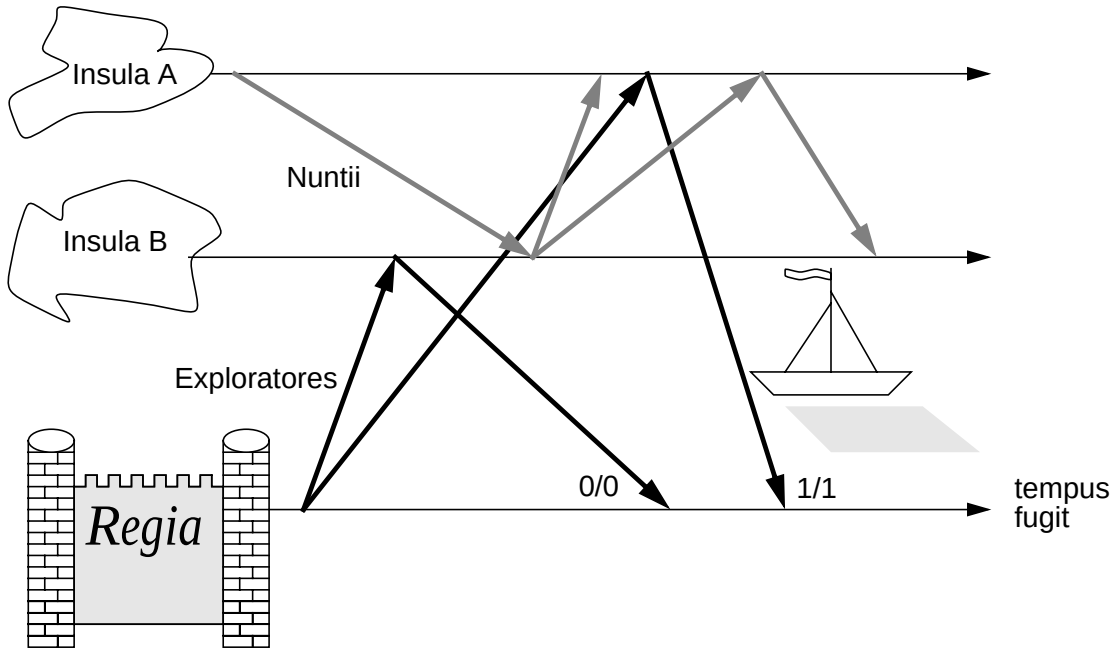
8.1 Die Fragestellung

❑ Das Märchen von der verteilten Terminierung



Verteilte Terminierung: Überblick

- Berechnungen sollen nach folgendem Muster erfolgen:
  1. Prozesse befinden sich stets in einem der Zustände *aktiv* oder *passiv*.
  2. Nur aktive Prozesse können Basisnachrichten aussenden.
  3. Aktive Prozesse können jederzeit passiv werden.
  4. Passive Prozesse können nur durch Empfang von Basisnachrichten in den aktiven Zustand wechseln.

- Interpretationen für *passiv*:

  Typische Interpretationen für *passiv*

  1. Das Programm, das ein Prozeß ausführt, ist beendet.
  2. Der Prozeß befindet sich in einer Endlosschleife, die mit Sicherheit nicht mehr zum Aussenden von Basisnachrichten führt.
  3. Der Prozeß hat seine letzte Basisnachricht versandt, rechnet aber lokal noch einige Zeit weiter.
  4. Der Prozeß wartet blockiert auf den Empfang einer Basisnachricht.

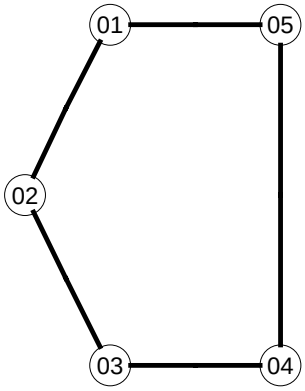
Verteilte Terminierung: Überblick

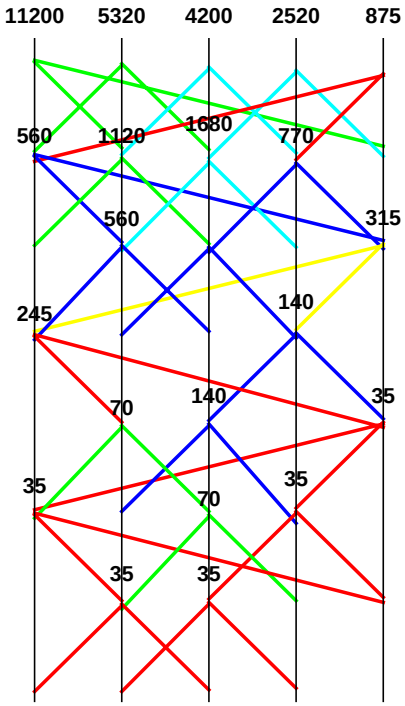
- Eine Berechnung heißt *terminiert*, wenn alle Prozesse passiv sind und keine Basisnachricht unterwegs ist.
- Beispiel: Berechnung des größten gemeinsamen Teilers
  - Jede der Zahlen, zu denen der größte gmeinsame Teiler berechnet werden soll, ist einem Prozeß als sein Anfangswert zugeordnet.
  - Start:

    Sende an alle Nachbarn den eigenen Anfangswert
  - Empfang einer Nachricht:

    Falls der empfangene Wert kleiner als der eigene ist, wird folgendermaßen verfahren:
    - Ist der empfangene Wert ein Teiler des eigenen, so wird der empfangene Wert an alle Nachbarn verteilt,
    - ansonsten wird der Rest, der bei der Division des eigenen Wertes durch den empfangenen bleibt, an alle Nachbarn verteilt.

Nachrichtenspiel

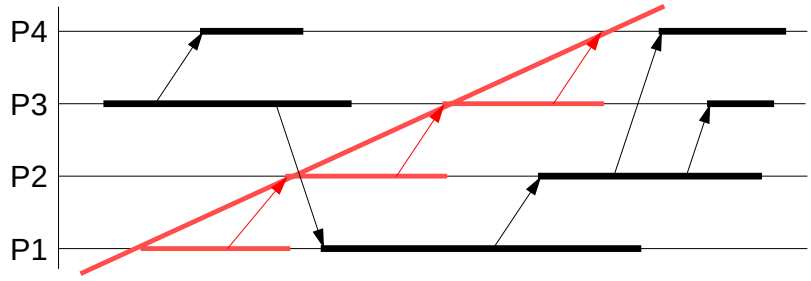




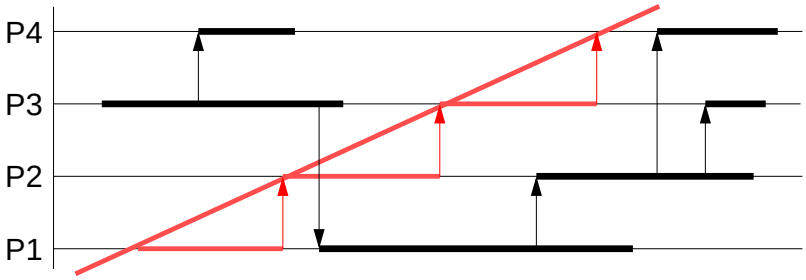
8.2 Analyse des Problems

❑ Kommunikationsmodelle

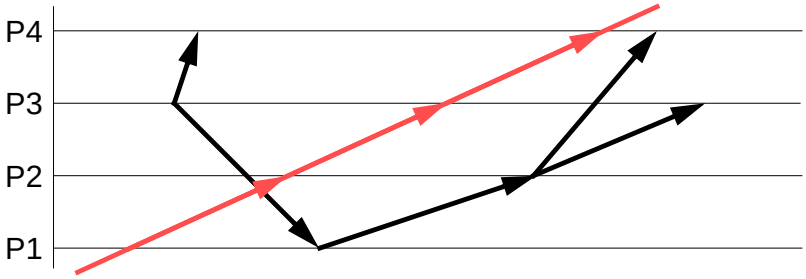
- **Asynchrones Modell (Berechnungen und Übertragungen benötigen Zeit)**



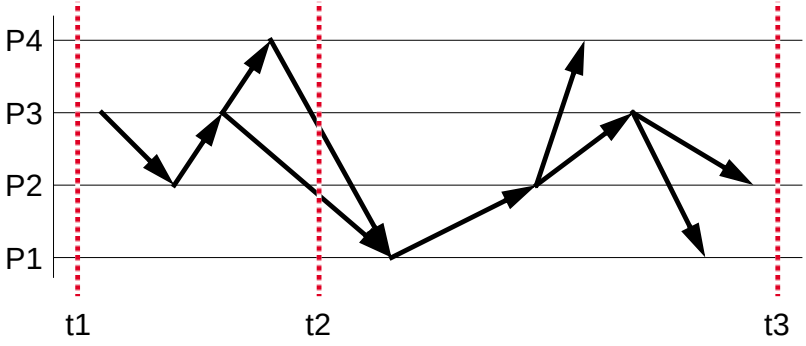
Synchrones Modell (Übertragung zeitlos)

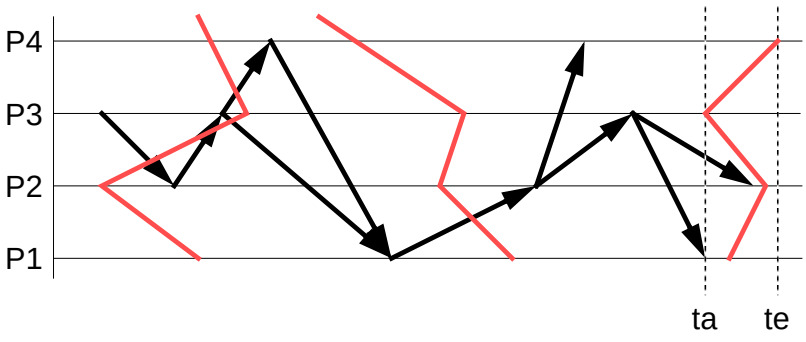


Atommodell (Bearbeitung zeitlos)



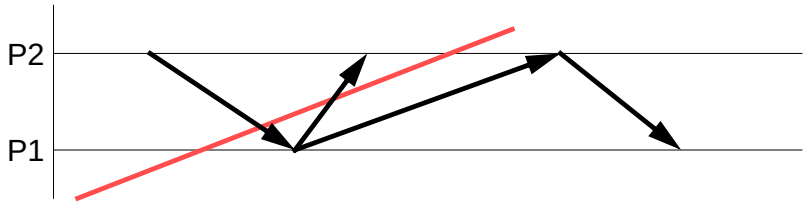
Zeitdiagramm einer Berechnung im Atommodell

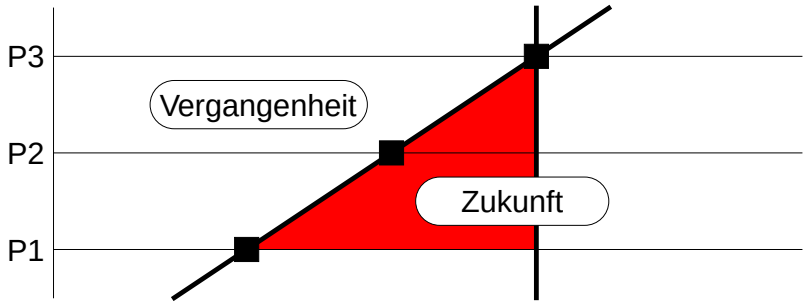




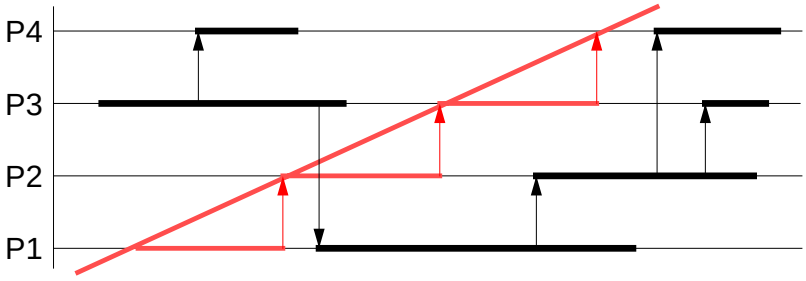
8.3 Zählverfahren

- **Einfaches Zählen genügt nicht**

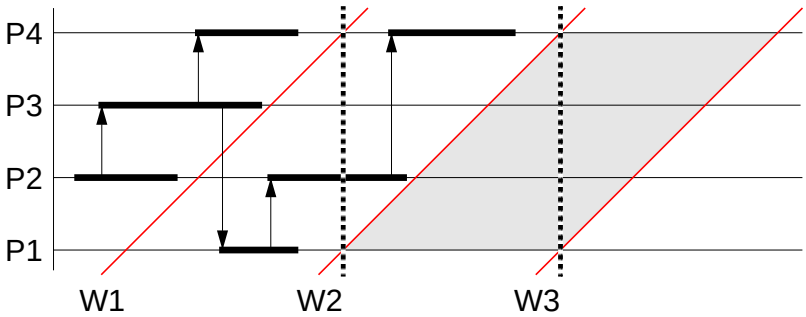




- **Auch im synchronen Modell tritt das Phänomen der Nachricht aus der Zukunft auf**



- **Doppelzählung**



Verteilte Terminierung: Zählverfahren

Lösungsideen zur "Reparatur" der Zählverfahren

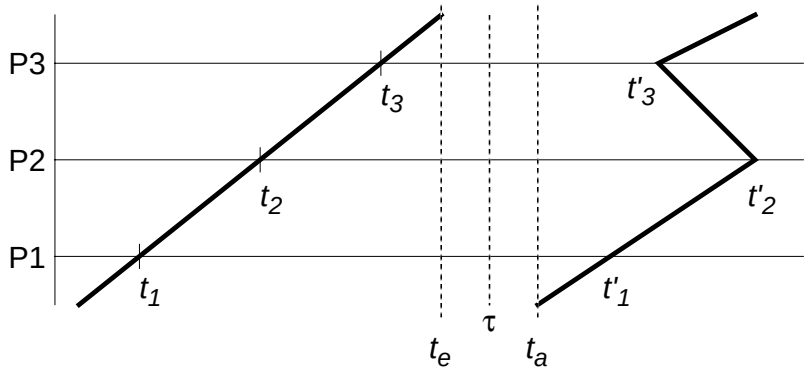
1. Einfrieren der Basiskommunikation im gefährlichen Dreieck
2. Erkennen von inkonsistenten Zeitschnitten durch geeignete logische Zeitstempel
3. Differenzierteres Zählen durch Identifikation der Nachrichten
4. Nachträgliches Überprüfen

Beurteilungskriterien

1. Einfachheit, Verifizierbarkeit, Korrektheit
2. Zugrundeliegendes Modell
3. Topologieabhängigkeit
4. Globales Wissen
5. Dynamik
6. Symmetrie
7. Fehlertoleranz
8. Implementierbarkeit
9. Effizienz

Verteilte Terminierung: Doppelzählverfahren

Bezeichnungen



t_i **Zeitpunkt, zu dem die erste Welle Prozeß i erreicht.**

t'_i **Zeitpunkt, zu dem die zweite Welle Prozeß i erreicht.**

$s_i(t)$ **Zahl der Basisnachrichten, die Prozeß i bis Zeitpunkt t versandt hat.**

$r_i(t)$ **Zahl der Basisnachrichten, die Prozeß i bis Zeitpunkt t empfangen hat.**

t_e **Ende der ersten Welle**

t_a **Beginn der zweiten Welle**

Weiter sei: $S^* = \sum_i s_i(t_i)$, $S'^* = \sum_i s_i(t'_i)$,

$R^* = \sum_i r_i(t_i)$ **und** $R'^* = \sum_i r_i(t'_i)$.

S8.1 **Satz**

Wird festgestellt, daß die Zahl der gesendeten und empfangenen Basisnachrichten entlang einer Welle identisch ist und stimmen die Werte für zwei sich nicht überlappende Wellen überein, dann ist die zugrundeliegende Berechnung terminiert.

Beweis: Es gelten fogende Beziehungen

(1) $t \le t' \Rightarrow (s_i(t) \le s_i(t') \wedge r_i(t) \le r_i(t'))$

(2) $t \le t' \Rightarrow (S(t) \le S(t') \wedge R(t) \le R(t'))$

(3) $R^* \le R(t_e)$

(4) $S'^* \ge S(t_a)$

(5) $\forall t(R(t) \le S(t))$

Verteilte Terminierung: Doppelzählverfahren

$R^* = S'^* \Rightarrow R(t_e) \geq S(t_a)$ wegen (3) u. (4)

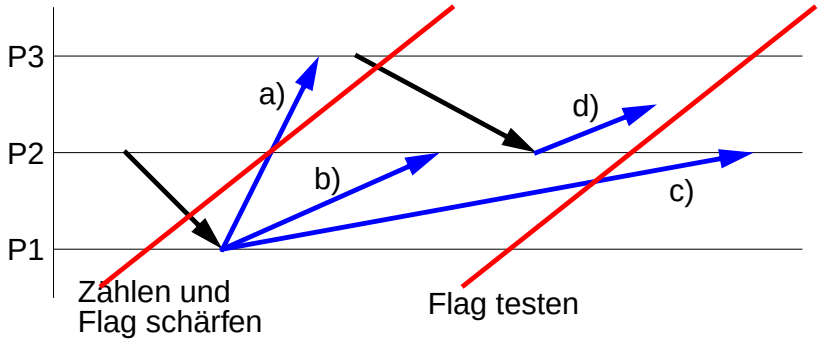
$\Rightarrow R(t_e) \geq S(t_e)$ wegen $t_e \leq t_a$ u. (2)

$\Rightarrow R(t_e) = S(t_e)$ wegen (5)

Also ist zum Zeitpunkt t_e das System terminiert, da bezüglich des Schnitts t_e das kritische Dreieck leer ist.

Verteilte Terminierung: Skeptischer Algorithmus

Skeptischer Algorithmus



Wahlbasierte Algorithmen

Bei Empfang einer Basisnachricht:

```
s--; com = true;
```

Beim Senden einer Basisnachricht:

```
s++;
```

Start des Algorithmus durch Prozeß s:

```
n = 0; predecessor = 0; accu = 0; com = false;
clock.t++; clock.i = s;  /* clock enthält als logische Zeit
                            den Wert eines lokalen Zählers
                            und die Prozeßnummer.
                         */
send(allNeighbors, EXPLORER(clock));
```

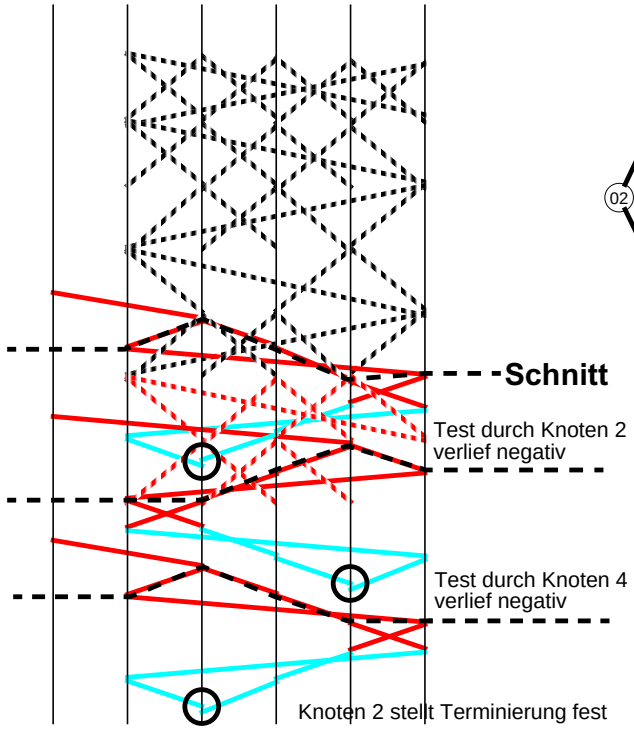

Bei Empfang einer Nachricht EXPLORER(t) von Prozeß p:

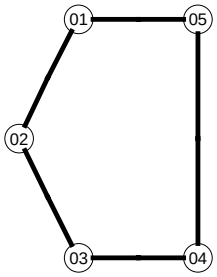
```
if (t > clock) {
   clock = t;
   if (predecessor == 0 && n != numberOfNeighbors) testFailed();
   predecessor = p;
   if (numberOfNeighbors == 1) {
      send(predecessor, ECHO(t, s, false));
   } else {
      n = 1; accu = 0; com = false;
      send(allNeighborsWithoutPredecessor, EXPLORER(clock));
   }
} else if (t == clock){
   checkCompleted();
}
```

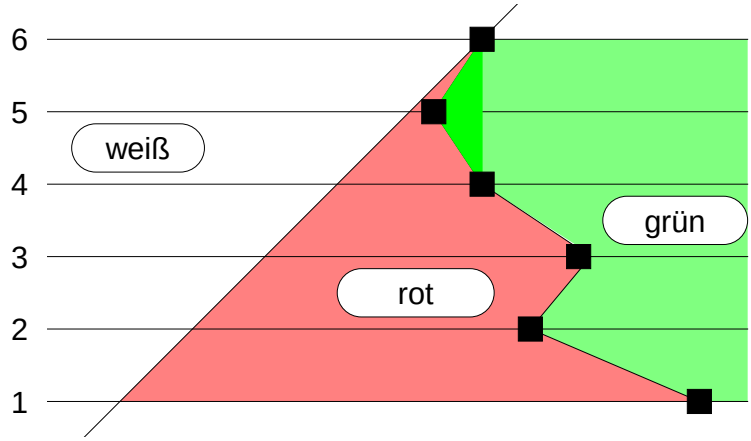
Beim Empfang von ECHO(t, sum, failed):

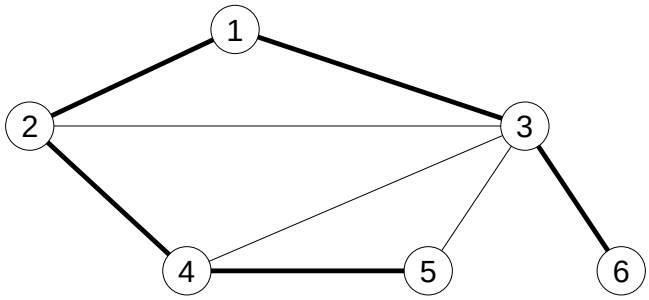
```
if (t == clock) {
   accu += sum;
   com |= failed;
   checkCompleted();
}

checkCompleted()
{  n++;
   if (n == numberOfNeighbors) {
      if (predecessor != 0)
         send(predecessor, ECHO(t, accu + s, com));
      else if (accu + s == 0 && !com) systemTerminated();
      else testFailed();
   }
}
```


❑ **Beispiel: Größter gemeinsamer Teiler**







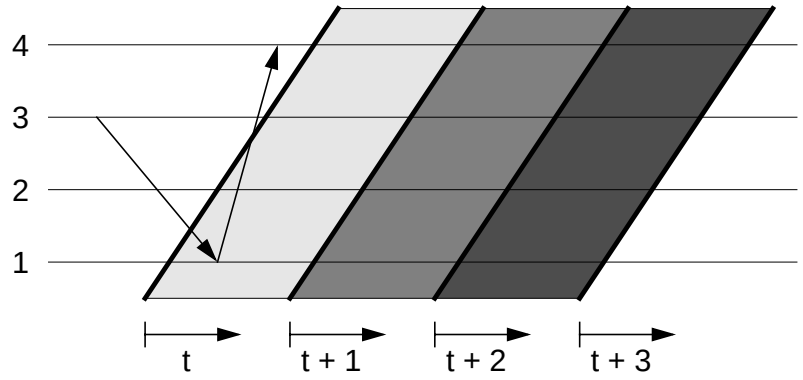


1. **Wenn von einem roten Prozeß eine Basisnachricht empfangen wird, wird das lokale Kommunikationsflag gesetzt.**
2. **Wenn eine Basisnachricht von einem roten Prozeß versandt wird, wird schließlich das Kommunikationsflag des Initiators gesetzt.**
3. **Ein weißer Prozeß kann keine von einem grünen versandte Basisnachricht empfangen.**
4. **Wenn am Ende des Algorithmus das Flag des Initiators nicht gesetzt wurde, sind alle Nachrichten, die von einem weißen Prozeß empfangen wurden, auch von einem weißen Prozeß versandt worden.**
5. **Eine Basisnachricht von einem weißen Prozeß, die von einem weißen empfangen wurde, ändert den akkumulierten Nachrichtenzähler nicht.**
6. **Wenn am Ende des Algorithmus das System nicht terminiert ist, aber das Flag des Initiators nicht gesetzt wurde, dann existiert eine von einem weißen Prozeß ausgesandte Basisnachricht, die von einem grünen Prozeß empfangen wurde oder noch von einem solchen Prozeß empfangen werden wird.**
7. **Wenn eine von einem weißen Prozeß ausgesandte Basisnachricht von einem grünen empfangen wird und das Flag nicht gesetzt ist, dann ist der Wert des akkumulierten Zählers größer als 0.**

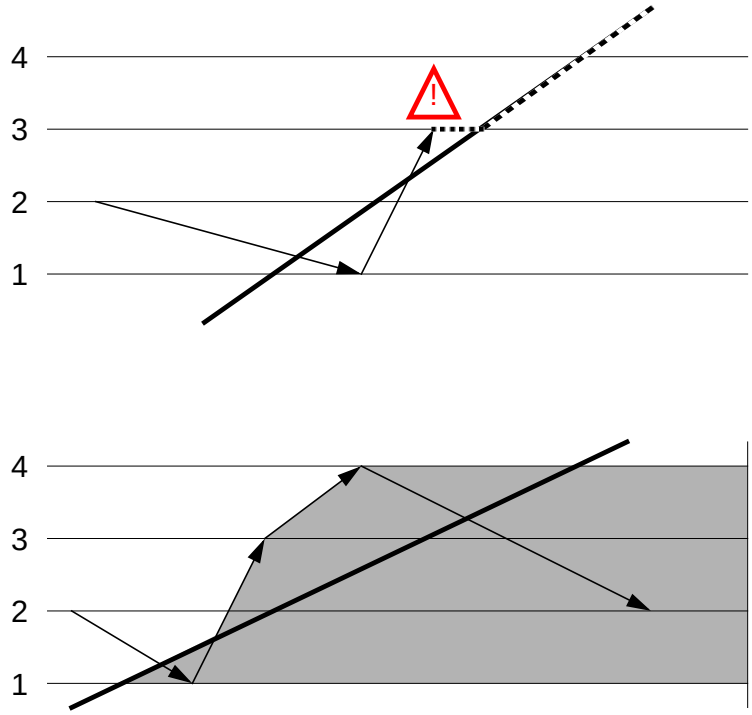
S8.2 **Satz**

Wenn die Basisberechnung am Ende des Terminierungsalgorithmus noch nicht terminiert ist, dann ist das Flag gesetzt oder der akkumulierte Zähler ist größer als 0.

8.4 **Zeitzonenverfahren**



Verteilte Terminierung: Zeitzonenverfahren



Verteilte Terminierung: Terminierung (Zeitzonen-Algorithmus)

Symmetrischer, ringbasierter Zeitzonenalgorithmus

```
int CLOCK = 0;  // aktuelle, lokale Zeit
int COUNT = 0;  // Zahl gesendeter minus Zahl empfangener
                // Nachrichten
int TMAX = 0;   // Höchster Zeitstempel der empfangenen
                // Basisnachrichten
```

(a) Senden einer Basisnachricht an P_i

```
1.  COUNT = COUNT + 1;
2.  send(CLOCK, ..., Pi);
```

(b) Empfang einer Basisnachricht (TSTAMP, ...) durch P_j

```
3.  COUNT = COUNT - 1;
4.  TMAX = max(TSTAMP, TMAX);
5.  Bearbeiten der Nachricht;
```

(c) Empfang einer Kontrollnachricht (TIME, ACCU, INVALID, INIT) durch P_j

```
6.   CLOCK = max(TIME, CLOCK);
7.   if (INIT == j)
8.      if (ACCU == 0 & !INVALID)
           terminated();
9.      else tryAgain();
10.  else send(TIME, ACCU + COUNT, INVALID | TMAX >= TIME, INIT,
                P(j + 1) % n) + 1);
```

(d) Bei Start einer Kontrollrunde durch P_j

```
12.  CLOCK := CLOCK + 1;
14.  send(CLOCK, COUNT, false, j, P(j + 1) % n) + 1);
```


- Zeitzonenalgorithmus für sternförmige Architektur mit endlichem Epochenzähler und Sternmittelpunkt P_0

(a) Senden einer Basisnachricht an P_i

```
1. COUNT = COUNT + 1;
2. send(TIME_ZONE, ..., Pi);
```

(b) Empfang einer Basisnachricht <TSTAMP, ...>

```
3. COUNT = COUNT - 1;
4. TIME_WARP = TIME_WARP | TSTAMP == (TIME_ZONE + 1) % k;
5. Bearbeiten der Nachricht;
```

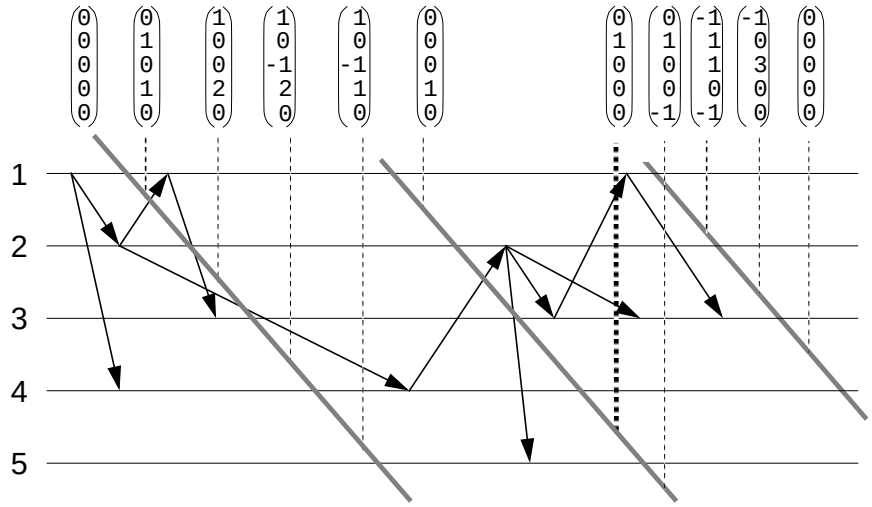
(c) Empfang einer Kontrollnachricht von P_0

```
6. send(COUNT, TIME_WARP, P0);
7. TIME_WARP = false;
8. TIME_ZONE = (TIME_ZONE + 1) % k;
```

8.5 Vektorverfahren

Jeder Prozeß P_j führt einen lokalen Vektor V der Länge n (= Gesamtzahl der Prozesse). V ist mit dem Nullvektor initialisiert. Wenn P_j eine Basisnachricht an P_i verschickt, wird V[i] um 1 erhöht, wenn P_j eine Basisnachricht erhält, wird V[j] um 1 erniedrigt.

Ein Kontrollvektor C umläuft einen (virtuellen) Ring. Wenn er bei einem Prozeß vorbeikommt, wird der lokale Vektor hinzuaddiert und V selbst wieder auf den Nullvektor gesetzt.



- **Verhalten von Prozeß P_j**

(a) Senden einer Basisnachricht an P_i

```
1.    COUNT[i] = COUNT[i] + 1;
```

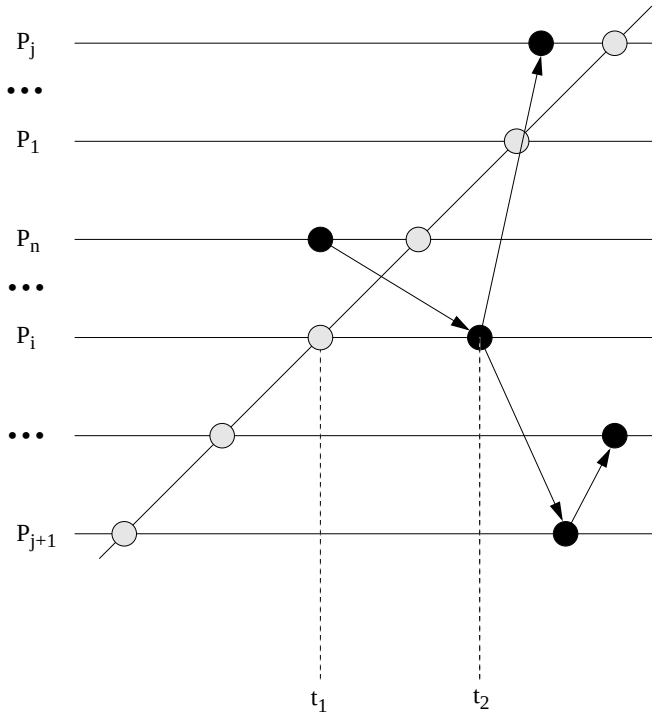
(b) Nach lokalen Aktivitäten bei Empfang einer Basisnachricht

```
2.    COUNT[j] = COUNT[j] - 1;
```

(c) Empfang einer Kontrollnachricht accumulate(ACCU)

```
3.    COUNT = COUNT + ACCU;
4.    if (COUNT == {0, 0, ..., 0}) systemTerminated();
5.    else {
        send(accumulate(COUNT), P(j + 1) % n);
6.       COUNT = {0, 0, ..., 0};
7.    };
```


Nachweis der Richtigkeit des Verfahrens



8.6 Kreditverfahren

- **Forderungen an die Kreditanteile:**
  1. **Zu jedem Zeitpunkt besitzt die Summe aller Kreditanteile, die von Prozessen oder Nachrichten gehalten werden, den Wert 1.**
  2. **Ein aktiver Prozeß hält einen Kreditanteil größer 0.**
  3. **Eine Basisnachricht, die unterwegs ist, hält einen Kreditanteil größer 0.**

- **Mögliche Umsetzung der Idee**

(R1) **Wenn ein Prozeß passiv wird, übersendet er seinen Kreditanteil an den Urprozeß.**

(R2) **Wenn eine Basisnachricht mit Kreditanteil C bei einem aktiven Prozeß ankommt, wird C an den Urprozeß weitergesandt.**

(R3) **Wenn eine Basisnachricht mit Kreditanteil C bei einem passiven Prozeß ankommt, wird C dem dadurch aktivierten Prozeß übergeben.**

(R4) **Wenn ein aktiver Prozeß mit Kreditanteil C eine Basisnachricht versendet, behält der Prozeß C/2, die andere Hälfte erhält die Nachricht.**

- **Darstellung von C: CREDIT = - ld C**

- **Beim Versenden einer Basisnachricht:**

```
CREDIT++;
send(CREDIT, ...);
```

- **Beim Wechsel in den passiven Zustand:**

```
send(CREDIT, Urprozess, ...);
active = false; /* Als Darstellung für CREDIT = 0 */
```

- **Bei Empfang einer Basisnachricht (CR, ...):**

```
if (active) send(CR, Urprozess, ...);
else { active = true; CREDIT = CR; }
```

Verteilte Terminierung: Kreditverfahren

- **Urprozeß empfängt eine Basisnachricht (CR, ...):**

DEBTS Menge von Werten d_i derart, daß

$\text{CREDIT}_{\text{Urprozess}} = 1 - \sum_i 2^{-d_i}$

```
K = CR;
while (K ∉ DEBTS) {
   DEBTS = DEBTS ∪ {K};
   K--;
}
DEBTS = DEBTS - {K};
if (DEBTS == ∅) terminated();
```