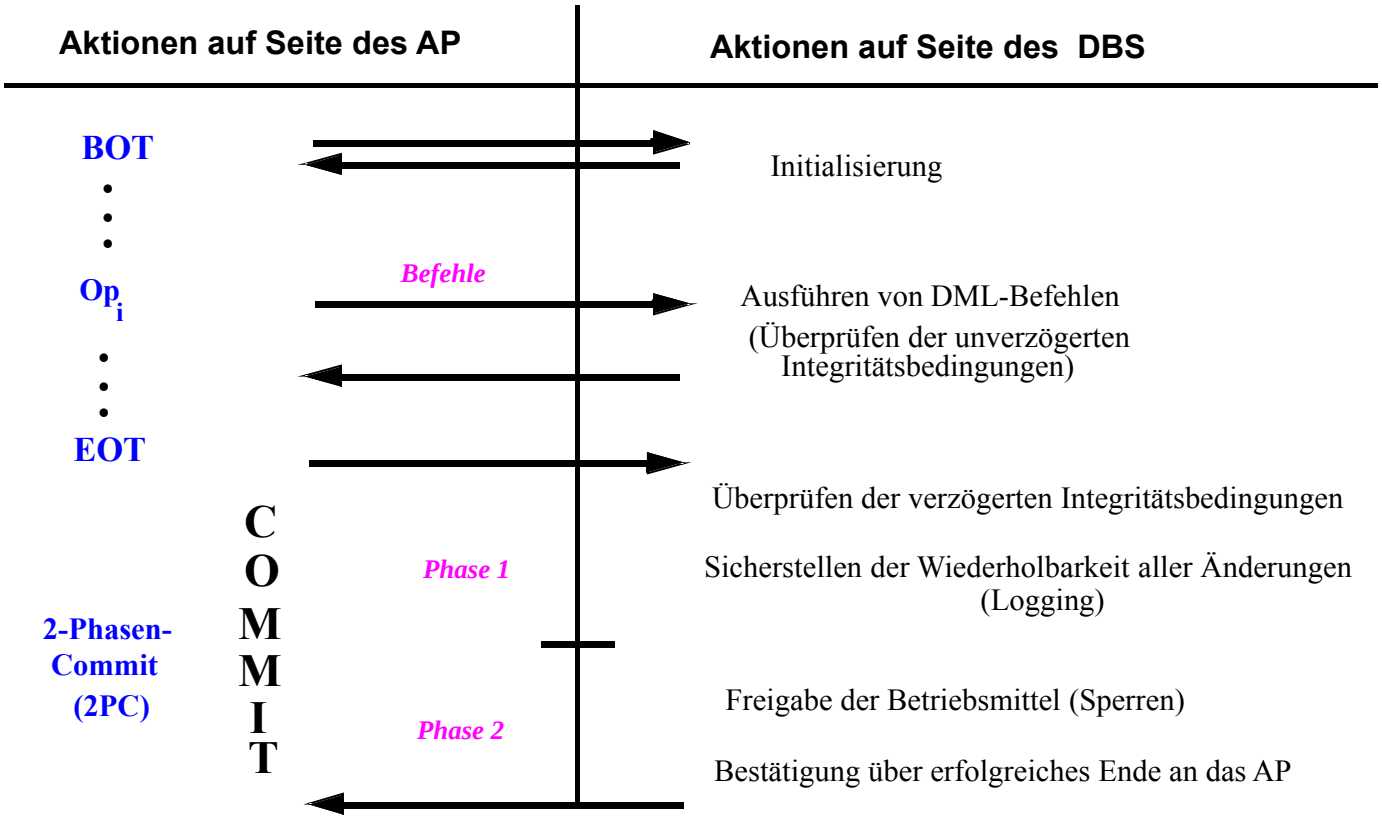
6. Verteilte Transaktionsverwaltung

- Einführung
- Synchronisation
  - Verfahrensüberblick, Sperrverfahren
  - Mehrversionen-Synchronisation
  - Deadlock-Behandlung
    - Timeout
    - Deadlock-Vermeidung: Wait/Die-, WoundWait-Verfahren
    - globale Deadlock-Erkennung
- Commit-Protokolle: Anforderungen
- Basis-Protokoll: 2-Phasen-Commit
  - Ablauf
  - Fehlerbehandlung
  - Lineares 2PC
  - Hierarchisches 2PC, XA-Protokoll
- 1-Phasen-Commit, 3-Phasen-Commit, Paxos-Commit

Transaktionsverarbeitung in zentralisierten DBS

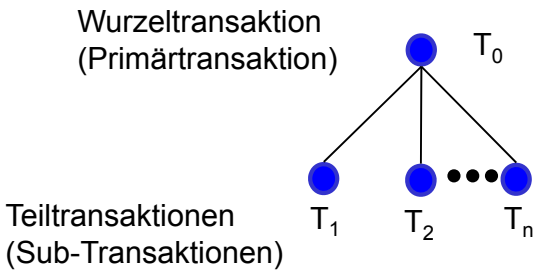


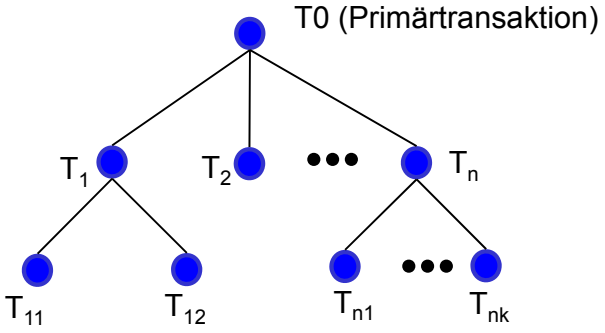
Transaktionsverwaltung in VDBS

- ACID-Eigenschaften von Transaktionen auch im verteilten Fall zu garantieren: Atomarität, Konsistenz, Isolation, Dauerhaftigkeit
- Logging und Recovery
  - wesentliche Neuerung: globales Commit-Protokoll (Consensus-Verfahren)
  - Robustheit gegenüber partiellen Fehlern, insbesondere Kommunikationsfehlern (Netzwerkpartitionierungen u. ä.)
- Integritätssicherung auf verteilten Daten
  - Überwachung zB im Rahmen eines erweiterten Commit-Protokolls
- Synchronisation
  - Wahrung der globalen Serialisierbarkeit
  - rechnerübergreifende Abhängigkeiten (globale Deadlocks u. ä.)
- Replikation
  - Wahrung von Replikationstransparenz
  - Optimierung von Leistung und Verfügbarkeit

Transaktionsstruktur

- Transaktionsaufbau: BOT, OP_1, OP_2, ..., OP_n, COMMIT
- Transaktionsbaum: Kontrollstruktur in einer verteilten Umgebung
  - Transaktionsbaum repräsentiert Aufrufbeziehungen
  - 1-stufige oder geschachtelte Teiltransaktionen
  - isoliertes Rücksetzen von Sub-Transaktionen wird i.a. nicht unterstützt: Abbruch einer Teiltransaktion führt zum Abbruch der Gesamttransaktion





Synchronisation

- Mehrbenutzerbetrieb führt ohne Synchronisation zu Anomalien:
  - Lost Updates
  - Dirty Reads
  - Non-repeatable Reads
  - Phantome

- Korrektheitskriterium der (globalen) Serialisierbarkeit:
  gleichzeitige (und verteilte) Ausführung mehrerer Transaktionen ist äquivalent zu wenigstens einer seriellen Ausführung derselben Transaktionen

Synchronisation (2)

- Sperrprotokolle garantieren Serialisierbarkeit
  - vor jedem Objektzugriff muß Sperre mit ausreichendem Modus angefordert werden
  - gesetzte Sperren anderer Transaktionen sind zu beachten
  - am Transaktionsende (2. Commit-Phase) werden alle Sperren freigegeben
- Einfachster Ansatz: RX - Sperrverfahren

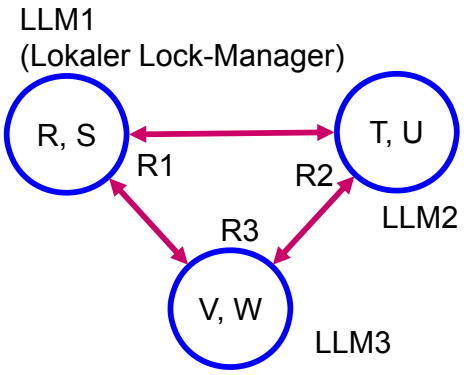
angeforderter Sperrmodus \ aktueller Sperrmodus	NL	R	X
R	+	+	-
X	+	-	-

NL: no lock,
R: read lock;
X: eXclusive lock

\+ Sperre wird gewährt
\- Sperrkonflikt

Synchronisation in VDBS

- Zentrale Verfahren problematisch
  - Knotenautonomie
  - Kommunikationsaufwand



- Verteilte Sperrverfahren
  - jeder Rechner verwaltet Sperren für lokale Daten
  - keine eigene Nachrichten für Sperranforderungen
  - Sperrfreigabe innerhalb des Commit-Protokolls
  - am weitesten verbreiteter Ansatz für VDBS
  - globale Deadlocks zu behandeln

Synchronisation in VDBS (2)

- Zeitmarkenverfahren (timestamp ordering)
  - Transaktionen T erhalten global eindeutige Zeitmarken ts(T) bei BOT, z.B. <lokale Uhrzeit, Rechner-ID>
  - alle Zugriffe müssen in Reihenfolge der BOT-Zeitmarken erfolgen, anderenfalls erfolgt Rücksetzung der zugreifenden Transaktion
- Zeitmarken an den Datenobjekten
  - Schreibzeitstempel (*write timestamp*) *wts* und Lesezeitstempel *(read timestamp*) *rts*: Transaktionszeitstempel des letzten Lesers / Änderers
  - Lesezugriff einer Transaktion *T* auf Objekt *x* scheitert, falls jüngere Transaktion O geändert hat, d.h. wts(x) > *ts(T)*
  - Schreibzugriff scheitert, falls *ts(T)* < *rts(x)* oder *ts(T)* < *wts(x)*
- Eigenschaften
  - lokale Zeitstempelverwaltung und Konflikttests
  - keine Deadlocks
  - jedoch viele Rücksetzungen

Synchronisation in VDBS (3)

- Optimistische Synchronisation
  - keine Sperren, sondern Konflikttest am Transaktionsende (Validierung)
  - potentiell viele Rücksetzungen
  - zentrale Validierung erfordert wenig Kommunikationsaufwand

- Zentrale Validierung
  - Prüfung, ob Objekte im *Read-Set* (RS) einer Transaktion T_j noch aktuell sind
  - erfolgreich validierte Änderungstransaktionen erhalten Commit-Zähler TNC (monoton wachsend), der für Objekte im *Write-Set* (WS) hinterlegt werden

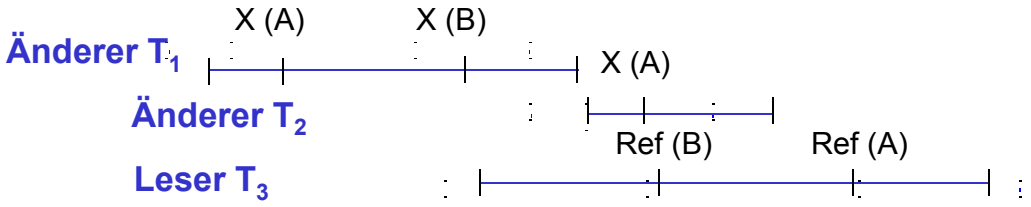
```
VALID := true;
FOR(all r in RS(Tj)DO
   IF rts(r,Tj) < wts(r)
   THEN VALID := false
END;
IF VALID THEN DO
  TNC := TNC+1;
  FOR(all w in WS(Tj)DO
        wts(w):=TNC END;
  Schreibphase für Tj
END;
ELSE  ROLLBACK (Tj);
```

- Verteilte Validierung im Rahmen des Commit-Protokolls möglich
  - Nutzung von (synchronisierten) EOT-Zeitstempeln pro Transaktion

Mehrversionen-Synchronisation

- Prinzip
  - jede Änderung erzeugt neue Objektversion
  - Lesezugriffe sehen den beim BOT gültigen DB-Zustand

  -> keine Synchronisation mehr für Lese-Transaktionen (dennoch Serialisierbarkeit)
  - erheblich weniger Synchronisationskonflikte
  - Realisierung über Commit-Zähler für Änderungen
  - anwendbar für Sperrverfahren und optimistische Synchronisation



- Variation *Snapshot Isolation* (i.a. nicht serialisierbar)
  - auch Änderer lesen Daten vom BOT-Zustand; quasi-optimistische Synchronisation (keine WS-Überlappung mit anderen Transaktionen)
- Nutzung in rel. DBS (Oracle, PostgresSQL, …) und NoSQL Stores

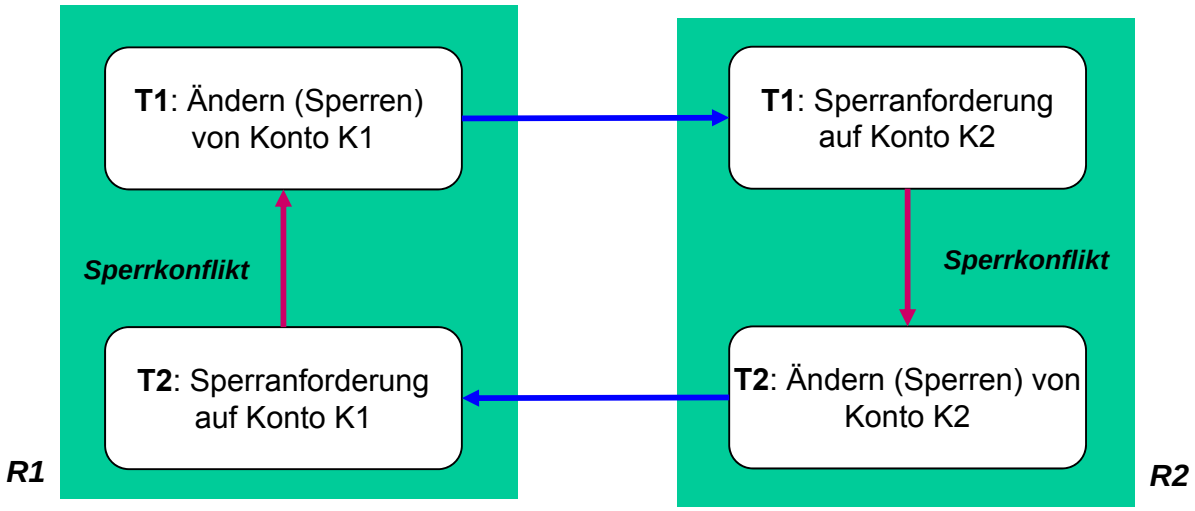
Mehrversionen-Synchronisation in VDBS

- Zentrale Kontrolle (u.a. in HBase)
  - zentrale Vergabe von Commit- und BOT-Zeitstempel
  - zentrale Validierung von Änderungstransaktionen

- Verteilte Kontrolle (u.a. in Google Spanner)
  - verteilte Vergabe von BOT- und Commit-Zeitstempeln erfordert eng synchroniserte lokale Uhren (Spanner: max 10 ms über Atomuhren + GPS)
  - Optimistische oder pessimistische Synchronisation von Änderungstransaktionen
  - Spanner: Sperren für Änderungstransaktionen nur während Commit-Protokoll, Leser lesen zu BOT gültige Versionen

Deadlock-Behandlung

- Sperrverfahren erfordern Deadlock-Behandlung
- Deadlock: zyklische Wartebeziehung zwischen zwei oder mehr Transaktionen
  - Deadlock-Behandlung in DBS erfordert Rücksetzung von Transaktionen
- Beispiel (Überweisungen zwischen Konten K1 und K2)



Deadlock-Behandlung: Timeout-Verfahren

- Festsetzen einer maximalen Wartezeit auf eine Sperre (Timeout)
- Nach Überschreiten des Timeout wird wartende Transaktion zurückgesetzt
- Vorteile
  - jeder Deadlock wird irgendwann aufgelöst
  - geringer Implementierungsaufwand
  - keine zusätzliche Nachrichten für Deadlock-Behandlung
  - kann auch bei heterogenen DBS genutzt werden
- Nachteile
  - unnötige Rücksetzungen (Erreichen des Timeout auch ohne Deadlock)
  - Wahl des Timeouts (viele Rücksetzungen vs. später Auflösung von Deadlocks)
- Nutzung in den meisten VDBS bzw. verteilten Transaktionssystemen

Deadlock-Behandlung: Weitere Lösungsmöglichkeiten

Deadlock-Vermeidung (Avoidance)

- Zuweisung einer eindeutigen ***Transaktionszeitmarke*** bei BOT
- Entscheidung bei einem Sperrkonflikt, ob Warten zugelassen wird oder eine Transaktionsrücksetzung erfolgt
- kein Wartezyklus (Deadlock) möglich, falls stets nur ältere Transaktionen auf jüngere warten dürfen (Bsp.: Wait/Die) bzw. nur jüngere auf ältere Transaktion warten (Bsp.: Wound/Wait-Verfahren)
- Behandlung globaler Deadlocks ohne Kommunikation!

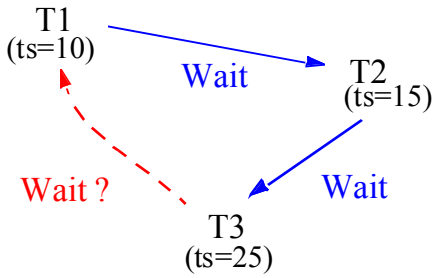
Deadlock-Erkennung (Detection)

- Führen eines globalen Wartegraphen (-> Kommunikationsbedarf)
- Zyklensuche zur Erkennung von Deadlocks
- potentiell geringste Anzahl von Rücksetzungen

Deadlock-Vermeidung: Wait/Die

- Zuweisung einer eindeutigen *Transaktionszeitmarke* ts (T) bei Beginn jeder Transaktion T
- WAIT/DIE-Verfahren
  - anfordernde Transaktion Ti wird zurückgesetzt, falls sie in Sperrkonflikt mit älterer Transaktion gerät
  - Ti wartet bei einem Sperrkonflikt, falls sie älter ist als Sperrbesitzer
  - kein Zyklus möglich, da nur ältere auf jüngere Transaktionen warten

```
Ti fordert Sperre, Konflikt mit Tj:
   if ts (Ti) < ts (Tj) {Ti älter als Tj}
   then WAIT (Ti)
   else Rollback (Ti) {'Die'}
```



- Bewertung
  - Behandlung globaler Dealocks ohne Kommunikation
  - unnötige Rückstzungen auch ohne Deadlock

Deadlock-Vermeidung: Wound/Wait

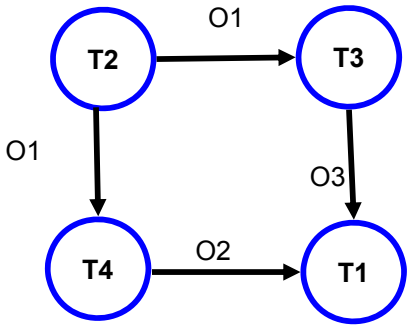
- preemptiver Ansatz: Sperrbesitzer wird zurückgesetzt, wenn er bei einem Sperrkonflikt jünger als anfordernde Transaktion ist
- kein Zyklus möglich, da nur jüngere auf ältere Transaktionen warten

```
Ti fordert Sperre, Konflikt mit Tj:
   if ts (Ti) < ts (Tj) {Ti älter als Tj}
   then ABORT (Tj) {'Wound'}
   else Wait (Ti)
```

- Bewertung ähnlich zu Wait/Die
  - noch stärkere Bevorzugung älterer Transaktionen positiv zu bewerten

Deadlock-Erkennung

- Explizites Führen eines ***Wartegraphen*** (wait-for graph)
  - Knoten: laufende Transaktionen
  - gerichtete Kanten: Wartebeziehungen aufgrund von Sperrkonflikten
  - Zyklus kennzeichnet Deadlock
- Zyklensuche zur Erkennung von Verklemmungen
  - bei jedem Sperrkonflikt bzw.
  - verzögert (z. B. über Timeout gesteuert)
- Deadlock-Auflösung durch Zurücksetzen einer oder mehrerer Transaktionen, deren Wegfall Zyklus auflöst
  - z. B. Verursacher des Deadlocks
  - "billigste" Opfer



Deadlock-Erkennung in Verteilten DBS

- Nachrichtenaustausch zur Erstellung des Wartegraphen
- Zentrale Deadlock-Erkennung
  - ausgezeichneter Knoten verwaltet globalen Wartegraphen
  - hoher Kommunikationsaufwand (Übertragung aller neuen / wegfallenden Wartebeziehungen)
  - Einschränkung der Knotenautonomie
- Verteilte Deadlock-Erkennung
  - verteilte Verwaltung eines globalen Wartegraphen
  - korrektes Verfahren schwierig zu realisieren:
    - Nachrichtenverzögerungen
    - Empfangs- ≠ Sendereihenfolge
    - Rechnerausfälle
  - oftmals
    - doppelte Erkennung von Deadlocks
    - "falsche" Deadlocks

Verteilte Deadlock-Erkennung in R*

- "Deadlock Detector" pro Rechner, der periodisch lokalen Wartegraphen auf Zyklen durchsucht
  - spezieller Knoten "EXTERNAL" im Wartegraph zur Darstellung von Wartebeziehungen zu Sub-Transaktionen auf anderen Rechnern
  - Zyklus mit EXT-Knoten kennzeichnet potentiellen globalen Deadlock
- Weitergabe der Zyklusinformation an andere Rechner, um globalen Deadlock ggf. zu erkennen
- Ausgangsbeispiel

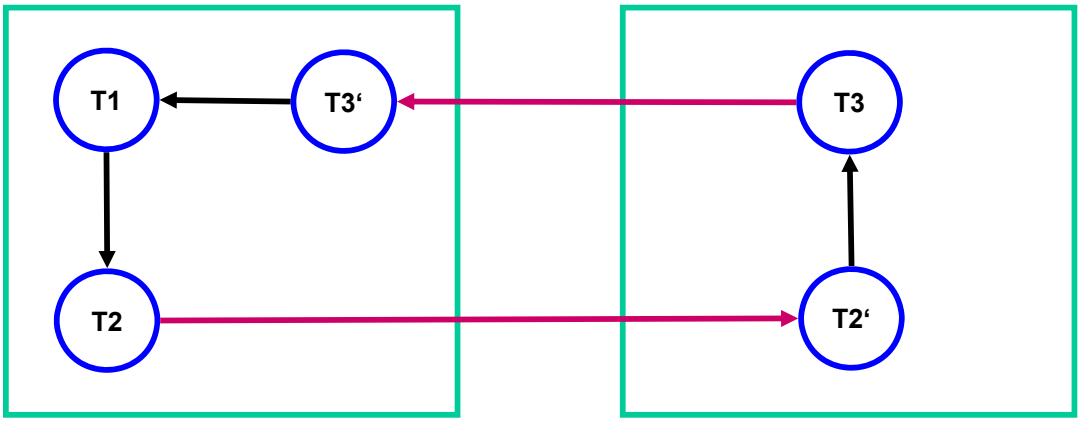
R1:

R2:

Deadlock-Erkennung in R* (2)

- Kooperation mit anderen Rechnern:
  1. Empfange Deadlock-Information anderer Rechner
  2. Erweitere damit lokalen Wartegraphen
  3. löse lokale/vollständige Zyklen durch Bestimmung und Rücksetzung eines "Opfers" auf
  4. für globale Zyklen sende lineare Darstellung:
     EXT -> T_1 -> T_2 -> ... -> T_n -> EXT
     an Rechner, auf den T_n wartet (falls $T_1 > T_n$)
- Bewertung
  - **max. N (N-1)/2 Nachrichten** zur Erkennung eines globalen Deadlocks bei N beteiligten Rechnern
  - Erkennung falscher Deadlocks möglich

Beispiel



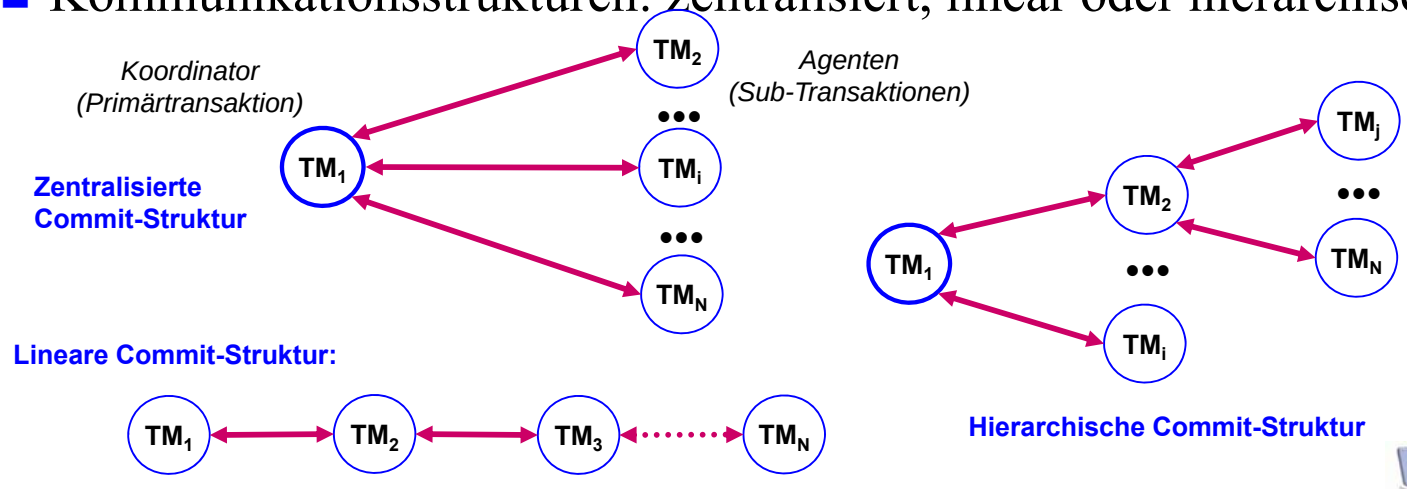
Commit-Protokolle

- Sicherstellen der ***Atomizität*** verteilter Transaktionen durch rechnerüber-greifendes Mehrphasen-Commit-Protokoll
- Anforderungen an geeignetes Commit-Protokoll:
  - Korrektheit
  - Geringer Aufwand (#Nachrichten, #Log-Writes)
  - Geringe Antwortzeitverlängerung
  - Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern
  - Knotenautonomie: jeder an einer verteilten Transaktionsausführung beteiligte Rechner soll möglichst lange das Recht des einseitigen Transaktionsabbruchs (*unilateral abort*) haben

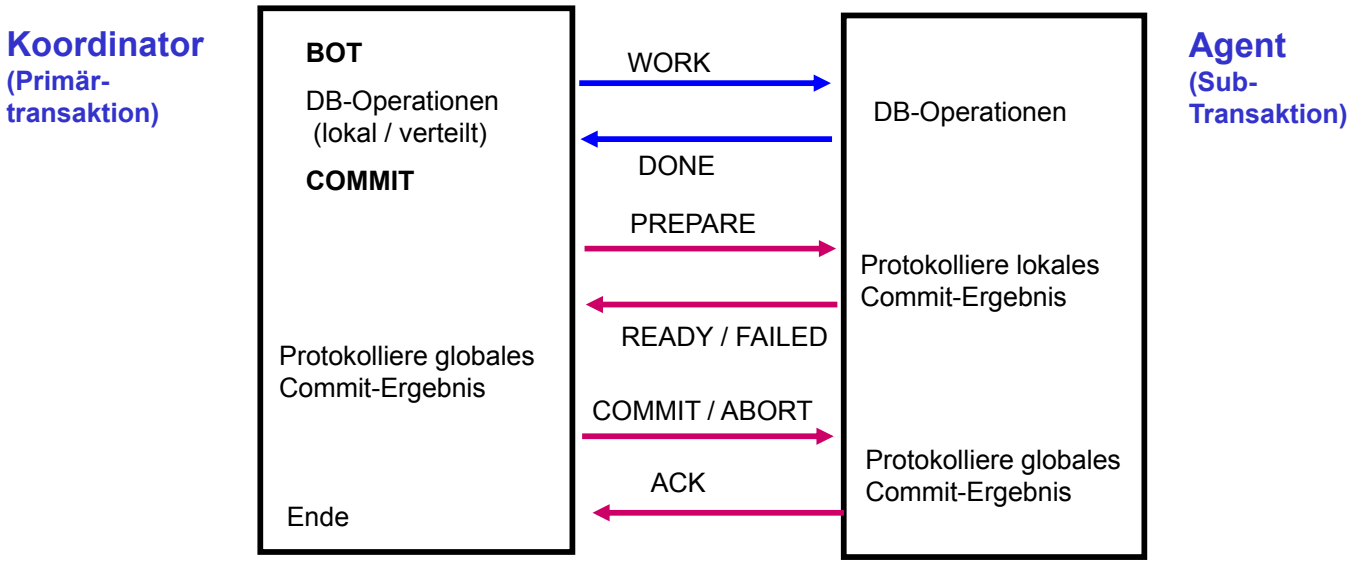
"Nicht-Fehler-Fall" ist zu optimieren

Commit-Protokolle (2)

- Ausführung des Commit-Protokolls erfolgt durch *Transaktions-Manager (TM)* an jedem Knoten (1 Koordinator + N-1 Agenten)
- Wesentliche Alternativen
  - 2-Phasen-Commit (zentral, linear, hierarchisch)
  - 1-Phasen-Commit
  - 3-Phasen-Commit
- Kommunikationsstrukturen: zentralisiert, linear oder hierarchisch

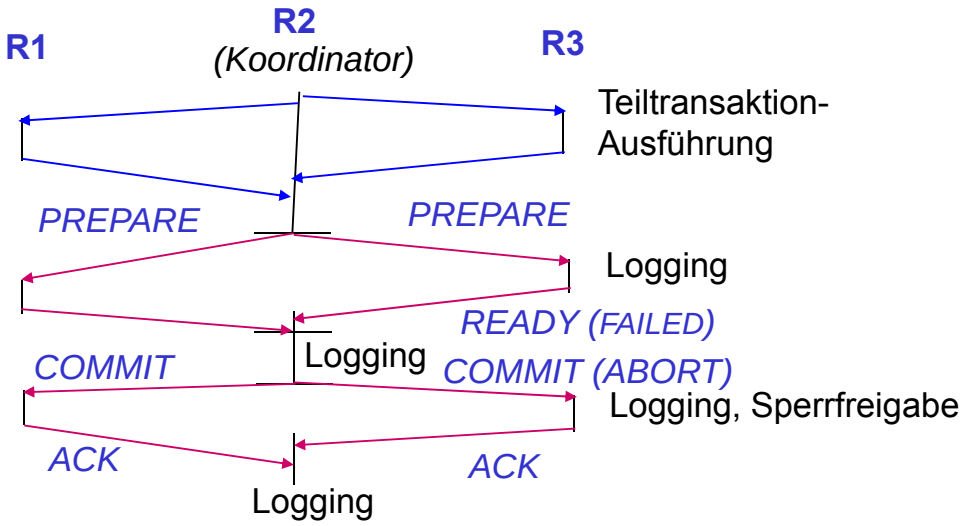


Zentralisiertes 2-Phasen-Commit (N=2)



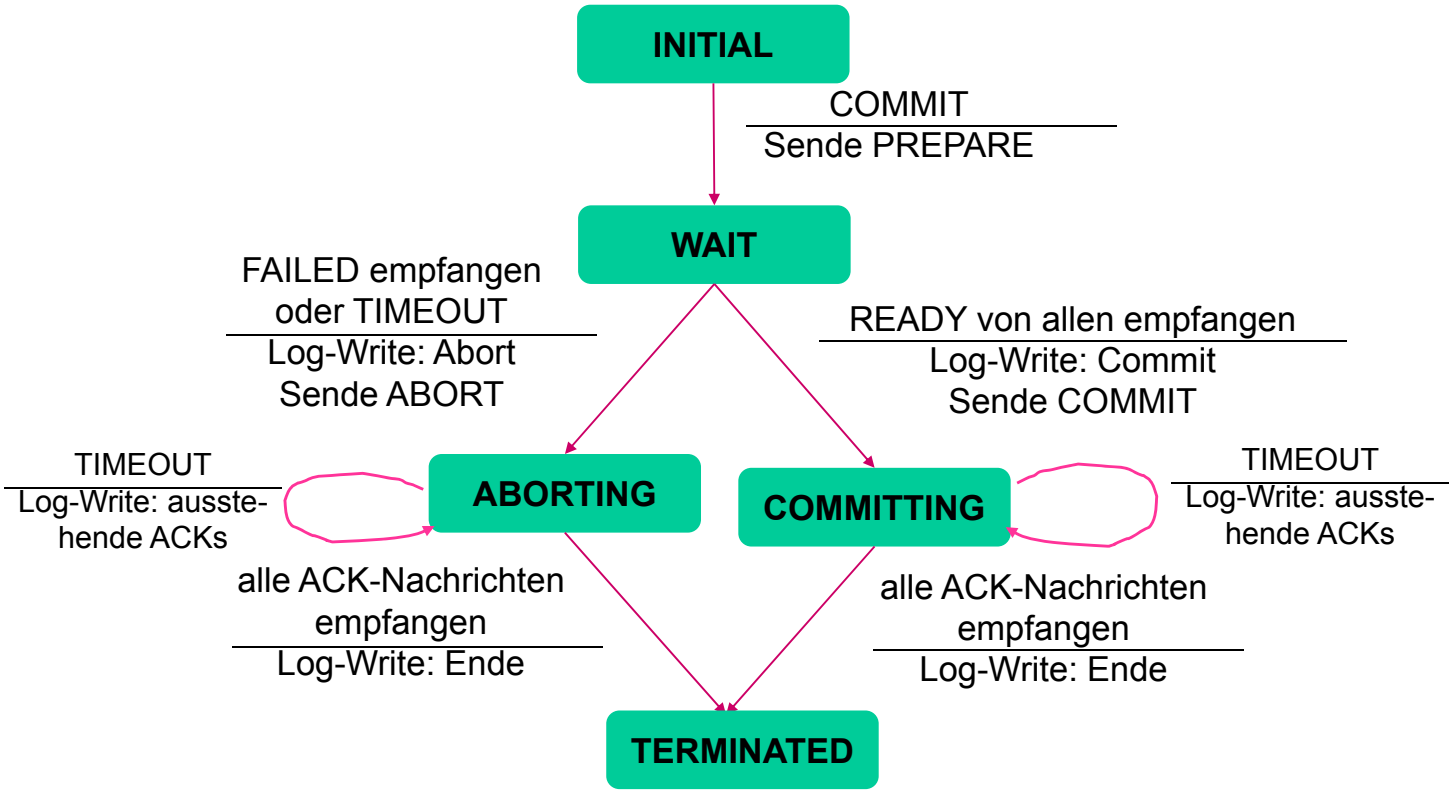
- Aufwand
  - Erfolgreicher Ausgang: 4 Nachrichten, 4 Log-Writes
  - ABORT-Nachrichten gehen nur an Teiltransaktionen, die nicht mit FAILED gestimmt haben
- Kernproblem Koordinatorausfall => ggf. lange Blockierung

Zentralisiertes 2-Phasen-Commit (N=3)



- Basisverfahren:
  - 4 (N-1) Nachrichten (N = Anzahl der beteiligten Knoten)
  - 2 N Log-Writes
- Optimierung für lesende Sub-Transaktionen (Anzahl M)
  - 4 (N-1) - 2M Nachrichten für M < N, 2 (N-1) für M=N
  - 2 N - M Log-Writes

Zustandsgraph Koordinator (2PC)



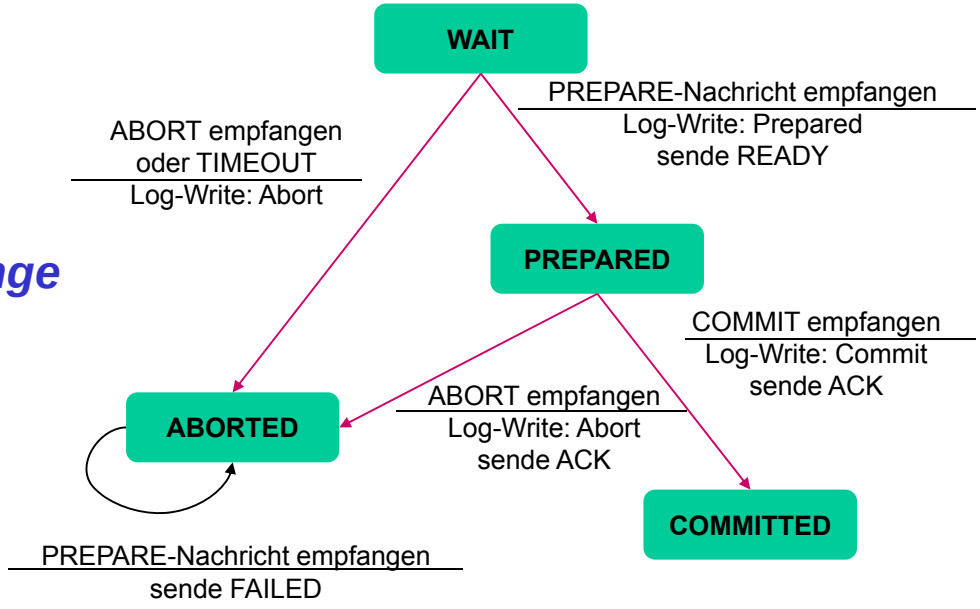
2PC: Fehlerbehandlung

- Timeout-Bedingungen Koordinator:
  - WAIT => setze Transaktion zurück; verschicke ABORT-Nachr.
  - ABORTING, COMMITTING => vermerke Agenten, für die ACK noch aussteht

- Ausfall des Koordinatorknotens:
  - Log-Zustand TERMINATED:
    - UNDO bzw. REDO-Recovery, je nach Transaktionsausgang
    - keine "offene" Teiltransaktionen möglich
  - Log-Zustand ABORTING:
    - UNDO-Recovery
    - ABORT-Nachricht an Rechner, von denen ACK noch aussteht
  - Log-Zustand COMMITTING:
    - REDO-Recovery
    - COMMIT-Nachricht an Rechner, von denen ACK noch aussteht
  - Sonst: UNDO-Recovery

2PC: Fehlerbehandlung (2)

Zustandsübergänge Agent



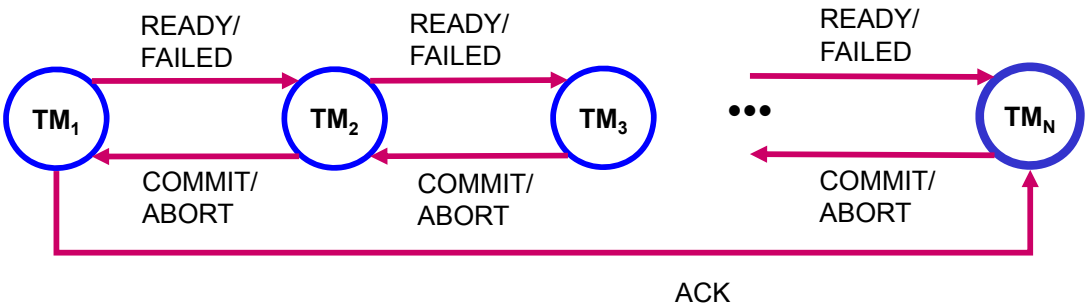
- Timeout-Bedingungen für Agenten:
  - WAIT => setze Teiltransaktion zurück (unilateral ABORT)
  - *PREPARED => erfrage Transaktionsausgang bei Koordinator (bzw. anderen Rechnern)*

2PC: Fehlerbehandlung (3)

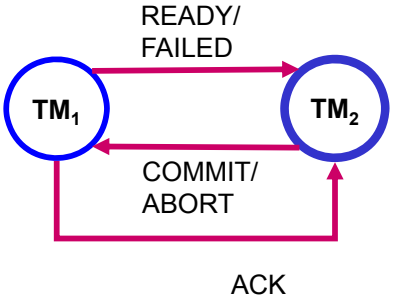
- Rechnerausfall für Agenten:
  - Log-Zustand COMMITTED: REDO-Recovery
  - Log-Zustand ABORTED bzw. kein 2PC-Log-Satz vorhanden: UNDO-Recovery
  - *Log-Zustand PREPARED: Anfrage an Koordinator-Knoten, wie Transaktion beendet wurde (Koordinator hält Information, da noch kein ACK erfolgte)*

Lineares 2PC

- Sequentielle Commit-Behandlung, dafür Halbierung der Nachrichtenanzahl: (N-1) + N = 2N-1
- Transfer der Commit-Koordinierung: Koordinatorrolle geht auf letzten Agenten über *("Last Agent"-Optimierung)*

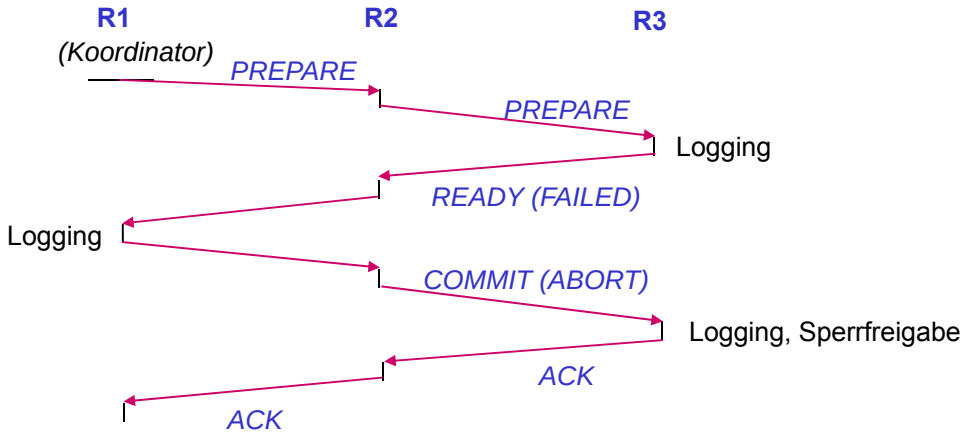


- Besonders vorteilhaft für N=2: **3 Nachrichten**



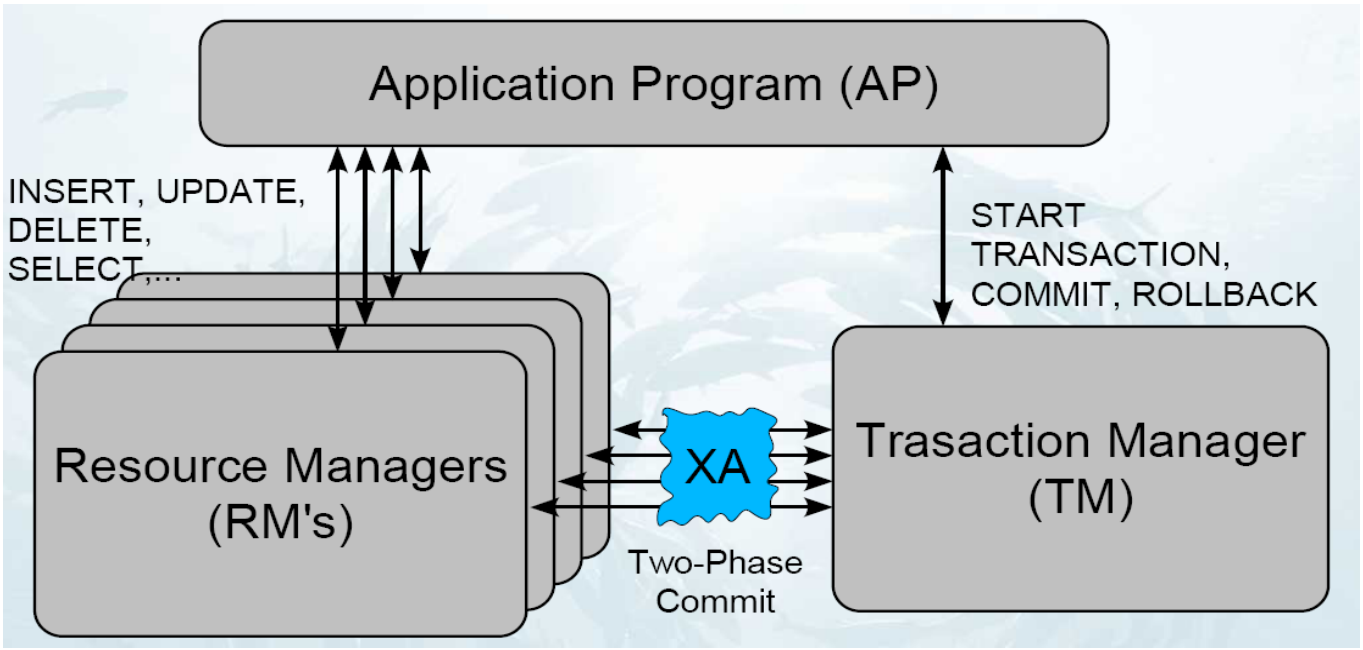
Hierarchisches 2PC

- allgemeineres Ausführungsmodell mit beliebiger Schachtelungstiefe
  - Nutzung im XA-Standard für verteilte Transaktionen
  - Antwortzeiterhöhung steigt mit Schachtelungstiefe
- Bekanntester Vertreter: *Presumed-Abort-Protokoll*
  - Optimierung für ABORT: keine ACK-Nachrichten und kein synchrones Logging
    - wenn keine Angaben im Log gefunden werden, wird per Default ABORT angenommen
  - Optimierung für lesende Teiltransaktionen:
    - kein Logging, Sperrfreigabe in Phase 1
    - Kommunikation für zweite Phase wird umgangen

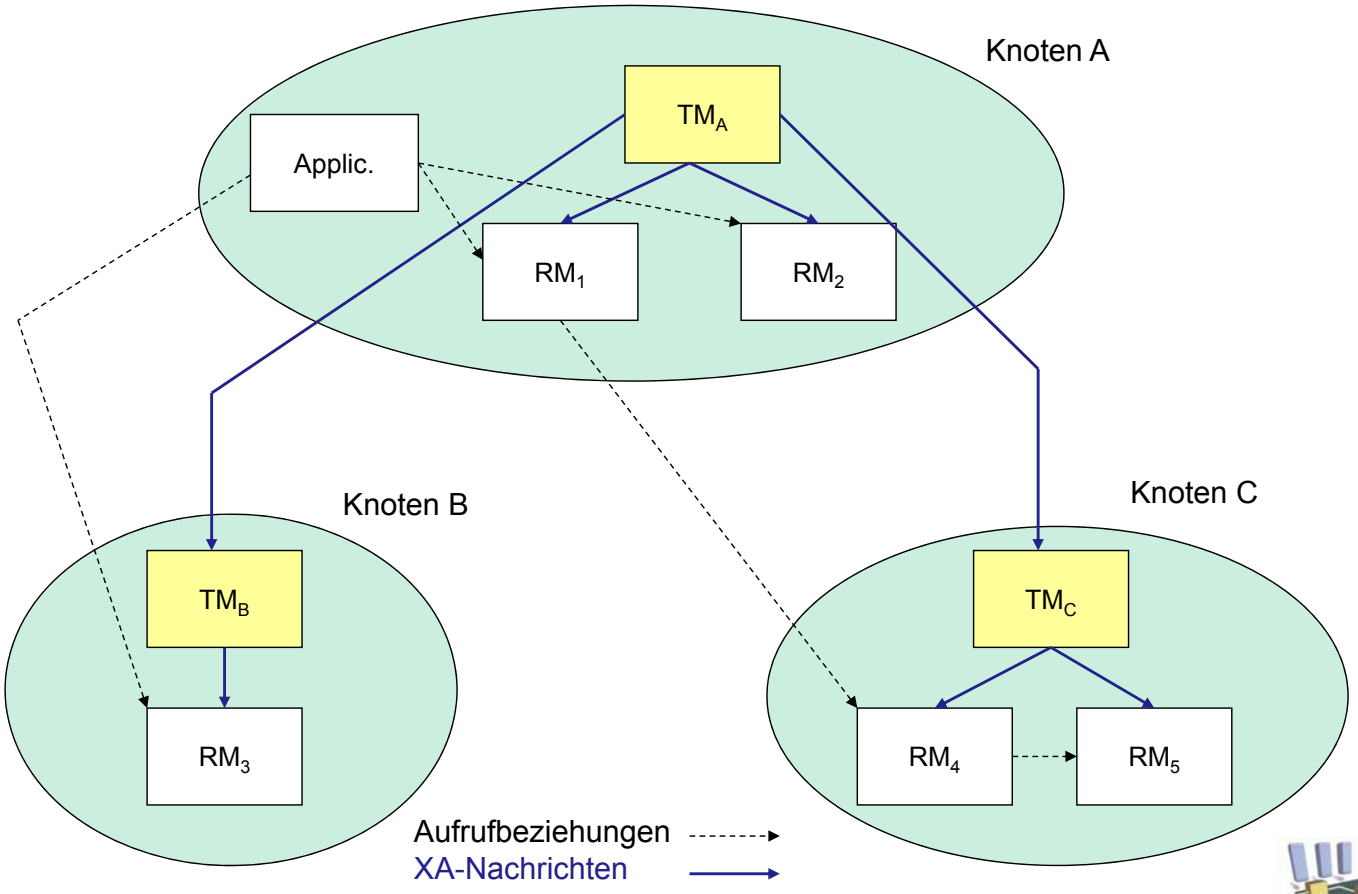


X/OPEN Modell für verteilte Transaktionen

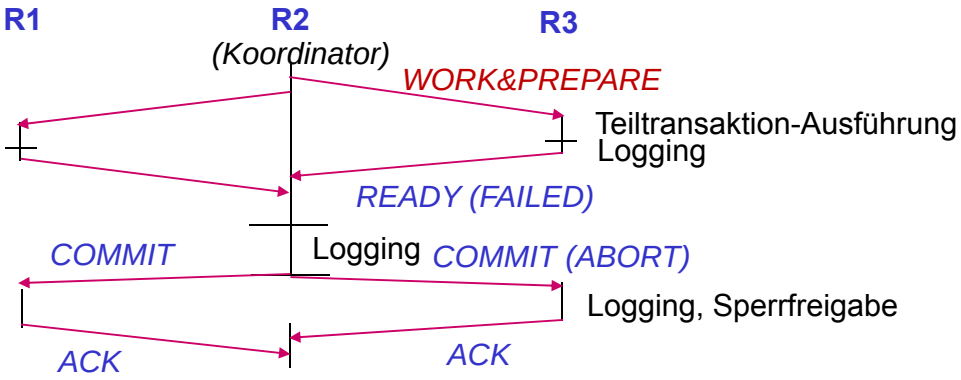
- TM: Teil von Betriebssystem oder Applikations-Server
- DBS wichtiger RM-Typ
  - XA-Unterstützung durch Oracle, DB2, SQLServer, MySQL, …



Verteilte Transaktionsausführung



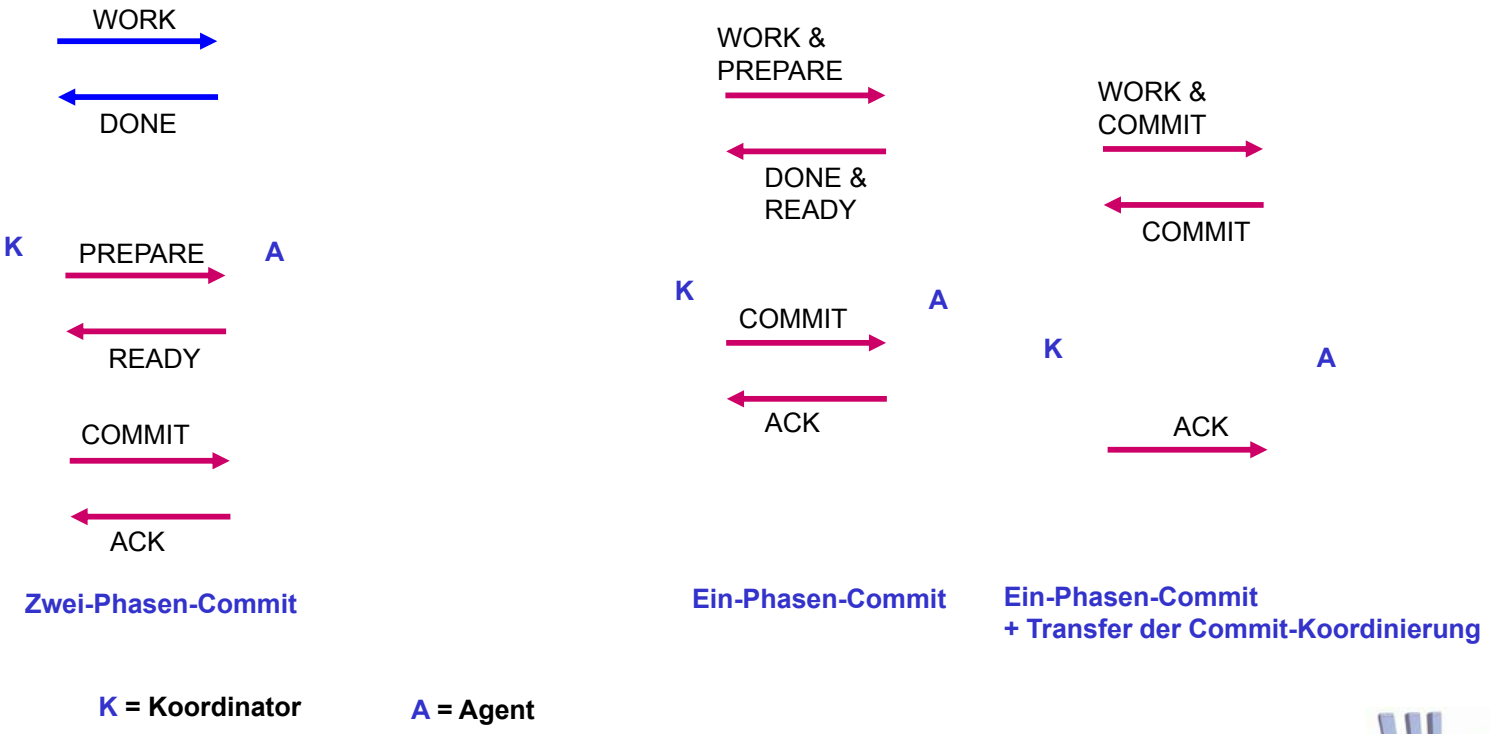
1-Phasen-Commit



- Teil-Transaktionen sichern ihre Änderungen bereits vor Rückgabe der Ergebnisse an Primärtransaktion
  - nach lokalem Commit am Koordinator steht Erfolg der Transaktion fest
- Einsparung von 2 Nachrichten pro Agent: 2 (N-1) Nachrichten
  - Besonders vorteilhaft für kurze (verteilte) Transaktionen
- Nachteile
  - starke Abhängigkeit vom Koordinator durch frühzeitigen Verzicht auf Unilateral Abort
  - mehrfaches Logging (READY) pro Transaktion und Knoten möglich

1-Phasen-Commit (2)

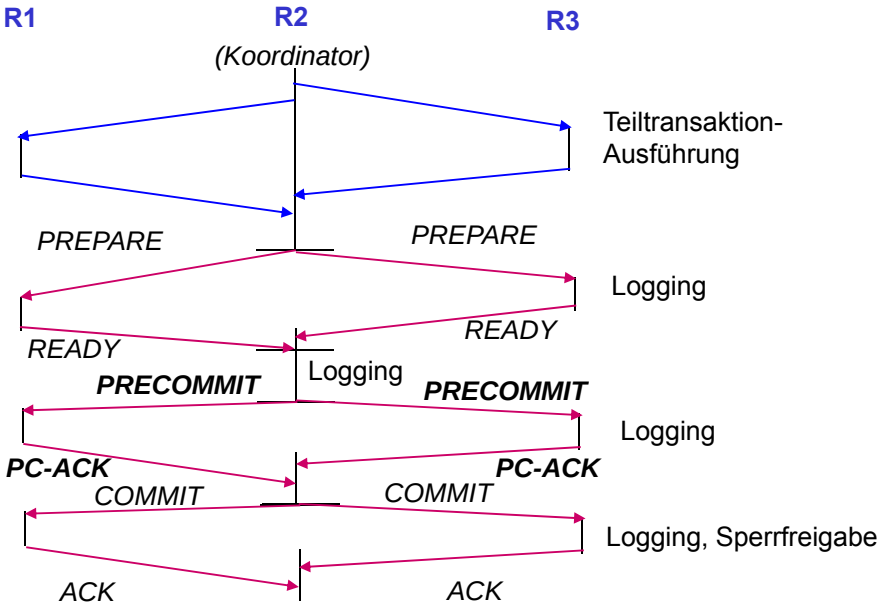
- Für N=2 kann weitere Nachricht durch Transfer der Commit-Koordinierung eingespart werden



K = Koordinator **A = Agent**

3-Phasen-Commit

- Nicht-blockierendes Verfahren
- Annahmen:
  - keine Netzwerkpartitionierung
  - höchstens F < N Rechner fallen gleichzeitig aus



3-Phasen-Commit (2)

- ABORT-Behandlung wie im 2PC
- neue Zwischenphase (Zustand PRECOMMIT) falls alle Agenten Phase 1 mit READY abschließen:
  - Koordinator teilt PRECOMMIT allen Teiltransaktionen mit
  - nach Eingang von F Quittungen (PC-ACK) erfolgt COMMIT-Entscheidung
  - erst jetzt ist Überleben der Transaktion gewährleistet
- Koordinatorausfall: Wahl eines neuen Koordinators
  - Erfragen des Transaktionszustandes noch nicht abschließend bearbeiteter Transaktionen bei überlebenden Rechnern:
  - Commit / Abort (bzw. keine Information): Mitteilung des Transaktionsausgangs
  - Precommit bei wenigstens einem überlebenden Rechner: Commit-Protokoll wird von neuem Koordinator mit Verschicken von Precommit-Nachrichten fortgeführt
- löst Blockierungsproblem im Prepared-Zustand
  - lag negative Koordinator-Entscheidung vor: kein Precommit möglich
  - positive Koordinator-Entscheidung: wenigstens 1 Knoten muss Precommit-Zustand haben
  - Precommit-Zustand beim Koordinator: positive oder negative Entscheidung möglich

Paxos: verteiltes Consensus-Protokoll

- Mehrere Knoten müssen sich auf auf einen Wert einigen
  - Szenarien: Commit-Entscheidung, Datenreplikation (welche Kopie ist gültig); Neuwahl eines Primary-Knotens, …
- Vorteile
  - fehlertolerant, d.h. geringer Einfluss von Knotenausfällen (im Gegensatz zu 2PC)
  - garantierte Korrektheit und Terminierung (im Gegensatz zu 3PC)
- Beteiligte
  - N beteiligte Knoten, z.B. Resource Manager (RM)
  - ein Client inititiert Entscheidungsprozess mit einem bestimmten Wert
  - Leader / Proposer (kann nach Ausfall gewechselt werden)
  - Abstimmung unter 2F+1 *Acceptor*-Knoten
- bis zu F Acceptor-Ausfälle verkraftbar
  - Mehrheitsentscheid noch möglich
- Nutzung u.a. in Google Chubby (Lock-Service), Spanner (Replikation)

Paxos – Commit*

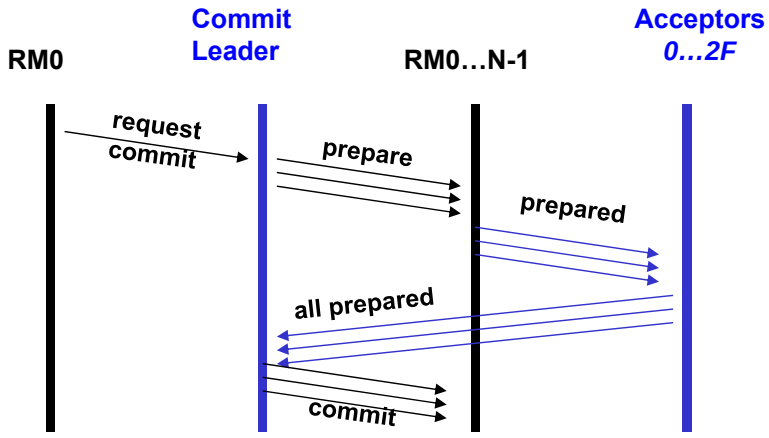
- Paxos-Entscheidung über Prepared-Ergebnisse der einzelnen Knoten / Resource-Manager
- Leader veranlasst auf Client-Anforderung N Prepared-Abstimmungsprozesse mit den selben 2F+1 Acceptor-Knoten
- Commit-Entscheidung, wenn F+1 Acceptors Prepared-Ergebnis für alle RM bestätigen
- Nachrichtenaufwand

  N+ N(2F+1)+(2F+1)+N
  = 3N+2F(N+1)+1

  falls Leader=RM: 3N + 2F(N+1)-2

  für F+1 statt 2F+1: 3N +F(N+1) -2

  - F=0: analog 2PC (ohne ACK)
  - F=1: 4N-1 Nachrichten



* *J. Gray, L. Lamport: Consensus on transaction commit. ACM Trans. Database Syst. 31(1): 133-160 (2006)*

Nachrichtenbedarf verteilter Commit-Protokolle

N: Anzahl beteiligter Knoten

F: #erlaubter Koordinator/Acceptor-Ausfälle (Paxos)

	Allgemein	Beipiel 1 (N=2, F=1)	Beipiel 2 (N=10, F=1)
1-Phasen-Commit	2*(N-1)	2	18
Lineares 2PC	2*N-1	3	19
zentralisiertes/hierarchisches 2PC	4*(N-1)	4	36
3-Phasen-Commit	6*(N-1)	6	54
Paxos-Commit	3N+2F(N+1)-2 3N+F(N+1)-2	10 7	50 39

Zusammenfassung

- ACID-Eigenschaften für verteilte Transaktionen
- Synchronisation
  - Sicherstellung der globalen Serialisierbarkeit
  - verteilte Sperrverfahren verursachen geringen Kommunikationsaufwand
  - Trend zu Mehrversionen-Synchronisation / Snapshot Isolation
- Globale Deadlock-Behandlung
  - einfachste Lösung: Timeout
  - Deadlock-Vermeidung (z.B. Wound/Wait) vermeidet Kommunikation, führt jedoch zu unnötigen Rücksetzungen
  - verteilte Deadlock-Erkennung aufwändig: reduziert aber Rücksetzungen
- Verteilte Commit-Protokolle
  - Sicherstellung der Atomarität und Dauerhaftigkeit bei verteilten Änderungen
  - Standardverfahren: hierarchisches 2PC
  - Varianten mit verbesserter Leistungsfähigkeit oder Verfügbarkeit (1PC, 3PC, Paxos)
  - relativ hoher Aufwand