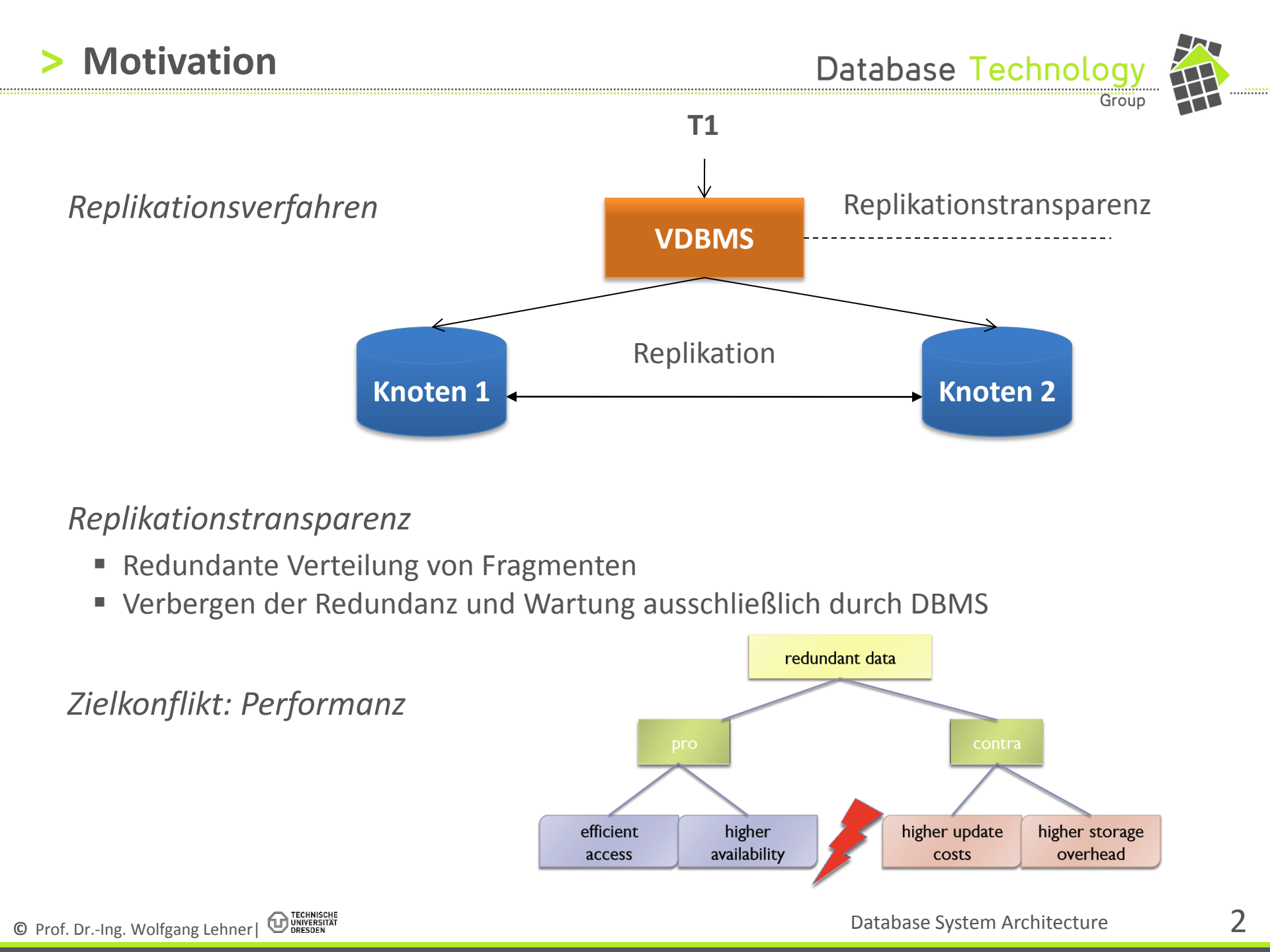
8 Replikation



Replikationsverfahren

Replikationstransparenz

- Redundante Verteilung von Fragmenten
- Verbergen der Redundanz und Wartung ausschließlich durch DBMS

Zielkonflikt: Performanz

- Motivation
- Vorbetrachtung
  - Anwendungsmöglichkeiten
  - Vor- und Nachteile
  - Zielkonflikte
  - Synchronisation
- Aktualisierungsstrategien
  - Klassifikation der Aktualisierungsstrategien
  - ROWA / ROWAA
  - Primary Copy
  - Votierungsverfahren
- Asynchrone Aktualisierungsstrategien
  - Klassifikation synchroner und asynchroner Verfahren
  - Need-to-Know-Prinzip
  - Snapshot Refresh
  - Lazy Update

Verteilte und Parallele DBS

- Replizierte DB-Tabellen / -Fragmente
- Replizierte Katalogdaten (Metadaten)
- Massiv-verteilte Key-Value Stores

Web

- Replizierte Daten und Metadaten für schnelles Lesen und Verfügbarkeit (Mirror Sites, Suchmaschinen, Portale)
- Katastrophen-Recovery

Data Warehousing

- Übernahme transformierter Daten in eigene Datenbank für Entscheidungsunterstützung

Mobile Computing

- Datenbank-Ausschnitte, ... auf Notebook, mobilen Endgeräten etc.

Vorteile

- **Effizienter lesender Zugriff**
- **Erhöhte Verfügbarkeit**
  (bessere Antwortzeiten, Kommunikationseinsparungen)
- Erhöhte Möglichkeit zur Lastbalancierung/Virtualisierung
- Enge Kopplung von Quellsystemen an Data-Warehouse-System

Nachteile

- **Erhöhter Änderungsaufwand**
- **Erhöhter Speicherplatzbedarf**
- Erhöhte Systemkomplexität
  - Update-Algorithmus
  - Erweiterte Synchronisationsanforderungen (1-copy-serializability)
  - Recovery-Probleme v.a. bei Netzwerk-Partitionierungen
  - Query-Optimierung

Erhaltung der Datenkonsistenz

- Kopien wechselseitig konsistent halten: 1-Kopien-Äquivalenz
- Kleine Kopienzahl

Erhöhung der Verfügbarkeit, effizienter Lesezugriff

- Große Kopienzahl
- Zugriff auf beliebige und möglichst wenige Kopien

Minimierung des Änderungsaufwands

- Kleine Kopienzahl
- Möglichst wenige Kopien synchron aktualisieren

One-Copy-Serialisierbarkeit (1SR im strengen Sinne: kein Lesen veralteter Replikate)

- Eine nebenläufige Ausführung von verteilten Transaktionen ist one-copy-serialisierbar, wenn sie mit einer seriellen Ausführung auf einer nicht replizierten Datenbank äquivalent ist. Folgerung: Eine replizierte Datenbank verhält sich wie eine nicht replizierte Datenbank.

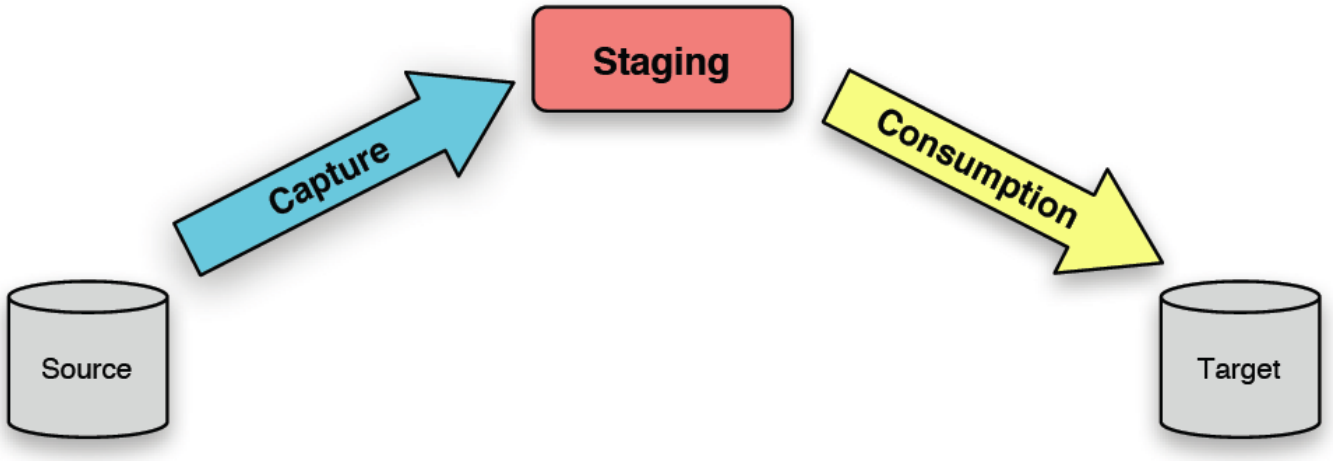
1SR (Lesen veralteter Replikate zulässig)

- z.B. Mehrversionen-Serialisierbarkeit: Jede Schreiboperation auf einem Datenobjekt erzeugt eine neue Version
- DBMS verwaltet die Versionen der Objekte und sorgt für eine konsistente Sicht auf die Datenbank. Es wird vermerkt, auf welche Version eines Objektes zugegriffen wurde. Lesezugriff auf veraltete Versionen ist erlaubt. Das DBMS sorgt für konsistente Verarbeitung, ggf. müssen Transaktionen zurückgesetzt werden.

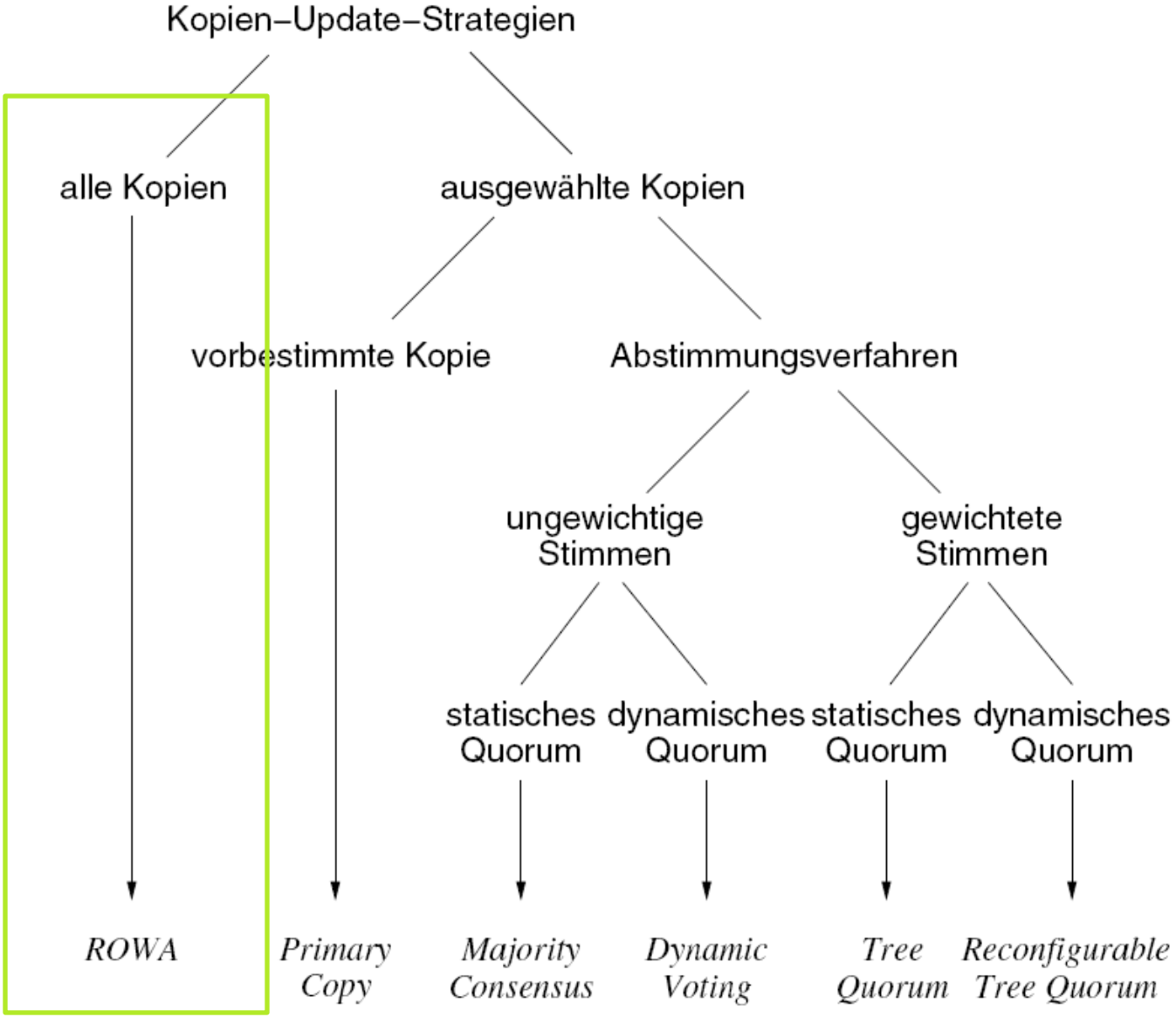
Transaktionale Konsistenz meist zu starke Forderung

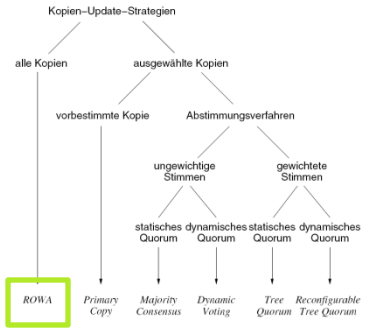
Daher „nur“ Ziel der Konvergenz

- Alle Kopien konvergieren zum selben konsistenten Stand der Informationen (schwächste Form)
- Nach einer gewissen änderungsfreien Zeit alle Kopien einen identischen Zustand annehmen → Capture-Staging-Consumption-Methode



Aktualisierungsstrategien


Kopien–Update–Strategien
alle Kopien
ausgewählte Kopien
vorbestimmte Kopie
Abstimmungsverfahren
ungewichtige Stimmen
gewichtete Stimmen
statisches Quorum
dynamisches Quorum
statisches Quorum
dynamisches Quorum
ROWA
Primary Copy
Majority Consensus
Dynamic Voting
Tree Quorum
Reconfigurable Tree Quorum

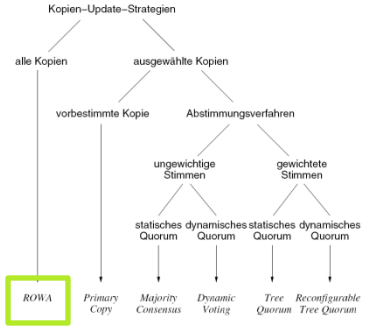


ROWA (Read-One/Write-All, ROWA)

- Write-All/Read-Any-Strategie

Konzept

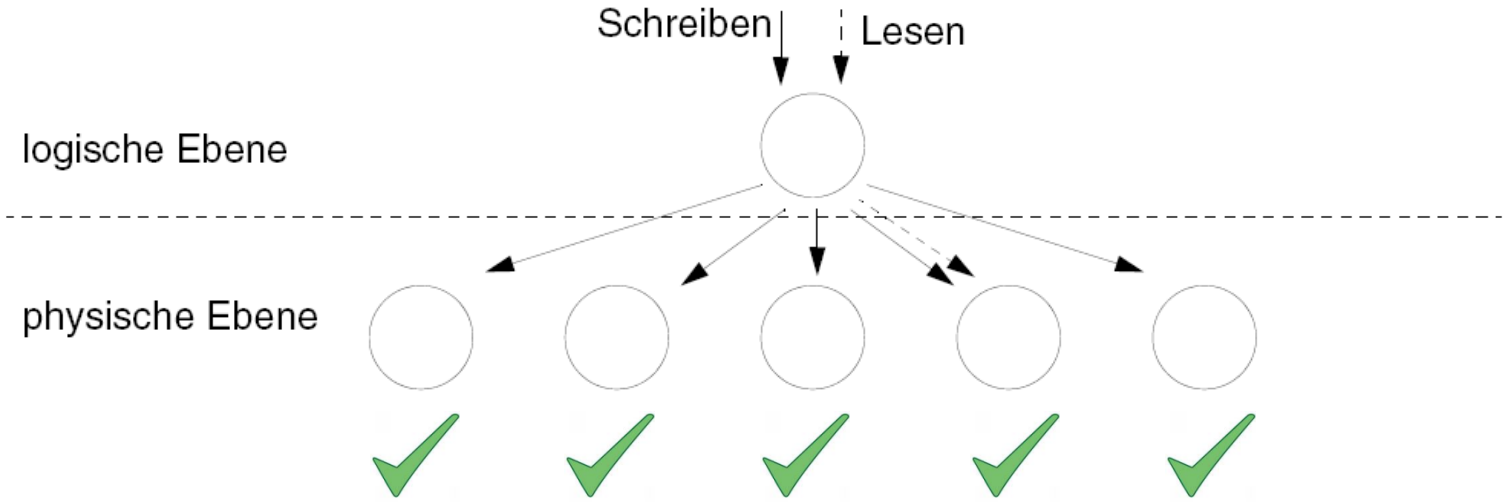
- Eine logische Schreiboperation → physische Schreiboperation auf allen Kopien
  - Änderungsoperationen synchron auf allen Replikaten (“write all”)
  - Alle Replikate sind immer aktuell
- Eine logische Leseoperation → eine physische Leseoperation auf einem beliebigen Replikat (“read one”)
- Bevorzugte Behandlung von Lesezugriffen → Erhöhte Verfügbarkeit für Leser
- Sehr hohe Kosten für Änderungstransaktionen
  - Bei Sperrverfahren: Schreibsperren bei allen Replikaten
  - Propagierung der Änderungen im Rahmen eines erweiterten Commit-Protokolls:
    1. Phase: Änderungen an alle Knoten übertragen
    2. Phase: Änderungen an allen Replikaten vornehmen
    → Aufwand steigt mit dem Replikationsgrad

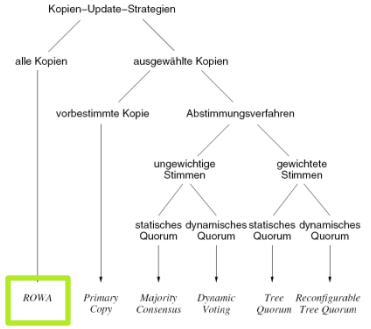


Abhängigkeit der Änderungstransaktion von Verfügbarkeit aller beteiligten Knoten

Verfügbarkeitsproblem

- Änderung nur erfolgreich, wenn alle Replikate erreichbar
- Beim Ausfall eines Knotens sind keine Änderungen mehr möglich
- Lesen ist möglich, solange noch mindestens ein Replikat erreichbar ist



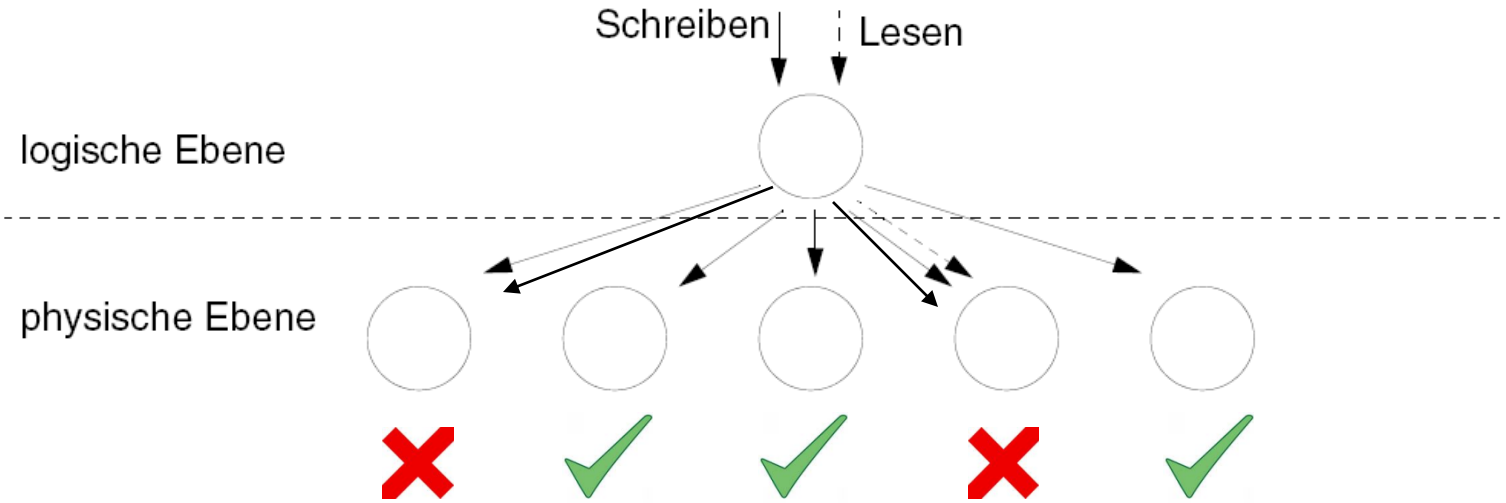


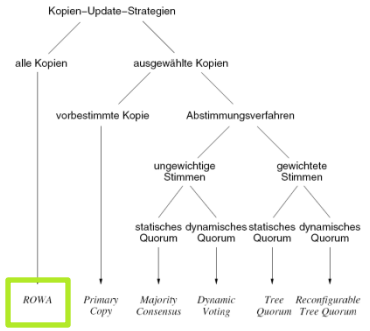
ROWAA (Write-All-Available-Variante)

Konzept

- Nur verfügbare Replikate werden geändert (Eine Änderung ist auch dann erfolgreich wenn ein Knoten nicht erreichbar ist)
- Für ausgefallene Rechner werden Änderungen protokolliert und bei Wiederanlauf nachgefahren.
- Hoher Aufwand für Änderungsoperationen bleibt
- Zusätzlicher Validierungsaufwand bei Knotenausfällen

Beispiel



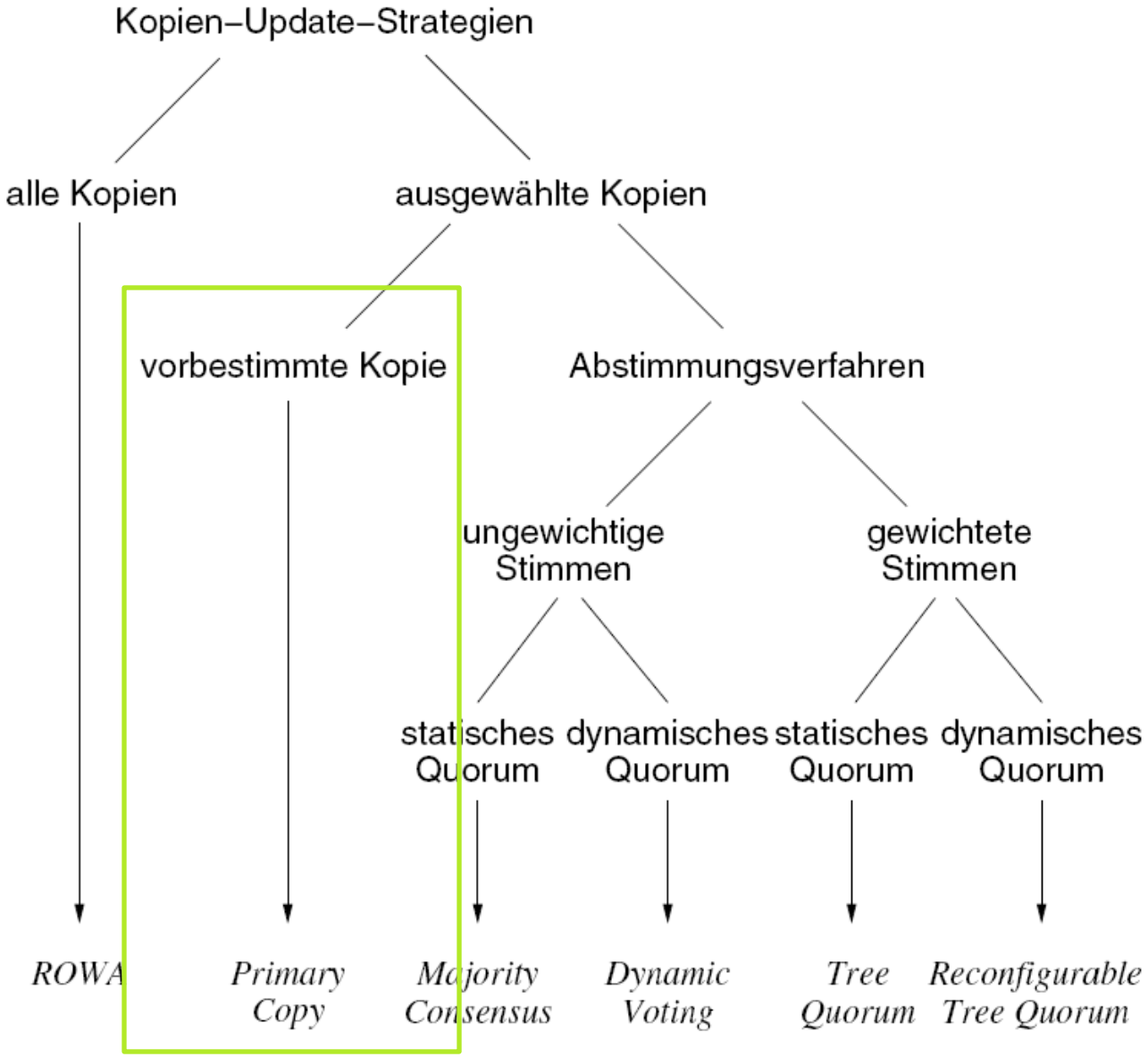


Zitat

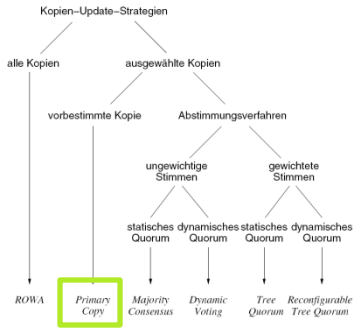
- „Update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up: **a ten-fold increase in nodes and traffic gives a thousand fold increase in deadlocks or reconciliations**. Master copy replication (primary copy) schemes reduce this problem.”

Literatur

- [**Jim Gray**, Pat Helland, Patrick E. O'Neil, Dennis Shasha: The Dangers of Replication and a Solution, SIGMOD 1996, pages 173-182.]

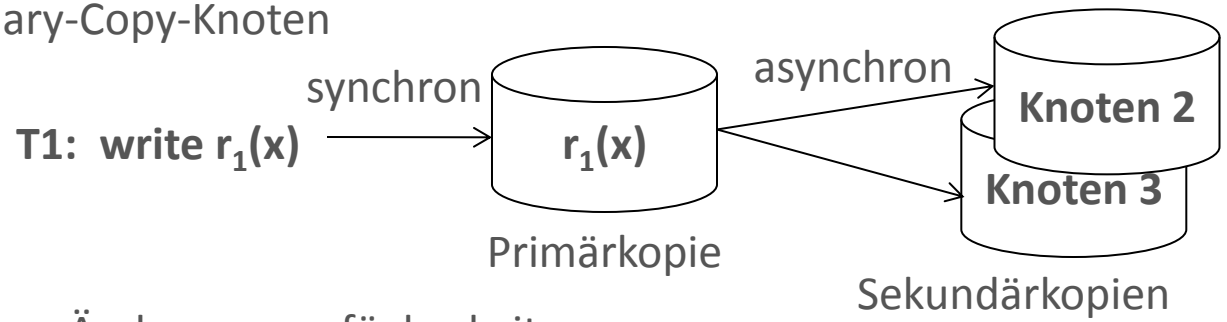

Kopien–Update–Strategien
alle Kopien
ausgewählte Kopien
vorbestimmte Kopie
Abstimmungsverfahren
ungewichtige Stimmen
gewichtete Stimmen
statisches Quorum
dynamisches Quorum
statisches Quorum
dynamisches Quorum
ROWA
Primary Copy
Majority Consensus
Dynamic Voting
Tree Quorum
Reconfigurable Tree Quorum

[Dadam, 1996]



Primary-Copy-Verfahren (Stonebraker, 1979)

- Effizientere Bearbeitung von Änderungen
- Alle Änderungen synchron nur an einer Kopie: Primärkopie
- Verzögerte/asynchrone Aktualisierung der anderen Kopien durch Primary-Copy-Knoten

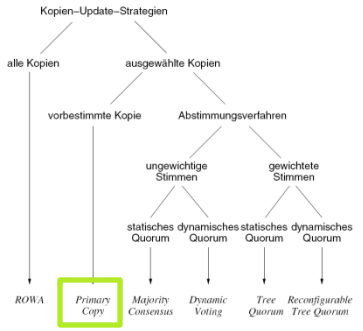


Vorteile

- Höhere Änderungsverfügbarkeit
- Einsparen von Nachrichten
  (Mehrere Aktualisierungen können gebündelt verschickt werden)
- Um Engpässe zu vermeiden können die Primärkopien verschiedener Objekte an unterschiedlichen Knoten abgelegt werden
  (Möglichst da, wo das Objekt am häufigsten benötigt wird)

Nachteil

- Lesetransaktion auf lokaler, möglicherweise veralteter Replikate



Realisierungsalternativen für Primary-Copy

***Sperren aller Replikate bei Änderung**, aber synchrones Aktualisieren nur auf der Primärkopie (sonst wie ROWA)*

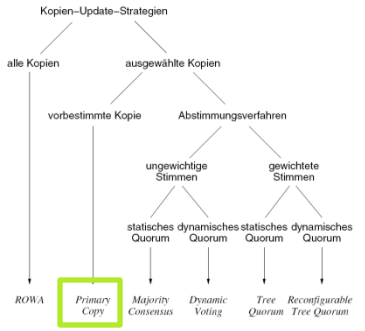
- Änderungen werden an allen Knoten zumindest bemerkt
- Lesen eines beliebigen Replikates möglich, sofern alle bemerkten Änderungen auch tatsächlich eingetroffen sind
- Geringe Änderungsverfügbarkeit - ähnlich ROWA

Sperren der Primärkopie beim Lesen

- Primärkopie lesen (Replikation wird nicht genutzt!)
- Sperren beim Primary-Copy-Rechner, Zugriff auf lokale Kopie
  → Es muss überprüft werden, ob die zu lesende Kopie alle Änderungen erhalten hat

Lesen ohne Synchronisation

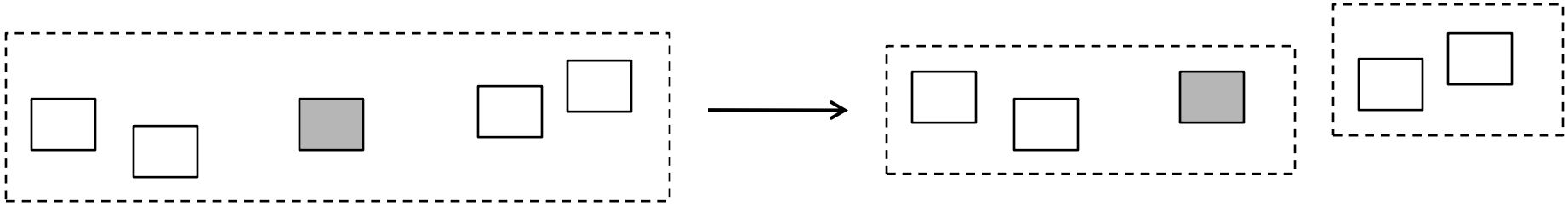
- Lokale Kopie lesen ohne Sperre beim Primary-Copy-Rechner
- Aktualität des gelesenen Replikats nicht gewährleistet
- Inkonsistente Sicht auf die Datenbank möglich!
- 1SR ist nicht gewährleistet, ABER: Konvergenz gewährleistet!



Fehlerszenarien

Netzwerk-Partitionierung

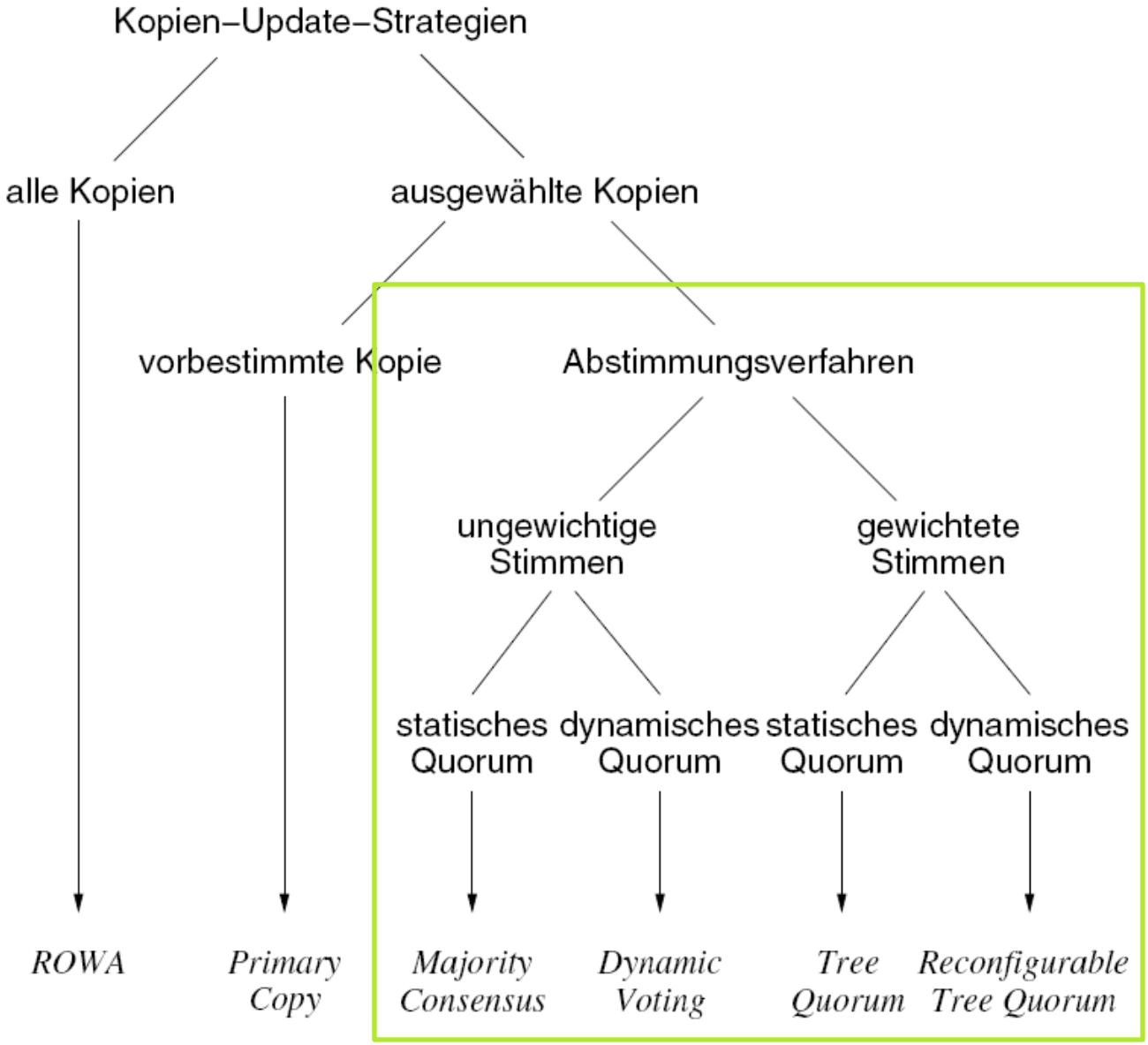
- Nur in Partition mit Primärkopie können Änderungen erfolgen
- Ist die Primärkopie in einer Partition nicht erreichbar:
  Lesezugriffe nur mit dem Risiko eventuell veraltete Daten zu lesen und inkonsistente Sicht auf die Datenbank in Kauf zu nehmen

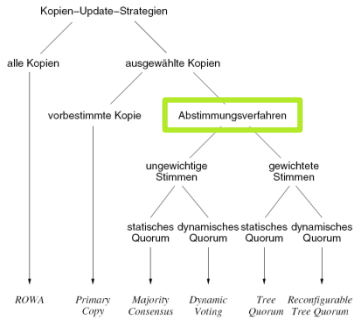


Fehlerszenarien

Knotenausfall

- Statischer Ansatz
  - keine weiteren Schreibzugriffe möglich bis Primary-Copy-Rechner wieder aktiv
- Dynamische Ansatz
  - Bestimmung eines neuen Primary-Copy-Rechners
  - Bei Netzwerk-Partitionierungen darf höchstens in einer Partition Verarbeitung fortgesetzt werden (z.B. wo Mehrzahl der Kopien vorliegt)
  - Neuer PC-Rechner muss ggf. zunächst seine Kopie auf den neuesten Stand bringen („pending updates“)


Kopien–Update–Strategien
alle Kopien
ausgewählte Kopien
vorbestimmte Kopie
Abstimmungsverfahren
ungewichtige Stimmen
gewichtete Stimmen
statisches Quorum
dynamisches Quorum
statisches Quorum
dynamisches Quorum
ROWA
Primary Copy
Majority Consensus
Dynamic Voting
Tree Quorum
Reconfigurable Tree Quorum



Idee

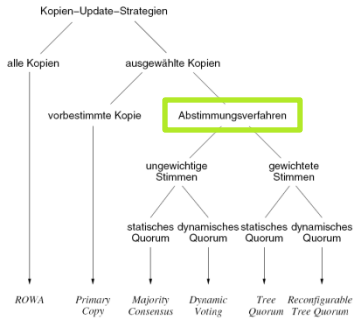
- Vor einem Zugriff wird abgestimmt, ob dieser Zugriff zulässig ist oder nicht.
- Jedes Replikat hat eine Stimme

Zugriffkontrolle

- Festlegung der erforderliche Stimmen für einen Zugriff
- Für Lesezugriffe (Lesequorum; Q_L)
- Für Schreibzugriffe (Schreibquorum; Q_U)

Quorum-Überschneidungsregel

- Sei Q die Gesamtstimmenzahl, dann muss stets die folgende Quorum-Überschneidungsregel beachtet werden, um die 1-Kopie-Serialisierbarkeit zu gewährleisten:
- $Q_L + Q_U > Q$
- $Q_U + Q_U > Q$

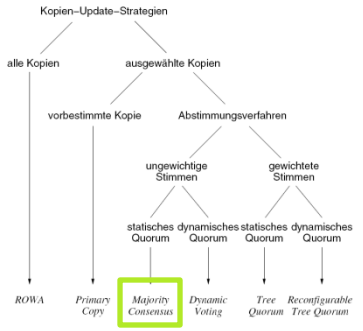


Qualität der Stimme

- Ungewichtete Stimmen: Alle Replikate werden gleich behandelt
  - Jedes Replikat zählt einfach
  - Anfragen in beliebiger Reihenfolge
- Gewichtete Stimmen: Replikate werden unterschiedliche behandelt
  - Kopien erhalten unterschiedliche Stimmgewichte
  - Anfragen in festgelegter Reihenfolge (absteigend nach Gewichten)
  - Reduktion des Kommunikationsaufwands

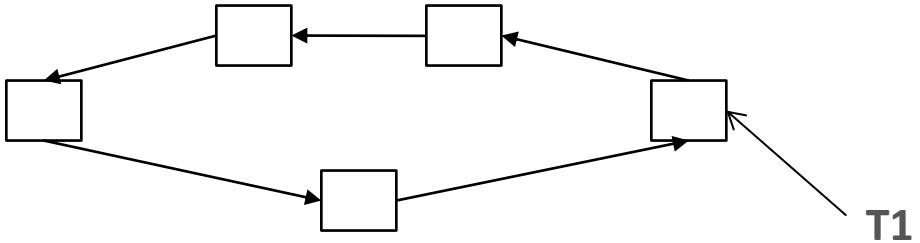
Quantität der Stimmen (Anzahl an Stimmen, die für das Erreichen der Mehrheit erforderlich sind)

- Statisches Quorum: bei Systemstart fest vorgegeben
- Dynamisches Quorum: dynamisch zur Laufzeit festgelegt
  - Quoren werden entsprechend der Verfügbarkeit von Knoten immer wieder neu angepasst



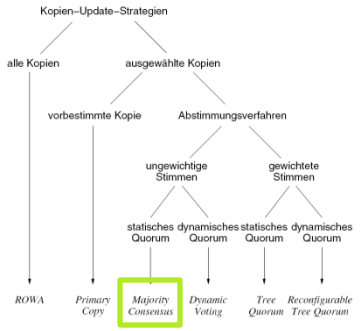
Konzept

- Lesen oder Schreiben eines Objektes verlangt Zugriff auf Mehrheit der Replikate
- Schreiben
  - Sperren und Ändern der Mehrheit der Replikate
- Lesen
  - Sperren der Mehrheit der Replikate
  - Lesen des aktuellsten Replikats von allen gesperrten Replikaten
- Logisch konfligierende Operationen, werden an mindestens einem Knoten erkannt!
- Synchronisation: Zeitstempelverfahren (Zeitstempel, Versionsnummer)
- Verwaltung als Ring
  - Frühzeitiges ABORT/COMMIT möglich



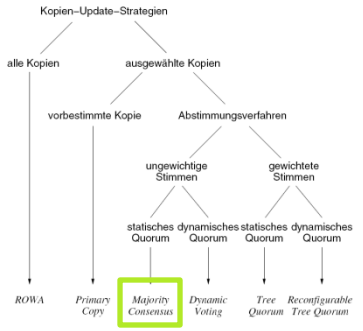
Fehlerfall

- Fortsetzung der Verarbeitung möglich, solange Mehrheit der Replikate noch erreichbar



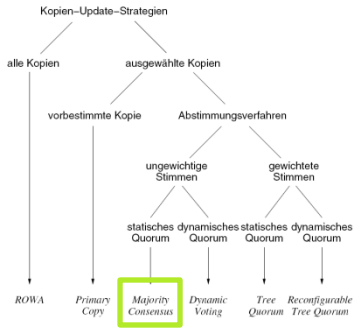
Vorgehensweise

- Jedes DB-Objekt ist mit einem Zeitstempel der letzten Änderung (bzw. der Erzeugung) versehen
- Jede Änderungs-Transaktion
  - Führt alle Änderungen zunächst rein lokal durch (macht sie aber noch nicht anderen Transaktionen sichtbar)
  - Erstellt eine Liste aller Ein- und Ausgabe-Objekte mit den jeweiligen Zeitstempeln
  - Schickt diese Liste zusammen mit ihrem Transaktions-Zeitstempel an alle anderen Knoten
  - Darf die Änderungen permanent machen, wenn die Mehrheit der Knoten (>n/2) zustimmt



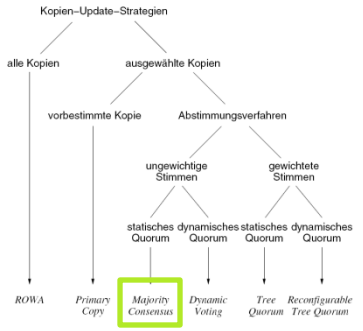
Jeder Knoten stimmt über eingehende Änderungs-Anträge wie folgt ab und reicht sein Votum zusammen mit den anderen Voten an den nächsten Knoten weiter

- Er stimmt mit **ABGELEHNT**, wenn einer der übermittelten Objekt-Zeitstempel veraltet ist
- Er stimmt mit **OK** und markiert den Auftrag als schwebend ("pending"), wenn alle übermittelten Objekt-Zeitstempel aktuell sind und der Antrag nicht in Konflikt mit einem anderen Antrag steht
- Er stimmt mit **PASSIERE**, wenn alle Objekt-Zeitstempel zwar aktuell sind, der Antrag mit einem anderen schwebenden Antrag mit höherer Priorität (d.h. mit aktuellerem Zeitstempel) in Konflikt steht
  - Falls durch das Votum "PASSIERE" keine Mehrheit mehr zustande kommen kann, stimmt er mit **ABGELEHNT**
- Er **verzögert seine Abstimmung** über den Antrag, wenn der Antrag in Konflikt mit einem Antrag niedrigerer Priorität in Konflikt steht oder wenn einer der übermittelten Objekt-Zeitstempel höher als der des korrespondierenden lokalen Objektes ist



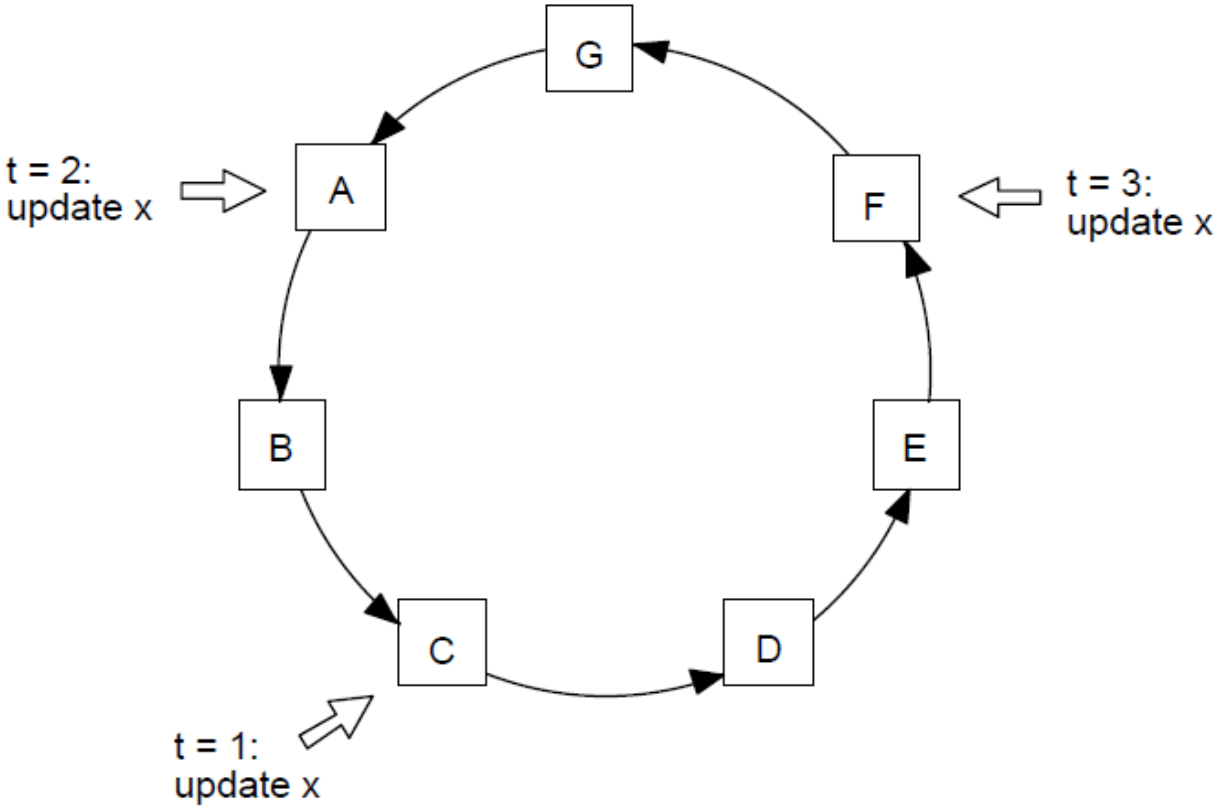
Annahme bzw. Ablehnung

- Der Knoten, dessen Zustimmung (**OK**) dem Antrag die Mehrheit verschafft, erzeugt die globale **COMMIT**-Meldung für die gewünschte Änderung
- Jeder Knoten, der mit **ABGELEHNT** stimmt, löst ein globales **ABORT** des Änderungs-Auftrages (d.h. der Transaktion) aus
- Wird eine Änderungs-Transaktion **akzeptiert**, werden die Änderungen bei Eintreffen eingebracht (Ausnahme siehe unten) und die Objekt-Zeitstempel entsprechend aktualisiert
- Wird eine Änderungs-Transaktion **abgelehnt**, so werden die "verzögerten Abstimmungen" je Knoten jeweils überprüft, ob jetzt eine Entscheidung möglich ist
- Bei Ablehnung muss die Transaktion komplett wiederholt werden, einschließlich Objekt-Lesen

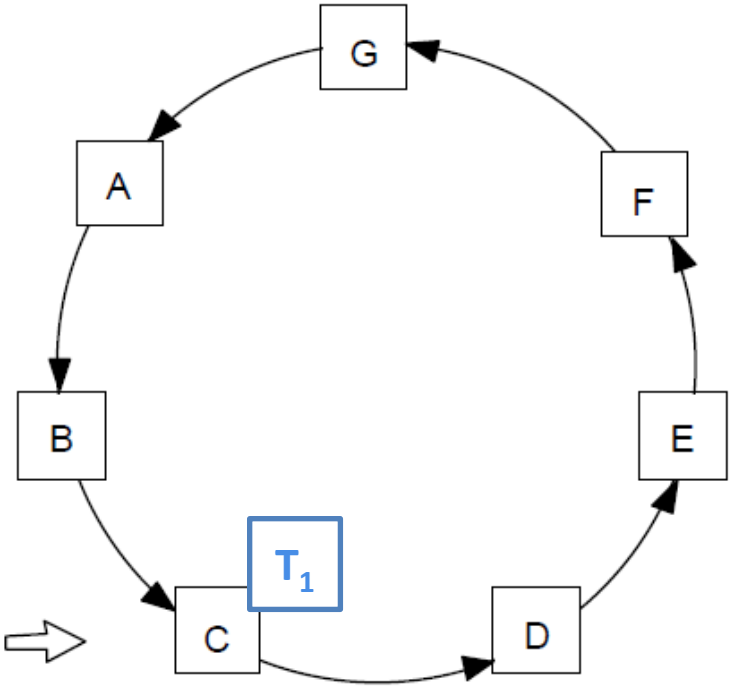


Beispiel

- n=7 → n/2=4

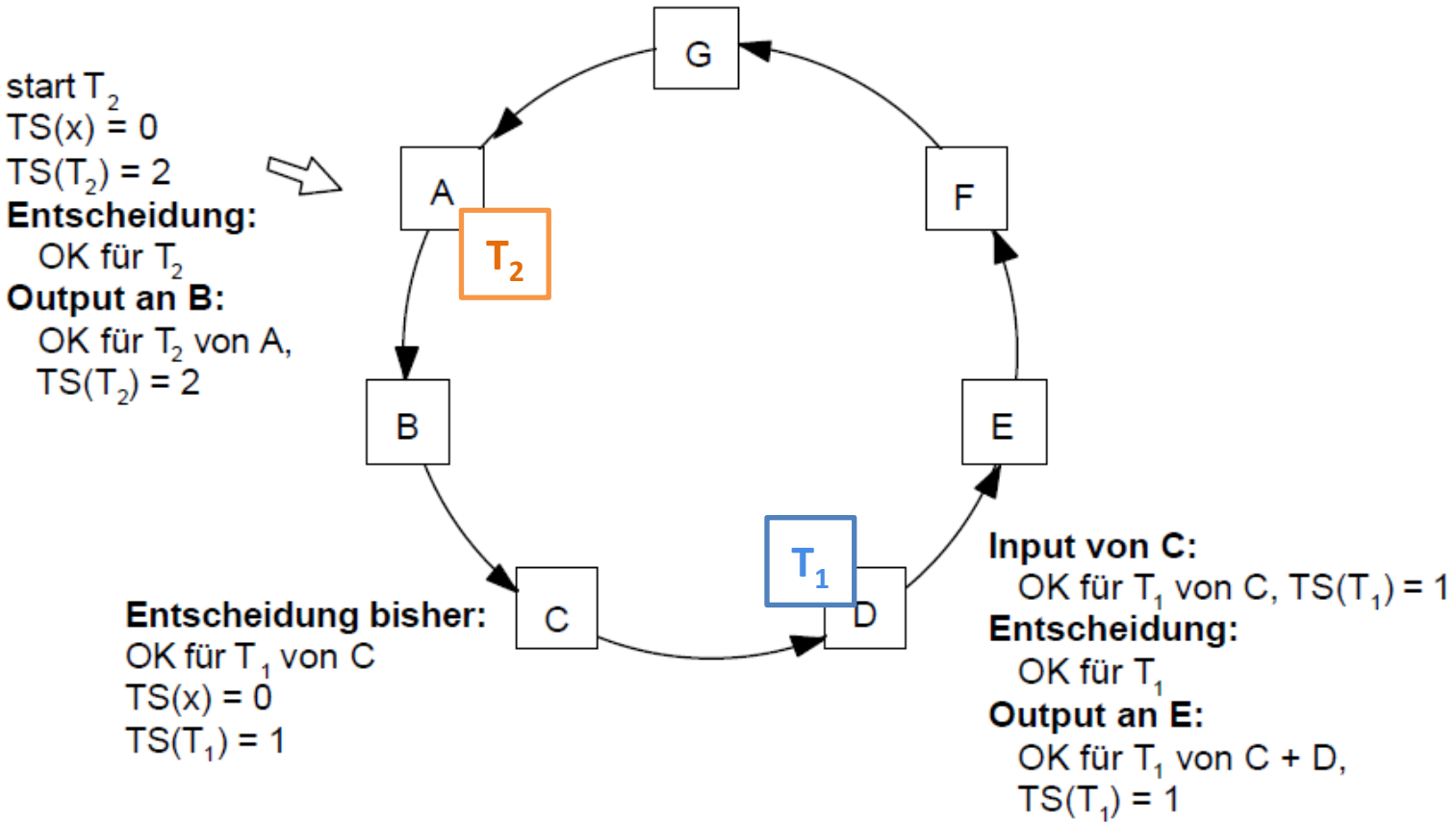


Beispiel (t=1)

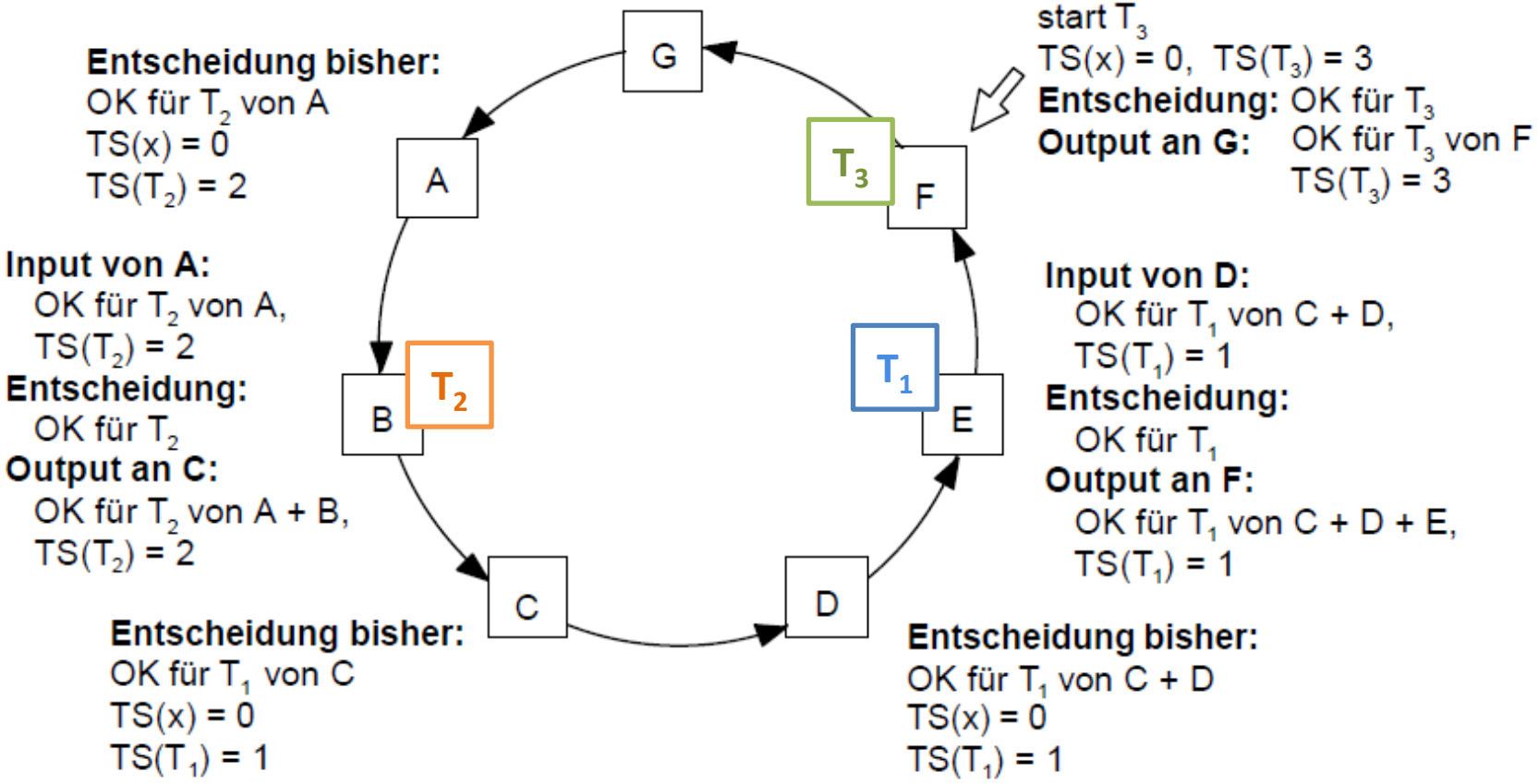


start T_1
$TS(x) = 0$
$TS(T_1) = 1$
Entscheidung: OK für T_1
Output an D: OK für T_1 von C, $TS(T_1) = 1$

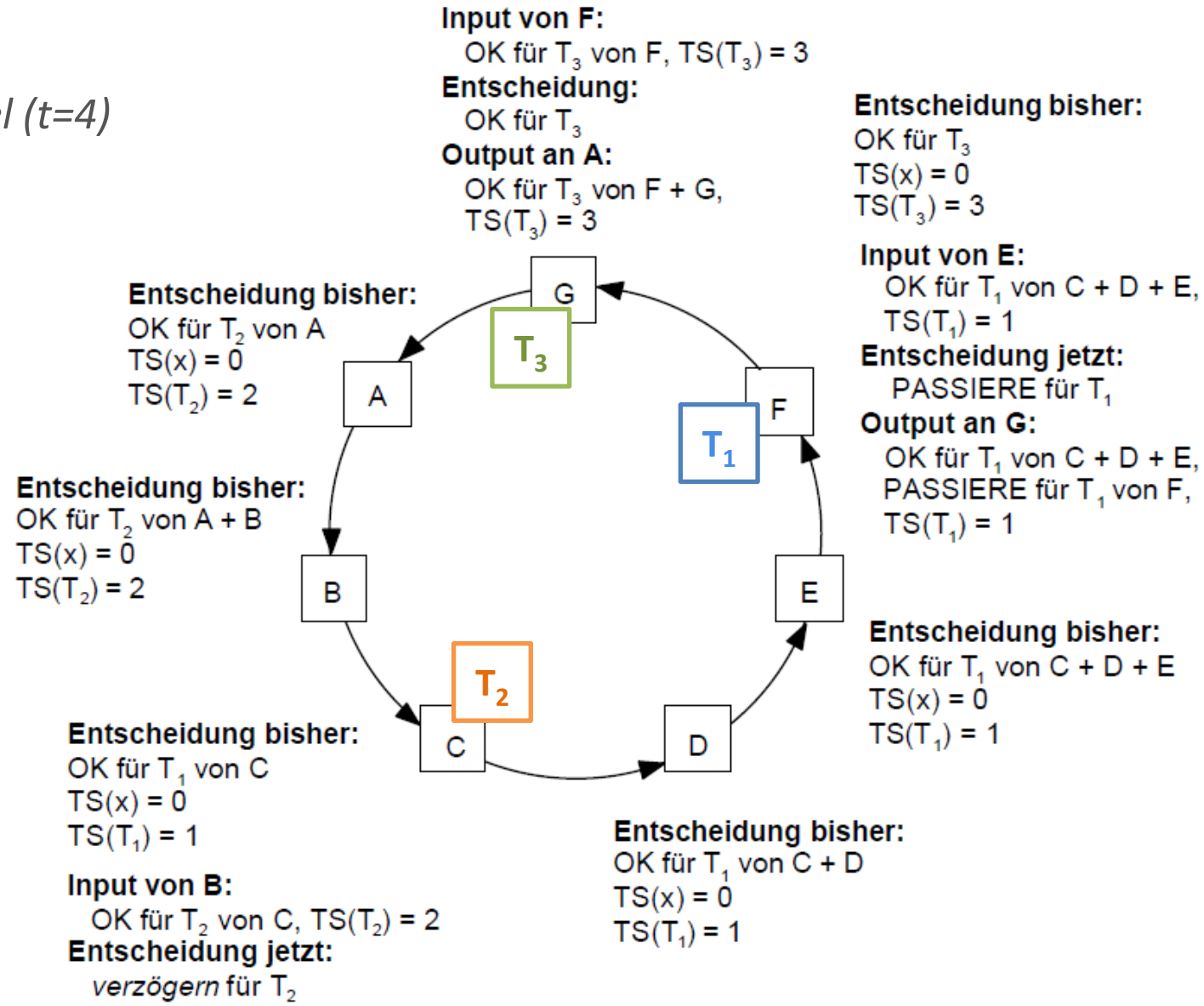
Beispiel (t=2)


start T_2
$TS(x) = 0$
$TS(T_2) = 2$
Entscheidung:
OK für T_2
Output an B:
OK für T_2 von A,
$TS(T_2) = 2$
G
A
F
T_2
B
E
T_1
C
D
Entscheidung bisher:
OK für T_1 von C
$TS(x) = 0$
$TS(T_1) = 1$
Input von C:
OK für T_1 von C, $TS(T_1) = 1$
Entscheidung:
OK für T_1
Output an E:
OK für T_1 von C + D,
$TS(T_1) = 1$

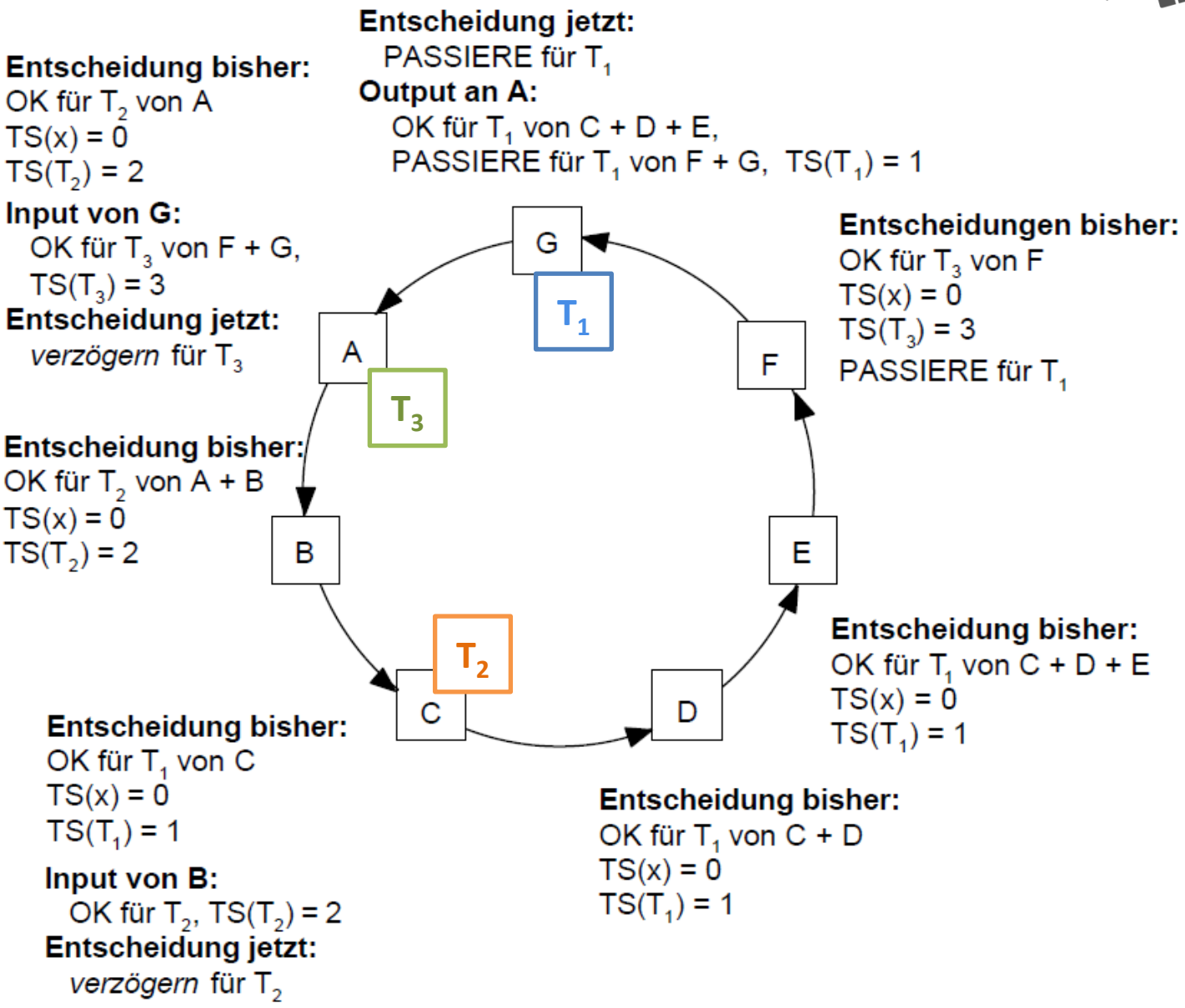
Beispiel (t=3)


G
A
F
B
E
C
D
T_3
T_2
T_1
start T_3
TS(x) = 0, TS(T_3) = 3
Entscheidung: OK für T_3
Output an G: OK für T_3 von F
TS(T_3) = 3
Entscheidung bisher:
OK für T_2 von A
TS(x) = 0
TS(T_2) = 2
Input von A:
OK für T_2 von A,
TS(T_2) = 2
Entscheidung:
OK für T_2
Output an C:
OK für T_2 von A + B,
TS(T_2) = 2
Input von D:
OK für T_1 von C + D,
TS(T_1) = 1
Entscheidung:
OK für T_1
Output an F:
OK für T_1 von C + D + E,
TS(T_1) = 1
Entscheidung bisher:
OK für T_1 von C
TS(x) = 0
TS(T_1) = 1
Entscheidung bisher:
OK für T_1 von C + D
TS(x) = 0
TS(T_1) = 1

Beispiel (t=4)


Input von F:
OK für T_3 von F, $TS(T_3) = 3$
Entscheidung:
OK für T_3
Output an A:
OK für T_3 von F + G,
$TS(T_3) = 3$
Entscheidung bisher:
OK für T_3
TS(x) = 0
$TS(T_3) = 3$
Input von E:
OK für T_1 von C + D + E,
$TS(T_1) = 1$
Entscheidung jetzt:
PASSIERE für T_1
Output an G:
OK für T_1 von C + D + E,
PASSIERE für T_1 von F,
$TS(T_1) = 1$
Entscheidung bisher:
OK für T_2 von A
TS(x) = 0
$TS(T_2) = 2$
Entscheidung bisher:
OK für T_2 von A + B
TS(x) = 0
$TS(T_2) = 2$
Entscheidung bisher:
OK für T_1 von C + D + E
TS(x) = 0
$TS(T_1) = 1$
Entscheidung bisher:
OK für T_1 von C
TS(x) = 0
$TS(T_1) = 1$
Input von B:
OK für T_2 von C, $TS(T_2) = 2$
Entscheidung jetzt:
verzögern für T_2
Entscheidung bisher:
OK für T_1 von C + D
TS(x) = 0
$TS(T_1) = 1$
G
A
F
B
E
C
D
T_3
T_1
T_2

Beispiel (t=5)

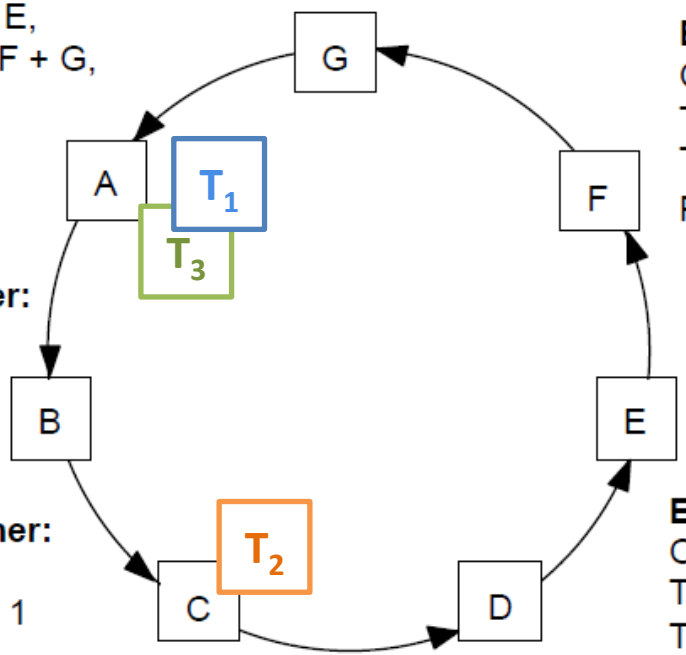

Entscheidung jetzt:
PASSIERE für T1
Output an A:
OK für T1 von C + D + E,
PASSIERE für T1 von F + G, TS(T1) = 1
Entscheidung bisher:
OK für T2 von A
TS(x) = 0
TS(T2) = 2
Input von G:
OK für T3 von F + G,
TS(T3) = 3
Entscheidung jetzt:
verzögern für T3
Entscheidungen bisher:
OK für T3 von F
TS(x) = 0
TS(T3) = 3
PASSIERE für T1
G
T1
A
T3
F
Entscheidung bisher:
OK für T2 von A + B
TS(x) = 0
TS(T2) = 2
B
E
T2
C
D
Entscheidung bisher:
OK für T1 von C + D + E
TS(x) = 0
TS(T1) = 1
Entscheidung bisher:
OK für T1 von C
TS(x) = 0
TS(T1) = 1
Input von B:
OK für T2, TS(T2) = 2
Entscheidung jetzt:
verzögern für T2
Entscheidung bisher:
OK für T1 von C + D
TS(x) = 0
TS(T1) = 1

Beispiel (t=6)

Entscheidung bisher:
OK für T_2 von A
verzögern für T_3
$TS(x) = 0$
$TS(T_2) = 2$
$TS(T_3) = 3$

Input von G:
OK für T_1 von C + D + E,
PASSIERE für T_1 von F + G,
$TS(T_1) = 1$
Entscheidung jetzt:
PASSIERE für T_1

Input von F:
OK für T_1 von C + D + E,
PASSIERE für T_1 von F, $TS(T_1) = 1$
Entscheidung:
PASSIERE für T_1
Output an A:
OK für T_1 von C+ D + E,
PASSIERE für T_1 von F + G, $TS(T_1) = 1$



Entscheidungen bisher:
OK für T_3 von F
$TS(x) = 0$
$TS(T_3) = 3$
PASSIERE für T_1

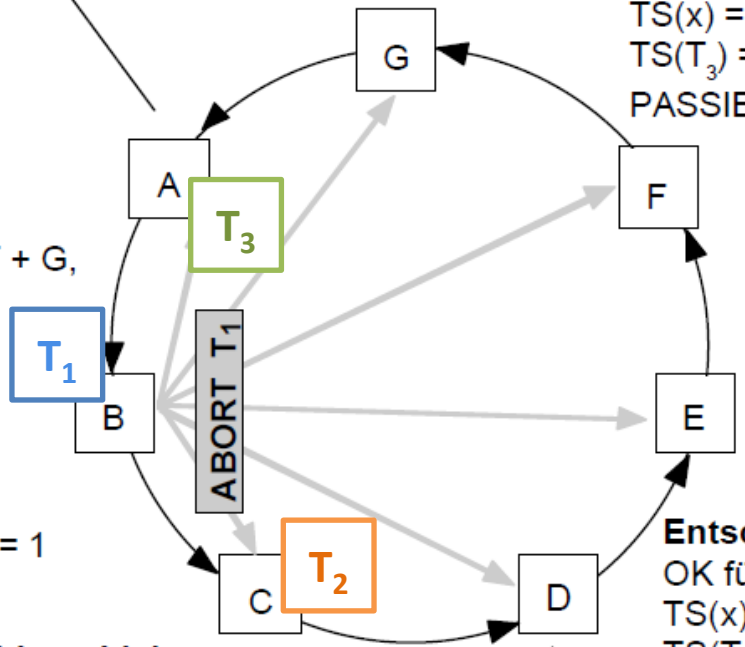
Entscheidung bisher:
OK für T_2 von A + B
$TS(x) = 0$
$TS(T_2) = 2$

Entscheidung bisher:
OK für T_1 von C + D + E
$TS(x) = 0$
$TS(T_1) = 1$

Entscheidung bisher:
OK für T_1 von C
$TS(x) = 0$, $TS(T_1) = 1$
Input von B:
OK für T_2, $TS(T_2) = 2$
Entscheidung jetzt:
verzögern für T_2

Entscheidung bisher:
OK für T_1 von C + D
$TS(x) = 0$, $TS(T_1) = 1$

Beispiel (t=7)



PASSIERE für T_1 von A + F + G
verzögern für T_3
TS(x) = 0, TS(T_2) = 2,
TS(T_3) = 3, TS(T_1) = 1

Entscheidung bisher:
OK für T_3 von F + G
OK für T_1 von C + D + E,
PASSIERE für T_1 von F + G,
TS(x) = 0,
TS(T_1) = 1
TS(T_3) = 3

Entscheidungen bisher:
OK für T_3 von F
TS(x) = 0
TS(T_3) = 3
PASSIERE für T_1

Entscheidung bisher:
OK für T_2 von A + B
TS(x) = 0
TS(T_2) = 2

Input von A:
OK für T_1 von C + D + E,
PASSIERE für T_1 von A + F + G,
TS(T_1) = 1

Entscheidung jetzt:
ABGLEHNT für T_1
(da mit PASSIERE keine Mehrheit mehr möglich)

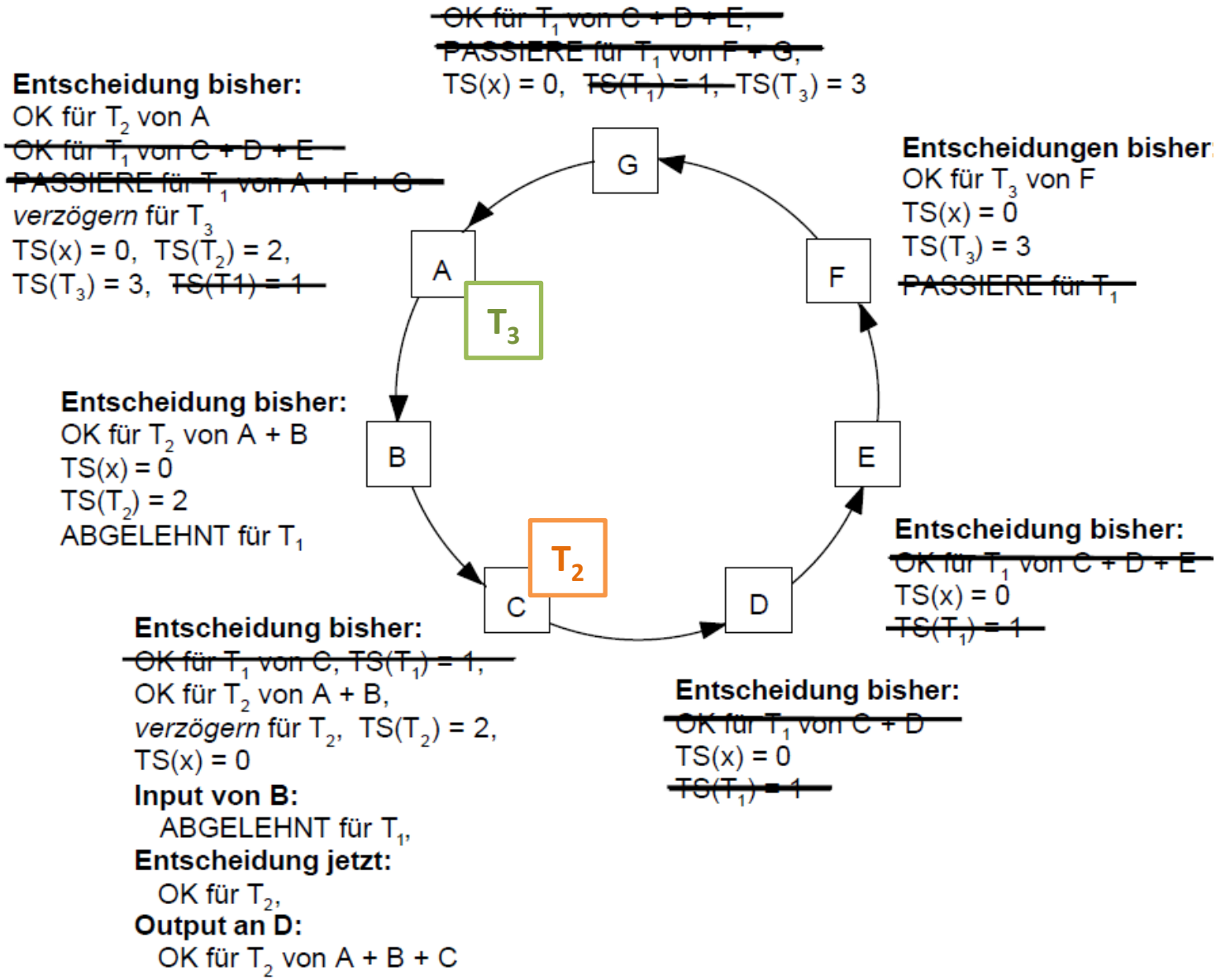
Output an alle:
ABGELEHNT für T_1, TS(T_1) = 1

Entscheidung bisher:
OK für T_1 von C + D + E
TS(x) = 0
TS(T_1) = 1

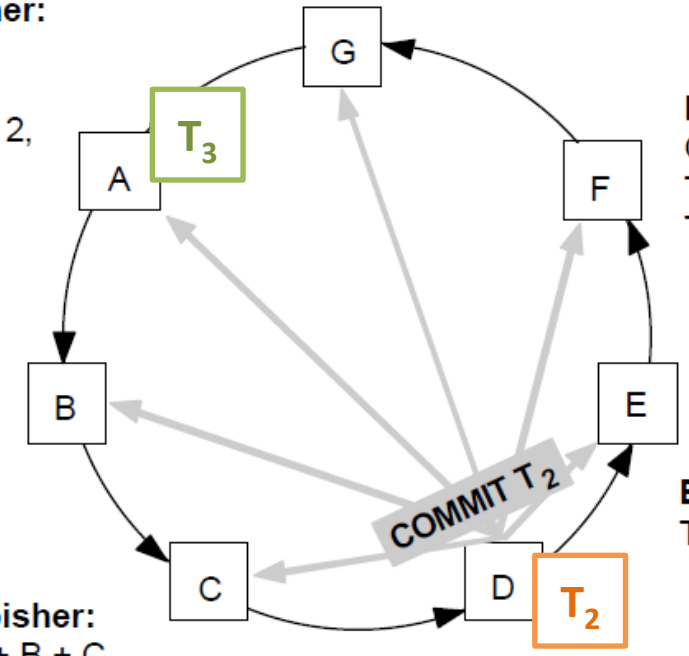
Entscheidung bisher:
OK für T_1 von C, TS(T_1) = 1,
OK für T_2 von A + B,
verzögern für T_2, TS(T_2) = 2,
TS(x) = 0

Entscheidung bisher:
OK für T_1 von C + D
TS(x) = 0
TS(T_1) = 1

Beispiel (t=8)


~~OK für T_1 von C + D + E,~~
~~PASSIERE für T_1 von F + G,~~
TS(x) = 0, ~~$TS(T_1) = 1$,~~ $TS(T_3) = 3$
Entscheidung bisher:
OK für T_2 von A
~~OK für T_1 von C + D + E~~
~~PASSIERE für T_1 von A + F + G~~
verzögern für T_3
TS(x) = 0, $TS(T_2) = 2$,
$TS(T_3) = 3$, ~~TS(T1) = 1~~
Entscheidungen bisher:
OK für T_3 von F
TS(x) = 0
$TS(T_3) = 3$
~~PASSIERE für T_1~~
G
A
F
T_3
Entscheidung bisher:
OK für T_2 von A + B
TS(x) = 0
$TS(T_2) = 2$
ABGELEHNT für T_1
B
E
T_2
Entscheidung bisher:
~~OK für T_1 von C + D + E~~
TS(x) = 0
~~$TS(T_1) = 1$~~
C
D
Entscheidung bisher:
~~OK für T_1 von C, $TS(T_1) = 1$,~~
OK für T_2 von A + B,
verzögern für T_2, $TS(T_2) = 2$,
TS(x) = 0
Input von B:
ABGELEHNT für T_1,
Entscheidung jetzt:
OK für T_2,
Output an D:
OK für T_2 von A + B + C
Entscheidung bisher:
~~OK für T_1 von C + D~~
TS(x) = 0
~~$TS(T_1) = 1$~~

Beispiel (t=9)



Entscheidung bisher:
OK für T_3 von F + G
TS(x) = 0, TS(T_3) = 3

Entscheidung bisher:
OK für T_2 von A
verzögern für T_3
TS(x) = 0, TS(T_2) = 2,
TS(T_3) = 3

Entscheidungen bisher:
OK für T_3 von F
TS(x) = 0
TS(T_3) = 3

Entscheidung bisher:
OK für T_2 von A + B
TS(x) = 0
TS(T_2) = 2
ABGELEHNT für T_1

Entscheidung bisher:
TS(x) = 0

Entscheidung bisher:
OK für T_2 von A + B + C,
TS(T_2) = 2,
TS(x) = 0

Entscheidung bisher:
OK für T_2 von A + B + C,
TS(x) = 0, TS(T_2) = 2,
Input von C:
OK für T_2 von A + B + C
Entscheidung jetzt:
OK für T_2 ➞ COMMIT
Output an alle:
COMMIT für T_2

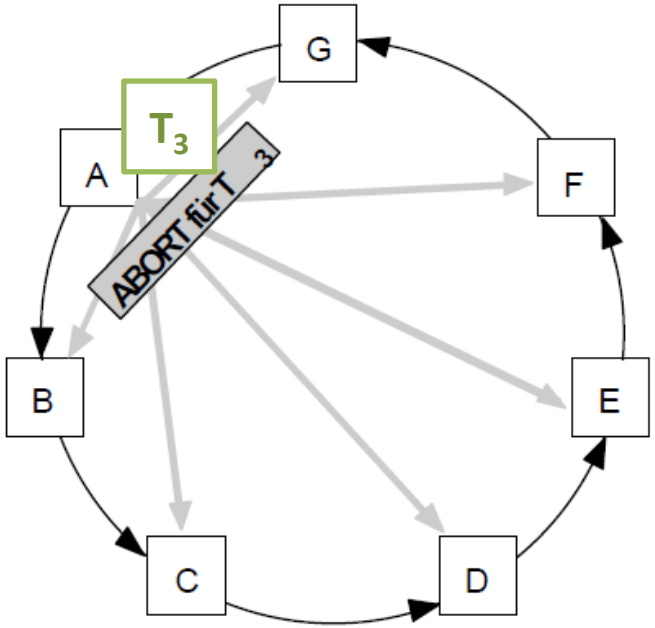
Beispiel (t=10)

verzögern für T_3
$TS(x) = 0$, $TS(T_2) = 2$,
$TS(T_3) = 3$
Input von D:
COMMIT für T_2
Entscheidung jetzt:
$TS(x) = 2$,
ABGELEHNT für T_3
(da Inputobjekte von T_3 nun veraltet)
Output an alle:
ABORT für T_3

Entscheidung bisher:
OK für T_2 von A + B
$TS(x) = 0$
$TS(T_2) = 2$
Input von D:
COMMIT für T_2
Entscheidung jetzt:
$TS(x) = 2$

Entscheidung bisher:
OK für T_2 von A + B + C,
$TS(T_2) = 2$,
$TS(x) = 0$
Input von D:
COMMIT für T_2
Entscheidung jetzt:
$TS(x) = 2$

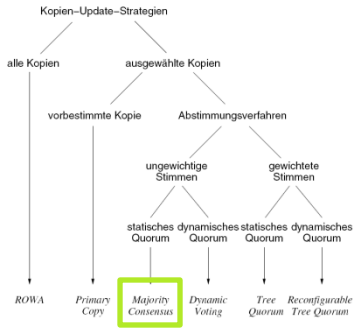
$TS(x) = 0$, $TS(T_3) = 3$
Input von D:
COMMIT für T_2
Entscheidung jetzt:
$TS(x) = 2$



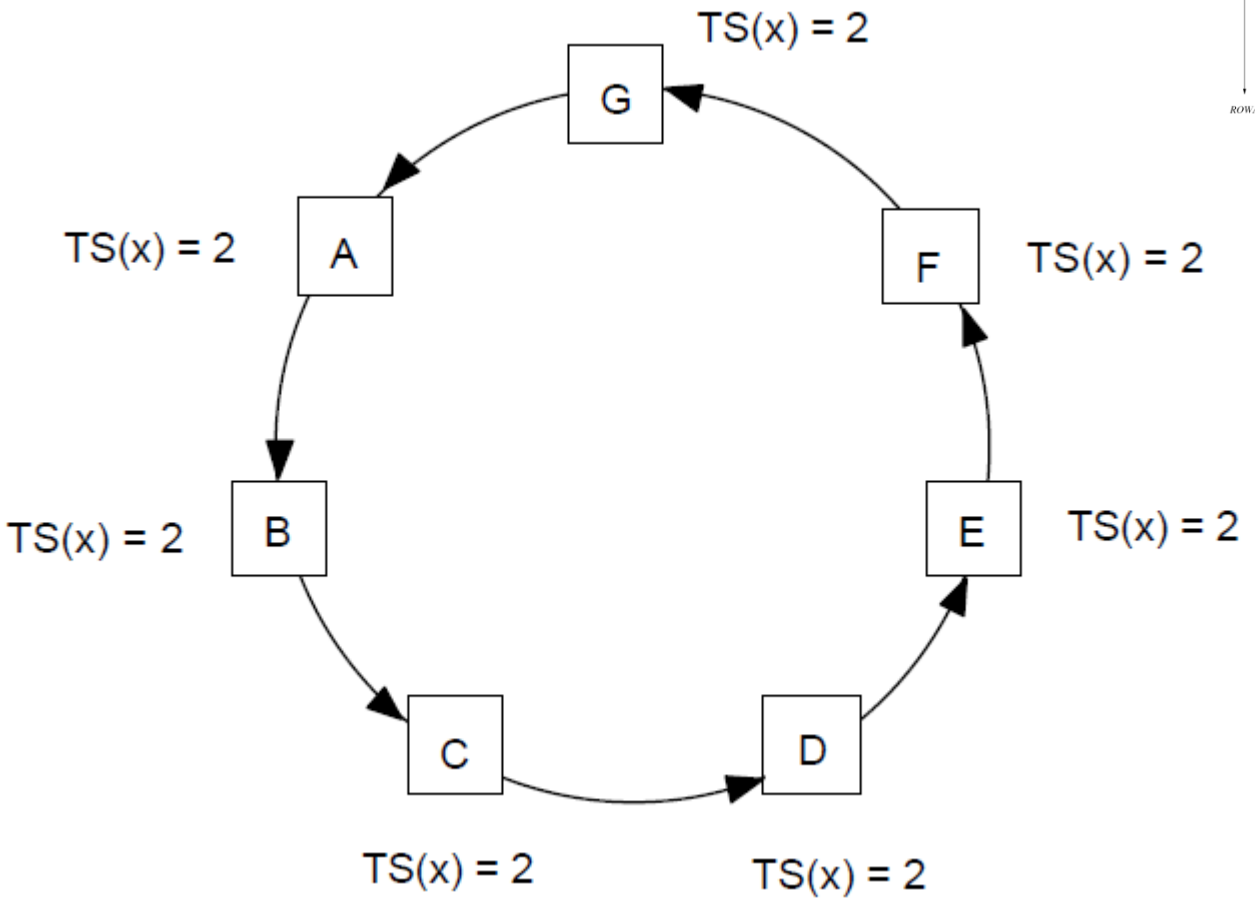
Entscheidungen bisher:
OK für T_3 von F
$TS(x) = 0$
$TS(T_3) = 3$
Input von D:
COMMIT für T_2
Entscheidung jetzt:
$TS(x) = 2$

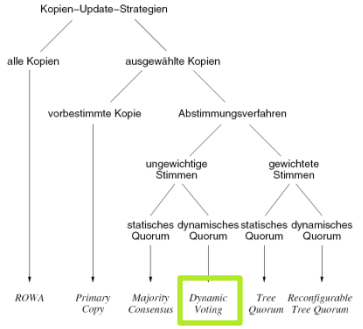
Entscheidung bisher:
$TS(x) = 0$
Input von D:
COMMIT für T_2
Entscheidung jetzt:
$TS(x) = 2$

Entscheidung bisher:
$TS(x) = 2$



Beispiel (t=11)



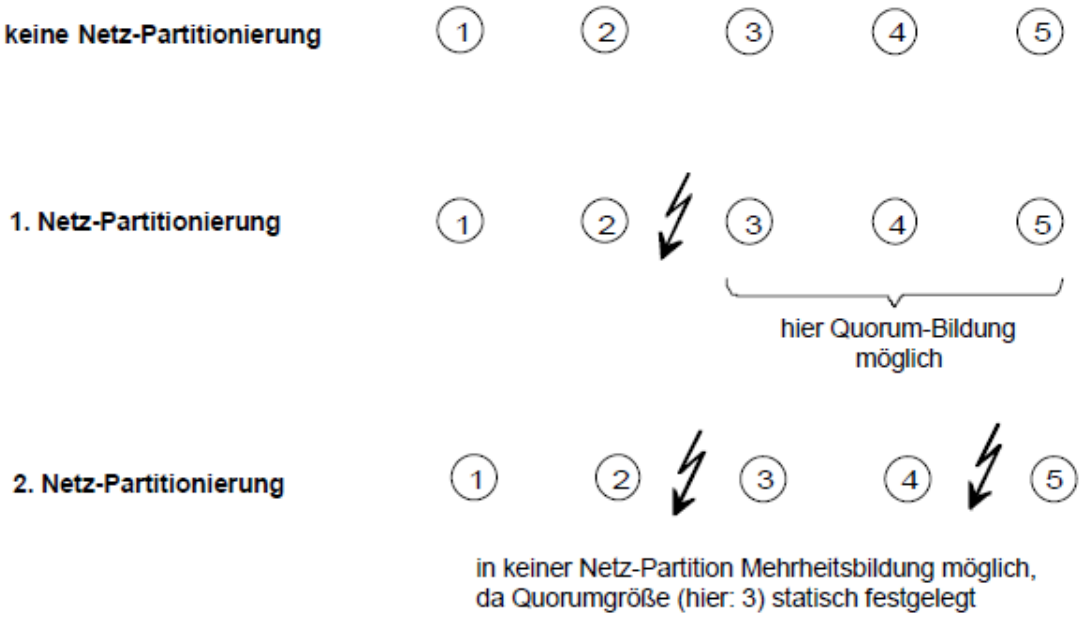


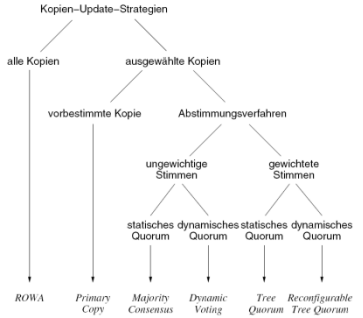
Dynamic Voting

- Majority Consensus mit dynamischer Quorumbildung

Problem Majority Voting

- Bei mehrfacher Netzpartitionierung / Knotenausfällen, Quorumbildung ggf. nicht mehr möglich





Idee

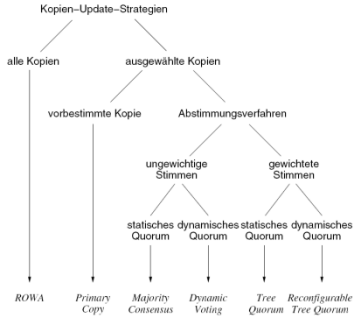
- Nur noch diejenigen Knoten, die beim letzten Update beteiligt waren besitzen Stimmrecht
- VN: Versionsnummer der Kopie
- SK: Anzahl der Knoten die beim letzten Update der Kopie beteiligt waren (gegenwärtige Gesamtstimmenzahl)

Beispiel

- Ausgangssituation: 9 Updates auf allen 5 Kopien erfolgreich → VN = 9, SK = 5

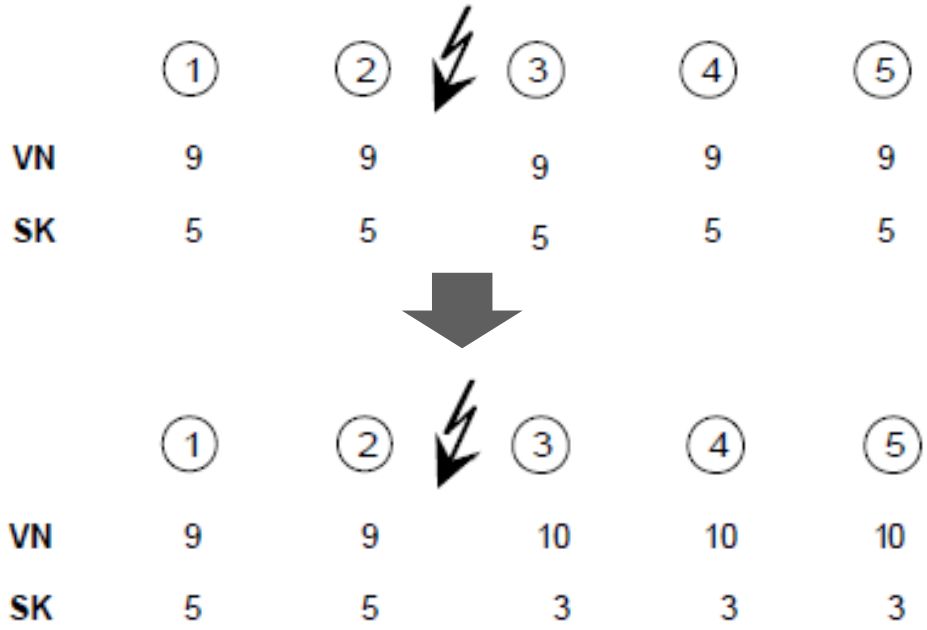
	1	2	3	4	5
VN	9	9	9	9	9
SK	5	5	5	5	5

> Dynamic Voting (3)



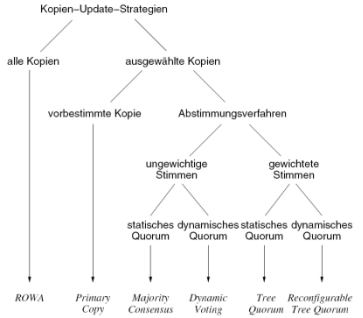
Beispiel

- Erste Netz-Partitionierung, anschl. Update-Anforderung an Knoten 3



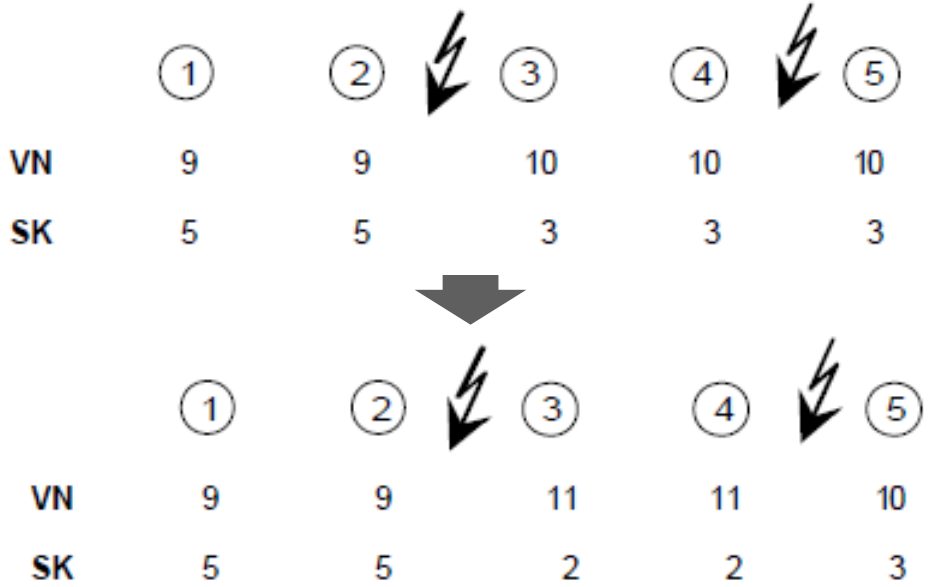
Analyse

- Knoten 3 gehört zur Hauptpartition (mit Knoten 3,4,5), da Quorum-Bildung möglich: round(SK/2) = round(5/2) = 3
- Update zulässig: neue VN = 10
- nur Knoten 3, 4, 5 führen Update durch → SK = 3 (dort)



Beispiel

- Zweite Netz-Partitionierung, anschl. Update-Anforderung an Knoten 4

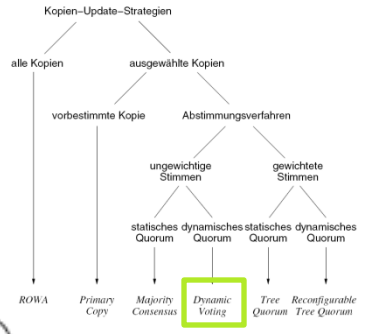


Analyse

- Knoten 4 gehört zur Hauptpartition (bestehend aus Knoten 3 und 4), da nur noch 3 Knoten stimmberechtigt (SK = 3) und im geg. Fall round(SK/2) = round(3/2) = 2 gilt
- Update zulässig, VNneu = 11

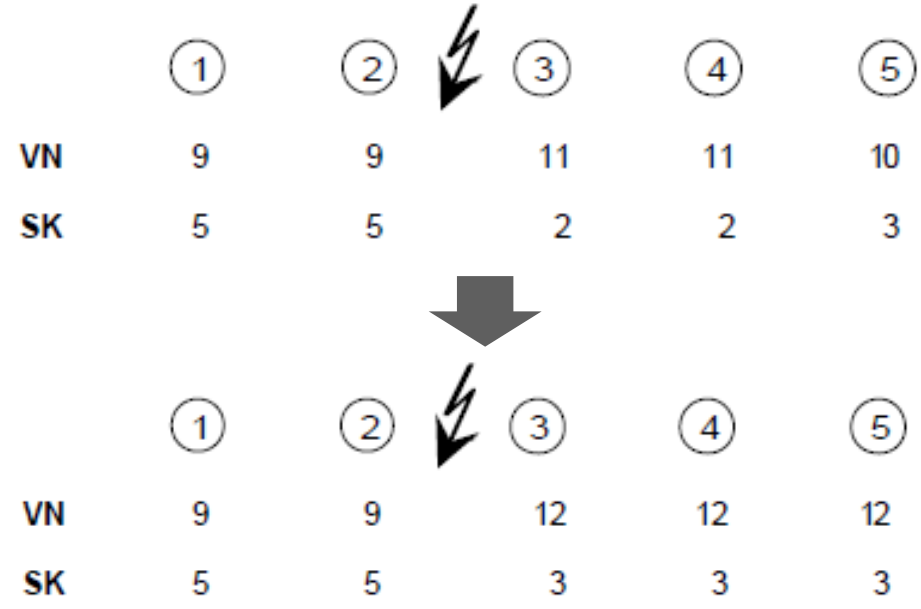
Anmerkung

- Das "normale" Majority Consensus-Verfahren würde hier solange blockieren bzw. den Request zurückweisen (da hier als Quorum 3 benötigt wird), bis die (zweite) Netz-Partitionierung wieder beseitigt ist.



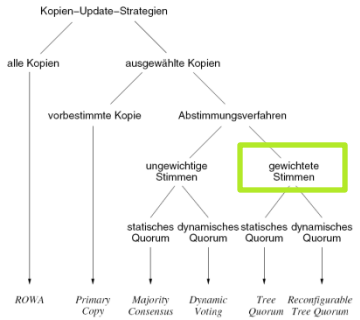
Beispiel

- Zweite Netz-Partitionierung beendet, Update-Anforderung an Knoten 4
- Wiederherstellung des Stimmrechts für Knoten 5



Analyse

- Anfrage von Knoten 4 an alle erreichbaren Knoten
- Knoten 3 und 5 melden sich (jeweils mit VN und SK)
- Knoten 3 besitzt Stimmrecht (VN und SK sind aktuell) und stimmt Update-Anforderung zu
- Update kann durchgeführt werden: VNneu = 12, SKneu = 3
- Update mit VNneu und SKneu wird an Knoten 3 und 5 verschickt



Gewichtetes Votieren (QC / Quorum Consensus)

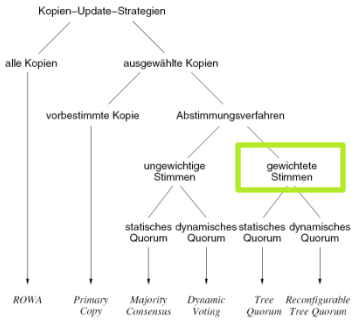
- Verallgemeinerung des Mehrheitsvotierens
- Jede Kopie erhält bestimmte Anzahl von Stimmen (votes)

Protokoll

- Lesen erfordert R Stimmen (Lese-Quorum)
  - ist ein Lese-Quorum erworben, so wird das „neueste“ Replikat aus dem Quorum gelesen
- Schreiben erfordert W Stimmen (Schreib-Quorum)
  - ist ein Schreibquorum erworben, werden alle Replikate, die am Quorum beteiligt sind, geschrieben

Bestimmung der Quoren so dass gilt (siehe Quorum-Überschneidungsregel)

- R + W > V (= Summe aller Stimmen)
  - Jedes Lesequorum überlappt mit jedem Schreibquorum
  - Garantiert erkennen von Schreib/Lese-Konflikten
  - Jedes Lesequorum enthält mindestens ein aktuelles Replikat
- 2 * W > V
  - Zwei Schreibquoren müssen überlappen
  - Garantiert erkennen von Schreib/Schreib-Konflikten

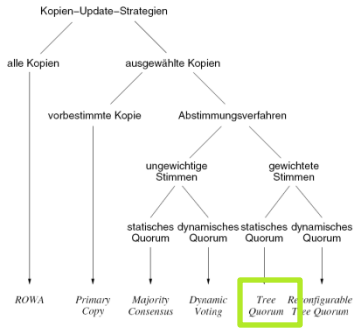


Gewichtetes Votieren (QC / Quorum Consensus)

- Festlegung von V, R und W erlaubt Trade-Off zwischen Lese- und Schreibkosten sowie zwischen Leistung und Verfügbarkeit
- Flexibilität
  - Hohes Gewicht an einem Replikat: Besonders schneller Zugriff
  - Kleines R, Großes W:
    Hohe Verfügbarkeit für Leser, Geringe Verfügbarkeit für Schreiber
  - Kleines W (soweit möglich), Großes R:
    Geringere Verfügbarkeit für Leser dafür höhere Verfügbarkeit für Schreiber

Spezialfälle

- Mehrheitsvotieren: Stimmengewicht 1; R=W=(V/2 +1)
- ROWA: Stimmengewicht 1; R=1; W=V
- Primary Copy: Gewicht 1 für Primärkopie - Alle übrigen: Gewicht 0

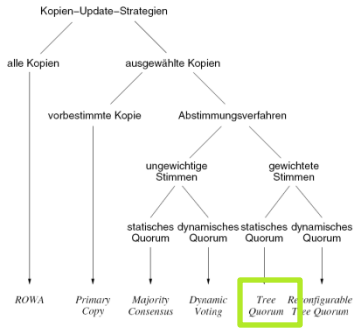


Tree Quorum Consensus (Hierarchisches Votieren)

- Knoten/Kopien sind in einer logischen Baumstruktur angeordnet
  → reduziert Nachrichtenanzahl
  - h enspricht der Höhe und
  - v dem Verzweigungsgrad des Baums

Vorgehen

- Die Mehrheit der Ebenen muss zustimmen (statt Mehrheit der Knoten)
  - Mehrheit einer Ebene = Mehrheit der Knoten in dieser Ebene
- Dadurch Serialisierbarkeit gewährleistet
- Antwortet ein Knoten nicht (timeout), kann dessen Akzeptanz durch die Mehrheit seiner Kinder ersetzt werden
- Zahl der zu sperrenden Replikate hängt davon ab, welche Replikate ausfallen

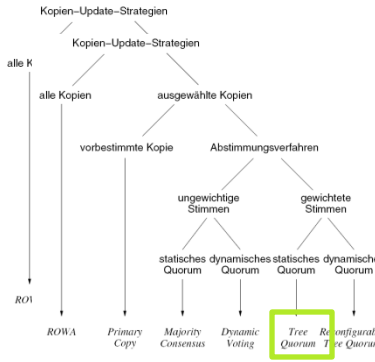


Lese- und Schreibquoren

- Definiert durch $q_r = (l_r, a_r)$ und $q_w = (l_w, a_w)$
- Mindestens a Knoten in l Ebenen müssen akzepzieren
- Hat eine Ebene weniger als a Knoten so müssen alle Knoten akzeptieren
- Parameter l und a müssen in Abhängigkeit der Baumhöhe (h) und der –verzweigung bestimmt werden

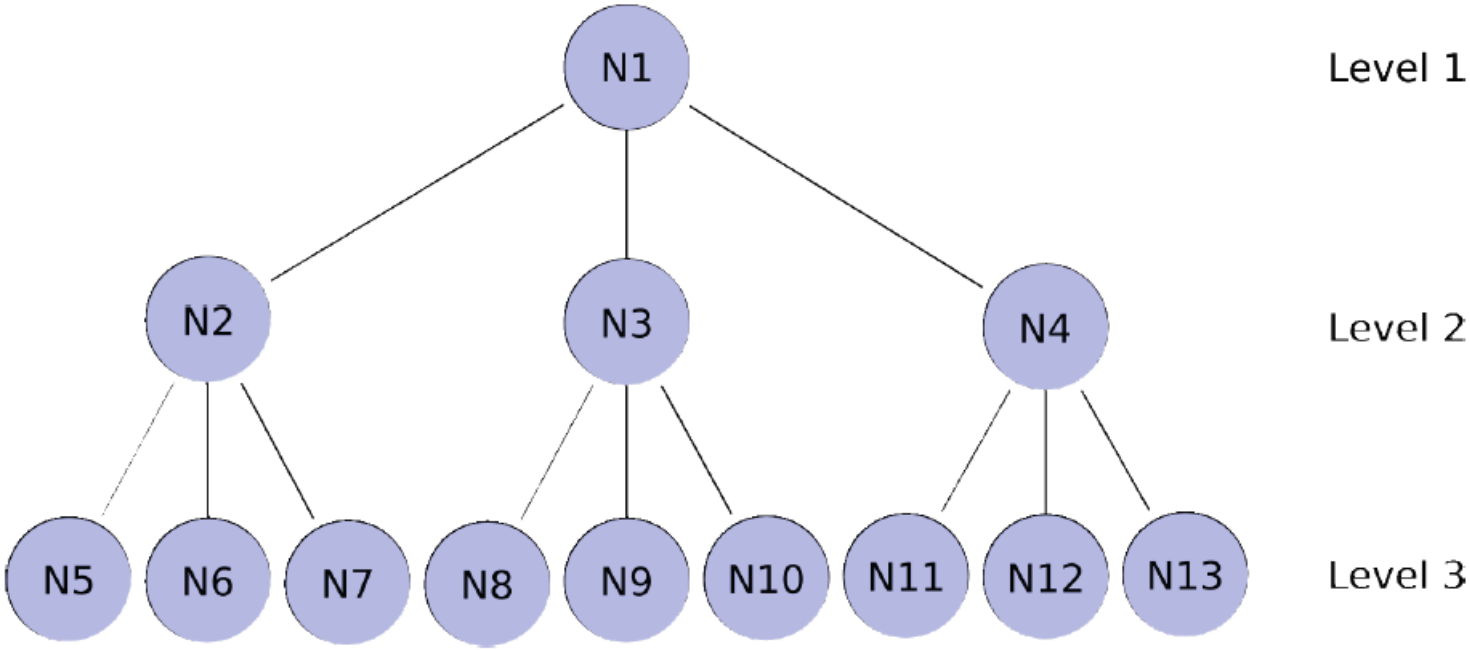
Um Serialisierbarkeit zu gewährleisten, wähle l und a so dass

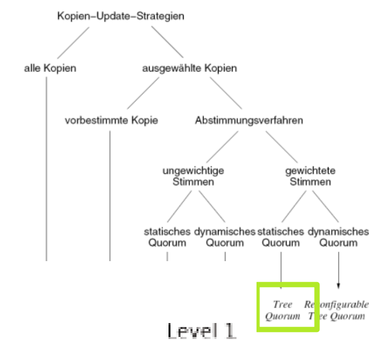
- $2 \cdot l_w > h$, $2 \cdot a_w > v$ (Schreib-Schreib-Konflikte)
- $l_r + l_w > h$, $a_r + a_w > v$ (Lese-Schreib-Konflikte)



Beispiel

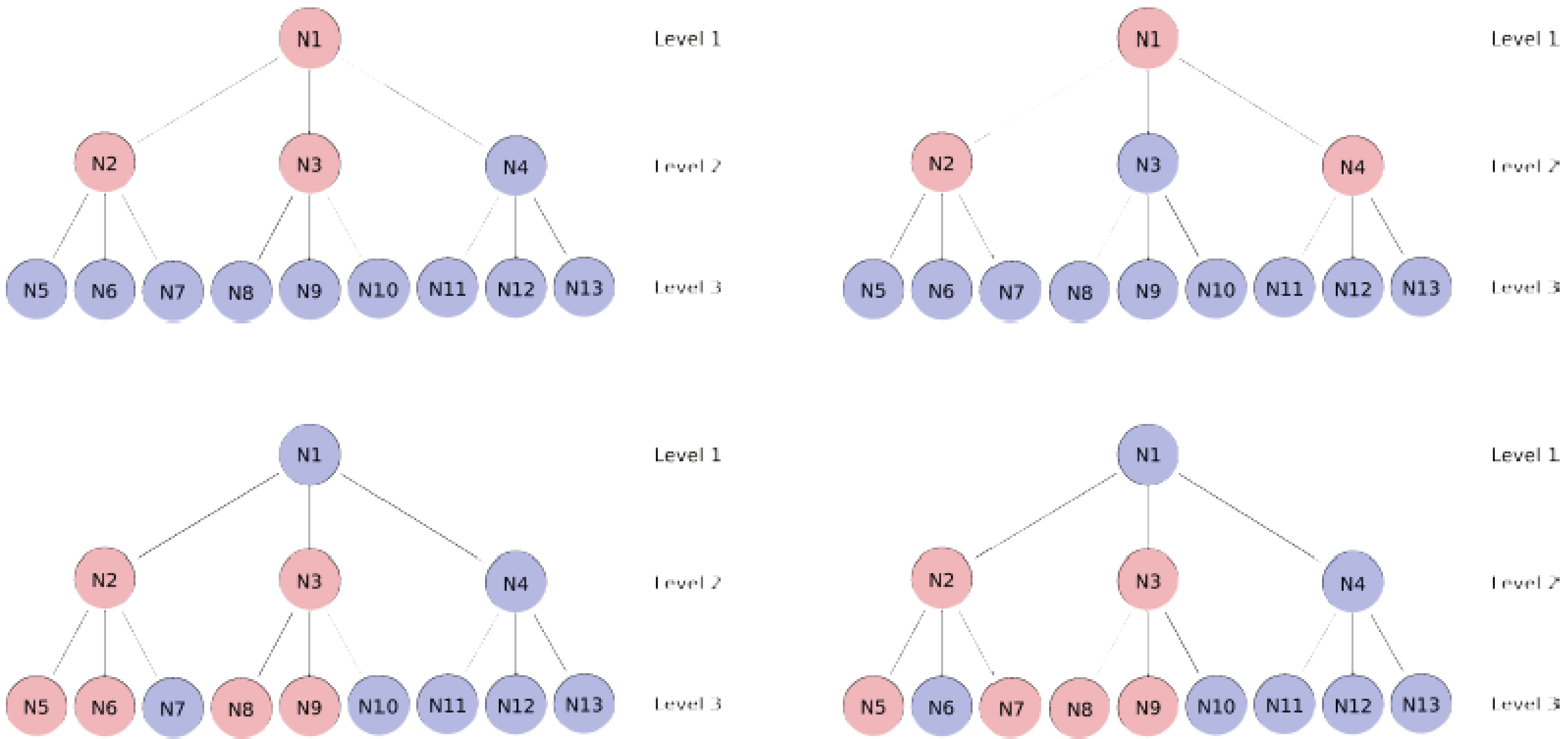
- h = 3, v = 3
- Q_w = (2,2), Q_r = (2, 2)



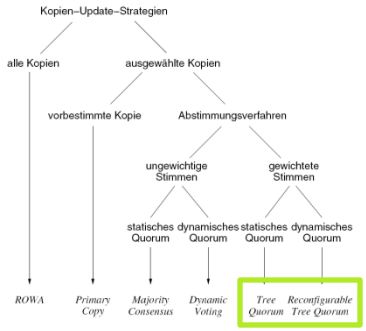


Beispiel (fortgesetzt)

- h = 3, v = 3
- $Q_w = (2,2)$, $Q_r = (2, 2)$



→ *Majority Consensus würde immer 7 Stimmen benötigen*



Vorteile gegenüber dem ungewichteten Votieren

- Weniger Stimmen notwendig für Lese- bzw. Schreibanforderungen

Nachteile

- Knoten im oberen Teil des Baums müssen mehr Anfragen bewältigen
- Falls Knoten aus dem oberen Teilen ausfallen, müssen mehr Stimmen auf tiefere Ebene gesammelt werden

Reconfigurable Tree Quorum

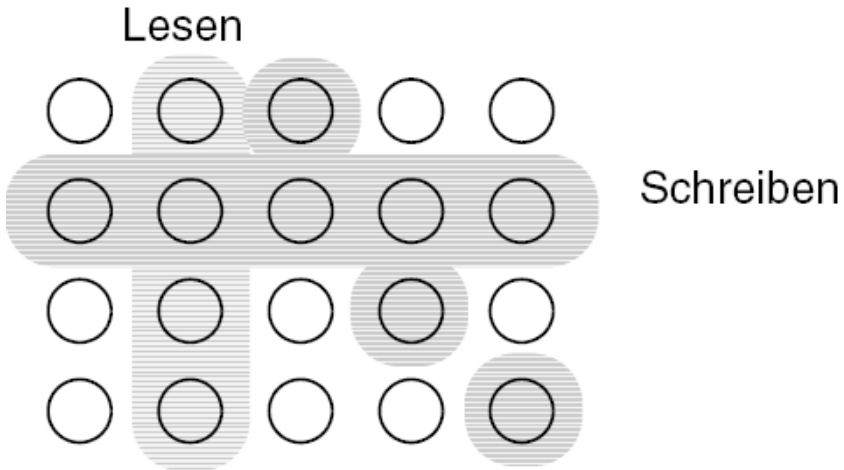
- Bestimmung von q zur Laufzeit (bzgl. Partitionierung und Quorum-Überschneidungsregel)

Varianten der Votierungsverfahren bei großer Zahl von Replikaten

- Gitterprotokoll
  - Anordnung der Replikate als logische Matrix
  - Quorenbildung nur durch bestimmte Kombinationen von Replikaten
  - Überlappung von Schreibquoren mit anderen Quoren muss garantiert werden
  - Weniger Replikate müssen zugegriffen werden

Beispiel

- für zweidimensionale Anordnung
  (Hier im Beispiel: V= 20; R= 4; W= 8)



Example: Amazon Dynamo

Sloppy Quorum of Dynamo

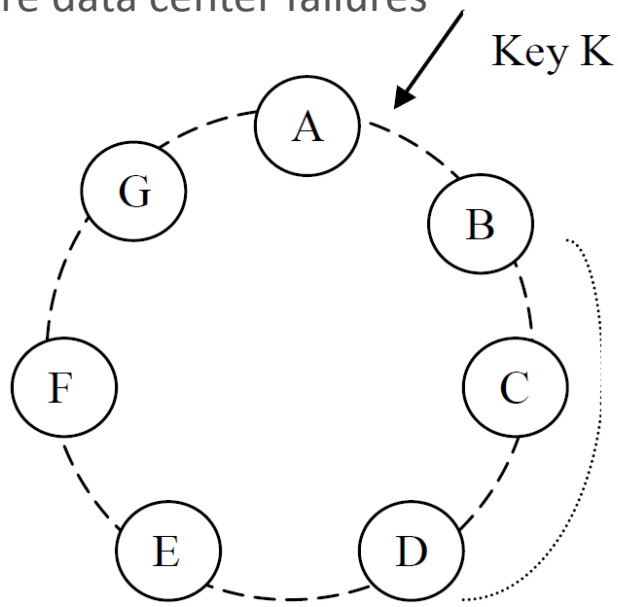
- R (/W) is the minimum number of nodes that must participate in a successful read (/write) operation
- N is the number of highest-ranked **reachable** nodes
- Setting R + W > N yields a quorum-like system
- The latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas
- Hence, R and W are usually configured to be less than N, to provide better latency

Receiving a put() request

- Coordinator generates the vector clock and writes the new version locally
- Coordinator sends the new version (with the new vector clock) to the N highest-ranked reachable nodes
- If at least W-1 nodes respond then the write is considered successful

Receiving a get() request

- Coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key
- Waits for R responses before returning the result to the client



Handling failures → Hinted Handoff

- All read and write operations are performed on the first N *healthy* nodes from the preference list, which may NOT always be the first N nodes encountered while walking the consistent hashing ring
- Ensures that read and write operations are not failed due to temporary node or network failures (→ "always writeable")
- Each object is replicated across multiple data centers, which are connected through high-speed network links → compensate entire data center failures

Example (N=3)

- If node b is down during a write operation then a replica that would normally have lived on B will now be sent to node E
- Replica sent to B includes metadata about node which was intended recipient (e.g. B)
- Hinted replicas are kept in a separate local database that is scanned periodically
- Upon detecting that B has recovered, E delivers replica back to B

Hinted Handoff requires replica synchronization (anti-entropy) protocol

- Periodically, check if replicas are in sync
- If replicas out-of-sync, bring replicas in sync

Problem

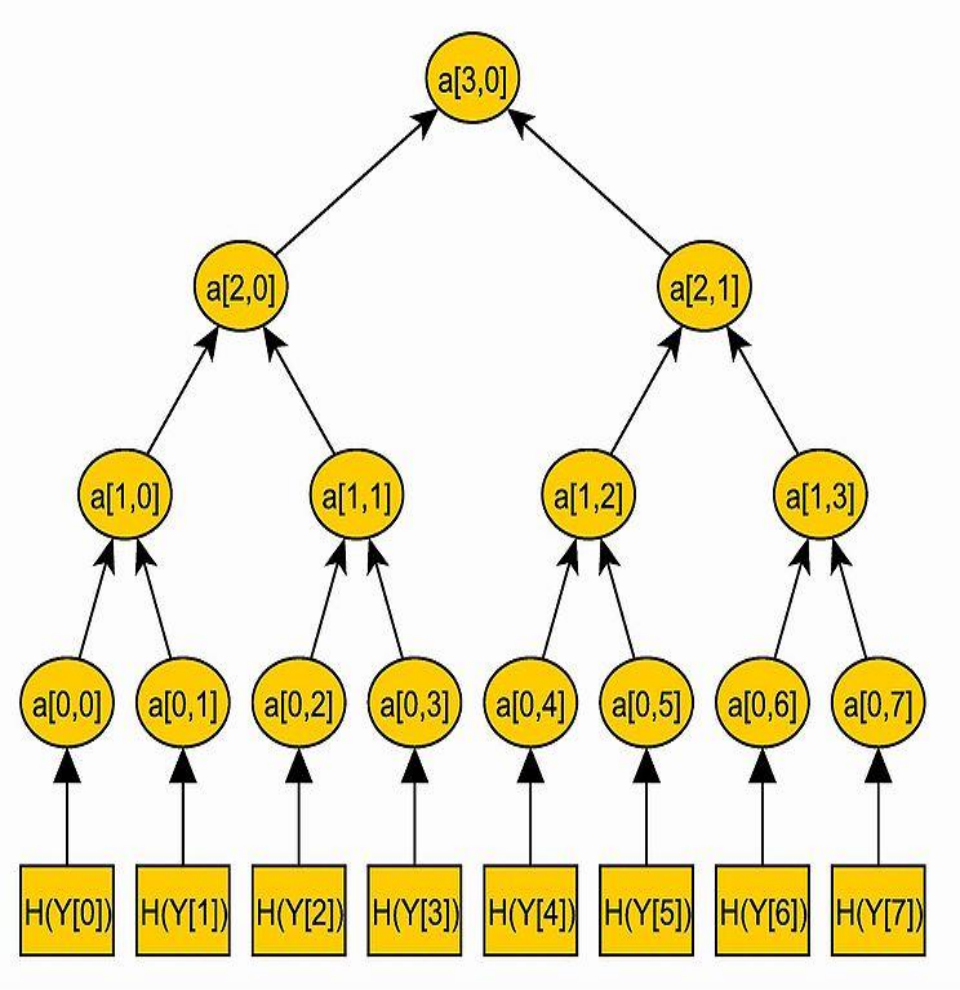
- Large state of replicas

Approach

- Use **Merkle Trees** to minimize data transfer and to detect inconsistencies between replicas faster
- Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process
- Disadvantage – When a new node leaves or joins, the trees need to be recalculated

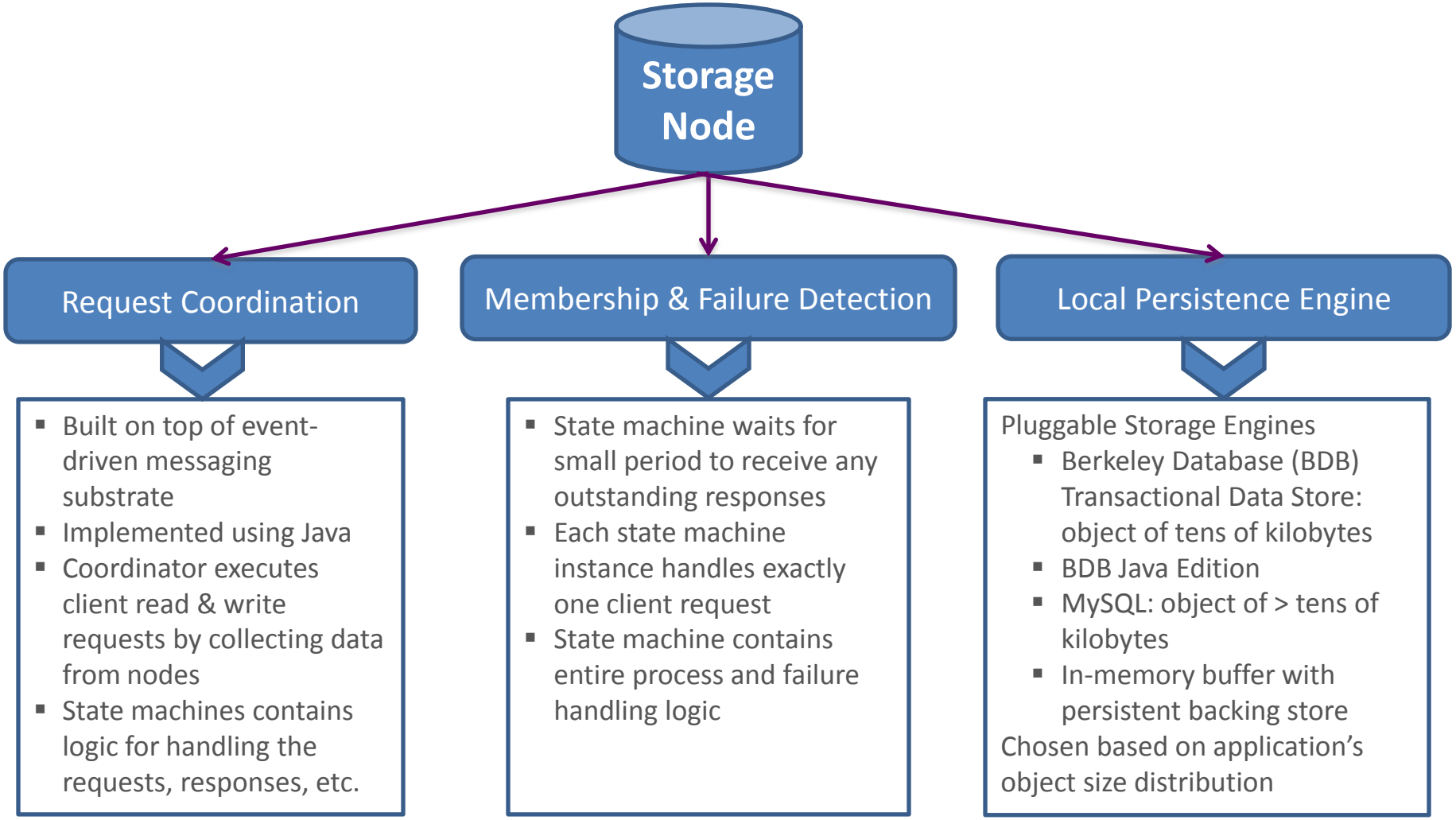
Approach

- A Merkle tree is a hash tree where leaves are hashes of the values of individual keys
- Parent nodes higher in the tree are hashes of their respective children
- Advantages
  - Each branch can be checked independently without requiring nodes to download the entire data set
  - Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas



Ring Membership

- Explicit mechanism to add and remove nodes from a Dynamo ring
- Command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring
- Node that serves the request writes the membership change and its time of issue to persistent store
- Membership changes form a history because nodes can be removed and added back multiple times
- Use of **seed nodes** to prevent partition
- A gossip-based protocol propagates through the ring and discovers other nodes
- Failure detection is done in a similar gossip protocol to avoid attempts to communicate with unreachable peers
- Preserves the symmetry design goal


Storage Node
Request Coordination
Membership & Failure Detection
Local Persistence Engine
Built on top of event-driven messaging substrate
Implemented using Java
Coordinator executes client read & write requests by collecting data from nodes
State machines contains logic for handling the requests, responses, etc.
State machine waits for small period to receive any outstanding responses
Each state machine instance handles exactly one client request
State machine contains entire process and failure handling logic
Pluggable Storage Engines
Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
BDB Java Edition
MySQL: object of > tens of kilobytes
In-memory buffer with persistent backing store
Chosen based on application's object size distribution

Characteristics of Dynamo can be tuned via

- Replication factor N
- Read quorum R
- Write quorum W

N	R	W	Application
3	2	2	Consistent, durable, interactive user state (typical configuration)
n	1	n	High-performance read engine
1	1	1	Distributed web cache

Note: Dynamo does not enforce strict **quorum membership**.

- Reads and writes might go to non-overlapping nodes even when R+W > N.

Asynchrone Replikationsverfahren

Bisher

- Ubiquitätsprinzip
  - Vollständige Replikationstransparenz
  - Jede Kopie ist immer auf dem aktuellsten Stand (strenge Konsistenz)
  - Jede Kopie kann gelesen und geändert werden
  - Resultierendes Korrektheitskriterium für Transaktionen: One-Copy-Serialisierbarkeit
- Probleme
  - Hohe Änderungskosten, da Änderungen synchron propagiert werden
  - Verfügbarkeitsprobleme (z.B. geringe Schreibverfügbarkeit bei ROWA)
  - Vermindert die Zugriffslokalität → Verminderung der Performance
  - Besonders schlecht bei geographischer Verteilung
  - Synchronisation von Lesern und Schreibern bzw. Schreiber untereinander

Need-To-Know-Prinzip

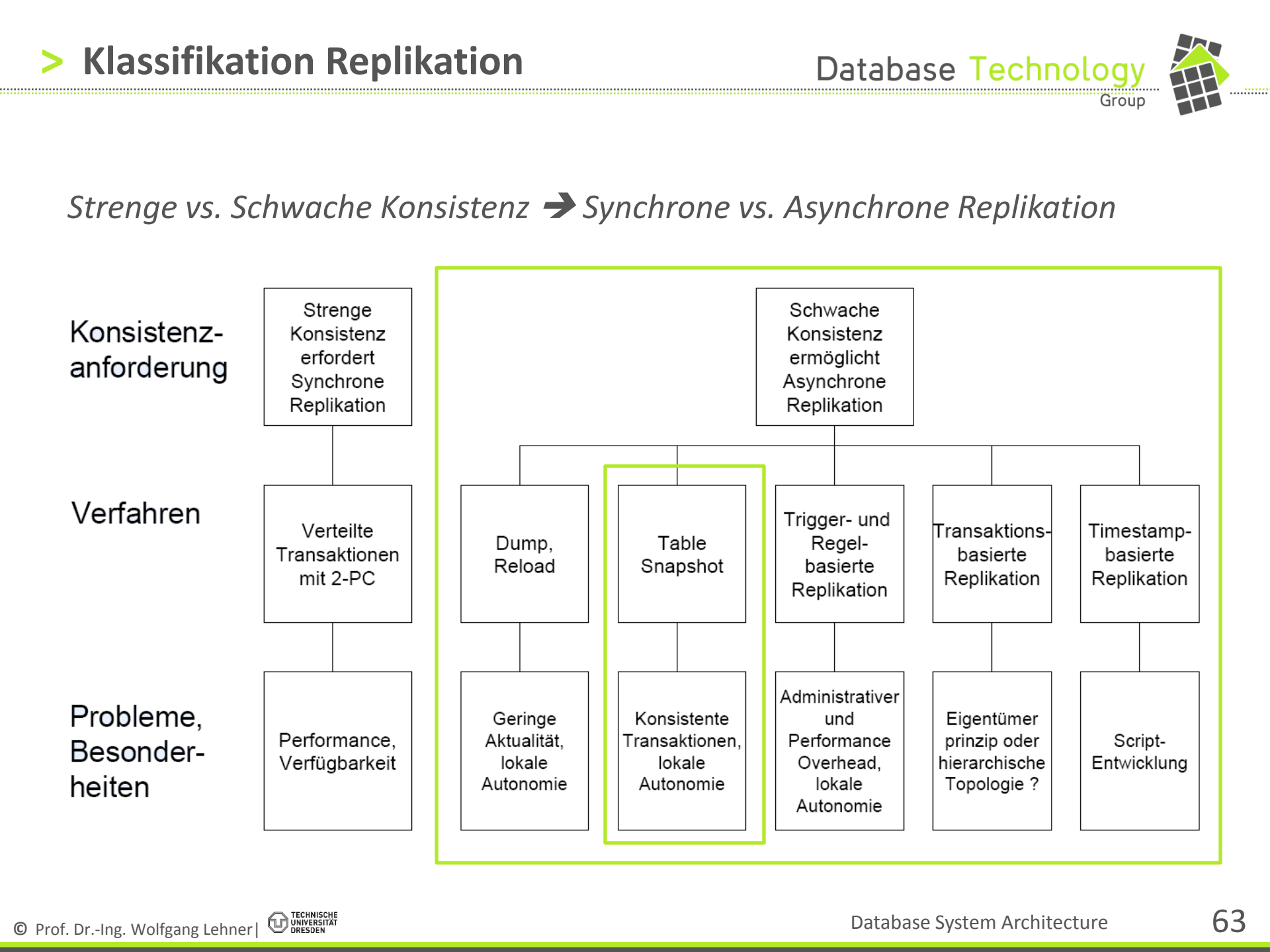
- Grundlage der asynchronen Replikation
- Einer Applikation werden nur die Daten bereitgestellt, die sie benötigt
  - Daten werden nur so aktuell bereitgestellt wie nötig
  - Daten werden nur so konsistent bereitgestellt wie nötig

Lese-/Schreib-Synchronisation

- Verzögerte Aktualisierung von Replikaten (asynchrone Propagierung)
- Ereignisorientierte Aktualisierung / kontinuierliche Aktualisierung

Schreib-/Schreib-Synchronisation

- Unabhängige Änderungen (nur eingeschränkt verwendbar)
- Nicht verwendbar wenn mindestens ein aktuelles Replikat benötigt wird
- Problem: Sicherstellung der Konvergenz von Replikaten
- Zusammenführung (Merge) erforderlich



Strenge vs. Schwache Konsistenz ➔ Synchrone vs. Asynchrone Replikation

Prinzip

- Nur-Lesekopie eines konsistenten DB-Ausschnitts (materialisierte Sicht) mit periodischem oder explizit veranlasstem „Refresh“
  - SPJ: Selektion, Projektion und Join
  - SPJG: zusätzlich Aggregationsoperationen
- Änderungen üblicherweise nur am Masterknoten (primary copy-Ansatz)

Änderungen

- Ereignisorientierte (asynchrone) Aktualisierung durch Refresh-Operationen
  - Snapshot-Knoten kann Refresh-Häufigkeit bestimmen
- Zentral: effiziente Realisierung der Aktualisierung
- Updatable Snapshots: Materialisierte Sichten, die eine Rückwärtspropagierung der lokal durchgeführten Änderungen ermöglichen - Konfliktauflösung am Masterknoten!

Ziel

- Performanceverbesserung der Replikatverwaltung durch Anwendungsanalyse

Prinzip

- Definition separater Datenbestände für Benutzergruppe bzw. Anwendungsklassen (Beispiele: Statistik, Test)
- individuelle Definition der Replikatsverwaltung

Modell

- Basisrelation
  - Primärkopie: Schreib- und Lesekopie
- Snapshot-Relation
  - abgeleitete Sekundärkopie über Query (SELECT ...): Nur-Lesekopie
- Refreshoperation
  - periodisches Aktualisieren des Snapshots durch Übermittlung veränderter Basisdaten
  - anwendungsspezifisch

Snapshot-Refresh

Full Refresh

- Neuauswerten der Snapshotquery bei jedem Refresh und vollständige Übertragung
- Ersetzten des alten Snapshotinhalts durch den transferierten
- Einfaches Verfahren, anzuwenden für Recovery

Immediate Refresh

- sofortige Übertragung geänderter Tupel
- Widerspruch zur Definition

Buffering in Basis-Relation

- Puffern von Änderungen in Basisrelation
- Löschen der Puffer erst wenn alle Snapshots versorgt sind

Differential Refresh

- Kennzeichnen aller geänderten Tupel
- nur Transferieren geänderter Tupel

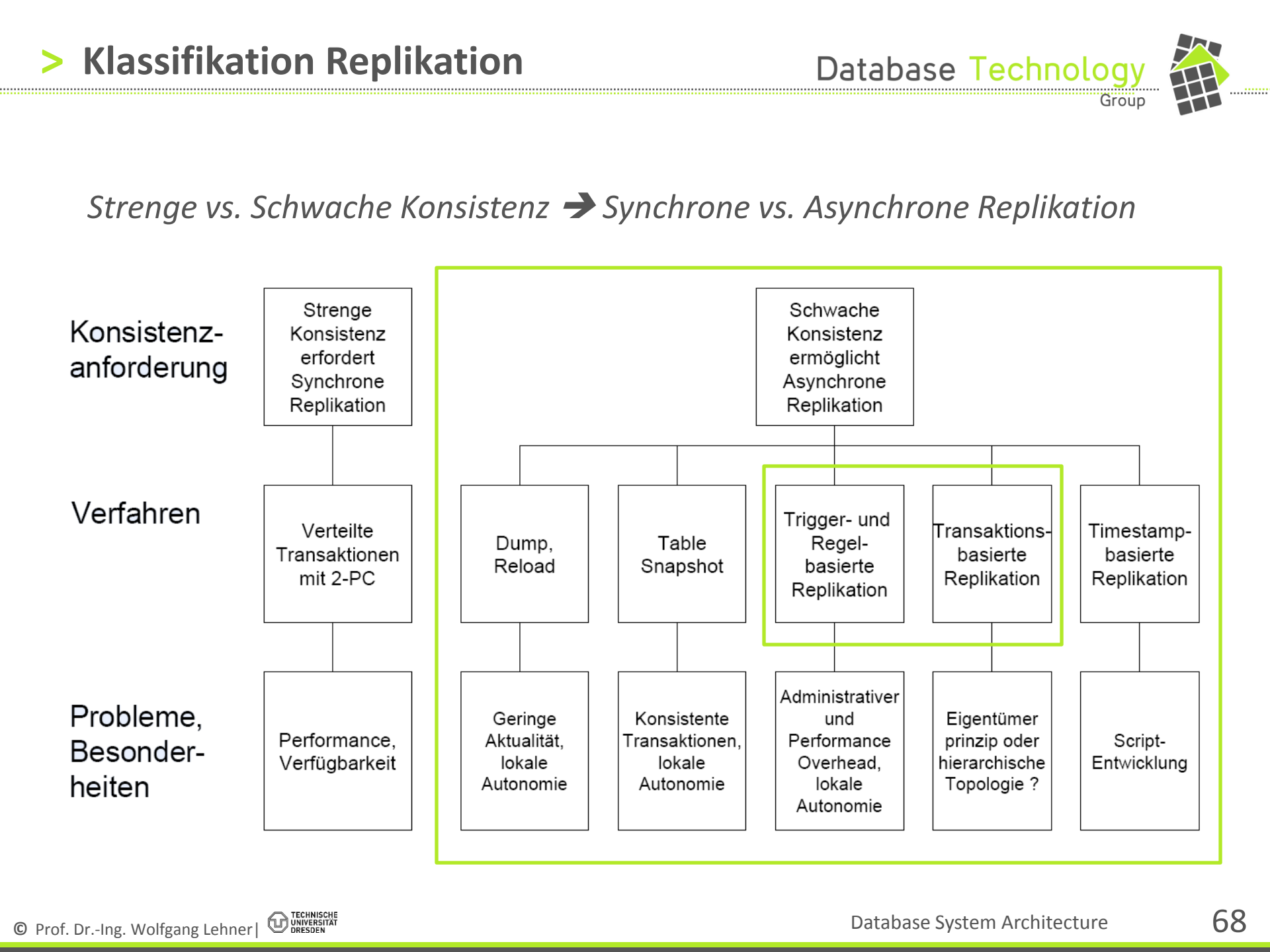
Voraussetzung

- Erweiterung der Basisrelation
  - addr — eindeutige Kennzeichnung des Tupels
  - timestamp — Zeitpunkt der letzten Änderung
  - status — Zustand des Tupels (existiert / ist gelöscht)
- Erweiterung der Snapshotrelation
  - baseaddr — korrespondierendes Tupel der Basisrelation
  - snaptime — Zeitpunkt des letzten Refreshs
  - snaprestrict — Snapshot Query

Durchführung

- Extrahieren der Tupel aus Basisrelation mit: timestamp > snaptime
- Neusetzen der snaptime
- Einbringen der transferierten Tupel am Snapshot

```
if status = empty then
    delete tuple
else
    insert/update tuple
```



Strenge vs. Schwache Konsistenz ➔ Synchrone vs. Asynchrone Replikation

Konsistenz-anforderung

Verfahren

Probleme, Besonder-heiten

Strenge Konsistenz erfordert Synchrone Replikation

Schwache Konsistenz ermöglicht Asynchrone Replikation

Verteilte Transaktionen mit 2-PC

Dump, Reload

Table Snapshot

Trigger- und Regel-basierte Replikation

Transaktions-basierte Replikation

Timestamp-basierte Replikation

Performance, Verfügbarkeit

Geringe Aktualität, lokale Autonomie

Konsistente Transaktionen, lokale Autonomie

Administrativer und Performance Overhead, lokale Autonomie

Eigentümer prinzip oder hierarchische Topologie ?

Script-Entwicklung

Zeitpunkt der Aktualisierung

- Synchron: innerhalb einer Transaktion (eager replication)
  - z.B. triggerbasierte Replikation
- Asynchron: separate Transaktion (lazy replication)
  - z.B. TX-basierte Replikation

Probleme synchroner Updates

- Eingeschränkte Skalierbarkeit
- Warten auf „langsamsten“ Knoten bzw. Probleme bei Knotenausfall

Lazy Replication

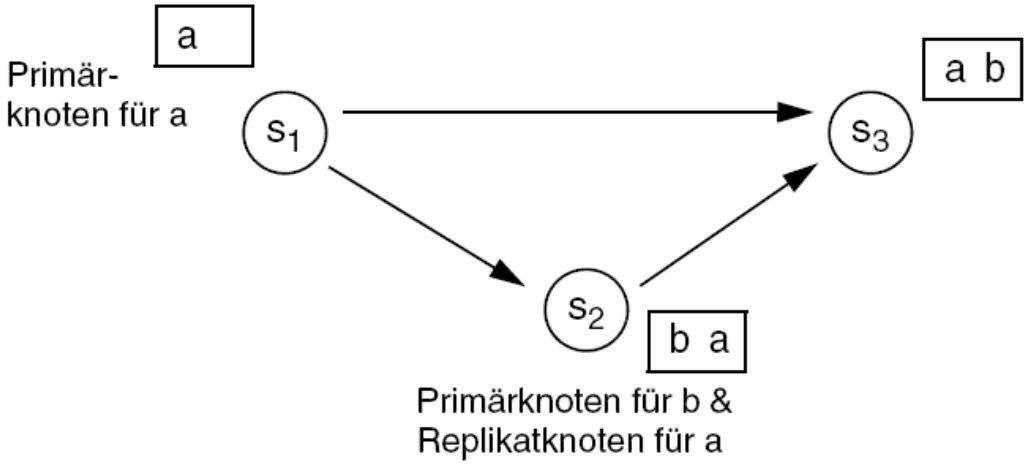
- Annahme: Wenn ein Update erfolgreich an einem Knoten war, kann das Update auch an anderen Knoten (erfolgreich) durchgeführt werden
  - Commit bereits nach Update auf erstem Knoten
  - Update-Propagierung durch separate Transaktion
- Kein 2PC notwendig
- Probleme mit Serialisierbarkeit, daher in der Praxis: Konfliktbehandlungsstrategien

Systemmodell

- Eigentümerprinzip
- Jedes Datenobjekt besitzt Primärknoten (Primärkopie), alle anderen Kopien: Sekundärkopie oder Replikat
- Jede Transaktion hat Ursprungsknoten
  - kann dort alle Datenobjekte lesen
  - Updates nur auf Primärkopien des Ursprungsknotens
- Weitere Annahmen: 2PL-Protokoll, zuverlässiges Netzwerk, Nachrichtenverarbeitung in FIFO-Reihenfolge

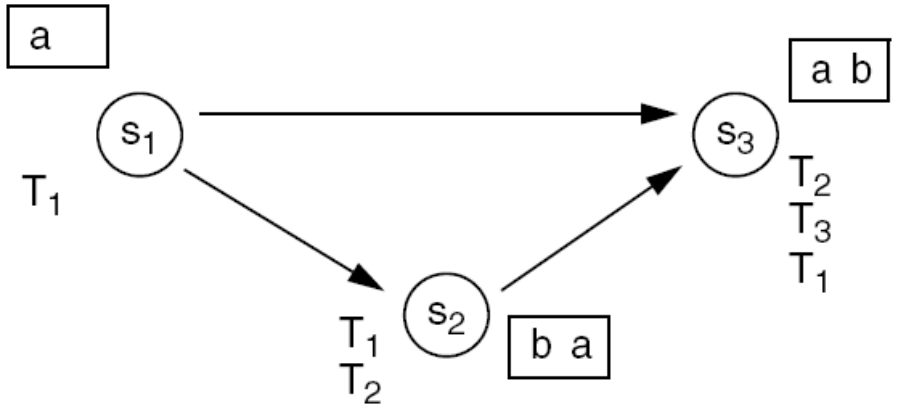
Copy Graph

- Knoten (Server), gerichtete Kante von s_i nach s_j , wenn Primärkopie an s_i und Replikat davon an s_j



Beispiel für nichtserialisierbaren Ausführungsplan

- $T_1@s_1$: w(a)
- $T_2@s_2$: r(a) w(b)
- $T_3@s_3$: r(a) r(b)
- Lazy Propagation
  - $@s_2$: $w_1(a)$ $r_2(a)$ $w_2(b)$
  - $@s_3$: $w_2(b)$ $r_3(a)$ $r_3(b)$ $w_1(a)$

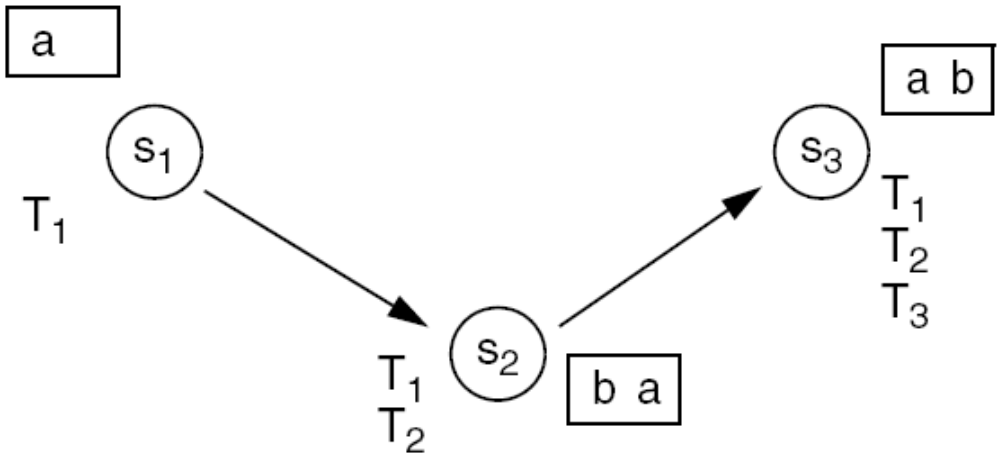


--> nicht serialisierbar!

Lösungsansatz:

DAG(WT)-Protokoll (DAG without Timestamps)

- Voraussetzung: azyklischer Copy Graph
  - vorhandene Zyklen ggf. durch Eliminierung einer Backedge auflösen (Backedges: Kanten, deren Entfernen den Graph azyklisch macht)
- Konstruktion eines Baumes aus Copy Graph
  - Primäre TA an s_i und Propagierung zu Nachfahren/Kindern s_i im Baum (Sekundär-TAs)
  - Commit-Reihenfolge: Reihenfolge des Eintreffens



Szenario

- Update von a: $s_1 \rightarrow s_2 \rightarrow s_3$
- erfolgt Update-Propagierung $w_1(a)$ vor $w_2(b)$ an s_2, so muss dies auch an s_3 gelten
  → Reihenfolge an s_3: $T_1T_2T_3$

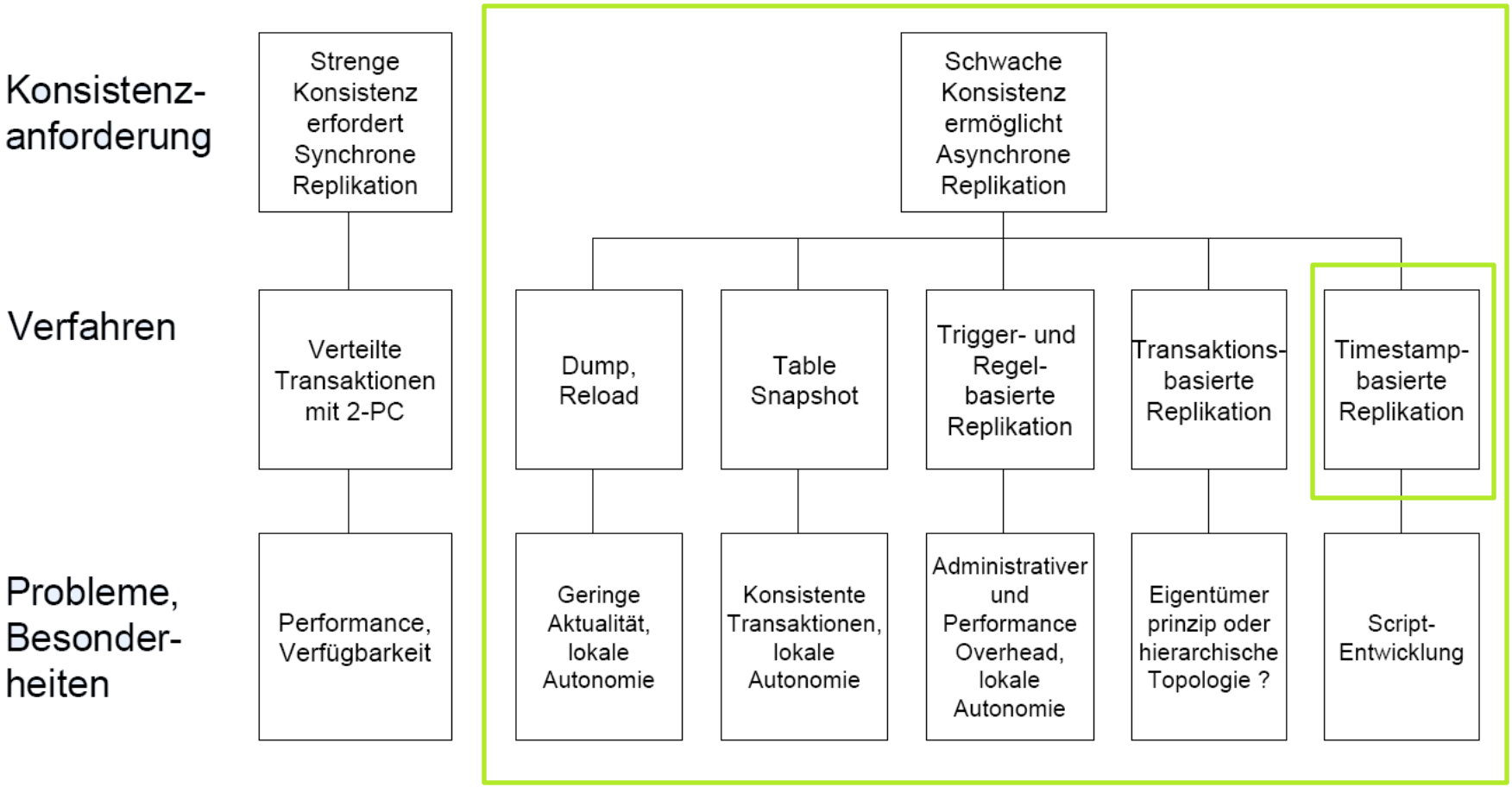
Probleme des DAG(WT)-Protokolls

- Propagierung der Updates entlang der Kanten des Baumes (kein Graph mehr)
- Routing über irrelevante Knoten, d.h. Knoten, die keine Replikate verwalten
  → kein Update
- Folge: Nachrichtenoverhead, Verzögerungen

Lösungsansatz: DAG(T)-Protokoll

- Ordnung der sekundären TAs durch Zeitstempel der primären TA (Commit-Zeitpunkt)
- Ausführung der TAs an jedem Knoten entsprechend der Zeitstempel

Strenge vs. Schwache Konsistenz ➔ Synchrone vs. Asynchrone Replikation



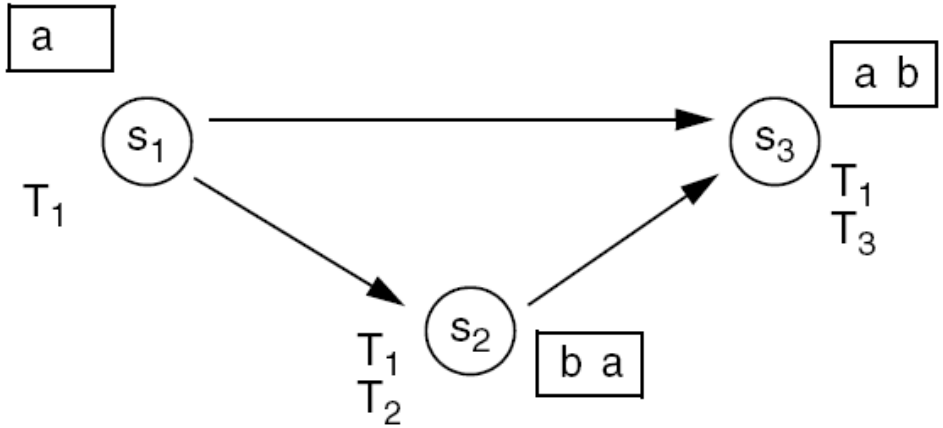
Zeitstempel für Lazy Update

- Azyklischer Copy Graph und totale Ordnung der Knoten, z.B. $s_1 < s_2 < ... < s_m$
- Jeder Knoten: lokaler Zeitstempelzähler LTS_i
  - Anzahl der primären Transaktionen, die an s_i committed haben
  - korrespondierendes Tupel (s_i, LTS_i)
- Zeitstempel $TS(s_i)$ eines Knoten s_i: Vektor von Tupeln
  - Tupel von s_i, d.h. (s_i, LTS_i)
  - Tupel der Vorfahren von s_i im Copy Graph sortiert nach Knotenordnung
- Zeitstempel $TS(T_i)$ einer Transaktion T_i
  - Zeitstempel an dem Knoten, an dem die primäre Transaktion Commit ausgeführt hat

Zeitstempel für serialisierbares Lazy-Update

Beispiel

- TS_1: <(s_1, 1)>
  TS_2: <(s_1, 1), (s_2, 1)>
  TS_3: <(s_1, 1), (s_3, 1)>



Lexikographische Ordnung

- $TS_1 < TS_2$ wenn gilt
  - TS_1 ist Präfix von TS_2
  - $TS_1 = X(s_i, LTS_i)Y$, $TS_2 = X(s_j, LTS_j)Z$
    1. $s_i > s_j$ oder (kürzere Wege)
    2. $s_i = s_j$ und $LTS_i < LTS_j$

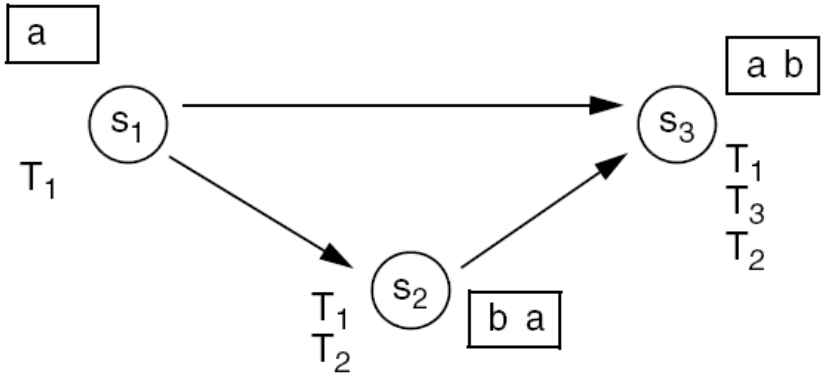
Beispiele

- (s_1, 1) < (s_1, 1), (s_2, 1)
- (s_1, 1), (s_3, 1) < (s_1, 1), (s_2, 1)
- (s_1, 1), (s_2, 1) < (s_1, 1), (s_2, 2)

Zeitstempel für serialisierbares Lazy-Update

Beispiel

- @s_2: T_1T_2
  ➔ $TS(s_2) := (s_1,1),(s_2,1)$
- @s_3: $T_1T_3T_2$
  ➔ $TS(T_3) < TS(T_2)$
  da: $(s_1, 1),(s_3, 1) < (s_1, 1),(s_2, 1)$



- Primäre Transaktion T_3 könnte direkt Commit ausführen
- Ordnung der Zeitstempel soll Commit-Reihenfolge entsprechen

Notwendige Datenstrukturen pro Knoten

- Zeitstempel-Vektor: Zeitstempel der letzten abgeschlossenen Sekundärtransaktion + Tupel des Knotens
- Warteschlange für jeden Vorgängerknoten

Zeitstempel für serialisierbares Lazy-Update

Ablauf für Commit bei Primärtransaktion T_i am Knoten s_i

- 1. Inkrementieren des lokalen Zeitstempelzählers LTS_i im korrespondierenden Tupel $TS(s_i)$ des Zeitstempel-Vektors
- 2. $TS(T_i) := TS(s_i)$ /* Zeitstempel für Primärtransaktion und alle ihre Sekundärtransaktionen */
- 3. Weiterleiten der Sekundärtransaktionen von T_i an relevante Kinder von s_i
  - Liste der Schreiboperationen + $TS(T_i)$

Ablauf für Sekundärtransaktionen

- Annahme: nur eine Sekundär-TA gleichzeitig
  - 1. Wähle Transaktion mit minimalen Zeitstempel aus allen Warteschlangen
  - 2. bei Commit von T_i : $TS(T_i) := TS(T_i);<(s_i, LTS_i)>$
- mind. eine Sekundärtransaktion in jeder Warteschlange vor Auswahl
  → Garantie für Serialisierbarkeit

Vorbetrachtung

- Anwendungsmöglichkeiten
- Vor- und Nachteile
- Zielkonflikte
- Synchronisation

Aktualisierungsstrategien

- Klassifikation der Aktualisierungsstrategien
- ROWA / ROWAA
- Primary Copy
- Votierungsverfahren

Asynchrone Aktualisierungsstrategien

- Klassifikation synchroner und asynchroner Verfahren
- Need-to-Know-Prinzip
- Snapshot Refresh
- Lazy Update