Folienkopien zur Vorlesung

Verteilte Algorithmen

F. Mattern

FB Informatik, Technische Universität Darmstadt

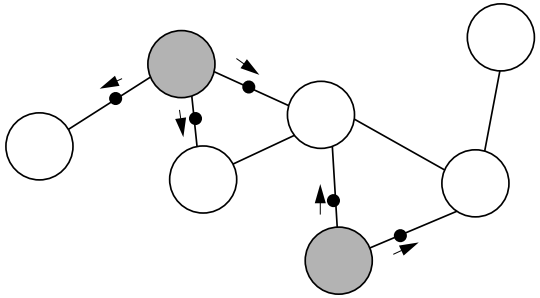
WS 97/98 (4-stündig mit Übungen)

Achtung: Prüfungsrelevant ist der Inhalt der Vorlesung, nicht alleine der Text dieser Foliensammlung!

Verteilte Algorithmen

F. Mattern

Fachbereich Informatik
Technische Universität Darmstadt



Verteilte Algorithmen

= Algorithmen für / in verteilten Systemen

Grundkonzept der Informatik (praktisch / theoretisch)	Interessantes Gebiet der praktischen Informatik sowie wichtiges Anwendungsfeld

- Was ist "verteilt" an einem verteilten Algorithmus?

  Vorläufige Antwort: *Zustand* und *Kontrolle* auf verschiedene "Orte".

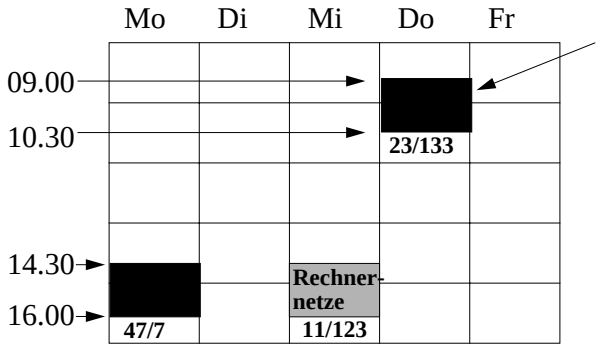
Wintersemester 1997/98

Einordnung der Vorlesung

4-stündige Vorlesung (inkl. Übungen) der praktischen Informatik (Prüfungsfach Informatik II)

Sinnvolle Vorkenntnisse:

- Verteilte Systeme (< 10% Überschneidung)
- Grundkenntnisse der Informatik und Mathematik (Vordiplom)



i.a. Vorlesung, aber Übung je nach Thema und Bedarf (mit gelegentlichen Aufgaben)

- Folienkopien befinden sich *nach* der jeweiligen Vorlesung im Lernzentrum Informatik (später auch im WWW im .ps-Format)

- Abweichungen von der Vorlesung vor 2 Jahren!

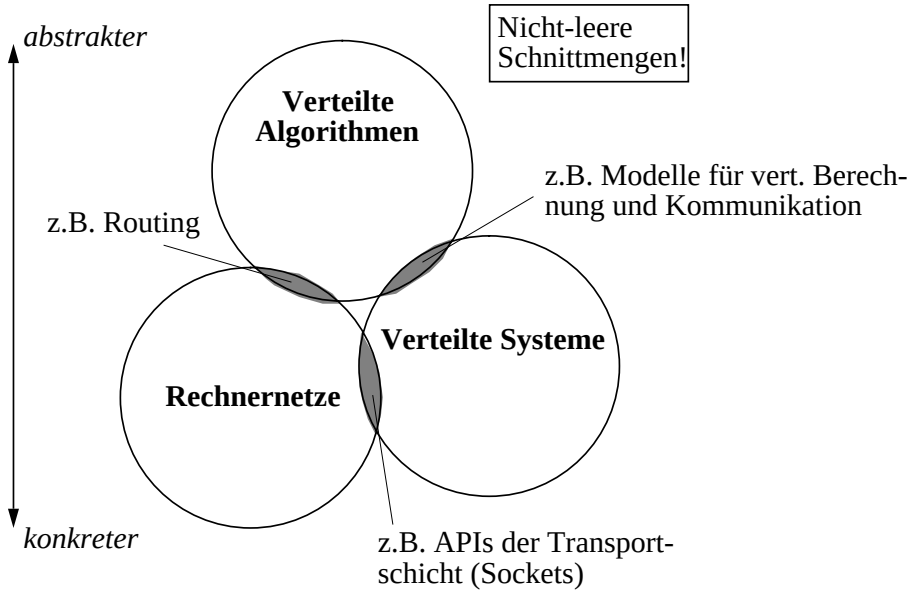
Einordnung der Vorlesung (2)

Weitere relevante Veranstaltungen:

Cocktail für Diplomprüfung

Mattern: *Verteilte Systeme*, 4st, WS 98/99
Mattern: *Rechnernetze*, 2st, WS 97/98
Theel: *Verteilte Betriebssysteme*, 2st, WS 97/98
Kammerer: *Betriebssysteme I*, 2+2st, WS 97/98
Kammerer: *Betriebssysteme II*, 2+2st, SS 98
Mattern et al.: *Praktikum*
Mattern et al.: *Seminar*

Es gibt noch weitere am FB angebotene Veranstaltungen



Vorlesungsankündigung

```
Veranstaltungsform:  V3 + Ü1
Hochschullehrer:     Prof. Mattern
Ort / Zeit:          V: 47/7,   Mo 14.25-16.05
                        23/133, Do  8.55-10.35 (14-tägig)
                     Ü: 23/133, Do  8.55-10.35 (14-tägig)
Beginn:              V: 21.10.1997, Ü: 30.10.1997
Voraussetzungen:     Vordiplom

Anmeldung:           ./.
Vorbesprechung:      ./.
Turnus:              alle 1 - 2 Jahre
```

Inhalt:

Verteilte Algorithmen sind Verfahren, die dadurch charakterisiert sind, daß mehrere autonome Prozesse gleichzeitig Teile eines gemeinsamen Problems in kooperativer Weise bearbeiten und der dabei erforderliche Informationsaustausch ausschließlich über Nachrichten erfolgt. Derartige Algorithmen kommen im Rahmen verteilter Systeme zum Einsatz, bei denen kein gemeinsamer Speicher existiert und die Übertragungszeit von Nachrichten i.a. nicht vernachlässigt werden kann. Da dabei kein Prozeß eine aktuelle konsistente Sicht des globalen Zustands besitzt, führt dies zu interessanten Problemen.

Im einzelnen werden voraussichtlich folgende Themen behandelt:

Modelle verteilter Berechnungen;
Raum-Zeitdiagramme;
virtuelle Zeit; logische Uhren und Kausalität;
Wellenalgorithmen;
verteilte und parallele Graphtraversierung;
Berechnung konsistenter Schnappschüsse;
wechselseitiger Ausschluß;
Election und Symmetriebrechung;
verteilte Terminierung;
Garbage-Collection in verteilten Systemen;
Beobachten verteilter Systeme; Berechnung globaler Prädikate;
Testen verteilter Systeme;
parallele und verteilte Simulation;
distributed / virtual shared memory.

Literatur:

F. Mattern: Verteilte Basisalgorithmen. Springer-Verlag, 1989.
G. Tel: Topics in Distributed Algorithms. Cambridge University Press, 1991.
G. Tel: Introduction to Distributed Algorithms. Cambridge University Press, 1994.
V. Barbosa: An Introduction to Distributed Algorithms, MIT Press, 1996.
N. Lynch: Distributed Algorithms, Morgan Kaufmann Pub., 1996.
Artikel aus Fachzeitschriften (wird in der Vorlesung bekanntgegeben).

Vertiefung: Vorlesungen "Rechnernetze", "Verteilte Systeme", "Verteilte Datenbanksysteme", "Betriebssysteme I und II"; Praktikum "Verteilte Systeme".

Einordnung in Studienplan und Prüfungsordnung: Informatik II.

Literatur...

Zu einigen Aspekten verteilter Algorithmen, die in dieser Vorlesung angesprochen werden, findet man mehr in **F. Mattern: Verteilte Basisalgorithmen,** Springer-Verlag, IFB 226, 1989. Dort werden auch weitere Literaturhinweise auf spezielle Forschungsliteratur gegeben.

Weitere Aspekte verteilter Algorithmen werden ferner in **G. Tel: Topics in Distributed Algorithms**, Cambridge University Press, 1991 gut, wenn auch etwas formaler und abstrakter, behandelt. 1994 erschien vom gleichen Autor das Buch **Introduction to Distributed Algorithms**, Cambridge University Press, welches eine empfehlenswerte und gute Einführung und Übersicht darstellt.

Aus den genannten drei Büchern wurden für die vorliegende Vorlesung einige Dinge entnommen.

Das Buch von **V. Barbosa: An Introduction to Distributed Algorithms**, MIT Press, 1996, ist *nicht* empfehlenswert.

Das Buch von **N. Lynch: Distributed Algorithms**, Morgan Kaufmann Pub., 1996, behandelt die Problematik umfassend vom theoretischen Standpunkt.

Von **M. Raynal** sind einige Bücher in französischer und teilweise auch in englischer Sprache zum Themengebiet "verteilte Algorithmen" erschienen (z.B. "Distributed Algorithms and Protocols, Wiley, 1988"; oder "Networks and Distributed Computation, MIT-Press, 1988"). Diese geben jedoch zum größeren Teil nicht den Stand der Vorlesung wieder.

Leider sind die meisten der in der Vorlesung behandelten Dinge nicht in Buchform, sondern nur in Form von Aufsätzen in verschiedenen Fachzeitschriften veröffentlicht. Die folgende Auswahl führt nur einige wenige auf, die in engerem Bezug zur Vorlesung stehen.

Eine knappe Übersicht zu einigen Themen gibt folgender Aufsatz: "**F. Mattern: Distributed Control Algorithms (Selected Topics)**". In: F. Ozguner (Hg.): Parallel Computing on Distributed Memory Multiprocessors, Springer-Verlag, 1993.

Zum Deadlockproblem stellt "**E. Knapp: Deadlock Detection in Distributed Databases**, Computing Surveys 19:4, 303-328, 1987" eine gute Übersicht dar.

Für das Problem des wechselseitigen Ausschlusses in verteilten Systemen gibt **"B.A. Sanders: The Information Structure of Distributed Mutual Exclusion Algorithms, ACM Transactions on Computer Systems 5:3, pp. 284-299, 1987"** eine interessante Klassifikation.

Die folgenden Aufsätze des Autors behandeln einige Themen, die in der oben als erstes angeführten Monographie nicht oder nur kurz angesprochen werden, genauer:

F. Mattern: Virtual Time and Global States of Distributed Systems. In: Cosnard M. et al. (Hg.): Proc. Workshop on Parallel and Distributed Algorithms, North-Holland / Elsevier, pp. 215-226, 1989.

... Literatur

F. Mattern: Über die relativistische Struktur logischer Zeit in verteilten Systemen. In: J. Buchmann, H. Ganzinger, W.J. Paul (Eds.): Informatik -Festschrift zum 60. Geburtstag von Günter Hotz, Teubner, pp. 309-331, 1992.

G. Tel, F. Mattern: The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. ACM Transactions on Programming Languages and Systems 15:1, 1-35, 1993.

R. Schwarz, F. Mattern: Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Distributed Computing 7:3, 149-174, 1994.

B. Charron-Bost, F. Mattern, G. Tel: Synchronous, Asynchronous, and Causally Ordered Communication. Distributed Computing, 9:4, 173-191, 1996.

F. Mattern et al.: Global Virtual Time Approximation with Distributed Termination Detection Algorithms. Technischer Bericht RUU-CS-91-32, Department of Computer Science, Universität Utrecht, 1991.

F. Mattern: Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. Journal of Parallel and Distributed Computing 18:4, pp. 423-434

Die Vorlesung basiert auf einer Vielzahl weiterer Zeitschriftenartikel. Es seien hier nur noch zwei einschlägige aufgeführt:

Der Sammelband "**Global States and Time in Distributed Systems**", herausgegeben von **Z. Yang** und **T.A. Marsland** (IEEE Computer Society Press, 1994), enthält eine größere Sammlung von Nachdrucken einschlägiger Zeitschriftenartikel der genannten Thematik.

Der Sammelband "**Distributed Systems (second edition)**", herausgegeben von **S. Mullender** (Addison-Wesley, 1993), enthält u.a. den Aufsatz "**Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms**" (pp. 55-96) von **Ö. Babaoglu** und **K. Marzullo**.

Themenspektrum

- Wellenalgorithmen, Traversierungsverfahren
- Verteilte Terminierung
- Verteiltes Garbage-Collection
- Globaler Zustand, Schnappschuß
- Konsistente Beobachtungen und globale Prädikate
- Logische Zeit, Kausalität, Konsistenz
- Wechselseitiger Ausschluß
- Election
- Kommunikationssemantik (synchron, asynchron, kausal)
- Verteilte Simulation ("time warp")

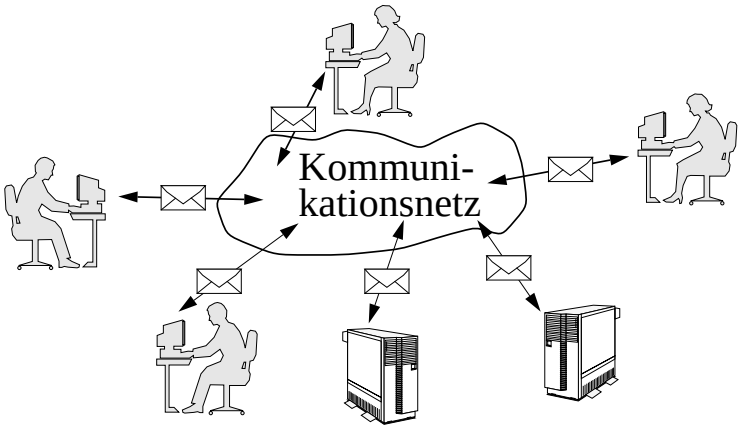
Algorithmen

- Prinzipielle Phänomene und Begriffe
  - Kausalität, Parallelität,...
- Anwendungsbezug
  - verteilte Betriebssysteme, Datenbanken, Anwendungen, Tools...

- Techniken, Einsichten, Zusammenhänge,...
- Basisverfahren, grundsätzliche Probleme,...
- Abstraktion, Modellbildung, Formalisierung,...
- Problemlösungstechniken, Analysetechniken,...
- Qualitätsbewertung, Komplexitätsabschätzung,...
- Verifikationstechniken

(verallgemeinerbare) Erkenntnisse

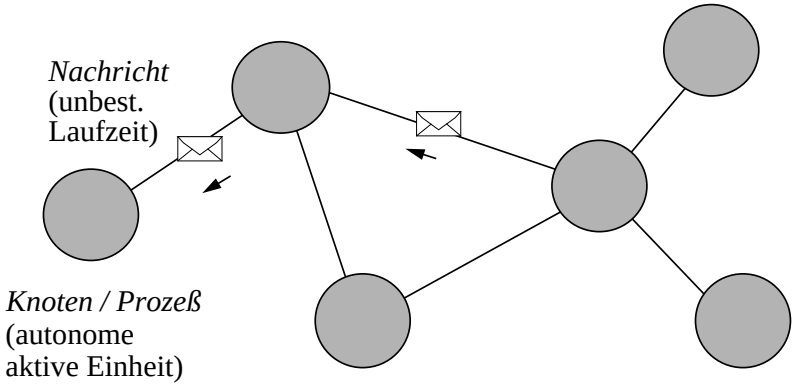
Verteiltes System



- Rechner, Personen, Prozesse, “Agenten” sind an *verschiedenen Orten*.

- Autonome Handlungsträger, die jedoch gelegentlich *kooperieren* (und dazu *kommunizieren*).

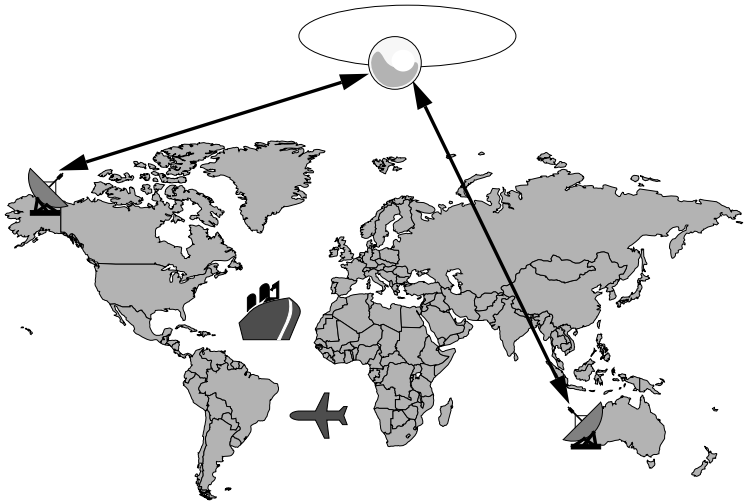
Verteiltes System II



Physisch verteiltes System:
Mehrrechnersystem ... Rechnernetze

Logisch verteiltes System: Prozesse
- Verteilung des Zustandes (keine globale Sicht)
- Keine gemeinsame Zeit (globale, genaue "Uhr")

Verteiltes System III

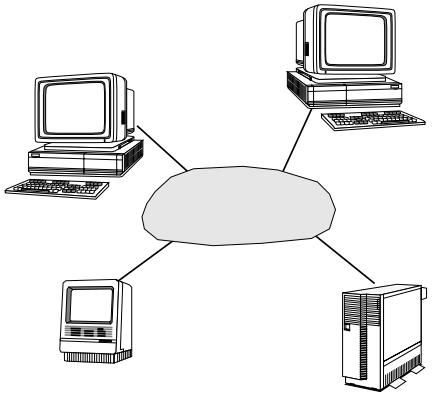


Auch die uns umgebende Welt ist ein verteiltes System:

- Viele gleichzeitige (“parallele”) Aktivitäten
- Exakte globale Zeit nicht erfahrbar / vorhanden
- Keine konsistente Sicht des Gesamtzustandes
- Kooperation durch explizite Kommunikation
- *Ursache* und *Wirkung* zeitlich (und räumlich) getrennt

- “Inkonsistente Zustände”: Kriegsende zwischen England und Frankreich war in den Kolonialgebieten erst später bekannt!
- Heute: “zeitkompakter Globus” - weitgehend synchronisierte Uhren.

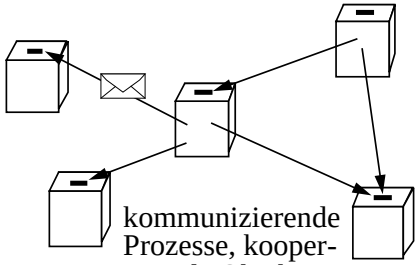
Sichten verteilter Systeme



Rechnernetz mit Rechenknoten:

- Multicomputer (Parallelrechner)
- LAN = Local Area Network
- WAN = Wide Area Network
- Routing, Adressierung....

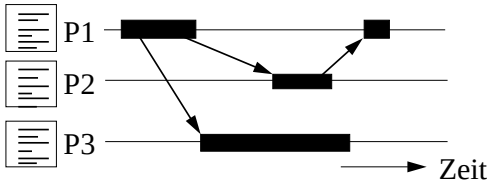
zunehmende Abstraktion



Objekte eines *Betriebssystems* bzw. einer *Programmiersprache*

kommunizierende Prozesse, kooperierende Objekte

==> “Programmierersicht” (Client, Server...)



Algorithmen- und Protokollebene

- Aktionen, Ereignisfolgen
- Konsistenz, Korrektheit

Parallele <--> verteilte Algorithmen

Sequentieller Algorithmus

↓

Paralleler Algorithmus

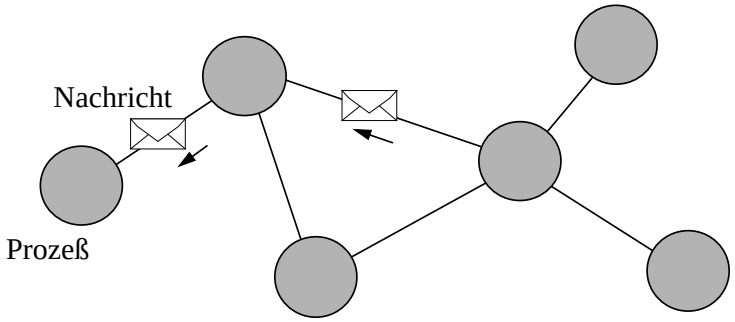
- mehrere, oft gleichartige Prozesse
- Synchronisation über gem. Speicher

↓

Verteilter Algorithmus

- mehrere Prozesse kommunizieren über Nachrichten
- gemeinsames Ziel --> Kooperation
- ideal: keine zentrale Kontrolle

oft asynchron



Aspekte verteilter Systeme

im Vergleich zu *sequentiellen* Systemen:

- Größe und Komplexität
- Heterogenität → jede(r) ist anders
- Nebenläufigkeit → vieles gleichzeitig
- Nichtdeterminismus → morgen anders als heute...
- Zustandsverteilung → niemand weiß alles

- Programmierung *komplexer*
- Test und Verifikation *aufwendiger*
- Verständnis der Phänomene *schwieriger*

==> gute Werkzeuge (“Tools”) und Methoden

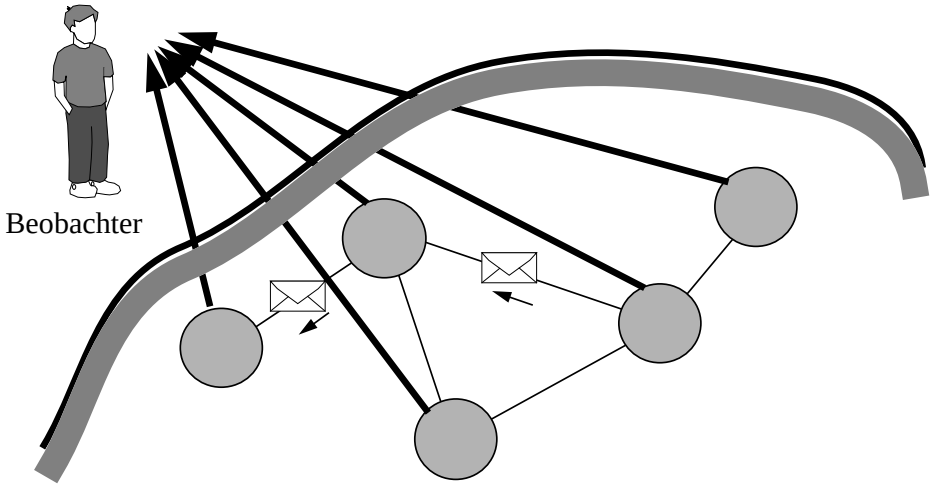
- z.B. Middleware als Software-Infrastruktur

==> adäquate Modelle, Algorithmen, Konzepte

- zur Beherrschung der neuen Phänomene

Ziel: Verständnis der grundlegenden Phänomene, Kenntnis der geeigneten Konzepte und Verfahren

Das Beobachtungsproblem



Beobachten geht nur über das Empfangen von "Kontrollnachrichten" (mit unbestimmter Laufzeit)

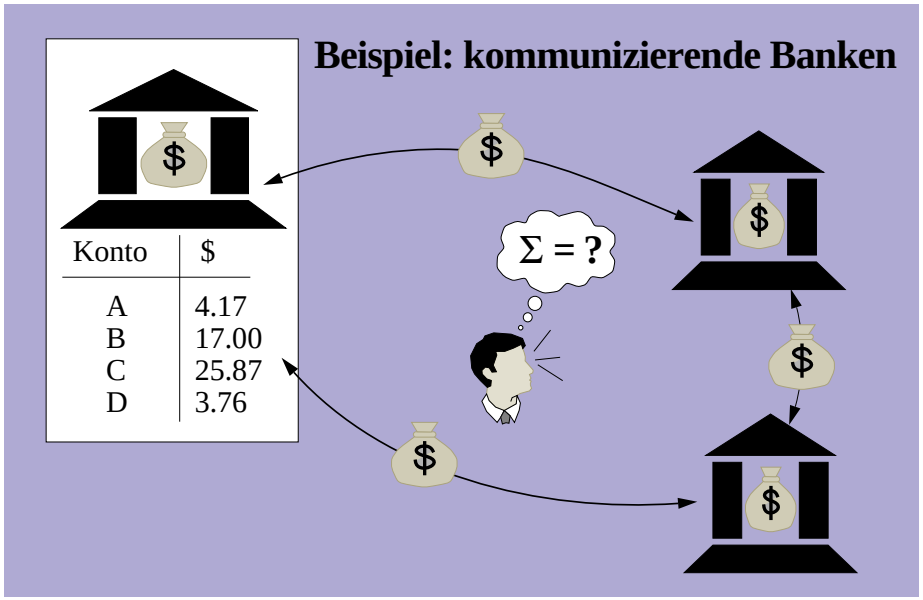
"Axiom": Mehrere Prozesse können "niemals" gleichzeitig beobachtet werden.

was heißt das?

"Korollar": Aussagen über den globalen Zustand sind schwierig.

Ein erstes Beispiel: Wieviel Geld ist in Umlauf?

- konstante Geldmenge, oder
- monotone Inflation (--> Untergrenze)

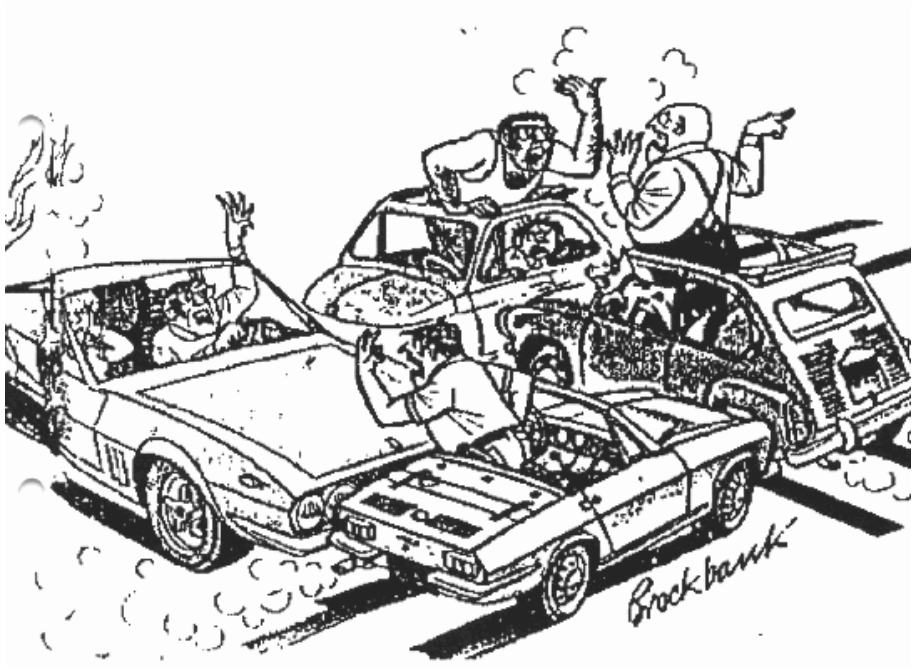


- Modellierung:
  - verteilte Geldkonten
  - ständige Transfers zwischen den Konten
- Erschwerte Bedingungen:
  - niemand hat eine globale Sicht
  - es gibt keine gemeinsame Zeit ("Stichtag")

--> Geht das dann überhaupt?
--> Ist das überhaupt ein wichtiges Problem?

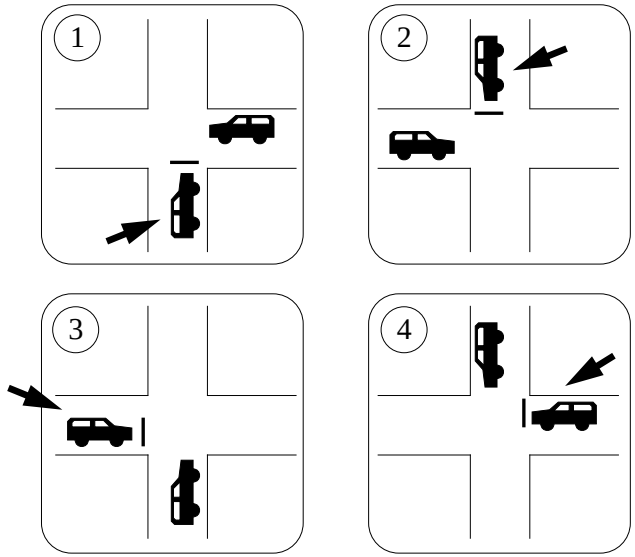
==> *Schnappschußproblem*

Das Deadlock-Problem



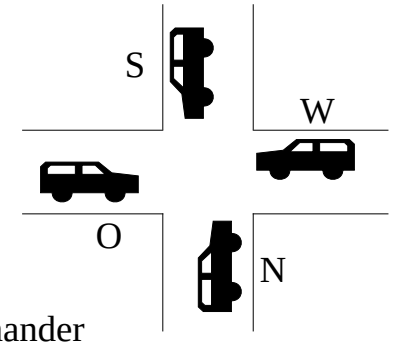
„Neapolitanischer Hakenkreuzstau“
(Also sprach Bellavista, Luciano De Crescenzo)

Phantom-Deadlocks



Vier Einzelbeobachtungen der Autos N, S, O, W

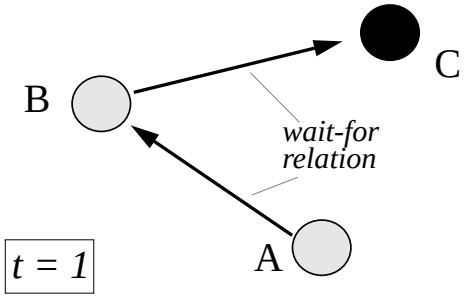
1) N wartet auf W
2) S wartet auf O
3) O wartet auf N
4) W wartet auf S



zu notwendigerweise verschiedenen Zeitpunkten liefert den *falschen* Eindruck, als würden zu einem *einzigen* Zeitpunkt alle zyklisch aufeinander warten (--> Verklemmung)

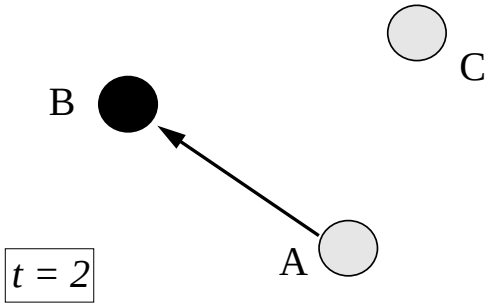
==> *verteiltes Deadlockproblem*

Phantom-Deadlocks

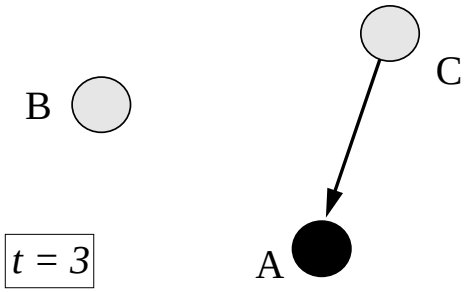


(C benutzt ein exklusives Betriebsmittel)

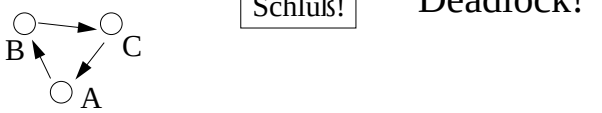
beobachte B:
==> B wartet auf C

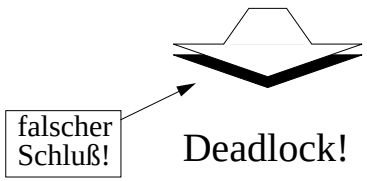


beobachte A:
==> A wartet auf B



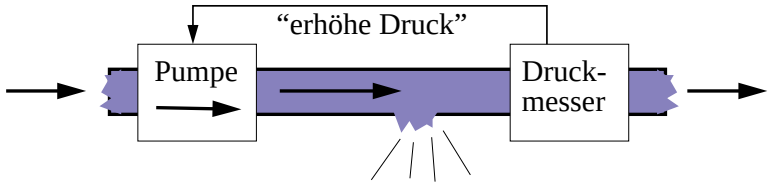
beobachte C:
==> C wartet auf A

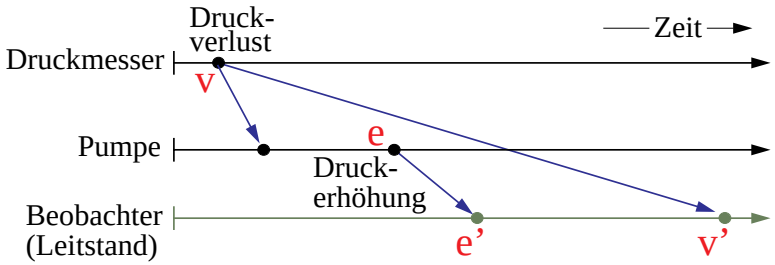




Kausal (in)konsistente Beobachtungen

- Gewünscht: Eine Ursache stets vor ihrer (u.U. indirekter) Wirkung beobachten

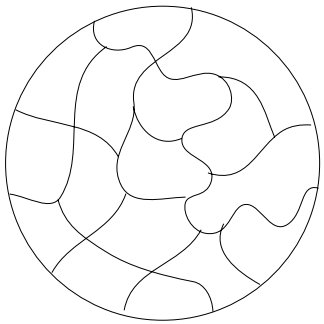


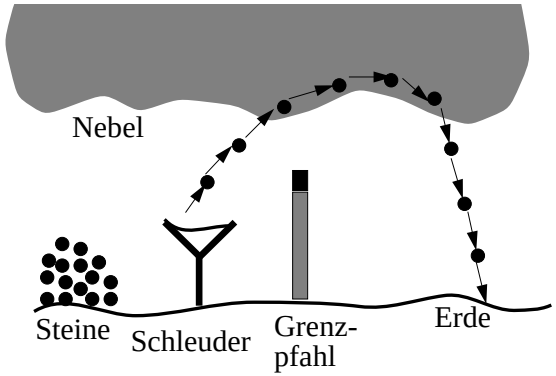


Falsche Schlußfolgerung des Beobachters:
Eine unbegründete Pumpenaktivität erhöhte den Druck bis zum Bersten der Pipeline;daraufhin trat das Öl aus dem Leck aus, was durch den Druckverlust angezeigt wird!

==> *"Causal order-Problem"*

Wie stellt man fest, daß der ewige Friede ausgebrochen ist?





Die Welt ist vollständig in einzelne Gebiete parzelliert

Regel: Nur wer von einem Stein getroffen wird, *darf* Steine wegschleudern.

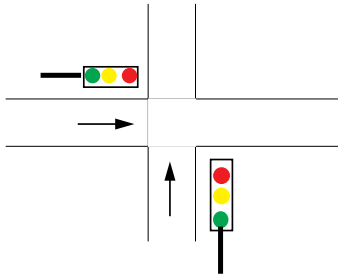
- Urknall: Ein einziger Meteorit traf die Erde...
- Steine fliegen beliebig hoch und brauchen beliebig lang...
- Niemand hat den Gesamtüberblick...

- Der Friede ist ewig ("stabil"; "monotone Eigenschaft")
- Wie stellt man sicher fest, *daß* er gilt (*wenn* er gilt)?

==> *Terminierungserkennungs-Problem*

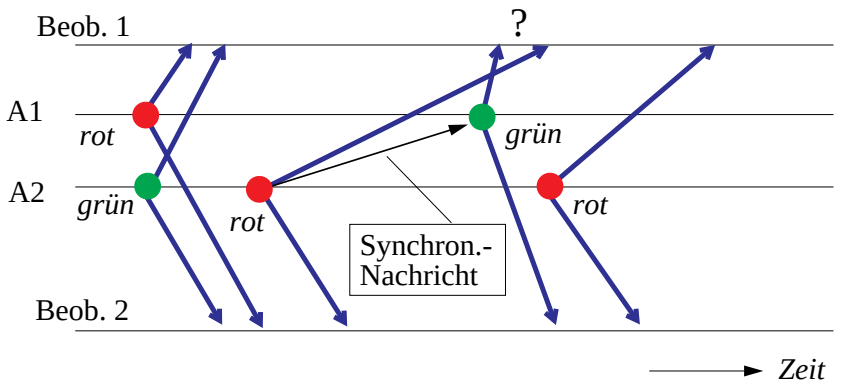
Beispiel: Verteilte Ampelsteuerung

(hier für 2 Ampeln)



Konsistenzproblem bei *mehreren Beobachtern*

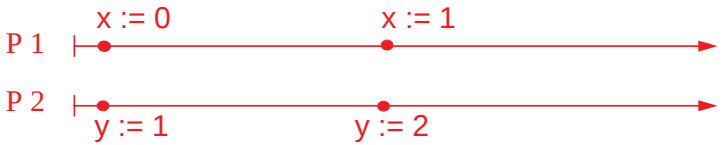
- Jede Ampel darf autonom auf rot schalten
- Eine Ampel darf nur dann auf grün schalten, wenn sie erfahren hat, daß die andere rot ist
- Umschalten der Ampel ist ein *Ereignis* ●
  (*atomar*: zeitlos, nicht unterbrechbare Aktion)

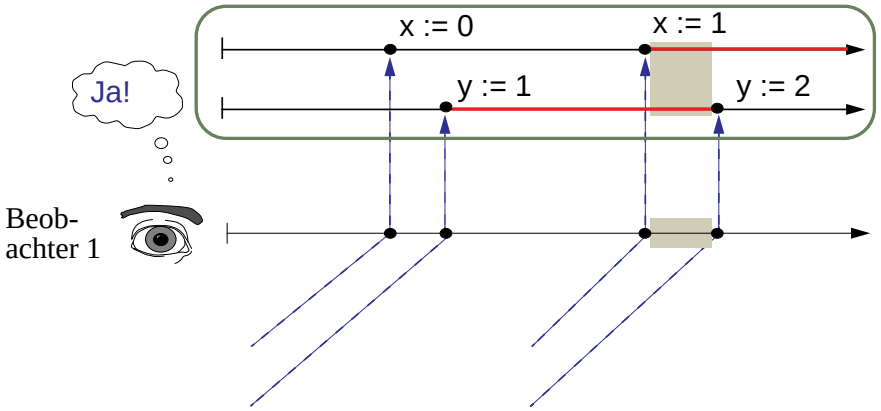


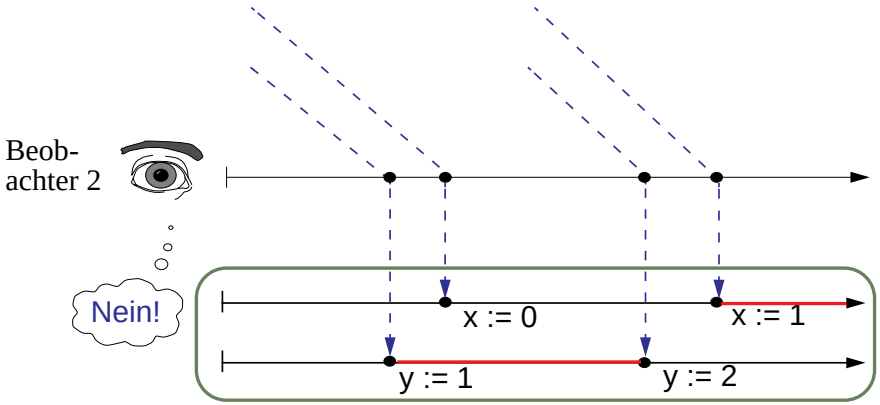
- Welcher Beobachter hat Recht? (Wieso?)
- Ähnliche Probleme (in komplexerem Umfeld) tauchen in der Praxis tatsächlich auf (z.B. Flugkontrollsystem)

Das Problem globaler Prädikate

Frage: Gilt in der vorliegenden Berechnung x = y ?





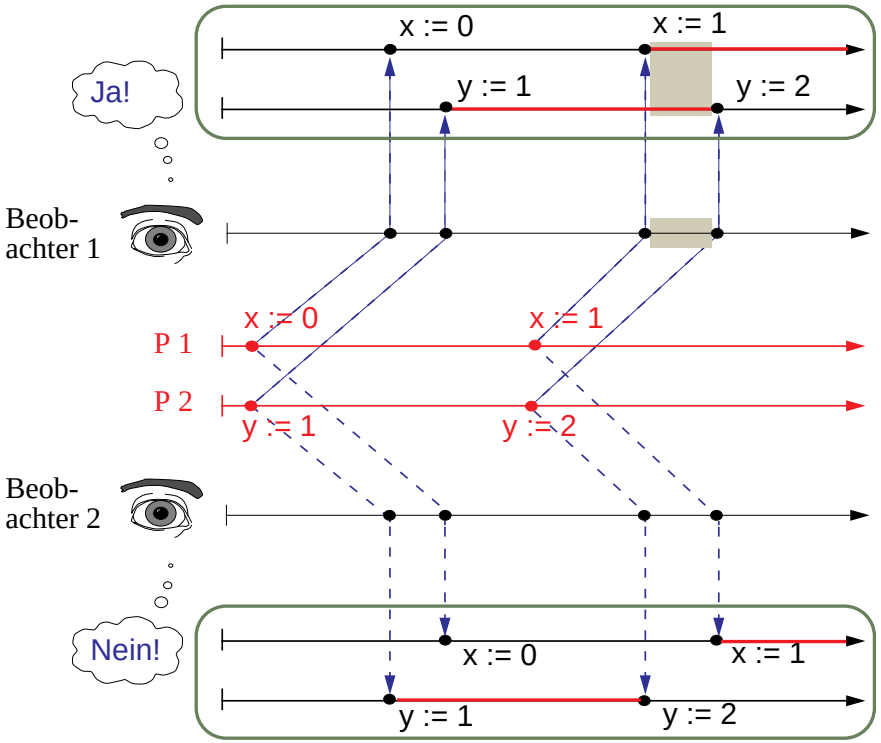


- Beide Beobachtungen sind gleich “richtig”
- Die Beobachter stimmen bzgl. x = y nicht überein!

Aber was denn nun: gilt x=y in dieser Berechnung oder nicht?

Das Problem globaler Prädikate

Frage: Gilt in der vorliegenden Berechnung x = y ?



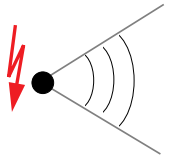
- Beide Beobachtungen sind gleich “richtig”
- Die Beobachter stimmen bzgl. x = y nicht überein!

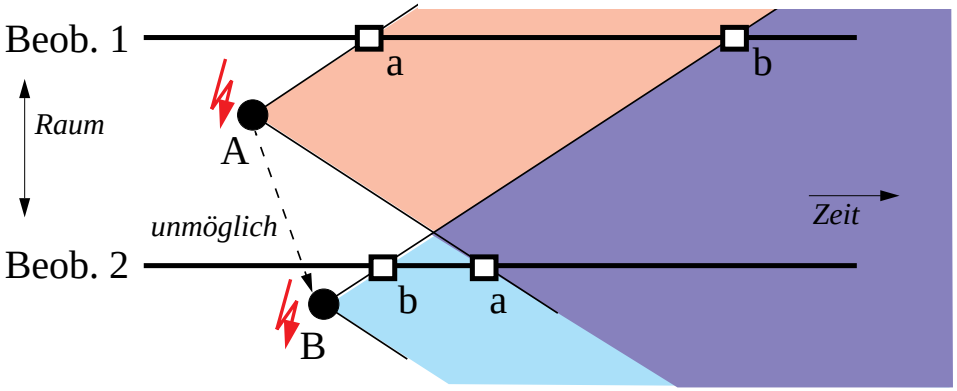
Aber was denn nun: gilt x=y in dieser Berechnung oder nicht?

Relativierung der Gleichzeitigkeit

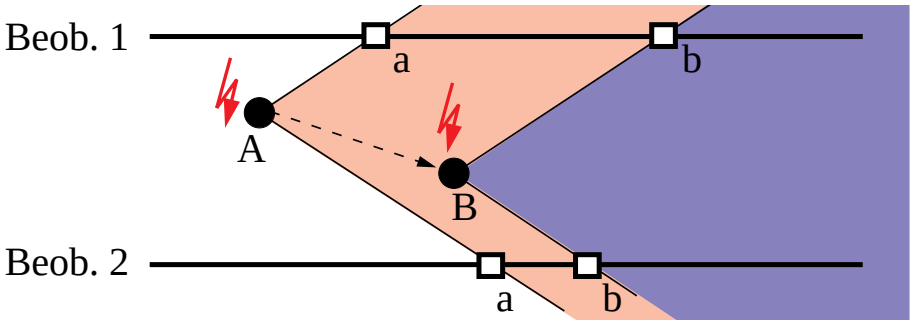
Zwei "kausal unabhängige" Ereigniss können in beliebiger Reihenfolge beobachtet werden!

Lichtkegel-Prinzip der relativistischen Physik





A und B geschehen "gleichzeitig"
(jedenfalls *kausal unabhängig*!)

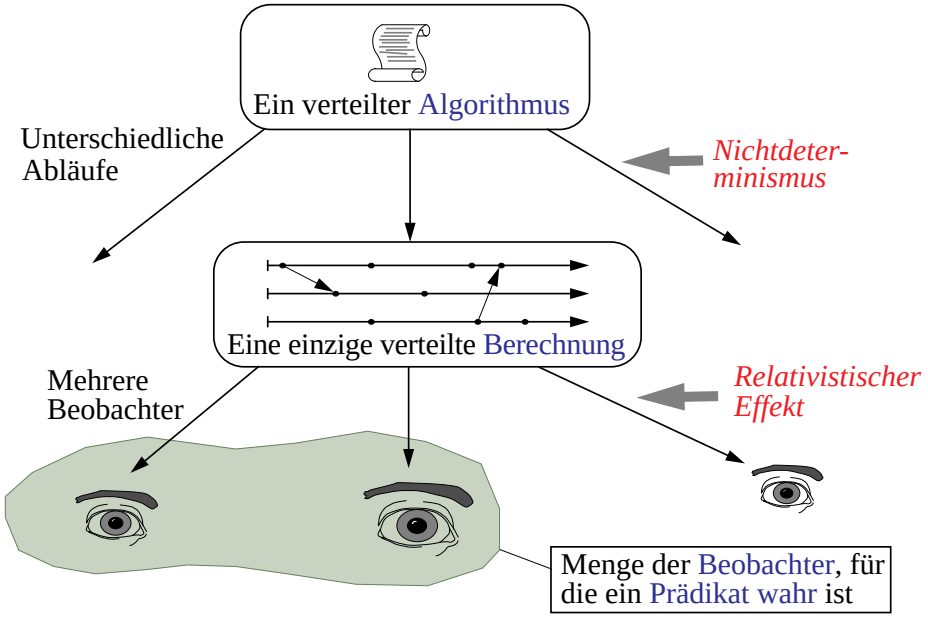


beobachterinvariant
==> *objektive Tatsache*

B liegt im Kegel von A -->
B hängt kausal ab von A -->
Alle Beobachter sehen B nach A

Die "Beobachtungsvielfalt"

- Verschiedene Beobachter sehen *verschiedene Wirklichkeiten*



Konsequenz:

Es ist naiv (d.h. falsch!), einen verteilten Debugger zu entwickeln, mit dem man solche (im sequentiellen Fall “richtigen”) Fragen beantworten kann!

Ursache:

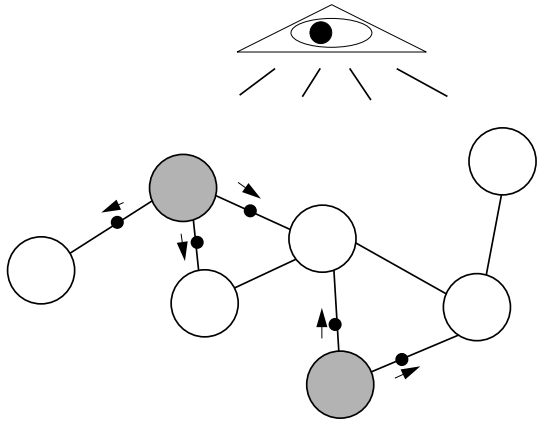
Bei sequentiellen Berechnungen fallen Berechnung und Beobachtung zusammen, im verteilten Fall nicht!

Halbordnung / lineare Ordnung

Globale Prädikate...

Eigenschaften globaler Zustände

... sind aber wichtig!



Bsp.:

- Wieviel Geld ist in Umlauf?
- Ist die parallele Approximation gut genug?
- Ist die verteilte Berechnung terminiert?
- Ist ein bestimmtes Objekt "Garbage"?
- Konsistenter Sicherungspunkt (vert. Datenbank)?
- Wie hoch ist die momentane Last?
- Liegt eine zyklische Wartebedingung vor?

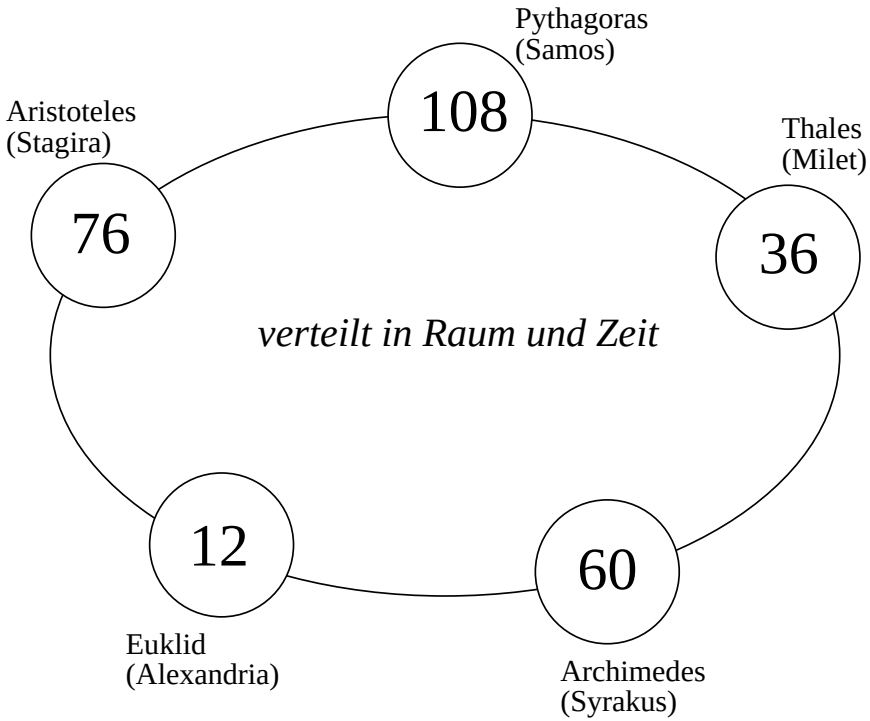
Einige globale Prädikate lassen sich "eindeutig" bestimmen

- Welche? --> fundamentale Aspekte analysieren
- Wie? --> *Algorithmen*

u.a. Thema der Vorlesung

Ein erster verteilter Algorithmus: Verteilte Berechnung des ggT

Fünf griechische Philosophen wollen zusammen den größten gemeinsamen Teiler ihres Alters berechnen...



- Was ist der ggT einer Menge $\{a_1,...,a_n\}$ nat. Zahlen?
- Wie funktioniert der *euklidische Algorithmus*?

"Elemente" Buch 7, Satz 1 und 2

Der größte gemeinsame Teiler (ggT)

- Nachfolgend werden nur natürliche Zahlen betrachtet.
- t heißt *Teiler* einer Zahl p, falls es q ≠ 0 gibt mit q t = p.
  - Bsp: 6 ist Teiler von 30 (da $5 \times 6 = 30$)
    3 ist Teiler von 30 (da $10 \times 3 = 30$)
    aber: 8 ist kein Teiler von 30
- 1 ist Teiler jeder Zahl (klar nach Definition).

- t heißt *gemeinsamer Teiler* von u, v
  wenn t Teiler von u ist und t Teiler von v ist.
  - Bsp: 6 ist gemeinsamer Teiler von 30 und 18.
    Aber 3, 2, 1 sind auch gemeinsamer Teiler.
    Jedoch gibt es keine anderen gemeinsamer Teiler
    ==> 6 ist der größte gemeinsame Teiler (ggT).
- Zu je zwei Zahlen ≠ 0 gibt es stets einen ggT, da die Menge der gemeinsamen Teiler mindestens die 1 enthält und endlich ist. (Was aber ist ggT(x,0)?)
- Kanonische Erweiterung auf n ≥ 1 Argumente
  - Bsp: ggT(108, 36, 60, 12, 76) = 4

- *Anwendung*: Z.B. Kürzen von Brüchen.

Satz von Euklid

Der ggT zweier positiver ganzer Zahlen x,y (mit x≥y>0) ist gleich dem ggT von y und dem Rest, der bei ganzzahliger Division von x durch y entsteht.

- Offenbar ist ggT(x,x) = x für alle x.
- Man setzt nun noch ggT(x,0) = x für alle x.
- Obiger Satz legt eine rekursive Realisierung nahe

```
ggt(x,y) --> ggt(y,mod(x,y)
```

  - Für x > y werden in der Rekursion beide Argumente jeweils kleiner
  - Rekursion geht irgendwann mit ggT(,,,.0) zuende
  - Oft: iterative Formulierung (statt rekursiver)

==> *Euklidischer Algorithmus*

Vorteil: Zur Bestimmung des ggT sind Primfaktorzerlegung ("Schulmethode") oder Probedivisionen nicht notwendig

Hinweis: Der Beweis des Algorithmus (und des Satzes) ist eine interessante Wiederholungsübung!

Das (abstrakte) Lösungsprinzip

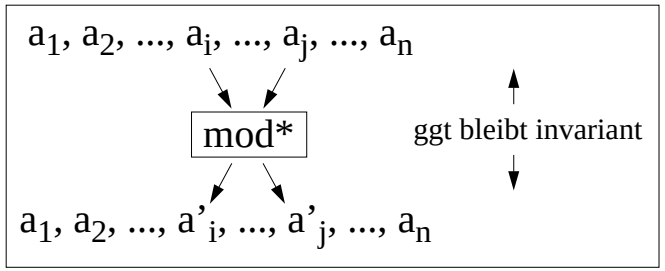
- *Invariante* (für $x \geq y > 0$):

  $ggT(x,y) = ggT(x,mod^*(x,y))$

  Wie *Restfunktion* mod, soll jedoch y liefern, falls x ganzzahliges Vielfaches von y ist. Also: $mod^*(a,b) = mod(a-1,b)+1$

- *Idee:* Ersetze x durch $mod^*(x,y)$. ← Ist i.a. kleiner als x

- Erweiterung auf n Zahlen:



Frage: Wieso wurde mod zu mod* modifiziert?

Greife zwei beliebige Elemente a_i, a_j ($i \neq j$) heraus und ersetze den größeren Wert durch $mod^*(a_i,a_j)$

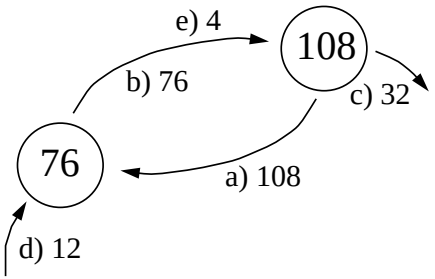
==> konvergiert gegen den ggT.

- Dies nun "verteilt" realisieren! ← In "Teamarbeit", damit es schneller geht...

Nachrichtenaustausch

Prinzip: Kooperation durch Kommunikation

zwischen den Nachbarn auf dem Ring



- Nachrichten sind lange unterwegs
- Parallele Aktivitäten

Idee:

- Initial Nachbarn spontan informieren.
- Danach nur auf eintreffende Nachrichten *reagieren.*
- Neue Erkenntnisse an alle Nachbarn übermitteln.

Verhaltensbeschreibung eines Prozesses P_i:

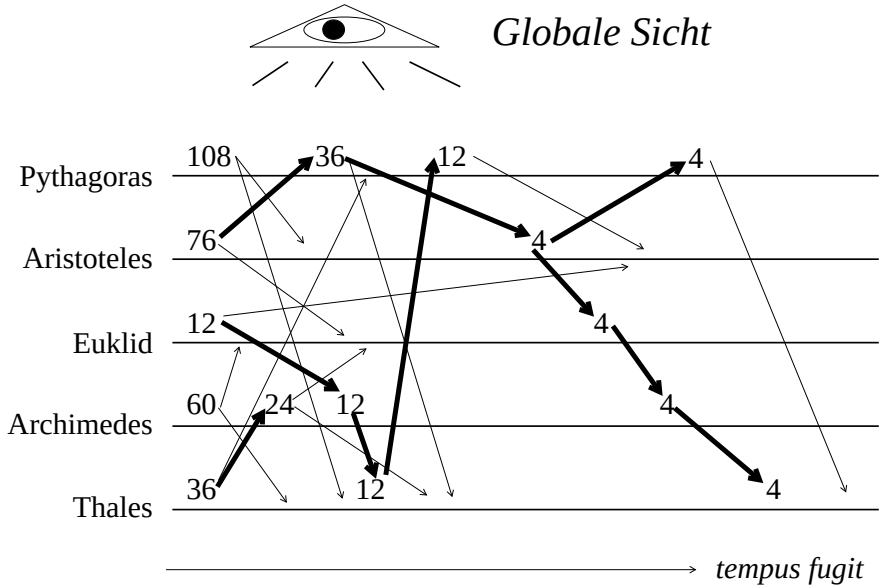
{Eine Nachricht <y> ist eingetroffen}
if $y < M_i$ **then**
  $M_i := mod(M_i-1,y)+1;$
  send $<M_i>$ **to** all neighbours;
fi

Jeder Prozeß P_i hat seine eigene Variable M_i.

Jeder Prozeß hat das gleiche prinzipielle Verhalten.

Atomare Aktion:
In der Zwischenzeit u.U. eintreffende Nachrichten werden im "Hausbriefkasten" zwischengepuffert...

Berechnungsablauf und Zeitdiagramm



Ablauf einer *möglichen* "verteilten Berechnung" mit einem *Zeitdiagramm*.

Nicht-deterministisch, abhängig von der Nachrichtenlaufzeit!

- *Terminiert* wenn (?):

  (a) jeder Prozeß den ggT kennt,

  (b) alle den gleichen Wert haben,

  (c) alle Prozesse passiv sind und keine Nachricht unterwegs ist.

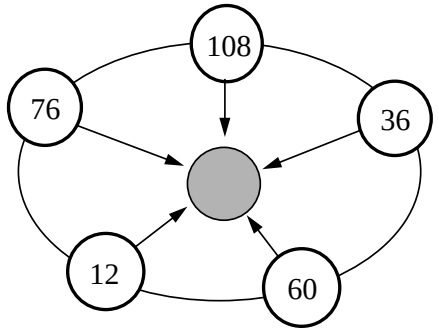
Das ist unrealistisch!

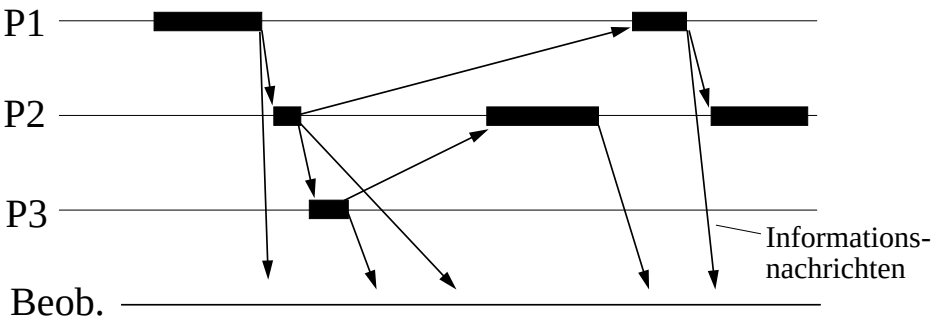
Beweisbare math. Eigenschaft

Diese "stabile" Stagnationseigenschaft ist *problemunabhängig*!

- Was ist adäquat?
- Wie feststellen?

Globaler ggT-Beobachter?





Bekommt der Beobachter ein "richtiges Bild" des Geschehens?

- was heißt das genau?
- und wenn Informationsnachrichten verschieden schnell sind?
- könnte der Beobachter einen Zwischenzustand (z.B. "alle haben den Wert 12") irrtümlich als Endzustand interpretieren?
- wie stellt er überhaupt das Ende der Berechnung fest?

Übungen

- Man zeichne Raum-Zeit-Diagramme für verschiedene Abläufe des verteilten ggT-Algorithmus

- Wie kann man beweisen, daß für *jeden* denkbaren Ablauf das Endergebnis stets der ggT ist?

- Bleibt der Algorithmus (und/oder der Beweis) korrekt, wenn im Algorithmus x<M durch x≤M ersetzt wird?

- Man vergleiche die verteilte Berechnung des ggT-Algorithmus für zwei Zahlen mit dem üblichen sequentiellen ggT-Algorithmus für zwei Zahlen

- Genügt es auch, nur in Uhrzeigerrichtung eine Nachricht zu senden (anstatt an beide Nachbarn)?

- Kann statt des Ringes eine andere Topologie verwendet werden? Welche?

- Formalisieren Sie für Zeitdiagramme den Begriff (potentiell, indirekt) "kausal abhängig" als Halbordnung über "Ereignissen"

- Wie kann man erreichen, daß ein ggT-Beobachter (der über jede Wertänderung eines Prozesse informiert wird) eine "kausaltreue" Beobachtung macht?

- Beobachtungen sind eine lineare Ordnung von (beobachteten) Ereignissen. In welcher Beziehung steht die oben erwähnte Halbordnung zu dieser linearen Ordnung? Können Sie eine Vermutung darüber anstellen, was der Schnitt aller möglichen kausaltreuen Beobachtungen einer verteilten Berechnung aussagt?

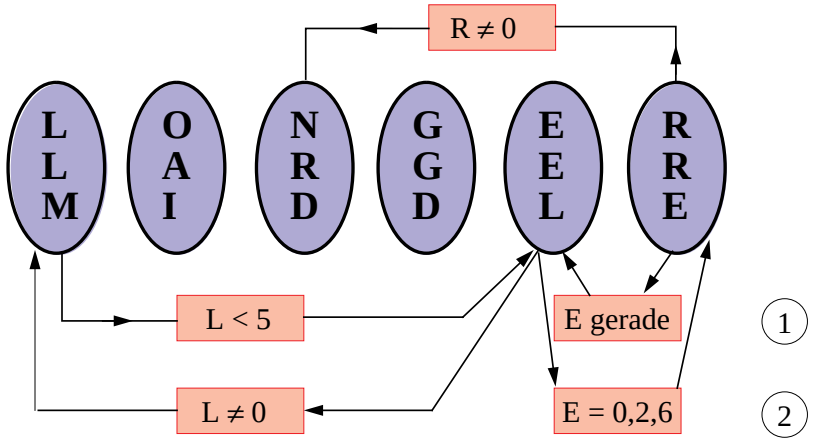
- Wie kann der Beobachter die Terminierung erkennen?

Ein weiteres Beispielproblem: Paralleles Lösen von "Zahlenrätseln"

```
 LONGER          207563
+LARGER    →    +283563
-------         -------
 MIDDLE          491126
```

Pro Spalte ein Prozeß

- Reaktives Verhalten: Auslösen atomarer Aktionen
- Propagieren neuer Erkenntnisse (= Einschränkungen)
  ("parallele Constraint-Propagation")



Endergebnis:

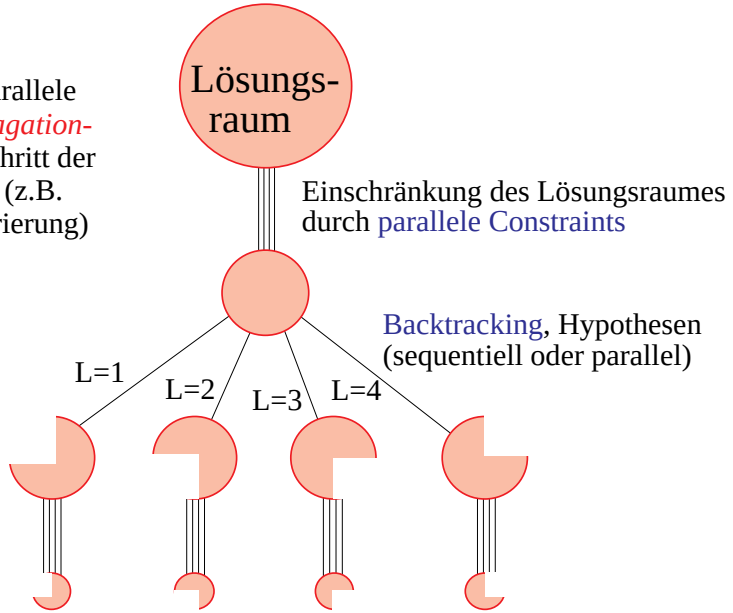
$1 \le L \le 4$
$M \ge 2$
R = 1,3,5,6,8
E = 0,2,6
sonst keine Einschränkung

Probleme:

1) Keine eindeutige Lösung
   --> Backtrack-Algorithmus
2) Entdeckung der "Stagnation"?
   (Ende der Parallelphase)

Offensichtlich *gleiches Schema* wie ggT-Berechnung!

Der aufgesetzte Backtrack-Algorithmus



Abwechselnd: Parallele *Constraint-Propagation-Phase* und ein Schritt der *Backtrack-Phase* (z.B. Hypothesengenerierung)

Einschränkung des Lösungsraumes durch parallele Constraints

Backtracking, Hypothesen (sequentiell oder parallel)

- Hypothese = beliebige Menge von Constraints
  (von einem “Orakel” statt von einer Spalte)

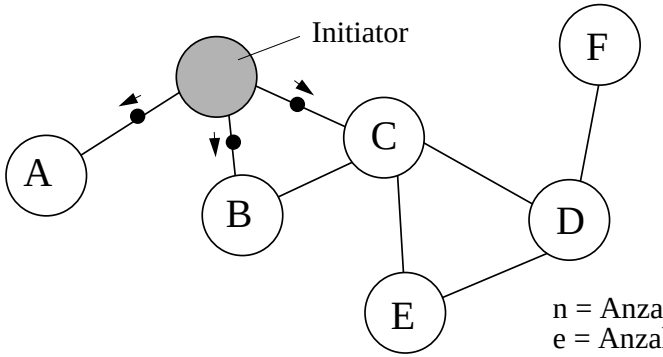
- Einheit, die die Hypothesen generiert und verwaltet, muß die Terminierung einer Constraint-Phase feststellen können
  - Wie würde man hier die Terminierung zweckmäßigerweise definieren?
  - Problembezogen wie bei ggT ("alle Werte identisch") hier nicht einfach.
  - "Alle passiv und keine sinnvolle Nachricht mehr unterwegs"?

Bemerkungen:

- Nicht unbedingt beste Parallelisierungsstrategie!
- Problem ist (in Verallgemeinerung) NP-vollständig

Informationsverteilung durch “Fluten”

“flooding”



n = Anzahl der Knoten
e = Anzahl der Kanten

- Voraussetzung: zusammenhängende Topologie
- Prinzip: jeder erzählt *neues* Gerücht allen anderen Freunden
- Kein Routing etc. notwendig

wieder das gleiche Prinzip wie beim ggT und beim Zahlenrätsel!

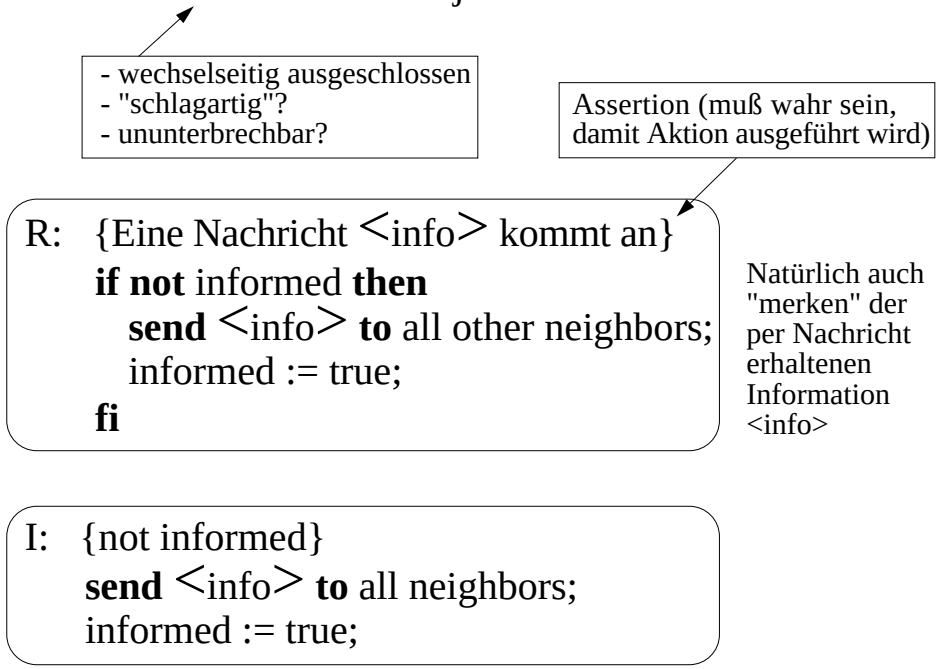
- Wieviele Nachrichten werden versendet?
  - Jeder Knoten sendet über alle seine inzidenten Kanten (--> 2e)
  - Jedoch nicht über seine Aktivierungskante zurück (--> -n)
  - Ausnahme: Initiator (--> +1)

  ==> *Also*: 2e - n + 1

- Frage: Wie *Terminierung* feststellen?
  D.h.: wie erfährt der Sender (= Initiator), wann alle erreicht wurden?
  (Das ist für “sicheren” oder “synchronen” Broadcast notwendig.)

Flooding-Algorithmus - eine etwas formalere Spezifikation

- Zwei *atomare Aktionen* für jeden Prozeß:



- initial sei informed=false

- Aktion R wird nur bei Erhalt einer Nachricht ausgeführt
  - "message driven"

- Aktion I wird vom Initiator *spontan* ausgeführt
  - Darf es mehrere *konkurrente* Initiatoren geben?

Terminierungserkennung von Flooding

1) Jeder Prozeß informiert (ggf. indirekt) Initiator (oder einen Beobachter) per *Kontrollnachricht*, wenn er eine *Basisnachricht* erhält. Initiator zählt bis 2e-n+1.

   - Nachteile?
     - n und e müssen dem Initiator bekannt sein
     - indirektes Informieren kostet ggf. viele Einzelnachrichten

   - Nachrichtenkomplexität?

   - Variante: Prozeß sendet Kontrollnachricht, wenn er *erstmalig* eine Basisnachricht erhält; Initiator zählt bis n-1.
     - n muß dem Initiator bekannt sein
     - Terminierung in dem Sinne, daß alle informiert sind - es können dann aber noch (an sich nutzlose) Basisnachrichten unterwegs sein

2) *Überlagerung* eines geeigneten Kontrollalgorithmus, der die Berechnung des Flooding-Verfahrens beobachtet und die Terminierung meldet.

   - Später mehr zu überlagerten Terminierungserkennungsverfahren

3) *Bestätigungsnachrichten* (acknowledgements)

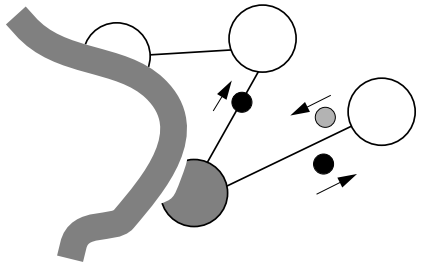
   - Direktes Bestätigen einer Nachricht funktioniert nicht
   - Indirekte Bestätigungsnachrichten: Ein Knoten sendet erst dann ein ack, wenn er selbst für allen seine Nachrichten acks erhalten hat
   - Klappt diese Idee? Auch wenn der Graph Zyklen enthält? Wieso?

Flooding mit Quittungsmeldungen

(Originalversion des *Echo-Algorithmus* von Chang ’82)

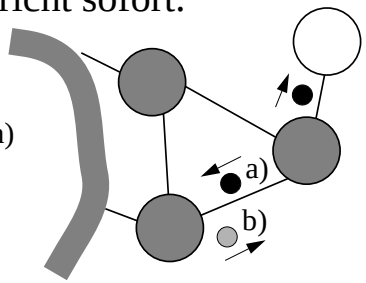
Prinzip: Ein Prozeß versendet eine Quittung für eine empfangene Nachricht erst dann, wenn er für alle von ihm selbst versendeten Nachrichten Quittungen erhalten hat.

- Ein Knoten mit Grad 1 ("Blatt") sendet sofort eine Quittung zurück.



Nachricht des zugrundeliegenden (flooding) Algorithmus

- Ein Knoten, der bereits eine Basisnachricht erhalten hat, quittiert jede weitere Basisnachricht sofort.
  - *Prinzip*: "Bin schon informiert"
  - *Wirkung*: Zyklen werden aufgebrochen (als wäre die Kante gar nicht vorhanden)
  - *Konsequenz*: Es entsteht ein Baum



- Terminiert, wenn Initiator alle Quittungen erhalten hat.

- Wieviele Quittungen / Nachrichten insgesamt?

Der Echo-Algorithmus

(PIF-Variante von A. Segal, 1983)

Propagation of Information with Feedback

- Ähnliches Verfahren 1980 von Dijkstra/Scholten: "Diffusing Computations"

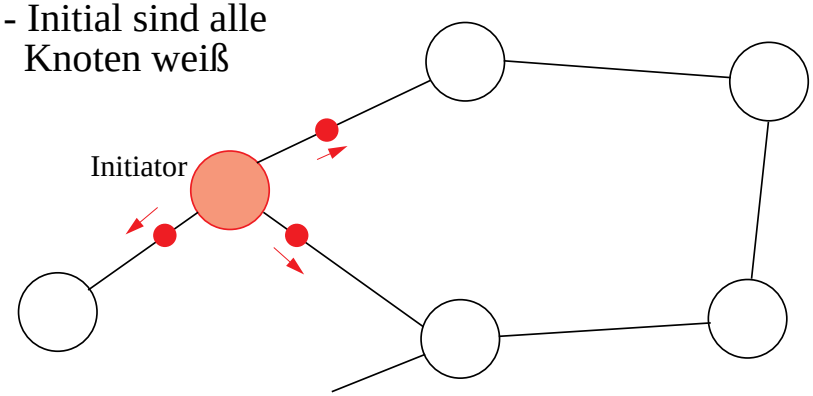
Ausgehend von einem einzigen Initiator:

- Paralleles *Traversieren* eines bel. (zusammenhängenden ungerichteten) Graphen mit 2e Nachrichten
- Terminierung klar durch "Vollzugsmeldung"
- Idee: *Indirektes* acknowledge
- Hinwelle durch "*Explorer*": Verteilen von Information
- Rückwelle durch "*Echos*": Einsammeln einer verteilten Information
- Aufbau eines *spannenden Baumes* ("Echo Kanten": jeder Knoten sendet genau ein Echo)

Paralleler *Wellenalgorithmus*:

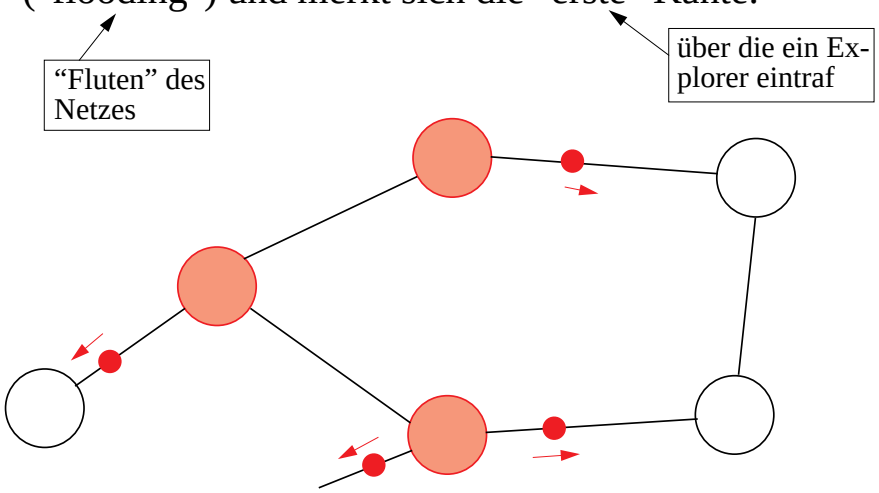
- virtueller broadcast
- *Basisalgorithmus* für andere Verfahren ("underlying algorithm"; "superimposition")

- Initial sind alle Knoten weiß

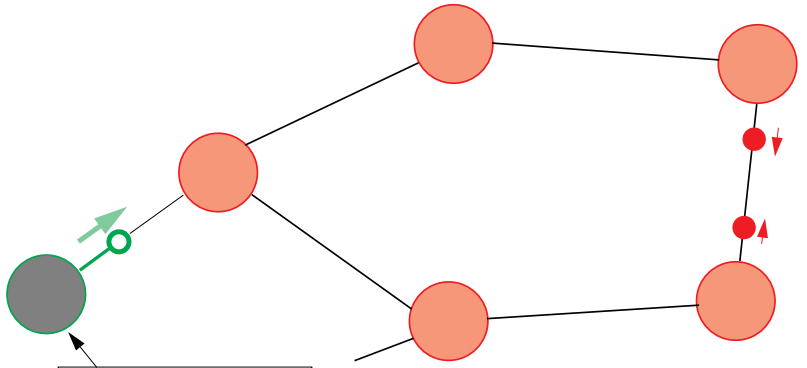


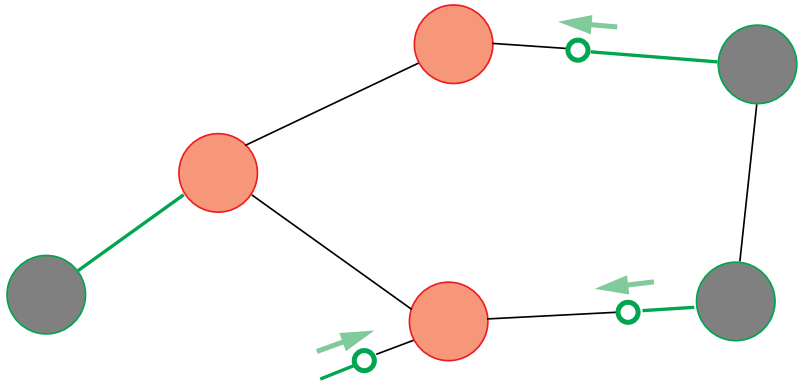
- Der (eindeutige) Initiator wird rot und sendet (rote) *Explorer* über alle seine Kanten.

- Ein weißer Knoten, der einen Explorer bekommt, sendet Explorer über alle seine anderen Kanten ("flooding") und merkt sich die "erste" Kante.



- Ein (roter) Knoten, der über alle seine Kanten einen Explorer *oder* ein Echo erhalten hat, wird grün und sendet ein (grünes) *Echo* über seine "erste" Kante.

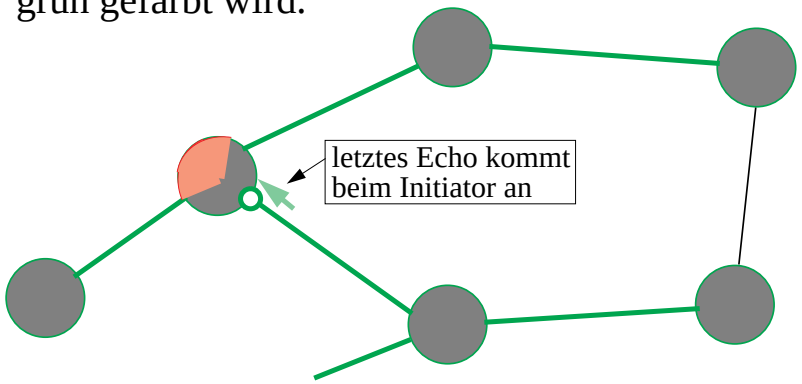




Beachte: *Atomare Aktionen* --> Explorer können sich höchstens auf Kanten begegnen, nicht aber bei Knoten!

Auf einer Kante, wo sich zwei Explorer begegnen, wird der Zyklus aufgebrochen.

- Das Verfahren ist beendet, wenn der Initiator grün gefärbt wird.



- Grüne Kanten bilden einen *Spannbaum*.
  - alle Knoten bis auf den Initiator haben eine "erste" Kante
  - “grüner Graph” ist zusammenhängend

- Über jede Kante laufen genau 2 Nachrichten:
  entweder ein Explorer und ein gegenläufiges Echo, oder zwei Explorer, die sich begegnen --> Nachrichtenkomplexität = 2e

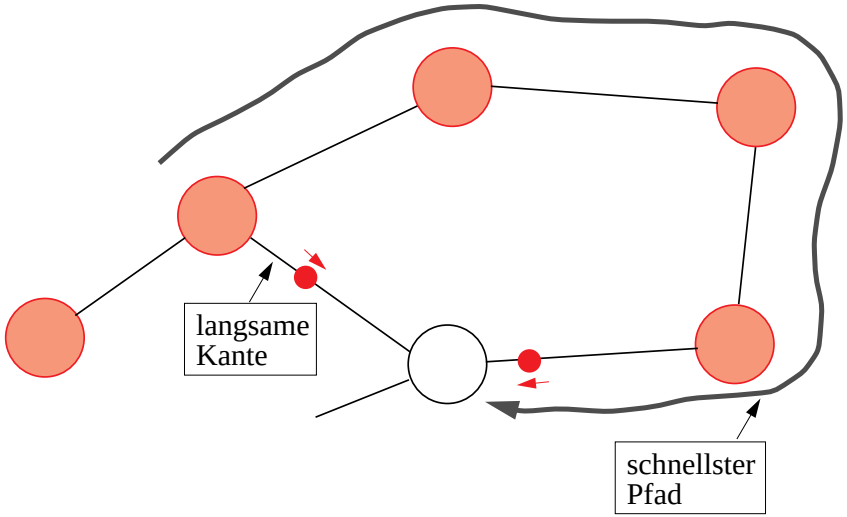
- Verfahren ist *schnell*: parallel und "bester" (?) Baum.

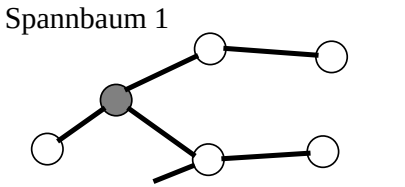
- Ereignis "rot werden" in jedem Prozeß definiert eine Welle (*Hinwelle*).

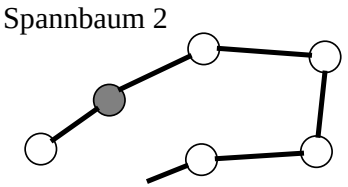
- Ereignis "grün werden" in jedem Prozeß --> *Rückwelle*.

- Es darf nicht mehr als einen Initiator geben:
  - was geschieht sonst?
  - wie kann man dem Problem mehrerer Initiatoren begegnen?

Echo-Algorithmus ist *nicht-deterministisch*, es können (je nach Geschwindigkeit einer "Leitung") verschiedene spannenden Bäume entstehen!



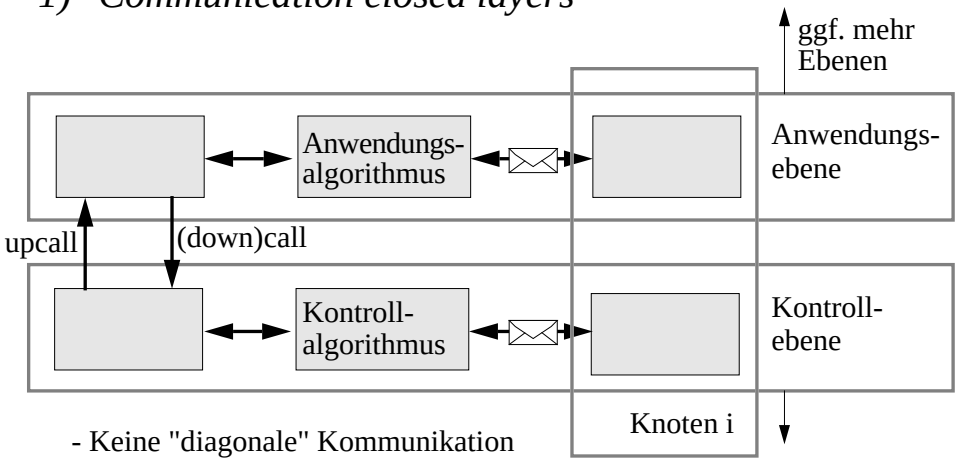




- Inwiefern ist diese Variante besser als die Originalversion?
  - Nachrichtenkomplexität
  - Einfachheit / Eleganz

Überlagerung ("Superposition")

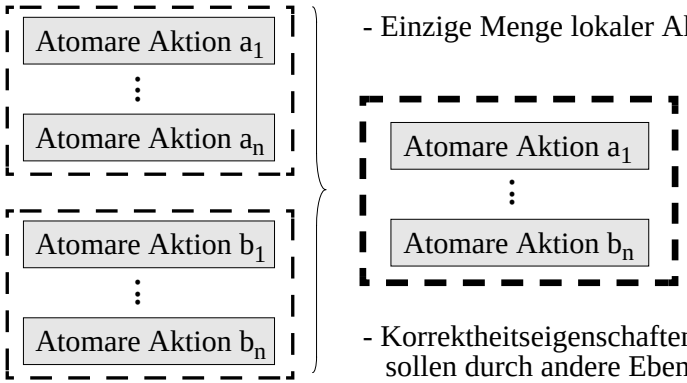
1) "Communication closed layers"



- Keine "diagonale" Kommunikation
- call und upcall (innerhalb eines Knotens) nicht notw. mittels Nachrichten sondern i.a. durch Aufruf lokaler Aktionen (z.B. Prozeduren)
- Kommunikation zwischen der Anwendungs- und Kontrollebene (innerhalb eines Knotens) typw. über gemeinsame Variablen (wobei i.a. nur eine Ebene schreiben darf).

---

2) Zusammenbau und Vereinigung von Aktionen



- Einzige Menge lokaler Aktionen
- Korrektheitseigenschaften einer Ebene sollen durch andere Ebenen nicht verletzt werden!

Echo-Algorithmus und upcall-Technik

```
receive <ECHO(...)> or <EXPLORER(...)> from p;
if COLOR = white then
    upcall "first EXPLORER(...) received";
    COLOR := red;
    N := 0; PRED := p;
    send <EXPLORER(...)> to neighbors\{PRED};
fi;
if echo received then upcall "ECHO(...) received"; fi;
N := N+1;
if N = | neighbors | then
    COLOR := green;
    if INITIATOR then upcall  "terminated";
                 else upcall "ready to send echo";
                      send <ECHO(...)> to PRED;
    fi;
fi;
```

Mit "upcalls" wird die darüberliegende Anwendung vom Echo-Algorithmus benachrichtigt; die Anwendung kann entweder die Kontrolle sofort zurückgeben oder z.B. Parameterwerte vorbereiten, die dann mit Nachrichten des Echo-Algorithmus mitgesendet werden.

```
{COLOR = white}
INITIATOR := true;
COLOR := red;
N := 0;
send <EXPLORER(...)> to neighbors;
```

Aktion, die von der Anwendung mit einem "normalen downcall" gestartet wird.

Echo-Algorithmus...

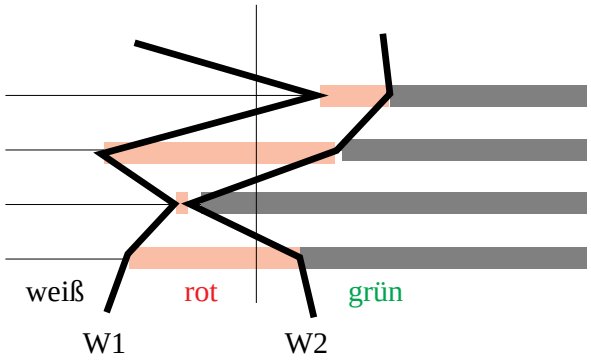
- Jeder Knoten wird erst *rot* und dann *grün*.
- Rote Phase ist bei "Blättern" allerdings recht kurz.
- Ein grüner Knoten hat keine weiße Nachbarn.

==> Eine von einem grünen Knoten versendete Basisnachricht wird nicht von einem weißen Knoten empfangen.

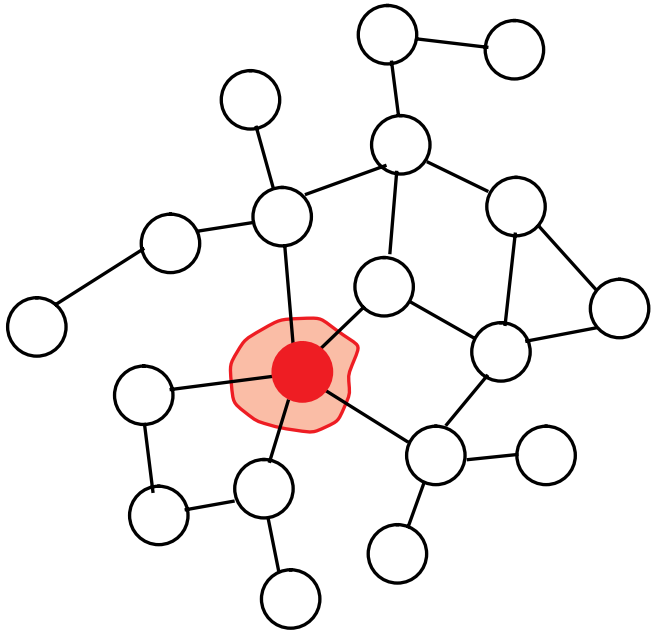
- Beachte: Gilt nur für *direkte* Nachrichten, nicht für *Nachrichtenketten!*

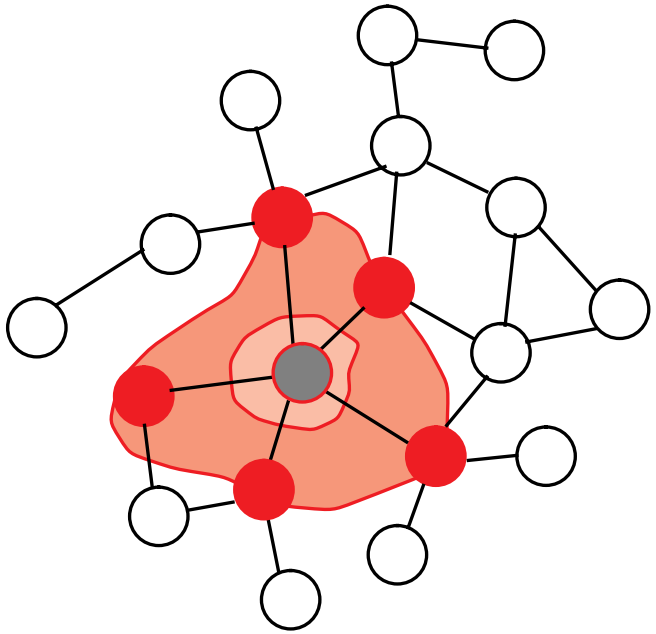
==> Weiße und grüne Phase sind in "gewisser Weise" disjunkt.

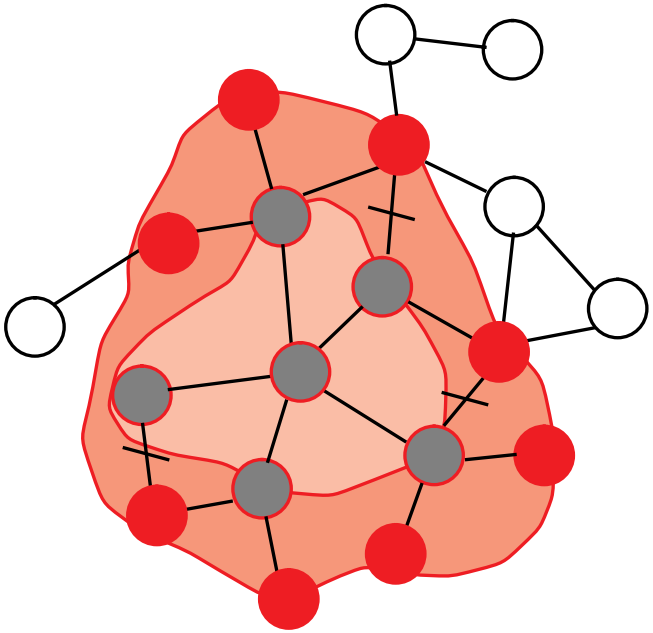
- Obwohl es globale Zeitpunkte geben kann, wo ein Knoten bereits grün ist, während ein anderer (nicht direkt benachbarter!) noch weiß ist!

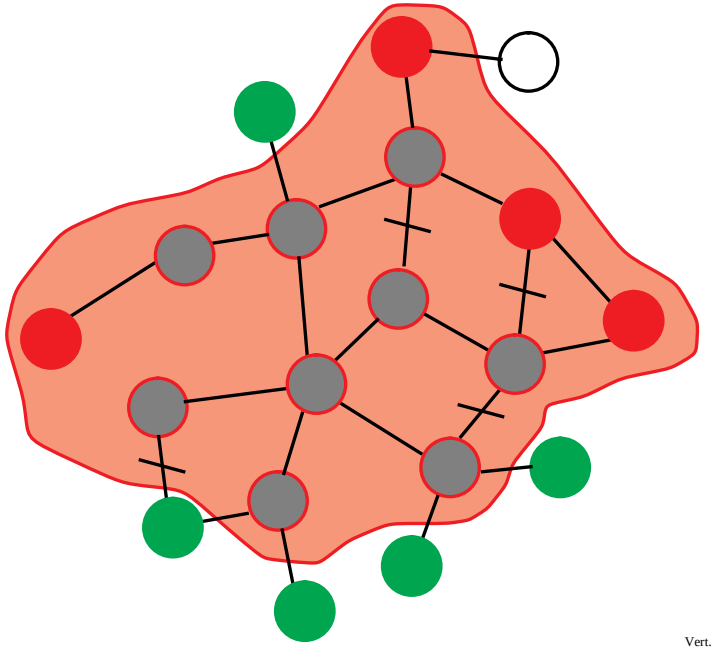


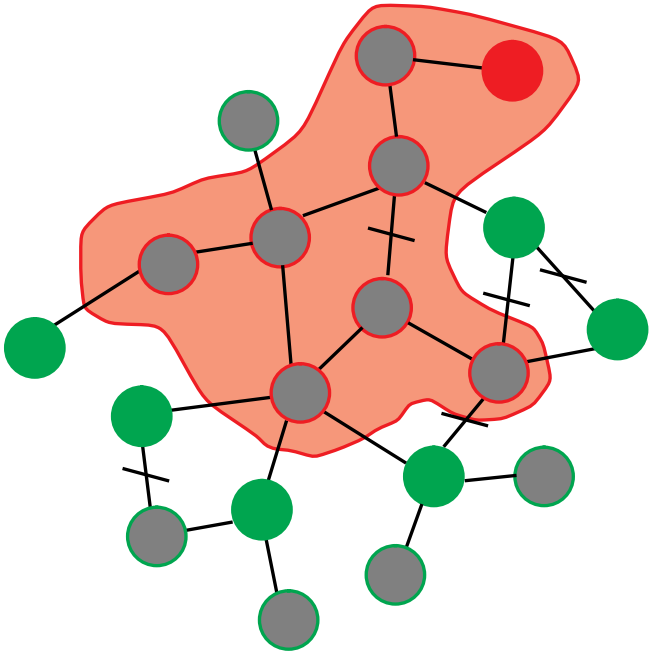
- "Rot werden" und "grün werden" definieren zwei *Wellen*.
  - Mit diesen Wellen kann Information transportiert werden (verteilen bzw. akkumulieren).
  - Echo-Algorithmus wird daher oft als Basis für andere Verfahren verwendet.

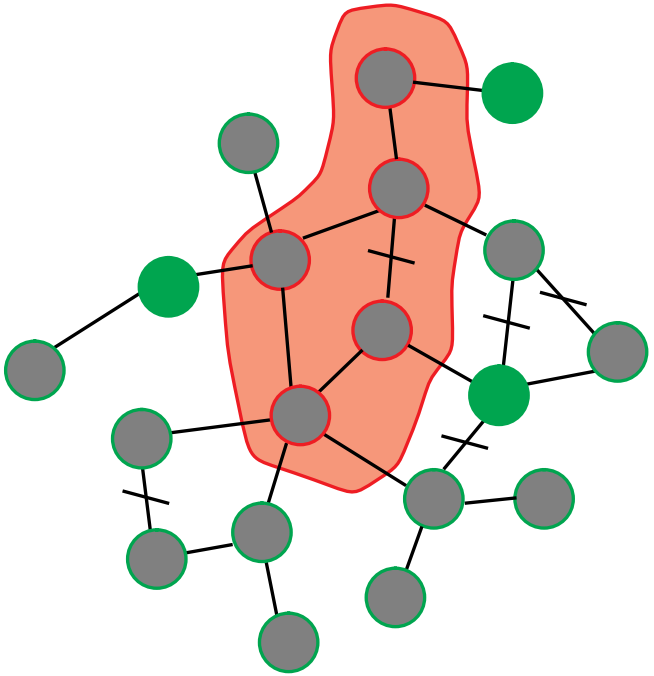


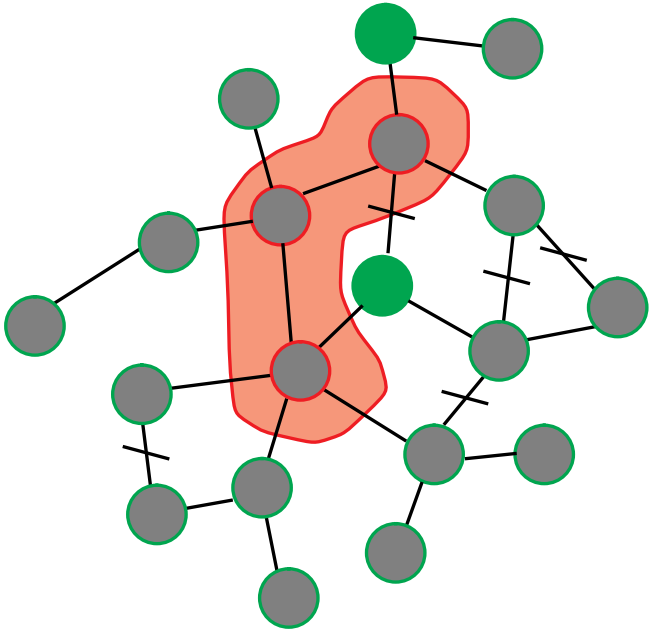


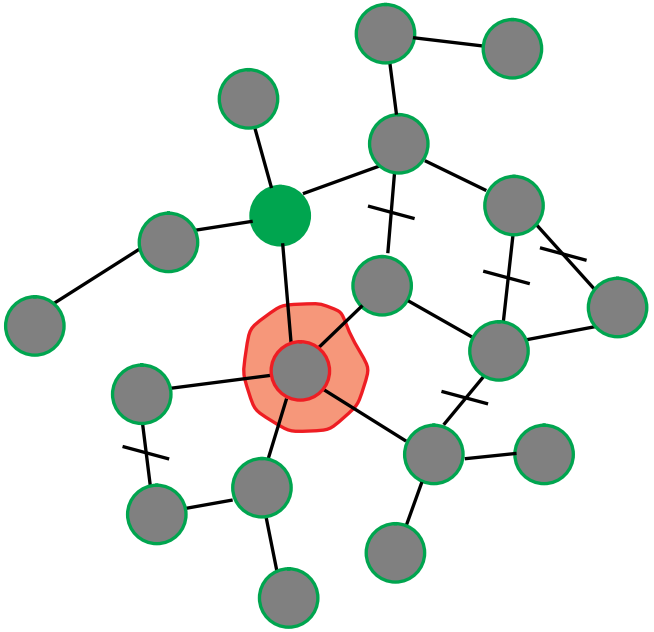


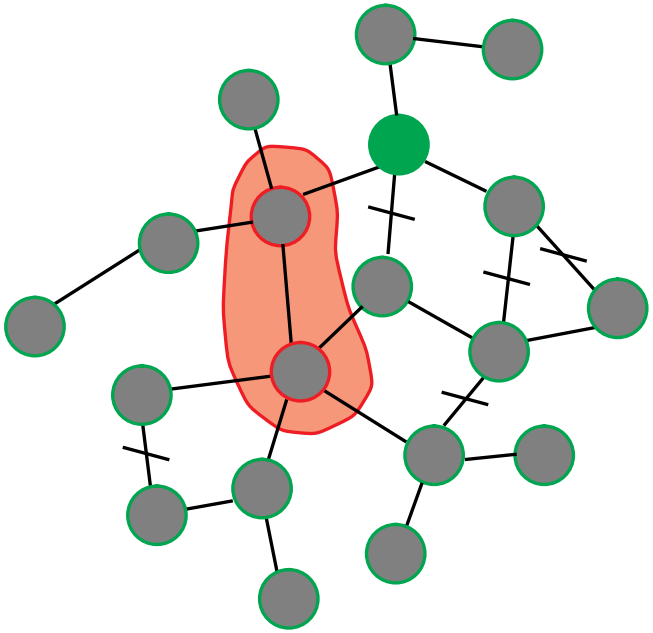


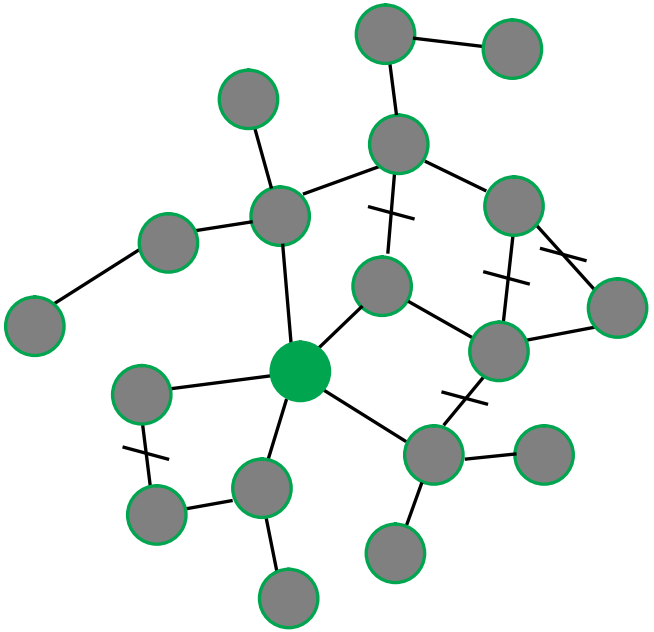










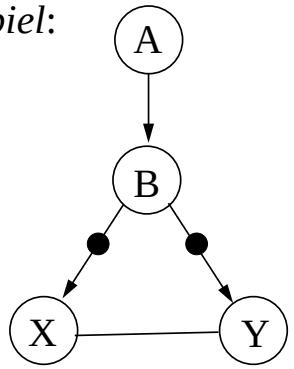


Verbesserung des Echo-Algorithmus?

[Helary et. al.]

- *Idee*: vermeide Besuch von Knoten, von denen man weiß, daß sie von anderen Explorern besucht werden

Beispiel:



Nachricht von B an X enthält Information, daß Y nicht besucht zu werden braucht.

- *Voraussetzung*: Identitäten der Nachbarn bekannt

Schema (Modifikation gegenüber PIF-Echo):

Menge von Knotenidentitäten

```
receive <..., z>
...
y := neighbors \ z
send <..., z ∪ y> to all y
(* Registrieren, über welche Kanäle
Echos oder Explorer eintreffen müssen *)
```

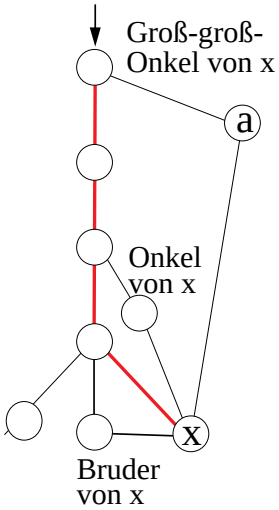
Statt neighbors ohne Vater im Original

- Initiator i: send <..., neighbors ∪ {i} >

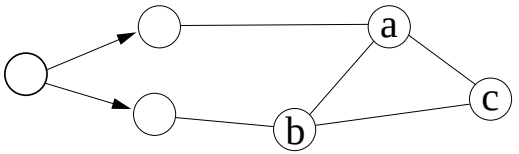
Was wird gespart?

- Keine Nachricht an einen Vorgänger oder einen direkten Nachbarn eines Vorgängers (z.B. "Brüder" und "Onkel")

- Allerdings: Obwohl x nicht an a sendet, sendet a an x! Über diese Kante fließt dann auch ein Echo zurück --> nichts gespart, da 2 Nachrichten über die Kante!



- Auch in diesem Fall spart man nicht (immer?) etwas:



Knoten a und b werden "gleichzeitig" erreicht: wissen nichts voneinander: --> senden beide gegenseitig und an c.

- Wieviel wird bei vollständigen Graphen gespart? Und bei Bäumen? Spielt der "Vermaschungsgrad" eine Rolle?

- Ersparnis nicht ganz klar.

- Nachteile:
  - Lange Nachrichten (O(n))
  - Nachbaridentitäten müssen bekannt sein

Zeitkomplexität

Beachte: Algorithmen i.a. nichtdeterministisch --> *mehrere* mögl. Berechnungen!

Variable Zeitkomplexität eines vert. Algorithmus = max. "Zeit" aller Berechnungen des Algo unter:

Z1: Lokale Berechnungen erfolgen in Nullzeit
Z2: Eine Nachricht benötigt *maximal* 1 Zeiteinheit

Einheitszeitkomplexität eines vert. Algorithmus = max. "Zeit" aller Berechnungen des Algo unter:

E1: Lokale Berechnungen erfolgen in Nullzeit
E2: Eine Nachricht benötigt *exakt* 1 Zeiteinheit

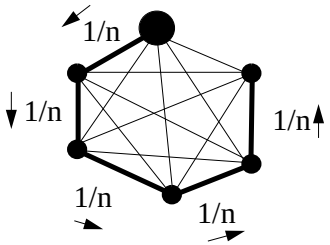
Behauptung:
Es gilt *nicht* immer var. Zeitkplx ≤ Einheitszeitkplx.

- Grund: Einheitszeitkplx erlaubt nicht alle Berechnungen!
- Frage: Gilt Umkehrung?

Bsp. Echo-Algorithmus auf vollständigem Graph

(1) Einheitszeitkomplexität = 3 ← Phase 1: Alle werden rot
Phase 2: Alle bis auf Initiator werden grün
Phase 3: Initiator wird grün

(2) Variable Zeitkomplexität ≥ n



- Explorer "außen": 1/n Zeiteinheiten
- Jede sonstige Nachricht 1 Zeiteinheit
- Entarteter Baum Tiefe n-1 nach einer Zeiteinheit aufgebaut
- Echo beim Initiator nach n Einheiten

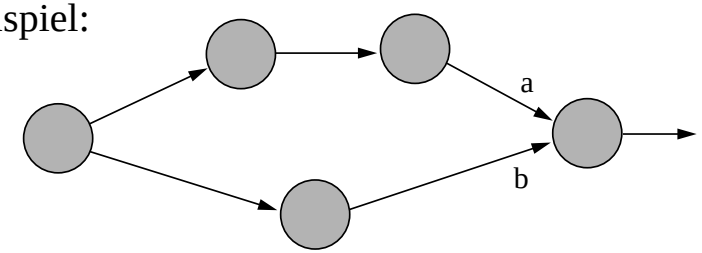
Zeitkomplexität: Welche Definition?

- *Einheitsztkplx*: Einige Berechnungen bleiben unberücksichtigt!

Nicht bei var. Ztkplx! (Wieso?)

unwahrscheinliche?

Beispiel:



Trifft a vor b ein --> sehr lange Berechnung, sonst terminiert

mag vielleicht in 10% aller Fälle der Fall sein...

Aber wie oft wirklich?

Systemabhängig!

- *Variable Ztkplx*: Resultat wird u.U. durch extrem unwahrscheinliche Berechnungen bestimmt

- Für worst-case: variable Ztkplx? Aber: average case?

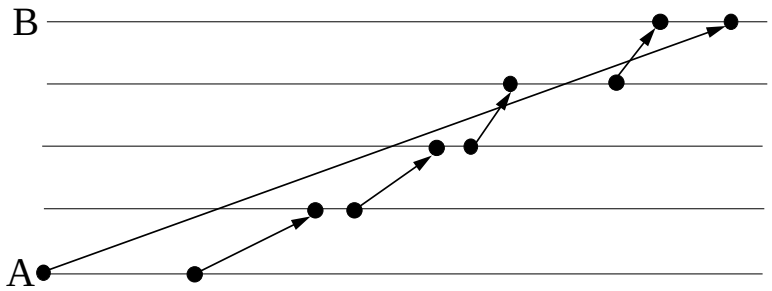
- Genauer: Wahrscheinlichkeitsverteilung --> Erwartungswert
  - systemabhängig
  - schwierig
  - jeden Tag anders...

Ein anderes Zeitkomplexitätsmaß

Längste Nachrichtenkette einer Berechnung

Dann max. (oder Mittelwert?) über alle Berechnungen

Beispiel:



Sende erst direkt, dann indirekt an B

- Prozeß A initiiert den Algorithmus
- Beendet, wenn B direkt oder indirekt von B hört
  (Also: Wenn B eine Nachricht empfängt)

- Einheitsztkplx. = 1
- Var. Ztkplx. = 1
  (worst case)
- Längste Kette = 4

größer!

- Wie sinnvoll ist dieses Maß?
- Was ist das "richtige" Maß für die Zeitkomplexität?

Bem.: Solche Ketten spielen im Sinne eines "critical path" bei Beschleunigungsuntersuchungen auf Parallelrechnern eine Rolle.

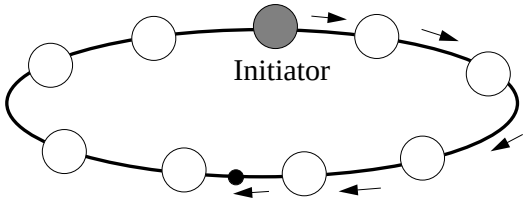
Broadcast auf speziellen Topologien

- Echo-Algorithmus realisiert einen Broadcast

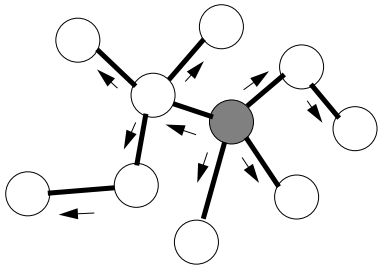
auf bel. zusammenh. Topologie

  - Verteilen von Information ausgehend von einem Initiator
  - für beliebige (zusammenhängende) Topologien
  - liefert sogar "Vollzugsmeldung" durch Echo-Nachrichten

- Auf *speziellen* Topologien läßt sich der Broadcast auch effizienter realisieren
  - Beispiel *Ring*: Ein "Token" zirkuliert mit der Information; alle sind informiert, wenn das Token wieder beim Initiator eingetroffen ist
  - ggf. kann einer anderen Topologie ein Ring überlagert werden



- Beispiel *(Spann)baum* (tatsächlich Unterschied zum Echo-Algorithmus?)



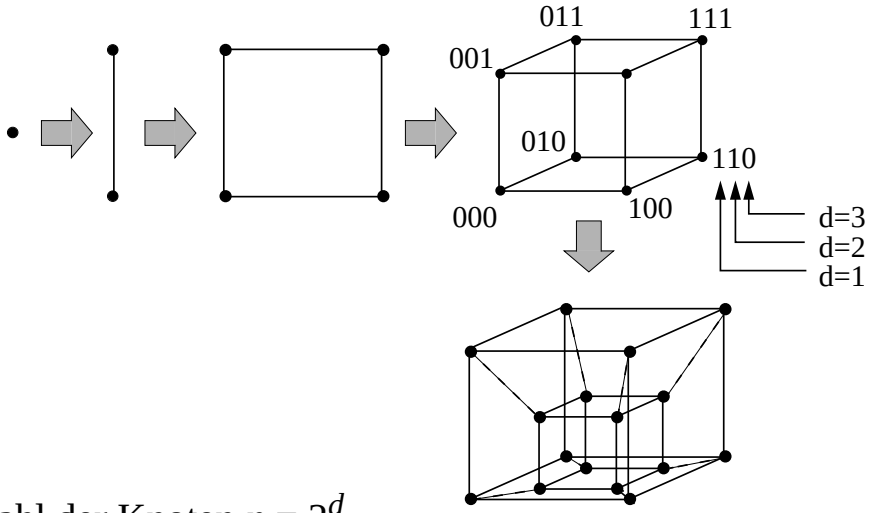
vorausgesetzt wird jeweils, daß der Algorithmus "weiß", daß eine spezifische Topologie vorliegt!

- Beispiel *vollständiger Graph* (als Denkübung)

Broadcast in Hypercubes (1)

- Hypercube = "Würfel der Dimension d"
- Rekursives Konstruktionsprinzip
  - Hypercube der Dimension 0: Einzelrechner
  - Hypercube der Dimension d+1:

    „Nimm zwei Würfel der Dimension d und verbinde korrespondierende Ecken“



- Anzahl der Knoten $n = 2^d$
- Anzahl der Kanten = $d\ 2^{d-1}$ (Ordnung O(n log n))
  - viele Wegalternativen (Fehlertoleranz, Parallelität!)
  - maximale Weglänge: d = log n
  - mittlere Weglänge: d/2 (Beweis als Denkübung!)

wieviele verschiedene Wege der Länge k gibt es insgesamt?

- Knotengrad = d (nicht konstant bei Skalierung!)
- Einfaches Routing von einzelnen Nachrichten
  - xor von Absende- und Zieladresse...

Kombinatorische Aspekte (1)

- Wieviele verschiedene Wege der Länge k (ausgehend von Knoten 0) gibt es?
  - man wiederhole die Begriffe und Methoden der Kombinatorik...
  - für den Anfang: was ergibt sich für k=1, k=n?
- Ein ähnliches Problem: Wege in Manhatten...

Brian Hayes, American Scientist, January-February 1996

A Walk in Manhattan (Auszug)

When I was a commuter in New York, I walked a diagonal route across midtown Manhattan morning and evening. I used to wonder how many different paths I could find through the grid of east-west and north-south streets without taking extra steps. Eventually I grew curious enough to calculate some solutions. For a square lattice of n-by-n blocks, any minimum-length diagonal path is necessarily 2n blocks long. How many such paths are there? For the 1-by-1 lattice the answer is 2: You can go east and then north or you can go north and then east. In a two-block square there are six paths, and in a 3-by-3 block there are 20. The sequence of values I calculated begins like this: 2, 6, 20, 70, 252, 924, 3432, 12870, 48620....

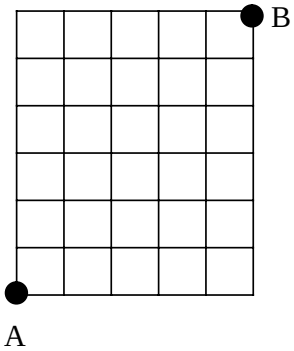
Do those numbers look familiar? They should. They are prominent members of one of the most ancient and famous families of numerical sequences. And yet I did not identify them until several years later, when I had a chance to submit them to Sloane's sequence server. The answer came back immediately: They were recognized as sequence M1645, the central binomial coefficients, the numbers that run down the middle of Pascal's triangle. My reaction was "of course! Why didn't I see it all along?" But I doubt that I would have made the connection without help.

Kombinatorische Aspekte (2)

The discovery was a productive one, which led not just to an answer but to an insight. I saw that counting the shortest diagonal paths for all rectangular lattices--not just the square ones--would fill in the rest of Pascal's triangle. Each row of the triangle consists of all the lattices with a given minimum diagonal path length. For example, the fifth row includes all the lattices with a path length of 4, namely the 0-by-4, 1-by-3, 2-by-2, 3-by-1 and 4-by-0 lattices. The corresponding counts of diagonal paths (and the corresponding entries in Pascal's triangle) are 1, 4, 6, 4 and 1.

http://www.sigmaxi.org/amsci/issues/comsci96/compsci96-01.html

---

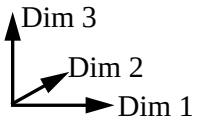


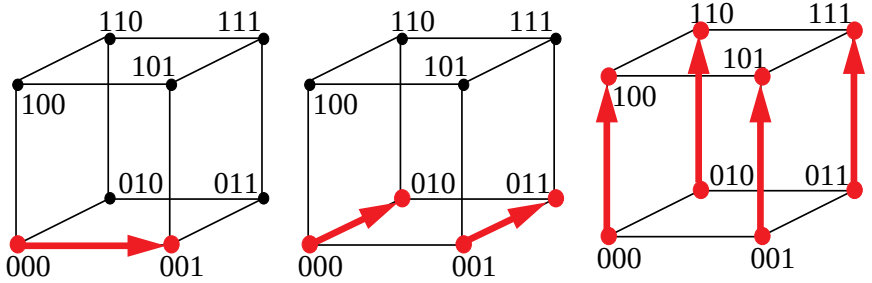
- Jeder kürzeste Weg von A nach B hat die Länge 11
- Repräsentation durch ein Wort der Länge 11
- 5 Nullen und 6 Einsen
- 0: gehe nach Osten
- 1: gehe nach Norden
- Wieviele solche Worte gibt es?

- Gegeben sei eine 11-elementige Menge
- Dem i-ten Element sei die Position i (in einem Wort) zugeordnet
- Anzahl der 6-elementigen Teilmengen = Anzahl der Worte der Länge 11 mit genau 5 Nullen und 6 Einsen
- ==> Binomialkoeffizienten

Broadcast in Hypercubes (2)

- Initiator habe die Nummer 00...00 (binär)
- Wir verzichten hier auf Vollzugsmeldung
  (also keine Acknowledgemnents oder Endeerkennung)





- Analog zum rekursiven Aufbau des Hypercube
  - zunächst in Dimension 1 senden: Teil-Hypercube der Dimension 1 ist damit informiert
  - dann senden alle Knoten der Dimension 1 in Dimension 2
  - etc.
- Nach d "Takten" sind alle Knoten informiert
  - Zeitkomplexität ist daher d (unter welchem Zeitmaß?)
  - Nachrichtenkomplexität: $1 + 2 + 4 + ... + 2^{d-1} = 2^{d-1} - 1 + 2^{d-1} = n-1$
- Welche Komplexität hat ein *optimaler* Broadcast-Algo.?
- Geht es besser? ← was heißt überhaupt "besser"?
  - Algorithmus startet ziemlich "langsam": am Anfang geschieht wenig parallel!
  - Kann man dies durch *gleichzeitiges* Versenden "in alle Richtungen" beschleunigen?

Broadcast in Hypercubes (3)

- Ein anderes Verfahren (Vergleich als Denkübung!)

- Initiator sendet an alle seine Nachbarn:

  0...01, 0...010, 0...100, ... , 10...0

  in “kanonischer” Numerierung

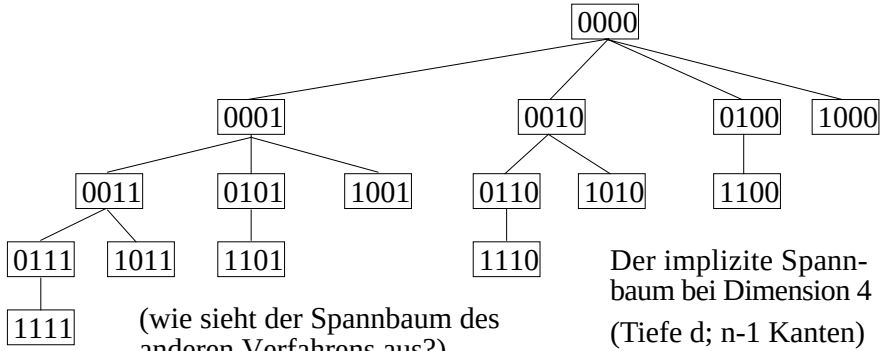
  am besten gleichzeitig, wenn dies technisch geht!

  linkeste 1 | beliebiges Restmuster

- Ein Knoten mit der Nummer 0...01x...y...z leitet die Information an alle seine “höheren” Nachbarn weiter:

  0...00**1**1x...y...z
  0...0**1**01x...y...z
  0...**1**001x...y...z
  ...
  10...001x...y...z

  Von welchem (eindeutigen) Knoten X wird Knoten Y informiert?
  Setze *vorderste 1* von Y auf 0
  --> = Nummer von X



(wie sieht der Spannbaum des anderen Verfahrens aus?)

Der implizite Spannbaum bei Dimension 4
(Tiefe d; n-1 Kanten)

- Der Algorithmus wird z.B. in Mehrprozessorsystemen (z.B. NCube) verwendet
- Wie effizient ist der Algorithmus? (Geht es besser?)
- Denkübung: Formuliere Algorithmus für einen beliebigen Initiator (schließlich sind Hypercubes symmetrisch...)
- Denkübung: Vergleich mit Flooding bzw. Echo-Algorithmus

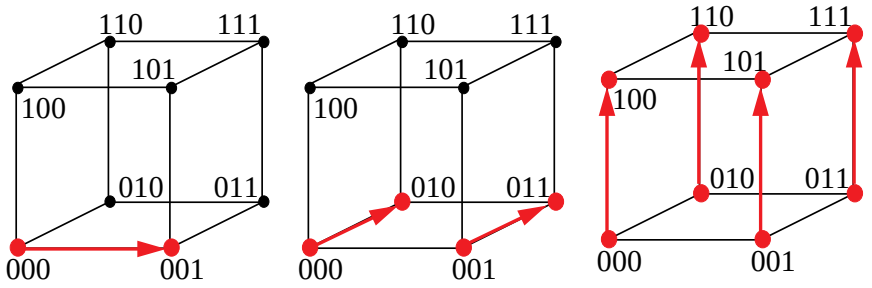
Noch ein anderer (besserer?) Algorithmus

- Beobachtungen:
  - Ein Baum verwendet im Hypercube relativ wenig Kanten --> schlechte Ausnutzung potentielle Parallelität
  - Es gibt *mehrere* Spannbäume in Hypercubes --> diese nutzen?

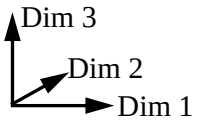
- Sender 0...0 teilt die Nachricht in d Pakete
- Sender startet für jedes Paket eine eigene “Welle”:
- *1. Paket* in *Dimension 1* senden --> 0...01
- Dann: Alle informierten Knoten (also 0...0 und 0...01) senden das Paket in Dimension 2
- Etc. Welle für Paket 1 breitet sich analog zur rekursiven Definition des Hypercubes in einer jeweils zusätzlichen Dimension aus
- Das *2. Paket* wird erst in *Dimension 2*, dann 3,..., d und erst zuletzt in Dimension 1 gesendet
- Das *3. Paket*: Dimensionsreihenfolge 3, 4, ..., d,1, 2
- Etc. Das *d.-Paket* in Dimensionsreihenfolge d, 1, 2,..., d-1

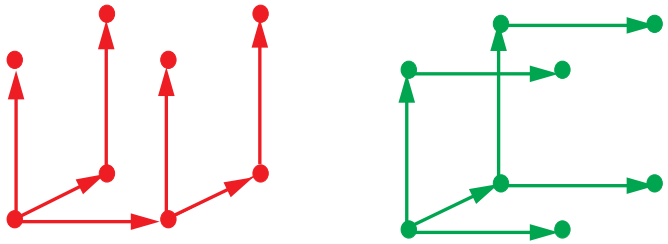
Denkübungen:
- Können so die Pakete gleichzeitig verschickt werden?
- Ist dann in jedem “Takt” pro Kante nur eine Nachricht unterwegs?
- Wieviele (kantendisjunkte ?) Spannbäume gibt es in einem Hypercube?

Veranschaulichung des Algorithmus



Die drei “Takte” der Welle von Paket 1

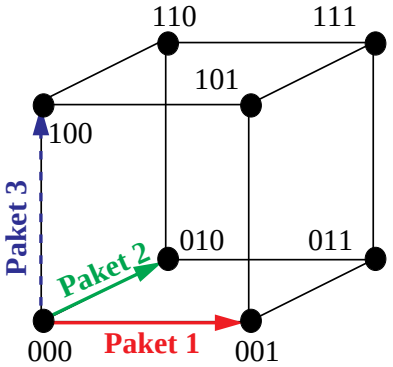




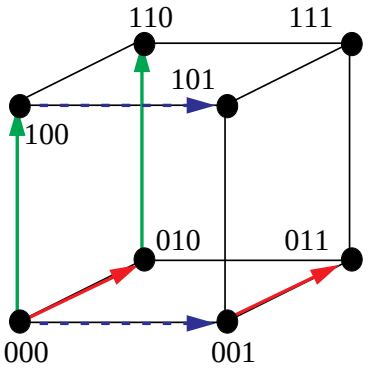
Die Spannbäume bzgl. Paket 1 und Paket 2

Parallelausführung der drei Wellen

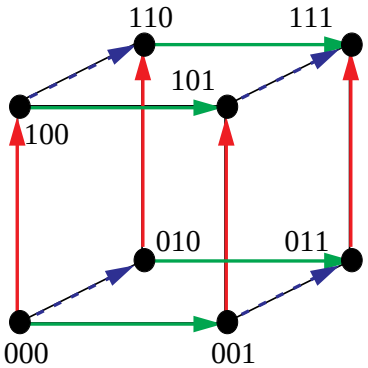
Takt 1



Takt 2



Takt 3



Takt

1 2 3

Paket 1: 1. 2. 3. Dim.

Paket 2: 2. 3. 1. Dim.

Paket 3: 3. 1. 2. Dim.

Pro Takt laufen die Pakete in jeweils unterschiedlicher Dimension!

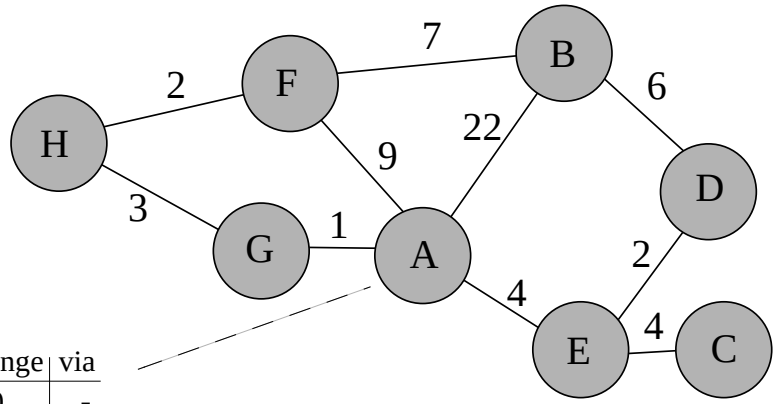
Es können also tatsächlich die drei Wellen parallel ausgeführt werden, ohne daß diese sich gegenseitig stören!

--> Dies ist das (im Prinzip) schnellere Verfahren!

Beachte: Ein globaler Takt ist gar nicht nötig!

Verteilte Berechnung von Routingmatrizen für kürzeste Wege

Gegeben: ungerichteter zusammenhängender Graph mit bewerteten Kanten ("Länge")



zu	Länge	via
A	0	-
B	22	B
C	∞	?
D	∞	?
E	4	E
F	9	F
G	1	G
H	∞	?

Wurde im Vorläufer des Internet (ARPANET) verwendet ("distance vector routing")

- Jeder kennt anfangs die Entfernung zu den Nachbarn
- "Spontanstart": Sende eigene Tabelle an Nachbarn
- Bei Empfang einer Tabelle über Kante mit Gewicht g:
  Für alle Zeilen i der Tabelle:
  Falls Nachricht.Länge[i]+g < Knoten.Länge[i]:
  ersetze Zeile (Länge := Länge+g; via := Absender)
- Falls sich Tabelle verändert hat:
  Neue Tabelle an alle Nachbarn (Ausnahme: Sender)
- Wie Terminierung feststellen?

- Ist eine verteilte Version des Bellman-Ford-Algorithmus
  - ähnlich dem bekannten Dijkstra-Algorithmus für kürzeste Wege
  - " Relexationsprinzip" (Bellman 1958, Dijkstra 1959, Ford 1962)

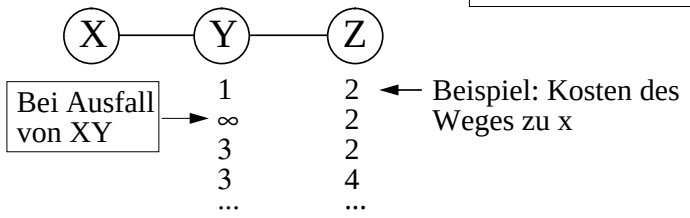
Kürzeste Wege in Rechnernetzen

- Algorithmus wird oft als dynamisches ("adaptives") Routing-Verfahren verwendet, wo in regelmäßigen Zeitabständen die Tabellen neu berechnet und ausgetauscht werden
  - bzw. dann, wenn sich etwas ändert (Kosten einer Verbindung, z.B. Ausfall einer Leitung oder Änderung der Lastsituation)

- Metrik für die Kosten z.B.:
  - (gewichtete)Anzahl der hops
  - Bitrate einer Verbindung
  - Verzögerung einer Verbindung (z.B. gemessen mit Testpaketen)
  - Länge der Warteschlange vor einer Verbindung

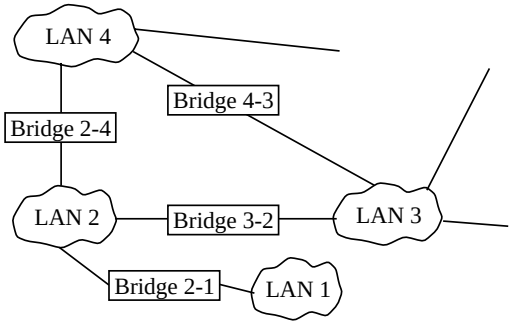
- "Count to infinity-Problem"

mehr zu diesen Dingen in der Vorlesung "Rechnernetze"



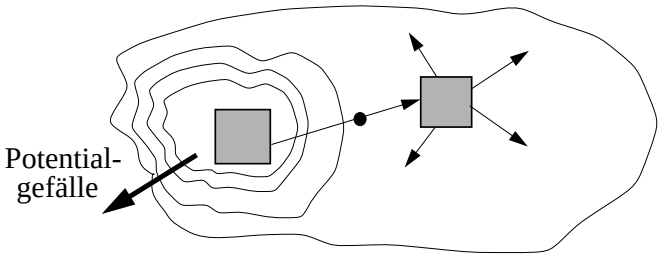
-Durch die kürzesten Wege zu einem festen Knoten ("Wurzel") ist ein kostenminimaler Baum gegeben

- Algorithmus wird in LANs eingesetzt, um einen Spannbaum zu bestimmen (Knoten = Teil-LAN; Kante = Bridge)
- Zyklenfreiheit ist wichtig, da kein Routing in LANs
- Ende wird heuristisch durch Abwarten einer Zeitspanne festgestellt



Das Paradigma der vert. Approximation

Prinzip:
- Anfang: Informiere alle Nachbarn spontan.
- Bei Empfang einer Nachricht:
  - Berechne neue Approximation.
  - Falls diese "besser": informiere Nachbarn.

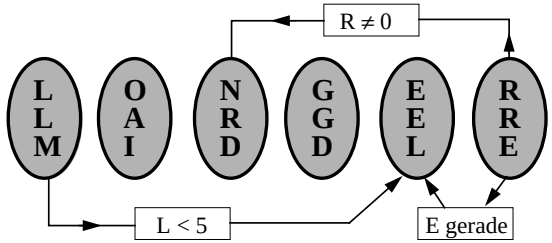


- *Nachrichtengesteuert* (aber "Spontanstart")
- Alle Prozesse arbeiten gleich, alle sind beteiligt
- *Nichtdeterministischer* Ablauf, determin. Ergebnis
- Beliebige stark zusammenhängende Topologie
- Assoziative Operatoren (min, max, $\cap$, $\cup$, +, and, or, ...)
- *Stagnation* bei globalem Gleichgewicht ("Optimum")
  --> Potentialunterschiede ausgeglichen
  --> Terminierungsproblem

Beispiele ("Instanzen der Algorithmenklasse"):
- ggT
- Zahlenrätsel
- Verteilen von Information ("Wissensausgleich")
- Routingmatrizen (inkl. Spannbaum)
- Maximale Identität ("election") } (noch) nicht behandelt
- Lastausgleich (Approx. eines dyn. Optimums) } (noch) nicht behandelt
- Relaxationsverfahren (Lösen von DGL) } (noch) nicht behandelt

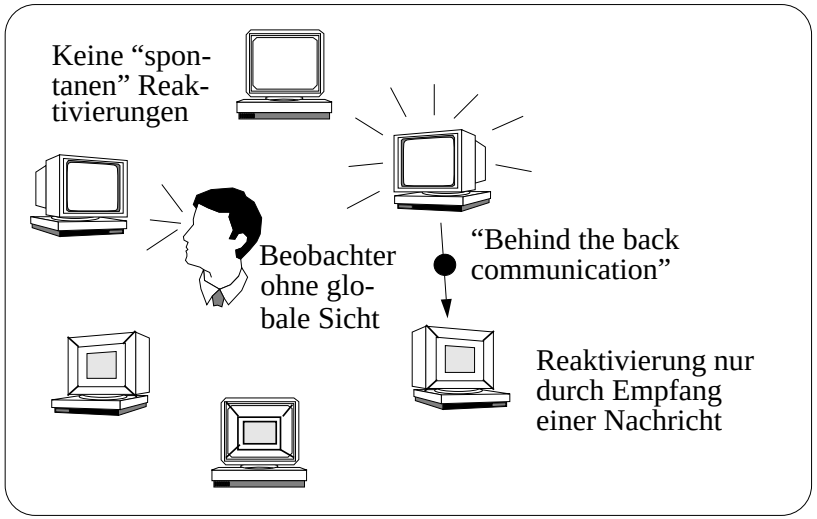
Das Problem der Terminierung

- Bsp: Zahlenrätsel (oder ggT) auf einem Workstation-Cluster



- pro Spalte (bzw. "Philosoph") jeweils ein Display
- dort jeweiligen Zustand und neue Wertemengen anzeigen

aktiv oder *passiv*



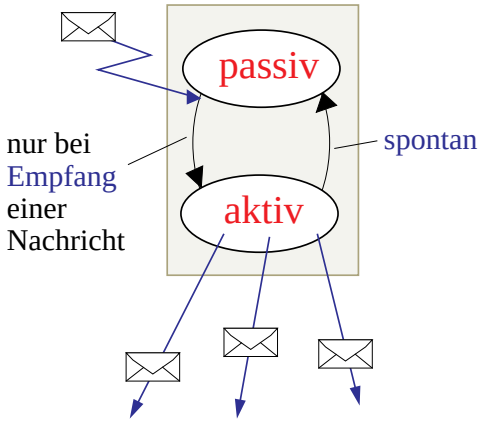
- *Erkennung* der verteilten Terminierung?

Alle passiv und keine Nachricht unterwegs

Verteilte Terminierung: Problemdefinition

Nachrichtengesteuertes Modell einer vert. Berechnung:

- Prozesse sind *aktiv* oder *passiv*
- Nur aktive Prozesse versenden Nachrichten
- Prozeß kann "spontan" passiv werden
- Prozeß wird durch ankommende Nachricht reaktiviert



Problem:

Feststellen, ob (zu einem Zeitpunkt)
- alle Prozesse passiv sind
- keine Nachricht unterwegs ist

- "Globales Prädikat"
- "Stabiler Zustand"

Die Aktionen der Basisberechnung im nachrichtengesteuerten Modell

Prozeß p:

"guard": Prädikat über dem lokalen Zustand

S_p: {Zustand = *aktiv*}
send message <M> **to** ...

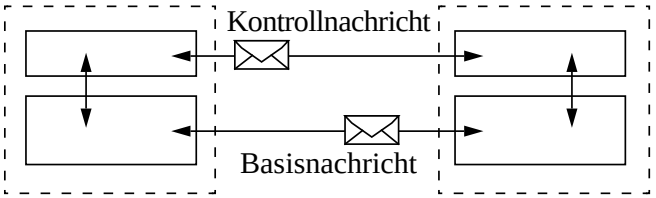
R_p: {Eine Nachricht ist angekommen}
receive <M> ; Zustand := *aktiv*

I_p: {Zustand = aktiv}
Zustand := *passiv*

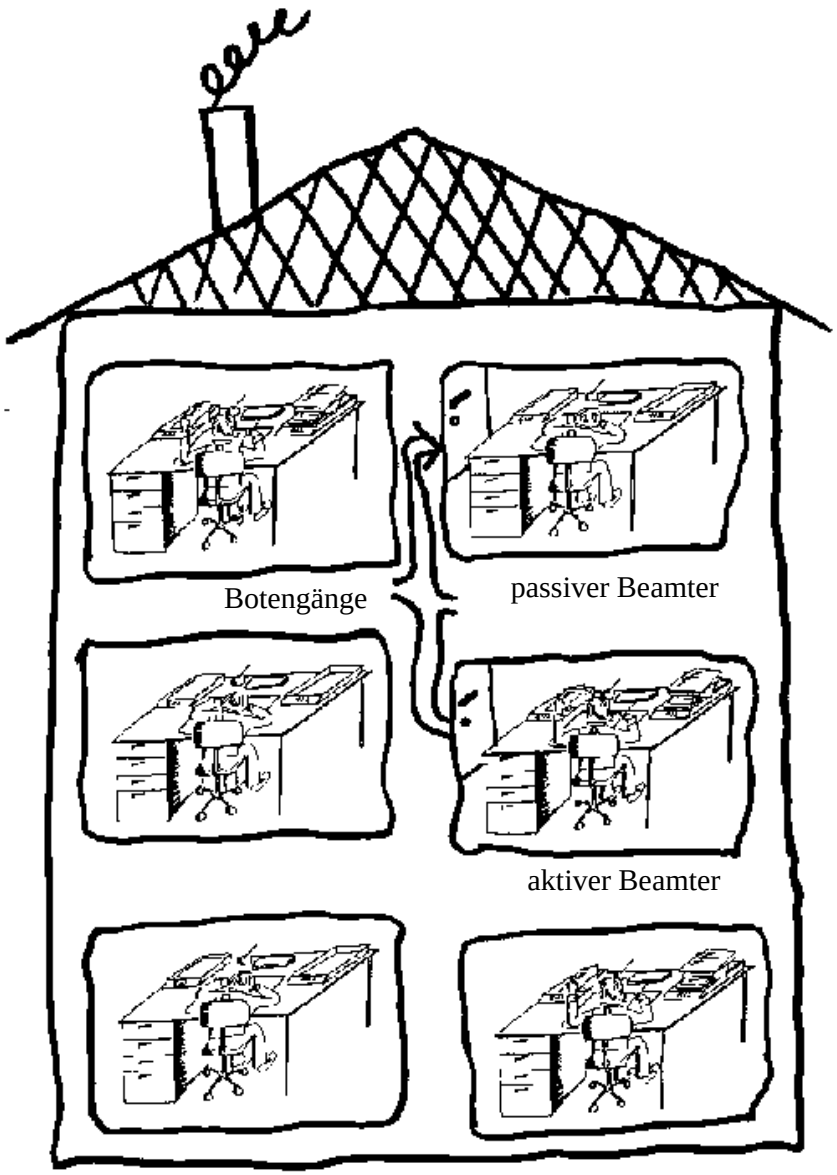
Typischerweise werden die Aktionen als "atomar" betrachtet

Abstraktes Verhalten einer verteilten Berechnung hinsichtlich Terminierung (ggf. existieren weitere Aktionen)

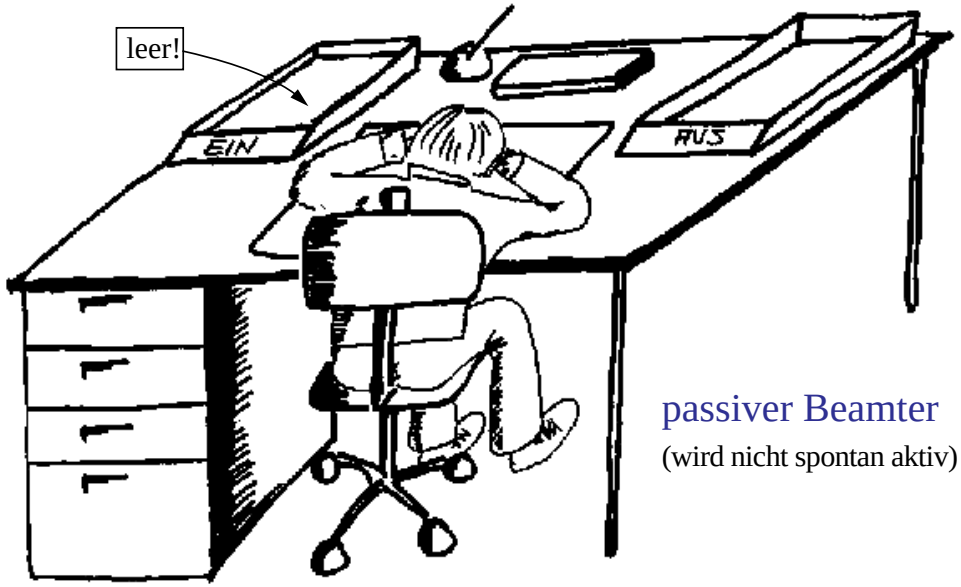
- Durch einen "überlagerten" Kontrollalgorithmus werden weitere Aktionen hinzugefügt
- "Anreichern" der Semantik der Basisberechnung für Zwecke des Kontrollalgorithmus
  - z.B. Verändern spezifischer (lokaler) Variablen
- Überlagerter Algorithmus soll Basisberechnung nicht stören
  - darf aber die Variablen, die der lokalen Kommunikation mit dem Basisalgorithmus dienen, lesen und schreiben



Eine typische Behörde



aktiver Beamter



passiver Beamter
(wird nicht spontan aktiv)

Die Funktionsweise der Behörde

(1) Publikumsverkehr nur bis 12 Uhr
(2) Schließt erst, wenn alle Vorgänge bearbeitet
(3) Vorgänge werden von Beamten erledigt
(4) Die Bearbeitung eines Vorganges *kann* neue Vorgänge für andere Beamte auslösen
(5) Aktenaustausch per (bel. langsame) Boten
(6) Keiner hat den Gesamtüberblick
(7) Beamte sind *aktiv* oder *passiv*
(8) Ein Beamter wird nicht spontan aktiv

Terminiert, wenn alle passiv und nichts "unterwegs"

Das ist ein stabiler Zustand!

Variante: Beamter läßt sich während der Arbeit nicht stören (anklopfen/warten auf "herein") -->

! → Beamte *scheinen* immer passiv (Atommodell)

Speedup ist durch die Maximalzahl gleichzeitig aktiver Beamten begrenzt.

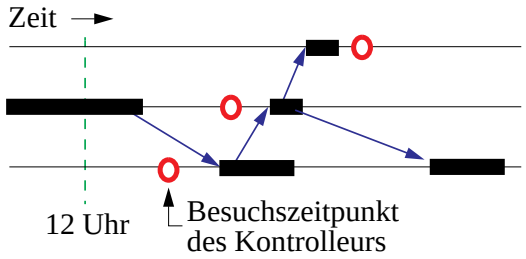
Dieser ist oft erstaunlich niedrig...

Das schiefe Bild des Kontrolleurs

Kontrolleur wandert durch die Behörde, um die Terminierung feststellen zu können.

Problem: Wie stellt der Kontrolleur fest, ob der stabile Terminierungszustand eingetreten ist?

Die *Illusion* Kontrolleurs:

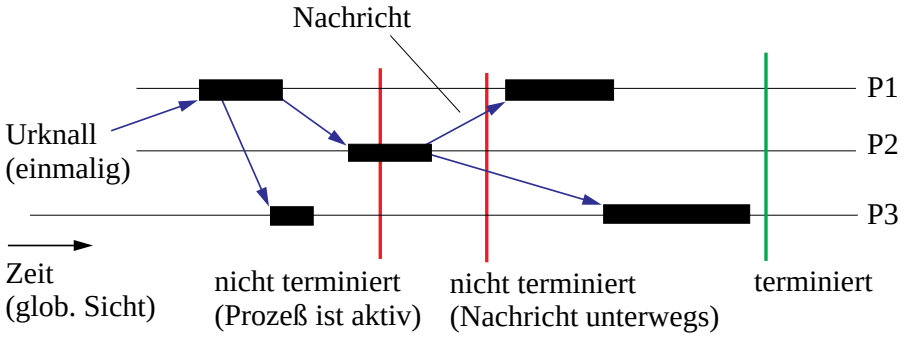


"behind the back communication"

- Alle Beamten stets passiv
- Σ Nachrichten versendet = Σ Nachrichten empfangen

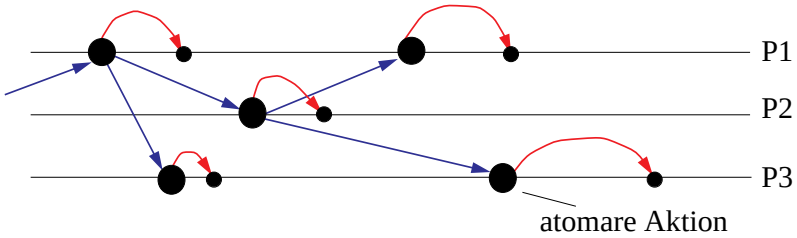
Kontrolleur hat ein *schiefes Bild!*

Zeitdiagramme und Atommodell



Idee: Dauer der Aktivitätsphasen "gegen Null" gehen lassen

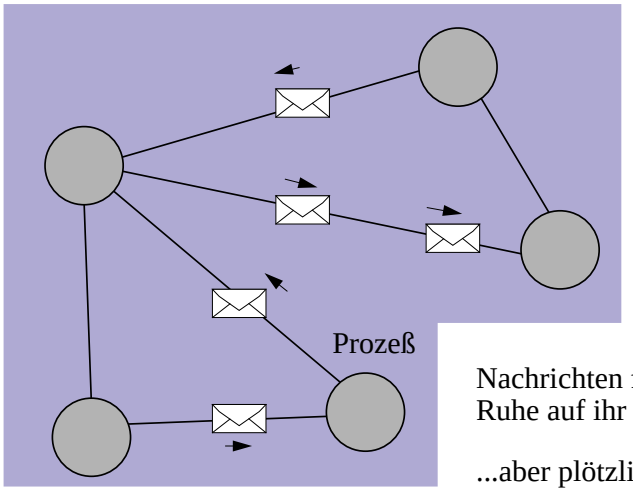
Modellierung: Prozeß sendet (virtuelle) Nachricht an sich selbst, sobald er aktiv wird; ist "unterwegs", solange er aktiv ist



Terminiert (Atommodell) <==>
Keine (echte oder virtuelle) Nachricht unterwegs

Zur Lösung des Terminierungsproblems also feststellen, *ob* noch Nachrichten unterwegs sind

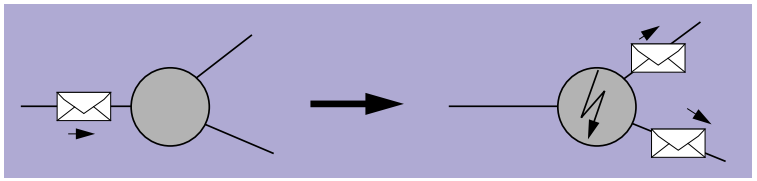
Globale Sicht ’atomarer’ Berechnungen



idealisierter Beobachter

Nachrichten fließen in aller Ruhe auf ihr Ziel zu...

...aber plötzlich "explodiert" ein Prozeß, wenn er von einer Nachricht getroffen wird!



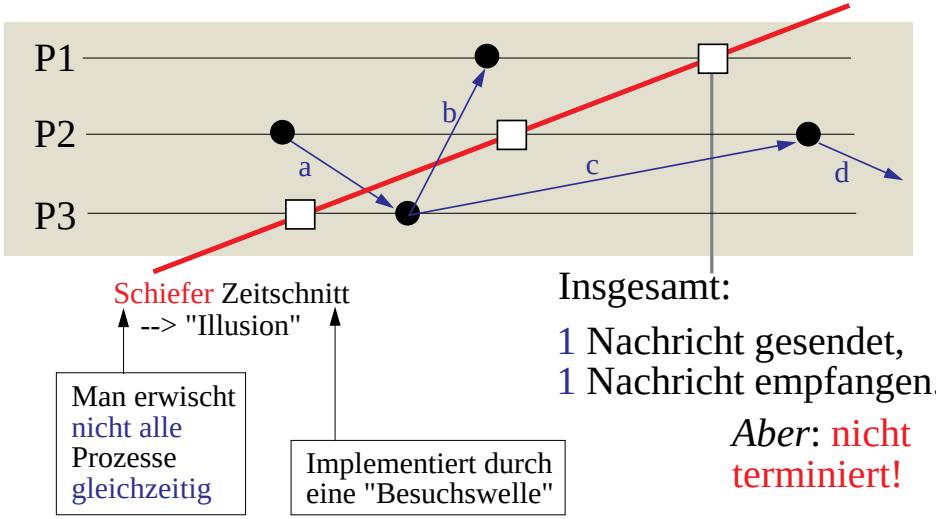
Terminiert, wenn in der globalen Sicht kein ✉ existiert

- Statt im "passiv/aktiv-Modell" genügt es, im Atommodell die Terminierungserkennung zu lösen (wieso?)
- Wie sehen die Aktionen der Basisberechnung in diesem Modell aus?

Verteilte Terminierung: Lösungen durch Zählen von Nachrichten?

- Genügt das (verteilte) Zählen von gesendeten und empfangenen Nachrichten?

- Einfaches Zählen genügt nicht, Gegenbeispiel:



Insgesamt:

1 Nachricht gesendet,
1 Nachricht empfangen.

Aber: nicht terminiert!

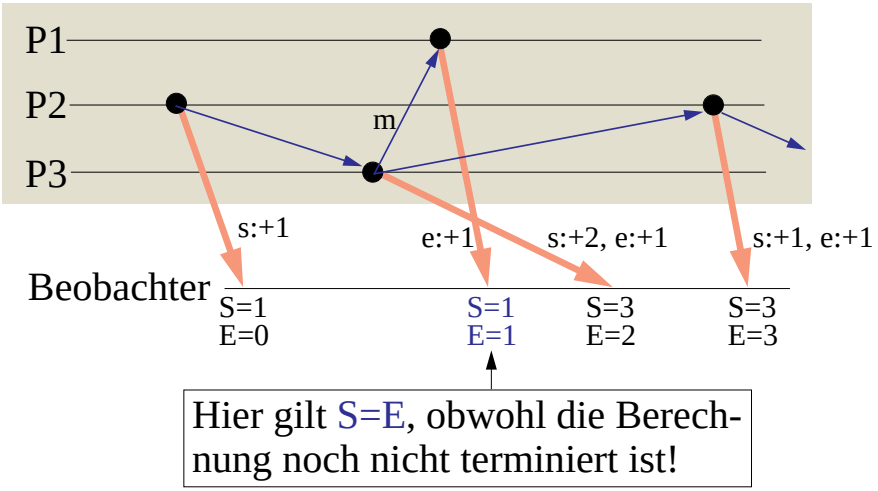
Ursache (informell):

- Nachricht aus der "Zukunft"
  - kompensiert die Zähler
- Inkonsistenter Schnitt
  - ist nicht äquivalent zu einem senkrechten Schnitt

Lösung durch "Ursachenvermeidung"? Ideen vielleicht:

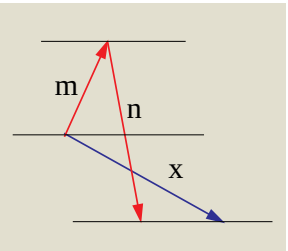
- Nachrichten aus der Zukunft *vermeiden* oder zumindest *erkennen*?
- Senkrechten Schnitt simulieren durch *Einfrieren* der Prozesse?

Beobachter über gesendete und empfangene Nachrichten informieren?



- Gleiches Szenario wie eben: Beobachter erfährt, daß m *gesendet* wurde, aber nicht, daß m *empfangen* wurde!

- Man beachte auch, daß hier eine Nachricht (x) in indirekter Weise (via m und n) "überholt" wurde!



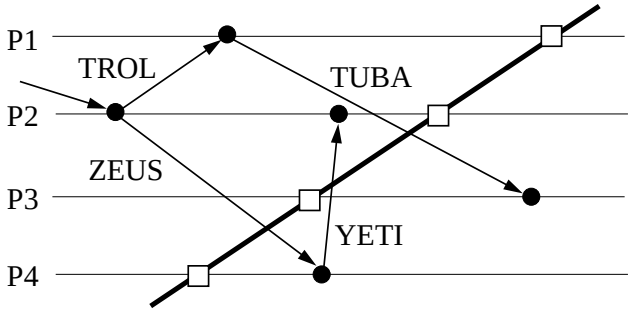
Vermutung: Wenn Informationsnachrichten nicht überholt werden können, dann kann das Phänomen eines "schiefen Bildes" nicht auftreten!

- worauf gründet sich die Vermutung?
- kann man solchermaßen korrekte ("kausaltreue") Beobachtungen erzwingen?

Nachrichten eindeutig benennen?

Prinzip: Jede Nachricht bekommt einen (global) eindeutigen Namen:

- TROL, ZEUS, TUBA, YETI,... (?)
- Nachricht kennt ihren Namen
- Sender weiß, welche Nachrichten gesendet wurden
- Empfänger weiß, welche Nachrichten empfangen wurden



ZEUS und TUBA müssen Schnitt von links nach rechts überquert haben

- Welle akkumuliert Namen der gesendeten und Namen der empfangenen Nachrichten
- Wenn eine gesendete nicht empfangen wurde, muß sie den Schnitt überquert haben ==> Terminierung nicht melden
- Terminiert, wenn alle "bekanntermaßen gesendeten" auch empfangen wurden? (Beweis?)
  - Tip: Wenn keine Nachricht den Schnitt (von links nach rechts??) überquert, ist der Lebensfaden des Systems gerissen; rechts des Schnittes kann dann keine Aktivität mehr entfacht werden (wieso?)

Frage: Wie geht das ganze überhaupt initial los?

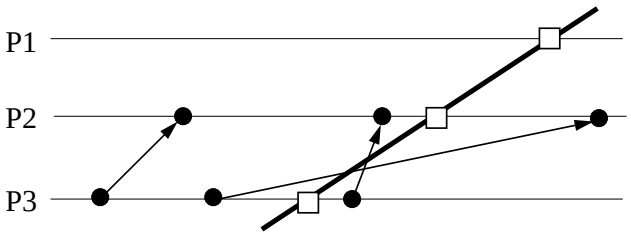
Eindeutige Nachrichtennamen?

- Sender könnte Nachrichten fortlaufend numerieren und seinen eigenen eindeutigen Namen hinzufügen
  - läßt sich einfacher verwalten als beliebige (global eindeutige) Namen
- Es genügt wohl auch eine fortlaufende Numerierung pro Sender-Empfänger-Beziehung ("Kanal")
  - z.B. 17.4.239 ("239. Nachricht von Knoten 17 an Knoten 4")
  - Verwaltungsaufwand ist recht hoch (bei FIFO benötigt man keine Mengen, es genügen $O(n^2)$ Zähler)

Date: Tue, 10 Dec 91 17:45:05 +0100
From: Bernadette CHARRON-BOST <bcb@litp.ibp.fr>
...
Voila comme promis la liste des noms des RER...
TROL, TSIN, TUBA, TJAO
UPAC, UTAH, UGON, UJIR, UXAM,
ZEBU, ZEUS, ZEMA, ZARA, ZITA, ZHAN, ZWIC
XILO, XERU, XUAN,
YVAN,
BROU, BRIO, BLEU, BUBU, BYLL, BOUL
YETI, YACK,
ELSA,
ANNE, AMIE, AOUT,
BALI,
DUFY, DEBA
EOLE,
...
En tous les cas, je ne sais pas si ce systeme est vraiment une trouvaille car hier on a annonce que "le prochain train n'etait pas un ELSA mais un YETI" ce qui a laisse les voyageurs (comme moi) dans une certaine perplexite! Mais peut-etre certains y trouvent une certaine poesie?

Genügt pauschales Zählen pro Kanal?

(anstatt Nachrichten pro Kanal individuell zu betrachten)



- Welle stellt folgendes fest:

  Auf Kanal P3P2 sind 2 Nachrichten gesendet worden und 2 Nachrichten empfangen worden

  ==> Keine Nachricht auf diesem Kanal überquert den Schnitt... ??

falsch!

Korrekt bei FIFO-Kanälen?

auch bei non-FIFO!

Behauptung:

Wenn entlang eines Schnittes *alle* Kanalzähler bzgl. send/receive ausgeglichen sind, dann überquert keine Nachricht den Schnitt.

- Wieso? (intuitives Argument?)
- Beweis?

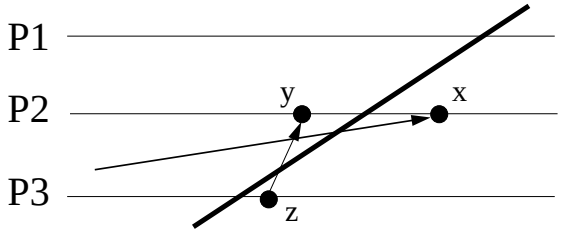
wieso ist obige Skizze kein Gegenbeispiel?

Beweisskizze für das Kanalzählerkriterium

Behauptung: Wenn entlang eines Schnittes pro Kanal gleich viele Nachrichten gesendet wie empfangen wurden, dann ist die Berechnung terminiert

Betrachte frühestes Ereignis (x) *nach* dem Schnitt:

Bei globaler (von links nach rechts fließender) *Zeit* in der Abb. ist dies klar; wenn man ohne solche graphischen Veranschaulichungen auskommen will, muß man statt dessen die *Kausalrelation* bemühen!

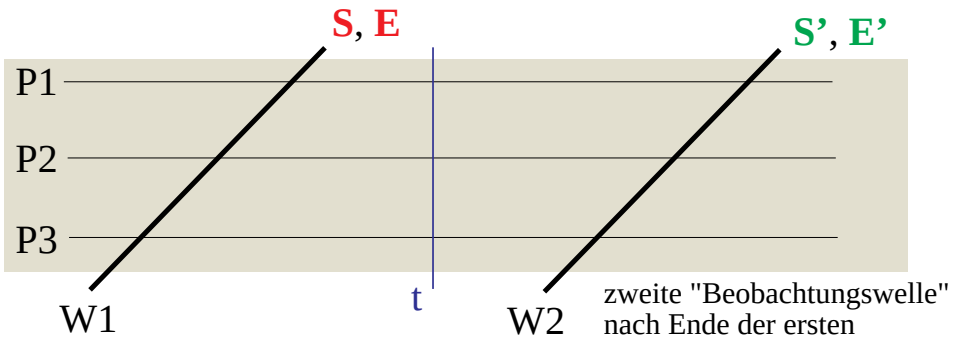


Wir zeigen durch *Widerspruch*: es gibt kein frühestes Ereignis nach dem Schnitt ==> Terminierung

- Dieses ist ein Empfangsereignis mit zugehörigem Sendeereignis *links* des Schnittes
- Zugehöriger Kanalzähler kann nicht getäuscht werden, da für eine *Kompensationsnachricht* gilt: Empfangen (y) vor dem Schnitt, gesendet (z) danach
- Sendeereignis der Kompensationsnachricht wäre *früheres* Ereignis *nach* dem Schnitt ==> *Widerspruch*
  - Senden ist immer früher als das Empfangen einer Nachricht!
  - z früher als y, y früher als x ==> z früher als x

Zählen pro Kanal ist aber etwas aufwendig ($O(n^2)$ Zähler); geht es nicht doch mit "ganz pauschalen" Zählern?

Das Doppelzählverfahren



d.h. keine Nachricht unterwegs

Behauptung: S=E=S'=E' ==> terminiert

Beweis (Skizze; läßt sich auch formalisieren):

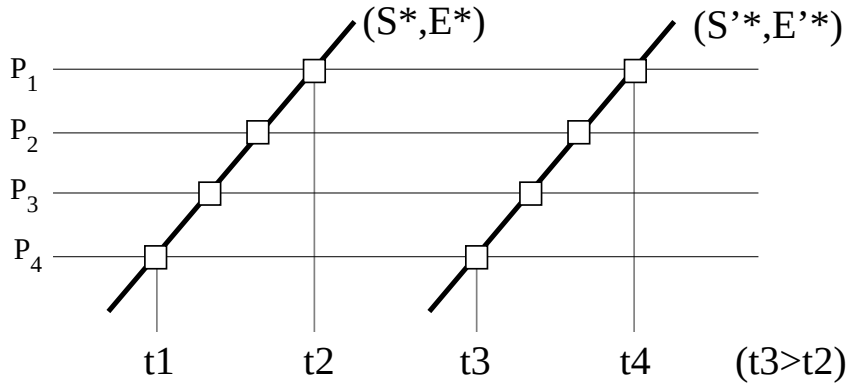
S=S' ==> Keine Nachricht zwischen W1, W2 gesendet.
E=E' ==> ——— "" ——— empfangen.
==> Werte bei t = Werte von W1.

Also: S=E ==> zum globalen Zeitpunkt t gilt:

Anzahl gesendeter = Anzahl empfangener Nachrichten
==> zum Zeitpunkt t ist keine Nachricht unterwegs
==> zum Zeitpunkt t terminiert
==> Berechnung war nach W1 terminiert □

Es gelingt also, für einen bestimmten Zeitpunkt eine kausaltreue Beobachtung (als senkrechten Schnitt) im Nachhinein zu rekonstruieren!

Formaler Beweis des Verfahrens



Notation:

- Lokaler Send-Zähler des Prozesses P_i zur Zeit t: $s_i(t)$
- Lokaler Empf.-Zähler des Prozesses P_i zur Zeit t: $e_i(t)$
- $S(t) := \sum s_i(t)$ $E(t) := \sum e_i(t)$

Lemmata:

(1) $t \le t' \Rightarrow s_i(t) \le s_i(t'), e_i(t) \le e_i(t')$ [Def.]
(2) $t \le t' \Rightarrow S(t) \le S(t'), E(t) \le E(t')$ [Def., (1)]
(3) $E^* \le E(t2)$ [(1), e_i wird "eingesammelt" vor t2]
(4) $S'^* \ge S(t3)$ [(1), s_i wird "eingesammelt" vor t3]
(5) Für alle t: $E(t) \le S(t)$ [Induktion über die atomaren Aktionen]

Beweis:

$E^* = S'^*$ ==> $E(t2) \ge S(t3)$ [(3), (4)]
==> $E(t2) \ge S(t2)$ [(2)]
==> $E(t2) = S(t2)$ [(5)]
==> terminiert zum Zeitpunkt t2 □

Zwei Zähler genügen!

Anzahl der "in-transit" Nachrichten bei t2 = 0

Eigenschaften des Doppelzählverfahrens

- Vergleich des Empfangszählers der ersten Welle mit mit dem Sendezähler der zweiten Welle (d.h. E = S') ist ein hinreichendes Kriterium

- Falls Terminierung nicht entdeckt wird: Benutze zweite Welle der vorherigen Runde als erste Welle der neuen Runde

---

"ablaufinvariant"

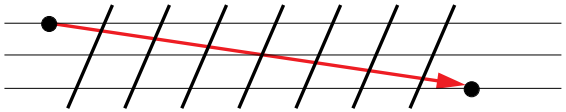
- Algorithmus ist *reentrant*: Lokaler Zustand des Prozesses wird durch Kontrollalgorithmus nicht geändert

  ==> jeder Prozeß kann unabhängig eine eigene / neue Kontrollrunde starten ("symmetrischer Algorithmus")

- Viele Varianten (zugrundeliegender Wellenalgorithmus)

- Anzahl der Kontrollrunden ist a priori nicht durch die Anzahl der Basisnachrichten begrenzt

  - es kann eine ganz langsame Basisnachricht geben, während der beliebig viele Kontrollrunden gestartet werden können



  - allerdings ist folgende Variante denkbar: Ein Prozeß, der eine Basisnachricht erhält ohne eine neue auszusenden, startet eine Doppelrunde

Safety- und liveness-Eigenschaften

Safety: Something bad will never happen...
(Typisch: "für alle eingenommenen Zustände gilt...")

- Bsp.: Nie mehr als 1 Prozeß im kritischen Abschnitt
- Bsp.: Variable x wird nie negativ
- Bsp.: Invariante wird nicht verletzt

oft auch "progress" — *schließlich*

Liveness: Something good will eventually happen...
(Typisch: "es wird ein Zustand eingenommen, so daß...")

- Bsp.: Variable x wird schließlich positiv
- Bsp.: Programm terminiert
- Bsp.: erfolgte Terminierung wird schließlich auch gemeldet

Korrektheit: Algorithmus erfüllt Safety *und* Liveness

Beispiel verteilte Terminierung:

- aufgesetzter Kontrollalgorithmus
  - sagt "ja", wenn Basisberechnung terminiert ist
  - sagt "weiß nicht" sonst ("nein" geht nicht!)

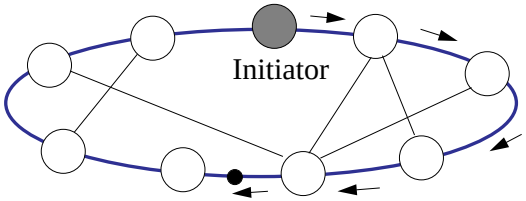
- safe, aber nicht live: meldet stets "weiß nicht"
- live, aber nicht safe: meldet stets "ja"

==> "Kunst": Algorithmus, der safe *und* live ist!
==> Es ist stets safety *und* liveness zu zeigen!

Kontrolltopologien

- Die für den Terminierungsalgorithmus benötigten "Wellen" können unterschiedlich realisiert werden:

1.) *Ring / Hamilton'scher Zyklus:*



Kontrollnachricht ("Token") besucht zyklisch zwei Mal alle Prozesse und akkumuliert dabei die Zählerstände.

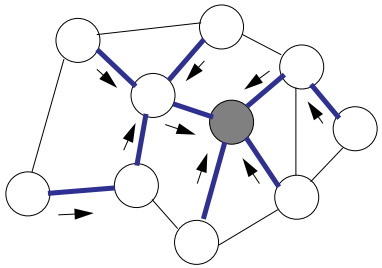
"Logischer" Ring genügt!

- In einem zusammenhängenden ungerichteten Graphen kann ein logischer Ring immer gefunden werden, indem man einen Spannbaum "umfährt".

- Zeit- und Nachrichtenkomplexität: O(n)

2.) *Spannbaum:*

- Vereinfachter Echo-Algorithmus: An den Blättern reflektierte Welle sammelt die Zählerstände in akkumulierender Weise ein.



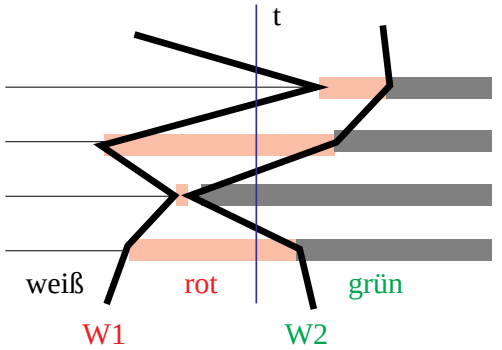
- Auch hier werden *zwei* solche zum Initiator hinfließende Wellen benötigt

- Bei nicht-entartetem Spannbaum: Viele Nachrichten parallel unterwegs ==> bessere Zeitkomplexität!

Echo-Algorithmus für die beiden Wellen des Doppelzählverfahrens?

- Anwendbar für beliebige zusammenhängende Topologien.

- Ausnutzen der beiden "Halbwellen" eines einzigen Laufs des Echo-Algorithmus für die beiden Kontrollrunden!



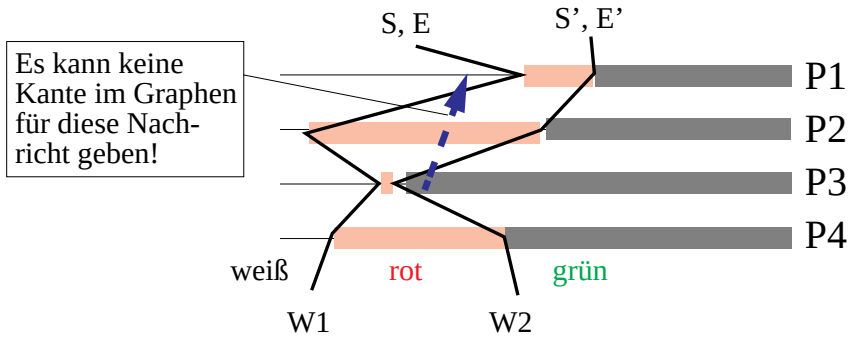
Der Echo-Algorithmus wird als *Transportdienst* zur Realisierung von *zwei* Wellen benutzt, wobei durch die Echo-Nachrichten sowohl die bei W1 lokal gemerkte Information als auch die bzgl. W2 relevante lokale Information an den Initiator übermittelt wird.

- Welle W1: "rot werden"; Welle W2: "grün werden"

- *Behauptung*: Das so realisierte Doppelzählverfahren ist korrekt.

- *Problem*: Formeller oder informaler Beweis lassen sich so nicht anwenden, da sich W1 und W2 *überlappen*!

Bemerkung:
Auf vorhandenen Spannbäumen kann man aber *nicht* einfach die vom Initiator ausgesendete Hinwelle und die reflektierte Rückwelle verwenden!



- "Trick": Es gibt keine nach W2 gesendete Nachricht, die vor W1 ankommt (grüne Knoten haben keine weißen Nachbarn!)

  - Wenn ein Knoten grün wird, sind seine Nachbarn bereits vorher rot geworden

  --> Täuschung der Zähler durch Kompensation mittels Nachrichten "rückwärts" über 2 Wellen ist unmöglich!

- *Beweisskizze*:

  1) Im roten Gebiet (d.h. zwischen W1 und W2) findet keine Aktivität statt, wenn das Terminierungskriterium S=E=S'=E' gilt:
  Dazu müßte eine vor W1 (im weißen Gebiet) ausgesendete Nachricht im roten Gebiet ankommen. Dann ist aber E' > E.

  2) Es kann Nachrichten geben, die beide Wellen "vorwärts" überqueren (d.h. im weißen Gebiet gesendet werden und im grünen ankommen). Solche Nachrichten werden auf S (und S') als gesendet registriert, jedoch weder auf E noch auf E' als empfangen registriert. Da eine Kompensation der Zähler S, E mittels Nachrichten "rückwärts" über 2 Wellen unmöglich ist (wie im Gegenbsp. zum einfachen Zählen), ist dann S > E.

  Also: Es gibt keine Nachricht, die W1 überquert; nach W1 findet daher keine Aktivität statt; das System ist nach W1 terminiert

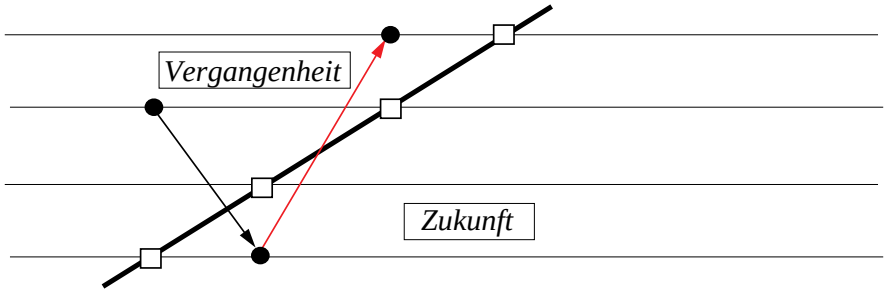
Zeitzonenverfahren

- Erkenntnis: Beim *einfachen Zählen* war eine *irreführende Kompensation* der Zähler durch Nachrichten aus der Zukunft möglich.

- Idee: Entlang des Schnittes (induziert durch Welle!) zählen, aber Nachricht aus der Zukunft *erkennen*.

- Nachricht aus der Zukunft --> Schnitt inkonsistent

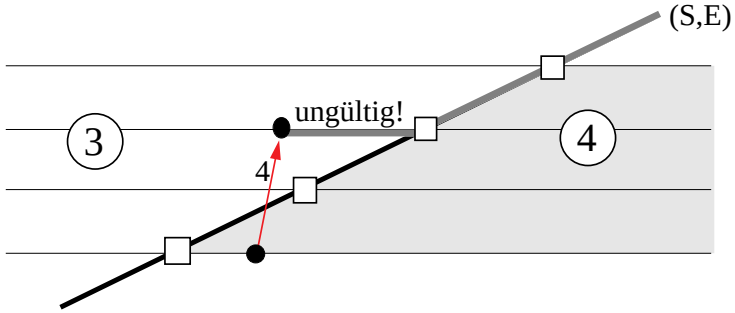
- Keine solche Nachricht --> Schnitt konsistent
  ("äquivalent" zu senkrechtem Schnitt)



- Zählergebnis verwerfen, wenn inkonsistent

- Sequenz von Wellen, bis Terminierung festgestellt.
  (Liveness ist klar: Schnitt nach Terminierung ist konsistent)

Prinzip: *Erkennen* inkonsistenter Schnitte



Lokale Nachrichtenzähler akkumulieren, aber:
Ergebnis *invalidieren*, wenn eine Nachricht aus der Zukunft empfangen wurde.

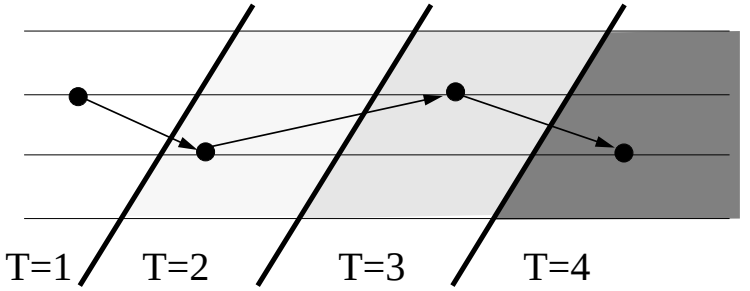
Implementierung durch:

- "Piggybacking" der Zeitzonennummer auf Nachrichten.
- Flag setzen, wenn eine Nachricht aus einer höheren Zeitzone empfangen wurde.
- Welle registriert Flag (und setzt es zurück) und erhöht die Zeitzone.

Terminiert, wenn Welle kein gesetztes Flag feststellt und die beiden akkumulierten Zähler übereinstimmen.

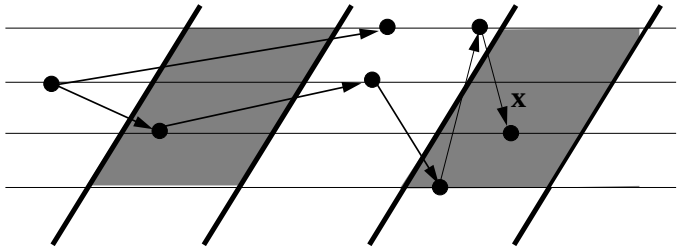
(Formaler Korrektheitsbeweis?)

- Wiederholte Ausführung des zugrundeliegenden Wellenalgorithmus:



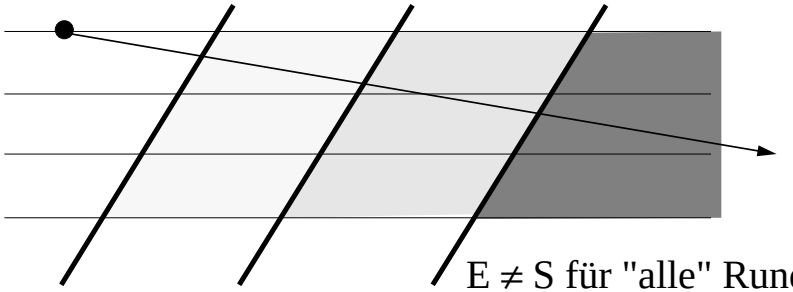
- Zeitzone bei jeder Runde inkrementieren

- "Zyklische" schwarz/weiß-Zeit genügt.
  Höhere Zeitzone --> "andere" Zeitzone.



- *Jede* Nachricht aus der Zukunft wird erkannt
  Nachricht aus der Zukunft trägt *andere* Farbe
  (Nachricht rückwärts über zwei oder mehr Wellen existiert nicht)
- Einige Nachrichten aus der Vergangenheit führen zu "Fehlalarmen" (vgl. Nachricht x)
  --> evtl. eine (einzige) zusätzliche Runde nötig

- Anzahl notwendiger Kontrollrunden? Unbeschränkt!



E ≠ S für "alle" Runden...

- Vergleich von Doppelzähl- und Zeitzonenverfahren?

  - Aufwand an Kontrollnachrichten und Speicher unwesentlich verschieden
  - Eingriff in die Basisnachrichten bei Zeitzonenverfahren nötig!
  - Zeitzonenverfahren ist nicht "reentrant"
    - lokaler Zustand wird verändert
    - dadurch nicht symmetrisch: Wellen mehrerer Initiatoren ("multi-source") können sich gegenseitig stören

==> Doppelzählverfahren scheint eleganter und einfacher / universeller einsetzbar.
(allerdings u.U. eine "Runde" mehr nötig)

- Denkübung: Könnte man nicht Nachrichten aus der Zukunft von vornherein *vermeiden*?

  - z.B. durch Einfrieren des Systems (dann in Ruhe zählen und ggf. in einer weiteren Runde wieder auftauen): ist das korrekt?
  - klappt das Vermeidungsprinzip auch ohne Einfrieren?