Verteilte Netzwerk-Algorithmen

Bericht Modul BTI7311 Informatik Seminar

Studiengang:	Informatik
Autor:	Adrian Bärtschi
Betreuer:	Peter Schwab
Datum:	09.05.2015

Versionen

Version	Datum	Status	Bemerkungen
0.1	28.03.2015	Entwurf	Setup Template
0.2	26.04.2015	Entwurf	Disposition, Einleitung
0.3	05.05.2015	Entwurf	Inhalte
1.0	09.05.2015	Final	Fazit, Korrekturen, Abschluss

Inhaltsverzeichnis

1. Einleitung

Um in verteilten, miteinander verbunden Systemen Information zu gewinnen, werden Algorithmen mit speziellen Eigenschaften benötigt. Diese Algorithmen laufen meistens gleichzeitig auf den verschiedenen Knoten eines Netzwerks. Die einzelnen Teilprogramme kennen dabei nicht das ganze Netz, sondern lediglich ihre direkten Nachbarn. Netzwerkalgorithmen müssen also autonom auf den Knoten laufen können und dürfen nicht auf feste Grössen (zum Beispiel Anzahl Netzwerkteilnehmer) angewiesen sein. Es muss dazu sichergestellt sein, dass Entscheidungen und Berechnungsresultate alle gewünschten Knoten erreichen und dass sich so eine gewisse Konsistenz bilden kann.

Typische Schwierigkeiten von verteilte Algorithmen sind das Koordinieren der einzelnen Prozesse bei teilweisen Ausfällen und unzuverlässigen Verbindungen.

Praktische Anwendungen von verteilten Algorithmen sind zum Beispiel Routing, Spanning Trees oder Transaktionssteuerung bei verteilten Datenbanken.

2. Grundlagen

2.1. Message-Passing Modell

Um Abläufe in verteilten System zu beschreiben, wird das Message-Passing Modell genutzt. Dabei wird das Netzwerk als Graph abgebildet. Prozessoren stellen dabei die Knoten dar und direkte Verbindungen werden durch die dazugehörigen Kanten repräsentiert. Die Teilnehmer kommunizieren untereinander, indem sie Messages versenden und empfangen (in der Praxis z. Bsp. TCP/IP Pakete). Dazu muss jeder Knoten eindeutig identifizierbar sein, beispielweise via MAC-Adresse, und die direkten Nachbarn müssen bekannt sein.

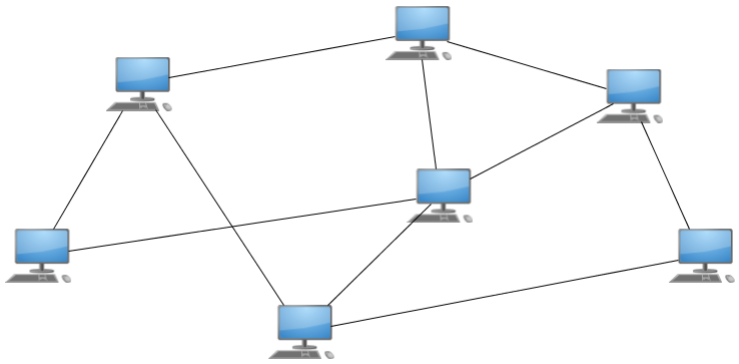

Abbildung 2.1.: Schematische Darstellunge eines Netzwerks als Graph

Teilweise ist auch noch die Gesamtanzahl Teilnehmer in Netzwerk bekannt, häufig ist dies aber aufgrund der Grösse und Dynamik nicht möglich.
In verteilten Systemen kann es vorkommen, dass sich die Struktur ändert, sich neue Teilnehmer anmelden oder dass Knoten und Verbindungen sich fehlerhaft verhalten oder ganz ausfallen. Verteilte Algorithmen müssen grundsätzlich auch mit solchen Problemen umgehen können, in diesem Bericht wird aber darauf nicht näher eingegangen.
Bei den behandelten Anwendungen wird angenommen, dass sich das Netzwerk zur Laufzeit nicht ändert und dass alle Teilemhmer und Verbindungen ohne Fehler funktionieren.

Ein wichtiger Aspekt des Message-Passings ist die Synchronisation der Prozesse. Es gibt dabei verschiedene Ansätze:

Synchrones Modell: Beim synchronen Modell geht man davon aus, dass bei jedem Prozessor ein interner Timer läuft. Die Timer laufen alle genau gleich schnell und die Prozessoren benötigen für die gleiche Aufgabe die gleiche Zeit. Ausserdem wird angenommen, dass das Versenden von Nachrichten immer die gleiche Zeit dauert, egal über welche Verbindung dies geschieht.

Asynchrones Modell: In asynchronen Modellen gibt es keinen einheitlichen Takt unter den Prozessoren. Man muss annehmen, dass die Knoten unterschiedlich schnell arbeiten. Um trotzdem verteilte Algorithmen einsetzten zu

können, wird bei jedem Knoten eine Queue eingeführt, mit welcher ankommende Nachrichten zwischengespeichert werden.

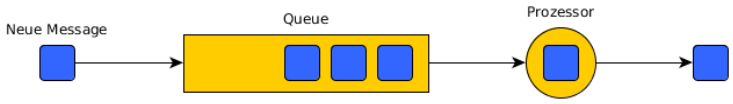

Abbildung 2.2.: Prozessor mit first-in-first-out Queue

Die meisten Netzwerke in der Praxis orientieren sich eher am asynchronen Modell. Gewisse Aspekte aus dem synchronen Modell, wie zum Beispiel ein Timeout falls die Übertragung zu lange dauert, sind jedoch auch häufig anzutreffen. Netzwerkalgorithmen für synchrone Netzwerke können meistens für das asynchrone Modell erweitert werden. Ein Algorithmus, der für ein asynchrones Netzwerk funktioniert, läuft mit Sicherheit auch bei in einen synchronen Netz.

2.2. Einheiten für die Komplexität der Algorithmen

Bei normalen, sequenziellen Algorithmen sind für die Ermittlung der Komplexität meistens die Laufzeit und der benötigte Speicherplatz relevant. Bei verteilten Systemen reichen diese Kriterien nicht mehr aus, um einen Algorithmus beurteilen zu können. Folgende Kennwerten spielen eine Rolle:

- **Anzahl Berechnungsrunden (Computational Rounds)**
  Die Algorithmen kommen nach einer gewissen Anzahl Berechnungsrunden zum Ergebnis. Bei synchronen Algorithmen kann man die Anzahl Ticks der Timer als Messgrösse benutzen. Im asynchronen Modell laufen die Algorithmen häufig in Wellen von Events durch das Netz. Hier kann somit die Anzahl Wellen eine relevante Grösse sein.
- **Benötigter Speicherplatz**
  Es gibt zwei Kenngrössen für den benötigen Speicher des Algorithmus. Zum einen kann der lokale Speicherplatz auf einem Knoten eine wichtige Information sein. Zum anderen kann über das ganze Netz die Gesamtsumme des benötigten Speichers ausgewertet werden.
- **Lokale Laufzeit**
  Die lokale Laufzeit der Algorithmen auf den Knoten ist auch in verteilten System noch eine wichtige Grösse. Es kann vorkommen, dass die Knoten unterschiedliche Laufzeiten haben, da z. Bsp. Knoten am Rand des Netzes spezielle Aufgaben übernehmen müssen.
- **Anzahl und Grösse der Messages**
  Typischerweise wird analysiert, wie viele Messages für die Ausführung einer Berechnungsrunde versendet werden müssen. Bei Algorithmen mit unterschiedlichen Messagetypen ist auch die Grösse der unterschiedlichen Messages zu beachten.

2.3. Leader Election Ring

Ein typisches Beispiel eines Netzwerkalgorithmus ist Leader Election. Die Ausgangslage ist ein Ringnetzwerk, bei dem jeder Knoten über eine Nummer identifizierbar ist.
Das Ziel ist nun, den Knoten mit der höchsten (oder tiefsten) Nummer zu finden und diese Information im gesamten Ring bekannt zu geben. Somit sollen sich alle Knoten einig sein, welcher als Leader gewählt wurde.
Die Knoten kennen lediglich ihre Nummer und den nächsten Knoten, and den sie als einziges Messages senden können.

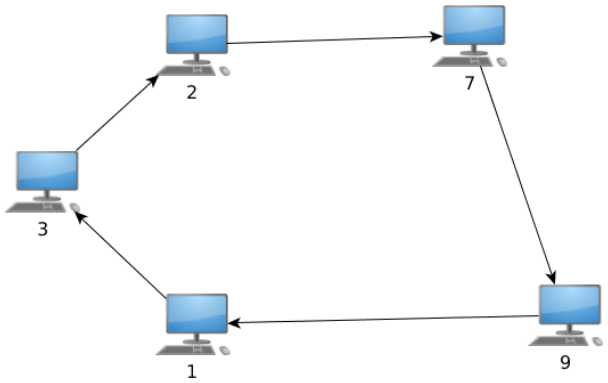

Abbildung 2.3.: Ausgangslage Leader Election Ring

Die Lösung dieses Problems funktioniert nach folgendem Ansatz:

1. Jeder Knoten sendet seine Nummer zum nächsten Knoten
2. Wenn eine Nummer empfangen wurde, wird diese mit der eigenen Nummer verglichen
    a) Die grössere der beiden Nummern wird weitergeschickt
    b) Falls die empfangene Nummer die eigenen Nummer ist, ist der Leader gefunden worden

Es werden zwei verschiedene Messagetypen eingeführt. 'Candidate is ...' um einen möglichen Leader dem nächsten Knoten zu melden und 'Leader is ...', sobald klar ist, welcher Knoten als Leader bestimmt wurde. In der ersten Phase werden so lange 'Candidate is' Messages versendet, bis die Nummer des Leaders den gesamten Ring durchlaufen hat. Sobald die Message den Ursprungsknoten erreicht hat, wird begonnen 'Leader is' Messages zu versenden um die anderen Knoten zu informieren.
Dieser Lösungsansatz setzt voraus, dass die Nummern der Knoten eindeutig sind.

```
send Message [Candidate is myId]

finished = false
while not finished
    M = Get Next Message from incoming Queue
    if M is "Candidate is ..." Message then
        if M.id = myId them
            send Message [Leader is myId]
            finished = true
        else
            send Message [Candidate is max(M.id, myId)]
    else
        send M
        finished = true
```

Listing 2.1: Pseudocode Asynchrone Leader Election in einen Ring Netzwerk

Eine Implementation des obigen Algorithmus in Java ist in Anhang A zu finden.

Bei einem Ring mit n Knoten sendet in der ersten Phase jeder Knoten n 'Candidate is' Messages. Somit werden in der ersten Phase $O(n^2)$ Messages versendet.
Sobald der Leader bestimmt ist, muss die 'Leader is' Message eine komplette Runde durch den Ring absolvieren. In dieser Zeit senden die anderen Knoten weiter 'Candidate is' Messages, bis die 'Leader is' Message sie erreicht hat. Durchschnittlich versendet jeder Knoten noch $\frac{n}{2}$ 'Candidate is' Messages. Die Message Komplexität des Ring Leader Election Algorithmus entspricht also $O(n^2)$.

2.4. Leader Election Tree

Für ein Netzwerk das eine Baumstuktur hat, muss der Leader Election Algorithmus natürlich angepasst werden. Der Algorithmus läuft in zwei Phasen ab.

1. Die **Akkumulationsphase** startet bei den externen Knoten (Knoten mit nur einem Nachbarn). Diese senden ihre Nummer an den Nachbar. Jeder Knoten wartet und speichert die eingehenden Nachrichten, bis er von allen ausser einem Nachbarn eine Nachricht erhalten hat. Dann wird die grösste bekannte Nummer an den verbleibenden Knoten gesendet. Sobald ein Knoten von allen Nachbarn Nachrichten erhalten hat, ist der gesamte Baum durchlaufen worden und der verbleibende Knoten kann entscheiden, welcher Teilnehmer als Leader gewählt wird.

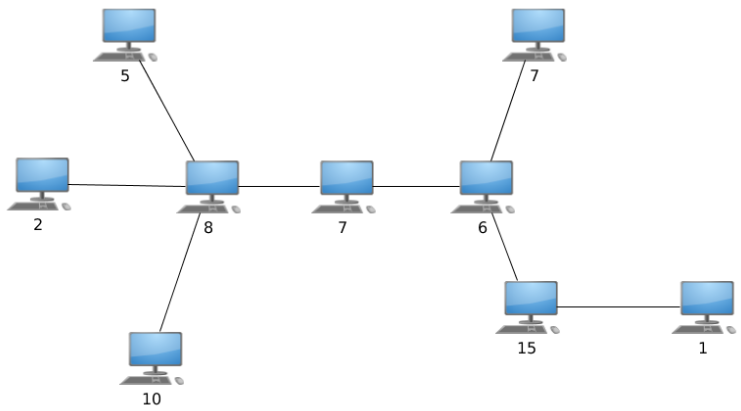

Abbildung 2.4.: Leader Election Tree Ausgangslage

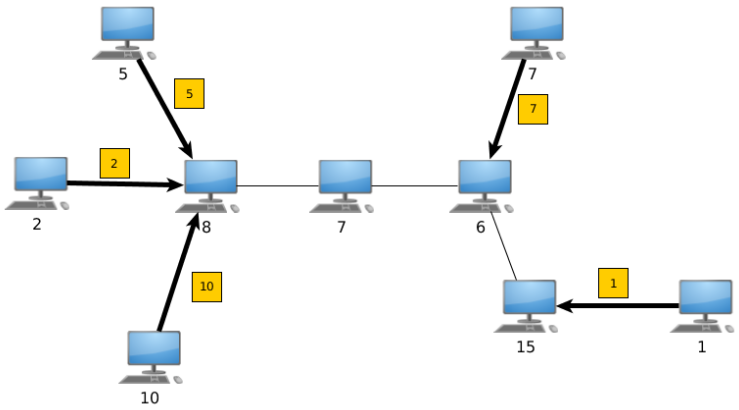

Abbildung 2.5.: Leader Election Tree Accumulation Schritt 1

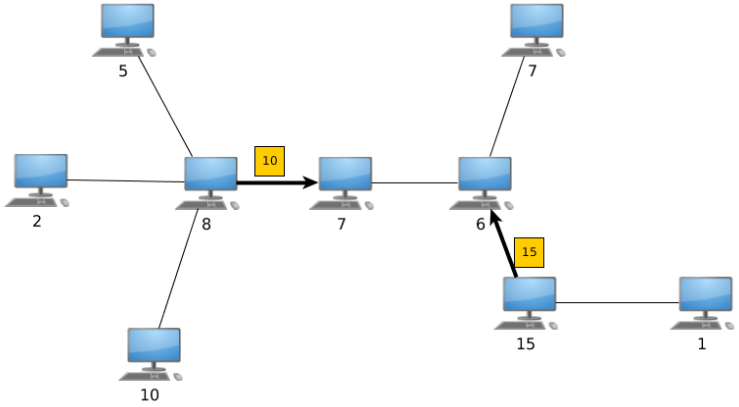

Abbildung 2.6.: Leader Election Tree Accumulation Schritt 2

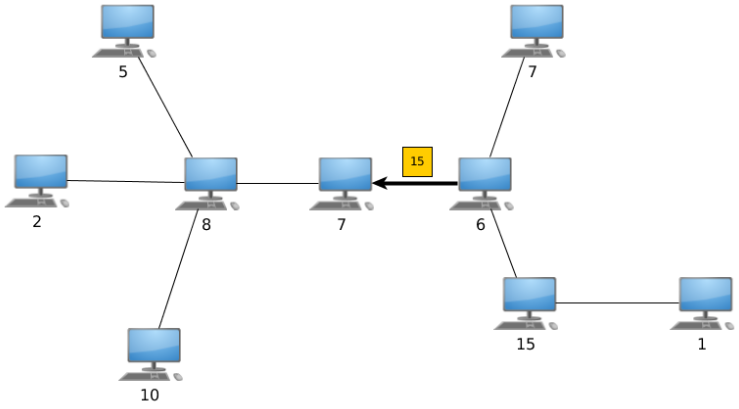

Abbildung 2.7.: Leader Election Tree Accumulation Schritt 3

2. Der letzte Node startet nun die **Broadcastphase**, bei der die Leader Information an jeweils alle Knoten gesendet wird (ausgenonnen vom Nachbarn, von dem die Broadcast Nachricht erhalten wurde).

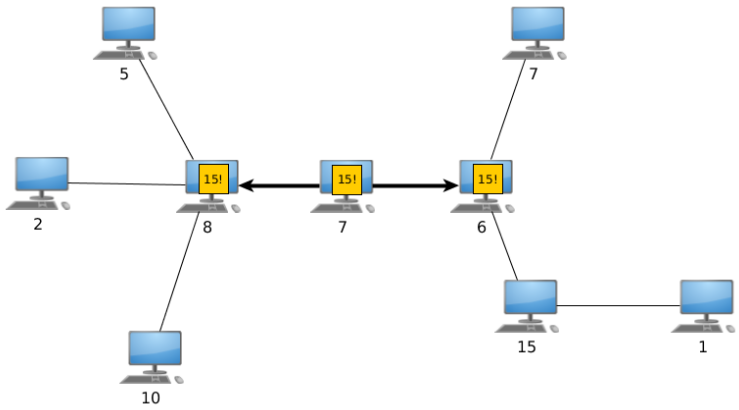

Abbildung 2.8.: Leader Election Tree Broadcast Schritt 1

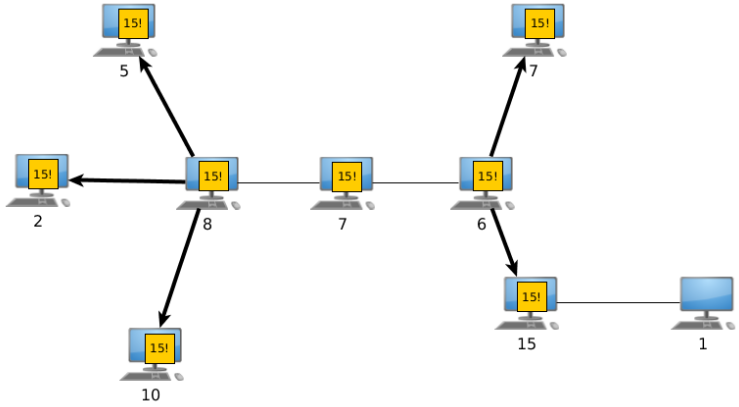

Abbildung 2.9.: Leader Election Tree Broadcast Schritt 2

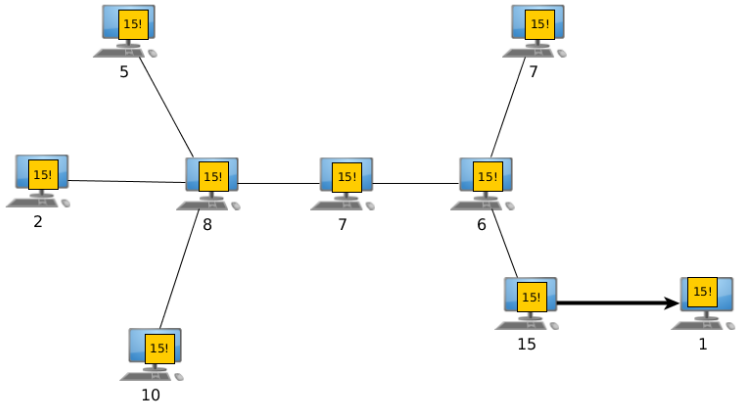

Abbildung 2.10.: Leader Election Tree Broadcast Schritt 3

Bei der asynchronen Variante dieses Algorithmus sendet während der Akkumulationsphase jeder Knoten genau eine 'Candidate is' Nachricht. Für den Broadcast wird schlussendlich an jeden Knoten eine 'Leader is' Message versendet. Die Messages haben eine Grösse von $O(1)$. Somit ist die Message Komplexität bei einem Baum mit n Knoten $O(n)$.

Ein Beispiel für eine praktische Anwendung von Leader Election ist der Clusterverbund von mehreren Servern. Dabei muss ein Server als Leader ermittelt werden, von dem dann koordinative Aufgaben wie Deployment und Logging übernommen werden.

2.5. Minimum Spanning Trees

Der Minimum Spanning Tree eines Graphen ist ein Subgraph, der alle Knoten erreicht und dessen summiertes Gewicht der Kanten minimal ist [7].

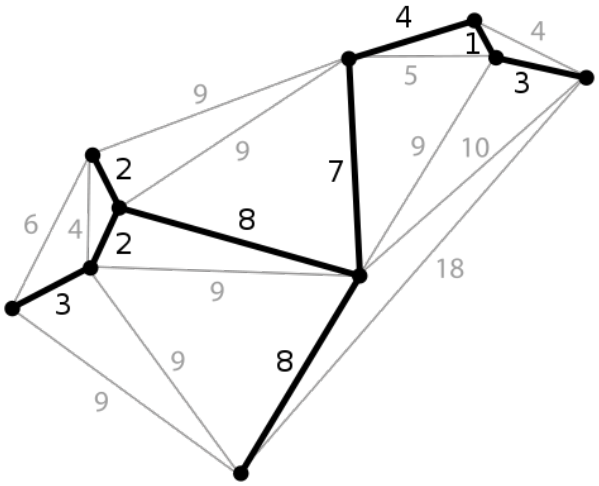

Abbildung 2.11.: Beispiel eines Minimum Spanning Trees [7]

Eine effiziente sequenzielle Lösung für das finden des Minimum Spanning Trees ist der Algorithmus von Baruvka [5]. Die Idee dabei ist, von jedem Knoten ausgehend Cluster mit dem Nachbarn zu bilden, der über die Kante mit dem kleinsten Gewicht verbunden ist. Der am Schluss entstandenen Graph ist der Minimum Spanning Tree.

Um in verteilten Systemen zu funktionieren, muss der Algorithus angepasst werden. Nachfolged ist die synchrone, verteilte Variante der Minimum Spanning Tree Generierung beschrieben.

1. Da die einzelnen Knoten nur ihre direkten Nachbarn kennen, müssen mit Hilfe des Tree Leader Election Algorithmus (siehe Kapitel 2.4) die zusammenhängenden Komponenten bestimmt werden.
2. Zu jeder zusammenhängenden Komponente muss die Kante gefunden werden mit dem tiefsten Gewicht, die einen neuen Knoten mit der zusammenhängenden Komponente verbindet. Auf jedem Knoten wird nun erneut eine Leader Election durchgeführt, nun mit den angrenzenden Kanten respektiv deren Gewicht.
3. Dies läuft so lange, bis alle Knoten erschlossen wurden.

In einem Graphen mit n Knoten und m Kanten werden pro Runde $O(m)$ Messsages versendet. Die Anzahl zusammenhängender Komponenten im Graphen halbiert sich mit jeder Runde. Somit gibt es $O(\log n)$ Runden, bis der Spanning Tree gefunden wurde. Dies ergibt eine Message Komplexität von $O(m \log n)$.

Minimum Spanning Trees werden beispielsweise verwendet, um in einem Netzwerk möglichst effizient eine Nachricht an alle Teilnehmer zu versenden (Broadcast).

3. Routing

3.1. Broadcast Routing mit Flooding

Ein einfacher Weg, um eine Nachricht von einem Teilnehmer aus an alle anderen des Netzwerks zu senden, ist Flooding. Der Knoten, der senden möchte, sendet die Nachricht an alle seine Nachbarn. Diese leiten die Nachricht an alle Nachbarn weiter, ausser dem Knoten, von dem sie die Nachricht erhalten haben. Das Problem dabei ist, dass dieser Algorithmus so nie enden würde (ausgenommen es gibt keine Zyklen im Netz).

Ein Lösungsansatz ist die Einführung eines numerischen **Hop Counters**, der auf den Nachrichten geführt wird. Bei der Weiterleitung auf einen Noten wird der Hop Counter immer um eins verringert. Ist der Counter bei Null angelangt, wird die Nachricht nicht mehr weitergeleitet, sondern verworfen. Um sicher zu sein, dass alle Knoten des Netzes erreicht werden, wird der Hop Counter initialisiert mit dem Durchmesser (längste Distanz zwischen zwei Knoten) des Netzes.

Ein andere Möglichkeit, unendliche Broadcasts zu verhindern, ist mittels **Sequenznummern**. Wenn der Quellknoten die Nachrichten erstellt, wird jeder Nachricht eine eindeutige Sequenznummer zugewiesen. Jeder Knoten führt bei sich eine Tabelle, in der gespeichert wird, welche Sequenznummern von welchen Ursprungsknoten bereits erhalten wurden. Falls bereits ein Eintrag existiert, wird die Nachricht verworfen, andernfalls wird sie in die Tabelle eingetragen und anschliessend weitergeleitet an alle Nachbarn. Problematisch kann bei dieser Variante der Speicherplatzbedarf auf den Knoten werden, um sich alle Nachrichten aller Knoten zu merken.

Da die Sequenznummern immer hochgezählt werden, kann pro Knoten auch nur die jeweils höchste Nummer gespeichert werden. Man nimmt dabei an, dass die Broadcast Nachrichten grösstenteils mit aufsteigender Sequenznummer eintreffen.

Wegen der tieferen Message Komplexität (Tabelle 3.1) wird normalerweise die Sequenznummer Variante der Hop Counter Methode vorgezogen. Nur bei Netzen resp. Graphen mit sehr geringem Durchmesser würde die Hop Counter Variante Vorteile bieten.

3.2. Unicast Routing mit Distance Vector

Um in einem Netzwerk die kürzesten Pfade zwischen den verschiedenen Knoten zu finden, kann das Distance Vector Protokoll verwendet werden. Distance Vector kann mit verschienden Algorithmen realisiert werden. Hier wird die Variante basierend auf dem Algorithmus von Bellmann und Ford [4] beschrieben.

Es wird angenommen, dass alle Kanten im Graphen ein positives Gewicht haben. Das Ziel ist, dass jeder Knoten weis, an welchen Knoten er ein eingegangenes Paket weiterleiten muss, damit es über den kürzesten Weg (geringstes Gewicht) zum Zielknoten kommt. Dafür wird auf jedem Knoten eine Liste geführt, welche das Gesamtgewicht des Pfades zu jedem anderen Knoten enthält.

In einem Graphen mit vier Knoten A, B, C und D könnte die Liste auf Knoten A zum Beispiel so aussehen, wenn der Algorithmus fertig gelaufen ist:

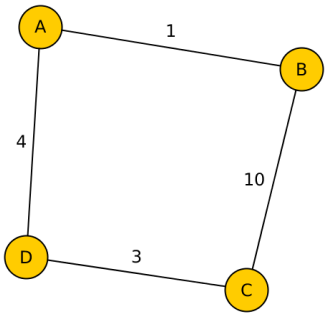

Abbildung 3.1.: Beispielgraph für Distance Vector

Ziel	Gewicht	weiter via
A	-	-
B	1	B
C	7	D
D	4	D

Abbildung 3.2.: Distance Vector auf Knoten A

Die Einträge für die direkten Nachbarn können sofort erstellt werden. Die Gewichte für die anderen Knoten werden mit "Infinity" initialisiert. Nun werden die Distance Vector Vector Informationen wird rundenbasiert aufgebaut.

In jeder Runde sendet jeder Knoten seine Distance Vector Information an alle direkten Nachbarn. Sobald ein Knoten eine Distance Vector Nachricht erhalten hat, wird diese ausgewertet. Wenn eine kürzere Route gefunden wurde, wird der entsprechende interne Distance Vector Enitrag aktualisiert.

Um die Distance Vector Tabelle aufzubauen, muss jeder Knoten x $O(d_x n)$ Messages pro Runde verarbeiten (d_x ist Anzahl Kanten des Knoten x). Im gesamten Netzwerk ergibt das $O(nm)$ Messages pro Runde. Der Algorithmus braucht so viele Runden, wie der Durchmesser D des Graphen ist. Somit ergibt sich eine Message Komplexität von $O(Dnm)$ [1].

Das Routing Infomation Protocol (RIP) [2] verwendet den Distance Vector Algorithmus.

3.3. Unicast Routing mit Link-State

Ein anderer Algorithmus für Unicast Routing ist Link-State. Die Idee ist, dass in einer ersten Phase alle Knoten die Gewichte ihrer angrenzenden Kanten via Broadcast mit Sequenznummern im Neztwerk bekanntgeben. Danach ist auf jedem Knoten der 'Link-State' aller Knoten gespeichert und mittels dem Algorithmus von Dijkstra werden lokal die kürzesten Wege zu jedem Knoten berechnet und der jeweils erste Knoten im Pfad dahin wird gespeichert.

Das Routing Protocol OSPF [3] ist ein bekanntes Beispiel für Link-State Routing.

Die lokale Berechung mittels einer Standard Implementation von Dijkstra läuft mit Zeitkomplexität $O(m \log n)$ [6]. Der Speicherverbrauch auf den Knoten ist jeweils $O(n)$.

3.4. Vergleich von Broadcast und Unicast Algorithmen

Folgende Tabelle vergleicht die vier beschriebenen Routing Algorithmen bezüglich folgender Kennwerte:

- Messages: Anzahl Messages, die versendet werden
- Lokaler Speicher: Speicher, der auf jedem Knoten benötigt wird
- Lokale Zeit: Laufzeitverhalten auf den Knoten für die Weiterleitung einer Meldung
- Routing Zeit: Laufzeitverhalten für die Wegfindung einer Message

Der lokale Berechnungsaufwand, der auf den Knoten für den Aufbau von Datenstrukturen benötigt wird, wird hier nicht berücksichtigt.

Algorithmus	**Messages**	**Lokler Speicher**	**Lokle Zeit**	**Routing Zeit**
Flooding Hop Counter	$O(1)$	$O(1)$	$O(d)$	$O((d_{max}-1)^D)$
Flooding Sequenz Nr.	$O(1)$	$O(n)$	$O(d)$	$O(m)$
Distance Vector	$O(Dnm)$	$O(n)$	$O(1)$	$O(p)$
Link-State	$O(m^2)$	$O(n)$	$O(1)$	$O(p)$

Tabelle 3.1.: Vergleich der asymptotischen Komplexität von statischen Routing Algorithmen [1]

- n: Anzahl Knoten im Netz
- m: Anzahl Kanten im Netz
- d: Maximaler Grad (max. Anzahl Kanten eines Knoten)
- D: Durchmesser des Netzwerks
- p: Anzahl Knoten des kürzesten Pfades

4. Fazit

Verteilte Algorithmen sind in der heutigen Zeit nicht mehr wegzudenken und viele Anwendungen haben sich über viele Jahre und Jahrzehnte in der Praxis bewährt.

Um die Problemstellungen in verteilten System zu lösen, müssen die Algorithmen spezielle Bedingungen erfüllen. Korrekte und dabei immer noch effiziente Lösungen zu finden, wird besonders bei asynchronen Abläufen schwieriger.

Es hat sich gezeigt, dass viele Konzepte von sequenziellen Algorithmen übernommen werden können, um ähnliche Probleme in Netzwerken zu lösen. Es sind jedoch zusätzliche Kennwerte für die Komplexitätsanalyse notwendig, um die Algorithmen beurteilen zu können.

Mit den momentan sehr wichtigen Themen Cloud- und Mobile Computing sind verteilte Systeme und Algorithmen vermehrt wieder relevant geworden in modernen Anwendungen.

Literaturverzeichnis

[1] M. T. Goodrich and R. Tamassia, *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2006.

[2] C. L. Hedrick, "Routing information protocol," 1988.

[3] J. Moy, "rfc 2328: Ospf version 2," 1998.

[4] Wikipedia, "Bellman–ford algorithm — wikipedia, the free encyclopedia," 2015, [Online; accessed 9-May-2015]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Bellman%E2%80%93Ford_algorithm&oldid=659009747

[5] ——, "Borůvka's algorithm — wikipedia, the free encyclopedia," 2015, [Online; accessed 3-May-2015]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Bor%C5%AFvka%27s_algorithm&oldid=653267410

[6] ——, "Dijkstra's algorithm — wikipedia, the free encyclopedia," 2015, [Online; accessed 9-May-2015]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=661260204

[7] ——, "Minimum spanning tree — wikipedia, the free encyclopedia," 2015, [Online; accessed 3-May-2015]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Minimum_spanning_tree&oldid=660266484

Abbildungsverzeichnis

A. Implementation Leader Election Ring

Der Sorucecode ist ebenfalls auf folgendem Github Repository verfügbar:
`https://github.com/barta3/ch.bfh.bti7311.DistributedAlgo`

```
1  package ch.bfh.bti7311.DistributedAlgo.leaderElection;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class LeaderElectionRing {
7
8    public static void main(String[] args) throws InterruptedException {
9
10     List<Node> ring = createRing();
11
12     for (Node n : ring) {
13       System.out.println(n.toString());
14       new Thread(n).start();
15     }
16   }
17
18   private static List<Node> createRing() {
19
20     int[] testdata = new int[] { 29, 74, 14, 55, 58 };
21     int size = testdata.length;
22
23     List<Node> ring = new ArrayList<Node>();
24     for (int i = 0; i < size; i++) {
25       Node n = new Node(testdata[i]);
26       ring.add(n);
27     }
28
29     for (int i = 0; i < ring.size(); i++) {
30       Node n = ring.get(i);
31       Node next = ring.get(i < size - 1 ? i + 1 : 0);
32       n.setNext(next);
33     }
34
35     return ring;
36   }
37
38 }
```

```java
package ch.bfh.bti7311.DistributedAlgo.leaderElection;

public class Message {

  private int id;
  private Type type;

  public enum Type {
    CANDIDATE_IS, LEADER_IS
  }

  public Message(Type type, int id) {
    this.setId(id);
    this.setType(type);
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Type getType() {
    return type;
  }

  public void setType(Type type) {
    this.type = type;
  }

  @Override
  public String toString() {
    return type + " " + id;
  }

}
```

```java
package ch.bfh.bti7311.DistributedAlgo.leaderElection;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

import ch.bfh.bti7311.DistributedAlgo.leaderElection.Message.Type;

public class Node implements Runnable {

  private int id;
  private int currLeader;
  private Node next;

  private BlockingQueue<Message> queue = new LinkedBlockingQueue<Message>();

  public Node(int value) {
    this.id = value;
    this.currLeader = value;
  }

  public Node getNext() {
    return next;
  }

  public void setNext(Node next) {
    this.next = next;
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public void putInQueue(Message m) {
    try {
      queue.put(m);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  @Override
  public String toString() {
    return String.format("ID %d next: %d", id, next.getId());
  }

  public void run() {

    this.getNext().putInQueue(new Message(Type.CANDIDATE_IS, this.id));
    boolean done = false;

    Message msgArr;
    Message msgSend = null;
    try {
      while (!done) {
```

```
59          msgArr = queue.take();
60          System.out.println(id + " Received " + msgArr);
61
62          if (msgArr.getType().equals(Type.CANDIDATE_IS)) {
63            if (msgArr.getId() == this.id) {
64              done = true;
65              msgSend = new Message(Type.LEADER_IS, this.currLeader);
66            } else {
67              currLeader = Math.max(id, msgArr.getId());
68              msgSend = new Message(Type.CANDIDATE_IS, currLeader);
69            }
70          } else {
71            done = true;
72            this.currLeader = msgArr.getId();
73            msgSend = new Message(Type.LEADER_IS, msgArr.getId());
74          }
75          System.out.println(id + " Send " + msgSend);
76          this.getNext().putInQueue(msgSend);
77
78        }
79      } catch (InterruptedException e) {
80        e.printStackTrace();
81      }
82
83      System.out.println(id + " FINISHED - Leader is " + currLeader);
84    }
85
86  }
```