Group XX
24 February 2016

Traffic Lights
Obligatory Hand-In

Introduction:

These two applications are a simple Java based traffic light server and client where you can run a server, have multiple traffic lights connected to it and assign them to certain positions to make them synchronise properly.

Our developer group consists of three people:

- s198755 Sondre Husevold
- s198761 Magnus Tønsager
- s198760 Magnus Knalstad

Features:

The complete feature list of the applications are as follows:

- Full graphical user interface for both server and client.
- Clients can be both traffic lights and walking signs.
- Logging mechanic that allows you to see what happens in real time.
- Light frequency sliders to change how fast each light should switch on the client.
- Synchronisation so that all clients are acting simultaneously.
- Easily managed clients with a map showing their position.
- Assigning said traffic lights or walking signs to positions on the crossroads map.

- Fully encrypted communication between server and client through AES.
- Sparsely sent packets where the traffic lights themselves are controlling themselves except when receiving changes from the server
- Supports TCP connections through any port you want
- Full documentation available

Workflow:

Our group's workflow was as follows:

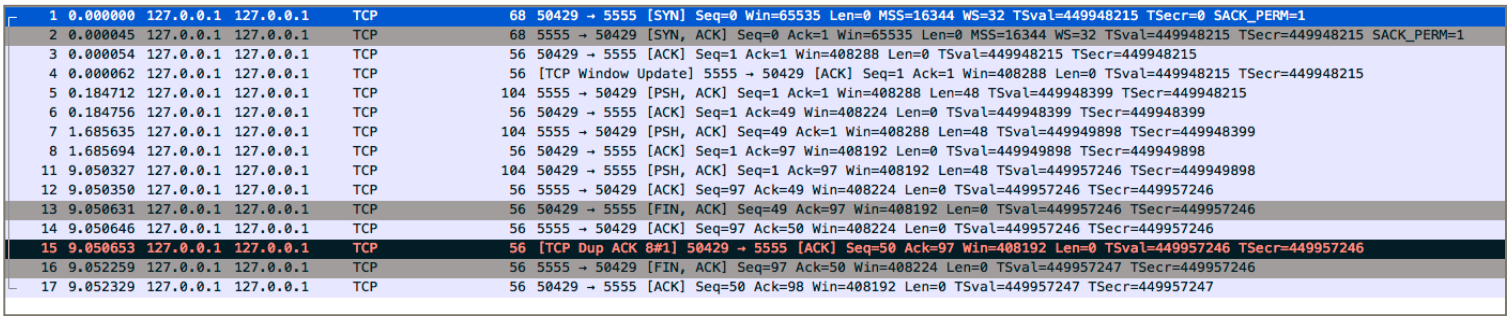
- s198761 was assigned to client connections, synchronisation and scheduling of light switches
- s198755 was assigned to cryptography, crossroads map, walking signs, GUI designer and blackbox tester.
- s198760 was assigned to the actual GUI creation, logging mechanics, JavaFX methodic and GUI exception handling.

All of us were involved with the questions 2 and 3 as we did that together.

We feel like the project was well balanced and that everyone got to do a meaningful part in the project. We are also working on our bachelor project simultaneously with this hand-in which means several members were busy with planning and development of that project. With that in mind, we feel each individual's time was evenly spent on each of the projects.

Question 2:

Here is a screenshot of the packets being sent once a client connects and disconnects from the server:



1	0.000000	127.0.0.1	127.0.0.1	TCP	68	50429 → 5555 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=32 TSval=449948215 TSecr=0 SACK_PERM=1
2	0.000045	127.0.0.1	127.0.0.1	TCP	68	5555 → 50429 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=32 TSval=449948215 TSecr=449948215 SACK_PERM=1
3	0.000054	127.0.0.1	127.0.0.1	TCP	56	50429 → 5555 [ACK] Seq=1 Ack=1 Win=408288 Len=0 TSval=449948215 TSecr=449948215
4	0.000062	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 5555 → 50429 [ACK] Seq=1 Ack=1 Win=408288 Len=0 TSval=449948215 TSecr=449948215
5	0.184712	127.0.0.1	127.0.0.1	TCP	104	5555 → 50429 [PSH, ACK] Seq=1 Ack=1 Win=408288 Len=48 TSval=449948399 TSecr=449948215
6	0.184756	127.0.0.1	127.0.0.1	TCP	56	50429 → 5555 [ACK] Seq=1 Ack=49 Win=408224 Len=0 TSval=449948399 TSecr=449948399
7	1.685635	127.0.0.1	127.0.0.1	TCP	104	5555 → 50429 [PSH, ACK] Seq=49 Ack=1 Win=408288 Len=48 TSval=449949898 TSecr=449948399
8	1.685694	127.0.0.1	127.0.0.1	TCP	56	50429 → 5555 [ACK] Seq=1 Ack=97 Win=408192 Len=0 TSval=449949898 TSecr=449949898
11	9.050327	127.0.0.1	127.0.0.1	TCP	104	50429 → 5555 [PSH, ACK] Seq=1 Ack=97 Win=408192 Len=48 TSval=449957246 TSecr=449949898
12	9.050350	127.0.0.1	127.0.0.1	TCP	56	5555 → 50429 [ACK] Seq=97 Ack=49 Win=408224 Len=0 TSval=449957246 TSecr=449957246
13	9.050631	127.0.0.1	127.0.0.1	TCP	56	50429 → 5555 [FIN, ACK] Seq=49 Ack=97 Win=408192 Len=0 TSval=449957246 TSecr=449957246
14	9.050646	127.0.0.1	127.0.0.1	TCP	56	5555 → 50429 [ACK] Seq=97 Ack=50 Win=408224 Len=0 TSval=449957246 TSecr=449957246
15	9.050653	127.0.0.1	127.0.0.1	TCP	56	[TCP Dup ACK 8#1] 50429 → 5555 [ACK] Seq=50 Ack=97 Win=408192 Len=0 TSval=449957246 TSecr=449957246
16	9.052259	127.0.0.1	127.0.0.1	TCP	56	5555 → 50429 [FIN, ACK] Seq=97 Ack=50 Win=408224 Len=0 TSval=449957247 TSecr=449957246
17	9.052329	127.0.0.1	127.0.0.1	TCP	56	50429 → 5555 [ACK] Seq=50 Ack=98 Win=408192 Len=0 TSval=449957247 TSecr=449957247

The first four packets (1-3) are designated to handshaking and connecting the server and the client.

- SYN (packet #1) means it tries to synchronise and thus initiate a connection.
- Packet #2 is a SYN and ACK response from the server, where it synchronises and acknowledges the received data.
- Packet #3 is an ACK response from the client acknowledging that it has received packet #2. A 3-way handshake is now complete.
- Packet #4 is a Window Update which means the sender has increased the TCP receive buffer space.

In packet #5, we actually send 48 bytes of data. This data when captured looks like this:

```
Data (48 bytes)
    Data: 5748704f4731496f62794d434d744558594b716e47770a58...
    [Length: 48]
```

5748704f4731496f62794d434d744558594b716e47770a5863736f5a7a495954586e5962616c495235387546773d3d0a

These are the bytes that are sent as a command by the cryptography class. When converted from hexadecimal numbers to a string it will give us this information:

WHpOG1IobyMCMtEXYKqnGw

XcsoZzIYTXnYbaIIR58uFw==

The first line of information is the unpadded base64 encoded 16-byte IV used to encrypt/decrypt the text to make the text randomised. This means the encrypted text will never be the same, even though the same string is encrypted twice. By decoding the base64 string and converting it to hex using xxd or some other program it'll give us the 16-byte IV.

The second part which begins with 'Xcso' is the actual encrypted text in Base64 form. When decoded by using Base64, we'll get the encrypted text. However since the text is encrypted it is complete gibberish unless decrypted using the 16-byte IV and the secret 128-bit key using AES.

Thus without the key, this information is completely useless, however when decrypted with the proper key we get the following string: **sync0**

sync0 is a command the server sends when it tells the client to synchronise to the first lane, where "0" is the red colour designated as a constant integer in the code. Thus the first command that isn't handshake and typical connection setups is packet #5 and contains a synchronisation command from the server.

This pushing of encrypted information is continued through packets 5, 7 and 11. When decrypted, packet #5 is the "**sync0**" command, #7 is "**restart**" which means the server told the client to start scheduling lights again, and 11 is "**kill**" for telling the client it is preparing to shut down and that the client needs to do its disconnection rituals.

After “kill”, at packet #13, there is a packet sent from the client called FIN, ACK which means it wants to close the connection. At #14, the server acknowledges this. And on #15 it disconnects completely.

Question 3.

We believe that using UDP for such an application would be disastrous as the commands can come directly from any computer due to UDP not being directly connected to the sender/receiver and is thus oblivious to where the packet came from unless you program the application itself to specifically do so and even then you can’t be one hundred percent sure it actually came from the sender.

Not only that, but if the packets are sent too fast, the packets will not come in orderly fashion, which means the commands might come unordered and thus do something before it was supposed to causing crashes or other anomalies.

Thirdly, some packets may be lost during sending with UDP and unlike TCP which automatically resends it if it didn’t get a response. You’d need to program the application itself to resend the packets if they weren’t properly received.

So in conclusion: It is possible to use UDP in this application, but it is not recommended as packet control would need to be coded manually, there’s the order problem and if packets are lost you need to code the application to handle it otherwise the lights would be unsynchronised.