# Algorithmen für Ad-hoc- und Sensornetze

## VL 12 – Synchronisation und Zeitsynchronisation

Dr. rer. nat. Bastian Katz

Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Karlsruher Institut für Technologie

15. Juli 2009
(Version 3 vom 21. Juli 2009)



## Heute

- Synchronisation: Was sind synchrone Algorithmen in asynchronen Netzen wert?
  - Was sind asynchrone Netze und was brauche ich für synchrone Algorithmen?
  - ein Synchronisierer in drei Varianten
- Zeitsynchronisation: Wie erreicht man, dass alle Knoten dasselbe Zeitgefühl haben?
  - Warum schon einfache Lösungen ihre Tücken haben
  - Gradientensynchronisation: Was man alles falsch machen kann



## Erinnerung: Synchrone Kommunikation

*(ganz informell, siehe erste Vorlesung für Details)*

- Zeit vergeht in *Runden*, in denen
- in jeder Runde kann jeder Knoten
  1. beliebige Berechnungen ausführen
     - (und dabei Nachrichten der vorangeg. Runde verwenden)
  2. Nachrichten an alle Nachbarn schicken
  3. Nachrichten empfangen („Posteingang leeren“)

\+ sehr klares Modell, gut verstanden

– so direkt hat das wenig von Sensornetzen
  - Schon die Zeit für eine einzelne Übertragung schwer vorherzusehen
  - Wer gibt den Startschuss für eine Runde?



## Asynchrones Knotenmodell

### Definition: Knotenkonfiguration

Jeder Knoten lässt sich zu jedem Zeitpunkt beschreiben durch

- Menge $Q$ von *Zuständen*, in denen er sich befinden kann
- einen Menge `out` von *ausgehenden Nachrichten*
- einen Menge `in` von *angenommenen Nachrichten*

Eine *Konfiguration* $c$ eines Knotens umfasst seinen Zustand und beide Mengen von Nachrichten.
Eine Konfiguration $C$ eines verteilten Systems besteht aus den Konfigurationen aller Knoten $C = (c_1, c_2, \ldots, c_n)$.

Viel realistischer: Senden/Empfangen ist aus Sicht des Programmes typischerweise von der Programmausführung unabhängig. Knoten rechnen, übergeben ausgehende Pakete an den Netzwerkstack und fragen, ob Pakete empfangen wurden.

## Asynchrone Kommunikation

**Definition: Asynchrones Modell**

Eine *Ausführung* eines Programms im *Asynchronen Modell* ist eine Folge von Konfigurationen und Schritten $C_0, \phi_0, C_1, \phi_1, \ldots$, in der

- $\phi_i$ darin besteht, dass entweder
  - eine beliebige Nachricht zugestellt wird
  - ein Knoten einen Berechnungsschritt macht, und dabei
    - alle eingehenden Nachrichten entgegennimmt
    - beliebige Berechnungen durchführt
    - ggf. ausgehende Nachrichten erzeugt
- $C_{i+1}$ aus $C_i$ durch Schritt $\phi_i$ hervorgeht
- jede Nachricht irgendwann zugestellt wird

- Aber: Keine Annahme an die Reihenfolge der Schritte!
  - noch nicht mal Reihenfolge der Nachrichten!





## Komplexitäten

Wie kann man von Zeitkomplexität sprechen, wenn Nachrichten beliebig „verschleppt“ werden können?

**Definition Zeitkomplexität**

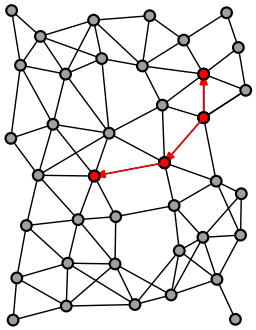
Die Zeitkomplexität eines Algorithmus im asynchronen Modell ist die maximale Zeit, die bis zur Terminierung vergehen kann, wenn man die Zeit für das Zustellen einer Nachricht auf 1 begrenzt.

- Zustellen einer Nachricht: Zeit zwischen dem Berechnungsschritt, der die Nachricht erzeugt und dem Berechnungsschritt, der die Nachricht entgegennimmt!

- Vorsicht: So ein Modell „verbirgt“ höhere Wartezeiten für Übertragungen bei stark ausgelastetem Kanal!
- Nachrichtenkomplexität ist in dem Modell dasselbe



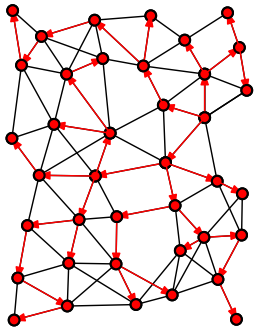
## Beispiel: Fluten



- Knoten sendet Nachricht an alle Nachbarn, sobald er die erste Nachricht empfängt
  - wie gehabt..
- ohne weitere Kniffe: Reihenfolge beliebig
  - Baum ist nicht unbedingt Kürzeste-Wege-Baum zur Senke!



## Beispiel: Fluten



- Knoten sendet Nachricht an alle Nachbarn, sobald er die erste Nachricht empfängt
  - wie gehabt..
- ohne weitere Kniffe: Reihenfolge beliebig
  - Baum ist nicht unbedingt Kürzeste-Wege-Baum zur Senke!
- stört uns ja nicht
- Zeitkomplexität? $O(D)$!
  - Beweis: Induktion über Abstand zum Startknoten!

Nicht immer geht das so glimpflich! Was ist zum Beispiel mit Färbungsalgorithmen ohne klare Runden?

## Synchronisatoren

### Synchronisator

Ein *Synchronisator* ist ein Verfahren, einen beliebigen *synchronen* Algorithmus mit asynchroner Kommunikation auszuführen.
Dazu fügt ein Synchronisator dem Zustand eines Knotens einen Zähler $r$ hinzu mit folgenden Eigenschaften:

- ≫ der Zähler erhöht sich nur zu Beginn einer Berechnung
- ≫ springt der Zähler auf $i$, sind alle Nachrichten empfangen worden, die der Knoten im synchronen Algorithmus in Runde $i-1$ empfangen hätte

⇒ Bis der Zähler hochspringt, merken wir uns eingehende Nachrichten nur
⇒ Wenn der Zähler hochspringt, führen wir die synchrone Runde $i$ aus!



## Wir bauen uns einen Synchronisator

### Zutaten: Synchronizator $\alpha$

1. Jede Nachricht des synchronen Algorithmus wird bestätigt
   - ≫ hat ein Knoten eine Nachricht empfangen, schickt er eine Bestätigung
2. Hat ein Knoten für alle Nachrichten, die er in Runde $i$ erzeugt hat, eine Bestätigung erhalten, schickt er eine Nachricht `SAFE-`$i$ an alle Nachbarn
3. Hat ein Knoten `SAFE-`$i$ von allen Nachbarn empfangen, springt der Zähler auf $i+1$



## Komplexität Synchronisator $\alpha$

### Definition: Zeitkomplexität eines Synchronisators

Ein Synchronisator hat Zeitkomplexität $T$, wenn nach $iT$ Zeiteinheiten alle Knoten einen Zähler von mindestens $i$ haben.
Bei der Nachrichtenkomplexität zählen Bestätigungen nicht mit!

### Satz

Mit Synchronisator $\alpha$ hat Zeitkomplexität $O(1)$ und Nachrichtenkomplexität $O(m)$/Runde.

- ≫ Bew. Zeitkomplexität durch Induktion für $T = 3$:
  - ≫ gilt für $i = 0$
  - ≫ haben alle Knoten Zähler $i$, dauert es maximal 2 ZE, bis alle Knoten `SAFE-`$i$ gesendet haben...
- ≫ pro Runde verschickt jeder Knoten $u$ $\Theta(\deg u)$ Nachrichten.



## Andere Synchronisatoren?

Synchronisator $\alpha$ braucht keinerlei Vorbereitung. Was, wenn wir uns vorher Strukturen zur Synchronisation aufbauen können?

### Synchronisator $\beta$

Berechne einen beliebigen gerichteten Spannbaum $T$
Wenn der Zähler auf $i$ springt

- ≫ verschicke alle Nachrichten aus Runde $i$
- ≫ warte auf Bestätigungen
- ≫ warte, bis alle Kinder `SAFE-`$i$ gesendet haben
- ≫ sende `SAFE-`$i$ an Vorgänger
  - ≫ Wurzel sendet stattdessen `START-`$(i+1)$ an Kinder
- ≫ warte auf `START-`$(i+1)$-Signal und setze Zähler auf $i+1$
  - ≫ Wurzel wartet nicht

## Komplexität Synchronisator $\beta$

**Satz (ohne Beweis)**

Synchronisator $\beta$ hat Zeitkomplexität $O(D_T)^a$ und Nachrichtenkomplexität $O(n)$/Runde.

[a] $D_T$: Durchmesser/größte Entfernung im Baum

- ≫ Bessere Nachrichtenkomplexität, aber viel schlechtere Zeitkomplexität.
- ≫ Dafür: Irgendeinen Baum kann man auch asynchron verteilt bestimmen (ohne Beweis)!

Kann man das beste aus beiden Welten bekommen? TIPP:

- ≫ $\alpha$ wartet nur auf Nachbarn, das geht schnell
- ≫ $\beta$ synchronisiert im Baum, das kostet weniger Nachrichten

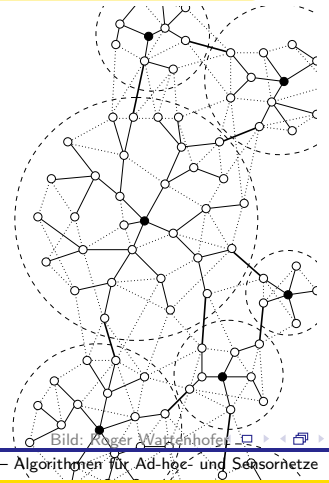




## Synchronisator $\gamma$

**Idee Synchronisator $\gamma$**

Partitioniere das Netz in kleine Gruppen!



- ≫ In jeder Gruppe wähle einen Baum und eine Wurzel aus
- ≫ zu je zwei benachbarten Gruppen wähle eine *Brückenkante* aus

Wie synchronisieren wir uns jetzt?

Bild: Roger Wattenhofer



## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal `NB-SAFE-`$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass `NB-SAFE-`$i$ über alle Brückenkanten *empfangen* wurde

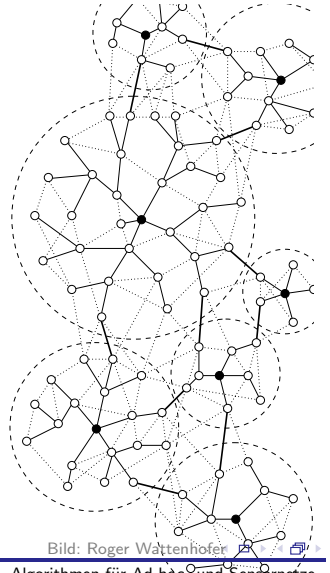


Bild: Roger Wattenhofer



## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal `NB-SAFE-`$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass `NB-SAFE-`$i$ über alle Brückenkanten *empfangen* wurde

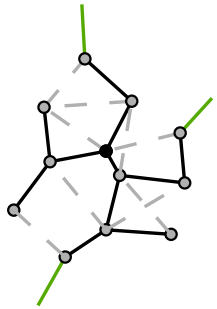


Bild: Roger Wattenhofer

## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal NB-SAFE-$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass NB-SAFE-$i$ über alle Brückenkanten *empfangen* wurde

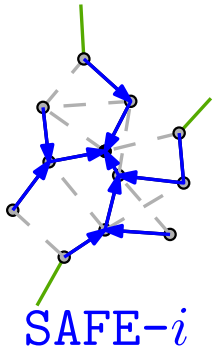



Bild: Roger Wattenhofer



Synchronisation Zeitsynchronisation

## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal NB-SAFE-$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass NB-SAFE-$i$ über alle Brückenkanten *empfangen* wurde

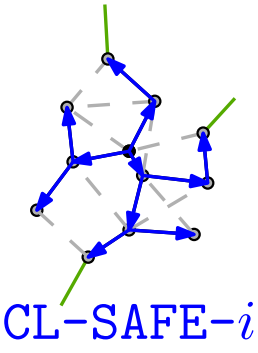



Bild: Roger Wattenhofer

Bastian Katz – Algorithmen für Ad-hoc- und Sensornetze
Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Karlsruher Institut für Technologie
14/ 32
Synchronisation Zeitsynchronisation

## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal NB-SAFE-$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass NB-SAFE-$i$ über alle Brückenkanten *empfangen* wurde

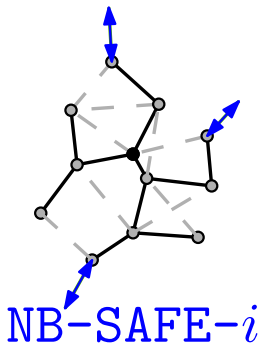



Bild: Roger Wattenhofer

Bastian Katz – Algorithmen für Ad-hoc- und Sensornetze
Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Karlsruher Institut für Technologie
14/ 32

## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal NB-SAFE-$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass NB-SAFE-$i$ über alle Brückenkanten *empfangen* wurde

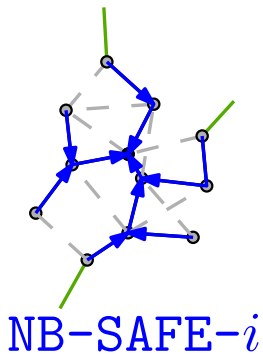



Bild: Roger Wattenhofer

Bastian Katz – Algorithmen für Ad-hoc- und Sensornetze
Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Karlsruher Institut für Technologie
14/ 32

## Synchronisator $\gamma$

**Synchronisation**

Zum Beenden von Runde $i$

1. lassen sich alle Knoten wie in Synchronisator $\beta$ bestätigen, dass im eigenen Cluster alle fertig sind
2. Knoten mit Brückenkanten senden ein Signal `NB-SAFE-`$i$ über Brückenk.
3. wie in Synchronisator $\beta$ lassen sich alle bestätigen, dass `NB-SAFE-`$i$ über alle Brückenkanten *empfangen* wurde

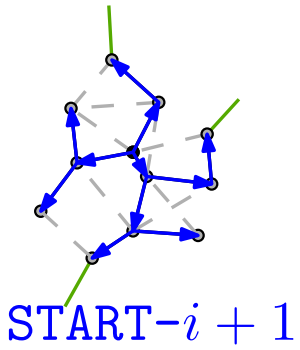



Bild: Roger Wattenhofer





## Analyse Synchronisator $\gamma$

**Satz**

Ist $k$ der maximale Abstand eines Knotens zur Wurzel in seinem Cluster und gibt es insgesamt $m_C$ Brückenkanten, dann hat Synchronisator $\gamma$ Zeitkomplexität $O(k)$ und benötigt $O(n + m_C)$ Nachrichten pro Runde.

- ≫ über jede Baumkante gehen 4, über jede Brückenkante 2 Nachrichten
- ≫ Zeitkomplexität ergibt sich vor allem aus der Synchronisation in der Gruppe
- ≫ genau betrachtet sind $\alpha$ und $\beta$ nur Sonderfälle von $\gamma$
  - ≫ $\alpha$: Jeder Knoten ist seine eigene Gruppe
  - ≫ $\beta$: Alle Knoten sind in derselben Gruppe

Ist das nun besser? Gibt es *gute* Gruppierung?

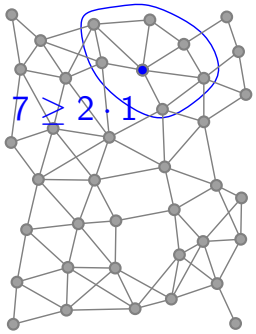


## Gute Partitionen

**Satz**

Jeder Graph lässt sich so partitionieren, dass $k \leq \log_2 n$ und $m_C \leq n$.

1. Wähle bel. unmarkierten Knoten $v$ aus
2. lasse Radius $r$ wachsen, so lange sich Menge der unmarkierten Knoten in $r$-Hop-Nachbarschaft von $v$ noch verdoppelt
3. markiere unmarkierte Knoten in $r$-Hop-Nachbarschaft
4. falls es unm. Knoten gibt, gehe zu 1



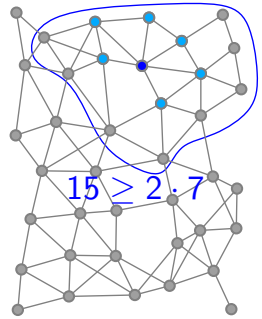



## Gute Partitionen

**Satz**

Jeder Graph lässt sich so partitionieren, dass $k \leq \log_2 n$ und $m_C \leq n$.

1. Wähle bel. unmarkierten Knoten $v$ aus
2. lasse Radius $r$ wachsen, so lange sich Menge der unmarkierten Knoten in $r$-Hop-Nachbarschaft von $v$ noch verdoppelt
3. markiere unmarkierte Knoten in $r$-Hop-Nachbarschaft
4. falls es unm. Knoten gibt, gehe zu 1

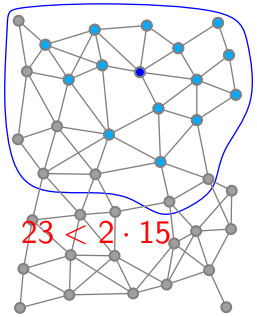


## Gute Partitionen

**Satz**

Jeder Graph lässt sich so partitionieren, dass $k \leq \log_2 n$ und $m_C \leq n$.

1. Wähle bel. unmarkierten Knoten $v$ aus
2. lasse Radius $r$ wachsen, so lange sich Menge der unmarkierten Knoten in $r$-Hop-Nachbarschaft von $v$ noch verdoppelt
3. markiere unmarkierte Knoten in $r$-Hop-Nachbarschaft
4. falls es unm. Knoten gibt, gehe zu 1





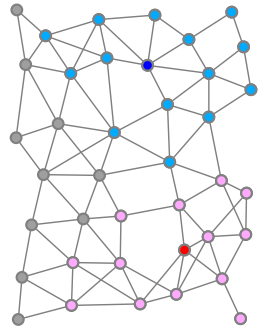

Synchronisation Zeitsynchronisation

## Gute Partitionen

**Satz**

Jeder Graph lässt sich so partitionieren, dass $k \leq \log_2 n$ und $m_C \leq n$.

1. Wähle bel. unmarkierten Knoten $v$ aus
2. lasse Radius $r$ wachsen, so lange sich Menge der unmarkierten Knoten in $r$-Hop-Nachbarschaft von $v$ noch verdoppelt
3. markiere unmarkierte Knoten in $r$-Hop-Nachbarschaft
4. falls es unm. Knoten gibt, gehe zu 1



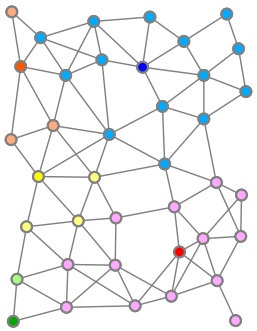
Bastian Katz – Algorithmen für Ad-hoc- und Sensornetze
Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Karlsruher Institut für Technologie
16/ 32
Synchronisation Zeitsynchronisation

## Gute Partitionen

**Satz**

Jeder Graph lässt sich so partitionieren, dass $k \leq \log_2 n$ und $m_C \leq n$.

1. Wähle bel. unmarkierten Knoten $v$ aus
2. lasse Radius $r$ wachsen, so lange sich Menge der unmarkierten Knoten in $r$-Hop-Nachbarschaft von $v$ noch verdoppelt
3. markiere unmarkierte Knoten in $r$-Hop-Nachbarschaft
4. falls es unm. Knoten gibt, gehe zu 1



Bastian Katz – Algorithmen für Ad-hoc- und Sensornetze
Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Karlsruher Institut für Technologie
16/ 32

## Bemerkungen

- So eine Partition kann man auch verteilt im asynchronen Modell berechnen
  - damit ist Synchronisation mit logarithmischem Zeitoverhead realistisch!
  - $O(n)$ Zeit und $O(m + n \log n)$ Nachrichten fürs Setup (o.B.)
  - lohnen sich die Initialisierungskosten?
  - was passiert in dynamischen Netzen?
- Viele synchrone Algorithmen müssten nicht immer alle Knoten synchronisieren
  - in solchen Situationen kann man mit speziellen Synchronisatoren viel Ressourcen sparen

Bastian Katz – Algorithmen für Ad-hoc- und Sensornetze

## Zeitsynchronisation: Motivation

Viele Anwendungen brauchen nicht nur vage Runden, sondern echte Zeitstempel, um

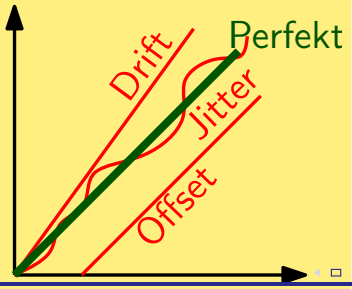
- zum richtigen Moment aufzuwachen
- gemessene Daten richtig zeitlich einordnen zu können
- gemeinsame Schedules einzuhalten

Zeitsynchronisation ist notwendig, um

- Offset (einmalig?) und
- Drift (permanent?)

auszugleichen.

Beispiel: TinyNode: Drift bis zu 30-50$\mu$s/s






## Externe Zeitgeber

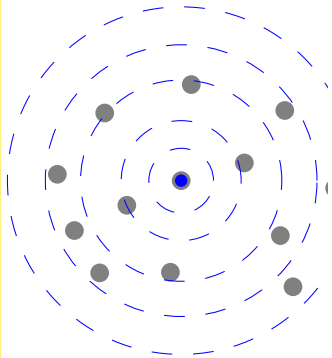
- DCF77 in Frankfurt
  - Synchronisation wie Funkuhren
  - braucht speziellen Empfänger
  - Laufzeit zertört Genauigkeit!
- GPS
  - nur bei freier Sicht
  - nur mit spezieller Hardware
  - dafür für bessere zeitliche Auflösung ausgelegt



## Reference-Broadcast

Einzelne Knoten (*Beacons*) senden eine kurze Nachricht, jeder Knoten merkt sich dann die Ankunftszeit gemäß der eigenen Uhr.

- Knoten können austauschen, wann sie das Signal empfangen haben und daraus ihren gegenseitigen Offset berechnen
- Pakete können dann sogar Multi-Hop geleitet werden, wenn je zwei kommunizierende Knoten einen gemeinsames Signal gehört haben



\+ Sehr einfaches Protokoll, regelmäßige Signale reichen aus
– keine wirklich gemeinsame Zeit
– Auflösung begrenzt durch Signallaufzeit!



## Sender/Empfänger-Synchronisation

Idee: Nutze Roundtrip-Zeit, um Signallaufzeit $\delta$ und (aktuellen) Offset $\theta$ auszurechnen:

$$\delta = \frac{(t_4 - t_1) - (t_3 - t_2)}{2} \qquad \theta = \frac{(t_1 - t_2) + (t_3 - t_4)}{2}$$

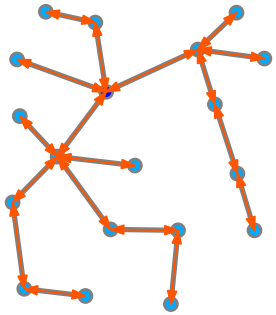

Damit man wirklich *nur* die Signallaufzeit misst: Zeitstempel so spät wie möglich in das Paket einfügen, und beim Empfangen eines Paketes direkt den Zeitstempel ermitteln.

- Es geht nicht um exaktes $\delta$, sondern um die Vermeidung von Effekten, die die beiden Übertragungen asymmetrisch lang machen!

## Baum-basierte Multi-Hop-Synchronisierung

**Typische Struktur für Multi-Hop Synchronisierung**

- ≫ Spanne Baum auf
- ≫ synchronisiere in regelmäßigen Abständen gegen die Zeit der Wurzel
  - ≫ entweder einfach durch Fluten der Zeiten oder
  - ≫ durch paarweisen Abgleich im Baum



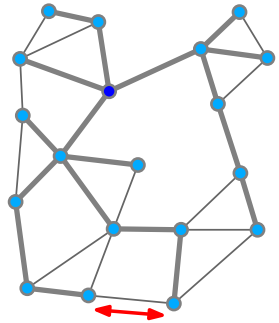


Synchronisation Zeitsynchronisation

## Baum-basierte Multi-Hop-Synchronisierung

**Typische Struktur für Multi-Hop Synchronisierung**

- ≫ Spanne Baum auf
- ≫ synchronisiere in regelmäßigen Abständen gegen die Zeit der Wurzel
  - ≫ entweder einfach durch Fluten der Zeiten oder
  - ≫ durch paarweisen Abgleich im Baum



Was, wenn das Netz eigentlich gar kein Baum ist?

- ≫ nahe Knoten liegen in Baum weit auseinander und sind schlecht synchronisiert!



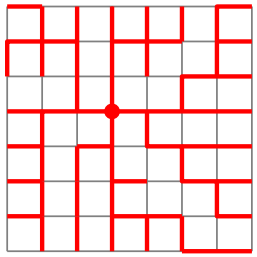
## „Gute“ Bäume für Synchronisation

Um große Fehler zwischen nahen Knoten zu vermeiden, bräuchten wir Bäume mit geringem *Stretch*, z. B. einen der

$$\max_{u,v \in V} \frac{d_T(u,v)}{d_G(u,v)}$$

minimiert.

Das ist nicht nur schwer zu optimieren, sondern auch unbefriedigend: Im $m \times m$-Gitter ist der minimale Stretch $m$. (o.B.)





## Was wir wollen

1. Geringen Zeitunterschied zwischen beliebigen Knoten
   - ≫ besser als durch Bäume bekommt man das nicht hin – Knoten mit Abstand $d$ sind im schlimmsten Fall $O(d)$ auseinander, wenn sich die kleinen Fehler alle in eine Richtung aufsummieren.
2. Nachbarn sollen einen möglichst kleinen Zeitunterschied haben
   - ≫ das ist oft viel wichtiger!
   - ≫ mit Bäumen ist das nicht zu schaffen!
3. Uhren sollten nicht rückwärts springen
   - ≫ dann käme die Ordnung völlig durcheinander
4. Uhren sollen sich immer vorwärts bewegen
   - ≫ sonst wäre eine stehende Uhr eine gute Uhr..

## Gradientensynchronisation

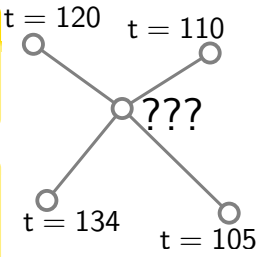
**Idee**

Nutze ein lokales Protokoll, das zur Synchronisation aus den Werten der Nachbarn einen vernünftigen Wert für die eigene Uhr ableitet.

**Modell**

Jede Uhr bewegt sich mit einer Geschwindigkeit zwischen $1-\epsilon$ und $1+\epsilon$.

Was wäre eine gute Strategie, um geringen globale Unterschiede mit geringen lokalen Unterschieden zu kombinieren?








## Zeitsynchronisation `MAX`

Wenn einer der Nachbarn eine höhere Zeit sendet, passe die eigene Uhr an.

**Satz**

`MAX` kann einen lokalen Unterschied von bis zu $D$ erzeugen!


- betrachte Pfad mit $D$ Knoten
  - alle haben Geschwindigkeit 1, nur der linke hat $1+\epsilon$
  - wenn lange Zeit alle Signale Laufzeit 1 haben, hat irgendwann jeder Knoten Abstand 1 nach links und rechts!
  - Kette von schnellen Nachrichten verbreitet die höchste Zeit!
  - $\Rightarrow$ Unterschied von $D$ zwischen Nachbarn!

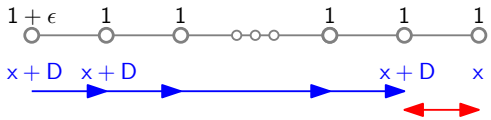


## Zeitsynchronisation `MAX`

Wenn einer der Nachbarn eine höhere Zeit sendet, passe die eigene Uhr an.

**Satz**

`MAX` kann einen lokalen Unterschied von bis zu $D$ erzeugen!




- betrachte Pfad mit $D$ Knoten
  - alle haben Geschwindigkeit 1, nur der linke hat $1+\epsilon$
  - wenn lange Zeit alle Signale Laufzeit 1 haben, hat irgendwann jeder Knoten Abstand 1 nach links und rechts!
  - Kette von schnellen Nachrichten verbreitet die höchste Zeit!
  - $\Rightarrow$ Unterschied von $D$ zwischen Nachbarn!



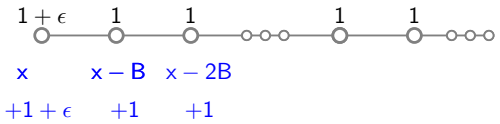
## Mehr schlechte Ideen

- `MIN`: Wähle Minimum
  - Halt! Das dürfen wir nicht! Uhren müssen vorwärts laufen!
- `AVERAGE`: Bilde Durchschnitt der umgebenden Knoten
  - *noch viel schlimmer*! Unterschied zwischen Nachbarn bis zu $\Theta(D)$, globale Unterschiede bis zu $\Theta(D^2)$! (ohne Beweis)

## BOUND-$B$: Warum sollte das besser sein?

Wähle Maximum umgebender Knoten, aber höchstens Minimum$+B$!

- Durch Sychronisation kann man keinen aktiven Fehler machen!
  - kein Knoten kann sich durch Synchronisation von langsamerem Nachbarn entfernen!
- aber durch Warten auf aufschließende Knoten




- wenn die Kette lang genug ist haben wir irgendwann *genau* Abstand $B$ zwischen Nachbarn
- ab dem Moment driftet der schnellste Knoten, ohne dass der zweite aufholen kann
- $\Rightarrow$ Skip wächst auf bis zu $\Theta(D)$ an, so weit kann der Knoten ganz links hinterherhängen!





## BOUND-$\sqrt{D}$

Genauere Analyse: Unterschied zwischen Nachbarn durch *Warten* kann durch $O(D/B)$ beschränkt werden. (Machen wir nicht)

$\Rightarrow$ Setze $B = \sqrt{D}$ für einen Unterschied von höchstens $\sqrt{D}$!

**Satz (o.B.)**

BOUND-$\sqrt{D}$ synchronisiert benachbarte Knoten bis auf einen Unterschied von höchstens $\sqrt{D}$.



## Bemerkungen

- $\sqrt{D}$ ist noch nicht optimal!
- seit 2004 ist eine untere Schranke von $\Omega(\log D / \log \log D)$ bekannt
  - das heißt ja noch nicht, dass es so gut geht, aber
- seit 2008 Lenzen et al. Algorithmus mit $O(\log D)$ Zeitunterschied zwischen Nachbarn
  - das scheint optimal zu sein



## Zum Mitnehmen

- Synchronisation erlaubt uns, synchrone Algorithmen auch im Asynchronen Kommunikationsmodell auszuführen
  - Synchrone Algorithmen sind besser zu verstehen
  - aber Synchronisation kostet Zeit und Energie
  - Mit höherem Initialen Aufwand billigere Synchronisation
  - angepasste asynchrone Algorithmen trotzdem überlegen
- Zeitsynchronisation ist oft notwendig, aber nicht ganz leicht
  - lokale Uhren driften ab und starten asynchron
  - externe Zeitgeber sind nicht immer denkbar
  - Perfekte Synchronisation geht nicht
  - schon lokal vernünftige Synchronisation ist eine harte Nuss

# Literatur


1. B. Awerbuch: *Complexity of network synchronization*. In: *Journal of the ACM*, 32(4):804–823, 1985
2. T. Locher and R. Wattenhofer: *Oblivious Gradient Clock Synchronization*. In: *20th Int'l Symp. on Distributed Computing (DISC'06), 2006*