VERTEILTE SYSTEME

(Zusammenfassung)

Buch Silberschatz 4. Auflage
Kapitel 15 bis 18

1 Einführung Netzwerk-Struktur

Ein **Verteiltes System** ist ein System mit räumlich verteilten Komponenten, die keinen gemeinsamen Speicher benutzen und einer dezentralen Administration unterstellt sind. Zur Ausführung gemeinsamer Ziele ist eine Kooperation der Komponenten möglich.

⊕ Greift ein Prozess auf Daten zu, macht es für die Anwendung keine Unterschied, ob die Daten im lokalen Speicher oder in dem Speicher eines Nachbarprozesses liegen.
⊕ Eine Systemerweiterung minimiert das Risiko der Überlastung einzelner Systemkomponenten.
⊕ Nutzung gemeinsamer Ressourcen zur Kostenersparnis.
⊕ Einzelne Fehler/Ausfälle können von anderen Komponenten toleriert werden.
⊕ Stetige Anpassung der Größe eines Systems an aktuelle Anforderungen.

⊖ Zunahme der Komplexität durch Verteilung und Heterogenität
⊖ Verwaltung von komplexen Netzinfrastrukturen
⊖ Softwaredefizit
⊖ Zusätzliche Fehlermöglichkeiten durch neue Netzwerkkomponenten; Datenschutz
⊖ Konsistenzprobleme: Zugriff auf verteilt gehaltene Daten

Netzwerk-Betriebssysteme NOS:
• NOS sind Betriebsysteme die Netzwerkfunktionalität anbieten z.B (UNIX)
− keine unmittelbare, gemeinsame Nutzung von Betriebsmitteln
− jeder Rechner besitzt Kontrolle über seine Betriebsmittel
− Autonomie: bei NOS wesentlich höher als bei DOS
⊖ Fehlertoleranz: bei NOS wesentlich schlechter als bei DOS.
Verteilte Anwendungen: Remote-Login File-Transfer, E-Mail, WWW.

Verteilte Systeme charakterisieren sich mit folgenden Eigenschaften:
Ressourcen-Sharing, Zuverlässigkeit, Schnelligkeit, Kommunikation

Netzwerktopologie

Die Topologie beschreibt, in welcher Beziehung Kommunikationsteilnehmer zueinander stehen.

- **Vollständige verbundene Netzwerke**
  ⊕ Kommunikation zwischen Knoten ist sehr schnell
  ⊖ Kosten sind sehr hoch
- **Partielle verbundene Netzwerke**
  − Kosten sind abhängig von Verbindungsleitungen
  ⊖ Kommunikation kann schlecht sein
- **Hierarchische Netzwerke:**
  − Baum-Struktur

- **Stern:**
  – eine der wichtigsten Topologie für kommerziellen LAN
  – Alle Stationen sind über eine einzelne Verbindungsleitung mit einem zentralen Verbindungsknoten (HUB, Switch) verbunden
  ⊕ Bei Ausfall eines Knotens ist nur dieser betroffen
  – Beispiel: Token-Ring, sehr schnelle Netzwerk
  – Kosten linear

- **Bus:**
  – eine der wichtigsten für kommerziellen LAN
  – Broadcast Topologie, da das gemeinsame genutzte Medium alle Stationen miteinander verbindet
  – Kontrollmedium ist nötig um Kollisionen zu vermeiden
  ⊕ Einfache Verkabelung und sehr minimale Installationskosten
  ⊖ Bei Ausfall eines Knotens ist das ganze Netz betroffen, aufwendige Fehlersuche

- **Ring:**
  – vor allem im Backbonebereich spielen eine wichtige Rolle.
  – Jede Station ist mit ihrer Nachbarstation über eine unidirektionale oder Bidirektionale Verbindungsleitung zusammengeschaltet
  – Kommunikation nur in eine Richtung möglich Gliederung
  – Jeder Rechner kann senden, empfangen und weiterleiten.
  ⊕ Keine Kollisionen, schnelle Netzwerke möglich
  ⊕ Kosten sind linear (auf die Anzahl der Knoten
  ⊖ Kommunikation kann im Worst Case sehr hoch sein!
  Bei unidirektionalen $(n-1)$ und bidirektionalen $\frac{n}{2}$
  ⊖ Bei Ausfall eines Knotens ist das ganze Netz betroffen.

- **Vermaschtes Netz:**
  – Punkt zu Punkt Verbindung
  – Nicht alle Stationen sind direkt verbunden

- **Hybride Netzwerke:**
  – Ring-Bus Netzwerke

Netzwerktypen

- **LAN:**
  – sind lokale Netze die auf kleinere Gebiete beschränkt sind (z.B. Firmen
  – Die Übertragung erfolgt über Koaxialkabel und Glasfaserkabel
  – Die Übertragungsrate beträgt hier zwischen 10 und 1000 Mbps
  – LANs können untereinander direkt, über WANs, zu größeren Strukturen verbunden werden
  – Allgemeine Topologie: Stern, Bus und Ring

– Für Implementierung von Hochgeschwindigkeit Netzwerken bieten sich gegenwärtig zwei konventionelle Technologien an:
1. Asynchronus Transfer Mode (ATM) und
2. Gigabit Ethernet als Weiterentwicklung des Ethernet
• ATM arbeitet verbindungsorientiert und garantiert grosse Bandbreiten ist daher auch in WAN einsetzbar • Nachteil von ATM ist die Inkompatibilität zu den verbreitesten LAN Technologie • Ethernet und Gigabit Ethernet arbeitet verbindungslos und ist zur Ethernet-Technologien kompatibel • Es können auch Gigabit Ethernet LANs gebaut werden mit ATM ähnlichen Eigenschaften

- **WAN:**
  – Entfernungen bis Tausende Kilometer können vernetzt werden
  – Komplexe Protokolle
  – Können von Organisationen, unabhängig von Benutzern, verwaltet werden
  – Höhere Fehlerrate 1 von 100.000
  – Im Allgemeine Verwendung von Punkt- zu-Punkt Verbindungsleitung
  – Allgemeine Topologie: Maschennetz und Stern

Kommunikation

- **Naming** bezeichnet man das Abbilden der logischen Datenobjekte (Dateinamen) auf physikalische Objekte(Adresse der Daten auf dem Speichermedium) in einem verteilen System.

- **Routing-Strategien:**
  Virtual Circuit, Fixed Routing, Dynamische Routing

- **Kommunikation-Strategien:**
  Packet-Switching, Message-Switching und Circuit-Switching

- **Verbindung-Strategien:**
  CSMA/CD, Token Passing, Message Slots

2 Struktur von Verteilte Systeme

Data-Migration bedeutet das Verlagern von Dateien von einem Ort an einen anderen, ohne dass sich ihre Namen verändern müssen.

Um den Leistungsanforderungen zu genügen, werden Zugriffsverfahren benutzt, die den Netzwerk reduzieren, und wenn möglich den Server entlasten. Beim zentralen Ansatz, dem sogenannten **Remote Service**, werden Schreibe-, und Lesewünsche immer an den Datei-Server gesendet. Dies verursacht hohen Datentransfer im Netz und eine hohe Belastung des File-Servers.

• RPC ist ein spezielles Kommunikationssystem für Client-Server Kommunikation.

Wenn ein Programm, der ein Unterprogramm aufruft, welches sich auf einem anderen Rechner befindet, bezeichnet man **Remote Procedure Call**, RPC (entfernter Unterprogrammaufruf).
Remote Procedure Call (RPC) ermöglicht es den Programmen Prozeduren aufzurufen, die auf andere Maschinen laufen. RPC ist ein Beispiel für die synchronen Nachrichtenaustausch. RPC gehört im Sitzungsschicht, da er auf ein vollständiges Transport aufsetzt.

Client-Server-Modell basiert auf einem einfachen verbindungslosen Frage/Antwort Protokoll. Der Client sendet eine Nachricht an den Server und fordert einen Service an. Der Server erledigt diese Arbeit und gibt die Daten zurück bzw. einen Fehlercode, falls die Arbeit nicht ausgeführt werden konnte.
⊕ CS-Modell ist sehr einfach und man braucht keine Verbindung auf- und abzubauen.

Threads (Faden)
• Threads sind Leichtgewichprozesse, die einen globalen Zustandsraum gemeinsam teilen
• Threads wurden eingeführt, um Parallelität von Prozesse und blockierenden Systemaufrufe zu ermöglichen.
• Jeder Thread verfügt über seinen eigenen Programmzähler, Stack und Registersatz.
• Threads sind nicht so unabhängig wie Prozesse: Sie teilen sich einen Adressraum und können so alle globalen Variablen gemeinsam nutzen.
• Jeder Thread kann auf jede virtuelle Adresse zugreifen und folglich den Stack von anderen Threads lesen, schreiben oder löschen. Ein Schutz davor ist nicht möglich und sollte in der jeweiligen Implementierung nicht notwendig sein.

Zustandlose vs. zustandbehaftete Server
Ein **zustandsbehafteter Server** verwaltet Informationen über Clients, so dass für die Dauer der Dienstleistung eine logische Verbindung zwischen Client und Server besteht.
⊕ einfache Caching-Strategien
⊕ schnelle Zugriff
⊕ Es enstehen keine Probleme, wenn ein Client abstürzt
⊕ Es wird kein Server-Speicherplatz für Tabellen verschwendet.
⊖ Aufwand, der nötig ist, um in Fehlersituationen korrekt weiter zu arbeiten .

Ein **zustandloser Server** speichert keine Daten über Clients. Bei einem Datenaustausch mit einem zustandslosen Server muss der Client bei jedem Zugriffswunsch sämtliche Parameter übergeben.
⊕ einfache Fehlerkorrektur
⊕ bessere Leistung
⊕ Es ist möglich Dateien zu sperren.
⊕ Vorausschauendes Lesen möglich
⊕ kürzere Anfragenachrichten
⊖ aufwendige Cache-Verwaltung.

3 Verteilte Dateisysteme

Ein verteiltes Dateisystem stellt dem Benutzer auf allen Maschinen im Netz ein einheitliches Dateisystem zu Verfügung. Der Dienst ist orttransparent, d.h. de Benutzer muss nicht wissen, wo im System seine Daten gespeichert sind.

Naming und Transparenz

Transparenz: ist ein Ziel von Entwicklern das Systemsicht einheitlich zu erreichen. Beim Datei-Zugriff auf Daten ist das Dateisystem dafür Verantwortlich, diese zu lokalisieren.

• *Location transparency (Ortstransparenz):* Der Pfadname gibt keinen Hinweis darauf, wo sich die Datei (oder ein anderes Objekt) befindet.
• *Location Independency (Ortsunabhängig):* Die Datei kann an jede beliebige Stelle im Netzwerk verschoben werden, ohne dass der Name verändert werden muss.

Remote File Access RFA

Caching: Unter Caching versteht man das Benutzen eines lokalen Speichers, der Kopien von Daten des File-Servers enthält. Bei einem Schreib- oder Lesewunsch wird zunächst lokal gesucht, ob die entsprechende Datei bzw. Dateiseite vorhanden ist. Falls Ja, wird der Dateizugriff lokal ausgeführt, falls Nein, erfolgt eine Anfrage beim File-Server, d.h. Remote Service.
Die Vorteile von eine Caching:
⊕ Reduzierung des Datenverkehrs im Netz
⊕ File-Server-Entlastung
⊕ Hohe Verfügbarkeit der Daten im System.

Im Client-Server-Systemen, die jeweils einen eigenen Hauptspeicher und eine Festplatte umfassen, gibt es vier Möglichkeiten, Dateien oder Teile von Dateien zu speichern:
1. Auf der Festplatte des Servers:
⊕ einfachste Möglichkeit, da viel sehr Platz zu Verfügung steht.
⊕ keine Konsistenzprobleme
⊖ Bevor ein Client eine Datei lesen kann, muss sie von der Festplatte des Servers in den Hauptspeicher des Servers übertragen werden und von dort über das Netzwerk in den Hauptspeicher des Clients. Beide Übertragungen kosten Zeit.
2. Im Hauptspeicher des Servers:
⊕ schneller Zugriff
3. Auf der Festplatte des Clients:
⊕ ist einfach zu realisieren.
⊕ keine Konstistenzprobleme.
⊖ sehr langsam
4. Im Hauptspeicher des Clients: gibt es drei Möglichkeiten
− direkt innerhalb der Adressraums des jeweiligen Benutzerprozesses
− Die Unterbringung des Caches ist der Kernel.
− Spezieller Cache-Managerprozess auf Benutzerebene
⊖ Bei Systemabsturz können Daten, die noch nicht geschrieben wurden, verloren ge-

hen.

Für die Lösung des Konsistenzproblems ist die Verwendung von :
1. Write-Through-Algorithmus
⊕ einfachste Methode
Hierbei werden nur Daten im Cache gespeichert, auf die lesend zugegriffen wird. Bei Schreibzugriffen wird auf Remote Service umgeschaltet. Somit besitzt der Server immer die aktuelle Kopie und es kann zu keinen Konsistenzproblemen kommen.
⊖ Bei Schreiben wird das Netz und Server belastet
2. Write-On-Close
− Erst nach schließen der Datei werden Daten zurückgeschrieben.
3. Verzögertes-Schreiben
− Beim Verzögertes-Schreiben werden Blöcke zunächst in den Cache geschrieben und bleiben dort eine gewisse Zeit bevor sie zum Server zurückgeschrieben werden.
⊕ Bessere Leistungen beim Schreiben und die Last bei Netz und Server wird reduziert
⊖ Mögliche Konsistenzprobleme und mögliche Verlust von Daten.
4. Zentrale-Steuerung

Vergleich Caching und Remote-Service

- Bei RPC müssen die nachrichten über's Netz gesendet werden. Wenn die Bandbreite klein ist, muss man mit längere Wartezeiten rechnen.
  ⊖ Hohe Datentransferim Netz und eine hohe Belastung eines File-Server
  ⊖ Sicherheitsnetz

- Ein Nachteil der Caching ist, dass um so mehr Datenkopien im System gespeichert werden, desto schwieriger wird es, diese Kopien immer auf dem aktuellen Stand zu halten (⇒ Konstistenzprobleme)

Replikation bezeichnet man die Vervielfältigung von Ressourcen, um höhere Fehlertoleranz und höhere Verfügbarkeit der Ressourcen zu erreichen. Ressourcen können sowohl Server-Prozesse, die auf verschiedene Maschinen installiert werden, als auch Daten von denen mehrere Kopien angelegt werden, sein.

Ein prinzipieller Unterschied liegt in der Speicherung der Dateien und im Paging. Bei NFS und AFS wurden für Programmbinaries und Paging lokale Platten benutzt, bei Sprite hingegen wurde alles vom Server bedient.

- **UNIX** teilt das Dateisystem in verschiedene physikalische Dateisystem auf, die auf unterschiedlichen Platenpartitionen abgespeichert werden

- **Network-File-System von SUN** macht aus Unix-File-System ein verteiltes System.
  NFS ist am weitesten verbreitet verteilte Dateisystem.
  NFS ist Remote-Service basiert.

- **Andrew:** ist ein kommerzielles System , das im Jahre 1983 entwickelt wurde. Alle Dateien werden in einen hierarchischen Dateibaum verwaltet, der in einen local *subtree* und in eine *shared subtree* aufgeteilt ist. Der shared subtree ist auf allen Maschinen des Systems identisch und enthält Programm-binaries et., im local subtree hingegen werden benutzerspezifische Dateien verwaltet.
  AFS ist derzeit das einzige System im Einsatz, das Dateimigration unterstützt.

- **Sprite** ist ein experimentelles verteiltes Betriebsystem. Die Sicht des Benutzers auf das Dateisystem ist auf jeder Maschine identisch. Das globale Dateisystem wird in sogenannte **Domains** aufgeteilt. Jedes Domain wird einem bestimmten File-Server zugeordnet.

4 Synchronisation in verteilten Systeme

Die Notwendigkeit zur Synchronisation ist immer dann gegeben, wenn mehrere Prozesse gemeinsam an einer Arbeit tätig sind.

Lamport hat eine Relation angegeben, die eine zeitliche Halbordnung in verteilten Systemen darstellt. E sei im folgenden die Menge von Ereignissen in einem verteilten System.
Definition: Zwei Ereignisse $A, B \in E$ erfüllen die Relation **Happend-Befor**, $A \rightarrow B$ (A geschieht vor B), falls eine der folgenden Bedienungen erfüllt ist:

1. A und B sind Ereignisse innerhalb desselben Prozesses, und A wurde vor B ausgeführt.

2. A ist das Ereignis Senden der Nachricht N und B ist das Ereignis Empfangen der Nachricht N.

3. Falls $A \rightarrow B$ und $B \rightarrow C$, dann gilt auch $A \rightarrow C$

Wechselseitiger Ausschluss:

Ein kritischer Bereich ist ein Teil eines Programms, in dem auf Ressourcen (z.B. Speicher) zugegriffen wird, die von mehreren Rechnern oder Prozessen benutzt werden. Ist zu jedem Zeitpunkt gewährleistet, dass nicht mehr als ein Prozess auf dieselben gemeinsam benutzten Ressourcen lesend oder schreibend zugreift, so können insbesondere keine Inkonsistenzen auftreten. Man spricht dann von wechselseitigem Ausschluss.

- **Ein zentraler Algorithmus**
  Beim zentralen Algorithmus wird ein Einprozessorsystem simuliert. Ein Prozess wird hier zum Koordinator ernannt. Der Koordinator gewährleistet, dass sich zu jedem Zeitpunkt höchstens 1 Prozess in einem bestimmten kritischen Bereich befindet.

Der Algorithmus arbeitet in folgender Weise: Möchte ein Prozess in einen kritischen Bereich eintreten, so stellt er eine Anfrage an den Koordinator. Dieser hat Informationen darüber, ob sich bereits andere Prozesse im kritischen Bereich befinden oder der kritische Bereich zum Anfragezeitpunkt von keinem anderen Prozess benutzt wird. In letzterem Fall wird seitens des Koordinators ein OK an die anfragende Einheit gesendet. Erfolgen weitere Anfragen von Prozessen, die ebenfalls in den kritischen Bereich eintreten wollen, so werden die zugehörigen Prozessnummern in der Reihenfolge ihres Anfragens in einer Warteschlange des Koordinators gespeichert. Gibt der benutzende Prozess den kritischen Bereich mit einer Fertigmeldung frei, so erhält der am längsten wartende Prozess ein OK und seine Prozessnummer wird aus der Warteschlange entfernt.

⊕ **Fairneß**: Die Reihenfolge der Anfrage bleibt erhalten und kein Prozess wartet ewig,
⊕ Relativ einfache Implementierung der 3 Operationen:
Anforderung (= Darf Ich?), Zuteilung (= OK) und Freigabe (= Fertig!).
⊖ Der Koordinator bildet einen Engpaß.
⊖ Der Koordinator darf nicht ausfallen.

- **Ein verteilter Algorithmus:**
  Voraussetzung hierfür ist eine totale Ordnung der Ereignisse im System, d.h. für jedes Paar von Ereignissen ist die Reihenfolge ihres Auftretens eindeutig bestimmt
  Das **Prinzip**: Wenn ein Prozess in den kritischen Bereich eintreten will, so sendet er eine Anfragenachricht mit dem Namen des kritischen Bereichs, seiner Prozessnummer und der aktuellen Zeit (total geordnet) an alle Prozesse des Verteilten Systems (alle anderen und sich selbst). Das Senden soll zuverlässig sein, d.h. jede Nachricht wird bestätigt. Was passiert nun, wenn eine Komponente des Verteilten Systems eine solche Anfrage erhält? Dazu sind drei Fälle zu unterscheiden:
  a) Der Empfänger befindet sich nicht im kritischen Bereich und will auch nicht in diesen eintreten → sendet OK an den Sender.
  b) Der Empfänger befindet sich im kritischen Bereich → sendet kein OK, sondern merkt sich die Anfrage in Warteschlange vor.
  c) Der Empfänger will selbst in den kritischen Bereich, hat "fast" zeitgleich eine eigene Anfrage abgesendet → er vergleicht die Zeitstempel der Nachrichten, der frühere gewinnt.
  Der Sender wartet auf alle OKs, dann tritt er in den kritischen Bereich (Kb) ein. Beim Verlassen sendet er OKs an alle Prozesse in seiner Warteschlange und entfernt die Prozesse hieraus

  ⊕ Fairneß und Deadlockfreiheit.
  ⊖ Der Algorithmus ist zwar verteilt, aber auch langsamer, komplizierter und weniger robust als der zentrale.
  ⊖ Die Netzwerkbelastung ist durch $2 \cdot (n-1)$ Nachrichten für eine Anfrage bei

n Komponenten im System recht hoch.
$\ominus$ Bei Ausfall einer Komponente ist die die Antwort auf eine Anfrage nicht möglich, was ggfs. als Ablehnung interpretiert wird. Somit sind alle folgenden Versuche blockiert, in den kritischen Bereich einzutreten. Ein Ausweg wäre, dass falls nach einem Timeout noch keine Antwort eingetroffen ist, erneut eine Anfrage gestellt wird und dann ggfs. davon ausgegangen wird, dass der Empfänger ausgefallen ist.

- **Ein Token-Ring Algorithmus**
  Gegeben seien n Prozesse. Auf diesen wird ein logischer Ring aufgebaut, in dem jedem Prozess eine Position zugewiesen wird. Damit hat jeder Prozess einen Nachfolger (der letzte Prozess hat wieder den ersten als Nachfolger).
  Bei der Initialisierung erhält ein Prozess ein Token. Will dieser Prozess in einen kritischen Bereich eintreten, so hat er die Möglichkeit. Verlässt er den kritischen Bereich, so gibt er das Token an seinen Nachfolger. Will er nicht eintreten, so gibt er es gleich weiter. Will kein Prozess in einen kritischen Bereich eintreten, so zirkuliert das Token mit hoher Geschwindigkeit im Ring. Der Algorithmus ist korrekt, da zu jedem Zeitpunkt nur ein Prozess das Token besitzen und somit auch nur dieser Prozess in den kritischen Bereich eintreten kann. Durch die Zirkulation ist gewährleistet, dass jeder Prozess nur endlich lange auf den Eintritt in einen kritischen Bereich warten muss (bei unendlicher Wartezeit spricht man auch von einem starvation problem).
  $\odot$ Prozess i kennt nur Prozess $i + 1 mod$ Ringgrösse
  $\ominus$ Verlust des Tokens und Ausfall eines Prozesses.
  $\ominus$ Das Entdecken des Tokenverlusts ist insbesondere schwierig, da die Zeit zwischen zwei Token auf dem Netzwerk nicht begrenzt ist.

Beim Vergleich der drei Algorithmen erweist sich der zentrale Algorithmus als am günstigsten. Dieser erfordert jedoch die Benennung eines Koordinators. Diese Auswahl wird durch bestimmte Regeln (z.B. höchste Netzwerkadresse) oder sog. Wahlalgorithmen getroffen.

Atomare Transaktionen

Die atomare Transaktion ist eine höhere Abstraktionsstufe, die technische Details verbirgt. Bei der Transaktion bestehe das verteilte System aus n unabhängigen Prozessen, die zufällig ausfallen können. Unsichere Kommunikation werde von der Software auf unteren Schichten behoben. Es wird **Stabiler Speicher** vorausgesetzt, da ein RAM-Inhalt bei Stromausfall verloren gehen kann und ein Plattenspeicher bei einem Hardwaredefekt zu Problemen führen kann.
Transaktionen müssen drei Eigenschaften besitzen:

- **Atomarität:** Eine Transaktion ist - von außen betrachtet - unteilbar.
  Eine Transaktion wird entweder vollständig ausgeführt oder überhaupt nicht.

- **Konsistent:** Die Transaktion verletzt keine System-Invarianten.
- **Isoliert:** Nebenläufige Transaktionen stören sich gegenseitig nicht - es macht also keinen Unterschied, ob sie seriell oder parallel ausgeführt werden.
- **Permanenz:** Falls eine Transaktion abgeschlossen wird,dann weren die Veränderungen festgeschrieben.

2-Phasen Commit Protokoll erlaubt allen beteiligten Servern miteinander zu kommunizieren, um bei der Beendigung der Transaktion zu einer gemeinsamen Entscheidung zu kommen ("Commit" oder "Abort")

- *Abstimmungsphase:* Der Koordinator sendet an alle beteiligten Prozesse eine Nachricht, in dem es erwartet wird, dass alle beteiligten Server bereit sind, das Commit für die verteilte Transaktion durchzuführen. Jeder angesprochene Rechner teilt ihm seine individuelle Antwort mit.
- *Abschlußphase:* es wird entschieden, ob die Transaktion erfolgreich abgeschlossen werden kann, oder abgebrochen werden muss; alle beteiligten Server müssen sich an diese Entscheidung halten.

Systemausfälle vor der ersten oder in der ersten Phase führen zum Abbruch der Transaktion. Systemausfälle in der zweiten Phase ändern nichts am Zustand der Transaktion (Commit/Abort) sondern zögern nur die Beendigung hinaus, da Wiederholungen stattfinden.

⊖ Die große Schwierigkeit des Algorithmus liegt dabei im blockierenden Verhalten. Fällt der Koordinator aus, müssen die Teilnehmer der Transaktion bis zum Wiederanlauf des Koordinators warten.
⊕ Die Lösung dieses Problems ist die Einführung eines weiteren Zustandes in den Automaten, wodurch das Drei-Phasen-Commit-Protokoll entsteht. Allerdings hat dieses Protokoll das Problem, dass bei gleichzeitigen Auftreten von 2 Fehlern die Konsistenz nicht gewährleistet werden kann.

Kontrolle der Nebenlaüfigkeit

Wenn gleichzeitig mehrere Transaktionen durchgeführt werden, benötigt man Mechanismen, die verhindern, dass sich die Transaktionen gegenseitig stören. Hierfür können unterschiedliche Verfahren angewendet werden:

• **Sperren:** Der am häufigsten eingesetzte Mechanismus zur Steuerung der Nebenläufigkeiten ist das Sperren. In der einfachsten Form wird bei einem Zugriff die ganze Datei gesperrt. Die Sperren können von einem zentralen oder jeweils von lokalen Sperrmanagern verwaltet werden. Das sehr restriktive Sperrschema kann verbessert werden, wenn Lese- und Schreibsperren unterschieden werden, da jeweils mehrere Lesesperren gleichzeitig existieren können. Weiterhin kann statt auf Dateiebene auf Blockebene gesperrt werden. Das Anfordern und Freigeben von Sperren zu dem Moment, in dem

sie benötigt bzw. nicht mehr benötigt werden, kann zu Inkonsistenzen und Deadlocks führen. Die meisten Transaktionen benutzen deshalb das sog. Zwei-Phasen-Sperren. Hierbei fordert der Prozess in der ersten Phase (Aufbau-Phase) alle Sperren an, die er benötigt und gibt sie in der zweiten Phase (Abbau-Phase) wieder frei. Falls mit der Aktualisierung der Daten erst nach der ersten Phase begonnen wird, können bei einem Fehlschlagen einer Sperrung einfach alle Sperren wieder freigegeben werden und die Sperrungen nach einer kurzen Zeit wieder versucht werden. Weiter kann bewiesen werden, dass, wenn alle Transaktionen das Zwei-Phasen-Sperren verwenden, alle möglichen Abläufe serialisierbar sind. Häufig erfolgt die zweite Phase erst, wenn die Transaktion beendet oder abgebrochen wurde. Sperren können zu Deadlocks führen, wenn Transaktionen auf gleiche Dateien aber in unterschiedlicher Reihenfolge zugreifen und bei nicht erfolgreichem Sperrversuch die belegten Sperren nicht freigeben. Hier können die bekannten Vermeidungs- oder Erkennungsalgorithmen eingesetzt werden.

• **Optimistische Nebenläufigkeitskontrolle** In diesem Fall arbeitet eine Transaktion so, als ob sie allein auf den Datenbestand zugreift. Erst bei dem Commit wird überprüft, ob eine der benutzten Dateien inzwischen verändert wurde. In diesem Fall wird die Transaktion abgebrochen. Dieser Ansatz ist gut geeignet, falls mit privaten Arbeitsbereichen gearbeitet wird. Sie ermöglicht maximale Parallelität und ist frei von möglichen Deadlocks. Der Nachteil ist, dass die Transaktion im Konfliktfall wiederholt werden muss, d.h. in einem stark belasteten System kann die Leistung ab einem Punkt plötzlich drastisch abfallen.

• **Zeitstempel** Hier wird jeder Transaktion bei $BEGINTRANSACTION$ ein Zeitstempel zugewiesen. Mit Hilfe von Lamports Algorithmus kann sichergestellt werden, dass jeder Zeitstempel eindeutig ist. Jeder Datei im System wird ein Lese- und ein Zeitstempel zugeordnet, die angeben, welche abgeschlossene Transaktion die Datei zuletzt gelesen bzw. geschrieben hat. Bei kurzen und zeitlich ausreichend auseinander liegenden Transaktionen, wird der Zeitstempel einer Datei bei einem Zugriff durch einen Prozess kleiner (älter) sein als der der aktuellen Transaktion. Damit werden die Transaktionen in der richtigen Reihenfolge verarbeitet. Wenn die Reihenfolge nicht korrekt ist, bedeutet dies, dass eine Transaktion die später gestartet wurde als die aktuelle, auf die Datei zugegriffen und sie geändert hat. Damit kommt die aktuelle Transaktion zu spät und muss abgebrochen werden. Diese Methode ist ebenfalls frei von Deadlocks und wie die Methode von Kung und Robinson in gewisser Weise optimistisch. Ein großes Problem bei dieser Methode ist ihre Komplexität und damit ihre relativ geringe Performance.

Deadlocks

Sind ähnlich mit denen in Einprozesorsystemen, sind aber schlimmer! Sie sind schwieriger zu verhindern, zu vermeiden oder sogar zu entdecken und, weil die relevante Informationen über viele Rechner verstreut ist, schwieriger zu beheben.

Alle 4 Vorbedingungen müssen für ein Deadlock erfüllt sein:

1. *Wechselseitiger Ausschluß* (mutual exclusion): Betriebsmittel sind exklusiv benutzbar, d.h. jedes Betriebsmittel ist entweder an genau einen oder an keinen

Prozess zugewiesen

2. *Nichtunterbrechbarkeit* (non preemption): Bereits zugewiesene Betriebsmittel können Prozessen nicht entzogen werden. Prozesse geben bewilligte Betriebsmittel selbständig zurück, wenn sie sie nicht mehr benötigen, spätestens aber bei ihrer Termination

3. *Wartebedingung* (hold & wait): Es können Prozesse existieren, die Betriebsmittel halten während sie auf die Freigabe weiterer Betriebsmittel durch andere Prozesse warten.

4. *Kreisbedingung* oder zyklische Wartebedingung (circular wait) Es existiert eine geschlossene Kette von Prozessen, in der jeder Prozeß auf die Freigabe eines Betriebsmittels wartet, das an seinen Nachfolger in der Kette zugewiesen ist.

Es gibt vier Strategien mit Deadlocks umzugehen:
1. Ignorieren des Problem Der Vogel-Strauß-Algorithmus
2. Erkennung
3. Vermeidung
4. Verhinderung wird in verteilten Systemen nicht verwendet.

Deadlock-Erkennung Zentraler Ansatz
− Jeder Rechner verwaltet die Betriebsmitteln eigene Prozesse. Zusätzlich verwaltet ein Koordinator die Betriebsmitteln für das Gesamtsystem. Entdeckt der Koordinator einen Zyklus, so bricht er zur Auflösung des Zyklus einen Prozess ab. Im Unterschied zu einem zentralen System müssen hier die einzelnen Komponenten ihre Betriebsmitteln explizit an den Koordinator senden.
⊕ Die gesamte Information wird an einer einzigen Stelle gesammelt und verwaltet.

Deadlock-Erkennung Verteilte Ansatz
− Prozesse können gleichzeitig mehrere Ressourcen (lokal, global) anfordern.
⊖ Prozesse müssen auf zwei oder mehr Ressourcen warten.

Deadlock-Vermeidung
Wait-Die Algorithmen: Wenn der alte Prozess ein Betriebsmittel, das durch einen jungen Prozess belegt ist anfordert, dann muss er warten aber umgekehrt wird die Transaktion unterbrochen.
Wound-Wait Algorithmen Wenn ein alter Prozess ein Betriebsmittel, das von einem jungen belegt ist anfordert, dann wird die Transaktion unterbrochen und ungekehrt muss gewarten werden.

Wahl-Algorithmen

Wenn der Koordinator Ausfällt benutzt man folgende Algorithmen:

• **Bully-Algorithmus** "Der Stärkste gewinnt"

Gegeben sind n Synchronisationsprozesse $S_1, S_2, \ldots S_n$. Der Bully-Algorithmus wählt immer denjenigen Synchronisationsserver mit dem höchsten Index als Master aus, d.h.

der Master ist S_m mit $m = max$, $\{i|1 \leq i \leq n, A_i$ aktiv$\}$. Bemerkt ein Prozess S_i, dass der Master ausgefallen ist, wird der Bully-Algorithmus gestartet:

1. S_i schickt eine Panikmeldung an alle $S_j, j > i$ und wartet ein Zeitintervall T lang auf Antwort. Die Panikmeldung wird auch an den alten Master geschickt, um ausgefallen Server von überlasteten Servern zu unterscheiden.

2. Erhält S_i kein Reply, bestimmt S_i sich selbst zum neuen Master und benachrichtigt alle Prozesse $S_j, j < i$. Erhält S_i ein Reply, wartet er ein weiteres Zeitintervall T' auf die Bestätigung, dass ein Prozess $S_j, j > i$ neuer Master ist. Trifft innerhalb von T' keine Antwort ein, wird Schritt 1 wiederholt.

Somit wird sichergestellt, das immer derjenige Server mit dem größten Index Master wird. Auch im Falle, dass ein abgestürzter Synchronisationsserver S_i wieder aktiv wird, muss er den aktuellen Master kennen.

Der Bully-Algorithmus erhöht zwar die Fehlertoleranz des zentralen Ansatzes, doch dieser Vorzug wird leider mit hohem Kommunikationsaufwand bezahlt. Bei n Synchronisationsservern im Worst-Case-Fall ist die Komplexität $\mathcal{O}(n^2)$
Dieser Aufwand ist akzeptabel, dass es sich um einen Algorithmus handelt, der in seltenen Fehlersituationen benutzt wird.

Ring-Algorithmus

Ein weiterer Wahl-Algorithmus basiert auf der Verwendung eines Rings, diesmal aber ohne Token. Wir nehmen an, dass die Prozesse sich in einer physikalischen oder logischen Reihenfolge befinden, so dass jeder Prozess weiß, wer sein Nachfolger ist. Wenn ein Prozess feststellt, dass der Koordinator nicht mehr arbeitet, erzeugt er eine WAHL-Nachricht, die seine eigene Prozessnummer enthält, und sendet diese Nachricht an seine Nachfolger. Wenn der Nachfolger nicht aktiv ist, dann überspringt der Sender den Nachfolger und geht zum nächsten Mitglied im Ring oder zum übernächsten, bis er schließlich einen laufenden Prozess gefunden hat. Bei jedem Schritt fügt der Sender der Liste seine eigene Prozessnummer hinzu.

Irgendwann kommt die Nachricht zurück zu dem Prozess, der sie ausgegeben hat. Der Prozess erkennt das, wenn eine Nachricht ankommt, die seine eigene Prozessnummer enthält. Zu diesem Zeitpunkt wird der Nachrichtentyp in Koordinator geändert, und die Nachricht wird erneut auf den Weg geschickt, diesmal, um alle darüber zu informieren, wer der Koordinator ist (das Listenelement mit der höchsten Nummer) und welche Mitglieder der neue Ring umfasst. Wenn diese Nachricht einmal durchgelaufen ist, wird sie entfernt, und jeder setzt seine Arbeit fort.