Algorithmen für verteilte Systeme **Universität Salzburg**
Sommersemester 2019 **FB Computerwissenschaften**

Vorlesung 9: 22. Mai 2019

Vortrag: Sebastian Forster *Mitschrift: Florian Feik*

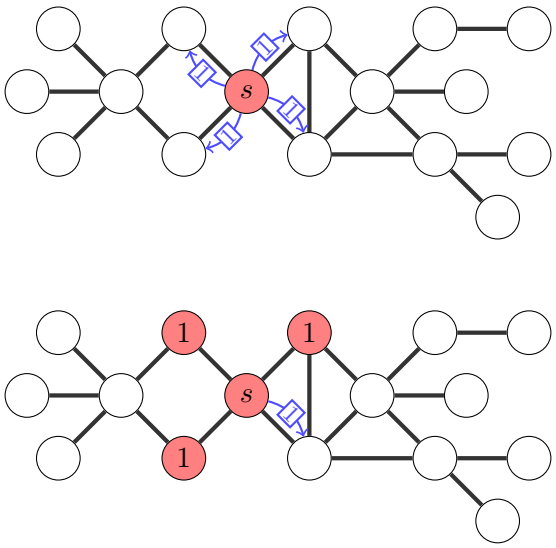
9.1 Synchronisierung

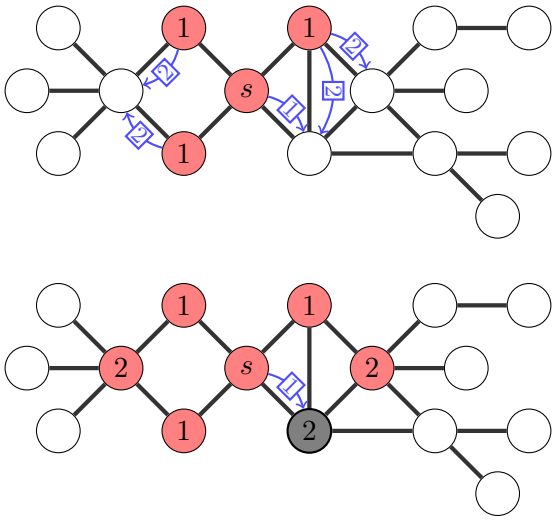
9.1.1 Motivation

Die Annahme, dass es in verteilten Systemen synchrone Runden gibt, ist idealisiert, da in realen Anwendung die Übermittlung von Nachrichten nicht unbedingt in konstanter Zeit geschieht. Im CONGEST Modell wird diese Eigenschaft vernachlässigt, da jegliche Kommunikation synchron und ohne Verzögerung abläuft. Das Ziel ist die synchrone Algorithmen auf einem asynchronen Modell mit moderaten Overheads in der Zeit- und Nachrichtenkomplexität zu simulieren. Bei einer naiven Ausführung von synchronen Algorithmen in asynchronen Modellen würde es zu Inkonsistenzen bei den Ergebnissen kommen.

9.1.2 Synchrone Breitensuche im asynchronen Modell

Bei einer Ausführung des Breitensuche Algorithmus ohne Adaptionen im asynchronen Modell kann es zu Inkonsistenzen kommen, wenn sich die Übertragungsdauer zwischen den einzelnen Knoten unterscheidet.
Dem grau markiertem Knoten wird die falsche Distanz zugewiesen. Wir müssen also in der adaptieren Version nachträgliche Reduzierung der Distanzen ermöglichen und diese Änderungen im Netzwerk weitergeben.





9.2 Asynchrones Modell

Der größte Unterschied zum synchronen Modell ist der Wegfall der Runden, also den diskreten Zeitschritten. Stattdessen gibt es zwei verschiedene Events:

- Die Initialisierung eines Knotens von außerhalb des Netzwerks.
- Der Empfang einer Nachricht an einem Knoten von einem anderen Knoten.

Der Empfang einer Nachricht aktiviert den Knoten und dieser kann dann die erhaltene Nachricht verarbeiten. Danach kann er eine neue Nachricht an seine Nachbarknoten schicken, wobei die Dauer der Übertragung endlich sein muss. Für eine Laufzeitanalyse, wird in der Regel, pro Übertragung, der maximale Delay mit einer Zeiteinheit betrachtet. Die Korrektheit eines Algorithmus muss jedoch für beliebig Delays gelten.

9.3 Synchronizer

Synchronizer ermöglichen das simulieren von synchronen Algorithmen in asynchronen Netzwerken.

9.3.1 Pulsgeber

Auf jedem Knoten läuft ein zusätzlicher Pulsgeber-Algorithmus (Synchronizer), der lokal für jeden Knoten einen Puls generiert. Dieser Puls ist jedoch nicht synchron mit den Pulsen der anderen Knoten. Dieser Puls kann analog zu einer Runde im synchronen Modell betrachtet werden. Der Knoten führt also im Puls i die Runde i aus (Verarbeiten, Senden und Empfangen von Nachrichten). Eine Simulation gilt dann als korrekt, wenn Puls $i+1$ immer erst generiert wird, wenn der Knoten als *safe* gilt, was der Fall ist, wenn die Nachrichten aller Nachbarn für Puls i empfangen wurden. Das heißt jedoch nicht, dass er von jedem Nachbarn eine Nachricht erhalten muss, da dieser nicht in jeder Runde eine Nachricht verschicken muss.

9.3.2 Safety

Die Safety beschreibt den Zustand eines Knotens für den Puls i. Der Knoten gilt genau dann als safe, wenn er die Simulation (Verarbeiten, Senden und Empfangen von Nachrichten) der aktuellen Runde i bereits abgeschlossen hat.

Definition 9.1. *Ein Knoten ist* safe *für Puls i, wenn alle seine in der Simulation von Runde i gesendeten Nachrichten die jeweiligen Nachbarn erreicht haben.*

Die Safety kann durch das Versenden von Bestätigungsnachrichten nach dem Empfangen jeder Nachricht überprüft werden. Jeder Empfänger einer Nachricht verschickt also eine Bestätigung an den Sender. Dadurch verdoppelt sich die Nachrichten- und Zeitkomplexität, da für jede gesendet Nachricht zusätzlich eine Bestätigung gesendet werden muss.

In weiterer Folge bedeutet dies auch, dass ein synchroner Algorithmus korrekt simuliert wird, wenn für jeden Knoten der Puls $i+1$ immer erst dann generiert wird, wenn der Knoten selbst und alle Nachbarn safe für Puls i sind. Für den einzelnen Knoten hat das zur Folge, dass er jede Information für die Runde $i+1$ erhalten hat, da er nur Nachrichten von seinen Nachbarn erhalten kann und diese keine Nachrichten mehr schicken werden, da sie bereits safe sind. Für die Überprüfung der Safety können verschiedene Synchronizer eingesetzt werden.

9.3.3 Synchronizer α

Beim Synchronizer α entscheidet jeder Knoten, selbst ob er safe ist und in den nächsten Puls übergehen kann.

9.3.3.1 Algorithmus [Awe85]

- Sobald ein Knoten safe ist, also alle Bestätigungsnachrichten für seine gesendeten Nachrichten empfangen hat, dann sendet er eine safe-Nachricht an alle seine Nachbarn.
- Der Knoten erkennen dadurch sofort, wenn alle Nachbarn safe sind.
- Sobald ein Knoten von allen Nachbarn eine safe-Nachricht empfangen hat, kann er mit dem nächsten Puls fortsetzen.

9.3.3.2 Analyse

Der größte Vorteil diese Algorithmus ist, dass er rein lokal auf jedem Knoten funktioniert. Durch das Verschicken der safe-Nachricht entsteht nur ein konstanter Overhead in der Zeitkomplexität. Die Nachrichtenkomplexität wird jedoch enorm erhöht, da jeder Knoten zusätzlich in jedem Puls an alle Nachbarn eine safe-Nachricht schicken muss, was einen großen Nachteil dieses Algorithmus darstellt.

Theorem 9.2. *Jeder synchrone Algorithmus, der R Runden benötigt und dabei M Nachrichten versendet, kann im asynchronen Modell in Zeit $O(R)$ mit $O(M + R \cdot m)$ Nachrichten simuliert werden.*

9.3.4 Synchronizer β

Beim Synchronizer β wir der Puls von einem globalen Leader generiert.

9.3.4.1 Algorithmus [Awe85]

- Ausgangssituation: Im Netzwerk wurde ein Breitensuchbaum berechnet, an dessen Wurzel sich der Leader des Netzwerks befindet.
- Der Leader sendet den Puls i mittels Downcast an alle Knoten im Baum.
- Jeder Knoten führt beim Erhalt des Pulses i die Runde i aus.
- Knoten ist wieder safe, sobald er alle Empfangsbestätigungen erhalten hat.
- Sobald ein Blatt safe ist, sendet es eine safe-Nachricht an den Elternknoten.
- Sobald der Elternknoten von allen Kindern eine safe-Nachricht erhalten hat, sendet er diese wiederum an seinen Knoten.
- Durch diesen aggregierten Upcast erfährt schlussendlich der Leader, dass alle Knoten safe sind.
- Sobald alle Knoten safe sind, schickt der Leader den Puls $i + 1$.

9.3.4.2 Analyse

Im Gegensatz zum Synchronizer α hat der Synchronizer β eine niedrigere Nachrichtenkomplexität, da für jede Kante im Baum (maximal n Kanten) genau zwei zusätzliche Nachrichten verschickt werden. Der große Nachteil ist jedoch, dass ein Breitensuchbaum und ein Leader im asynchronen Netzwerk bereits bestimmt werden mussten. Zusätzlich erhöht sich die Laufzeit um $2 \cdot D$, da der Up- und Downcast jeweils ein Laufzeit von $O(D)$ haben.

Theorem 9.3. *Jeder synchrone Algorithmus, der R Runden benötigt und dabei M Nachrichten versendet, kann im asynchronen Modell in Zeit $O(R \cdot D)$ mit $O(M + R \cdot n)$ Nachrichten simuliert werden, wenn bereits ein Leader und ein Breitensuchbaum berechnet wurden.*

9.3.5 Asynchrone Breitensuche

Der benötigte Breitensuchbaum kann durch verschiedene Algorithmen berechnet werden. Zwei wurden in der Vorlesung erwähnt:

1. **Dijkstra-Algorithmus**: Zeit $O(D^2)$, Nachrichten: $O(m + n \cdot D)$
2. **Bellman-Ford-Algorithmus**: Zeit: $O(D)$, Nachrichten: $O(n \cdot m)$

   Der Algorithmus verwendet den gleichen Breitensuche-Algorithmus, wie wir ihn in der Vorlesung für das synchronen Modell besprochen haben, nur, dass die Distanz eines Knotens zur Wurzel, durch später ankommende Nachrichten noch verringert werden kann. Damit kann das in 9.1.2 entstandene Problem verhindern werden und ermöglicht, dass jedem Knoten die korrekte Distanz zugewiesen wird. Das führt jedoch dazu, dass durch das mehrfache setzten der kürzesten Distanzen, eine höhere Nachrichtenkomplexität $O(n \cdot m)$ statt $O(m)$ erhält. Durch die Verwendung des Synchronizer α erreicht man jedoch die gleichen Garantien im bezug auf Korrektheit.

9.4 Hybride Synchronisierung

9.4.1 Netzwerkpartitionierung

Für den folgenden Algorithmus gehen wir davon aus, dass wir einen partitionierten Graphen haben. Das ist eine Aufteilung des Graphen in Subgraphen mit den Parametern δ und μ. In jedem dieser Subgraphen ist zusätzlich ein Breitensuchbaum, der von einem Zentrum (analog zu Leader) aus, mit Durchmesser höchstens δ, konstruiert worden ist. Formal gesagt bedeutet das:

Definition 9.4. *Eine (δ, μ)-Netzwerkpartitionierung ist eine Partitionierung von V in Cluster $V_1, \ldots, V_k$ so dass für jedes i der von V_i induzierte Subgraph $G[V_i]$ Durchmesser höchstens δ hat und die Anzahl an Kanten mit Endpunkten in verschiedenen Knotenmengen höchstens μ ist.*

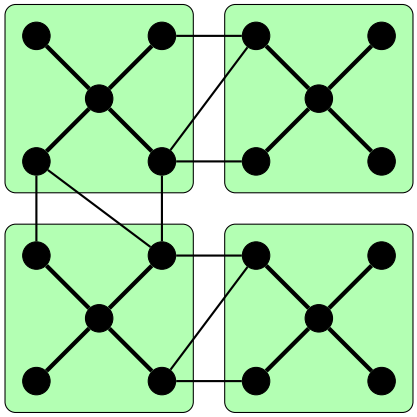


Abbildung 9.1: Beispiel für Netzwerkpartitionierung

Abbildung 9.1 ist ein Beispiel für eine solche Partitionierung mit den Parametern $\mu = 9$ und $\delta = 2$. Die Kanten der Breitensuchbäume in den Clustern sind dicker dargestellt, als die Verbindungskanten zwischen den Clustern.

9.4.2 Synchronizer γ

Der Synchronizer γ ist eine Kombination der zuvor beschriebenen Synchronizer. Synchronizer α wickelt die Pulsweitergabe zwischen den Clustern ab und Synchronizer β findet innerhalb der Cluster Verwendung.

9.4.2.1 Algorithmus [Awe85]

- Ausgangssituation: Das Netzwerk wurde in Cluster partitioniert.
- Der Algorithmus beginnt indem alle Zentren, also die Leader der Cluster, einen Puls senden, was jeden Knoten dazu bringt, eine Runde zu simulieren.
- Innerhalb jedes Clusters läuft nun der Algorithmus nach dem Synchronizer β, wie in 9.3.4.1 beschrieben ist, ab. Dabei starten die Blätter des Breitensuchbaum im Cluster

einen aggregierten Upcast, der dazu führt, dass des Zentrum im Cluster weiß, wann alle Knoten im Cluster safe sind.
- Sobald alle Knoten im Cluster safe sind, gilt der gesamte Cluster als safe. Das Zentrum des Clusters schickt nun eine cluster-safe-Nachricht mittels Broadcast an alle Nachbarcluster. Knoten im Cluster leiten diese Nachricht einfach an alle Kanten, die aus dem Cluster führen, weiter.
- Jeder Cluster wird nun wie ein Knoten im Algorithmus des Synchronizer α (9.3.3.1) betrachtet. Da jedes Zentrum alle Nachbarcluster kennt, weiß es, sobald er von jedem dieser Cluster eine cluster-safe-Nachricht erhalten hat, dass es wieder einen Puls senden darf.
- Nun beginnt der Algorithmus von vorne und das Zentrum im Cluster wartet wieder bis es von allen Knoten im Cluster ein safe-Nachricht bekommen hat.

9.4.2.2 Laufzeit

Die Laufzeit ist nun Abhängig von den Partitionierungsparameter δ und μ. Je nachdem wie die Cluster ausgewählt werden kann zwischen der optimalen Zeitkomplexität des Synchronizer α und der besseren Nachrichtenkomplexität des Synchronizer β variiert werden. Im allgemeinen gilt, dass bei kleineren Clustern die Zeitkomplexität verringert, die Nachrichtenkomplexität jedoch erhöht wird.

Theorem 9.5. *Jeder synchrone Algorithmus, der R Runden benötigt und dabei M Nachrichten versendet, kann im asynchronen Modell in Zeit $O(R \cdot \delta)$ mit $O(M + R(\mu + n))$ Nachrichten simuliert werden, wenn bereits eine (δ, μ)-Netzwerkpartitionierung berechnet wurde.*

9.5 Zusammenfassung

9.5.1 Zeit- und Nachrichtenkomplexität

	Zeit	**Nachrichten**	**Initialisierungsaufwand**
Synchronizer α	$O(R)$	$O(M + R \cdot m)$	–
Synchronizer β	$O(R \cdot D)$	$O(M + R \cdot n)$	Breitensuchbaum
Synchronizer γ	$O(R \cdot \delta)$	$O(M + R(\mu + n))$	(δ, μ)-Netzwerkpartitionierung

Tabelle 9.1: Übersicht der Synchronizer [Awe85]

9.5.2 Spezielle Netzwerkpartitionierung

Theorem 9.6. *Für jedes $k \geq 1$ gibt es eine (δ, μ)-Netzwerkpartitionierung mit $\delta = O(k)$ und $\mu = O(n^{1+1/k})$.*

Aufgrund des Theorems 9.6 kann man eine Netzwerkpartitionierung finden, die eine asymptotische Laufzeit des Algorithmus von $O(R \cdot \log n)$ und eine Nachrichtenkomplexität von $O(R \cdot n)$. Das bedeutet jedoch nicht automatisch, dass so eine Aufteilung in der Praxis sinnvoll ist, da der Partitionierungsvorgang dadurch sehr komplex werden kann.

9.5.3 Randomisierten Synchronizer [APSP+92]

Es gibt einen randomisierten Synchronizer mit polylogarithmischen Overheads in Zeit- und Nachrichtenkomplexität und ohne Initialisierungsaufwand, der mit hoher Wahrscheinlichkeit korrekt ist.

Literatur

[APSP+92] Baruch Awerbuch, Boaz Patt-Shamir, David Peleg und Michael Saks. "Adapting to Asynchronous Dynamic Networks (Extended Abstract)". In: *Proc. of the Symposium on Theory of Computing (STOC)*. 1992, S. 557–570. DOI: 10.1145/129712.129767 (siehe S. 9-7).

[Awe85] Baruch Awerbuch. "Complexity of Network Synchronization". In: *Journal of the ACM* 32.4 (1985), S. 804–823. DOI: 10.1145/4221.4227 (siehe S. 9-3, 9-4, 9-5, 9-6).