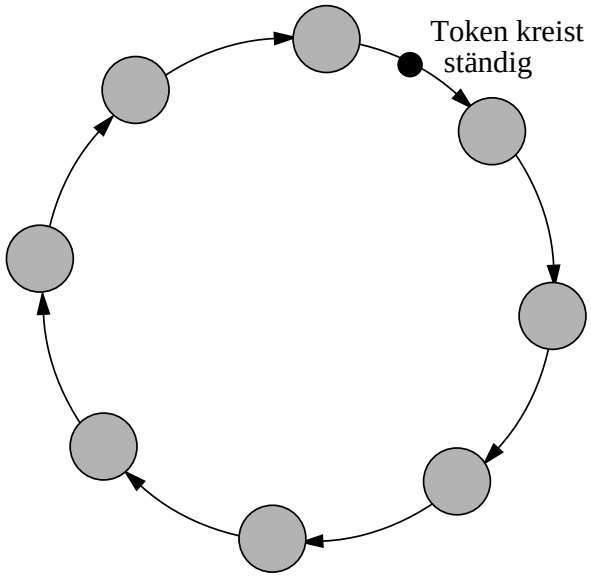
Die Token-Ring-Lösung



- Nur der Tokeninhaber darf
  - Safety ist klar
  - Liveness: Token muss weitergegeben werden
  - Fairness intuitiv gegeben

- Probleme?
  - Bei vielen Prozesse --> lange Wartezeiten, Gefahr von Tokenverlust
  - Anzahl der Einzelnachrichten nicht begrenzt (ständiges Kreisen)
  - Für jedes Betriebsmittel eigenes Token vorsehen

Token-Request-basierte Algorithmen

- Token soll nicht dauernd nutzlos unterwegs sein
  - Token wandert nur bei Bedarf

- Grundidee: "Token zu mir" an alle (?) anderen senden
  - per *broadcast* (falls entspr. Kommunikationsprimitiv existiert)
  - oder z.B. mittels *Echo-Algorithmus*
  - Aufwand ist hoch, wenn man nicht weiss, wo das Token sein könnte

- Fairness muss aber gewahrt bleiben

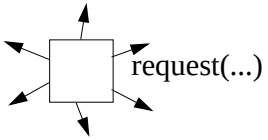
- Safety ist vergleichsweise trivial (Tokenbesitz)

- Liveness mit möglichst wenig Aufwand garantieren

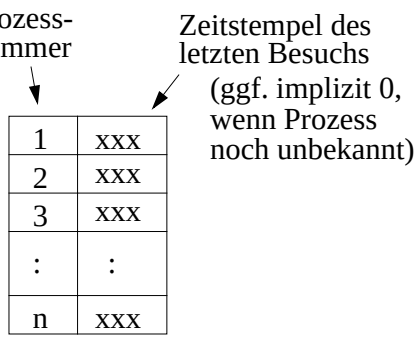
Ricart / Agrawala 1983

1) Es gibt ein einziges Token; nur Besitzer darf kritischen Abschnitt betreten

2) Request-Nachricht (mit Zeitstempel) an alle senden



3) Token hat "Gedächtnis":



4) Jeder Prozess registriert Anforderungen *aller* anderen

5) Nach Verlassen des kritischen Abschnitts wird das Token an denjenigen Prozess geschickt, der am "längsten" wartet (Anforderung muss jünger als Zeitstempel des letzten Besuchs beim Prozess sein)

--> Nachrichtenkomplexität: n (n-1 für request, + 1 Token)
(bzw. 0, wenn inzwischen kein anderer wollte)

Fragen:
- Wie lange muss ein Prozess max. (auf Mitbewerber) warten?
- Liveness? Fairness?
- Geht es auch mit weniger Nachrichten?

Token-Request-basierte Algorithmen

aber wie bekommt man diesen?

- Andere Idee: *Spannbaum* verwenden

  - "Suchnachrichten" wandern nur auf Kanten des Baumes
  - Token benutzt ebenfalls nur Baumkanten
  - Aufwand: Maximal n-1 Einzelnachrichten, um jeden der n Knoten über den Tokenwunsch zu informieren

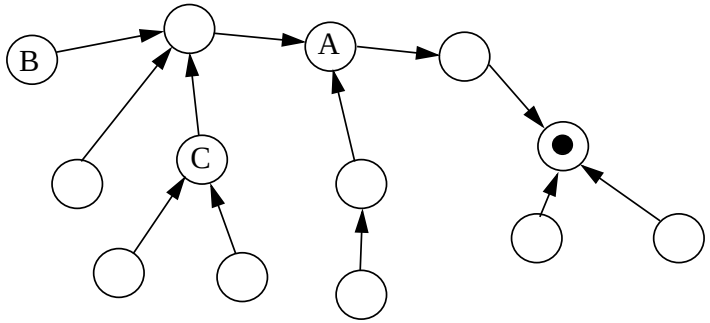
---

- Bessere Idee: Spannenden Baum mit *gerichteten Kanten*

  - Kanten zeigen immer in Richtung des Tokenbesitzers
  - typischerweise O(log n) Einzelnachrichten, um den Tokenbesitzer zu erreichen (aber bei gutartig / bösartig entarteten Bäumen?)
  - wenn Token seinen Ort wechselt, müssen Kantenrichtungen aktualisiert werden! (Aufwand?)

Der O(log n)-Algorithmus

- Das wäre aber besser als das "optimale" O($\sqrt{n}$)- Verfahren!?!
- Unabhängig veröffentlicht 1987 (van de Snepscheut) und 1989 (Raymond)
- Ähnliches Verfahren auch von Naimi / Trehel (1987)



- Token wandert entgegen der Pfeilrichtung zum anfordernden Prozess und dreht jede durchlaufene Kante um

- Ein Prozess sendet (bis er das Token erhält) nur ein Mal ein request auf seiner ausgehenden Kante
  - nochmals "ungeduldig" nachfragen hilft auch nichts
  - Prozess merkt sich aber, wer ihn alles um Weiterleitung des requests bat

- Bei Empfang des Tokens wird es (in fairer Weise) in eine der anfordernden Richtungen weitergeleitet
  - vorher selbst benutzen, wenn Bedarf besteht? ← vgl. "Lift-Algorithmus"
  - Fairness: Nur beschränkt oft in eine andere Richtung weiterleiten, bevor es über eine wartende Kante weitergeleitet wird
  - falls Anforderungen aus mehreren Richtungen vorliegen: Dem ausgesendeten Token sofort ein request hinterhersenden (Optimierung: Token und request zusammenfassen)

- Sind Nachrichtenüberholungen ein Problem?
  - z.B.: request überholt Token

Skizze des Algorithmus

- *Nachrichtenarten*:
  - REQUEST
  - TOKEN

- *Lokale Variablen eines Prozesses P_i*:
  - HOLDER = 'self' oder Name des Nachbarn, in dessen Richtung das Token liegt
  - ASKED: bool = 'true' $\Leftrightarrow$ es ist ein REQUEST in Richtung HOLDER versendet worden
  - REQ_QUEUE = FIFO-Warteschlange für angekommene REQUEST-Nachrichten von Prozessen, die das Token möchten (enthält Namen von Nachbarn oder 'self')

- Garantiert 'FIFO'-Eigenschaft der Warteschlange Fairness?

Skizze des Algorithmus (2)

Einige Details müssten in naheliegender Weise ergänzt werden

- Prozess P_j möchte Token, hat es jedoch nicht:

```
Füge 'self' an REQ_QUEUE an;
if not ASKED then
      send REQUEST to HOLDER; ASKED := true;
```

- P_j empfängt REQUEST von P_i:

```
Füge 'Pi' an REQ_QUEUE an;
if not ASKED and HOLDER ≠ self then
      send REQUEST to HOLDER; ASKED := true;
```

- P_j empfängt Token von P_k:

```
HOLDER := dequeue(REQ_QUEUE);
if HOLDER = self then  <kritischer Abschnitt>
                  else send TOKEN to HOLDER
                       if |REQ_QUEUE| > 0 then
                          send REQUEST to HOLDER;
                          ASKED := true;
```

- P_j hat Token und möchte es loswerden:

```
if |REQ_QUEUE| > 0 then
   HOLDER := dequeue(REQ_QUEUE);
   send TOKEN to HOLDER;
   if |REQ_QUEUE| > 0
      then send REQUEST to HOLDER;
      else ASKED := false;
```

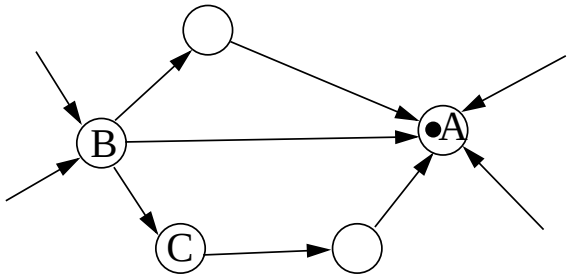

Verallgemeinerung auf allg. Graphen

- Gerichtet, azyklisch und schwach zusammenhängend
- Kann man die Kanten jedes ungerichteten Graphen so orientieren, dass kein Zyklus entsteht?
  - Bsp. Stadtplanung: Einbahnstrassenrichtung so festlegen, dass man nicht im Kreis fahren kann
  - Ja: Willkürliche Ordnung auf den Knoten festlegen (z.B. Knoten numerieren) und Kanten entsprechend dieser Ordnungsrelation orientieren!

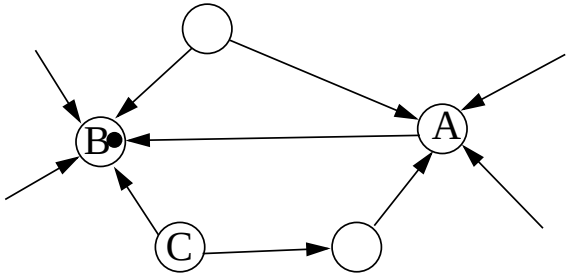
- Präzisere Forderung: Jeder gerichtete Weg soll beim (eindeutigen) Tokenbesitzer enden
  - Klar, dass Tokenbesitzer nur eingehende Kanten hat
  - Auch das geht!
    - Starte Echo-Algorithmus vom Tokenbesitzer aus
    - Durch die Flussrichtung der Echos entsteht ein auf diesen Knoten hin gerichteter Baum
    - Nun müssen noch die Nicht-Baumkanten orientiert werden
    - Dazu bekommt jeder Knoten eine Ordnungsnummer aus:
      - seiner Höhe (= Entfernung zur Wurzel = Tokenbesitzer) als primärem Ordnungskriterium,
      - einem eindeutigen sekundären Ordnungskriterium
    - "Flussrichtung" der Kanten von der höheren zur niedrigeren Ordnungsnummer --> ist azyklisch!

Tokenanforderung

- Schicke request über irgendeine ausgehende Kante
  - klappt mit jeder Kante (Weg zum Token ist aber i.a. verschieden lang)
  - auch denkbar: über alle Pfade gleichzeitig... (--> mehr Aufwand!)



Regel: Wenn das Token wandert, werden *alle* Ausgangskanten des Empfängers invertiert



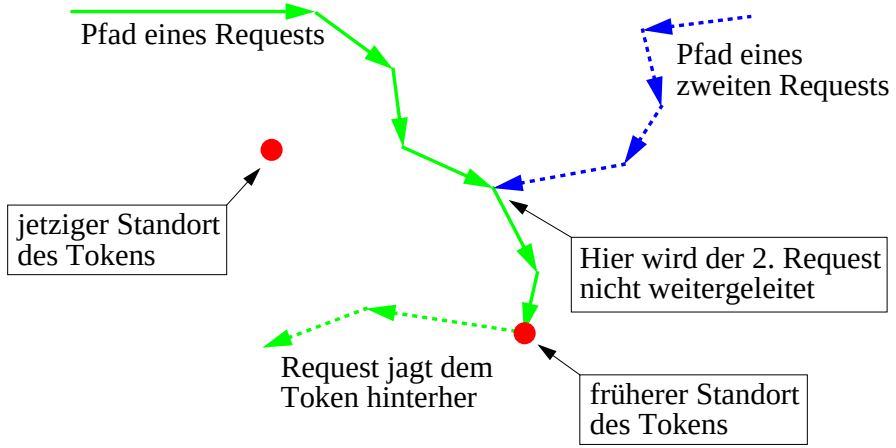
- Behauptung: *Zyklenfreiheit ist eine Invariante*

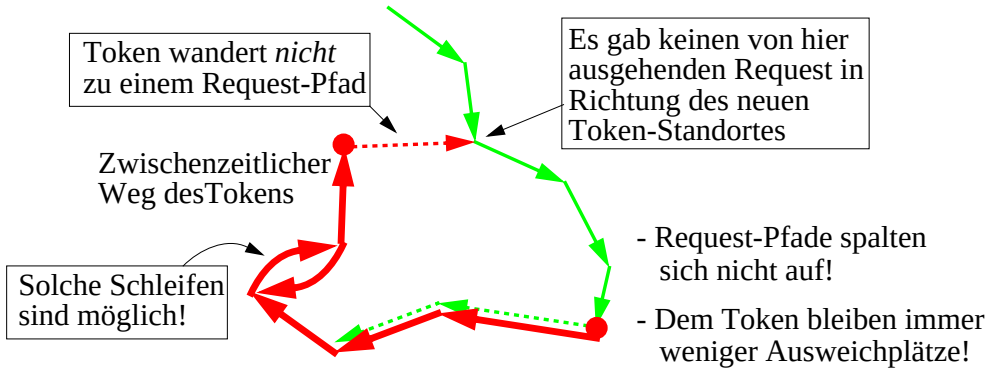
  Beweis: Neue Zyklen können höchstens durch die invertierten Kanten entstanden sein - mit diesen ist jedoch kein Zyklus möglich, da sie alle zum neuen Tokenbesitzer gerichtet sind, welcher keine ausgehenden Kanten besitzt

- Weitere Invariante: *Jeder gerichtete Pfad endet beim Tokenbesitzer*

Glücklose Token-Jagd?

- Wieso holt ein Request das Token schliesslich immer ein?





Aufwand und Varianten

- Bemerkung: Richtung einer Kante sei als "Wegweiser" im ausgehenden Knoten gespeichert
- Allen Nachbarn des neuen Tokenbesitzers muss daher eine Meldung gesendet werden, damit diese den "Wegweiser" entsprechend setzen
  - tatsächlich allen, oder kann man sich das für einige sparen?
  - "Gewinn" des Verfahrens: Nachbarn haben nun i.a. einen kürzeren Weg zum Token
  - Denkübung: muss der Empfang aller dieser Meldungen abgewartet werden (Quittungen!), bevor das Token weiterreisen darf?

- Variante: Statt die Nachbarn zu informieren:
  - Nur die Richtung der einen vom Token durchlaufenen Kante ändern
  - Beim neuen Tokenbesitzer alle ausgehenden "Wegweiser zum Token" löschen

    --> es entstehen ungerichtete Kanten!
- Dies spart Nachrichtenaufwand! Vergleich mit:
  - obigem "Originalverfahren"?
  - Verfahren mit spannendem gerichteten Baum?
- Welche Entfernung legt das Token im Mittel zurück?
- Ist ein vollst. Graph ein interessanter Sonderfall?