NVS Projekt: Distance-Vector Algorithmus

Florian Schwarzl, 5BHIF, 21

8. April 2021

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Internet als globales Netzwerk ist schon seit geraumer Zeit ein zentrales Element unserer Gesellschaft. Die Technologien, auf denen es Aufgebaut ist, sind mannigfaltig und werden zunehmend mehr. Trotzdem sind alte und gut bewährte Ansätze weiterhin im Einsatz. Zu diesen bereits lange im Einsatz befindlichen Technologien gehört auch das Routing. Routing, zu Deutsch Weiterleitung, ist ein Kernkonzept einer jeden Kommunikation. Ohne eine funktionierende Weiterleitung von einem Knoten zum Nächsten ist eine Kommunikation über mehrere Stellen nicht möglich. Die Notwendigkeit für Routing besteht in jedem Netzwerk, welches aus mehreren Knoten besteht. Dieses Projekt befasst sich mit einem dieser Routing-Algorithmen, dem Distance-Vektor-Algorithmus. Das Projekt ist in C++ geschrieben und verwendet das Meson-Buildsystem.[1]

[1] *Meson Build System.*

Kapitel 2

Aufgabenstellung

Die Aufgabenstellung bestand daraus, den Routing-Algorithmus in einer lokalen Simulation zu implementieren. Die lokale Implementierung soll auf einer Maschine laufen, jedoch trotzdem den Netzwerk-Stack durchlaufen. Daher werden anstatt von IP- oder MAC-Adressen, welche normalerweise für das Routing verwendet werden, die Port-Nummern zur Identifikation verwendet. Zur Simulation des Algorithmus gehört auch ein fiktiver Netzwerkgraph. Jeder Knoten soll selbstständig sein und von anderen unabhängig seine Aufgaben erfüllen. Zum Aufbau der Netzwerkverbindung wird die Bibliothek asio verwendet.[1] Asio ist eine weit verbreitete Bibliothek zur synchronen und asynchronen Netzwerkkommunikation in C++.

[1] *asio C++ library.*

Kapitel 3

Routing

Das Routing umfasst das Suchen des schnellsten Weges zwischen zwei Knoten eines Netzwerkes. Es ist nicht zu verwechseln mit dem Weiterleiten (engl. Forwarding), welches sich genau genommen nur mit dem Weiterleiten von Nachrichten beschäftigt, unter der Annahme, dass bereits der kürzeste Weg bekannt ist. Diese beiden Aufgaben hängen jedoch insofern zusammen, als dass für das Weiterleiten einer Nachricht der kürzeste Weg bekannt sein muss. Daher, und weil Routing-Protokolle meist auch die Weiterleitung übernehmen, werden Routing und Weiterleiten oft als Synonyme verwendet. Die zentrale Aufgabe des Routings ist es, eine Routingtabelle (Weiterleitungstabelle) zu erstellen. Diese Tabelle enthält für jeden bekannten Knoten den kürzesten Weg, genauer gesagt den nächsten Knoten des kürzesten Weges zum Ziel. Diese Weiterleitungstabelle wird, wie der Name vermuten lässt, beim Weiterleiten von Nachrichten verwendet, um den nächsten Knoten (den nächsten Hop) zu bestimmen. Die Metrik, welche für die Bestimmung des kürzesten Weges verwendet wird, wird als Kosten bezeichnet. Jeder Kante im Netzwerkgraph werden bestimmte Kosten zugeordnet. Diese Kosten können sich aus mehreren Eigenschaften wie beispielsweise der Übertragungsleistung, der Zuverlässigkeit oder ähnlichem zusammensetzen. In diesem vereinfachten Beispiel werden als Kosten die Distanz zum nächsten Knoten genommen. Somit hat jede Kante im Netzwerkgraph einen Kostenwert von 1.[1]

3.1 Routing-Algorithmen

Um die oben genannten Aufgaben zu erfüllen wurden im Laufe der Zeit eine Vielzahl von Algorithmen mit unterschiedlichen Eigenschaften entworfen. Es gibt verschiedene Ansätze die sich in Aufbau, Anpassungsfähigkeit und allgemeiner Funktionsweise unterscheiden. In den folgenden Sektionen wird die Aufteilung der Verfahren und ihre Unterschiede besprochen.

[1] Kolousek, *Entwurf und Implementierung verteilter Systeme.*

3.1.1 Statisches Verfahren

Dieses Verfahren ist das simpelste, jedoch in der Praxis kaum anwendbar. Die Weiterleitungstabelle eines jeden Knoten wird beim Aufsetzen des Netzwerks einmalig angelegt, danach kann diese nur noch manuell geändert werden. Aufgrund der statischen Eigenschaft dieses Verfahrens können temporäre Ausfälle von Knoten nur schwer berücksichtigt werden, daher wird es nur wenn überhaupt für provisorische Netzwerke mit simpler, sich kaum verändernder Topologie verwendet.

3.1.2 Zentrale Verfahren

Hier liegt die Routingtabelle bei einer zentralen Kontrolleinheit, welche sie an alle Knote weitergibt. Änderungen im Netzwerk müssen von der Zentrale erkannt werden und an alle Knoten weitergegeben werden. Um dies zu ermöglichen muss der Zentrale das gesamte Netzwerk bekannt sein und eine entsprechende Leistung und Ausfallsicherheit gegeben sein. Wie bei allen zentral geregelten Algorithmen ist ein Single-Point-of-Failure gegeben, welcher bei einem Ausfall das gesamte System lahmlegen kann.

3.1.3 Isolierte Verfahren

Im Unterschied zu den meisten Routing-Verfahren werden bei isolierten Verfahren keine Routing-Informationen zwischen den Knoten des Netzwerkes versendet. Jeder Knoten ist so gesehen auf sich allein gestellt und kann nur auf die ihm direkt zu Verfügung stehenden Informationen zugreifen. Ein Beispiel für solch eine Information ist die Größe der Warteschlangen an den Ausgängen des Knoten. Zu den isolierten Verfahren zählt auch das sogenannte Hot-Potato-Verfahren (Heiße Kartoffel). Dabei ist das Ziel des Knoten, ein erhaltenes Paket möglichst schnell wieder weiter zusenden. Dadurch hat zwar jeder Knoten einen geringen Rechenaufwand, jedoch kann es bei Paketen zu sehr großen Verzögerungen kommen.[2]

3.1.4 Verteilte Verfahren

Bei einem verteilten Verfahren gibt es, wie bei anderen verteilten Systemen, keine zentrale Steuereinheit. stattdessen ist jeder Knoten selbst für seinen Teil des Rouings verantwortlich und besitzt damit auch seine eigene Weiterleitungstabelle. Genauer werden diese Verfahren auch als verteilte adaptive Verfahren bezeichnet, da sie sich an Ausfälle im Netzwerk selbstständig anpassen können. Ein zentraler Punkt dieser Verfahren ist der Austausch von Routing-Informationen zwischen den Knoten des Netzwerks. Welche Informationen ausgetauscht werden, hängt vom konkreten Verfahren ab.

[2] *Wikipedia, Routing.*

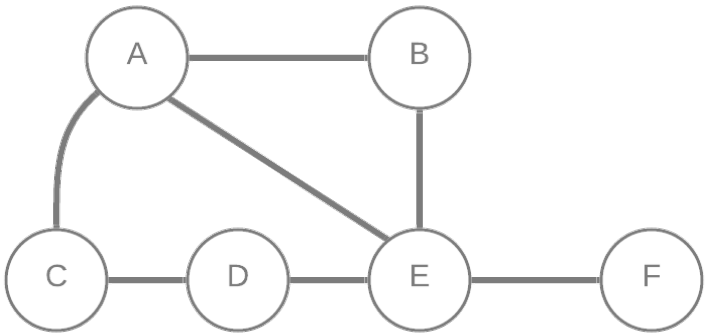

Abbildung 3.1: Beispiel eines Netzwerkgraphen

3.2 Distanzvektor-Algorithmus

Dieser Algorithmus basiert darauf, dass jeder Knoten eine Tabelle (Vektor) verwaltet, in dem die kürzesten Wege zu den anderen Knoten eingetragen sind. Im Gegensatz zu statischen Verfahren wird diese Tabelle dynamisch vom Knoten selbst erstellt und an Veränderungen im Netzwerk angepasst.

3.2.1 Prinzip

Die Tabelle bzw. der Distanzvektor enthält für jeden dem Knoten bekannten Knoten einen Eintrag (einschließlich des Knoten selbst). Zu jedem dieser Einträge wird der nächste Knoten am Weg und die gesamten Kosten bis zum Ziel gespeichert. Die Kosten sind abhängig von der konkreten Implementierung im Routing-Protokoll, für dieses Beispiel werden die Anzahl der Knoten (der Hop-Count) als Kosten verwendet. Der Graph in Abbildung 3.1 zeigt ein Beispielnetzwerk, anhand dessen dieser Algorithmus erklärt wird. In diesem Beispiel gehen wir davon aus, dass jeder Knoten nur seine direkten Nachbarn und nicht das gesamte Netzwerk kennt. Er muss daher nicht nur die Kosten für jeden Knoten, sondern auch die Knoten an sich speichern.

Das Prinzip dieses Algorithmus ist simpel. Zu Beginn werden alle direkten Nachbarn des Knoten und der Knoten selbst in die Tabelle eingetragen. Am Beispiel betrachtet trägt der Knoten A somit B, C und E als seine Nachbarn ein. Zu jedem dieser Nachbarn sind die Kosten gleich 1. Sich selbst trägt der Knoten mit den Kosten von 0 ein. Die Tabelle 3.1 zeigt, wie der Distanzvektor nun in diesem Beispiel aussieht.

Die im Vektor gespeicherten Informationen werden nun an alle Nachbarn weitergeleitet. Wenn ein Knoten von seinem Nachbarn einen Vektor erhält, wird dieser mit dem gespeicherten verglichen. Ist für einen Knoten noch kein Eintrag vorhanden, werden die erhaltenen Werte übernommen, wobei die Kosten um eins erhöht werden. Gibt es für ein Ziel bereits einen Eintrag, werden die Kosten

Tabelle 3.1: Beispiel des Distanzvektor am Beginn

Ziel	**Kosten**	**Nächster Knoten**
A	0	A
B	1	B
C	1	C
E	1	E

Tabelle 3.2: Beispiel des Distanzvektors nach erstem Update

Ziel	**Kosten**	**Nächster Knoten**
A	0	A
B	1	B
C	1	C
E	1	E
D	2	C

verglichen. Sind die um eins erhöhten Kosten des erhaltenen Vektors kleiner, wird der Nachbar, von dem der Vektor stammt, als nächster Knoten eingetragen und die Distanz angepasst. Wenn in unserem Beispiel nun A den Vektor von C erhält sieht der neue Vektor von A wie in Tabelle 3.2 dargestellt aus. Dieser Vorgang geschieht bei jedem Knoten, bis schließlich alle Routen bekannt und im Distanzvektor eingetragen sind. Der endgültige Vektor kann für dieses Beispiel etwa so aussehen wie in Tabelle 3.3

3.2.2 Ausfall einer Verbindung

Um den Ausfall einer Verbindung anhand unseres Beispiels zu erläutern, nehmen wir an, dass die Verbindung zwischen C und D ausgefallen ist. Wird der Ausfall vom Knoten C erkannt, setzt dieser seine Kosten für den Weg nach D auf Unendlich. Dies teilt er A mit, worauf hin auch A seine Kosten für D aktualisiert und dies E mitteilt. E wird seine Kosten für D jedoch nicht ändern, da E den Knoten D noch direkt erreichen kann. Danach teilt es A mit, dass D über ihn erreicht werden kann. A ändert seinen Eintrag für den nächsten Knoten nach D auf E und kann den Knoten D wieder erreichen.

Tabelle 3.3: Beispiel des Distanzvektors am Ende

Ziel	**Kosten**	**Nächster Knoten**
A	0	A
B	1	B
C	1	C
E	1	E
D	2	C
F	2	E

3.2.3 Count-to-Infinity Problem

Bei dieser Art der Ausfallerkennung kann es jedoch zu einem Problem kommen, dass als Count-to-Infinity (Bis zur Unendlichkeit zählen) bezeichnet wird. Angenommen wird, dass die Verbindung zwischen E und F ausfällt. E setzt somit seine Kosten für F auf den Wert für Unendlich. Wenn nun eine Aktualisierung von B kommt, welche für F die Kosten 2 beinhaltet, wird E diese übernehmen, um eins erhöhen und B als nächsten Knoten eintragen. Diese Information sendet er nun zurück an B, welcher ebenfalls seine Kosten für F um eins erhöht und diese Aktualisierung erneut aussendet. Dieser Vorgang wiederholt sich, bis eine gewisse Schranke erreicht wird, ab der abgebrochen wird. Je nach Höhe dieser Schranke und Häufigkeit der Aktualisierungen kann es zu beträchtlichen Verzögerungen kommen, bis ein Ausfall erkannt wird. Um dieses Problem zu umgehen gibt es die Verbesserung des Split Horizon (geteilter Horizont). Diese Verbesserung legt fest, dass ein Knoten keine Aktualisierungen an den Knoten zurückschickt, von dem er den Wert erhalten hat. Dadurch werden jedoch nur Schleifen mit zwei Knoten vermieden.[3]

3.2.4 Anwendungsbeispiel - RIP

Das Routing Information Protocol (RIP) ist ein Beispiel für ein Routing-Protokoll, welches auf Basis des Distanz-Vektor-Algorithmus arbeitet. Es wird primär innerhalb von lokalen Netzwerken verwendet. Es agiert wie der oben beschriebene Algorithmus. Es ist in drei verschiedenen Verionen verfügbar, RIPv1, RIPv2 und RIPng. Letzteres ist eine Erweiterung für die Verwendung von IPv6-Adressen.[4]

[3] Kolousek, *Entwurf und Implementierung verteilter Systeme.*
[4] *Wikipedia, Routing Information Protocol.*

Kapitel 4

Bedienung

Das Programm ist über die Kommandozeile zu starten, indem man das Hauptprogramm `distance_vector_sim` aufruft. Die gesamte Simulation besteht aus zwei ausführbaren Dateien, dem Hauptprogramm und den Knoten. Jeder Knoten ist ein eigener Prozess, in dem die Datei node ausgeführt wird. Das Hauptprogramm dient nur zum erstellen des Netzwerkgraphen und starten der Knoten. Sobald alle Knoten gestartet sind, erfolgt keine Kommunikation von oder zu dem Hauptprogramm. Die folgende Tabelle 4.1 zeigt alle Parameter, welche am Hauptprogramm eingestellt werden können. Die Konfiguration des Programms kann sowohl über die Kommandozeilenparameter als auch über eine JSON-Datei erfolgen.

Keiner der angegebenen Parameter ist verpflichtend, jeder kann weggelassen werden, wodurch die Standardwerte herangezogen werden. Die Parameter der Konfigurationsdatei sind die selben wie jene, die über die Kommandozeile angegeben werden können. Die eiinzige Ausnahme ist der Parameter für die Konfigurationsdatei selbst, welcher nicht in der Datei angegeben werden kann. Die Tabelle 4.2 zeigt alle Parameter, welche in der Konfigurationsdatei angegeben werden können. Für die Parameter der Konfigurationsdatei gelten die gleichen Anforderungen wie für jene der Kommandozeile. So braucht auch hier die Option debug zuvor die Option log.

4.1 Knoten

Jeder Knoten ist ein eigenes Programm, welches automatisch vom Hauptprogramm gestartet wird. Es nicht empfohlen, die Knoten manuell zu starten, da so auch ein sinnvoller Netzwerkgraph händicsh erstellt werden muss und an die Knoten übergeben werden muss. Jeder Knoten bekommt als Parameter seine eigene Portnummer und die Portnummern seiner direkten Nachbarn im Simulierten Netzwerk. Weiter werden die Optionen für Logging analog zum Hauptprozess gesetzt. Die Option für den simulierten Ausfall wird mit –failure und einer danach folgenden Portnummer an zwei benachbarte Knoten übergeben.

Tabelle 4.1: Parameter des Hauptprogramms

Parameter	**Datentyp**	**Standardwert**	**Beschreibung**
node count	ganze Zahl größer 0	7	Anzahl der zu erstellenden Knoten
`-l, -log`	boolean	false	Aktiviert logging
`-d, -debug`	boolean	false	Setzt Log-Level auf Debug (benötigt `-log`
`-f, -file`	Dateipfad		Pfad zur JSON-Konfigurationsdatei
`-failure`	boolean	false	Simuliert den Ausfall einer Verbindung

Tabelle 4.2: Parameter der Konfigurationsdatei

Parameter	**Datentyp**	**Standardwert**
nodes	ganze Zahl größer 0	7
log	boolean	false
debug	boolean	false
failure	boolean	false

Die beiden Knoten erhalten über diese Option die Portnummer zu dem Knoten, dessen Verbindung ausfallen wird. Die beiden Knoten müssen benachbart sein, damit eine direkte Verbindung zwischen ihnen besteht. Auch den Knoten können die Parameter über eine Konfigurationsdatei übergeben werden.

4.2 Logging

Wenn die Option des Logging aktiviert wird, erstellt jeder Prozess eine eigene Datei. In diese Datei wird je nach gesetzten Log-Level Information über den Ablauf des Programms gespeichert. Vermerkt werden sollte, dass in der Datei des Hauptprogramms, welche controller.log heißt, der erstellte Netzwerkgraph gespeichert wird. Somit kann die Korrektheit der Routing-Informationen überprüft werden. Abgesehen von dieser Information werden je nach Parameter primär Informationen über den Programmablauf gespeichert, beispielsweise werden auf Debug-Level jeder Verbindungsauf- und -abbau vermerkt.

Kapitel 5

Implementierung

Dieses Kapitel umfasst die Implementierung des Distanzvektor-Algorithmus und eine Beschreibung des Quellcodes. Das Projekt ist auf zwei ausführbare Dateien aufgeteilt. Die Hauptdatei mit dem Namen `distance_vector_sim` beinhaltet den als Controller bezeichneten Teil. Der zweite Teil ist ein Programm namens node. Dieses stellt einen Netzwerkknoten dar und wird duch den Controller gestartet. Die Knoten können zwar auch manuell gestartet werden, das ist jedoch mit deutlich mehr Aufwand verbunden, da dann der Nutzer einen sinnvollen Netzwerkgraphen selbst erstellen muss und die entsprechenden Nachbarknoten manuell an jeden Knoten übergeben.

5.1 Controller

Der Controller agiert als Ausgangspunkt und ist für das Starten und Beenden der Knoten verantwortlich. Der dazugehörige Code befindet sich im gleichnamigen Unterverzeichnis des src-Verzeichnisses. Der Controller besteht aus einer Main-Funktion und zwei in die Datei utils.cpp ausgelagerte Funktionen. Am Beginn des Controllers werden alle Parameter für die Kommandozeile mit ihren Standardwerten initialisiert. Danach werden mithilfe der Bibliothek CLI11 die übergebenen Parameter gelesen.[1] Sollte eine Konfigurationsdatei angegeben sein, wird diese im nächsten Schritt abgearbeitet, wobei die Werte der Konfigurationsdatei jene der Kommandozeile überschreiben. Dieses Verhalten ist gewollt und erwartet. Die JSON-Datei wird durch die Bibliothek JSON for Modern C++ bearbeitet.[2] Im Codebeispiel 5.1 ist die Initialisierung der Parameter und die Angabe mittels der Bibliothek CLI11 dargestellt.

[1] *CLI11.*
[2] *JSON for Modern C++.*

```
1  bool use_logging{false};
2  bool log_level_debug{false};
3  bool simulate_error{false};
4  size_t node_cnt{7};
5  string config_file{"N/A"};
6
7  const char node_program[7]{"./node"};
8
9  app.add_option("node count", node_cnt, "Total number of nodes. Must
       be > 0.")->check(CLI::PositiveNumber);
10 CLI::Option* log_flag{app.add_flag("-l, --log", use_logging, "Write
       log file dist_sync_log.log.")};
11 app.add_flag("-d, --debug", log_level_debug, "Set log level to
     debug.")->needs(log_flag);
12 app.add_option("-f, --file", config_file, "Path to json config file
     .")->check(CLI::ExistingFile);
13 app.add_flag("--failure", simulate_error, "Simulate failure of one
     connection");
```

Listing 5.1: Angabe der Kommandozeilenparameter

Wenn die Parameter sowohl aus der Datei als auch von der Kommandozeile gelesen und übernommen wurden, werden die Einstellungen für das Logging gesetzt. Dabei wird die Bibliothek spdlog verwendet. Ein sogenannter rotating-file-logger wird als Standard-Logger gesetzt und das Log-Level wird den übergebenen Parametern angepasst.

Der nächste Schritt ist das Erstellen des Netzwerkgraphen. Hierfür wird ein einfacher Algorithmus verwendet, welcher versucht, jeden Knoten mit mindestens einem anderen zu Verbinden. Um sicherzustellen, dass ausreichend Verbindungen erstellt werden, ist die Anzahl der Kanten auf zwei mehr als die Anzahl der Knoten festgelegt. Die Funktion für das Erstellen befindet sich in der Datei utils.cpp. In der dazugehörigen Header-Datei ist auch ein C++-Struct, welches alle Informationen zu einem Knoten speichert. Wenn der Netzerkgraph erstellt ist, wird in einer Schleife ein Vektor mit den Einträgen für die Knoten befüllt. Identifiziert werden die Knoten einerseits über eine Nummer, die auch ihrem Index in diesem Vektor entspricht, andererseits auch über ihre Portnummer. Die Portnummern werden beginnend mit 9900 aufsteigend zugewiesen. Der erste Knoten erhält somit die Nummer 9900, der zweite die Nummer 9901 und so weiter. Ist die Option für die Simulation eines Verbindunsausfalls gesetzt, wird im nächsten Abschnitt festgelegt, welche Verbindung ausfällt. Hierfür wird ein zufälliger Knoten ausgewählt. Von diesem Knoten wird sein erste Nachbar aus der Liste gewählt. Die Verbindung zwischen diesen beiden Knoten wird dann als fehlerhaft gekennzeichnet.

Sind alle Einstellungen gesetzt beginnt das eigentliche Erstellen der Knoten. Dafür werden in einer Schleife die Kommandozeilenparameter der Knoten in einem Vektor gespeichert. Der erste Parameter ist der Name des Programms, in diesem Fall `./node`. Von dieser Zeichenkette wird ein Zeiger auf das erste Zeichen in einem Vektor gespeichert. Die nächsten beiden Parameter sind die Indikatoren für das Logging, welche analog zu denen des Controllers gesetzt werden. Danach folgt der Port des Knoten, welcher mit der Option -p übergeben wird. Nach dem

eigenen Port folgen jene Ports der direkten Nachbarn. Jeder Knoten erhält mit der Option -n die Portnummern seiner Nachbarn. Die letzte Option wird nur gesetzt, wenn zuvor dieser Knoten als fehlerhaft deklariert wurde. In Zeile 22 und 23 des Codebeispiels 5.2 sieht man, dass dem Knoten mit der Option –failure der Port übergeben wird, zu dem ein Verbindungsfehler simuliert wird. Der letze Schritt ist das eigentliche Starten der Knoten. Hierfür wird wie in Zeile 28 des Code-Ausschnitts zu sehen sit ein neuer Prozess erstellt. In diesem Prozess wird mittels execv das Programm node ausgeführt.

```
1  for (size_t i{0}; i < node_cnt; i++) {
2      vector<char*> node_cmd_args;
3      node_cmd_args.push_back((char*)"./node");
4
5      if (use_logging) {
6          node_cmd_args.push_back((char*)"--log");
7          if (log_level_debug) {
8              node_cmd_args.push_back((char*)"--debug");
9          }
10     }
11
12     node_cmd_args.push_back((char*)"-p");
13     node_cmd_args.push_back(&nodes[i].port[0]);
14
15     if (network[i].size() > 0) {
16         node_cmd_args.push_back((char*)"-n");
17         for (size_t j{0}; j < network[i].size(); j++) {
18             node_cmd_args.push_back(&nodes[(network[i][j])].port
       [0]);
19         }
20     }
21     if (nodes[i].failure) {
22         node_cmd_args.push_back((char*)"--failure");
23         node_cmd_args.push_back(&nodes[i].failed_connection[0]);
24     }
25     node_cmd_args.push_back(NULL);
26     nodes[i].cmd_args = node_cmd_args;
27
28     nodes[i].pid = fork();
29     if (nodes[i].pid == -1) {
30         spdlog::error("Creating node process {} failed.", i);
31     } else if (nodes[i].pid> 0) {
32         fmt::print("[{}] Create node process with pid {}.\n",
       format(fg(fmt::color::magenta), "Controller"), nodes[i].pid);
33         spdlog::info("Create node process with pid {}.", nodes[i].
       pid);
34     } else {
35         char** node_argv{node_cmd_args.data()};
36         execv(node_program, &node_argv[0]);
37         perror("execl");
38         exit(EXIT_FAILURE);
39     }
40 }
```

Listing 5.2: Erstellen der Knoten

Sobald die Knoten erstellt sind, sind alle Aufgaben des Controllers erfüllt und

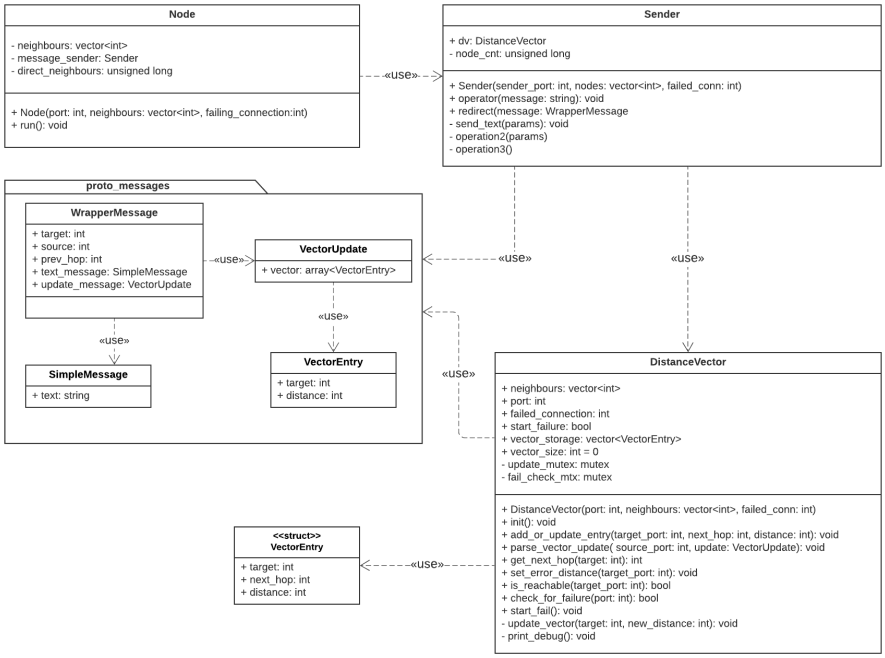

Abbildung 5.1: Klassendiagramm des node-Programms

er wartet nur noch auf einen Keyboard-Interrupt, bei welchem er die Prozesse der Knoten beendet.

5.2 Knoten

Das Programm des Knoten ist jener Teil, der den Algorithmus implementiert. Das Programm ist unter dem Namen node aufrufbar und besteht aus insgesamt drei Klassen und einer Main-Funktion. In dieser werden nur die Parameter der Kommandozeile ausgelesen und gespeichert. Dies geschieht gleich wie im Controller und wird daher nicht weiter erläutert. Der relevante Teil beginnt mit dem Aufruf der Methode run der Node-Klasse. Um einen Überblick über die einzelnen Klassen und deren Beziehungen zu erhalten ist in Abbildung 5.1 ein Klassendiagramm dargestellt.

5.2.1 Node-Klasse

Diese Klasse dient als Hülle für alle weiteren Klassen und beinhaltete somit die gesamte Logik. Die Klasse hat nur zwei Methoden. In der run-Methode wird ein Socket erstellt und in einer Endlosschleife auf neue Verbindungen gewartet.

Ist eine Verbindung dieses Knotens als fehlerhaft gekennzeichnet, wird außerdem ein Thread gestartet, welcher nach einer Verzögerung von 30 Sekunden die Simulation des Fehlers startet. Dabei wird eine Variable gesetzt, welche beim Verbindungsaufbau abgefragt wird. Wird eine Verbindung zum Socket aufgebaut, wird der Socket an einen neu erstellten Thread weitergegeben, welcher die Verbindung bearbeitet. Der Thread führt die zweite Methode der node-Klasse aus, `serve_request`. Hier wird die eingehende Nachricht gelesen und zur Bearbeitung weitergegeben. Ist es eine Textnachricht, wird ihr Inhalt auf der Kommandozeile ausgegeben. Handelt es sich um eine Vektor-Aktualisierung, wird diese von der DistanceVector-Klasse bearbeitet. Die letzte Option ist, dass die Nachricht nur weitergeleitet werden muss. In diesem Fall wird sie an die redirect-Methode der Sender-Klasse übergeben. Im Codebeispiel 5.3 ist die Methode zum Bearbeiten einer Verbindung angegeben. Je nach Art der Nachricht wird anders mit ihr verfahren.

```
asio::streambuf buf;
asio::read_until(sock, buf, '\n');
proto_messages::WrapperMessage message;
istream is{&buf};
message.ParseFromIstream(&is);
if (message.target() == this->message_sender.dv.port) {
    if (message.has_text_message()) {
        fmt::print("[{}] ({}) Received: "
        + message.text_message().text() + "\n"
        , format(fg(fmt::color::cyan)
        , "Node " + to_string(this->message_sender.dv.port))
        , format(fg(fmt::color::pale_green), "TextMessage"));
        spdlog::info("Received message: '{}'"
        , message.text_message().text());
    } else if (message.has_update_message()) {
        this->message_sender.dv.parse_vector_update(
            message.source(), message.update_message());
    }
} else {
    this->message_sender.redirect(message);
}

sock.close();
```

Listing 5.3: Bearbeiten einer Verbindung

5.2.2 Protobuf-Klassen

Dieser Abschnitt befasst sich mit den Nachrichten, welche mit der Protocol-Buffer-Bibliothek erstellt werden. Google Protocoll Buffers ist eine Bibliothek zum Serialisieren und Deserialisieren von Daten. Für die Implementierung des Distanz-Vektor-Algorithmus werden insgesamt vier Nachrichten verwendet. Zu besseren Lesbarkeit sind in den folgenden Sekionen die Datentypen meist nicht angegeben. Ist kein Datentyp angegeben, so handelt es sich um einen 32-Bit Integer.

WrapperMessage

Um das Deserialisieren zu vereinfachen, sind die eigentlichen Nachrichten in eine Wrapper-Nachricht namens WrapperMessage verpackt. Diese enthält neben der eigentlichen Nachricht noch drei Felder für das Ziel, den Herkunftsknoten und den zuletzt besuchten Knoten. In dieser Verpackungs-Klasse sind eine der beiden folgenden Klassen gespeichert.

SimpleMessage

Die erste Nachrichten-Klasse ist SimpleMessage, welche nur ein Feld, eine Zeichenkette enthält. Diese Nachricht dient als Testnachricht und soll die in einem echten Netzwerk versendeten Nutzdaten simulieren.

VectorUpdate

Die Klasse VectorUpdate versendet die Routing-Informationen. Sie enthält eine Liste von Einträgen der Klasse VectorEntry, eine in diese Nachricht eingebettete Nachricht. Jeder dieser Einträge repesentiert einen Eintrag in der Weiterleitungstabelle des Sender-Knotens und hat daher zwei Felder, eines für das Ziel und eines für die Distanz zu diesem.

5.2.3 Sender-Klasse

Die Sender-Klasse ist, wie der Name schon sagt, für das Senden von Nachrichten über das Netzwerk verantwortlich. Sie ist in der Node-Klasse gespeichert. Sie beinhaltet zusätzlich zu den Funktionen zum Senden auch die DistanceVektor-Klasse. Die Klasse wird als Thread ausgeführt, welcher periodisch in zufälligen Zeitabschnitten von 8 bis 12 Sekunden Nachrichten aussendet. Bevor das Aussenden der Nachrichten beginnt, wird die DistanceVector-Klasse mit ihrer Methode init initialisiert.

Nach der ersten Hälfte des Intervalls wird eine Vektor-Aktualisierung an alle Nachbarknoten versendet. Dafür wird die Methode `send_update_to_neighbours` aufgerufen, welche wiederum die Funktion `send_update` für jeden Nachbarknoten aufruft. Dabei wird eine neune Instanz der WrapperMessage angelegt und mit den in der Weiterleitungstabelle vorhandenen Einträgen befüllt. Ist die zweite Hälfte des Intervalls verstrichen, wird eine SimpleMessage an alle bekannten Knoten, nicht nur an alle Nachbarn ausgesendet. Dieser Vorgang wiederholt sich in einer Endlosschleife.

Die Methode redirect zum Weiterleiten einer Nachricht ist ebenfalls Teil dieser Klasse. Ihr wird eine WrapperMessage als Parameter übergeben. In der Weiterleitungstabelle wird nach dem nächsten Knoten gesucht, an den die Nachricht gesendet werden muss, um zum Ziel zu gelangen. Ist der Knoten gefunden, wird eine neue Verbindung geöffnet und die Nachricht versendet.

Vor jedem Versenden einer Nachricht, unabhängig davon, ob dies nur Weiterleiten oder das Versenden einer neuen Nachricht ist, wird überprüft, ob die

zum nächsten Knoten noch besteht. Diese Übrprüfung bezieht sich auf den simulierten Ausfall und ist keine tatsächliche Überprüfung der Erreichbarkeit. Ist die Verbindung als fehlerhaft gekennzeichnet, wird in der Weiterleitungstabelle der Wert für Unendlich, in dieser Implementierung das Maximum eines 16-Bit Integers, eingetragen.

5.2.4 DistanceVector-Klasse

Diese Klasse beinhaltet die eigentliche Implementierung des Algorithmus. Das Herzstück ist ein Vektor des Structs VectorEntry. In diesem wird für jeden bekannten Knoten seine Portnummer, der nächste Knoten, über den er erreicht werden kann und die Distanz zu ihm gespeichert. Dieser Vektor wird von der Mehtode `add_or_update_entry` bearbeitet. Die Methode erhält als Parameter den Ziel-Port, den nächsten Hop und die Distanz. Zuerst wird überprüft, ob es sich bei dem Ziel um den Knoten selbst handelt. In diese Fall wird nichts geändert. Danach wird die Weiterleitungstabelle durchgegangen und nach einem Eintrag für dieses Ziel durchsucht. Wird kein Eintrag gefunden, wird der neu erhaltene mit einer um 1 erhöhten Distanz übernommen. Wurde ein Eintrag gefunden, werden die Distanzen verglichen. Ist die um 1 erhöhte erhaltene Distanz kleiner als die bereits gespeicherte Distanz, wird die neue Distanz übernommen und der Sender der Aktualisierung als nächster Knoten eingetragen. Eine Ausnahme aus diesem Ablauf entsteht, wenn die erhaltene Distanz den Wert für Unendlich hat. In diesem Fall wird der neue Wert übernommen, wenn der bereits gespeicherte nächste Knoten dem Senderknoten der Aktualisierung entspricht. Dadurch wird sichergestellt, dass auch andere Knoten vom Ausfall erfahren.

Ausgelöst wird eine Änderung der Weiterleitungstabelle durch den Erhalt einer Vektor-Aktualisierung von einem benachbarten Knoten. Die erhaltene VectorUpdate-Nachricht wird zusammen mit dem Port des Senders an die Methode `parse_vector_update` übergeben. Diese geht die Liste an Einträgen durch und ruft für jeden Eintrag die oben beschriebene Methode `add_or_update_entry` auf. Das gleiche geschieht auch beim Aufrug der init-Methode. Diese wird einmalig nach dem Start des Knoten aufgerufen und befüllt die Weiterleitungstabelle mit den Einträgen der Nachbarknoten und einem Eintrag für den Knoten selbst.

Abgesehen von den bereits beschriebenen Methoden gibt es noch weitere, von denen vor allem die Methode `get_next_hop` relevant ist. Sie durchsucht die Weiterleitungstabelle nach einem Eintrag für den übergebenen Port und gibt die Portnummer des nächsten Knoten zurück, an den eine Nachricht gesendet werden muss, um das Ziel zu erreichen.

Literatur

Kolousek, Dr. Günter. *Entwurf und Implementierung verteilter Systeme*. 26. Sep. 2010.

asio C++ library. URL: `https://think-async.com/Asio/index.html` (besucht am 08. 04. 2021).

CLI11. URL: `https://github.com/CLIUtils/CLI11` (besucht am 11. 04. 2021).

JSON for Modern C++. URL: `https://github.com/nlohmann/json` (besucht am 11. 04. 2021).

Meson Build System. Meson development team. URL: `https://mesonbuild.com/index.html` (besucht am 08. 04. 2021).

Wikipedia, Routing. URL: `https://de.wikipedia.org/wiki/Routing` (besucht am 11. 04. 2021).

Wikipedia, Routing Information Protocol. URL: `https://de.wikipedia.org/wiki/Routing_Information_Protocol` (besucht am 11. 04. 2021).