

# Token-Request-basierte Algorithmen

aber wie bekommt man diesen?

## - Andere Idee: *Spannbaum* verwenden

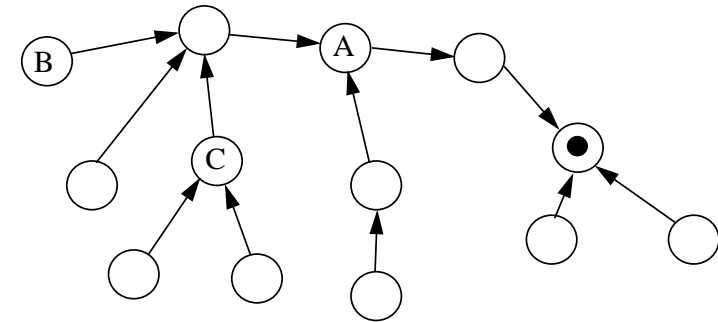
- "Suchnachrichten" wandern nur auf Kanten des Baumes
- Token benutzt ebenfalls nur Baumkanten
- Aufwand: Maximal  $n-1$  Einzelnachrichten, um jeden der  $n$  Knoten über den Tokenwunsch zu informieren

## - 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!?!)



- 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 hinterher senden (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 ' $P_i$ ' an REQ_QUEUE an;  
if not ASKED and HOLDER  $\neq$  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 nummerieren) und Kanten entsprechend dieser Ordnungsrelation orientieren!

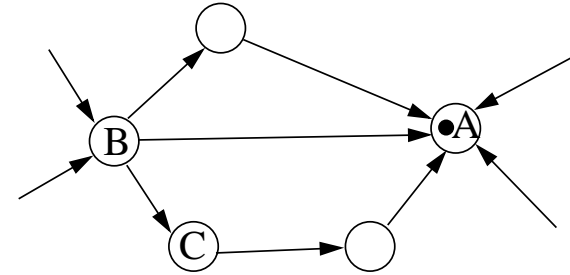
---

## - Präzisere Forderung: Jeder gerichtete Weg soll beim (eindeutigen) Tokenbesitzer enden

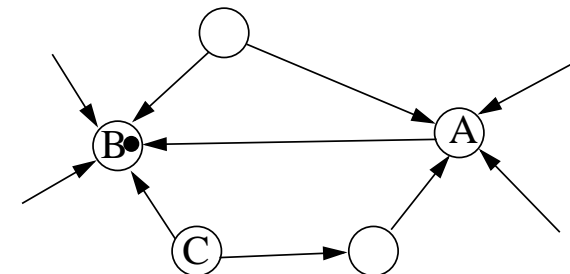
- Klar, dass Tokenbesitzer nur eingehende Kanten hat
- Auch das geht!
  - Starte Echo-Algorithmus vom Tokenbesitzer aus
  - Durch die Flussrichtung der Echos entsteht ein auf diesen Knoten hin gerichteter Baum
  - Nun müssen noch die Nicht-Baumkanten orientiert werden
  - Dazu bekommt jeder Knoten eine 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*



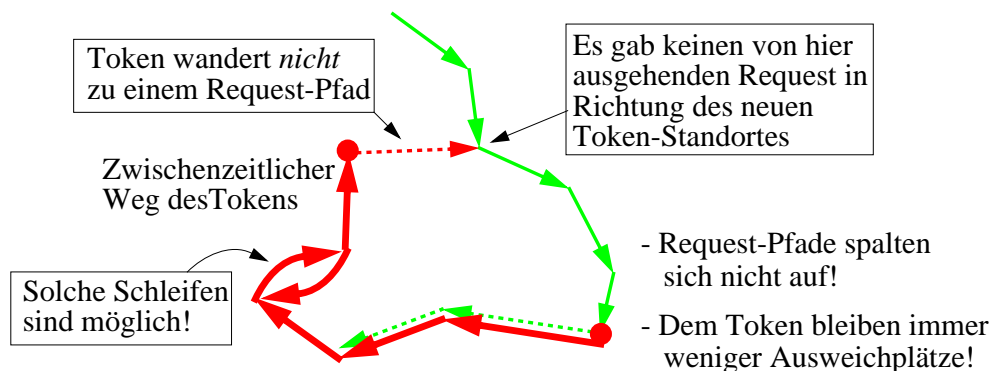
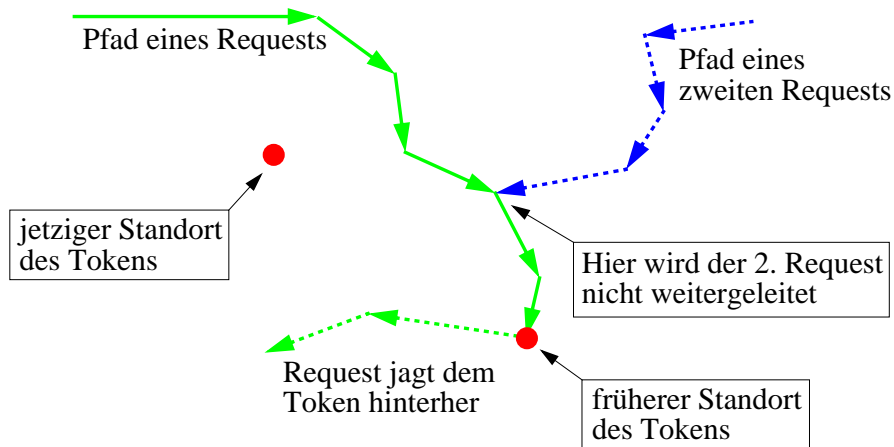
## - Behauptung: *Zyklenfreiheit ist eine Invariante*

*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?



- Request-Pfade spalten sich nicht auf!  
- Dem Token bleiben immer weniger Ausweichplätze!

# Aufwand und Varianten

- Bemerkung: Richtung einer Kante sei als "Wegweiser" im ausgehenden Knoten gespeichert
- Allen Nachbarn des neuen Tokenbesizers 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!

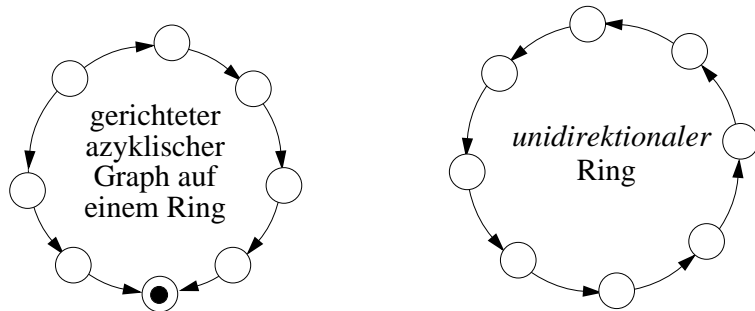
- Dies spart Nachrichtenaufwand! Vergleich mit:

- obigem "Originalverfahren"?
- Verfahren mit spannendem gerichteten Baum?

- Welche Entfernung legt das Token im Mittel zurück?

- Ist ein vollst. Graph ein interessanter Sonderfall?

## Sonderfall Ring ?



- Auf *unidirektionalen* Ringen ist folgende Variante möglich:

- Token wird "gejagt" (statt zurückgeholt)
- Token kommt damit schliesslich beim Anforderer an
- Jagdnachricht wird von einem Knoten nicht weitergeleitet, wenn dieser bereits eine solche (seit dem letzten Tokenbesuch) ausgesendet hat

- Frage: Sind FIFO-Kanäle notwendig?

- Variante: ständig kreisendes Token ("perpetuum mobile")

- keine "Jagdnachrichten" nötig
- --> Algorithmus hatten wir anfangs bereits genannt!

## Welcher Baum beim $O(\log n)$ -Verfahren?

- Baum mit inneren Knoten vom *Grad*  $k$ :

--> längster Weg hat Länge  $O(\log_k n)$ ; durchschnittliche Weglänge ebenfalls

--> bestimmt die Nachrichtenkomplexität bei *schwacher Last*

- Knoten im "Zentrum" werden mehr belastet als weiter aussen liegende Knoten!

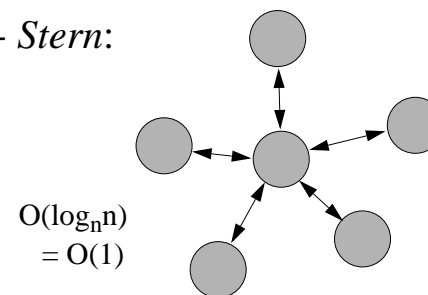
↑  
fast nie mehrere gleichzeitige Konkurrenten

- Grad an Fairness?

### Nachrichtenkomplexität bei *starker Last*:

- Idee: Jedes Weitersenden des Tokens ein "Treffer"
- Genauer: Token *traversiert* den Baum (wegen Fairness)
  - >  $n$  Knotenbesuche bei  $2(n-1)$  Token-Nachrichten
  - > weiterer Faktor 2, da gleiche Anzahl von requests
  - > ca. 4 Nachrichten pro Betreten des kritischen Abschnittes
- Algorithmus wird bei starker Last also "besser"!

- *Stern*:

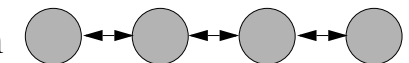


- Kürzeste Wege --> beste Topologie? (D.h. wenn schon Baum, dann diesen?)

- Unterschied zur Lösung mit zentralem Manager? (Und was war daran so schlecht?)

- Beachte: Einen Spannbaum gibt es immer; ein Stern erfordert aber u.U. zusätzliche (logische oder physische) Verbindungen!

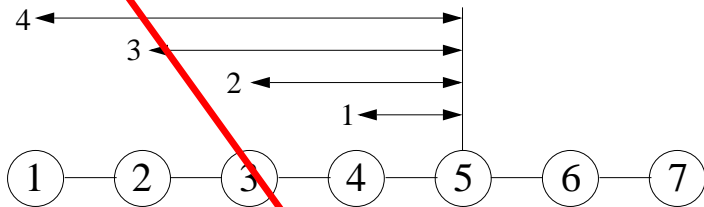
- *Lineare Kette* als entarteter Baum



--> mittlere Weglänge  $\approx n/3$  (Beweis: einfaches kombinator. Nachrechnen)

# Mittlere Weglänge im linearen Fall

(Für diejenigen, denen das Rechnen Spass macht)



$$\left. \begin{array}{l} 1 \\ 1+2 \\ 1+2+3 \\ 1+2+3+4 \\ \dots \\ 1+2+3+\dots+(n-1) \end{array} \right\} \sum_{j=1}^{n-1} \sum_{i=1}^j i = \sum_{j=1}^{n-1} \frac{j(j+1)}{2} = \frac{1}{2} \sum_{j=1}^{n-1} j^2 + \frac{1}{2} \sum_{j=1}^{n-1} j$$

Denn es gilt:

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6} \quad \Rightarrow \quad \frac{1}{2} \frac{(n-1)n(2(n-1)+1)}{6} + \frac{1}{2} \frac{n(n-1)}{2}$$

Gemittelt über alle  $n(n-1)/2$  ungerichteten Paare  $\{i, j\}$  mit  $i \neq j$ :

$$\Rightarrow \frac{2(n-1)+1}{6} + \frac{1}{2} = \frac{2n+2}{6} = \frac{1}{3}(n+1) \approx \underline{\underline{n/3}}$$

(ggf. abzüglich der Fälle, wo das Token bereits "vor Ort" ist)

# Klassifikation: Token $\Leftrightarrow$ Request

## 1) Token-basierte Lösungen

- Safety ist trivial
- Fairness bei Tokenweitergabe beachten
- Wie fordern Prozesse das Token an? --> unterschiedliche Lösungen
- Topologie
  - > "Reiseweg des Tokens"
  - > Zeitaufwand durch sequentielle Nachrichtenketten
- Fehlertoleranz:
  - wie Tokenverlust feststellen?
  - wer darf neues Token generieren? (Eindeutigkeit notwendig!)
- Nur anwendbar, wenn für das exklusive Betriebsmittel von vornherein ein Token eingerichtet wird --> nicht immer möglich!

## - Beispiel für *a priori unbekannte exklusive Betriebsmittel*: "Reservierungszeiten für einen Tennisplatz"

- "Ich benötige ihn übermorgen von 10.28 - 12.17 exklusiv"
- Dafür lässt sich nicht von vornherein ein Token generieren!
- Vielleicht: ein Token "Tennis <Zeitintervall>" dynamisch generieren?
- Aber: wer garantiert, dass ein anderer dies nicht gleichzeitig tut?
- Zurück zu einer zentralen Lösung mit allen Nachteilen? ("Tennisplatz als Monitor")

## - Anderes Bsp. für ein "abstraktes Betriebsmittel": Terminvereinbarung mit einer beliebigen Menge von Teilnehmern

- Exklusives Generieren eines Tokens unter symmetrischen Bedingungen --> *Election-Problem* (--> später)

# Klassifikation (2)

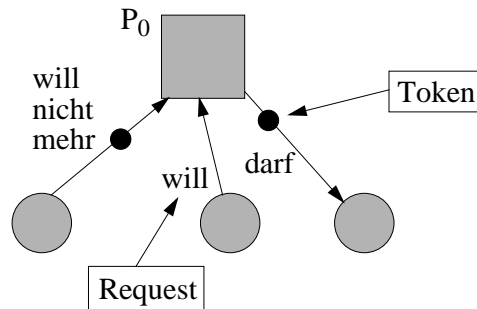
## 2) Request-basierte Lösungen

- wen sollen die Prozesse fragen? (request set)
  - Safety sicherstellen!
  - Deadlockfreiheit ist nicht trivial
- auch hierfür verschiedene Lösungen

## 3) Zentraler Manager

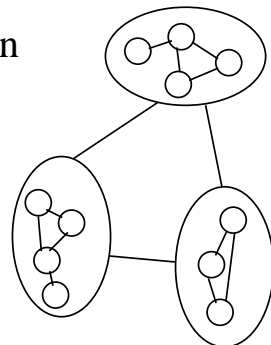
bzw. Server oder Monitor

- ist entarteter Fall beider Klassen!



## 4) Hierarchische / hybride Verfahren

- bei grossen Systemen mehrstufig (mittels "Stellvertreter")
- auf verschiedenen Stufen / in verschiedenen Clustern ggf. unterschiedliche Verfahren



# Vergleich der Nachrichtenkomplexität

(pro Betreten des kritischen Abschnittes bzw. Anforderung des Betriebsmittels)

Token-Ring (LeLann,1977)	.....	1 ... ∞
Lamport (1978)	.....	3 (n-1)
Ricart / Agrawala (1981)	.....	2 (n-1)
("an optimal algorithm...")		
Ricart / Agrawala (1983)	.....	n
Maekawa (1985)	.....	$O(\sqrt{n})$
Lift-Algorithmus auf Baum (1987)	.....	$O(\log_k n)$
Zentraler Manager	.....	2

Vorlesung "vert. Systeme"

- Offenbar soll es aber eben nicht alleine auf die Nachrichtenkomplexität ankommen!
- "Qualitative" Merkmale oft wichtiger!

# Wechselseitiger Ausschluss: Kriterien

- Nachrichtenkomplexität

für die Qualität eines  
Lösungsalgorithmus

- Symmetrie

- syntaktisch: gleicher Algorithmus für alle
- semantisch: gleiche Last für alle etc.

Inwiefern würde man  
den  $O(\log n)$ -Algo-  
rithmus als symme-  
trisch bezeichnen?

- Fehlertoleranz

- Verhalten des Algorithmus bei Fehlern
  - z.B. Nachrichtenverlust
  - oder: nach Abbruch von aussen wegen Deadlock der Anwendung
- Zusatzaufwand, um (etwa bei erkanntem Fehler) wieder einen konsistenten Zustand herzustellen

- Grad an Fairness

- inwieweit wird zeitlich globale Reihenfolge der Requests eingehalten?

- Zeitbedarf zwischen Freigabe und Benutzung durch einen anderen Prozess

- (minimale) Länge sequentieller Nachrichtenketten
- Bsp.: bei  $O(\log n)$ -Algorithmus ist Zeitbedarf auch  $O(\log n)$  (statt  $O(1)$  wie bei einigen anderen Algorithmen)!
- verschiedene Lastsituationen berücksichtigen:
  - schwache Last --> nur selten mehr als ein Konkurrent
  - hohe Last --> Betriebsmittel fast ständig in Benutzung

- Effizienz / Einfachheit der Implementierung

- z.B.: wie wird broadcast / multicast ("request an alle") realisiert? (als effiziente Systemoperation; auf Ring; mit Echo-Algorithmus...)
- wird eine spezielle Topologie vorausgesetzt (Ring, Baum,...) bzw. muss jeder Prozess jeden anderen kennen?