Kapitel 1

Verteilte Algorithmen

1.1 Verteilte Systeme

Unter einem *verteilten System* versteht man eine Ansammlung 'eigenständiger' Rechner, die Daten untereinander austauschen können.

Solche Systeme sind sinnvoll für:

- Informationsaustausch (etwa im WWW oder bei der Kontoverwaltung)
- Resource–Sharing (z. B.: Drucker etc.)
- Erhöhung der Zuverlässigkeit
- höhere Performance zum niedrigen Preis
- Vereinfachung des Softwaredesigns

Verteilte Systeme unterscheiden sich in Bezug auf

- Zuverlässigkeit der Datenübertragung
- Ausfallsicherheit einzelner Komponenten
- Kommunikationszeit
- Homogenität
- wechelseitiges Vertrauen

In einem verteilten System können die einzelnen Komponenten über sogenannte Wide–Area–Networks (WAN) oder Local–Area–Networks (LAN) miteinander kommunizieren.

Merkmale von WANs:

- geringe Zuverlässigkeit
- lange Kommunikationszeiten
- inhomogen
- wechselseitiges Mißtrauen
- Probleme:
  - Verläßlichkeit des Datenaustauschs
  - Wahl des Übertragungspfades (Routing)
  - Verstopfungskontrolle
  - Deadlock–Vorsorge
  - Sicherheit

Merkmale von LANs:

- hohe Zuverlässigkeit
- kurze Kommunikationszeiten
- wechselseitiges Vertrauen
- Probleme:
  - Broadcasting und Synchronisation
  - Leader–Election (Auswahl eines Prozessors aus einer Menge von Kandidaten)
  - Terminierungserkennung
  - wechselseitiger Ausschluß
  - Deadlock–Vermeidung/–Vorsorge

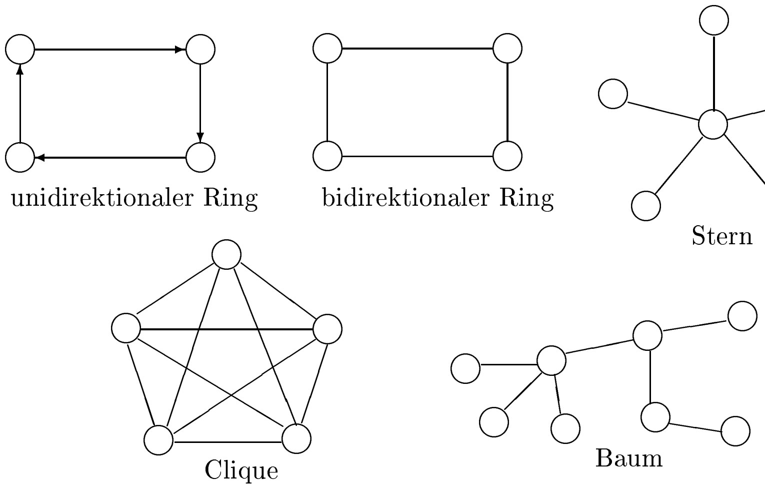
Die Schwierigkeiten bei der Realisierung solcher Systeme sind

- Mangel an Kenntniss des globalen Zustands des Systems.
- Mangel an globaler Zeit
- Nichtdeterminismus infolge unterschiedlicher Geschwindigkeit der Komponenten bzw. der Programmausführung auf der gleichen Komponente

1.2 Das Modell

Definition:
Ein *System* Σ besteht aus einer Menge von *Prozessen* P und einer Menge von *Kanälen* $E \subseteq P \times P$. Das Tupel (P, E) heißt *Topologie* von Σ. Wir verlangen, daß zwischen zwei Prozessen p und q ein Pfad von p nach q existiert.

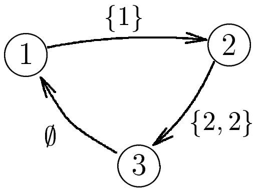
Beispiele für Topologien:



Bemerkung: Bei uns ändert sich die Topologie während des Programmablaufs **nicht** !

Jedem Kanal c wird eine Multimenge von *Nachrichten*, die bereits abgeschickt, aber noch **nicht** angekommen sind, zugeordnet.
Die Menge möglicher Nachrichten heißt M, die Funktion $\mu : E \longrightarrow M$ heißt *Nachrichtenbelegung*.

Beispiel: $M = \{1, 2\}$

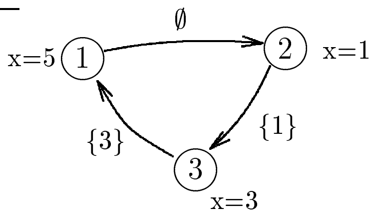


Definition:

- Jeder Prozeß $p \in P$ besitzt ein Programm t_p und befindet sich in einem Zustand s (Zustand = Programmpunkt + Variablenbelegung).
- S ist die Menge aller Zustände, $I \subseteq S$ die Menge aller Anfangszustände und $\sigma : P \longrightarrow S$ eine Belegung der Prozesse mit Zuständen.
  Das Paar $\gamma = (\sigma, \mu)$ heißt Konfiguration des Systems Σ.

  Eine *Anfangskonfiguration* (σ, μ) liegt vor, wenn für alle $p \in P, c \in E$ folgendes gilt:
  $\sigma(p) \in I(p) \wedge \mu(c) = \emptyset$.

Beispiel:



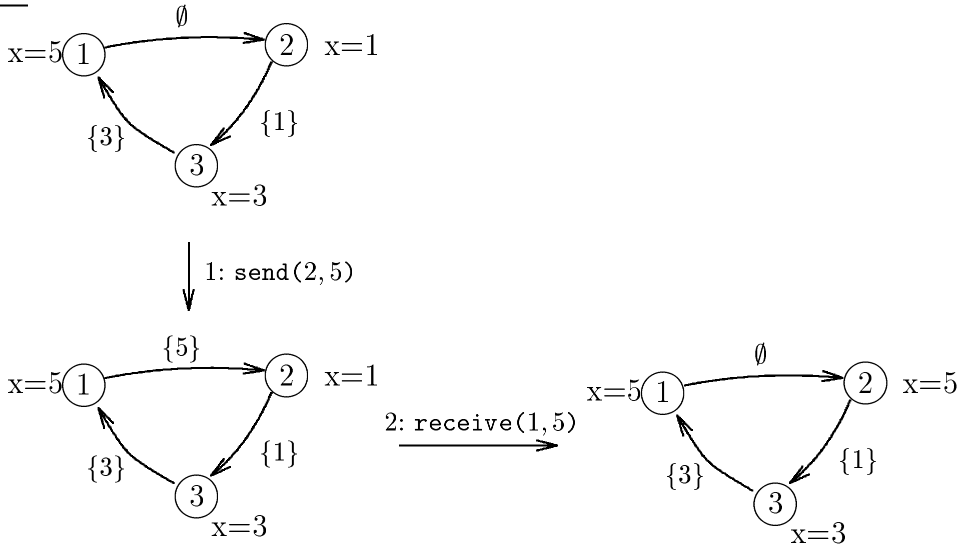
- Ein Berechnungsschritt eines Prozesses $p \in P$ (in einer Konfiguration γ von Σ) heißt *Ereignis* (*Event*).
  Ein *internes Ereignis* ändert nur den Zustand von p. Ein *externes Ereignis* erzeugt oder konsumiert Nachrichten.

  Externe Ereignisse für p sind:

`send`(q,m):	ändert den Zustand von p und schreibt m auf den Kanal (p,q)
`receive`(q,m):	ändert den Zustand von p und konsumiert m vom Kanal (q,p)

- Eine maximale Folge von Ereignissen, die in einer Anfangskonfiguration beginnt, heißt *Run* des Systems (solche Folgen müssen nicht unbedingt endlich sein).
  Für eine bestimmte Anfangskonfiguration können wir einen Run auch als maximale Folge von Konfigurationen auffassen.

Beispiel:



Achtung:
Σ ist nichtdeterministisch, weil in jedem Schritt **verschiedene** Prozesse agieren können und jeder Prozeß eventuell mehrere Ereignisse hat/zulässt. Folglich kann es für eine Anfangskonfiguration viele Runs geben.

Implizite Annahmen über das System:

1. Die Kanäle sind verläßlich.

2. Zwischen externen Schritten können beliebig (**aber endlich**) viele interne Zwischenschritte erfolgen.

3. Die Kanäle haben beliebige Kapazität.

4. Die Laufzeit der Nachrichten liegt bei Null.

5. Die Kommunikation ist asynchron; d. h. zwischen Senden und Empfangen kann beliebig viel Zeit vergehen. Die Nachrichten werden im Kanal gepuffert.

6. Nachrichten können sich überholen. Es sind keine Annahmen über die Reihenfolge von Nachrichten auf einem Kanal zulässig !

Programme t_p eines Prozesses p notieren wir in (Pidgin–)C mit den zusätzlichen Routinen send(destination,msg) und receive(source,msg).

Das „Raten", d.h. die nicht-deterministische Auswahl der ankommenden Nachricht bzw. der sendenden Quelle, modellieren wir mit Pattern-Matching wie folgt:

receive(1,&x);	bindet x an eine Nachricht von Prozeß 1
receive(&p,&x);	bindet zusätzlich p an PID des Senders
receive(-1,&x);	Wildcard (-1), wird verwendet, wenn der Sender nicht weiter interessiert
receive(1,<control,&x>);	empfängt Nachricht der Form <control,content> und bindet die Variable x an content

Zusätzlich erlauben wir den Test, ob eine Nachricht (eines bestimmten Typs) angekommen ist:

int probe(source,msg);	verändert **nicht** den Kanal blockiert **nicht**, falls keine Nachricht da ist Rückgabewert 1: Nachricht für source angekommen Rückgabewert 0: keine Nachricht für source vorhanden

Beispiel: Topologie: ein unidirektionaler Ring von Prozessen $0, \ldots, n-1$

Die Variable x sei initialisiert.

Programm für Prozeß i:

⋮

```
int succ, pred, t, x;
   succ = (i + 1) % n;
   pred = (n + (i - 1)) % n;
   for(t = 0; t < n; t++) {
     send(succ, x);
     receive(pred, &y);
     if (y > x)
       x = y;
     }
```

⋮

Fragen:

- Berechnet dieses Programm stets das Maximum ?
- Reichen vielleicht weniger Iterationen aus ? (→ siehe Übungsaufgabe)
- Gibt es wesentlich effizientere Programme? (**ja!**)

Kapitel 2

Methoden

2.1 Notation

Definition:

- Sei e ein Ereignis. Dann kann e geschrieben werden als $e = (p : a)$, d.h.: Der Prozeß p führt die Aktion a aus.
  Überführt e eine Konfiguration γ_1 in eine Konfiguration γ_2, dann schreiben wir: $\gamma_2 = e(\gamma_1)$
  (**Achtung:** e ist nun eine Funktion !)
- Eine Eigenschaft P (von Konfigurationen) heißt *Safety-Eigenschaft* von Σ genau dann, wenn sie in allen Konfigurationen aller Runs erfüllt ist.
- Eine Eigenschaft P heißt *Invariante* von Σ, falls P in allen Anfangskonfigurationen gilt und für alle Ereignisse e mit $\gamma_2 = e(\gamma_1)$ gilt: $P(\gamma_1) \Rightarrow P(\gamma_2)$

Satz:
Jede Invariante von Σ ist eine Safety-Eigenschaft. (Die Umkehrung gilt i.A. nicht).

Definition:

- Eine Eigenschaft P heißt *Lifeness-Eigenschaft*, falls jeder Run eine Konfiguration γ enthält mit $P(\gamma)$.
- Sei D eine Menge und $\sqsubseteq\ \subseteq D \times D$ eine zweistellige Relation über D.
  Eine *partielle Ordnung (partial ordered set (po))* ist ein Paar $(D, \sqsubseteq)$ für das gilt:
  1. $x \sqsubseteq x \quad \forall x \in D$ (Reflexivität)
  2. $x \sqsubseteq y$ und $y \sqsubseteq x \Rightarrow x = y \quad \forall x, y \in D$ (Antisymmetrie)
  3. $x \sqsubseteq y$ und $y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \forall x, y, z \in D$ (Transitivität)
- Eine *Wohlordnung* $(W, <)$ ist eine partielle Ordnung ohne unendlich absteigende Ketten.

Folgende partielle Ordnungen sind Wohlordnungen:

- $(I\!N, <)$
- $(I\!N, |)$ mit $a|b :\Leftrightarrow a$ teilt b

- Für Wohlordnungen W_1 und W_2 ist stets $(W_1 \times W_2, <)$ eine Wohlordnung, wobei $(w_1, w_2) < (w_1', w_2') :\Leftrightarrow w_1 < w_1'$ und $w_2 < w_2'$.

Definition:

- Sei *Conf* eine Menge von Konfigurationen des Systems Σ, W eine Wohlordnung.
  Eine *Rangfunktion* f für P und Σ ist eine Funktion $f : Conf \longrightarrow W$, so daß für alle Ereignisse e mit $\gamma_2 = e(\gamma_1)$ gilt: $P(\gamma_2)$ oder $f(\gamma_1) > f(\gamma_2)$

- Eine *terminale Konfiguration* ist eine Konfiguration, in der kein Ereignis möglich ist.

Satz:
Existiert eine Rangfunktion für P und Σ und gilt P in allen terminalen Konfigurationen, dann ist P eine Lifeness-Eigenschaft.

Beweis:
Sei f diese Rangfunktion.
Annahme: P ist keine Lifeness-Eigenschaft
$\Rightarrow$ es gibt einen Run π, in dem für jede Konfiguration γ in π gilt: $\neg P(\gamma)$
$\Rightarrow$ π enthält keine terminale Konfiguration
$\Rightarrow$ $\pi = \gamma_0\gamma_1\gamma_2\ldots$ ist unendlich
$\Rightarrow$ $f(\gamma_0) > f(\gamma_1) > f(\gamma_2) > \ldots$ ist unendlich
$\Rightarrow$ **Widerspruch** (es gibt in Wohlordnungen keine unendlich absteigenden Ketten!)

Definition:
P-echte Terminierung liegt genau dann vor, wenn P in allen terminalen Konfigurationen gilt.
P-deadlock liegt genau dann vor, wenn P in keiner terminalen Konfiguration gilt.

Die Ereignisse eines Runs π können unabhängig sein, d.h. $e_1(e_2(\gamma)) = e_2(e_1(\gamma))$.

Idee: Betrachte statt einer linearen Folge von Konfigurationen in π den sogenannten *Trace* von π, d.h. „π modulo Umordnung unabhängiger Ereignisse“.

Definition:
Ein *Trace* $[\pi]$ eines Runs $\pi = (p_1 : a_1)(p_2 : a_2)\ldots$, der in der Konfiguration γ_0 startet, besteht aus:
Knoten: $i \in \{1, 2, \ldots\}$ mit Label: $\lambda(i) = (p_i : a_i)$
Kanten: (i, j), sofern $i < j$ und die zugehörigen Ereignisse „direkt abhängig“sind, d.h.:

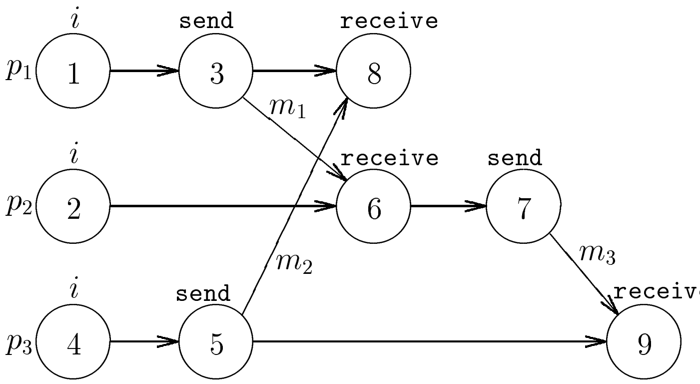
(1) Direkt aufeinander folgende Schritte eines Prozesses:
Für $\lambda(i) = (p_i : a_i)$ und $\lambda(j) = (p_j : a_j)$ mit $p_i = p_j$ gilt: $\forall i < k < j : p_k \neq p_i$

(2) Empfangen einer Nachricht von Prozeß p_i durch den Prozeß p_j:
Für $\lambda(i) = (p_i : a_i = \texttt{send}(p_j, m))$ und $\lambda(j) = (p_j : a_j = \texttt{receive}(p_i, m))$ gilt:
$$\begin{aligned} & \sharp\{k_1 < i \mid (p_{k_1} : a_{k_1}) = (p_i : \texttt{send}(p_j, m))\} \\ = \; & \sharp\{k_2 < j \mid (p_{k_2} : a_{k_2}) = (p_j : \texttt{receive}(p_i, m))\} \end{aligned}$$
D.h.: Genau alle vor i gesendeten Nachrichten m sind bereits vor j empfangen worden.

Traces, die bis auf Umnummerierung der Knoten isomorph sind, betrachten wir als gleich.

Beispiel:

1	$p_1 : i$
2	$p_2 : i$
3	p_1 :send(p_2, m_1)
4	$p_3 : i$
5	p_3 :send(p_1, m_2)
6	p_2 :receive(p_1, m_1)
7	p_2 :send(p_3, m_3)
8	p_1 :receive(p_3, m_2)
9	p_3 :receive(p_2, m_3)



wobei i eine internes Ereignis bezeichne.

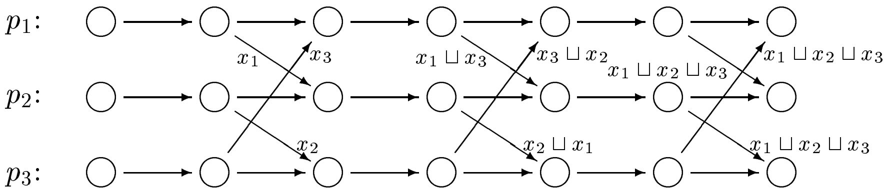
Fakten:
$[\pi]$ abstrahiert von der Anordnung der unabhängigen Ereignisse in π.

(1) Jeder Schnitt S durch $[\pi]$ repräsentiert eine erreichbare globale Konfiguration γ von Σ.

(2) Jede Permutation der Ereignisse links von S mit isomorphem Trace liefert genau γ ($\Rightarrow$ zwischen ihnen kann nicht unterschieden werden!)

Definition:
Seien i, j Konten eines Traces, e_1, e_2 Ereignisse.
Eine Ordnung heißt *kausale Ordnung*, wenn gilt: $(i, e_1) < (j, e_2) \iff \exists$ Pfad in $[\pi]$ von i nach j.

Beispiel:



Beobachtung:
Für obiges Beispiel gilt:

- Es gibt genau einen Trace!
- In der n–ten Runde senden offensichtlich alle Prozesse $x_1 \sqcup \cdots \sqcup x_n$ als Nachricht, d.h. bereits nach der $(n-1)$–ten Iteration verfügt jeder Prozeß über das Maximum aller x_i.

2.2 Logische Uhren

Ein Prozeß kann im wesentlichen nur von solchen Ereignissen „Kenntnis“ haben, von denen sein gegenwärtiges Ereignis kausal abhängt. Von Interesse sind nun Möglichkeiten für Prozesse, das Systemverhalten zu beobachten.
Beispiele solcher Selbstbeobachtungen sind logische Uhren.

Definition:
Eine *logische Uhr (LU)* ist eine Funktion Θ von Ereignisvorkommen nach einer (partiell) geordneten Menge, sodaß für Ereignisse e_1, e_2 gilt: $(v, e_1) < (w, e_2) \implies \Theta(v, e_1) < \Theta(w, e_2)$.

<u>LAMPORT's logische Uhr (LAMPORT 1978):</u>

$$\Theta(w,e) = \bigsqcup\{k \in I\!N \mid (v_1,e_1) < \cdots < (v_k,e_k) = (w,e)\}$$

Implementierung:

ursprünglicher Prozeß	Prozeß mit LAMPORT'S LU
	`int Θ=0; //Initialisierung`
`ch_state();`	`ch_state();` `Θ++;`
`send(p,m);`	`Θ++;` `send(p,<m,Θ>);`
`receive(p,m);`	`receive(p,<m,&Θ'>);` `Θ=(Θ ⊔ Θ')++;`

<u>Achtung:</u>

- Der 'neue' Algorithmus wird dem vorhandenen übergestülpt.
- Die Anzahl der Ereignisse erhöht sich.
- Die Nachrichtenlänge erhöht sich.

<u>Vektoruhr (MATTERN 1989):</u>

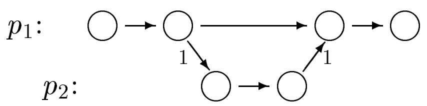
$$\Theta(v,e) : P \longrightarrow I\!N_0, \text{ wobei}$$
$$\Theta(v,e)(p) = \bigsqcup\{k \in I\!N_0 \mid (v_1,e_1) < \cdots < (v_k,e_k) \leq (v,e),\ e_i = (p : \_)\}$$

Die Vektor Uhr beobachtet das System viel präziser als LAMPORT's LU.
Hier gilt sogar: $(v,e_1) \leq (w,e_2) \Leftrightarrow \Theta(v,e_1) \leq \Theta(w,e_2)$.

<u>Implementierung:</u>
Analog zu LLU. Bei jeder Nachricht m muß jetzt aber das ganze P–Tupel mitgesandt werden. Das kann eventuell sehr teuer werden. Außerdem ist eine genaue Kenntnis von P erforderlich.

2.3 Komplexitätsmaße

1. Anzahl der Nachrichten
2. Anzahl der versendeten Bits
3. maximale logische Zeit eines Ereignisses (Nachteil: wichtet interne Ereignisse und Nachrichtenübertragung jeweils gleich)
4. Länge des längsten Pfades in $[\pi]$, wobei Kanten $\overset{\texttt{receive}}{\bigcirc} \longrightarrow \overset{\texttt{send}}{\bigcirc}$ Gewicht 1 und alle anderen Kanten Gewicht 0 erhalten.



$\rightsquigarrow$ liefert Länge 2

2.3.1 Zusätzliche Kriterien

- Topologie (Ring, Clique, beliebig)
- anfängliches Wissen über
  (a) eigene Identität (bei uns immer bekannt!),
  (b) die Nachbarn,
  (c) Einordnung in Topologie (z.B. Zuordnung von Kanälen zu Dimensionen in Hyperwürfel),
  (d) die Topologie (z.B. ihre Größe, Durchmesser).

Kapitel 3

Wellen–Algorithmen

Diese Algorithmen lösen wichtige Teilaufgaben komplizierterer Berechnungen.

Definition:
Ein *Wellen–Algorithmus* besitzt folgende Eigenschaften:

(T) Terminierung: Alle Traces (damit auch alle Runs) sind endlich.

(E) Entscheidung: Jeder Trace enthält mindestens ein *Entscheidungsereignis* (p : `decide`) (EE).

(A) Abhängigkeit: jedes EE hängt kausal von einem Ereignis in jedem Prozeß ab, d.h.:
Sei τ Trace, dann gilt:

$$\forall\ \tau : \forall (w,d) \in \tau : [(w,d) \text{ ist EE }] \implies [\forall q \in P : \exists (v,e) : (v,e) < (w,d),\ e = (q : \_)].$$

Sprechweise:

- *Starter* nennt man einen Prozeß, der die Welle auslöst.
- *Follower* heißen alle Prozesse, die in den Algorithmus „verwickelt “ werden.
- Gibt es nur einen Starter, so spricht man von einem *zentralisierten*, bei vielen Startern von einem *dezentralisierten* Algorithmus.

Im weiteren wollen wir Anwendungen von Wellen-Algorithmen untersuchen.

3.1 Broadcast mit Rückmeldung (PIF)

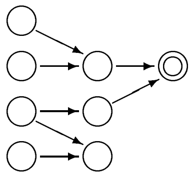
Arbeitsweise: Eine Teilmenge von Prozessen besitzt eine Nachricht m, die allen Prozessen mitgeteilt werden soll. Die Terminierung des Broadcasting soll bestimmten Prozessen mitgeteilt werden. Insgesamt soll der Trace terminieren.

Satz

1. Jeder PIF-Algorithmus ist ein Wellen–Algorithmus.
2. Jeder Wellen–Algorithmus kann für PIF benutzt werden.

Beweis:

1. (T) und (E) sind offensichtlich erfüllt. Bleibt zu zeigen, daß (A) gilt: Dazu nehmen wir an, die Notifikation hänge von keinem Ereignis in Prozeß p kausal ab.

   Dann kann ein Run gefunden werden, in dem alle Ereignisse von p nach Notifikation stattfinden. Das ist jedoch ein Widerspruch, da der Algorithmus PIF lösen soll.

2. Prozesse, die m besitzen, werden Starter des Wellen–Algorithmus A. An jede Nachricht n von A hängen wir m an. Notifikation findet statt, wenn A ein Entscheidungesereignis (EE) ausführt. □

3.2 Synchronisation (SYN)

Arbeitsweise: In jedem Prozeß q muß jede Aktion a_q ausgeführt werden und in einigen Prozessen p muß Aktion b_p ausgeführt werden, die zeitlich nach sämtlichen a_q kommt. Das Ganze soll terminieren.

Satz:

(1) Jeder SYN-Algorithmus ist ein Wellen–Algorithmus.

(2) Jeder Wellen–Algorithmus kann als SYN-Algorithmus benutzt werden.

Beweis:

(1) Die b_p sind die decide-Aktionen.

(2) Jeder Prozeß P muß erst a_q ausführen bevor er eine Nachricht des Wellen–Algorithmus sendet und darf erst b_p ausführen nachdem er `decide` ausgeführt hat.

3.3 Infimumsberechnung (INF)

Definition:
Sei $(D, \sqsubseteq)$ eine partielle Ordnung.
$x \in D$ heißt *untere Schranke* für eine Menge $X \subseteq D$, falls gilt: $\forall x' \in X : x \sqsubseteq x'$.
Eine untere Schranke x heißt *größte untere Schranke* von X, falls für jede untere Schranke $\tilde{x}$ gilt: $\tilde{x} \sqsubseteq x$.
Wir schreiben dann $x = \bigsqcap X$ und nennen x *Infimum* von X. Für $\bigsqcap\{a, b\}$ schreiben wir auch $a \sqcap b$. $\sqcap$ ist kommutativ, assoziativ und idempotent.
Falls jede Menge $X \subseteq D$ ein Infimum besitzt, heißt $(D, \sqsubseteq)$ *vollständiger Verband (complete lattice (cl))*.

INF-Problem:
Jeder Prozeß $p \in P$ besitzt anfänglich einen Wert $x_p \in D$, wobei D ein nichttrivialer vollständiger

Verband ist. Einige Prozesse besitzen eine Variable *out*. Nach endlich vielen Schritten schreiben diese Prozesse dann den Wert $\bigsqcap\{x_p \mid p \in P\}$ nach *out*. Insgesammt terminiert der Algorithmus.

Satz:

(1) Jeder INF-Algorithmus ist ein Wellen–Algorithmus.

(2) Jeder Wellen–Algorithmus kann das Infimum berechnen.

Beweis:

(1) (T): $\surd$
(E): $\surd$
(A): Sei A ein INF–Algorithmus.
Annnahme: Kein Ereignis eines Prozesses q beeinflußt die Variable *out* eines Prozesses p.
$\Longrightarrow$ Ersetze in der Anfangskonfiguration x_q durch $\bar{x}$, so daß $x = \bigsqcap\{x_{p'} \mid p' \in P\} \not\sqsubseteq \bar{x}$.
$\Longrightarrow$ Berechnug der Variablen *out* (von Prozeß p) ändert sich nicht.
$\Longrightarrow$ Widerspruch zur Voraussetzung A INF–Algorithmus

(2) Sei ein Wellen-Algorithmus A gegeben. Jeder Prozeß q erhält zusätzliche Variablen vom Typ D: $D\ \ x = x_q$;
$D\ \ y$;
Weiterhin ersetzen wir folgende Anweisungen:

vorher	nachher
`send`(p, m)`;`	`send`$(p, < m, x >)$`;`
`receive`(p, m)`;`	`receive`$(p, < m, \&y >)$`;` $x = x \sqcap y$;
`ch_state();`	`ch_state();`
`decide;`	`decide;` $out = x$;

Korrektheit:
"$\Rightarrow$" :
Sei $x[v, (p : a)]$ der Wert von x in Prozeß p nach ausführen des Ereignisses a am Knoten v.
Dann gilt:

(a) $x[v, (p : a)] \sqsubseteq x_p$

(b) $(v, e_1) \leq (w, e_2) \Rightarrow x[v, e_1] \sqsupseteq x[w, e_2]$

(c) Sei $x = \bigsqcap\{x_{p'} \mid p' \in P\}$. Dann gilt: $x \sqsubseteq x[v, (p : a)]$.

Nach (c) gilt dann: $x \sqsubseteq out$

"$\Leftarrow$" :
Betrachte das Entscheidungsereignis: $(w, (p : \texttt{decide}))$.
Da A Wellenalgorithmus folgt: $\forall p' \in P : \exists (v, (p' : a)) \leq (w, (p : \texttt{decide}))$.

Dann gilt: $x[w, (p : \texttt{decide})] \overset{(b)}{\sqsubseteq} x[v, (p' : a)] \overset{(a)}{\sqsubseteq} x_{p'}$
$\Longrightarrow$ Es gilt stets $out \sqsubseteq x_{p'}$ für alle p'.
$\Longrightarrow$ Der Wert für *out* ist untere Schranke von $\{x_{p'} \mid p' \in P\}$.
$\Longrightarrow$ $out \sqsubseteq x$ $\square$

3.4 Grundlegende Eigenschaften

(1) Für jedes Ereignis (w, e) gibt es einen Starter $p \in P$ und ein Ereignis $(v, (p : \_)) \leq (w, e)$ (der Starter löst also e aus).

(2) Sei τ eine Welle mit Starter p, $\Sigma = (P, E)$. Sei $T \subseteq E$ die Menge der Kanäle (q_1, q_2) so, daß das erste Ereignis (v, e) mit $e = (q_1 : a)$ dasjenige war mit $a =$ `receive(`$q_2, \_$`)`.
Dann ist (P, T) ein spannender Baum der Topologie Σ.
Insbesondere folgt, daß jeder Wellen–Algorithmus mindestens $|P| - 1$ Nachrichten benötigt (wozu benötigt ?).

3.5 Beispiele für Wellen–Algorithmen

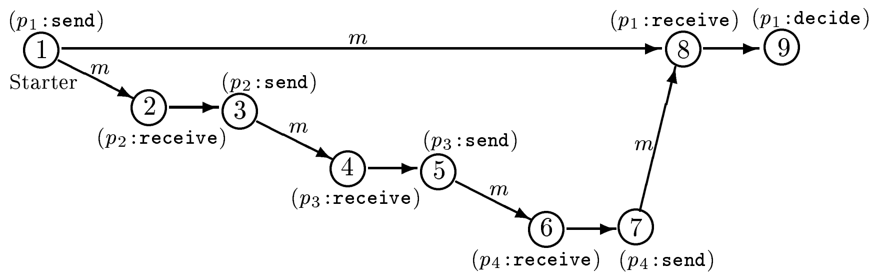
3.5.1 Zentralisierter unidirektionaler Ring–Algorithmus

Im Algorithmus ist `starter` eine boolesche Variable, die mitteilt, ob der Prozeß der Starter der Welle ist.
Algorithmus:

```
SetOfPid succ, pred;   /* Nachfolger–/Vorgängerprozesse */
int token;

  if (starter) {
    send(succ,token);
    receive(pred,&token);
    decide;
    }
  else {
    receive(pred,&token);
    send(succ,token);
    }
```

Für einen zentralisierten unidirektionalen Ring mit vier Prozessen $p_1, \ldots, p_4$ (p_1 ist Starter) erhält man als einzigen Trace:



Der Algorithmus terminiert nach $|P|$ Nachrichten, ist zentralisiert für einen unidirektionalen Ring und hat die Nachrichtenkomplexität $|P|$.

3.5.2 Der Echo-Algorithmus (Segal 83)

Der Echo–Algorithmus ist zentralisiert für beliebige ungerichtete Graphen $G = (V, E)$.

Algorithmus:

```
SetOfPid succ, pred;   /* Nachfolger–/Vorgängerprozesse */
int      token;
int      rec = 0;  /* Anzahl der erhaltenen Nachrichten */
Pid      father, p;

  if(starter) { /* Starterprozess */
    forall(p ∈ neighbours) /* neighbours: Menge der Nachbarprozesse */
      send(p,token);
    while(rec < |neighbours|){
      receive(-1,&token);
      rec++;
      }
    decide;
    }
  else { /* Follower–Prozess */
    receive(&father,&token);
    rec++;
    forall(p ∈ neighbours \ {father})
      send(p,token);
    while(rec < |neighbours|) {
      receive(-1,&token);
      rec++;
      }
    send(father,token);
    }
```

Satz: Der Echo–Algorithmus ist ein Wellen–Algorithmus mit einem Starter, der auch entscheidet.

Beweis von (T):
Maximal wird eine Nachricht durch jeden Kanal gesendet, d. h. die Nachrichtenkomplexität ist $\leq |E|$.

Zum Beweis von (E) und (A) betrachten wir terminale Konfiguration γ eines Traces τ.
Sei $T = (P, E_T)$ ein ger. Teilgraph von G mit $E_T = \{(q_1, q_2) \mid$ für q_1 gilt: $(\texttt{father == q}_2)\ \}$
(d. h. E_T ist die Menge der Kanten (q_1, q_2), für die q_2 Vater von q_1 ist).

Wir zeigen zunächst folgende Behauptungen:

Behauptung 1: T ist ein Spannbaum mit dem Wurzelknoten als Starter.

Beweis: Jeder Knoten hat maximal eine ausgehende Kante (die auch zu E gehört).
$\Longrightarrow$ T ist ein Wald mit Wurzeln $R = \{p \mid p : \texttt{father}$ undefiniert$\}$.
Sicherlich ist $Starter \in R$.
Sei $U = R \setminus \{Starter\}$

Zu zeigen: $U = \emptyset$

Annahme: $U \neq \emptyset$
Für jedes $p \in U$ und $(q,p) \in E$ gilt: q sandte keine Nachricht an p. $\implies q \in U$
$\implies P \subseteq U \implies$ Bei allen Prozessen ist der Vater undefiniert.
$\implies$ Widerspruch $\implies U = \emptyset$

$\implies T$ ist Spannbaum.

Kanten (q_1, q_2) mit $\{(q_1,q_2),(q_2,q_1)\} \cap E_T = \emptyset$ heißen *Querkanten*.

Behauptung 2: Über jede Querkante wird genau ein Token (pro Richtung) gesendet.

ohne Beweis

Behauptung 3: Seien T_p der Teilbaum von T mit Wurzel p und P_p die Prozesse in T_p.
Sei zudem v_p das Vorkommen des Ereignisses $e_p = (p$:`send(father,token)`)
(sofern e_p existiert). Dann gilt:

(1) v_p exisiert für jedes $p \neq Starter$.

(2) Sei $p \neq Starter$. Für jedes $s \in P_p$ gibt es ein Ereignis $(v,e) \leq (v_p, e_p)$ mit $e = (s : \_)$.

Beweis: Induktion über $|P_p|$
$|P_p| = 1$: d. h. p ist ein Blatt
$\implies$ alle Kanten (außer der zum Vater) sind Querkanten.
$\overset{Beh. 2}{\implies}$ Die `while`–Schleife wird verlassen. $\implies$ Beh. 3(1)

Beh. 3(2) gilt trivialerweise.

$|P_p| > 1$: Seien $p_1, \ldots, p_k$ Söhne von p in T_p.
Nach Induktionsvor. erhält p von jedem Prozeß p_i, $1 \leq i \leq k$ eine Nachricht (jeder Prozess p_i führt ja das Ereignis e_{p_i} aus).
Mit Beh. 2 folgt auch hier Beh. 3(1).

Zu Beh. 3(2): $P_p = \{p\} \cup P_{p_1} \cup \cdots \cup P_{p_k}$
Sicherlich gilt: $(v_p, e_p) \leq (v_p, e_p)$.
Weiterhin gilt für jedes $s \in P_j$: $\exists (v,e) \leq (v_{p_j}, e_{p_j})$ mit $e = (s : \_)$.
Da $(v_{p_j}, e_{p_j}) \leq (v_p, e_p)$, folgt die Beh. 3(2). □

Beweis von (E) und (A):

(E) Aus Beh. 2 und Beh. 3(1) folgt, daß der Starter seine `while`–Schleife verläßt.
$\implies$ Der Starter führt ein `decide` aus.

(A) Aus Beh. 1 und Beh. 3(2) folgt, daß in jedem Prozess $s \neq Starter$ ein Ereignis (v,e) $(e = (s : \_))$ stattgefunden hat, wobei (v,e) logisch vor dem `decide`–Ereignis liegt.
$\implies$ Beh. □

Komplexität:	Nachrichten:	$\lvert E \rvert$	(E Menge der gerichteten Kanten der Topologie)
	Zeit:	$O(\lvert P \rvert)$	(linear)

3.5.3 Der Phasen–Algorithmus (Tel 91)

Hier handelt es sich um einen dezentralen Algorithmus, bei dem keine bidirektionalen Kanäle gefordert werden. Allerdings muß der Durchmesser der Topologie bekannt sein.

Der *Durchmesser* D einer Topologie ist der maximale Abstand zweier Prozesse, wobei der *Abstand* zwischen zwei Prozessen q und p der Länge des kürzesten Weges von q nach p entspricht.

Algorithmus:

```
#define D  ...     /* Durchmesser der Topologie */
SetOfPid succ, pred;   /* Nachfolger–/Vorgängerprozesse */
int      rec[pred];
int      sent = 0;
Pid      q;

  forall (q ∈ pred)
    rec[q] = 0;
  if (starter) {   /* evtl. existieren mehrere Starter */
    forall (q ∈ succ)
      send(q,token);
    sent = 1;
    }
  while (∃q ∈ pred : rec[q] < D) {
    receive(&q,&token);
    rec[q]++;
    if ((sent < D) && (∀q ∈ pred : sent ≤ rec[q])) {
       forall (q ∈ succ)
         send(q,token);
       sent++;
       }
    }
  decide;   /* alle Prozesse entscheiden */
```

Satz: Der Phasen–Algorithmus ist ein Wellen–Algorithmus.

Beweis:

(T) Jeder Prozess kann maximal D Nachrichten an jeden seiner Nachfolger senden, d. h. insgesamt $D \cdot |E|$ Nachrichten.

(E) Sei γ eine terminale Konfiguration. Da aus jedem Kanal bis zu D Nachrichten gelesen werden, sind alle Kanäle leer.
Seien $\texttt{sent}_p$, $\texttt{rec}_q$ die Werte der entsprechenden Variablen im Prozess p in γ. Dann gilt für jeden Kanal (p, q):

(1) $\texttt{sent}_p = \texttt{rec}_q\texttt{[}p\texttt{]}$,

(2) $\texttt{rec}_q\texttt{[}p\texttt{]} > 0 \implies \texttt{sent}_p > 0$,

(3) $\texttt{sent}_p > 0$ für jeden Starter,

damit folgt schließlich

(4) $\texttt{sent}_p > 0$ für alle Prozesse p.

Sei p ein Prozeß mit minimalem $\texttt{sent}_p$, d. h. $\texttt{sent}_p \leq \texttt{sent}_q$ $(q \in P)$. Sei r ein Vorgänger von p. Dann gilt:

$$\texttt{sent}_r = \texttt{rec}_p\texttt{[}r\texttt{]} \geq \texttt{sent}_p \implies \texttt{sent}_p = D$$

Andernfalls wäre γ nicht terminal).

Daraus folgt, daß jeder Kanal genau D Nachrichten erhält.

$\implies$ Alle Prozesse verlassen die `while`–Schleife. $\implies$ Alle Prozesse entscheiden.

(A) Seien die Prozesse p, q gegeben. Der Durchmesser der Topologie sei $\leq D$ und es existiere ein Pfad $p = p_0, \ldots, p_\ell = q$ in der Topologie, der von p nach q führt mit $\ell \leq D$.
Seien weiterhin $s[c,i]$, $r[c,i]$ die i–ten Sende–/Empfangsereignisse im Kanal c.
Dann gilt: $s[(p_i,p_{i+1}),i+1] < r[(p_i,p_{i+1}),i+1]$ (nach Def. der Kausalität)
$r[(p_i,p_{i+1}),i+1] < s[(p_{i+1},p_{i+2}),i+2]$ (nach Konstr. des Algos)

$s[(p,p_1),1] = s[(p_0,p_1),1] < r[(p_{\ell-1},p_\ell),\ell] = r[(p_{\ell-1},q),\ell] \leq r[(p_{\ell-1},q),D]$
$\leq$ Entscheidung von q $\square$

Komplexität:	Nachrichten:	$D \cdot \lvert E \rvert$
	Zeit:	$O(D)$

3.5.4 FINN's Algorithmus (1979)

Auch hier handelt es sich um einen dezentralen Algorithmus, bei dem keine bidirektionalen Kanäle gefordert werden. Dieser Algorithmus verwendet die Prozeß–Identifier (PIDs) um Prozesse in zwei Mengen `I` und `J` einzutragen (`I`: Menge der bereits bekannten Prozesse, `J`: Menge der Prozesse, von denen bekannt ist, daß sie alle ihre Vorgänger kennen).

Algorithmus:

```
SetOfPid succ, pred;     /* Nachfolger–/Vorgängerprozesse */
SetOfPid I = {own_pid};  /* siehe oben */
SetOfPid J = {};         /* siehe oben */
SetOfPid nI = {}, nJ = {};
boolean  rec[pred];
Pid      p, q;

  forall (p ∈ pred)
    rec[p] = false;
  if (starter) /* evtl. existieren mehrere Starter */
    forall (p ∈ succ)
      send(p,<I,J>);

  while (I != J) {
    receive(&q,<&nI,&nJ>);
    nI = nI ∪ I;
    nJ = nJ ∪ J;
    rec[q] = true;
    if (∀p ∈ pred : rec[p] == true)
      nJ = nJ ∪ {own_pid};
    if ((I != nI) || (J != nJ)) {
      I = nI;
      J = nJ;
      forall (p ∈ succ)
        send(p,<I,J>);
      }
    }

  decide;
```

Satz: FINN's Algorithmus ist ein Wellen–Algorithmus.

<u>Beweis:</u>

Zu (T): Die Mengen `I` und `J` können nur größer werden $\Longrightarrow$ es gibt höchstens $|P|^2 \cdot |E|$ Nachrichten.

Zu (E) zeigen wir erst 2 Behauptungen:

Behauptung 1: Sendet p eine Nachricht an jeden Nachfolger, und ist q ein Nachfolger von p, dann sendet q mindestens eine Nachricht an jeden seiner Nachfolger
(d. h. durch jeden Kanal wird mindestens eine Nachricht gesendet).

Beweis:
Annahme: q sendet keine Nachricht.
Dann ist q kein Starter und für jedes empfangene Paar $\langle S_1, S_2 \rangle >$ gilt $S_1 \subseteq \{q\}$ und $S_2 = \emptyset$.
Sei $\langle S_1, S_2 \rangle$ jetzt ein Paar, das p sendet. Dann gilt $p \in S_1$. $\Longrightarrow$ $S_1 \not\subseteq \{q\}$. $\square$.

Aus Beh. 1 folgt, daß durch jeden Kanal mindestens eine Nachricht gesendet wird.

Sei γ die terminale Konfiguration.
Seien I_v, J_v die Werte der Variablen I, J im Prozeß p nach Ereignis $(v, (p, a))$, sowie I_p, J_p die entsprechenden Werte in der terminalen Konfiguration γ.

Behauptung 2: Für alle Prozesse p gilt $I_p = J_p$.

Beweis:
In γ gilt für jeden Kanal $(q_1, q_2) \in E$: $\quad I_{q_1} \subseteq I_{q_2}$ und $J_{q_1} \subseteq J_{q_2} \quad (*)$

Denn: Wir nehmen dazu an es gelte: $I_{q_1} \not\subseteq I_{q_2}$ oder $J_{q_1} \not\subseteq J_{q_2}$ für einen Kanal $(q_1, q_2) \in E$.

<u>Fall 1</u>: $I_{q_1} \not\subseteq I_{q_2}$ für Kanal (q_1, q_2)
Wegen Beh. 1 gibt es ein Ereignis $(v, (q_1, a))$ mit $I_v \subseteq I_{q_2}$.
Sei $(\bar{v}, (q_1, \bar{a}))$ ein letztes solches Ereignis, d. h. es gibt ein Nachfolgeereignis $(v', (q_1, a'))$, bei dem die Variable I einen Wert $I_{v'} \not\subseteq I_{q_2}$ erhält
$\Longrightarrow$ $I_{v'} \neq I_{\bar{v}}$ $\Longrightarrow$ q_1 sendet $I_{v'}$ an q_2
$\Longrightarrow$ $I_{v'} \subseteq I_{q_2}$ $\Longrightarrow$ **Widerspruch**

<u>Fall 2</u>: $J_{q_1} \not\subseteq J_{q_2}$ analog

Da die Topologie stark zusammenhängend ist, folgt aus $(*)$, daß sogar $I = I_{q_1} = I_{q_2}$ und $J = J_{q_1} = J_{q_2}$ für alle $q_1, q_2 \in P$ gilt.

Da für $q \in P$ stets $q \in I_q$ gilt, folgt $I = P$.
Außerdem impliziert Beh. 1, daß $q \in I_q$ für alle $q \in P$. Folglich gilt auch $J = P$. $\square$

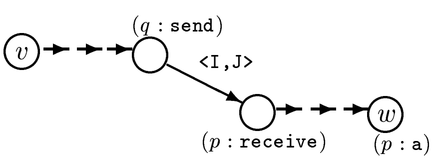
Wir schließen: Jeder Prozeß führt ein `decide` aus.

Bleibt (A) zu zeigen, d. h. jedes Entscheidungsereignis hängt von allen Prozessen ab.

Sei nun $(p : a)$ das interne Ereignis, bei dem Prozeß p den Test der `while`–Schleife ausführt.

Behauptung 3: Gilt $(v, (q : \_)) \leq (w, (p : a))$, dann ist $q \in I_w$.

Beweis:
Im Falle $q = p$ gilt die Behauptung stets. Sei nun $q \neq p$.
Dann muß mindestens eine Folge von Nachrichten von q nach p 'geflossen' sein.



$\Longrightarrow q \in I_w$

$\square$

Behauptung 4: Sei $\lambda(w) = (p : a)$.
Falls $q \in J_w$, dann gilt $(v, (r : \_)) \leq (w, (p : a))$ für alle Vorgänger r von q.

Beweis:
Für q sei (v_q, e_q) das interne Ereignis, bei dem erstmals $J := J \cup \{q\}$ ausgeführt wird.
Offenbar gilt:

- $q \in J_w \implies (v_q, e_q) \leq (w, (p : a))$
- Das Ereignis (v_q, e_q) kann nur dann stattfinden, wenn von jedem der Vorgänger eine Nachricht kam.

Daraus folgt die Behauptung. □

Folgerung:
Wenn Prozeß p entscheidet, dann verließ p die Schleife.
D. h.: Vor dem Testereignis w gilt $I_w = J_w$.
Sei nun $q \in I_w$. Dann ist $q \in J_w$.
Wegen Beh.4 gilt für alle Vorgänger r von q: $\exists (v_r, (r : \_)) \leq (w, (p : a))$.
Wegen Beh. 3 folgt $r \in I_w$.
Da die Topologie stark zusammenhängend ist und $I_w \neq \emptyset$ gilt, folgt $I_w = P = J_w$.
Wegen Beh. 4 folgt (A). □

Nachteil von FINN's Algorithmus:
Mit jeder Nachricht werden zwei Mengen von Prozessen versandt. Dies ist sehr aufwendig.

Nachrichtenkomplexität: $O(2 \cdot |P| \cdot |E|)$.

Verbesserung:
Sendet man mit jeder Nachricht nur die Änderungen der eigenen Mengen I, J, dann werden insgesamt $O(|P| \cdot \log |P| \cdot |E|)$ Bits versandt, falls jede PID mit $\log |P|$ Bit darstellbar ist.
Dies führt uns zum inkrementellen FINN–Algorithmus.

3.5.5 Inkrementeller FINN–Algorithmus

In diesem Algorithmus sind alle Prozesse Starter.

Algorithmus:

```
SetOfPid   succ, pred;    /* Nachfolger–/Vorgängerprozesse */
SetOfPid   I = {own_pid}, J = {};
int        rec[pred];
int        sent = 1, N = 2, n;
enum {i,j} tag;
Pid        p, q, new;

  forall (p ∈ pred)  rec[p] = 0;
  /* alle Prozesse sind Starter */
  forall (p ∈ succ)
    send(p,<i,own_pid,1>);
```

```
while ((∃p ∈ pred: rec[p] < N) || (I != J)) {
  receive(&q,<&tag,&new,&n>);
  rec[q]++;
  N = N ⊔ n;
  switch(tag) {
    case i: if (new ∉ I) {
              I = I ∪ {new};
              sent++;
              forall (p ∈ succ)
                send(p,<i,new,sent>);
              }
            break;
    case j: if (new ∉ J) {
              J = J ∪ {new};
              sent++;
              forall (p ∈ succ)
                send(p,<j,new,sent>);
              }
    }   /* switch */
  if ((own_pid ∉ J) && (∀p ∈ pred : rec[p] > 0))
    J = J ∪ {own_pid};
    sent++;
    forall (p ∈ succ)
      send(p,<j,own_pid,sent>);
    }
  }

decide;
```

Achtung:

- Falls von vornherein **nicht** alle Prozesse starten, muß eine Aufforderung zum Starten im Voraus durchs Netz geflutet werden !
- Der neue Algorithmus sendet exakt $2 \cdot |P|$ Nachrichten durch jeden Kanal. Insgesamt werden also $2 \cdot |P| \cdot |E|$ NAchrichten versandt.
- Jede Nachricht enthält $O(\log |P|)$ Bits.

Kapitel 4

Election–Algorithmen

Definition:
Ein *Election–Algorithmus* ist ein Algorithmus mit folgenden Eigenschaften:

(G) Jeder Prozeß hat den gleichen Algorithmus (Gleichheit).

(D) Der Algorithmus ist dezentral (Dezentralisiertheit).

(T) Der Algorithmus terminiert stets. In jeder erreichbaren terminalen Konfiguration ist genau ein Prozess im Zustand `leader`, alle anderen sind im Zustand `lost`.

Annahmen:

1. Alle Prozesse haben eine Variable `state` (`enum {leader,lost,...} state;`).
2. Jeder Prozeß besitzt eine eindeutige PID.
3. Jede Nachricht darf nur $O(\log |P|)$ Bits enthalten.

4.1 Election mittels Infimum–Berechnung

Idee: Wähle den Prozeß mit der kleinsten PID. Prozesse, die in der Variable *out* das Infimum berechnen, müssen anschließend diesen Wert allen anderen Prozessen mitteilen.

Geeignet sind hier der Phasen–Algorithmus (mit $O(D \cdot |E|)$) Nachrichten) und der inkrementelle Algorithmus von Finn (mit $O(|P| \cdot |E|)$) (Echo–Algorithmus ist hier ungeeignet).

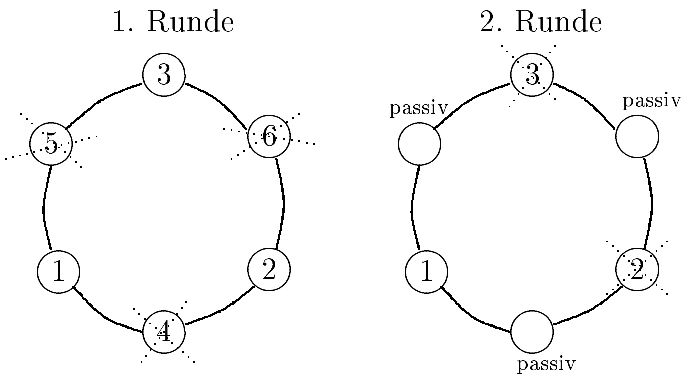
4.2 Election auf einem unidirektionalen Ring

Ideen:

(1) Organisiere die Election in Runden, wobei in jeder Runde mindestens die Hälfte aller verbleibenden Kandidaten wegfällt (mindestens einer soll aber stets übrig bleiben). Folglich genügen $\lfloor \log |P| \rfloor$ Runden.

(2) Vergleiche die PIDs dreier benachbarter Prozesse. Der kleinste PID bildet ein lokales Minimum.
Behalte einzig das lokale Minimum; die übrigen beiden Prozesse werden 'passiv'.
Das globale Minimum ist sicherlich auch lokales Minimum. Keine zwei lokalen Minima sind benachbart.

Beispiel:



Beobachtung: In jeder Runde bleibt maximal die Hälfte der Prozesse übrig.
Folglich sind $\lfloor \log |P| \rfloor$ Runden erforderlich, bis schließlich ein Prozeß übrig bleibt.

(3) Trenne die Identifier von den Prozessen. Da der Ring unidirektional ist, werden die Prozeßidentifier herumgereicht. In jeder Runde wandern die aktiven Identifier jeweils eine Etappe vor.
Diese Methode wird durch den folgenden Algorithmus realisiert.

4.2.1 Algorithmus von Peterson/Dolev–Kleve–Rodeh (1982)

Algorithmus:

```
Pid succ, pred;     /* Nachfolger–/Vorgängerprozeß */
Pid ci  = own_pid;  /* current identity */
Pid api = -1;       /* active predecessor identity */
Pid win = -1;       /* winner */
Pid q;
int round = 0;
enum {active,passive,leader,lost} state = passive;
enum {smal,one,two}                tag;

  if (starter)
    state = active;  /* nur aktive Prozesse nehmen an der Auswahl teil */
  while (win != -1) {
    if (state == active) {
      send(succ,<one,round,ci>);
      receive(pred,<one,round,&api>);
      if (api != ci) {   /* noch Mitbewerber im Rennen */
        send(succ,<two,round,api>);
        receive(pred,<two,round,&q>);
```

```
          if (api < (ci ⊓ q))   /* lokales Minimum */
            ci = api;    /* shift */
          else
            state = passive;
          }
        else {  /* api == ci, kein weiterer Prozeß aktiv ⇝ ci ist winner */
          send(succ,<smal,ci>);
          receive(pred,<smal,-1>);
          win = ci;
        }  /* if (ci != api)  */
      else {  /* state == passive */
        receive(pred,<one,round,&q>);
        send(succ,<one,round,q>);
        receive(pred,<&tag,round,&q>);
        send(succ,<tag,round,q>);
        if (tag == smal)
          win = q;
        }
      round++;
      } /* while */
    if (own_pid == win)
      state = leader;
    else
      state = lost;
```

Anzahl der Nachrichten pro Runde: $2 \cdot |P|$
Anzahl der Runden: $\leq \lfloor \log |P| \rfloor + 1$
$\Longrightarrow$ es werden höchstens $2 \cdot |P| \cdot (\lfloor \log |P| \rfloor + 1)$ Nachrichten verschickt. [1]

4.3 Extinction

Sei A ein zentralisierter Wellenalgorithmus. Dann erhält man daraus einen *Election-Algorithmus* $Ex(A)$ wie folgt:

1. jeder Starter p (von $Ex(A)$) initiiert eine Welle gemäß A, die durch Anhängen von `own_pid` von p an die Nachrichten kenntlich gemacht wird.

2. jeder Prozeß ist nach jedem Ereignis in maximal einer Welle **aktiv**, deren Initiator in der Variable *caw* vom Typ P abgelegt ist. Diese Variable („current active wave") wird mit -1 initialisiert; die Starter setzen *caw* anfangs auf `own_pid`.

3. Erhalte Prozeß q eine Nachricht aus der Welle p (p Starter der Welle).
   Fall 1: Ist $p < caw$ oder $caw == -1$, so stellt q seinen Anfangszustand wieder her, setzt $caw = p$ und schließt sich der Welle von p an.
   Fall 2: Ist $p == caw$, so wird die Nachricht gemäß Algorithmus A behandelt.
   Fall 3: Ist $p > caw$, so wird die Nachricht ignoriert ($\leadsto$ die Welle von p schlägt fehl).

[1] HIGHAM, PRZYTYCKA haben 1993 gezeigt, daß ein Faktor von 1,271 genügt.

4. Wenn das Entscheidungsereignis ausgeführt wird, ist *caw* der Gewinner. Falls dies nicht in allen Prozessen erfolgt, informieren die Entscheidungsprozesse alle anderen Prozesse über den Gewinner (Idee: sende entsprechende Nachricht genau einmal durch jeden Kanal).

Korrektheit:

1. Sei p kleinster Starter, dann nimmt jeder Prozeß an der Welle von p teil.
   $\Longrightarrow$ In dieser Welle findet ein Entscheidungsereignis statt.
   $\Longrightarrow$ p wird als Leader ausgerufen.
2. Sei p kein Starter, dann wird keine Welle für p gestartet. $\Longrightarrow$ p wird nicht Leader.
3. Sei p Starter, allerding nicht kleinster Starter, dann wird eine Welle für p gestartet.
   Der kleinste Starter nimmt aber an dieser Welle nicht teil.
   $\Longrightarrow$ In dieser Welle wird kein Entscheidungsereignis ausgeführt.
   $\Longrightarrow$ p wird nicht Leader. □

Sei w Anzahl der Nachrichten des Algorithmus A. Dann erfordert $Ex(A)$ mit t Startern höchstens $t \cdot w$ Nachrichten.

4.4 Election mittels minimaler Spannbäume

Wir konstruieren nun einen Spannbaum der Topologie und führen die Election auf dieser Teiltopologie aus. Wir betrachten allerdings hier nur Election auf ungerichteten Bäumen.

Idee:

(1) Die Starter wecken alle Blätter auf

(2) Das Infimum der PIDs wird berechnet

Algorithmus:

```
boolean  awake = false;
int      wrec = 0;   /* erhaltene wake-ups */
Pid      v = pid;
Pid      q, q', r;
SetOfPid silent = neighbors;

if (starter) {  /* initiiere wake-up */
  awake = true;
  forall (q ∈ neighbors)
    send(q,wakeup);
  }
```

```
while (wrec < |neighbors|) {
  receive(-1,wakeup);
  wrec++;
  if (!awake) {
    awake = true;
    forall (q ∈ neighbors)
      send(q,wakeup);
    }
  } /* end of wake-up */
/* Infimum-Berechnung */
while (|silent| > 1) { /* mind. 2 Nachbarn sendeten keine Botschaft */
  receive(&q,<tok,&r>);
  v = v ⊓ r;
  silent = silent \ {q};
  }
let silent = {q} in {
  send(q,<tok,v>);        /* reiche bisheriges Minimum weiter */
  receive(q,<tok,&r>);
  v = v ⊓ r;    /* Minimum */
  if (v == pid)
    state = leader;
  else
    state = lost;
  forall (q' ∈ neighbors\{q})
    send(q',<tok,v>);
  }
```

Es werden $4 \cdot (|P| - 1)$ Nachrichten geschickt.
Beachte: Die Infimumberechnung ist eine Welle auf einem Baum, bei der alle Blätter Starter sind.

Problem: Finde einen Spannbaum in einer beliebigen (ungerichteten) Topologie.
Der Algorithmus von GALLAGER–HUMBLET–SPIRA (1983) berechnet den minimalen Spannbaum mit
$O(|E| + |P| \cdot \log |P|)$ Nachrichten.[2]
Wir betrachten hier nur eine vereinfachte Version, die immerhin $O(\log |P| \cdot |E|)$ Nachrichten benötigt.

Annahme: Es existiere eine Kostenfunktion *cost* in die lineare Ordnung D $cost : E \longrightarrow D$ mit $cost(e_1) \neq cost(e_2)$ für $e_1 \neq e_2$.
Eine solche Kostenfunktion kann immer aus den PIDs konstruiert werden:

$$cost(\{q_1, q_2\}) = \begin{cases} (q_1, q_2) & \text{falls} \quad q_1 < q_2 \\ (q_2, q_1) & \text{falls} \quad q_2 < q_1 \end{cases} .$$

Die Paare sind lexikalisch geordnet.

Die vereinfachte Version von GALLAGER–HUMBLET–SPIRA benötigt nur $O(\log |P| \cdot |E|)$ Nachrichten auf einer ungerichteten Topologie.

[2] für einen Ring gibt es sogar eine untere Schranke von $\Omega(|P| \cdot \log |P|)$, falls auf PIDs nur Vergleiche erlaubt sind.

Sei *cost* : $E \longrightarrow D$ eine injektive Kostenfunktion mit linearer Ordung D.

Idee: Berechne minimalen Spannbaum und bestimme den Führer auf diesem.

Satz
Ist *cost* injektiv, dann gibt es genau einen minimalen Spannbaum (MST) $T = (P, E_T)$

Beweis: Übungsaufgabe

Definition
Sei $F \subseteq E$. F heißt *Fragment*, falls F in der Kantenmenge eines minimalen Spannbaums enthalten ist.
Sei P_F die zu F gehörige Menge von Prozessen. Dann heißt $e \in E$ *ausgehende Kante* von F, falls $e \cap P_F \neq \emptyset$, aber $e \notin P_F$.

(*Hier soll eins von Hutti's Bildern stehen.*)

Satz
Ist e billigste ausgehende Kante eines Fragments F, so ist $F \cup \{e\}$ wieder ein Fragment.

Ohne Beweis.

Idee:

- Lasse parallel Fragmente wachsen.
- Anfangs besteht jedes Fragment aus genau einem Knoten.
- In jeder Runde werden einige Fragmente verschmolzen.

  (*Und hier soll eins von Hutti's Bildern stehen.*)

  Dies geschieht, wenn es eine billigste ausgehende Kante des einen Fragments gibt, die zu einem anderen hinführt.

Satz
Es sind maximal $\lfloor \log |P| \rfloor$ Runden erforderlich.

Beweis
Sei $n[j]$ Anzahl der Fragmente nach Runde j.
Zu zeigen: $n[j] \leq \frac{|P|}{2^j}$ $(j \geq 0)$

$j = 0$: $n[j] = |P|$ $\surd$

$j > 0$: Sei $F_1, F_2, \ldots, F_n$ die Folge der Fragmente nach Runde $j-1$,
$e_1 \ldots, e_n$ die Folge der billigsten ausgehenden Kanten (d.h. $n = n[j-1]$).
$n[j]$ ist die Anzahl der Zusammenhangskomponenten des Graphen (F', E')
mit $F' = \{F_1, \ldots, F_n\}$ und $E' = \{\{F_i, F_j\} \mid (e_i \cap F_j \neq \emptyset) \vee (e_j \cap F_i \neq \emptyset)\}$
(*Und hier soll eins von Hutti's Bildern stehen.*)
Jede Zusammenhangskomponente enthält mindestens 2 Knoten.
$\Longrightarrow n[j] \leq \frac{n[j-1]}{2} \stackrel{\text{IV}}{\Longrightarrow}$ Behauptung

Probleme

1. Benennung der Fragmente:
   Benutze MST um Prozeß mit kleinster Pid im Fragment zu bestimmen.
   $\Longrightarrow$ Name (Nachrichten: $O(|P|)$ pro Runde)

2. Bestimmung der ausgehenden Kanten:
   Sende Fragmentname über jede alte ausgehende Kante (Kosten $O(|E|)$ pro Runde).
3. Bestimmung der billigsten ausgehenden Kante:
   Idee: Infimumsberechnung auf MST (Kosten $O(|P|)$ pro Runde).
   $\Longrightarrow$ insgesamt $O(|E| \cdot \log |P|)$.

Bemerkung

- untere Schranke: $\Omega(|E| + |P| \cdot \log |P|)$.
- Gallager–Humblet–Spira erreicht diese, indem Schritt 2 verbessert wird.

Algorithmus:

```
Pid Name = own_pid; /* Fragment-Name */
int round = 0;
SetOfPid neighbours;
SetOfPid Tree = {};  /* Baumnachbarn */
int min = ⊓{cost({own_pid,q}) | q ∈ Out};
SetOfPid minOut = {own_pid,q} with cost(own_pid,q)  = min;

while (min ≤ ∞) {
  connect(round);
  newName();
  newOut(round);
  newMinOut();
  round++;
  }

void connect (int round) {
int rec;
Pid q;

  forall (q ∈ Out) {
    if ({pid,q} == minOut) {
      Out = Out \ {q};
      Tree = Tree ∪ {q};
      send(q,<connect,round>);
      }
    else
      send(q,<nothing,round>);
    }
  for (rec = 0; rec ≤ |Out|; rec++) {
    receive(&q,<&tag,round>);
    if (tag == connect) {
      Out = Out \ {q};
      Tree = Tree ∪ {q};
      }
    }
}  //  end of connect()

void newName () {
  name = treeElect(); /* Election auf dem Fragment */
  }
```

```
void NewOut (int round) {
int rec;
Pid q, F;
  forall(q ∈ Out )
    send(q,<test,round,name>);
  for(rec = 0; rec < |Out|; rec++) {
    receive(&q,<test,round,&F>);
    if(name == F)
      Out = Out \ {q};
    }
}

void MinOut() {
   ··· /* Infimumberechnung auf Fragment */
  }
```

Kapitel 5

Terminierungserkennung

Eine Berechnung (Run) eines verteilten Algorithmus' terminiert, falls sie eine terminale Konfiguration erreicht hat; also eine Konfiguration in der kein weiterer Berechnungsschritt stattfinden kann. Allerdings könnten in einigen Prozessen durchaus noch Ereignisse möglich sein (etwa warten einige Prozesse noch auf Nachrichten, obwohl alle Kanäle des Systems leer sind).
In solchen Konfigurationen sind nicht alle Prozesse darüber informiert, daß die Berechnung terminiert ist (*implizite* Terminierung oder *Nachrichtenterminierung*).
Befindet sich hingegen jeder Prozeß in einem terminalen (Prozeß-)Zustand (kein Ereignis ist mehr anwenderbar), d. h. der Algorithmus hat terminiert, dann sprechen wir auch von *expliziter* Terminierung (*Prozeßterminierung*).

Hier wollen wir nun Methoden untersuchen, die implizit terminierende Algorithmen in solche überführen, die explizit terminieren.

Dazu geben wir zwei Algorithmen an:

- Einer, der die Ausführung des implizit terminierenden Algorithmus überwacht, und feststellt, ob dieser eine terminale Konfiguration erreicht hat.
- Ist eine terminale Konfiguration erreicht, flutet ein weiterer Algorithmus Nachrichten durch das System, die alle Prozesse von der terminierung in Kenntnis setzen und dazu veranlassen, in einen expliziten terminalen Zustand überzugehen.

Diese beiden Algorithmen bezeichen wir zusammenfassend als *Kontrollalgorithmus*. Der implizit terminierende Algorithmus, dem diese beiden "übergestülpt" werden, heißt *Basisalgorithmus*.

5.1 Vorüberlegungen

5.1.1 Formalismen

Die Zustände eines Prozesses werden eingeteilt in

- passive Zustände: nur ein `receive`-Ereignis ist möglich.
- aktive Zustände: nur ein interner Schritt ι oder ein `send`-Ereignis ist ausführbar.

Zur Beobachtung der Zustände statten wir jeden Prozess in folgender Weise mit einer Variablen `enum {active,passive} state` aus, die die jeweilige Eigenschaft angibt:

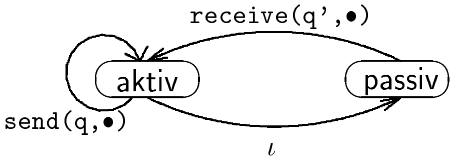
vorher	nacher

`receive(p,m);`	`state = passive;`
	`receive(p,m);`
	`state = active;`

`send(p,m);`	`send(p,m);`
`change_state();`	`change_state();`

Der Kontrollalgorithmus soll für jeden beliebigen Basisalgorithmus funktionieren (z. B. als Betriebssystemroutine).
Dabei hat er nur auf beschränkt viele Informationen des Basisalgorithmus' Zugriff:
Er kennt den Wert der `state`-Variablen und Sender bzw. Empfänger von Nachrichten, **nicht** aber den Inhalt von Nachrichten.

Aus der Sicht des Kontrollalgorithmus' sieht der Basisalgorithmus für einen Prozeß p folgendermaßen aus:



`q` ist ein Nachfolgerprozeß, `q'` ein Vorgängerprozeß von p, ι ein internes Ereignis in p.

Mindestens ein Prozeß muß in der Ausgangskonfiguration aktiv sein.

Der folgende Algorithmus steht in den weiteren Betrachtungen stellvertretend für irgendeinen Basisalgorithmus.

Der Basisalgorithmus:

```
enum {active,passive} state;
Pid q;

while (1)
  switch (state) {
    case active:  select {   /* nichtdeterm. Auswahl von Statements */
                    choose q;  /* nichtdeterm. Auswahl eines Empf. */
                    send(q,•);
                    }
                  or
                    state = passive;
                  break;
    case passive: receive(-1,•);
                  state = active;
    }
```

Achtung:

- Eine mögliche Blockierung infolge andauernd eintreffender unerwünschter Nachrichten ignorieren wir.
- Der Kontrollalgorithmus wird in den Basisalgorithmus eingefügt. Der Basisalgorithmus wird also transformiert.

Satz: Terminierung liegt in einer Konfiguration $\gamma = (\sigma, \mu)$ genau dann vor, wenn für alle Prozesse $p \in \mathcal{P}$ folgende Eigenschaft **Term** gilt: $\sigma(p) = \texttt{passive} \wedge (\forall c \in E : \mu(c) = \emptyset)$.

Beweis:
"$\Rightarrow$" : Sei γ terminale Konfiguration.
Dann sind alle Prozesse passiv, d.h. $\sigma(p) = \texttt{passive}$ für alle $p \in \mathcal{P}$.
Wenn ein Kanal $c = (p, q)$ eine Nachricht enthielte, so könnte q einen Schritt ausführen. D. h. γ wäre nicht terminal. $\Longrightarrow$ Widerspruch $\Longrightarrow$ $\mu(c) = \emptyset$.
"$\Leftarrow$" : Es gelte **Term** für γ.
Da alle Prozesse passiv sind, können nur `receive`-Ereignisse eintreten.
Da die Kanäle leer sind, kann nichts empfangen werden.
$\Longrightarrow$ γ ist terminale Konfiguration. $\square$

Der Kontrollalgorithmus *erkennt* das Vorliegen von Terminierung und überführt jeden Prozeß in einen neuen terminalen Zustand `halt`.

Ein Kontrollalgorithmus muß folgende Eigenschaften haben:

(U) Ungestörtheit: Der Basisalgorithmus wird vom Kontrollalgorithmus nicht beeinflußt.

(L) Lifeness: Falls Terminierung eintritt, geht jeder Prozeß nach endlich vielen Schritten in den `halt`-Zustand über.

(S) Safety: Erreicht ein Prozeß den Zustand `halt` erreicht, dann gilt **Term**.

Wir bezeichnen einen Kontrollalgorithmus zur Terminierungserkennung auch als *Termination-Detection Algorithmus*, kurz TD-Algorithmus.

5.1.2 Untere Schranken

Wir verwenden folgende Parameter:

$\lvert\mathcal{P}\rvert$	Anzahl der Prozesse
$\lvert E\rvert$	Anzahl der Kanäle
M	Anzahl der versendeten Nachrichten des Basisalgorithmus' bis zur Terminierung
W	Anzahl der Nachrichten eines besten Wellen-Algorithmus' für die vorliegende Topologie

Satz:
Für jeden TD-Algorithmus A gibt es einen terminalen Run des Basisalgorithmus' mit M Nachrichten, so daß A mindestens M Kontroll-Nachrichten austauscht.

Intuition: Jedem Sendeereignis muß Rechnung getragen werden.

Satz: (Chandrasekaran/Venkatesan (90))
Jeder TD-Algorithmus benötigt mindestens $|E|$ Nachrichten, sofern

(1) alle Kanäle FIFO sind

(2) der TD-Algorithmus zu einem beliebigen Zeitpunkt nach dem Basisalgorithmus gestartet wird.

Intuition: Eine letzte Kontrollnachricht im Kanal garantiert, daß keine weiteren Nachrichten mehr vorhanden sind.

5.1.3 Beenden aller Prozesse nach Terminierungserkennung

Um allen Prozessen die Terminierung mitzuteilen, wird eine `stop`-Nachricht durch das Netzwerk geflutet.
Dazu sendet jeder Prozeß eine `stop`-Nachricht an seine Nachbarn. Hat jeder Prozeß von allen seinen Nachbarn eine `stop`-Nachricht empfangen, dann geht er in den `halt`-Zustand über.

Folgender Algorithmus flutet `stop`-Nachrichten durch alle Kanäle.

```
boolean sentStop = false;
int     recStop = 0;
Pid     q;
enum {active,passive} state;

void announce() {
  if (!sentStop) {
    sentStop = true;
    forall (q ∈ succ)
      send(q,stop);
  }
}
```

Der Basisalgorithmus wird noch in geeigneter Weise erweitert.

<u>Modifizierter Basisalgorithmus:</u>

```
while (1)
  switch (state) {
    case active:  select {
                    choose q;
                    send(q,•);
                    }
                  or
                    state = passive;
                  break;
    case passive: receive(-1, &msg);
                  switch (msg) {
                    case •    : state = active;
                                break;
                   ┌─────────────────────────────────────────┐
                   │case stop: recStop++;                    │
                   │           announce();                   │
                   │           if (recStop == |pred|) halt;  │
                   └─────────────────────────────────────────┘
                    }
    }
  }
```

Die Benachrichtigung von der Terminierung läuft an, nachdem mindestens ein Prozeß eine `stop`-Nachricht an alle seine Nachbarn gesendet hat bzw. `announce` ausführte.

5.2 TD-Algorithmen für ungerichtete Topologien

Die beiden im folgenden vorgestellten Algorithmen basieren auf einem dynamischen *Berechnungsgraphen* G, dessen innere Knoten aktive Prozesse und dessen Blätter Basisnachrichten repräsentieren, die sich gerade im Übertragungskanal befinden. Ist der Berechnungsgraph leer, dann hat der Basisalgorithmus terminiert.

5.2.1 Zentralisierter Basisalgorithmus

Der hier vorgestellte Algorithmus von DIJKSTRA und SCHOLTEN (80) erkennt die Terminierung eines zentralisierten Basisalgorithmus'.

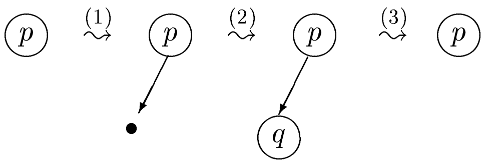
Sei p_0 der einzige aktive Prozeß der Anfangskonfiguration (der Starter).
Der DIJKSTRA-SCHOLTEN-Algorithmus verwaltet einen Berechnungsbaum $T = (V_T, E_T)$ mit folgenden Eigenschaften:

(1) T ist ein Baum mit Wurzel p_0.

(2) V_T umfaßt alle aktiven Prozesse und alle Nachrichten aller Kanäle.

Der Starter p_0 ruft `announce()` auf, sobald $V_T = \{p_0\}$ und p_0 passiv ist.

Um die Eigenschaften (1) und (2) zu sichern, muß der Berechnungsgraph folgendermaßen konstruiert werden:



- p: `send(q,•)`: Sendet p eine Basisnachricht, dann wird $\bullet$ als Sohn von p in den Berechnungsgraphen eingefügt (Übergang (1)).
- q: `receive(p,•)`: Wurde in Prozeß q eine Basisnachricht von p empfangen und gilt $q \notin V_T$, dann wird q an den (Vater-)Knoten p im Berechnungsgraphen eingefügt (Übergang (2)).
- $q : \iota$: Hat q keine Söhne und findet ein internes Ereignis ι in q satt, dann wird q aus dem Berechnungsgraphen gelöscht (Übergang (3)).

Repräsentation des Berechnungsgraphen im Algorithmus:

- Wir verwalten einen Zähler `sc` für die Anzahl der Söhne eines Knoten q und einen Verweis `father` auf den Vater-Knoten in T (für $q \in \mathcal{P} \cap V_T$).
- Die Wurzel p_0 hat als einziger Knoten sich selbst als Vater.
- Die Nachricht `sig` signalisiert einem Vaterknoten, daß einer seiner Söhne gelöscht wurde.

Algorithmus:

```
int recStop = 0, sc = 0;
Pid q, father = -1;
enum {•, sig, stop}     msg;
enum {passive, active} state;

if (starter)
  { father = own_pid; state = active; }

while (1)
  switch (state) {
    case active : select {
                    choose q;
                    send(q,•);
                    sc++;
                    }
                  or {
                    state = passive;
                    if (sc == 0)
                      if (father == own_pid)  /* ich bin die Wurzel */
                        announce();
                      else {
                        send(father, sig);
                        father = -1;
                        }
                    }
                  break;
    case passive: receive(&q, &msg);
                  switch (msg) {
                    case  •  : state = active;
                               if (father == -1)
                                 father = q;
                               else  /* sc des Vaters 'runterzählen */
                                 send(q, sig);
                               break;
                    case  sig: sc--;
                               if (sc == 0)
                                 if (father == own_pid)
                                   announce();
                                 else {
                                   send(father, sig);
                                   father = -1;
                                 }
                               break;
                    case stop: recStop++;
                               announce();
                               if (recStop == |pred|)
                                 halt;
                    }
    }   /* switch(state) */
```

Bleibt noch zu zeigen, daß T die gewünschten Eigenschaften besitzt.

Für jede Konfiguration γ (vor Ausführung von `anounce`) sei

$$\begin{array}{rcl} V_T &=& \{p \mid \texttt{father}_p \neq -1\} \cup \bigcup_{c \in E} \mu(c) \\ E_T &=& \{(p, \texttt{father}_p) \mid \texttt{father}_p \neq -1 \wedge \texttt{father}_p \neq \texttt{own\_pid}_p\} \\ &\cup& \{(\bullet, p) \mid \bullet \in \mu(p,q)\} \\ &\cup& \{(\texttt{sig}, p) \mid \texttt{sig} \in \mu(p,q)\} \end{array}$$

Sei P die folgende Eigenschaft:

$$\begin{array}{llr} & \texttt{state}_p = \texttt{active} \;\Rightarrow\; p \in V_T & (1) \\ \wedge & (u,v) \in E_T \;\Rightarrow\; (u \in V_T \wedge v \in V_T \cap \mathcal{P}) & (2) \\ \wedge & \texttt{sc}_p = |\{v \mid (v,p) \in V_T\}| & (3) \\ \wedge & T \text{ ist ein Baum mit Wurzel } p_0 & (4) \\ \wedge & (p \neq p_0 \wedge \texttt{state}_p = \texttt{passive} \wedge \texttt{sc} = 0) \Rightarrow p \notin V_T & (5) \end{array}$$

(1) besagt, daß T alle aktiven Prozesse enthält. (2) bedeutet, daß alle Kanten des Graphen auf Prozesse zeigen. Wegen (3) sind alle Sohn-Zähler `sc` korrekt und wegen (5) enthält der Baum nur noch p_0, falls Terminierung vorliegt.

<u>Lemma:</u> P ist Invariante des DIJKSTRA-SCHOLTEN-Algorithmus'.

Ohne Beweis.

<u>Satz:</u> Der DIJKSTRA-SCHOLTEN-Algorithmus ist ein korrekter TD–Algorithmus, der M Kontrollnachrichten benötigt (bevor p_0 `announce` ausführt).

<u>Beweis:</u>
Sei $S = \sum_{p \in \mathcal{P}} \texttt{sc}_p$. Anfangs ist $S = 0$.
S erhöht sich mit jeder Basisnachricht $\bullet$ und erniedrigt sich mit jeder empfangenen Kontrollnachricht `sig`. Wegen $S \geq 0$, kann die Anzahl der Kontrollnachrichten nie die Anzahl der Basisnachrichten übersteigen. $\Longrightarrow$ Nachrichtenkomplexität M.
Die Safety-Eigenschaft folgt direkt aus obigem Lemma: p_0 kann `announce` nur aufrufen, wenn p_0 keine Söhne mehr hat. $\Longrightarrow$ Alle Prozesse sind passiv und alle Kanäle leer. $\Longrightarrow$ **Term**.
Für die Lifeness-Eigenschaft gilt: Habe der Basisalgorithmus terminiert. Dann können nur noch Kontrollnachrichten vom Typ `sig` versandt werden. $\Longrightarrow$ S erniedrigt sich in jedem Schritt.
Für $S = 0$ muß $V_T = \{p_0\}$ gelten und p_0 ist passiv mit $\texttt{sc}_{p_0} = 0$. $\Longrightarrow$ p_0 ruft `announce` auf. □

5.2.2 Dezentraler Basisalgorithmus

Der SHAVIT-FRANCEZ-Algorithmus (86):

- Statt einem Baum T verwalten wir hier einen Wald F, dessen Menge von Wurzeln R eine Teilmenge der Initiatoren ist.
- Jeder Prozeß erhält ein Prädikat `root` mit $\texttt{root}_p = 1 \iff p \in R$.
- Initiatoren, deren Baum leer ist, setzen $\texttt{root} = 0$ und starten eine dezentrale Welle.
- An einer Welle nehmen nur Prozesse p mit $\texttt{root}_p = 0$ teil.
- Entscheidet die Welle, wird `announce` aufgerufen (sofern nicht alle entscheiden).

<u>Satz:</u> Der Algorithmus von SHAVIT-FRANCEZ ist ein korrekter TD–Algorithmus und benötigt maximal $M + W$ Kontrollnachrichten, wobei W die Anzahl der Nachrichten einer dezentralen Welle ist.

5.3 TD-Algorithmen für gerichtete Topologien

5.3.1 Zentralisierter Basisalgorithmus

Wir treffen die Zusatzannahme, daß jeder Prozeß **direkt** eine Nachricht an den Initiator p_0 senden kann; d. h. die Topologie enthält einen Stern mit Zentrum p_0. (Abschwächung: die Topologie enthält einen gerichteten Spannbaum mit Wurzel p_0.)

In diesem Fall kann man den *Credit-Recovery-Algorithmus* (MATTERN (89)) anwenden.

Idee: Jede Nachricht und jeder Prozeß erhält einen Kredit c mit $0 \leq c \leq 1$.

Der Algorithmus besitzt folgende Invarianten:

$S1$ Die Summe aller Kredite ist 1.

$S2$ Jede Basisnachricht hat einen Kredit > 0.

$S3$ Jeder aktive Prozeß hat einen Kredit > 0.

Ein Prozeß, der über positiven Kredit verfügt, sendet diesen an den Starter p_0.
Dieser agiert wie eine Bank und speichert die zurückgezahlten Kredite in einer Variablen `rsum`. Sind alle Kredite zurückgezahlt, kann es keine aktiven Prozesse und keine Nachrichten in den Kanälen mehr geben.

Kredite werden nach folgenden Regeln bearbeitet:

- **Regel 1**: Ist `rsum` $= 1$, ruft p_0 `announce` auf.
- **Regel 2**: Wird ein Prozeß p passiv, sendet p seinen Kredit an p_0 zurück.
- **Regel 3**: Wird ein Prozeß p aktiv ist und sendet eine Basisnachricht, teilt p seinen Kredit zwischen sich und der Nachricht auf.
- **Regel 4**: Empfängt ein passiver Prozeß eine Basisnachricht, so erhält er den Kredit dieser Nachricht.

Algorithmus:

```
int   recStop = 0;
float d, c = 0, rsum = 0;
Pid   q;
enum {<•,_>, <ret,_>, stop}     msg;
enum {passive, active} state;

 if (starter) {
   state = active;
   c = 1;
   }
```

```
while (1) {
  switch(state) {
    case active: select {
                      choose q;
                      c = c/2;          /* Regel 3 */
                      send(q,<•, c >);
                      }
                    or {
                      state = passive;
                      if (!starter)
                        send(p0,<ret,c>);
                      else {   /* p0 "sendet" an sich selbst */
                        rsum = rsum + c;
                        if (rsum == 1) announce();
                        }
                      c = 0; /* Regel 2 */
                      }
                    break;
    case passive: receive(-1,&msg);
                    switch(msg) {
                      case <•,&d > : state = active;
                                     c = d;   /* Regel 4 */
                                     break;
                      case <ret,&d>: rsum = rsum + d;
                                     if (rsum == 1) announce();
                                     break;
                      case  stop :   recStop++;
                                     announce();
                                     if (recStop == |pred|)
                                       halt;
                      }  /* switch(msg) */
    }  /* switch(state) */
```

<u>Satz:</u> Der Credit-Recovery-Algorithmus ist ein korrekter TD-Algorithmus.

<u>Beweis:</u>
Der Algorithmus implementiert die Regeln 1 bis 4.
$\Longrightarrow$ Er erfüllt die Invariante $S_1 \wedge S_2 \wedge S_3$, wobei:

$$S1 \equiv 1 = \sum_{\langle \bullet, \mathtt{c}\rangle} \mathtt{c} + \sum_{p \in \mathcal{P}} \mathtt{c}_p + \sum_{\langle \mathtt{ret}, \mathtt{c}\rangle} \mathtt{c} + \mathtt{rsum}$$

$$S2 \equiv \forall \langle \bullet, \mathtt{c}\rangle \text{ in Transit gilt } \mathtt{c} > 0$$

$$S3 \equiv \forall p \in \mathcal{P} : (\mathtt{state}_p = \mathtt{passive} \Rightarrow \mathtt{c}_p = 0) \wedge (\mathtt{state}_p = \mathtt{active} \Rightarrow \mathtt{c}_p > 0)$$

Aus **Regel 1** ($\mathtt{rsum} = 0$) und der Invarianten folgt **Term**. Damit folgt die Safety-Eigenschaft.
Für die Lifeness gilt: Habe der Basisalgorithmus eine terminale Konfiguration erreicht.
Dann sind alle Prozesse passiv ($\sum_{p \in \mathcal{P}} \mathtt{c}_p = 0$) und keine Basisnachrichten unterwegs ($\sum_{\langle \bullet, \mathtt{c}\rangle} \mathtt{c} = 0$).
Da p_0 passiv ist empfängt er alle Nachrichten `<ret,c>`.
Deren Anzahl verringert sich mit jedem `receive`.
Ist sie Null, so muß wegen $S1$ $\mathtt{sum} = 1$ sein und p_0 führt `announce` aus. □

Bemerkung:

Die Implementierung basiert auf der Beobachtung, daß alle Werte $\mathtt{c} > 0$ die Form 2^{-i} haben. Um die Anzahl der übertragenen Bits zu reduzieren bietet sich die Repräsentation durch den negativen Logarithmus $[2^{-i}] := -\log_2(2^{-i}) = i$ an. Dann ist $[2^{-i}/2] = [2^{-(i+1)}] = i+1$.

5.3.2 Dezentraler Basisalgoritmus

Wir treffen hier die zusätzliche Annahme, daß die gerichtete Topologie einen unidirektionalen Ring als Subtopologie enthält. Jeder Prozeß besitze eine Variable `next`, die den Nachfolger im Ring angibt.

Unter diesen Gesichtspunkten läßt sich der Algorithmus von SAFRA (87) einsetzen, der auf dem wiederholten Ausführen eines Wellenalgorithmus (hier Ring-Algorithmus) basiert.
Am Ende einer Welle wird entweder Terminierung erkannt oder eine neue Welle gestartet.

Idee: Sei B die Anzahl der Nachrichten, die sich in den Kanälen befinden.
Terminierung liegt genau dann vor, wenn $(\forall p \in \mathcal{P} : \mathtt{state}_p = \mathtt{passive}) \wedge B = 0$ gilt ($\equiv$ **Term**).
Zur verteilten Berechnung von B erhält jeder Prozeß einen Nachrichtenzähler $\mathtt{mc}_p$, so daß immer $P_m \;\equiv\; B = \sum_{p \in \mathcal{P}} \mathtt{mc}_p$ gilt.

Unter der Voraussetzung, daß am Anfang der Berechnung $\forall p \in \mathcal{P} : \mathtt{mc}_p = 0$ gilt und die Prozesse die folgende Regel M beachten, ist P_m eine Invariante des Algorithmus'.

- **Regel M**: Sendet ein Prozeß eine Nachricht, inkrementiert er `mc`; empfängt er eine Nachricht, dekrementiert er `mc`.

Um die Terminierung zu erkennen, sendet ein ausgezeichneter Prozeß p_0 ein Token entlang des unidirektionalen Rings. Mit dem Weitersenden des Tokens werden auch die Nachrichtenzähler $\mathtt{mc}_p$ der einzelnen Prozesse addiert.

Neben Regel M, müssen weitere Regeln dabei eingehalten werden.

- **Regel 1**: Nur ein passiver Prozeß p darf das Token bearbeiten. Hat p die Summe s empfangen, dann sendet er $s + \mathtt{mc}_p$ an seinen Nachbarprozeß $\mathtt{next}_p$ weiter.

Folgendes Problem tritt nun auf: Ein Prozeß, der noch nicht vom Token erreicht wurde, kann eine Nachricht an einen Prozeß senden, der bereits an der Summierung teilgenommen hat. Dies resultiert evtl. in einer falschen Summe.
Eine korrekte Summierung findet sicherlich dann statt, wenn alle Prozesse beim Starten der Terminierungserkennung passiv sind und während der Summenbildung auch passiv bleiben.

Um zu erkennen, ob eine Terminierungserkennung erfolgreich verlaufen war, "färben" wir die Prozesse schwarz oder weiß.

- **Regel 2**: Ein Prozeß p, der eine Basisnachricht empfängt, wird schwarz gefärbt (p wird aktiv).

- **Regel 3**: Leitet ein schwarzer Prozeß das Token weiter, wird das Token schwarz gefärbt.

Eine Summierung endet erfolglos, wenn entweder p_0 ein schwarzes Token zurückerhält oder ein weißes Token mit einer Summe $s \neq 0$.

- **Regel 4**: Endete die Welle erfolglos, startet p_0 eine neue Welle.

- **Regel 5**: Nach Weiterreichen des Tokens wird ein Prozeß weiß.

Algorithmus von SAFRA:

```
int     s, recStop = 0, mc = 0;
Pid     q, next;
boolean firstSent = false;
enum {white, black}    color;
enum {passive, active} state;

  if (state == active) color = black;
  else color = white;
  while (1)
    switch (state) {
      case active : select {
                      choose q;
                      send(q,•);
                      mc++;    /* Regel M */
                      }
                    or
                      state = passive;
                    break;
      case passive: if (own_pid == p0)
                      if (!firstSent) {  /* erste Welle initiieren */
                        send(next, <white,mc>);
                        color = white;   /* Regel 5 */
                        firstSent = true;
                        }
                    receive(-1,&msg);
                    switch (msg) {
                      case  • : mc--;  /* Regel M */
                                color = black;  /* Regel 2 */
                                state = active;
                                break;
                      case black: if (own_pid == p0)
                                    send(next,<white,mc>) /* Regel 4 */
                                  else
                                    send(next,black);  /* Regel 3 */
                                  color = white;   /* Regel 5 */
                                  break;
                      case <white,s>: if (own_pid == p0)
                                        if (s == 0)
                                          announce();
                                        else   /* Regel 4 */
                                          send(next,<white,mc>);
                                      else
                                        if (color == black)
                                          send(next,black); /* Regel 3 */
                                        else   /* Regel 1 */
                                          send(next,<white,s+mc>);
                                        color = white;  /* Regel 5 */
                                      break;
```

```
            case stop: recStop++;
                       announce();
                       if (recStop == |pred|)
                         halt;
            }  /* switch (msg) */
}  /* switch (state) */
```

Bemerkungen:

- Komplexität des Algorithmus' kann beliebig schlecht sein. Es gibt nämlich keine Garantie, daß zwischen zwei Summierungen der Basis-Algorithmus auch nur eine Nachricht versandte oder empfing. In der Praxis dies aber nie vor.
- Anstelle eine unidirektionalen Rings können wir auch jede andere Topologie benutzen, die verteilt summieren kann und bei der die Menge der Starter gleich der Menge der Summierer ist (hier war das p_0).

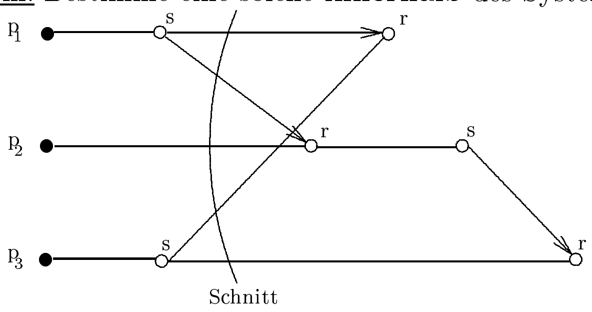
Kapitel 6

Schnappschüsse

Ziel: Berechnung einer **legalen** Konfiguration eines verteilten Systems, d.h. eines Schmittes durch den aktuellen Trace.

Problem: Bestimme eine solche **innerhalb** des Systems

Trace:



Schnitt: Teilmenge von Kanten, so daß der Trace

- in zwei Zusammenhangskomponenten zerfällt
- nicht zwei voneinander abhängige Kanten enthält
- keine Kante von rechts nach links enthält

Grundlegende Methode:

- jeder Prozeß speichert eigenen Zustand
- dann werden Zusatzbedingungen überprüft

Anwendungen

- Feststellen von Systemeigenschaften, etwa Eigenschaft P heißt stabil, falls $\gamma_1 \rightarrow \gamma_2$ impliziert $P(\gamma_1) \rightarrow P(\gamma_2)$.
  Beispiele: Terminierung, Deadlock, Verlust von Token, ...
- Falls ein Fehler auftritt, braucht das System nicht wieder in die Anfangskonfiguration zurückgesetzt zu werden $\rightarrow$ Schnappschüsse in regelmäßigen Abständen.
- Debugging: Um die Fülle der Daten zu reduzieren wird man von ausgewählten Zeitpunkten Schnappschüsse nehmen, die (hoffentlich) informativ genug sind, um die Ursache des (Fehl-) Verhaltens zu bestimmen.

6.1 Rekonstruktion der Kanalinhalte μ^*

1. Lai, Yang:
   q erhält Preshot–Nachricht eventuell noch lange, nachdem erste Preshot–Nachricht eintraf.

   - Lösung 1: Sobald p Schnappschuß aufgenommen hat, sendet p an q in Kontrollnachricht $\sharp$`sent` $p^*[q]$ (Anzahl der an q gesendeten Preshot–Nachrichten)
     $\Rightarrow$ q hat mitgezählt, wieviel Preshot–Nachrichten von p eingetroffen sind
     $\Rightarrow$ weiß deshalb, auf wieviele es noch warten muß
   - Lösung 2: Fasse System nach Aufnehmen des Schnappschusses auf als Basisalgorithmus für Termination Detection, wobei alle postshot–Nachrichten ignoriert werden. Termination Detection erkennt, wann keine Preshot–Nachrichten mehr unterwegs sind.

6.1.1 Rekonstruktion der abzuspeichernden Information

1. Chandy, Lamport: benötigt weder `sent[]` – noch `received[]` –Datenstruktur
2. Lai, Yang (Lösung 1): benötigen nur **Anzahlen** der von p an q gesendeten bzw. der von q von p empfangenen Nachrichten

In Abhängigkeit von der interessierenden Systemeigenschaft muß eventuell bei lokalen Schnappschüssen auch nur kleiner Teil des aktuellen Zustandes gespeichert werden.

6.1.2 Anwendung: Erkennen stabiler Systemeigenschaften

Zur Erinnerung: Die Eigenschaft P heißt *stabil*, falls für $\gamma \to \gamma'$ stets gilt: $P(\gamma) \Rightarrow P(\gamma')$.

Generelle Methode:

```
bool detect = false;
do {
  global_SS();  /* bestimme verteilt γ* */
  test_SS();  /* teste γ* verteilt, ob Eigenschaft P gilt */
              /* falls ja: detect = true */
} while (!detect);
```

So nur sinnvoll, sofern **alle** `global_SS()` initieren bzw. **alle** bei `test_SS` entscheiden und `detect` setzen. Andere Schnappschuß–Algorithmen (z.B. zentralisierte können leicht angepaßt werden.

Satz:

1. Falls System Konfiguration γ erreicht mit $P(\gamma)$, dann erkennt Algorithmus dieses endlich viele Schritte später.
2. Falls Algorithmus $P(\gamma^*)$ feststellt, hat System auch eine Konfiguration erreicht mit: $P(\gamma)$ gilt.

Beweisskizze:
Sei γ_s Konfiguration nach dem ersten Ereignis, bei dem ein lokaler Schnappschuß aufgenommen

wurde. Analog sei γ_f die Konfiguration nach dem lezten Ereignis, bei dem ein lokaler Schnappschuß aufgenommen wurde. Dann zeigt man:

$$\gamma_s \xrightarrow[\text{(a)}]{*} \gamma^* \xrightarrow[\text{(b)}]{*} \gamma_f$$
$$(a) \Rightarrow (1) \text{ und } (b) \Rightarrow (2)$$

Implizite Vorausetzung: Fairness, d.h. weder der Basisalgorithmus noch der Kontrollalgorithmus führt unendlich lange Berechnungen aus, ohne den jeweils anderen Algorithmus ein Ereignis ausführen zu lassen.

Bleibt für verschiedene stabile Eigenschaften P möglichst effiziente Implementierungen von `test_SS()` zu berechnen.

Generelle Methode:

- akkumuliere gesamte Konfiguration γ^* (bzw. deren relevanten Anteil) in einem Prozeß p_0.
- p_0 führt Test aus; initiiert gegebenenfalls nächsten Schnappschuß

Nachteil:

- viele Nachrichten werden gesendet
- Test selbst ist sequentiell

6.2 Terminierung

Relevanter Anteil der Konfiguration, den der Test berücksichtigen muß:

- $state^* \in \{$ active, passive$\}$
- $\sharp\mu^*(p,q)$, diese ist in Prozeß q vorhanden

Jeder Prozeß q legt Bool *quiet* an, mit $quiet_q = 1 \Leftrightarrow (state_q{==}\text{passive} \;\&\&\; \sum_p \mu^*(p,q) = 0)$. Zum Testen auf Terminierung genügt es, verteilt das *Infimum* der Variablen *quiet* zu berechnen $\Longrightarrow$ Wellenalgorithmus.

6.3 Deadlock–Detection

Modell des zugrunde liegenden Basisalgorithmus':

Algorithmus:

```
enum {active,blocked} state = active;
int  v = 0;   // Request No
Set_of_processes Reqs,   // Request Set
                 Grant;  // Received Grants
bool Free; // Freeing constraint
while (1) {
  switch (state) {
    case active:
      select {
        Reqs = choose S();
        Free = choose P();
        v++;  // neue Req. No
        forall (q ∈ Reqs)
        send(q,<req,v>);
        Grant = 0;
        state = blocked;
      } or {
        receive(&q,<req,&v'>);
        send(q,<grant,v'>);
      }
      break;
    case blocked:
      receive(&q,<grant,v>);
      // Grant zu gegenwaertigem Request erforderlich!
      Grant = Grant ∪ {q}
      if (Free(Grant)) state = active;
    } // end switch
  } // end while
```

Free ist ein beliebiges Prädikat mit:

(D1) $Free_p(a) ==$ false und $Free_p(Reqs_p) ==$ true;

(D2) $Free_p(A) ==$ true und $A \in B \Rightarrow Free_p(B) ==$ true

Definition:
Ein Prozeß p ist *lebendig* (alive) in γ, sofern es eine Konfiguration $\gamma = (\sigma', \gamma')$ gibt mit $\gamma \rightarrow \gamma'$, sodaß in $\sigma'(p)$ state==active.
Andernfalls heißt p tot.
γ heißt *Deadlock* (DL), sofern es einen in γ toten Prozeß p gibt. Offenbar ist DL eine stabile Eigenschaft.

Ein DLD (Deadlock Detection) Algorithmus erfüllt die 3 Eigenschaften:

I Non–Interference: Basisalgorithmus wird nicht gestört

L Lifeness

S Safety

Wir nehmen an, ein Schnappschuß liege vor und liefere in jedem Prozeß p Variablen $state^*_p$, v^*_p, $Reqs^*_p$, $Grant^*_p$, $Free^*_p$, $\mu^*(q,p)$.
Verteilter Test auf Deadlock: *Der Global-Marking-Algorithmus* nimmt pessimistisch an, alle Prozesse wären tot, d.h. die Menge der lebendigen Prozesse ist leer. Dann werden dieser Menge sukzessiv Prozesse hinzugefügt, die aktiv werden können. Übrig bleiben die „hoffnungslosen Fälle“.

Algorithmus:

```
bool alive = false;
// sichere Grants
Set_of_processes GrRec = Grants ∪ {q | ⟨Grant,v*⟩ ∈ μ*(q,pid)};
while (1) {
  if (!alive && state == active || Free(GrRec)) {
    alive = true;
    forall (q with <rec,-1> ∈ μ*(q,pid)) send(q,alive);
  } else {
    receive(&q,alive);
    if (!alive) {
      GrRec = GrRec ∪ {q};
      if (Free*(GrRec)) {
        alive = true;
        forall (q with <rec,-1>∈ μ*(q,pid)) send(q,alive);
        }
      }
    }
  } // while
```

Satz:
Der Global-Marking-Algorithmus terminiert. Bei Terminierung gilt:
$\texttt{alive}_p$ `== true` $\Leftrightarrow$ p ist lebendig in γ^*

Beweis:

- Jeder Prozess setzt `alive` max. einmal auf `true` und versendet deshalb max. einmal eine `alive`-Nachricht an alle Nachfolger.
  $\Rightarrow$ max. |E| Nachrichten
  $\Rightarrow$ Algorithmus terminiert.

- Algorithmus simuliert eine Berechnung, bei der jeder aktive Prozeß sofort alle Anforderungen, die an ihn gestellt werden, erfüllt.
  $\Rightarrow$ Alle Prozesse, die $\texttt{alive}_p$ `= true` setzen, sind lebendig.

- p alive in γ^*
  $\Rightarrow$ Es ex. Berechnungsfolge $\gamma^* = \gamma_0 \rightarrow ... \rightarrow \gamma_m$, wobei p in γ_m aktiv ist.
  Sei m minimal mit dieser Eigenschaft.

  Induktion nach m:

  - m = 0: trivial
  - m > 0: m minimal $\Rightarrow$ p nicht aktiv in $\gamma_0, ..., \gamma_{m-1}$
    $\Rightarrow$ Ereignis vor γ_{m-1} war Empfang eines Grants <Grant,v^*_p >, so daß anschließend $\texttt{Free}^*_p(G)$ erfüllt ist, wobei
    G: Menge der von p erhaltenen Zusicherungen für v^*_p

    Z.z.: G $\subseteq$ `GrRecs`

Sei q $\in$ G.

- $*$ Fall 1: <Grant,v_p^* >$\in \mu^*$(q,p)
  $\Rightarrow$ q wird von Algorithmus in `GrRecs` eingefügt.
- $*$ Fall 2: <Grant,v_p^* > wird in einem Ereignis der Berechnung
  $\gamma_0 \rightarrow ...\gamma_m$ abgeschickt.
  $\Rightarrow$ q aktiv in einem γ_j, j<m
  $\Rightarrow$ $\texttt{alive}_q$ wird von Algorithmus auf `true` gesetzt.
  $\Rightarrow$ q sendet p eine `alive`-Botschaft.
  $\Rightarrow$ p fügt q in `GrRecs` ein.

$\Rightarrow$ Behauptung

Bemerkung:
Oft werden eingeschränkte Modelle betrachtet:

- AND-Modell: **Alle** Requests müssen erfüllt werden
  ($\rightarrow$ Resource-Allokation).
- OR-Modell: **Mindestens ein** Request mußerfüllt werden
  ($\rightarrow$ Verklemmungen in verteilten Systemen).

Sie gestatten spezialisierte Algorithmen. In den beiden Spezialfällen ex. nämlich einfache graphentheoretische Charakterisierungen mit Hilfe des *Wait-For-Graphen* W.
Für γ^* def. wir W als den folgenden gerichteten Graphen:

- Knoten: p $\in$ P
- Kanten: (p,q) gdw. q $\in$ $\text{Reqs}_p^* \backslash \text{GrRec}_p^*$ und kein <Grant,v_p^* >$\in \mu^*$(q,p)

Deadlocks:

- AND-Modell: Deadlock $\Leftrightarrow$ W enthält einen Kreis (`circular wait`).
- OR-Modell: Deadlock $\Leftrightarrow$ W enthält `Knoten` , d.h. starke Zusammenhangskomponente ohne ausgehende Kanten.

Kapitel 7

Fehlertolerante Systeme

Zwei Ansätze:

- Robuste Algorithmen führen nur Schritte aus, wenn sie (hinreichend) sicher sind, keinen Fehler zu machen.
- Stabilisierende Algorithmen werden von Fehlern beeinflußt, der Algorithmus garantiert allerdings "Erholung", d.h. angestrebter Endzustand wird von bel. Konfiguration erreicht.

Letztere sind gut geeignet, wenn Fehler nur zeitweilig auftritt.
Erstere helfen gegen **permanente** Ausfälle.

Fehlermodelle für robuste Algorithmen:

- Crash: Ein Prozeß führt Programm eine Weile aus, um dann keinen weiteren Schritt zu tun.
- Byzantinisches Verhalten: Ein Prozeß führt Programm bis zu best. Zeitpunkt aus, um danach **beliebige** Schritte auszuführen, die bel. "bösartig" sind.

Entscheidungsprobleme:

Ziel: Jeder intakte Prozeß schreibt Entscheidung auf Ausgabe.
Diese Ausgabe hängt in relativ trivialer Weise von einer Eingabe ab.
Die Entscheidungen müssen "zusammenpassen".
(In sicherer Umgebung Wellen anwendbar.)
Präzisierte Anforderungen an Entscheidungsalgo:

- T: Terminierung
- C: Konsistenz
- NT: Nichttrivialität; in versch. Berechnungen versch. Entscheidungen möglich

Anwendungen:

- Verteilte Datenbanken: Commit/Abort
- Replizierte Berechnung: Consens über Input
- Leader Election
- Approximative Übereinstimmung

7.0.1 Consens

Definition:

- Eine *t-Crash-robuste faire Berechnung* ist eine Berechnung, bei der mindestens $\#P - t$ Prozesse unendlich viele Berechnungsschritte ausführen und jede Nachricht an einen korrekten Prozeß (d.h. einer, der unendlich lange rechnet) empfangen wird.
- Ein *Consens–Algorithmus* ist ein Entscheidungs–Algorithmus mit Konsistenzbed. *Gleichheit.*
- Ein Consens-Algorithmus heißt *t-Crash-robust*, falls in einer t-Crash-fairen Berechnung alle korrekten Prozesse entscheiden.

Satz:
Es gibt keinen deterministischen 1-Crash-robusten Consens-Algorithmus.

7.0.2 Consens-Algorithmus von Bracha und Toueg (1985)

Topologie sei Clique mit N Prozessen.

Idee:

- Organisiere in Runden.
- In jeder Runde sendet jeder Prozeß an jeden eine Botschaft.
- Botschaft enthält Votum $\in \{0,1\}$ und Gewicht = # erhaltener Voten für diesen Wert in letzter Runde.
  Votum mit Gewicht > N/2 heißt *Zeuge*
  $\Rightarrow$ In einer Runde gibt es max. einen Zeugen.
- In nächster Runde schließt sich Prozeß p einem Zeugen an (falls solcher eintraf), andernfalls der Mehrheit.
- Nach Ankunft von > t Zeugen entscheidet p.

Formal:

```
{0,1} value = x;        // input
int round = 0;
int msgs[2],witness[2];
int y = -1;             // output
int weight = 1;

while (y < 0) {
  msgs[0] = msgs[1] = 0;
  witness[0] = witness[1] = 0;
  forall (q in P)
    send(q,<round,value,weight>),
  while(msgs[0] + msgs[1] < N - t) {
    receive(-1,<&r,&v,&w>);
    if (r > round)
      send(pid,<&r,&v,&w>);
    elseif (r == round) {
      msgs[v]++;        // count votes
      if (w > N / 2)
        witness[v]++;
    }
    else ;              // r < round
  }                     // receive finished -> determine new vote!
  if (witness[0] > 0)
    value = 0;
  if (witness[1] > 0)
    value = 1;
  if (msgs[0] > msgs[1])
    value = 0;
  else
    value = 1:
  weight = msgs[value];
  if (witness[value] > t)
    y = value;          // decide
  round++;
}                       // end while (y < 0)
forall (p in P) {
  send(p,<round,value,N - t>);
  send(p,<round + 1,value,N + t>);
}
```

<u>Es gilt:</u>

1. In keiner Runde gibt es Zeugen für verschiedene Werte v_1 und v_2
   <u>Beweis</u>
   Annahme: es gäbe sie:
   $\rightarrow$ in Runde davor votierten $> N/2$ Prozesse für v_1 und $> N/2$ Prozesse für v_2
   $\rightarrow$ Widerspruch

2. Falls gilt Prozeß p entscheidet für v, dann entscheidet sich auch jeder andere korrekte Prozeß maximal 2 Runden später für v

Beweis
Sei k erste Runde, in der ein Prozeß sich für v entscheidet
$\rightarrow$ es gab mehr als t Zeugen in Runde k
$\rightarrow$ alle korrekten Prozesse erhielten mindestens einen Zeugen für v in Runde k
$\rightarrow$ in Runde k+1 senden alle korrekten Prozesse v
$\rightarrow$ in Runde k+2 werden nur Zeugen für v versandt
$\rightarrow$ alle entscheiden sich für v
Annahme über Wahrscheinlichkeiten
Sei $R(p,a,k)$ = Ereignis, daß q die Nachricht von p in Runde k entgegennimmt (d.h. die Nachricht sich unter den ersten N-t Botschaften für q befindet
Wir nehmen an:

(a) $\exists\ \epsilon < 0$: $\forall p, q, k : ProbP(p,q,k) \leq \epsilon$

(b) alle Ereignisse sind unabhängig

3. Prob (Algorithmus terminiert nicht) = 0
Beweis Sei S Menge von N-t korrekten Prozessen
Annahme
$\rightarrow \exists$ Prob $\varphi > 0$: In einer Runde k empfangen alle Prozesse aus S nur Nachrichten von Prozessen aus S
$\rightarrow$ mit Prob $\geq \varphi^3$ empfangen Prozesse aus S in den Runden $k_0, k_0{+}1, k_0{+}2$ nur Nachrichten aus S
$\rightarrow$ in k_0 erhalten alle Prozesse aus S die gleiche Menge von Voten
$\rightarrow$ alle voteieren für gleiches v
$\rightarrow$ in Runde $k_0 + 1$ werden nur Voten für v versendet
$\rightarrow$ in Runde $k_0 + 2$ erhält jeder Prozeß in S N-t Zeugen für v
$\rightarrow$ alle Prozesse in S entscheiden sich nach Runde $k_0 + 2$ für v
Prob(Prozesse in S haben nach Runde $k_0{+}2$ nicht entschieden) $\leq (1-\varphi^3)$ * Prob(Prozesse in S haben nach Runde k_0 noch nicht entschieden)
$\rightarrow$ Behauptung.

Bemerkung:

1. Es gibt keinen t-crash robusten Konsens-Algorithmus für $t \geq N/2$.

2. Eine Modifikation des vorliegenden Alogorithmus ist auch in der Lage t < N/3 Byzantinische Fehler zu tolerieren (Auch diese Schranke ist scharf).