Verteilte Algorithmen und Datenstrukturen

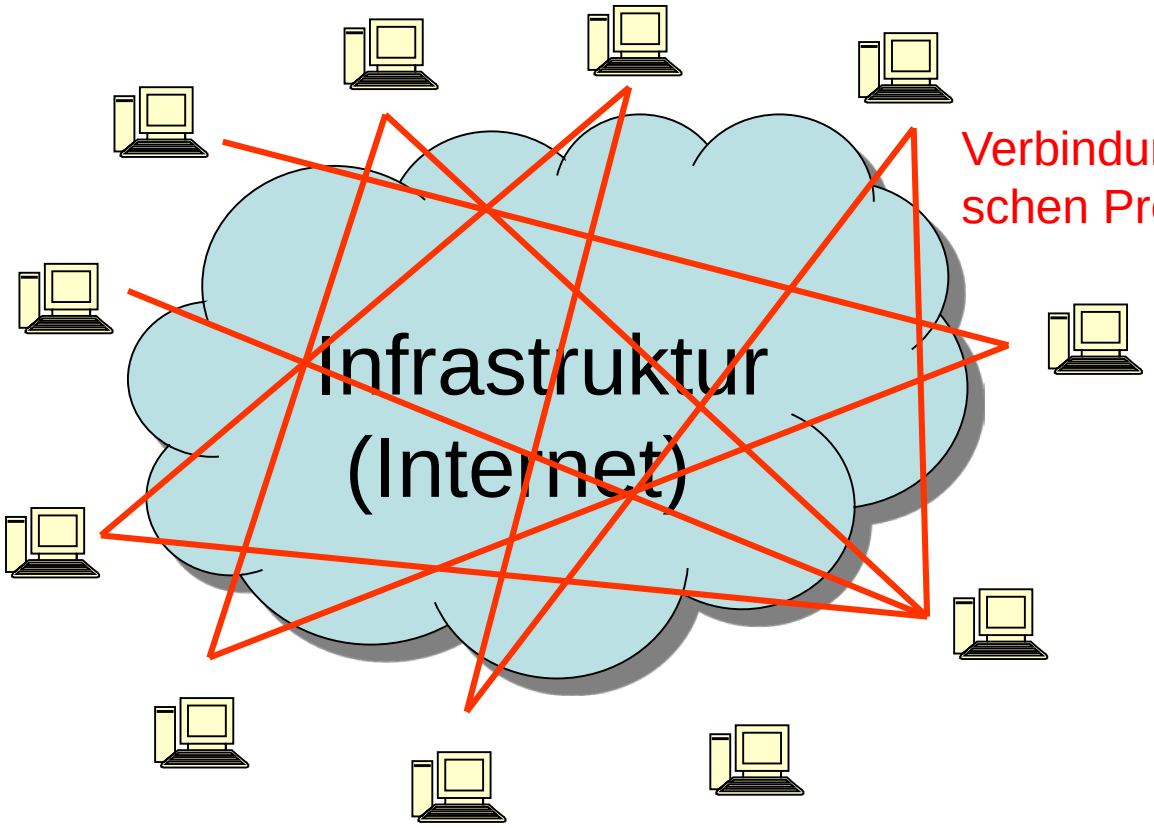
Kapitel 3: Designprinzipien für verteilte Algorithmen und Datenstrukturen

Prof. Dr. Christian Scheideler

Insitut für Informatik

Universität Paderborn

Verteilte Systeme



Korrektheit, Skalierbarkeit, Robustheit,...

Robustheit

Probleme:

- Prozesse arbeiten unterschiedlich schnell oder sind gar temporär offline
- Neue Prozesse kommen hinzu, alte Prozesse möchten das System verlassen oder sind fehlerhaft.
- Botschaften brauchen unterschiedlich lange oder gehen verloren.
- Gegnerische Angriffe

Robustheit

Verfügbarkeit ist alles!

→ Entkopplung von Zeit und Fluss.

- Zeitentkopplung: interagierende Prozesse müssen nicht zur selben Zeit miteinander interagieren
- Flussentkopplung: die Ausführung einer Aktion innerhalb eines Prozesses sollte nicht von anderen Prozessen abhängen

Entkopplung von Zeit und Fluss

Flussentkopplung:

- Nur asynchrone Kommunikation, d.h. Anfragen von Prozess A an Prozess B erfordern keine sofortige Antwort.

- Alle für die Ausführung einer Aktion in A notwendigen Variablen sind lokal zu A.

Zeitentkopplung: Die Auslieferung von Anfragen geschieht nebenläufig zur Bearbeitung von Aktionen, d.h. ihre Bearbeitung wird durch diese nicht behindert.

Robustheit

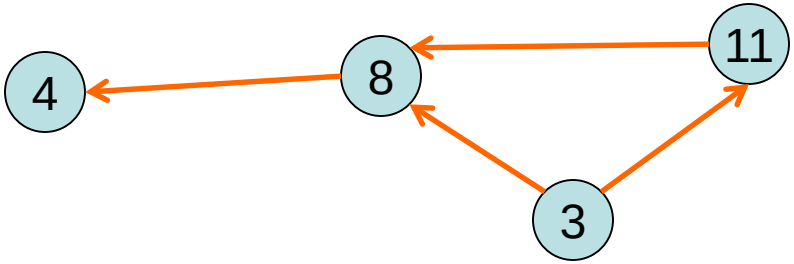
Verfügbarkeit ist alles!

→ Selbsterholung von jedem Zustand, von dem aus das möglich ist.

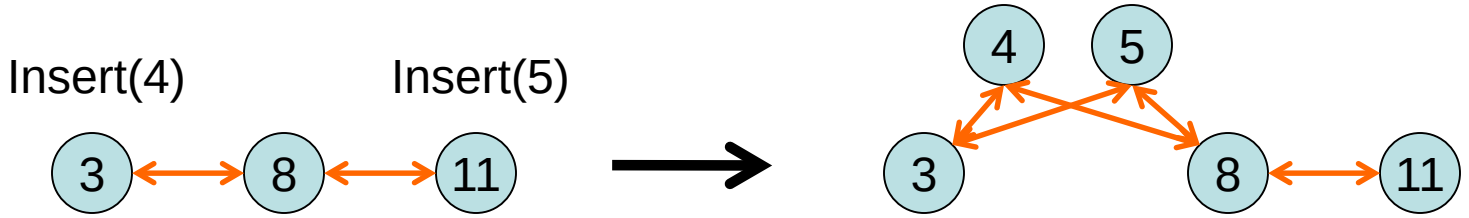
- Idealerweise: Fehler und gegnerisches Verhalten können lokal erkannt und korrigiert werden, so dass sich die betroffenen Teile wieder davon erholen können während die Teile des Systems, die davon nicht unmittelbar betroffen sind, weiterhin verfügbar bleiben.

Warum Selbsterholung?

- In verteilten Datenstrukturen können Fehler auftreten.

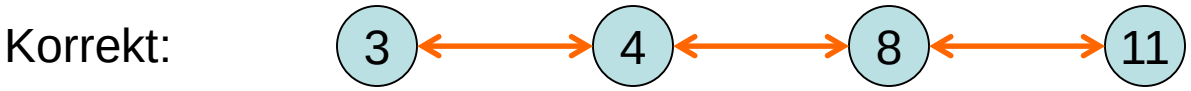


- In verteilten Datenstrukturen können mehrere Operationen gleichzeitig ausgeführt werden.

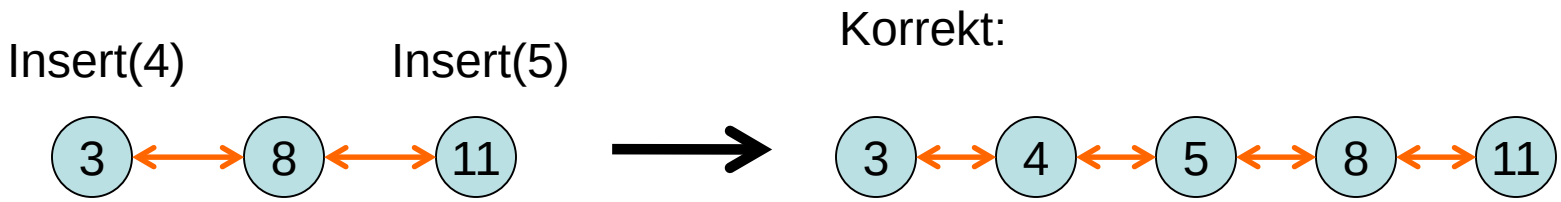


Warum Selbsterholung?

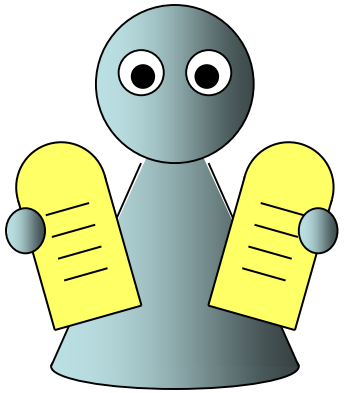
- In verteilten Datenstrukturen können Fehler auftreten.



- In verteilten Datenstrukturen können mehrere Operationen gleichzeitig ausgeführt werden.



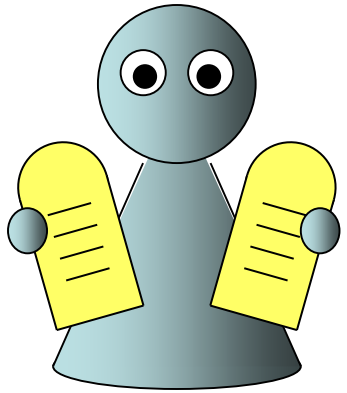
Gesetze der Robustheit



Entkopplung und Selbsterholung
von jedem möglichen Zustand

Nur dann können verteilte Systeme
hochgradig skalierbar und verfügbar sein

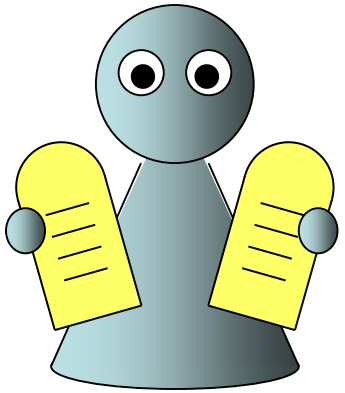
Gesetze der Robustheit



Vollständige Kontrolle über Ressourcen und Informationen

Nur dann können verteilte Prozesse sicher ausgetauscht werden und miteinander interagieren

Gesetze der Robustheit



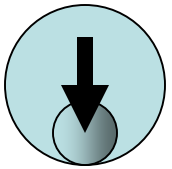
1. Eignerzustimmung und Kontrolle
2. Geringste Ausgesetztheit
3. Selbsterholung
4. Entkopplung

[siehe auch POLA, K. Cameron: The laws of identity, D. Epp: The eight rules of security,…]

Gesetze der Robustheit

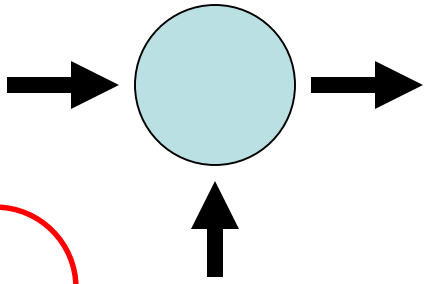
Eignerzustimmung und Kontrolle:

- Eindeutig definierte Zuständigkeiten
- Vollständige Kontrolle über Info & Ressourcen



Geringste Ausgesetztheit:

- Nicht mehr Wissen als notwendig
- Vollst. Kontrolle über Informationsfluss



Selbsterholung:

- Erholung muss von jedem Zustand aus möglich sein (solange die Plattform noch im legalen Zustand ist)

Entkopplung:

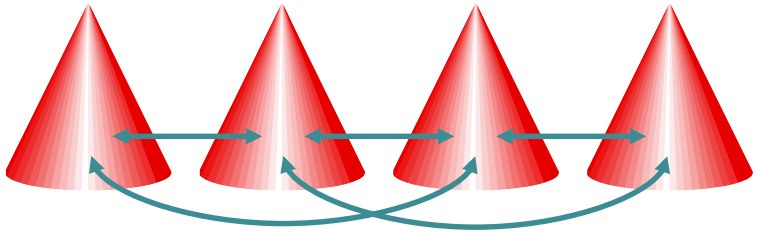
- keine Synchronisation notwendig für Primitive

diese Vorlesung

Gesetze der Robustheit

Wie können wir Selbsterholung und Entkopplung rigoros studieren?

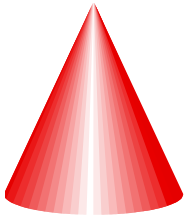
Hier: prozessorientierte Datenstrukturen



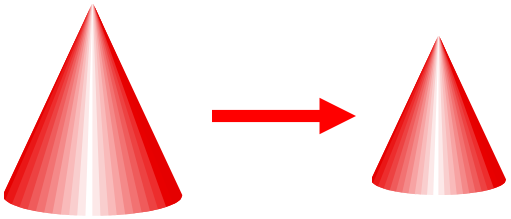
Designprinzipien für verteilte Systeme

- Prozessmodell und Pseudocode
- Netzwerkmodell und zulässige Verbindungsprimitive
- Selbststabilisierung
- Konsistenzmodelle

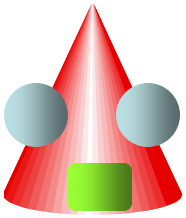
Prozessmodell

- Prozesse 
- Objekte  (repräsentieren Daten)
- Aktionen 

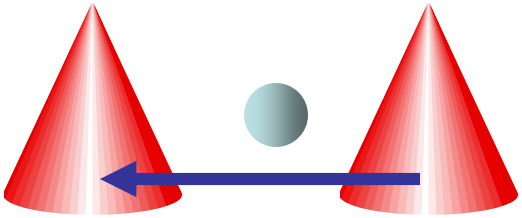
Prozessmodell



Prozess kann Kindprozesse (am selben Ort wie Mutter) erzeugen,



kann Objekte besitzen und Aktionen sequentiell ausführen, und

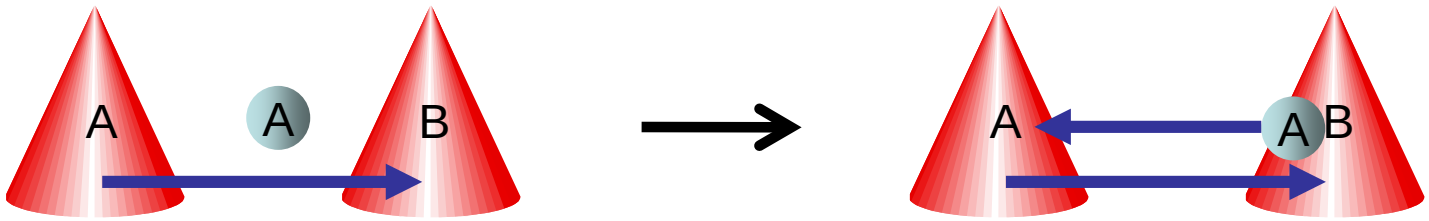


kann Verbindungen auf- und abbauen.

Prozessmodell

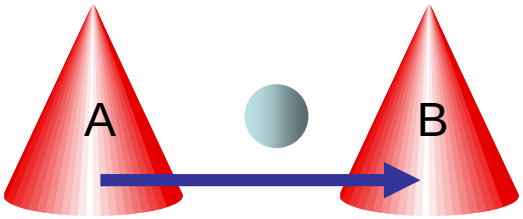
- Prozesse können beliebig parallel zueinander Aktionen ausführen, aber innerhalb eines Prozesses werden Aktionen streng sequentiell abgearbeitet.
  → jede Aktion muss terminieren!
- Prozesse verbinden sich, indem sie Referenzen austauschen

Beispiel: Prozess A erzeugt Referenz von sich und schickt diese zu Prozess B, so dass B mit A kommunizieren kann.



Prozessmodell

- Wir nehmen an, dass die Netzwerkschicht zuverlässig ist, d.h. jede Botschaft wird unverändert in endlicher Zeit ausgeliefert.
- ABER: Auslieferungszeit nicht vorhersehbar (mag sogar durch Gegner kontrolliert werden)

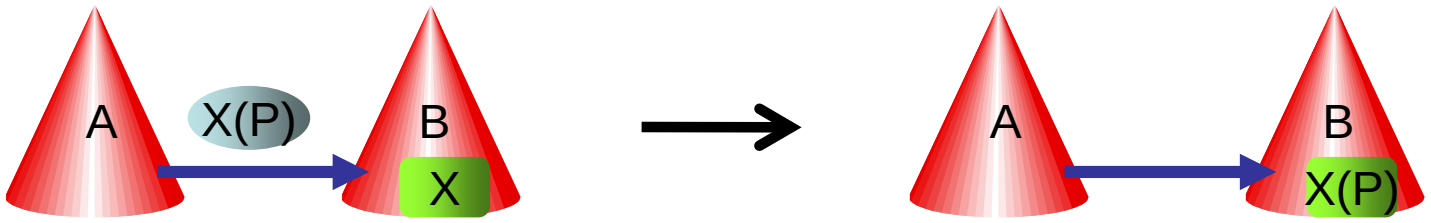


- Fehlertoleranz: Mastervorlesung

Prozessmodell

- Aktionen werden entweder durch eine Anfrage oder einen lokalen Zustand initiiert

Beispiel: Prozess A bittet Prozess B, die Aktion X mit Parameter P auszuführen.



Prozessmodell

Formen einer Aktion:

- Getriggert duch Aufrufanfrage:
  ⟨Name⟩(⟨Objektliste⟩) → ⟨Befehle⟩
- Getriggert duch lokalen Zustand:
  ⟨Name⟩: ⟨Prädikat⟩ → ⟨Befehle⟩

- Beispiel: minimum Aktion

  minimum(x,y) →
  if x<y then m:=x else m:=y
  print(m)

  Aktion „minimum“ wird ausgeführt, sobald eine Anfrage zum Aufruf von minimum(x,y) empfangen wird.

Prozessmodell

Formen einer Aktion:

- Getriggert duch Aufrufanfrage:
  ⟨Name⟩(⟨Objektliste⟩) → ⟨Befehle⟩
- Getriggert duch lokalen Zustand:
  ⟨Name⟩: ⟨Prädikat⟩ → ⟨Befehle⟩

- Beispiel: timeout Aktion

  timeout: true →
  print(„I am still alive!“)

  In jedem Prozess werden periodisch Timeouts ausgelöst. „true“ sorgt dann dafür, dass die Aktion „timeout“ bedingungslos ausgeführt wird, sobald ein Timeout erfolgt.

Pseudocode

Wie in objektorientierter Programmierung:

Subject ⟨Subjektname⟩: // deklariert Prozesstyp
    lokale Variablen
    Aktionen

Allgemeine Formen einer Aktion:

⟨Aktionsname⟩(Objektliste) →
    Befehle in Pseudocode

⟨Aktionsname⟩: ⟨Prädikat⟩ →
    Befehle in Pseudocode

Spezielle Aktionen:

init(Objektliste) → // Konstruktur
    Befehle in Pseudocode

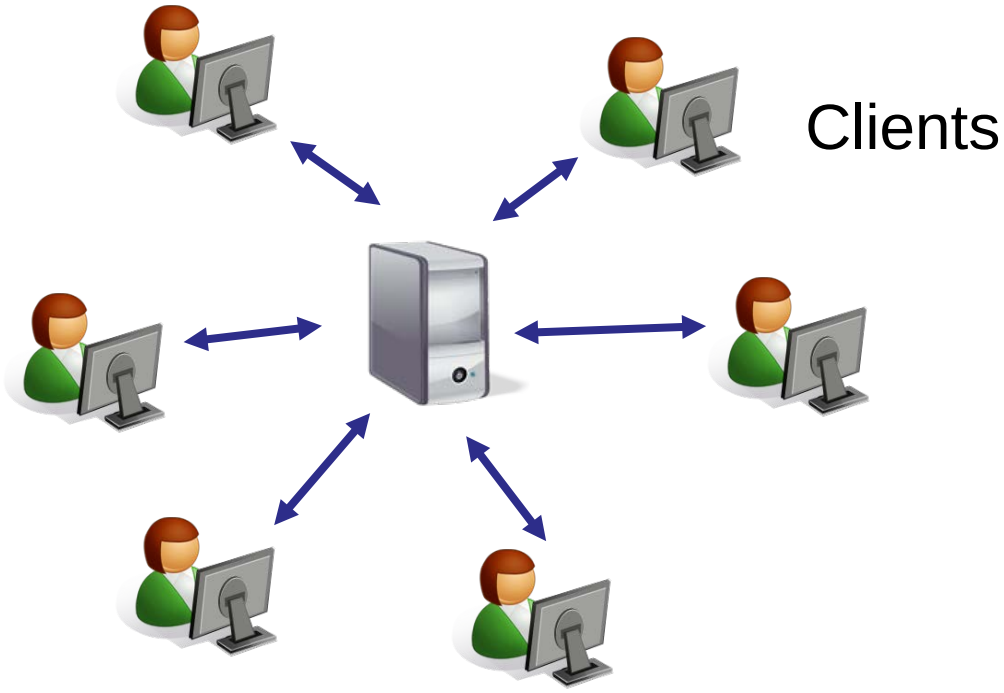
timeout → // bei jedem timeout (was periodisch erfolgt)
    Befehle in Pseudocode

Pseudocode

- Zuweisung durch :=
- Schleifen (for, while, repeat)
- Bedingtes Verzweigen (if – then – else)
- (Unter-)Programmaufruf/Übergabe (return)
- Kommentar durch { }
- Blockstruktur durch Einrückung
- Aufruf einer Aktion über Subjektreferenz: $\leftarrow$
- Referenzvariable leer: $\perp$, Menge leer: $\varnothing$
- Erzeugung neuer Subjekte und Objekte: new
  (new aktiviert init im Subjekt)

Beispiel

Einfacher Broadcast Service über Server



Broadcast Service

```
Subject Server:
    n: Integer              { aktuelle Anzahl Clients }
    Client: Array[1..MAX] of Subject    { für Subjektreferenzen}

    init() →                { Konstruktor }
       n:=0

    register(C) →           { registriere neuen Client mit Referenz C }
       n:=n+1
       Client[n]:=C

    broadcast(M) → { schicke M an alle Clients }
       for i:=1 to n do
          M´:=new Object(M)    {neues Objekt mit Inhalt von M}
          Client[i]←output(M´)
```

Broadcast Service

```
Subject Client:
   Server: Subject

   init(S) →                    { Konstruktor }
      Server:=S                 { S: Referenz auf Server }
      Server←register(this)
                                { eigene Ref. an Server }

   broadcast(M) →               { verteile M über Server }
      Server←broadcast(M)

   output(M) →                  { gib M aus }
      print M
```

Programmieransatz

- Ursprung: Hewitts Actor Modell (1973) für neuronale Netzwerke
- Seitdem verschiedene Arbeiten im Bereich der Programmiersprachen (E, Scala, Erlang,…)
- Wir werden eine Java-Erweiterung zur Umsetzung des Pseudocodes verwenden.

Beispiel in Java (Kapitel 4)

```
public class Hello extends Subject {
    public Hello() { }

    protected void init() {
        println("Hello World!");
    }

    protected void onMessageReceived(Object message) { }

    protected void onTimeout() { }

    public static void main(String[] args) {
        Hello hello = new Hello();
        hello.start();
    }
}
```

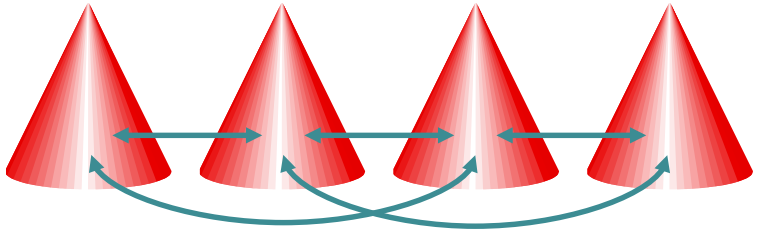
Designprinzipien für verteilte Systeme

- Prozessmodell und Pseudocode
- Netzwerkmodell und zulässige Verbindungsprimitive
- Selbststabilisierung
- Konsistenzmodelle

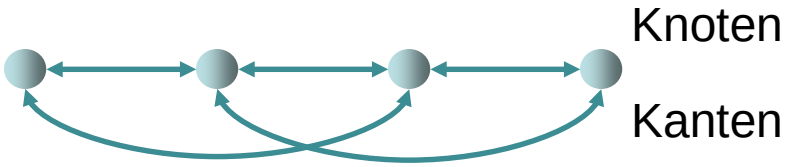
Netzwerkmodell

Modellierung der Referenzen im System als gerichteten Graph.

- Darstellung über Prozesse:



- Vereinfachte Graphendarstellung:



- Kante $A \rightarrow B$ bedeutet: A kennt B

Netzwerkmodell

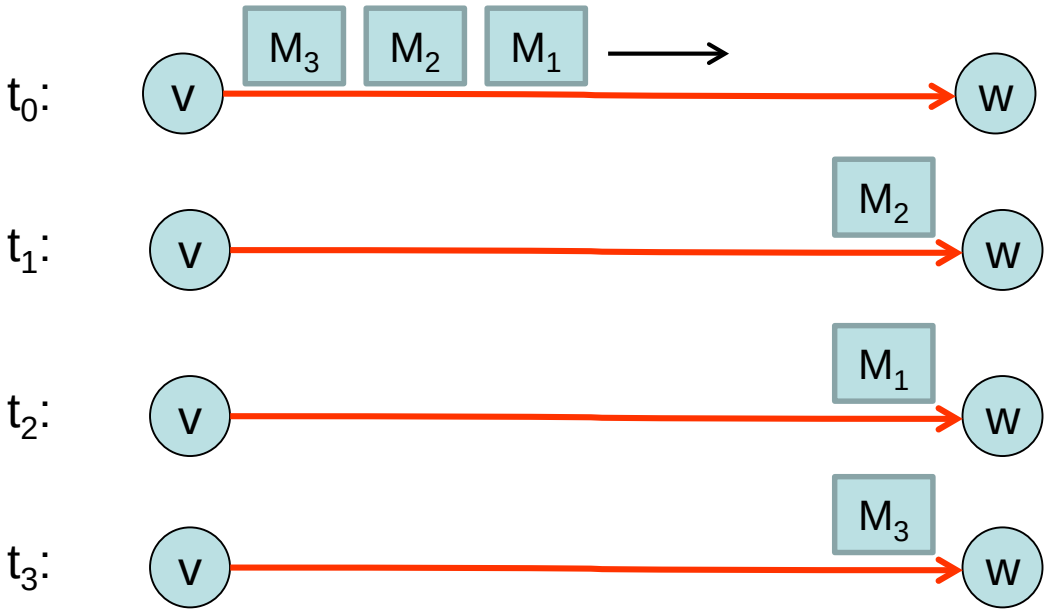
- Kantenmenge E_L: Menge aller (v,w), für die Prozess v Prozess w kennt (explizite Kanten).

- Kantenmenge E_M: Menge aller (v,w), für die eine Nachricht mit Referenz von w zu v unterwegs ist (implizite Kanten).

- Graph $G=(V,E_L \cup E_M)$: Graph aller expliziten und impliziten Kanten.

Netzwerkmodell

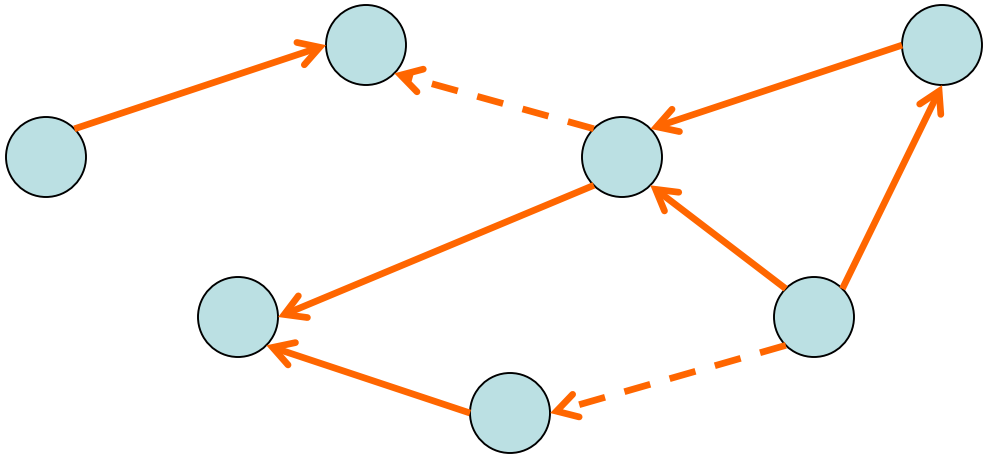
Nachrichten werden asynchron verschickt



- jede Nachricht wird in endlicher Zeit ausgeliefert, aber darüberhinaus keine Garantien wie FIFO Auslieferung

Zulässige Verbindungsprimitive

Grundlegendes Ziel: Prozessgraph (d.h. G) schwach zusammenhängend zu halten

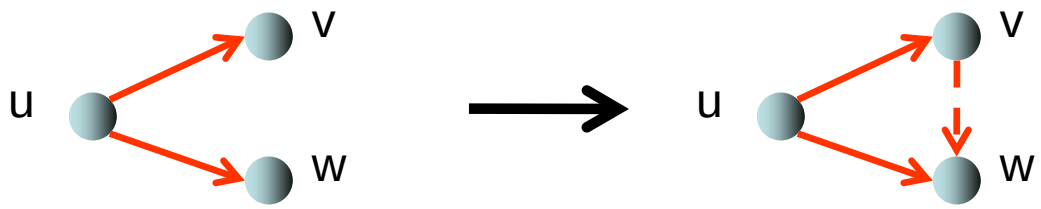


Grundlegende Regel: niemals Referenz einfach „wegwerfen“!

Zulässige Verbindungsprimitive

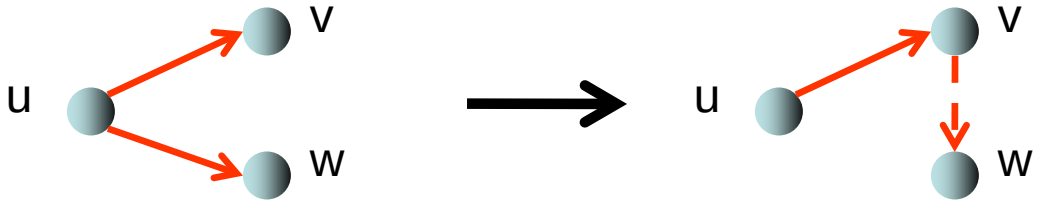
Zulässige Primitive für Prozessgraphen:

Vorstellung:



u schickt Kopie der Ref. zu w nach v

Weiterleitung:



u schickt Ref. zu w nach v

Zulässige Verbindungsprimitive

Zulässige Primitive für Prozessgraphen:

Verschmelzung:

u löscht redundante Referenz

Umkehrung:

u schickt Referenz von sich an v
und löscht dann Referenz zu v

Zulässige Verbindungsprimitive

Bemerkungen:

- Erlaubter Spezialfall für Vorstellung:

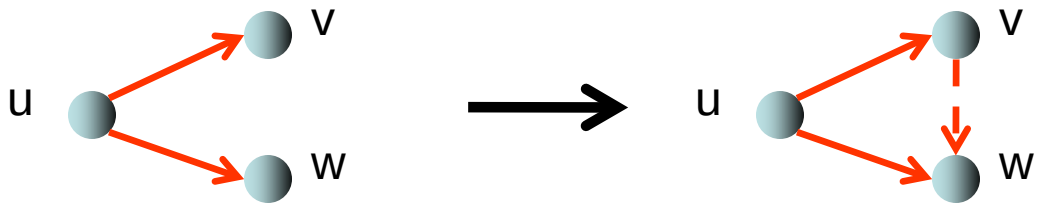
u schickt Referenz von sich an v

- Die Primitive nennen wir deshalb zulässig, da sie den schwachen Zusammenhang bewahren.
- Die Vorstellung, Weiterleitung und Verschmelzung bewahren sogar den starken Zusammenhang.
- Einzige notwendige Vergleichsoperation für Primitive: Test, ob Referenz1=Referenz2 (für Verschmelzung)

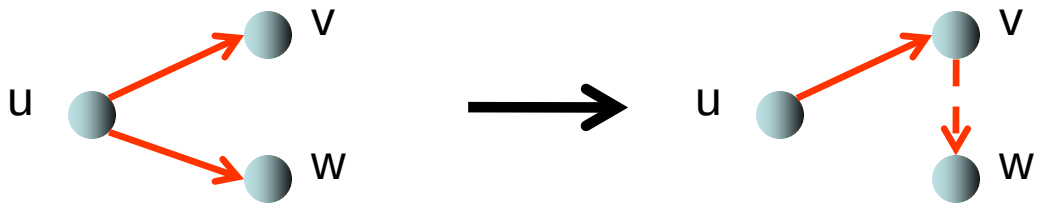
Zulässige Verbindungsprimitive

Satz 3.1: Die 3 Primitive unten sind schwach universell, d.h. man kann mit ihnen von jedem schwach zusammenhängenden Graphen G=(V,E) zu jedem stark zusammenhängenden Graphen G´=(V,E´) gelangen.

Vorstellung:



Weiterleitung:



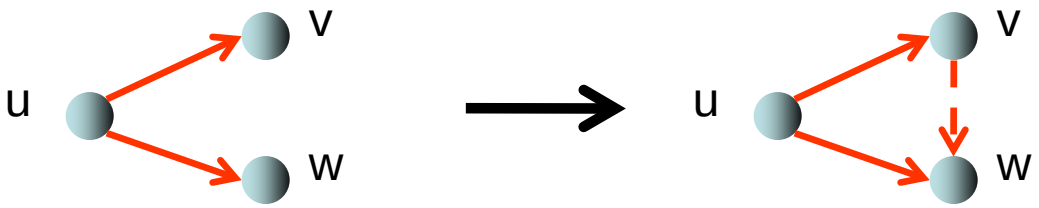
Verschmelzung:

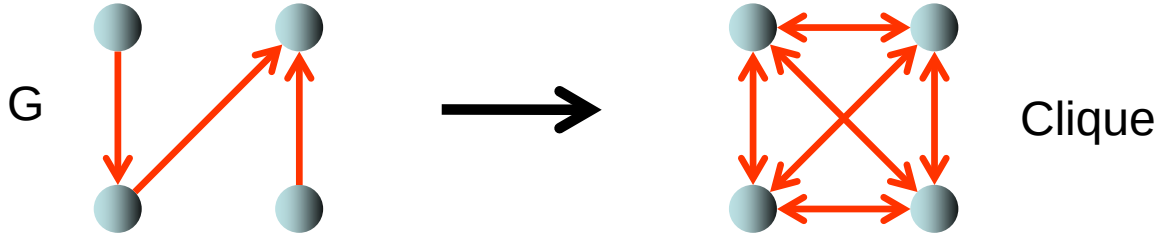
Zulässige Verbindungsprimitive

Beweis: besteht aus zwei Teilen

1. Mit der Vorstellungsregel gelangt man von jedem schwach zusammenhängenden Graphen G=(V,E) zu einer Clique.

Vorstellung:

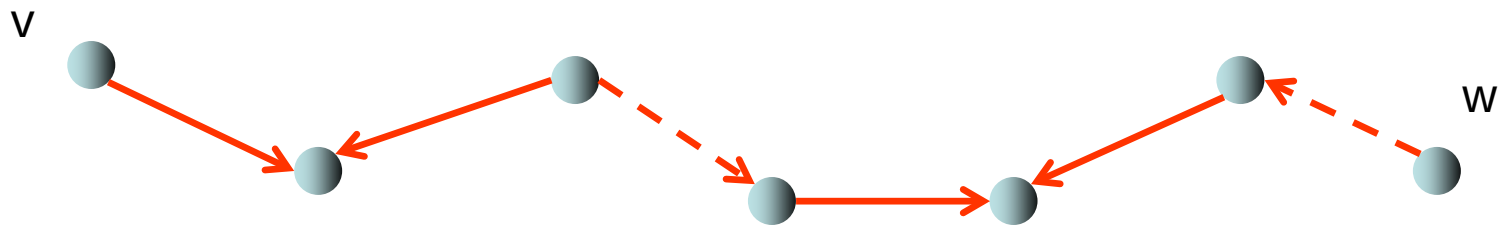




Zulässige Verbindungsprimitive

Warum funktioniert das?

Betrachte zwei Knoten v und w. Da G schwach zusammenhängend ist, gibt es einen Pfad von v nach w.

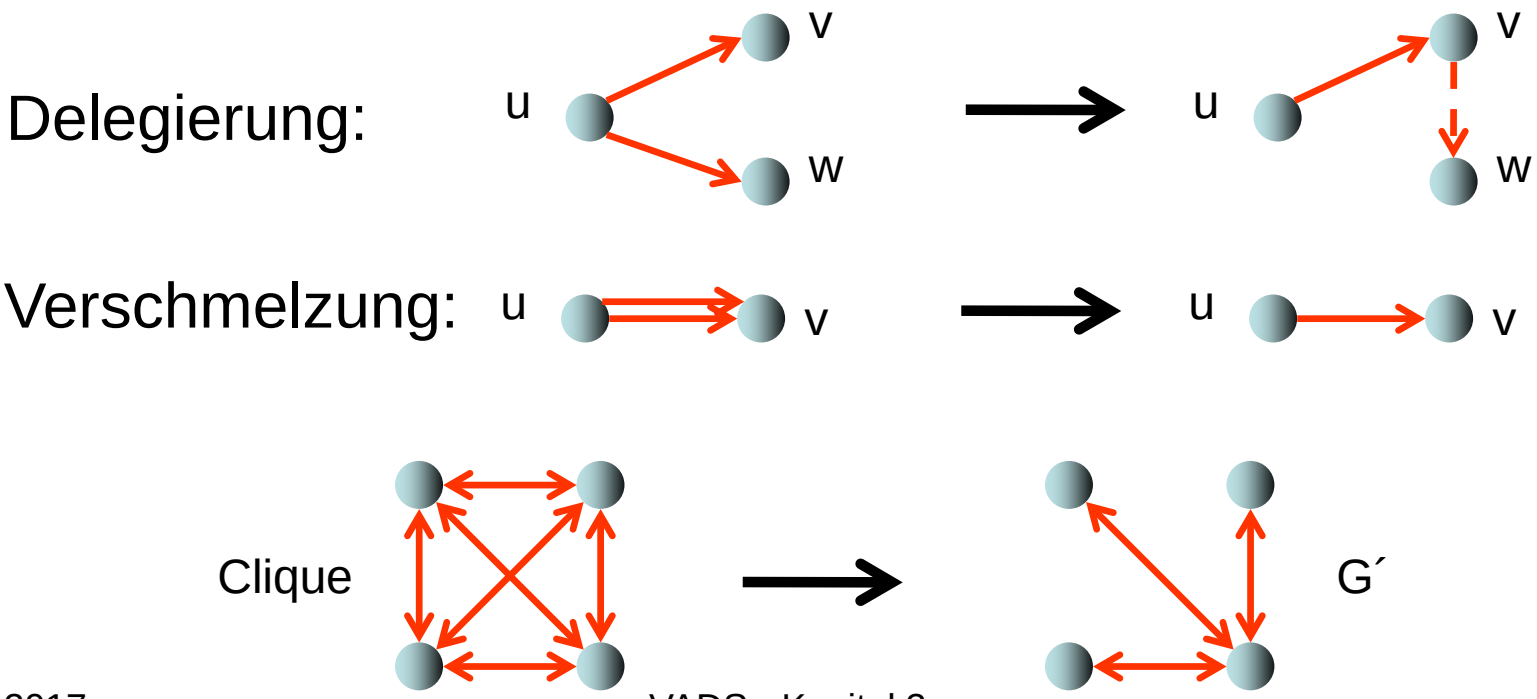


Übung: Falls in jeder Kommunikationsrunde jeder Knoten jedem seiner Nachbarn alle Nachbarn und sich selbst vorstellt, benötigt man dann nur O(log n) Kommunikationsrunden bis zur Clique.

Zulässige Verbindungsprimitive

Beweis:

2. Mithilfe der Delegierungs- und Verschmelzungsprimitive gelangt man von der Clique zu G´=(V,E´).

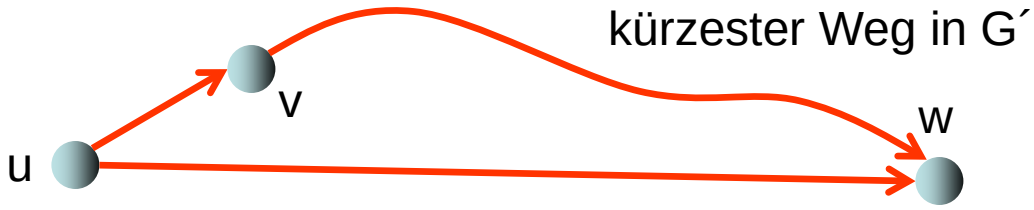


Zulässige Verbindungsprimitive

Beweis: (im Detail)

2. Angenommen, G=(V,E) ist eine Clique. Dann lässt sich G wie folgt in G´=(V,E´) transformieren (ohne Kanten in G´ zu entfernen).

- Sei (u,w) eine beliebige Kante, die noch zu entfernen ist (d.h. $(u,w) \notin E'$). Da G´=(V,E´) stark zusammenhängend ist, gibt es einen kürzesten gerichteten Weg von u nach w in G´. Sei v der nächste Knoten entlang dieses Weges.

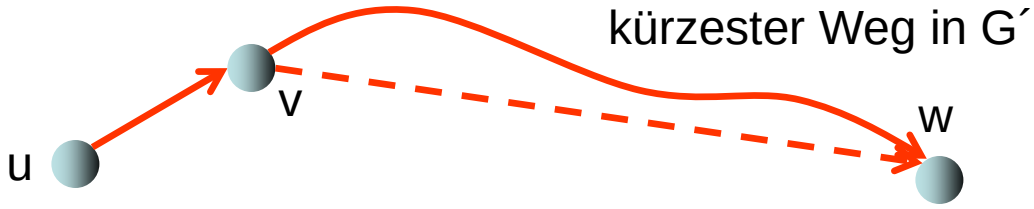


Zulässige Verbindungsprimitive

Beweis: (Fortsetzung)

2. Angenommen, G=(V,E) ist eine Clique. Dann lässt sich G wie folgt in G´=(V,E´) transformieren (ohne Kanten in G´ zu entfernen).

- Sei (u,w) eine beliebige Kante, die noch zu entfernen ist (d.h. (u,w)$\notin$E´). Da G´=(V,E´) stark zusammenhängend ist, gibt es einen kürzesten gerichteten Weg von u nach w in G´. Sei v der nächste Knoten entlang dieses Weges.
- Knoten u delegiert dann (u,w) an v weiter, d.h. aus (u,w) wird (v,w). Damit verkürzt sich die Distanz des Knotenpaares einer nicht benötigten Kante bzgl. G´ um 1.
- Da die Maximaldistanz n-1 ist, verschmilzt jede überflüssige Kante damit in höchstens n-1 Verkürzungen mit einer Kante in G´, d.h. am Ende erhalten wir G´.



Zulässige Verbindungsprimitive

Satz 3.2: Die vier Primitive Vorstellung, Weiterleitung, Verschmelzung, und Umkehrung sind universell, d.h. man kann mit ihnen von jedem schwach zusammenhängenden Graphen $G=(V,E)$ zu jedem schwach zusammenhängenden Graphen $G´=(V,E´)$ gelangen.

Beweis:

- Sei $G´´=(V,E´´)$ der Graph, in dem für jede Kante $(u,v)\in E´$ sowohl (u,v) also auch (v,u) in $E´´$ sind. Dann ist $G´´$ stark zusammenhängend.
- Nach Satz 3.1: wir können von G zu $G´´$ gelangen.
- Von $G´´$ nach $G´$: verwende Umkehrungsprimitiv, um unerwünschte Kanten loszuwerden (indem jede solche Kante (u,v) in (v,u) umgekehrt und dann mit der vorhandenen Kante (v,u) verschmolzen wird).

Zulässige Verbindungsprimitive

Bemerkung:

- Die 4 Primitive sind notwending für die Universalität.
  - Vorstellung: das einzige, das neue Kante generiert
  - Verschmelzung: das einzige, das Kante entfernt
  - Delegierung: das einzige, das Knotenpaar trennt
  - Umkehrung: das einzige, das Knoten unerreichbar macht

- Satz 3.2 zeigt lediglich, dass es im Prinzip möglich ist, von jedem schwach zusammenhängenden zu jedem anderen schwach zusammenhängenden Graphen zu gelangen.
- Unser Ziel: verteilte Verfahren dafür zu entwickeln

Designprinzipien für verteilte Systeme

- Prozessmodell und Pseudocode
- Netzwerkmodell und zulässige Verbindungsprimitive
- Selbststabilisierung
- Konsistenzmodelle

Selbststabilisierung

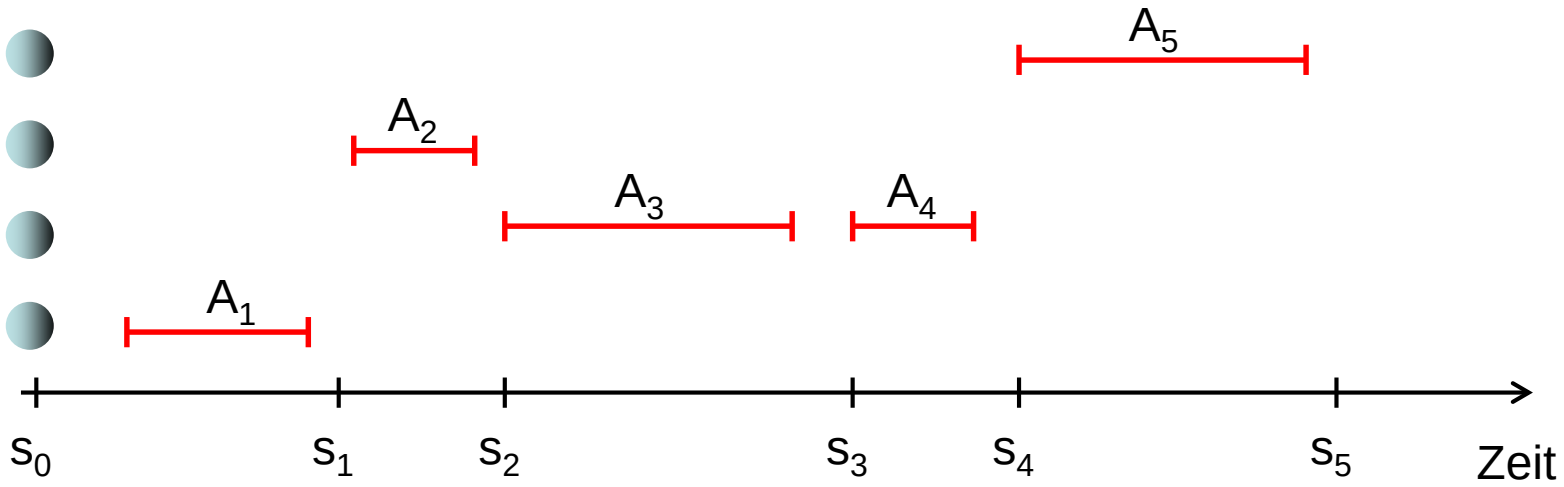
- Zustand eines Prozesses: Sämtliche variablen Informationen, die sich im Prozess befinden (was nicht die Nachrichten einschließt, die noch zu ihm unterwegs sind oder bereits abgeschickt worden sind).
- Zustand des Netzwerks: Umfasst alle Nachrichten, die zurzeit unterwegs sind.
- Zustand des Systems: Umfasst sämtliche Prozesszustände sowie den Netzwerkzustand.
- Allgemeine Formulierung eines Netzwerkproblems:
  Gegeben: initialer Systemzustand S
  Gesucht: legaler Systemzustand $S'(S)$

Beispiel: Routingproblem

- Initialer Systemzustand: Graph $G=(V,E)$ und eine Kollektion R an Quell-Ziel Paaren mit jew. einem Paket in s mit Ziel t für alle $(s,t) \in R$
- Gesucht: Systemzustand, in dem für alle $(s,t) \in R$ das entsprechende Paket in t angekommen ist.

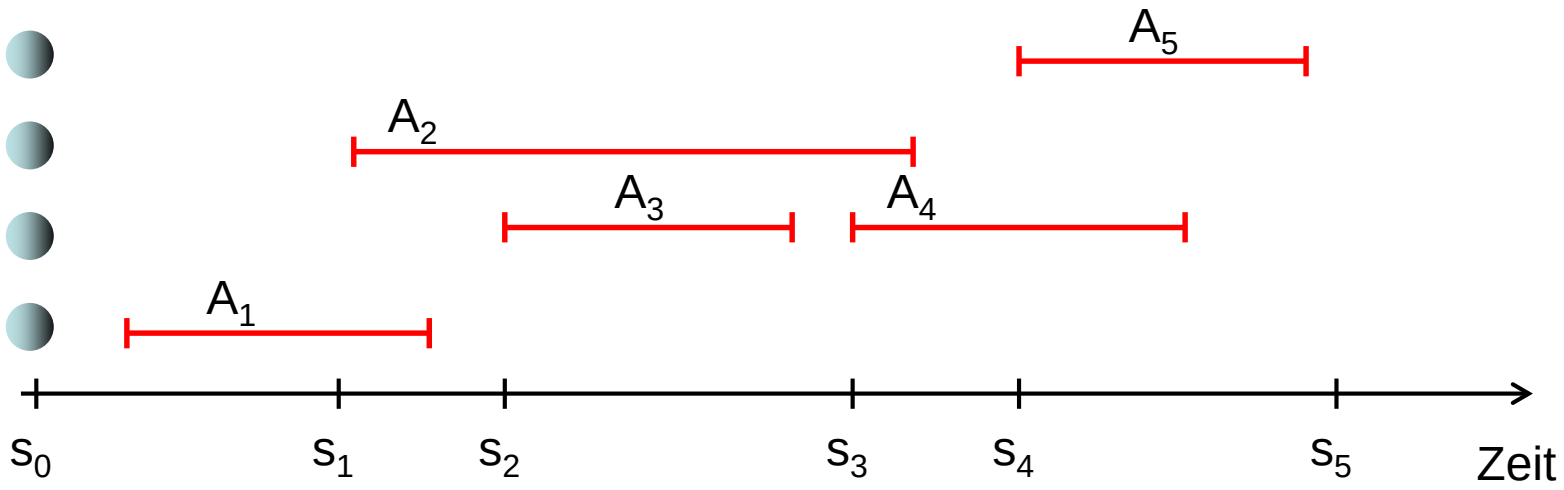
Selbststabilisierung

- Vereinfachte Annahme: zu jedem Zeitpunkt kann im gesamten System nur genau eine Aktion ausgeführt werden, d.h. wir nehmen eine global atomare Aktionsausführung an.
- Rechnung: potenziell unendliche Folge von Systemzuständen s_0, s_1, s_2,…, wobei sich Zustand s_{i+1} aus s_i duch die Ausführung einer Aktion ergibt.
- Einfach für formale Analyse, aber nicht realistisch.



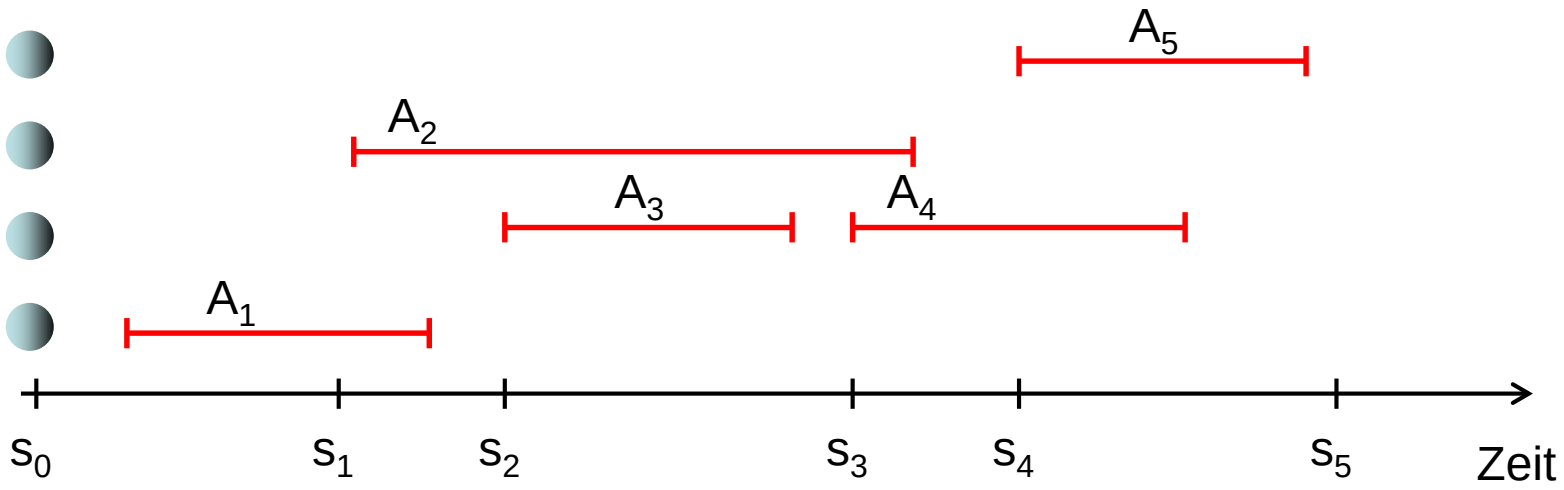
Selbststabilisierung

- Vereinfachte Annahme: zu jedem Zeitpunkt kann im gesamten System nur genau eine Aktion ausgeführt werden, d.h. wir nehmen eine global atomare Aktionsausführung an.
- Rechnung: potenziell unendliche Folge von Systemzuständen s_0, s_1, s_2,…, wobei sich Zustand s_{i+1} aus s_i duch die Ausführung einer Aktion ergibt.
- In Wirklichkeit:



Selbststabilisierung

Realistischere Annahme: zu jedem Zeitpunkt kann in jedem Prozess nur eine Aktion ausgeführt werden (d.h. Ausführung ist lokal atomar)



Selbststabilisierung

Realistischere Annahme: zu jedem Zeitpunkt kann in jedem Prozess nur eine Aktion ausgeführt werden (d.h. Ausführung ist lokal atomar)

Satz 3.3: Innerhalb unseres Prozess- und Netzwerkmodells kann jede endliche lokal atomare Aktionsausführung in eine global atomare Aktionsausführung mit demselben Endzustand transformiert werden.

→ Alle möglichen Endzustände können durch global atomare Aktionsausführungen abgedeckt werden.

→ „Keine schlechte global atomare Aktionsausführung“ impliziert „keine schlechte lokal atomare Ausführung“

Selbststabilisierung

Satz 3.3: Innerhalb unseres Prozess- und Netzwerkmodells kann jede endliche lokal atomare Aktionsausführung in eine global atomare Aktionsausführung mit demselben Endzustand transformiert werden.

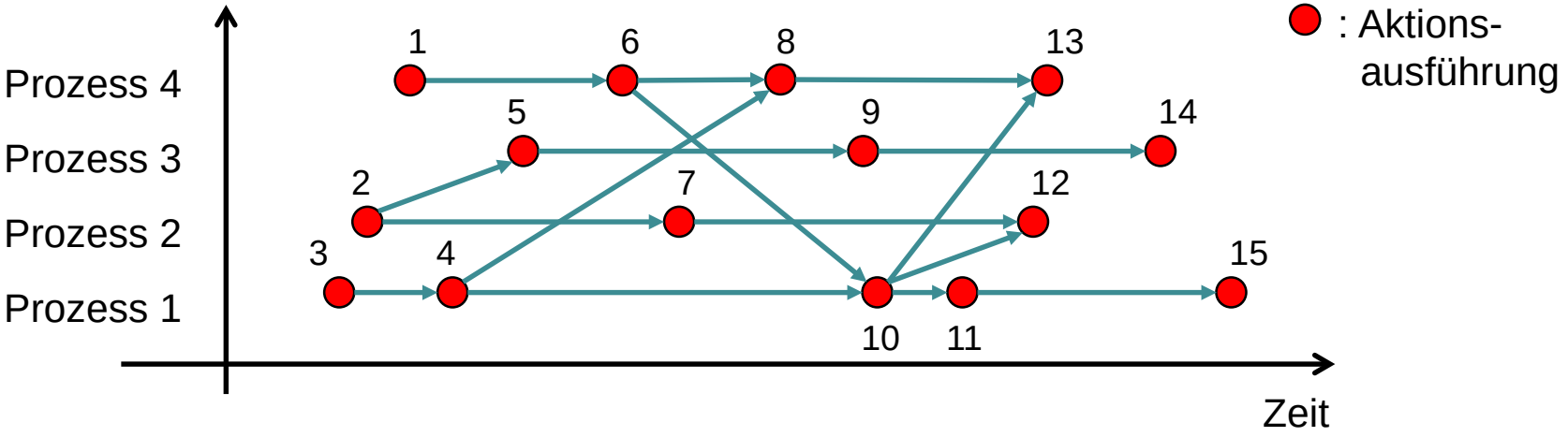
Beweis:

- Eine Aktion hängt nur von der sie aufrufenden Anfrage oder dem lokalen Zustand des Prozesses ab und kann nur auf die lokalen Variablen des Prozesses zugreifen.
- Aktionsausführung A_i hängt von A_j ab, wenn A_i aufgrund einer Anfrage von A_j ausgeführt wird oder wenn A_i direkt nach A_j im selben Prozess ausgeführt wird.
- Betrachte den Graph $G=(V,E)$, bei dem jeder Knoten eine Aktionsausführung darstellt und eine Kante (v,w) existiert genau dann wenn w von v abhängt.
- Für jede Kante $(v,w) \in E$ gilt, dass w erst nach dem Start von v ausgeführt worden ist. G ist also azyklisch (d.h. G hat keinen gerichteten Kreis).
- Daher können die Knoten von G in eine topologische Ordnung gebracht werden. Diese Ordnung stellt eine global atomare Aktionsausführung dar, die zur lokalen äquivalent ist. (Beweis evtl. Übung)

Selbststabilisierung

Satz 3.3 anschaulich:

- Lokal atomare Ausführung:



- Zahlen: mögliche topologische Sortierung
  (= Reihenfolge für global atomare Ausführung)

Selbststabilisierung

Wann führt ein Prozess eine Aktion aus?

→ Wir nehmen Fairness an, d.h. keine unbegrenzt ausführbare Aktion muss beliebig lange auf ihre Ausführung warten.

Aktion des Typs ⟨Name⟩(⟨Objektliste⟩) → ⟨Befehle⟩:

- Lokaler Aufruf von anderer Aktion A: wird sofort ausgeführt (gilt wie Unteraufruf, zählt dann zur Ausführung von A)
- Eingehende Anfrage: entsprechender Aktionsaufruf ist unbegrenzt ausführbar, da Anfrage nicht verfällt, d.h. die Aktion wird gemäß der Fairness in endlicher Zeit ausgeführt

Aktion des Typs ⟨Name⟩: ⟨Prädikat⟩ → ⟨Befehle⟩:

- Nur dann Ausführung in endlicher Zeit garantiert, wenn diese sonst unbegrenzt ausführbar wäre wie timeout.

Selbststabilisierung

Allgemeine Formulierung eines Netzwerkproblems P:
Gegeben: initialer Systemzustand S
Gesucht: legaler Systemzustand S´(S)

Annahmen:

- Global atomare Ausführung von Aktionen
- Fairness: jede unbegrenzt ausführbare Aktion wird irgendwann ausgeführt

Sei L(S) die Menge aller legalen Zustände für Anfangszustand S.

Definition 3.4: Ein System heißt selbststabilisierend bzgl. P, wenn die folgenden Anforderungen erfüllt sind für den Fall, dass keine Fehler auftreten und die Menge der Knoten statisch ist:

(1) Konvergenz: Für alle initialen Systemzustände S und alle fairen Rechnungen (d.h. Folgen fairer Aktionsausführungen) erreicht das System in endlicher Zeit einen Zustand S´ mit $S' \in L(S)$.

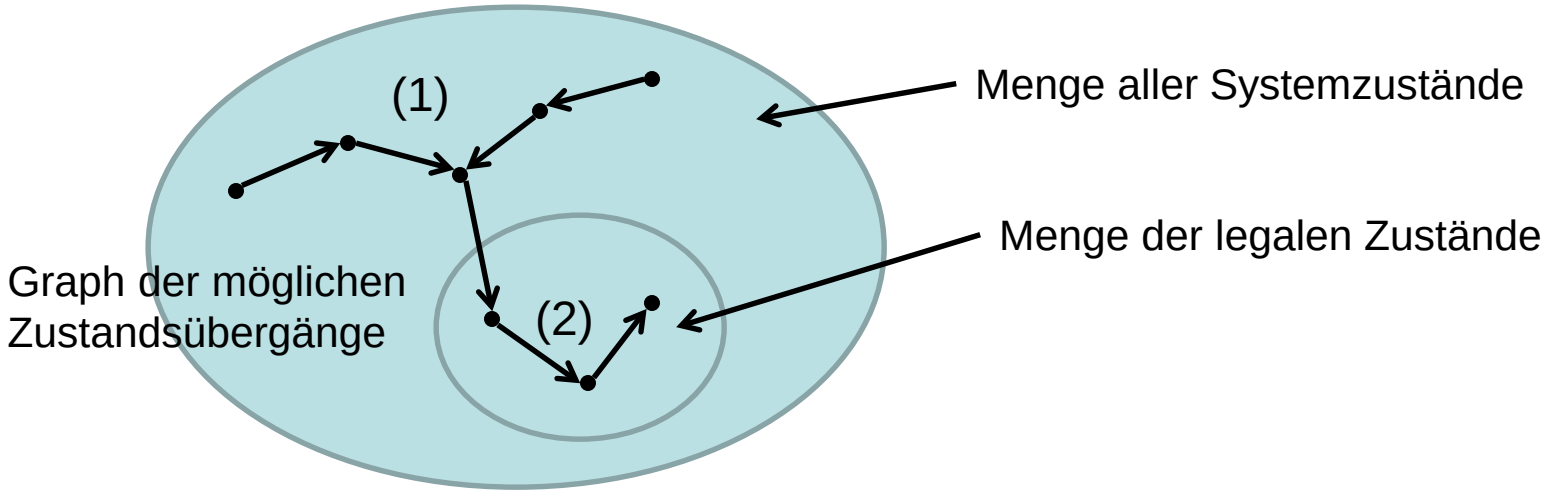
(2) Abgeschlossenheit: Für alle initialen Systemzustände S, die legal sind, ist auch jeder Folgezustand legal.

Selbststabilisierung

Definition 3.4: Ein System heißt selbststabilisierend bzgl. P, wenn die folgenden Anforderungen erfüllt sind für den Fall, dass keine Fehler auftreten und die Menge der Knoten statisch ist:

(1) Konvergenz: Für alle initialen Systemzustände S und alle fairen Rechnungen erreicht das System in endlicher Zeit einen Zustand S´ mit $S' \in L(S)$.

(2) Abgeschlossenheit: Für alle initialen Systemzustände S, die legal sind, ist auch jeder Folgezustand legal.



Selbststabilisierung

Definition 3.4: Ein System heißt selbststabilisierend bzgl. P, wenn die folgenden Anforderungen erfüllt sind für den Fall, dass keine Fehler auftreten und die Menge der Knoten statisch ist:

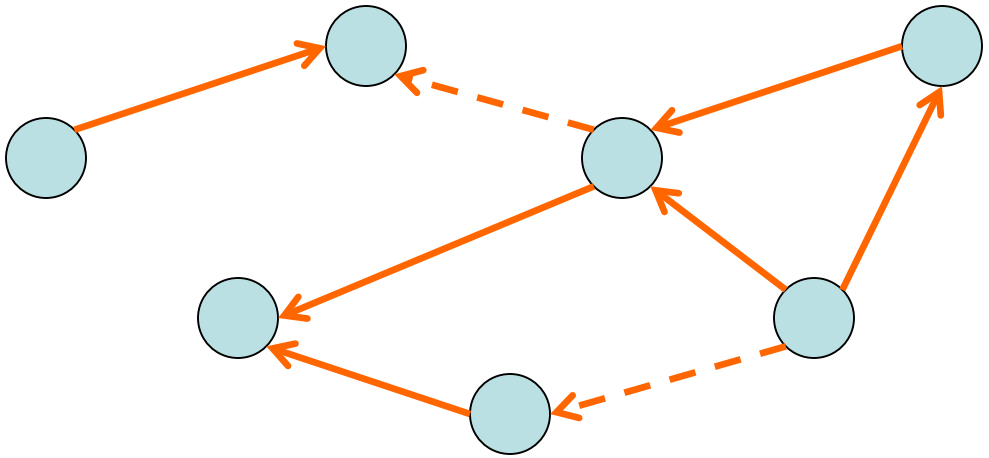
(1) Konvergenz: Für alle initialen Systemzustände S und alle fairen Rechnungen erreicht das System in endlicher Zeit einen Zustand S´ mit $S' \in L(S)$.

(2) Abgeschlossenheit: Für alle initialen Systemzustände S, die legal sind, ist auch jeder Folgezustand legal.

Bemerkung: Die Forderung in der Konvergenz ist wörtlich zu nehmen: ALLE initialen Systemzustände sind erlaubt, d.h. es muss kein wohlinitialisierter Systemzustand vorliegen. Die Prozesszustände und die Nachrichten mögen also anfangs beliebig fehlerhaft sein. Das macht die Entwicklung selbststabilisierender Systeme oft sehr kompliziert.

Selbststabilisierung

Weiteres Ziel: System erholt sich „monoton“ vom illegalen Zustand



D.h. wichtige Eigenschaften für das System, die zur Zeit t gelten, sollen auch zu jedem Zeitpunkt t´>t gelten unter der Annahme, dass keine Fehler auftreten und die Knotenmenge statisch ist.

Selbststabilisierung

Forderung für die Erreichbarkeit:

- Monotone Erreichbarkeit: Ist v von u aus zur Zeit t (über explizite oder implizite Kanten) erreichbar, dann auch zu jeder Zeit t´>t unter der Annahme, dass keine Fehler auftreten und die Knotenmenge statisch ist.

Satz 3.5: Die Vorstellungs-, Weiterleitungs- und Verschmelzungs-primitive garantieren monotone Erreichbarkeit.

Beweis: über Induktion (Übung).

Bemerkung:

- Satz 3.5 gilt nur, wenn keine Referenzen nicht existierender Prozesse im System sind. Sonst könnte bei einer Weiterleitung die Erreichbarkeit gefährdet sein.

Reicht das für eine Datenstruktur?

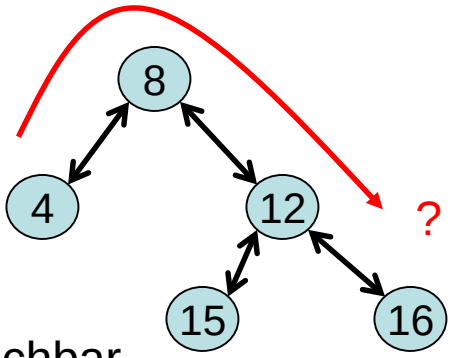
Selbststabilisierende Datenstrukturen

Reicht Erreichbarkeit für eine Datenstruktur?

Nein, da die Operationen der Datenstruktur nur dann funktionieren, wenn die Datenstruktur die erforderliche Form (z.B. binärer Suchbaum) hat.

→ Wir benötigen mehr: z.B. die Suchbarkeit, d.h. ist v von u aus zur Zeit t für die gegebene Suchoperation erreichbar, dann auch zu jeder Zeit t´>t unter der Annahme, dass keine Fehler auftreten und die Knotenmenge statisch ist.

Beispiel: Die ideale Topologie von DS sei ein Suchbaum und die Operationen verwenden die Suchbaumstrategie, um zu einem bestimmten Prozess zu gelangen. Dann ist Erreichbarkeit sicherlich nicht äquivalent zu Suchbarkeit.



15 ist von 4 erreichbar aber nicht suchbar

Selbststabilisierende Datenstrukturen

Im Allgemeinen:

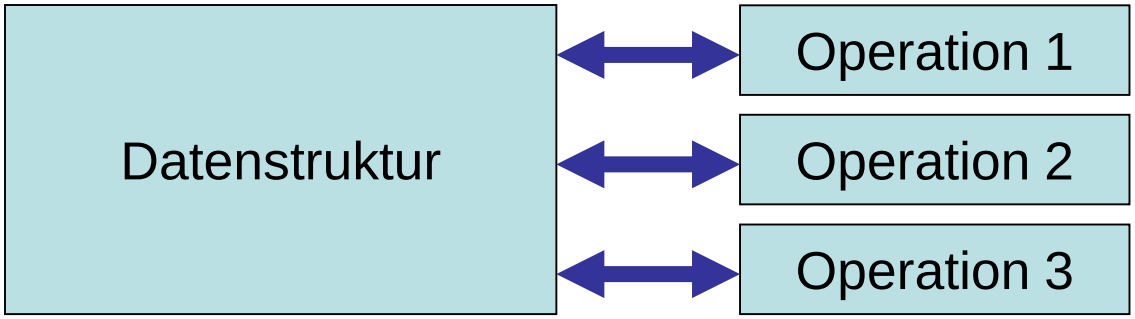
- Monotone Korrektheit bezüglich Eigenschaft E: Eigenschaft E (bestehend aus einer Menge an Forderungen) wird monoton erreicht
- E hängt von der gewünschten Funktionalität der Datenstruktur ab und sollte in jedem legalen Zustand vollständig erfüllt sein.

Beispiel: Suchstruktur

- Suchbarkeit: Für alle Prozesse v und w gilt, dass w von v aus suchbar ist.
- Monotone Suchbarkeit: Für alle v und w gilt, dass wenn w von v aus zur Zeit t suchbar ist, dann auch zu jeder Zeit t´>t.

Selbststabilisierende Datenstrukturen

Erinnerung: grundlegende Funktionsweise einer Datenstruktur



Standardoperation im sequentiellen Fall:

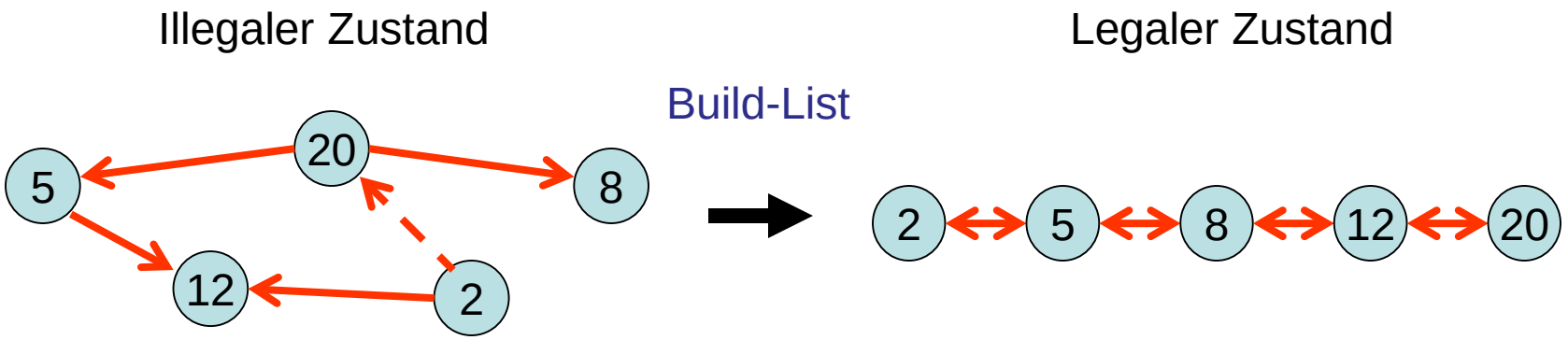
Build-DS(S): gegeben eine Elementmenge S, baue eine Datenstruktur DS für S auf

Verteilter Fall:

Build-DS: verteilt ausgeführtes Protokoll, das Datenstruktur DS aus beliebigem Systemzustand mit schwachen Zusammenhang heraus stabilisiert und idealerweise die monotone Korrektheit sicherstellt.

Selbststabilisierende Datenstrukturen

Beispiel: sortierte Liste



Definition 3.6: Build-DS stabilisiert die Datenstruktur DS, falls Build-DS

1. DS für einen beliebigen Anfangszustand mit schwachem Zusammenhang und eine beliebige faire Rechnung in endlicher Zeit in einen legalen Zustand überführt (Konvergenz) und
2. DS für einen beliebigen legalen Anfangszustand in einem legalen Zustand belässt (Abgeschlossenheit),

sofern keine Operationen auf DS ausgeführt werden und keine Fehler auftreten.

Selbststabilisierende Datenstrukturen

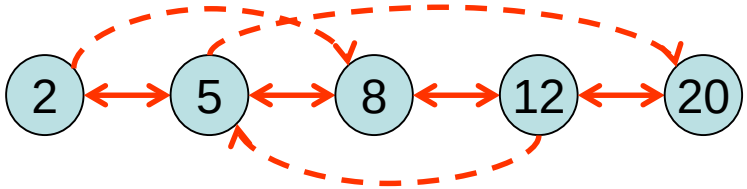
Gängige Anforderungen an einen legalen Zustand:

- Explizite Kanten formen gewünschte Topologie
- Keine korrumpierten Informationen mehr im System

} dann oft korrekt und stabil

Darüberhinaus kann es weitere DS-spezifische Anforderungen geben.

Beispiel: für eine sortierte Liste wäre folgender Systemzustand bereits legal



Wir wollen idealerweise auch eine monotone Erholung.

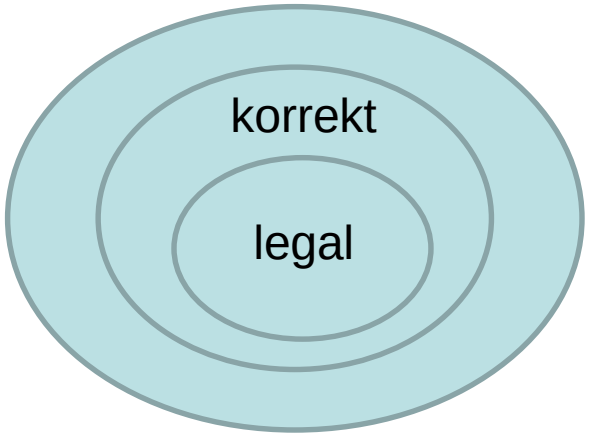
Definition 3.7: Build-DS stabilisiert die Datenstruktur DS monoton, falls Build-DS DS stabilisiert (siehe Def. 3.6) und zusätzlich die monotone Korrektheit sicherstellt.

Wir werden für konkrete Beispiele sehen, ob und wie das möglich ist.

Selbststabilisierende Datenstrukturen

Bemerkung:

- Im Allgemeinen ist nicht unbedingt jeder Zustand, in dem die Korrektheits-bedingung der Datenstruktur erfüllt ist, auch schon ein legaler Zustand.
- Problem: Es können noch Nachrichten mit korrumpierten Informationen im System unterwegs sein! Diese könnten nachträglich die Korrektheit gefährden.
- Sollte das System von einem Zustand s durch korrumpierte Informationen in einen Zustand s´ gelangen können, in dem die Datenstruktur nicht mehr korrekt ist, wäre s´ auch nicht mehr legal. D.h. s dürfte in diesem Fall auch nicht legal sein, da sonst die Abgeschlossenheits-forderung für ein selbststabilisierendes System verletzt wäre.
- Im Allgemeinen ist also die Menge der legalen Zustände nur eine Teilmenge der Zustände, in denen die Datenstruktur korrekt ist.

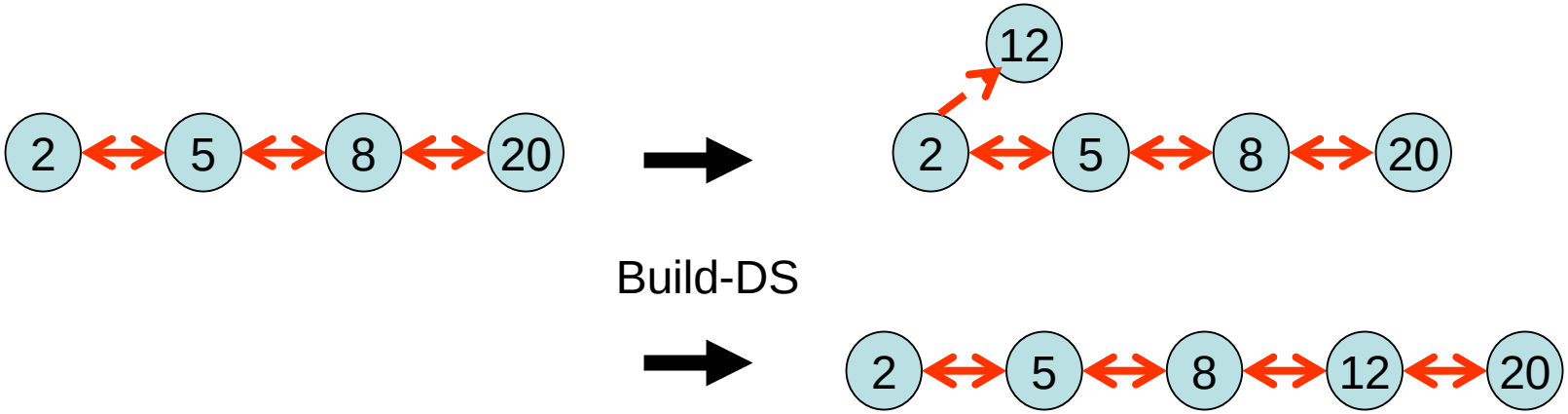


Menge der DS-Zustände

Selbststabilisierende Datenstrukturen

Idee: Stabilisiert Build-DS eine Datenstruktur, dann könnte Build-DS auch dazu benutzt werden, die Topologie bei einer veränderten Knotenmenge zu aktualisieren.

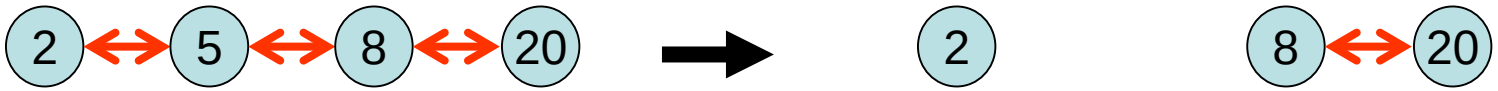
Beispiel: 2 lernt den neuen Knoten 12 kennen



Selbststabilisierende Datenstrukturen

Idee: Stabilisiert Build-DS eine Datenstruktur, dann könnte Build-DS auch dazu benutzt werden, die Topologie bei einer veränderten Knotenmenge zu aktualisieren.

Beispiel: 5 verlässt das System.

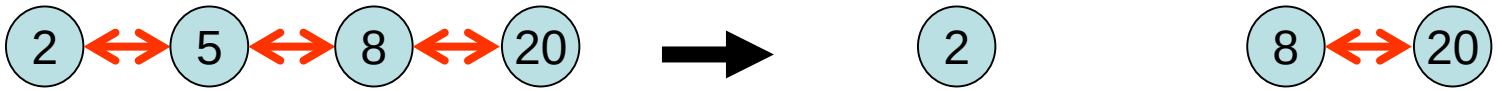


In diesem Fall kann Build-DS die Liste nicht mehr stabilisieren! Zur Stabilisierung des Falls, dass ein Knoten im System ohne Ankündigung verlässt, bräuchten wir also eine Topologie mit genügend hoher Expansion.

Selbststabilisierende Datenstrukturen

Idee: Stabilisiert Build-DS eine Datenstruktur, dann könnte Build-DS auch dazu benutzt werden, die Topologie bei einer veränderten Knotenmenge zu aktualisieren.

Beispiel: 5 verlässt das System.



Wir werden in Kapitel 4 sehen, wie Knoten aus einem System entfernt werden können, ohne dass der Zusammenhang verloren geht, sofern ein geeignetes Protokoll ausgeführt wird und es keine Fehler gibt.

Selbststabilisierende Datenstrukturen

Wie wollen wir die Qualität einer verteilten Daten-struktur DS messen?

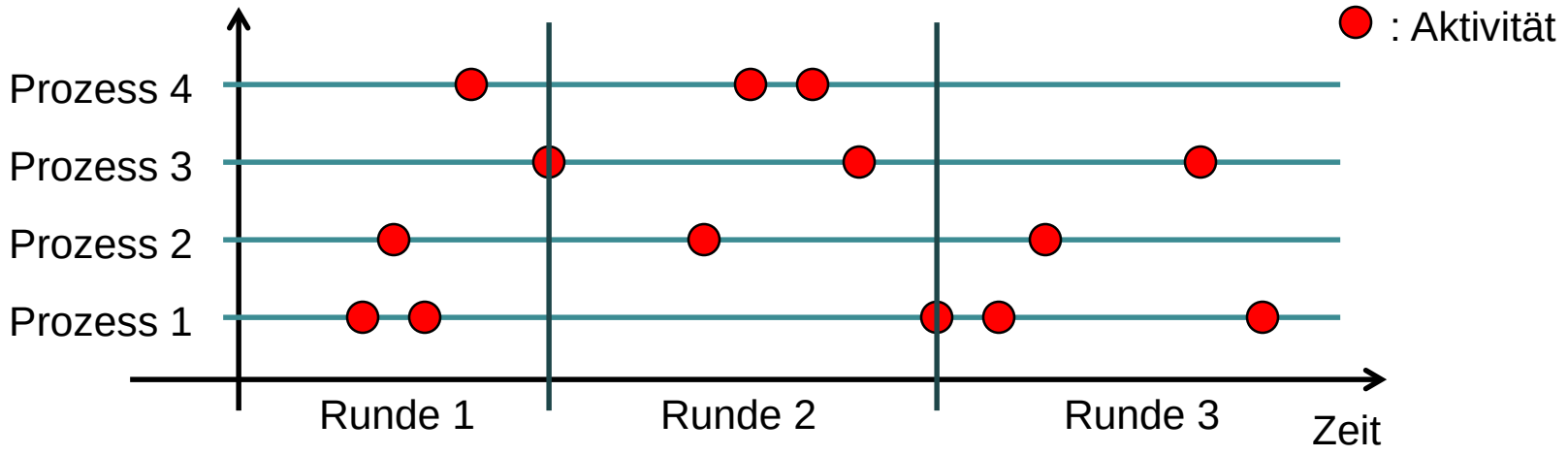
Build-DS Protokoll:

- Robustheitskriterien:
  - Selbststabilisierung von beliebigem schwachen Zusammenhang aus
  - Monotone Korrektheit
- Effizienzkriterien:
  - Geringe worst-case Arbeit (Anzahl Botschaften bzw. strukturelle Änderungen) für Selbststabilisierung
  - Geringe worst-case Arbeit zur Stabilisierung für eine einzelne Aktion (Join / Leave)

Selbststabilisierende Datenstrukturen

Zeitmodell:

- Wir erlauben beliebig asynchrone Bearbeitung der Anfragen durch die Prozesse. (D.h. wir fordern nicht, dass alle Prozesse gleichzeitig und in derselben Geschwindigkeit Anfragen bearbeiten.)
- Eine Runde ist dann vorbei, wenn jeder Prozess, der eine Anfrage zu bearbeiten hat, mindestens eine dieser Anfragen bearbeitet hat.
- Wir messen die Laufzeit dann in der Anzahl der Runden.



Designprinzipien für verteilte Systeme

- Prozessmodell und Pseudocode
- Netzwerkmodell und zulässige Verbindungsprimitive
- Selbststabilisierung
- Konsistenzmodelle

Konsistenzmodelle

Definition 3.8:

- Ein Ereignis ist eine Teilmenge des gegebenen Zustandsraums.
- Eine Sicherheitseigenschaft (safety property) ist eine Eigenschaft, die angibt, dass Ereignis E niemals eintritt.
- Eine Lebendigkeitseigenschaft (liveness property) ist eine Eigenschaft, die angibt, dass irgendwann Ereignis E eintritt.

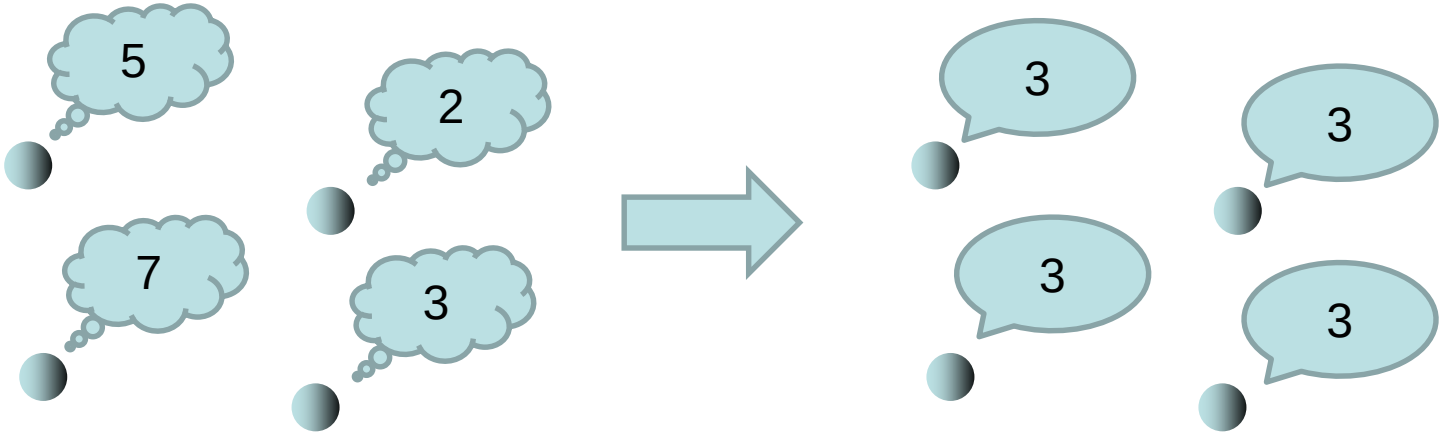
Beispiele in unserem Kontext:

- Ereignis: Menge der legalen Zustände
- Sicherheitseigenschaft: Abgeschlossenheit, monotone Erreichbarkeit
- Lebendigkeitseigenschaft: Konvergenz

Konsistenzmodelle

Die Beziehungen zwischen Sicherheits- und Lebendigkeits-eigenschaften sind bereits intensiv erforscht worden.

Beispiel 1: Fischer, Lynch, Paterson (1985): fehlertoleranter Konsensus ist unmöglich in einem asynchronen System



Konsistenzmodelle

Die Beziehungen zwischen Sicherheits- und Lebendigkeits-eigenschaften sind bereits intensiv erforscht worden.

Beispiel 1: Fischer, Lynch, Paterson (1985): fehlertoleranter Konsensus ist unmöglich in einem asynchronen System

Forderungen:

- Konsensus: jeder Prozess gibt denselben Wert aus (safety)
- Gültigkeit: jeder ausgegebene Wert muss einer der eingegebenen Werte sein (safety)
- Terminierung: irgendwann gibt jeder Prozess einen Wert aus (liveness)

Für die Aussage oben reicht es, dass ein einziger Prozess zu einem ungünstigen Zeitpunkt ausfällt.

Konsistenzmodelle

Beispiel 2: Brewers CAP Theorem (2000): In jedem verteilten System können nur zwei der drei Eigenschaften Konsistenz, Verfügbarkeit und Partitionstoleranz sichergestellt werden.

- Konsistenz: jede Antwort einer Anfrage ist korrekt (safety)
- Verfügbarkeit: jede Anfrage erhält in endlicher Zeit eine Antwort (liveness)
- Partitionstoleranz: die gegebenen Anforderungen sind auch dann erfüllt, wenn die Prozesse in mindestens zwei Gruppen unterteilt sind, die nicht miteinander interagieren können

Im Allgemeinen gilt: Sicherheit und Lebendigkeit können üblicherweise nicht gleichzeitig in unzuverlässigen verteilten Systemen garantiert werden.

$\rightarrow$ Schwerpunkt auf der Eigenschaft setzen, die wichtiger ist.

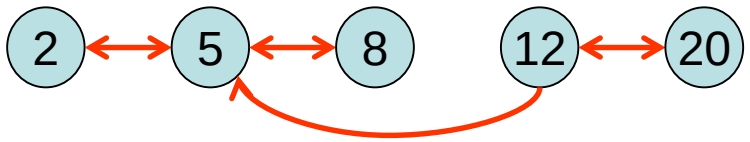
Konsistenzmodelle

Bei Verwendung unserer Verbindungsprimitive:

Kein Partitionsproblem (sofern keine Referenzen nicht existierender Prozesse im System und keine Ausfälle), also Konsistenz und Verfügbarkeit prinzipiell möglich

Problem: es ist im Allgemeinen nicht lokal entscheidbar, ob ein System bereits in einem legalen Zustand ist (und damit die Korrektheit einer Antwort garantiert ist).

Beispiel:



Aus der lokalen Sicht aller Knoten sieht sortierte Liste legal aus.

Konsistenzmodelle

Bei Verwendung unserer Verbindungsprimitive:
Kein Partitionsproblem (sofern keine Referenzen nicht existierender Prozesse im System und keine Ausfälle), also Konsistenz und Verfügbarkeit prinzipiell möglich

Problem: es ist im Allgemeinen nicht lokal entscheidbar, ob ein System bereits in einem legalen Zustand ist (und damit die Korrektheit einer Antwort garantiert ist).

Konsequenz:

- Verfügbarkeit: sollte grundsätzlich in asynchronen, selbststabilisierenden Systemen erfüllt werden, da es ohnehin im Allgemeinen nicht lokal entscheidbar ist, ob das System bereits in einem legalen Zustand ist.
- Konsistenz: im Allgemeinen bestenfalls garantierbar, dass Konsistenz monoton wiederhergestellt werden kann.

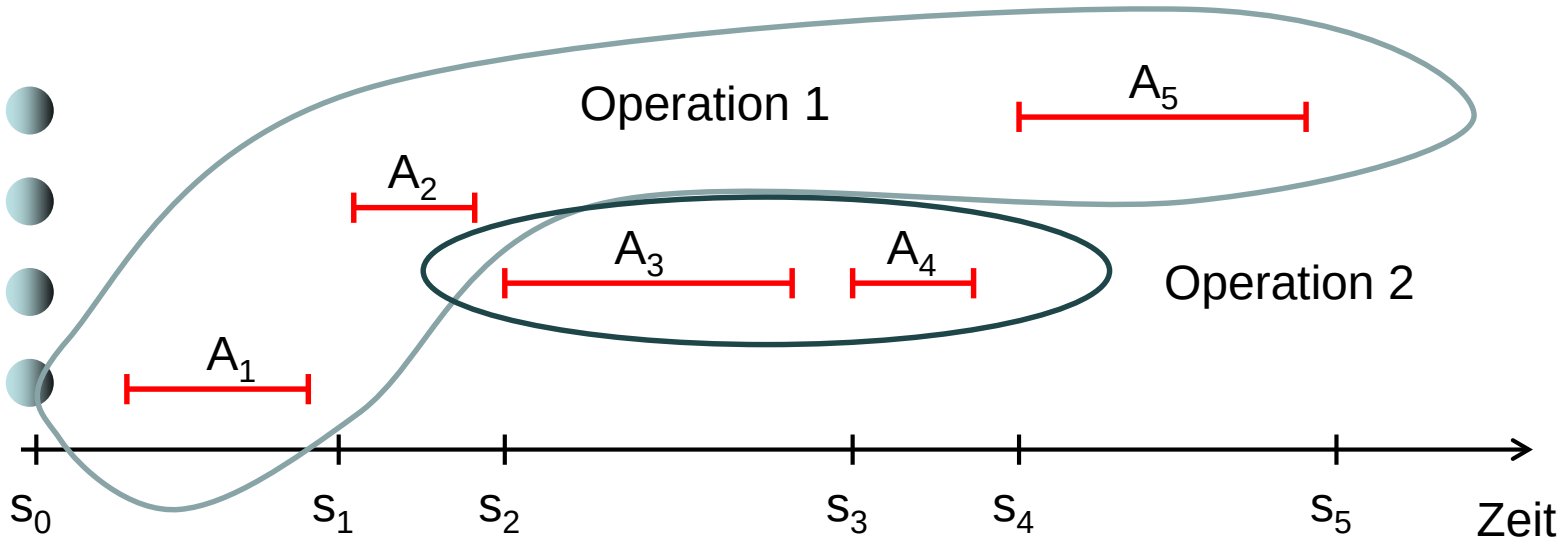
Konsistenzmodelle

Sobald Konsistenz wieder hergestellt ist,
kann diese zumindest dann aufrecht-
erhalten werden (sofern keine Fehler auftreten)?

Problem: Nebenläufigkeit (d.h. Ausführungen von Operationen überlappen sich)

Konsistenzmodelle

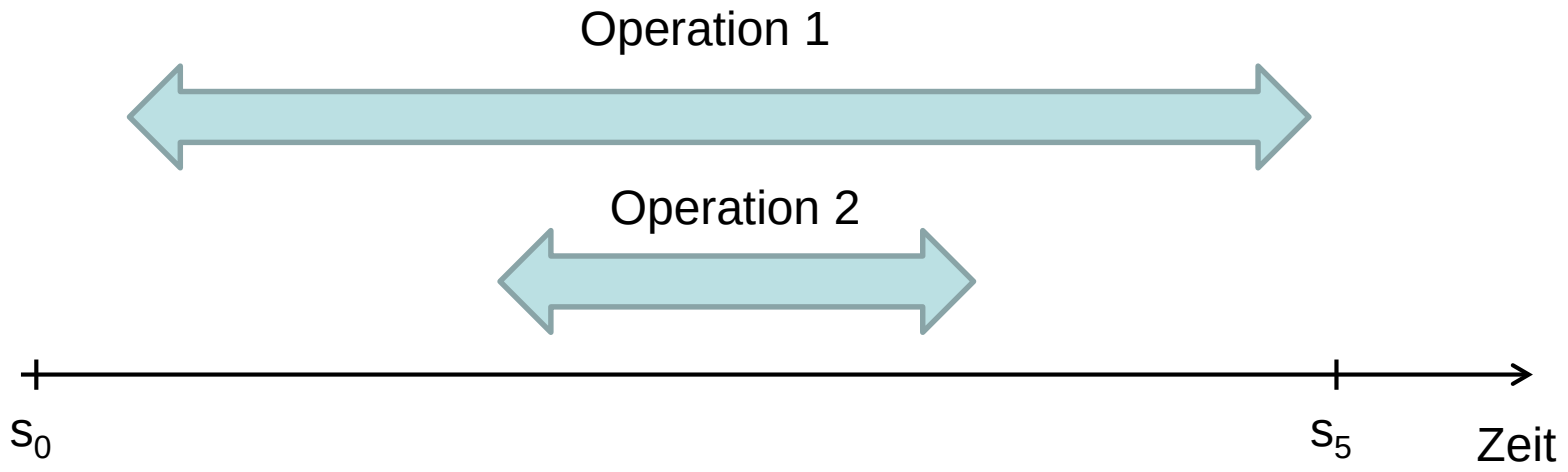
Selbst bei global atomarer Ausführung von Aktionen können nebenläufig ausgeführte Operationen verschiedenste Konsistenzprobleme verursachen.



Konsistenzmodelle

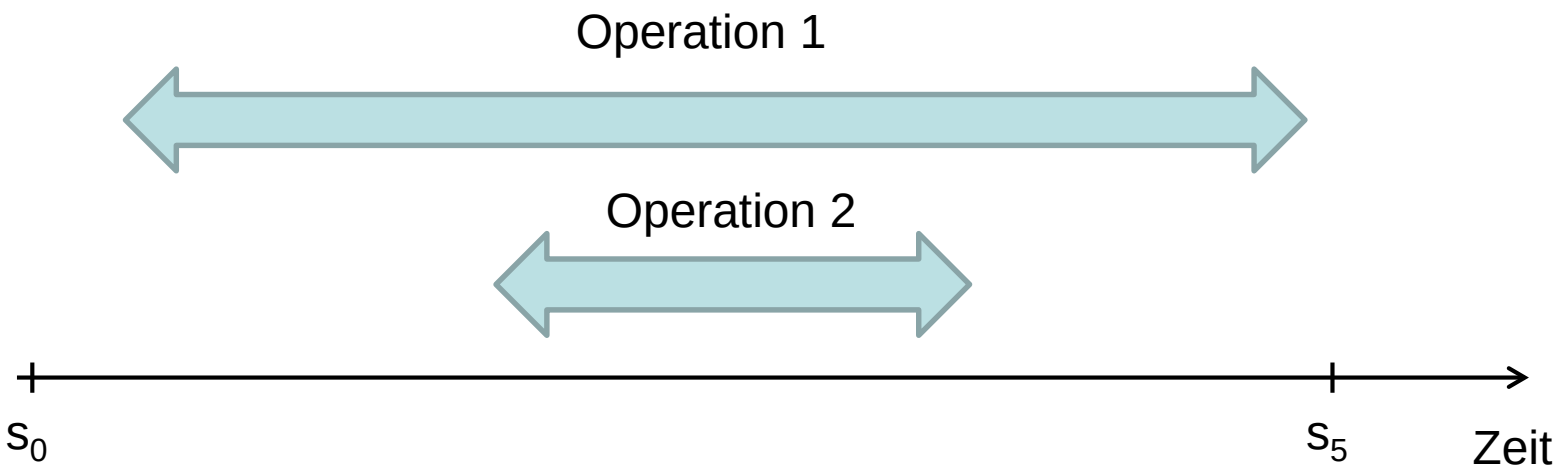
Selbst bei global atomarer Ausführung von Aktionen können nebenläufig ausgeführte Operationen verschiedenste Konsistenzprobleme verursachen.

Abstrakt:



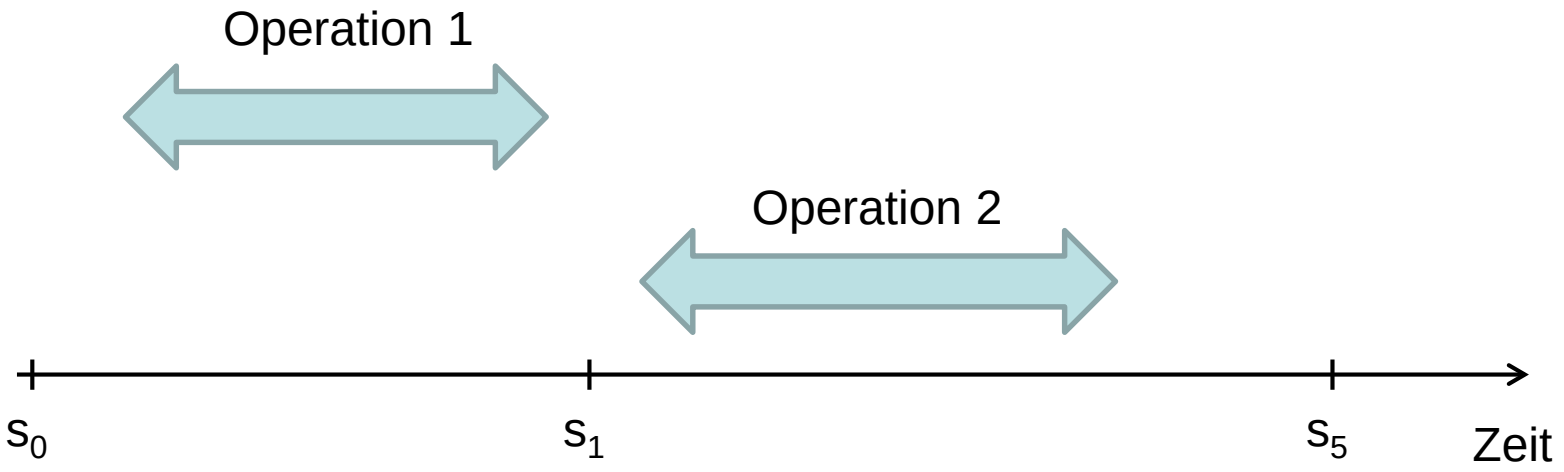
Konsistenzmodelle

Definition 3.9: Eine Rechnung R ist linearisierbar, wenn sie zu einer Rechnung S erweitert werden kann, so dass der Endzustand von S äquivalent zu einer global atomaren Ausführung von Operationen in S ist.



Konsistenzmodelle

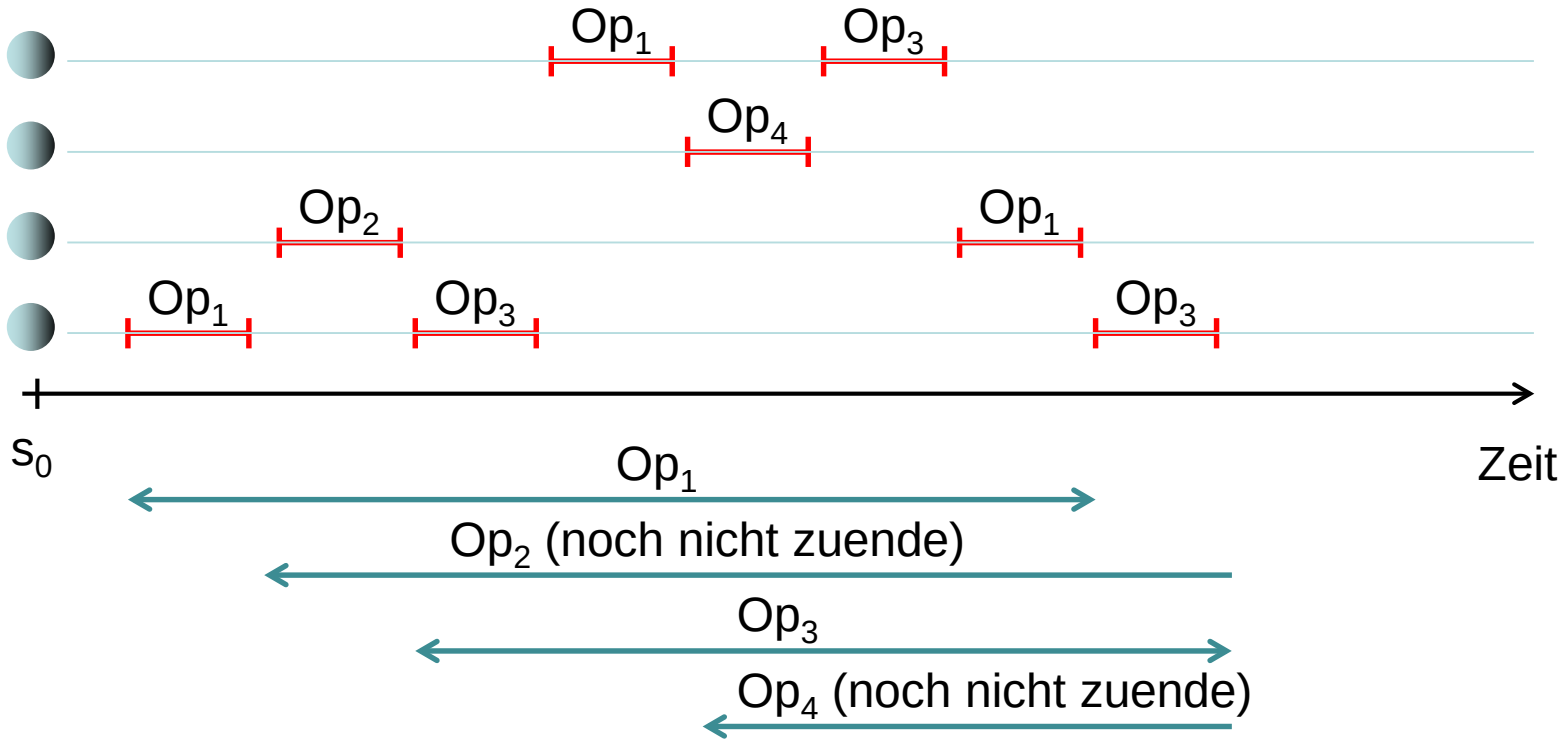
Definition 3.9: Eine Rechnung R ist linearisierbar, wenn sie zu einer Rechnung S erweitert werden kann, so dass der Endzustand von S äquivalent zu einer global atomaren Ausführung von Operationen in S ist.



Konsistenzmodelle

Beispiel für Erweiterung

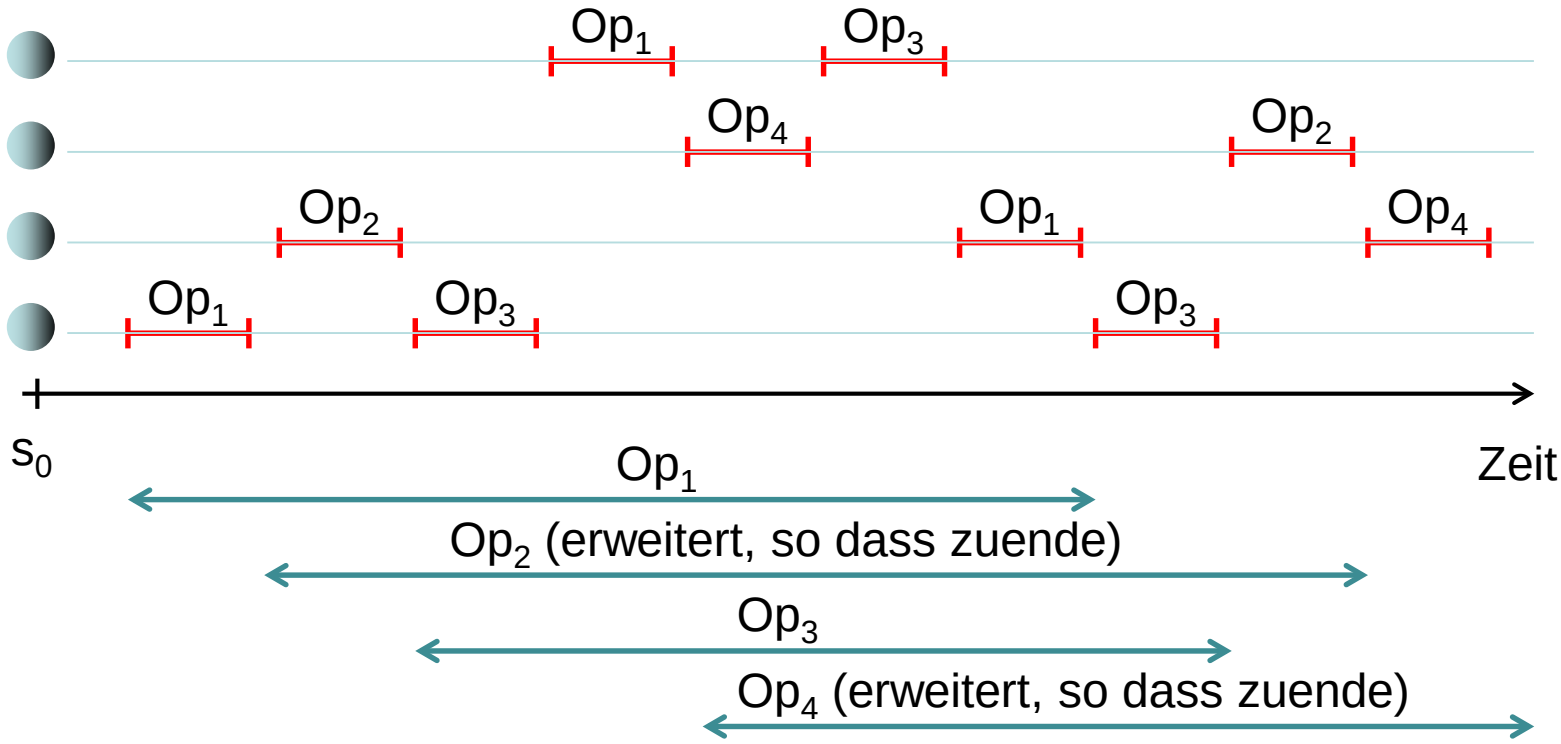
Rechnung R:



Konsistenzmodelle

Beispiel für Erweiterung

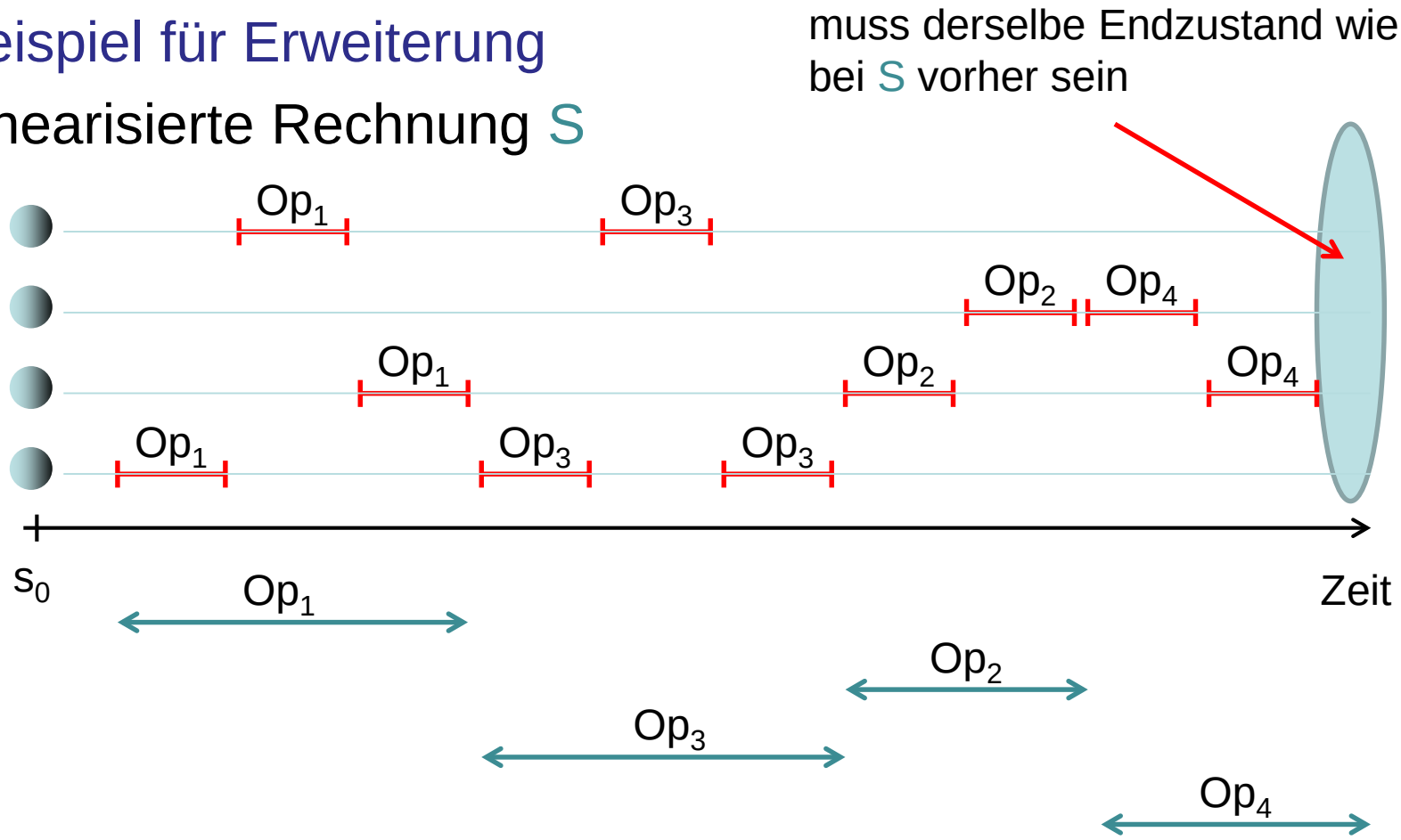
Rechnung S:



Konsistenzmodelle

Beispiel für Erweiterung

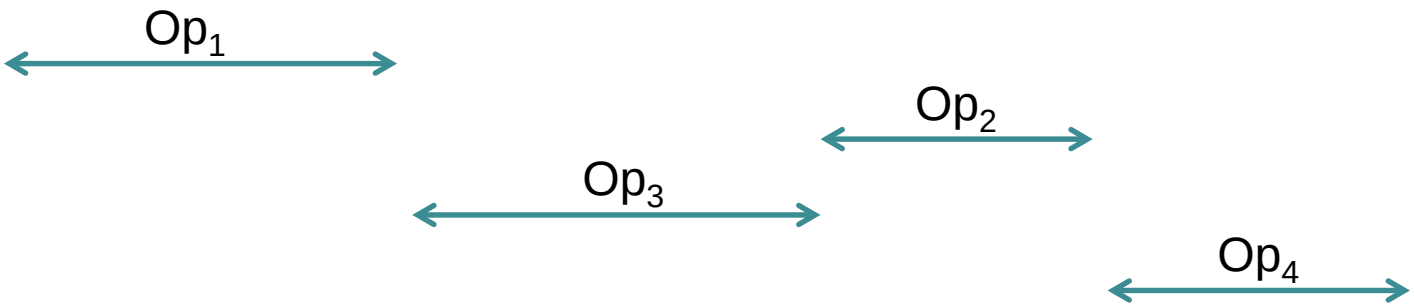
Linearisierte Rechnung S



Konsistenzmodelle

Definition 3.10: Eine Linearisierung L(S) einer Rechnung S ist lokal konsistent, wenn für jeden Prozess v gilt, dass die Operationen, die von v initiiert werden, in derselben Reihenfolge in L(S) auftauchen, wie sie von v initiiert worden sind.

Beispiel: für die Initiierungsfolge (Op_1, Op_3) von Prozess v ist folgende Linearisierung lokal konsistent



Konsistenzmodelle

Ideales Ziel für diese Vorlesung:

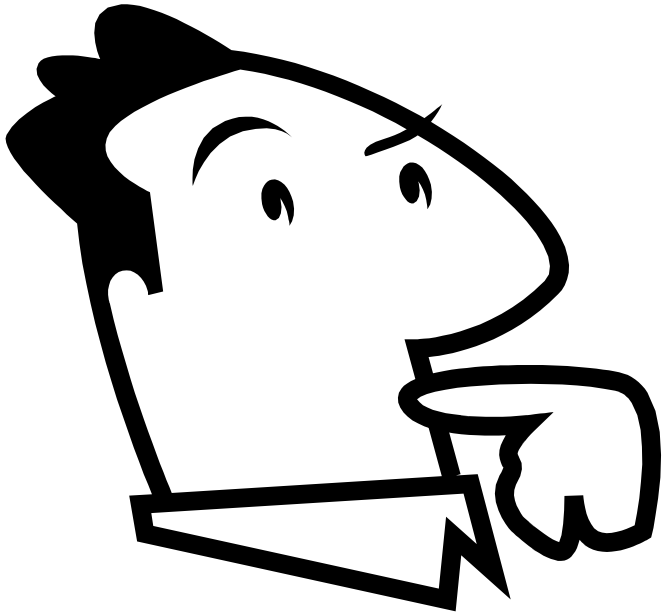
monoton selbststabilisierende Datenstrukturen, die linearisierbar und lokal konsistent sind

Andere Prinzipien in der Literatur und in Systemen:

- ACID-Prinzip (Atomicity, Consistency, Isolation, and Durability): Datenbanken
- BASE-Prinzip (Basically Available, Soft state, Eventual consistency): Web Services
- … und viele andere Prinzipien dazwischen

Referenzen

- Michael J. Fischer, Nancy A. Lynch, Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2): 374-382 (1985)
- S. Gilbert and N. Lynch. Perspectives on the CAP Theorem. *IEEE Computer* 25(2): 30-36, 2012.
- http://en.wikipedia.org/wiki/Consistency_model
- http://en.wikipedia.org/wiki/Linearizability
- http://en.wikipedia.org/wiki/Self-stabilization



Fragen?