Developer Study Guide

Bluetooth® Internet Gateways

Scalability

Release	:	2.0.1
Document Version:		2.0.0
Last updated	:	24th June 2021

Contents

1. Revision History

Version	Date	Author	Changes
1.0.0	18th November 2020	Martin Woolley Bluetooth SIG	Initial version
1.0.1	6th January 2021	Martin Woolley Bluetooth SIG	Added *enabled* configuration property to the Bluetooth firewall.
2.0.0	24th June 2021	Martin Woolley Bluetooth SIG	**Release** 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. **Document** This document is new in this release

2. Introduction

This module covers the topic of scalability as it applies to Bluetooth internet gateways. It applies to both gateways for LE Peripheral devices and gateways for Bluetooth mesh networks since many of the issues are the same. Where an issue only applies to one of these gateway types, this is made clear in the text.

3. Scalability

3.1 Bluetooth Internet Gateways and Scalability Issues

Scalability is concerned with being able to increase the capacity of a system in some respect by adding more computing resource. In the world of web applications, it's usually concerned with the throughput of requests and the number of concurrent users.

3.1.1 Concurrent Users

A Bluetooth internet gateway is unlikely to need to handle very large numbers of concurrent users or to handle very high volumes of requests per second. But this is not impossible. Imagine a situation where a sensor is being watched concurrently by millions of users around the world. To meet this requirement would require a very large number of concurrent websocket connections to be active and delivering sensor data from a single Bluetooth device.

3.1.2 Concurrent Connected Bluetooth Devices

A more significant and common issue concerns the number of concurrently connected Bluetooth devices a gateway can accommodate. A typical Bluetooth LE device can handle a relatively small number of concurrent connections. The precise number is not specified in the Bluetooth core specification since this is an implementation detail but it's common for the maximum number of connections to be around eight. That's a small number in the context of a Bluetooth internet gateway, which may be required to support multiple concurrent users, interacting with a multitude of Bluetooth devices.

The maximum number of concurrent Bluetooth connections is therefore one possible key scalability limit in gateways. LE Peripherals as described in module 03 use connection-oriented communication and therefore are affected by this issue. Bluetooth mesh devices usually perform connectionless communication but may use connections when the GATT bearer is in use. The gateway for mesh devices developed in module 04 only uses connectionless communication however and so that particular mesh gateway design is not affected by this issue.

3.1.3 Serialisation of HCI Commands and Events

A Bluetooth LE stack consists of a host component and a controller component. We sometimes refer to the controller and associated hardware and driver as an adapter. The Raspberry Pi Zero W, Raspberry Pi 4 model B and the Raspberry Pi 400 all have a Bluetooth LE controller built into the main board. Bluetooth adapters (and their controller) sometimes take the form of USB dongles which can be plugged into a USB port, making it possible to add Bluetooth LE support to a device which does not have an integral Bluetooth controller.

Communication between the host layers of the stack and the controller takes place over a logical interface called the Host Controller Interface (HCI) which defines a series of *commands* that the host may send to the controller and *events* that the controller can send to the host. Physically, a number of different transports are defined in the Bluetoth core specification, including but not limited to USB and UART.

Critically, HCI communication is completely serial and therefore will inevitably become a bottleneck and constraint for the gateway when it comes to supporting concurrent users and communication with multiple devices.

All concurrent requests received by the gateway from its TCP/IP clients result in messages being exchanged with the d-bus daemon which passes them on to the BlueZ daemon (bluetoothd for LE Peripherals or bluetooth-meshd for mesh). The BlueZ daemon queues and serialises these requests and submits corresponding HCI commands one at a time to the controller. Events received over HCI from the controller are received serially by the host (BlueZ daemon) and dispatched back to the corresponding application via d-bus. Figure 1 shows this.

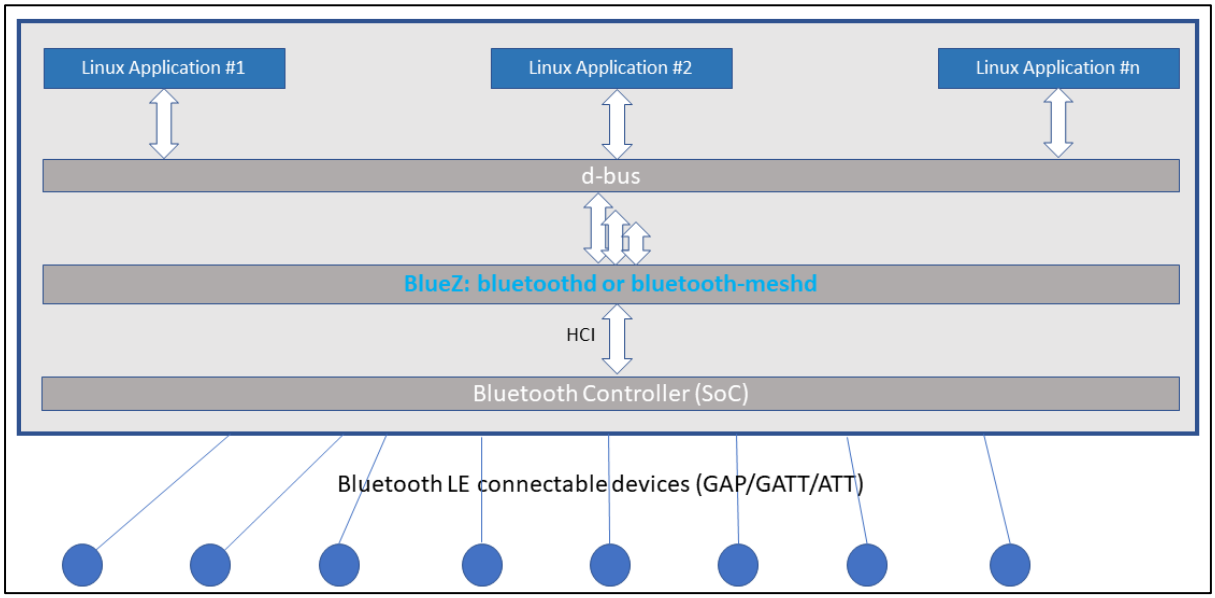

Figure 1 - Serialisation of HCI communication by BlueZ

3.1.4 Collisions

If multiple gateways or a gateway with multiple Bluetooth controllers are all physically situated close to each other (where *close* means within direct radio range), there is a risk that simultaneous or near-simultaneous transmissions by more than one gateway will cause *collisions* between packets in the air. Such packets will be lost but may be retrsnsmitted automatically, depending on stack implementation details and in the case of Bluetooth mesh, configuration choices.

3.1.5 BlueZ and Mesh Application Tokens

The implementation of Bluetooth mesh within BlueZ brings with it the concepts of *applications, tokens* and of *joining* a network. Neither of these are formal concepts from the Bluetooth mesh profile specification but it should be noted that only one BlueZ mesh application may join the mesh network with a given token value at a time. Depending on other aspects of the architecture and gateway implementation, this may be an issue which has a bearing on the subject of scalability.

3.2 Supporting more concurrent Bluetooth devices

There are several ways in which a gateway could support larger numbers of Bluetooth devices.

3.2.1 Bluetooth mesh

Bluetooth mesh uses connectionless Bluetooth communication and is message-oriented. It should be possible for a single Bluetooth internet gateway to support as many Bluetooth mesh nodes as a network can contain, or 32,767 to be precise. For situations where large numbers of devices must be supported and you have control over what those devices are, Bluetooth mesh is recommended.

Near-simultaneous, reliable communication with large numbers of Bluetooth mesh devices is affected by the issues in section 3.1 other than 3.1.2 and other issues beyond the scope of this topic, such as general network design and network density and device verbosity. In particular, issue 3.1.5 needs to be considered when implementing a gateway for a Bluetooth mesh network using BlueZ. The project in module 04 illustrates one possible response to this issue.

3.2.2 Bluetooth LE Peripherals

In the case of connectable Bluetooth LE devices *connections* are an integral part of the system and the source of our most limiting constraint.

3.2.3 Scalability Strategies

To address the issues described above, we need to include measures in our architecture which allow capacity to be added. There are at least two possibilities and each are described below with particular reference to connected LE Peripherals where the limited number of connections per controller is the most significant concern.

3.2.3.1 Multiple Bluetooth adapters

Some computers allow multiple Bluetooth adapters to be installed, perhaps using USB. Each distinct adapter should allow further concurrent Bluetooth device connections to be handled.

This approach is simple but limited and will impact adapter code.

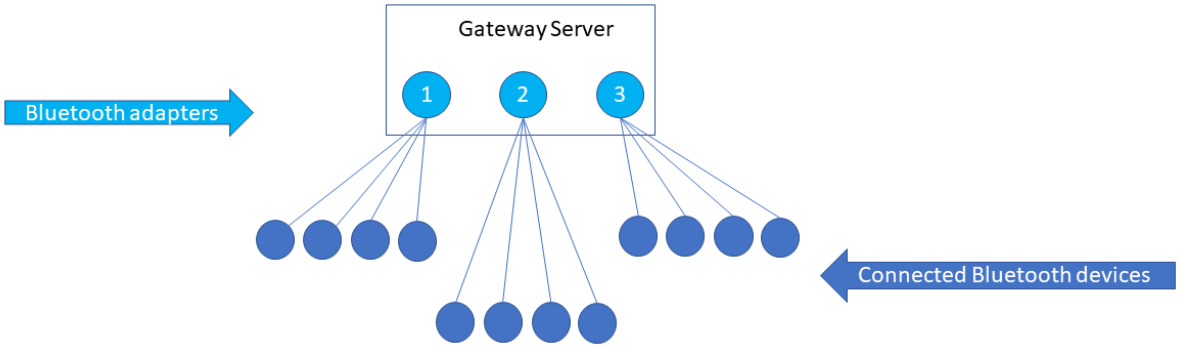

Figure 2 - A single gateway server with multiple Bluetooth adapters

3.2.3.2 Horizontal scaling with multiple gateway servers

A better way to approach the issue of concurrency and scalability is to take an approach which is very common in the world of large scale web applications. This involves placing a device or software service known as a load balancer in front of a *cluster* of web servers. The handling of requests from

clients is distributed across the individual web servers so that load is shared. Adding capacity can often be accomplished simply by adding further web servers to the cluster.

This approach can scale to very large levels, with large numbers of back end servers in the cluster across which load is balanced, and it is this approach which shall be discussed further here.

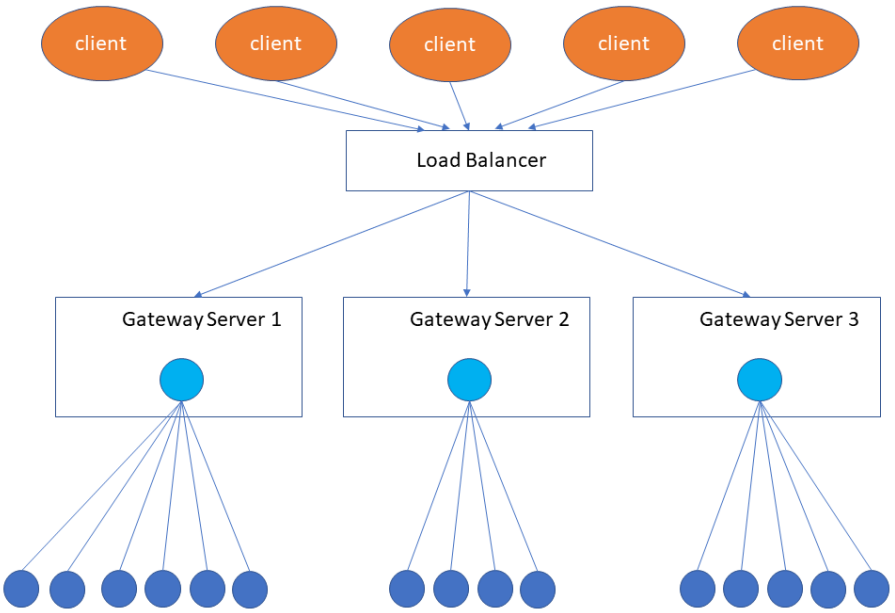

Figure 3 - Using a load balancer with multiple Bluetooth internet gateway servers

3.3 Load balancing and Bluetooth internet gateways

We start by considering a number of special issues which a load balancing solution for Bluetooth internet gateways that supports connection-oriented communication must accommodate.

3.3.1 Stickiness

Using load balancing with Bluetooth internet gateways involves an issue which the scalability solution must address if communication with the Bluetooth devices involves connections. In the greater majority of cases, a Bluetooth GAP peripheral device will support only one connection at a time and therefore, all communication with the device must be via the gateway server which has connected to it. It is not possible to adopt a simple, round-robin load balancing scheme because once a gateway server has been selected and its API used to cause it to connect to a particular Bluetooth device, all subsequent API calls which target that same Bluetooth device (e.g. to read or write characteristics), must be handled by the same gateway server within the cluster. Sets of operations relating to different Bluetooth devices may be load balanced across the cluster of gateway servers, but individual operations for the same Bluetooth device must “stick” to the same server which initially made the Bluetooth connection.

In the world of load balancers, routing requests to the same server within a cluster, based upon something at the application layer rather than a lower layer of the stack such as the client IP address, is called *persistence* and the result is a *sticky session*. See https://www.haproxy.com/blog/load-balancing-affinity-persistence-sticky-sessions-what-you-need-to-know/

Bluetooth mesh will typically use connectionless communication in the form of the *advertising bearer* and stickiness is not required.

3.3.2 Connect without scanning

The Bluetooth core specification defines a state machine for the link layer, describing the states a link may be in and the transitions between states that are permitted. It is permitted to transition from the standby state, straight into the connected state, for example. This is something you would want to do if you already knew the Bluetooth device address of the device you wished to connect to and knew it to be in range and available to connect to.

BlueZ 5.50 requires scanning to have taken place before a device can be connected to using the *Connect* method of DBus Device objects. However, an experimental function, *ConnectDevice* is available in BlueZ if it is started with the *--experimental* flag and its purpose is to allow a client to initiate connecting to a device without first scanning for it.

This is relevant to the subject of load balancing because it may be that device discovery is handled by one gateway server, but a request to connect to one of the devices that was discovered could be handled by another server. Its BlueZ stack may not yet have scanned and therefore not know of the device to be connected to.

It is desirable therefore that a load balancing solution allows individual gateway servers to connect to a Bluetooth device without previously having performed scanning.

3.3.3 Gateway load balancing requirements

The requirements for using load balancing with a Bluetooth internet gateway are as follows:

Table 1 - Load Balancer Requirements

Name	Details	Applies to Gateway Type(s)
connect without prior scan	The Bluetooth API used by gateway servers must allow connecting to a Bluetooth device, identified by *bdaddr* without first having discovered it through scanning.	LE Peripherals
scan using any server	Device discovery may be performed by any gateway server in the backend cluster, selected using any suitable algorithm such as *round-robin*.	LE Peripherals
bdaddr stickiness	Operations relating to a connected Bluetooth device must always be handled by the gateway server that owns that Bluetooth connection. This applies to both HTTP and websocket transported operations.	LE Peripherals

disconnect ends sticky session	On disconnecting from a Bluetooth device, the load balancing solution must release that Bluetooth device address from its table of sticky sessions.	LE Peripherals
Security	Load balancing must accommodate the same basic security requirements as were identified for a single server solution.	LE Peripherals Mesh

As indicated by Table 1, the connectionless nature of Bluetooth mesh communication makes the selection of a load balancer much easier than selecting one for use with gateways that handle LE Peripheral devices.

3.4 Security and load balanced Bluetooth gateways

When using load balancing, as depicted in Figure 3, there are a number of ways to approach the use of SSL for encryption of HTTP and websocket communication.

3.4.1 SSL Termination

In the first option, known as *SSL Termination*, SSL encryption is applied to requests and responses passing between clients and the load balancer only. Communication between the load balancer and each back end gateway server is then done in clear.

3.4.2 SSL Passthrough

In this approach, the end-points for SSL encryption are the clients and each backend server, with the load balancer passing encrypted packets through to backends without unencrypting them first.

3.4.3 Two-stage SSL

An alternative to SSL Passthrough is to make the load balancer an SSL endpoint, so that all traffic received from clients is unencrypted by the load balancer. But in contrast to SSL Termination, the load balancer establishes a distinct SSL session with each of its backend servers and so has to re-encrypt traffic from clients, before dispatching it over the new SSL connection to the selected backend.

3.4.4 Appraisal of SSL architectural options

Load balancing with application layer controlled sticky sessions can be complicated and requires the load balancer to apply rules which are based on the content of the communication from the client (e.g. URL, HTTP headers or payload). As such, the load balancer must decrypt traffic before it can apply its backend server selection algorithm and only in the simplest of cases can SSL Passthrough be used. With requirements such as those that apply to Bluetooth internet gateways, we must choose between SSL Termination and Two-stage SSL.

Two-stage SSL is more secure than SSL Termination. But this security comes at a price. Every server must handle SSL connection establishment and encryption and decryption, which can place a significant processing load on each server. Furthermore, every backend server must have a digital certificate and each certificate must be properly managed through its lifecycle.

Probably the most common approach is to use SSL Termination, with backend servers on a private local area network, isolated from other networks through network design measures which make it impossible to sniff the traffic between the load balancer and backend gateway servers.

3.5 Implementing load balancing with SSL Termination and sticky sessions

3.5.1 HAProxy

Loads balancers are not all the same and so a good understanding of their capabilities and your own requirements is important. HAProxy is a popular product with extensive capabilities. From a short review and some prototyping, it would appear that a load balanced Bluetooth internet gateway solution could be based upon HAProxy. Note that this is not a recommendation, but HAProxy is works well as a concrete example to consider how some of our requirements might be met.

The following extract from a HAProxy configuration, establishes an SSL enabled HTTP front end which listens on the standard port 443 and dispatches requests to either a cluster of three web servers which listen on port 8080 or a cluster of three websocketd daemons, each of which listens on port 9090.

Table 2 - HAProxy Configuration Extract

```
frontend https
  bind *:443 ssl crt /etc/apache2/ssl/apache.pem
  mode http
  default_backend gw-backend
  acl hdr_connection_upgrade hdr(Connection)  -i upgrade
  acl hdr_upgrade_websocket  hdr(Upgrade)     -i websocket
  use_backend bk-ws if hdr_connection_upgrade hdr_upgrade_websocket

backend gw-backend
   balance roundrobin
   mode http
   stick-table type string len 19 size 1k
   stick on "lua.parseRequest" table bk-ws
   server gw1 pi1:8080 check
   server gw2 pi2:8080 check
   server gw3 pi3:8080 check

backend bk-ws
  balance roundrobin
  stick-table type string len 19 size 1k
  stick on urlp(bdaddr)
  server ws1 pi1:9090 check
  server ws2 pi2:9090 check
  server ws3 pi3:9090 check
```

3.5.2 Load Balancing

The two lines that start “acl” in the frontend section detect the switch from HTTPS to a websocket taking place and result in the backend cluster named bk-ws being handed the work. In all other cases (i.e. HTTP operations), the backend cluster named gw-backend is used. Within each cluster, a *round robin* load balancing algorithm is used but with session stickiness.

3.5.3 Session Stickiness

Session stickiness, which is needed when we need to accommodate connections to Bluetooth LE Peripheral devices, is accomplished by using the bdaddr parameter in either the URL string associated with GET operations and when opening websockets or from within the JSON object in the message body, as used in HTTP PUT requests. In the latter case, extracting the bdaddr value is achieved by using a custom script, written in the Lua language, which HAProxy supports. URLs can be parsed using standard configuration options such as *urlp*.

A custom Lua script is as follows. Note that this is prototype code and not hardened against possible data format errors.

```
core.Alert("LUA script for BDADDR extraction loaded");

function get_bdaddr_from_url(payload)
    qs_start = string.find(payload,"?") + 1;
    qs_end = string.find(payload," ",qs_start);
    qs = string.sub(payload, qs_start,qs_end)
    bdaddr_param_start = string.find(qs,"bdaddr=");
    bdaddr_start = bdaddr_param_start + 7;
    bdaddr_end = bdaddr_start + 16;
    core.Alert(qs);
    core.Alert("bdaddr_start=" .. bdaddr_start);
    core.Alert("bdaddr_end=" .. bdaddr_end);
    bdaddr = string.sub(qs, bdaddr_start, bdaddr_end);
    core.Alert("URL contains bdaddr=<" .. bdaddr .. ">");
    return bdaddr;
end

function get_bdaddr_from_json(payload)
    json_start = string.find(payload,"{");
    if (json_start == nil)
    then
        return nil
    end
    json_end = string.find(payload,"}");
    -- {"bdaddr":"D6:01:BD:93:A8:3F"}
    json = string.sub(payload, json_start)
    bdaddr_key_inx = string.find(json,"\"bdaddr\"");
    bdaddr_colon_inx = string.find(json,":",bdaddr_key_inx);
    bdaddr_value_start = string.find(json,"\"",bdaddr_colon_inx) + 1;
    bdaddr = string.sub(json,bdaddr_value_start,bdaddr_value_start+16);
    return bdaddr;
end

function parseRequest(txn)
    core.Alert("HELLO from parseRequest");
    local payload = txn.req:dup()
    bdaddr = get_bdaddr_from_url(payload);
    if (bdaddr ~= nil) then
       core.Alert("URL bdaddr=" .. bdaddr);
       return bdaddr;
    end
    bdaddr = get_bdaddr_from_json(payload);

    if (bdaddr ~= nil) then
        core.Alert("JSON bdaddr=" .. bdaddr);
        return bdaddr;
    end
    return nil
end

-- register HAProxy "fetch"
core.register_fetches("parseRequest", parseRequest)
```

3.5.4 Application Impact

Gateway applications must ensure that all GET requests include a bdaddr parameter, except where stickiness is not required. This is limited to device discovery, from the various use cases our gateway implementation project addressed. When opening a websocket, the URL must also pass a bdaddr parameter. HTTP PUT requests must include a bdaddr parameter in the JSON object which is included. In this way, all relevant requests include a bdaddr value which can be used as the basis for sticky sessions and ensure that the right gateway is used for requests relating to previously established Bluetooth connections.

3.5.5 Releasing the sticky session

HAProxy has an API which can be used over a Unix socket or a TCP/IP socket. The API allows dynamic changes to be made, including the removal of entries from its session stickiness tables (known simply as Stick Tables in HAProxy).

A change based on the following code or similar needs to be incorporated in the gateway adapter script, *do_disconnect.py* to ensure the stick table entry for the device being disconnected is cleared. Note that this script has a hard-coded Bluetooth device address in it and is provided to illustrate the way such code would work.

```
#!/usr/bin/python
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('192.168.0.54', 9999)
sock.connect(server_address)
try:

    # Send HAProxy API data - NOTE THE NEWLINE CHARACTER!
    message = 'clear table bk-ws key 6B:99:FF:59:C6:A6\n'
    print message
    sock.sendall(message)

finally:
    print "Closing socket connection"
    sock.close()
```

3.5.6 Connect without Scanning

As described in 3.3.2, given the server which handles a device discovery request might not be the same server which handles a subsequent request to connect to a specific device that was discovered, on Linux the experimental ConnectDevice method must be used. The Python Bluetooth API would need to be changed to use this approach if deployed within a load balanced solution. Suitable code might look like this:

```
def connect_directly(bdaddr):
    bus = dbus.SystemBus()
    selected_adapter_addr = ""
    selected_adapter_path = bluetooth_constants.BLUEZ_NAMESPACE +
bluetooth_constants.ADAPTER_NAME
    # acquire the adapter interface so we can call its methods
    adapter_object = bus.get_object(bluetooth_constants.BLUEZ_SERVICE_NAME,
selected_adapter_path)
    adapter_interface=dbus.Interface(adapter_object, bluetooth_constants.ADAPTER_INTERFACE)
    try:
        device_path = adapter_interface.ConnectDevice({"Address" : bdaddr , "AddressType" :
"random" })
    except Exception as e:
        type, value, traceback = sys.exc_info()
        print(e.message)
        return bluetooth_constants.RESULT_EXCEPTION
    return bluetooth_constants.RESULT_OK
```

And code with which to test the connect_directly function might look like this:

```
print("Connecting....")
test_device = "D9:64:36:0E:87:01"
try:
    path = connect_directly(test_device)
    print("Done - path="+str(path))
except Exception as e:
    // an exception message of "Already Exists" means the device is already known to
    // BlueZ. To connect to the device you must now call the standard Connect method.
    type, value, traceback = sys.exc_info()
    print("Exception: " + str(e.message))
```

3.5.7 HTTP Server Impact

The configuration of the backend HTTP servers will be largely unaffected by the inclusion of a frontend load balancer. However, the reverse proxying configuration we used as part of the security solution will not be needed, since the load balancer is effectively providing that capability, allowing communication on a single port and forwarding requests on another.

3.5.8 More on HAProxy

A full tutorial on HAProxy is beyond the scope of this study guide but it is hoped that this section has given some useful insight into how a multi-server, scalable Bluetooth internet gateway solution could be created. For further information, consult the HAProxy web site.

Figure 4 - Three rack mounted Raspberry Pi Zeroes acting as a load balanced Bluetooth internet gateway

4. Close

Further scalability measures may still need to be considered, depending on the type of gateway. Return now to module 03 or 04 and complete the remainder of those modules.