Android Programming

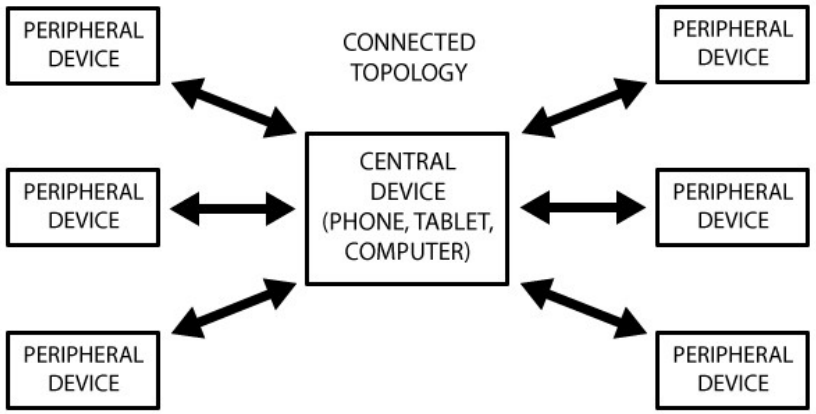
TX00CK66 Sensor Based Mobile Applications

Lecture Bluetooth LE

Jarkko.Vuori@metropolia.fi

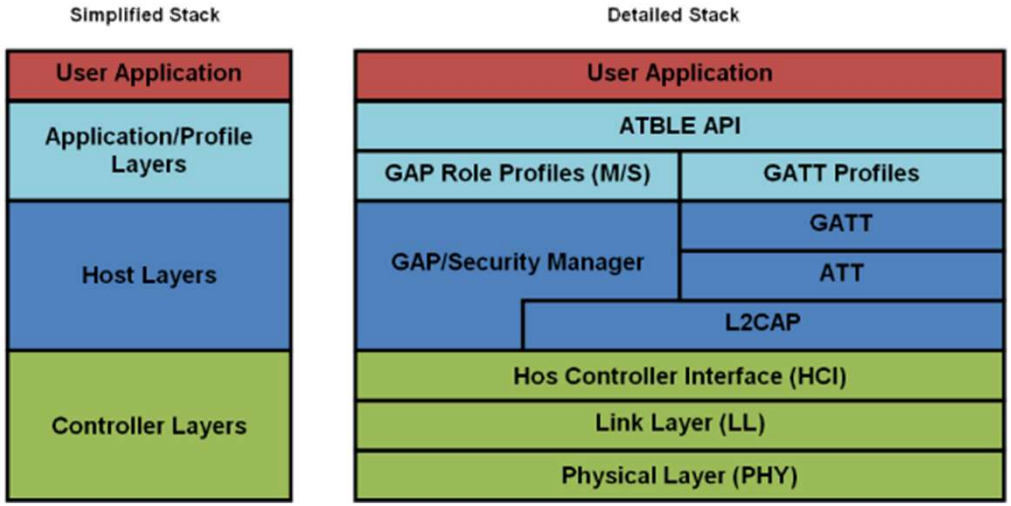
Bluetooth architecture

- A peripheral can only be connected to one central device (such as a mobile phone) at a time
  - the central device can be connected to multiple peripherals
- If data needs to be exchanged between two peripherals, a custom mailbox system will need to be implemented where all messages pass through the central device
- Once a connection is established between a peripherals and central device, communication can take place in both directions
  - different than the one-way broadcasting approach using only advertising data



Tech side

- Layered protocol architecture
  - HCI is often used in external Bluetooth device communication
  - GATT (Generic Attribute) and GATT Profiles makes it easy to provide services and data to clients (applications)
- Robustness
  - Adaptive fast frequency hopping
  - Forward Error Correction (FEC)
  - Fast Acknowledge (ACK) to the sender
- Total time to send data is typically 50 ms
  - With special hardware 6 ms (www.BlueRadios.com)
  - New Bluetooth 5.2 will improve transmission latencies
    - With a new LC3 codec, latency is 5 ms
- Only one general type of packet, specialized into advertisement and data types

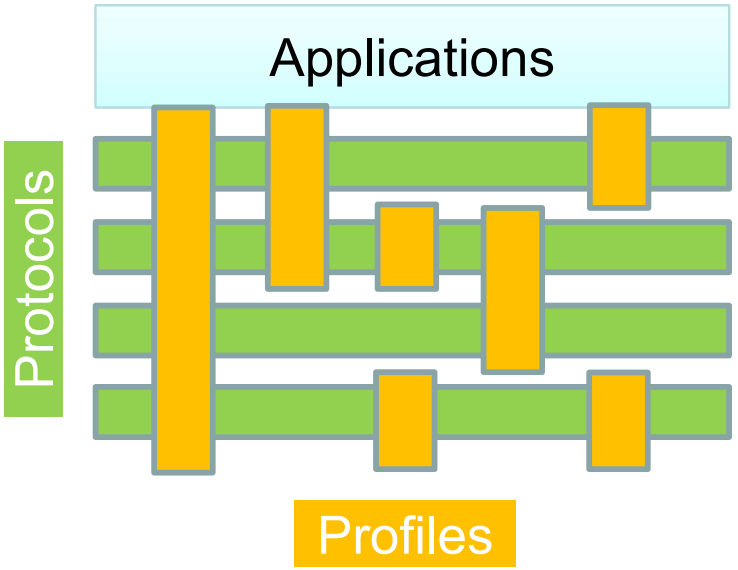


Bluetooth module

Processor module

Tech side

- Bluetooth documents (Bluetooth.com)
  - Core Specification
    - How the tech works
    - Bluetooth attribute protocols (ATT)
      - Specify how data (attribute) is changed
  - Generic Attribute Profiles (GATT)
    - How (and what for) the tech is used
    - How different parts of the spec shall be used to fulfill a desired function
    - Examples
      - Battery level detector
      - Heart rate monitor
- Bluetooth LE is not compatible with the Bluetooth classic
  - But a device can implement both modes
  - And both can co-exist on the same frequency band

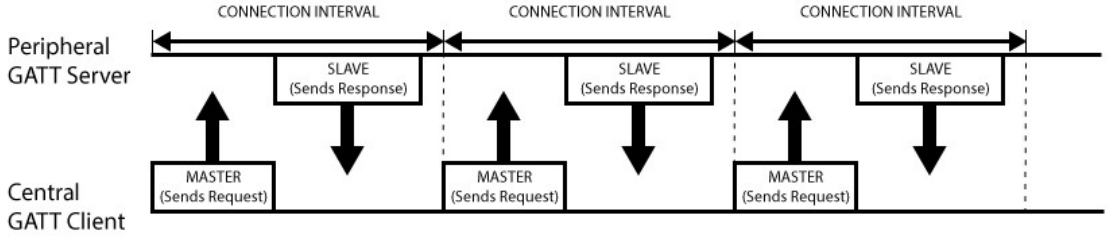


GATT is an acronym for the Generic Attribute Profile, and it defines the way that two Bluetooth Low Energy devices transfer data back and forth using concepts called **Services** and **Characteristics**.

It makes use of a generic data protocol called the **Attribute Protocol (ATT)**, which is used to store Services, Characteristics and related data in a simple lookup table using 16-bit IDs for each entry in the table.

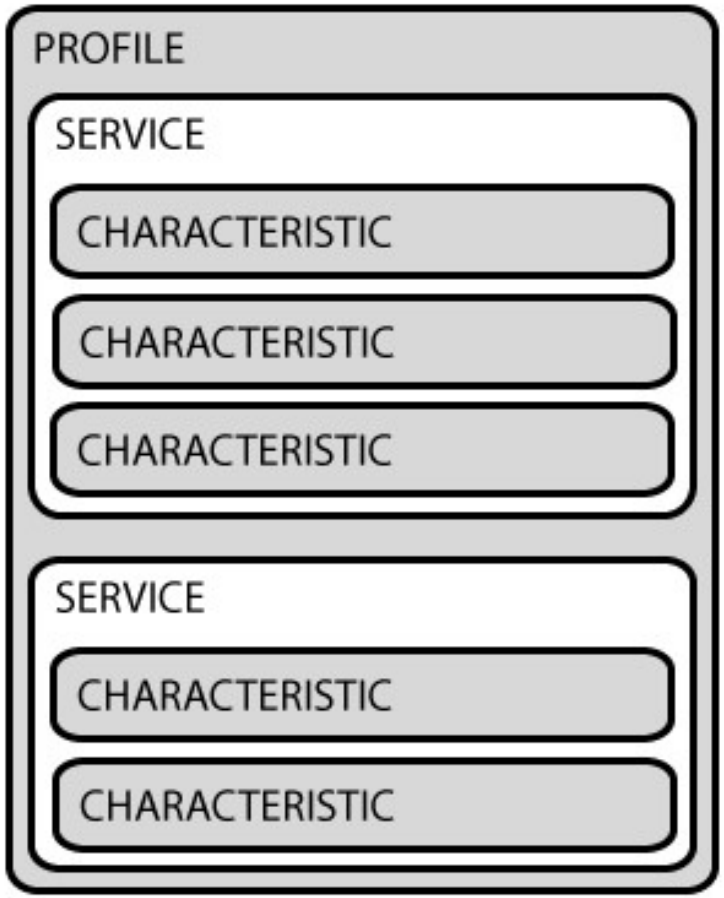
GATT

- GATT is based on the server/client relationship
- The peripheral is known as the GATT Server
  - holds the ATT lookup data and service and characteristic definitions
- The phone or table is the GATT Client
  - It sends requests to this server
- All transactions are started by the master device, the GATT Client, which receives response from the slave device, the GATT Server
- When establishing a connection, the peripheral will suggest a 'Connection Interval' to the central device, and the central device will try to reconnect every connection interval to see if any new data is available, etc.
  - This connection interval is really just a suggestion of the peripheral
    - Central device may not be able to honor the request because it's busy talking to another peripheral or the required system resources just aren't available



GATT transactions

- GATT transactions in BLE are based on high-level, nested objects called Profiles, Services and Characteristics
  - In HTTP communication we have URLs and ports
  - In Bluetooth communication we have services and characteristics
    - Characteristics are defined attribute types that contain a single logical value, i.e., characteristics are where the data is, that's what we want to read or write
    - Services and characteristics are identified by UUIDs



Profiles

- A Profile doesn't actually exist on the BLE peripheral itself, it's simple a pre-defined collection of Services that has been compiled by either the Bluetooth SIG or by the peripheral designers
- The Heart Rate Profile (available from here https://www.bluetooth.com/specifications/specs/), for example, combines the Heart Rate Service and the Device Information Service
- The complete list of officially adopted GATT-based profiles can be seen here: https://www.bluetooth.com/specifications/profiles-overview

Services

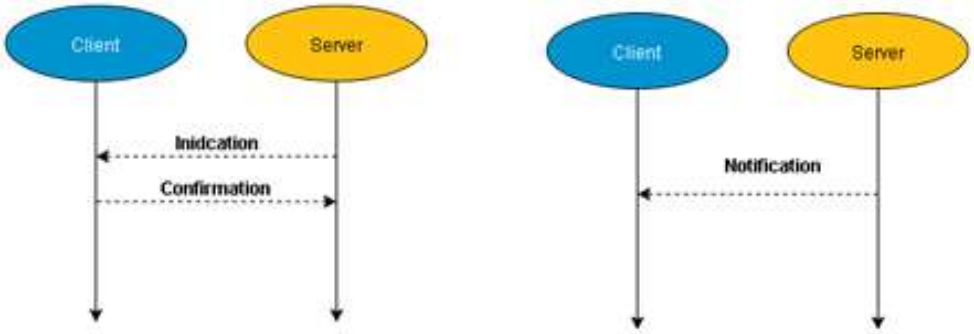
- Services are used to break data up into logic entities, and contain specific chunks of data called characteristics
- A service can have one or more characteristics, and each service distinguishes itself from other services by means of a unique numeric ID called a UUID, which can be either 16-bit (for officially adopted BLE Services) or 128-bit (for custom services)
- A full list of officially adopted BLE services can be seen on the Services page of the Bluetooth Developer Portal (https://www.bluetooth.com/specifications/gatt/services)
  - E.g., Heart Rate Service is officially adopted service that has a 16-bit UUID of 0x180D, and contains up to 3 characteristic, though only the first one is mandatory: Heart Rate Measurement, Body Sensor Location and Heart Rate Control Point

Characteristics

- The lowest level concept in GATT transactions is the Characteristic, which encapsulates a single data point (though it may contain an array of related data, such as X/Y/Z values from a 3-axis accelerometer, etc.)
- Similar to Services, each Characteristic distinguishes itself via a pre-defined 16-bit or 128-bit UUID
  - you're free to use the standard characteristics defined by the Bluetooth SIG (which ensures interoperability across and BLE-enabled HW/SW)
  - or define your own custom characteristics which only your peripheral and SW understands
- As an example, the Heart Rate Measurement characteristic is mandatory for the Heart Rate Service, and uses a UUID of 0x2A37
  - It starts with a single 8-bit value describing the HRM data format (whether the data is UINT8 or UINT16, etc.), and goes on to include the heart rate measurement data that matches this config byte

Indication and Notification

- In addition to the normal read and write, indication and notifications are commands that could be send through the attribute (ATT) protocol
- BLE standard define two ways to transfer data for the server to the client:
  1. Notification and
  2. Indication
  - Maximum data payload size defined by the specification in each message is 20 bytes
  - Notifications and indications are initiated by the Server but enabled by the Client
- Notification don't need acknowledged, so they are faster
  - Server does not know if the message reach to the client
- Indication need acknowledged to communicated
  - The client sent a confirmation message back to the server, this way server knows that message reached the client
  - You have to wait for the confirmation of each indication in order to send the next indication



Making a UUID

- Services available from the server side are indentified by the UUID (Universally Unique Identifier)
  - They are like port numbers in Internet
  - You can generate an one e.g. using the https://www.uuidgenerator.net/
    - Then your client must use the same UUID
  - Basic serial port service (RFCOMM) has UUID 00001101-0000-1000-8000-00805f9b34fb which can be converted to UUID object type using `UUID.fromString("00001101-0000-1000-8000-00805f9b34fb")`
  - Or you can use the Kotlin function below to generate Bluetooth UUID from the 16-bit number

```
val HEART_RATE_SERVICE_UUID             = convertFromInteger(0x180D)
val HEART_RATE_MEASUREMENT_CHAR_UUID    = convertFromInteger(0x2A37)
val CLIENT_CHARACTERISTIC_CONFIG_UUID   = convertFromInteger(0x2902)

/* Generates 128-bit UUID from the Protocol Indentifier (16-bit number)
 * and the BASE_UUID (00000000-0000-1000-8000-00805F9B34FB)
 */
private fun convertFromInteger(i: Int): UUID {
    val MSB = 0x0000000000001000L
    val LSB = -0x7fffff7fa064cb05L
    val value = (i and -0x1).toLong()
    return UUID(MSB or (value shl 32), LSB)
}
```

Making a connection

- From the Bluetooth LE scan you will have the `ScanResult` object
  - `ScanResult` object contains a field `BluetoothDevice`
  - This can be used to make a connection to the device (if possible)

```
val BluetoothDevice device = …
val gattClientCallback = GattClientCallback(this)
mBluetoothGatt = device.connectGatt(this, false,
mGattCallback);
```

  - Because connection can be happen at any time, it is asynchronous operation. Therefore we need again those callbacks
    - In this case it is the `GattClientCallback` class where you implement the operations for the connection

Making a connection

- When the connection to the Bluetooth LE device is established, `onConnectionStateChange()` with the status `STATE_CONNECTED` is called
- Then it is possible to discover what kind of services are available from the connected device by calling `discoverServices()` method
  - It uses the same callback object than `connectGatt()`
  - It calls the method `onServicesDiscovered()` with the List of `BluetoothGattService` objects

```
class GattClientCallback(): BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        super.onConnectionStateChange(gatt, status, newState)
        if (status == BluetoothGatt.GATT_FAILURE) {
            Log.d("DBG", "GATT connection failure")
            …
            return
        } else if (status == BluetoothGatt.GATT_SUCCESS) {
            Log.d("DBG", "GATT connection success")
            …
            return
        }
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            Log.d("DBG", "Connected GATT service")
            …
            gatt.discoverServices();
        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            …
        }
    }

    override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
        super.onServicesDiscovered(gatt, status)
        if (status != BluetoothGatt.GATT_SUCCESS) {
            return
        }

        Log.d("DBG", "onServicesDiscovered()")

        for (gattService in gatt.services) {
                …
        }
    }

    override fun onDescriptorWrite(gatt: BluetoothGatt, descriptor: BluetoothGattDescriptor, status: Int) {
        Log.d("DBG", "onDescriptorWrite")
    }

    override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
        Log.d("DBG", "Characteristic data received")
    }
}
```

Making a connection

- When you have found the service you are insterested on, it is time to ask notifications from there (if the service provides them to you)
  - You need also to enable the server to provide values

```
if (gatt.setCharacteristicNotification(characteristic, true)) {
    // then enable them on the server
    val descriptor = characteristic.getDescriptor(UUID_CLIENT_CHARACTERISTIC_CONFI
    descriptor.value = BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
    val writing = gatt.writeDescriptor(descriptor)
}
```

- You will get `onCharacteristicChanged()` method call in your `GattClientCallback()` class when characteristic's content has been changed
  - Data itself can be obtained using getValue(), getIntValue() etc. functions

```
for (gattService in gatt.services) {
    Log.d("DBG", "Service ${gattService.uuid}")

    if (gattService.uuid == HEART_RATE_SERVICE_UUID) {
        Log.d("DBG", "BINGO!!!")

        for (gattCharacteristic in gattService.characteristics)
            Log.d("DBG", "Characteristic ${gattCharacteristic.uuid}")

        /* setup the system for the notification messages */
        val characteristic = gatt.getService(HEART_RATE_SERVICE_UUID)
                .getCharacteristic(HEART_RATE_MEASUREMENT_CHAR_UUID)
        gatt.setCharacteristicNotification(characteristic, true)
                ….
    }
}
```

```
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
    Log.d("DBG", "Characteristic data received")

    …

}
```

Reading list

- Bluetooth technical specification
  - https://www.bluetooth.com/
- Bluetooth in Android
  - http://www.tutorialspoint.com/android/android_bluetooth.htm
  - https://developer.android.com/guide/topics/connectivity/bluetooth/ble-overview