

5. Transaktionsverwaltung

- **Transaktionskonzept**
 - führt ein neues Verarbeitungsparadigma ein
 - ist Voraussetzung für die Abwicklung betrieblicher Anwendungen (*mission-critical applications*)
 - erlaubt „Vertragsrecht“ in rechnergestützten IS zu implementieren
- **Gefährdung der DB-Konsistenz**
- **Transaktionsverwaltung**
 - ACID-Eigenschaften
 - Architektur
- **Transaktionsablauf**
 - SQL-Operationen: COMMIT WORK, ROLLBACK WORK (Beginn einer Transaktion implizit)
 - Zustände und Zustandsübergänge
- **Wie erreicht man Atomarität?**
 - Einsatz von Commit-Protokollen (zentralisierter TA-Ablauf)
 - **2PC (Zweiphasen-Commit-Protokoll)**
 - verteilter TA-Ablauf
 - Fehleraspekte
 - Kostenbetrachtungen
 - 2PC-Protokoll in Bäumen
 - Bestimmen eines Koordinators
 - Hierarchisches 2PC

Gefährdung der DB-Konsistenz

| | | |
|--|--|---|
| Übereinstimmung zwischen DB und Miniwelt | unzulässige Änderungen Integritätsüberwachung des DBVS TA-orientierte Verarbeitung | undefinierter DB-Zustand nach einem Systemausfall Transaktionsorientierte Fehlerbehandlung (Recovery) |
| Korrektheit der Abbildungshierarchie | Mehrbenutzer-Anomalien Synchronisation | Fehler auf den Externspeichern, Inkonsistenzen in den Zugriffspfaden Fehlertolerante Implementierung, Archivkopien (Backup) |
| | durch das Anwendungsprogramm | durch das DBVS und die Betriebsumgebung |

Transaktionsverwaltung

• DB-bezogene Definition der Transaktion:

Eine TA ist eine ununterbrechbare Folge von DML-Befehlen, welche die Datenbank von einem logisch konsistenten Zustand in einen neuen logisch konsistenten Zustand überführt.

➔ Diese Definition eignet sich insbesondere für relativ kurze TA, die auch als **ACID-Transaktionen** bezeichnet werden.

• Wesentliche Abstraktionen aus Sicht der DB-Anwendung

- Alle Auswirkungen auftretender Fehler bleiben der Anwendung verborgen (*failure transparency*)
- Es sind keine anwendungsseitigen Vorkehrungen zu treffen, um Effekte der Nebenläufigkeit beim DB-Zugriff auszuschließen (*concurrency transparency*)

➔ Gewährleistung einer fehlerfreien Sicht auf die Datenbank im logischen Einbenutzerbetrieb

• Transaktionsverwaltung

- koordiniert alle DBS-seitigen Maßnahmen, um ACID zu garantieren
- besitzt drei wesentliche Komponenten zur/zum
 - Synchronisation
 - Logging und Recovery
 - Konsistenzsicherung/Regelverarbeitung (wird später behandelt)
- kann zentralisiert oder verteilt (z.B. bei VDBS) realisiert sein
- soll Transaktionsschutz für heterogene Komponenten bieten

Transaktionsverwaltung (2)

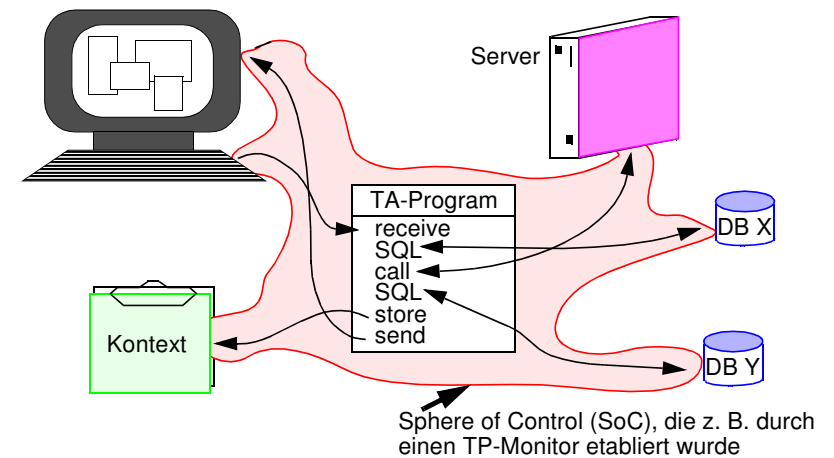
• Einsatz kooperierender Ressourcen-Manager (RM)

- RM sind Systemkomponenten, die **Transaktionsschutz für ihre gemeinsam nutzbaren Betriebsmittel (BM)** bieten
- RM gestatten die externe Koordination von BM-Aktualisierungen durch spezielle Commit-Protokolle

➔ Gewährleistung von ACID für DB-Daten und auch für andere BM (persistente Warteschlangen, Nachrichten, Objekte von persistenten Programmiersprachen)

• Ziel:

TA-orientierte Verarbeitung in heterogenen Systemen

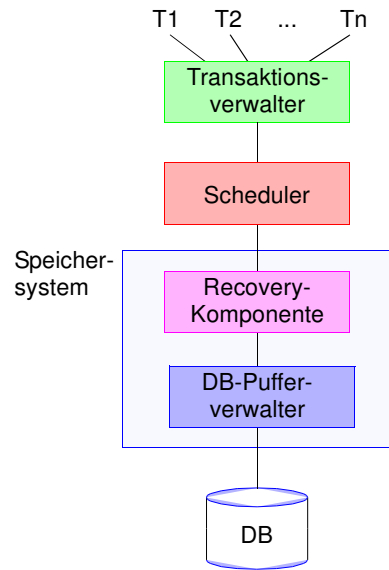


➔ Die gesamte verteilte Verarbeitung in einer SoC ist eine ACID-TA

- alle Komponenten werden durch die TA-Dienste integriert
- für die Kooperation ist eine Grundmenge von Protokollen erforderlich

Transaktionsverwaltung (3)

- **Abstraktes Architekturmodell für die Transaktionsverwaltung**
(für das Read/Write-Modell auf Seitenbasis)



- **Transaktionsverwalter**
 - Verteilung der DB-Operationen in VDBS und Weiterreichen an den Scheduler
 - zeitweise Deaktivierung von TA (bei Überlast)
 - Koordination der Abort- und Commit-Behandlung
- **Scheduler (Synchronisation)**
kontrolliert die Abwicklung der um DB-Daten konkurrierenden TA
- **Recovery-Komponente**
sorgt für die Rücksetzbarkeit/Wiederholbarkeit der Effekte von TA
- **DB-Pufferverwalter**
stellt DB-Seiten bereit und gewährleistet persistente Seitenänderungen

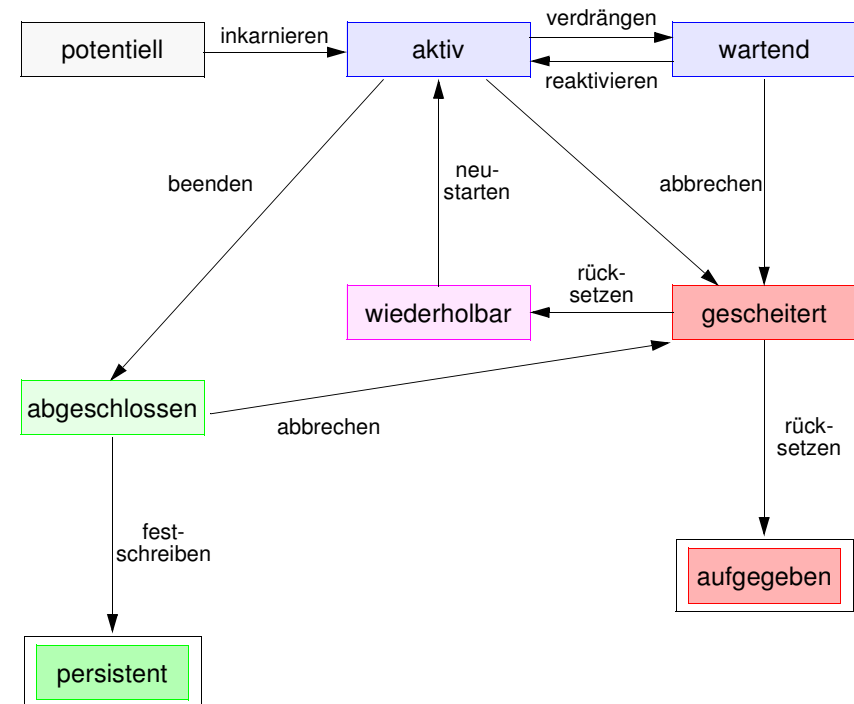
Zustände einer Transaktion

- **Transaktionsprogramm – Beispiel:**

```

BOT
UPDATE Konto
...
UPDATE Schalter
...
UPDATE Zweigstelle
...
INSERT INTO Ablage (...)
COMMIT;
    
```

- **Zustandsübergangs-Diagramm**



Zustände einer Transaktion (2)

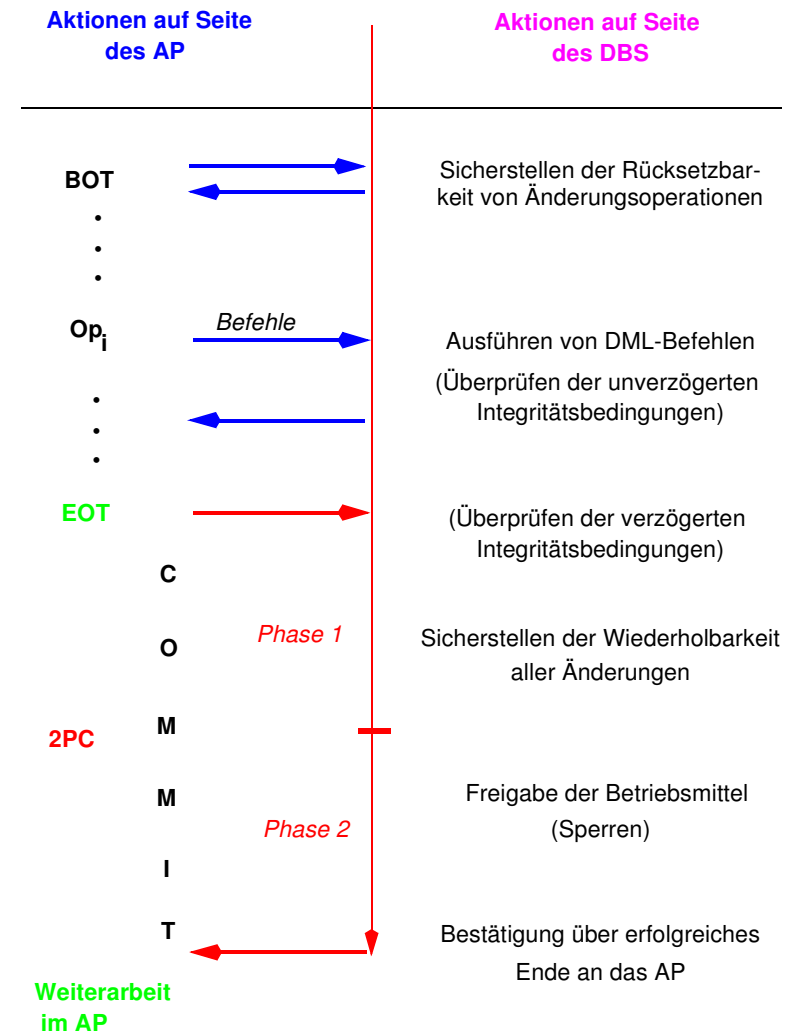
• Transaktionsverwaltung

- muß die möglichen Zustände einer TA kennen und
- ihre Zustandsübergänge kontrollieren/auslösen

• TA-Zustände

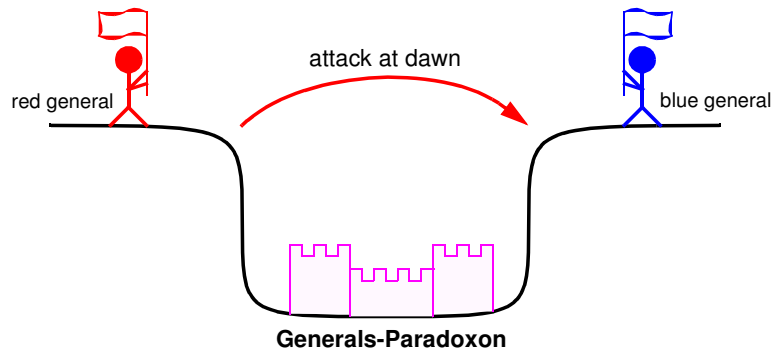
- **potentiell**
 - TAP wartet auf Ausführung
 - Beim Start werden, falls erforderlich, aktuelle Parameter übergeben
- **aktiv**
TA konkurriert um Betriebsmittel und führt Operationen aus
- **wartend**
 - Deaktivierung bei Überlast
 - Blockierung z.B. durch Sperren
- **abgeschlossen**
 - TA kann sich (einseitig) nicht mehr zurücksetzen
 - TA kann jedoch noch scheitern (z.B. bei Konsistenzverletzung)
- **persistent** (Endzustand)
Wirkung aller DB-Änderungen werden dauerhaft garantiert
- **gescheitert**
Vielfältige Ereignisse können zum Scheitern ein TA führen (siehe Fehlermodell, Verklemmung usw.)
- **wiederholbar**
Gescheiterte TA kann ggf. (mit demselben Eingabewerten) erneut ausgeführt werden
- **aufgegeben** (Endzustand)

Schnittstelle zwischen AP und DBS – transaktionsbezogene Aspekte



Verarbeitung in Verteilten Systemen

- Ein **verteiltes System** besteht aus autonomen Subsystemen, die koordiniert zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen
 - Client/Server-Systeme
 - Mehrrechner-DBS, . . .
- **Beispiel: The „Coordinated Attack“ Problem**



• Grundproblem verteilter Systeme

Das für verteilte Systeme charakteristische Kernproblem ist der Mangel an globalem (zentralisiertem) Wissen

- ➔ **symmetrische Kontrollalgorithmen sind oft zu teuer oder zu ineffektiv**
- ➔ **fallweise Zuordnung der Kontrolle**

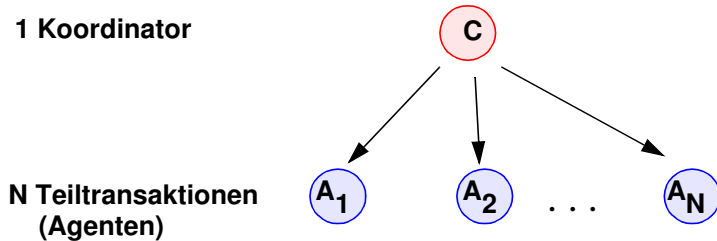
Verarbeitung in Verteilten Systemen (2)

- **Annahmen im allgemeinen Fall**
 - nicht nur Verlust (**omission**) von Nachrichten (Bote/Kurier wird gefangen)
 - sondern auch ihre bösartige Verfälschung (**commission failures**)
- ➔ **dann komplexere Protokolle erforderlich: Distributed consensus (bekannt als Byzantine agreement)**
- **Fehlermodell**
 - **allgemein:**
 - Transaktionsfehler
 - Crash (i. Allg. einiger Rechner)
 - Gerätefehler
 - **speziell (nur omission failures):**
 - Nachrichtenverluste
 - Nachrichten-Duplikate: dieselbe Nachricht erreicht ihr Ziel mehrmals, möglicherweise verschachtelt mit anderen Nachrichten
 - transiente Prozessfehler (soft crashes): Restart erforderlich, aber kein Datenverlust auf Externspeicher
 - Achtung: Wir schließen hier bösartige Verfälschungen von Nachrichten aus!
- ➔ **keine speziellen Annahmen:**
Kommunikationssystem arbeitet mit Datagrammen als einfachstem Typ von unbestätigten, sitzungsfreien Nachrichten
- ➔ **schwierige Fehlererkennung, z. B. oft über Timeout**

Verarbeitung in Verteilten Systemen (3)

• Erweitertes Transaktionsmodell

verteilte Transaktionsbearbeitung (Primär-, Teiltransaktionen) –
zentralisierte Steuerung des Commit-Protokolls (vergleiche „Heirat“)



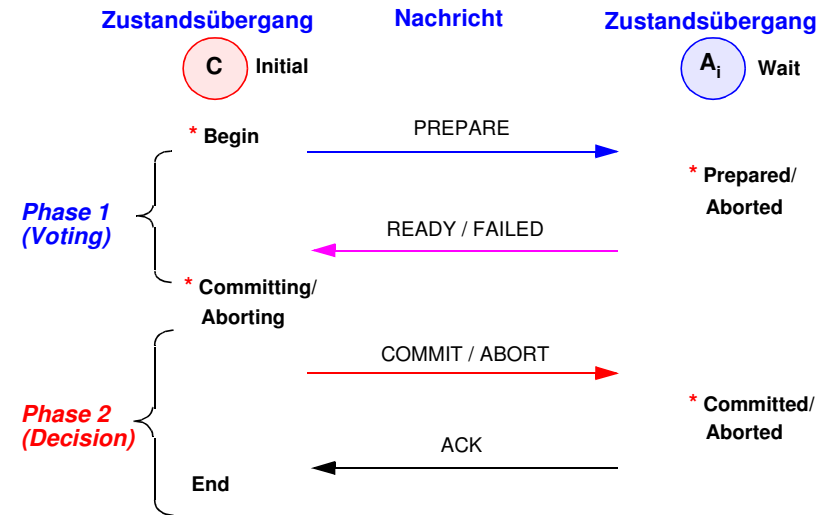
➔ *rechnerübergreifendes Mehrphasen-Commit-Protokoll notwendig, um Atomarität einer globalen Transaktion sicherzustellen*

• Allgemeine Anforderungen an geeignetes Commit-Protokoll:

- Geringer Aufwand (#Nachrichten, #Log-Ausgaben)
- Minimale Antwortzeitverlängerung (Nutzung von Parallelität)
- Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern

➔ *Zentralisiertes Zweiphasen-Commit-Protokoll stellt geeignete Lösung dar*

Zentralisiertes Zweiphasen-Commit



* synchrone Log-Ausgabe (log record is force-written)
 „End“ ist wichtig für Garbage Collection im Log von C

• Protokoll (Basic 2PC)

- Folge von Zustandsänderungen für Koordinator und für Agenten
 - Protokollzustand auch nach Crash eindeutig: synchrones Logging
 - Sobald C ein NO VOTE (FAILED) erhält, entscheidet er ABORT
 - Sobald C die ACK-Nachricht von allen Agenten bekommen hat, weiß er, dass alle Agenten den Ausgang der TA kennen

➔ *C kann diese TA vergessen, d. h. ihre Log-Einträge im Log löschen!*

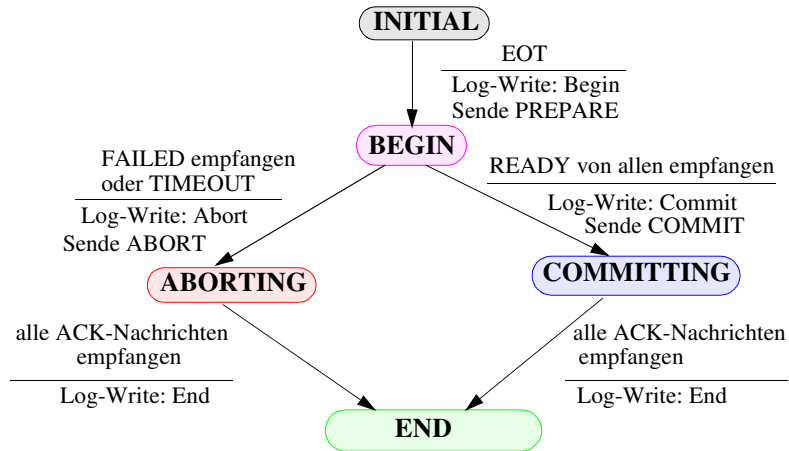
- Warum ist das 2PC-Protokoll blockierend?

• Aufwand im Erfolgsfall:

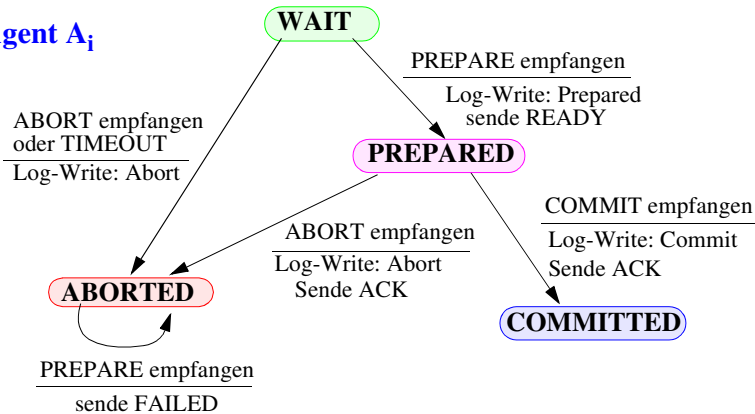
- Nachrichten:
- Log-Ausgaben (forced log writes):

2PC: Zustandsübergänge

Koordinator C



Agent A_i



2PC: Fehlerbehandlung

• Timeout-Bedingungen für Koordinator:

- **BEGIN** ➔ setze Transaktion zurück; verschicke ABORT-Nachricht
- **ABORTING, COMMITTING** ➔ vermerke Agenten, für die ACK noch aussteht

• Timeout-Bedingungen für Agenten:

- **WAIT** ➔ setze Teiltransaktion zurück (unilateral ABORT)
- **PREPARED** ➔ erfrage Transaktionsausgang bei Koordinator (bzw. bei anderen Rechnern)

• Ausfall des Koordinator-knotens:

Vermerkter Zustand auf Log

- **END:**
 - UNDO bzw. REDO-Recovery, je nach Transaktionsausgang
 - keine "offene" Teiltransaktionen möglich
- **ABORTING:**
 - UNDO-Recovery
 - ABORT-Nachricht an Rechner, von denen ACK noch aussteht
- **COMMITTING:**
 - REDO-Recovery
 - COMMIT-Nachricht an Rechner, von denen ACK noch aussteht
- Sonst: UNDO-Recovery

• Rechnerausfall für Agenten:

Vermerkter Zustand auf Log

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

2PC-Protokoll in TA-Bäumen

- **Typischer AW-Kontext: C/S-Umgebung**

- Beteiligte: Clients (PCs), Applikations-Server, DB-Server
- unterschiedliche *Quality of Service* bei Zuverlässigkeit, Erreichbarkeit, Leistung, ...

- **Wer übernimmt die Koordinatorrolle? – wichtige Aspekte**

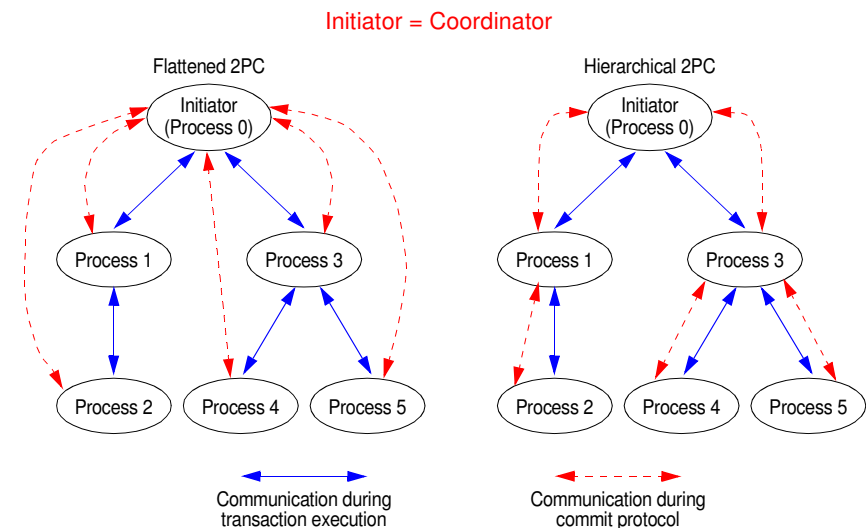
- TA-Initiator
- Zuverlässigkeit und Geschwindigkeit der Teilnehmer
 - Anzahl der Teilnehmer
 - momentane Last
 - Geschwindigkeit der Netzwerkverbindung
- Kommunikationstopologie und -protokoll
 - sichere/schnelle Erreichbarkeit
 - im LAN: Netzwerkadressen aller Server sind bekannt; jeder kann jeden mit Datagrammen oder neu eingerichteten Sitzungen erreichen
 - im WAN: „transitive“ Erreichbarkeit; mehrere Hops erforderlich; Initiator kennt nicht unbedingt alle (dynamisch hinzukommenden) Teilnehmer (z. B. im Internet)
- Im einfachsten Fall: TA-Initiator übernimmt Koordinatorrolle

➔ sinnvoll, wenn Initiator ein zuverlässiger, schneller und gut angebundener Applikations-Server ist!

2PC-Protokoll in TA-Bäumen

- **Drei wichtige Beobachtungen**

- Während der TA-Verarbeitung formen die involvierten Prozesse einen (unbalancierten) Baum mit dem Initiator als Wurzel. Jede Kante entspricht einer dynamisch eingerichteten Kommunikationsverbindung
- Für die Commit-Verarbeitung kann der Baum „flachgemacht“ werden
 - Der Initiator kennt die Netzwerkadressen aller Teilnehmerprozesse bei Commit (durch „piggybacking“ bei vorherigen Aufrufen)
 - Nicht möglich, wenn die Server die Information, welche Server sie aufgerufen haben, kapseln

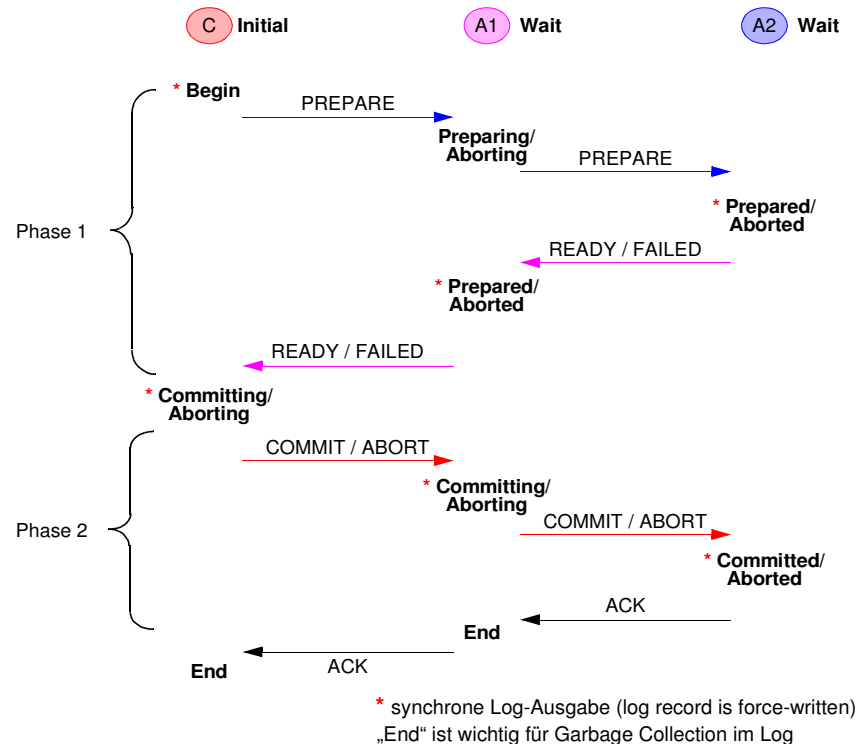


- „Flachmachen“ kann als spezieller Fall der Restrukturierung der C/S-Aufrufhierarchie gesehen werden
 - Es könnte auch ein zuverlässiger innerer Knoten als Koordinator ausgewählt werden
 - „Rotation“ der Aufrufhierarchie um den neu gewählten Koordinatorknoten

Hierarchisches 2PC

• Allgemeineres Ausführungsmodell

- beliebige Schachtelungstiefe, angepasst an Client/Server-Modell
- Modifikation des Protokolls für "Zwischenknoten"



• Aufwand im Erfolgsfall:

- Nachrichten:
- Log-Ausgaben (forced log writes):
- Antwortzeit steigt mit Schachtelungstiefe

• Problem Koordinator-/Zwischenknotenausfall ➔ Blockierung möglich!

Commit: Optimierungen

• Commit-Protokoll

- hat erheblichen Einfluss auf den TA-Durchsatz
 - substanzieller Anteil an der gesamten TA-Zeit bei kurzen TAs (Reservierungen, Buchungen, Kreditkartenvalidierung usw.)
 - Commit-Anteil bei TAs, die einen Satz ändern, etwa 1/3 der TA-Zeit
- Schnelleres Commit-Protokoll kann Durchsatz auf mehrfache Art verbessern
 - Reduktion der Commit-Dauer jeder TA
 - frühere Freigabe von Sperren, was die Wartezeit anderer TA verkürzt
 - Verkleinerung des Zeitfensters für die Blockierungsgefahr (Crash oder Nicht-Erreichbarkeit von C)

• Wünschenswerte Charakteristika eines Commit-Protokolls

- 1. A von ACID: stets garantierte TA-Atomarität¹**
- 2. Fähigkeit, den Ausgang der Commit-Verarbeitung einer TA „nach einer Weile“ vergessen zu können**
 - begrenzte Log-Kapazität von C
 - C muss sicher sein, dass alle Agenten den Commit-Ausgang für T_i kennen oder
 - C nimmt einen bestimmten Ausgang an, wenn er keine Information über T_i mehr besitzt
- 3. Minimaler Aufwand für Nachrichten und synchrones Logging**
- 4. Optimierte Leistungsfähigkeit im fehlerfreien Fall (no-failure case)**
 - ist wesentlich häufiger zu erwarten: optimistisches Verhalten
 - vor allem Minimierung von synchronen Log-Ausgaben (forced log write)
 - dafür zusätzlicher Recovery-Aufwand, um verlorengegangene Information im Fehlerfall wieder zu beschaffen
- 5. Spezielle Optimierung für Leser**
(ihre Commit-Behandlung ist unabhängig vom Ausgang der TA)

1. May all your transactions commit and never leave you in doubt! (Jim Gray)

Commit: Kostenbetrachtungen

- **vollständiges 2PC-Protokoll**

($N = \# \text{Teil-TA}$, davon $M = \# \text{Leser}$)

- Nachrichten: $4N$
- Log-Ausgaben: $2 + 2N$
- Antwortzeit:
längste Runde in Phase 1 (kritisch, weil Betriebsmittel blockiert)
+ längste Runde in Phase 2

- **Spezielle Optimierung für Leser**

- Teil-TA, die nur Leseoperationen ausgeführt haben, benötigen keine Recovery!
 - C und D von ACID sind nicht betroffen
 - für A und I genügt das Halten der Sperren bis Phase 1
- Lesende Teil-TA brauchen deshalb **nur an Phase 1** teilnehmen²
 - Mitteilung an (Zwischen-)Koordinator: „**read-only vote**“
 - dann Freigabe der Sperren
 - (Zwischen-)Koordinator darf (Teil-)Baum nur freigeben, wenn er ausschließlich *read-only votes* erhält
- Nachrichten:
- Log-Ausgaben:
für $N > M$

2. Leseroptimierung läßt sich mit allen 2PC-Varianten kombinieren, wird jedoch nachfolgend nicht berücksichtigt

Commit: Kostenbetrachtungen (2)

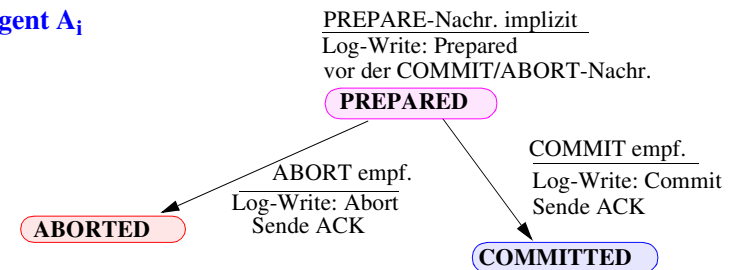
- **1PC-Protokolle**

- Bei Aufruf der Commit-Operation durch die TA sind bereits alle Agenten im PREPARED-Zustand
 - „implicit yes-vote“ (IYV)
 - Modifikation der API erforderlich

- **Variante 1: A_i geht nach jedem Auftrag in den PREPARED-Zustand**

- Jeder Aufruf von A_i : Work&Prepare
- Einschränkungen bei verzögerten Integritätsprüfungen erforderlich (deferred)

Agent A_i



- Nachrichten:
- durchschnittlich K Aufträge pro Agent;
- Log-Ausgaben:

- **Variante 2: A_i geht beim letzten Auftrag in den PREPARED-Zustand**

- Normaler Aufruf von A_i : Work
- Letzter Aufruf von A_i : Work&Prepare;
Lässt sich diese Art der Optimierung **immer** erreichen?
- Nachrichten:
- Log-Ausgaben:

Commit: Kostenbetrachtungen (3)

• Weglassen der expliziten ACK-Nachricht

- A_i fragt ggf. nach, falls ihn die Koordinatorentscheidung nicht erreicht
- C weiß nicht, ob alle A_i die Commit/Abort-Nachricht erhalten haben

➔ **impliziert "unendlich langes Gedächtnis" von C**

- Nachrichten:

- Log-Ausgaben:

• Spartanisches Protokoll

- A_i geht nach jedem Auftrag in den PREPARED-Zustand;
Weglassen der expliziten ACK-Nachricht

- Nachrichten:

- Log-Ausgaben:

- Nur letzter Aufruf: Work&Prepare;
Log-Ausgaben:

➔ **Log-Aufwand bleibt gleich (oder erhöht sich drastisch) !**

Zusammenfassung

• Transaktionsparadigma

- Verarbeitungsklammer für die Einhaltung von semantischen Integritätsbedingungen

- Verdeckung der Nebenläufigkeit (*concurrency isolation*)

➔ **Synchronisation**

- Verdeckung von (erwarteten) Fehlerfällen (*failure isolation*)

➔ **Logging und Recovery**

- im SQL-Standard: COMMIT WORK, ROLLBACK WORK

➔ **Beginn einer Transaktion implizit**

• Zweiphasen-Commit-Protokolle

- Hoher Aufwand an Kommunikation und E/A
- Optimierungsmöglichkeiten sind zu nutzen
- Maßnahmen erforderlich, um Blockierungen zu vermeiden!

➔ **Kritische Stelle: Ausfall von C**

• Einsatz in allen Systemen!

• Varianten des Commit-Protokolls:

- Hierarchisches 2PC:
Verallgemeinerung auf beliebige Schachtelungstiefe

- Reduzierte Blockierungsgefahr durch 3PC:
Nach Voting-Phase wird in einer zusätzlichen Runde (Dissemination-Phase) allen Agenten der TA-Ausgang mitgeteilt. Erst nachdem C sicher ist, dass alle Agenten das Ergebnis kennen, wird die Decision-Phase initiiert:
unabhängige Recovery, aber komplexes Fehlermodell