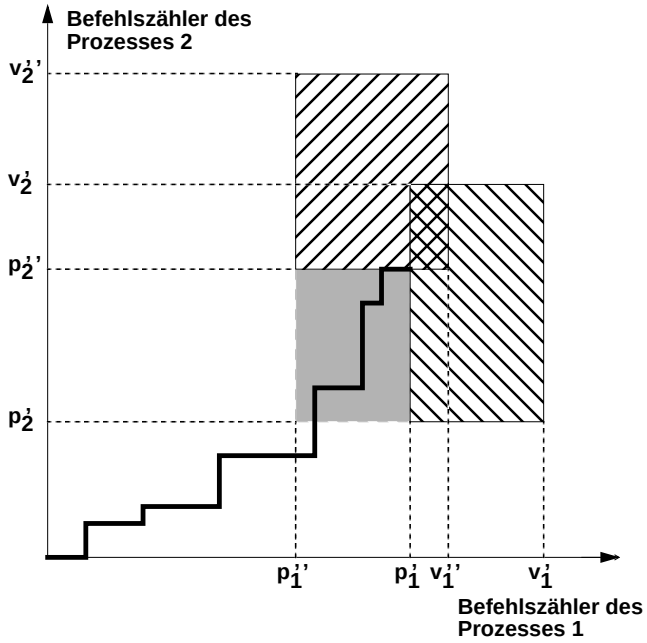
10 Verteilte Verklemmungserkennung

10.1 Die Fragestellung



Für die Existenz einer (partiellen) Verklemmung notwendige Bedingungen

- **Exklusive Betriebsmittelbelegung.**
- **Betriebsmittel können nachgefordert werden.**
- **Betriebsmittel können nicht entzogen werden.**
- **Existenz eines Ringes von Prozessen, in dem jeder Prozeß auf Betriebsmittel wartet, die der im Ring folgende Prozeß belegt hat.**

Problemkreise

- **Vermeidung von Verklemmungen (durch das Programm, prevention)**
- **Verhinderung von Verklemmungen (durch das Betriebsystem, avoidance)**
- **Erkennung von Verklemmungen**
- **Erholung aus Verklemmungen**

Verteilte Verklemmungserkennung: Fragestellung

Verhinderung (Banker's Algorithm)

Erkennung (Betriebsmittelgraph)

Vermeidung

- **alles sofort**
- **vor neuer Anforderung vollständige Freigabe**
- **lineare Anordnung der Betriebsmittel**

Verteilte Verklemmungserkennung: Erkennung von Verklemmungen

Erkennung von Verklemmungen

Wartegraphen

- **Knoten: Prozesse**
- **Kanten: Es führt eine Kante von Prozeß P_1 zu Prozeß P_2 genau dann, wenn P_1 seine nächste Aktion nur ausführen kann, nachdem P_2 eine Aktion ausgeführt hat.**

Ein System ist genau dann (partiell) verklemmt, wenn der Wartegraph einen Zyklus enthält.

Organisation der Verklemmungskontrolle

- **Zentralisiert: Ein Koordinator übernimmt den Aufbau und die Untersuchung des Wartegraphen.**

  Vorteil: Konzeptuell einfach

  Nachteil: Engpaßgefahr in der Netzumgebung des Koordinators.

- **Verteilt: Verantwortung auf alle gleichmäßig aufgeteilt.**

  Vorteil: Gleichmäßige Netzbelastung.

  Nachteil: Algorithmen schwierig, Inkonsistenz der gesammelten Daten könnte fälschlicherweise zur Feststellung einer (vermeintlichen) Verklemmung führen.

- **Hierarchisch: Knoten werden in einem Baum angeordnet, jeder Knoten untersucht Verklemmungen, die nur seinen Teilbaum betreffen.**

10.3 **Modelle**

Betriebsmittelmodell

- **Betriebsmittelgraph**
  - **Komponenten: Prozesse P_i, Betriebsmittelarten BM_k**
  - **Relationen**

    P fordert n Exemplare einer Betriebsmittelart BM an

    An P sind n Exemplare einer Betriebsmittelart BM zugewiesen
  - **Fortsetzbedingung**

    Ein Prozeß kann erst fortgesetzt werden, wenn alle seine momentanen Anforderungen erfüllt werden.
- **Wartegraph**
  - **Komponenten: Prozesse**
  - **Relation: P_i wartet auf P_j**
  - **Fortsetzbedingung**

    Ein Prozeß P kann erst dann weiterarbeiten, wenn alle Prozesse, auf die er wartet, weiterarbeiten können.

Kommunikationsmodelle

- **1-aus-q-Modelle**
  - Komponenten: Prozesse
  - Relation: P_i wartet auf Nachricht von P_j
  - Fortsetzbedingung

    Ein Prozeß P kann genau dann weiterarbeiten, wenn ihm einer der q Prozesse, von denen er eine Nachricht erwartet, eine Nachricht gesandt hat. Mit dem Eintreffen einer erwarteten Nachricht sind alle seine Erwartungen erledigt.

- **p-aus-q-Modelle**
  - Komponenten: Prozesse
  - Relation: P_i wartet auf Nachricht von P_j
  - Fortsetzbedingung

    Ein Prozeß P kann genau dann weiterarbeiten, wenn ihm p(P) von den q Prozessen, von denen er eine Nachricht erwartet, eine Nachricht gesandt haben. Mit dem Eintreffen der nötigen Zahl von Nachrichtn sind alle seine Erwartungen erledigt.

10.4 Zentralisierte Algorithmen

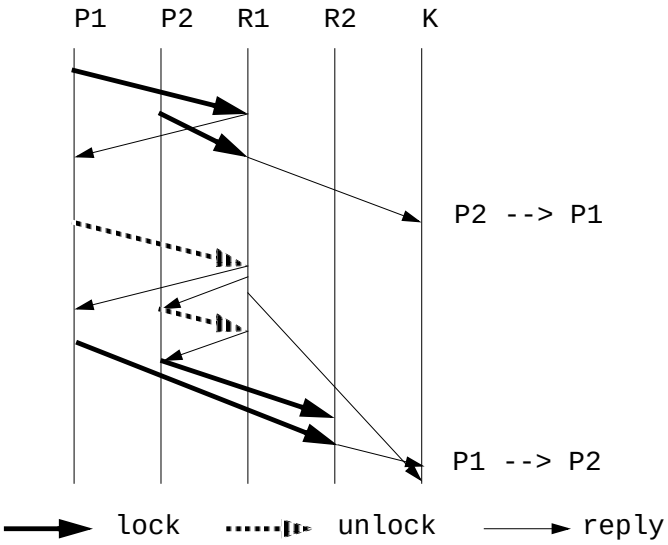
Vorgehensweise: Jeder Knoten teilt dem Koordinator seine Anforderungen und Freigaben mit so, wie sie entstehen.

Probleme

- **Starke Netzbelastung.**
- **Wegen der variierenden Nachrichtenübertragungszeiten und dem Fehlen einer exakten Zeitsynchronisation kann es zu Phantom-Verklemmungen kommen.**

Beispiel:

P1	P2
lock R1	lock R1
unlock R1	unlock R1
lock R2	lock R2
unlock R2	unlock R2



10.5 Algorithmus von Chandy-Misra-Haas für das Betriebsmittelmodell

Voraussetzung: Je zwei Knoten sind durch einen einzigen zuverlässigen FIFO-Kanal verbunden

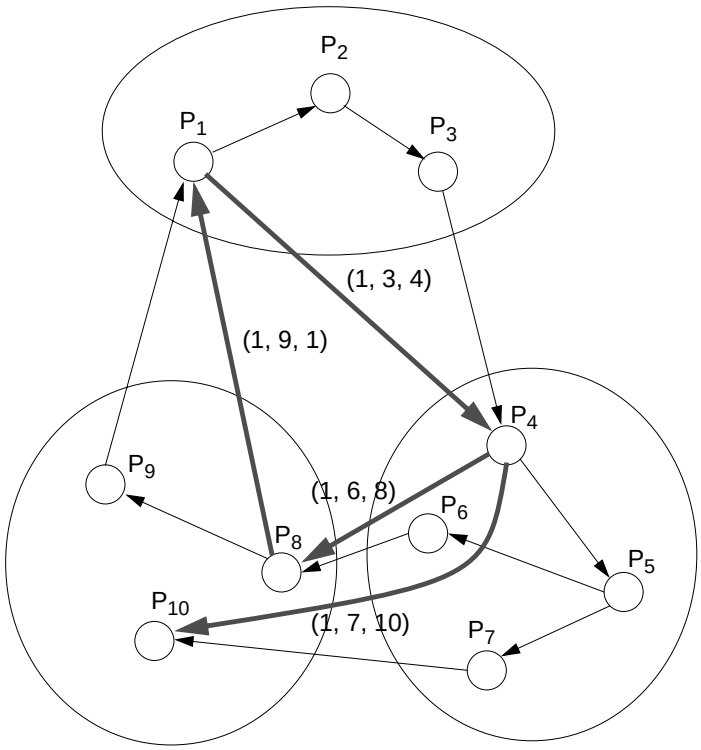
Zu Beginn: Für alle i, k ist abhängig(i, k) == false.

Start des Algorithmus durch P_i

Existiert lokaler Zyklus, der P_i enthält?	
ja	nein
verklemmt	für alle P_j und P_k mit 1. es ex. Pfad von P_i nach P_j 2. es ex. Pfad von P_j nach P_k 3. P_j und P_k sind nicht auf dem gleichen Knoten sende test(i, j, k) an den Knoten von P_k

Bei Empfang von test(i, j, k)

P_k blockiert und abhängig(k, i) == false und P_k hat nicht alle Anfragen von P_j beantwortet?	
ja	nein
abhängig(k, i) = true	
k == i ?	
ja	nein
P_i verklemmt	für alle P_m und P_n mit 1. es ex. Pfad von P_k nach P_m 2. es ex. Pfad von P_m nach P_n 3. P_m und P_n sind nicht auf dem gleichen Knoten sende test(i, m, n) an den Knoten von P_n



10.6

Algorithmus von Chandy-Misra-Haas für das 1-aus-q Kommunikationsmodell

Voraussetzung: Nachrichtenkanäle sind zuverlässige FIFO-Kanäle!

Datenstrukturen

Prozeß P_k:

DS_k	Menge aller Prozesse, von denen P_k eine Nachricht erwartet
$latest_k[i]$	Größte Sequenznummer einer empfangenen Nachricht, mit P_i als Initiator
$engager_k[i]$	Identität des Prozesses, dessen Nachricht die letzte Zuweisung an $latest_k[i]$ veranlaßte
$num_k[i]$	Zahl der zu $latest_k[i]$ gehörigen ausgesandten query-Nachrichten minus der Zahl der empfangenen reply-Nachrichten
$wait_k[i]$	Ist true genau dann, wenn P_k seit der letzten Zuweisung an $latest_k[i]$ ständig blockiert war.

Nachrichtenformat

Nachrichtentyp(Initiator, Sequenznummer, Sender, Empfänger)

Die Sequenznummer gibt an, zur wievielten von Initiator veranlaßten Abwicklung des Algorithmus die Nachricht gehört

Initiator sei ein passiver Knoten P_i

```
begin
   latest[i]++;
   wait[i] = true;
   for all j ∈ DS_i send query(i, latest[i], i, j);
   num[i] = |DS_i|;
end
```

Passive Prozesse P_k setzen bei ihrer Aktivierung `active = true` und für alle Indizes i `wait[i] = false`. Während ihrer aktiven Phase verwerfen sie alle Kontrollnachrichten.

Verteilte Verklemmungserkennung: Chandy-Misra-Haas

Bei Empfang von query(i, m, j, k) durch passiven Knoten P_k

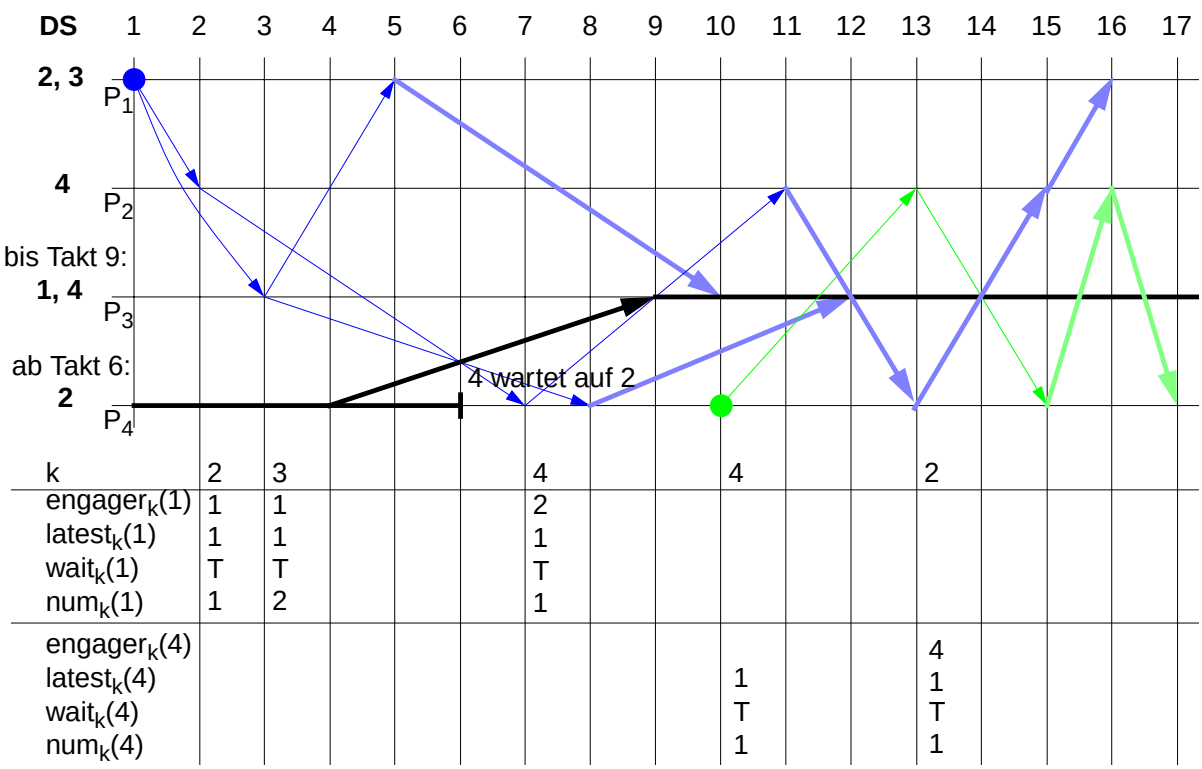
```
if (!active) {
   if (m > latest[i]) {
      latest[i] = m;
      engager[i] = j;
      wait[i] = true;
      for all r ∈ DS_k send query(i, m  k, r);
      num[i] = |DS_k|;
   } else if (wait[i] and m == latest[i])
      send reply(i, m, k, j);
}
```


Verteilte Verklemmungserkennung: Chandy-Misra-Haas

Bei Empfang von reply(i, m, r, k) durch P_k

```
if (!active) {
   if (m == latest[i] && wait[i]) {
      num[i]--;
      if (num[i] == 0) {
         if (i == k)
            declare_P_k_deadlocked();
         else send reply(i, m, k, j) where j == engager[i]
      }
   }
}
```

Verteilte Verklemmungserkennung: Chandy-Misra-Haas



Verteilte Verklemmungserkennung: Algorithmus von Bracha und Toueg

10.7

Synchrones Modell: Algorithmus von Bracha und Toueg

p-aus-q-Modell

Wartegraph: **Prozessoren als Knoten;**
E Menge der Wartekanten

Nachrichten: **REQUEST** **Anforderung von Betriebsmitteln**
REPLY **Zuteilung des Betriebsmittels**
RELINQUISH **Rücknahme der noch besgtehenden Anforderungen, wenn p Anforderungen erfüllt sind**

Initialisierung für jeden Knoten

```
OUT = {u | (v, u) ∈ E};
IN = {u | (u, v) ∈ E};
notified = false; free = false;
noGranted = 0;
n = <prozesorspezifischer Wert von p>
```

Verteilte Verklemmungserkennung: Algorithmus von Bracha und Toueg

Prozeduren

```
void notify()
{ notified = true;
  for (all w ∈ OUT) send(w, NOTIFY);
  if (n == 0) grant();
  for (all w ∈ OUT) await(w, DONE);
}

void grant()
{ free = true;
  for (all w ∈ IN) send(w, GRANT);
  for (all w ∈ IN) await(w, ACK);
}
```


Verteilte Verklemmungserkennung: Algorithmus von Bracha und Toueg

Wenn v von u eine NOTIFY-Nachricht erhält

```
if (!notified) notify();
send(u, DONE)
```

Wenn v von u eine GRANT-Nachricht erhält

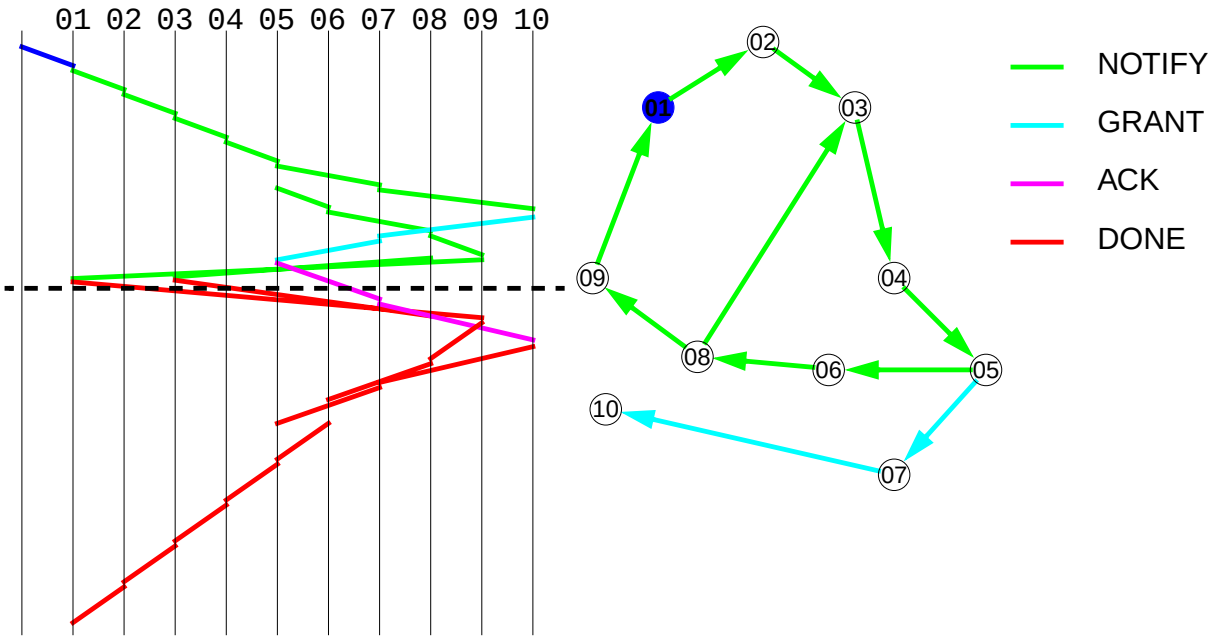
```
noGranted++;
if (!free && noGranted >= n) grant();
send(u, ACK);
```

Initiierung

```
notify();
```

Verklemmungskriterium

$$\texttt{deadlocked} == \texttt{!free}_{\texttt{initiator}}$$



Beweis für die Funktionsfähigkeit des Algorithmus

D10.1 Definition der Hülle eines Prädikats

Gegeben seien

- **eine Menge** $S \subseteq V$ **und**
- **ein Prädikat** $P : V \times \wp(V) \rightarrow \{\text{true}, \text{false}\}$ **mit** $P(v, \varnothing) = \text{false}$ **und** $S_1 \subseteq S_2 \Rightarrow (P(v, S_2) \Rightarrow P(v, S_1))$

Dann ist die Hülle $C(S, P)$ **rekursiv definiert durch**

1. $C(S, P)^0 = S$
2. $C(S, P)^{i+1} = C(S, P)^i \cup \{v \in V | P(v, IN(v) \cap C(S, P)^i)\}$
3. $C(S, P) = \bigcup_{i \in N} C(S, P)^i$

Anschauliches Beispiel:

In einem Kommunikationsgraphen sei $P(v, S)$ **genau dann erfüllt, wenn jeder Knoten in S von v aus über einen Kommunikationspfad erreichbar ist. Dann ist** $C(S, P)$ **die Menge aller Knoten, die von v aus erreichbar sind.**

Algorithmus zur Ermittlung der Hülle

Initialisierung für jeden Knoten

```
OUT = {u | (v, u) ∈ E};
in_C = false; ancestors = {}; lock = 1; // lock is a semaphore
```

Prozeduren

```
void Closure()
{ in_C = true;
  lock.V();
  for (all w ∈ OUT) send(w, NOTIFY);
  for (all w ∈ OUT) await(w, DONE);
}
```

Wenn v von u eine NOTIFY-Nachricht erhält

```
ancestors = ancestors + {u};
lock.P();
if (in_C) lock.V();
else { if (v ∈ S || P(v, ancestors))) Closure();
       else lock.V();
     }
send(u, DONE);
```


L10.1 **Im Laufe einer Ausführung des Hüllenalgorithmus ruft ein Knoten v die Methode `closure` genau dann auf, wenn $v \in C(S, P)$ ist.**

L10.2 **Der Hüllenalgorithmus terminiert und zwar nachdem alle $v \in C(S, P)$ ihren Teil terminiert haben.**

L10.3 **Während der Ausführung des Hüllenalgorithmus werden maximal 2|E| Nachrichten versandt.**

Beweis für die Richtigkeit des Verklemmungsalgorithmus

1. **Man betrachte das Prädikat `ADJ(v, D) = |D| > 1`. Die zugehörige Hülle C({Initiator}, ADJ) besteht aus allen Knoten, die vom Initiator aus erreichbar sind.**

   Dann stellt die notify-Welle die zugehörige Hüllenberechnung dar, indem im Hülenalgoritmus `Closure` ersetzt wird durch `Notify` und `in_C` durch `notified`.

   Dabei ist noch zu beachten, daß `ADJ(v, ancestors)` wegen `|ancestors| > 1` immer erfüllt ist und die zusätzliche Anweisung `if (n == 0) grant()` keinen Einfluß hat.

2. Man betrachte das Prädikat `SAT(v, D) = |D| > n_v` im Graphen $G^T=(V, E^T)$.
   Weiter sei ACTIVE die Menge der vom Initiator aus erreichbaren Knoten, die nicht auf Basisnachrichten warten. Dann ist die Hülle C(ACTIVE, SAT) die Menge der Knoten, die nicht verklemmt sind.

   Ersetzt man im Hüllenalgorithmus `Closure` durch `Grant` und `in_C` durch `free` und `OUT` durch `IN`, so stellt die grant-Welle die Berechnung dieser Hülle dar, weil in der Anweisung

   `if (!free && (v ∈ ACTIVE || SAT(v, ancestors)) grant()`

   der Ausdruck `SAT(v, ancestors)` äquivalent ist zu `#granted > n_v` und der Ausdruck `v ∈ ACTIVE` den Wert `false` hat.

S10.1 Wenn ein Knoten den Algorithmus in einem Wartegraphen G startet, dann terminiert er auch. Der Initiator terminiert mit Zustand $\texttt{free}_{\texttt{initiator}}$ `== true` genau dann, wenn er nicht verklemmt ist.

10.8 Asynchrones Modell mit Nachrichten im Kommunikationssystem, statischer Wartegraph

Kanäle zuverlässig, nicht notwendig FIFO.

Färbung der Kanten (u, v)

- grau: Knoten u hat an v eine REQUEST-Nachricht gesandt, aber die Nachricht ist noch nicht angekommen und u hat an v noch keine RELINQUISH-Nachricht gesandt
- schwarz: Knoten v hat von u eine REQUEST-Nachricht erhalten, v hat an u noch keine REPLY-Nachricht gesandt und u hat keine RELINQUISH-Nachricht an v gesandt
- weiß: Knoten v hat an u eine REPLY-Nachricht gesandt, aber u hat sie noch nicht erhalten und u hat noch keine RELINQUISH-Nachricht an v gesandt.
- transparent: Knoten u hat an v eine RELINQUISH-Nachricht gesandt, aber v hat sie noch nicht erhalten.

Initialisierung für jeden Knoten

```
OUTBlack = {u | (v, u) ∈ E && isBlack((v, u))};
INBlack = {u | (u, v) ∈ E && isBlack((u, v))};
notfied = false; free = false;
noGranted = 0;
```


Prozeduren

```
void notify()
{ notified = true;
  for (all w ∈ OUTBlack) send(w, NOTIFY);
  if (n - noGreyOrWhite <= 0) grant();
  for (all w ∈ OUTBlack) await(w, DONE);
}

void grant()
{ free = true;
  for (all w ∈ INBlack) send(w, GRANT);
  for (all w ∈ INBlack) await(w, ACK);
}
```

Verteilte Verklemmungserkennung: Algorithmus von Bracha und Toueg

Wenn v von u eine NOTIFY-Nachricht erhält

```
if (¬notified) notify();
send(u, DONE)
```

Wenn v von u eine GRANT-Nachricht erhält

```
noGranted++;
if (¬free && noGranted >= n - noGreyOrWhite)
  grant();
send(u, ACK);
```

Initiierung

```
Notify();
```


Verteilte Verklemmungserkennung: Algorithmus von Bracha und Toueg

10.9 Asynchrones System mit dynamisch veränderlichem Wartegraphen

Kanäle zuverlässig und FIFO

Algorithmus kann bestenfalls folgende Eigenschaften besitzen:

1. **Wenn der Initiator beim Start des Algorithmus verklemmt ist, dann wird eine Verklemmung erkannt.**
2. **Wenn eine Verklemmung erkannt wird, dann ist beim Ende des Algorithmus der Initiator verklemmt.**

D10.2 Definition

- **Ein Zeitschnitt heißt konsistent, wenn er nicht zu Nachrichten aus der Zukunft führt.**
- **Für Zeitschnitte S_1 und S_2 ist $S_1 \leq S_2$ genau dann, wenn der Zeitschnitt S_1 vollständig in der Vergangenheit von S_2 verläuft.**

S10.2 Satz

- **Sei $S_1 \leq S_2$ und p sei in S_1 verklemmt, dann ist p auch in S_2 verklemmt.**

Algorithmus

- **Der Initiator sendet an alle Prozesse in $IN \cup OUT$ eine FREEZE-Nachricht.**
- **Wenn ein Prozeß p zum ersten Mal eine FREEZE-Nachricht bekommt, sichert er seinen lokalen Zustand und sendet eine FREEZE-Nachricht an alle Prozesse in $IN_p \cup OUT_p$.**
- **Wenn ein Prozeß p an einen nicht in $IN_p \cup OUT_p$ enthaltenen Prozeß sendet, sendet er vorher eine FREEZE-Nachricht.**
- **Der Initiator leitet nach dem Versenden seiner FREEZE-Nachricht den Algorithmus für das asynchrone Modell mit statischem Wartegraph ein. Dieser Algorithmus arbeitet mit den gesicherten Zuständen (d. h. auf dem dadurch festgelegten, statischen Wartegraphen).**

S10.3 **Satz**

Der Algorithmus werde zum Zeitpunk t_1 gestartet und terminiere zum Zeitpunkt t_2. G_{t_1} und G_{t_2} seien die zugehörigen Wartegraphen.

- **Wenn der Initiator zum Zeitpunkt t_1 verklemmt ist, dann ist zum Zeitpunkt t_2 $free_{initiator}$ = false.**
- **Wenn zum Zeitpunkt t_2 $free_{initiator}$ = false ist, dann ist der Initiator zum Zeitpunkt t_2 verklemmt.**