Lab4 Microservices Exploring Integration Patterns with Bluetooth, CoAP, and MQTT

Lorin Speybrouck

Taks

Task1: Send and receive data with Bluetooth Classic

`Full source code:` *ESP32_serial_BC*

`Result:` see **task1.mp4**

`Code`

```
// ESP32_serial_BC.ino
BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  /* If no name is given, default 'ESP32' is applied */
  /* If you want to give your own name to ESP32 Bluetooth device, then */
  /* specify the name as an argument SerialBT.begin("myESP32Bluetooth"); */
  SerialBT.begin("ESP Lorin");
  Serial.println("Bluetooth Started! Ready to pair...");
}
void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

- A simple Bluetooth classis application, using the serial profile
- A serial connection is created with name 'ESP Lorin'
- When there serial bluetooth message available it is written to the console

Bonus Task2: Control a LED on the ESP32 using Bluetooth Classic

`Full source code:` *ESP32_led_BC*

`Result:` see **task2.mp4**

`Code`

```
BluetoothSerial SerialBT;
const int ledPin = 2;

void setup() {
  pinMode(ledPin, OUTPUT);
  Serial.begin(115200);
  /* If no name is given, default 'ESP32' is applied */
  /* If you want to give your own name to ESP32 Bluetooth device, then */
  /* specify the name as an argument SerialBT.begin("myESP32Bluetooth"); */
  SerialBT.begin("ESP Lorin");
  Serial.println("Bluetooth Started! Ready to pair...");
  digitalWrite(ledPin, HIGH);
}
void loop() {
  if (SerialBT.available()) {
    int code = SerialBT.read();
    Serial.println(code);

    if (code == (int)'0') {
      digitalWrite(ledPin, LOW);
    } else if (code == (int)'1') {
      digitalWrite(ledPin, HIGH);
    }
  }
  delay(20);
}
```

- Similar to the previous task but the read code is checked for 0 or 1 to control the built in LED

Task3: Send and receive data with Bluetooth Low Energy

`Full source code:` *RSP32_server_BLE* and *ESP32_client_BLE*

`Result:` see **task3.mp4**

`Code`

```
// ESP32_server_BLE.ino
MFRC522 rfid(SS_PIN, RST_PIN);
BLECharacteristic *pCharacteristic;

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  SPI.begin();
  rfid.PCD_Init();

  BLEDevice::init("ESP32 BLE Server Lorin");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
                                       CHARACTERISTIC_UUID,
                                       BLECharacteristic::PROPERTY_READ |
                                       BLECharacteristic::PROPERTY_WRITE|
BLECharacteristic::PROPERTY_NOTIFY
                                     );

  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
}
void loop() {
  if (rfid.PICC_IsNewCardPresent()) {
    if (rfid.PICC_ReadCardSerial()) {
      MFRC522::PICC_Type piccType = rfid.PICC_GetType(rfid.uid.sak);
      Serial.print("RFID/NFC Tag Type: ");
      Serial.println(rfid.PICC_GetTypeName(piccType));
      Serial.print("UID:");

      String uidStr = "";
      for (int i = 0; i < rfid.uid.size; i++) {
        Serial.print(rfid.uid.uidByte[i] < 0x10 ? " 0" : " ");
        Serial.print(rfid.uid.uidByte[i], HEX);
        uidStr += String(rfid.uid.uidByte[i], HEX);
      }

      pCharacteristic->setValue(uidStr.c_str());
      pCharacteristic->notify();
      Serial.println("BLE Characteristic updated with UID: " + uidStr);


      Serial.println();
      rfid.PICC_HaltA();
      rfid.PCD_StopCrypto1();
    }
  }
}
```

- On the BLE server a service with SERVICE_UUID where the characteristic with CHARACTERISTIC_UUID will be written to
- The RFID reader is also configured and when a tag is presented it will update the characteristics value using *setValue*. The clients then get this message using notify

```
// ESP32_client_BLE.ino
static void notifyCallback(
  BLERemoteCharacteristic* pBLERemoteCharacteristic,
  uint8_t* pData,
  size_t length,
  bool isNotify) {
    Serial.print("Notify callback for characteristic ");
    Serial.print(pBLERemoteCharacteristic->getUUID().toString().c_str());
    Serial.print(" of data length ");
    Serial.println(length);
    Serial.print("data: ");
    Serial.println((char*)pData);
}
```

- On the BLE client a connection is made with the server(not shown in code block)
- A callback is configured when a server notifies its clients, it then reads the characteristic and prints it to the terminal

Bonus Task4: Send data from ESP32 with BLE to the RPi

`Full source code:` *ESP32_send_BLE* and *RPi_receive_BLE*

`Result:` see **task4.mp4**

`Code`

```
// ESP32_send_BLE
int counter = 0;
BLECharacteristic *pCharacteristic;

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");
  BLEDevice::init("ESP32 Server Lorin For Pi");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE);
  pCharacteristic->setValue("0");
  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
  Serial.println("Characteristic defined!");
}
void loop() {
  counter++;
  pCharacteristic->setValue(counter);
  pCharacteristic->notify();
  Serial.println(counter);
  delay(2000);
}
```

- On the ESP32 a BLE server is configured again with a service with one characteristic. In this case we use a counter
- The characteristic is updated with setValue and then the clients are notified

```python
# main.py
from bluepy.btle import Peripheral, DefaultDelegate
import struct

class MyDelegate(DefaultDelegate):
    def __init__(self):
        DefaultDelegate.__init__(self)

    def handleNotification(self, cHandle, data):
        for i in range(0, len(data), 4):
            value = struct.unpack('I', data[i:i + 4])[0]
            print(value)

def main():
    ble_address = "24:6F:28:22:51:96"

    print("Connecting to BLE device...")
    peripheral = Peripheral(ble_address)
    print("Connected")

    # Set the delegate to handle notifications
    peripheral.setDelegate(MyDelegate())

    # Assuming the characteristic handle is known and is 0x0012
    # You may need to discover the services and characteristics
    characteristic_handle = 0x0012

    # Enable notifications
    peripheral.writeCharacteristic(characteristic_handle + 1, b"\x01\x00")

    print("Listening for notifications...")

    try:
        while True:
            if peripheral.waitForNotifications(1.0):
                # HandleNotification() was called
                continue
            print("Waiting...")
    except KeyboardInterrupt:
        print("Program terminated")
    finally:
        peripheral.disconnect()
```

- On the RPi we use bluepy to connect to a Peripheral, we than create a delegate with a notification handler. This handler will print the value of the updated characteristic.
- In the main loop we wait for notifications

Task5: Communicate sensor data with CoAP from ESP32 to RPi

`Full source code:` *ESP32_client_CoAP* and *RPi_server_CoAP*

`Result:` see **task5.mp4**

`Code`

```
// ESP32_client_CoAP.ino
WiFiUDP Udp;
Coap coap(Udp);

void callback_response(CoapPacket &packet, IPAddress ip, int port) {
  Serial.println("[Coap Response obtained]");

  char p[packet.payloadlen + 1];
  memcpy(p, packet.payload, packet.payloadlen);
  p[packet.payloadlen] = NULL;

  Serial.println(p);
}

void setup() {
  Serial.begin(9600);
  setup_wifi();

  Serial.println("Setup Response Callback");
  coap.response(callback_response);
  coap.start();
}
void loop() {
  if (rfid.PICC_IsNewCardPresent()) {
    if (rfid.PICC_ReadCardSerial()) {
      MFRC522::PICC_Type piccType = rfid.PICC_GetType(rfid.uid.sak);
      Serial.print("RFID/NFC Tag Type: ");
      Serial.println(rfid.PICC_GetTypeName(piccType));

      char buffer[rfid.uid.size];
      array_to_string(rfid.uid.uidByte, rfid.uid.size, buffer);
      Serial.println(buffer);

      Serial.print("Sending ");
      Serial.println(buffer);
      uint8_t* val = reinterpret_cast<uint8_t *>(buffer);
      int msgid =  coap.send(IPAddress(192, 168, 0, 46), 5683, "sensor", COAP_CON,
COAP_POST, NULL, 0, val, 8);

      Serial.println();
```

```
        rfid.PICC_HaltA();
        rfid.PCD_StopCrypto1();
        delay(500);
      }
    }

    coap.loop();
  }
```

- In the ESP32_client_CoAP the coap service is started. When the RFID reader detects a tag the coap service sends a POST request to a defined IPAddress, with route *sensor* and value *val*

```
# main.py
class BasicResource(Resource):
    def __init__(self, name="BasicResource", coap_server=None):
        super(BasicResource, self).__init__(name, coap_server, visible=True,
observable=True, allow_children=True)
        self.payload = "Basic Resource"

    def render_POST(self, request):
        res = BasicResource()
        res.location_query = request.uri_query
        res.payload = request.payload
        print(res.payload)
        return res


class CoAPServer(CoAP):
    def __init__(self, host, port):
        CoAP.__init__(self, (host, port))
        self.add_resource('sensor/', BasicResource())


def main():
    server = CoAPServer("0.0.0.0", 5683)
    print("Server started")
    try:
        server.listen(10)
    except KeyboardInterrupt:
        print("Server Shutdown")
        server.close()
        print("Exiting...")
```

- On the RPi a coap server is started that has one handler for POST requests.
- In the render post the payload of the request is written to the console
- In the main loop the server listens to incoming requests

Task6: Send CoAP data from RPi to Ingress API

`Full source code:` *ESP32_client_CoAP* and *RPi_gateway_CoAP*

Result: see **task6.mp4**

Code: ESP32_client_CoAP is unchanged

```
# main.py
HOST = "http://ingress-api.daellhin.cloudandmobile.ilabt.imec.be/"
sensor_name = "ESP32-via-RPI"
count = 0
RFID_tags = []

def process_RFID_tag(tag_id: int):
    global count

    if tag_id in RFID_tags:
        count = count - 1
        RFID_tags.remove(tag_id)
        print(f"        {tag_id} left the building")
    else:
        RFID_tags.append(tag_id)
        count = count + 1
        print(f"        {tag_id} entered the building")

    current_milliseconds = int(time.time() * 1000)
    data_point = RfidDataPoint(timestamp=current_milliseconds,
rfid_id=str(tag_id), count=count, sensor_name=sensor_name)

    try:
        r = requests.post(url=HOST + "data", params={}, json=data_point.to_dict())
        r.raise_for_status()
        print(f"        Data posted successfully: {r.status_code}")
    except requests.exceptions.RequestException as e:
        print(f"        Error posting data: {e}")

    print(f"        {count} people in the building")

class BasicResource(Resource):
    def __init__(self, name="BasicResource", coap_server=None):
        super(BasicResource, self).__init__(name, coap_server, visible=True,
observable=True, allow_children=True)
        self.payload = "Basic Resource"

    def render_POST(self, request):
        res = BasicResource()
        res.location_query = request.uri_query
        res.payload = request.payload
        print(res.payload)
        process_RFID_tag(res.payload)
        return res
class CoAPServer(CoAP):
    def __init__(self, host, port):
        CoAP.__init__(self, (host, port))
        self.add_resource('sensor/', BasicResource())
```

```
def main():
    server = CoAPServer("0.0.0.0", 5683)
    print("Server started")
    try:
        server.listen(10)
    except KeyboardInterrupt:
        print("Server Shutdown")
        server.close()
        print("Exiting...")
```

- The RPI code has now been adapted with code from the previous lab where the RFID tag is save to the influx DB. There are no other changes to this code

Task7: Communicate sensor data with MQTT from ESP32 to RPi

`Full source code:` *ESP32_client_MQTT*

`Result:` see **task7.mp4**

`Code`

```
// ESP32_client_MQTT.ino
WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 1234567890;

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Create a random client ID
    String clientId = "ESP8266Client-";
    clientId += String(random(0xffff), HEX);
    // Attempt to connect
    // CHANGE LINE BELOW FOR CONNECTING WITH USERNAME AND PASSWORD
    if (client.connect(clientId.c_str())) { //, mqtt_username, mqtt_password)) {
      Serial.println("connected");
      // Once connected, publish an announcement...
      // client.publish("esp32/tagscan", "hello world");
      // ... and resubscribe
      client.subscribe("esp32/tagscan");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
```

```
}

void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
}

void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, mqtt_server_port);
  client.setCallback(callback);

  SPI.begin();
  rfid.PCD_Init();
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  if (rfid.PICC_IsNewCardPresent()) {
    if (rfid.PICC_ReadCardSerial()) {
      char buffer[rfid.uid.size];
      array_to_string(rfid.uid.uidByte, rfid.uid.size, buffer);
      Serial.printf("Detected Tag: %s\n", buffer);

      Serial.printf("Publish message: %s\n", buffer);
      client.publish("esp32/tagscan", buffer);

      rfid.PICC_HaltA();
      rfid.PCD_StopCrypto1();
      delay(500);
    }
  }
  delay(100);
}
```

- The ESP32 code first sets up WiFi after this the PubSubClient(used for MQTT) can be configured, this being the server and its port and the callback. The callback prints the payload that is received from a topic to the terminal.
- In the main loop the when a RFID is detected it published this over the MQTT network on the esp32/tagscan route

Bonus Task8: Control an LED via MQTT (Mosquitto)

> Unfinished

Task9 Modify your Ingress to subscribe to sensor data with MQTT

`Full source code:` *ESP32_client_MQTT* and *Container_ingress_API*

`Result:` see **task9.mp4**

`Code:` ESP32_client_MQTT is unchanged

```python
# main.py
@fast_mqtt.on_connect()
def connect(client: MQTTClient, flags: int, rc: int, properties: Any):
    print(f"Connected to {client._host}")

@fast_mqtt.on_disconnect()
def disconnect(client: MQTTClient, packet, exc=None):
    print(f"Disconnected from {client._host}")

@fast_mqtt.on_subscribe()
def subscribe(client: MQTTClient, mid: int, qos: int, properties: Any):
    print("subscribed: ", mid, qos, properties)

@fast_mqtt.subscribe("esp32/tagscan", qos=1)
async def tagscan(client: MQTTClient, topic: str, payload: bytes, qos: int,
properties: Any):
    print("Tagsscan: ", topic, payload.decode(), qos)

    current_milliseconds = int(time.time() * 1000)
    await post_data(RfidDataPoint(timestamp=current_milliseconds,
rfid_id=payload.decode(), count=1, sensor_name="esp32"))
```

- In the main file of the ingress api MQTT handlers are added
  - on_connect: is called when the API connects to the MQTT broker
  - on_disconnect: is called when the API disconnects from the MQTT broker
  - on_subscribe: is called when a subscription on a topic is successful, in this case we only subscribe to *esp32/tagscan*
  - subscribe: is called when the passed subscription route "esp32/tagscan" has a update
    - We then get the timestamp and call the post_data of the API with the payload(the RFID id)

Questions

Question 1

> Compare and contrast the different protocols you have explored in this assignment: Bluetooth Classic, Bluetooth Low Energy, CoAP, and MQTT. What scenario is each targeted for or good at? Be specific, give examples.

- Bluetooth Classic: point to point connection for modest data speeds with modest energy usage.
  - Scenario: audio streaming, serial connection, Easy to connect/disconnect

- Bluetooth Low Energy: one to many connection for low data speeds with low energy usage. Works with a architectures with services that have characteristics that can be read and written to
    - Scenario: local sensor networks where sensors connect to a BLE server
- CoAP: IP network connection connection, with a HTTP GET, POST, PUT, DELETE like architecture that is optimised for low power devices
    - Scenario: low power networks where IP can be used to connect to a CoAP server, or a CoAP proxy to connect to HTTP services over the wider internet
- MQTT: publish-subscribe star network where clients connect to a broker over IP
    - Scenario: systems where multiple sensors publish data and multiple controllers/apps need to receive updates
- Extra Zigbee: low power mesh networks where some devices can be configured as routers to repeat signals over the network
    - Scenario: local low power local sensor network for large area
    - I am passionate about Zigbee because I use it in my own home for networking sensors and actuators with Home Assistant as coordinator, it is also easy to develop applications for using ESP32 using the Arduino framework