Simulation zweier Ampelsteuerung die miteinander kommunizieren

Projektnummer: 19

Valentino Lazarevic, 5CHIF

April 2021

Contents

1 Einführung

Heutzutage sind Ampeln aus der Gesellschaft nicht mehr wegzudenken. Sie regeln den Verkehr in bestimmten Situationen, sorgen für einen sicheren Verkehrsfluss und vermeiden Unfälle. Zu wissen ist, dass an einer Kreuzung alle Ampeln miteinander verbunden sind, sodass sie aufeinander abgestimmt umschalten. Die Ampeln funktionieren computergesteuert durch einen Algorithmus, der über einen Schaltkasten, zu den Ampeln der Kreuzung, durch Kabel miteinander verbunden ist.

1.1 Projekt Aufgabe

Die Aufgabe besteht darin, zwei Ampelsysteme miteinander kommunizieren zu lassen. Dabei kommuniziert die Südstraße der ersten Ampel mit der Nordstraße der zweiten Ampel, außerdem soll eine Lastverteilung der Straßen zustande kommen. Damit wird die Auslastung der einzelnen Straßen gemeint. Zum Beispiel, wenn an der Oststraße mehr als zehn Autos warten, dass die Ampel Grün wird, wird diese bei der nächsten Grünschaltung länger Grün bleiben, damit mehr Autos durchkommen.

1.2 Idee der Implementierung

Die Implementierung der Kommunikation wird über das Lesen und Schreiben auf Sockets realisiert. Sowohl die Südstraße der ersten Ampel, als auch die Nordstraße der zweiten Ampel besitzen solch ein Socket über den diese kommunizieren. Die Kommunikation zwischen den Straßen, welche nur die Nord- und die Südseite der jeweiligen Ampel betrifft, wird über die geteilte Queue erledigt, die sich durch die Ampel unterscheiden lässt. Dabei wird ein Tokensystem berücksichtigt, welches entscheidet welches Auto Element zur welcher Straße gehört:

- t1-South TokenNORTH → Richtung t1-North
- t1-South TokenSOUTH → Richtung t2-North
- t2-North TokenNORTH → Richtung t1-South
- t2-North TokenSOUTH → Richtung t2-South

Wichtig zu wissen ist, dass die Socket Straßen nur Autos von den gegenüberliegenden Straßen kriegen können. Das heißt, dass diese am Anfang des Starts leer sind.

Die Idee zur Lastverteilung ist, dass die Queue Größe abgefragt wird. Diese entscheidet, ob die Zeit bei der nächsten Grünphase vergrößert oder verkleinert wird.

1.3 Meine Problematiken

Bei der Umsetzung der Kommunikation, sowohl bei den Ampelschaltungen sind viele Probleme entstanden. Da die Straßen nicht miteinander verbunden sind, wissen diese nicht welche Autos z.B. im Norden ankommen bzw. vorbei fahren. Dies musste geändert werden, im Kapitel 3.2.3 wird dies auch erklärt. Weiters wurde die Kommunikationsarchitektur mehrmals überdacht und verändert, zuerst sollten die Nordstraße mit der Südstraße beider Ampel verbunden werden, danach sollte die Südstraße mit der Nordstraße der jeweiligen Ampeln kommunizieren. Jedoch reagierten, in dem Fall die Server, Nord- und Südstraße immer nur auf einen Socket und dieser war auf der gleichen Ampel schon belegt. Daher wurde die Implementierung der Kommunikation verändert, wie diese nun ist wird im Kapitel 3.2 besprochen. Da die Implemtierung der Kommunikation in der street.cpp durchgeführt wurde, wusste ich zuerst nicht wie die Ports den Straßen zugeteilt werden. Dies wurde gelöst in dem der Port bzw. der Empfänger Port im Konstruktor der Straße übergeben wurde.

2 Verwendung der Bibliotheken

2.1 Allgemeine Informationen

Die Bibliotheken wurden in der STANDALONE Variante verwendet, damit ist die Header Only Version gemeint. Diese wird einfach über die meson_options.txt hinzugefügt. Die Bibliotheken sind im Referenzverzeichnis verlinkt.

2.2 ASIO

Die ASIO Bibliothek, auch genannt "asynchronous stream input/output Library", wird für die Netzwerk Programmierung in C++ verwendet. Außerdem ist diese Open Source und somit für jeden zugänglich. Die Boost::Asio

Version enthält weitere Features, da diese nicht als Header Only Version kommt. In diesem Projekt wird ASIO zur Kommunikation zwischen der ersten und der zweiten Ampel verwendet. Die Kommunikation wird im Kapitel 3.2 genauer erklärt. [ASI]

2.3 CLI11

Die CLI11 Bibliothek wird zur Verbesserung der Bedienung verwendet, genauere Informationen zur Funktion und Einsetzung von CLI wird im Kapitel 4 besprochen. [anoa]

2.4 RANG

Die Rang Bibliothek dient zur Modifizierung der Ausgabe. Dies wird erreicht indem die Ausgabe in verschiedenen Farben bzw. Arten(**FETT**, *kursiv* oder unterstrichen) gekennzeichnet wird. Im Projekt wird die Ausgabe farblich markiert, dies führt zu einer übersichtlichen Ausgabe.[anob]

2.5 SPDLOG

Die spdLog Bibliothek dient zum Loggen, mit spdLog ist das Loggen in Dateien als auch in die Konsole möglich. Weiters bringt die Bibliothek viele weitere Features. Das Loggen wird in den Straße und der Ampel gemacht.[anoc]

2.6 JSON

Die Nholmann JSON Bibliothek ist eine moderne Lösung für die Verwendung von JSON in C++. Wichtige Aspekte dieser Bibliothek sind die Performance, da das Nholmann/Json z.B. schneller als Vektoren und Maps ist. Aber auch die Nutzung von Speicher ist effizienter gestaltet. Das JSON wird im Projekt beim serialisieren bzw. deserialisieren verwendet, als auch bei der Generierung der Autos in dem diese in eine JSON Datei geschrieben werden. [Loh]

3 Implementierung

3.1 Implementierung des Ampelsystems

Zur Darstellung der Archtitektur habe ich Sourcetrail verwendet. Sourcetrail ist ein kostenloser Open-Source-Quellcode-Explorer, der interaktive Abhängigkeitsgraphen bereitstellt. [Sou]

Als aller Erstes wird die Architektur durch die Dateien dargestellt. In der folgenden Abbildung werden die Dateien aufgelistet:

Figure 1: Auflistung der Dateien im Projekt

3.1.1 Enum

"Mit enum können Aufzählungstypen definiert werden. Dazu gehören Wochentage oder Farben. Den Elementen eines Aufzählungstyps werden Namen zugeordnet, obwohl sie intern natürlich als Zahlen codiert werden."[Wil]

In meinem Programm sind folgende enums definiert:

Listing 1: C++ enums.h

```
1 enum TrafficColor {RED = 1, YELLOW = 2, GREEN = 3};
2 enum Directions {NORTH = 1, SOUTH = 2, WEST = 3, EAST = 4};
```

3.1.2 Ampel - (trafficlight)

Aufgabe

Die **Ampel** ist in meinem Programm dafür zuständig, die richtigen Ausgaben zu tätigen. Um der Straße mitzueilen, welche Straßeneinmündung auf Grün, Gelb oder Rot gestellt ist. Weiters speichert die Klasse die Queue der Nord und Südstraße.

Klassenstruktur

Listing 2: C++ trafficlight.h - Klassenstruktur

```
1  class TrafficLight
2  {
3    private:
4      TrafficColor colorNorthSouth;    //enum
5      TrafficColor colorWestEast;      //enum
6      std::mutex l_mutex;
7      std::string name;
8      int north_south_timer;
9      int east_west_timer;
10     int counter;              //Entscheidet welche Seiten beginnen
11   public:
12     std::queue<Car> *NorthSouthcarQueue = new std::queue<Car>();
13
14     Trafficlight(std::string name, int counter);
15     TrafficColor getNorthSouthColor();
16     TrafficColor getWestEastColor();
17     std::string getName();
18     void setNorth_south_timer(int timer);
19     void setEast_west_timer(int timer);
```

```
20      void startTrafficLight();
21  };
```

Wie wir sehen können, besitzt die Klasse drei wichtige Funktionen. Die ersten beiden Funktionen - getNorthSouthColor und getWestEastColor liefern den aktuellen Status der Ampel an der gewünschten Straße. Die Funktion start-TrafficLight startet und verwaltet die Ausgaben der Ampel. Ein Beispiel:

[TrafficLight 1] North and South Light is now GREEN
[TrafficLight 1] West and East Light is now RED

Die Funktion getName wird zur Verbesserung der Ausgabe verwendet, damit jetzt zwei Ampel auftretten. Weiters gibt es die setter Funktion für die Lastverteilung, diese setzen die Zeit der Ampelausgabe.

Sehr wichtig ist die Queue auf die, die verbundenen Straßen der eigenen Ampel zugreifen. Diese wird in die Ampel Klasse ausgelagert, da die Ampel unterschieden werden kann zwischen den beiden Ampeln.

Abhängigkeitsdiagramm

An der folgenden Grafik kann die Abhängigkeit der Ampel Klasse beobachtet werden. Außerdem werden alle Funktionen und Variablen angezeigt:

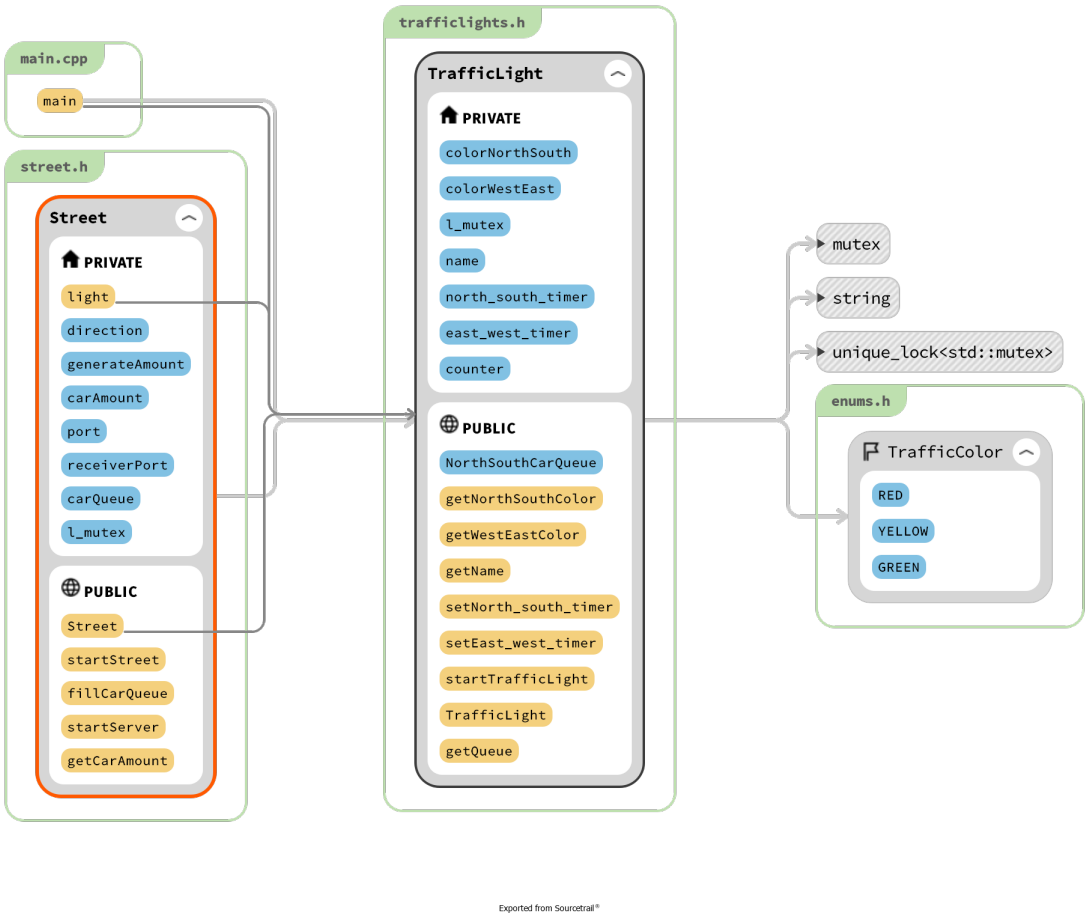

Figure 2: Abhängigkeitsdiagramm der Ampel Klasse

3.1.3 Straße - (street)

Aufgabe

Die **Straße** ist für die Auffüllung der carQueue zuständig. Weiters "schickt" die Klasse die Autos über die Straße. Dies wird mit einer Ausgabe markiert. Ein Beispiel:

[CAR] AUDI HL-B7GF9 drives away from NORTH

Außerdem erfolgt die Kommunikation in der Street.cpp, wie die Lastverteilung. Diese werden jedoch im Kapitel 3.2 und 3.3 erklärt. Somit speichert sich die Klasse den Port und den Empfänger Port.

Klassenstruktur

Listing 3: C++ street.h - Klassenstruktur

```
1  class Street
2  {
3  private:
4      TrafficLight* light;
5      Directions direction;
6      int generateAmount;
7      int carAmount;
8      unsigned short port;
9      unsigend short receiverPort;
10     std::queue<Car>* carQueue = new std::queue<Car>();
11     std::mutex l_mutex;
12 public:
13     Street(int generateAmount, TrafficLight* light,
14             Directions direction, int carAmount, unsigned short port,
15             unsigned short receiverPort);
16     void startStreet();
17     void fillCarQueue(int amount);
18     void startServer();
19     int getCarAmount();
20 };
```

Die Struktur enthält die wichtige *carQueue*, welche die Autos beinhaltet. Diese zählt jedoch nur für die Ost und Westseiten bzw. der Seite die nicht mit der anderen Ampel kommuniziert. Weiters enthält die Klasse die Menge - (Zahl) an nach-generierten Autos, sowie die Richtung in die, die Autos fahren. Die beiden Funktionen sind am Namen selbst erklärend, startStreet ist für die Ausgabe der fahrenden Autos zuständig bzw. löscht diese nach der Überfahrt aus der Queue heraus. Dazu kommt, dass die Funktion, die Autos auf die Sockets schreibt. Die Funktion StartServer wird ausgeführt um den Server zu starten, der auf die Verbindung wartet. Genaueres im Kapitel 3.2. Die Methode fillCarQueue füllt die Queue mit generierten Autos.

Abhängigkeitsdiagramm

An der folgenden Grafik kann die Abhängigkeit der Straße gesehen werden. Außerdem werden wieder alle Funktionen und Variablen angezeigt:

Figure 3: Abhängigkeitsdiagramm der Straße

3.1.4 Auto - (car)

Aufgabe

Die **Auto** Klasse stellt das generierte Auto zur Verfügung. Diese beinhaltet das Kennzeichen, die Automarke und die Zeit welche das Auto benötigt, um über die Straße zu fahren. Außerdem speichert die Klasse die Richtung, in welche das Auto fährt.

Klassenstruktur

Listing 4: C++ car.h - Klassenstruktur

```
1  class Car
2  {
3    private:
4      std::string name;
5      std::string licensePlate;
6      int speed;
7      int direction
8    public:
9      static nholmann::json generateCar(int amount);
10
11     Car(std::string name, std::string licensePlate, int speed,
12         int direction = 0) {
13         this->name = name;
14         this->licensePlate = licensePlate;
15         this->speed = speed;
16         this->direction = direction;
17     }
18
19     std::string getLicensePlate();
20     std::string getName();
21     int getSpeed();
22     int getDirection();
23     void setDirection(int direction);
24 };
```

Die Auto-Klassenstruktur generiert das *Auto* und ihre *Eigenschaften*. Diese werden in eine JSON-Datei ausgelegt. Dabei erstellt die generatCar Methode

das Auto, dieses beinhaltet das zufällig generierte Kennzeichen, sowie die Zeit die es benötigt die Straße zu überqueren. Außerdem wird die Richtung - wo das Auto ankommt, in dieser Klasse gespeichert. Für den Zugriff wurde eine Get und Set-Mehtode definiert.

Abhängigkeitsdiagramm

An der folgenden Grafik wird die Abhängigkeit der Auto Klasse dargestellt. Außerdem werden wieder alle Funktionen und Variablen angezeigt:

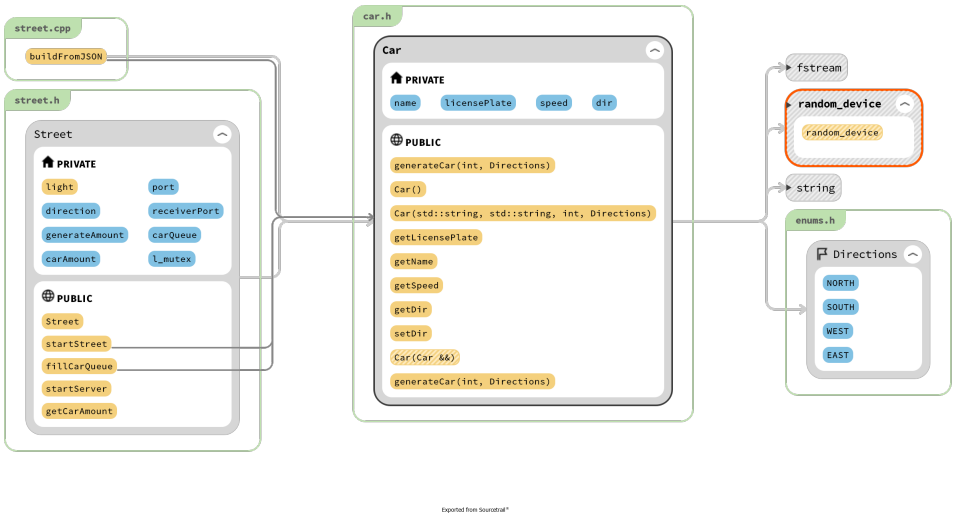

Figure 4: Abhängigkeitsdiagramm der Auto Straße

3.2 Implementierung der Kommunikation

Die Kommunikation wurde zwischen der Südstraße, der ersten Ampel, und der Nordstraße, der zweiten Ampel, implementiert. Generell wurde die Kommunikation in der street.cpp geschrieben. Im Konstruktor der Klasse Straße wird der Port und der Empfänger Port übergeben. Jedoch ist dies nur für die Nord und Südstraße wichtig. Bei den anderen Straßen sind diese mit 0 definiert. Als aller erstes wird durch die Überprüfung der Ports entschieden welche Straße der Server ist. Im Projekt ist es die Straße mit dem Port 45000. Diese ruft die Funktion startServer() auf. In der folgenden Struktur wird diese Funktion erklärt:

Listing 5: C++ street.cpp - Server Verbindung

```
1  //Server Socket Definition
2  asio::io_context ioCtx;
3  asio::ip::tcp::socket t1SouthReadSocket(ioCtx);
4
5  void Street::startServer() {
6      try {
7          asio::io_context ioCtx;
8          asio::error_code ec;
9
10         string host = "127.0.0.1";
11
12         asio::ip::address ip_address = asio::ip::make_address(host);
13
14         asio::ip::tcp::endpoint ep(ip_address, port);
15         asio::ip::tcp::acceptor acceptor(ioCtx);
16         acceptor.open(ep.protocol());
17         acceptor.bind(ep);
18         acceptor.listen();
19         acceptor.accept(t1SouthReadSocket);
20
21         spdlog::get("console")->info(
22                     "[T1 South] CONNECTION ESTABLISHED"
23                                         );
24     } catch (exception &e) {
25         spdlog::get("err_logger")->info("ERROR");
26         cout << e.what() << endl;
27     }
28 }
```

Im Allgemeinen wird in der Funktion beschrieben, dass der Socket gesetzt wird und darauf wartet, dass sich wer verbindet. Nach dem das accept richtig ausgeführt wird und sich ein anderer socket verbindet, wird in die Konsole CONNECTION ESTABLISHED geschrieben.

Der Client Socket wird ebenfalls durch den Port entschieden. Danach führt dieser Socket eine Verbindung auf den Empfänger Port durch. Der Empfänger Port ist in diesem Fall der Server Port. Im folgenden Code kann man sehen

wie dies durchgeführt wird:

Listing 6: C++ street.cpp - Client Verbindung

```
asio::io_context ioCtx;
asio::error_code ec;
asio::ip::tcp::socket t2NorthWriteSocket(ioCtx);
asio::ip::address ip_address =
            asio::ip::make_address("127.0.0.1", ec);

// establish connection
if (port == 47501) {
    //connection
    t2NorthWriteSocket.connect(
            asio::ip::tcp::endpoint(ip_address, receiverPort), ec
                                );
    spdlog::get("console")->info("
            [T2 North] Client connected to server [T1 South]"
                                );
}
```

Auch hier muss zuerst der Socket und dessen notwendigen Teile definiert werden. Danach wird der Port überprüft. Sollte dieser richtig sein, wird wie vorher besprochen die Verbindung durchgeführt. Nach erfolgreicher Verbindung, wird ebenfalls eine Ausgabe getätigt.

3.2.1 Lesen

Das Lesen wurde in die _read Funktion verlagert. Wie diese Funktion aufgebaut ist, sieht man am folgenden Code:

Listing 7: C++ street.cpp -Lese Funktion

```
string _read(asio::ip::tcp::socket &socket) {
    asio::streambuf buf;
    asio::read_until(socket, buf, "\n");
    asio::streambuf::const_buffers_type bufs = buf.data();
    string data(asio::buffers_begin(bufs),
                asio::buffers_begin(bufs) + buf.size());

    return data;
```

```
9 }
```

Die Funktion bekommt einen Socket der gerade lesen möchte. In der Funktion selbst wird gelesen bis ein "\n" ankommt. Nachdem wird der stream buffer in einen string konvertiert und zurückgegeben. Diese Funktion wird in der startStreet() aufgerufen. Da sowohl der Client als auch der Server lesen wollen, wird die Funktion für beide aufegrufen.

3.2.2 Schreiben

Das Schreiben wird getätigt wenn das Auto auf die andere Ampel möchte. Dabei wird die write Funktion aufgerufen. Diese wird im späteren Verlauf von der _read Funktion gelesen. Wie das Aufrufen der Ampel funktioniert, wird folgend beschrieben, außerdem wir durch das JSON serialisiert:

Listing 8: C++ street.cpp - Client Verbindung

```
1 string carJSON = getJSON(NorthSouthNextCar).dump() + "\n";
2
3 asio::write(t2NorthWriteSocket, asio::buffer(carJSON));
4 //oder
5 string carJSON = getJSON(NorthSouthNextCar).dump() + "\n";
6
7 asio::write(t1SouthWriteSocket, asio::buffer(carJSON));
```

3.2.3 Kommunikation der Straßen in der selben Ampel

Da Grundsätzlich nicht implementiert wurde, dass die Nordseite weiß welche Autos ankommen, musste dies anders gelöst werden. Dies wurde durch eine Queue in der trafficlights.h gelöst. Da die Straßen wissen an welcher Ampel der beiden sie sich befinden, können nur die Straßen der eigenen Ampel auf die Queue zugreifen. Im Projekt schreibt die Straße die sich nicht in der Mitte befindet in die Queue miteels push Funktion. Somit kann die Straße die sich in der Mitte befindet diese Queue abfragen und das Auto auf die andere Ampel mittels ASIO schicken. Danach wird das Element mittels einer pop Funktion aus der Queue gelöscht .

3.3 Implementierung der Lastverteilung

Die Idee der Lastverteilung ist, dass die Größe der Queue abgefragt wird. Je nach Größe der Queue wird entschieden, ob Zeit dazu gerechnet oder entzogen wird. Dies wird in einer Funktion der street.cpp eingerichtet. Im folgenden Code sieht man wie dies realisiert wurde:

Listing 9: C++ street.cpp - Lastverteilungsfunktion

```
1  int getTrafficTimer(int queueSize) {
2      if (queueSize > 50) {
3          return 15000;
4      } if (queueSize > 25) {
5          return 10000;
6      } if (queueSize > 10) {
7          return 7500;
8      } else {
9          return 5000;
10     }
11 }
```

Wie man sehen kann beinhaltet die Funktion einfache If-Abfragen die entscheiden wie lang gewartet werden soll. Es wird ein Integer zurückgegeben, dass in ms umgewandelt wird. Die Funktion wird innerhalb der startStreet() aufgerufen. Je Nach Straßenrichtung wird mittels einer Setter Methode der neue Wert gesetzt. Jedoch ist zu beachten, dass bei der Nord und Südstraße zwei Queues betroffen sind. Siehe Beispiel:

Listing 10: C++ street.cpp - Lastverteilungsfunktion

```
1  this->light->setEast_west_timer(getTrafficTimer(carQueue->size()));
2  //ode
3
4  if (carQueue->Size() < this->light->NorthSouthCarQueue->size()) {
5      this->light->setNorth_south_timer(getTrafficTimer(
6                                  this->light->NorthSouthCarQueue->size()
7                                                                  ));
8  } else {
9      this->light->setNorth_south_timer(
10                             getTrafficTimer(carQueue->size())
11                                                  );
```

```
12 }
```

Die Setter Methoden "setNorth_south_timer()" und "setEast_west_timer()" werden nachher bei der Grunschaltung aufgerufen. Dieser lässt den Thread die neue Zeit schlafen. Dadurch haben die Straßen länger Zeit die Autos über die Ampel zu schicken.

3.4 Schlusswort zur Implementierung

Die Implementierung dieses Projekt gestaltete sich weitaus komplexer als ich es dachte. Ich musste sehr viel Zeit damit verbringen, die ASIO Bibliothek zu verstehen. Außerdem wurde das Projekt mehrmals umgeändert. Am Anfang habe ich mehrere Ideen ausprobiert, jedoch sind die meisten gescheitert. Nach langen Nächten, habe ich die Kommunikation siehe 3.2 gemeistert als auch die Kommunikation unter der eigenen Ampel siehe Kapitel 3.2.3. Außerdem waren die Probleme, die entstanden sind, viel komplexer und schwieriger zu lösen, als diese des ersten Projekts. Es hat oft mehrere Stunden gedauert eine Lösung zu finden. Die Implementierung der Lastverteilung war verständlicher und leichter zu implementieren.

4 Bedienung

Zur besseren Bedienung wurde die Header-only Datei CLI11.hpp verwendet. Diese implementiert ein userfreundliches Kommandozeileninterface, über welches man das Programm konfigurieren kann. Alle eingegebenen Werte werden vor dem Start der Simulation überprüft und falls notwendig werden auch Fehler geworfen.

4.1 –help, -h

Mittels ./trafficlight-sim -h oder –help, werden die nötigen Parameter bzw. Funktionen angezeigt. In meinem Beispiel würde die Ausgabe folgendermaßen Aussehen:

Listing 11: Ausgabe - –help, -h

```
1  TrafficLight-Simulation
2  Usage: ./trafficlight-sim [OPTIONS] cars respawntime car-amount
3
4  Positionals:
5    cars INT:NUMBER REQUIRED
6    How many cars after each respawn time respawns
7
8    respawntime INT:NUMBER REQUIRED
9    The time interval in which new cars spawn
10
11   car-amount INT:NUMBER REQUIRED
12   The amount of cars which should be generated
13
14 Options:
15   -h,--help
16   Print this help message and exit
```

4.2 cars

Verpflichtend ist die Eingabe von **cars**. Damit ist die Menge an Autos gemeint, die nach einer gewissen Zeit wieder an die Straße kommen. Dieser Parameter muss ein Integer sein.

4.3 respawntime

Die **respawntime** ist ebenfalls verpflichtend. Diese regelt die Zeit, in der Autos generiert werden. Zum Beispiel: ./trafficligtht-sim 1 10 1 würde jede 10 sek ein Auto generieren lassen. Dieser Parameter muss ein Integer sein.

4.4 car-amount

Der **car-amount** ist ebenfalls verpflichtend. Dieser bestimmt die Menge der Autos bei der ersten Generation. Dieser Parameter muss ein Integer sein.

4.5 Beispiel

Die Ausführung wrid durch die Eingabe von ./trafficlight-sim getätigt. Zum Beispiel ./trafficlight 2 10 4. Dabei werden die Parameter gesetzt. In diesem Falle: cars = 2, respawntime = 10 und car-amount = 4. Will man mehr Interaktionen, kann man die Prameter höher setzen.

5 Schlusswort

Im Allgemeinen war die Umsetzung des Projekts eine sehr anspruchsvolle Arbeit, die mir sehr viel beigebracht hat. Meine Kenntnisse haben sich zum Thema Netzwerktechnik verbessert. Außerdem stärkte das Projekt meine Umgangsweise mit C++ und dessen Features. Ich habe insbesonders die folgenden Foliensätze durchgearbeitet um mehr über das gesamte Thema zu verstehen:

- 24_serialization
- 25_communication
- 26_tcpip_programming1
- 27_tcpip_programming2
- 28_tcpip_programming3
- 29_server_programming

Mittlerweile finde ich das Thema sehr spannend. Außerdem möchte Ich dem Internet danken, dass viele Dinge gut erklärt sind die z.B. nicht in der Dokumentation der Bibliotheken vorkommen.

References

[anoa] anonym. *CLI11: Command line parser for C++11*. Besucht am 11.04.2021. URL: https://github.com/CLIUtils/CLI11.

[anob] anonym. *Colors for your Terminal*. Besucht am 11.04.2021. URL: https://github.com/agauniyal/rang.

[anoc] anonym. *Fast C++ logging library*. Besucht am 11.04.2021. URL: https://github.com/gabime/spdlog.

[ASI] ASIO. *Asio C++ Library*. Besucht am 11.04.2021. URL: https://think-async.com/Asio/.

[Loh] Niels Lohmann. *JSON for Modern C++*. Besucht am 11.04.2021. URL: https://github.com/nlohmann/json.

[Sou] Sourcetrail. *Sourcetrail*. Besucht am 11.04.2021. URL: https://www.sourcetrail.com/.

[Wil] Arnold Willemer. *Aufzählungstyp enum*. Besucht am 09.02.2021. URL: willemer.de/informatik(cpp/enum.htm.