



Verteilte Betriebssysteme

8. Kapitel Gegenseitiger Ausschluss

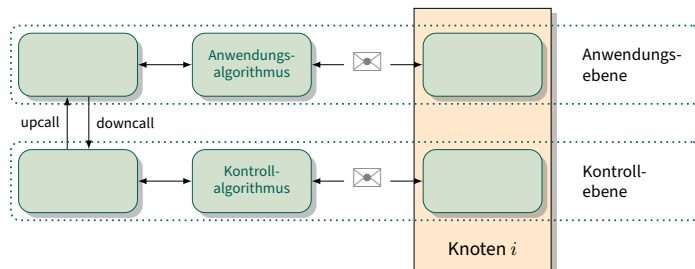
Matthias Werner
Professur Betriebssysteme

8.1 Grundlagen

- ▶ Koordination des exklusiven Zugriffs auf Ressourcen
 - ▶ Das Programmstück, das den Zugriff realisiert, heißt **kritischer Abschnitt** (critical section)
 - ▶ Beispiele für Ressourcen: Datei, Eventqueue, Drucker
- ▶ **Annahme:** Hat ein Prozess das Zugriffsrecht, so gibt er dieses nach endlicher Zeit wieder ab
- ▶ Meist: **Maximal ein** Prozess darf zugreifen
 - ▶ Manchmal auch Verallgemeinerung: maximal n Prozesse dürfen gleichzeitig zugreifen, oder unterschiedliche Zugreifer/Arten des Zugriffs $\Rightarrow$ Mehrsortenprobleme, Leser-Schreiber-Problem

Kontrollalgorithmen

- ▶ **Allgemein:** Gegenseitiger Ausschluss u.a. Koordinierungen werden in verteilten wie in nicht-verteilten Betriebssystemen gebraucht
 - ▶ Problem im verteilten BS schwieriger
- ▶ Was ist wenn Programme (Algorithmen) selbst verteilt sind?
- ▶ Algorithmen zur Feststellung/Herstellung des Zustandes von verteilten Algorithmen
- ▶ Unterscheidung „Anwendungsalgorithmus“ und „Kontrollalgorithmus“ (auch: **Steueralgorithmus**)



Anforderungen an eine Realisierung

- ▶ Kontrollalgorithmen müssen bestimmte Eigenschaften aufweisen
- ▶ **Sicherheit** (engl.: *safety*): Etwas nicht wieder gutzumachendes Schlechtes soll nie passieren
 - ▶ Hier: Zu jedem Zeitpunkt darf höchstens einem Prozess der Zugriff gestattet werden
- ▶ **Lebendigkeit** (engl.: *liveness*): Etwas das geschehen soll, geschieht schlussendlich auch
 - ▶ Hier: Gibt es mindestens einen Bewerber, muss einem der Bewerber der Zugriff nach endlicher Zeit gestattet werden
- ▶ Algorithmen müssen i.d.R. Sicherheit **und** Lebendigkeit erfüllen
- ▶ Fordert man **nur eines** von beiden, ist meist triviale Lösung möglich
 - ▶ **Beispiel:** nur Sicherheit $\Rightarrow$ Zugriff wird immer verweigert
nur Lebendigkeit $\Rightarrow$ Zugriff wird immer gewährt

Anforderungen an eine Realisierung (Forts.)

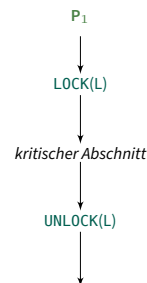
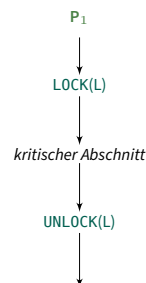
- ▶ Meist zusätzlich gefordert: **Fairness**
 - ▶ **Kein Verhungern:** Begehrt ein Prozess den Zugriff, muss ihm irgendwann (nach endlicher Zeit) der Zugriff gestattet werden
 - ▶ **Stärkere Fairnessanforderung:** Die Gewährung des Zugriffs berücksichtigt (auf die eine oder andere Weise) die Reihenfolge der Anforderungen

Leistungskriterien

- ▶ Zur Beurteilung der **Leistungsfähigkeit** eines Mechanismus für gegenseitigen Ausschluss können verschiedene Kriterien herangezogen werden
 - ▶ **Anzahl** n_m der versendeten Nachrichten pro CS (Nachrichtenkomplexität) ($n_m \rightarrow \min$)
 - ▶ **Abstimmungsverzögerung** d pro CS ($d \rightarrow \min$)
 - ▶ **Antwortzeit** t_r , d.h. Zeitspanne vom Anfordern des CS bis zum Verlassen ($t_r \rightarrow \min$)
 - ▶ **Durchsatz** T_P , d.h. Anzahl der Durchläufe durch einen CS pro Zeiteinheit ($T_P \rightarrow \max$)
- ▶ Im Folgendem Fokus auf Nachrichtenkomplexität und Abstimmungsverzögerung

Wiederholung: Gegenseitiger Ausschluss in nichtverteilten Systemen

- ▶ In zentralen Systemen gibt es für die Realisierung eines gegenseitigen Ausschlusses Kernoperationen des BS
- ▶ Realisierung basiert letztendlich auf gemeinsamer Variable

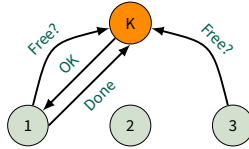


Wiederholung: Gegenseitiger Ausschluss in nichtverteilten Systemen (Forts.)

- ▶ Verschiedene Mechanismen bekannt (u.a. aus **Betriebssysteme I**):
 - ▶ Spinlock
 - ▶ Prozesslock
 - ▶ Semaphore
 - ▶ Monitor
- ▶ Mechanismen basieren auf atomarem Zugriff auf gemeinsamen Speicher (**atomares** Testen und Setzen einer Speicherzelle)
- ▶ Nicht gegeben in verteilten Systemen!
- ▶ Wie kann wechselseitiger Ausschluss in verteilten System realisiert werden?

Zentralisierte Lösung für verteilte Systeme

- Rückführung auf lokalen Fall:
Ein Prozess ist bezüglich einer Ressource Koordinator (z.B. durch Auswahlalgorithmus, siehe späteres Kapitel)
- Alle Anforderungen und Freigaben werden dem Koordinator mitgeteilt
- Zuteilungen vergibt der Koordinator
- Leicht zu implementieren
- 3 Nachrichten pro Zugriff



Verteilte Koordination

- Zentrale Lösungen sind einfach zu realisieren, haben aber auch Nachteile
 - „single point of failure“
 - Engpassgefahr (Flaschenhals)
 - Verzögerung: mindestens doppelte Nachrichtenlaufzeit
- Daher soll über verteilte Lösungen nachgedacht werden
- **Annahme:** im verteilten Fall existieren ebenfalls die Operationen **LOCK** und **UNLOCK**
- Zusätzlich können noch weitere Hilfsprozesse erforderlich sein

8.2 Broadcast-basierte Algorithmen

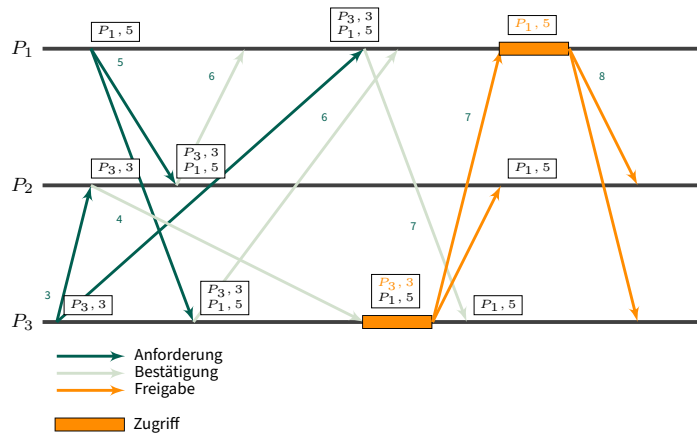
Broadcast Algorithmus von LAMPORT, 1978

- **Voraussetzungen:**
 - Verlustfreie FIFO-Kommunikationskanäle
 - Alle Nachrichten tragen eindeutige logische Zeitstempel ➡ siehe Kapitel 5 und 7
- **Grundidee:**
 - Jeder Prozess verwaltet eine nach Zeitstempeln geordnete Nachrichtenwarteschlange (Queue)
 - Anforderungen und Freigaben werden per Broadcast **an alle** gesendet

Broadcast Algorithmus von LAMPORT, 1978 (Forts.)

- **Anforderung abgeben:** Anforderung mit Zeitstempel in eigene Schlange einreihen und an alle senden
- **Anforderung empfangen:** Anforderung in eigene Schlange einreihen, Anfragebestätigung an anfragenden Prozess senden
- **Freigabe senden:** Anforderung aus eigener Schlange entfernen und Freigabe an alle senden
- **Freigabe empfangen:** Anforderung aus eigener Schlange entfernen
- Ein Prozess darf zugreifen, wenn
 - Eigene Anforderung ist **erste** Anforderung in eigener Schlange
 - Von **jedem** anderen Prozess wurde bereits eine Nachricht mit größerem Zeitstempel empfangen (z.B. die Anfragebestätigung)

Broadcast Algorithmus von LAMPORT, 1978 (Forts.)



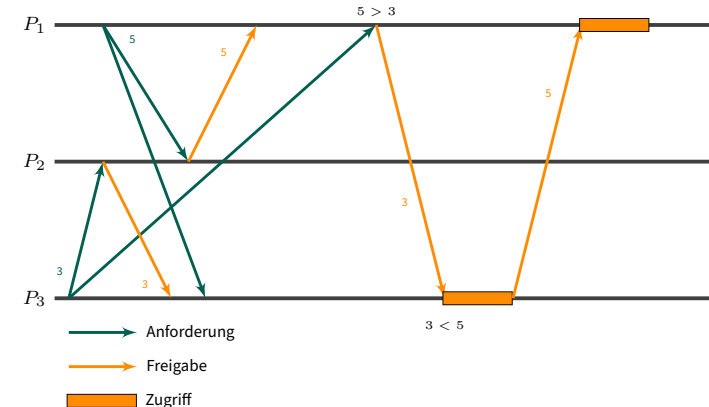
Broadcast Algorithmus von LAMPORT, 1978 (Forts.)

- ▶ Früheste Anforderung ist **global eindeutig**, nachdem von allen anderen Prozessen eine Nachricht mit größerem logischen Zeitstempel empfangen wurde
- ▶ **Nachrichtenkomplexität:**
 - ▶ Versenden der Anforderung an $(n - 1)$ Prozesse
 - ▶ $(n - 1)$ Prozesse versenden eine Bestätigung
 - ▶ Versenden der Freigabe an $(n - 1)$ Prozesse
 - ➡ $3(n - 1)$ Nachrichten pro Zugriff insgesamt ➡ bei Broadcast-Nachrichten gleich gut wie zentralisierte Lösung, sonst schlechter
- ▶ **Abstimmungsverzögerung:**
 - ▶ eine Nachrichtenlaufzeit ➡ besser als bei zentralisierter Lösung

Verbesserter Broadcast Algorithmus

- ▶ RICART und AGRAWALA, 1981
- ▶ Benötigt keine FIFO-Kanäle
- ▶ Benötigt nur $2(n - 1)$ Nachrichten pro Zugriff
- ▶ **Grundidee:**
 - ▶ **Anforderung** (mit Zeitstempel) an alle anderen $(n - 1)$ Prozesse senden
 - ▶ Zugriff, nachdem $n - 1$ Einwilligungen **für diese Anforderung** erhalten wurden
 - ➔ jede Anforderung muss eine eindeutige ID haben
- ▶ **Bei Eintreffen einer Anfrage:**
 - ▶ Einwilligung **sofort** schicken, wenn
 - ▶ wenn nicht selbst beworben
 - oder*
 - ▶ Sender „ältere Rechte“ (erkennbar am logischen Zeitstempel) hat
 - ▶ Bei gleichen Zeitstempeln entscheidet die Knoten-ID
 - ▶ Ansonsten Einwilligung erst **nach Beendigung** des eigenen Zugriffs schicken

Verbesserter Broadcast Algorithmus (Forts.)



- Nummern an der Einwilligungsnachricht zeigen Zeitstempel der zugehörigen Anforderung, nicht der Nachricht selbst

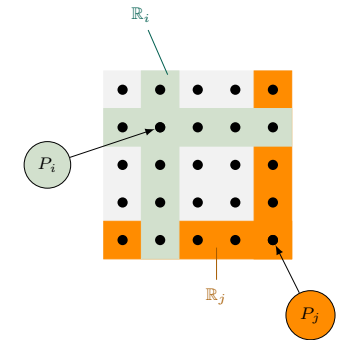
8.3 Quorum-basierende Algorithmen

Bessere Algorithmen?

- Ist eine Lösung möglich, die mit
 - **weniger Nachrichten** pro Zugriff auskommt und
 - die trotzdem die **Last gleichmäßig** auf alle Prozesse verteilt?
- Gibt es eine Lösung, bei der
 - **nicht alle Prozesse** an jeder Koordination beteiligt werden und
 - die trotzdem die **Last gleichmäßig** auf alle Prozesse verteilt?

Prozessgitter-Algorithmus

- MAEKAWA, 1985
- Die n Prozesse werden in einem quadratischen Gitter der Maße $\sqrt{n} \times \sqrt{n}$ angeordnet
- Ein Prozess P_i muss eine bestimmte Menge von Prozessen (seine **Bewilligungsmenge** $\mathbb{R}_i$) vor dem Zugriff um Erlaubnis fragen
- Für alle Paare von Prozessen P_i und P_j sind deren $\mathbb{R}_i$ und $\mathbb{R}_j$ so angeordnet, dass sie mindestens zwei Prozesse gemeinsam haben



Prozessgitter-Algorithmus (Forts.)

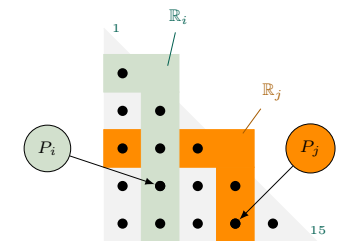
- Bewilligungsmengen haben¹ die Mächtigkeit $2\sqrt{n} - 1$
- **Nachrichtenkomplexität:**
 - Anfrage an $2(\sqrt{n} - 1)$ Prozesse senden
 - $2(\sqrt{n} - 1)$ Prozesse senden Einwilligung
 - Freigabe an $2(\sqrt{n} - 1)$ Prozesse senden
 - $6(\sqrt{n} - 1)$ Nachrichten insgesamt pro Zugriff
- **Problem:** Es können Verklemmungen auftreten
 - Durch die Einführung weiterer Nachrichtentypen vermeidbar
 - Erhöht die Nachrichtenkomplexität nur um konstanten Faktor

Gibt es eine andere Anordnung der Prozesse, bei der die Mächtigkeit der Bewilligungsmengen kleiner ist?

¹inklusive des „Antragsstellers“

Dreieckige Anordnung

- In einem quadratischen Gitter haben zwei verschiedene Bewilligungsmengen mindestens je zwei Prozesse gemeinsam
- Ein **einzig** gemeinsamer Prozess wäre aber ausreichend!
- **Lösung:**
 - Prozesse werden in einem Dreieck angeordnet
 - Bewilligungsmengen haben ungefähr die Mächtigkeit $\sqrt{2n}$
- **Problem:** Einwilligung mancher Prozesse wird häufiger benötigt als die anderer!
 - Prozess 15 kommt nur in **einer** Bewilligungsmenge vor
 - Prozess 1 kommt in **neun** Bewilligungsmengen vor



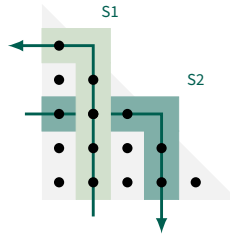
Bildung der Bewilligungsmenge

Gleiche Spalte und die Zeile eins überhalb als oberster Knoten der Spalte

Lösung zur Lastbalancierung

► Ansatz: Zwei unterschiedlichen Schemata

- S1: Gleiche Spalte und Zeile eins über obersten Knoten der Spalte (nach oben und nach links)
- S2: Gleiche Zeile und Spalte eins weiter rechts als rechterster Knoten der Zeile (nach rechts und nach unten)

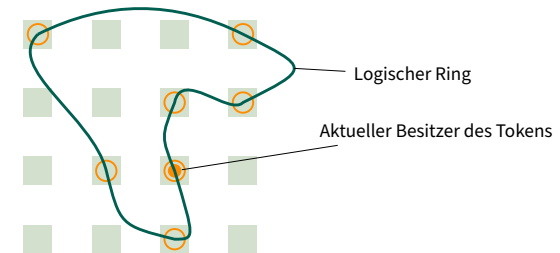


► Eigenschaften

- Jede Bewilligungsmenge überschneidet sich mit jeder Bewilligungsmenge des **gleichen** Schemas
- Jede Bewilligungsmenge des **einen** Schemas überschneidet sich mit jeder Bewilligungsmenge des **anderen** Schemas
- Alle Prozesse kommen **gleich oft** in S1US2 vor
- Alternierende (oder auch zufällige) Nutzung der beiden Schemata
→ Lastbalancierung

8.4 Token-basierte Verfahren

- **Topologie:** Die am gegenseitigen Ausschluss beteiligten Knoten bilden eine bestimmte Topologie, z.B. einen logischen Ring
 - Alternativ wird tatsächliche Topologie genutzt
- Es gibt genau eine **Zugriffsberechtigung (Token)**, die entlang der Topologie weitergegeben wird



Einfache Token Ring-Lösung

- LE LANN, 1977
- Prozesse werden in logischem Ring angeordnet
- Eigenes Token für jede Ressource
- Zugriff wird durch zirkulierendes Token gesteuert
- Bewerber wartet mit Zugriff, bis ihn das Token erreicht
- Zugreifender Prozess leitet bei Freigabe das Token weiter
- Prozess ohne Zugriffsabsicht leitet das Token direkt weiter

Einfache Token Ring-Lösung (Forts.)

- **Vorteile**
 - Einfacher, korrekter, fairer Algorithmus
 - Keine Verklemmungen
 - Kein Verhungern
 - Prioritäten sind möglich
- **Probleme**
 - Token ist ständig unterwegs, unter Umständen nutzlos
 - Nachrichtenzahl pro Anforderung nicht beschränkt
 - Bei großer Anzahl von Prozessen lange Wartezeit

Lift-Algorithmus

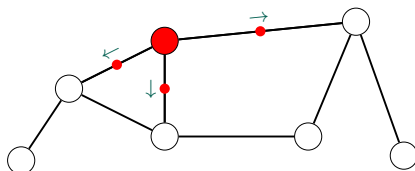
- **Alternative Idee:** Verwendung eines **Spannbaums**
- **Vorteil:** Keine nebenläufigen Pfade
- **Problem:** Wie erhält man einen Spannbaum
→ **Echo-Algorithmus**

Exkurs: Echo-Algorithmus

- Der Echo-Algorithmus wird zur **Konstruktion von Spannäumen** genutzt
- Er kann auf beliebigen, zusammenhängenden Kommunikationstopologien eingesetzt werden
- Angewendet u.a. in Algorithmen zu
 - Gegenseitigem Ausschluss (hier 😊)
 - Terminierungsproblem
 - Auswahlproblem

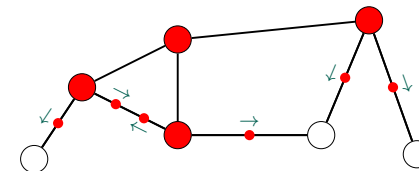
Exkurs: Echo-Algorithmus (Forts.)

- Zustand von Knoten wird durch Farben beschrieben
- Initial sind alle Knoten **weiß**
- Der eindeutige Initiator wird **rot** und schickt rote Nachrichten (**Explorer**) an alle seine Nachbarn



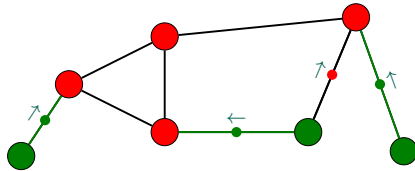
Exkurs: Echo-Algorithmus (Forts.)

- Ein weißer Knoten, der einen Explorer empfängt, wird selber rot, merkt sich diese „erste“ Kante (**Aktivierungskante**) und schickt Explorer an alle seine anderen Nachbarn
- Wird ein Explorer von einem roten Knoten empfangen, wird der Explorer nicht weitergeleitet („verschluckt“)



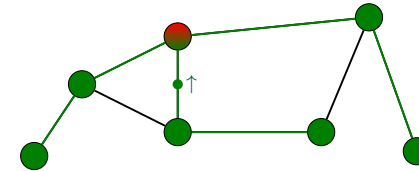
Exkurs: Echo-Algorithmus (Forts.)

- ▶ Ein roter Knoten, der über **alle** seine Kanten einen Explorer oder ein Echo erhalten hat wird **grün** und sendet ein grünes **Echo** über seine Aktivierungskante, die auch grün wird
- ▶ **Blätter** senden beim Empfang eines Explorers sofort ein Echo zurück



Exkurs: Echo-Algorithmus (Forts.)

- ▶ Nach und nach werden alle Knoten und ein Teil der Kanten grün
- ▶ Der Algorithmus terminiert, wenn der Initiator über alle seine Kanten ein Echo oder einen Explorer empfangen hat

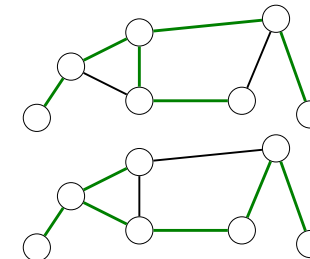


Echo-Algorithmus – Eigenschaften

- ▶ Über jede Kante laufen genau zwei Nachrichten
 - ▶ Entweder ein Explorer und ein gegenläufiges Echo oder zwei gegenläufige Explorer
- ▶ Paralleles Traversieren eines (zusammenhängenden ungerichteten) Graphen mit $2e$ Nachrichten
- ▶ Der Echo-Algorithmus ist ein **Wellenalgorithmus**
 - ▶ **Hinwelle:** Rot werden
 - ➔ Verteilen einer Information an alle Knoten
 - ▶ **Rückwelle:** Grün werden
 - ➔ Einsammeln von Information von allen Knoten

Echo-Algorithmus – Eigenschaften (Forts.)

- ▶ Echo-Kanten bilden einen **Spannbaum**
- ▶ Je nach Nachrichtenlaufzeit sieht der Spannbaum anders aus, weil schnelle gegenüber langsamen Kanten bevorzugt werden



Verbesserung des Echo-Algorithmus?

- ▶ **Idee:** Vermeide den Besuch von Knoten, von denen bekannt ist, dass sie von anderen Explorern besucht werden
- ▶ Zusammen mit einem Explorer wird eine Menge von Tabuknoten Z verschickt und empfangen
- ▶ Die vom Initiator verschickte Tabumenge ist $Z = \langle \text{Nachbarn vom Initiator} \rangle \cup \langle \text{Initiator} \rangle$
- ▶ Explorer nur an die Menge von Nachbarn Y schicken, die nicht in Z sind
 - ▶ Dabei wird die neue Tabumenge $Z' = Z \cup Y$ angehängt
- ▶ **Vorteil:** Einsparung von Nachrichten
- ▶ **Nachteile:**
 - ▶ Nachrichtenlänge $\mathcal{O}(n)$
 - ▶ Nachbaridentität muss bekannt sein

Lift-Algorithmus (Forts.)

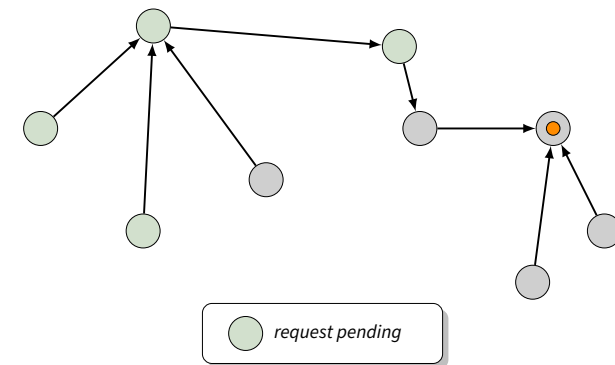
- ▶ Empfängt ein Prozess das Token...
 - ▶ darf er bei eigener Anforderung zugreifen
 - ▶ leitet er es in **eine** der anfordernden Richtungen weiter
 - ▶ gibt es noch weitere (gemerkte) Anforderungen aus anderen Richtungen, so schickt er dem Token eine Anforderung **hinterher**
- ▶ Um Fairness zu gewährleisten, darf ein Prozess **nicht beliebig oft** eine gleiche anfordernde Richtung nicht bedienen

Lift-Algorithmus

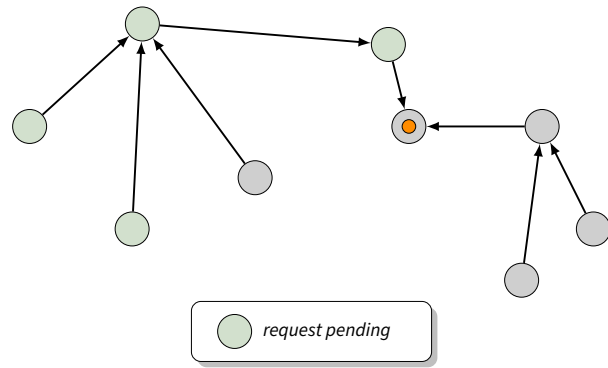
Zurück zum Lift-Algorithmus: Nutzung des Spannbaums

- ▶ Kanten des Spannbauums haben einen **Zustand** (Richtung)
 - ▶ Sie können jeweils in eine von zwei Richtungen zeigen
- ▶ Selektive Weiterleitung von Zugriffsanforderungen in Richtung Token (anstatt Senden an alle anderen Prozesse)
 - ▶ Token wandert **entgegen der Kantenrichtung** und **dreht** dabei die Richtung jeder durchlaufenen Kante um
 - ▶ Prozess, der Token will, sendet Anforderung über seine **ausgehende** Kante
- ▶ Hat ein Prozess Anfrage erhalten, sendet er **einmalig** dem Token eine Anforderung hinterher
 - ▶ Weitere Anfragen werden „gemerkt“

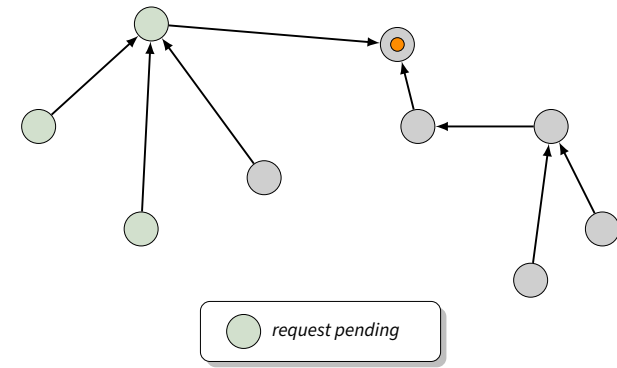
Lift-Algorithmus (Forts.)



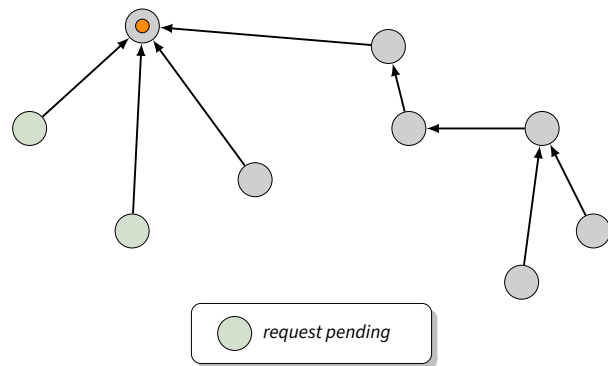
Lift-Algorithmus (Forts.)



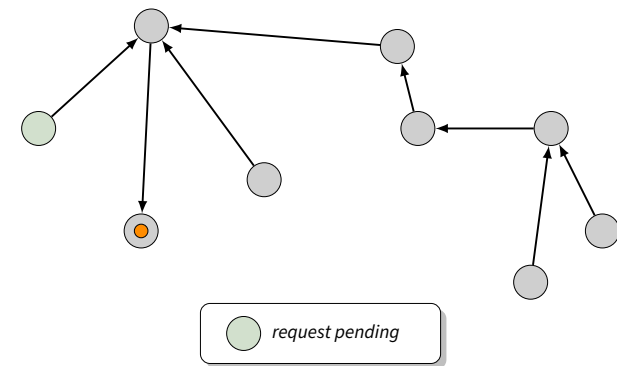
Lift-Algorithmus (Forts.)



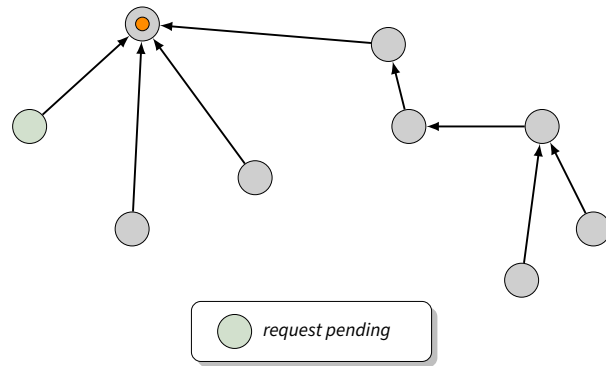
Lift-Algorithmus (Forts.)



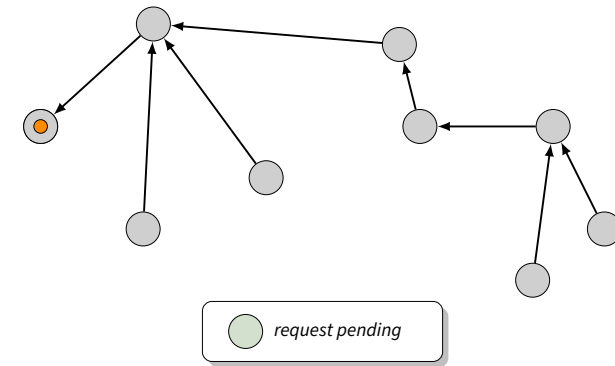
Lift-Algorithmus (Forts.)



Lift-Algorithmus (Forts.)



Lift-Algorithmus (Forts.)



Lift-Algorithmus (Forts.)

- **Invariante:** Von jedem Prozess aus führt ein gerichteter Pfad zum Token
- In einem k -ären balancierten Baum ist die maximale Weglänge zwischen beliebigen Prozessen $\mathcal{O}(\log_k n)$
- Entsprechend werden nur $\mathcal{O}(\log_k n)$ Nachrichten pro Zugriff gebraucht
- **Startzustand:** Sieger einer Auswahl (siehe später) bekommt das Token und startet einen Echo-Algorithmus, um einen Spannbaum mit auf ihn gerichteten Kanten zu erzeugen
- Verfahren lässt sich so auf beliebige zusammenhängende Topologien verallgemeinern

Literatur

- 📖 [RiAg81] Ricart, G. und A. K. Agrawala: „An Optimal Algorithm for Mutual Exclusion in Computer Networks“. *Communications of the ACM*, 24(1)1981, 9–17
- 📖 [Mae85] Maekawa, M.: „ $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems“. *ACM Transactions on Computer Systems*, 3(2)1985, 145–159
- 📖 [LuWo97] Luk, W. S. und T. T. Wong: „Two New Quorum Based Algorithms for Distributed Mutual Exclusion“, in *17th International Conference on Distributed Computing Systems (ICDCS '97)*, 100–107
- 📖 [SuKa85] Suzuki, I. und T. Kasami: „A distributed mutual exclusion algorithm“. *ACM Transactions on Computer Systems*, 3(4)1985, 344–349