Zusammenfassung „Betriebssysteme II" WS2000/2001

Michael Jaeger (michael.jaeger@in-flux.de)

29. Oktober 2001

Inhaltsverzeichnis

1 Verteilte Systeme

Vorteile:

- **Modularität** und **Flexibilität**,
- **Inkrementelles Wachstum** (Erweiterung durch einfaches Hinzufügen einer Komponente möglich),
- **Ressource Sharing**,
- **Leistungs-** und **Lastverbund**,
- **Verfügbarkeit**,
- **Wirtschaftlichkeit**.

Nachteile:

- **Komplexität** (Separation, Parallelität, Heterogenität, Autonomie),
- **Wirtschaftlichkeit** (Gesamtkosten),
- **Schwierige Programmierung** (echte Parallelität, Kommunikation, Test),
- **Sicherheit**.

Probleme, die in einem verteilten System auftreten, und nicht gelöst werden können:

- **Signallaufzeiten** sind nicht deterministisch,
- **Zustandsinformation** des Systems verteilt sich auf mehrere Knoten,
- keine **globale Uhr**,

- Entscheidungen in Knoten können nur auf Basis des dort verfügbaren Wissens getroffen werden.

Entwurfsentscheidungen:

- **System dezentral** konstruieren (→„**single point of failure**"),
- **Skalierbarkeit** gewährleisten,
- „**information hiding**" / „**need to know**",
- **Transparenz** (Mechanismen, die das System verwendet, sollten soweit wie möglich vor dem Programmierer/Benutzer verborgen bleiben) → ist natürlich nicht immer möglich (z.B. bei Kontext-bewussten Anwendungen oder adaptiven Systemen).

Hardware (Klassifikation nach Anweisungs- und Datenströmen):

- SISD: Ein-Prozessor
- SIMD: Array-Prozessor
- MISD: Prozessor?
- MIMD: Mehr-Prozessoren Systeme ((N)UMA) oder Multi-Computer (Netzwerke ohne gemeinsamen Speicher).

Die Vernetzung mehrerer Rechner kann über folgende Topologien realisiert werden:

- Bus, Ring, Gitter, Kreuzschiene, Hypercube.

Mikrokern:

- Minimale Funktionalität im Kern → klare Struktur, leicht verteilbar
- Fast alle Dienste laufen als Anwendungen
- die IPC findet über Nachrichten statt
- Aufgaben des Mikrokern: Prozessabstraktion, IPC, Prozessumschaltung, elementare Speicherverwaltung und I/O
- Problem: Performance
- Bsp.: Amoeba, Mach, QNX

1.1 Verteilte Basisalgorithmen

Aufgrund der Verteilung des Betriebssystems treten bisher nicht bekannte Probleme auf:

- Verteilung/Einsammeln von Informationen an Prozesse
- Aufbau einen kostenminimalen spannenden Baumes
- Bestimmung eindeutiger Prozesse
- Schnappschüsse des globalen Zustandes
- Feststellung der globalen Terminierung eines Vorgangs (Bsp.: ggT)
- Berechnung von Routing-Informationen.

Algorithm 1 Verteilter ggT-Algorithmus

Jeder der n Prozesse beginnt mit eigener Zahl $z \in \mathbb{N}$. Dann werden alle Prozess in einem Ring angeordnet, so dass in Prozess i der folgende Algorithmus angewendet werden kann:

1. Sende(z, linker Nachbar);
2. Sende (z, rechter Nachbar);
3. e =empfangen(von links oder rechts);
4. **if** $e < z$ **then**
   (a) $z :=$ mod $(z - 1, e) + 1$;
   (b) **goto** 1.

Nach Beendigung des Algorithmus haben alle Prozesse den gleichen Wert.

Echo-Algorithmus:

- Variablen "Initiator", "erhalten", "Anzahl".

2 Kommunikation

7-Schichten-Modell (Anwendung, Darstellung, Sitzung, Transport, Vermittlung, Sicherung, Bitübertragung) ist zu ineffizient. Außerdem werden Multi-/Broadcast-Netze benötigt und die Protokolle dürfen nur einen sehr geringen Overhead haben. In der einfachsten Version werden lediglich in Schicht 1 und 2 und eine Request/Reply-Steuerung benötigt.

IPC-Modelle:

Algorithm 2 Echo-Algorithmus

Dieser Algorithmus dient dem Versenden einer Nachricht an alle Prozesse in einem beliebigen zusammenhängenden Graphen.
Wenn der Initiator startet werden folgende Initialisierungen vorgenommen:

1. Initiator := TRUE;
2. erhalten := TRUE;
3. Anzahl := 0;
4. schicke Nachrichten an alle Vorgänger.

Beim Empfangen einer Nachricht von Prozess p wird in den einzelnen Prozessen folgender Algorithmus durchlaufen:

1. **if not** erhalten **then**
   (a) erhalten := TRUE;
   (b) Anzahl := 0;
   (c) Vorgänger := p;
   (d) schicke Nachrichten an alle Nachbarn ausser p;

Wenn ein Prozess ein Echo empfängt wird folgende Prozedur durchlaufen:

1. Anzahl := Anzahl+1
2. **if** Anzahl = #Nachbarn **then**
   (a) erhalten := FALSE;
   (b) **if not** Initiator **then**
      i. schicke Echo an Vorgänger;
   (c) **else**
      i. fertig.

- **Message Passing** (`send/receive`, blockierend/nicht-blockierend, gepuffert/nicht-gepuffert, bestätigt/unbestätigt), **RPC**, **RMI**, **virtueller gemeinsamer Speicher**.

2.1 Remote Procedure Call

Client-Server-Modell: Client ruft über einen `Stub` eine Funktion auf, die auf einem Server ausgeführt wird:

1. **Client-Prozedur** ruft Client-Stub auf
2. **Client-Stub** baut Nachricht zusammen und schickt diese an den Kern (System Call)
3. **Client-Kern** schickt die Nachricht an den Empfänger
4. **Server-Kern** gibt die Nachricht an den Server-Stub
5. **Server-Stub** packt die Nachricht aus und schickt ruft die Prozedur auf
6. **Server-Prozedur** wird ausgeführt und gibt die Ergebnisse an den Server-Stub
7. **Server-Stub** baut Nachricht zusammen und schickt diese an den Kern (System call)
8. **Server-Kern** schickt die Nachricht an den Client-Kern
9. **Client-Kern** übergibt die Nachricht an den Client-Stub
10. **Client-Stub** packt die Ergebnisse aus und gibt die Kontrolle an den Client zurück.

Bestandteile eines RPC:

- **Schnittstellen-Beschreibung** (Prozedurname, Parameter etc.)
- **Aufrufer (Client)** ("Absender")
- **Aufgerufener Dienst** ("Empfänger")
- **Stubs** (Platzhalter)
- **Laufzeitsystem** (Schnittstelle zu Betriebssystem und Kommunikation – "Middleware")
- **Infrastruktur** (Dienste, die den RPC unterstützen)
- **Kommunikationsnetz** (Datentransport).

Ein RPC kann sowohl **synchron** (**blockierend**), als auch **asynchron** durchgeführt werden. Synchronizität bringt den Nachteil, dass die Parallelität nicht genutzt wird – parallele Anfragen an mehrere Server können aber durchaus durchgeführt werden.

Bei den übergebenen **Parametern** muss darauf geachtet werden, dass keine Referenzen auf Objekte übergegeben werden, da verschiedene Prozesse i.d.R. getrennte Adressräume haben. Komplexe Datentypen können also nicht übergeben werden, es sei denn, sie werden auf Basisdatentypen reduziert.

Durch den Ausfall von Prozessen können beim Ausfall des Aufrufers **„Waisen"** entstehen. Für die **Aufruf-Semantik** wurden folgende Strategien entwickelt, die von der Anwendung abhängen:

- **exactly once**, **at least once**, **at most once**, **maybe**.

Bei einem Timeout stellt sich die Frage, wie der Client reagieren soll, da verschiedene Ursachen vorliegen können: Anfrage/Antwort gingen verloren, Netzwerk ist blockiert, Server ist abgestürzt, Server arbeitet noch. Beim verteilten ggT-Algorithmus ist eine Terminierung ohne weiteres z.B. nicht feststellbar.

Erweiterungen des klassischen RPC:

- **asynchrone Aufrufe**, **Futures**, **call-by-copy/restore**, **Delegation** (sichere Objektreferenzen durch copy/restore), **Pipes**, **Broadcast RPC** (DCE RPC).

Damit die Performance von RPC's in verteilten Systemen gut ist, werden folgende Mechanismen eingesetzt:

- kein teures Kopieren von Daten
- sogar weniger Aufwand als UDP (Prüfsummen, weniger Felder)
- Effiziente Timer-Steuerung
- sparsames Umgehen mit Prozesswechseln
- Übertragungszeit im Netz vergrössern.

Leider ist eine vollständige Transparenz des Netzes kaum zu erreichen und das Client-Server-Modell ist für derartige Systeme auch nicht immer adäquat.

Um Anwendungen flexibel zu halten, soll das Binden der Clients an den Server auch dynamisch erfolgen können: Dazu wird ein **Verzeichnis** benötigt, in dem ein Server sein Angebot über **Namen, Versionsnummer, Handle** und **eindeutige Kennung** anbietet. Der Client fragt dann beim Verzeichnis nach dem gewünschten Dienst nach und bekommt ein **Handle** zurück zusammen mit einer Serveradresse und Protokollinformationen. Mit diesen Informationen kann sich der Client an den Server **binden**. Dieser Vorgang ist sehr aufwendig und sollte sparsam eingesetzt werden (→**dynamisches Binden**).

2.2 Gruppenkommunikation

Im Gegensatz zum Client-Server-Modell, das ein $1:1$-Kooperationsmodell darstellt, gibt es Anwendungen, die ein $1:n$-Kooperationsmodell benötigen (z.B. bei Abstimmungen oder dem Update von Replikaten).

Unter einem **Broadcast** verstehen wir die Versendung von Nachrichten an **alle** Teilnehmer und unter einem **Multicast** verstehen wir die Versendung einer Nachricht an einen ausgewählten Kreis der Teilnehmer. **Punkt-zu-Punkt** Kommunikation findet zwischen genau zwei Knoten statt.

Wieder ergeben sich Entwurfsfragen bei der Realisierung von $1:n$-Kommunikation abhängig davon, welche Möglichkeiten das Kommunikationssystem zur Verfügung stellt:

- **Multicast:** eindeutige Multicast-Adresse für Gruppen (Adressierung durch die Adresse)
- **Broadcast:** Gruppenkommunikation durch Filterung der Pakete (Adressierung durch Prädikat P)
- **Punkt-zu-Punkt:** es muss an jeden Teilnehmer ein Paket verschickt werden

Neben diesen Varianten stehen natürlich immer noch die bereits bekannten Entwurfsprobleme (gepuffert? blockierend? etc.). Auch die Frage, wer mit Gruppen kommunizieren darf (geschlossene Gruppen vs. offene Gruppen) muss geklärt werden.

Die Struktur einer Gruppe kann **hierarchisch** (Koordinator: **central point of failure**) oder gleichberechtigt sein (führt u.U. zu Inkonsistenzen). Auch die **Verwaltung der Gruppenmitgliedschaft** muss geklärt werden (anlegen/löschen, Ausfall, Mitglied in mehreren Gruppen). Wieder kann diese Verwaltung zentral und dezentral organisiert sein:

- **Zentrale Verwaltung:** Ein Gruppenserver übernimmt die gesamte Gruppenverwaltung
- **Verteilte Verwaltung:** Alle Mitglieder beteiligen sich an der Verwaltung (kann zu Inkonsistenzen, unsicheren Zustandsinformationen und komplexer Fehlerbehandlung führen).

RPC und Gruppenkommunikation passen nicht so recht zueinander, da bei RPC's immer genau eine Antwort erwartet wird – bei der Gruppenkommunikation können aber bis zu n Antworten eintreffen.

Auch stellen sich bzgl. der **Semantik** des Multicast neue Fragen:

- Wer empfängt die Nachricht tatsächlich?

- In welcher Reihenfolge treffen die Nachrichten bei den Gruppenmitgliedern ein?
- Wie steht es mit der Reihenfolge bei Nachrichten von verschiedenen Sendern?

Bei Gruppen ist es sehr schwer die **Atomizität** sicherzustellen, da dynamische Veränderungen bei Gruppen nicht immer berücksichtigt werden können. **Atomische Broadcasts** sind aber sehr wichtig, um das System in einem konsistenten Zustand zu halten. Eine Lösung wäre das **kontrollierte Fluten**, was allerdings sehr ineffizient ist (→Echo-Algorithmus).

Die Reihenfolge, in der Nachrichten bei Mitgliedern eintreffen können kann **inkonsistent**, **absolut** oder **serialisiert** sein. Dabei darf nicht die Kausalität verletzt werden – die Ursache muss also vor der Wirkung beobachtbar sein. Bei kausal unabhängigen Ereignissen müssen dagegen keine Reinfolgegarantien gegeben werden. Für die Notation von Kausalitäten wurde folgende Schreibweise vereinbart:

Transitiv, irreflexiv, asymmetrisch

- $e_1 < e_2$
  e_2 ist von e_1 **kausal abhängig** $\Leftrightarrow$ e_1 kann Auswirkungen auf e_2 haben
- $e_1 \not< e_2 \wedge e_2 \not< e_1$
  e_1 und e_2 sind **kausal unabhängig**
- $e_1 < e_2 \wedge e_2 < e_3 \Rightarrow e_1 < e_3$
  Kausalitäten sind transitiv und die Abhängigkeit definiert eine **partielle Ordnung** auf den Ereignissen.

reflexiv!

In einem Prozess sind nacheinander stattfindende Ereignisse kausal abhängig voneinander (z.B. das Senden/Empfangen von Nachrichten).

Die **Ordnungsgrade** beim Multicast können wie folgt eingeordnet werden (locker bis streng):

- **FIFO-Ordnung:** Nachrichten an eine Gruppe kommen bei den Empfängern in der Sendereihenfolge an
- **Kausale Ordnung:** gilt $m_1 < m_2$, dann erscheint bei den Empfängern erst m_1, dann m_2
- **Totale Ordnung:** es gilt die kausale Ordnung und alle Nachrichten, die nicht kausal abhängig sind werden von allen Gruppenmitgliedern in der gleichen (beliebigen) Reihenfolge gesehen

Ein Beispiel für den Einsatz von Gruppenkommunikation ist das ISIS-System, das einen Aufsatz auf UNIX darstellt. In ISIS wird zwischen verschiedenen Stufen der Kopplung eines verteilten Systems unterschieden:

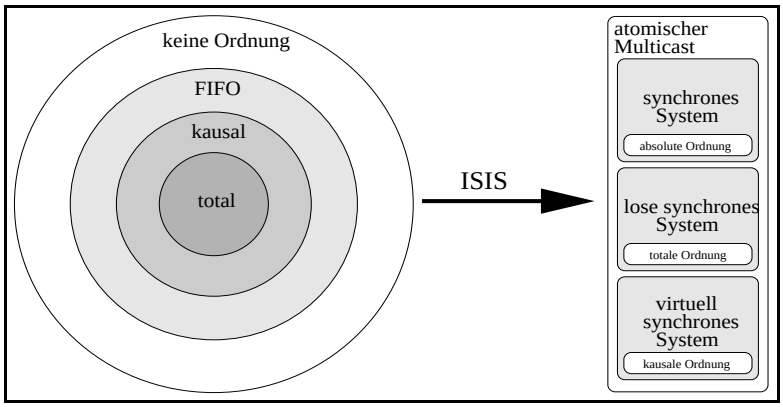

Abbildung 1: Ordnungsgrade beim Multicast

- in **synchronen** Systemen werden alle Ereignisse streng sequentiell durchgeführt (absolute Ordnung, atomischer Broadcast), ist jedoch nur von theoretischem Interesse
- in einem **lose gekoppelten** System sehen alle Beteiligten die Ereignisse in exakt gleicher Reihenfolge (totale Ordnung, atomischer Broadcast)
- die Beteiligten eines **virtuell synchronen** Systems sehen kausal zusammenhängende Ereignisse in exakt der gleichen Reihenfolge (kausale Ordnung, atomischer Broadcast

ABCAST (lose synchron)

Ein Beispiel für die Kopplung ist der **ABCAST** in ISIS (Multicast mit aktueller Uhrzeit, Gruppenmitglieder justieren ihre Uhren wenn nötig und schicken Zeitstempel zurück, Sender wählt größten Zeitstempel und schickt ein `commit` an die Gruppenmitglieder – damit erhalten alle Anwendungen die Nachrichten in der gleichen Reihenfolge). Dabei handelt es sich um ein lose synchrones System.

CBCAST (virtuell synchron)

Ist die Anzahl der Prozesse immer bekannt?

Ein andere Broadcastoperation ist der **CBCAST**: hier pflegt jeder Sender einen Vektor L mit i Komponenten und $i = |\text{Prozesse im System}|$, wobei L_i der (logische) Zeitstempel der letzten Nachricht von i ist („Vektorzeit"), der jeder Nachricht beim Versenden angehängt wird. Wird eine Nachricht mit V von Prozess x an y geschickt, so überprüft y, ob gilt $V_x = L_x + 1$ (alle Nachrichten von x vor der aktuellen wurden empfangen) und $V_y \leq L_y$ (der Prozess x ist auf dem aktuellen Stand, der Rest darf nicht weiter sein – kausale Abhängigkeiten!). Ist das nicht der Fall, so ist der Absender schon „weiter in der Zeit" als der Empfänger. In diesem Fall wird die Nachricht vorerst nicht an den Empfängerprozess weitergereicht. Dieses System wird als **virtuell synchron** bezeichnet.

Das Kommunikationsmodell in verteilten Systemen in meist nachrichtenbasiert oder fußt auf dem RPC. Bei der Verwendung von Prozessgruppen ergeben sich Fragen bzgl. der Verwaltung, Atomizität und Konsistenz der Reihenfolge. In ISIS werden verschiedene Konsistenzmodelle unterstützt.

3 Verteilte Basisalgorithmen

Zentrale Lösungen in verteilten Systemen sind schwierig zu realisieren, da es **keinen gemeinsamen Speicher**, keine globale Zeit und (in der Regel) **nur asynchronen Nachrichtenaustausch** gibt. Synchronisation ist wichtig für **Deadlockvermeidung**, **Uhrensynchronisation** und **wechselseitigen Ausschluss** – und vor allem für **atomische Transaktionen**.

3.1 Uhrensynchronisation

Zeitsynchronisation ist vor allem für die Beurteilung der **Ereignisreihenfolge** notwendig. Für „Zeitpunkte" gibt es folgende Modelle/Eigenschaften, von denen wir die für uns wichtigen auswählen müssen:

- **transitiv**, **irreflexiv**, **linear** (ergibt eine **irreflexive totale Ordnung**)
- **kontinuierlich**, **unbeschränkt** ($\forall x \exists y : x < y$, $\forall y \exists x : x < y$)
- **homogen**
- **dicht** (jeder Zeitpunkt wird erreicht: $\forall x, y : x < y \, \exists z : x < z < y$).

Reden wir von Zeitpunkten in verteilten Systemen, so meinen wir in der Regel die **Einhaltung der Reihenfolge von Zugriffen** – die absolute Zeit spielt dort meist keine Rolle. Eine **zentraler Synchronisationspunkt** ist nicht akzeptabel und führt zu der Verwendung von **logischen Uhren**, die eine **virtuelle Zeit** messen, die nicht – wie die Realzeit – unbeeinflussbar vergeht, sondern als Folge von Ereignissen wahrgenommen wird.

Logische Uhren von Lamport

Bei **Lamport's logischen Uhren** wird die Relation $a \to b$ als „Ereignis a liegt vor Ereignis b" definiert. Diese Relation ist **transitiv** und **nicht-reflexiv** und a kann b kausal beeinflussen. Wenn $a \in P_1$, $b \in P_2$ und keine Nachricht zwischen P_1 und P_2 verschickt wird, so handelt es sich bei a und b um nebenläufige Ereignisse.

Die Funktion $C\,(a)$ gibt den logischen Zeitpunkt eines Ereignisses a an und muss die folgenden **Uhrenbedingungen** erfüllen:

1. **Interne Ereignisse:** $a, b \in P_1 : a \to b \Rightarrow C_1\,(a) < C_1\,(b)$
2. **Externe Ereignisse:** $a = (\mathbf{SEND}\ m\ \mathbf{in}\ P_1)$ und $b = (\text{RECEIVE}\ m\ \text{in}\ P_2) \Rightarrow C_1\,(a) < C_2\,(b)$
3. $C\,()$ ist monoton steigend.

Die Zeit $C\,()$ bildet also eine partielle Ordnung auf der Menge kausal abhängiger Ereignisse. Eine total Ordnung könnte z.B. durch eine Prozessnr. in der Zeit erreicht werden.

Jedes Ereignis in einem Prozess schaltet die logische Uhr weiter und alle Nachrichten tragen einen **Zeitstempel** t_m. Sei L_1 die logische Uhrzeit von Prozess P_1:

- Eine Nachricht m wird in P_1 geschickt: $t_m := L_1$ und $L_1 := L_1 + 1$
- Eine Nachricht m wird in P_1 empfangen: $L_1 := \max(L_1, t_m) + 1$.

Jede Nachricht trägt also einen Zeitstempel, der größer ist als alle bisher gesehenen. Damit erfüllt die Uhr von Lamport die Eigenschaften 1 (es existiert eine partielle Ordnung auf der Menge kausal zusammenhängender Ereignisse), 2 und 3 (es existiert eine totale Ordnung, da die Uhren beim Empfangen von Nachrichten entsprechend gestellt werden).

Ein Problem bei der Lamport-Zeit ist, dass voneinander unabhänge Ereignisse, die parallel zueinander stattfinden, Zeitstempel erhalten, als würden sie in einer bestimmten Reihenfolge ausgeführt werden. Das kann in einigen Anwendungen (z.B. beim verteilten Testen) zu Problemen führen. Es wird also einem verteilten System ein lineares Zeitkonzept aufgeprägt, das es dort eigentlich überhaupt nicht gibt.

Bei der Synchronisierung der **physischen Zeit** ergeben sich eine Reihe von Problemen, da schon die Zeitgeber in den Computern nicht exakt laufen. Der Faktor, um den die Uhr falsch läuft wird i.d.R. vom Hersteller mit der sog. **Driftrate** ρ angegeben:

$$1 - \rho \leq dC/dt \leq 1 + \rho,$$

wobei C die Uhrzeit und t die **UTC**-Zeit (**Universal Time Coordinated**) ist. Wenn also zwei Uhren nicht mehr also δ auseinander laufen sollen, so ist eine Resynchronisation alle $\delta/2\rho$ Sekunden nötig.

Zeitserver-Modell von Christian

Christian führte 1989 ein Modell ein, bei dem ein **Zeit-Server** die UTC-Zeit empfängt und die Clients die Zeit alle $\delta/2\rho$ Sekunden abfragen. Der Client misst das die Zeit Δ vom Stellen der Anfrage bis zum Erhalt der Antwort und addiert dann $\Delta/2$ zu der Antwort (**Propagationszeit**).

Zeit-Daemon-Modell von Gusella und Zatti

Ein Modell nach **Gusella** und **Zatti** (1989) verwendet ein System ohne Funk-Zeit. Ein **time daemon** fragt periodisch alle Rechner nach ihrer Zeit und berechnet dann die Durchschnittszeit, die er wiederum allen mitteilt.

Bei der dezentralen Uhrensynchronisation gibt es festgelegte **Resynchronisationsintervalle** R und jeder Rechner sendet zu Beginn von R seine lokale Zeit als Broadcast. Nach Ablauf eines Intervalls S berechnet jeder aus den bis dahin eingetroffenen Zeiten die neue Zeit.

DCE Distributed Time Service

In DCE gibt es den **DCE Distributed Time Service**, in dem die Zeit durch

$$\text{Zeit} = \text{UTC} + \text{Verschiebung} \cdot \text{Ungenauigkeit}$$

berechnet wird. Die **time clerks** (Clients) synchronisieren sich periodisch mit allen **time servern** ihrer Zelle – es wird also mindestens ein time server pro Zelle benötigt (meist gibt es aber mehr als zwei). Die Zeit wird dabei aus dem Mittelwert der gemeldeten Zeitintervalle gebildet, wobei völlig falsche Intervalle ignoriert werden. **Globale time server** können für mehrere Zellen arbeiten und auch Signale von außen empfangen.

Eine gute Uhrensynchronisation bringt viele Vorteile in Netzen, z.B. bei der Konsistenzerhaltung von Caches bei verteilten Dateisystemen.

3.2 Wechselseitiger Ausschluss

In Ein-Prozessor-Systemen werden kritische Abschnitte durch **busy waiting**, **Semaphore**, **Monitore** etc. geschützt. Diese Systeme basieren auf einem gemeinsam genutzen Speicher.
Anforderungen für eine Lösung sind:

- **höchstens n Prozesse im KA** (meist $n = 1$)
- **beliebig viele Prozessoren**
- **kein Blockieren eines Prozesses außerhalb eines KA**
- **jeder Prozess kommt irgendwann in den KA.**

Dijkstra: zentraler Koordinator

Dijkstra führte 1965 die Semaphore mit den Operationen P und V ein. Eine Lösung auf Basis von Semaphoren benötigt einen zentralen Koordinator (der z.B. über einen Auswahl-Algorithmus bestimmt werden kann). Dieser teilt dann Anforderungen zu. Problem: central point of failure.

Lamport: P und V über Broadcasts

Ein dezentraler Algorithmus nach **Lamport** (1978) überträgt die Operationen $P()$ und $V()$ als Broadcast zusammen mit der eigenen Prozessnummer, der lokalen Zeit und dem KA. Die Empfänger verschicken ein ACK mit Zeitstempel per Broadcast und der Prozess zählt, wie viele Nachrichten er erhalten hat. Bei allen Operationen muss die **Semaphor-Invariante** mit Initialisierungswert von $S = S_0$

$$\#\{P - \text{Operationen}\} \leq \#\{V - \text{Operationen}\} + S_0$$

gewährleistet werden. Die Prozesse zählen also die P- und V-Operationen mit und benutzen eine Lamport-Uhr. Dazu besitzt jeder Prozess eine **Nachrichtenschlange**, in der empfangene Nachrichten aufsteigend nach Zeitstempeln gespeichert werden. Eine Nachricht ist genau dann **stabil**, wenn von allen Prozessen eine Nachricht mit einem größeren Zeitstempel (also z.B. ein `Ack` oder eine andere Nachricht) vorliegt. Genau dann kann jeder Prozess anhand der gespeicherten Nachrichten entscheiden, ob er in den KA eintreten kann (ob also der letzte Prozess, der vor ihm ein P() gemacht hat ein V() gesendet hat). Ein Nachteil sind die hohe Netzlast pro Eintritt ($3\,(n-1)$-viele Nachrichten für Anfrage, Bestätigung und Freigabe) und die Vorausssetzungen, dass alle Knoten operational und up-to-date sein müssen. Ausserdem zirkuliert permanent ein Token, wenn kein Prozess in den KA eintreten möchte.

Algorithmus von Ricard und Agrawala

Ein Algorithmus nach **Ricard** und **Agrawala** nutzt Lamport's totale Ordnung. Voraussetzung ist, dass das Senden einer Nachricht zuverlässig ist (z.B. durch Verwendung eines bestätigenden Dienstes). Wenn ein Prozess P in einen KA eintreten möchte, dann schickt P eine Broadcastnachricht [KA,$P()$, eigene Nr., eigene Zeit]. Die Empfänger verfahren dann wie in Algorithmus 3 auf der nächsten Seite.

Der Prozess P kann erst in den KA eintreten, wenn ein `OK` von allen anderen Prozessen da ist. Nach Beendigung des KA schickt er ein `OK` an alle Prozesse

Algorithm 3 Algorithmus von Ricard und Agrawala (Reaktion der Empfänger eines Broadcasts)

1. **if** nicht in KA und keine Absicht **then**
   (a) Schicke `OK`.
2. **else if** in KA **then**
   (a) Hebe Anfrage auf und antworte nichts.
3. **else** // es besteht die Absicht
   (a) **if** Zeit(Anfrage)<Zeit(self) **then**
      i. Schicke `OK`.
   (b) **else**
      i. Hebe Anfrage auf und antworte nichts.

in der Warteschlange. Möchte kein Prozess in den KA eintreten, so behält der Prozess mit dem Token dasselbe und wartet, bis ein Prozess in den KA eintreten möchte. Auf diese Art und Weise wird ein permanent zirkulierendes Token verhindert. An der Liste der zuletzt zugeteilten KA's kann der Prozess ausserdem erkennen, ob eine Anfrage eines Prozesses bereits erfllt wurde, oder ob er das Token noch braucht.

Die Hauptprobleme dieses Algorithmus sind die Gefahr des „**Verhungerns**" (z.B. wenn ein Prozess stirbt), die **vielen Nachrichten** pro Eintritt ($2\,(n-1)$ Nachrichten) und die Voraussetzung, dass jeder Prozess **immer aktiv sein muss**, um alle Nachrichten zu verarbeiten. Eine Verbesserung könnte z.B. die Mehrheit der `OK` sein anstelle einer Antwort von allen.

Token Ring

Im **Token Ring** ist das Problem des wechselseitigen Auschluss durch die Verwendung eines Tokens gelöst, so dass immer nur der Prozess den KA betreten kann, der im Besitz des Tokens ist. Nach Beendigung des KA oder bei fehlendem Bedarf wird das Token weitergeben. Probleme stellen **verlorene oder doppelte Token** dar.

Verbesserung des Token Ring nach Ricart und Agrawala

Eine Verbesserung des Token Ring nach **Ricart** und **Agrawala** (1983) betrifft den Aufbau des Tokens: das Token speichert in dem Vektor V in V_k den Zeitstempel für die letzte Zuteilung des KA an Prozess k. Will jetzt ein Prozess in den KA eintreten, so teilt er dies allen anderen Prozessen zusammen mit einem Zeitstempel mit. Besitzt nun ein Prozess P_j das Token und hat den KA beendet, so sucht er V der „bedienten" Prozesse ab und schickt das Token dem nächsten Prozess in der Liste (von $j+1, \ldots, n, 1, \ldots j-1$), für den der letzte Token-Anforderungs-Zeitpunkt größer ist als der letzte Token-Zuteilungszeitpunkt in V. Daraus ergibt sich eine Nachrichtenkomplexität von $(n-1)$ Anforderungen

und einer Nachricht für das Token – macht n Nachrichten.

Wegen des Tokens ist der wechselseitige Ausschluss gegeben. Ein Deadlock wird durch die Anwesenheit des Tokens in immer mindestens einem Prozess verhindert. Da für das Verschicken des Tokens an den nächsten Prozess immer beim Nachbarn in aufsteigender Reihenfolge begonnen wird, kommt jeder Prozess nach endlicher Zeit zum Zug – der Algorithmus ist also fair.

Dem Algorithmus liegt die Annahme zugrunde, dass keine Nachrichten verloren gehen. Sollte das bei Bedarfsanfragen der Fall sein, so arbeitet der Algorithmus weiter, allerdings wird die Anforderung des Clients nicht erfüllt werden, falls dieser nicht mit Timeouts arbeitet. Wenn das Token verloren geht kann es Probleme bis hin zu Deadlocks kommen. In diesem Fall muss das Token wieder hergestellt werden.

Algorithmus von Maekawa (nachrichtenoptimal)

Ein (Nachrichten-)optimaler Algorithmus von **Maekawa** (1985) mit Nachrichtenkomplexität von nur $\sqrt{n}$ ordnet die Prozesse in einem $\sqrt{n} \times \sqrt{n}$-Gitter an. Jeder Prozess P_i muss eine Menge von Prozessen (die **request granting**-Menge R_i) vor Eintritt in den KA um Erlaubnis fragen. In dem Gitter überschneiden sich nun die Mengen R_i, was zu einer Nachrichtenkomplexität von $3 \cdot |R_i|$-vielen Nachrichten führt (jeweils $|R_i|$ Nachrichten für Anforderung, Gewährung und Freigabe, also $2 \cdot |R_i|$-viele Nachrichten vor KA-Eintritt). Ist eine Ressource belegt, so gibt es aufgrund der Anordnungsstruktur immer einen Schnittpunkt.

Vergleich der Algorithmen

Vergleichend ergibt sich damit Tabelle 2.

Algorithmus	#Nachrichten	Kritikpunkte
zentral	2	central point of Failure, election
Lamport	$3\,(n-1)$	n mögliche points of failure
Ricart, Agrawala 1	$2\,(n-1)$	n mögliche points of failure
Ricart, Agrawala 2	n	n mögliche points of failure
Token Ring	maximal $n-1$	verlorenes Token, Prozessausfall
Maekawa	$3\,(\sqrt{n}-1)$	Dedlocks, Konfiguration, Ausfälle

Tabelle 2: Vergleich der Algorithmen für den wechselseitigen Ausschluss

3.3 Auswahl

Bei vielen Algorithmen wird ein Prozess für die Bearbeitung von Sonderaufgaben (z.B. als Koordinator, Tokenmonitor) benötigt. Ziel eines Election-Algorithmus ist, dass der Gewinner und alle anderen wissen, wer gewonnen hat. Ein Algorithmus, der das Election-Problem löst, muss folgende Anforderungen erfüllen:

- Jeder Prozess im System kann den Auswahlprozess starten, ohne dass eine Absprache erfolgt sein muss

- Er muss fehlertolerant sein (der Ausfall eines Systems während des Auswahlprozesses soll den Ablauf nicht gefährden)

Diese Voraussetzungen beinhalten nicht die **Fairness**, dass also kein Prozess übergangen wird. Es werden folgende Annahmen getroffen:

- jeder Prozess hat einen eindeutigen Namen
- es gibt eine totale Ordnung auf den Namen
- jeder Prozess kennt die Namen aller anderen Prozesse
- kein Prozess besitzt ein Wissen über Ausfälle bei anderen Prozessen.

Das Election-Problem ist eng verwandt zu anderen Problemen verteilter Syteme, wie z.B. dem **spannenden Baum**. Das Hauptproblem bei den meisten Election-Algorithmen ist, dass die Knoten nicht wissen, wann der Algorithmus terminiert ist – lösen wir also das Problem der verteilten Terminierung, so haben wir auch gleichzeitig eine Lösung für das Auswahlproblem.

Bully-Algorithmus

In dem 1982 von **Garcia-Molina** veröffentlichten **Bully-Algorithmus** schickt der Sender S eine Nachricht `Neuwahlen` an alle Prozesse mit einer höheren Nummer. Wenn keine Antwort kommt, so ist S der Gewinner. Der Gewinner gibt das Wahlergebnis immer allen bekannt und neu gestartete Prozesse fordern immer zu einer kompletten Neuwahl auf.

Broadcast-Election – löschen von Nachrichten im Ring

Bei der **Broadcast-Election** speichert jeder Prozess den Sieger einer Election in der Variable M. Der Initiator P_i setzt nun $M := i$ und schickt einen Broadcast M. Jeder Prozess überprüft für den einkommenden Wert K ob gilt $M < K$ und setzt in diesem Fall $M := K$ und schickt schließlich einen Broadcast mit K. Wenn der Vorgang terminiert, steht der Sieger fest – bloß: Wann ist der Vorgang genau terminiert?

3.3.1 Ringbasierte Auwahlverfahren

Message Extinction Prinzip

Bei Verwendung des **message-extinction-Prinzip** wird das Problem der verteilten Terminierung dadurch gelöst, dass die Nachricht im Laufe der Wanderung u.U. verändert wird und daran erkannt werden kann, wenn eine Nachricht alle Knoten passiert hat. Bezüglich der mittleren Nachrichtenkomplexität ist es optimal, besitzt allerdings eine sehr schlechte worst-case Nachrichtenkomplexität – vermutlich ist ein hohes Nachrichtenaufkommen jedoch eher selten.

Election auf einem Ring

Auf einem **Ring** kann eine Election folgendermaßen stattfinden: der Initiator schickt eine Nachricht `Neuwahlen` mit der eigenen Nummer los. Der Empfänger fügt seine eigene Nummer an und gibt die Nachricht weiter. Wenn jetzt ein Prozess eine Nachricht empfängt, in der die eigene Nummer enthalten ist, dann verschickt er die Nachricht `Koordinator` zusammen mit der Liste der Prozessnummern und jeder Prozess kann die Nummer des Koordinators (der mit

der höchsten Nummer) ablesen. Insgesamt werden bei n Prozessen und k Initiatoren insgesamt $k \cdot 2 \cdot n$ Nachrichten verschickt und ein kompletter Umlauf pro Initiator benötigt. Diese Verfahren verwendet das oben erwähnte message-extinction-Prinzip.

Election nach Chang und Roberts

Einen ähnlichen Weg gehen **Chang** und **Roberts** 1979, setzen aber voraus, dass alle Prozesse logisch in einem **unidirektionalen Ring** angeordnet sind. Hier wird eine Nachricht `Neuwahlen` in die Runde geschickt und jeder Prozess merkt sich, ob er schon mal dran war (so werden unnötige Nachrichten „geschluckt" – z.B. bei mehreren Initatoren). Außerdem wird geprüft, ob die eigene Nummer größer als die in der Nachricht ist – in diesem Fall wird die Nachricht mit der eigenen Nummer weitergeschickt. Ansonsten wird die Orginalnachricht weiterversendet. Entdeckt ein Prozess seine eigene Nummer in der `Neuwahlen`-Nachricht, so schickt er eine `Gewinner`-Nachricht mit seiner Nummer los. In einer Variable `Teilnehmer` speichert ein Prozess, ob er schon mal dran war. In dem Fall, dass es mehrere Initiatoren gibt, also mehrere `Neuwahlen`-Nachrichten kursieren, verwirft der Prozess eine Nachricht, wenn er bereits einmal dran war und und die eigene Nummer größer ist, als die, die in dem `Neuwahlen`-Nachricht vermerkt ist, da es dann offensichtlich mehrere Initiatoren gibt. Im worst-case werden bei n Prozessen und k Initiatoren $\sum_{i=0}^{k-1}(n-i)$ Nachrichten benötigt, was einer Nachrichtenkomplexität von $O\left(n^2\right)$ entspricht. Im Idealfall bei nur einem Initiator $P_{\max}$ werden nur n Nachrichten benötigt.

Die Nachrichtenkomplexität für das Löschen von Nachrichten im Ring läßt sich mit folgenden Größen berechnen

$$
\begin{aligned}
P(i,j) &= Ws(\text{Nachricht von } i \text{ geht } j \text{ Schritte weit}) \\
P(i,j) &= \frac{\binom{i-1}{j-1}}{\binom{n-1}{j-1}} \cdot \frac{n-i}{n-j} \quad , i < n \\
\mathbb{E}_i &= \sum_{j=1}^{n-1} j \cdot P(i,j) \quad \text{Erwartungswert von Knoten } i \\
\mathbb{E}_n &= n \quad \text{Erwartungswert von Knoten } n \\
\mathbb{E} &= n + \sum_{i=1}^{n-1} \mathbb{E}_i \quad \rightarrow \quad O(n \cdot \log n)
\end{aligned}
$$

Hierbei ist:

$\binom{i-1}{j-1}$	#günstige $(j-1)$-Tupel, deren Zahlen $< i$ sind.
$\binom{n-1}{j-1}$	#mögliche $(j-1)$-Tupel aus $n-1$ Zahlen
$\frac{\binom{i-1}{j-1}}{\binom{n-1}{j-1}}$	es wird maximal j Schritte weit gegangen
$\frac{n-i}{n-j}$	Nach j Schritten wird soll mindestens einer weiter gegangen werden
$\mathbb{E}$	#Schritte, die von einem beliebigen Knoten aus abgehen

Nach j Schritten gibt es also $(n-j)$ mögliche Knoten, die kommen können, es soll aber einer kommen, der $> i$ ist – und dafür gibt es $n-i$ viele günstige Knoten.

Zur Nachrichtenkomplexität beim Algorithmus von Chang/Roberts ist zu sagen, dass er im optimalen Fall eine Laufzeit von

$$n \cdot H_n = n \cdot \sum_{i=1}^{n} \frac{1}{n} \approx n \cdot \ln n$$

und im worst-case Zeit $n\,(n+1)\,/2$ benötigt.

Election in Bäumen

3.3.2 Baumbasierte Auswahlverfahren

Die **Auswahl in Bäumen** erfolgt in drei Phasen:

1. **Explosion:** Der Initiator startet die Explosion und alle Knoten reichen die Nachricht beim erstmaligem Erhalt in alle Richtungen weiter.
2. **Kontraktion:** Die Blätter „reflektieren" die Explosion durch Kontraktionsnachrichten – innere Knoten reflektieren das Maximum über die letzte Kante.
3. **Information:** Verbreitung des Ergebnisses durch Fluten des Netzes.

Bei n Knoten und k Initiatoren ergibt sich damit folgende Nachrichenkomplexität:

- Explosion: $(n-1)+(k-1)$ (bei $k-1$ Bewegungskanten und $n-1$ Kanten insgesamt)
- Kontraktion: $(n-1)+1$ (über alle Kanten und $+1$ Begegnung)
- Information: $(n-1)-1$ (über die Bewegungskante wird keine Nachricht verschickt).

Probabilistische Election

Macht also insgesamt $2n+k-2$ Nachrichten.

Eine **Las Vegas Election** nach **Itai** und **Rodeh** (1981) basiert auf dem Algorithmus von Chang/Roberts und funktioniert auch wenn nicht alle Prozesse bekannt sind, es muss lediglich die Größe des Rings bekannt sein. Jeder Prozess sucht sich **zufällig eine Identität** aus $[1,n]$ aus und alle Nachrichten führen einen **hop count** mit. Wird eine Nachricht mit dem eigenen Namen empfangen, so wird geprüft, ob der hop count gleich n ist. Ist dies nicht der Fall, so gibt es einen Prozess mit gleichem Namen (merken!), wenn nicht, dann ist der Knoten der Sieger (zusammen mit anderen, wenn es doppelte Namen gibt). Sollte es mehrere Sieger geben, so wird eine zweite Election durchgeführt – diesmal nur unter den Siegern und die Nachrichten erhalten eine neue Rundenkennung.

Es gibt also ganz verschiedene Algorithmen je nach Kenntnis der Umgebung (Anzahl der Prozesse, Namen, Topologie, Zuverlässigkeit etc.). Die beste Nachrichtenkomplexität besitzt der Algorithmus von Chang/Roberts mit $O(n \log n)$ – in Bäumen geht es aber noch effizienter (in günstigen Konfigurationen).

Laufzeiten nach Topologien

Zum Vergleich haben verschiedene Topologien nach [Mattern1989, Kapitel 2.3.2] unterschiedliche Nachrichtenkomplexitäten,wie $O(n)$ (Bäume) oder $O(n \log k)$ (Ringe) oder $O(e + n \log k)$ für beliebige Netze – Ringe sind also wegen Ihrer inhärenten Symmetrie relativ „teuer".

3.4 Deadlocks

Die vier notwendigen Bedingungen für einen Deadlock sind nach Coffman:

1. mutual exclusion
2. hold and wait
3. non-preemtable
4. circular wait

Die vier Strategien zum Verhindern von Deadlocks sind

1. Ignorieren
2. Entdecken und beheben
3. Vermeidung durch Aufhebung einer Vorbedingung
4. Vorbeugung bei der Ressourcenvergabe

Wegen der nicht-deterministischen Nachrichtenlaufzeiten ist die Entdeckung von Deadlocks in verteilten Systemen sehr schwierig. Ein zentraler Koordinator kann so nur aufgrund von Ressourcenbelegungen falsche Entscheidungen treffen, da ein „Schnappschuss" der Ressourcenbelegung einen Deadlock anzeigen kann, wo keiner ist.

Verteilte Deadlockerkennung

Für die Entdeckung eines Deadlock haben **Chandy/Misra/Haas** 1983 einen verteilten Algorithmus entwickelt: Zur Anforderung einer Ressource von S schickt P eine Nachricht (m_1, m_2, m_3), wobei m_1 der blockierende Prozess, m_2 der Sender und m_3 der Empfänger ist. Der Empfänger E der Nachricht prüft nun, ob $m_1 = m_3$ ist (dann existiert ein Deadlock). Ist der Empfänger selbst durch die Prozesse F_i blockiert, so schickt er (m_1, E, F_i) an alle F_i.

Verteilte Vorbeugung

Bei der **verteilten Vorbeugung** soll eine der vier Deadlock-Bedingungen gebrochen werden. Dafür benötigt jede Transaktion einen **eindeutigen Zeitstempel** (!). Die **Vermeidung des circular wait** wird dann durch Beachtung der Zeitstempel erreicht: Warten ist nur erlaubt, wenn der blockierte Prozess einen älteren Zeitstempel aufweist, sonst wird der Prozess abgebrochen (hier wird am

besten nicht der Initiator-Prozess sondern derjenige mit der höchsten Nummer beendet). Dadurch, dass die Zeitstempel immer aufsteigend sortiert sind, ist eine zyklische Wartekette nicht mehr möglich.

Die **Vermeidung von non-preemptive** wird durch Einführung eines Transaktionskonzepts erreicht, nach dem Transaktionen abgebrochen und wieder neu gestartet werden können. Dabei unterbrechen ältere Transaktionen jüngere.

Bei der Synchronisation in verteilten Systemen lassen sich die meisten Probleme auf wenige Basis-Fragestellungen herunterbrechen:

- alle informieren
- einen auswählen
- verteilte Terminierung
- globale zeitliche Reihenfolge feststellen

Theorie und Praxis klaffen oft weit auseinander, da sich viele Annahmen in der Theorie als praktisch nicht haltbar erweisen (so ist die Vogel Strauss-Methode in der Praxis immer noch die am meisten eingesetzte).

4 Fehlertoleranz

Wenn wir von Fehlertoleranz reden unterscheiden wir zwischen **Störung**, **Fehler** und **Ausfall**. Nicht jede Störung führt gleich zu einem Ausfall und die Auswirkung von Fehlern sind stark abhängig von der jeweiligen Anwendung: **fault→error→failure**.

Wir unterscheiden die **Fehlerklassen**

- **transiente Fehler** (z.B. Naturkatastrophen)
- **Aussetzer** (z.B. Wackelkontakt, timing Fehler)
- **permanente Fehler** (z.B. Plattencrash, Bug)
- **Fehler-und-Schluss** (**fail-silent fault**, z.B. bei fehlerhaften Komponenten)
- **beliebiger/byzantinischer Fehler** (z.B. zeigt eine Komponente beliebiges fehlerhaftes Verhalten)

Sei

$$p = Ws \text{ (eine Komponente faellt aus)},$$

dann ist die mittlere Zeit bis zu einem Fehler (**mean time before failure**) (entspricht dem Erwartungswert bei der

$$MTBF = 1/p.$$

Da es sich um eine geometrische Verteilung handelt berechnet sich der Erwartungswert, also die MTBF wir folgt:

$$\begin{aligned} E &= \sum_{k=1}^{\infty} kp\,(1-p)^k = p\sum_{k=1}^{\infty} k\,(1-p)^k \\ &= p \cdot \frac{1}{p^2} = \frac{1}{p}, \end{aligned}$$

da gilt

$$\begin{aligned} \sum_{k=1}^{\infty} k \cdot q^{k-1} &= (1-q)^2 \\ \Leftarrow \quad \sum_{k=0}^{\infty} q^k &= (1-q)^{-1}. \end{aligned}$$

4.1 Redundanz

Replikation

Fehlertoleranz kann durch das redundante Vorhalten von Komponenten erreicht werden, wie z.B.

- **aktive Replikation**, bei der alle Replikate produktiv arbeiten (z.B. RAID5) oder mehrere Komponenten das Gleiche tun und ein `Voter` ein Ergebnis auswählt
- **passive Replikation**, bei der Replikate nur im Fehlerfall in Aktion treten (z.B. RAID1 oder Notstromaggregate)

Redundanzgrad

Für die Bestimmung des **Redundanzgrades** müssen die **Fehlerklasse** und die **tolerierte Anzahl gleichzeitiger Ausfälle** von Komponenten festgelegt werden. Unter der **k-Zuverlässigkeit** eines Systems mit n Komponenten verstehen wir, dass das System den Ausfall von k Komponenten tolerieren kann. Außerdem ist das System bei $n = k + 1$ Komponenten im Fall von nicht-byzantinischen Fehlern k-zuverlässig und bei $n = 2k + 1$ (→Abstimmung) Komponenten und byzantinischen Fehlern k-zuverlässig. Ein System mit nicht-byzantinisches System schickt z.B. entweder die richtige Antwort oder gar keine (→Fileserver: `read()`) und ein System mit byzantinischem Fehler fällt Entscheidungen z.B. nach dem Mehrheitsprinzip(→Fileserver stimmen ab, welche Antwort auf einen `read()` die richtige ist).

Probleme von Backup-Systemen

Damit ein Backup-System im Fehlerfall übernehmen kann, muss es stets auf dem aktuellen Stand sein – jede Operation auf dem Primary-System zieht also eine Operation auf dem Backup-System nach sich. Das kann zu folgenden Fehlern führen:

- Auftrag geht verloren

- Ausführung auf dem Server misslingt
- Update geht verloren
- Auführung des Backup misslingt
- `OK` vom Backup-System geht verloren
- Antwort an den Client geht verloren

Primary/Backup-Systeme

Wir unterscheiden deshalb folgende **Primary/Backup-Systeme**:

- **hot standby**: alle Änderungen werden online ausgeführt
- **cold/warm standby**: Änderungen werden gespeichert und periodisch vom Backup-System nachgezogen

Umschalten auf das Backup-System

Bleibt die Frage, wie auf das Backup-System umgeschaltet werden soll. Beim **time-out** soll das Backupsystem einspringen, wenn der Primary nicht auf eine `alive`-Anfrage antwortet. Die **Koordinierung der Übernahme** stellt dann ein Problem dar, schließlich könnte der Server ja nur einen „Aussetzer" haben. Es muss also ein **Agreement** gefunden werden, in der sich zwei Komponenten eine Meinung über den Systemzustand bilden. Nach Übernahme durch das Backup-System muss nun noch der Client informiert werden, was stark von der **Fehlertransparenz** abhängt.

Redundanz bei Softwareentwicklung

Bei der Entwicklung von Software kann auch Redundanz zu erhöhter Qualität führen, indem Module von verschiedenen Programmierteams unabhängig voneinander entwickelt werden. Die Ergebnisse der Module werden dann zur Laufzeit durch **Mehrheitsentscheid** bewertet. Dabei können Probleme entstehen, wie die **zulässige Ungenauigkeit** (z.B. bei numerischer Approximation) oder der Vergleich mehrerer korrekter Lösungen (z.B. bei der Nullstellenberechnung bei Polynomen).

Fehlertoleranter RPC

Ein **fehlertoleranter RPC** wird durch Replikation einer Prozedur auf n Server erreicht. Der Client-Stub sendet dabei seine Anfragen an alle Server und kann dann z.B. die erste Antwort verwenden oder alle Antworten abwarten und vergleichen.

Übereinstimmung bei fehlerhaften Prozessoren

Alle bisherigen Algorithmen sind davon ausgegangen, dass die Prozessoren in den Computern korrekt arbeiten. Für allgemeiner **Agreement-Protokolle** müssen aber **verlorene Nachrichten**, **Prozessorausfälle** ((nicht-)byzantinisch) und **synchrone und asynchrone Kommunikation** berücksichtigt werden, damit ein **Konsens nach endlich vielen Interaktionsschritten** erreicht wird. Ein bekanntes Problem ist das der unzuverlässigen Kommunikation bei Generälen, die einen Angriff verabreden wollen, deren Boten aber feindliche Gebiete passieren müssen, so dass die Nachricht verfälscht werden kann und nie beide gleichzeitig wissen, ob Ihre letzte Nachricht angekommen ist.

Byzantinische Generäle

Bei einer anderen Variante des Problems der **Byzantinischen Generäle** möchten sich n Generäle über ihre Truppenstärken abstimmen, jedoch gibt es m Verräter

unter ihnen. Eine Lösung von **Lamport** (1982) ist, dass jeder General allen anderen seine Stärke mitteilt (der Guten sagen dann die Wahrheit und die Bösen lügen und sagen immer etwas anderes). Im nächsten Schritt teilt jeder General allen anderen seine Ergebnisse mit und jeder nimmt als i-tes Element die Mehrheit der gesehenen Werte oder `unbekannt`. Nach diesem Algorithmus ist also ein agreement bei m fehlerhaften Prozessoren und $2m + 1$ Prozessoren korrekt (es muss also mindesten $3m + 1$-viele Prozessoren geben). Gibt es nämlich weniger Prozessoren, z.B. nur $2m + 1$ viele, so gibt es bei m byzantinischen Prozessoren immer zu der Situation, dass ein Prozess bei der Abstimmung seines eigenen Wertes m falsche Antworten hat, und somit nicht entscheiden kann, ob dieser richtig ist (da keine absolute Mehrheit erreicht wurde). Voraussetzung ist also, dass 2/3 aller Prozessoren korrekt arbeiten und die Kommunikation zuverlässig ist. Nach **Fisher** et al. (1985) ist in einem System mit asynchroner Kommunikation und unbeschränkter Kommunikationsverzögerung kein agreement möglich.

Für kritische Steuerungs- und Informationssysteme ist Fehlertoleranz sehr wichtig. Entscheidend für Maßnahmen sind Annahmen zu Fehlerarten. Redundanz ist die Basis für Fehlertoleranz. Diese ist aber nicht automatisch gegeben, sondern muss explizites Entwurfsziel sein. Das Problem der defekten Prozessoren (**byzantine agreement**) lässt sich unter bestimmten Voraussetzungen lösen.

5 Verteilte Terminierung

Actor-Modell

In einem objektorientierten Ansatz spielt im Gegensatz zur prozessorientierten Programmierung die Parallelität kaum eine Rolle. Als ein Berechnungsmodell für lose gekoppelte parallele Prozesse sei hier das sog. **Actor-Modell** genannt, in dem Prozesse existieren, die untereinander Nachrichten austauschen. Der eigentliche Actor ist eine autonome Berechnungseinheit, die u.U. gleichzeitig mit anderen aktiven Einheiten kommuniziert. Die Berechnungseinheiten (Prozesse) können dabei die Zustände `aktiv` und `passiv` annehmen, wobei nur im Zustand `passiv` Nachrichten empfangen und im Zustand `aktiv` Nachrichten versendet werden können. Dabei werden die Actor nur dann aktiv, wenn sie eine Nachricht empfangen, was in einer **atomaren Operation** vollzogen wird. Berechnungen in einem Actor-System sind i.A. nichtdeterministisch, da Nachrichten eine **Laufzeit** und Prozesse eine **Ausführungszeit** haben. Man unterscheidet zwei Arten von Terminierungen:

- **kommunikationsorientiert**: ein Prozess sendet eine Nachricht nach seiner Terminierung und die Berechnung ist beendet, wenn keine Nachrichten mehr gesendet werden. Ist ein Prozess wirklich fertig, wenn keine Nachricht kommt oder rechnet er noch?

- **ergebnisorientiert**: eine verteilte Berechnung ist fertig, wenn ein **Prädikat** erfüllt ist – das macht nur für **monotone Prädikate** Sinn, die nicht so leicht gefunden werden können. Dafür muss aber der Gesamtzustand festgestellt

werden unter dem Ausschluss der Möglichkeit, dass dieser sich nochmal ändert.

Im beiden Fällen ist eine globale Terminierung lokal schwer festzustellen, da im ersten Fall kein Prozess weiss, ob nicht noch ein anderer eine Nachricht verschicken möchte und im zweiten Fall kein Prozess den gesamten Zustand des Systems kennt. Das Fehlen einer globalen Zeit verschärft das Problem noch.

Kommunikationsorientierte Terminierung

Bei der **kommunikationsorientierten Terminierung** gibt es zwei Arten von Nachrichten: **Basis-Nachrichten** und **Kontroll-Nachrichten**. Dabei können Basis-Nachrichten nur von aktiven Prozessen gesendet werden und ein aktiver Prozess kann jederzeit passiv werden (Achtung: ein aktiver Prozess kann auch blockiert sein oder sich in einer Endlosschleife befinden). Ein passiver Prozess, der eine Basis-Nachricht empfängt, wird wieder aktiv. Eine verteilte Berechnung ist also genau dann terminiert, wenn alle Prozesse passiv sind und keine Nachrichten mehr unterwegs sind. Diese Definition impliziert die Existenz einer globalen Zeit. Was passiert aber, wenn ein Prozess eine `passiv`-Nachricht verschickt, und dann durch eine Basis-Nachricht **reaktiviert** wird? -Dieses Problem ist die Hauptschwierigkeit bei Terminierungsalgorithmen.

Variationen

Auch beim **synchronen Kommunikationsmodell**, wo die Laufzeit von Nachrichten vernachlässigt wird und die Berechnung terminiert, wenn alle Prozesse passiv sind, also keine Nachrichten „unterwegs" sind, besteht das Problem der Reaktivierung. In diesem Fall wird die Terminierung der Berechnung über die Passivität aller Prozesse definiert. Im **Atommodell** wird die Berechnungszeit vernachlässigt, es gibt also keine Zustände. Antworten auf Nachrichten werden unmittelbar nach Erhalt verschickt und die Berechnung ist abgeschlossen, wenn keine Nachrichten mehr kursieren. Diese Methode ist sehr praxisnah und leicht handhabbar.

Wird ein Prozess aktiv, so sendet er eine virtuelle Basisnachricht an sich selbst oder einen anderen Prozess. Diese Nachricht wird erst dann als angekommen angesehen, wenn der Prozess wieder passiv wird. Visualisiert man das Atommodell mit einem Zeitdiagramm, so erkennt man die Terminierung des Systems zu einem Zeitpunkt t daran, dass keine Nachrichtenlinie die vertikale Linie durch t schneidet. In der Praxis kann eine Terminierung durch Kontrollprozesse festgestellt werden, die für jeden Prozess die ein- und ausgehenden Nachrichten überwacht. Zur Feststellung der Terminierung werden dann sog. **Kontrollwellen** erzeugt.

Vektoralgorithmus

Die Idee der Kontrollprozesse wird im **Vektoralgorithmus** verwendet: Jede Prozess zählt die Nachrichten, die er erhalten (positiv) und die er versendet (negativ) hat. Ein **Kontrollvektor** repräsentiert dann den aktuellen Stand des Systems – ist er der Nullvektor, so ist die Berechnung terminiert. Die Vorteile liegen auf der Hand: die Methode ist unabhängig von der Netzwerktopologie und kann nicht „getäuscht" werden. Zum Feststellen der Terminierung wird lediglich eine Kontrollwelle benötigt und die Basisnachrichten bleiben unberührt. Leider werden die Kontrollnachrichten proportional zu der Anzahl der Prozesse länger, da sie den Zustandsvektor mit sich führen müssen.

Im Atommodell überprüft man ob eine Berechnung terminiert ist, indem man einen Schnitt durch das Zeitdiagramm macht und prüft, ob ein Ereignis rechts von der Schnittlinie liegt. In diesem Fall liegt die Ursache links vom Schnitt und ein Prozess hat einen Sendevorgang, der noch nicht ausgeglichen ist $\Rightarrow$ die Berechnung ist nicht terminiert. Für diese Argumentation ist es bequem von einer globalen Zeit auszugehen, was in der Praxis mit synchronisierten Uhren und dem **Alibi-Prinzip** realisiert wird. Damit gilt

$$e < e \quad \Rightarrow \quad t(e) < t(e')$$

Vektorzeit

und wir haben den Begriff **Vektorzeit**. Jeder Prozess besitzt einen **lokalen Zeitvektor** C, dessen Komponenten $C[i]$ bei Ereignissen um 1 erhöht wird. Die Ereigniszeit $C[j]$ von anderen Prozessen wird als **Approximation** augefasst. Die lokalen Zeitvektoren werden global abgeglichen – da eine lokale Approximation nie „vorgehen" kann, wird beim Abgleich immer die größere Zeitkomponente $C[j]$ verwendet.

Die Vektorzeit beachtet also die **Ereignisordnung** und zwar **komponentenweise**. Jeder Prozess hat in seiner Komponente die exakte Zeit und in den anderen eine untere Approximation der anderen Prozesse

$$C_i[i] \geq C_k[i] \ \forall i,k \qquad e \| e' \Rightarrow C(e)[i] \geq C(e')[i] \ (e \text{ ist Ereignis von } P_i).$$

$e \| e'$ bedeutet, dass e und e' **virtuell gleichzeitig** geschehen

Damit ist die Ordnung der Vektorzeit nicht transitiv bei der Parallelität und es gilt $u \leq v :\Leftrightarrow u[i] \leq v[i]$ für alle $i = 1, \ldots, n$. Die **Parallelität** ist definiert durch

$$u \| v \quad :\Leftrightarrow \quad u = v \ \vee \ \underbrace{(\neg(u \leq v) \wedge \neg(v \leq u))}_{\text{Nebenlaeufigkeit}}.$$

Schnitte

Mit einem **Schnitt** möchte man die globale Zeit zu einem Zeitpunkt X feststellen. Dabei unterscheidet man zwischen **konstistenten** und **inkonsistenten Schnitten**. Bei einem konsistenten Schnitt liegt jede Ereignis, das vor einem Ereignis im Schnitt stattfand auch im Schnitt, was bei inkonsistenten Schnitten nicht gegeben ist. Um einen Schnitt durchführen zu können werden **Schnittereignisse** eingeführt. Dabei handelt es sich um **Pseudoereignisse**, die später als alle realen Ereignisse passieren, jedoch noch zum Schnitt gehören. Damit stehen zwei Schnittereignisse eines konsistenten Schnitts nie in kausaler Abhängigkeit, da es sonst mind. ein reales Ereignis im Schnitt geben müsste, das später als das Schnittereignis stattfindet.

Konsistente Schnitte werden in Realzeit als gleichzeitig angenommen. Ein Schnitt X Schnittereignis s_i ist genau dann **konsistent**, wenn

$$t_X = (C(s_1)[1], \ldots, C(s_n)[n])$$

ist. Wäre X inkonsistent, so gäbe es ein Nachricht von P_i nach P_k, so dass $s_i[i] < t[i] \leq s_k[i]$ ist. Das würde dann aber bedeuten, dass

$$t_X > \left(\underbrace{C(s_1)[1]}_{\text{eigene Zeit}}, \ldots, C(s_n)[n] \right)$$

ist – Widerspruch.

Um nun die Terminierung mit der Vektorzeit zu erkennen, müssen die Vektoren t_X entlang der Schnittlinie berechnet werden. Sind sie alle gleich, so liegt ein konsistenter Schnitt vor und es muss noch die Anzahl der empfangenen und versendeten Nachrichten geprüft werden. Ist diese gleich, so liegt eine Terminierung vor.

Die verteilte Terminierung wir also im Großen und Ganzen auf **zwei Modell** reduziert, wobei **einmal die Nachrichtenlaufzeit und andermal die Berechnungszeit vernachlässigt** werden. Aufgrund des Fehlens einer globalen Zeit wird die Vektorzeit als Ersatz verwendet – so ist eine Prüfung auf konsistente Schnitte möglich.

6 Verteilter Speicher

NUMA, NORMA

Bei Mehrprozessorsystemen ist es möglich, einen gemeinsamen Speicher für alle Prozessoren zu realisieren und somit eine **UMA** (Uniform Memory Access) Architektur zu konstruieren. Multicomputer ohne einen gemeinsamen (physischen) Speicher verwenden besondere Mechanismen, um etwas ähnliches wie einen gemeinsamen Speicher zu schaffen. Dort unterscheidet man **NUMA** (Non Uniform Memory Access) und **NORMA** (No Remote Memory Access) Architekturen. Bei der NUMA-Architektur wird ein einziger Adressraum zur Verfügung gestellt, der für alle CPU's sichtbar ist. Dabei ist allerdings nicht ersichtlich, ob benutzter Speicher lokal oder entfernt liegt. Der entfernte Zugriff auf Speicher ist aber teuer, so dass bei der NORMA-Architektur der entfernte Zugriff auf den Adressraum anderer Prozessoren gar nicht erst zugelassen wird.

Verteilter Speicher

Die Einführung von **DSM** (Distributed Shared Memory) soll die Verwendung von verteiltem Speicher für Programmierer transparent machen. Dabei wird der virtuelle Speicher wie gehabt verwaltet: liegt eine Speicher-Seite nicht im lokalen Hauptspeicher, so wird sie von der DSM-Schicht geholt und eingelagert und für den nächsten Zugriff zwischengespeichert. Eine derartige Abstraktion kann zu Problemen führen, da Daten in einen **inkonsistenten Zustand** geraten können und die **Performance** schlecht werden kann, da der Programmierer selbst gar nicht optimieren kann – von der Zuverlässigkeit ganz zu schweigen.

Neuere Techniken versuchen vor allem, den Netzwerkverkehr zu minimieren und die Latenzzeit beim Zugriff auf Speicherseiten zu verringern. Beispielsweise kann DSM über eine Art globale Variablen erreicht werden, so dass der Programmierer selbst optimieren kann, da er dann genau weiss, wann entfernte Zugriffe durchgeführt werden.

6.1 Konsistenzmodelle

Ein großes Problem im DSM ist die **Konsistenz der Daten**. Um das in den Griff zu bekommen, wird eine Art **Vertrag** zwischen dem Betriebssystem/Speicher

und der Anwendungssoftware geschlossen, der Annahmen und Garantien zu Konsistenz der Daten macht – und meist ein trade-off zwischen Geschwindigkeit und Einschränkung ist. Im Allgemeinen werden die folgenden **Konsistenzmodelle** unterschieden, sortiert nach dem Grad der Einschränkung:

- **strenge Konsistenz**: Jede `read`-Operation liefert den zuletzt geschriebenen Wert, alle Schreibvorgänge sind also unmittelbar für alle Prozesse sichtbar (Standardannahme bei nicht-verteilten Systemen).
  ▷ Nachteil: Setzt eine globale Zeit voraus und ist somit in verteilten Systemen nicht praktikabel.

- **sequentielle Konsistenz**: (1979) Die sequentiellen Ausführung der Operationen auf allen Prozessoren in einer beliebigen gültigen Verschachtelung wird von allen Prozessen gleich wahrgenommen – es wird keine Aussage über die Zeit oder den „aktuellsten" Wert gemacht, lediglich **Speicherkohärenz** wird gefordert (eine `read`-Operation liefert immer den Wert der letzten `write`-Operation). Die wiederholte Ausführung eines Programms muss also nicht zwangsläufig das gleiche Ergebnis liefern, falls keine Synchronisierungsoperationen vorgenommen werden.
  ▷ Nachteil: Es gilt $r + w \geq t$, wird die `read`-Performance erhöht, so geschieht das zu Lasten der `write`-Performance (t ist die Übertragungszeit von Paketen zwischen den Knoten) – und das führt dann zu einem echten Performance-Problem.

- **kausale Konsistenz**: (1990) Eine Abschwächung der sequentiellen Konsistenz, wo nur die gleiche Reihenfolge bei Speicherzugriffen garantiert wird, die kausal zusammenhängende Konsistenz.
  ▷ Nachteil: Die Verwaltung kausaler Zusammenhänge erfordert die Pflege eines Abhängigkeitsgraphs, was zu einem beträchtlichen Overhead führt.

- **PRAM-Konsistenz**: (1988) Bei der Pipelined RAM Konsistenz werden Schreibvorgänge von einem einzigen Prozess von allen anderen Prozessen in der gleichen Reihenfolge empfangen, in der sie abgesetzt wurden – Schreibvorgänge unterschiedlicher Prozesse können aber von verschiedenen Prozessen in unterschiedlicher Reihenfolge gesehen werden.
  Nachteil: Speicherkohärenz ist nicht gegeben (Bsp.: P_1 und P_2 killen sich gegenseitig mit a=1, if(b==0) kill P2)

- **schwache Konsistenz**: (1986) Basis dieser Konsistenz ist die Synchronisierung des (gemeinsam benutzten) Speichers durch **Synchronisations-Variablen**. Dabei wird der Speicherzustand erst nach dem Verlassen des KA synchronisiert und ggf. propagiert. Der Zugriff auf Synchronisierungs-Variablen ist dabei sequentiell-konsistent, ein Zugriff auf sie ist erst dann möglich, wenn alle vorhergehenden Schreibvorgänge abgeschlossen sind und ein Zugriff auf Daten ist erst dann möglich, wenn wiederum alle Zugriffe auf Synchronisierungs-Variablen ausgeführt sind. Ein Lesevorgang heißt dabei „ausgeführt", wenn keine darauf folgenden Schreibvorgänge

den Wert verändern und ein Schreibvorgang heißt „ausgeführt", wenn alle folgenden Lesevorgänge den Wert zurückgeben.
▷ Nachteil: Die Transparenz für den Programmierer nimmt ab, über die Synchronisationsvariable kann er jedoch die Aktualisierung des Speichers steuern und erhält dadurch ein Werkzeug zur Performance-Steigerung, was Sinn macht, wenn nur wenige Zugriffe auf gemeinsamen Speicher erfolgen.

- **Freigabe-Konsistenz**: (1990) Eine Differenzierung zwischen der Synchronisierung zu Beginn eines KA und zum Austritt aus einem KA soll eine Effizienzsteigerung bringen – es werden also zwei Arten von Synchronisierungs-Variablen und -Operationen benötigt: Ein `release`-Zugriff besagt, dass ein KA verlassen wird und ein `aquire`-Zugriff wird zum Eintritt in einen KA getätigt. Durch einen `aquire`-Aufruf wird der Prozess also blockiert bis die „überwachten" Variablen konsistent sind. Der `release`-Aufruf sorgt dann dafür, dass Veränderungen an diesen Variablen an alle Prozesse propagiert werden. Wieder muss ein `aquire` vor einer `read`-Operation erfolgen, um zu garantieren, dass sie den korrekten Wert liefert. Die `aquire`'s und `release`'s müssen wieder PRAM-konsistent sein, wobei es einen zentralen Manager gibt, der die Zuteilungen verwaltet.
  Bei der **trägen Release-Konsistenz**, werden nach einem `release` die Variablen nicht an alle Prozesse propagiert, sondern es wird versucht, bei einem `aquire` den aktuellen Wert der Variablen zu holen. Diese Methode ist auch deshalb empfehlenswert, weil sich viele KA's in Schleifen befinden.

- **Eintritts-Konsistenz**: (1993) Jede gemeinsam benutzte Variable ist einer Synchronisations-Variable zugeordnet, so dass die Datenobjekte vor jedem Zugriff konsistent gemacht werden können. Das reduziert den Kommunikations-Overhead und mehrere KA's mit voneinander unabhängig gemeinsam genutzten Variablen werden möglich. Dabei muss der Speicher den folgenden Bedingungen genügen:

  - alle Aktualisierungen der bewachten gemeinsamen Variablen müssen vor einem `aquire`-Zugriff durchgeführt worden sein
  - wenn ein Zugriff im exklusiven Modus auf eine Synchronisierungs-Variable erfolgt, darf kein anderer Prozess die Snchronisierungsvariable halten – nicht einmal nicht-exklusiv
  - vor dem (nicht-)exklusiven Zugriff auf eine Synchronisierungs-Variable müssen alle exklusiven Zugriffe beendet sein.

- ▷ Nachteil: Erhöhter Verwaltungsaufwand und kompliziertere Programmierung.

Zusammenfassung

Wir unterschieden bei Konsistenzmodellen bzgl. der Reihenfolge **Striktheit**, **Sequentialität** und **Kausalität**. Bezüglich der Speicherkonsistenz unterscheiden wir **schwache**, **Freigabe- und Eingangs-Konsistenz** – diese benötigen spezielle Programmierkonstrukte für ihre Realisierung.

6.2 Entwurfsfrage

Granularität

Bei der Implementierung von DSM muss man zunächst die **Granularität** einer Speicher-Transfer-Einheit festlegen (Wort/Seite(n)). Die Wahl der Granularität hängt von der Übertragungsgeschwindigkeit und dem Replikationsverfahren ab. Im Folgenden werden wir Seiten als Speichereinheit verwenden.

Naive Implementierung

Prozesse verwenden einen gemeinsamen Speicher und bei jeder Seite, die nicht lokal verfügbar ist wird ein Trap ausgelöst und die Seite eingelagert. Ein `write` auf die Seite muss dann an den DSM weitergereicht werden.

Replikation

Bei der Replikation können read-only-Bereiche an Prozesse verteilt werden, so dass diese sie lokal im Zugriff haben. Ein Problem stellt sich bei beschreibbaren Seiten, da bei einem `write`-Vorgang die Änderung an alle Kopien der Seite propagiert werden muss – meist werden einfach die entfernten Seiten als ungültig markiert. Diese Strategie wird **Single Write – Multiple Read** genannt. Bleibt die Frage, wie der Besitzer einer Seite gefunden wird. Hier operiert man mit Broadcasts, einem **Page Manager** oder man verfolgt einfach alle Seitenbewegungen und Suchanfragen. Genauso werden zum Finden von Kopien einer Seite Broadcasts oder Page Manager eingesetzt.

Seitenersetzungsstrategie

Die Frage nach der Seitenersetzungsstrategie ist dabei prinzipiell die gleiche wie die in nicht-verteilten Betriebssystemen, mit der Ausnahme, dass Kopien nicht gesichert werden müssen und u.U. der Besitzer der Seite informiert werden muss bzw. wenn der Prozess selbst der Besitzer ist, eine Weitergabe der Besitzrechte organisiert werden muss.

Linda

Eine Erweiterung einer Programmiersprache stellt **Linda** dar, das einen gemeinsam genutzten stark strukturierten Speicher für mehrere Rechner darstellt. Die Basisabstraktion dabei ist ein **Datentupel** im **Tupelraum**, der für den Benutzer wie ein riesiger globaler DSM aussieht. Ein **Tupel** ist dabei eine Datenstruktur, die aus beliebig vielen Feldern bestehen kann, wobei der erste Paramter N (**Aktualparameter**) vom speziellen Typ „Name" ist und die restlichen Parameter (**Formalparameter**) von einem beliebigen Typ sein können. Auf Tupel kann mit den folgenden Operationen zugegriffen werden:

- `out()`: stellt ein Tupel in den Tupelraum
- `read()`: liest ein Tupel (blockierend)
- `in()`: entnimmt ein Tupel (blockierend)
- `eval()`: startet einen Prozess, der auf dem Tupelraum arbeitet, wobei die Argumente parallel ausgewertet werden und das resultierende Tupel wieder im Tupelraum abgelegt wird.

Die Adressierung von Tupeln kann **inhaltsorientiert** erfolgen, also über **assoziative Adressierung**. Dabei passt ein Tupel a zu einem Tupel b, wenn gilt $N_a = N_b$, $|a| = |b|$ und die Formal- und Aktualparameter positionsweise kompatibel sind. Ist eine Adressierung erfolglos, so wird der Prozess so lange blockiert,

bis ein gesuchtes Tupel in den Tupelraum gestellt wird – praktisch bei der Realisierung von Semaphoren. Auch ein **replicated worker model** lässt sich einfach realisieren, indem Arbeiter im Tupelraum mittels `in(„Aufträge",?job)` nach Jobs suchen, die Auftraggeber mittels `out(„Aufträge",job)` eingestellt haben.

JavaSpaces

Eine Wiederbelebung der Linda-Idee stellen **JavaSpaces** dar, bei denen ein JavaSpace (analog zum Tupelraum) (getypte) Objekte verwaltet. Auf diese läßt sich mit den Operationen `read()` und `write()` zugreifen. JavaSpaces und RMI sind Kernbausteine von **Jini**, das die logische Netzwerkinfrastruktur zur Verfügung stellt.

Zusammenfassung

Bei vielen Lösungen für DSM muss sich der Programmierer nicht auf eine neue Abstraktion der Programmierung einlassen. Ein Problem stellt allerdings der Austausch von Ereignissen zwischen Prozessen und die Replikation von Daten mit unterschliedlichen **Konsistenzgarantien** dar. Letztendlich sind alle Lösungen ein trade-off zwischen Performance und Konsistenz.

7 Optimistische Replikation

Schwache Konsistenz wird heute sehr gerne in verschiedensten Systemen eingesetzt, da sie sich durch Hochverfübarkeit, gute Skalierbarkeit und einfaches Design auszeichnen.

Bei **pessimistischen Konsistenzprotokollen** werden `write`-Operationen immer propagiert und verschiedene Konsistenzmodelle verwendet. Bei der **optimistischen Replikation** wird dagegen toleriert, dass Anwendungen temporär inkonsistente Daten sehen. Diese Vorgehensweise ist nicht für alle Anwendungen akzeptabel, bei Bankkonten ist das im Prinzip schon heute der Fall, bei Kalendern kann das aber zu Problemen führen. Ein Anwendungsgebiet für die optimistische Replikation stellen **mobile Systeme** dar, bei denen der Benutzer auf lokalen Kopien weiterarbeitet und diese später synchronisiert.

Bayou

Im Forschungszentrum XEROX PARC wurde 1994 das Projekt **Bayou** gestartet, bei dem mobile Benutzer auf gemeinsame Datenobjekte zugreifen können sollten. Genauso sollte das Arbeiten einzelner Benutzer an verschiedenen Kopien ermöglicht werden. Der Zugriff auf die Replikate war beliebig und `write`-Operationen sollten bei Gelegenheit propagiert werden. Als einzige Garantie bietet Bayou eine **Konsistenzgarantie für Sitzungen**, wobei unter einer Sitzung logisch zusammengehörige Datenoperationen verstanden werden. Angestrebt wurde also ein schwach konsistentes Replikationssystem, das kollaborative Anwendungen in einer mobilen Umgebung unterstützen sollte. Die Replikation soll dann aussehen wie eine Sitzung an einem Server (→**Journalling**). Insgesamt werden vier Konsistenzgarantien für Sitzungen gegeben:

- **read your writes**: `read`-Operationen sehen alle vorangehenden `write`-Operationen der Sitzung.
- **monotonic reads**: `read`-Operationen sehen mind. den Zustand, den vorherige `read`-Operationen sahen.

- **writes follow reads**: eine `write`-Operation einer Sitzung wird immer nach den `read`-Operationen gesehen, von denen sie abhängt.
- **monotonic writes**: alle `write`-Operationen der Sitzung werden von allen Replikationen in der gleichen Reihenfolge gesehen.

7.1 Sitzungsgarantien

Modell

In der Modellvorstellung existieren mehrere Daten-Server, die vollständige Replikate einer Datenbank besitzen. Ein Zugriff von Mobilen Clients auf die Datenbank geschieht durch die Operationen `read(R)` und `write(W)`, wobei $DB\,(S,t)$ eine geordnete Folge von Paaren ist, bestehend aus dem Server S und dem Zeitpunkt des Zugriffs t. Unter $DB\,(S)$ verstehen wir dabei die Zugriffe auf den Server S bis zum aktuellen Zeitpunkt.

Sitzungsgarantien

Wir nehmen im Folgenden an, dass gilt $DB\,(S_1,t) \neq DB\,(S_2,t)$ und dass wir mit schwacher Konsistenz arbeiten. Die Veränderungen von Daten sollen außerdem in endlicher Zeit zu allen Replikaten wandern. Für den Fall, dass gilt `WriteOrder(W1,W2)`, dass also zwei Schreiboperationen in einer festgelegten Reihenfolge durchgeführt werden sollen, muss diese Reihenfolge bei allen Servern eingehalten werden. Dabei werden keine Annahmen über die Realisierung dieser Garantien gemacht. Schreibkonflikte müssen manuell von dem Benutzer gelöst werden. Damit können die bereits besprochenen Sitzungsgarantien formal aufgeschrieben werden:

- read your writes

  $\Leftrightarrow$ ($W < R$ in einer Sitzung $\wedge$ R wird zum Zeitpunkt t auf S ausgefuehrt)

  $\Rightarrow$ $W \in DB\,(S,t)$.

  Beispiele: Passwort-Synchronisation, Mail-Reader (Löschen von Nachrichten)

- monotonic reads

  $\Leftrightarrow$ ($R_1 < R_2$ in einer Sitzung $\wedge$ R_1 greift um t_1 auf S_1 und R_2 um t_2 auf S_2 zu)

  $\Rightarrow$ $\text{RelevantWrites}\,(S_1,t_1,R_1) \subseteq DB_2\,(S_2,t_2)$.

  Unter $\text{RelevantWrites}(S,t,R)$ verstehen wir die kleinste Menge an `write`-Operationen, die die Operation R eindeutig bestimmen.
  Beispiele: Kalender, Mail-Reader (neue Nachrichten)

- writes follow reads

  $\Leftrightarrow$ ($R_1 < W_2$ in einer Sitzung $\wedge$ R_1 greift um t_1 auf S_1 zu)

  $\Rightarrow$ $\forall$Server S_2 mit $W_2 \in DB\,(S_2) : \forall W_1 \in \text{RelevantWrites}\,(S_1,t_1,R_1)$ ist $W_1 \in DB\,(S_2) \wedge \text{WriteOrder}\,(W_1,W_2)$.

  Beispiele: verteilte Bibliographie-Datenbank (Update), betrifft auch andere Benutzer.

- monotonic writes

  $\Leftrightarrow$ ($W_1 < W_2$ in einer Sitzung)
  $\Rightarrow$ $\forall$Server $S_2 : W_2 \in DB(S_2) \Rightarrow W_1 \in DB(S_2) \wedge \text{WriteOrder}(W_1, W_2)$.

  Beispiele: Texteditor (speichert Version n, dann Version $n+1 \Rightarrow n$ ersetzt $n+1$), Update von Bibliotheken und Anwendungen, die diese nutzen

7.2 Implementierung

Zur Implementierung dieser Replikation wird eine global eindeutige **WID** (write ID) benötigt, die jeweils von dem Server vergeben wird, der die Schreiboperation akzeptiert. Unter einem **Read-Set** verstehen wir eine Menge von WIDs der Schreiboperationen, die für die Operationen relevant sind. Analog bezeichnet ein **Write-Set** eine Menge von WIDs der Schreiboperationen der Sitzung. Der **Session-Manager** als Teil des Client-Stubs führt alle `read-` und `write-`Operationen aus und speichert das Read-Set und das Write-Set jeder Sitzung.

Über das Verhalten der Server treffen wir folgende Annahme:

$$\forall W_1 \text{ um } t \text{ auf } S : \text{WriteOrder}(W, W_1)\ \forall W \in DB(S, t).$$

Für Schreib-Operationen gilt außerdem:

W_2 wandert von S_1 nach S_2 um t $\Rightarrow$ Alle $W_1 \in DB(S_1, t)$ mit $\text{WriteOrder}(W_1, W_2)$ werden zu S_2 propagiert.

Die Datenbank wird also **vollständig und geordnet propagiert**. Im Folgenden wird unter „Lesen" (R) und „Schreiben" (W) immer das Lesen/Schreiben von/auf Server S zum Zeitpunkt t verstanden:

7.2.1 read your writes

- Schreiben: Füge WID zum Write-Set hinzu.
- Lesen:
  - Sicherstellen, dass Write-Set Teilmenge von $DB(S, t)$ oder Write-Set zum Server schicken oder Liste der WIDs vom Server zum Client schicken
  - Wenn S nicht erfüllbar, dann anderen Server probieren
  - Falls kein Server gefunden, dann Ausnahme melden.

7.2.2 monotonic reads

- Schreiben: Füge WID zum Write-Set hinzu.
- Lesen:
  - Sicherstellen, dass Read-Set Teilmenge von $DB\,(S,t)$
  - Füge WIDs der `write`-Operationen aus RelevantWrites(S,t,R) zum Read-Set der Sitzung hinzu.

7.2.3 writes follow reads

- Schreiben: Stelle sicher, dass Read-Set der Sitzung Teilmenge von $DB\,(S,t)$ ist.
- Lesen: Füge WIDs der Schreib-Operationen aus RelevantWrites(S,t,R) zum Read-Set der Sitzung hinzu.

7.2.4 monotonic writes

- Schreiben:
  - Stelle sicher, dass Write-Set der Sitzung Teilmenge von $DB\,(S,t)$ ist
  - Füge WID zum Write-Set der Sitzung hinzu.
- Lesen: Stelle sicher, dass das Write-Set Teilmenge von $DB\,(S,t)$ ist.

Tabelle 5 gibt zusammenfassend einen Überblick über die Zustandsänderungen nach `read`-/`write`-Operationen. Der Zustand einer Sitzung besteht hierbei aus dem Write- und dem Read-Set.

	Zustand der Sitzung verändert bei	Zustand der Sitzung überprüfen bei
read your writes	Write	Read
monotonic reads	Read	Read
writes follow reads	Read	Write
monotonic writes	Write	Write

Tabelle 5: Übersicht Zustandsänderungen der Sitzung nach `read`-/`write`-Operationen

7.3 Praktische Implementierung der Garantien

Die Verwendung von WIDs hat in der Praxis einige Nachteile:

- WIDs können sehr lange werden
- die Menge der relevanten WIDs, die als Antwort einer `read`-Operation zurückgeliefert wird, kann sehr groß werden
- die Menge der WIDs, die bei einer `read`-/`write`-Operation geprüft werden muss kann sehr groß werden
- das Log der `write`-Operationen auf den Servern kann sehr groß werden
- das Finden eines passenden Servers für eine anstehende `write`-Operation kann lange dauern
- die Verwaltung der RelevantWrites für eine `read`-Operation kann sehr aufwendig werden.

In der Praxis kann man viele dieser Probleme mit dem Einsatz von **Versions-Vektoren** umgehen, die aus Tupeln $\langle$`Server`, `clock`$\rangle$ bestehen, wobei `clock` die monoton steigende lokale Zeit des `Servers` ist, die sich bei jeder `write`-Operation um eins erhöht ($\Rightarrow$Lamport). Diese Tupel eignen sich sehr gut als WIDs, wobei jeder Server seinen eigenen Versions-Vektor verwaltet. Der Versionsvektor wird dabei wie folgt eingesetzt:

- Um einen Versions-Vektor zu erhalten, der eine kompakte Repräsentation für eine Menge von WIDs Ws enthält, setze $V[S]$ =Zeitpunkt, zu dem die letzte WID von Server S in Ws zugewiesen wurde oder 0.
- Um einen Versions-Vektor zu erhalten, der die Vereinigung zweier Mengen von WIDs Ws_1 und Ws_2 darstellt, hole erst V_1 von Ws_1 und V_2 von Ws_2 und setze $V[S] = \max(V_1[S], V_2[S])$ für alle S.
- Um zu prüfen, ob eine Menge von WIDs Ws_1 eine Teilmenge einer anderen Ws_2 ist, hole wieder V_1 von Ws_1 und V_2 von Ws_2 und prüfe, ob V_1 von V_2 dominiert wird, d.h. ob V_2 in allen Komponenten größer oder gleich V_1 ist.

Es werden also zwei Versions-Vektoren verwendet: einer für die `reads` und einer für die `writes`. Der Session-Manager findet dann einen geeigneten Server, indem es prüft, ob einer oder beide Versions-Vektoren (abhängig von der Operation) des Servers dominieren.

Als Ergebnis einer `read`-Operation liefert ein Server immer die Menge der RelevantWrites mit zurück. Da diese Menge u.U. schwer zu bestimmen sein kann, ist es zulässig, wenn der Server in bestimmten Situationen den aktuellen Versions-Vektor zurückliefert als Schätzung der RelevantWrites. Diese Vorgehensweise ist

verletzt nicht die Garantien monotonic reads und writes follow reads und kann lediglich zu einer längeren Suche nach einem geeigneten Server auf Client-Seite führen.

Die `read`- und `write`-Operationen können wie in Algorithmus 4 implementiert werden.

Algorithm 4 Implementierung der `read`- und `write`-Operationen

`read`:

1. **if** MR **then**
   (a) Prüfe, ob der S-Vektor den `read`-Vektor dominiert.
2. **if** RYW **then**
   (a) Prüfe, ob der S-Vektor den `write`-Vektor dominiert.
3. $\langle$result, RelevantWrites $-$ Vektor$\rangle$:=read R from S.
4. `read`-Vektor:= max (`read` $-$ Vektor, RelevantWrites $-$ Vektor).
5. **return**(result).

`write`:

1. **if** WFR **then**
   (a) Prüfe, ob der S-Vektor den `read`-Vektor dominiert.
2. **if** MW **then**
   (a) Prüfe, ob der S-Vektor den `write`-Vektor dominiert.
3. WID:=`write` W in S.
4. `write`-Vektor[S] :=WID.clock.

Die Prüfungen können natürlich entfallen, wenn immer auf dem selben Server gearbeitet wird.

8 Prüfungsfragen

- **Welche verteilten Algorithmen kennen sie?**
  Echo, Schnappschuss.
- **Erläutere den Echo-Algorithmus!**
  Ziel des Algorithmus ist das Versenden einer Nachricht an alle Prozesse

in einem beliebigen zusammenhängenden Graphen. Dabei schickt ein Initiator die Nachricht an alle weiter, die er kennt. Jeder Empfänger gibt die Nachricht, sofer er sie das erste mal sieht, an alle weiter. Prozesse, die keine Nachfolger besitzen schicken ein Echo zurück. Wenn ein Knoten von allen bis auf einen Knoten ein Echo bekommen hat, schickt er ein Echo an seinen Vorgänger. Hat der Initiator ein Echo von all seinen Nachfolgern bekommen, so terminiert der Algorithmus.
Auf diese Art und Weise spannen die Echo-Nachrichten einen Baum auf.

- **Warum benutzt man anstelle des Echo-Algorithmus nicht einfach einen Broadcast?**
  Ein Broadcast wäre nicht kontrolliert und würde mehr Nachrichten al notwendig benötigen. Ausserdem setzt das zunächst einmal ein Broadcast-Netz voraus! Im Nachhinein kann ausserdem nicht garantiert werden, dass wirklich alle Prozesse die Nachricht erhalten haben.

- **Was bedeutet kausale Abhängigkeit?**
  Zwei Ereignisse e_1 und e_2 sind kausal abhängig genau dann, wenn e_1 Auswirkungen auf e_2 haben kann ($e_1 < e_2$). Ein Vertauschung der Reihenfolge würde also das Ergebnis ändern.

- **Erkläre das Problem der globalen Zeit!**
  In einem verteilten System kann es keine eindeutige globale Realzeit geben, da alle Quartz-Uhren in Computern nur mit einer relativen Genauigkeit arbeiten (→Driftrate).
  Eine Möglichkeit, trotzdem ein verteiltes System mit einer „relativ" exakten Realizeit zu realisieren, bietet der Einsatz von Zeit-Servern oder Time Daemons, die in Regelmässigen Zeitabständen die Uhren der Knoten Resynchronisieren. Bei dieses Ansätzen gehen die Uhren jedoch niemals exakt gleich, es wird lediglich ein „Genauigkeitsintervall" garantiert.
  Ein anderer Ansatz ist die Einführung einer logischen Zeit. Da bei einem verteilten System vor allem die „relative" Zeit eine Rolle spielt – also z.B. was vor etwas oder nach etwas passiert ist – muss die globale Zeit nicht gleich der exakten Zeit sein. In diesem Fall bekommt ein Zeitpunkt einen logischen Zeitstempel, der im System eindeutig ist und seine Posis in der Zeitachse eindeutig definiert.

- **Wie funktioniert die Lamport'sche Uhr?**
  Lamport's Uhr liegt die Relation $a \rightarrow b$ (a passiert vor b) zugrunde. Diese Relation ist anti-symmetrisch, nicht-reflexiv und transitiv. Die logische Uhren-Funktion in Prozess P_i $C_i(a)$, die den Zeitpunkt eines Ereignisses a beschreibt gelten drei Uhrenbedingungen:
  1. Bei internen Ereignissen $a, b \in P_1$ gilt $a \rightarrow b \Rightarrow C_1(a) < C_1(B)$.
  2. Bei externen Ereignissen $a =$ (SEND m in P_1) und $b =$ (RECEIVE m in P_2) gilt $C_1(a) < C_2(b)$
  3. $C()$ ist monoton steigend
  Jedes Ereignis in einem Prozess erhöht nun die logische Uhr in einem Prozess und alle Nachrichten tragen einen Zeitstempel t_i, so dass die logische

Zeit L_i von Prozess i beim internen Verschicken einer Nachricht um 1 erhöht wird. Bei empfangenen externen Nachrichten wird als neue logische Zeit L_i das Maximum aus L_i und t_j genommen.
Nachteil: auch voneinander unabhängige Ereignisse die parallel stattfinden tragen Zeitstempel als würden sie in einer bestimmten Reihenfolge ausgeführt werden. Das kann z.B. beim verteilten Testen zu Problemen führen – dem verteilten System wird also eine lineare Zeit aufgeprägt, die es dort eigentlich gar nicht gibt.

- **Wie funktioniert Lamport's Ansatz zu Lösung verteilter Semaphore?**
  Lamport's Ansatz verlangt ein Broadcast-Netz als Grundlage. Jeder Prozess sendet sein $P()$ oder $V()$ Operation zusammen mit seiner Prozessnr., seiner lokalen Zeit und dem KA als Broadcast an alle anderen Prozesse, die daraufhein mit einem ACK antworten. Diese Nachrichten werden von jedem Prozess in einer Nachrichten Warteschlange nach aufsteigenden Zeitstempeln sortiert gespeichert. Möchte nun ein Prozess in den KA eintreten, so prüft er die Semaphor-Invariante mit Initialisierungswert S_0 $\#\{P-\text{Operationen}\} \leq \#\{V-\text{Operationen}\} + S_0$. Ist die Nachricht stabil, liegt also von allen Prozessen eine Nachricht mit größerem Zeitstempel vor, so kann der Prozess den KA betreten. Beim Verlassen des KA sendet der Prozess schliesslich ein $V()$ als Broadcast an alle.

- **Wieviele Nachrichten werden für das Abarbeiten eines KA insgesamt benötigt?**
  $3\,(n-1)$-viele: $(n-1)$ Nachrichten als Broadcast an alle, $(n-1)$ ACK-Nachrichten und $(n-1)$ $(n-1)$ $V()$-Nachrichten beim Austritt.

- **Stehen beim dezentralen Semaphor-Algorithmus auch $V()$-Operationen in der Warteschlange?**
  Ja, genau wie auch die Ack's, mit deren Hilfe eine Nachricht als stabil markiert werden kann. Würden die $V()$ nicht gespeichert werden, so könnte kein Prozess entscheiden, ob er in den KA eintreten darf.

- **Kann der Algorithmus noch verbessert werden?**
  Ricart und Agrawala nutzen Lamport's totale Ordnung aus und setzen ein zuverlässiges Versenden von Nachrichten voraus. Genau wie bei Lamport wird das $P()$ also Broadcast verschickt. Der Empfänger schickt ein OK, wenn er nicht im KA ist und keine Absicht hegt. Ist er im KA, so hebt er die Anfrage auf und antwortet mit einem OK, wenn er den KA verlässt. Im Falle der Absicht schickt er nur dann ein OK, wenn die Anfrage einen kleineren Zeitstempel hat als die eigene. Auf diese Weise werden $n-1$ Nachrichten gespart.

- **Was sind Vor- und Nachteile?**
  Ein Vorteil ist, dass es ein dezentraler Algorithmus ist, dafür blockiert der Ausfall eines Prozesses gleich das ganze System und jeder Prozess muss aktiv sein um die Nachrichten zu verarbeiten,

- **Gibt es weitere Ansätze?**
  Beim Token-Ansatz bilden alle Prozesse einen logischen Ring, in dem ein Token rotiert, das immer nur im Besitz eines Prozesses sein kann. In den KA eintreten kann nun nur der Prozess, der gerade das Token besitzt.

- **Was ist der Nachteil?**
  Nicht dezentral, da es einen Token-Monitor geben muss.

- **Welche Informationen werden bei Agrawala und Ricard abgespeichert und warum?**
  Agrawala und Ricard speichern in dem Token zusätzlich noch für jeden Prozess den Zeitpunkt der letzten Zuteilung und schicken das Token dann an den ersten folgenden Prozess im Ring, dessen letzte Anforderung später als die letzte Zuteilung erfolgt ist.

- **Wieviele Nachrichten werden für einen KA benötigt?**
  $(n-1)$ Nachrichten für das Eintrittsgesuch (Broadcast) und 1 Nachricht für das Token – macht n Nachrichten.

- **Gibt es noch einen Algorithmus, der besser bzgl. der Nachrichtenkomplexität ist?**
  Der Algorithmus von Maekawa besitzt nur eine Nachrichtenkomplexität von $6\,(\sqrt{n}-1)$. Er ordnet alle Prozesse in einem Gitter an. Damit ein Prozess in den KA eintreten kann benötigt er die Erlaubnis aller Prozesse in seiner Zeile und Spalte. Für die Erlaubnis werden also $2\,(\sqrt{n}-1)$-viele P()-Nachrichten mit $2\,(\sqrt{n}-1)$ ACK-Nachrichten und für das Verlassen nochmal $2\,(\sqrt{n}-1)$ V()-Nachrichten benötigt.

- **Warum ist die Terminierung in verteilten Systemen ein Problem?**
  Da jeder Knoten nur sein lokales Wissen besitzt und keinen globalen Zustand kennt, kann ein lokaler Knoten nie entscheiden, in welchem Zustand sich das ganze System befindet.

- **Was für Ansätze gibt es zur Lösung der verteilten Terminierung?**
  Es gibt zwei Ansätze, um die Terminierung eines verteilten Algorithmus festzustellen: der kommunikationsorientierte, bei dem nach Abschluss eine Nachricht geschickt wird, und der ergebnisorientierte, bei dem ein Prädikat erfüllt sein muss.
  Beim kommunikationsorientierten Ansatz werden zwei Arten von Nachrichten unterschieden: Basis- und Kontroll-Nachrichten. Ein aktiver Prozess kann nur Basis-Nachrichten versenden und ein passiver Prozess wird durch Empfangen einer Basis-Nachricht aktiv. Der Algorithmus ist genau dann terminiert, wenn keine Nachricht mehr auf der Leitung ist, also alle Prozesse im passiven Zustand sind. Genau das ist aber schwer festzustellen, da niemand den globalen Zustand kennt und keiner weiss, ob nicht ein Porzess noch eine Nachricht schicken möchte.
  Beim synchronen Kommunikationsmodell wird nun die Laufzeit der Nachrichten vernachlässigt, die Berechnung ist also terminiert, wenn alle Prozesse passiv sind und der Algorithmus muss nicht feststellen, ob noch

Nachrichten unerwegs sind – nur das Feststellen der Passivität bei allen Prozessen wird zum Problem, wenn z.B. „Kommunikation hinter dem Rücken" stattfindet. In diesem Fall müssen mehrere Kontrollwellen gestartet werden, die die Flags der Prozesse abfragen.
Beim Atommodell werden die Berechnungszeiten vernachlässigt, Folgenachrichten werden also sofort nach Erhalt einer Nachricht versendet und wieder ist die Berechnung terminiert, wenn keine Nachrichten unterwegs sind.
In diesen Modellen kann man jetzt die Terminierung durch Kontrollwellen oder Schnitte überprüfen. Hierbei werden konsistente (alle Ereignisse vor einem beliebigen Ereignis im Schnitt liegen ebenfalls im Schnitt) und inkonsistente Schnitte unterschieden.

- **Wie kann man das Terminierungsproblem lösen?**
  Beim Doppelzählverfahren, einem Verfahren mit zwei Wellen, werden sog. Wellen mit Basisnachrichten an alle Prozesse geschickt. Dies kann z.B. durch Broadcasts mit Rückantwort, Token, mit dem Echo-Algorithmus oder das direkte Ansprechen aller Prozesse geschehen. In der ersten Welle werden die bis dahin gesendeten und empfangenen Basisnachrichten gezählt und aufaddiert. Das gleiche wird in der zweiten Welle gemacht. Sind beide Summen identisch, so wurde zwischen den beiden Wellen keine Nachricht versendet – der Algorithmus ist also terminiert.
  Beim Vektoralgorithmus zählt jeder Prozess (positiv) in seinem Vektor V in Element V_j, wieviele Nachrichten von ihm an einen anderen Prozess j verschickt wurden und (negativ), wieviele Nachrichten er erhalten hat. Nun zirkuliert ein Kontrollvektor zwischen den Prozessen, auf den die jweiligen lokalen Vektoren aufaddiert werden. Ist der Kontroll-Vektor nach einem Rundlauf gleich dem Nullvektor, ist die Berechnung terminiert.
  Vorteile sind die Unabhängigkeit von der Netztopologie, nur eine Welle wird benötigt und die Basis-Nachrichten bleiben unberührt. Leider wird die Länge der Kontrollnachrichten immer größer, da der Kontrollvektor immer mitgesendet wird.

- **Wie genau funktioniert das Vektorverfahren?**
  Jeder Prozess besitzt eine eigene Uhr C_i, die er bei jedem Ereignis um 1 erhöht. Ein externer Beobachter könnte nun alle diese Zeiten aller n Prozesse zu einem Zeitpunkt t in einem Vektor V_t speichern. Da dies aufgrund von Nachrichtenlaufzeiten aber nicht möglich ist, wurde die Vektorzeit als optimale lokale Approximation einer globalen Zeit eingeführt: Jeder Prozess P_i besitzt einen lokalen Zeitvektor C_i, der aus n Komponenten besteht. Bei jedem Ereignis (z.B. einem Empfangsereignis) wird $C_i[i]$ um 1 erhöht und C_i als Zeitstempel verwendet, der an zu verschickende Nachrichten angehängt wird. Empfängt ein Prozess eine Nachricht, so überprüft er jede Komponente des empfangenen Zeitvektors und übernimmt die Komponentenwerte in seinen eigenen Vektor falls diese größer sind als die ensprechenden Werte im eigenen Zeitvektor. Dadurch ist es

egal, in welcher Reihenfolge voneinander unabhängige Ereignisse stattfinden. Jeder Prozess P_i hat als alleiniger die Macht, den Wert der i-ten Komponente in der globalen Zeit zu erhöhen und besitzt als einziger immer garantiert die exakten Wert.
Beweis: Angenommen, es gibt ein Ereignis rechts der Schnittlinie, dann liegt die Ursache des frühesten Ereignisses im Schnitt und ein Prozess hat einen Sendevorgang, der nicht ausgeglichen ist. Damit kann der Vektor nicht ein Nullvektor sein.

- **Was für Probleme können bei der Vektorzeit auftreten?**
  Es gibt keine globale Zett – Nachrichten aus der Zukunft können die Folge sein. Deshalb werden das Alibi-Prinzip und synchronisierte Uhren eingesetzt.

- **Wie funktioniert die Broadcast-Election?**
  Bei der Broadcast-Election speichert jeder Prozess den Sieger in einer Variable M. Der Initiator i setzt also $M := i$ und schickt einen M als Broadcast. Jeder Empfänger prüft, ob sein M kleiner ist als das empfangene. Ist das der Fall, so setzt er sein M auf das empfangene, wenn nicht, dann schickt er sein M als Broadcast.

- **Gibt es Probleme?**
  Wann ist der Algorithmus terminiert? -Wahrscheinlich, wenn keine Nachricht mehr gesendet wird, nur wann kann man das mit Sicherheit sagen?

- **Gibt es einen besseren Algorithmus?**
  Beim Algorithmus von Chang und Roberts wird das message extinction Prinzip angewandt. Hier wird eine Nachricht im Ring geschickt und währenddessen verändert. Das führt zu einer optimalen mittleren Nachrichtenkomplexität, aber einer schlechten worst-case-Nachrichtenanzahl.
  Es geht also eine Nachricht im Kreis und jeder merkt sich, ob er die Nachricht schonmal bekommen hat. Empfängt ein Prozess eine Nachricht, so schaut er, ob seine Nummer größer als die in der Nachricht ist. Ist das der Fall so tauscht er die Nummer gegen seine aus und schickt die Nachricht weiter. Findet er seine Eigene Nummer in der Nachricht, so schickt er eine Gewinner-Nachricht weiter, in der er seine Nummer verbreitet.

- **Wie ist da die Nachrichtenkomplexität?**
  Im Idealfall werden nur n Nachrichten benötigt, bis der Gewinner feststeht. Dabei wird die Gewinnerrunde weggelassen. Im worst-case bei k Initiatoren und n Prozessen werden $\sum_{i=0}^{k-1} n-i$ viele Nachrichten benötigt. Im Mittel erwarten wir aber nur $O(n \cdot \log n)$ viele Nachrichten.

- **Wie funktioniert gemeinsam genutzter verteilter Speicher?**
  Problem ist, dass jeder Rechner einen eigenen Speicher hat, aber kein gemeinsamer physikalischer Speicher vorhanden. Also wird versucht, soetwas virtuell zur Verfügung zu stellen.

- **Wo liegen die Probleme?**
  Ein Methode ist das Zentrale Verwalten eines gemeinsamen Speichers mit allen Nachteilen eines central point of failures und den Netzwerk und Verwaltungskosten. Eine andere Möglichkeit ist das Replizieren von Daten. Falls replizierte Daten aber geändert werden, kann es zu Inkonsistenzen kommen.

- **Wie kann das Inkonsistenzproblem gelöst werden?**
  Seiten können repliziert werden, falls dann ein Prozess schreiben will werden alle Kopien für ungültig erklärt; Existenz nur einer Master-Kopie nur zum Schreiben. Eine andere Möglichkeit ist die optimistische Replikation bei der Änderungen im Nachhinein synchronisiert werden.

- **Wie findet man die Kopien?**
  Page-Manager oder Broadcast an alle Prozesse

- **Welche Konsistenzmodelle gibt es?**
  Absolute, sequentielle, kausale, PRAM, schwache, Freigabe- und Eintritts-Konsistenz.

- **Wie erreicht man sequentielle Konsistenz?**
  z.B. mit Lamport's Algorithmus (Lamport Uhren)

- **Wie können RPC's in verteilten Systemen beschleunigt werden?**
  - kein teures Kopieren von Daten
  - sogar weniger Aufwand als UDP (Prüfsummen, weniger Felder)
  - Effiziente Timer-Steuerung
  - sparsames Umgehen mit Prozesswechseln
  - Übertragungszeit im Netz vergrössern.

- **Was ist mit einer effizienten Timer-Steuerung bei RPC's gemeint?**
  Timeouts?

- **Wie wird der Erwartungswert, die MTBF ermittelt?**
  Wenn p die Wahrscheinlichkeit ist, dass eine Komponente ausfällt, dann ist die MTBF, also der Erwartungswert, wie lange es dauert, bis die Komponente ausfällt, gleich $1/p$.

Literatur

[Mattern1989] F. Mattern: „Verteilte Basisalgorithmen", Springer-Verlag, 1989

[Tanenbaum1995] A. S. Tanenbaum: „Verteilte Betriebssysteme", Prentice-Hall-Verlag, 1995

[Raynal1988] M. Raynal: „Distributed Algorithms and Protocols", John Whiley & Sons 1989.

[Mattern1999] Skript „Verteilte Algorithmen" von F. Mattern, http://www.informatik.tu-darmstadt.de/VS/Lehre/SS99/VertAlg/20.193.1.html, http://www.informatik.tu-darmstadt.de/VS/Lehre/WS97-98/VertAlg/20.146.1.html