des Knotens, auf dem sich Prozess P_j befindet
  - Die Nachricht enthält die ID des „initiierenden“ Prozesses, die des wartenden Prozesses[1] und die ID des blockierenden Prozesses

[1] ...was anfangs der gleiche Prozess ist...

Chandy-Misra-Haas-Algorithmus (Forts.)

- Empfängt ein Koordinator eine Test-Nachricht (P_i, P_j, P_k), so überprüft er, ob P_k ebenfalls auf andere Prozesse (mindestens einen) wartet
- Wenn nicht, wird die Test Nachricht ignoriert
- Fall doch, wird überprüft ob $P_i = P_k$ gilt ➡ **Verklemmung erkannt**
- Sonst wird an für blockierenden Prozesse eine modifizierte Nachricht gesendet, bei der P_j und P_k durch den blockierten bzw. blockierenden Prozess ersetzt wird

Auswahl einer abzubrechenden Transaktion

- Wird ein Zyklus entdeckt, so muss eine der beteiligten Transaktionen **abgebrochen** (abort) werden
- Im Gegensatz zu einer „normalen“ Prozessverklemmung ist ein Abbruch in Transaktionen bereits vorgesehen
- Mögliche Auswahlkriterien:
  - bereits geleistete Arbeit der Transaktion
  - Rücksetzkosten der Transaktion
  - die erwarteten Restkosten (Zeit) zur vollständigen Bearbeitung der Transaktion
  - die Anzahl der Deadlock-Zyklen, in der sich eine Transaktion befindet
  - die Anzahl der Abbrüche, die eine Transaktion schon erlebt hat

Zeitstempel-Verfahren

- In einer Verklemmungssituation wartet eine Transaktion auf sich selbst
- Durch eine globale Ordnung der Transaktionen (Zeitstempel, time stamp), die im Wartegraph berücksichtigt wird (topologische Sortierung), können Zyklen verhindern werden:

T_i darf nur auf T_j warten, wenn $ts(T_i) < ts(T_j)$ ist, sonst erfolgt ein Abbruch.

- Beliebige Relation möglich
- Keine Synchronisation nötig, bei Gleichheit entscheidet Knoten-ID o.ä.
- **Problem:** es können zyklische Abbrüche auftreten (➡ Livelock)

Diskussion

- Bei den hier diskutierten Ansätzen wird davon ausgegangen, dass die Identität des Blockierenden bekannt ist
  - Dies ist aber nicht im Allgemeinen der Fall: Insbesondere im Fall der Blockierung bei **Mehrexemplarbetriebssmitteln** ist der Blockierer nicht eindeutig
  - Es existieren jedoch z.B. verteilte Varianten des **Banker-Algorithmus**
- Bei Erkennung eines Deadlock wird Transaktion abgebrochen (*abort*) und erneut versucht ➡ es kann zu zyklischen Abrüchen kommen (**Livelocks**)

12.3 Bestätigung verteilter Transaktionen

- Die Korrektheit eines Transaktionskonzepts baut auf der **Atomarität** der Commit-Operation
- Gewährleistung im verteilten Fall, wo es einer Abstimmung der Knoten bedarf, ist schwierig ➡ Knoten können ausfallen
- Anforderungen an **atomares Commit-Protokoll** (ACP)
  1. Alle Knoten, die eine Entscheidung treffen, treffen **dieselbe** Entscheidung
  2. Ein Knoten kann seine Entscheidung **nicht nachträglich** ändern
  3. Die Entscheidung, eine Transaktion zu bestätigen, kann nur von allen Knoten **einstimmig** getroffen werden
  4. Falls keine Ausfälle vorliegen und alle Knoten zugestimmt haben, wird die Transaktion **bestätigt**
  5. Nach Behebung eventueller Ausfälle muss eine Entscheidung getroffen werden

- Häufig sind wir bei verteilten Algorithmen von der Fehlerfreiheit oder **Atomarität** bestimmter Vorgänge ausgegangen
  - z.B. Broadcast
- Im nichtverteilten Fall kann Atomarität durch globale Auswirkung von Fehlern hergestellt werden
- Im verteilten Fall, wo **lokale** Ausfälle nicht ausgeschlossen werden können, wird dies schwierig
- Betrachen **Zustimmungsproblem** (commit problem)
  - Typisch: Bestätigung von Transaktionen

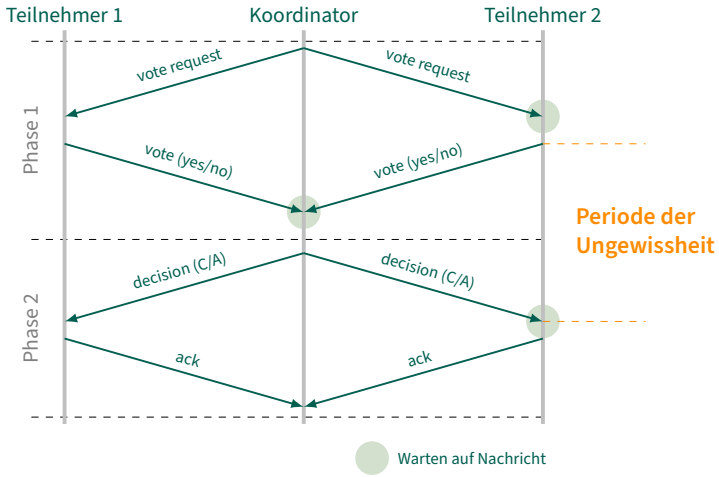
Anforderungen an Commit-Protokoll

- Ein **atomares Commit-Protokoll** (ACP) hat folgenden Anforderungen zu genügen:
  1. Alle Knoten, die eine Entscheidung treffen, treffen dieselbe Entscheidung
  2. Ein Knoten kann seine Entscheidung nicht nachträglich ändern
  3. Die Entscheidung, eine Transaktion zu bestätigen, kann nur von allen Knoten einstimmig getroffen werden
  4. Falls keine Ausfälle vorliegen und alle Knoten zugestimmt haben, wird die Transaktion bestätigt
  5. Nach Behebung eventueller Ausfälle muss eine Entscheidung getroffen werden

Das 2-Phasen-Commit-Protokoll (2PC)

- Einer der involvierten Knoten (üblicherweise der „Heimatknoten“ einer Transaktion) übernimmt die Rolle des **Koordinators**, alle anderen sind **Teilnehmer** (*participants*)
- Jeder Knoten unterhält eine spezielle Log-Datei, in die alle relevanten Ereignisse geschrieben werden
- Das Protokoll besteht aus vier Schritten:
  1. Aufforderung zur Stimmabgabe (*vote request*)
  2. Stimmabgabe (*vote*)
  3. Mitteilung über die Entscheidung (*decision*)
  4. Bestätigung der Entscheidung (*acknowledge*)

Struktur des Zwei-Phasen-Commit-Protokolls



Verhalten im Fehlerfall

- Fehler wird durch Fristablauf (Deadline) erkannt
- Mögliche Fälle:
  a) **Teilnehmer wartet auf Vote-Request**
     ➡ Falls keine Nachricht eintrifft, entscheidet der Teilnehmer „Abort“ und stoppt
  b) **Koordinator wartet auf Vote-Messages**
     ➡ Falls nicht alle Stimmabgaben eingetroffen sind, entscheidet der Koordinator „Abort“, sendet die entsprechenden Decision-Messages und stoppt
  c) **Teilnehmer wartet auf Entscheidung**
     ➡ Da die Entscheidung zu diesem Zeitpunkt vielleicht bereits getroffen ist, ist eine einseitige Entscheidung wie in a) nicht mehr möglich
     ➡ Teilnehmer p fragt einen anderen Teilnehmer q nach dem Ergebnis (Terminierungsprotokoll)

Verhalten im Fehlerfall (Forts.)

- Im Fall c) frage p einen Knoten q
- Dabei gibt es drei Möglichkeiten:
  - q hat die Entscheidung erhalten und gibt sie an p weiter
  - q hat noch nicht abgestimmt ➡ q kann einseitig „Abort“ entscheiden und eine entsprechende Nachricht an p senden
  - q ist selbst in der Ungewissheitsphase und kann p nicht helfen ➡ Fragen eines anderen Teilnehmers

Verhalten der Knoten bei Crash (Recovery)

1. Log enthält keinen „Start 2PC“-Eintrag (➡ Knoten ist Teilnehmer)
   a) Log enthält „Commit“: Verhalten gemäß Recovery-Verfahren (**redo**)
   b) Log enthält „Abort“: Verhalten gemäß Recovery-Verfahren (**undo**)
   c) Log enthält „Yes“, aber weder „Abort“ noch „Commit“: Terminierungsprotokoll starten
   d) Alle anderen Fälle: „Abort“ entscheiden, „Abort“ ins Log schreiben und wie b) verfahren
2. Log enthält „Start 2PC“-Eintrag (➡ Knoten ist Koordinator)
   a) Log enthält „Commit“: Verhalten gemäß Recovery-Verfahren (**redo**)
   b) Log enthält „Abort“: Verhalten gemäß Recovery-Verfahren (**undo**)
   c) Alle anderen Fälle: „Abort“ entscheiden, „Abort“ ins Log schreiben und wie b) verfahren

Eigenschaften des 2PC-Protokolls

- **Fehlertoleranz**:
  - Toleriert Knotenausfälle und Kommunikationsfehler (durch Fristablauf)
- **Blockierung**:
  - 2PC blockiert, wenn bei einem Teilnehmer ein Fristablauf während seiner Ungewissheitsperiode entsteht und nur Knoten erreicht werden können, die sich ebenfalls in Ungewissheit befinden
- **Zeitkomplexität**:
  - Das 2PC-Protokoll benötigt drei Runden
  - Im Fehlerfall kommen zwei Runden für das Terminierungsprotokoll hinzu
- **Nachrichtenkomplexität**:
  - Bei n Teilnehmern und 1 Koordinator: $3n$ Nachrichten
  - Im Fehlerfall weitere $\mathcal{O}(n^2)$ Nachrichten (worst case)

Diskussion

- Es gibt ein **Drei-Phasen-Commit-Protokoll** (3PC) als nichtblockierende Verbesserung des 2PC-Protokolls
- **Idee:**
  - Blockieren im 2PC resultiert aus einer Situation, in der alle nicht ausgefallenen Knoten ungewiss sind und nicht einseitig „Abort“ entscheiden können, weil einige Knoten vielleicht schon „Commit“ entschieden haben
  - 3PC verhindert diese Situation, indem es sicherstellt, dass kein Knoten „Commit“ entschieden haben kann, solange noch ein nicht ausgefallener Knoten ungewiss ist

12.4 Replikation

Grundlagen

- In verteilten (Transaktions-)Systemen können Zugriffe auf Daten
  - zu lange Zeit in Anspruch nehmen, weil sie von weit her geholt werden müssen
  - kurzfristig nicht möglich sein, weil sie exklusiv gesperrt sind
  - kurzfristig nicht möglich sein, weil Kommunikationsfehler auftreten
  - längerfristig nicht möglich sein, weil der Knoten, auf dem die Daten liegen, ausgefallen ist
- Dagegen kann Abhilfe geschaffen werden, wenn man die Daten in mehreren Kopien auf unterschiedlichen Rechnern bereitstellt ➡ **Replikation**
- Betrachten Replikation von Daten im Kontext der Transaktionen, Problem ist aber allgemein

Anforderungen

- Anforderung: **Replikationstransparenz**
  - Dem Benutzer sollte die Existenz von Replikaten verborgen bleiben
  - Er sollte seine übliche Sicht auf die Daten haben
  - Bezüglich Korrektheit (ACID-Eigenschaften) sollte sich das Transaktionssystem verhalten wie im replikationsfreien Fall

Annahmen

- Ein Datum hat einen (ggf. veränderbaren) „Heimatort“ und Repliken
- Replikationstransparenz wird vom Transaktionsmanager gegenüber Scheduler (und Nutzer etc) hergestellt
- Scheduler behandelt Zugriffe ohne Wissen um weitere Repliken
- Bei Ausfällen können einige Kopien nicht verfügbar sein

Schreiben aller verfügbarer Kopien

- **Write-All-Algorithmus**
  - Leseoperationen können auf jeder beliebigen Kopie durchgeführt werden ➡ diejenige, die zu den geringsten Kosten erreichbar ist
  - Schreiboperationen müssen auf **allen** Kopien durchgeführt werden
  - Ist ein Knoten A ausgefallen, so kann eine Schreiboperation nicht durchgeführt werden ➡ die Operation wird verzögert
  - ➡ I.d.R. nicht tolerabel
- Abhilfe: **Write-All-Available-Algorithmus**
  - Schreiboperationen werden nur bei aktuell erreichbaren Kopien durchgeführt
  - Führt zu neuen Problemen

Schreiben aller verfügbarer Kopien (Forts.)

- Wegen der Unerreichbarkeit einiger Kopien können globale Konflikte zwischen Operationen gelegentlich unerkannt bleiben

➡ Zusätzliche Validierung nötig

- Ein Teil der für die Validierung notwendigen Kommunikation kann einfach in das Commit-Protokoll integriert werden
- Write-all-Available-Algorithmus ermöglicht „billige“ Leseoperationen auf Kosten „teuer“ Schreiboperation

Primärkopie-Algorithmus

- **Idee**: nur Änderung einer Kopie, der **Primärkopie** (*primary copy* oder *master copy*)
  ➡ reduziert Kosten für Schreiboperation
- Schreiben der anderen Kopien erst nach der Bestätigung der Transaktion
- In Kombination mit dem Zwei-Phasen-Sperren gibt es zwei Alternativen:
  a) Schreiboperation sperrt alle (verfügbare) Kopien
     - Gewährleistet konsistente Sicht
     - Leseoperation werden verzögert, da sie auf Freigabe der Sperren warten müssen.
  b) Schreiboperation sperrt lediglich Primärkopie
     - Lesetransaktionen, die nicht unbedingt die jüngste Version benötigen, können irgendeine Kopie lesen
     - Lesetransaktionen, die die jüngste Version brauchen, müssen eine Lesesperre auf die Primärkopie anfordern

Quorum-Algorithmus

- Der **Quorum-Algorithmus** liegt „zwischen“ dem Primary-Copy-Verfahren und Write-all-Available-Verfahren
- **Idee**: Es genügt, eine qualifizierte Mehrheit (**Quorum**) der Kopien zu aktualisieren
  - Jede Kopie x_A von x erhält ein nichtnegatives Gewicht (Anzahl der Stimmen)
  - Eine Leseschwelle RT (*read threshold*) und eine Schreibschwelle WT (*write threshold*) sind so definiert, dass gilt:

$$WT > \frac{TW}{2} \quad (1)$$

$$RT + WT > TW \quad (2)$$

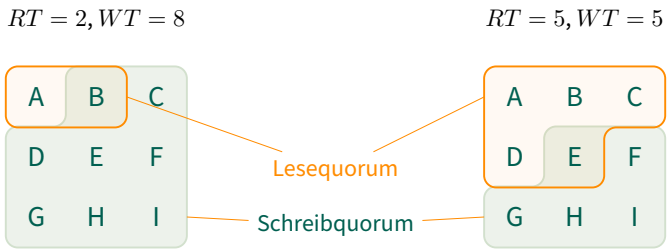
  - Dabei ist TW die Summe aller Gewichte (total weight), die den Kopien von x zugewiesen wurden

Quorum-Algorithmus (Forts.)

- Eine Menge von Kopien von x heißt **Lese-Quorum** (bzw. **Schreib-Quorum**), wenn ihr akkumuliertes Gewicht größer als die Leseschwelle (bzw. Schreibschwelle) ist
- Der Transaktionsmanager erzeugt Leseoperationen/Schreiboperationen) für jede Kopie eines Lese-Quorums (bzw. Schreib-Quorums
  - Schwellenbedingung (1): nicht mehr als eine Schreiboperation zu einem Zeitpunkt möglich
  - Schwellenbedingung (2): kein nebenläufiges gleichzeitiges Lesen und Schreiben **und** jedes Lese-Quorum enthält mindestens einen aktuellen Wert

Beispiele für Quoren

- Jedes der neun Elemente habe eine Stimme



Versionssteuerung bei Quoren

$RT = 5,\ WT = 5$

Zahlen geben Versionsnummern an

12.5 Skalierung
Einfluss Fehler und Latenz

- Seit den 90er Jahren exponentiell steigende Datenmengen
- Skalierbare globale Konsistenzversprechen wurden immer schwieriger
- Ausfälle und große Latenzen können nicht mehr als „Ausnahme“ angesehen werden
- Reale Beispiele für Netzwerkausfälle[2]
  - Microsoft Datacenter: 40.8 defekte Links/Tag für ca. 5 Minuten
  - HP: 11.4% der Tickets sind Netzwerkprobleme, im Schnitt 3h
  - Google: Cluster hat im ersten Jahr typischerweise 3 Router-Ausfälle
  - Blockierendes I/O in Kombination mit *garbage collection*
  - Fehlerhafte Netzwerktreiber
  - Fehlerhafte Firmware in Routern

[2] https://aphyr.com/posts/288-the-network-is-reliable

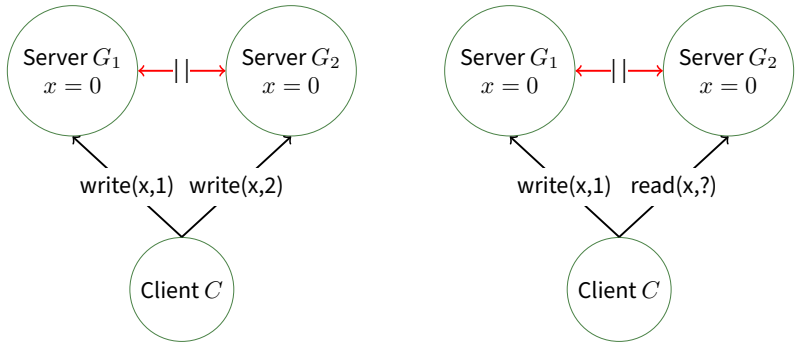
CAP-Theorem

- Die Schwierigkeit einer globalen Konsistenz führt zu weiteren Abschwächungen ➡ Aufgabe ACID-Forderungen (NoSQL-Datenbanken)
- Teilweise unterschiedliche (und manchmal nicht sehr exakte) Definitionen
- Was ist überhaupt möglich?
- Prof. Eric A. Brewer, 2001:

Definition 1 (CAP Theorem)

Man kann höchstens zwei dieser Eigenschaften in einem System mit verteilten Daten haben:
- Strikte/sequentielle Konsistenz - *(C)onsistency*
- Perfekte Verfügbarkeit - *(A)vailability*
- Toleranz gegenüber beliebiger Partitionierung - *(P)artition tolerance*

Szenarien im CAP-Theorem



Netzwerkpartitionierung

- TCP verspricht doch verlässliche Übertragung?
  - Lange zeitliche Verzögerungen sind trotzdem möglich
  - Halteproblem ➡ Echte Verzögerungen von Ausfällen nicht zu unterscheiden
  - Middleware muss das Problem also berücksichtigen (Ende-zu-Ende Prinzip)

PACELC Theorem

- CAP-Theorem vielfach missverstanden - Erweiterung zu PACCELC durch Abadi
- Genauere Betrachtung der Fehlerfälle bei Partitionierung
- Dient der Klassifikation von Systemen / Algorithmen
- PA/EL: Bei Partitionierung (P) hat die Verfügbarkeit (A) Vorrang, sonst (E) hat Latenz (L) Vorrang
  ➡ Normalerweise schnell, im Fehlerfall verfügbar, Konsistenz immer zuletzt
- PC/EC: Mit (P) und ohne (E) Partitionierung hat die Konsistenz (C) immer Vorrang
  ➡ Konsistenz auf Kosten der Latenz und Verfügbarkeit (klassische Datenbanken)
- PA/EC: Bei Partitionierung (P) hat die Verfügbarkeit (A) Vorrang, sonst (E) ist Konsistenz wichtiger
  ➡ System liefert nur im Fehlerfall inkonsistente Daten
- PC/EL: Bei Partitionierung (P) hat die Konsistenz (C) Vorrang, sonst Verfügbarkeit
  ➡ Normalerweise schnell, im Fehlerfall konsistent

Clienten-bezogene Konsistenz

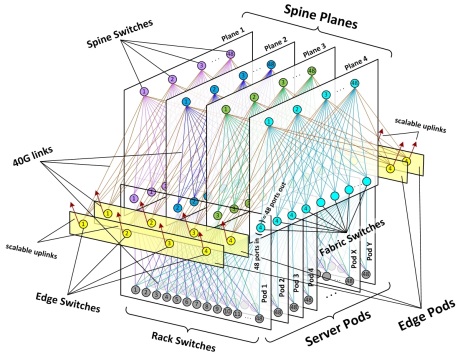
- In vielen Anwendungen mit hohem Datenaufkommen sind die Transaktionsgarantien nicht notwendig
- Konsistenz und Verfügbarkeit bezieht sich nur noch auf Klient-Knoten-Beziehung
  - Knoten ist hier der Server des verteilten Datenbanksystems, mit dem der Klient „spricht“

Werner Vogels, Amazon CTO

„Each node in a system should be able to make decisions purely based on local state. If you need to do something under high load with failures occurring and you need to reach agreement, you’re lost. If you’re concerned about scalability, any algorithm that forces you to run agreement will eventually become your bottleneck. Take that as a given.“

Beispiel: Facebook

- 2016: 1.13 Milliarden aktive Nutzer pro Tag
- Hunderttausende von Servern
- Facebook Fabric
  - Rack: Hunderte von Servern mit *top of rack (TOR)* Switch
  - Server Pod: 48 TOR Switches redundant verbunden mit 4 Fabric-Switches
  - Edge Pod: Verbindung zur Außenwelt



Klienten-bezogene Konsistenz-Modelle

- Mögliche Klienten-bezogene Konsistenz-Anforderungen:
  - **Monotones Lesen**: Wenn x zum Zeitpunkt t gelesen wurde, wird bei jedem $t' > t$ kein älterer Wert als x gelesen.
  - **Monotones Schreiben**: Wenn x zum Zeitpunkt t geschrieben wird, sind alle Schreibvorgänge zu $t' < t$ bereits vollzogen.
  - ***Read-your-writes (RYW)***: Wenn x zum Zeitpunkt t geschrieben wurde, wird ohne weiteren Schreibvorgang zu jedem $t' > t$ der Wert x gelesen.
  - ...
- Die Gesamtsicht (Systemsicht) wird dann mitunter als **letztendliche Konsistenz** definiert:

Definition 2 (Letztendliche Konsistenz)

Ein System realisiert **letztendliche Konsistenz** (*eventual consistency*), wenn es funktioniert und nach einer Zeitspanne hinreichend langen Zeitspanne ohne Schreiboperationen alle Leseoperationen mit (*einer gegebenen Wahrscheinlichkeit*) den gleichen Zustand ergeben.

Beispiel: Schwache Quoren

- Abschwächung der Konsistenz erlaubt z. T. Generalisierungen
- Z. B. lässt sich die Idee der Quoren (siehe Folie 37ff) verallgemeinern:
- Strenge Konsistenz für CP-System: $R + W > N$
  - Nicht-leere Schnittmenge von lesenden und schreibenden Knoten
  - Beispiele:
    - $W = N, R = 1$ ➡ Schnelles Lesen, langsames Schreiben
    - $W = 1, R = N$ ➡ Langsames Lesen, schnelles Schreiben
- Schwächere Konsistenz für AP-System: $R + W \leq N$
  - Beispiele:
    - $N = 3, W = 2, R = 1$ ➡ Lesen alter Werte möglich
    - $N = 3, W = 1, R = 2$ ➡ Lesen alter Werte und Fehler möglich

Literatur

[Sin97] Sinha , P. K. : *Distributed Operating Systems*. IEEE Press, 1997 , Abschnitt 6.5

[ChMi83] Chandy , K. M. und J. Misra: „Distributed Deadlock Detection“. *ACM Transactions on Computer Systems*, 1(2)1983, 144–156

[GrRe93] Gray , J. und A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1993, Abschnitt 10.4

[BaHa03] Bacon , J. und T. Harris: *Operating Systems*. Addison Wesley, 2003, Abschnitt 22.8