



## **Developer Study Guide**

# **Bluetooth® Internet Gateways**

**Bluetooth Mesh Networks**

Release : 2.0.1

Document Version: 2.0.0

Last updated : 24th June 2021

# Contents

<b>1. REVISION HISTORY .....</b>	<b>5</b>
<b>2. INTRODUCTION.....</b>	<b>6</b>
<b>3. PREREQUISITES .....</b>	<b>6</b>
3.1 Equipment .....	6
3.2 Knowledge and Experience .....	7
<b>4. PROJECT - BUILDING A BLUETOOTH® INTERNET GATEWAY FOR MESH NETWORKS.....</b>	<b>8</b>
4.1 Mesh Gateway Requirements .....	8
4.2 Refining the Physical Architecture for Bluetooth Mesh Devices .....	9
4.3 Selected Physical Architecture .....	10
Gateway Applications .....	11
Gateway API and Supporting Protocols .....	11
Application Protocol Handlers .....	12
Adapters.....	12
Bluetooth API .....	12
Bluetooth LE Stack .....	12
Operating System.....	12
Hardware Platform .....	13
4.4 BlueZ and Bluetooth Mesh - Special Concepts .....	13
4.4.1 Applications, nodes and node composition.....	13
4.4.2 D-Bus Basics .....	14
4.4.3 Important Interfaces .....	14
4.4.4 The BlueZ Mesh API .....	15
4.4.5 Attaching to the network.....	15
4.4.6 Foundation models and network management .....	15
4.5. Gateway Development .....	16
4.5.1 Platform Set-up .....	16
4.5.2 Apache Web Server.....	22
4.5.3 websocketd .....	27

4.5.4 Mesh Network.....	29
4.5.5 Coding the HTTP adapter and Bluetooth mesh API components.....	37
4.5.6 Gateway status.....	66
<b>5. PROJECT - BUILDING GATEWAY APPLICATIONS .....</b>	<b>66</b>
5.1 Introduction.....	66
5.2 Task - Complete the mesh network control application.....	67
5.2.1 Overview .....	67
5.2.2 Preparation .....	67
5.2.3 Task - Completing the mesh network control application.....	72
5.3 Gateway application status.....	76
<b>6. SECURITY .....</b>	<b>77</b>
6.1 Warning!.....	77
6.2 Further Security Measures for the Bluetooth Mesh Gateway.....	77
6.3 Encrypted web sockets communication.....	78
6.4 Authentication.....	78
6.5 Reverse proxying and the mesh gateway.....	79
6.5.1 Web server configuration changes.....	79
6.5.2 Adjusting the gateway application.....	80
6.6 Task - Securing the Bluetooth Devices.....	80
6.6.1 Task - The Bluetooth firewall .....	80
<b>7. SCALABILITY .....</b>	<b>86</b>
7.1 Additional scalability issues relating to gateways for Bluetooth mesh.....	86
7.1.1 BlueZ and concurrency limits with mesh application nodes .....	86
7.1.1.1 BlueZ tokens and concurrency.....	86
7.1.1.2 Exploring BlueZ tokens and concurrency.....	87
7.1.1.3 Increasing mesh node application concurrency .....	87
<b>8. TROUBLESHOOTING.....</b>	<b>91</b>
8.1 Issue: It's not working.....	91
8.2 Issue: mesh-cfgclient is waiting to connect to bluetooth-meshd.....	91
8.3 Issue: dbus-org.bluez.mesh.service not found.....	91
8.4 Issue: Unexpected Warning: config file /home/pi/.config/meshcfg/config_db.json not found.....	92
8.5 Issue: Node is not responding to messages.....	92
8.6 Issue: Websocket problems.....	92
8.7 Issue: Inconsistent use of tabs and spaces in Python code.....	93

<b>9. CLOSE .....</b>	<b>94</b>
<b>APPENDIX A - RESOURCES .....</b>	<b>95</b>
A1 Example Provisioning and Configuration Summary	95

## 1. Revision History

Version	Date	Author	Changes
1.0.0	18th November 2020	Martin Woolley Bluetooth SIG	Initial version
1.0.1	6 <sup>th</sup> January 2021	Martin Woolley Bluetooth SIG	Added <i>enabled</i> configuration property to the Bluetooth firewall.
2.0.0	24 <sup>th</sup> June 2021	Martin Woolley Bluetooth SIG	<b>Release</b> Added new module covering gateways for Bluetooth mesh networks. Modularised the study guide to accommodate the two main cases of LE Peripherals and mesh networks. Updated to be based on Python 3 rather than Python 2. <b>Document</b> New in this release

## 2. Introduction

This module continues on from module 02 First Steps and covers the completion of the hands-on project to develop a Bluetooth internet gateway for Bluetooth mesh networks.

## 3. Prerequisites

### 3.1 Equipment

To complete the practical work in this guide, you will need some equipment, and the following table lists the hardware and platform software that was used by the author in creating this resource. You are free to use compatible alternatives, but the information in the guide relates to the items listed here.

Item	Version(s)	Comments
Raspberry Pi	Zero W or 4 Model B or 400 <i>Any Raspberry Pi which supports Bluetooth</i>	For use as a <i>Provisioner</i> and <i>Configuration Manager</i> . These terms are explained in section 4. Any Raspberry Pi will work but the Zero W was found to be a little slow at times and timeouts during configuration operations sometimes occurred. A type 4 Model B is preferred.
Raspberry Pi	400 or 4 Model B	For use as the Bluetooth internet gateway. A Raspberry Pi Zero W will work but response times from the gateway will be slow. The author used a Raspberry Pi 400 as the gateway device.
Linux, Raspbian distribution	Kernel version 4.19.97	Created using the Raspberry Pi Imager tool.
BlueZ Bluetooth stack	5.50	This is the version that was included in Raspbian, installed by the Raspberry Pi Imager when this study guide was created.
Python 2	2.7.16	This is the version that was included in Raspbian, installed by the Raspberry Pi Imager when this study guide was created.
1 or more Nordic Thingy devices		To act as a node or nodes in the mesh network. Any device or combination of devices which implement or can be programmed to implement the Bluetooth mesh <i>generic on off server model</i> , the <i>light HSL server model</i> and the <i>sensor server model</i> will suffice.
Nordic command line tools for erasing and flashing to Nordic Thingy devices	Latest	Available from <a href="https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download">https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download</a>
Segger J-Link Debugger	Any	This hardware device allows you to flash code to the Nordic Thingy. See

		<a href="https://www.segger.com/products/debug-probes/j-link/">https://www.segger.com/products/debug-probes/j-link/</a>
Segger RTT Viewer tool	Latest	<a href="https://www.segger.com/products/debug-probes/j-link/tools/rtt-viewer/">https://www.segger.com/products/debug-probes/j-link/tools/rtt-viewer/</a>

Regarding the Nordic Thingy and its use as a Bluetooth mesh node, we do not cover how to develop software for these devices in this study guide. A binary for the Nordic Thingy which implements enough of the required Bluetooth mesh models to support the project use cases is provided and steps to install will be described. Source code is provided and is based on the [Zephyr RTOS](#). It should be possible to adapt this code for use with other boards. Please note that the implementation is incomplete, is not suitable for production use and would not pass qualification. It is provided for and fit for purpose for completing the mesh gateway implementation exercise only.

Other software will be installed or created in the practical exercises themselves and where appropriate, version information will be provided at that point.

### 3.2 Knowledge and Experience

To get the best out of this resource, you should already understand the fundamental concepts relating to Bluetooth mesh networking including terms such as *node*, *element*, *model*, *state*, *message* and “*publish and subscribe*”. If any of these terms are new to you then you are advised to read the *Basic Theory* module in the Bluetooth Mesh Developer Study Guide, which you can download from <https://www.bluetooth.com/bluetooth-resources/bluetooth-mesh-developer-study-guide/>.

You should also be comfortable issuing basic commands from a Linux shell and understand Linux fundamentals such as users and permissions. There are numerous Linux tutorials for beginners available on the internet if you need to acquire this knowledge first.

## 4. Project - Building a Bluetooth® Internet Gateway for Mesh Networks

In this section, we'll continue with the work initiated in section 7 of module 02 First Steps. In module 02 we'd got as far as considering requirements, had defined a logical architecture and a tentative physical architecture. We'd noted though that when examined at a more detailed level, we'd need the physical architecture to vary depending on whether we were building a gateway for LE Peripheral devices or Bluetooth mesh nodes.

### 4.1 Mesh Gateway Requirements

The Nordic Thingy device that is to be used in this project as a mesh node has a coloured LED and contains a number of sensors. We want our gateway to allow clients to switch LEDs on and off and to set their colour. We also want nodes to be able to publish temperature readings and for the gateway to send this data to those clients that have indicated that they it. When the gateway supports these use cases, we'll be able to build client applications like the one depicted in Figure 1.

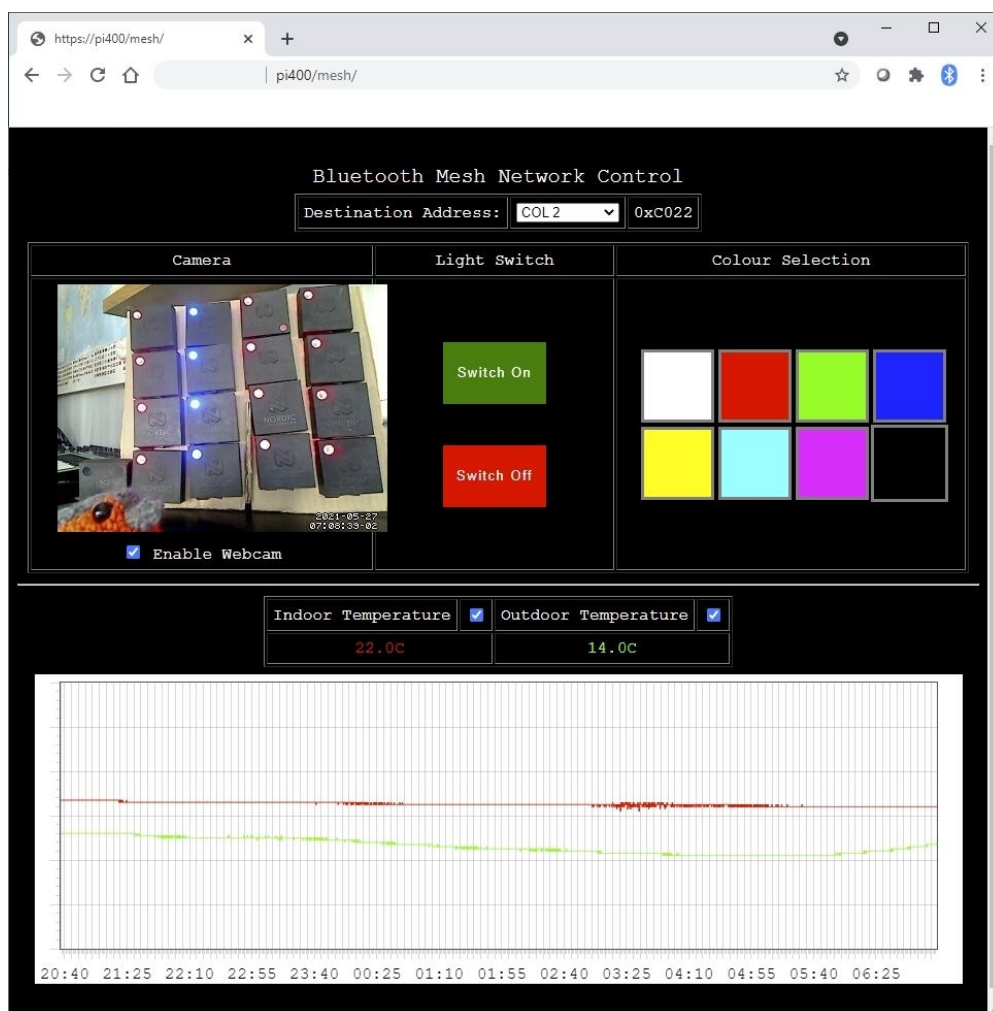


Figure 1 - Mesh Network Controller Application

We need to add to our requirements therefore and specify the types of Bluetooth mesh node that our gateway must allow clients to send and receive messages to and from. Nodes acquire their behaviours by implementing *models* and to support the on/off control, colour control and sensor



data reporting requirements this means that our test nodes must implement the *generic on off server model*, the *light HSL server model* and the *sensor server model*.

To support these types of node and their models and associated use cases, our gateway must implement the corresponding client models, specifically the *generic on off client model*, the *light HSL client model* and the *sensor client model*.

## 4.2 Refining the Physical Architecture for Bluetooth Mesh Devices

The physical architecture defined in module 02 is repeated here in Figure 2.

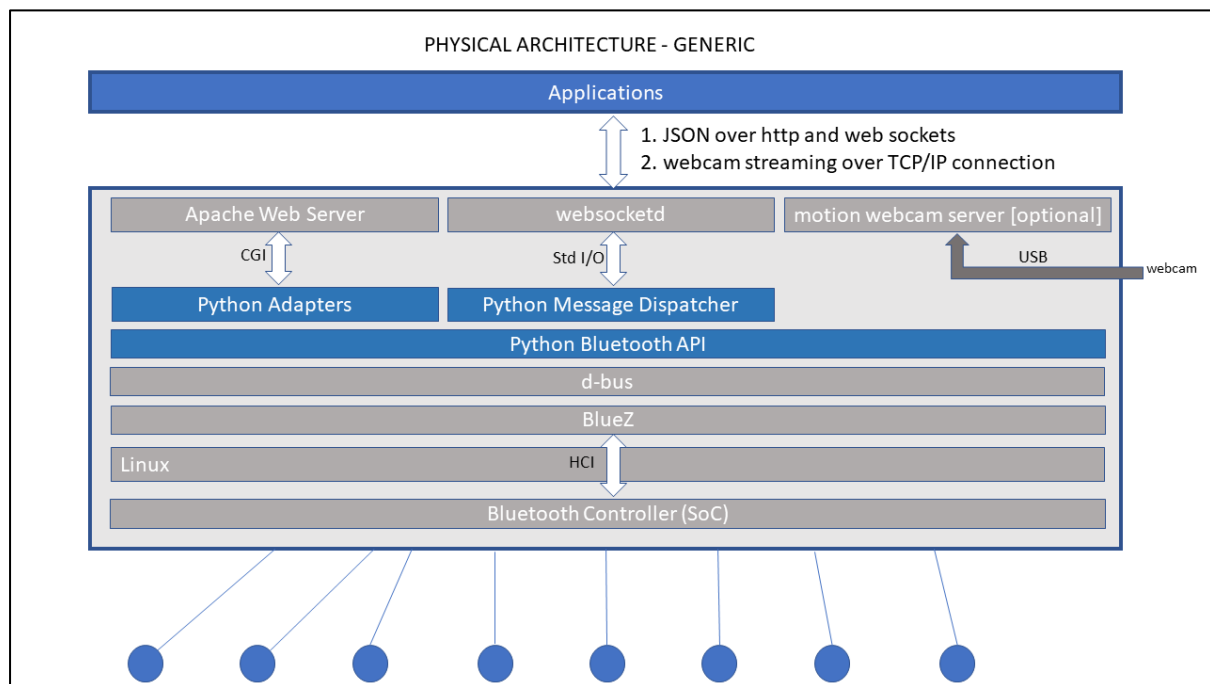


Figure 2 - Generic Physical Architecture for a Bluetooth Internet Gateway

BlueZ provides an API for use with Bluetooth mesh devices which involves a Linux system service called *bluetooth-meshd*. For LE Peripherals a different service called *bluetoothd* is used. The *bluetoothd* daemon and the *bluetooth-meshd* daemon may not be run at the same time on the same Linux computer. One of the reasons for this is that they each serve to queue and serialise requests from applications to and from the (usually) single Bluetooth controller that the device has. If both daemons were running at the same time, it would not be possible for that serialised use of the Bluetooth controller to be achieved.

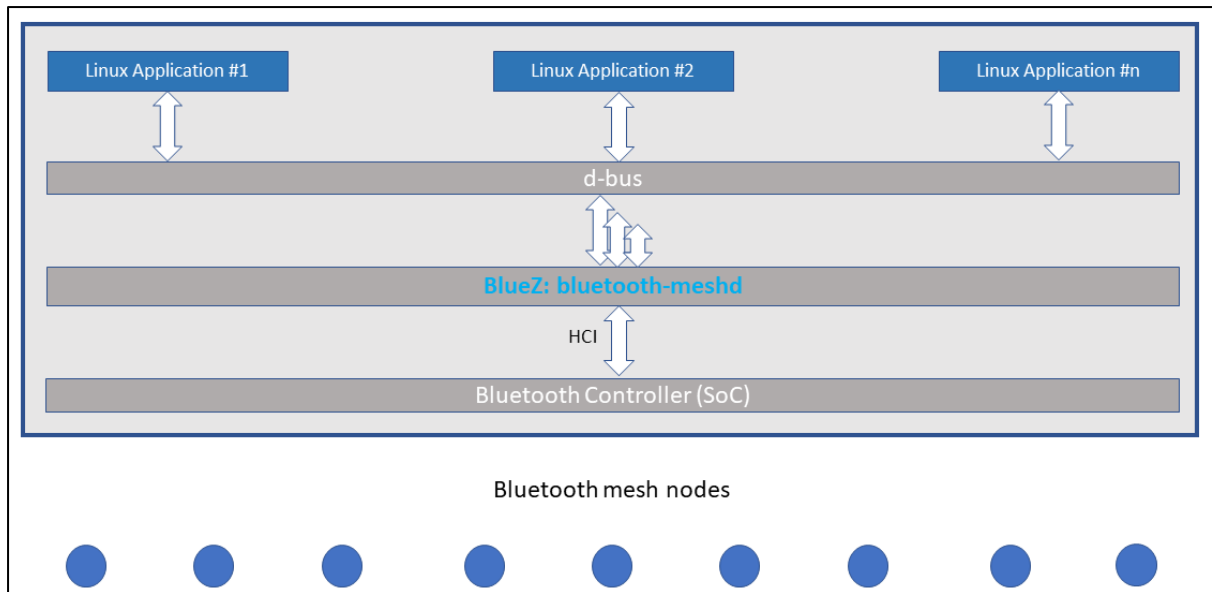


Figure 3 - bluetooth-meshd and the serialisation of HCI commands and events

Consequently, a Bluetooth internet gateway that is based on Linux may only service Bluetooth LE GAP/GATT devices or Bluetooth mesh devices but not both at the same time.

The API provided by BlueZ for mesh is very different to the one provided for use with Bluetooth LE GAP/GATT devices too and so API code for one is very different to code for the other.

### 4.3 Selected Physical Architecture

The physical architecture for a Bluetooth internet gateway which supports the types of mesh device described in section 4.1 is shown in Figure 4 where you can see that the BlueZ service that is required is bluetooth-meshd and the role of the adapter, dispatcher and Bluetooth API Python scripts has been specified more clearly. The acronym I2MN means *Internet To Mesh Network* and the acronym MN2I stands for *Mesh Network To Internet*. These terms will be used to refer to the two directions of flow of message through the gateway.

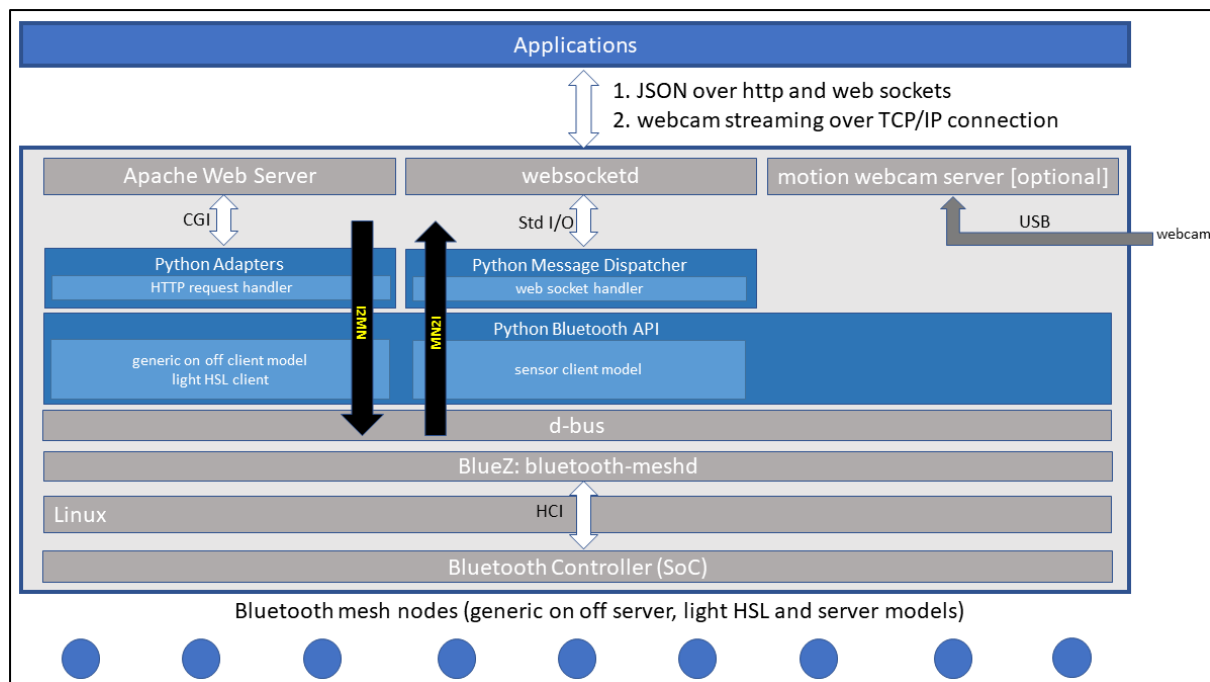


Figure 4 - Physical Architecture of a Bluetooth internet gateway supporting mesh nodes

Study Figure 4 and the explanatory text for each component below. Note that blue rectangles are components we will develop or complete. Others are ready-made components that we shall install, configure and integrate. By selecting largely off the shelf components, we've saved ourselves a lot of work.

### Gateway Applications

Gateway applications fall outside of the scope of our gateway architecture. They can be written in any language, provided the gateway API can be used and the gateway's TCP/IP application protocols are supported.

### Gateway API and Supporting Protocols

The gateway shall support HTTP and web sockets for API communication.

The API shall use the HTTP GET for making requests which do not change mesh model states in nodes with an HTTP query string used to pass associated parameters.

The API shall use the HTTP PUT for making requests which do change mesh model states in nodes with JSON objects in the HTTP request body used to encode associated parameters.

Responses will be delivered as JSON objects in all cases.

In the case of mesh status messages web sockets shall be used since this use case involves node-initiated communication. Clients must be able to subscribe to particular destination addresses so that they only receive relevant messages over the web socket connection and this shall be achieved by writing an API JSON control object to the web socket. Status messages with destination addresses to which a client has subscribed will be delivered to the client application using the same web socket connection, also expressed as a JSON object.

Streaming USB webcam video data over a TCP/IP connection may be supported.

## Application Protocol Handlers

This layer of the architecture consists of two mandatory components and one optional component. Given HTTP and web sockets have been chosen as the TCP/IP interfaces to our gateway, we need one handler for each.

The [Apache](#) web server is an open source, free of charge, mature, reliable and easy to use HTTP server daemon. It has an extensive collection of plugins called modules, with which its functionality can be extended, and custom code to be executed in response to HTTP requests can be written in a multitude of languages. For these reasons, it has been selected as the component that will act as an HTTP handler for our gateway.

Choosing the Apache HTTP server has implications for other aspects of the physical architecture, including the platform the gateway will run on. Apache runs on many platforms, another of its advantages. But if you had envisaged implementing your gateway on a constrained device like a microcontroller, it will not be suitable. It's perfect for this education resource, however.

Web sockets require a server to service connections and so we'll be using a web socket server implementation called [websocketd](#). It's simple, lightweight and perfect for our needs.

We'll support plugging a USB webcam into our gateway so we can point it at a device or two, just because we can, which is often a good enough reason to do something! To that end, our architecture will include the open source webcam server software, [motion](#).

## Adapters

Adapters shall be written in the popular programming language [Python](#). The Apache web server will invoke a particular python script, according to the URL in HTTP requests and by writing to standard output, the python scripts will be able to return a JSON object back over the connection to the requesting HTTP client.

There are many ways to integrate custom code with web servers. The simplest, supported by most web servers, is called the Common Gateway Interface (CGI). It's not designed for high performance or efficiency. There are alternative approaches which fare better in those respects. But neither of these issues is a priority for us. The expectation is that a small number of concurrent users will make a small number of requests at a time.

## Bluetooth API

An API which provides makes it easy for Python scripts to send and receive Bluetooth mesh messages has been partially developed. It will need to be finished during this project.

## Bluetooth LE Stack

We'll be using Linux and therefore will use its standard BlueZ stack which is included in Linux distributions and specifically, per section 4.2, we'll run the *bluetooth-meshd* daemon. The hardware platform must have a Bluetooth LE adapter for BlueZ to work.

## Operating System

Our gateway will run on the Linux operating system.

## Hardware Platform

You can run the gateway on any computer which supports Linux and has a Bluetooth adapter. We used a Raspberry Pi 400 and we'll be referring to this environment in the development stage.

### 4.4 BlueZ and Bluetooth Mesh - Special Concepts

When using BlueZ to create Bluetooth mesh solutions, there are some special concepts to understand that you will not find in the standard Bluetooth specifications.

#### 4.4.1 Applications, nodes and node composition

The Bluetooth mesh profile specification explains the concept of node composition. In simple terms, a node consists of one or more elements, each of which has a 16-bit unicast address. Each element implements one or more models and each model contains items of state and produces and consumes messages of various types. Messages act upon state. Figure 5 shows this.

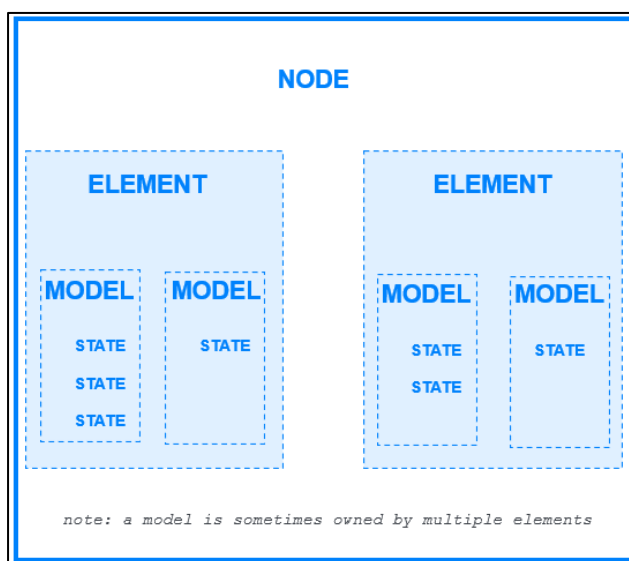


Figure 5 - Node Composition

Usually a physical device acts as a single node in a mesh network and is transformed into a node from an *unprovisioned device* by *provisioning*. But the mesh profile specification does not actually stipulate that there must be a 1-1 correspondence between a node and the physical device which acts as its host. And on Linux with BlueZ providing the mesh stack, this is very much not the case.

On Linux, a node is a software entity whose top compositional level is called an Application (capitalised because Application is often implemented as class). A physical Linux computer can host multiple of these node Applications each of which is composed of Elements, Models and sub-classes of Model per the usual compositional structure. In fact it is more accurate to talk about application **instances** because the same application code can be run in the appropriate mode and provisioned multiple times and this allows multiple instances to be run concurrently and be regarded as multiple separate nodes hosted by the same physical device. Each instance of the node application has a unique identity which is known to bluetooth-meshd and each element in each instance of this application has a unique unicast address in the network because each instance, despite being the same code which is run on the same computer, was independently provisioned.

From now on the term *node application* will be used to refer to one set of source code which implements particular elements and models or one or more runtime instances of an application that implements and has been provisioned as a mesh node. Where the distinction between the source and one or more runtime instances is important, it will be made clear.

This is a flexible arrangement which allows more powerful computers as opposed to simpler microcontrollers to be fully exploited.

#### 4.4.2 D-Bus Basics

It's not intended that this resource will attempt to explain D-Bus programming. You'll pick up what you need to know to work with Bluetooth mesh using BlueZ but if you want to know more, the specification is the key document for this information:

<https://dbus.freedesktop.org/doc/dbus-specification.html>

A very short summary of key points that are relevant to this study guide are:

1. Applications register objects with the D-Bus daemon using a string path value. This is called the *object path* and acts as an identifier for the path. Some system object paths are defined as well as those you invent for use in your application.
2. Objects implement interfaces. Some are used by D-Bus itself and some by the bluetooth-meshd daemon. The absence of a required interface will cause runtime errors.
3. D-Bus applications often need an event loop to ensure callbacks are serviced. This is how asynchronous events are handled. If an event loop is not running, callbacks will not be received. Your code will not exit while the event loop is running so when finished handling a request, whatever the outcome it is usual to stop and exit the event loop so that the application can exit. It is usual to use an event loop provided by the Gnome system library Glib like this:

```
global mainloop
mainloop = GLib.MainLoop()
```

#### 4.4.3 Important Interfaces

Communication between a node application and bluetooth-meshd via D-Bus is bidirectional and to work a node application must implement certain mandatory interfaces. Optional interfaces are also defined and are used only in certain use cases like provisioning. Some validation takes place and if an object (identified by the registered D-Bus object path) does not implement a required interface, errors such as "org.bluez.mesh.Error.Failed: Failed to create node from application" may occur.

A summary of important interfaces used in this project appears in Table 1. The code provided in the implementation/solutions/mesh directory and in bluez/test/test-mesh illustrates their use.

**Table 1 - Node applications and required interfaces**

Object Path	Interface	Comment
<i>Your application's root object path such as /mesh/gateway</i>	org.freedesktop.DBus.ObjectManager	mandatory
e.g.	org.bluez.mesh.ProvisionAgent1	mandatory

/mesh/gateway/agent		
e.g. /mesh/gateway/ele	org.bluez.mesh.Element1	mandatory

#### 4.4.4 The BlueZ Mesh API

A D-Bus service called org.bluez.mesh exposes functions which collectively act as a BlueZ mesh API. Node applications will make use of this API.

#### 4.4.5 Attaching to the network

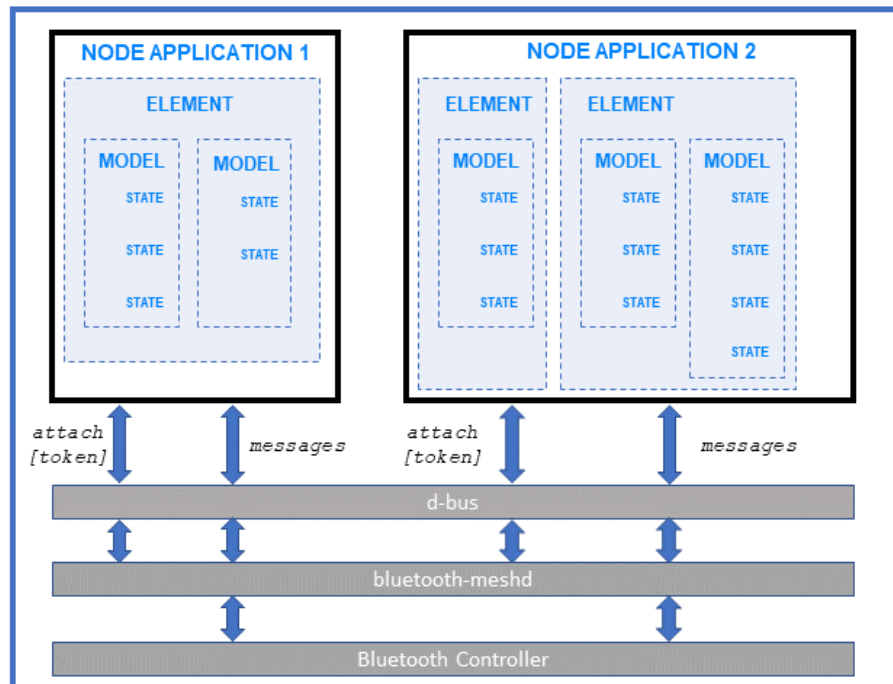


Figure 6 - BlueZ mesh applications

Figure 6 depicts two applications which act as Bluetooth mesh nodes running on a single Linux computer. It also introduces the BlueZ mesh terms *attach* and *token*.

When an application is provisioned, in the terminology used by BlueZ, it *joins* the mesh network. On joining or being successfully provisioned, it is issued with a unique numeric identifier called a *token*.

When the provisioned application is run, before it can send or receive mesh messages in the network, it must *attach*. Attaching requires the token value that was issued to the application (instance) when it was provisioned to be provided. This acts as an authentication step and also enables the bluetooth-meshd daemon to exchange data with the right application instances.

You will gain experience of these concepts in practice later in this project.

#### 4.4.6 Foundation models and network management

The bluetooth-meshd daemon implements the mesh foundation models for the primary element of each node application provisioned on its host computer. This means that the important foundation models such as the configuration server model are provided by the daemon as standard and node applications should not seek to implement such models. Additionally, mesh network management

procedures and features including but not limited to Publication, Subscription, Binding, Key Refresh, IV Update, TTL setup and Relaying are actively available without any node application needing to be running and attached.

## 4.5. Gateway Development

By the end of this section, you should have a working gateway which meets the requirements which were selected in module 02 and extended in this module.

*The initial implementation will not be secure and should not be connected to the internet or other insecure networks!*

Two Raspberry Pis are needed for this project but only one of them will host the gateway components. The second Pi will act as a Bluetooth mesh provisioner and configuration manager. It will be referred to as the *Provisioner* for short.

Some of the following set-up instructions must be completed for both Raspberry Pis. Some only apply to the gateway device only.

### 4.5.1 Platform Set-up

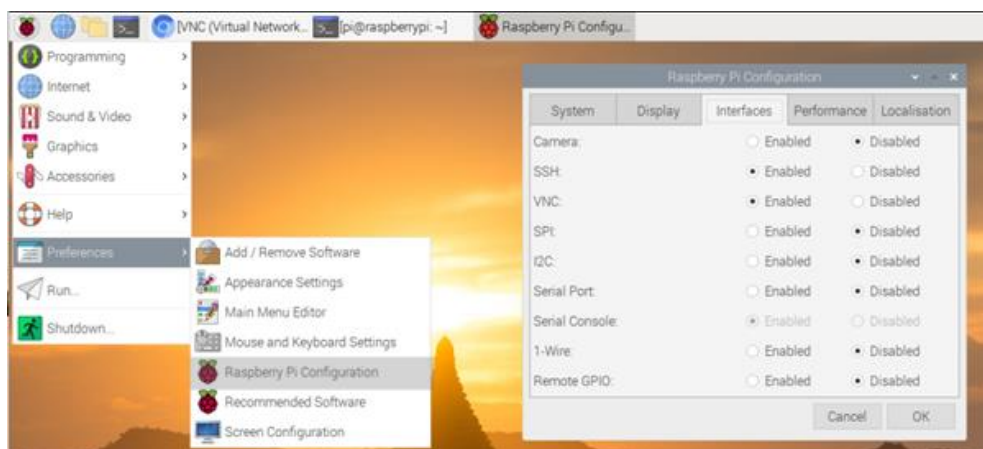
We need to establish a platform upon which to build the gateway. We'll define this as a computer which has the required Bluetooth capabilities and supports our selected programming language, Python.

#### 4.5.1.1 Task - Raspberry Pi set-up

This section must be completed for **both** Raspberry Pi devices.

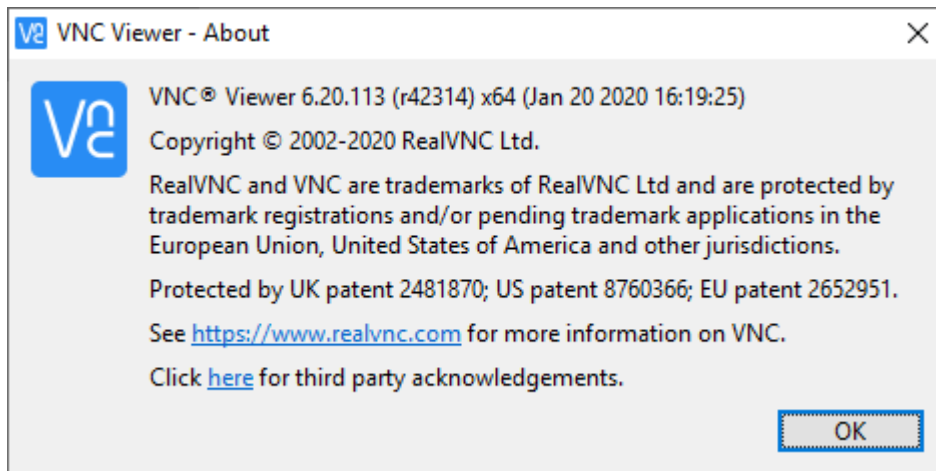
Set up each Raspberry Pi by following the instructions at <https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up>.

Enable access to your Raspberry Pi via SSH and VNC by going into the desktop menu and navigating to Preferences/Raspberry Pi Configuration.





You may now work directly on your Raspberry Pi, with a USB keyboard and mouse connected to it and a monitor connected via HDMI. Alternatively, with a suitable VNC Viewer application on your usual PC, you can use remote access to work with the Raspberry Pi.



Your final option is to use an SSH terminal such as Putty from <https://putty.org/>.

Determine the IP address of each Raspberry Pi by running either *ifconfig* or the alternative command, *ip a*. Make a note of the IP address.

```
pi@raspberrypi:~$ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether dc:a6:32:0e:83:e0 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 25 bytes 1484 (1.4 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 25 bytes 1484 (1.4 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.39 netmask 255.255.255.0 broadcast 192.168.0.255
    inet6 fe80::4666:142d:850b:c0f6 prefixlen 64 scopeid 0x20<link>
    ether dc:a6:32:0e:83:e1 txqueuelen 1000 (Ethernet)
    RX packets 10909 bytes 7774618 (7.4 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 10963 bytes 8064070 (7.6 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

pi@raspberrypi:~$
```

Windows users note: One of the commands for acquiring your IP address on a Linux machine is *ifconfig*. On Windows however, the command is *ipconfig*. Make sure you issue the command for Linux!

#### 4.5.1.2 Task - Prepare the Linux Kernel and Libraries

**This section must be completed for both Raspberry Pi devices.**

Bluetooth mesh uses cryptographic functions such as AES-CMAC. A number of libraries must be installed and the Linux Kernel must be compiled with the required cryptographic functions included. Execute the following steps to achieve this.

### 1. Get your package lists up to date and then install dependencies.

```
sudo apt-get update

sudo apt-get install -y git bc libusb-dev libdbus-1-dev libglib2.0-dev libudev-dev libical-dev libreadline-dev autoconf bison flex libssl-dev libncurses-dev glib2.0 libdbus-1-dev
```

### 2. Download the Linux kernel source and install its dependencies

```
cd ~

wget https://github.com/raspberrypi/linux/archive/raspberrypi-kernel_1.20210303-1.tar.gz

tar -xvf raspberrypi-kernel_1.20210303-1.tar.gz

cd ./linux-raspberrypi-kernel_1.20210303-1

sudo apt install bc bison flex libssl-dev make
```

### 3. Configure, build and the install the recompiled kernel

```
# If you're using a Pi Zero:
KERNEL=kernel
make bcmrpi_defconfig

# If you're using a Pi 4 or Pi 400:
KERNEL=kernel7l
make bcm2711_defconfig

make menuconfig

# In the menus that appear, select the following options
Cryptographic API-->
* CCM Support
* CMAC Support
* User-space interface for hash algorithms
* User-space interface for symmetric key cipher algorithms
* User-space interface for AEAD cipher algorithms

# Save to .config and exit menuconfig
make -j4 zImage modules dtbs

sudo make modules_install

sudo cp arch/arm/boot/dts/*.dtb /boot/

sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/

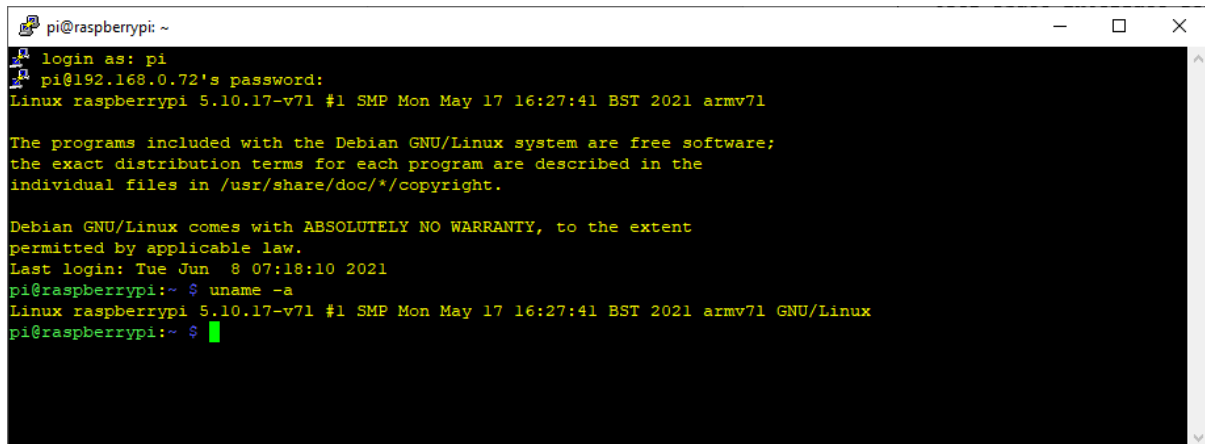
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/

sudo cp arch/arm/boot/zImage /boot/$KERNEL.img

sudo reboot
```

### 4. After rebooting, check the kernel version

```
uname -a
```

A terminal window titled 'pi@raspberrypi: ~' showing the output of the 'uname -a' command. The output is: 'Linux raspberrypi 5.10.17-v7l #1 SMP Mon May 17 16:27:41 BST 2021 armv7l GNU/Linux'. Above this, there is a login prompt and a password prompt. Below the output, there is a message about Debian GNU/Linux being free software and a warranty disclaimer. The prompt 'pi@raspberrypi:~ \$' is followed by 'uname -a' and the output is shown on the next line. The prompt is then 'pi@raspberrypi:~ \$' again.

```
pi@raspberrypi:~  
login as: pi  
pi@192.168.0.72's password:  
Linux raspberrypi 5.10.17-v7l #1 SMP Mon May 17 16:27:41 BST 2021 armv7l  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
Last login: Tue Jun  8 07:18:10 2021  
pi@raspberrypi:~ $ uname -a  
Linux raspberrypi 5.10.17-v7l #1 SMP Mon May 17 16:27:41 BST 2021 armv7l GNU/Linux  
pi@raspberrypi:~ $
```

#### 4.5.1.3 Task - Install BlueZ 5.58 and Dependencies

This section must be completed for **both** Raspberry Pi devices.

Check the version of BlueZ running on your machine by launching the *bluetooth* tool and entering the *version* command.

```
pi@raspberrypi:~ $ bluetoothctl  
Agent registered  
[bluetooth]# version  
Version 5.50  
[bluetooth]#
```

The version reported should be at least version 5.58. If an earlier version is reported, install dependencies and then download and build the required version of BlueZ. Note that this resource was created and tested using version 5.58. Later versions may be OK but this is not guaranteed.

Exit bluetoothctl by entering the *quit* command.

#### 1. Install libraries

```
sudo apt-get install libglib2.0-dev libusb-dev libdbus-1-dev libudev-dev libreadline-dev  
libical-dev
```

#### 2. Install JSON library

```
sudo apt install cmake  
  
wget https://s3.amazonaws.com/json-c_releases/releases/json-c-0.15.tar.gz  
tar xvf json-c-0.15.tar.gz  
  
cd ~/json-c-0.15  
mkdir build  
cd build  
  
cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=Release -DBUILD_STATIC_LIBS=OFF ..  
make
```

```
sudo make install
```

### 3. Install the Embedded Linux Library (ELL)

```
cd ~
wget https://mirrors.edge.kernel.org/pub/linux/libs/ell/ell-0.6.tar.xz
tar -xvf ell-0.6.tar.xz
cd ell-0.6/
sudo ./configure --prefix=/usr
sudo make
sudo make install
```

### 4. Install BlueZ

```
cd ~

pi@raspberrypi:~ $ wget http://www.kernel.org/pub/linux/bluetooth/bluez-5.58.tar.xz
URL transformed to HTTPS due to an HSTS policy
--2021-04-08 12:28:05-- https://www.kernel.org/pub/linux/bluetooth/bluez-5.58.tar.xz
Resolving www.kernel.org (www.kernel.org)... 136.144.49.103, 2604:1380:40b0:1a00::1
Connecting to www.kernel.org (www.kernel.org)|136.144.49.103|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://mirrors.edge.kernel.org/pub/linux/bluetooth/bluez-5.58.tar.xz [following]
--2021-04-08 12:28:06-- https://mirrors.edge.kernel.org/pub/linux/bluetooth/bluez-
5.58.tar.xz
Resolving mirrors.edge.kernel.org (mirrors.edge.kernel.org)... 147.75.101.1,
2604:1380:2001:3900::1
Connecting to mirrors.edge.kernel.org (mirrors.edge.kernel.org)|147.75.101.1|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 2060368 (2.0M) [application/x-xz]
Saving to: 'bluez-5.58.tar.xz'

bluez-5.58.tar.xz
100%[=====] 1.96M 6.12MB/s in 0.3s

2021-04-08 12:28:06 (6.12 MB/s) - 'bluez-5.58.tar.xz' saved [2060368/2060368]

tar -xvf bluez-5.58.tar.xz
cd bluez-5.58/
./configure --enable-mesh --enable-testing --enable-tools --prefix=/usr --
mandir=/usr/share/man --sysconfdir=/etc --localstatedir=/var
sudo make
sudo make install
```

#### 4.5.1.4 Task - Bluetooth DBus Communication

The Bluetooth API uses the Linux DBus inter-process communication system. A default security policy restricts use of DBus for Bluetooth purposes to processes owned by users that are a member of the *bluetooth* system group. You can see this in the `/etc/dbus-1/system.d/bluetooth.conf` file, here:

```
<!-- allow users of bluetooth group to communicate -->
<policy group="bluetooth">
  <allow send_destination="org.bluez"/>
</policy>
```

Edit `/etc/group` and add the *www-data* and *pi* users to the bluetooth group.

```
pi@raspberrypi:~ $ sudo vi /etc/group
pi@raspberrypi:~ $ cat /etc/group|grep bluet
bluetooth:x:112:www-data,pi
# you need to start a new bash shell for the change to be activated - do this by entering
# su - pi
# or exit this shell and start a new one
```

```
pi@raspberrypi:~ $ groups
pi adm dialout cdrom sudo audio www-data video plugdev games users input netdev bluetooth
gpio i2c spi
```

Now restart the DBus daemon:

```
pi@raspberrypi:~ $ sudo service dbus restart
```

#### 4.5.1.5 Task - Start bluetooth-meshd

Create a script called mesh.sh which will stop the bluetoothd daemon (this will usually be running after rebooting) and starts the bluetooth-meshd daemon instead.

```
pi@raspberrypi:~ $ cat mesh.sh
#!/bin/bash
sudo service bluetooth stop
sudo service bluetooth-mesh start
sudo service bluetooth-mesh status
```

Make the script executable and then run it.

```
pi@raspberrypi:~ $ chmod 755 mesh.sh
pi@raspberrypi:~ $ ./mesh.sh
• bluetooth-mesh.service - Bluetooth mesh service
   Loaded: loaded (/lib/systemd/system/bluetooth-mesh.service; disabled; vendor preset:
   enabled)
   Active: active (running) since Tue 2021-06-08 07:48:42 BST; 68ms ago
   Main PID: 1606 (bluetooth-meshd)
   Tasks: 1 (limit: 4915)
   CGroup: /system.slice/bluetooth-mesh.service
           └─1606 /usr/libexec/bluetooth/bluetooth-meshd -ndb

Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 65 73 41 64 64 65 64 00 02 01
73 00 22 00 00 00 esA
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 6f 72 67 2e 66 72 65 65 64 65
73 6b 74 6f 70 2e org
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 44 42 75 73 2e 4f 62 6a 65 63
74 4d 61 6e 61 67 DBu
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 65 72 00 00 00 00 00 00 08 01
67 00 0a 6f 61 7b er.
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 73 61 7b 73 76 7d 7d 00
sa{
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 0f 00 00 00 2f 6f 72 67 2f 62
6c 75 65 7a 2f 6d ...
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 65 73 68 00 20 00 00 00 17 00
00 00 6f 72 67 2e esh
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 62 6c 75 65 7a 2e 6d 65 73 68
2e 4e 65 74 77 6f blu
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: [DBUS] 72 6b 31 00 00 00 00 00
rkl
Jun 08 07:48:42 raspberrypi bluetooth-meshd[1606]: LE Scan disable failed (0x0c)
```

You can ignore the “LE Scan disabled failed” message at the end.

#### 4.5.1.6 Task - Python check

This section must be completed for **both** Raspberry Pi devices.

Check that Python 3 is installed and that it is functioning, as follows:

```
pi@raspberrypi:~ $ python3 --version

Python 3.7.3
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World

>>> quit()
```

Install the *termcolor* Python library which is used by some BlueZ scripts.

```
sudo pip3 install termcolor
```

## 4.5.2 Apache Web Server

This section only needs to be completed for the gateway Raspberry Pi device.

### 4.5.2.1 Task - Install Apache2

Install the Apache2 web server as follows:

```
sudo apt install apache2
```

Check that Apache is running using *service* like this:

```
pi@raspberrypi:~ $ sudo service apache2 status
• Apache2.service - The Apache HTTP Server
  Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset: enabled)
  Active: active (running) since Mon 2020-04-27 13:42:48 BST; 3min 10s ago
    Docs: https://httpd.apache.org/docs/2.4/
  Main PID: 2950 (Apache2)
    Tasks: 55 (limit: 4915)
   Memory: 4.0M
   CGroup: /system.slice/apache2.service
           └─2950 /usr/sbin/apache2 -k start
             └─2951 /usr/sbin/apache2 -k start
               └─2952 /usr/sbin/apache2 -k start

Apr 27 13:42:48 raspberrypi systemd[1]: Starting The Apache HTTP Server...
Apr 27 13:42:48 raspberrypi Apachectl[2939]: AH00558: Apache2: Could not reliably determine
the server's fully qualified domain name, using 127.0.1.1. Set the 'ServerName' directive
globally to suppress this
Apr 27 13:42:48 raspberrypi systemd[1]: Started The Apache HTTP Server.
```

Note the highlighted message regarding *ServerName*. Address this by editing the */etc/apache2/apache2.conf* configuration file, adding a *ServerName* directive with a value of *raspberrypi* and restarting Apache.

```
sudo vi /etc/apache2/apache2.conf
```

```
pi@raspberrypi: ~  
# ServerRoot: The top of the directory tree under which the server's  
# configuration, error, and log files are kept.  
#  
# NOTE! If you intend to place this on an NFS (or otherwise network)  
# mounted filesystem then please read the Mutex documentation (available  
# at <URL:http://httpd.apache.org/docs/2.4/mod/core.html#mutex>);  
# you will save yourself a lot of trouble.  
#  
# Do NOT add a slash at the end of the directory path.  
#  
#ServerRoot "/etc/apache2"  
  
ServerName raspberrypi  
#  
# The accept serialization lock file MUST BE STORED ON A LOCAL DISK.  
#  
#Mutex file:${APACHE_LOCK_DIR} default  
#  
# The directory where shm and other runtime files will be stored.  
#  
DefaultRuntimeDir ${APACHE_RUN_DIR}  
#  
# PidFile: The file in which the server should record its process  
"/etc/apache2/apache2.conf" 228 lines, 898964 characters written
```

Restart the Apache2 service and check its status:

```
pi@raspberrypi:~ $ sudo service apache2 restart  
pi@raspberrypi:~ $ sudo service apache2 status  
● Apache2.service - The Apache HTTP Server  
   Loaded: loaded (/lib/systemd/system/Apache2.service; enabled; vendor preset: enabled)  
   Active: active (running) since Mon 2020-04-27 13:52:17 BST; 6s ago  
     Docs: https://httpd.apache.org/docs/2.4/  
  Process: 3527 Exec Start=/usr/sbin/Apachectl start (code=exited, status=0/SUCCESS)  
 Main PID: 3531 (Apache2)  
    Tasks: 55 (limit: 4915)  
   Memory: 3.8M  
   CGroup: /system.slice/Apache2.service  
           └─3531 /usr/sbin/Apache2 -k start  
             └─3532 /usr/sbin/Apache2 -k start  
               └─3533 /usr/sbin/Apache2 -k start  
  
Apr 27 13:52:17 raspberrypi systemd[1]: Starting The Apache HTTP Server...  
Apr 27 13:52:17 raspberrypi systemd[1]: Started The Apache HTTP Server.
```

You should also be able to ping the host name *raspberrypi*.

```
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $  
pi@raspberrypi:~ $ ping raspberrypi  
PING raspberrypi (127.0.1.1) 56(84) bytes of data:  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=1 ttl=64 time=0.078 ms  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=2 ttl=64 time=0.071 ms  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=3 ttl=64 time=0.080 ms  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=4 ttl=64 time=0.070 ms  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=5 ttl=64 time=0.066 ms  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=6 ttl=64 time=0.070 ms  
64 bytes from raspberrypi (127.0.1.1): icmp_seq=7 ttl=64 time=0.074 ms  
^C  
--- raspberrypi ping statistics ---  
7 packets transmitted, 7 received, 0% packet loss, time 277ms  
rtt min/avg/max/mdev = 0.066/0.072/0.080/0.011 ms  
pi@raspberrypi:~ $
```

Apache is a modular system and by default, various modules are enabled. Find out which are currently enabled using the *apache2ctl* command as shown.

```
pi@raspberrypi:~ $ apache2ctl -t -D DUMP_MODULES
Loaded Modules:
  core_module (static)
  so_module (static)
  watchdog_module (static)
  http_module (static)
  log_config_module (static)
  logio_module (static)
  version_module (static)
  unixd_module (static)
  access_compat_module (shared)
  alias_module (shared)
  auth_basic_module (shared)
  authn_core_module (shared)
  authn_file_module (shared)
  authz_core_module (shared)
  authz_host_module (shared)
  authz_user_module (shared)
  autoindex_module (shared)
  deflate_module (shared)
  dir_module (shared)
  env_module (shared)
  filter_module (shared)
  mime_module (shared)
  mpm_event_module (shared)
  negotiation_module (shared)
  reqtimeout_module (shared)
  setenvif_module (shared)
  status_module (shared)
```

Our plan is to use the Common Gateway Interface (CGI) for executing Python scripts in response to certain HTTP requests. For this to work, we need the Apache *cgid* module enabled. As you can see from the list of modules reported by *apache2ctl*, by default it is not. Enable it using *a2enmod*, list the enabled modules again to check it has been enabled, and then restart Apache.

```
pi@raspberrypi:~ $ sudo a2enmod cgid
Enabling module cgid.
To activate the new configuration, you need to run:
  systemctl restart Apache2

pi@raspberrypi:~ $ Apache2ctl -t -D DUMP_MODULES
Loaded Modules:
  core_module (static)
  so_module (static)
  watchdog_module (static)
  http_module (static)
  log_config_module (static)
  logio_module (static)
  version_module (static)
  unixd_module (static)
  access_compat_module (shared)
  alias_module (shared)
  auth_basic_module (shared)
  authn_core_module (shared)
  authn_file_module (shared)
  authz_core_module (shared)
  authz_host_module (shared)
  authz_user_module (shared)
  autoindex_module (shared)
  cgid_module (shared)
  deflate_module (shared)
  dir_module (shared)
  env_module (shared)
  filter_module (shared)
  mime_module (shared)
  mpm_event_module (shared)
  negotiation_module (shared)
  reqtimeout_module (shared)
  setenvif_module (shared)
  status_module (shared)
```



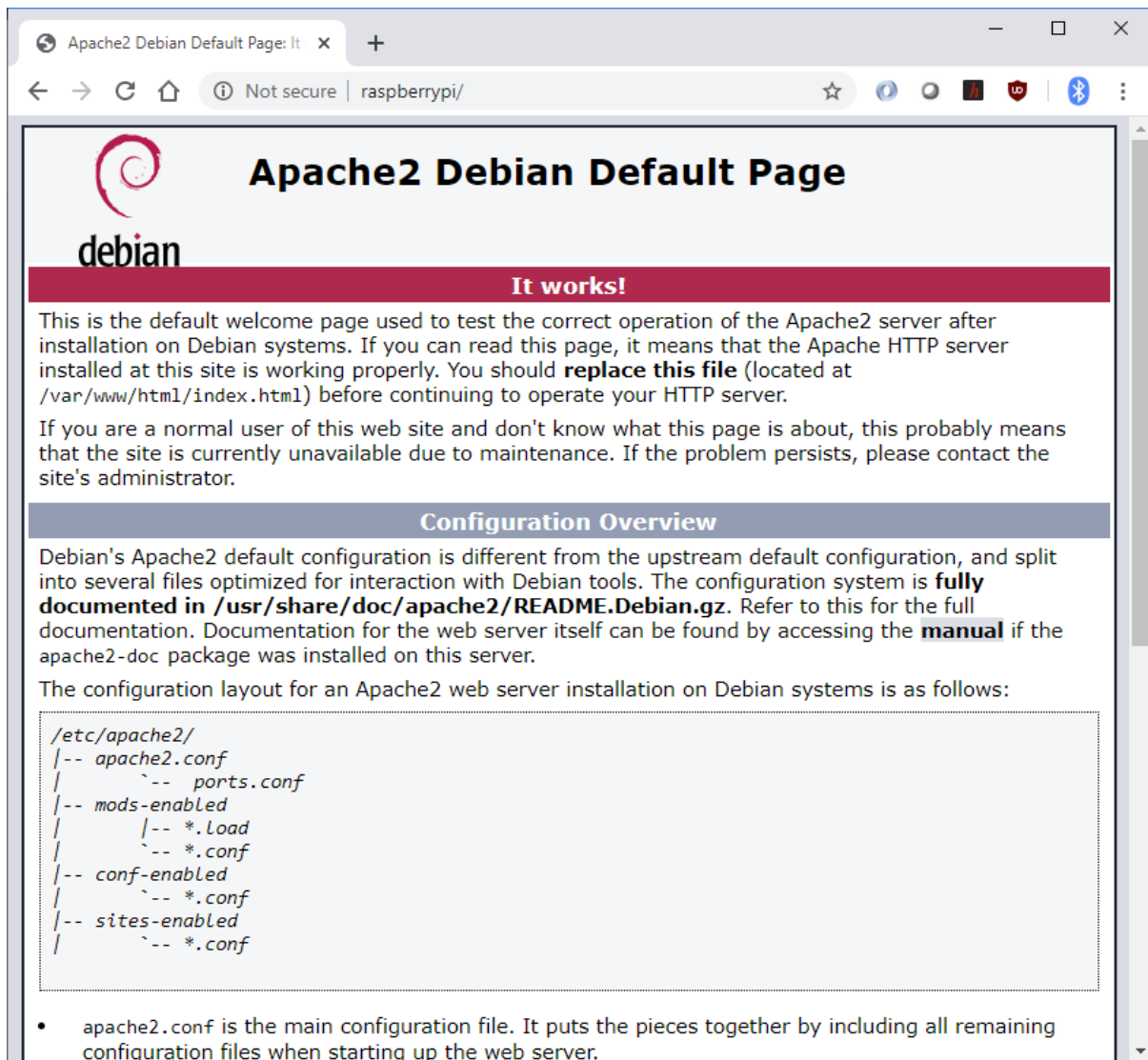
```
pi@raspberrypi:~ $ sudo service apache2 restart
```

If you are intending to test from a machine other than the gateway itself (and this is recommended), add an entry to the local *hosts* file on the other computer, and associate name *raspberrypi* with the IP address you acquired from your Raspberry Pi earlier. On Windows, this involves starting an editor as Administrator and editing the file `c:\windows\system32\drivers\etc\hosts` and adding an entry such as:

```
192.168.0.39    raspberrypi
```

You should now be able to ping *raspberrypi* from this computer, provided it is connected to the same LAN as the Raspberry Pi.

Check that you can reach the default home page of your Apache web server by going to <http://raspberrypi> in your web browser. You should see:



The screenshot shows a web browser window with the title "Apache2 Debian Default Page: It works!". The address bar shows "Not secure | raspberrypi/". The page content includes the Apache logo, the text "It works!", and a message stating that the Apache HTTP server is working properly. It also provides instructions on how to replace the default index.html file and how to access the configuration overview. The configuration overview section lists the default configuration files and their locations.

**Apache2 Debian Default Page**

**It works!**

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Debian systems. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

**Configuration Overview**

Debian's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Debian tools. The configuration system is **fully documented in `/usr/share/doc/apache2/README.Debian.gz`**. Refer to this for the full documentation. Documentation for the web server itself can be found by accessing the **manual** if the `apache2-doc` package was installed on this server.

The configuration layout for an Apache2 web server installation on Debian systems is as follows:

```
/etc/apache2/
|-- apache2.conf
|   |-- ports.conf
|-- mods-enabled
|   |-- *.load
|   |-- *.conf
|-- conf-enabled
|   |-- *.conf
|-- sites-enabled
|   |-- *.conf
```

- `apache2.conf` is the main configuration file. It puts the pieces together by including all remaining configuration files when starting up the web server.

#### 4.5.2.2 Task - Verify that CGI and Python script execution is working

Using the editor of your choice, create a script file called `hello.py` located at `/usr/lib/cgi-bin`. It should contain the following, simple Python script:

```
#!/usr/bin/python
print "Content-type: text/html\n\n"
print "Hello World."
```

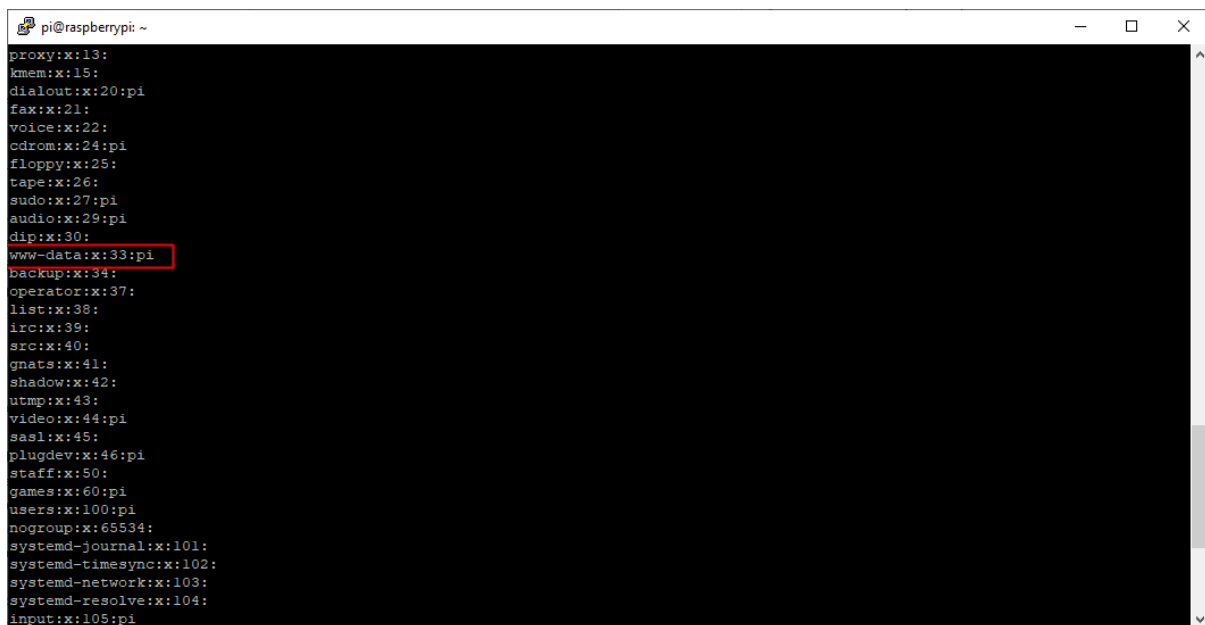
**Note:** for this and many other steps in the exercises, you must ensure you have the required permissions to perform the required action. The easiest way to do this is to prefix any commands with *sudo*. This will temporarily give your command the superuser (root) privileges.

Change the file's owner and group to `www-data` (the user which Apache runs as) and permissions to `775` so that both the `www-data` user and group members have read, write and execute access.

```
pi@raspberrypi:~ $ sudo chown www-data:www-data /usr/lib/cgi-bin/hello.py
pi@raspberrypi:~ $ sudo chmod 775 /usr/lib/cgi-bin/hello.py
pi@raspberrypi:~ $ ls -l /usr/lib/cgi-bin/
total 4
-rwxrwxr-x 1 www-data 77 Apr 27 14:03 hello.py
```

It will be assumed from now on that you will give all CGI scripts created in this directory, this ownership and set of permissions.

Edit `/etc/group` with *sudo vi /etc/group* and add the *pi* user to the *www-data* group.



```
pi@raspberrypi: ~
proxy:x:13:
kmem:x:15:
dialout:x:20:pi
fax:x:21:
voice:x:22:
cdrom:x:24:pi
floppy:x:25:
tape:x:26:
sudo:x:27:pi
audio:x:29:pi
dip:x:30:
www-data:x:33:pi
backup:x:34:
operator:x:37:
list:x:38:
irc:x:39:
src:x:40:
gnats:x:41:
shadow:x:42:
utmp:x:43:
video:x:44:pi
sas1:x:45:
plugdev:x:46:pi
staff:x:50:
games:x:60:pi
users:x:100:pi
nogroup:x:65534:
systemd-journal:x:101:
systemd-timesync:x:102:
systemd-network:x:103:
systemd-resolve:x:104:
input:x:105:pi
```

Now your default Linux user will have write and execute access to CGI scripts, which will make working with these scripts easier.

Go to `http://raspberrypi/cgi-bin/hello.py` in your browser. You should see a web page containing only "Hello World."

If you did, you have verified that your Apache web server is both working and correctly configured to be able to execute Python scripts. If not, carefully check the previous steps to ensure you have not

missed anything. The last few entries of the Apache error log, at `/var/log/apache2/error.log` may provide some insight.

### 4.5.3 websocketd

This section only needs to be completed for the gateway Raspberry Pi device.

Our architecture includes the use of web sockets for the enabling, disabling and delivery of Bluetooth notifications and indications and a daemon which will act as a dispatcher for web socket traffic called `websocketd`.

#### 4.5.3.1 Task - Install `websocketd`

Go to the `websocketd` web site at <http://websocketd.com/> . Follow the instructions there to download and install `websocketd` and to test it with a simple Python script.

**Tip: You need the Linux ARM version for a Raspberry Pi**

```
# expand the package archive

pi@raspberrypi:~/Downloads $ unzip websocketd-0.3.0-linux_arm.zip
Archive:  websocketd-0.3.0-linux_arm.zip
  inflating: websocketd
  inflating: README.md
  inflating: LICENSE
  inflating: CHANGES

# which directories are already on the PATH?

pi@raspberrypi:~/Downloads $ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games

# copy binary to a suitable directory

pi@raspberrypi:~/Downloads $ sudo cp websocketd /usr/local/bin

# check websocketd is available

pi@raspberrypi:~/Downloads $ which websocketd
/usr/local/bin/websocketd
```

#### 4.5.3.2 Task - Verify `websocketd`

Next we'll check that `websocketd` is working by setting up a simple web page which will receive a series of numbers over a web socket connection, generated by a Python script running on the Raspberry Pi. Follow the steps below to create a web page and the required Python script.

```
# create directory and web page

pi@raspberrypi:~ $ sudo mkdir /var/www/html/wstest
pi@raspberrypi:~ $ sudo vi /var/www/html/wstest/index.html

# index.html must contain the following:

<html>
<head>
  <script src="js/wstest.js"></script>
</head>
<body>
  <h2>Web Socket Test</h2>
```

```

        <table><tr><td><button id="btn_test" class="button" onclick="onTest();"
/>TEST</button></td></tr></table>
        <div id="message" class="important"></div>
        <pre id="content"></pre>

</body>
</html>

# create a directory for JavaScript files and the JavaScript needed by our test page
pi@raspberrypi:~ $ sudo mkdir /var/www/html/wstest/js
pi@raspberrypi:~ $ sudo vi /var/www/html/wstest/js/wstest.js

# Content of wstest.js:

function onTest() {

    var ws = new WebSocket('ws://raspberrypi:8080/');

    ws.onmessage = function(event) {
        document.getElementById('content').innerHTML = 'Notification: ' + event.data;
    };

}

# set ownership and permissions
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /var/www/html/wstest/
pi@raspberrypi:~ $ sudo chmod -R 775 /var/www/html/wstest/

# create a Python script which will write to websockets
pi@raspberrypi:~ $ sudo vi /usr/lib/cgi-bin/wstest.py

# /usr/lib/cgi-bin/wstest.py content:

#!/usr/bin/python
from sys import stdout
from time import sleep

# Count from 1 to 10 with a sleep
for count in range(0, 10):
    print(count + 1)
    stdout.flush()
    sleep(0.5)

# set ownership and permissions
pi@raspberrypi:~ $ sudo chown www-data:www-data /usr/lib/cgi-bin/wstest.py
pi@raspberrypi:~ $ sudo chmod 775 /usr/lib/cgi-bin/wstest.py

```

Now start websocketd, listening on port 8080 like this:

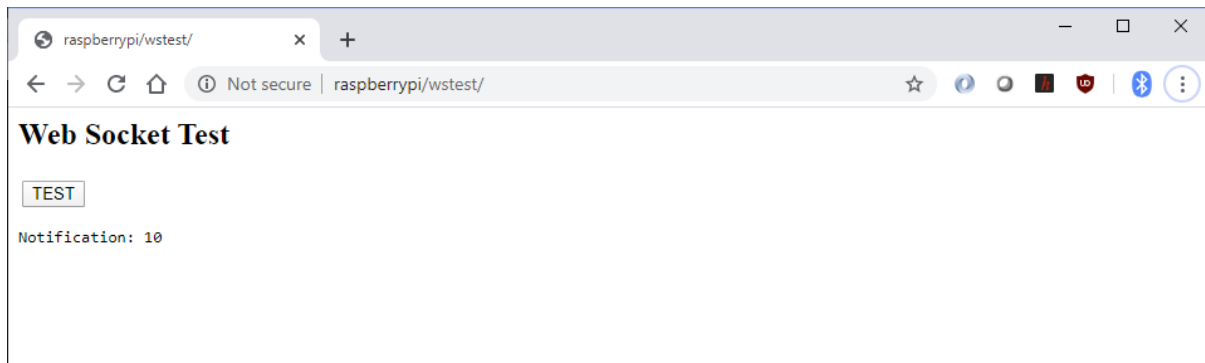
```

pi@raspberrypi:~ $ websocketd --port=8080 /usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:25:55 +0100 | INFO    | server      | | Serving using application :
/usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:25:55 +0100 | INFO    | server      | | Starting WebSocket server :
ws://raspberrypi:8080/

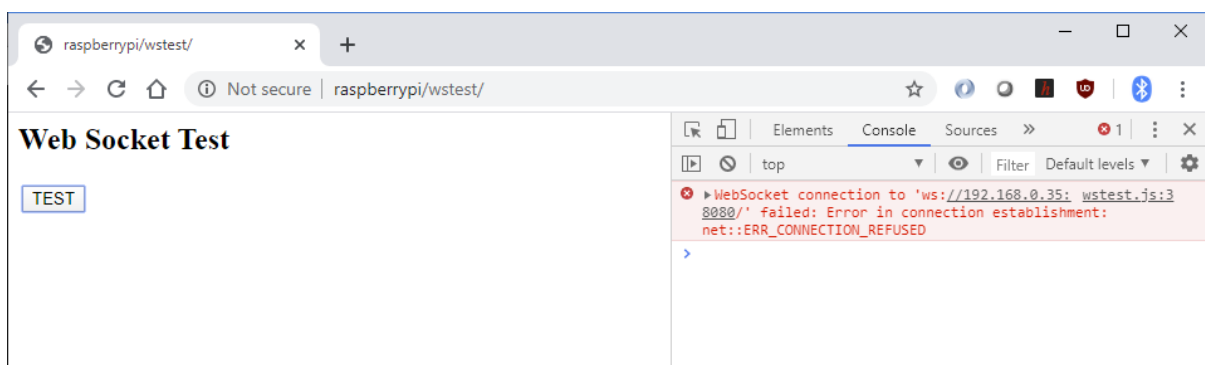
```

In a browser, access the test page here: <http://raspberrypi/wstest/>

Click the start button and you should see a series of integer values, from 1 to 10, appearing on the web page. These values were generated by the Python script, which was launched by websocketd and written to the web socket connection.



If it didn't work for you, open the developer console in your browser. Using Google Chrome on Windows, CTRL+SHIFT+I will achieve this result quickly. The console should offer clues, the most likely issues being that either websocketd is not running, it's running but listening on a different port to that which is used in the JavaScript or the JavaScript specifies the wrong IP address / host name.



After successfully executing, the Python script will exit and the websocketd console will indicate that the web socket has been disconnected.

```
pi@raspberrypi:~ $ websocketd --port=8080 /usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:28:40 +0100 | INFO    | server      | | Serving using application   : /usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:28:40 +0100 | INFO    | server      | | Starting WebSocket server   : ws://raspberrypi:8080/
Tue, 28 Apr 2020 08:28:42 +0100 | ACCESS  | session     | url:'http://192.168.0.35:8080/' id:'1588058922060392454' remote:'192.168.0.20' command:'/usr/lib/cgi-bin/wstest.py' origin:'http://raspberrypi' | CONNECT
Tue, 28 Apr 2020 08:28:47 +0100 | ACCESS  | session     | url:'http://192.168.0.35:8080/' id:'1588058922060392454' remote:'192.168.0.20' command:'/usr/lib/cgi-bin/wstest.py' origin:'http://raspberrypi' pid:'2531' | DISCONNECT
```

## 4.5.4 Mesh Network

### 4.5.4.1 Task - Create the mesh network

This section must be completed using the **Provisioner** device

Log on to the raspberry Pi which will act as your Provisioner. If this is the first time you have accessed the device since booting, run the mesh.sh script that you created in section 5.1.4. From now on it will be assumed that you have the bluetooth-meshd daemon running.

BlueZ includes a command line tool called mesh-cfgclient. It allows a new mesh network to be created, devices to be added to the network as nodes through provisioning and the various aspects of a node to be configured. Your first job involving mesh-cfgclient is to create a new mesh network of which your Provisioner device will be the first member node.

Run mesh-cfgclient, review the available commands by entering *help* and then create your mesh network and its security keys as shown:

```
pi@raspberrypi:~ $ mesh-cfgclient

Warning: config file "/home/pi/.config/meshcfg/config_db.json" not found
[mesh-cfgclient]# help
Menu main:
Available commands:
-----
create [unicast_range_low]                Create new mesh network with one initial
node                                     node
discover-unprovisioned <on/off> [seconds] Look for devices to provision
appkey-create <net_idx> <app_idx>         Create a new local AppKey
appkey-import <net_idx> <app_idx> <key>    Import a new local AppKey
appkey-update <app_idx>                   Update local AppKey
appkey-delete <app_idx>                   Delete local AppKey
subnet-create <net_idx>                   Create a new local subnet (NetKey)
subnet-import <net_idx> <key>             Import a new local subnet (NetKey)
subnet-update <net_idx>                   Update local subnet (NetKey)
subnet-delete <net_idx>                   Delete local subnet (NetKey)
subnet-set-phase <net_idx> <phase>        Set subnet (NetKey) phase
list-unprovisioned                       List unprovisioned devices
provision <uuid>                         Initiate provisioning
node-import <uuid> <net_idx> <primary> <ele_count> <dev_key> Import an externally
provisioned remote node
list-nodes                               List remote mesh nodes
keys                                     List available keys
version                                 Display version
quit                                   Quit program
exit                                  Quit program
help                                 Display help about this program
export                                Print environment variables

[mesh-cfgclient]# create
Created new node with token fac47414b6bb4d91
Proxy added: org.bluez.mesh.Node1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Proxy added: org.bluez.mesh.Management1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Attached with path /org/bluez/mesh/node8c758f3726d147d488a42a7007957471
```

You can ignore the warning about config\_db.json. After creating your network the file will exist and contain details of all nodes. You should not see this message again. If you do, check section 9 for troubleshooting tips.

#### 4.5.4.2 Task - Program a Nordic Thingy

Plug your Nordic Thingy into a Segger J-Link debugger and the J-Link into your computer using a USB cable. Connect a USB cable to the Thingy and a power source such as your computer. Switch the Thingy on and use the `nrfjprog` command to flash the provided `zephyr.hex` binary (see `implementation\solutions\mesh\nodes\thingy\Light_and_Sensor\binary`) to your Nordic Thingy device. Do this for each of your devices if you have more than one.

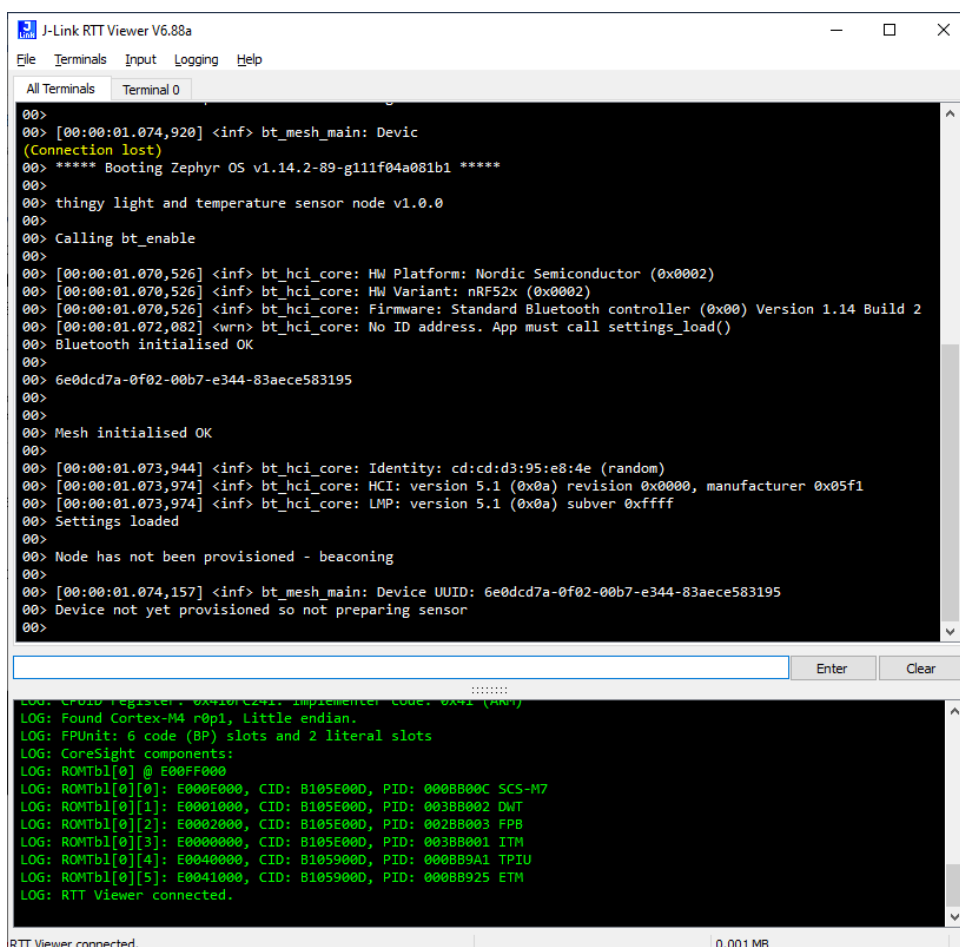
```
C:\BIG_SG\implementation\solutions\mesh\nodes\thingy\Light_and_Sensor\binary>nrfjprog --
program zephyr.hex
Parsing image file.
Reading flash area to program to guarantee it is erased.
Checking that the area to write is not protected.
Programming device.
```

#### 4.4.4.3 Task - Provision a Nordic Thingy

This section must be completed using the **Provisioner** device, once per device being used as a test node

The Nordic Thingy device has been provided with an implementation of the server models needed for the project but has not yet been provisioned. That's your next task.

Start the J-Link RTT Viewer tool and connect to your Thingy. The console should look like this. You can see that the device has not yet been provisioned and is therefore beaconing.



```
J-Link RTT Viewer V6.88a
File Terminals Input Logging Help
All Terminals Terminal 0
00>
00> [00:00:01.074,920] <inf> bt_mesh_main: Device
(Connection lost)
00> ***** Booting Zephyr OS v1.14.2-89-g111f04a081b1 *****
00>
00> thingy light and temperature sensor node v1.0.0
00>
00> Calling bt_enable
00>
00> [00:00:01.070,526] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
00> [00:00:01.070,526] <inf> bt_hci_core: HW Variant: nRF52x (0x0002)
00> [00:00:01.070,526] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00) Version 1.14 Build 2
00> [00:00:01.072,082] <warn> bt_hci_core: No ID address. App must call settings_load()
00> Bluetooth initialised OK
00>
00> 6e0cd7a-0f02-00b7-e344-83aece583195
00>
00>
00> Mesh initialised OK
00>
00> [00:00:01.073,944] <inf> bt_hci_core: Identity: cd:cd:d3:95:e8:4e (random)
00> [00:00:01.073,974] <inf> bt_hci_core: HCI: version 5.1 (0x0a) revision 0x0000, manufacturer 0x05f1
00> [00:00:01.073,974] <inf> bt_hci_core: LMP: version 5.1 (0x0a) subver 0xffff
00> Settings loaded
00>
00> Node has not been provisioned - beaconing
00>
00> [00:00:01.074,157] <inf> bt_mesh_main: Device UUID: 6e0cd7a-0f02-00b7-e344-83aece583195
00> Device not yet provisioned so not preparing sensor
00>

LOG: CPUID Register: 0x410c241: Implementer Code: 0x41 (ARM)
LOG: Found Cortex-M4 r0p1, Little endian.
LOG: FPUUnit: 6 code (BP) slots and 2 literal slots
LOG: CoreSight components:
LOG: ROMTbl[0] @ E00FF000
LOG: ROMTbl[0][0]: E000E000, CID: B105E000, PID: 000BB00C SCS-M7
LOG: ROMTbl[0][1]: E0001000, CID: B105E000, PID: 003BB002 DWT
LOG: ROMTbl[0][2]: E0002000, CID: B105E000, PID: 002BB003 FPB
LOG: ROMTbl[0][3]: E0000000, CID: B105E000, PID: 003BB001 ITM
LOG: ROMTbl[0][4]: E0040000, CID: B1059000, PID: 000BB9A1 TPIU
LOG: ROMTbl[0][5]: E0041000, CID: B1059000, PID: 000BB925 ETM
LOG: RTT Viewer connected.

RTT Viewer connected. 0.001 MB
```

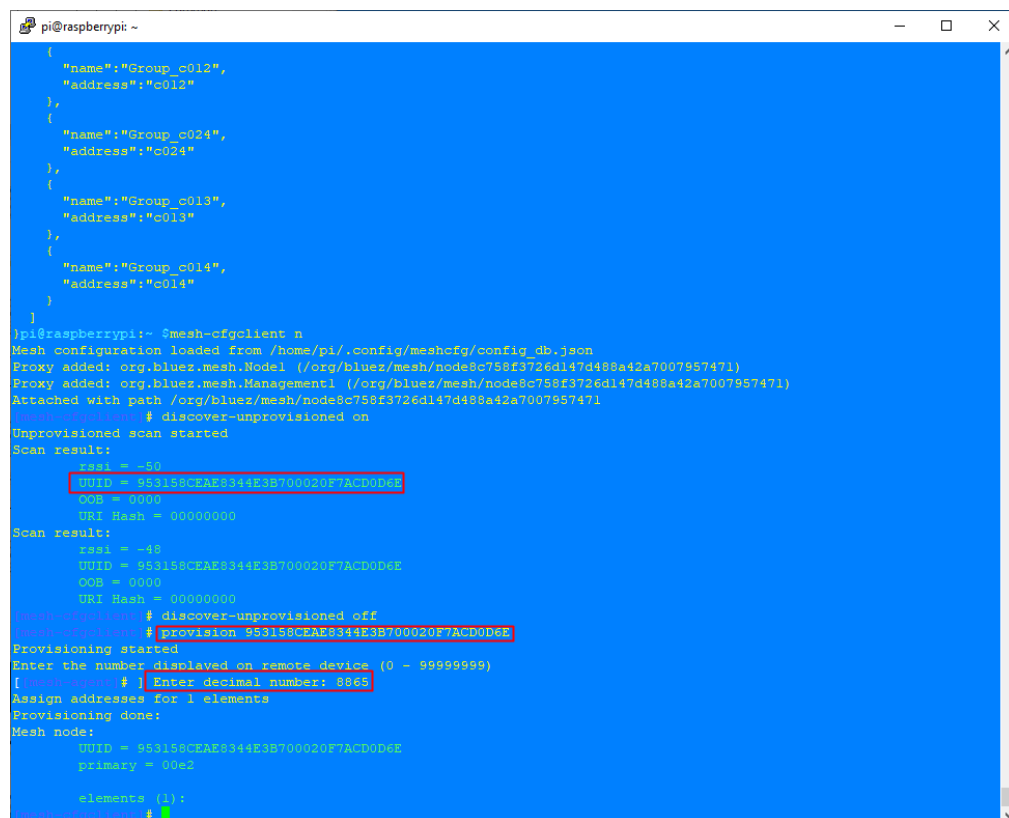
Start mesh-cfgclient and use the commands shown to enable the discovery of unprovisioned devices.

```
pi@raspberrypi:~ $mesh-cfgclient
Mesh configuration loaded from /home/pi/.config/meshcfg/config_db.json
Proxy added: org.bluez.mesh.Node1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Proxy added: org.bluez.mesh.Management1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Attached with path /org/bluez/mesh/node8c758f3726d147d488a42a7007957471
[mesh-cfgclient]# discover-unprovisioned on
Unprovisioned scan started
Scan result:
    rssi = -50
    UUID = 953158CEAE8344E3B700020F7ACD0D6E
    OOB = 0000
    URI Hash = 00000000
Scan result:
    rssi = -48
    UUID = 953158CEAE8344E3B700020F7ACD0D6E
    OOB = 0000
    URI Hash = 00000000
[mesh-cfgclient]# discover-unprovisioned off
```

Figure 7 - Unprovisioned device being discovered

The scan results list beacons from unprovisioned devices. Details include the advertised UUID from each device.

Use the command *provision [UUID]* to initiate provisioning the device. For example to provision the device shown being discovered in Figure 5, execute the command *provision 953158CEAE8344E3B700020F7ACD0D6E*. At the same time, watch the RTT Viewer console. Provisioning involves an authentication step and in this case, you should see a 4 digit PIN displayed in the RTT console. Enter this number into the mesh-cfgclient console when prompted.



```
pi@raspberrypi:~ $mesh-cfgclient n
Mesh configuration loaded from /home/pi/.config/meshcfg/config_db.json
Proxy added: org.bluez.mesh.Node1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Proxy added: org.bluez.mesh.Management1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Attached with path /org/bluez/mesh/node8c758f3726d147d488a42a7007957471
[mesh-cfgclient]# discover-unprovisioned on
Unprovisioned scan started
Scan result:
    rssi = -50
    UUID = 953158CEAE8344E3B700020F7ACD0D6E
    OOB = 0000
    URI Hash = 00000000
Scan result:
    rssi = -48
    UUID = 953158CEAE8344E3B700020F7ACD0D6E
    OOB = 0000
    URI Hash = 00000000
[mesh-cfgclient]# discover-unprovisioned off
[mesh-cfgclient]# provision 953158CEAE8344E3B700020F7ACD0D6E
Provisioning started
Enter the number displayed on remote device (0 - 99999999)
[mesh-cfgclient]# 8865
Assign addresses for 1 elements
Provisioning done:
Mesh node:
    UUID = 953158CEAE8344E3B700020F7ACD0D6E
    primary = 90x2
    elements (1):
```

Figure 8 - A device being provisioned using mesh-cfgclient



The sole element of your new node will have been allocated an address which in Figure 6 is 0x00E2. Your value will almost certainly differ. Make a note of the address as you will need it when completing the configuration steps in the next task.

```

J-Link RTT Viewer V6.88a
File Terminals Input Logging Help
All Terminals Terminal 0
00>
00> Mesh initialised OK
00>
00> [00:00:01.073,944] <inf> bt_hci_core: Identity: cd:cd:d3:95:e8:4e (random)
00> [00:00:01.073,974] <inf> bt_hci_core: HCI: version 5.1 (0x0a) revision 0x0000, manufacturer 0x05f1
00> [00:00:01.073,974] <inf> bt_hci_core: LMP: version 5.1 (0x0a) subver 0xffff
00> Settings loaded
00>
00> Node has not been provisioned - beaconing
00>
00> [00:00:01.074,157] <inf> bt_mesh_main: Device UUID: 6e0dcd7a-0f02-00b7-e344-83aece583195
00> Device not yet provisioned so not preparing sensor
00>
00> attention_on()
00>
00> OOB Number: 8865
00>
00> [00:07:51.532,897] <wrn> bt_mesh_prov: Data for unknown transaction (2 != 0)
00> [00:07:51.732,879] <wrn> bt_mesh_prov: Data for unknown transaction (2 != 0)
00> [00:07:59.230,285] <wrn> bt_mesh_prov: Data for unknown transaction (5 != 0)
00> [00:07:59.638,977] <inf> bt_mesh_main: Primary Element: 0x00e2
00> [00:07:59.638,977] <dbg> bt_mesh_main.bt_mesh_provision: net_idx 0x0000 flags 0x00 iv_index 0x0000
00> [00:07:59.645,904] <dbg> bt_mesh_access.bt_mesh_comp_provision: addr 0x00e2 elem_count 1
00> [00:07:59.645,904] <dbg> bt_mesh_access.bt_mesh_comp_provision: addr 0x00e2 mod_count 6 vnd_mod_count 0
00> [00:07:59.645,904] <dbg> bt_mesh_main.bt_mesh_provision: Storing network information persistently
00> Provisioning completed
00>
00> [00:07:59.830,017] <wrn> bt_mesh_prov: Resending ack
00> attention_off()
00>

LOG: CPUID Register: 0x410c241: Implementer Code: 0x41 (ARM)
LOG: Found Cortex-M4 r0p1, Little endian.
LOG: FPUUnit: 6 code (BP) slots and 2 literal slots
LOG: CoreSight components:
LOG: ROMTbl[0] @ E00FF000
LOG: ROMTbl[0][0]: E000E000, CID: B105E00D, PID: 000BB00C SCS-M7
LOG: ROMTbl[0][1]: E0001000, CID: B105E00D, PID: 003BB002 DWT
LOG: ROMTbl[0][2]: E0002000, CID: B105E00D, PID: 002BB003 FPB
LOG: ROMTbl[0][3]: E0000000, CID: B105E00D, PID: 003BB001 ITM
LOG: ROMTbl[0][4]: E0040000, CID: B105900D, PID: 000BB9A1 TPIU
LOG: ROMTbl[0][5]: E0041000, CID: B105900D, PID: 000BB925 ETM
LOG: RTT Viewer connected.

RTT Viewer connected. 0.002 MB

```

Figure 9 - The device being provisioned

#### 4.5.4.4 Task - Configure a Nordic Thingy

We now need to configure the new node. This involves a number of steps which will:

1. Create and then add an application key
2. Bind the application key to each model
3. Subscribe the generic on off server model and light HSL server model to 3 different destination addresses.
4. Set a publish address for the sensor server model

Use the following examples and explanatory notes as a guide to configuring your node:

```
pi@raspberrypi:~ $ mesh-cfgclient

[mesh-cfgclient]# appkey-create 0 0

[mesh-cfgclient]# menu config

[mesh-cfgclient]# target 00e2

[config: Target = 00e2]# timeout 5

[config: Target = 00e2]# composition-get 0

[config: Target = 00e2]# appkey-add 0

[config: Target = 00e2]# bind 00e2 0 1000
[config: Target = 00e2]# bind 00e2 0 1100
[config: Target = 00e2]# bind 00e2 0 1307

[config: Target = 00e2]# sub-add 00e2 c001 1000
[config: Target = 00e2]# sub-add 00e2 c011 1000
[config: Target = 00e2]# sub-add 00e2 c021 1000

[config: Target = 00e2]# sub-add 00e2 c001 1307
[config: Target = 00e2]# sub-add 00e2 c011 1307
[config: Target = 00e2]# sub-add 00e2 c021 1307

[config: Target = 00e2]# pub-set 00e2 c002 0 0 0 1100
```

Figure 10 - Example configuration commands

### Configuration Notes

- Make sure you use the element address that was allocated to your device when you provisioned it in place of address 00e2 that is being used in Figure 10.
- **appkey-create 0 0** creates a new application key and associates it with the network key with index 0. It assigns application key index 0 to the new key. Note that the network key was created when you first created your network.
- **menu config** switches mesh-cfgclient into the configuration sub-menu.
- **target** specifies a default unicast address for operations.
- **timeout** specifies a timeout for operations in seconds. A slow device like a Raspberry Pi Zero may need this.
- **composition-get 0** retrieves details of elements, models and features from the target device. See Figure 11 and note the model ID values which we use in other commands.
- **appkey-add 0** adds the application key with index 0 to this device.
- **Commands** such as **bind 00e2 0 1000** bind the application key at index 0 to the specified model (with model ID 1000 here) within the element with the given address (here the address is 00e2). So this example binds the new appkey to the generic on off server model in the device's primary element.
- **sub-add**: There are a series of sub-add commands shown in Figure 10. The first three subscribe the generic on off server model to addresses C001, C011 and C021. The final three subscribe the light HSL server model to those same three group addresses. The significance of the addresses is explained below.
- **pub-set** this command configures a publish address of C002 to be used by the sensor server model when publishing status messages. The provided code for the Nordic Thingy has been

designed to check for a publish address associated with the sensor server model and if one is present, start sampling the temperature every 10 seconds and publishing sensor status messages containing the measured temperature. If no publish address associated with the sensor server model is found then the temperature sampling process does not get started. This means you have configuration control over whether or not a device acts as a sensor as well as a coloured light. See Figure 13 for the code that handles this and see the implementation/solutions/mesh/nodes/thingy/Light\_and\_Sensors/src/main.c for the entire source for the test node(s).

- **addresses** - the gateway application that features in section 6 was designed to be used with a grid of 4 x 4 devices. Each device in the grid subscribes to three addresses, one which represents membership of the grid as a whole (C001), one for the column the node is physically placed in (CC021 - C024) and one for the row (C011 - C014). This is illustrated in Figure 10. You don't need 16 devices to complete the project of course. One test node will suffice but if you have more, you'll be able to explore the publish and subscribe capabilities of Bluetooth mesh more.

```
[config: Target = 00e2]# composition-get 0
Received DeviceCompositionStatus (len 27)
Received composion:
  Feature support:
    relay: no
    proxy: yes
    friend: no
    lpn: no
  Element 0:
    location: 0000
    SIG defined models:
      Model ID      0000 "Configuration Server"
      Model ID      0002 "Health Server"
      Model ID      1000 "Generic OnOff Server"
      Model ID      1307 "Light HSL Server"
      Model ID      1100 "Sensor Server"
      Model ID      1101 "Sensor Setup Server"
```

Figure 11 - composition details

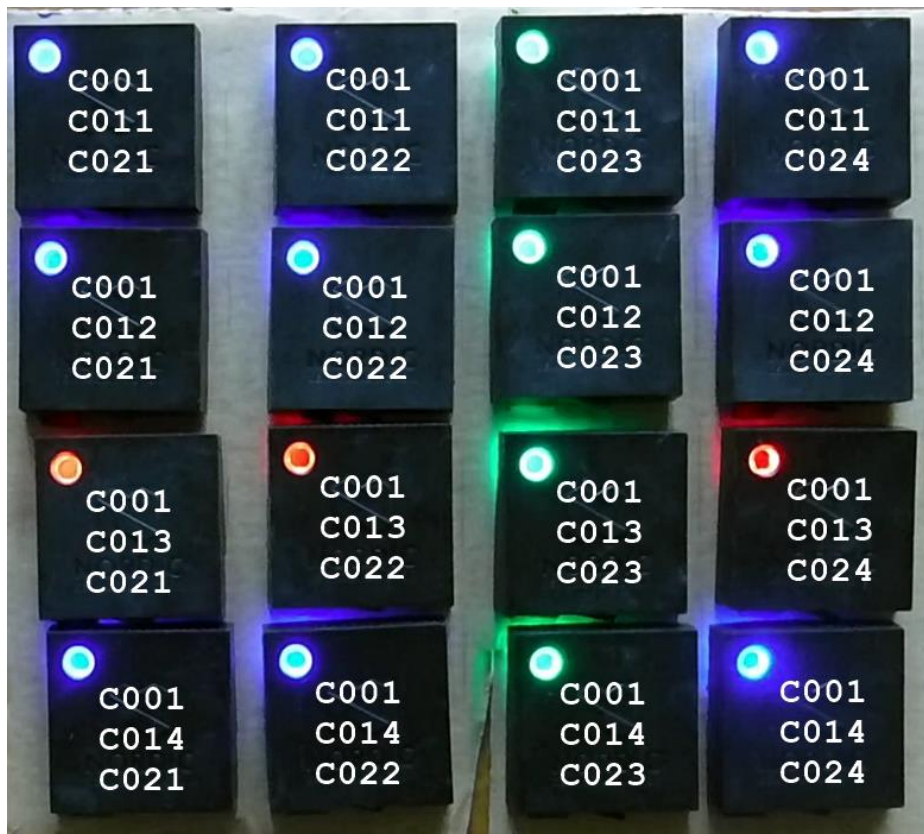


Figure 12 - Subscribe addresses and the Grid of Thingy Nodes

```

    if (provisioned == 1) {
        struct bt_mesh_model *model = &sig_models[SENSOR_SERVER_MODEL];
        publish_address_available = 1;

        if (model->pub->addr == BT_MESH_ADDR_UNASSIGNED) {
            printk("No publish address assigned to the sensor server model -
sensor data will not be published\n");
            publish_address_available = 0;
        }

        if (publish_address_available == 1) {
            printk("Preparing sensor\n");
            configure_hts221_sensor();
            if (sensor == NULL) {
                printk("Could not get HTS221 (sensor) device\n");
            }

            k_delayed_work_init(&temperature_timer, sample_sensor);
            printk("starting sensor sampling\n");
            k_delayed_work_submit(&temperature_timer, K_SECONDS(10));
        }
    } else {
        printk("Device not yet provisioned so not preparing sensor\n");
    }
}

```

Figure 13 - Code checking for a publish address before starting temperature sampling

#### 4.5.5 Coding the HTTP adapter and Bluetooth mesh API components

Developing the adapter code and completing the Bluetooth mesh API code will be approached in two major stages. We'll start with the internet to mesh network (i2mn) path and implement support for our on / off and colour control use cases and progress in a series of small steps. We'll then work on the mesh network for internet path (mn2i) and implement support for our temperature monitoring use case.

We will need to write two Python scripts for each of the two paths through the system. Two will act as the HTTP and websocket adapters respectively and the other two will provide a Bluetooth mesh API for the i2mn uses cases of on/off control and colour control and the mn2i temperature monitoring use case. This is depicted in Figure 14.

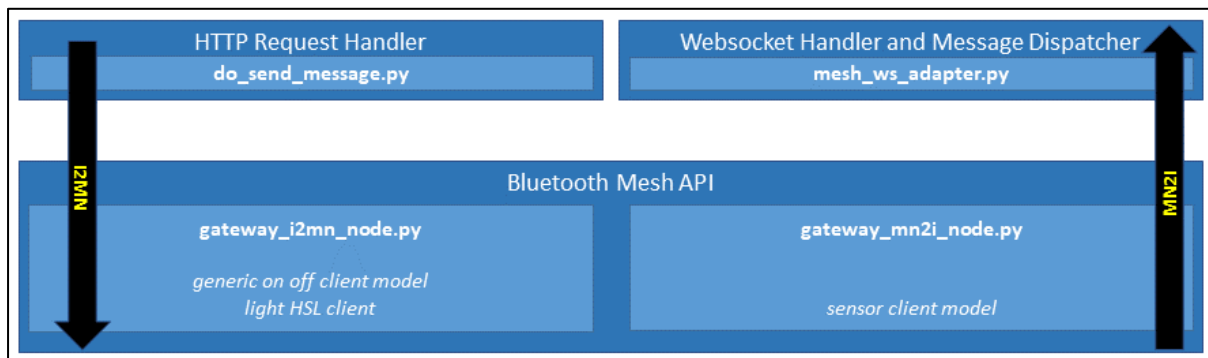


Figure 14 - The four scripts implementing the HTTP adapter and mesh API components

##### 4.5.5.1 Coding the I2MN Path - on / off control

The first is the adapter which handles HTTP requests, validates them and then invokes a function in the Bluetooth API to execute the request, receiving the results in a call back before formulating and sending a response object back to the client. This script does not yet exist and you will create it from scratch.

The second is a script which when run, instantiates a mesh node which implements the models needed to support our two use cases, specifically the *generic on off client model* and the *light HSL client model*. This script exists but is incomplete. Following the guidance below, you will complete it.

To begin with, we'll focus only on the switch on/off use case with the gateway implementing a *generic on off client model*.

##### 4.5.5.1.1 Approach

You will now proceed to develop the gateway adapter scripts that will service requests received over HTTP from the Apache web server.

We'll proceed one use case at a time and will generally be concerned with extracting and validating parameters, executing one or more calls to our Python Bluetooth mesh API, and transforming the results into a JSON object to be sent back to the HTTP client which submitted the original request.

For some of you, this may be the first time you have programmed in Python, so we'll take it very slowly to begin with.

There are numerous Python tutorials on the internet, including this one:

<https://docs.python.org/3/tutorial/index.html>

Initially you will test against an incomplete implementation of the Bluetooth mesh API which has stub implementations of the functions needed by the HTTP adapter script. Once the HTTP adapter is working when tested from an HTTP client (command line or browser), we'll complete the Bluetooth API code for the I2MN path.

#### 4.5.5.1.2 Preparation

If you previously developed a gateway for LE Peripherals following module 03 on this device, first backup your work as we will be reusing some of the same directories for the mesh gateway code.

Create a directory for the gateway adapter code and set its ownership and permissions so that it can be used by both the Apache web server and you, logged into the default *pi* account.

```
pi@raspberrypi:~ $ sudo mkdir /usr/lib/cgi-bin/gateway
pi@raspberrypi:~ $ sudo chown www-data:www-data /usr/lib/cgi-bin/gateway/
pi@raspberrypi:~ $ sudo chmod 775 /usr/lib/cgi-bin/gateway/
```

It will be assumed from now on that all files created in this directory will be given the same ownership and permissions.

Copy the file **constants.py** from study guide folder implementation/start\_state/mesh/common to your raspberry Pi's new directory /usr/lib/cgi-bin/gateway.

Copy the files **gateway\_i2mn\_node.py** and **i2mn\_config.json** from study guide folder implementation/start\_state/mesh/bluetooth\_mesh\_api to your raspberry Pi's new directory /usr/lib/cgi-bin/gateway.

Your cgi-bin/gateway folder should look like this:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ls -l
total 24
-rwxrwxr-x 1 www-data www-data  936 May 26 15:15 constants.py
-rwxrwxr-x 1 www-data www-data 17294 Jun  9 14:23 gateway_i2mn_node.py
-rwxrwxr-x 1 www-data www-data   29 Jun  9 14:23 i2mn_config.json
```

Tip: Watch out for Windows carriage return/line feed characters, which may be visible in your source when you open it in an editor on your Raspberry Pi. Eliminate them by running the **dos2unix** command like this:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ sudo dos2unix gateway_i2mn_node.py
dos2unix: converting file gateway_i2mn_node.py to Unix format...
```

You'll need to install **dos2unix** if it is not present on your system with **sudo apt-get install dos2unix**

#### 4.5.5.1.3 Task - Implement the common HTTP Handler Code

##### 4.5.5.1.3.1 Coding

All HTTP requests will be handled by the same script. Depending on the requested action, a particular mesh message type will be sent using the Bluetooth mesh API (which at this stage only contains stub implementations of the required functions).

Start by creating a file in /usr/lib/cgi-bin/gateway called **do\_send\_mesh\_message.py** and using the editor of your choice, add the following content to it:

```
#!/usr/bin/env python3
#
# Invoke over HTTP with a suitable JSON object containing args which select and inform the
# mesh message type to be sent by the gateway_i2mn_node script.
#
#####

import sys
sys.path.insert(0, '.')
import json
import os
import sys
import cgi
import constants
import gateway_i2mn_node
```

To help with troubleshooting, we'll include a very simple logging system which you can remove later on when everything is working. Add the following code:

```
fo = open("http_log.txt", "a")

def log(line):
    fo.write(line+"\n")
    fo.flush()
```

All the log function does is to append a provided line of text to a file called http\_log.txt.

Declare and initialise some variables which will be used in the main request handling code:

```
dst_addr = None
state = None
rc = None
result = {}
```

The functions that the Bluetooth mesh API will provide each use a callback pattern to inform the HTTP request handler (our current script) of the outcome of calls made to those functions. Add this function to act as a general callback function to our script now:

```
def result_cb(rc):
    result['result'] = rc
    print(json.JSONEncoder().encode(result))
```

The result\_cb function gets passed a result code (rc) by the Bluetooth mesh API and values for result codes are defined in constants.py. It then adds the result code to a JSON object and outputs it to the HTTP client by printing to standard output, which due to the way CGI works, results in the object being written over the TCP/IP connection between the Apache Web Server and the HTTP client.

Next, create a nested IF construct which checks that the script was invoked by HTTP (as opposed to the command line), uses the HTTP PUT method and then outputs the required Content-Type HTTP header:

```
if 'REQUEST_METHOD' in os.environ:

    if (os.environ['REQUEST_METHOD'] != 'PUT'):
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
    else:
        log("PUT")
        print("Content-Type: application/json;charset=utf-8")
        print()
else:
    print("ERROR: Not called by HTTP")
```

Set ownership and permission of the new file:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ sudo chown www-data:www-data *
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ sudo chmod 775 *
```

Tip: you'll probably need to set and reset file ownership and permissions several times while developing and testing. So put these commands in a script in your `/usr/lib/cgi-bin/gateway` folder to give yourself a quick and easy way of accomplishing this.

Like this one:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ cat permset.sh
#!/bin/bash
sudo chown www-data:www-data *
sudo chmod 775 *
```

#### 4.5.5.1.3.2 Testing

Test the basic conditions in the script from the command line. First check that it cannot be run from the command line itself, only invoked over HTTP via CGI.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ./do_send_mesh_message.py
ERROR: Not called by HTTP
```

If your test produced no output, you probably have errors in your script, or maybe it is in the wrong directory or has the wrong permissions. Look at the end of `/var/log/apache2/error.log` for clues.

Tip: Python has very strict [rules about indentation](#).

Execute the following tests to check that the validation of the HTTP method.

The only acceptable HTTP method for this script is PUT since both of the use cases it will support involve changing a model's state value.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
request GET http://localhost/cgi-bin/gateway/do_send_mesh_message.py
Status-Line: HTTP/1.0 405 Method Not Allowed
```

The same URL accessed using PUT should work and evidence for this will be visible in the `http_log.txt` file:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py

pi@raspberrypi:/usr/lib/cgi-bin/gateway $ cat http_log.txt
-----
PUT
```

#### 4.5.5.1.4 Task - Implement the on/off control HTTP request

##### 4.5.5.1.4.1 API Details

Request
<code>do_send_mesh_message.py</code>
HTTP Method
PUT
Arguments Object
<pre>// switch on lights that subscribe to group address C001 {   "action":"generic_onoff_set_unack",</pre>



```

"dst_addr":"C001",
"state":"01"
}

// switch off lights that subscribe to group address C001
{
  "action":"generic_onoff_set_unack",
  "dst_addr":"C001",
  "state":"00"
}

```

### Response Description

Returns a JSON object containing a result code.

### Response Example - Successful Execution

```
{"result": 0}
```

### Response Example - Error - Invalid Args

```
{"result": 5}
```

### Response Example - Error - Attach Failed

```
{"result": 6}
```

#### 4.5.5.1.4.2 Coding

Edit `do_send_mesh_message.py`. Change your IF statement so it includes the new code (highlighted in green) provided here:

```

if 'REQUEST_METHOD' in os.environ:

    if (os.environ['REQUEST_METHOD'] != 'PUT'):
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
    else:
        log("PUT")
        print("Content-Type: application/json;charset=utf-8")
        print()
        args = json.load(sys.stdin)
        if (not "action" in args):
            result['result'] = constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif (args["action"] != "generic_onoff_set_unack" and args["action"] !=
constants.ACTION_LIGHT_HSL_SET_UNACK):
            result['result'] = constants.RESULT_ERR_NOT_SUPPORTED
            print(json.JSONEncoder().encode(result))
        elif (args["action"] == constants.ACTION_GENERIC_ON_OFF_SET_UNACK):
            log("generic on off set unack action")
            result['result'] = constants.RESULT_OK
            print(json.JSONEncoder().encode(result))
        elif (args["action"] == constants.ACTION_LIGHT_HSL_SET_UNACK):
            log("light HSL set unack action")
            result['result'] = constants.RESULT_OK
            print(json.JSONEncoder().encode(result))
        else:
            result['result'] = constants.RESULT_ERR_NOT_SUPPORTED
            print(json.JSONEncoder().encode(result))

```

After checking that this is an HTTP PUT, the JSON arguments object is parsed with the result stored in a Python dictionary variable called `args`. It is then validated, checking that it contains a property called “action” and that action has a value of either “generic\_onoff\_set\_unack” or “light\_hsl\_set\_unack”. These are the names of the two mesh message types that our gateway will allow clients to request are sent by the gateway. The current task concerns the first only, the *generic on off set unacknowledged* message type which is used to switch the Thingy LEDs on or off.

Any validation errors cause a non-zero result code to be returned in the JSON response object to the client. If the request is valid and the action is supported, the action name will be logged and a result with code zero (success) returned to the client.

Test with curl from the command line like this:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --  
data '{"action": "generic_onoff_set_unack", "dst_addr": "C001", "state": "01" }' --  
request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py  
  
{"result": 0}
```

The foot of your http\_log.txt file should indicate that the generic\_onoff\_set\_unack action was recognised in the request.

```
PUT  
generic on off set unack action
```

You now have a partial implementation of the HTTP adapter script, do\_send\_mesh\_message.py with a clear structure into which we will add more code to complete the implementation soon. You will now turn your attention to the Bluetooth mesh API script, gateway\_i2mn\_node.py which is substantially incomplete at this stage.

#### 4.5.5.1.5 Task - Familiarisation with the gateway\_i2mn\_node.py script

Open gateway\_i2mn\_node.py in an editor and scroll through it. Much of this code has its origins in a script which is packaged with BlueZ, test/test-mesh.

Some key points to note about this script are highlighted next.

The script defines a number of important classes as shown here:

```
class Agent(dbus.service.Object):  
    def __init__(self, bus):  
        log("Constructing Agent")  
  
class Application(dbus.service.Object):  
  
    def __init__(self, bus):  
        log("Constructing Application")  
  
class Element(dbus.service.Object):  
    PATH_BASE = '/mesh/gateway/ele'  
  
    def __init__(self, bus, index):  
        log("Constructing Element")  
  
class Model():  
    def __init__(self, model_id):  
        log("Constructing Model")
```

Application, Element and Model are part of the node composition. Agent is a class that is involved in provisioning. You'll also notice lines like this one:

```
@dbus.service.method(AGENT_IFACE, in_signature="", out_signature="")
```

This is an annotation which exports a method or function as part of a named interface to D-Bus so that it can be used in interprocess communication. This relates to the interfaces described in section 4.4.3.

The important point to note is this. Code which implements the application which creates a node, with its elements and models and ability to be provisioned must have the right constituent parts and

interface. Failure to include the expected components and expose the expected interface will result in your mesh node not working when you try to attach it to the bluetooth-meshd daemon.

The client models that the gateway needs to support for the i2mn path are each implemented as classes which at present are incomplete. Look for class OnOffClient(Model) and class LightHslClient(Model) in the code and note the TODO comments. Your task will be to complete the code required to formulate and send a mesh message of the required type.

Scroll to the end of the file and notice the final two TODO comments and the finished() function.

```
# TODO implement getting a token from the pool

# TODO Join from command line

def finished():
    global exit
    log("finished")
    exit.set()
```

We're going to implement a simple pooling mechanism for token values which will be explained further below. It will need to be possible to provision the node application (multiple times) and we'll do that by executing the script directly from the command line.

We also need to make sure that regardless of the outcome of an operation, we finish processing correctly and the finish() function acts as a common function called for this purpose, working in conjunction with a timeout function which we will be meeting later on.

#### 4.5.5.1.6 Task - Provisioning the i2mn gateway node application

In order that your node application, the gateway\_i2mn\_node.py script is recognised as a bona fide node in your mesh network it must be provisioned. To accomplish this, we need the script to execute a call to a mesh API function called Join. On completion, a callback will be made to one of two functions depending on whether the operation was successful or not. During the course of executing the provisioning or *join* operation, the Agent will participate and may for example, display an authentication value which needs to be entered into the Provisioner (in our case the Provisioner is the command line tool mesh-cfgclient).

We will run the script in "provisioning mode" like this:

```
./gateway_i2mn_node.py join
```

The "join" value is an argument that indicates we wish to join the network i.e. provision this node application.

#### 4.5.5.1.6.1 Coding

At the very end of your gateway\_i2mn\_node.py script, add this code after the TODO comment:

```
# TODO implement JOIN command line function

if __name__ == '__main__':
    if (len(sys.argv) != 2):
        print("Error: incorrect number of arguments specified")
        sys.exit(1)

    cmd = sys.argv[1]
    if (cmd == 'join'):
        join()
    else:
```

```
print("Unrecognised option - join is the only argument supported over the  
command line")
```

This checks for the script being run from the command line using a special Python system variable called `__name__`. It then checks for the expected number of arguments if it was. Note that the first argument (`sys.argv[0]`) is always the name of the script.

If an argument of “join” has been provided, we’ll execute a function of that name.

Under the comment “# TODO Join from command line” add the following code:

```
def join_cb():  
    log('.....')  
  
def join_error_cb(reason):  
    log('Join procedure failed: ', reason)  
  
def join():  
    initNode()  
    global app  
    global mesh_net  
    global mainloop  
    uuid_bytes = uuid.uuid4().bytes  
    uuid_str = array_to_string(uuid_bytes)  
  
    log("Requesting to join network with UUID "+uuid_str)  
    mesh_net.Join(app.get_path(), uuid_bytes,  
                  reply_handler=join_cb,  
                  error_handler=join_error_cb)  
    mainloop = GLib.MainLoop()  
    mainloop.run()
```

The first two functions are callback functions, the first for a positive outcome and the second for when an error occurred. The `join()` function itself first calls a function `initNode()` which per the name performs some important initialisation, constructing the node from its parts (elements and models), connecting to the D-Bus system bus and obtaining a reference to the BlueZ mesh service which implements the API.

The provisioning process requires an unprovisioned device (or node application in our case) to identify itself with a UUID in unprovisioned device beacons. We generate a UUID and then call the mesh API function `Join`, specifying the UUID as a byte array and the two callback functions. The `mesh_net` object is an interface to the D-Bus BlueZ mesh service which implements the mesh API. This gets set up in the `initNode` function.

We finish the `join()` function by starting the Glib event loop. This allows callbacks to be received.

#### 4.5.5.1.6.2 Testing

Test command line argument validation with the following (at least)

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ./gateway_i2mn_node.py  
Error: incorrect number of arguments specified
```

In a second SSH terminal, tail the file `i2mn_log.txt`. Now run `./gateway_i2mn_node.py join`

The log file being tailed will have the following lines added to it:

```
Constructing Application  
Constructing Agent  
Constructing Element  
Constructing OnOffClient  
Constructing Model  
Constructing LightHslClient  
Constructing Model
```

```
Requesting to join network with UUID 25948d51758d4b0a8cbceab38ad575c8  
Agent: get_properties  
Agent: get_path
```

Note the generated UUID.

Start an SSH session on your Raspberry Pi that acts as your Provisioner. Make sure bluetooth-meshd is running and then execute mesh-cfgclient. Mimic the session captured next, entering commands shown in bold:

```
pi@raspberrypi:~ $ mesh-cfgclient  
Mesh configuration loaded from /home/pi/.config/meshcfg/config_db.json  
Proxy added: org.bluez.mesh.Node1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)  
Proxy added: org.bluez.mesh.Management1  
(/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)  
Attached with path /org/bluez/mesh/node8c758f3726d147d488a42a7007957471  
  
[mesh-cfgclient]# discover-unprovisioned on  
Unprovisioned scan started  
Scan result:  
    rssi = -63  
    UUID = 25948D51758D4B0A8CBCEAB38AD575C8  
    OOB = 0001  
  
[mesh-cfgclient]# discover-unprovisioned off  
  
[mesh-cfgclient]# provision 25948D51758D4B0A8CBCEAB38AD575C8  
Provisioning started  
Request hexadecimal key (hex 16 octets)  
[[mesh-agent]# ] Enter key (hex number): d886e8dd33fb51cef8d3d279fc8fc422  
Assign addresses for 1 elements  
Provisioning done:  
Mesh node:  
    UUID = 25948D51758D4B0A8CBCEAB38AD575C8  
    primary = 00e3  
  
    elements (1):
```

## Notes

discover-unprovisioned on	Starts scanning for unprovisioned device beacons. Any discovered are reported as a Scan Result and each Scan Result includes the UUID of the beaconing device. You should see the UUID of your node application that was reported on the other Pi in the i2mn_log.txt file.
discover-unprovisioned off	Stops scanning for unprovisioned device beacons.
provision [UUID]	Initiates provisioning of the device identified by the specified UUID.

After a delay, the gateway\_i2mn\_node.py script will output a 16 octet authentication value like this:

```
PromptStatic ( static-oob )  
Enter 16 octet key on remote device: d886e8dd33fb51cef8d3d279fc8fc422
```

At the same time, the Provisioner will output a prompt requesting that you enter the authentication value and you should do so like this:

```
Request hexadecimal key (hex 16 octets)
[[mesh-agent]# ] Enter key (hex number): d886e8dd33fb51cef8d3d279fc8fc422
```

If everything works, the Provisioner will output something like this:

```
Assign addresses for 1 elements
Provisioning done:
Mesh node:
    UUID = 25948D51758D4B0A8CBCEAB38AD575C8
    primary = 00e3

    elements (1):
```

and the application node will output the token value that was assigned to the application.

```
Node provisioned OK - token=0x5d54b6aff549a36a
```

Your i2mn application has been provisioned. Make a note of the allocated token value now and do not lose it. If you do, you will need to provision the application again from scratch. Also note the element primary address that was assigned (00e3 in the example above).

#### 4.5.5.1.7 Task - Configuring the gateway node application

The provisioned node application must also be configured so that it can function as required as a node with its two client models. Using Table 2 as a guide, configure your node application so that it has an application key which is bound to both client models owned by the primary element. Make sure you get a positive response to all commands. If you get no response, just submit the command again.

**Table 2 - Configuring the i2mn node application**

```
[mesh-cfgclient]# target 00e3
Configuring node 00e3

[config: Target = 00e3]# timeout 5
Timeout to wait for remote node's response: 5 secs

[config: Target = 00e3]# composition-get 0
Received DeviceCompositionStatus (len 21)
Received composion:
    Feature support:
        relay: yes
        proxy: no
        friend: yes
        lpn: no
    Element 0:
        location: 0000
        SIG defined models:
            Model ID      0000 "Configuration Server"
            Model ID      1001 "Generic OnOff Client"
            Model ID      1309 "Light HSL Client"

[config: Target = 00e3]# appkey-add 0
Received AppKeyStatus (len 4)
Node 00e3 AppKey status Success
NetKey  0 (0x000)
AppKey  0 (0x000)

[config: Target = 00e3]# bind 00e3 0 1001
Received ModelAppStatus (len 7)
Node 00e3: Model App status Success
Element Addr      00e3
Model ID          1001 "Generic OnOff Client"
AppIdx            0 (0x000)

[config: Target = 00e3]# bind 00e3 0 1309
```

```
No response for "ModelAppBind" from 00e3
```

```
[config: Target = 00e3]# bind 00e3 0 1309
Received ModelAppStatus (len 7)
Node 00e3: Model App status Success
Element Addr    00e3
Model ID        1309 "Light HSL Client"
AppIdx          0 (0x000)
```

#### 4.5.5.1.8 Task - Using the assigned token value

When the i2mn node application needs to attach to the mesh network it will need to present the token it was allocated when provisioned. We'll implement this in a function and will make the mechanism for allocating a token value more sophisticated later on. For now, using the value that was allocated to your gateway i2mn node application when you provisioned it, add the `getToken()` function to your code like this:

```
# TODO implement getting a token from the pool

def getToken():
    return "5d54b6aff549a36a"
```

#### 4.5.5.1.9 Task - Implement the `gateway_i2mn_node.send_onoff` function

The `do_send_mesh_message.py` HTTP adapter needs an API function it can call to instruct the i2mn node application to formulate and send a *generic on off set unack message* with a specified *generic on off state* value to a specified destination address. On completion, the API function must inform the HTTP adapter of the outcome. As you will come to appreciate, most of what you will do with D-Bus and BlueZ involves callbacks and we'll use that approach here as well. The function we need to implement will have this signature therefore:

```
def send_onoff(dst_addr_str, state_str, cb_function):
```

To send a message, the i2mn node application will need to *attach* to the mesh network using a token value, then send the message. At each of these stages, the outcome might be successful or an error. Sending a message involves several of the classes and objects in the i2mn script. Before we get into the code, have a look at Figure 15.

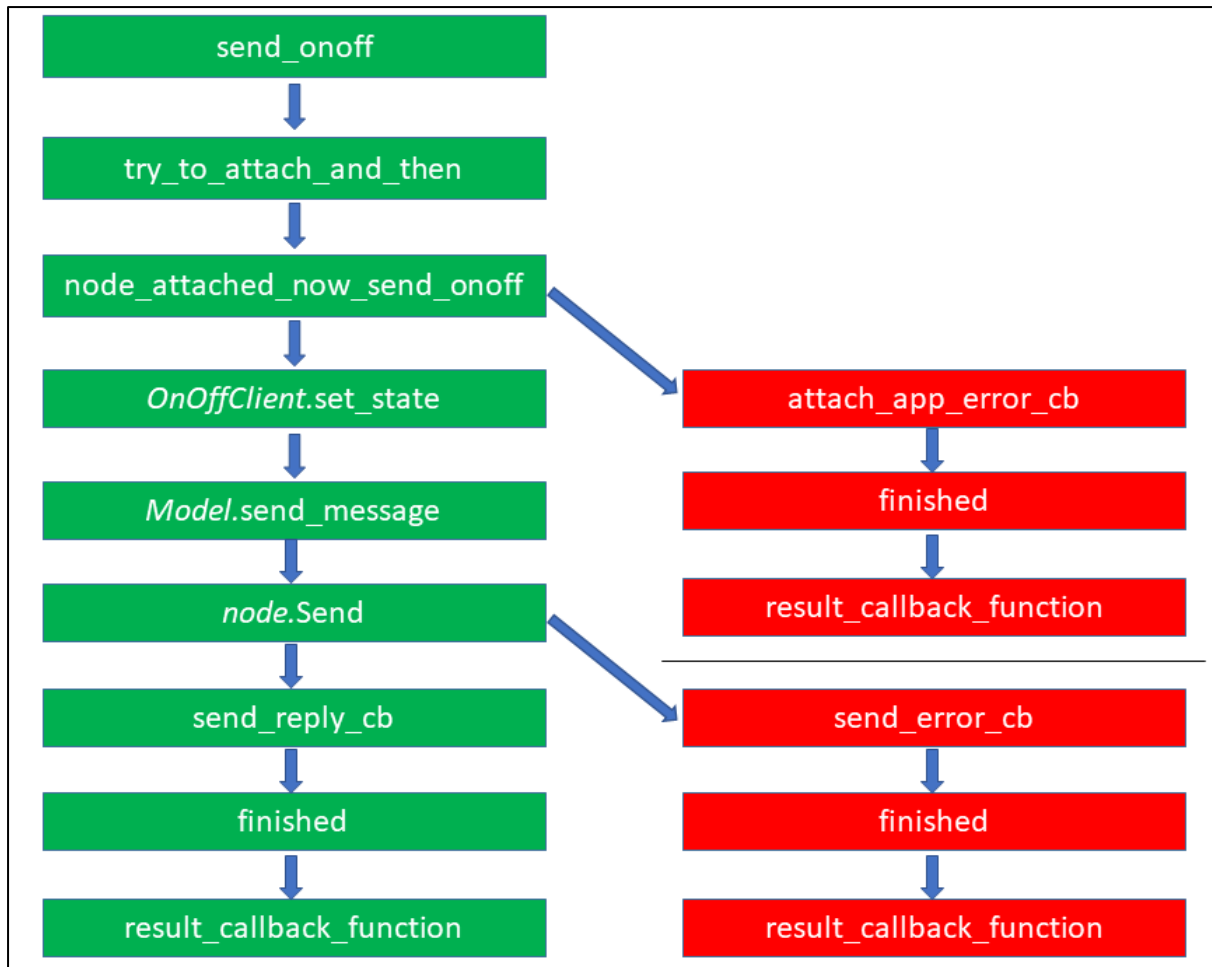


Figure 15 - Flow of execution sending an on off set message

One point to note is that regardless of the final outcome, we always call the finished() function and then the callback function that was supplied as an argument to the initial send\_onoff function so that the called (the HTTP handler do\_send\_mesh\_message.py) always gets called with the result.

Proceed as follows. Add the following function to gateway\_i2mn\_node.py.

```

# TODO implement the send_onoff API function
def send_onoff(dst_addr_str, state_str, cb_function):
    t = threading.Thread(target=timeout)
    t.start()
    global result_callback_function
    global dst_addr
    global state
    initNode()
    dst_addr_int = int(dst_addr_str, 16)
    dst_addr = numpy.uint16(dst_addr_int)
    state = numpy.uint8(int(state_str))
    result_callback_function = cb_function
    log(constants.ACTION_GENERIC_ON_OFF_SET_UNACK+ " action starting")
    try_to_attach_and_then(node_attached_now_send_onoff)
    return constants.RESULT_OK
  
```

This code:

- Starts by kicking off a timeout detection function in a background thread. Have a look at the timeout() function and exit Event object to see how this works.



- Calls `initNode`. This acquires a reference to the mesh D-Bus service which implements the BlueZ mesh API functions. It then constructs the node application and the parts of its node composition and obtains a reference to the Glib event loop.
- Converts the destination address and state arguments, which have been passed as hex strings, into 16 and 8 bit unsigned int values respectively.
- Stores the provided callback function in `result_callback_function` (see Figure 15).
- Initiates attaching to the bluetooth-meshtd daemon by calling `try_to_attach_and_then` with the function which must be executed after successfully attaching, as an argument.
- Returns a result code meaning OK to the caller. This just means the request was successfully accepted. The final outcome will be communicated in a call to the specified callback function.

`try_to_attach_and_then(..)` has already been implemented and looks like this:

```
def try_to_attach_and_then(next):
    global mesh_net
    global app
    global token_inx
    global next_action
    next_action = next
    token = getToken()
    tid = str(threading.current_thread().ident)
    log(tid+" attempting to attach with "+token)
    token_uint64 = numpy.uint64(int(token, 16))
    log(tid+" attempting to attach NOW: app="+str(app)+" mesh_net="+str(mesh_net))
    mesh_net.Attach(app.get_path(), token_uint64,
                    reply_handler=next,
                    error_handler=attach_app_error_cb)
    if (token_inx == 0):
        log(tid+" starting mainloop")
        mainloop.run()
        log(tid+" mainloop no longer running")
```

Notice that a token value is acquired by calling the `getToken()` function and that after converting it to a `uint64` from its hex format, this is used in a call to the BlueZ mesh API function `Attach`. Don't worry about `token_inx` for now.

The *next* argument is the function which must be called after successfully attaching. In our current use case, that's the function *node\_attached\_now\_send\_onoff*.

The function *node\_attached\_now\_send\_onoff* is already in your starter code and looks like this:

```
def node_attached_now_send_onoff(node_path, dict_array):
    log("node_attached_now_send_onoff")
    global dst_addr
    global app_idx
    global state

    obj = bus.get_object(MESH_SERVICE_NAME, node_path)
    global node
    node = dbus.Interface(obj, MESH_NODE_IFACE)
    log("sending to "+str(dst_addr))
    app.elements[PRIMARY].models[GENERIC_ON_OFF_CLIENT].set_state(dst_addr, app_idx,
state)
```

This code:

- Obtains a D-Bus proxy object for the node and assigns it to a global for later use

- Calls the `set_state` method of the generic on off client model with the specified destination address, application key index and state value as arguments.

In the `OnOffClient` class, we now need to complete the `set_state` function so that it looks like this:

```
def set_state(self, dest, key, state):
    log("set_state dest="+str(dest)+" state="+str(state)+"\n")
    opcode = 0x8203
    self.data = struct.pack('>HBB', opcode, state, self.tid)
    self.tid = (self.tid + 1) % 255
    self._send_message(dest, key, self.data)
```

The opcode value 0x8203 is the specified identifier for the generic on off set unacknowledged message type and you'll find this in the Bluetooth SIG Mesh Model specification.

Server Model	Message Name	Opcode
Generic OnOff	Generic OnOff Get	0x82 0x01
	Generic OnOff Set	0x82 0x02
	Generic OnOff Set Unacknowledged	0x82 0x03
	Generic OnOff Status	0x82 0x04

The call to `struct.pack` involves a format specifier which ensures that the variable assigned to has the opcode, state and mesh transaction ID (tid) data using the right byte ordering and byte lengths. See <https://docs.python.org/3/library/struct.html> for more information on the Python struct module.

#### 4.5.5.1.10 Task - Call the `i2mn` API function from the HTTP handler script

Everything should now be in place to allow the HTTP handler script to invoke the `send_onoff` `i2mn` mesh API function and for a message to be sent into the mesh network.

##### 4.5.5.1.10.1 Coding

Update your `do_send_mesh_message.py` code so that the block dealing with *generic on off set unack* actions looks like this:

```
elif (args["action"] == constants.ACTION_GENERIC_ON_OFF_SET_UNACK):
    log("generic on off set unack action")
    if (not "state" in args or not "dst_addr" in args):
        result['result'] = constants.RESULT_ERR_BAD_ARGS
        print(json.JSONEncoder().encode(result))
    else:
        action = args["action"]
        log(constants.ACTION_GENERIC_ON_OFF_SET_UNACK+ " preparing
arguments")

        dst_addr = args["dst_addr"]
        state = args["state"]
        result['dst_addr'] = dst_addr
        result['state'] = state
        log("calling gateway_i2mn_node.send_onoff()")
        rc = gateway_i2mn_node.send_onoff(dst_addr, state, result_cb)
        result['result'] = rc
        if (rc != constants.RESULT_OK):
            print(json.JSONEncoder().encode(result))
```

This code:

- Checks that the arguments which are required for this action are present in the JSON object received from the client.

- Calls the `send_onoff` function in `gateway_i2mn_node.py` using the argument values received from the client. The function `result_cb` is provided for use as a callback function.

#### 4.5.5.1.10.2 Testing

Execute the following tests from the command line. Make sure that your test node (Nordic Thingy) is powered on and has been provisioned and configured per earlier sections.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "generic_onoff_set_unack", "dst_addr": "C001", "state": "01"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"dst_addr": "C001", "state": "01", "result": 0}
```

The LED in your test node should switch on.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "generic_onoff_set_unack", "dst_addr": "C001", "state": "00"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"dst_addr": "C001", "state": "00", "result": 0}
```

The LED in your test node should switch off.

#### 4.5.5.1.11 Task Completed

Congratulations! If you've made it this far, then hopefully this means you have a gateway which lets you switch your test node(s) on and off using HTTP.

If you are having a few problems, check your work carefully against the instructions in this section and don't forget, the full solution is available in `implementation/solutions/mesh`. The Troubleshooting information in section 9 might also help.

#### 4.5.5.2 Coding the I2MN Path - colour control

We'll now turn our attention to the second of the i2mn use cases; colour control. The Nordic Thingy has a coloured LED and implements the Bluetooth mesh *light HSL server model*. HSL stands for Hue Saturation Lightness and is one of the commonly used colour representation schemes. The *light HSL server model* allows mesh clients to set the colour used by an element of a node by sending messages such as *light HSL set unacknowledged*, which our gateway will use in response to corresponding HTTP requests.

The general pattern we'll use will be recognisable from implementing the on / off control use case. The HTTP client will send a JSON object requesting that a *light HSL set unacknowledged* message be sent, together with required parameter values. The HTTP handler `do_send_mesh_message.py` will need some logic with which to validate and process these requests. The `gateway_i2mn_node.py` script will need a function which the HTTP handler can call and its execution flow will be similar to that of Figure 15 except of course that you will ultimately be invoking `set_state` on the `LightHslClient` class rather than the `OnOffClient` class. As such, you will be given a little less detailed direction for this task.

#### 4.5.5.2.1 Task - Implement the colour control HTTP request

##### 4.5.5.2.1.1 API Details

Request
<code>do_send_mesh_message.py</code>
HTTP Method
PUT
Arguments Object
// Example: set the colour of lights that subscribe to group address C001 to red { "action": "light_hsl_set_unack",

```

"dst_addr": "C001",
"h_state": "0000",
"s_state": "FFFF",
"l_state": "7FFF"
}

```

### Response Description

Returns a JSON object containing a result code.

### Response Example - Successful Execution

```

{"result": 0}

```

### Response Example - Error - Invalid Args

```

{"result": 5}

```

### Response Example - Error - Attach Failed

```

{"result": 6}

```

#### 4.5.5.2.1.2 Coding

Your `do_send_mesh_message.py` script currently includes this fragment:

```

elif (args["action"] == constants.ACTION_LIGHT_HSL_SET_UNACK):
    log("light HSL set unack action")
    result['result'] = constants.RESULT_OK
    print(json.JSONEncoder().encode(result))

```

Given the API details above including the example JSON object and considering the code in the equivalent code block which handles `ACTION_GENERIC_ON_OFF_SET_UNACK`, update the block of code which handles `ACTION_LIGHT_HSL_SET_UNACK` so that the presence of the expected attributes in the JSON object is checked and ultimately, a call to `gateway_i2mn_node.send_light_hsl_set(dst_addr, h_state, s_state, l_state, result_cb)` is made. We have not yet implemented this function of course.

Your updated code should look something like this:

```

elif (args["action"] == constants.ACTION_LIGHT_HSL_SET_UNACK):
    log("light HSL set unack action")
    if (not "h_state" in args or not "s_state" in args or not "l_state"
in args or not "dst_addr" in args):
        result['result'] = constants.RESULT_ERR_BAD_ARGS
        print(json.JSONEncoder().encode(result))
    else:
        action = args["action"]
        log(constants.ACTION_LIGHT_HSL_SET_UNACK+ " preparing
arguments")

        dst_addr = args["dst_addr"]
        h_state = args["h_state"]
        s_state = args["s_state"]
        l_state = args["l_state"]
        result['dst_addr'] = dst_addr
        result['h_state'] = h_state
        result['s_state'] = s_state
        result['l_state'] = l_state
        log("calling gateway_i2mn_node.send_light_hsl_set()")
        rc = gateway_i2mn_node.send_light_hsl_set(dst_addr, h_state,
s_state, l_state, result_cb)
        result['result'] = rc
        if (rc != constants.RESULT_OK):
            print(json.JSONEncoder().encode(result))

```

Temporarily comment out the call to `send_light_hsl_set` and hard code the assignment

```

rc = constants.RESULT_OK

```

so that some basic testing can be carried out before we move on to the gateway\_i2mn\_node.py changes.

#### 4.5.5.2.1.3 Testing

Perform the following simple tests from the command line:

```
# missing l_state
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "light_hsl_set_unack", "dst_addr": "C001", "h_state": "0000", "s_state": "FFFF"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"result": 8}

# missing s_state
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "light_hsl_set_unack", "dst_addr": "C001", "h_state": "0000", "l_state": "7FFF"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"result": 8}

# missing h_state
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "light_hsl_set_unack", "dst_addr": "C001", "s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"result": 8}

# missing dst_addr
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "light_hsl_set_unack", "h_state": "0000", "s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"result": 8}
```

Reinstate the call to `send_light_hsl_set` and remove the hard coded assignment to `rc` and continue.

#### 4.5.5.2.2 Task - Implement the gateway\_i2mn\_node.send\_light\_hsl\_set function

The gateway\_i2mn\_node.py script needs to be changed so that it includes the function `send_light_hsl_set` which the HTTP adapter script will call. In terms of structure the complete set of code required is similar to that which services on/off set requests. Required functions and the possible paths through them are depicted in Figure 16. Some of these functions already exist in your starter code.

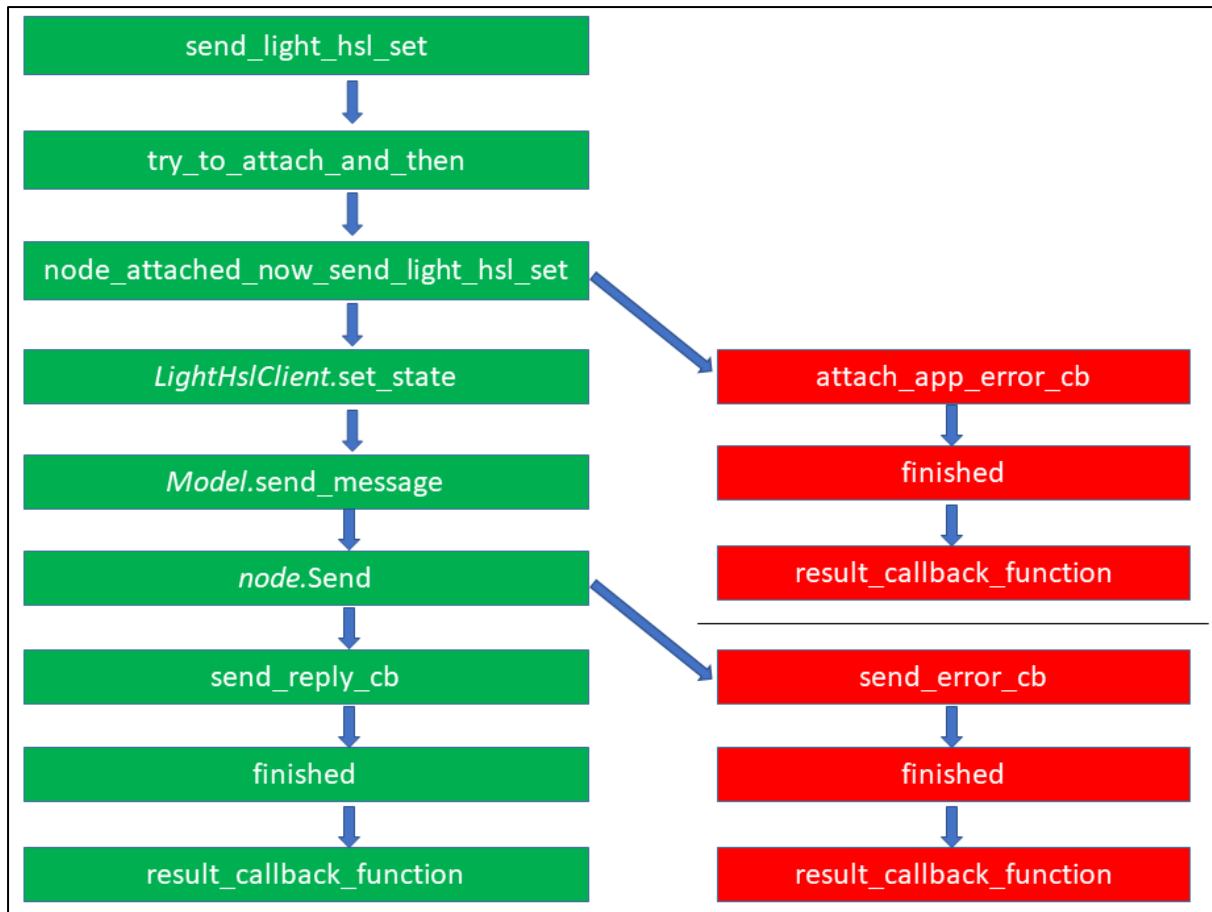


Figure 16 - Flow of execution sending a light HSL set message

The `send_light_hsl_set` function which your HTTP adapter code will call requires this signature:

```
def send_light_hsl_set(dst_addr_hex, h_hex, s_hex, l_hex, cb_function):
```

Using the work you did to implement support for generic on off set unack messages previously as an example and Figure 16 to provide an overview of structure and flow, try to implement as much of this use case as you can. The Bluetooth [mesh models specification](#) will give you details of the mesh message and its fields. We'll review the solution from the next page onwards.

#### 4.5.5.2.2.1 Coding

`send_light_hsl_set` should look something like this:

```
def send_light_hsl_set(dst_addr_hex, h_hex, s_hex, l_hex, cb_function):
    t = threading.Thread(target=timeout)
    t.start()
    global result_callback_function
    global dst_addr
    global h_state
    global s_state
    global l_state
    log("send_light_hsl_set("+dst_addr_hex+", "+h_hex+", "+s_hex+", "+l_hex+")")
    initNode()
    dst_addr_int = int(dst_addr_hex, 16)
    dst_addr = numpy.uint16(dst_addr_int)
    h_state = numpy.uint16(int(h_hex, 16))
    s_state = numpy.uint16(int(s_hex, 16))
    l_state = numpy.uint16(int(l_hex, 16))
    result_callback_function = cb_function
    log(constants.ACTION_LIGHT_HSL_SET_UNACK+ " action starting")
    try_to_attach_and_then(node_attached_now_send_light_hsl_set)
    return constants.RESULT_OK
```

This code:

- Starts by kicking off a timeout detection function in a background thread. Have a look at the `timeout()` function and `exit` Event object to see how this works.
- Calls `initNode`. This acquires a reference to the mesh D-Bus service which implements the BlueZ mesh API functions. It then constructs the node application and the parts of its node composition and obtains a reference to the Glib event loop.
- Converts the destination address and HSL state arguments, which have been passed as hex strings, into 16 bit unsigned int values.
- Stores the provided callback function in `result_callback_function` (see Figure 16).
- Initiates attaching to the bluetooth-meshtd daemon by calling `try_to_attach_and_then` with the function which must be executed after successfully attaching, as an argument.
- Returns a result code meaning OK to the caller. This just means the request was successfully accepted. The final outcome will be communicated in a call to the specified callback function.

The `try_to_attach_and_then` and `node_attached_now_send_light_hsl_set` functions are already in your starter code.

In the `LightHslClient` class, the `set_state` function should now look something like this:

```
def set_state(self, dest, key, h_state, s_state, l_state):
    log("set_state dest="+str(dest)+" state="+str(h_state)+" "+str(s_state)+" "+str(l_state) + "\n")
    # Light HSL Set Unacknowledged
    opcode = 0x8277
    # Opcode (16), L (16), H (16), S (16). NB H, S and L must be little endian.

    # And YES! Order is L, H then S
    h_state_LE = struct.pack('<H', h_state)
    s_state_LE = struct.pack('<H', s_state)
    l_state_LE = struct.pack('<H', l_state)
    self.data = struct.pack('>HHBBBB', opcode, l_state_LE[0],
l_state_LE[1], h_state_LE[0], h_state_LE[1], s_state_LE[0], s_state_LE[1], self.tid)
    self.tid = (self.tid + 1) % 255
    log("Sending:")
    log(byteArrayToHexString(self.data))
    self._send_message(dest, key, self.data)
```

The opcode value 0x8277 is the specified identifier for the light HSL set unacknowledged message type and you'll find this in the Bluetooth SIG Mesh Model specification.

Server Model	Message Name	Opcode
	Light HSL Saturation Set	0x82 0x73
	Light HSL Saturation Set Unacknowledged	0x82 0x74
	Light HSL Saturation Status	0x82 0x75
	Light HSL Set	0x82 0x76
	Light HSL Set Unacknowledged	0x82 0x77
	Light HSL Status	0x82 0x78
	Light HSL Target Get	0x82 0x79
	Light HSL Target Status	0x82 0x7A
	Light HSL Default Get	0x82 0x7B
	Light HSL Default Status	0x82 0x7C
	Light HSL Range Get	0x82 0x7D
	Light HSL Range Status	0x82 0x7E

Per the comments in the code above, watch out for the fact that the order of the H, S and L state values in the mesh message must be L, H and S. The full message layout, including a number of optional or conditional fields taken directly from the specification follows.

Field	Size (octets)	Notes
HSL Lightness	2	The target value of the Light HSL Lightness state
HSL Hue	2	The target value of the Light HSL Hue state
HSL Saturation	2	The target Light HSL Saturation state
TID	1	Transaction Identifier
Transition Time	1	Format as defined in Section 3.1.3. (Optional)
Delay	1	Message execution delay in 5 millisecond steps (C.1)

It's also important that byte ordering is correct. The H, S and L state values are 16-bit fields in little-endian order.

#### 4.5.5.2.2.2 Testing

Make sure you reinstated the temporarily commented out call to `gateway_i2mn_node.send_light_hsl_set` function in your HTTP handler code in `do_send_mesh_message.py` because you should be now ready to test this use case.

Execute the following tests from the command line. Make sure that your test node (Nordic Thingy) is powered on and has been provisioned and configured per earlier sections.

```
# First switch on the LED
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --data '{"action": "generic_onoff_set_unack", "dst_addr": "C001", "state": "01"}' --request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py {"dst_addr": "C001", "state": "01", "result": 0}
```

The LED in your test node should switch on.



```
# set the colour to red
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001", "h_state": "0000",
"s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "0000", "s_state": "FFFF", "l_state": "7FFF", "result": 0}
```

The LED in your test node should change colour to red (unless it was already red).

```
# set the colour to blue
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001", "h_state": "AAAA",
"s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "AAAA", "s_state": "FFFF", "l_state": "7FFF", "result": 0}
```

The LED in your test node should change colour to blue.

```
# set the colour to green
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001",
"h_state": "5555", "s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "5555", "s_state": "FFFF", "l_state": "7FFF": "7FFF", "result":
0}
```

The LED in your test node should change colour to green.

```
# set the colour to yellow
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001",
"h_state": "2AAA", "s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "2AAA", "s_state": "FFFF", "l_state": "7FFF", "result": 0}
```

The LED in your test node should change colour to yellow.

```
# set the colour to cyan
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001", "h_state": "8000",
"s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "8000", "s_state": "FFFF", "l_state": "7FFF", "result": 0}
```

The LED in your test node should change colour to cyan.

```
# set the colour to magenta
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001", "h_state": "D555",
"s_state": "FFFF", "l_state": "7FFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "D555", "s_state": "FFFF", "l_state": "7FFF", "result": 0}
```

The LED in your test node should change colour to magenta.

```
# set the colour to white
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "light_hsl_set_unack" , "dst_addr" : "C001",
"h_state": "0000", "s_state": "0000", "l_state": "FFFF"}' --request PUT http://localhost/cgi-
bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "h_state": "0000", "s_state": "0000", "l_state": "FFFF", "result": 0}
```

The LED in your test node should change colour to white.

```
# switch off
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --header "Content-Type: application/json" --
data '{"action" : "generic_onoff_set_unack" , "dst_addr" : "C001", "state" : "00" }' --
request PUT http://localhost/cgi-bin/gateway/do_send_mesh_message.py
{"dst_addr": "C001", "state": "00", "result": 0}
```

The LED in your test node should switch off.

#### 4.5.5.2.3 Task Completed

Congratulations! Your gateway should now allow you to switch your test node(s) on and off and set the colour of the LED, all using HTTP and JSON objects. Pretty cool!

If you are having a few problems, check your work carefully against the instructions in this section and don't forget, the full solution is available in [implementation/solutions/mesh](#). The Troubleshooting information in section 8 might also help.

#### 4.5.5.3 Coding the MN2I Path - temperature monitoring

We will now turn our attention to the mesh network to internet (mn2i) path through the system. We have a single use case to support which involves this path. We want sensor status messages published by nodes in our network to be received by the gateway and relayed over a web socket to connected TCP/IP gateway clients that have subscribed to the destination address of the message. In our particular case, one or more nodes running the Zephyr code provided will act as temperature sensors (assuming that they have been configured with a publish address).

As with the i2mn path, we will develop or complete the development of two Python scripts. One script which we will call `mesh_ws_adapter.py` will be invoked by `websocketd` whenever a new gateway client opens a websocket. It will receive and handle subscription requests from clients encoded as JSON objects. The second script, called `gateway_mn2i_node.py` will receive mesh sensor status messages and provide them to the websocket adapter script. If the gateway client that created the websocket has subscribed to the destination address of the mesh message it will be encoded as a JSON object and transmitted to the gateway client over the websocket connection.

##### 4.5.5.3.1 Task - Create `mesh_ws_adapter.py`

In `/usr/lib/cgi-bin/gateway`, create the file `mesh_ws_adapter.py` and insert the following code into it:

```
#!/usr/bin/python3
import threading
import json
import sys
sys.path.insert(0, '.')

sub_addresses = set()

fo = open("ws_log.txt", "a")
def wslog(line):
    fo.write(line+"\n")
    fo.flush()
```

This just imports code libraries and our `gateway_mn2i_node.py` node application script, creates a set object which will keep track of destination addresses the gateway client has subscribed to and implements a simple logging function.

Save the file and set its ownership and permissions as for other files.

##### 4.5.5.3.2 Task - Handle subscribe and unsubscribe requests

Next we'll add some code which will create a loop within which to accept and process subscribe and unsubscribe requests from the client. Don't forget that `websocketd` starts a new process containing the `mesh_ws_adapter.py` script for each connected websocket client and so we do not need to keep track of which client subscribed to which addresses in our code, only what the subscriptions requested by the owning client are.

#### 4.5.5.3.2.1 Coding

Open the file in your editor again and add the following:

```
keep_going = 1
tid = str(threading.current_thread().ident)

wslog(tid+" starting loop")
while keep_going == 1:
    try:
        line = sys.stdin.readline()
        if len(line) == 0:
            # means websocket has closed
            keep_going = 0
            wslog(tid+" Websocket closed")
            # tell the mesh node application to exit
            gateway_mn2i_node.finished()
        else:
            line = line.strip()
            message = {}
            wslog(tid+" RX> "+line)
            subscription_control = json.loads(line)
            if (subscription_control["action"] == "subscribe"):
                sub_addresses.add(subscription_control["dst"].upper())
                wslog(tid+" subscribed:"+str(sub_addresses))
            elif (subscription_control["action"] == "unsubscribe"):
                sub_addresses.remove(subscription_control["dst"])
                wslog(tid+" unsubscribed:"+str(sub_addresses))
    except Exception as err:
        wslog(tid+" Exception: {0}".format(err))
        gateway_mn2i_node.finished()
```

Take a look at this code. It consists of a while loop which will continue to run until the websocket connection is closed (signified by a zero length result when reading from it). We're expecting a JSON object and so parse what we receive and handle action parameters with values "subscribe" and "unsubscribe" by adding or removing the value of the accompanying *dst* property to the *sub\_addresses* set. In this way we keep track of the mesh message destination addresses that the client is interested in.

#### 4.5.5.3.2.2 Testing

To test the mesh websocket adapter script we will take a different approach and instead of using *curl* we will use a Python module and commands issued within the interactive Python interpreter. Install the WebSocket-client module as follows:

```
sudo pip3 install websocket-client
```

Start the *websocketd* daemon with parameters as shown here:

```
websocketd --port=8082 /usr/lib/cgi-bin/gateway/mesh_ws_adapter.py
```

*Hint: create a script to make it easier to issue this command in the future.*

In another terminal, start a Python interpreter by executing the command *python3*. From within the shell that starts, issue the following commands to test subscribe to and unsubscribing from destination addresses.

```
# get ready
import websocket
import json
ws = websocket.WebSocket()
ws.connect("ws://localhost:8082")
command = {}

# subscribe to destination address C002
command = {}
command["action"] = "subscribe"
```

```

command["dst"] = "C002"
ws.send(json.JSONEncoder().encode(command))

# unsubscribe to destination address C002
command = {}
command["action"] = "unsubscribe"
command["dst"] = "C002"
ws.send(json.JSONEncoder().encode(command))

```

Tail the `ws_log.txt` file in another terminal window. You should see something like this:

```

3070118608 starting loop
3070118608 RX>
{"action": "subscribe", "uuid": "67a3fba9ecb808c444088df87a9edb", "dst": "C002"}
3070118608 subscribed: {'C002'}
3069622992 RX> {"action": "subscribe", "dst": "C002"}
3069622992 subscribed: {'C002'}

3069622992 RX> {"action": "unsubscribe", "dst": "C002"}
3069622992 unsubscribed: set()

```

#### 4.5.5.3.3 Task - Create, provision and configure gateway\_mn2i\_node.py

Copy the `gateway_mn2i_node.py` and `mn2i_config.json` starter files from `implementation/start_state/mesh/bluetooth_mesh_api` onto your Raspberry Pi and into the `/usr/lib/cgi-bin/gateway` directory and set its ownership and permissions. Run `dos2unix` against both files to make sure they contain the right carriage return characters.

Review `gateway_mn2i_node.py`. You will find that it has much in common with `gateway_i2mn_node.py` as well as certain differences. It implements the mesh sensor client model in the `SensorClient` class and you'll find a number of `TODO` comments that indicate areas where the code is currently incomplete.

The functions that support joining (provisioning) are already in place in the starter code since it's the same as the code you already encountered in the `i2mn` node application.

Run `gateway_mn2i_node.py` from the command line with an argument *join*. On the Raspberry Pi which you are using as your provisioner, provision and configure this node application so that its sensor client model is subscribed to destination addresses `C002` and `C003`. Use Figure 17 and Figure 18 as a guide.

```

pi@raspberrypi:~ $ mesh-cfgclient
Mesh configuration loaded from /home/pi/.config/meshcfg/config_db.json
Proxy added: org.bluez.mesh.Node1 (/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Proxy added: org.bluez.mesh.Management1
(/org/bluez/mesh/node8c758f3726d147d488a42a7007957471)
Attached with path /org/bluez/mesh/node8c758f3726d147d488a42a7007957471

[mesh-cfgclient]# discover-unprovisioned on
Unprovisioned scan started
Scan result:
    rssi = -49
    UUID = 552FCD43CC8D4B569A29937CF6BC246C
    OOB = 0001
Scan result:
    rssi = -45
    UUID = 552FCD43CC8D4B569A29937CF6BC246C
    OOB = 0001

[mesh-cfgclient]# discover-unprovisioned off

[mesh-cfgclient]# provision 552FCD43CC8D4B569A29937CF6BC246C
Provisioning started
Request hexadecimal key (hex 16 octets)

```

```

[[mesh-agent]# ] Enter key (hex number): 0ba81bbde1289ba2a3c2acb773fc7578
Assign addresses for 1 elements
Provisioning done:
Mesh node:
    UUID = 552FCD43CC8D4B569A29937CF6BC246C
    primary = 00e4
    elements (1):

[mesh-cfgclient]# menu config

[mesh-cfgclient]# target 00e4
Configuring node 00e4

[config: Target = 00e4]# timeout 5
Timeout to wait for remote node's response: 5 secs

[config: Target = 00e4]# composition-get 0
Received DeviceCompositionStatus (len 19)
Received composition:
    Feature support:
        relay: yes
        proxy: no
        friend: yes
        lpn: no
    Element 0:
        location: 0000
        SIG defined models:
            Model ID      0000 "Configuration Server"
            Model ID      1102 "Sensor Client"

[config: Target = 00e4]# appkey-add 0
Received AppKeyStatus (len 4)
Node 00e4 AppKey status Success
NetKey  0 (0x000)
AppKey  0 (0x000)

[config: Target = 00e4]# bind 00e4 0 1102
Received ModelAppStatus (len 7)
Node 00e4: Model App status Success
Element Addr  00e4
Model ID      1102 "Sensor Client"
AppIdx        0 (0x000)

[config: Target = 00e4]# sub-add 00e4 c002 1102
Received ModelSubStatus (len 7)

Node 00e4 Subscription status Success
Element Addr  00e4
Model ID      1102 "Sensor Client"
Subscr Addr   c002

[config: Target = 00e4]# sub-add 00e4 c003 1102
Received ModelSubStatus (len 7)

Node 00e4 Subscription status Success
Element Addr  00e4
Model ID      1102 "Sensor Client"
Subscr Addr   c003

```

**Figure 17 - mesh-cfgclient provisioning and configuring the mn2i node**

```

pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ./gateway_mn2i_node.py join
PromptStatic ( static-oob )
Enter 16 octet key on remote device: 0ba81bbde1289ba2a3c2acb773fc7578

Node provisioned OK - token=0xd59015c1f32d0271
Quitting mainloop

```

**Figure 18 - mn2i node application being provisioned**

Add a function called `getToken()` which returns the token value allocated to your mn2i node application when it was provisioned. The function should look like this but with your particular token value:

```

# TODO implement getting a token from the pool

```

```
def getToken():
    return "d59015c1f32d0271"
```

#### 4.5.5.3.4 Task - Establish HTTP adapter callback functions

In `gateway_mn2i_node.py` find the functions `set_message_dispatcher` and `set_on_error`. These functions need to be called from within the `mesh_ws_adapter.py` script and provide references to the functions implemented in `mesh_ws_adapter.py` that `gateway_mn2i_node.py` must call to pass data received in mesh sensor status messages or error codes and messages respectively.

Edit `mesh_ws_adapter.py` and add the required callback functions.

```
def on_error(err_code, err_message):
    wslog(tid+" error reported: "+str(err_code))
    result = {}
    result['result'] = err_code
    result['message'] = err_message
    sys.stdout.write(json.JSONEncoder().encode(result)+"\n")
    sys.stdout.flush()

def message_dispatcher(dst, sensor_data):
    global sub_addresses
    tid = str(threading.current_thread().ident)
    wslog(tid+" message_dispatcher:"+dst.upper()+" "+sensor_data)

    if (dst.upper() in sub_addresses):
        wslog(tid+" sending sensor_data to subscribed client")
        message = {}
        message['dst'] = dst.upper()
        message['temperature'] = sensor_data
        print(json.JSONEncoder().encode(message))
        sys.stdout.flush()
```

The `on_error` function:

- creates a dictionary object called *result*
- sets key values pairs named *result* and *message* to the error code and error message provided as function parameters
- transforms the dictionary object into a JSON string and outputs it to standard out, which websocketd redirects to the websocket connection with the client.

The `message_dispatcher` function:

- receives a destination address and sensor data item
- checks the set of destination addresses to which the client has subscribed and if the sensor data is addressed to one of the subscribed addresses sends the data as a JSON string to the client.

Provide references to each of these two functions to the `gateway_mn2i_node.py` script by adding the highlighted code:

```
keep_going = 1
tid = str(threading.current_thread().ident)
gateway_mn2i_node.set_message_dispatcher(message_dispatcher)
gateway_mn2i_node.set_on_error(on_error)
```

And at the top of the script, import the `gateway_mn2i_node.py` script like this:

```
#!/usr/bin/python3
import threading
```

```
import json
import sys
sys.path.insert(0, '.')
import gateway_mn2i_node
```

#### 4.5.5.3.5 Task - Implement the mesh message receiver loop

In `gateway_mn2i_node.py` find the comment which reads “TODO implement the message receiver loop”. The next task is to add code to the `mn2i` script which attaches to the mesh network, receives and processes sensor status messages by passing their content back to the websocket adapter. We need to ensure that the websocket adapter script can continue to service subscribe and unsubscribe requests from the client though, so receiving and handling mesh messages will be performed in a background thread.

In `gateway_mn2i_node.py` add the following code:

```
def receive():
    global mainloop
    mainloop = GLib.MainLoop()
    tid = str(threading.current_thread().ident)
    try:
        initNode()
        log(tid+" SENSOR CLIENT")
        log(tid+" -----")
        log(tid+" receive starting")
        try_to_attach()
    except Exception as err:
        log(tid+" exception in receive(): "+str(err))
    finished()
```

This code:

- initialises the node in the standard way which you have already seen expect that this time the node’s composition includes an implementation of the sensor client model in `SensorClient`.
- Attaches to the network and starts the event loop

Now edit `mesh_ws_adapyer.py` and add the highlighted code:

```
keep_going = 1
tid = str(threading.current_thread().ident)
gateway_mn2i_node._message_dispatcher(message_dispatcher)
gateway_mn2i_node.set_on_error(on_error)

t = threading.Thread(target=gateway_mn2i_node.receive)
t.daemon = True
t.start()
```

This code:

- creates a thread with the `receive` function in `gateway_mn2i_node.py` as its entry point
- designates the thread a daemon thread so that it does not stop the process as a whole from exiting when required
- starts the thread running

#### 4.5.5.3.6 Task - Complete the sensor client model message handler

In `gateway_mn2i_node.py` find the comment which reads “# TODO implement the Sensor Client Model message handler”. This comments appears under a method called `process_message` which belongs to the `SensorClient` class. It gets called by the `Element` class when a message is received.

The process\_message method in SensorClient needs now to be completed. The opcode of the message must be checked to determine whether or not the message passed by the Element class is of a type that this model should process. We're only interested in Sensor Status messages.

Sensor	Sensor Descriptor Get	0x82 0x30
	Sensor Descriptor Status	0x51
	Sensor Get	0x82 0x31
	Sensor Status	0x52
	Sensor Column Get	0x82 0x32
	Sensor Column Status	0x53
	Sensor Series Get	0x82 0x33
	Sensor Series Status	0x54

Figure 19 - a subset of message types belonging to the sensor models

The message payload then needs to be processed such that sensor data is extracted and passed to the mesh\_ws\_adapter.py script by making a call to its message\_dispatcher callback function.

The payload of a sensor status message uses Tag/Length/Value formatting so that multiple values of more than one type can be transported in a single message (see Figure 20). A *property ID* acts as the tag, identifying the type of data. Properties and their identifiers are defined in the Mesh Device Properties specification.

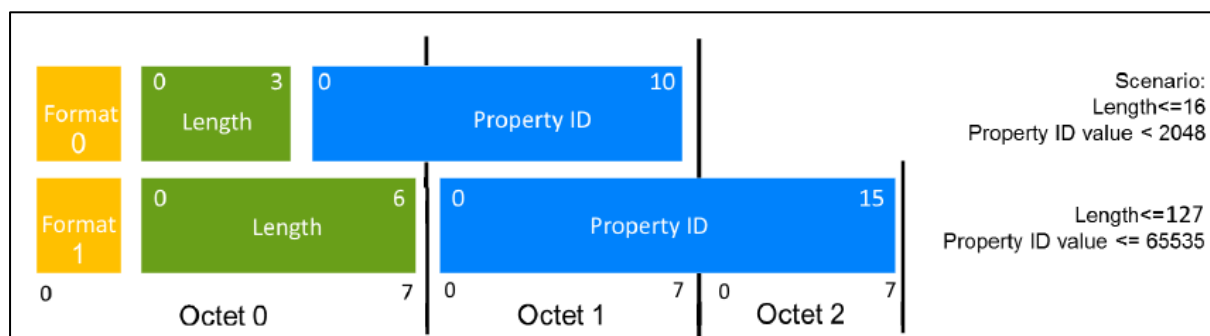


Figure 20 - sensor status message TLV format

Our incomplete implementation of the sensor status message handler will only extract temperature data of property type Present Ambient Temperature. This property type has an ID of 0x004F.

Property	Characteristic	Property ID
Present Ambient Noise	Noise	0x0079
Present Ambient Relative Humidity	Humidity	0x0076
Present Ambient Temperature	Temperature 8	0x004F



Update process\_message belonging to SensorClient so that it looks like this:

```
def process_message(self, source, dest, key, data):
    global message_dispatcher
    datalen = len(data)
    opcode = bytes(data[0:1])[0]
    if (opcode == 0x52):
        format_length_byte = bytes(data[1:2])[0]
        sensor_data_format = (format_length_byte >> 7)
        # only Format A is supported
        if (sensor_data_format == 0):
            sensor_value_length = ((format_length_byte & 0b01111000) >>
3)

            id1 = format_length_byte & 0b00000111
            id2 = bytes(data[2:3])[0]
            property_id = (id1 << 8) | id2
            # only temperature data is supported
            if (property_id == 0x004F):
                # Example: 52084f27 = opcode, format and length,
                # property ID, value
                sensor_value = bytes(data[3:4])[0]
                sensor_value = sensor_value * 0.5
                tid = str(threading.current_thread().ident)
                log(tid+' SensorClient opcode=' + str(hex(opcode)))
                #
                + ' value=' + str(sensor_value))
                message_dispatcher(format(dest, "x"), str(sensor_value))
```

This code:

- selects the message for processing if its opcode indicates it is a Sensor Status message (0x52)
- demarshalls the payload (subject to the limitations of our implementation) and extracts sensor data values if and only if the property ID indicates that it is a Present Ambient Temperature value.
- passes the sensor value as a hex string back to the mesh\_ws\_adapter.py script by making a callback to its message\_dispatcher function.

Note that for the purposes of this project we have only implemented a subset of the sensor client model. This is a very versatile model with many message types and its status messages are able to contain multiple sensor data values of multiple types. A model implementation with such limitations in a real product would not pass SIG qualification.

#### 4.5.5.3.7 Task - Test the MN2I path

Start websocketd with your script:

```
#!/bin/bash
websocketd --port=8082 /usr/lib/cgi-bin/gateway/mesh_ws_adapter.py
```

Place your Nordic Thingy somewhere not too far away where the temperature is different (e.g. on the window ledge).

Start a python3 shell in another terminal windows and execute the commands shown here:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import websocket
>>> import json
>>> ws = websocket.WebSocket()
>>> ws.connect('ws://localhost:8082')
>>> command = {}
>>> command = {}
```

```

>>> command["action"] = "subscribe"
>>> command["dst"] = "C002"
>>> ws.send(json.JSONEncoder().encode(command))
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "25.0"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> ws.recv()
'{"dst": "C002", "temperature": "24.5"}'
>>> quit()

```

Execute `ws.recv()` approximately every 10 seconds since this is frequency with which the Thingy is sampling the temperature.

#### 4.5.6 Gateway status

If you've completed the work to this point then you should now have a working Bluetooth mesh internet gateway which supports the three required use cases. You can switch groups of devices on or off and can set the colour of LEDs. You can subscribe to messages by group address and receive sensor status messages containing ambient temperature measurements. Excellent!

## 5. Project - Building Gateway Applications

### 5.1 Introduction

You should now have a working gateway. This section will review and explore applications that use gateways and will give you some experience of developing one. If you didn't complete the tasks of section 4 you can still set up a working gateway, using the solution in the `implementation/solutions/mesh/gateway` directory.

Applications can take many different forms. All that is required, when developing a gateway client application is that it be possible to work with HTTP and web sockets. You may be using a language which has high level APIs for both these things, or you may be working with a constrained microcontroller, with rudimentary support for TCP/IP sockets only, meaning that you will need to develop the HTTP and web socket capability.

The most common platforms to use a Bluetooth internet gateway are likely to be mobile applications and web applications, all of which have rich collections of APIs.

We're going to focus on web applications since they are the easiest and fastest to work with and even if you have no prior experience with HTML, JavaScript, AJAX and the web socket APIs, the learning curve is shallow.

Your next set of tasks will involve completing an application which is designed to allow control of a network of Bluetooth mesh lights and/or sensors to be exercised from a web browser via the gateway.

## 5.2 Task - Complete the mesh network control application

### 5.2.1 Overview

You will now have the opportunity to complete the development of a simple web application which you can use with your network of devices (even if you only have one Nordic Thingy). The web application will

1. Allow you to select a group address to use with transmitted mesh messages
2. Allow groups of devices in the mesh network to be switched on
3. Allow groups of devices in the mesh network to be switched off
4. Allow groups of devices in the mesh network to have the colour of their LED changed
5. Allow you to subscribe to and receive sensor status messages sent to one or two hard-coded destination addresses, one for interior temperature monitoring and one for external temperature monitoring.
6. Receive temperature readings from mesh sensor status messages and plot the values on a chart in real time.
7. Optionally, stream video from a webcam which is pointing at the devices in your network so that a remote user can see the results of their actions.

### 5.2.2 Preparation

#### 5.2.2.1 Task - Install the motion daemon

Ideally, you should plug a webcam into one of the USB ports of your Raspberry Pi. You can then point it at the micro:bit to be controlled and watch it respond from within the gateway application. This is an optional step but recommended.

To support the streaming of images from a webcam to a web page, we need to install some software called *motion*. See <https://motion-project.github.io/index.html> for further details about *motion*.

To install *motion*, follow execute the command `sudo apt install motion`.

Edit the configuration file `/etc/motion/motion.conf`, find the following properties and set them to the values indicated. If a property is not present in the default file, add it.

```
framerate 50
stream_maxrate = 25
daemon on
stream_quality 100
stream_localhost off
```

Do not yet attempt to run *motion*. We want to run it as a non-root user for security reasons and a few additional steps are required to achieve this.

Edit the file `/etc/default/motion` and set the property `start_motion_daemon=yes`

Run the command `sudo systemctl daemon-reload`

Run the command `sudo /etc/init.d/motion restart`

Yes, use the **restart** argument. This is correct.

Check the status of motion with:

```
pi@raspberrypi:~ $ sudo /etc/init.d/motion status
• motion.service - LSB: Start Motion detection
  Loaded: loaded (/etc/init.d/motion; generated)
  Active: active (running) since Thu 2021-06-24 07:30:49 BST; 2s ago
    Docs: man:systemd-sysv-generator(8)
  Process: 3941 ExecStart=/etc/init.d/motion start (code=exited, status=0/SUCCESS)
    Tasks: 3 (limit: 4915)
   CGroup: /system.slice/motion.service
           └─3969 /usr/bin/motion

pi@raspberrypi:~ $ ps -ef|grep motion
motion  4030      1  10 07:33 ?        00:00:07 /usr/bin/motion
pi       4036    1090  0 07:34 pts/0    00:00:00 grep --color=auto motion
```

Note that the `/usr/bin/motion` binary is running in a process owned by user *motion*. This is what we wanted.

#### 5.2.2.2 Task - Run the websocketd daemon

If it's not already running, start the websocketd daemon with the commands:

```
websocketd --port=8082 /usr/lib/cgi-bin/gateway/mesh_ws_adapter.py
```

#### 5.2.2.3 Task - Install the incomplete application

The study guide package includes an incomplete version of the micro:bit controller application.

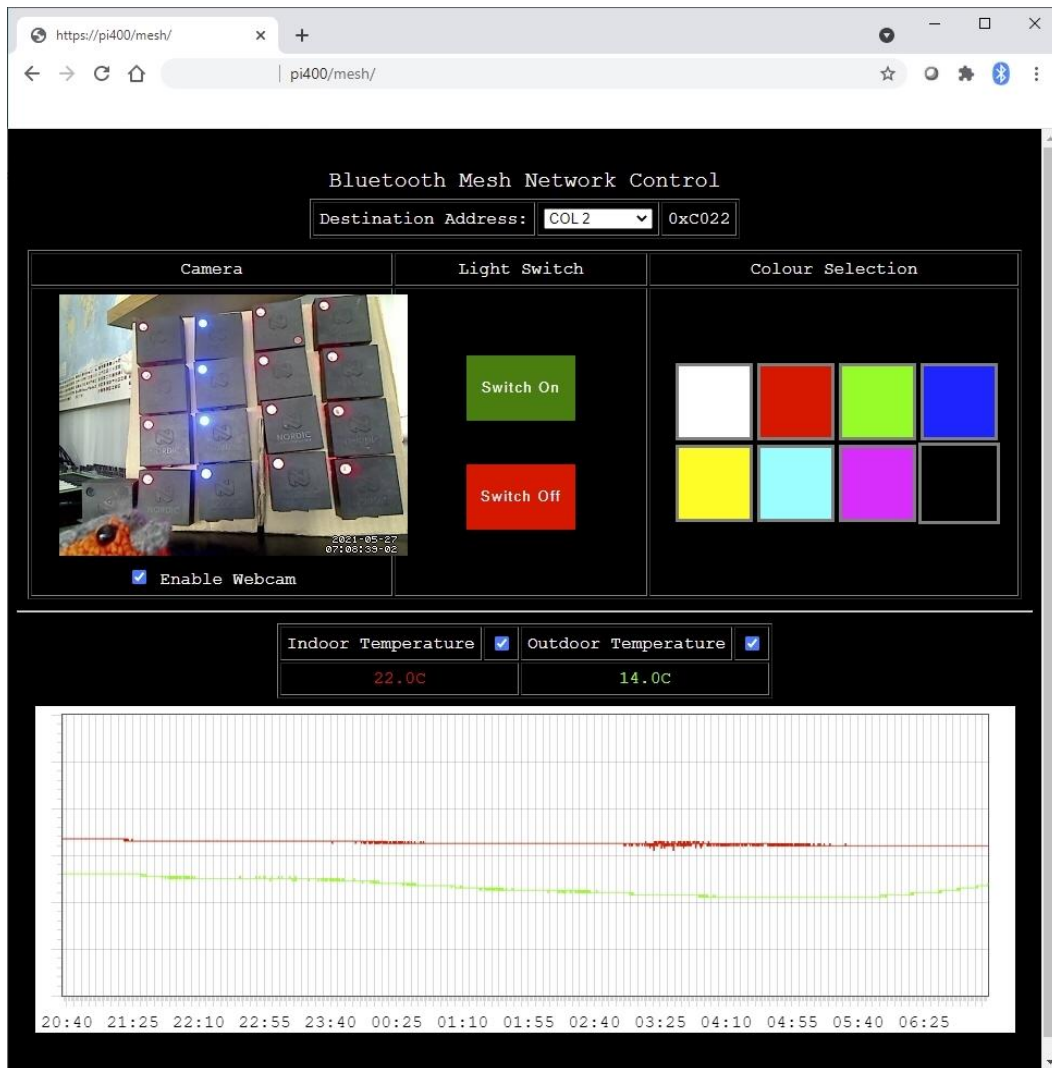
Create the directory `/var/www/html/mesh` on your Raspberry Pi and set ownership and permissions.

```
pi@raspberrypi:/var/www/html $ sudo mkdir mesh
pi@raspberrypi:/var/www/html $ sudo chown www-data:www-data mesh/
pi@raspberrypi:/var/www/html $ sudo chmod 775 mesh/
```

Then copy the contents of the study guide's `implementation\start_state\mesh\web` directory into it.

#### 5.2.2.4 Task - Application Orientation

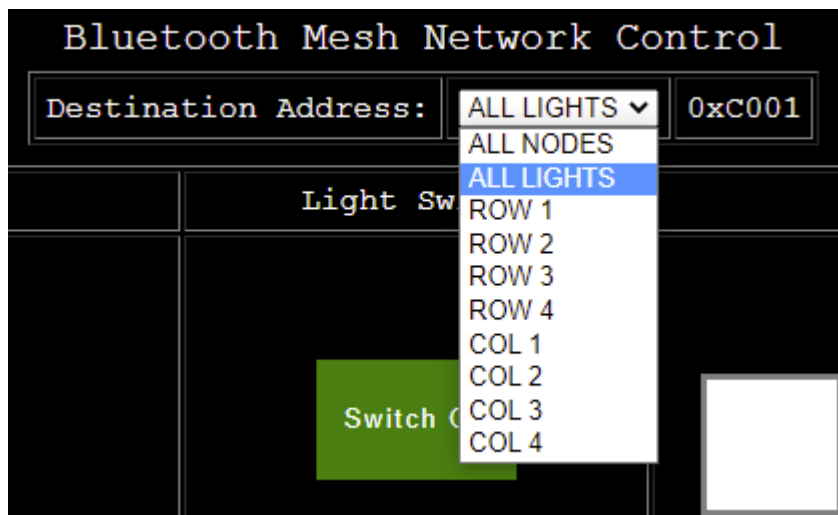
When finished, the mesh network controller application will look like the screenshot shown in Figure 21.



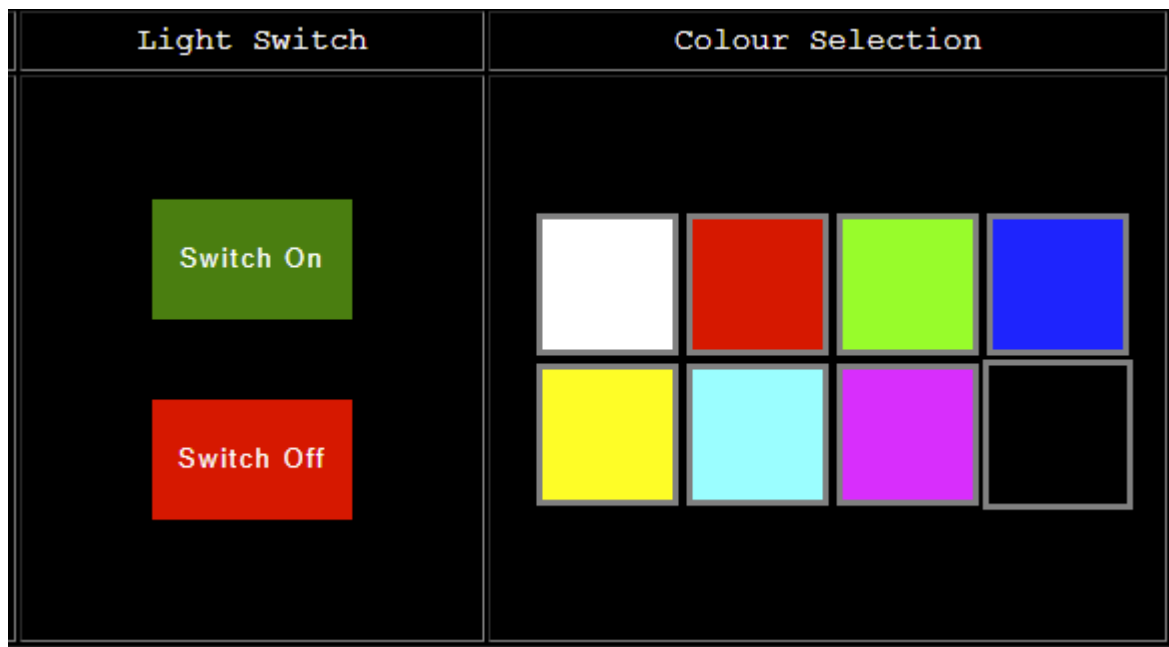
**Figure 21 - The finished mesh network control application**

As you can see, it displays a live video feed from the webcam on the left.

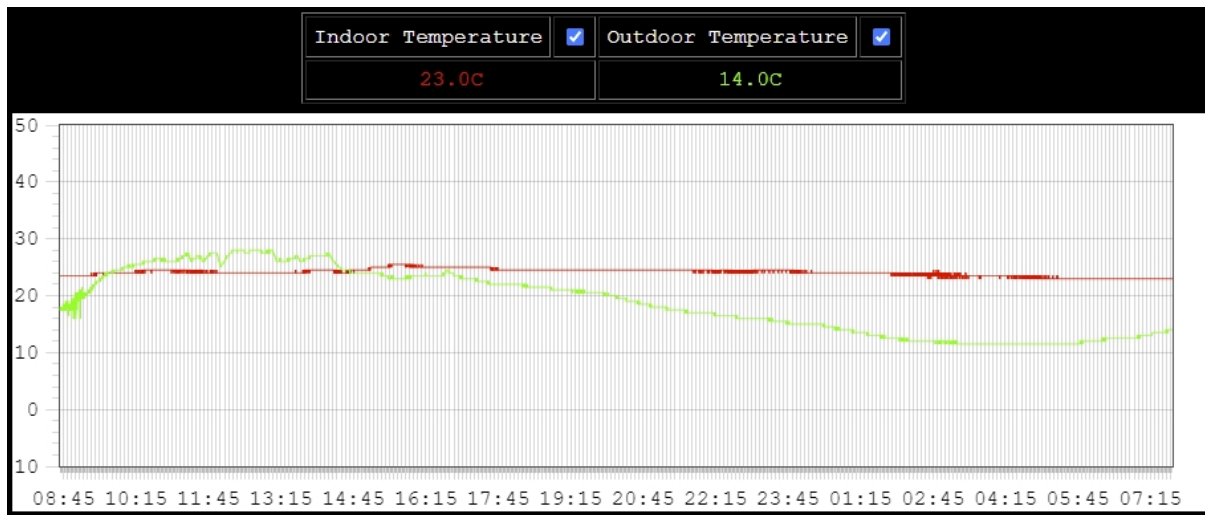
There's a dropdown list from which to select a destination address. The addresses that nodes have been subscribed to are available and are listed using names that correspond to the expected row or column a node occupies in a grid. In addition address C001 which all nodes in the network that can act as lights is available under the name "ALL LIGHTS" and the special system address 0xFFFF which all nodes implicitly subscribe to is available under the name "ALL NODES".



The user interface also provides buttons for switching on or off nodes or changing their colour.

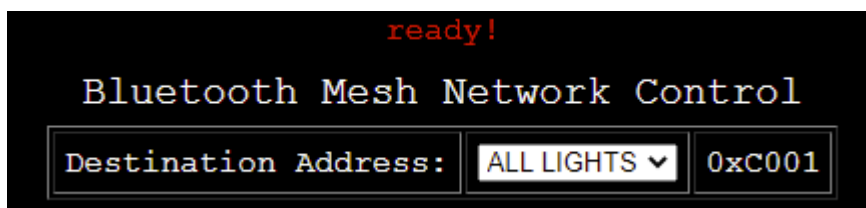


Finally, the bottom part of the UI allows addresses to which sensor status messages will be published to be subscribed to or unsubscribed from and includes an area in which a chart of temperature values will be drawn.



The starter version of the application which you have installed, contains everything it needs except for some of the functions which invoke gateway API operations. Your next task will be to complete the application by implementing these functions.

Point your web browser at the application's sole page. The URL should be <http://raspberrypi/mesh/> but this may vary if you are using a different host name for your Pi. If websocketd is running and all is well, the page will appear and the message "ready!" will be displayed at the top of the page.



#### 5.2.2.5 Task - Invoking HTTP on/off and light colour requests from JavaScript

The mesh network control application is a web application. Consequently, it consists of HTML, CSS and JavaScript. JavaScript is used to write functions which provide most of the dynamic aspects of the application and it is in JavaScript that gateway related functions must be written.

JavaScript includes the XMLHttpRequest API which allows HTTP requests to be sent and the response to be received asynchronously and handled.

Study the following example which shows how to use JavaScript to initiate HTTP PUT operations with a JSON object sent in the HTTP request entity body and a JSON object returned as the response.

```
function httpPut() {
    var xhttp = new XMLHttpRequest();

    // this is a callback function which is called when the request state changes
    xhttp.onreadystatechange = function() {

        // this means the request has completed, a response received and the HTTP status
        // of the request is OK (status == 200)

        if (this.readyState == 4 && this.status == 200) {
            result = JSON.parse(this.responseText);
            if (result.result == 0) {
                // it worked! Perhaps the JSON object contains other data?
            }
        }
    }
}
```

```

    }
};

var args = {};
args.paramA = data_1;
args.paramB = data_2;
var json = JSON.stringify(args);

var target = "do_something.py";
// configure the request
xhttp.open("PUT", CGI_ROOT+target, true);

// set a HTTP header to indicate that the body contains a JSON object
xhttp.setRequestHeader('Content-type','application/json; charset=utf-8');

// send it with the JSON data in the entity body of the request
xhttp.send(json);
}

```

**Figure 22 - HTTP PUT with JSON from JavaScript**

### 5.2.3 Task - Completing the mesh network control application

Open the file `js/gateway.js` in a text editor. This file contains all of the functions which the application uses to interact with the gateway. Some of these functions are currently empty placeholders. Your task is to complete these functions.

#### 5.2.3.1 Sending a JSON request object to the gateway

Each of the switch on, switch off and set colour use cases will require the web application to formulate and send a JSON object over using HTTP PUT to gateway with the same resource as the execution target. We'll start by implementing a function which takes a JSON object as an argument and uses the JavaScript XMLHttpRequest API to send it to the `do_send_mesh_message.py` CGI script.

Find the empty function `sendMeshMessage` in your code and update it so it looks like this:

```

function sendMeshMessage(json) {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        console.log(this.responseText);
        if (this.readyState == 4 && this.status == 200) {
            result = JSON.parse(this.responseText);
            message(result_string(result.result));
        }
    };

    var target = "do_send_mesh_message.py";

    xhttp.open("PUT", CGI_ROOT+target, true);
    xhttp.setRequestHeader('Content-type','application/json; charset=utf-8');
    xhttp.send(json);
}

```

This is a simple function which creates a request and sets the HTTP content header to the MIME type `application/json` and then sends the JSON object in string form to the remote CGI script, `do_send_mesh_message.py`. The `onreadystatechange` function is a callback function which gets called whenever the state of the request changes. Here we're checking that the HTTP result was OK and displaying the content of the JSON response object returned by `do_send_mesh_message.py`.

#### 5.2.3.2 Switching nodes on and off

##### 5.2.3.2.1 Coding

Find the function named `sendOnOffSetMessage` in your editor. Update it as shown so that it formulates a JSON object that represents a request to send a mesh *generic on off set unacknowledged* message to a selected destination address and with a selected state value and then



uses the `sendMeshMessage` function to send it to the gateway for processing by the `do_send_mesh_message.py` script.

```
function sendOnOffSetMessage(state_hex) {
    var args = {};
    args.action = ACTION_GENERIC_ON_OFF_SET_UNACK;
    args.dst_addr = selected_dst;
    args.state = state_hex;
    var json = JSON.stringify(args);
    console.log(json);
    sendMeshMessage(json);
}
```

#### 5.2.3.2.2 Testing

Make sure that you have the `bluetooth-meshd` daemon running and your test node(s) switched on before proceeding.

Clear the browser cache (by pressing `CTRL + SHIFT + DEL` in Chrome on Windows) and reload the page in your browser so that your latest JavaScript code is in use.

With the ALL LIGHTS destination address selected, click the Switch On button. Your node's LED should switch on. Now click Switch Off. The LED should go off. Experiment with the destination address selector. Your node should respond only to messages sent to addresses it has been subscribed to.

Open the developer console in your browser (`CTRL + SHIFT + DEL` in Chrome on Windows). It should contain messages such as:

```
{"action":"generic_onoff_set_unack","dst_addr":"C001","state":"01"}
gateway.js:78
gateway.js:78
gateway.js:78 {"dst_addr": "C001", "state": "01", "result": 0}

gateway.js:104 {"action":"generic_onoff_set_unack","dst_addr":"C001","state":"00"}
gateway.js:78
gateway.js:78
gateway.js:78 {"dst_addr": "C001", "state": "00", "result": 0}
```

Here you can see two pairs of JSON objects. The first of each pair is the request object being sent to address C001 (all lights) with a state value of 0x01 to switch the LEDs on or 0x00 to switch them off. The second is the response object returned by the gateway with a result of 0 meaning the action was successful.

#### 5.2.3.3 Changing the colour of LEDs

##### 5.2.3.3.1 Coding

Find the function named `sendLightHslSetMessage` in your editor. Update it as shown so that it formulates a JSON object that represents a request to send a mesh *light HSL set unacknowledged* message to a selected destination address and with specified hue, saturation and lightness state values and then uses the `sendMeshMessage` function to send it to the gateway for processing by the `do_send_mesh_message.py` script.

```
function sendLightHslSetMessage(h_hex, s_hex, l_hex) {
    var args = {};
    args.action = ACTION_LIGHT_HSL_SET_UNACK;
    args.dst_addr = selected_dst;
    args.h_state = h_hex;
    args.s_state = s_hex;
```

```
args.l_state = l_hex
var json = JSON.stringify(args);
console.log(json);
sendMeshMessage(json);
}
```

#### 5.2.3.3.2 Testing

Clear your browser cache and reload the page.

With the ALL LIGHTS destination address selected, click the Switch On button. Your node's LED should switch on. Now click the red colour selector button. The LED should change colour to red. Experiment with the other colours and ensure the result is as expected. Switch the light(s) off and then click to change the colour. You will not immediately see the colour change of course because your lights are currently off. But the internal HSL state values should have changed. Switch the light(s) on and they should switch on but using the latest selected colour.

Open the developer console in your browser (CTRL + SHIFT + i in Chrome on Windows). It should contain messages such as:

```
{"action":"light_hsl_set_unack","dst_addr":"C001","h_state":"0000","s_state":"FFFF","l_state":"7FFF"}
gateway.js:78
gateway.js:78
gateway.js:78 {"dst_addr": "C001", "h_state": "0000", "s_state": "FFFF", "l_state": "7FFF", "result": 0}
```

Here you can see a pair of JSON objects. The first is the request object being sent to address C001 (all lights) with HSL state values of H=0x0000, S=0xFFFF and L=0x7FFF which between them represent the colour red. The second is the response object returned by the gateway with a result of 0 meaning the action was successful.

#### 5.2.3.4 Temperature sensor measurements

The final use case our web application must support concerns the charting of ambient temperature measurements from a sensor or sensors. If you only have a single node to test with, it should have been configured with a publish address of 0xC002. In our web user interface, this is deemed to be the address used by a sensor which measures the Indoor Temperature. If you have more than one node to test with, one of the others should have been configured to publish sensor status messages to group address 0xC003, which our application interprets as being the Outdoor Temperature. Mount this device somewhere outdoors but make sure it remains in range of your Raspberry Pi gateway node.



Figure 23 - A Bluetooth mesh node measuring the outdoor temperature

#### 5.2.3.4.1 Coding

The temperature charting use case breaks down into a number of functions. The two checkboxes above the chart area allow the user to switch on or off charting of each of the two temperature measurements. Enabling or disabling one of these checkboxes must cause the web application to send a JSON request object over a websocket connection rather than make an HTTP PUT request as with the previous use cases. Depending on the selected state, the object sent to the gateway will indicate that the client (the browser) wishes to subscribe to the associated address or unsubscribe from it.

The MN2I receiver code running on the gateway will be receiving sensor status messages from the mesh network, addressed to both 0xC002 and 0xC003 (see section 4.5.5.3.3 to remind yourself how the gateway's MN2I script was configured as a mesh node application). It will then dispatch temperature measurements as JSON objects over web socket connections to those clients that have subscribed to the address that each mesh message was addressed to.

Find the JavaScript function *startWs()* in your code and review it. This code establishes a web socket connection and implements the required callback functions.

You should note that

1. When a web socket is created it uses a URL which is in a constant defined in the constants.js file
2. The web socket has three callback functions names *onerror*, *onopen* and *onmessage*.
3. A new web socket is created only the first time an action requiring the connection is invoked. Otherwise, the web socket object, stored when originally created is reused.
4. Objects sent from the client to gateway over the web socket connection represent either a request to subscribe to an address or unsubscribe from an address.
5. The *onmessage* callback function gets called whenever the gateway sends a sensor status message in JSON format. The string representation of the JSON object is turned into an actual JSON object in the line *result = JSON.parse(event.data)*. After that, if the object has a *dst* property, we extract the temperature reading and treat it as an indoor temperature measurement or an outdoor temperature measurement depending on the destination address. Values are immediately displayed and are also added to the end of an array which drives the charting process.

Find the functions *subscribe* and *unsubscribe*, both of which are currently incomplete. Update them so that they each formulate a JSON object and send it over the websocket to the gateway.

```
function subscribe(dst) {
  if (websocket.readyState === WebSocket.OPEN) {
    var sub_json = {action: "subscribe", dst: dst};
    console.log(JSON.stringify(sub_json));
    websocket.send(JSON.stringify(sub_json));
  }
}

function unsubscribe(dst) {
  if (websocket.readyState === WebSocket.OPEN) {
    var sub_json = {action: "unsubscribe", dst: dst};
    console.log(JSON.stringify(sub_json));
    websocket.send(JSON.stringify(sub_json));
  }
}
```

This code:

- checks that the websocket connection is ready
- creates a JSON object with an action property of “subscribe” or “unsubscribe” plus the destination address as the property *dst*.
- Writes the object in string format over the web socket connection to the gateway

To see how these functions are used, look for and review *onIndoorTempSubSwitchClicked* and *onOutdoorTempSubSwitchClicked*. You’ll notice that it is in these functions that we check for the availability or otherwise of an existing websocket connection and if necessary, first create one by calling `startWs()` and then executing the required subscribe or unsubscribe action.

#### 5.2.3.4.2 Testing

Start your websocketd daemon. Clear your browser cache and reload the web page. Check the Indoor Temperature checkbox. If you have more than one test node, also check the Outdoor Temperature checkbox. Watch for messages at the top of the page such as **C002 says it's 24.0C** and temperature values appearing under the checkbox(es). Monitor the developer console where you will see output such as:

```
creating websocket connection
gateway.js:220 {"action":"subscribe","dst":"C002"}
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"23.5\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"23.5\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
gateway.js:162 message received: "{\"dst\": \"C002\", \"temperature\": \"24.0\"}"
```

When sufficient values have been received, you should also see the chart being drawn. Leave this test running for several hours if you can.

Unsubscribe from one or more of the temperature sensor addresses and note that messages addressed to that destination address are no longer received. Leave it in the unsubscribed state for a few minutes before subscribing again and checking that values are once again received and charted.

### 5.3 Gateway application status

You should now have a working, example mesh gateway web application and can exercise the various functions supported. Enjoy!

## 6. Security

### 6.1 Warning!

At this point, you should have a working gateway for Bluetooth mesh networks and a working gateway application. However, you should note the following:

*The current implementation is insecure and should not be connected to the internet or other insecure networks!*

It's insecure because we have not yet turned our attention to the subject of security. Module 05 covers security for both LE Peripheral gateways and gateways for Bluetooth mesh. Proceed now to study module 05 and start to secure your mesh gateway. Return here when you have completed module 05 and then complete those remaining steps that apply specifically to the gateway for Bluetooth mesh.

### 6.2 Further Security Measures for the Bluetooth Mesh Gateway

Module 05 identified generally applicable actions to take in securing a Bluetooth internet gateway of either of the two types dealt with in this study guide. The table of issues and countermeasures from section 3.1.4 of module 5 is repeated here but highlights any special actions required to secure the gateway for LE Peripherals.

Ref	Short Name	LE Peripheral Gateway Countermeasure
1	No user authentication or access control	The countermeasures from module 05 apply.
2	TCP/IP port visibility	The countermeasures from module 05 apply.
3	Data over TCP/IP is not private	The countermeasures from module 05 apply.
4	Multiple TCP/IP ports in use	The countermeasures from module 05 apply.
5	websocketd is directly accessible	The countermeasures from module 05 apply.
6	motion is directly accessible	The countermeasures from module 05 apply.
7	Bluetooth data in flight is not necessarily private	All Bluetooth mesh messages are encrypted using a network key and an application key and are therefore private in flight. As such this issue does not apply to a Bluetooth mesh gateway which must itself act as a node in the associated network in order that it can publish and receive mesh messages.
8	Not all Bluetooth devices should be accessible	In a Bluetooth mesh network, messages are sent by nodes and received and acted upon by zero or more other nodes in the same network. Messages have a unicast source address and a destination address which may be unicast or more likely will be a group address or a virtual address. These latter two address types

		<p>are subscribed to by nodes that wish to receive and process these messages. There exists, therefore, a set of zero or more nodes which have subscribed to each destination address to which messages are sent in the network.</p> <p>Restricting access to particular nodes or sets of nodes can be achieved by restricting the addresses that the gateway can send messages to and which the gateway will dispatch to gateway clients. A component we will call a <i>Bluetooth mesh firewall</i> can be added to our gateway to accomplish this.</p>
9	Not all Bluetooth device capabilities should be available	In the case of Bluetooth mesh nodes, only particular message types will be supported by the gateway and this will automatically restrict access to specific mesh node capabilities without the need for the firewall to do more.
10	Physical access	The countermeasures from module 05 apply.
11	Gateway server login control	The countermeasures from module 05 apply.
12	User access rights	The countermeasures from module 05 apply.

### 6.3 Encrypted web sockets communication

You should already have generated a self-signed SSL certificate and keys for your web server following the instructions in module 05.

Create a script with which to start websocketd like this:

```
pi@raspberrypi:~ $ vi start_secure_ws.sh
#!/bin/bash
cd /usr/lib/cgi-bin/gateway
websocketd --address=raspberrypi --port=8082 --ssl --sslcert=/etc/apache2/ssl/apache.crt --
sslkey=/etc/apache2/ssl/apache.key /usr/lib/cgi-bin/gateway/mesh_ws_adapter.py
```

Save the file and change its permissions with `chmod 755 start_secure_ws.sh`

### 6.4 Authentication

Create user credentials as described in module 05 section 3.2.2.3 and then continue here.

Secure the mesh network control web application directory with a <Directory> entry in the virtual host config which we have enabled for SSL use:

```
sudo vi /etc/apache2/sites-enabled/000-default-ssl.conf

# inside the configuration block which starts with...

<IfModule mod_ssl.c>

    <VirtualHost _default_:443>

# add this Directory entries
```

```
<Directory /var/www/html/mesh/>

    # Choose authentication protocol

    AuthType Basic

    # Define the security realm

    AuthName "Mesh Control"

    # Location of the user password file

    AuthUserFile /etc/apache2/auth.users

    # Valid users can access this folder and no one else

    Require valid-user

</Directory>
```

Restart apache with `sudo service apache2 restart`

Try to access the gateway application at <https://raspberrypi/mesh>. You will now be challenged to enter a valid user ID and password.

## 6.5 Reverse proxying and the mesh gateway

### 6.5.1 Web server configuration changes

After taking the generally applicable actions defined in module 05, complete the steps required to use reverse proxying for the mesh gateway as follows:

Edit `/etc/apache2/sites-enabled/000-default-ssl.conf` and add or set the following properties:

```
SSLProxyEngine on

ProxyRequests Off

ProxyPass "/ws/" "wss://raspberrypi:8082/"

ProxyPass "/camera/" "http://raspberrypi:8081/"
```

Add the following Location blocks, which will ensure that any requests which will be forwarded to the web sockets daemon or to the webcam *motion* daemon are also subject to authentication.

```
<Location "/ws/">

    AuthType Basic

    AuthName "web sockets"

    AuthUserFile /etc/apache2/auth.users

    Require valid-user

</Location>

<Location "/camera/">

    AuthType Basic
```

```
AuthName "webcam authentication"

AuthUserFile /etc/apache2/auth.users

Require valid-user

</Location>
```

Restart the apache web server.

### 6.5.2 Adjusting the gateway application

The gateway application needs two small changes. The URL to be used for web sockets must use the secure protocol scheme “wss://” rather than “ws://” and instead of addressing the websocketd port directly, should now use a URL which will be matched by the reverse proxy configuration and forwarded.

Edit the JavaScript file `/var/www/html/mesh/js/constants.js`. Comment out the current `WS_SERVER` constant value and uncomment the alternate secure URL value so that the secure web sockets protocol WSS is used by the mesh network control application.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ vi /var/www/html/mesh/js/constants.js
// let WS_SERVER = 'ws:// raspberrypi:8082/';
let WS_SERVER = 'wss:// raspberrypi:443/ws/';
```

In addition, the URL used for streaming images from a connected webcam must be changed. In the `constants.js` file change the value of the `webcam_source` constant to:

```
let webcam_source = "<img src='http://raspberrypi:8081?action=stream' alt='--- webcam feed not found ---'/>"
```

Make sure you use the appropriate host name or IP address for your setup.

Clear the browser cache and access the application web page over https. Enable the webcam stream which should now work via the reverse proxy.

## 6.6 Task - Securing the Bluetooth Devices

To further secure Bluetooth mesh devices we need a component which was referred to as a *Bluetooth firewall* in module 05. The Bluetooth firewall will allow us to restrict access to selected addresses only.

### 6.6.1 Task - The Bluetooth firewall

Your next task is to add address filtering to the `i2mn` and `mn2i` paths. Create a file called `/usr/lib/cgi-bin/gateway/firewall_config.json` and populate it with:

```
{
  "enabled": true,
  "i2mn_address_acceptlist": [ "C001", "C011", "C012", "C013", "C014", "C021", "C022",
    "C023", "C024"],
  "mn2i_address_acceptlist": [ "C002", "C002"]
}
```



Copy the file `bluetooth_firewall.py` from `implementation\solutions\mesh\gateway\bluetooth_mesh_api` to `/usr/lib/cgi-bin/gateway` on your Raspberry Pi gateway device. Set ownership and permissions as for other Python files in that directory.

#### 6.6.1.1 Integrating the firewall within the I2MN path

We shall now modify the HTTP adapter in the `i2mn` path so that it checks with the firewall before proceeding to complete processing of a request. Edit `do_send_mesh_message.py` and add the import statement and new function shown here:

```
#!/usr/bin/env python3
#
# Invoke over HTTP with a suitable JSON object containing args which select and inform the
# mesh message type to be sent
# by the gateway_i2mn_node script.
#
#####
import sys
sys.path.insert(0, '.')
import json
import os
import sys
import cgi
import constants
import gateway_i2mn_node
import bluetooth_firewall

fo = open("http_log.txt", "a")

def log(line):
    fo.write(line+"\n")
    fo.flush()

dst_addr = None
state = None
rc = None
result = {}

def result_cb(rc):
    result['result'] = rc
    print(json.JSONEncoder().encode(result))

def firewall_allows(addr):
    if not bluetooth_firewall.i2mn_address_allowed(addr):
        print('Status: 400 Bad Request')
        print()
        print('Status-Line: HTTP/1.0 400 Bad Request')
        print()
        return False
    else:
        return True
```

Now add conditional checks against the firewall using the new `firewall_allows` function and return an HTTP response of 400 (Bad Request) if the firewall does not allow the use of the destination address received in the client's JSON request object. We also need to send a content type header of `application/json` only when returning a JSON object so we need to adjust that aspect of the code as well.

```

def json_content_type():
    print("Content-Type: application/json;charset=utf-8")
    print()

if 'REQUEST_METHOD' in os.environ:
    args = json.load(sys.stdin)
    if (os.environ['REQUEST_METHOD'] != 'PUT'):
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
    else:
        log("PUT")
        if (not "action" in args):
            json_content_type()
            result['result'] = constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif (args["action"] != constants.ACTION_GENERIC_ON_OFF_SET_UNACK and
args["action"] != constants.ACTION_LIGHT_HSL_SET_UNACK):
            json_content_type()
            result['result'] = constants.RESULT_ERR_NOT_SUPPORTED
            print(json.JSONEncoder().encode(result))
        elif (args["action"] == constants.ACTION_GENERIC_ON_OFF_SET_UNACK):
            log("generic on off set unack action")
            if (not "state" in args or not "dst_addr" in args):
                json_content_type()
                result['result'] = constants.RESULT_ERR_BAD_ARGS
                print(json.JSONEncoder().encode(result))
            else:
                action = args["action"]
                log(constants.ACTION_GENERIC_ON_OFF_SET_UNACK+ " preparing
arguments")

                dst_addr = args["dst_addr"]
                if firewall_allows(dst_addr):
                    state = args["state"]
                    result['dst_addr'] = dst_addr
                    result['state'] = state
                    log("calling gateway_i2mn_node.send_onoff()")
                    rc = gateway_i2mn_node.send_onoff(dst_addr, state,
result_cb)

                    result['result'] = rc
                    if (rc != constants.RESULT_OK):
                        json_content_type()
                        print(json.JSONEncoder().encode(result))
                    else:
                        log("firewall denied use of address "+dst_addr)
                elif (args["action"] == constants.ACTION_LIGHT_HSL_SET_UNACK):
                    log("light HSL set unack action")
                    if (not "h_state" in args or not "s_state" in args or not "l_state"
in args or not "dst_addr" in args):
                        result['result'] = constants.RESULT_ERR_BAD_ARGS
                        json_content_type()
                        print(json.JSONEncoder().encode(result))
                    else:
                        action = args["action"]
                        log(constants.ACTION_LIGHT_HSL_SET_UNACK+ " preparing
arguments")

                        dst_addr = args["dst_addr"]
                        if firewall_allows(dst_addr):
                            h_state = args["h_state"]
                            s_state = args["s_state"]
                            l_state = args["l_state"]
                            result['dst_addr'] = dst_addr
                            result['h_state'] = h_state
                            result['s_state'] = s_state
                            result['l_state'] = l_state
                            log("calling gateway_i2mn_node.send_light_hsl_set()")
                            rc = gateway_i2mn_node.send_light_hsl_set(dst_addr,
h_state, s_state, l_state, result_cb)

                            result['result'] = rc
                            if (rc != constants.RESULT_OK):
                                json_content_type()
                                print(json.JSONEncoder().encode(result))
                            else:
                                log("firewall denied use of address "+dst_addr)

```

```

else:
    json_content_type()
    result['result'] = constants.RESULT_ERR_NOT_SUPPORTED
    print(json.JSONEncoder().encode(result))
log("exiting script")
else:
    print("ERROR: Not called by HTTP")

```

Next we need to modify the web application so that it handles the possibility of an HTTP status 403 (Forbidden) response. Update the sendMeshMessage function in gateway.js so that it looks like this:

```

function sendMeshMessage(json) {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        console.log(this.responseText);
        if (this.readyState == 4) {
            if (this.status == 400) {
                message("Requested action was not carried out by the gateway");
                alert("Requested action was not carried out by the gateway");
            } else if (this.status == 200) {
                result = JSON.parse(this.responseText);
                message(result_string(result.result));
            }
        }
    };

    var target = "do_send_mesh_message.py";

    xhttp.open("PUT", CGI_ROOT+target, true);
    xhttp.setRequestHeader('Content-type','application/json; charset=utf-8');
    xhttp.send(json);
}

```

Comment: The HTTP Status value 403 means Forbidden. This seems a reasonable status code to use when the firewall denies access. But is this wise from a security perspective? Informing the client that its request has been forbidden could be interpreted as informing the client that there is something of value being protected there and signal to a hacker that this is an area worthy of further work in pursuit of an exploit. This issue is called [information leakage](#).

#### 6.6.1.2 Testing the firewall within the I2MN path

Clear your browser cache and reload so that the latest JavaScript is downloaded.

On the gateway device the firewall\_config.json file which is supplied in the solutions folder looks like this:

```

{
    "enabled": true,
    "i2mn_address_acceptlist": [ "C001", "C011", "C012", "C013", "C014", "C021", "C022",
    "C023", "C024"],
    "mn2i_address_acceptlist": [ "C002", "C003"]
}

```

All addresses used by the mesh network control web application are allowed except for the special *all nodes* address of 0xFFFF. Try to send both a *generic on off set* message and a *light HSL set* message from the UI to the All Nodes address. Tail the log file http\_log.txt during these tests. You should see messages like these whenever the firewall denies access to an address:

```
PUT
```

```
generic on off set unack action
generic_onoff_set_unack preparing arguments
firewall denied use of address FFFF
exiting script
-----
PUT
light HSL set unack action
light_hsl_set_unack preparing arguments
firewall denied use of address FFFF
exiting script
```

#### 6.6.1.3 Integrating the firewall within the MN2I path

We shall now modify the mesh websocket adapter in the mn2i path so that the firewall is checked before allowing a client to subscribe to an address.

Modify the mesh\_ws\_adapter.py script and import the bluetooth\_firewall script and implement a function which will be used to check the firewall for permission to subscribe to a particular address.

```
import bluetooth_firewall

sub_addresses = set()

fo = open("ws_log.txt", "a")
def wslog(line):
    fo.write(line+"\n")
    fo.flush()

def firewall_allows(addr):
    if not bluetooth_firewall.mn2i_address_allowed(addr):
        print('Status: 403 Forbidden')
        print()
        print('Status-Line: HTTP/1.0 403 Forbidden')
        print()
        return False
    else:
        return True
```

Now modify the main processing loop to include a check against the firewall.

```
        if (subscription_control["action"] == "subscribe"):
            dst_addr = subscription_control["dst"].upper()
            if firewall_allows(dst_addr):
                sub_addresses.add(dst_addr)
                wslog(tid+" subscribed:"+str(sub_addresses))
            # else silently ignore
            else:
                wslog(tid+" firewall denied subscribe request to "+dst_addr)
```

#### 6.6.1.4 Testing the firewall within the MN2I path

Edit the firewall\_config.json file so that only address C002 may be subscribed to.

```
{
    "enabled": true,
    "i2mn_address_acceptlist": [ "C001", "C011", "C012", "C013", "C014", "C021", "C022", "C023", "C024"],
    "mn2i_address_acceptlist": [ "C002" ]
}
```

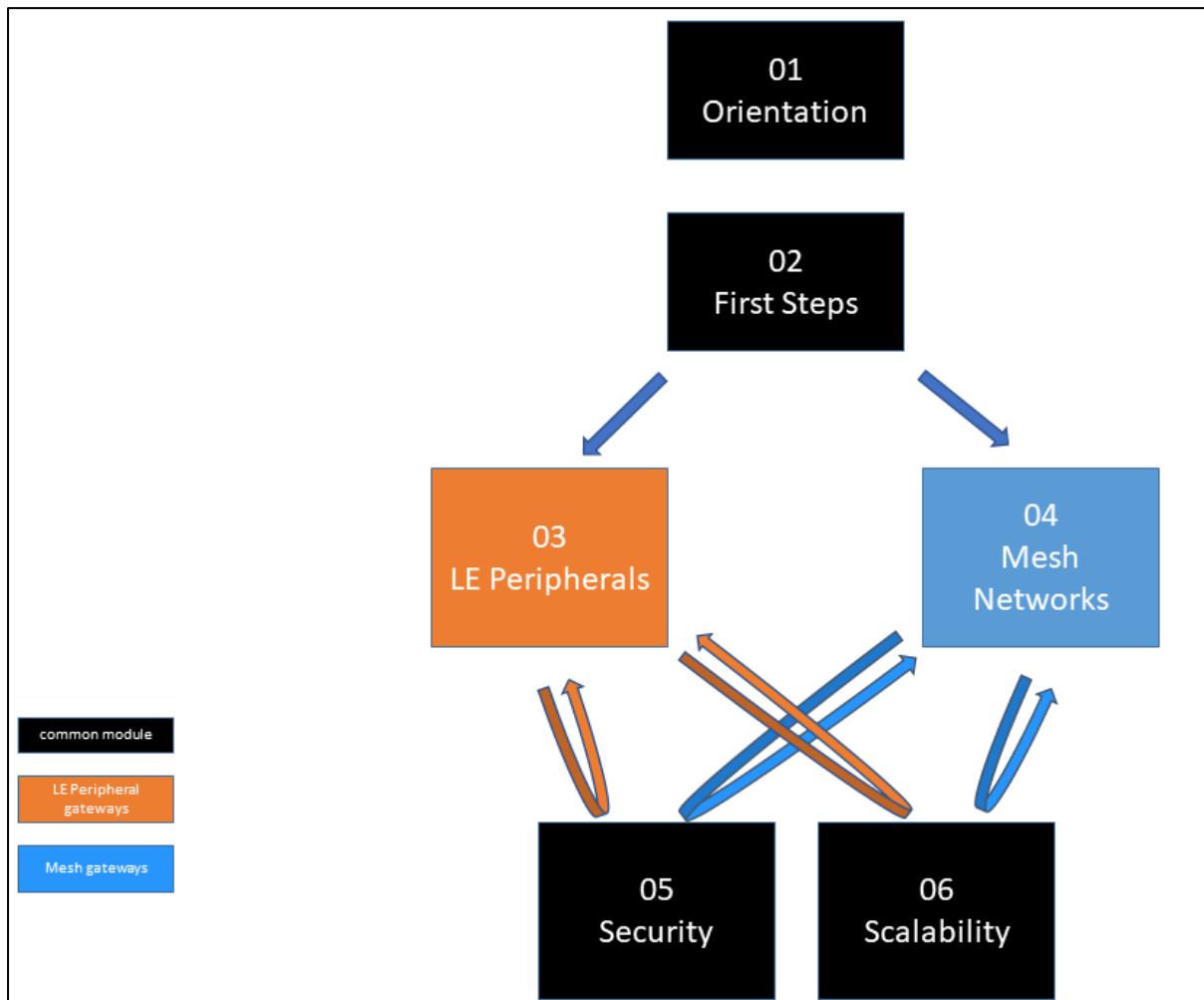
With your secured websocket daemon running, reload the mesh network control web page in your browser. Subscribe to the indoor temperature (address C002). This should work and if your node is publishing sensor readings to that address you will start to see values received.

Next, subscribe to the outdoor temperature (address c002). You will not see an error in the web UI but in the ws\_log.txt log file you will see evidence that the request was denied:

```
3069299408 RX> {"action":"subscribe","dst":"C003"}  
3069299408 firewall denied subscribe request to C003
```

## 7. Scalability

Module 06 covers the topic of scalability for both LE Peripheral gateways and gateways for Bluetooth mesh. Proceed now to study module 06.



### 7.1 Additional scalability issues relating to gateways for Bluetooth mesh

There are a few additional issues relating to scalability and gateways for Bluetooth mesh that are worth mentioning here, one of which you can take practical steps to address.

#### 7.1.1 BlueZ and concurrency limits with mesh application nodes

##### 7.1.1.1 BlueZ tokens and concurrency

A mesh application node must attach to the network before it can send or receive messages and attaching requires the token which it was issued to the node when it was provisioned to be presented for authentication purposes. BlueZ's bluetooth-meshd will only allow one application to attach at a time with a given token value though. In the i2mn direction, an instance of the node application gateway\_i2mn\_node.py must be running to service a gateway client HTTP request. The instance is launched via the web server's common gateway interface and executes in an operating system process. In the mn2i direction when a new websocket connection is requested by a gateway client, websocketd forks a new process in which to execute the mesh\_ws\_adapter.py script which itself runs the node application gateway\_mn2i\_node.py.

### 7.1.1.2 Exploring BlueZ tokens and concurrency

With websocketd running, visit the mesh control application in your webbrowser or reload the page if it's already open. You'll see websocketd indicate that a connection has been established and if you run `ps -ef | grep web` in another terminal, you'll see websocketd running your script.

Now start another browser instance in a new window. You should see the message "Error: concurrency limit reached" at the top of the page.

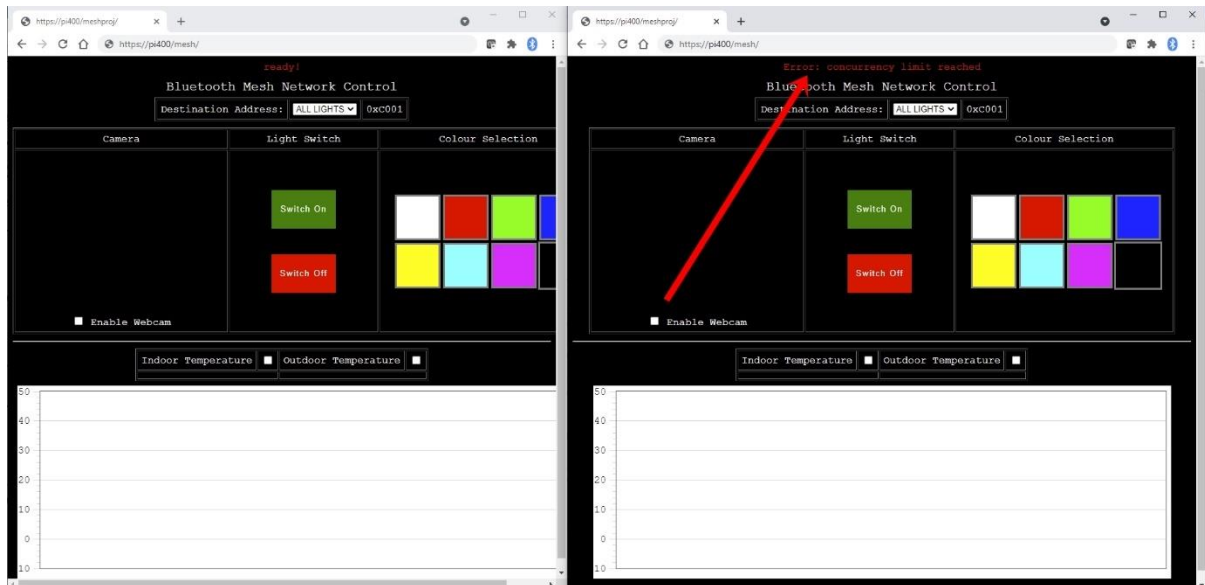


Figure 24 - Two browser instances, one showing the concurrency limit reached error

Open the developer console and you'll see

```
message received: "{\"result\": 6, \"message\": \"org.bluez.mesh.Error.AlreadyExists: Attach failed\"}"
gateway.js:209 The websocket connection has been closed
```

The message attribute of the JSON response object contains "org.bluez.mesh.Error.AlreadyExists: Attach failed". This is straight from the D-Bus call to the BlueZ mesh API and indicates that an attempt to attach to the network using a token value which another application has already attached with has been rejected.

### 7.1.1.3 Increasing mesh node application concurrency

To allow more than one mesh operation to be handled in each direction, `gateway_i2mn_node.py` and `gateway_mn2i_node.py` each need to be provisioned again perhaps multiple times. Each time you provision, using the *join* procedure you used in 4.5.5.1.6 and 4.5.5.3.3 you will be allocated a new token value. All token values associated with this code are valid and known to bluetooth-meshtd. Therefore, with a set of token values, it's possible to modify the node application code so that a token value is selected from the full set of tokens and used to attach. In this way, as long as each node application uses a distinct token value, a greater degree of concurrency becomes possible.

In general terms, what's needed is a token value pooling system that allocates an unused token to an application on request and locks that value as *already in use*, releasing it when the token is returned to the pool by the application or perhaps after a time limit. There are many ways to

approach the implementation of a token pool. We'll take a very simple but fit for purpose approach for our gateway.

You should already have copied the files `i2mn_config.json` and `mn2i_config.json` to your gateway directory `/usr/lib/cgi-bin/gateway`. Review these files. You'll find that they're identical and currently contain only a string array property called *tokens* which contains a single element with the empty string as a value.

```
{
    "tokens" : [
        ""
    ]
}
```

These configuration files will be used as a container for the token values allocated to our node applications through provisioning them multiple times.

Open `gateway_i2mn_node.py` in your editor. You may already have noticed the following code which is not yet fully used:

```
# Node token housekeeping
token = None
token_inx = 0
tokens = None

# and
with open(I2MN_CONFIG) as f:
    config = json.load(f)
    tokens = config["tokens"]
```

Find the `getToken()` function. Note the hard coded token value so you do not lose it and then and update the function it so it looks like this:

```
def getToken():
    global tokens
    global token_inx
    tid = str(threading.current_thread().ident)
    token = tokens[token_inx]
    log(tid+" Selected token "+token)
    return token
```

This code now takes a token value from the array which was loaded from the `i2mn_config.json` file using the current value of `token_inx`.

Edit your `i2mn_config.json` file and insert the currently used token value into the `tokens` array.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ cat i2mn_config.json
{
    "tokens" : [
        "5d54b6aff549a36a"
    ]
}
```

Review the function `attach_app_error_cb` which is the callback functioned that gets executed when an attempt to attach fails.



```
def attach_app_error_cb(error):
    tid = str(threading.current_thread().ident)
    log(tid+" attach_app_error_cb "+str(error))
    global mainloop
    global token_inx
    global next_action
    token_inx = token_inx + 1
    log(tid+' token_inx=' + str(token_inx))
    if (token_inx < len(tokens)):
        # token was already in use so retry with next in tokens list
        log(tid+' retrying')
        try_to_attach_and_then(next_action)
    else:
        log(tid+' Failed to register application: ' + str(error)+""")
        finished()
        result_callback_function(constants.RESULT_ATTACH_FAILED)
```

This code

- assumes that the previous attempt to attach failed for a reason such as the token already being in use and so tries to attach again using the next token from the tokens array, if one is available
- if the value of token\_inx indicates that all tokens have been tried then the overall attempt to attach fails and the HTTP handler script dp\_send\_mesh\_message.py is notified by callback.

Using your provisioner device and the mesh-cfgclient tool, provision the gateway\_i2mn\_node.py application again, running `./gateway_i2mn_node.py join` on the gateway node and then add the allocated token value to the i2mn\_config.json file.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ./gateway_i2mn_node.py join
PromptStatic ( static-oob )
Enter 16 octet key on remote device: 8ba4b6be784289a825467aefb378d2b7

Node provisioned OK - token=0x6438a5b9058ecd57
pi@raspberrypi:/usr/lib/cgi-bin/gateway $

pi@raspberrypi:/usr/lib/cgi-bin/gateway $ vi i2mn_config.json

pi@raspberrypi:/usr/lib/cgi-bin/gateway $ cat i2mn_config.json
{
    "tokens" : [
        "5d54b6aff549a36a",
        "6438a5b9058ecd57"
    ]
}
```

Make a note of the unicast address that mesh-cfgclient allocated.

```
[mesh-cfgclient]# provision 055AD7B422EC424FAF418FA6F73B5120
Provisioning started
Request hexadecimal key (hex 16 octets)
[[mesh-agent]# ] Enter key (hex number): 8ba4b6be784289a825467aefb378d2b7
Assign addresses for 1 elements
Provisioning done:
Mesh node:
    UUID = 055AD7B422EC424FAF418FA6F73B5120
    primary = 00e5

    elements (1):
```

Continuing to use mesh-cfgclient on your provisioner device, configure the newly provisioned node in exactly the same way as you did previously but remembering to use the new unicast address in commands.

You should now increase the concurrency of the mn2i node application in the same way. Edit `gateway_mn2i_node.py` and replace `getToken()` with the new version of the function exactly as used in the i2mn node application. The `attach_app_error_cb` function is already in the right state to

allocate tokens from the tokens array. Provide an additional token value by provisioning gateway\_mn2i\_node.py again and configuring as you did originally in 4.5.5.3.3, remembering to use the new unicast address. Add the new token value to mn2i\_config.json.

Start websocketd. Launch your browser and access the mesh control application. Launch another browser instance and access the web application. This time the websocket connection will be established and no concurrency related error message will appear. Launch a third browser window and you'll see the error.

Using the Linux command ps, find the process ID of websocketd and then find the child processes it has forked:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ps -ef|grep web
pi          6901   6900   0 15:05 pts/0    00:00:00 websocketd --port=8082 --ssl --
sslcert=/etc/apache2/ssl/apache.crt --sslkey=/etc/apache2/ssl/apache.key /usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py
pi          6923   1275   0 15:06 pts/1    00:00:00 grep --color=auto web

pi@raspberrypi:/usr/lib/cgi-bin/gateway $ ps -ef|grep 6901
pi          6901   6900   0 15:05 pts/0    00:00:00 websocketd --port=8082 --ssl --
sslcert=/etc/apache2/ssl/apache.crt --sslkey=/etc/apache2/ssl/apache.key /usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py
pi          6908   6901   1 15:05 pts/0    00:00:00 /usr/bin/python3 /usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py
pi          6915   6901   2 15:06 pts/0    00:00:00 /usr/bin/python3 /usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py
```

You should see two processes running the mesh\_ws\_adapter.py script, each servicing a different web client.

That's it! If you made it this far, congratulations. You have a working and sufficiently secure mesh gateway and an understanding of some of the key issues relating to scalability. If you have hit problems, check the troubleshooting section below and the solution code.

## 8. Troubleshooting

This section contains a collection of troubleshooting tips.

### 8.1 Issue: It's not working

This is a little vague but if what we mean is that on sending a request over HTTP, the expected action does not occur and there are no obvious errors then here are some things to check:

1. Is your logic correct? Did you recently comment out a function call intending that this be a temporary change but forget to reinstate the call?
2. Are runtime errors occurring? Watch `/var/log/apache2/error.log` for errors when you execute the test. Maybe your Python indentation is not correct or some other Python language issue is breaking your code. Maybe something is going wrong with D-Bus or your interaction with the bluetooth-meshd daemon. Enable tracing in bluetooth-meshd by editing `/lib/systemd/system/bluetooth-mesh.service` (sudo vi `/lib/systemd/system/bluetooth-mesh.service`), adding `-ndb` to the end of the `ExecStart` property and restarting the daemon (sudo service bluetooth-mesd restart). Then tail `/var/log/syslog` and run your test again.

Don't forget that your Python scripts perform logging to simple text files. Tail those files (`i2mn_log.txt`, `mn2i_log.txt`, `http_log.txt` and `ws_log.txt`).

3. Is your target node provisioned and configured properly? Common mistakes include forgetting to bind an appkey to each model or forgetting to subscribe each model to the right group addresses.

### 8.2 Issue: mesh-cfgclient is waiting to connect to bluetooth-meshd

```
pi@raspberrypi:~ $ mesh-cfgclient
Mesh configuration loaded from /home/pi/.config/meshcfg/config_db.json
Waiting to connect to bluetooth-meshd...
```

This means the bluetooth-meshd daemon is not running. Start it with your `mesh.sh` script as defined in section 5.1.4.

### 8.3 Issue: dbus-org.bluez.mesh.service not found

This issue may be reported when running a Python script or other application which attempts to communicate with the bluetooth-meshd daemon via the D-Bus service. It happens when bluetooth-meshd is not running.

```
pi@raspberrypi:~/bluez-5.58/test $ ./test-mesh
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/dbus/bus.py", line 175, in activate_name_owner
    return self.get_name_owner(bus_name)
  File "/usr/lib/python3/dist-packages/dbus/bus.py", line 361, in get_name_owner
    's', (bus_name,), **keywords)
  File "/usr/lib/python3/dist-packages/dbus/connection.py", line 651, in call_blocking
    message, timeout)
dbus.exceptions.DBusException: org.freedesktop.DBus.Error.NameHasNoOwner: Could not get
owner of name 'org.bluez.mesh': no such name

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "./test-mesh", line 1201, in <module>
    main()
  File "./test-mesh", line 1165, in main
    "/org/bluez/mesh"),
  File "/usr/lib/python3/dist-packages/dbus/bus.py", line 241, in get_object
```

```
follow_name_owner_changes=follow_name_owner_changes)
File "/usr/lib/python3/dist-packages/dbus/proxies.py", line 248, in __init__
self._named_service = conn.activate_name_owner(bus_name)
File "/usr/lib/python3/dist-packages/dbus/bus.py", line 180, in activate_name_owner
self.start_service_by_name(bus_name)
File "/usr/lib/python3/dist-packages/dbus/bus.py", line 278, in start_service_by_name
'su', (bus_name, flags)))
File "/usr/lib/python3/dist-packages/dbus/connection.py", line 651, in call_blocking
message, timeout)
dbus.exceptions.DBusException: org.freedesktop.systemd1.NoSuchUnit: Unit dbus-
org.bluez.mesh.service not found.
```

Start the bluetooth-meshd daemon with your mesh.sh script as defined in section 5.1.4.

## 8.4 Issue: Unexpected Warning: config file /home/pi/.config/meshcfg/config\_db.json not found

If you see this message on starting mesh-cfgclient and had already created your network then the likelihood is that you are running mesh-cfgclient on a device which is not acting as your Provisioner (the gateway device for example). Switch to the Provisioner device and try again.

## 8.5 Issue: Node is not responding to messages

Check that you have both provisioned and configured your target node(s). In particular ensure that an appkey has been bound to each model and that each model subscribes to the appropriate group addresses. Review the help information in the mesh-cfgclient config menu. You'll find a number of commands which allow you to check the configuration of a node. For example, to check that the models are subscribed to the right addresses:

```
[config: Target = 00e2]# sub-get 00e2 1000
Received ModelSubList (len 11)

Node 00e2 Subscription List status Success
Element Addr    00e2
Model ID        1000 "Generic OnOff Server"
Subscriptions:
                c001
                c011
                c021

[config: Target = 00e2]# sub-get 00e2 1307
Received ModelSubList (len 11)

Node 00e2 Subscription List status Success
Element Addr    00e2
Model ID        1307 "Light HSL Server"
Subscriptions:
                c001
                c011
                c021
```

## 8.6 Issue: Websocket problems

If you suspect your websocket is misbehaving, you may be able to find some clues by setting the logging level of the websocketd daemon to debug like this:

```
#!/bin/bash
websocketd --loglevel=debug --port=8082 /usr/lib/cgi-bin/gateway/mesh_ws_adapter.py
```

Even when using the default logging level, websocketd will output script errors such as this one:

```
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
```

```

bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | Traceback (most recent
call last):
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | File
"/usr/lib/python3/dist-packages/dbus/connection.py", line 607, in msg_reply_handler
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' |
*message.get_args_list())
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | File
"./gateway_mn2i_node.py", line 558, in attach_app_error_cb
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | on_error(rc)
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | File "/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py", line 20, in on_error
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | result['result'] =
rc
Wed, 16 Jun 2021 07:39:24 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1623825558642214632' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3100' | NameError: name 'rc' is
not defined

```

## 8.7 Issue: Inconsistent use of tabs and spaces in Python code

You're seeing errors indicating that you have inconsistent use of spaces and tabs in your Python code. For example:

```

Fri, 18 Jun 2021 10:03:15 +0100 | ACCESS | session | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' | CONNECT
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | Traceback (most recent
call last):
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | File "/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py", line 6, in <module>
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | import
gateway_mn2i_node
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | File
"./gateway_mn2i_node.py", line 512
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | def
process_message(self, source, dest, key, data):
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' |
^
Fri, 18 Jun 2021 10:03:15 +0100 | ERROR | stderr | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | TabError: inconsistent
use of tabs and spaces in indentation
Fri, 18 Jun 2021 10:03:15 +0100 | ACCESS | session | url:'http://pi400:8082/'
id:'1624006995078652357' remote:'192.168.0.59' command:'/usr/lib/cgi-
bin/gateway/mesh_ws_adapter.py' origin:'http://pi400' pid:'3663' | DISCONNECT

```

This is easily fixed using the Visual Studio Code editor. Simple press CTRL+, (Windows, Linux) or ⌘+, (Mac) and then in the field which appears type "Convert indentation to Tabs".

You can also have spaces and tabs visually indicated with this Visual Studio Code setting:

```
"editor.renderWhitespace": "all",
```

You can see the tab characters in this example:

```
45  wslog(tid+" starting loop")
46  while keep_going == 1:
47      try:
48          line = sys.stdin.readline()
49          if len(line) == 0:
50              # means websocket has closed
51              keep_going = 0
52              wslog(tid+" Websocket closed")
53              # tell the mesh node application to exit
54              gateway_mn2i_node.finished()
```

## 9. Close

This brings us to the end of this module. It is hoped that you have learned all that you wanted to learn about Bluetooth internet gateways for mesh networks and feel ready to include Bluetooth mesh in your Internet of Things (IoT) solutions.

If you are interested in Bluetooth internet gateways for Bluetooth LE Peripherals, proceed now to study module 03.

# Appendix A - Resources

## A1 Example Provisioning and Configuration Summary

```
pi@raspberrypi:~ $ mesh-cfgclient

[mesh-cfgclient]# discover-unprovisioned on

[mesh-cfgclient]# discover-unprovisioned off

[mesh-cfgclient]# provision 7CE04BAEA73211EB951CC30E4F6696CF

[mesh-cfgclient]# menu config

[mesh-cfgclient]# target 00ac

[config: Target = 00ac]# timeout 5

[config: Target = 00ac]# composition-get 0

[config: Target = 00ac]# appkey-add 0

[config: Target = 00ac]# bind 00ac 0 1000

[config: Target = 00ac]# bind 00ac 0 1307

[config: Target = 00ac]# sub-add 00ac c001 1000
[config: Target = 00ac]# sub-add 00ac C011 1000
[config: Target = 00ac]# sub-add 00ac C021 1000

[config: Target = 00ac]# sub-add 00ac c001 1307
[config: Target = 00ac]# sub-add 00ac C011 1307
[config: Target = 00ac]# sub-add 00ac C021 1307

[config: Target = 00ac]# pub-set 00ac C002 0 0 0 1100
```