Project Work in Mobile Systems
Bluetooth LE Script Runner

Lorenzo Felletti

2022

Contents

1 Introduction

The project work consists in an Android application implementing a GATT Server (taking the role of BLE Peripheral), and a Python application acting as a GATT Client (taking the role of BLE Central).

The server can communicate to the client a script it wants to be invoked on the client machine through a service exposed by the server, exposing a characteristic whose value is the name of a script to be invoked.

The client is continuously searching for devices exposing the said service, shutting down after a configurable amount of time.

The script invocation is triggered by the server by sending a notification that the characteristic value has changed to the connected devices. When the clients receive the notification, they read the characteristic value and tries to execute the corresponding script.

If the server device goes out of range, or the server is shut down, the client disconnects and starts again the discovery.

2 Usage Scenarios

There are several possible scenarios in which such an infrastructure could be used.
For example, there could be an employee in a factory that by walking around the warehouse could start a variety of analysis, or command some machinery in their vicinity to perform some kind of actions, and may do this in a "batch manner", i.e. by commanding the same task to **all** the machines near they that are able to perform it at once.

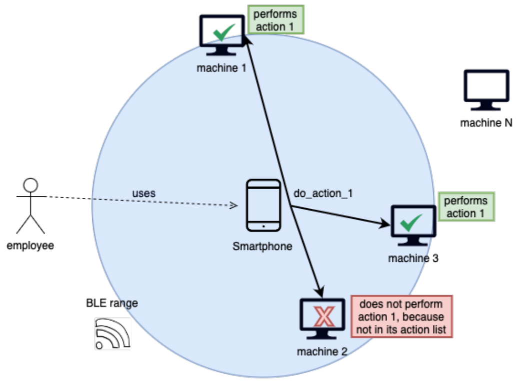


Figure 1: Possible usage scenario. The employee can command nearby machines to perform some action, and all the machines within the range performs it if the action is within their action's list.

Bluetooth LE is especially useful when your goal is to communicate with nearby devices, such as in the depicted scenario. In fact, its relatively small connection range is perfect for communicating only with nearby machines.
When you need to communicate only to devices near to you, BLE, compared with other possible communicating technologies with bigger ranges, avoids bloating the bandwidth with "wasted" communications transmitted to devices you need then to manually filter out, and moreover it avoids having to manually manage the calculation of what devices are in fact near to you.
Last but not least, BLE power consumption is significantly lower when compared to other wireless communication technologies such as Wi-Fi.

3 Bluetooth LE Architecture

Bluetooth Low Energy (or Bluetooth LE, or BLE) is a wireless personal area network (PAN) designed and marketed by Bluetooth SIG, and independent of classic Bluetooth.

The original specification was developed by Nokia in 2006 under the name Wibree which was integrated into Bluetooth 4.0 in December 2009 as Bluetooth Low Energy.

Bluetooth Low Energy is intended to provide considerably *reduced power consumption* and cost while maintaining a similar communication range.

3.1 Functioning

Bluetooth Low Energy has two ways of communicating. The first one is using **advertisements**, where a BLE peripheral device broadcasts packets to every device around it. The receiving device can then act on this information or connect to receive more information. The second way to communicate is to receive packets using a **connection**, where both the peripheral and central send packets.

3.1.1 Advertising and Discovery

BLE devices are detected through a process based on **broadcast advertising packets**.

BLE devices broadcast some information to make any other near device know about its presence. The broadcasters are called **advertisers**, and devices that receive these advertisements are called **scanners**.

The advertising packets can be observed by all scanners, thus they are initially *connectionless packets.*

3.1.2 Communication

Bluetooth LE uses the Client/Server approach to communication, defining two roles: the **Peripheral** role (typically the server) and the **Central** (typically the client).

3.1.3 ATT and GATT Protocols

BLE makes use of two protocols: **Attribute Protocol** (ATT), and **Generic Attribute Protocol** (GATT).

Note 3.1: Availability of GATT Server APIs Documentation

During my work, I found out that usually, regardless of the chosen language/framework, the documentation on how to implement a GATT server is much less detailed than that on how to connect to one.
This, I believe, is due to the fact that the most common use case of BLE is gathering data from external sensors (e.g. heart rate monitors) that have GATT servers already implemented by the manufacturer.

3.2 Attribute Protocol (ATT)

The **Attribute Protocol** (ATT) defines the basic structure of how data are stored and exchanged in BLE.

At a very basic level of understanding, you can see a Bluetooth LE device has a database table with entries with special format and fields.

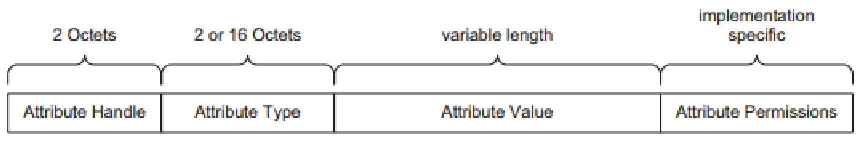


Figure 2: ATT entry specification.

The fields of an ATT entry are:

- **Attribute Handle** - 2 Octets - represents like a unique ID for each entry (an unsigned number unique for the attribute)
- **Attribute Type** - 2 or 16 Octets - is a unique number called **UUID** (Universally Unique IDentifier)
- **Attribute Value** - variable length - is the value of the attribute
- **Attribute Permissions** - implementation specific length - defines whether read or write access is permitted for the attribute.

The device holding this database is the *server* device, while the one accessing it is the *client*.

The server contains a number of attributes, and here the **GATT Profile** comes to define a more generic structure to control how to use the Attribute Protocol to discover and interact with the attributes.

From the point of view of ATT, the value is amorphous; it is just an array of bytes. The actual meaning of the value depends entirely on its UUID, and ATT does not check if the value length is consistent with the given UUID.

3.3 Generic Attribute Protocol (GATT)

The **GATT Protocol** dictates how ATT is employed in service composition. The GATT Protocol has a logical hierarchy that consists of:

- one or more **services**
- each service must have one or more **characteristics**
- each characteristic has optionally a **descriptor**.

Remember that all have the general structure of attributes (Handler, UUID and value).

4 Android Bluetooth LE GATT Server API's Overview

To implement Bluetooth LE in Android, you need to use two main components:

- the `BluetoothGattServer`
- the `BluetootLeAdvertiser`.

The former, as the name suggests, is used to set up a GATT server, the latter enables BLE advertising on the smartphone.

Note 4.1: Support for Advertising in Android

Note that not all Android smartphones support LE advertising. You must ensure that your smartphone is supporting it in order to run an application using a BLE GATT server on it.

In order to create a functioning GATT server on Android, you must get a reference to the **Bluetooth adapter** (through the **Bluetooth manager** provided by Android), and use it to start the advertising, only then the server would be visible by other devices.

The main classes that need to be used in order to create a functioning GATT server are:

- `BluetoothGattServer` class - provides the methods to open a GATT server and set its services (with their characteristics and descriptors)
- `BluetoothLeAdvertiser` class - a reference to an object of this class can be got from the Bluetooth adapter provided by the Bluetooth manager

- `BluetoothGattServerCallback` abstract class - the developer must subclass this class to provide a concrete implementation of this callback, that is used to manage the connections to the GATT server, as well as the incoming requests and responses to send
- `BluetoothGattService` class - describes a GATT service, and provides methods to add characteristics to it
- `BluetoothGattCharacteristic` class - describes a GATT characteristics, and provides methods to manage its descriptor, its permissions, and so on
- `BluetoothGattDescriptor` class - describes a GATT characteristic descriptor, its permissions, and so on
- `AdvertiseData` class - together with its builder class provides a method to describe what data need to be advertised by the LE advertiser
- `BluetoothDevice` class - describes a Bluetooth device, and provides method to access its properties (MAC address, name, and other properties)
- `ByteArray` class - the values of the BLE characteristics have this type. So, the developer must be aware of the actual type of the specific data in order to perform correctly casting to the actual type needed by the application.

5 Android Application

5.1 Overview

The Android application was designed to be easily extensible for future developments. It uses the Android's API provided to implement a functioning GATT server, and tries to build upon them reusable components.

There are four main packages in the application's package structure:

- `ble.gattserver` - this package provides reusable components to manage a GATT server, as decoupled as possible from the specific server implemented by the application
- `blescriptrunner` - this package uses the components defined in the above package to implement the specific server needed by the application
- `fragments` - defines the fragment used in the `MainActivity` to show the connected devices
- `permissions` - this package provides useful components to handle the permission grant flow.

The application's package structure is shown in Figure 3.

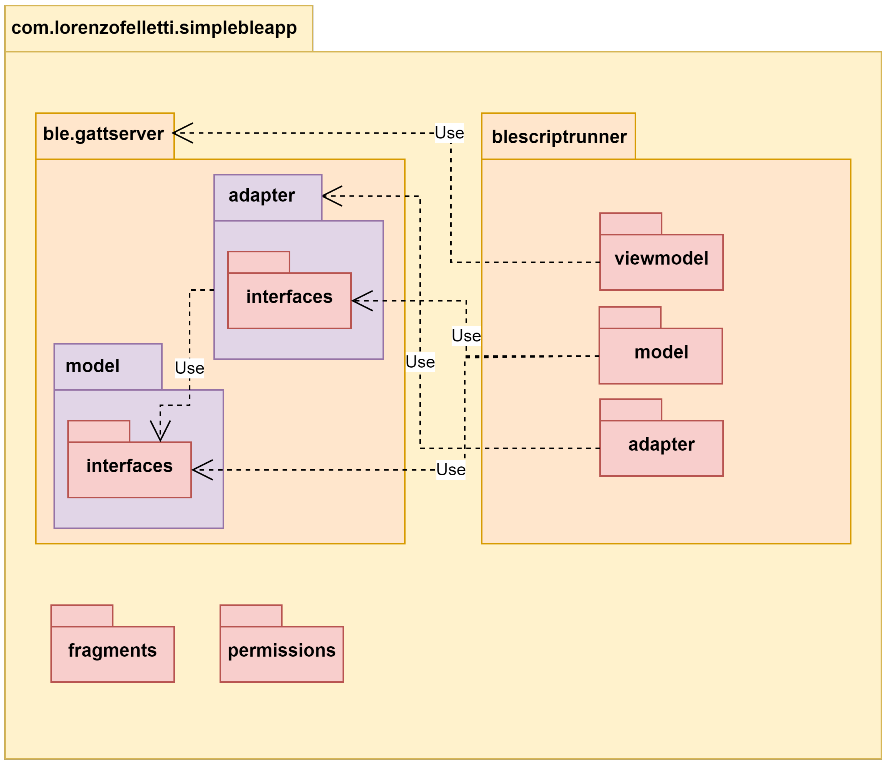


Figure 3: Package structure of the application.

5.2 Application's GATT Structure

The Application's GATT is structured as a service exposing two characteristics. One characteristic holds the name of the *script* to invoke, the other is used by the client to provide the server with the execution status. The characteristic used to invoke the script has a descriptor which is used by clients to (un)subscribe to the notifications for the characteristic (i.e. for enabling and/or disabling the updates on the script to run):

- a first **write** to the descriptor is used to subscribe to notifications
- a second **write** is used to unsubscribe
- a **read** to the descriptor returns the notification status.

The characteristic to provide the execution status is used as follows by the client:

- *when* the notified script is started, a **read** to the characteristic value is made by the client to inform the server that the execution has started

- *when* the execution stops, a **write** request to the characteristic value is made (by the client) with the exit code of the script as data.

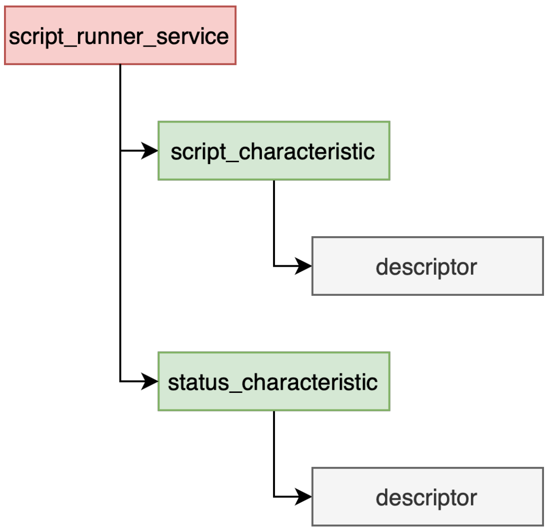


Figure 4: Application GATT server structure.

The characteristic has read and notify permissions enabled, while the descriptor has the permissions to perform read and write requests.

Figure 4 shows the application's GATT server structure, particularly the hierarchy of the exposed service.

Additionally, in Figure 5 it is shown an example of the messages exchanged by client and server during a session in which a user asks for the execution of a script. For the sake of brevity, the Bluetooth connection is supposed to be already established.

5.3 Main Activity

The `MainActivity` extends `AppCompatActivity` and handles the start and stop of both the GATT server and the BLE advertising, as well as the user interactions with the UI components. Also, it creates the **fragment** containing the `RecyclerView` that shows the list of connected devices and their notification status.

The main members of the class are:

- `gattServerManager` - a reference to an instance of `GattServerManager` that is initialized in the `onCreate` method of the class. It is used to manage the GATT server easily.

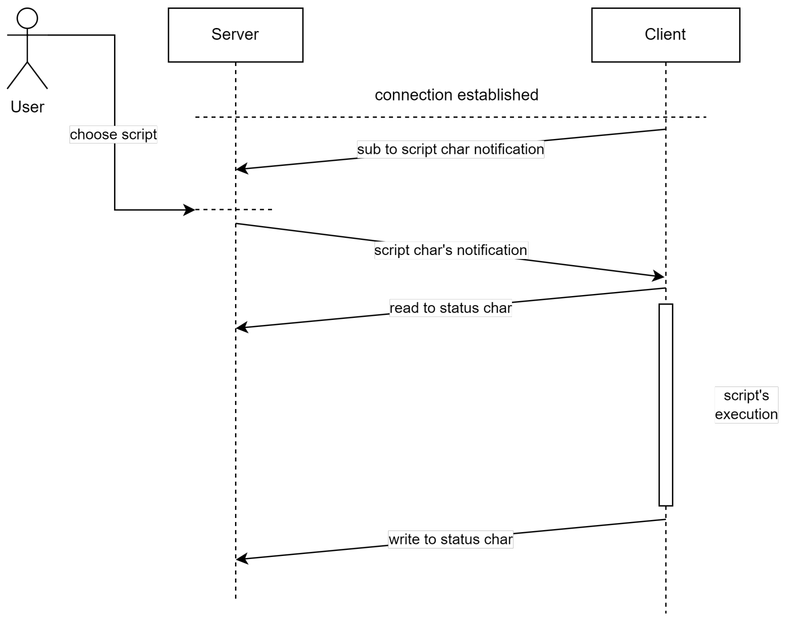
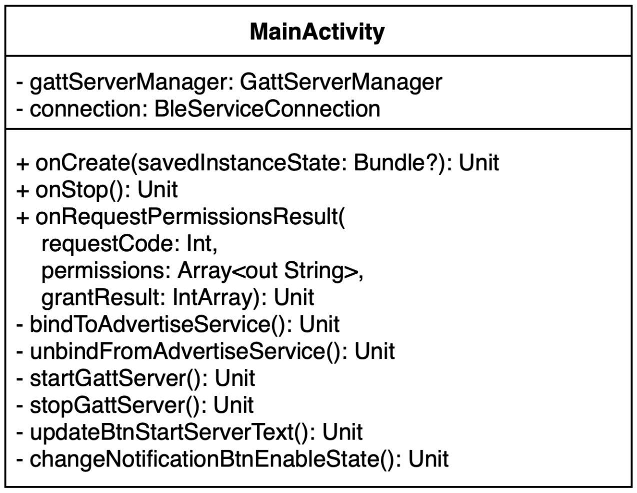


Figure 5: Sequence of messages exchanged after a connection between a client and a server is established, and after the user triggers the execution of a script.



```
MainActivity
- gattServerManager: GattServerManager
- connection: BleServiceConnection
+ onCreate(savedInstanceState: Bundle?): Unit
+ onStop(): Unit
+ onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>,
    grantResult: IntArray): Unit
- bindToAdvertiseService(): Unit
- unbindFromAdvertiseService(): Unit
- startGattServer(): Unit
- stopGattServer(): Unit
- updateBtnStartServerText(): Unit
- changeNotificationBtnEnableState(): Unit
```

Figure 6: `MainActivity` class UML diagram. The diagram shows the main members and methods of the class.

- `connection` - of type `BleServiceConnection`, holds the actions to be performed when the service is bound or unbound to the activity.

The main methods of the class are:

- `onRequestPermissionsResult` - overrides the `AppCompatActivity`'s method, and is used in the management of the permissions' request flow to dispatch what to do in the possible permissions requests outcome (granted or not granted)
- `bintToAdvertiseService` - binds the activity to the Service managing the advertising
- `unBindFromAdvertiseService` - unbinds the activity to the Service managing the advertising
- `startGattServer` - starts the GATT server
- `stopGattServer` - stops the GATT server
- `changeNotificationBtnEnableState` - enables/disables the notification updates sending based on whether the advertising and the server are on or off.

The class also manages the permissions' granting flow.
The UI has two `Button`s and a `EditText`:

- a `Button` to start and stop the server, whose listener to the **click** event also checks that the required permissions are granted before starting the server, and in case they are not granted starts the flow for requesting them (that will later conclude with the call to `onRequestPermissionsResult`)
- the `EditText` is used to change the value of the characteristic
- a second `Button` is used to send the notification to the devices subscribed to it.

Note 5.1: Notification Triggering

The notification is not triggered immediately when the `EditText`'s text is changed. By doing so, one will send a lot of *spurious* notifications while the user is typing the script they want to invoke, and that could also cause the wrong script or the wrong script options to be invoked.
To avoid this problem, notification is triggered only when the user explicitly states their intention to send a notification, thus the need for an ad hoc `Button` for this.

5.4 Connected Devices Fragment

The `ConnectedDevicesFragment` class extends `Fragment`, and handles a `RecyclerView` showing currently connected devices, and their notification status.

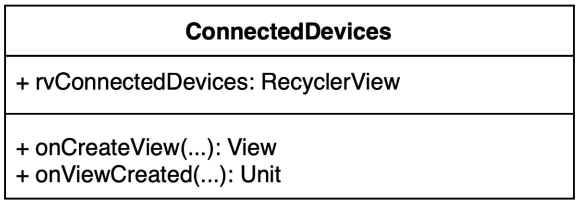

ConnectedDevices
+ rvConnectedDevices: RecyclerView
+ onCreateView(...): View
+ onViewCreated(...): Unit

Figure 7: `ConnectedDevices` class UML diagram showing the main members and methods of the class.

The class uses a `ViewModel` to get the `GattServerManager` reference from the `MainActivity`.

5.5 Bluetooth LE Package

The package named `ble.gattserver` was created with the purpose of creating a general API to simplify the creation and management of a GATT server.

It was made as independent as possible from the actual purpose of the GATT server, so that it could be reused in different scenarios just by supplying a specific implementation of some interfaces or abstract classes.
The package has The following structure:

- contains a class named `GattServerManager`
- contains a class named `PeripheralAdvertiseService`
- has a sub-package named `model` containing all the model classes and interfaces that are necessary for implementing a GATT server
- has a sub-package named `adapter` containing useful interfaces and classes to simplify the creation of adapters.

5.5.1 GattServerManager

The `GattServerManager` class provides a decoupled from the actual usage implementation of a GATT server. The class implements two interfaces, defined in the sub-package `model.interfaces`:

- `HasConnectedDevicesAdapter`
- `HasConnectedDevicesMap`.

These interfaces manage the connected devices, the interaction with them, and their displaying.

The primary constructor of the class has the following arguments:

- the context to use

- a `MutableMap` of connected devices (which can be null, in which case it is created
- a reference to the adapter used to display the connected devices, implementing the `ConnectedDeviceAdapterInterface` interface, and that can be null (if the connected devices should not be displayed)
- the class to be used to create the `GattServerCallback`, extending `AbstractBleGattServerCallback`.

The members that can be accessed of the class are:

- the adapter, extending `ConnectedDeviceAdapterInterface`
- the `bluetoothConnectedDevices`, that is the `MutableMap` of devices that are connected, mapping each connected device to a `Boolean` that is `true` if the device has enabled the notifications for a characteristic, false otherwise.

The methods exposed by the class are:

- `startGattServer()`
- `stopGattServer()`
- `addService(service: BluetoothGattService)`
- `addCharacteristicToService(serviceUUID: UUID, characteristic: BluetoothGattCharacteristic)`
- `setCharacteristic(characteristicUUID: UUID, value: T)`.

5.5.2 PeripheralAdvertiseService

The `PeripheralAdvertiseService` class extends `Service`, and implements the Android service dedicated to the BLE advertising. Since the service is **bound**, an inner class called `PeripheralAdvertiseBinder` (inheriting from `Binder`) is defined.

The class has a public member called `onStopServiceActions`, that is a list of functions taking no arguments and returning `Unit`. All the functions in this list, as the name suggest, are called when the service is stopped.

The class *overrides* the following `Service` methods:

- `onCreate()`
- `onDestroy()`
- `onBind()`.

Moreover, the following private methods are defined:

- `initialize()`, that get a reference to the Bluetooth LE Advertiser (of type `BluetoothLeAdvertiser`)
- `startAdvertising()`
- `stopAdvertising()`.

The `onCreate` method calls `initialize` and `startAdvertising` to start the LE advertising.
The `onDestroy` method stops the LE advertising and calls the `stopAdvertising` function, which executes the `onStopServiceActions` and then clears said list.

5.5.3 Model

The `model` package is composed of:

- the sub-package `interfaces`
- the `AbstractBleGattServerCallback` abstract class
- the `BleAdvertiseCallback` class
- the `BleServiceConnection` class.

5.5.4 Model: Interfaces

The interfaces defined in the package are two:

- `HasConnectedDevicesAdapter`, that must be implemented by all classes that expose an adapter implementing the `ConnectedDeviceAdapterInterface` interface.
- `HasConnectedDevicesMap` that must be implemented by all classes that expose a map of connected devices of type `MutableMap<BluetoothDevice, Boolean>`.

Both interfaces only defines the presence of a public variable of the relative type, without constraining the interaction. However, these interfaces are useful to seamlessly change the actual class implementing said interfaces, but without posing constraints on the inheritance hierarchy of the implementations.

5.5.5 Model: AbstractBleGattServerCallback

The `AbstractBleGattServerCallback` class is an *abstract class* that extends `BluetoothGattServerCallback`, and implements `HasConnectedDevicesMap`, and `HasConnectedDevicesAdapter`.

This class serves as a baseline to start implementing an application specific GATT server to be used by the `GattServerManager`.

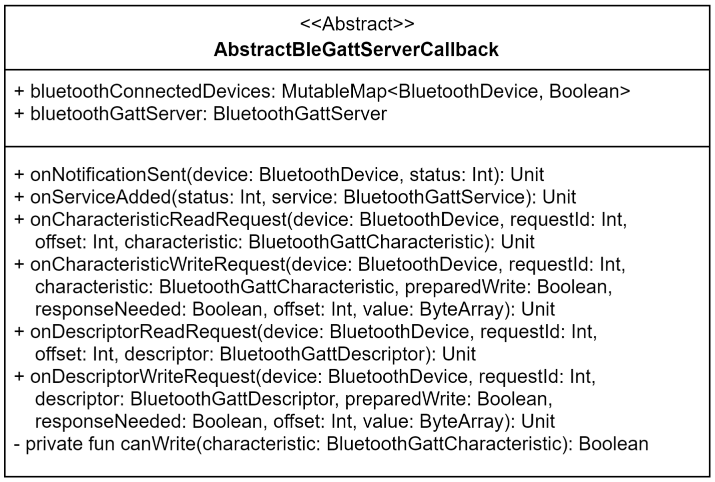

<<Abstract>>
AbstractBleGattServerCallback
+ bluetoothConnectedDevices: MutableMap<BluetoothDevice, Boolean>
+ bluetoothGattServer: BluetoothGattServer
+ onNotificationSent(device: BluetoothDevice, status: Int): Unit
+ onServiceAdded(status: Int, service: BluetoothGattService): Unit
+ onCharacteristicReadRequest(device: BluetoothDevice, requestId: Int, offset: Int, characteristic: BluetoothGattCharacteristic): Unit
+ onCharacteristicWriteRequest(device: BluetoothDevice, requestId: Int, characteristic: BluetoothGattCharacteristic, preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray): Unit
+ onDescriptorReadRequest(device: BluetoothDevice, requestId: Int, offset: Int, descriptor: BluetoothGattDescriptor): Unit
+ onDescriptorWriteRequest(device: BluetoothDevice, requestId: Int, descriptor: BluetoothGattDescriptor, preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray): Unit
- private fun canWrite(characteristic: BluetoothGattCharacteristic): Boolean

Figure 8: `AbstractBleGattServerCallback` class UML diagram.

5.5.6 Model: BleAdvertiseCallback

The `BleAdvertiseCallback` class extends `AdvertiseCallback`.

The class overrides two methods of its base class:

- `onStartSuccess(settingsInEffect: AdvertiseSettings)` that defines what to do in the case the BLE advertising is started correctly
- `onStartFailure(errorCode: Int)` that defines what to do in the case the advertising cannot be started.

To customize the behavior to have in the two cases, two member variables are defined that must be passed to the constructor:

- `doOnStartSuccess: () -> Unit = {}`
- `doOnStartFailure: () -> Unit = {}`.

5.5.7 Model: BleServiceConnection

The `BleServiceConnection` class extends the `ServiceConnection` class.

The constructor of this class is *private*, because the **builder** pattern is used to create instances of the class.

The class `BleServiceConnection.Builder` is used to create an instance of `BleServiceConnection` specialized for the user's needs. In particular, it has methods to add functions (of type `() -> Unit`) that can be executed before or after the service is connected. The use of the builder pattern helps keeping the code simple and readable.

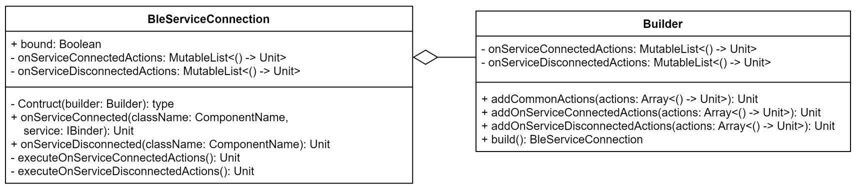

BleServiceConnection
+ bound: Boolean
- onServiceConnectedActions: MutableList<() -> Unit>
- onServiceDisconnectedActions: MutableList<() -> Unit>
- Contruct(builder: Builder): type
+ onServiceConnected(className: ComponentName, service: IBinder): Unit
+ onServiceDisconnected(className: ComponentName): Unit
- executeOnServiceConnectedActions(): Unit
- executeOnServiceDisconnectedActions(): Unit
Builder
- onServiceConnectedActions: MutableList<() -> Unit>
- onServiceDisconnectedActions: MutableList<() -> Unit>
+ addCommonActions(actions: Array<() -> Unit>): Unit
+ addOnServiceConnectedActions(actions: Array<() -> Unit>): Unit
+ addOnServiceDisconnectedActions(actions: Array<() -> Unit>): Unit
+ build(): BleServiceConnection

Figure 9: `BleServiceConnection` and its builder classes UML diagram.

5.5.8 Adapter

The `adapter` package contains the code to help implement an adapter for the connected devices map. In the package is defined the class `AbstractRecyclerViewConnectedDeviceAdapter` that serves as the base class for the adapters that are used in a `RecyclerView`.

The package also contains a sub-package named `interfaces` that defines the `ConnectedDeviceAdapterInterface`.

5.5.9 Adapter: Interfaces

The package named `interfaces` that defines the `ConnectedDeviceAdapterInterface`.

This interface must be implemented by any adapter the user wants to use in the `GattServerManager`.

The interface defines the following methods:

- `clearDevices()`
- `addDevice(device: BluetoothDevice)`
- `removeDevice(device: BluetoothDevice)`
- `blinkDevice(device: BluetoothDevice, blinkStatus: Int, duration: Long)`
- `toggleDeviceNotification(device: BluetoothDevice)`.

The latter method, when called, changes the notification status for a particular device: if it was `true` it changes it to `false`, and vice-versa.

5.5.10 Adapter: AbstractRecyclerViewConnectedDeviceAdapter

The class `AbstractRecyclerViewConnectedDeviceAdapter` extends `RecyclerView.Adapter<T>`, and implements the `ConnectedDeviceAdapterInterface` and `HasConnectedDevicesMap` interfaces.

5.6 Script Runner Package

This package contains the code that is needed for handling the Bluetooth LE GATT server of the specific scenario and application. It uses the classes and interfaces defined in the `ble.gattserver` package, and provide the concrete implementations to be used.

It also defines a Kotlin object called `Constants` holding the UUIDs of the application's service, characteristics, and descriptor:

- service UUID: `0000ffe0-0000-1000-8000-00805f9b34fb`
- script characteristic UUID: `0000ffe1-0000-1000-8000-00805f9b34fb`
- script characteristic descriptor UUID: `00002902-0000-1000-8000-00805f9b34fb`
- execution status characteristic UUID: `0000ffe2-0000-1000-8000-00805f9b34fb`
- execution status characteristic descriptor UUID: `00002903-0000-1000-8000-00805f9b34fb`.

It also defines an `enum class`, and special constants, to handle the execution status results obtained by the client.

This package has three sub-packages:

- `model`
- `viewmodel`, containing the `VieModel` holding the `GattServerManager` reference shared by the `MainActivity` and the `ConnectedDevices` fragment.
- `adpater`, containing the class implementing the actual adapter of the application.

5.6.1 The Model Package

The `blescriptrunner.model` package contains a class named `BleGattServerCallback` that implements the callback to be used by the GATT server of the application.

The class inherits from `AbstractBleGattServerCallback`, and override its `onConnectionStateChange`, `onDescriptorReadRequest`, and `onDescriptorWriteRequest` methods to implement application-specific behavior.

5.6.2 The Adapter Package

The `blescriptrunner.adapter` package contains a class named `ConnectedDeviceAdapter` extending `AbstractRecyclerViewConnectedDeviceAdapter` that is used as adapter in the `RecyclerView` showing the list of connected devices.

The adapter implements directly the methods to be used to update the value of the data, so to avoid code repetition among different classes performing the same kind of update.

It is worth noting that in order to work, the adapter must have a reference to the activity in which it is used, and that the code calling the notification for the change of the data must be in a lambda passed to `activity.runOnUiThread`, otherwise unexpected behavior (such as a delay in the update of the UI) may occur.

6 Python Application

The Python application is much simpler than the Android one. It was developed using the Bleak library as the Bluetooth LE framework.

Bleak is an open source, multiplatform, library written in Python to manage Bluetooth LE with a very active development community.

The application continuously tries to discover and connect to devices exposing the script runner's service UUID (defined in Subsection 5.6). Once one of such devices is found, and the application connects to it, it enables the notification for the script characteristic, and waits to receive notifications.

Once a notification is received, the notified characteristic value is put into an *asynchronous queue*, that is consumed (also asynchronously) by the coroutine that has the task to try the script's invocation. This task parses the characteristic value (in a POSIX compliant way), and then tries to execute the script obtained by prepending `arg[0]` with the path to the script directory. If a script with that name does not exist in the script directory, the error is saved in the log file, but the program continues to run without being blocked.

Note 6.1: The Bleak Library

The Bleak library works on different platforms, given that the right Bluetooth backend is installed. For instance, on macOS, the installation of `pyobjc` is mandatory in order for the library to work properly.
Bleak makes massive use of Python's which enables for the use of *coroutines* in Python. Thus, the applications developed using this library permits to write highly concurrent, asynchronous, code, without the bigger overhead of using threads.

Note 6.2: Log Analysis

Through the analysis of the application's logs, it is possible to have a better understanding of the application's functioning, and monitor the system.
Logs filtering is one of the best ways to check if some errors occurred during the time the application was running. Filtering the log file to show only the errors that occurred can be achieved by running in the terminal:

```
cat logs/main.log | egrep 'ERROR'
```

An example of output is:

```
ERROR:__main__:[Errno 2] No such file or directory:
↪  '/Users/user/app/scripts/hellw.sh'
ERROR:__main__:[Errno 2] No such file or directory:
↪  '/Users/user/app/scripts/hellno.sh'
```

In the example, there was an attempt to invoke two scripts, named `hellw.sh` and `hellno.sh`, that were not present in the `scripts` folder, thus the shell trying to invoke them resulted in an error, that was subsequently logged by the Python application. Anyway, the two errors were not blocking, allowing for the execution to continue.

6.1 Application Structure

The application is composed of a file named `main.py`, that defines all the functions used in the application, as well as containing the code to run the application, a `strings.py` file, that contains all the strings that are used in the usage helpers, and a file named `config.py` that defines the object containing the application configuration (uuids, contant values, and so on).

The application can be run from the command line with different configurations, configurable via the use of a set of command line options.

To run the application, it is mandatory to have the requirements (described in the file `requirements.txt` installed (preferably in a dedicated virtual environment). Once the requirements are satisfied, it is sufficient to run `python main.py`, optionally with optional arguments specified next.

Note 6.3: Application's Usage

By running `python main.py --help` the following usage helper is displayed:

```
usage: main.py [-h] [--log-level LOG_LEVEL] [--log-file LOG_FILE]
↪  [--notification-window-size NOTIFICATION_WINDOW_SIZE]
↪  [--max-running-time MAX_RUNNING_TIME] [--follow-log]

Bluetooth Low Energy client searching for devices with a given service
↪  uuid and executing scripts indicated by a specific characteristic
↪  value on notification. If the script is not found, the client will
↪  ignore it.

optional arguments:
  -h, --help            show this help message and exit
  --log-level LOG_LEVEL
                        Provide logging level. Example: --log-level
                        ↪  DEBUG, default: INFO
  --log-file LOG_FILE   Provide logging file. Example: --log-file
  ↪  /tmp/main.log
  --notification-window-size NOTIFICATION_WINDOW_SIZE
                        Provide notification window size (in seconds).
                        ↪  Example: --notification-window-size 20.
                        ↪  Default: 10
  --max-running-time MAX_RUNNING_TIME
                        Provide max running time (in seconds), i.e. the
                        ↪  time after which the client will exit.
                        ↪  Example: --max-running-time 60. Default: 8h
                        ↪  (28800s)
  --follow-log, -f      Follow the log file. Example: --follow-log-file
```

The main lines of code in `main.py` are:

```python
async def run_ble_client(device: BLEDevice, queue:
↪  asyncio.Queue):
    ''' Connects to the device and reads the characteristic, then
    ↪  puts the data into the queue '''
    disconnection_event = asyncio.Event()  # set by the
    ↪  disconnection callback

    def disconnection_callback(_):
        disconnection_event.set()

    async def disconnection_handler(client:
    ↪  Optional[BleakClient]):
        ''' Disconnects from the device and waits for the
        ↪  disconnection event. Pass client=None when the client
        ↪  is already disconnected (e.g. after timeout). '''
        # ...

    try:
        async with BleakClient(device,
        ↪   disconnected_callback=disconnection_callback) as
        ↪   client:
        async def notification_callback(_, data: bytearray):
            await queue.put((time.time(), data, client))

        await client.connect()
        await client.start_notify(C.CHAR_UUID,
        ↪   notification_callback)
        try:
            await
            ↪   asyncio.wait_for(disconnection_event.wait(),
            ↪   C.NOTIFICATION_WINDOW_SIZE)
        except asyncio.TimeoutError:
            # ...
        except BleakError as e:
            disconnection_event.set()
        finally:
            await disconnection_handler(None if
            ↪   disconnection_event.is_set() else client)
    except Exception as e:
        await disconnection_handler(None)

async def run_queue_consumer(queue: asyncio.Queue):
    ''' Consumes the queue and invokes the script indicated by
        ↪   the queue data. The queue data is a tuple of (timestamp,
        ↪   data, client) '''
    def run_script(data: bytearray) -> int:
        ''' Runs the script indicated by the data. Returns the
        ↪   exit code. '''
        # Manipulate data so that is runnable
        try:
        process = subprocess.run(data)
        return process.returncode
        except Exception as e:
        return 255

    while True:
        epoch, data, client = await queue.get()
        if data is None:
        break
        if client is not None and client.is_connected():
        await client.read_gatt_char(C.CHAR_MONITORING_UUID)
        res = run_script(data)
        if client is not None and client.is_connected():
            await client.write_gatt_char(C.CHAR_MONITORING_UUID,
            ↪  bytearray(str(res % 256).encode("utf-8")), True)

async def app():
    ''' Scan for devices with a given service uuid, then connect
    ↪  to them and read the characteristic on notification, then
    ↪  invoke the script indicated by the characteristic. '''
    device_to_connect_to: Optional[BLEDevice] = None
    stop_event = asyncio.Event()

    def scan_callback(device: BLEDevice, advertising_data:
    ↪  AdvertisementData):
        device_to_connect_to = device
        stop_event.set()  # awakens stop_event.wait()

    async with BleakScanner(scan_callback,
    ↪  service_uuids=[C.SERVICE_UUID]) as _:
        await stop_event.wait()

    if isinstance(device_to_connect_to, BLEDevice):
        queue = asyncio.Queue()
        client_task = run_ble_client(device_to_connect_to, queue)
        consumer_task = run_queue_consumer(queue)
        await asyncio.gather(client_task, consumer_task)

if __name__ == "__main__":
    # Configuration setup
    # ...
    # Run the app
    try:
        while not has_max_running_time_elapsed():
            asyncio.run(app())
    except Exception as e:
        # log error
    finally:
        # close resources and print statistics
```

From the code, it is noticeable that the connection is always terminated after a configurable amount of time. This is especially useful for testing purposes, enabling to gather data about latencies quickly by setting a small connection window, so that many different connections are established and terminated within a small time window.

7 Performance Assessment

The tested conducted on the framework were of two kind:

- CPU and memory usage for the Android application
- connection latency for the client application.

7.1 Testing environment

The tests were carried out, Android-side, on a **Google Pixel 3a** onboarded with **Android 12L** (API level 32), and using the release version of the application. The tool used to assess the performances is **Android Studio**'s *Profiler*.

On the other end, the device used to test the Python client was an **Apple MacBook Air M1 2020**, mounting **macOS 12.6**.

7.2 Android Application Performance

The hardware requirements to use the Android application are relatively low, as proved by the tests carried out to assess the application's performance.

The results show that the memory (RAM) required by the Android application to run is about 80 MB, and the CPU usage peaked at about 10%.

The results of a run of the test are shown in Figure 10.

As it is visible in Figure 10 the application is not consuming a lot of resources, and especially CPU usage remains really low during the whole execution of the test.

The obtained results are consistent throughout different runs of the application, further confirming the low performance requirements required to run the Android application, that is this way suitable to run on low-end devices. This is a huge pro in the case of the usage scenario by a factory, since the factory would not be forced to provide their employee with costly, high-end, devices to make use of the framework.

7.3 Bluetooth Performance

The Bluetooth performance was assessed on the Python application side, by using the `time` library

Two metrics were measured:

- **discovery latency** - time elapsed from scanning start to device discovery
- **connection latency** - time elapsed between the discovery and successful connection to a device.

Latency times can vary based on many parameters, such distance between the devices (the greater the distance, the greater the latency), or the number of other devices around (the more the devices, the greater the latency), and many others difficult to measure parameters.

Another set of parameters that can tweak the mean discovery latency is tuning the GATT server advertising data parameters, above all:

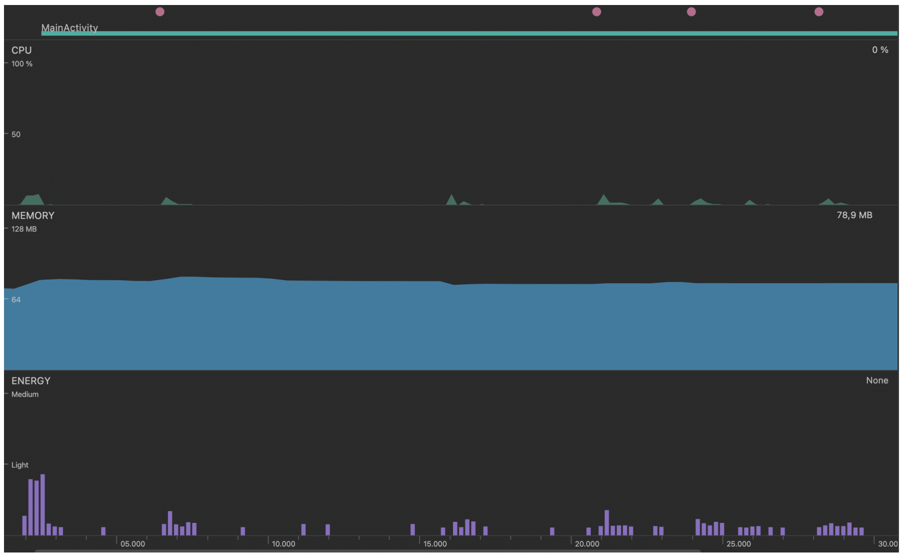


Figure 10: CPU, memory, and energy performance measured while the application is running. At the top, there are 4 noticeable red dots, corresponding to four user actions on the UI. The first and last dot correspond to the start and shut down of the server, while the dots in the middle correspond to sent notifications.
In the CPU usage line, it is possible to notice several peaks of CPU usage, the first, and highest, corresponding to the start of the application. The other peaks all corresponds more or less to the user interaction with the UI triggering some actions, apart from the third peak from the left, which corresponds to the moment when the Python client connected to the GATT server. The fourth peak from the left corresponds to the first notification sent by the user to the client. Right after it

- transmission power - the higher, the smaller the latency
- advertising interval - the smaller, the smaller the latency.

7.3.1 Discovery Latency

The discovery latency was measured as the time elapsed between the moment the LE scanning is started to the time the LE device is found.

Measurements of discovery latency over 12 trials, with low latency and high transmission power settings:

- mean latency: 2.74s
- variance: 1.89s^2
- max latency: 4.88s

- min latency: 0.93s.

7.3.2 Connection Latency

The measured connection latency, i.e. the time elapsed between discovery and connection, varied a lot across various tests, ranging from 1s to 3s, but in any case it was never under the second.

Measurements of connection latency over 12 trials, with low latency and high transmission power settings:

- mean latency: 1.96s
- variance: 2.24s^2
- max latency: 5.05s
- min latency: 0.70s.

7.3.3 Considerations

The measured latencies, both discovery and connection, showed a high level of variance and were generally a little high. That is, I suspect, at least partly to be attributed to the Bleak library. This suspect is supported by the fact that my smartwatch also automatically connects to the GATT server, and does it much faster than the Python client.

8 Security Advise

Security is an important aspect of modern applications, and is especially important in a context such as this one, where a device can be used to invoke scripts on a remote platform.

While the application is not implementing much of the security aspects that will be described below, that doesn't mean the aspect was overlooked in the application design.

8.1 Avoid Arbitrary Remote Script Execution

One of the first security problems that were taken into account was that of disabling arbitrary scripts to be executed on the target platform from the device.

In fact, the only scripts that can be invoked must reside in a specific directory. Currently, the implementation involves:

- splitting the characteristic values following the POSIX semantics (i.e. preserving the quotes and double quotes behavior one expects from the shell) into an array that is interpreted as
`<script_name>, <arg1>, ..., <argN>`

- to the `<script_name>` is prepended the path to the directory that contains the scripts by the target machine (i.e. the machine that has to invoke the script)
- all the other elements of the array are interpreted as arguments to the script `/path/to/script/directory/<script_name>`.

This way, a potential attacker has no way to execute a script that is not in the specific script directory that the target machine has defined, not even by using pipelining or logical operators to alter the execution flow, as these operators would be interpreted as arguments for the script.

8.2 Confidentiality, Authentication, and Authorization

At the moment, the mechanisms described in the following lines are not implemented.

8.2.1 Confidentiality

To ensure confidentiality, a Diffie-Hellman key exchange is advised to negotiate a symmetrical key unique to the connection. This will cover against scenarios in which an attacker is trying to attack the system by spreading over messages sniffed in other connections.

In fact, due to the fact that each connection will have its own key, the attacker cannot connect to a machine and send messages they previously sniffed, because the attacker's key would not be the same of the connection they sniffed, so the message could also arrive to the machine, but the machine would not be able to decrypt, and thus use, it.

However, this is not preventing in any way from DoS attacks, that are not completely avoidable, but could be mitigated with an authentication and authorization mechanism.

8.2.2 Authentication and Authorization

An authentication and authorization mechanism could be achieved in different ways:

- you could prepend an `auth_token` (i.e. a secret shared by client and server) to the characteristic value
- you could create a second characteristic containing the `auth_token`.

Either way, the token should be exchanged securely outside the scope of the Bluetooth LE connection, for example using a shared repository to which both the client and the server look to get the `auth_token`s, that should be changed regularly.

In this way, authentication is achieved by the use of the tokens, and authorization can be achieved by using different tokens for different levels of authorization.

8.2.3 DoS Mitigation

DoS mitigation could be done by blacklisting for a given amount of time addresses of devices supplying wrong `auth_token`s, and incrementing exponentially the blacklisting time at each error.

This could mitigate the effect of DoS attacks in the case one or a few devices are used, but not if the attack is coming from many different devices simultaneously.

However, given the limited range of Bluetooth, attacks from many devices simultaneously are difficult enough to achieve that no countermeasures is the right solution for most scenarios. Bluetooth LE range is a sufficient DoS mitigation mechanism most of the time. This is another example of how the relatively small range of Bluetooth can be used as an advantage and not a disadvantage.

9 Conclusion

Overall, the framework built provides a tool that, with appropriate refinements, could be used in real scenarios. Bluetooth Low Energy proved to be a powerful enough wireless technology to perform what could be called **pRPC** (proximity RPC), coming with the advantage of having really low power consumption, and a localized message exchange that would be difficult to intercept for an attacker, as they need to be very close to the source.

On the other side, the possible downsides are the lower *reliability* in the connection performance, as the latencies can vary a lot between different trials, thus requiring applications to be tolerant to relatively long connection times.

9.1 Future Developments

Further developments could go in the direction of speeding up the BLE Central's performance, especially the discovery and connection latencies.

Another area that may need some refinements is the Android application's UI, as it is very basic as of now.

Also, some features that could be developed are, other than the security ones that were described in Section 8, the possibility to gather from the connected devices the list of available scripts to run, so that the user is not forced to know the script's name a priori, nor to remember it.