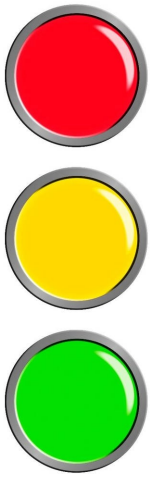
Simulation zweier Ampelsteuerung die miteinander kommunizieren

Projektnummer: 19

Valentino Lazarevic, 5CHIF

April 2021



Contents

1 Einführung

Heutzutage sind Ampeln aus der Gessellschaft nicht mehr wegzudenken. Sie regeln den Verkehr in bestimmten Situationen und sorgen für einen sicheren Verkehrsfluss und vermeiden Unfälle. Zu Wissen ist das 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 Ampelsystem miteinander zu 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 Aulastung der einzelnen Straßen gemeint. Zum Beispiel wenn die Oststraße mehr als 10 Autos hat, die warten das 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, was nur die Nord und die Südseite jeweiliger Ampel betrifft, wird über die geteilte Queue Ressource erledigt. Dabei wird ein Token System berücksichtigt, dass entscheidet welches Auto Element zur welcher Straße gehört:

- 2 = Süd
- 1 = Nord
- 0 = Gegenüberlige Straße der anderen Ampel

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

1.3 Meine Problematiken

Bei der Umsetzung der Kommunikation, sowohl bei der 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 (ref) wird dies auch erklärt. Weiters wurde die Kommunikationsarchitektur mehrmals überdacht und verändert. Zuerst sollten die Nordstraße mit der Südstraße beider Ampel verbunden sein. Danach sollte die Südstraße mit der Nordstraße der jeweiligen Ampeln kommunizieren. Jedoch reagieren 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 (ref) besprochen. Da die Implemtierung der Kommunikation (ref) 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 (ref) genauer erklärt.

2.3 CLI11

Die CLI11 Bibliothek wird zur Verbesserung der Bedienung verwendet. Genauere Information zur Funktion und Einsetzung von CLI wird im Kapitel (ref Kapitel Bedienung) besprochen.

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.

2.5 SPDLOG

Die spdLog Bibliothek dient wie der Name schon sagt zum Loggen. Mit spdLog ist das Loggen in Dateien als auch in die Konsole möglich. Weiters bringt die Bibliothe viele weiter Features. Das Loggen wird explizit im Kapitel (ref) nohcmals erklärt und beschrieben.

2.6 JSON

Die Nholmann JSON Bibliothek ist eine moderne Lösung für die Verwendung von JSON in C++. Wichtige Aspekte dieser Bibliothek ist 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. Genauere Erklärung folgt im Kapitel (ref).

3 Implementierung des Ampelsystems

3.1 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};
```

3.2 Ampel - (trafficlight)

Aufgabe

Die **Ampel** ist in meinem Programm zuständig, die richtigen Ausgaben zu tätigen. Um der Straße mitzueilen, welche Straßeneinmündung auf Grün oder Rot bzw. auf Gelb 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 startTrafficLight 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, das 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.

3.3 Straße - (street)

Aufgabe

Die **Straße** ist für die Auffüllung der Auto-Queue 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 ref() und ref() 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*, die die Autos beinhaltet, diese zählt jedoch nur für die Ost und Westseiten der Ampel. Weiters enthält die Klasse die Menge - (Zahl) an nach-generierten Autos. Die Richtung in die die Autos fahren. Wie bei den Aufgaben schon erkärt. 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 das 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 (ref). Die Methode fillCarQueue füllt die Queue mit generierten Autos.

3.4 Auto - (car)

Aufgabe

Die **Auto** Klasse stellt das generierte Auto zur Verfügung. Diese beinhaltet das Kennzeichen, die Automarke und die Zeit wie lang das Auto benötigt um über die Straße zu fahren. Außerdem speichert die Klasse die Richtung, dass 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. Die Methode generateCar erstellt das Auto, dieses beinhaltet das zufällig generierten Kennzeichen. Zur der genauen Erklärung kommen wir im Kapitel 7.2. Außerdem wird die Direction - wohin das Auto fährt, in dieser Klasse gespeichert. Für den Zugriff wurde ein Get und Set-Mehtode definiert.

4 Implementierung der Kommunikation

5 Implementierung der Lastverteilung

6 Bedienung

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

6.1 –help, -h

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

Listing 5: Ausgabe - –help, -h

```
1  TrafficLight-Simulation
2  Usage: ./project [OPTIONS] cars respawntime
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 Options:
12   -h,--help
13   Print this help message and exit
```

6.2 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.

6.3 respawntime

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

7 Zustatzfunktion

7.1 Logger

Der **Logger** ist für das loggen bestimmter Funktionen zuständlich. Diese Informationen schreibt die Funktion in eine externe Datei hinein. Der Dateiname besteht aus dem log_ und den aktuellen Tag, an den der Logger fungiert hat. Ein Beispiel: log_2021-02-10.txt. Sollte an dem Tag der Logger bereits geschrieben haben, wird diese Datei erweitert. Um solche Logs zu unterscheiden steht in der Datei die aktuelle Uhrzeit des Logs. Diese Datei wird im Log Ordner abgespeichert. Am folgenden Beispiel können Sie sehen, wie der Logger aufgebaut ist:

Die Methode **getCurrentTime** stellt jeweils einen String zur Verfügung der entweder den aktuellen Tag oder die aktuelle Uhrzeit zurück gibt.

Die Methode **logger** erstellt die Datei mit der Verwendung der vorherigen Methode und schreibt den angegeben Parameter in diese Datei hinein.

7.2 Auto - Generierung

Die Auto Generierung erfolgt in der Car Klasse siehe: 3.4.

Besonders dabei ist die Generierung des **Kenzeichen**, da erstens alle Bezirksabkürzungen des Bundeslandes Niederösterreich zur Verfügung stehen. Weiters wird das Kennzeichen mittels einer Funktion die zufällige Zeichenketten zurückgibt erstellt. Im folgenden Code Stück sehen Sie wie dies funktioniert:

Die Ermittlung der **Marke** erfolgt durch einen zufälligen Zugriff auf einen Vector der mit Automarken befüllt ist.

Die Ermittlung der **Zeit**, dass das Auto für die Überquerung benätigt wird von einer Zufälligen Zeit bestimmt. Diese liegt zwischen 1 und 2.5 Sekunden.

References

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