Traffic Light Sample

Sockets, Threads, Protocols

This isn't as complicated as it looks.

Contents

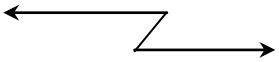
Contents

The Scenario

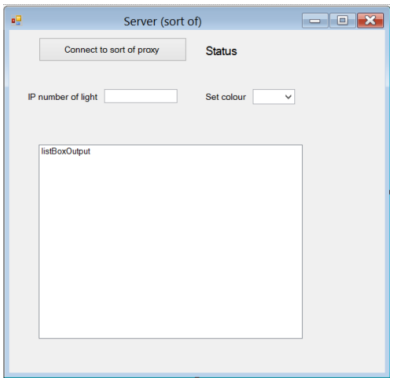
…is slightly contrived, I admit, but basically a remote traffic light sends instructions to a server (it isn't a server in the usual sense of the word) over a computer network. The server controls the state of the lights, over a network of course.

Traffic light "client"

Local Area Network (LAN)



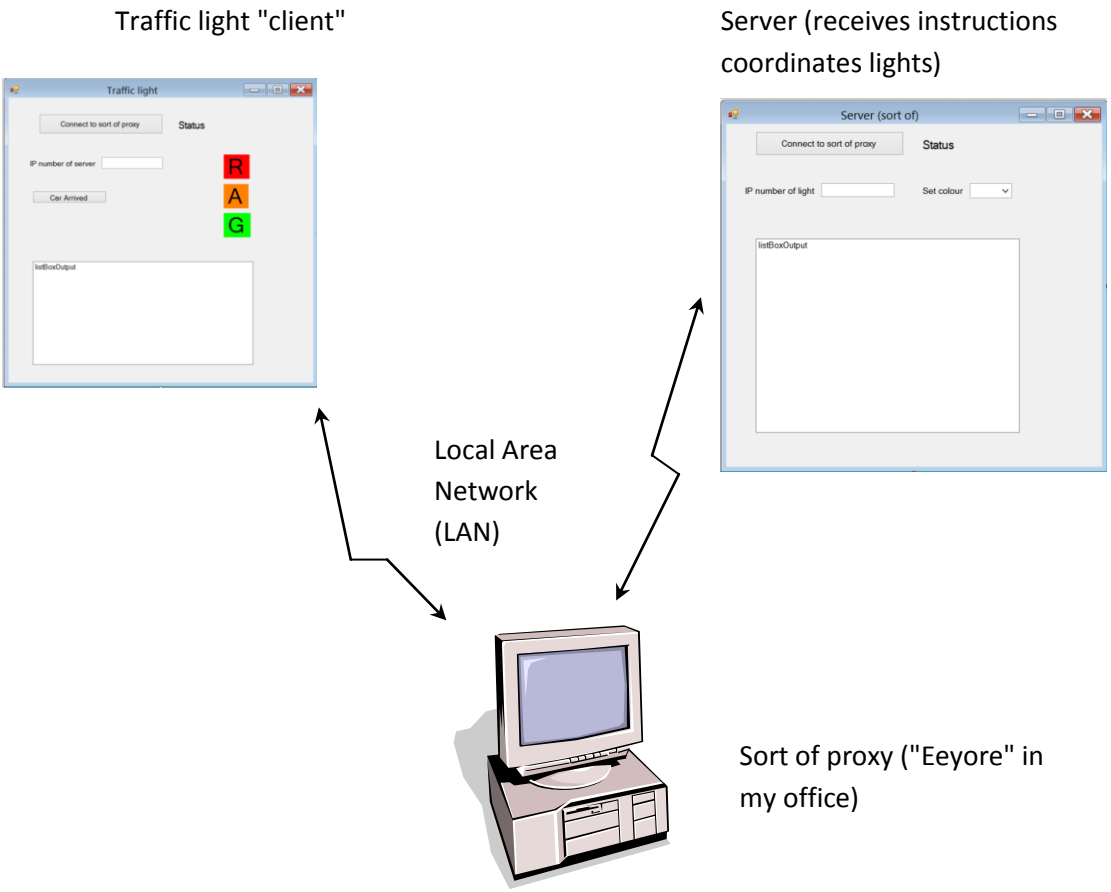
Server (receives instructions coordinates lights)



The only thing that makes the "server" look like a server is that the server waits to be connected to and the client initiates the network connection.

However we have a problem with the above model in all of the computing labs. By default for the lab computers have the local firewall enabled, and the firewall prevents any application other than known privileged to accept incoming network connections ☹

So (being nerds, for this module anyway) we think our way round the problem with a rather more complicated solution. We only make outgoing connections, and the communication between two computers is managed by a third “sort of proxy” running on "Eeyore (so named because it is a bit of a work-donkey) which is a computer than resides in my office. "Eeyore" can accept incoming connections.



Both traffic light and "server" the networking component “Socket” or "ClientSocket", which is something you will find in any operating system. It allows is to create a socket, a reliable connection between two computers.

You will find three programs in the folder "Nigel Barlow (OS) -> Sample Code -> C# -> Traffic Lights"

You don't need to touch (or even download) "sort of proxy"

Sockets

A socket can reliably transmit and receive bytes. That's all it does. It has no knowledge of what these bytes mean or why you wanted to send them in the first place. Transporting bytes over a network transport layer stuff (you probably know about that already and you certainly will know if you are a networking student).

Here we are talking application layer stuff. That's the whole point of this network layered model. As a developer, we don't know (or particularly care) what the underlying network is. It could be a wired LAN (as in our labs), WiFi, home broadband, a mobile network (2G / 3G / 4G) or other network technologies I am forgetting. Programmatically everything above the socket appears to us to be the same.

At this (application) level we generally need to consider three things

1. An interface which defines how the two applications will communicate.
2. A mechanism to serialise (or marshal) data by flattening a complicated data structure into a stream of bytes. And hopefully reconstruct the data structure as it originally was once it has transmitted and received.
3. A mechanism to translate data formats on one computer into those on another.

We don't need to consider (3) here as we have C# programs talking to C# programs. What (3) means is that there is no guarantee that, say, an int, float, double in C# looks anything like the equivalents in, say, Java, PHP, Python... But we don't need to worry about that here.

The first thing we need consider is a protocol (what the data means) for the sort of proxy.

The Sort of Proxy

(you won't have to change this one unless you want to)

This starts with `program.cs` makes a listening socket. This socket just blocks (sits and waits) until a client connects to it. Writing programs on a desktop computer that block is not usually considered good practice, as blocking it this way causes the whole application to lock up. There isn't much of an application here.

The proxy also maintains a list of connections that have been made.

If a client connects, than a new `ThreadConnection` (in the separate file `ThreadConnection.cs`). Each connection is managed by a new instance of `ThreadConnection`. This is object oriented jargon, we can make as many instances of `ThreadConnection` as we like. A `Thread` runs in its parent's memory space, so the parent can easily talk to it, but it treated by the low level scheduler as a separate process and runs asynchronously (in its own time). The idea here is that the sort of proxy can manage many connections, each of which runs asynchronously.

Digression

Ask yourself how scaleable this (each connection handled in a separate Thread) is – how many connections can we really support? 200? (we have 200 students on the module). 2,000? 20,000 connections?

We will soon know what a `semaphore` is. The sort of proxy creates a list of the many connections it is supporting, stored in a C# `List`. The C# `List` is not thread safe. If several `Thread`s try to modify a C# `List` concurrently the list is likely to get scrambled. I use a `semaphore` (as it is something we have met) to stop the scrambling; there are many other solutions.

`ThreadConnection` loops forever reading data, if any, from its socket. If data is present, the program iterates through a list of all connections, looking for any that match the IP number as represented by the first 4 bytes of the packet that was received. If there is a connection that matches that IP, the packet is transmitted to that connection.

Or put another way

The Protocol

The client programs open a connection to the sort of proxy. They then transmit a packet of bytes. The packet is *always* 200 bytes long, an arbitrary number. The first 4 bytes represent the (old IP4) IP number of the client that the packet is to be sent to. OK, OK, IP4 numbers aren't that simple, but they are here.

That's it. The remaining 196 bytes are for you to fill.

The things you will have to modify are

The Traffic Light

There is a simple sample program that gets you started.

When you run it, start by pressing the button "Connect to sort of proxy". If this works, the status box will go green.

You then enter the IP number of the computer running the server. All you can then currently do is lush a button indicating that a car has arrived.

As with the sort of proxy, a network connection is handled in a separate Thread `ThreadConnection` (in the separate file `ThreadConnection.cs`). I won't say don't touch it, but you probably don't need to.

The Thread `ThreadConnection` spends most of its time sitting and waiting (blocking is the jargon) on the line

```
inStream.Read(packet, 0, bufferSize);
```

That's why the connection is handled in a separate Thread. If it wasn't then the form would become frozen and unresponsive.

The nasty part (which I nave done for you) is in the C# Threading rules, which say that you can only touch the UI components from the thread that made them, which is the form's thread here. To get over this we use a `SynchronizationContext`. This allows us to post messages to ourselves. How sad is that? Well, the good part is that the message we then receive comes back on the form's Thread, and we can now use that call-back to modify the form's UI components. A call-back is just like a button click, literally "don't call me, I'll call you".

I think the traffic light will only ever need one `ThreadConnection`.

In short

When a traffic light receives a message, the method

```
public void MessageReceived(Object message)
```

is executed. With the sample code I have given you, this is always a String containing three things, separated by spaces, all separated by spaces.

1. An IP number representing the computer that sent the string
2. the word "sent"
3. The message

The protocol, such as it is, is that the server may send one of three strings, "Red", "Green", "Amber"

These strings are interpreted by the method

```
private void ChangeLights(String command)
```

And you will see that the program code is very lazy. It doesn't try to solit the string into its component parts (which would be equivalent to step (2), the data un-marshalling), is just looks to see if the message contains the strings "Red", "Green", "Amber".

Where to start?

So why can't we turn a light off? We need to extend out protocol a little. How about starting by extending the protocol so that the server can transmit "Red on", "Red off" etc.....

So let's takes a look at the server.

The "Server"

Maybe controller would be a better description.

There is a simple sample program that gets you started.

When you run it, start by pressing the button "Connect to sort of proxy". If this works, the status box will go green.

You then enter the IP number of the computer running the traffic light. All you can then currently do is to use a drop-down box (sends the message "Red", "Green", "Amber" to the client) to turn a light on. Never off again.

The server looks very similar to the client; it has a single Thread `ThreadConnection` which spends most of its time sitting and waiting for something to be received from the network. It uses the same solution to the C# Threading rules as the client, and has the same call-back

```
public void MessageReceived(Object message)
```

which always contains a String containing three things, separated by spaces, all separated by spaces.

1. An IP number representing the computer that sent the string
2. the word "sent"
3. The message

At present, it only ever displays the message that was received in a list box.

Where to start?

For the minute, use a server with one `ThreadConnection` only. You will only be able to control one set of traffic lights. When 10 cars have arrived (you will need to count them), change the lights to green and reset the cars counter.

If you want to be really ambitious (for the brave only; you don't have to do this) create a server (or controller) that maintains lists of traffic lights and which can control several sets of lights in some notional city. In this case you will need to maintain a list of connections, a little like my sort of proxy.

On a Home Network

Please feel free to run "sort of proxy" on your home network of you wish. You will need to adjust the network addresses in the client and the controller that they connect to.

Otherwise, to see "Eeyore" from home you may well need to make a VPN (Virtual Private Network) as "Eeyore" is not visible across the corporate firewall.

Exactly how you do this depends on the operating system you are running, the place to look is probably "create to a corporate network"

Once you have found the right setting, create a VPN to vpn.plymouth.ac.uk and use your usual Plymouth name and password when asked. If asked for a domain. that is "UoPNET" (shouldn't it be PUNET?).