UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

ACQUISIZIONE REAL-TIME DI DATI DA SENSORI INERZIALI INDOSSABILI PER IL RICONOSCIMENTO DI ATTIVITÀ QUOTIDIANE.

REAL-TIME ACQUISITION OF DATA FROM WEARABLE INERTIAL SENSORS FOR THE RECOGNITION OF DAILY ACTIVITIES.

Relatore: Chiar.mo

Prof. *Paolo Crippa*

Correlatore: Chiar.mo

Prof. *Giorgio Biagetti*

Tesi di Laurea di:

Giorgia Pellicciari

A.A 2019/2020

INDICE

INTRODUZIONE……………………………………..…………………2

CAPITOLO1) FASE DI STUDIO

1.1 Tecnologia Bluetooth Low Energy………………..……..………4

1.1.1 Link Layer………………………………………………….6

1.1.2 Protocollo degli Attributi (ATT)…………………………7

1.1.3 Profilo Attributi Generico (GATT)………………………9

1.2 Approccio al mondo UNIX……………………………………..15

1.2.1 Linguaggi C++ e Python…………………………………19

CAPITOLO 2) FASE DI ANALISI

2.1 Programma “receive.py”……………………………………………22

2.2 Definizione dell’obiettivo…………………………………………28

2.3 Gestione dei processi in UNIX……………………………………30

2.4 Programma “btle.py” e protocollo di comunicazione…………..33

CAPITOLO 3) FASE DI SVILUPPO

3.1 Comunicazione tra i processi……………………………………..40

3.2 Cuore del programma “real_time.c”……………………………46

3.3 Funzioni del programma “real_time.c”…………………………..55

3.4 Abilitazione alla ricezione delle notifiche………………………..65

CAPITOLO 4) CONCLUSIONI ………………………………………………71

SITOGRAFIA……………………………………………………………...72

INTRODUZIONE

Il lavoro svolto mira a potersi affiancare ad un ben più ampio progetto riguardante la realizzazione di un dispositivo costituito da un sensore wireless per l'acquisizione di segnali bioelettrici come quello elettromiografico (EMG) e quello elettrocardiografico (ECG), accoppiati a un sensore inerziale adatto al rilevamento dell'attività umana e sviluppato presso il DII – Dipartimento di Ingegneria dell'Informazione, Università Politecnica delle Marche.

In particolare, il dispositivo appena citato sfrutta il collegamento wireless Bluetooth a basso consumo (BLE) in modo tale da consentire un facile interfacciamento con molti dispositivi di elevata fruibilità, come smartphone o tablet e pc, secondo un protocollo di comunicazione ben definito dalle specifiche, in modo da essere compatibile con ogni device.

Fin ora, l'approvvigionamento a questi dati, è avvenuto in modalità off-line in quanto i valori utili ottenuti dal sensore EMG/inerziale, venivano salvati su un file su disco, secondo le istruzioni dettate da un programma scritto in linguaggio Python.

Solo in un secondo momento, cioè alla fine dell'acquisizione, questi dati potevano essere accessibili e quindi resi utilizzabili dall'utente. Perciò, fin ora, si è potuto riscontrare solo un feed-back di avvenuta connessione tra il suddetto sensore e il dispositivo centrale. Questa notifica risultava necessaria per comprendere il momento di inizio di rilevazione dei campioni sul soggetto indossante il dispositivo.

Alla luce di ciò si riscontra la necessità di monitorare in tempo reale i dati acquisiti, allo scopo di avere un migliore controllo su quest'ultimi e, quindi, di permettere all'operatore, di agire tempestivamente nella correzione di uno qualsiasi dei fattori concorrenti un eventuale mismatching verificatosi, evitando cioè, il passaggio di scrittura dei dati sul file.

Queste considerazioni hanno portato alla necessità di migliorare la comunicazione, trasformando questa in real-time.

Il lavoro illustrato in seguito, evidenzia le varie fasi affrontate per conseguire questo scopo, partendo dalla presentazione dei concetti generali che verranno utilizzati nel corpo della trattazione, proseguendo con l'analisi del materiale fornito sul quale ci si accinge ad intervenire, passando per la delineazione, più dettagliata, dell'obiettivo che si intende raggiungere, per poi presentare la parte riguardante lo sviluppo vero e proprio, cuore del lavoro svolto. In ogni capitolo, comunque, non mancheranno chiarimenti ed osservazioni, in quanto l'elaborato ripercorre abbastanza fedelmente gli step realizzati in fase di programmazione. Quindi alcuni risultati verranno anticipati di pari passo agli avanzamenti compiuti, per poi essere riassunti nella conclusione a fine elaborato.

CAPITOLO 1

FASE DI STUDIO

1.1 Tecnologia Bluetooth Low Energy

La tecnologia wireless Bluetooth Low Energy (Bluetooth LE) è stata sviluppata dal Special Interest Group (SIG) ed è stata quindi adottata dalla specifica Bluetooth Core versione 4.0 col fine di raggiungere uno standard di consumo di potenza molto basso.

Questa versione dello standard Bluetooth supporta due sistemi di tecnologia wireless:

- Basic Rate
- Bluetooth Low Energy

La tecnologia Bluetooth Low Energy opera nella banda ISM (industriale, scientifica e medica) da 2,4 a 2,485 GHz. In particolare, questa tecnologia è stata creata allo scopo di trasmettere dei pacchetti di dati molto piccoli per volta, dove ogni pacchetto rappresenta un dato etichettato mediante metadati così da renderlo pronto per essere trasmesso da un dispositivo e ricevuto da uno o più dispositivi, consumando significativamente meno energia rispetto ai dispositivi Basic Rate.

Lo stack Bluetooth Low Energy è costituito da due componenti:

- Controller: include il livello fisico (Phisic Layer) e il livello di collegamento (Link Layer)
- Host: include il controllo del collegamento logico, il protocollo di adattamento (L2CAP), il gestore della sicurezza (SM), il protocollo

degli attributi (ATT), il profilo degli attributi generici (GATT) e il profilo di accesso generico (GAP).

L'interfaccia tra i due componenti è denominata Interfaccia Controller-Host (HCI).

Gli elementi appena elencati, verranno descritti in maniera proporzionale all'importanza che rivestono per la comprensione dei concetti trattati nel presente studio.

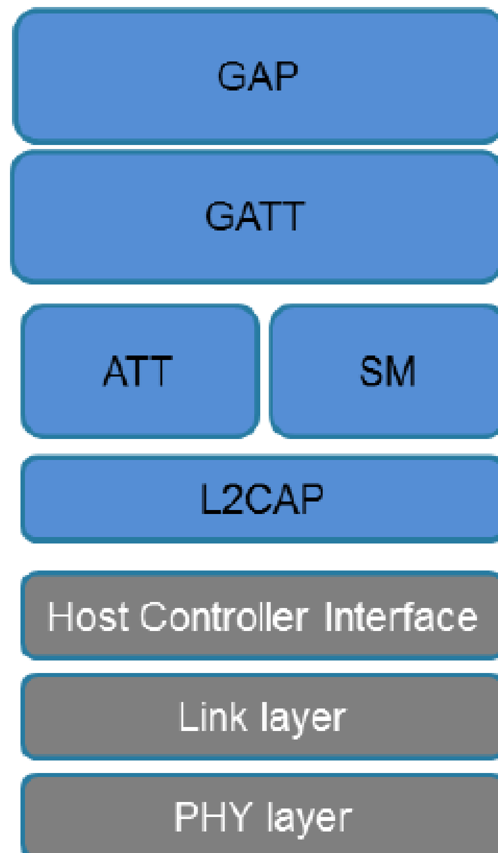

Figura 1 Architettura dello stack Bluetooth Low Energy

1.1.1 Link Layer (LL)

Definisce il modo in cui un collegamento radio mette in comunicazione due dispositivi tra loro, così che questi possano trasmettere informazioni grazie alla pianificazione dei set di advertising definiti dal livello *host*. Considerando entrambi i dispositivi, si possono presentare in totale cinque stati:

- Standby: è lo stato in cui si trovano master e slave quando non ci sono dati da trasmettere.
- Advertising: il dispositivo trasmette pacchetti pubblicitari nei canali addetti, acquisendo la nomina di advertiser device.
- Scansione: il secondo dispositivo ricerca advertiser devices e, per questo, prende il nome di dispositivo scanner.
- Initiating: il dispositivo scanner tenta una connessione col dispositivo che si sta pubblicizzando.
- Connessione: il dispositivo che ha tentato la connessione, se questa va a buon fine, assume il ruolo di master: comunica con il dispositivo nel ruolo di slave e definisce i tempi delle trasmissioni dei pacchetti. Di conseguenza, l'advertiser device assume il ruolo di slave, nome legato al fatto che potrà comunicare con un singolo dispositivo nel ruolo di master.

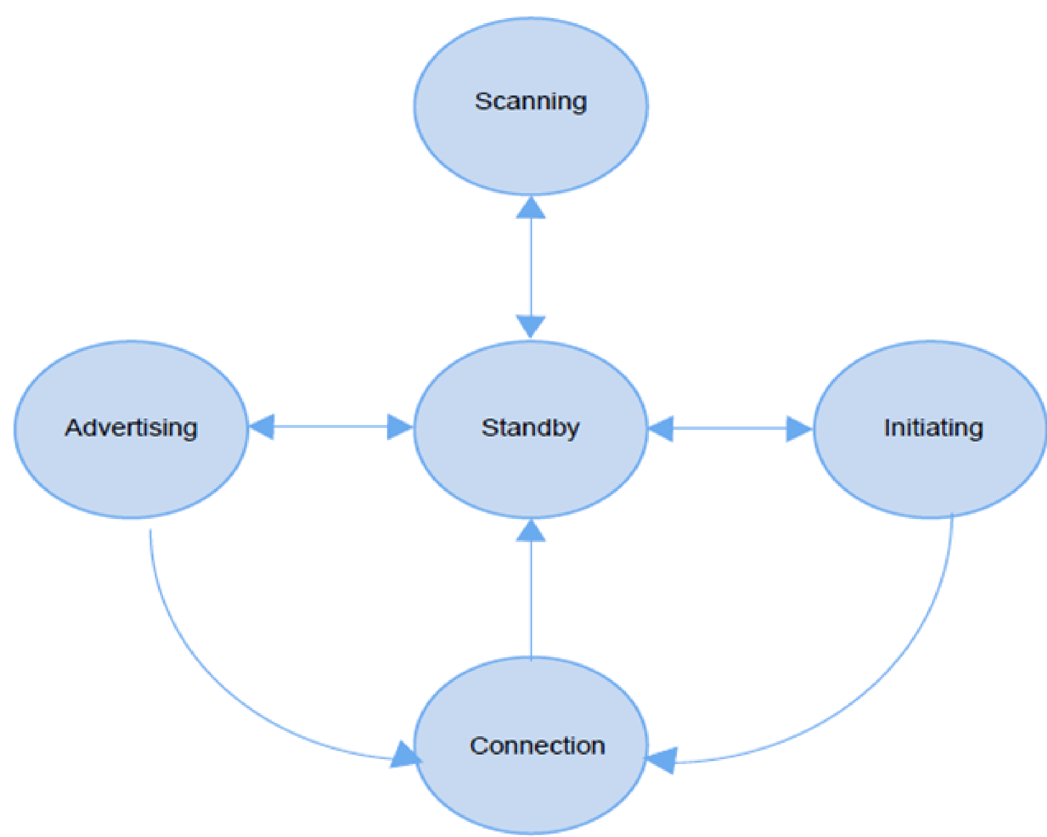

Figura 2 Macchina a stati del Link Layer

Quando un dispositivo in fase di scanning riceve un pacchetto pubblicitario da un dispositivo in advertising, come risposta, può inviare un pacchetto di richiesta di connessione. Se viene stabilita una connessione, il master deve trasmettere un pacchetto allo slave ad ogni evento di connessione per consentire allo slave stesso di inviare pacchetti utili.

1.1.2 Protocollo degli attributi (ATT)

Gli *attributi* sono una forma di scambio di informazioni e possono riferirsi ai dati utili o ai relativi metadati, andando a costituire la più piccola entità di elementi definita sia da GATT che da ATT e detta unità di dati del protocollo (PDU, comunemente noti come pacchetti). Infatti, entrambi i protocolli possono funzionare solo con questi *attributi*, quindi, affinché il

dispositivo centrale e quello periferico possano interagire, tutte le informazioni devono essere organizzate in questa forma. Grazie a ATT, un dispositivo server che definisce gli attributi è in grado di esporre questi dati, noti appunto come pacchetti, ad un altro dispositivo che li utilizza e al quale viene attribuito il nome di *client*.

Protocol data unit (PDU message)	Sent by	Description
Request	Client	Client asks server (it always causes a response)
Response	Server	Server sends response to a request from a client
Command	Client	Client commands something to server (no response)
Notification	Server	Server notifies client of a new value (no confirmation)
Indication	Server	Server indicates to client a new value (it always causes a confirmation)
Confirmation	Client	Confirmation to an indication

Figura 3 Messaggi del Protocollo degli Attributi

Un attributo presenta i seguenti componenti:

- *Handle*: è un valore a 16 bit (2 byte), che identifica un *attributo* di un server. La sua presenza consente al client di fare riferimento all'*attributo* specifico nelle richieste di lettura o scrittura che il client stesso può presentare. All'interno di un server GATT, gli handle sono disposti in ordine crescente determinando una sequenza ordinata ma non continua di attributi, sequenza alla quale un client può accedere. Gli spazi tra gli handle sono consentiti, quindi un client non può fare affidamento su una sequenza contigua per indovinare la posizione dell'attributo successivo.
- *Tipo* di *attributo*: è definito da un Identificatore Univoco Universale (UUID) che determina il *significato del valore*. Può essere un UUID a 16, 32 o 128 bit, che occupa rispettivamente 2, 4 o 16 byte. Questo

campo determina il *tipo di dati* presenti nel *valore* dell'attributo e può rappresentare l'UUID del servizio, gli UUID delle caratteristiche, ecc.

- *Valore*: il *valore* dell'*attributo* comprende il contenuto effettivo dei dati. Non ci sono restrizioni sul tipo di dati che può contenere e, a seconda del *tipo* di attributo, il *valore* può contenere informazioni aggiuntive, descrittive dell'attributo stesso o direttamente relative al dato utile. Questa è la parte di un attributo a cui un client può accedere, ovviamente con i permessi appropriati che lo consentono, sia per la lettura che per la scrittura. Tutte le altre entità costituiscono la struttura dell'attributo e non possono essere accessibili o modificate direttamente dal client (sebbene il client utilizzi indirettamente l'handle e l'UUID nella maggior parte degli scambi con il server).

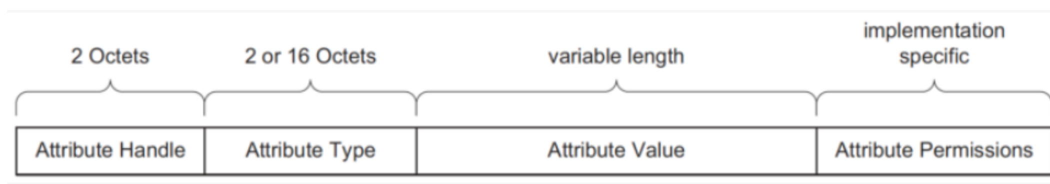

Figura 4 Struttura di un Attributo

1.1.3 Profilo Attributi Generico (GATT)

Il Profilo di Attributo Generico (GATT) definisce un framework cioè un'architettura logica di supporto per l'utilizzo del protocollo ATT, come registro di trasporto per lo scambio di dati tra i dispositivi. In sostanza questo protocollo stabilisce in dettaglio come scambiare tutti i dati del profilo attraverso una connessione BLE. A differenza di GAP che definisce le interazioni di basso livello, GATT si occupa solo dei formati di trasferimento dei dati tra i dispositivi.

Come anticipato nel protocollo ATT, quando due dispositivi sono connessi, si stabiliscono i rispettivi ruoli:

- Client GATT: il dispositivo accede ai dati sul server GATT remoto tramite lettura, scrittura, notifica.
- Server GATT: il dispositivo contiene tutti gli attributi (database degli attributi) e fornisce al client GATT remoto i metodi di accesso ai dati oltre che notificare la variazione di un *valore* di attributo.

Il ruolo GATT di un dispositivo è separato dal ruolo di master o slave: infatti i ruoli di master e slave definiscono la modalità di gestione della connessione radio Bluetooth LE, mentre i ruoli client e server GATT sono determinati dall'archiviazione e dal flusso dei dati, anche se comunemente si ha che il dispositivo slave ricopre il ruolo di server GATT mentre il dispositivo master è il client GATT.

Gli *attributi* in un server GATT sono raggruppati in *servizi*, ognuno dei quali può contenere più *caratteristiche*. Queste caratteristiche, a loro volta, possono includere zero o più *descrittori*. Questa gerarchia viene applicata rigorosamente a qualsiasi dispositivo che pretenda la compatibilità GATT, il che significa che tutti gli attributi in un server GATT sono inclusi in una di queste tre categorie, senza eccezioni.

Per la maggior parte dei tipi di dato nella gerarchia GATT, è importante distinguere tra la loro *definizione*, cioè l'intero gruppo di *attributi* che lo compongono, e la *dichiarazione* specifica: quest'ultima è un *singolo attributo* che viene sempre posizionato per primo secondo l'ordine crescente degli handle all'interno della *definizione*. Il ruolo della *dichiarazione* è quello di introdurre la maggior parte dei metadati per i dati che seguono. Tutte le

dichiarazioni hanno autorizzazioni di sola lettura senza necessità di sicurezza, perché non possono contenere dati sensibili.

Tipo di attributo: Servizio

I servizi sono dei contenitori comprendenti ognuno alcune caratteristiche specifiche del suddetto servizio. In realtà si può affermare che gli *attributi* di un server GATT sono una successione di *definizioni di servizio*, ognuna delle quali inizia con una *dichiarazione di servizio* rappresentata da un singolo *attributo* che segna l'inizio di un servizio.

Attribute handle	Attribute type	Attribute value	Attribute permissions
0xNNNN	0x2800 – UUID for "Primary Service" or 0x2801 – UUID for "Secondary Service"	0xuuuu – 16 bits or 128 bits for service UUID	Read only, no authentication, no authorization

Figura 5 Dichiarazione del servizio

Tipo di attributo: Caratteristica

È possibile intendere le caratteristiche come dei contenitori per i *dati* sensibili dell'utente. Queste includono sempre almeno due attributi: il primo è la *dichiarazione* della caratteristica che fornisce i metadati relativi ai dati utili dell'utente, mentre il secondo è il *valore* della caratteristica che è un *attributo* contenente i dati dell'utente. Inoltre, il *valore* della caratteristica può essere seguito da descrittori, che ampliano ulteriormente i metadati contenuti nella *dichiarazione* delle caratteristiche.

La *dichiarazione*, il *valore* e l'insieme di tutti i descrittori, formano la *definizione* della caratteristica. Tutte le caratteristiche GATT fanno sempre parte di un servizio. Una caratteristica descrive il *tipo* di dati che il *valore*

rappresenta, definisce se il *valore* può essere letto o scritto, determina come configurare il *valore* da notificare, e dice cosa significa un *valore*.

Attribute handle	Attribute type	Attribute value	Attribute permissions
0xNNNN	0x2803 (UUID for characteristic attribute type)	Characteristic value properties (read, broadcast, write, write without response, notify, indicate, …). Determine how characteristic value can be used or how characteristic descriptor can be accessed	Read only, no authentication, no authorization
		Characteristic value attribute handle	
		Characteristic value UUID (16 or 128 bits)	

Figura 6 Dichiarazione della caratteristica

- *Handle* della caratteristica: come già riportato, l'handle è costituito da due byte e contiene il valore effettivo della caratteristica.
- UUID della caratteristica: l'UUID di una caratteristica selezionata, può essere un UUID approvato da SIG nei profili standard o un UUID a 128 bit se personalizzato.
- *Valore* della caratteristica: contiene i *dati* utente effettivi che possono essere letti e scritti dall'utente per gli scambi di informazioni pratiche.

Il *valore* di una caratteristica può contenere qualsiasi *tipo* di dato: tutto ciò che può essere utilmente trasmesso tra due dispositivi BLE può essere assegnato a questo campo.

Tipo di attributo: Descrittori delle caratteristiche

I descrittori vengono utilizzati per, appunto, descrivere il *valore* della caratteristica, fornendo dei metadati ossia un significato aggiuntivo specifico, così da renderla comprensibile per il client. Questi elementi sono

sempre posizionati all'interno della *definizione* della caratteristica e sono costituiti da un singolo attributo il cui UUID è sempre il *tipo* di descrittore e il cui *valore* contiene quanto definito da quel particolare *tipo* di descrittore.

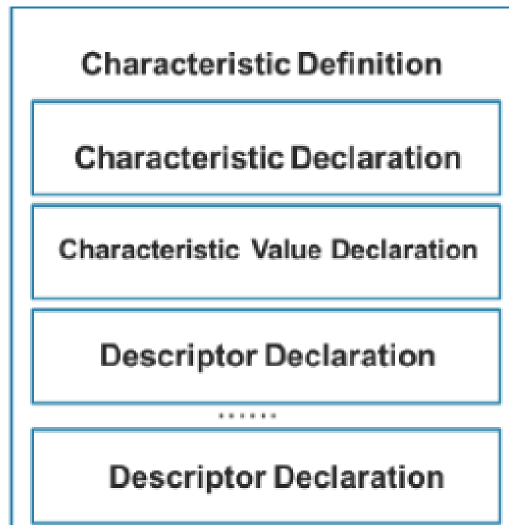

Figura 7 Esempio di definizione della caratteristica

Tra i descrittori, quello che assume un ruolo di spicco, è il Client Characteristic Configuration Descriptor. Questo tipo di descrittore (spesso abbreviato in CCCD) è essenziale per il funzionamento della maggior parte dei profili. La sua funzione è quella di agire abilitando o disabilitando gli aggiornamenti delle notifiche da parte del server, ma solo per la caratteristica specifica che contraddistingue. Il server invia gli aggiornamenti costituiti dal cosiddetto "Handle Value Notifications" in modo asincrono ogni volta che il *valore* di una caratteristica cambia, attraverso un pacchetto formattato che contiene solo un handle di *attributo* e un array di byte per il *valore*. Il valore di un CCCD, che verrà richiamato anche nel capitolo della programmazione così come avverrà per tutti i concetti qui esposti, non è altro che un campo di due bit, dove un bit corrisponde alle notifiche e l'altro alle indicazioni. Un client può impostare

quei bit in un qualsiasi momento e il server li controllerà ogni volta che la caratteristica che li racchiude avrà cambiato *valore*.

Ogni volta che un client desidera abilitare notifiche o indicazioni per una particolare caratteristica, utilizza semplicemente un pacchetto "Write Request ATT" per impostare il bit 1. Il valore del CCCD in questo caso è 0x0001, ad indicare che le notifiche sono abilitate per la specifica caratteristica che lo contiene. Il server risponderà quindi con una risposta in scrittura e inizierà a inviare i pacchetti ogni volta che desidera avvisare il client di una variazione del *valore*.

Il Profilo di Attributo Generico (GATT) definisce quindi un insieme standard di procedure che consentono di conoscere servizi, caratteristiche e descrittori correlati, insieme alla modalità di utilizzo associata ad ognuno di questi elementi.

Tutti i concetti espressi più o meno dettagliatamente in questo capitolo, verranno ampiamente utilizzati in fase di programmazione e progettazione del software, fase indispensabile affinché l'acquisizione diventi real-time.

1.2) Approccio al mondo Unix

In fase iniziale è stata posta l'attenzione sulla scelta del sistema operativo da utilizzare: l'intera applicazione software già esistente, era stata sviluppata in ambiente Linux, per cui erano già in uso quegli strumenti e quelle funzionalità caratterizzanti questo ambiente.

Nonostante ciò, è stata compiuta una piccola ricerca per provare a sviluppare il nuovo codice nel sistema operativo Windows, così da verificare l'esistenza di supporti alla programmazione validi anche su altre piattaforme. Sono state dunque prese in considerazione le librerie Qt rappresentanti lo stack Bluetooth Low Energy per il suddetto OS. Queste librerie, solo da poco tempo, hanno messo a disposizione degli sviluppatori un modulo supplementare chiamato Qt Add-On che implementa le funzioni del classico Bluetooth con quelle Low Energy richieste nel progetto. Sfortunatamente però non esiste uno stack Bluetooth o API completo e multipiattaforma che funzioni su Windows, Linux ecc., per cui, per motivi di continuità e portabilità del lavoro già esistente, abbiamo optato per l'utilizzo del sistema operativo Linux.

Per quanto affermato, parallelamente alla documentazione riguardante l'aspetto della modalità di trasmissione radio Bluetooth, è stato necessario prendere dimestichezza con il sistema operativo Linux e con quegli aspetti necessari per comprendere, verificare ed implementare sia i concetti espressi nel capitolo precedente, sia i codici veri e propri. Il codice a cui ci si riferisce è sia quello già in uso, da modificare e considerato come base di riferimento, sia quello che siamo stati in grado di sviluppare. La scelta del sistema operativo ha comportato quindi la necessità di installazione di una Virtual Machine con distribuzione Ubuntu versione 20.04 e, di

conseguenza, è stato necessario prendere un minimo di dimestichezza con il Prompt dei Comandi o Terminale, così da poter importare le librerie, il compilatore e tutti gli strumenti necessari alla valutazione del codice già esistente e all'implementazione del nuovo.

Per poter interagire con i dispositivi Bluetooth, è stata necessaria l'installazione di una libreria contenente tutte le funzioni appartenenti alla sfera BLE. Per il sistema operativo Linux è disponibile la libreria Bluez. Questa piattaforma mette a disposizione degli sviluppatori sia un intero protocollo API, che alcuni strumenti utili per poter agire direttamente da terminale nominati *hcitool* e *gatttool* che permettono la scansione e la connessione delle caratteristiche dei servizi.

- *sudo hcitool lescan*: utilizza l'interfaccia del controller host del pc per eseguire una scansione di eventuali dispositivi BLE nelle vicinanze. La presenza di un device è rivelata dalla stampa del MAC Address del dispositivo stesso.

```
giorgia@giorgia-VirtualBox:~$ sudo hcitool lescan
LE Scan ...
FC:0D:87:07:B6:77 BT5.1  Mouse
FC:0D:87:07:B6:77 (unknown)
73:ED:AB:D0:42:71 (unknown)
```

Figura 8 Output da terminale

- *sudo gatttool – I*: questa istruzione permette di avviare lo strumento in modalità interattiva così che, digitando *help*, si possa prendere visione di tutte le opzioni e i comandi disponibili.

```
giorgia@giorgia-VirtualBox:~$ sudo gatttool -I
[                 ][LE]>
```

Figura 9 Output da terminale

- *connect <BD_ADDR>:* eseguendo questo comando di seguito al precedente con indicato il MAC Address del dispositivo, si ha la possibilità di evidenziare i servizi e le caratteristiche attribuite alla periferica.

```
[                    ][LE]> connect FC:0D:87:07:B6:77
Attempting to connect to FC:0D:87:07:B6:77
Connection successful
[FC:0D:87:07:B6:77][LE]>
```

Figura 10 Output da terminale

In presenza di eventuali avversità nella connessione si può ricorrere al comando *bluetoothctl* le cui istruzioni appaiono nel terminale in risposta ad *help* e permettono di monitorare le varie fasi della connessione.

```
giorgia@giorgia-VirtualBox:~$ bluetoothctl
Agent registered
[CHG] Controller 5C:F3:70:9C:C1:45 Pairable: yes
[bluetooth]# help
Menu main:
Available commands:
-------------------
advertise                              Advertise Options Submenu
scan                                   Scan Options Submenu
gatt                                   Generic Attribute Submenu
list                                   List available controllers
show [ctrl]                            Controller information
select <ctrl>                          Select default controller
devices                                List available devices
paired-devices                         List paired devices
system-alias <name>                    Set controller alias
reset-alias                            Reset controller alias
power <on/off>                         Set controller power
pairable <on/off>                      Set controller pairable mode
discoverable <on/off>                  Set controller discoverable mode
discoverable-timeout [value]           Set discoverable timeout
```

Figura 11 Output da terminale

Queste azioni preliminari, permettono, in fase di analisi del codice fornito, una migliore capacità di orientamento tra i vari problemi da risolvere come

ad esempio l'individuazione corretta del tipo di una variabile (che nel linguaggio C è fissa) a partire da variabili di tipo mutevole (con riferimento al linguaggio Python), concetti che verranno ampiamente ripresi in seguito.

Inoltre, limitatamente alla fase iniziale, è stato impiegato di un modulo adattatore Bluetooth LE 4.0 USB da applicare al computer. Come emulatore del sensore EMG/GYRO, è stato impiegato, invece, un mouse dotato di tecnologia Bluetooth 5.1.

In un secondo momento è stata messa a disposizione la possibilità di collegarsi da remoto direttamente con il sensore attraverso una Raspberry la quale, tramite i comandi impartiti a distanza, instaurava una connessione Bluetooth direttamente con il sensore.

Quindi, tutte le evidenze riportate in questo documento sono effettivamente quelle risultanti dal dispositivo EMG/GYRO inerziale sviluppato presso il DII e, a tal proposito, si ringraziano nuovamente i docenti per aver permesso lo studio direttamente sul sensore.

1.3 Linguaggi C++ e Python

Il codice già esistente a cui si fa riferimento, rappresenta la parte software che regola lo scambio di informazioni tra il centrale e il peripheral. L'intera applicazione fornita, si presenta costituita da più file sorgenti e più file di header, tutti scritti in C++ tranne che per uno scritto in Python rinominato come *receive.py*, nonché il file di interesse.

Per riuscire a capire le operazioni svolte nel receive.py, è stata necessaria una documentazione sulle caratteristiche del C++, limitatamente alla parte riguardante i concetti di programmazione ad oggetti, di classi, metodi e funzioni, necessari anche per la comprensione del linguaggio Python.

Il linguaggio C++ è un'estensione del linguaggio C. In particolare ne conserva i punti di forza come la flessibilità di gestione dell'interfaccia hardware e software e la possibilità di programmare a basso livello ma, in più, introduce alla programmazione orientata agli oggetti che rende tale linguaggio una piattaforma ideale per l'astrazione dei problemi di alto livello.

Gli *oggetti* costituiscono il costrutto fondamentale e sono definibili come singole entità che combinano sia strutture, dati che comportamenti: i dati sono visti come variabili e le procedure per accedere ai dati sono viste come metodi (o funzioni). Gli oggetti, in sostanza, possono essere visti come dei mattoni assemblabili e caratterizzati da un alto potenziale di riutilizzo.

Un secondo concetto alla base della programmazione ad oggetti è quello di *classe* la quale rappresenta un tipo di dati astratto che può contenere elementi in stretta relazione tra loro e che condividono gli stessi attributi. Un *oggetto*, di conseguenza, è un'istanza di una *classe*.

Le *classi* hanno anche altre funzionalità: un *costruttore* è una *funzione* membro di una classe. I *costruttori* sono utili per inizializzare le variabili della *classe* o per allocare aree di memoria. Ogni volta che viene creato un *oggetto* di una determinata *classe*, viene sempre eseguito il *costruttore* di quella *classe*.

Acquisite queste nozioni, è stato quindi possibile passare allo studio della particolare sintassi Python, anch'essa basata sul concetto di programmazione ad oggetti, la quale però prevede che le istruzioni possano essere trattate in maniera procedurale a differenza della sintassi del linguaggio C.

Un programma scritto con il linguaggio Python assume il nome di *script* in quanto identifica una sequenza di istruzioni che viene interpretata o portata a termine da un altro programma invece che dal processore stesso come nei linguaggi compilati.

Una particolarità che è stata ampiamente percepita nel corso del processo di rielaborazione del programma Python, è che questo linguaggio non si preoccupa dell'efficienza intesa in termini di velocità di esecuzione e di memoria allocata: le variabili vengono continuamente sottoposte a casting di tipo, modificandone la forma, anche solo per pochi passaggi per poi, in alcuni casi, tornare, attraverso semplici conversioni, al tipo di partenza. Questa caratteristica del suddetto linguaggio, lo rende assai contorto da analizzare se l'intenzione è quella di coglierne una visione di insieme, soprattutto se ciò che si ricerca tra le righe di codice è l'essenzialità delle istruzioni.

L'esistente codice Python, agiva registrando i suddetti dati utili in un file il quale, solo successivamente, veniva letto dal corposo codice C++ il cui

compito è di implementare i filtri di estrazione a diverse latenze descritti nel documento che illustra l'hardware e il software alla base del funzionamento del sensore.

CAPITOLO 2

FASE DI ANALISI

3.1 Programma *"receive.py"*

Una volta acquisite le conoscenze necessarie, per comprendere concretamente ciò che il recieve.py compieva, sono state esaminate le classi e i metodi utilizzati al suo interno. Qui si potevano distinguere da un lato le librerie importate per lo svolgimento delle istruzioni generali, come ad esempio *datetime*, *binascii*, *math*, *sys*, e dall'altro il modulo di più interesse denominato *bluepy.btle*, dove *btle* è la libreria contenente le classi e i metodi di nostro interesse, e *bluepy* è la libreria di interfaccia per le comunicazioni BLE per il linguaggio Python.

Si possono quindi esaminare le classi e le funzioni importate nello script recive.py:

- *Peripheral*(): questa classe di Bluepy permette la connessione ad una periferica Bluetooth LE: si crea un *oggetto Peripheral* specificando il suo indirizzo MAC (*arg.host*) e, una volta stabilita la connessione, i servizi e le caratteristiche offerti dal dispositivo possono essere visionati, letti e scritti. Essenzialmente si ricrea ciò che lo strumento *gatttool* aveva rivelato in precedenza.
  I parametri fondamentali sono: il *arg.host,* che come anticipato corrisponde al MAC Address ed è necessario per creare l'*oggetto pheriperal* chiamato *device*; il parametro *addrType* che seleziona il tipo di indirizzo *public* o casuale *random*; il parametro *iface* che permette di impostare l'interfaccia Bluetooth su cui effettuare la connessione: nel codice viene inserito 0 in quanto è l'unica interfaccia in uso.

```
while True:
  try:
    log('* CONNECTING TO [' + arg.host + ']...')
    device = None
    sensor = None
    device = Peripheral(arg.host, 'random', 0)
    break
```

Figura 12 "receive.py": alcune istruzioni del main()

Di questa classe, vengono utilizzate, in ordine, le seguenti funzioni definite in btle.py:

I. *_connect*(): la funzione effettua una connessione al dispositivo indicato dal parametro formale *addr*. Questa sarà una delle funzioni rilevanti anche per produrre il nuovo codice in C, per cui, una volta esaminate le sue istruzioni, le quali verranno ampiamente descritte successivamente, è stata convertita in linguaggio C.

```
self._startHelper(iface)
self.addr = addr
self.addrType = addrType
self.iface = iface
if iface is not None:
  self._writeCmd("conn %s %s %s\n" % (addr, addrType, "hci"+str(iface)))
else:
  self._writeCmd("conn %s %s\n" % (addr, addrType))
rsp = self._getResp('stat')
while rsp['state'][0] == 'tryconn':
  rsp = self._getResp('stat')
if rsp['state'][0] != 'conn':
  self._stopHelper()
```

Figura 13 "btle.py": alcune istruzioni della funzione _connect

II. *withDelegate*(): in ordine di chiamate, dopo la connessione con la periferica viene chiamata questa funzione che memorizza un riferimento a un oggetto il quale nel receive.py prende il nome di *device*. Quest'ultimo verrà quindi richiamato quando si verificano eventi come le notifiche Bluetooth.

```
try:
    device.setDelegate(MyDelegate(device))
    log('* CONNECT')
```

Figura 14 "receive.py": alcune istruzioni del main()

```
def withDelegate(self, delegate_):
    self.delegate = delegate_
    return self
```

Figura 15 "btle.py": definizione della funzione withDelegate() appartenente alla classe BluepyHelper

III. *getServiceByUUID*(): funzione richiamata dalla classe *Sensor*() definita in receive.py. Quest'ultima ha come scopo finale quello di memorizzare l'handle del CCCD e il *valHandle* della caratteristica, cioè l'handle della caratteristica con i dati utili, nella variabile *ctrl*. Questi handle sono sempre gli stessi quando ci si connette ad un dispositivo specifico, per cui si era ipotizzato di ottenere tali valori direttamente tramite la stampa su terminale, senza implementare queste funzioni in C, ma poi, esaminando meglio il codice Python, si è scelto di provare a editare tutte le istruzioni.

La *getServiceByUUID*() restituisce un'istanza di un *oggetto* della *classe Service* che ha l'UUID indicato. *uuidVal*, cioè il parametro passato, è utilizzato per costruire un *oggetto* UUID: il programma passa come argomento un *uuidVal* ottenuto dalla doppia trasformazione conseguita eseguendo dapprima la funzione *uuid5*() su stringhe e valori numerici, quindi attraverso la funzione *UUID*() che processa il risultato della funzione precedente. Il prodotto finale di queste operazioni è memorizzato nell'*oggetto service*.

```
for servicename in self.servicenames:
   service = self.periph.getServiceByUUID(UUID(EMGyro2(servicename)))
```

Figura 16 "receive.py": alcune istruzioni della classe Sensor

```
def getServiceByUUID(self, uuidVal):
   uuid = UUID(uuidVal)
   if self._serviceMap is not None and uuid in self._serviceMap:
       return self._serviceMap[uuid]
   self._writeCmd("svcs %s\n" % uuid)
   rsp = self._getResp('find')
   if 'hstart' not in rsp:
       raise BTLEGattError("Service %s not found" % (uuid.getCommonName()), rsp)
   svc = Service(self, uuid, rsp['hstart'][0], rsp['hend'][0])
```

Figura 17 "btle.py": alcune istruzioni della definizione della funzione getServiceByUuid

IV. *getCharacteristics(startHnd, endHnd, uuid*): restituisce un elenco di *oggetti Characterist* che evidenziano le caratteristiche della periferica. Se non vengono forniti argomenti, restituirà tutte le caratteristiche. Se vengono forniti *startHnd* o *endHnd*, l'elenco è limitato alle caratteristiche i cui handle si trovano all'interno dell'intervallo evidenziato, o alternativamente, si passa l'UUID associato alla caratteristica da individuare.

```
for servicename in self.servicenames:
   service = self.periph.getServiceByUUID(UUID(EMGyro2(servicename)))
   c = service.getCharacteristics(EMGyro2(servicename, 1))
```

Figura 18 "receive.py": chiamata alla funzione getCharacteristics

```
def getCharacteristics(self, startHnd=1, endHnd=0xFFFF, uuid=None):
  cmd = 'char %X %X' % (startHnd, endHnd)
  if uuid:
       cmd += ' %s' % UUID(uuid)
  self._writeCmd(cmd + "\n")
  rsp = self._getResp('find')
  nChars = len(rsp['hnd'])
  return [Characteristic(self, rsp['uuid'][i], rsp['hnd'][i],
          rsp['props'][i], rsp['vhnd'][i])for i in range(nChars)]
```

Figura 19 "btle.py": definizione della funzione getCharacteristics

V. *getDescriptors(startHnd, endHnd)* che restituisce un elenco contenente *oggetti Descriptor* per un certo servizio della periferica connessa.

Attraverso questa funzione si possono ottenere gli handle dei descrittori. Se vengono forniti *startHnd* e *endHnd*, l'elenco è limitato ai descrittori i cui handle sono all'interno dell'intervallo dato.

I dati che si sta cercando di acquisire in maniera real-time provenienti dal sensore, sono contenuti negli handle dei descrittori di cui alcune caratteristiche sono fornite. Le caratteristiche come tali riportano, invece, la descrizione dei dati, ossia il formato. Gli handle sono di 2 byte cioè 16 bit, mentre gli UUID sono di 128 bit.

```
for servicename in self.servicenames:
   service = self.periph.getServiceByUUID(UUID(EMGyro2(servicename)))
   c = service.getCharacteristics(EMGyro2(servicename, 1))
   d = c[0].getDescriptors(self.cccdUUID)
```

Figura 20 "receive.py": chiamata alla funzione getDescriptors

```
def getDescriptors(self, startHnd=1, endHnd=0xFFFF):
   self._writeCmd("desc %X %X\n" % (startHnd, endHnd) )
   resp = self._getResp('desc')
   ndesc = len(resp['hnd'])
   return [Descriptor(self, resp['uuid'][i], resp['hnd'][i]) for i in range(ndesc)]
```

Figura 21 "btle.py": definizione della funzione getDescriptors

VI. *writeCharacteristic(handle, val)*: questa funzione è importante per lo scopo prefisso, in quanto scrive i dati *val* nella caratteristica identificata dall'handle passatole. È attraverso questa funzione che viene inviato alla periferica un pacchetto detto *Write Request* il quale permette di apprezzare il flusso di dati dalla periferica al device centrale.

```
def enable(self, servicename):
  self.periph.writeCharacteristic(self.cccd[servicename], struct.pack("<H", 0x0001))
  print(datetime.datetime.now(), "<", servicename[0:3].ljust(3).upper(), "ENABLE")
  sys.stdout.flush()
```

Figura 22 "receive.py": invocazione della funzione writeCharacteristic

```
def writeCharacteristic(self, handle, val, withResponse=False):
  cmd = "wrr" if withResponse else "wr"
  self._writeCmd("%s %X %s\n" % (cmd, handle, binascii.b2a_hex(val).decode('utf-8')))
  return self._getResp('wr')
```

Figura 23 "btle.ph": definizione della funzione writeCharacteristic

VII. *disconnect*(): questa funzione di btle.py, richiamata alla fine del programma receive.py da una funzione della *classe Peripheral*, elimina la connessione col server GATT e resetta le risorse del sistema operativo associate.

Degno di nota è il fatto che, soprattutto nella fase iniziale di approccio al linguaggio e alle nuove funzioni, è stato ampiamente utilizzato il programma Visual Studio Code e installata l'estensione Microsoft denominata IntelliSense che consente di ottenere informazioni sui parametri (come ad esempio il tipo), sugli elenchi dei membri presenti nel codice (cioè se si aveva a che fare con una funzione, un metodo e l'ambito in cui questo era definito), oltre che di risalire alle definizioni delle varie funzioni usate nel codice e importate attraverso i file di header inclusi. Infatti, le librerie presenti nel file sono molto specifiche e strettamente riguardanti l'ambito delle comunicazioni BLE e quindi le funzioni, non di uso comune, dovevano essere comprese anche attraverso la loro stessa definizione presente in altri file sorgenti così da permettere la comprensione di cosa facessero realmente.

2.2 Definizione dell'obiettivo

In generale l'intero modulo bluepy è caratterizzato dal fatto che internamente utilizza il codice del progetto BlueZ: installando la libreria Bluez e selezionando la cartella bluepy, ci si accorge che questa contiene, oltre che alcuni file Python tra cui appunto il btle.py, anche un file sorgente scritto in linguaggio C di nome "bluepy-helper.c" necessario per l'esecuzione dell'interprete Python. Come anticipato, il Python è un linguaggio interpretato cioè si serve di un software chiamato appunto interprete che recepisce il codice e lo esegue, quindi non c'è nessuna compilazione e non viene creato nessun file eseguibile. Questi linguaggi sono ampiamente utilizzati grazie alla loro alta portabilità che permette di eseguire il codice senza alcuna modifica su tutte le piattaforme per le quali è disponibile il linguaggio utilizzato dall'interprete.

In sostanza, l'utilizzo dello script receive.py, che richiama le funzioni definite nel btle.py, risulta essere subordinato al parallelo processo rappresentato dal bluepy-help.c. Appare evidente quindi che, oltre alla scrittura sul file, per la quale viene impegnata la lenta memoria di massa, vi è un primo passaggio dal linguaggio C++ verso quello Python che, essendo un linguaggio interpretato, utilizza nuovamente il sorgente con estensione .c per essere eseguito, comportando un ulteriore passaggio che si traduce in un rallentamento del programma stesso.

Per cui, l'obiettivo è dunque quello di togliere lo strato Python così che il recieve.py che richiama il bluepy, possa essere emulato in un altro programma scritto in C, senza altri passaggi. Togliendo il doppio passaggio evidenziato, si va ad aumentare di efficienza, e a questo punto ci si può permettere di fare un unico programma in C che riceva tutti i dati

richiamando direttamente le funzioni del bluepy-helper.c. Quindi si deve eliminare l'interprete Python così da ridurre l'inefficienza dovuta ad un linguaggio interpretato che risulta più lento di uno compilato, ottenendo quindi meno strati software che permettono al sistema di diventare più stabile. Infatti la trasformazione in tempo reale, caratteristica che tautologicamente comporta una repentinità di approvvigionamento ai dati, non può essere rallentata da trasformazioni ed interpretazioni, e per questo motivo il nuovo programma verrà implementato in linguaggio C, così che le funzioni che questo utilizza comunicheranno direttamente con alcune delle funzioni presenti nel bluepy-helper.c. Quest'ultimo, in risposta, permetterà al nuovo programma di eseguire correttamente le istruzioni, realizzare il channel con la periferica e scambiare i servizi. È dunque evidente che i due programmi per comunicare, dovranno rispettare un protocollo ben specifico così che vi sia una cooperazione tra il chiamante e il chiamato.

2.3 Gestione dei processi in UNIX

Un processo è un'istanza di un programma in esecuzione cioè è un'attività controllata da un programma che si svolge su un processore. Ogni istanza viene considerata come un processo separato anche se questi possono condividere lo stesso codice, mentre i dati e lo stato rimangono separati.

La gestione dei processi comprende la fase di *creazione* del processo, di *esecuzione* del processo e *controllo* del suo ciclo di vita, quindi la *terminazione* del processo.

La fase di *creazione* induce una naturale gerarchia, chiamata albero dei processi, basata sulla relazione *padre* (creatore) e *figlio* (nuovo processo). Un processo in esecuzione si dice *cooperante* quando può influenzare o può essere influenzato dall'esecuzione di un altro processo: la gestione di processi cooperanti è necessaria quando si devono *condividere risorse* e quando risulta utile permettere delle forme di *comunicazione tra i processi,* oltre che per accelerare il calcolo e la repentinità di approvvigionamento ai servizi offerti nell'altro processo. Secondo quest'ultima considerazione, si può ricondurre l'esempio di processi cooperanti al problema del Client e del Server: il produttore produce informazioni e che sono inviate ed elaborate da un consumatore.

Meccanismi di comunicazione Client-Server:

- *Socket*: un *socket* è un'estremità di un canale di comunicazione virtuale, cioè un'astrazione software progettata per utilizzare le API standard per la trasmissione e la ricezione di dati. Il server rimane in attesa delle richieste del client su una porta specificata; quando riceve una richiesta, se accetta la connessione proveniente dal client, si stabilisce la comunicazione attraverso un *socket*.

Per cui, questo strumento, rappresenta il punto in cui il codice di un processo accede al canale di comunicazione per mezzo di una porta, ottenendo una comunicazione tra processi che lavorano su due macchine fisicamente separate. A questo punto, attraverso il *socket*, il Client sarà in grado di leggere e scrivere sui servizi della periferica attraverso pacchetti di comunicazione (PDU) da trasmettere o ricevere.

- *Pipeline*: Le *pipes* sono dei canali virtuali che permettono il reindirizzamento dei canali standard, cioè il trasferimento, ad esempio, dell'output standard di un programma, ad un'altra destinazione sempre locata nella stessa macchina fisica.

  Queste strutture vengono utilizzate in Linux ed altri sistemi operativi Unix per inviare l'output di un comando o di un programma, nonché di un processo, verso un altro programma o processo, così che sia possibile eseguire ulteriori elaborazioni. Al bisogno, possono essere associate ad un file fisico o ad un altro programma per dar luogo alle *pipelines* cioè vere e proprie condutture virtuali che permettono ad un processo di ricevere un input, di elaborarlo con un programma così da produrre un output e di reindirizzare questo come input di un altro, formando una catena di montaggio in cui ogni programma elabora delle modifiche su questi dati. Alla fine, si otterrà un risultato senza aver adoperato un unico programma monolitico, pesante e complesso ma, al contrario, avendo utilizzato programmi in cascata, più snelli e comprensibili.

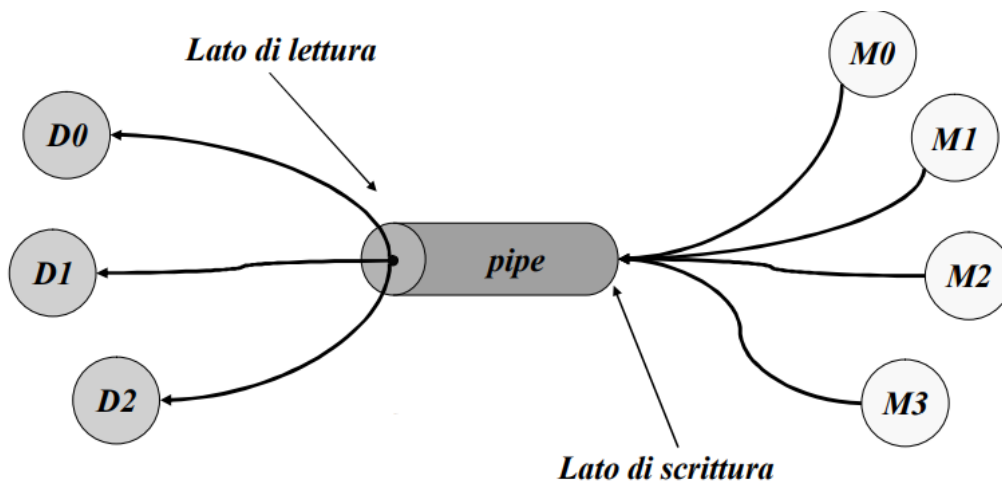

Figura 24 Comunicazione attraverso la pipe tra un Mittente e un Destinatario

I programmi hanno almeno, come standard, 3 canali di comunicazione: *stdin*, *stdout*, *stderr*, uno per dare normalmente i comandi al SO, come ad esempio quelli dati da terminale, uno di output che di default è il video, e l'ultimo è lo standard di error, che anche esso per default va a video ma è riservato per i messaggi di errore.

2.4 Programma “btle.py” e protocollo di comunicazione

Presa coscienza della presenza del bluepy-helper.c e del suo ruolo, è necessario approfondire l’analisi eseguita sul btle.py mirando più in dettaglio verso la ricerca dei binding indicanti lo scambio di informazioni. A tal proposito, se si analizzano in maggior dettaglio le funzioni sopra citate, ci si accorge che queste contengono altre funzioni come ad esempio la _*startHelper*(), la _*writeCmd*(), la _*waitResp*() e la _*stopHelper*() le cui definizioni appartengono ad una classe del sorgente stesso, chiamata *BluepyHelper*.

- “_startHelper()“ : esaminando le sue istruzioni, ci si accorge dell’utilizzo di funzioni di sistema particolari come la *subprocess*() e la *Popen*() che rappresentano il nome che questo linguaggio attribuisce a quell’aspetto caratteristico della programmazione del mondo Linux trattata nel capitolo precedente e che costituiscono la chiave per il metodo di comunicazione tra i due programmi.

```
def _startHelper(self,iface=None):
  if self._helper is None:
      DBG("Running ", helperExe)
      self._lineq = Queue()
      self._mtu = 0
      self._stderr = open(os.devnull, "w")
      args=[helperExe]
      if iface is not None: args.append(str(iface))
      self._helper = subprocess.Popen(args,
                     stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE,
                     stderr=self._stderr,
                     universal_newlines=True,
                     preexec_fn = preexec_function)
```
Figura 25 "btle.py": alcune istruzioni della funzione _startHelper della classe BluepyHelper

Il modulo *subprocess*, così come riportato nella documentazione Python, consente di generare nuovi processi che possano connettersi tra loro per mezzo delle *pipeline* di *input/output/error*, ottenendo i loro valori di ritorno che li identificano. In sostanza, consente di eseguire un programma figlio generando un nuovo processo. Viene utilizzato assieme al costruttore della classe *Popen* che permette la creazione e la gestione del processo figlio, il quale è eseguito solo quando viene richiamato attraverso le *pipeline*. Quest'ultime sono dei veri e propri tubi di collegamento virtuale che permettono l'invio di elementi tra i due processi in quanto risultano connessi sui loro flussi standard, cioè l'output di un processo viene utilizzato come input dell'altro processo.

In sostanza quello che accade è che il processo padre (btle.py) attraverso la *Popen*, genera un processo figlio e tramite le pipeline è in grado di leggere e scrivere a questo processo (il bluepy-helper.c).

Stdin, *stdout* e *stderr* specificano rispettivamente lo standard input, lo standard output e quello di errore standard del programma eseguito: il valore per *stdin* e *stdout* è *PIPE* e ciò indica che deve essere creata una nuova *pipe* verso il figlio, mentre per *stderr* è *devnull* e indica che per questo standard verrà utilizzato il file speciale *os.devnull* che punta a NULL.

Quindi, ad esempio, l'istruzione *stdin=subprocess.PIPE* non farà altro che specificare che lo standard di input dell'helper deve essere connesso alla pipe che si sta creando, e lo stesso discorso deve valere per lo standard di output. Dunque ogni volta che il programma btle.py scrive qualcosa sulla *pipe*, questa verrà letta dal programma in C attraverso una *scanf()* se la lettura avviene ad alto livello e formattata, o *read*() se avviene a basso livello cioè attraverso i *file descriptors*.

Ora il tassello mancante è rappresentato dal protocollo di comunicazione tra i due programmi: è necessario individuare gli elementi di scambio tra i due processi.

A tal proposito si nota che, sempre all'interno del btle.py, sono presenti delle istruzioni di scrittura sullo standard di input *stdin.write*() del processo figlio e delle istruzioni di lettura *stdout.readline*() dallo standard di output sempre del processo figlio e contenute all'interno delle funzioni *_writeCmd*(), *_getResp*() e *_waitResp*().

```
def _writeCmd(self, cmd):
  if self._helper is None:
      raise BTLEInternalError("Helper not started (did you call connect()?)")
  DBG("Sent: ", cmd)
  self._helper.stdin.write(cmd)
  self._helper.stdin.flush()
```

Figura 26 "btle.py": funzione _writeCmd

```
def _getResp(self, wantType, timeout=None):
   if isinstance(wantType, list) is not True:
       wantType = [wantType]
   while True:
       resp = self._waitResp(wantType + ['ntfy', 'ind'], timeout)
       if resp is None:
           return None
       respType = resp['rsp'][0]
       if respType == 'ntfy' or respType == 'ind':
           hnd = resp['hnd'][0]
           data = resp['d'][0]
           if self.delegate is not None:
               self.delegate.handleNotification(hnd, data)
       if respType not in wantType:
           continue
       return resp
```

Figura 27 "btle.py": alcune istruzioni della _getResp

Dallo studio di queste funzioni ci si rende conto che quello che viene scambiato sono delle singole stringhe o, in alcuni casi più specifici, alcuni comandi costituiti dall'unione di più elementi.

Per controprova si può eseguire una simile ricerca sul file bluepy–helper.c e qui ci si accorge che, a livello globale, è definito un *array di strutture* i cui campi sono formati dalle *stringhe* di comando ricevute, dalla *firma* della funzione in cui vengono lette, da eventuali *parametri* che sono da considerarsi come attributi del comando e quindi ne descrivono un comportamento piuttosto che un altro e, infine, dalla *descrizione* dell'azione concreta collegata a tale comando. Queste parole corrispondono a quelle utilizzate nelle funzioni dal btle.py.

```
static struct {
  const char *cmd;
  void (*func)(int argcp, char **argvp);
  const char *params;
  const char *desc;
} commands[] = {
  { "help",       cmd_help,    "",
      "Show this help"},
  { "stat",       cmd_status, "",
      "Show current status" },
  { "quit",       cmd_exit,    "",
      "Exit interactive mode" },
  { "conn",       cmd_connect,    "[address [address type [interface]]]",
      "Connect to a remote device" },
  { "disc",       cmd_disconnect, "",
      "Disconnect from a remote device" },
  { "svcs",       cmd_primary,    "[UUID]",
      "Primary Service Discovery" },
  { "incl",       cmd_included,   "[start hnd [end hnd]]",
      "Find Included Services" },
  { "char",       cmd_char,    "[start hnd [end hnd [UUID]]]",
      "Characteristics Discovery" },
  { "desc",       cmd_char_desc,  "[start hnd] [end hnd]",
      "Characteristics Descriptor Discovery" },
```

Figura 28 "bluepy-helper.c": alcuni membri della struttura dei comandi

Oltre a questa struttura, vengono definite, sempre globalmente, delle *stringhe* non modificabili costituite da tutte le parole che il bluepy-helper scrive sul suo standard di output.

```
static const char
  *tag_RESPONSE  = "rsp",
  *tag_ERRCODE   = "code",
  *tag_ERRSTAT   = "estat",
  *tag_ERRMSG    = "emsg",
  *tag_HANDLE    = "hnd",
  *tag_UUID      = "uuid",
  *tag_DATA      = "d",
  *tag_CONNSTATE = "state",
```

```
static const char
  *rsp_ERROR       = "err",
  *rsp_STATUS      = "stat",
  *rsp_NOTIFY      = "ntfy",
  *rsp_IND         = "ind",
  *rsp_DISCOVERY   = "find",
  *rsp_DESCRIPTORS = "desc",
  *rsp_READ        = "rd",
  *rsp_WRITE       = "wr",
  *rsp_MGMT        = "mgmt",
```

Figura 29 "bluepy-helper.c": alcune stringhe della comunicazione

Ancora, si possono notare nelle prime funzioni del documento, dei *simboli* che vengono aggiunti come prefissi alle stringhe inviate attraverso le *printf*() sullo standard di output, e cioè, per quanto detto sopra, sullo standard di input del processo padre, ovvero il btle.py. L'aggiunta di questi simboli tornerà utile in fase di analisi delle stringhe nel programma padre.

```
static void resp_begin(const char *rsptype)
{
  printf("%s=$%s", tag_RESPONSE, rsptype);
}
```

```
static void send_uint(const char *tag, unsigned int val)
{
  printf(RESP_DELIM "%s=h%X", tag, val);
}
```

Figura 30 "bluepy-helper.c": esempio di due funzioni in cui vengono prefissi i due simboli $ e h

Infine, dando uno sguardo d'insieme all'helper, tra le librerie incluse si nota la presenza di *gatttool.h*, *hci.h* e *glib.h* dove quest'ultima è la libreria che fondamentalmente gestisce i canali IO attraverso l'utilizzo dei *socket*: in particolare, la funzione *GIOChannel *g_io_channel_unix_new(int fd)* è utilizzata per la creazione di un nuovo *GIOChannel** a partire da un *descrittore di file*. Una volta creato il canale, questo può essere gestito dalle funzioni *g_io_channel_read_chars()*, *g_io_channel_write_chars()*.

Per il bluepy-helper.c, *glib* è la libreria che contiene le funzioni di lettura per il programma intero. Questa libreria richiama l'evento *main_loop* al quale passa i dati letti: utilizzando quindi le opportune funzioni di libreria, permette di multiplexare gli eventi che arrivano dal *socket* (strumento utilizzato per creare il canale) e quelli che arrivano dallo standard di input. Il problema, quando si hanno più di due canali di comunicazione aperti come in questo caso, è che se si rimane in ascolto attraverso una funzione di lettura, il programma si blocca e non va avanti finché non arrivano altri dati da leggere. Se contemporaneamente dovessero arrivare dati sull'altro canale, questi verrebbero persi perché il programma è rimasto in ascolto sul canale sbagliato. Per ovviare a questo problema, la *glib* presenta il meccanismo delle *call_back* che servono per multiplexare questi ingressi molteplici, mettendo in evidenza il canale dal quale andare a leggere: il *main_loop* ha il ruolo di attendere un evento, ci sarà un *watcher* che funge da sentinella così che quando arriva un messaggio, ovvero un segnale di nuovi dati su una di queste due porte, sarà in grado di mettere in evidenza il canale da cui ascoltare.

Quindi, al ciclo di eventi principale *g_main_loop,* sarà aggiunto un *GIOChannel** attraverso *g_io_add_watch*(). Qui si specifica a quali eventi si è interessati e si fornisce una funzione da chiamare ogni volta che si verificano questi eventi.

Pe i motivi appena evidenziati non si può prescindere dall'impostare una comunicazione con l'helper in quanto contiene tutte le funzioni e i meccanismi necessari per stabilire una connessione BLE.

Dunque, ora che i concetti base sono stati esaminati e che è stato compreso dove si deve intervenire, ci si accingerà a produrre un programma che chiameremo real_time.c, scritto appunto in linguaggio C, che sia in grado

di interpellare il bluepy-helper.c, stabilendo una comunicazione tra i due processi così come avveniva col precedente programma receive.py e che sia anch’esso in grado di comprendere i messaggi di risposta, ma che, a differenza del programma Python, renda disponibili all’utente i dati ottenuti appena ricevuti by-passando la scrittura sul file.

CAPITOLO 3

Fase di sviluppo

Col senno di poi, è possibile rendersi conto che la scelta di optare per le librerie BlueZ, e quindi per la programmazione in Linux, è stata lungimirante per risolvere i problemi connessi all'esecuzione di processi simultanei attraverso strumenti che, nel sistema operativo Windows, sarebbero stati altamente limitati.

Come evidenziato nel capitolo precedente, sarà necessario utilizzare una funzione di linguaggio C che permetta la coordinazione contemporanea tra il file bluepy-helper.c e il nostro programma da editare. Nel capitolo 2, a grandi linee, sono stati riportati i mezzi messi a disposizione da questo SO per poter affrontare il problema.

La fase di scrittura del codice real_time.c, è iniziata con l'estrapolazione delle istruzioni fondamentali compiute dai due programmi Python, seguendo l'ordine delle situazioni analizzate nel receive.py.

3.1) Comunicazione tra i processi

In UNIX ogni processo in esecuzione viene identificato univocamente da un numero intero positivo o nullo progressivo che viene detto ID di processo (Process Identifier, PID). Quest'ultimo viene usato per riferirsi al processo se si lavora a basso livello cioè al livello del kernel.

Uno dei meccanismi per la comunicazione e la sincronizzazione tra i processi, simile a quello che era stato incontrato nel Python, è rappresentato, anche in linguaggio C, dalle *pipe* cioè un canale di comunicazione tra processi che in questo caso, però, risulta essere unidirezionale, nel senso che

un processo può solo scrivere nella *pipe* mentre l'altro può solo leggere dalla *pipe*.

Attraverso la chiamata alla funzione *pipe*() si crea una nuova pipeline specificando il parametro fd[2] che è un vettore di 2 *file descriptors*: uno che si riferisce all'estremità di lettura della *pipe* mentre l'altro che si riferisce all'estremità di scrittura e che ricoprono una funzione paragonabile a quella di due porte di comunicazione, per cui ogni processo è in grado di chiudere quella che non viene utilizza perché riservata all'utilizzo dell'altro processo.

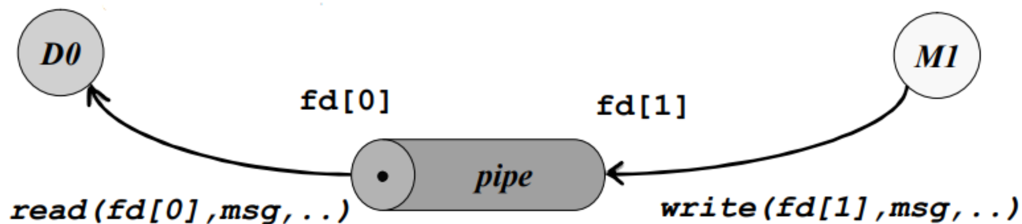

Figura 31 Esempio di comunicazione con descrittori di file, funzioni di lettura e scrittura

fd[0] rappresenta il lato di lettura dalla *pipe* mentre fd[1] è il lato di scrittura nella *pipe*. Se la funzione ha successo: vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi *file descriptor* vengono assegnati a fd[0] e fd[1].

Quanto descritto è ciò che avviene per la creazione di una singola *pipe*. Dunque, affinché la comunicazione avvenga in maniera bidirezionale si necessita l'impiego di due tubi di comunicazione, quindi si sfrutteranno due invocazioni alla suddetta funzione utilizzando come parametro l'indirizzo della prima locazione di memoria di un array di 4 interi, dove i primi due elementi verranno utilizzati per la prima *pipe* mentre i restanti due per la seconda.

```
pid_t launch(){
  pid_t pid;// process identifier
  int fds[2];
  int pipes[4];

  pipe(&pipes[0]);
  pipe(&pipes[2]);
```
Figura 32 "real_time.c": alcune istruzioni della funzione launch()

Creati i collegamenti, si richiama la funzione *fork*() che permette di clonare il processo attuale, cioè di dividere in due processi identici quello in uso, così che uno dei due processi, il figlio, possa essere adoperato per eseguire il nuovo processo (e in questo caso la funzione *fork*() assume come valore di ritorno il PID=0) mentre l'altro processo continua normalmente nell'esecuzione delle istruzioni successive (e la funzione *fork*() ritorna un PID maggiore di 0). Il processo figlio è un duplicato esatto del processo padre tranne che per il proprio ID di processo che lo identifica univocamente.

```
pid=fork();

  if (pid<0){
    printf("error! cannot fork!!!\n");
    exit(1);
    }
  else

  {
    if(pid==0){ // processo figlio
```
Figura 33 "real_time.c": alcune istruzioni della funzione launch()

A questo punto le due *pipes* del figlio devono essere copiate su *stdin* e *stdout* cioè sullo standard di input e su quello di output (ovvero sul *file descriptor* 0 e sul *file descriptor* 1). Infatti, ad ogni canale standard, definito come uno

stream (cioè di tipo *FILE**), utilizzato ad alto livello ed associato a funzioni di lettura/scrittura formattata, corrisponde un *file descriptor* usato dal kernell al quale è associata una funzione di lettura/scrittura di più basso livello, cioè ad una *read*() o una *write*(). Le *pipes* hanno numeri progressivi associati, quindi si devono assegnare queste nuove *pipes* del processo figlio a quelle vecchie, e per fare ciò si utilizza la funzione *dup2*() che assegna il vecchio *file descriptor* al nuovo.

Perciò, all'interno del processo figlio, affinché l'immagine di processo corrente possa essere sostituita con quella di un nuovo processo, verrà chiamata la funzione *exec*(). Questa permette di eseguire il nuovo programma, nonché il bluepy-helper.c, ricevendo come parametri il percorso di inclusione, il nome del file ed eventuali altre stringhe di argomenti che nel nostro caso non servono, per cui sarà necessario indicare semplicemente che la lista degli argomenti è costituita da una stringa vuota.

```
close(pipes[0]); //il figlio chiude la porta da cui il padre legge
   close(pipes[3]); //il figlio chiude la porta su cui il padre scrive

   dup2(pipes[2], 0);
   dup2(pipes[1], 1);

   execl("/usr/local/lib/python3.5/dist-packages/bluepy/bluepy-helper", "bluepy-helper",(char*)0);
```

Figura 34 "real_time.c": alcune istruzioni della funzione launch()

Come accennato, le funzioni *read*() e *write*() agiscono a basso livello, quindi necessiterebbero di una successiva formattazione. Per questo motivo, nel real_time.c, si cercherà di leggere e scrivere ad un livello più elevato, utilizzando funzioni come la *getline*() e la *fprintf*(). Questo salto di livello si traduce nella necessità di utilizzo non più del *file descriptor* come per le funzioni *read*() e *write*() ma del corrispondente *file pointer*. Da notare è il fatto che queste funzioni appena introdotte non sono istantanee ma utilizzano

un buffer sul quale memorizzano i dati. *fflush*() agisce sul *file pointer* ed è necessario per forzare le funzioni di libreria del C nello spedire effettivamente il buffer nella pipe.

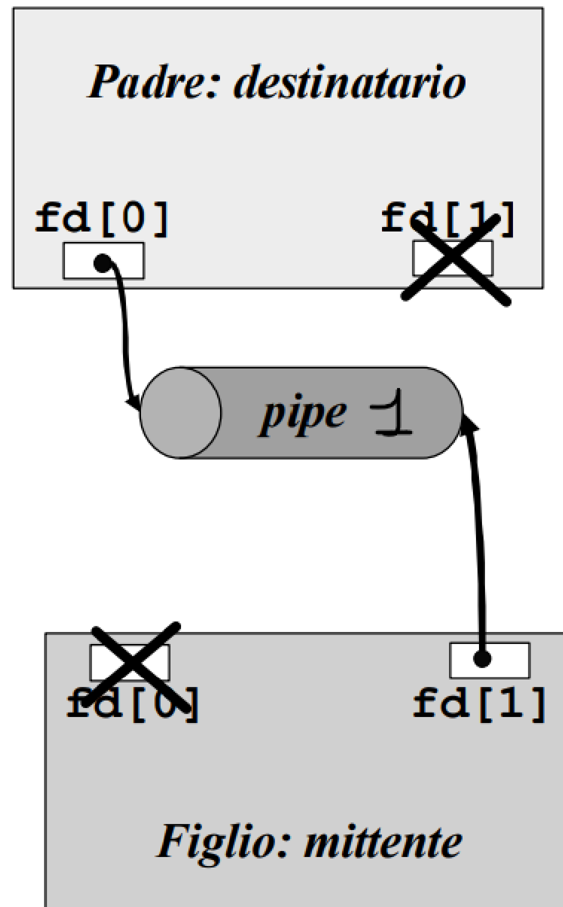

Figura 35 Esempio di comunicazione tra processo padre e figlio nel caso di una unica pipe

Esaminate le istruzioni per gestire l'avvio del processo figlio, si prende in considerazione ciò che avviene nel processo padre. Questo, in ordine, assegnerà ai *descrittori di file,* identificanti *l'input* e *l'output*, le due porte di comunicazione create con le *pipes*, mentre chiuderà quelle all'altro capo della comunicazione. Ora, per poter eseguire operazioni di lettura e scrittura sull'helper, come avviene in generale per tutti i file, si dovrà aprire quest'ultimo sia in modalità di lettura che in modalità di scrittura. Per poter

fare ciò, però, non si può ricorrere all'utilizzo della funzione *fopen*(). Quindi si è optato per l'impiego della funzione *fdopen*() che permette di creare un *file pointer* a partire dal *file descriptor* fornitogli come parametro. In questo modo, indicando una volta l'apertura della comunicazione in modalità di lettura e una volta in modalità di scrittura, sono stati ottenuti come valori di ritorno rispettivamente i *file pointers fpr* e *fpw*.

```
else{ // processo padre

  fds[0]=pipes[0];
  fds[1]=pipes[3];

  close(pipes[1]);
  close(pipes[2]);

  if(((fpw=fdopen(fds[1], "w" )) == NULL) || ((fpr=fdopen(fds[0], "r" )) == NULL )){
    printf("error! cannot use file pointer!!!");
    exit(2);
  }
  else{
    return pid;
  }
```

Figura 36 "real_time.c": alcune istruzioni della funzione launch()

Arrivati a questo punto, per tenere d'occhio eventuali errori nella scrittura del codice, essendo il discorso delle *pipes* relativamente intricato, è stato utile richiamare da terminale il tool *strace ./real_time* (dove *strace* sta per system trace). Secondo quanto riportato nel manuale utenti Unix, con questo comando, costituendo uno strumento di debugging, si riesce a tener traccia delle chiamate di sistema. Utilizzando la funzionalità *strace*, si verifica che la prima riga, cioè quella di intestazione dell'helper, viene effettivamente trasmessa nella *pipe*: se questa riga venisse stampata a schermo vorrebbe dire che il sistema di comunicazione costruito con le *pipes* non ha funzionato correttamente.

L'output dell'helper deve essere letto dalla funzione *getline*() eseguita richiamando la *_getR*().

```
write(1, "Fri Nov 27 10:42:01 2020\n", 25 Fri Nov 27 10:42:01 2020) = 25
write(1, " CONNECTING TO: E3:E8:06:20:8B:9"..., 33 CONNECTING TO: E3:E8:06:20:8B:99) = 33
pipe([3, 4])                          = 0
pipe([5, 6])                          = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x76f2cfd8) = 771
close(4)                              = 0
close(5)                              = 0

read(3, "# bluepy-helper.c version 1.3.0 "..., 4096) = 129
write(1, "*# bluepy-helper.c version 1.3.0"..., 66*# bluepy-helper.c version 1.3.0 built at 12:54:59 on Dec  3 2018) = 66
```

Figura 37 "real_time.c": output del tool strace

Strace ./real_time quindi, permette di vedere tutte le chiamate di sistema che vengono fatte nel programma: si evidenziano le chiamate alle *pipes* e i numeri progressivi interi assegnati alle porte di comunicazione 3-4 per la prima e 5-6 per l'altra. I *file descriptors* 0,1,2 non sono assegnati a queste *pipes* da noi generate perché vengono aperti ed utilizzati dal sistema. Si nota anche la presenza della *fork*() che appare con il nome di *clone*(). Tutto quello che è evidenziato, sono le azioni compiute dal padre, il processo figlio non appare. Inoltre vengono indicate le *read*() e le *write*() alle quali viene passato il *file descriptor* sul quale devono agire, oltre che la stringa da inviare nel tubo di comunicazione. Come valore di ritorno si ottiene il numero di bytes corrispondenti ai caratteri inseriti da tastiera.

3.2 Cuore del programma "real_time.c"

Fissate le basi che permettono l'esecuzione in cascata dei due processi, si può dunque proseguire riprendendo le funzioni che inizialmente erano state esaminate e appartenenti al receive.py per, sostanzialmente, implementarle nel nuovo programma in linguaggio C, non perdendo di vista la ricerca dell'essenzialità delle istruzioni al fine di una maggiore efficienza.

Nella parte iniziale del programma si chiederà all'utente l'indirizzo MAC del device BLE a cui connettersi, supponendo che la fase di pairing sia già stata eseguita in precedenza.

Verrà stampato sulla linea di comando il messaggio "CONNETTING TO" seguito, appunto, dall'indirizzo della periferica con l'orario e la data. Questa azione sarà richiesta fintanto che non verrà individuato un indirizzo valido, mentre, al suo termine, si otterrà l'equivalente di quello che era nel Python l'oggetto *device*: in C quest'ultimo sarà rappresentabile attraverso una *struttura* i cui campi sono costituiti dal suo MAC Address, dal tipo di indirizzo *random* o *public* e dall'interfaccia *HCI* che, per default avrà valore 0 essendo l'unica disponibile.

```
typedef struct{
    char *addr;
    char *addressType;
    int iface;
}device;

device dev={NULL,NULL,0};
```

Figura 38 "real_time.c": struttura device

Posta la struttura tra le variabili globali, verrà richiamata la funzione *peripheral*() che provvederà da subito a far partire i due processi ampiamente descritti sopra, permettendo quindi l'invio da parte del processo padre delle prime stringhe di comando in favore del processo figlio, il bluepy-helper.c. Ovviamente, da questo momento in poi, si esclude intenzionalmente l'intervento dell'utente in quanto si presuppone la conoscenza delle stringhe di comunicazione tra i due processi, per cui, avendo come obiettivo quello di far compiere al codice un'azione specifica, il comando sarà già impostato in fase di editing.

```
void main(){
  size_t len=0;
  char *command=NULL;
  services serv=EMG;
  uuid_t out;
  char mess[40]="CONNECTING TO: ";
  dev.addr=(char*)malloc((20)*sizeof(char));
  getline(&dev.addr,&len,stdin);
  strcat(mess,dev.addr);
  while(1){
  login(mess);
  dev.addressType="random";
  dev.iface=0;
  peripheral();
  if(dev.addr!=0)
    break;
  }
  login("CONNECT\n");
```

Figura 39 "real_time.c": alcune istruzioni del main()

Quindi, il bluepy-helper, attraverso la *pipe* dalla quale legge, riceverà la richiesta di connessione all'indirizzo salvato nella struttura *device*. La risposta che questo invia, sarà letta attraverso la funzione _*getR*().

Eseguendo dal terminale quanto fatto fin ora, e inviando alcuni comandi all'helper manualmente (azione possibile in fase di sperimentazione col fine di compiere un testing di quanto editato), si può comprendere in maniera più coscienziosa i prossimi passi da percorrere: le risposte del sotto processo risultano essere delle intere righe formate da più coppie di stringhe, separate dal carattere speciale ASCII di terminazione record e, in particolare, in ogni coppia di stringhe, l'operatore "=" risulta essere il separatore tra le due parti. Si può evidenziare quindi una corrispondenza tra la stringa precedente tale simbolo e quella successiva: la prima risulta essere il *tag*, cioè la *chiave* della comunicazione, mentre la seconda rappresenta il *valore* corrispondente. A seconda della fase di comunicazione

Bluetooth, quindi, si riceveranno un numero e una tipologia di coppie *chiave-valore* diverso.

```
rsp=$findhnd=h13props=h10vhnd=h14uuid='cef92ff1-4179-5abe-9ab4-f41a779bd20f
hnd=h17props=h8vhnd=h18uuid='cef92ff2-4179-5abe-9ab4-f41a779bd20f
```

Figura 40 "real_time.c": output della riga di risposta dell'helper

In più, c'è da evidenziare il fatto che l'helper, come anticipato precedentemente, aggiunge come prefisso ai valori delle coppie *tag-valore*, un operatore la cui presenza è necessaria per il programma ricevente e sta ad indicare il formato di interpretazione della risposta: il "$" indicherà una stinga, la "h" anticiperà un valore esadecimale, il " ' " significherà un intero e infine una " b " un tipo binario. In realtà questa convenzione risulta più utile al programma Python che a quello in C per via della facilità di casting di tipo delle variabili Python e quindi della possibilità di cambiare il formato dei dati nella maniera più conveniente. In ogni caso, si è cercato di non perdere nemmeno questa informazione, utile magari in una successiva fase di interpretazione della ricezione Bluetooth.

Operativamente, quindi, le righe della risposta dell'helper sono state splittate in primo luogo in corrispondenza del carattere speciale ASCII e quindi in corrispondenza dell'uguale.

Ora, se il programma Python utilizzava una lista di dizionari *chiave-valore* in cui i *valori* erano trattati a loro volta come delle *liste*, la tipologia di rappresentazione più somigliante ed adatta nel linguaggio C, è quella di un *array di strutture* di lunghezza massima di elementi ampiamente sufficiente per contenere il reale numero di coppie di stringhe da estrapolare, mentre i campi *tag* e *valore* saranno costituiti da vettori di tipo *char* la cui dimensione sarà allocata dinamicamente attraverso la funzione *malloc*(), nella quale si

riserveranno tanti byte di tipo *char* quanti sono i caratteri che compongono la suddetta stringa, più un byte finale riservato all'operatore "\0", necessario per ottenere una stringa ben formata. Un terzo campo viene aggiunto per memorizzare il prefisso del *valore* e un ulteriore campo è inserito per contenere un numero progressivo il cui valore scorrerà in un range tra 0 e la lunghezza reale del vettore di strutture stesso e che tornerà utile in fase di ordinamento dei campi della struttura, dove sarà necessario rintracciare l'ordine di acquisizione degli elementi.

```
typedef struct{
  char *tag;
  int index;
  char type;
  char *tval;
}tagtval;

tagtval tgvl[1000];

int len_tgvl; //lunghezza effettiva del vettore di strutture
```

Figura 41 "real_time.c": array di strutture tgvl

La funzione *parseResp*() è la responsabile nell'eseguire queste azioni, ricevendo come parametro la riga di risposta utile dell'helper, quest'ultima precedentemente selezionata dalla funzione *_getR*().

```
parseResp: 0 tgvl[0].tag=*rsp* tgvl[0].tval=*stat*
parseResp: 1 tgvl[1].tag=*state* tgvl[1].tval=*tryconn*
parseResp: 2 tgvl[2].tag=*dst* tgvl[2].tval=*E3:E8:06:20:8B:99*
parseResp: 3 tgvl[3].tag=*mtu* tgvl[3].tval=*0*
parseResp: 4 tgvl[4].tag=*sec* tgvl[4].tval=*low*
parseResp: 0 tgvl[0].tag=*rsp* tgvl[0].tval=*stat*
parseResp: 1 tgvl[1].tag=*state* tgvl[1].tval=*conn*
parseResp: 2 tgvl[2].tag=*dst* tgvl[2].tval=*E3:E8:06:20:8B:99*
parseResp: 3 tgvl[3].tag=*mtu* tgvl[3].tval=*0*
parseResp: 4 tgvl[4].tag=*sec* tgvl[4].tval=*low*
```

Figura 42 "real_time.c": output dei campi tgvl.index, tgvl->tag, tgvl->tval della stringa letta dall'helper ed analizzata

La lettura delle righe avverrà in maniera continuativa all'interno di un loop infinito, capace di interrompersi solo nel caso in cui il valore di ritorno della funzione *waitResp*(), richiamata dalla stessa *_getR*(), abbia valore 0, ad indicare che il tipo di *tag* ricercato all'interno della riga acquisita è stato individuato oppure, in caso contrario, continuerà ad esaminare le righe successive.

```
void _getR(char *wantType){ //"wantType" è il type value associato al tag
  size_t size=0, n;
  char *line;
  while((n=getline(&line,&size,fpr))!=-1)
    {
      if((n>0) && (line[n-1]=='\n'))
        line[--n]=0;
      if((line[0]=='#')||(line[0]=='\n')||(strlen(line)==0))
         continue;
      int cntrl=waitResp(wantType, line);
      if (cntrl==-1){
        fflush(stdout);}
      else {
        if(cntrl==0)
        break;
       }
    }
}
```

Figura 43 "real_time.c": funzione _getR()

La funzione chiamante, ossia la *waitResp*(), subito dopo, si preoccuperà di invocare la funzione *search*(), indicando come parametro il *tag* da cercare.

```
int waitResp(char *wantType, char *line){
  int i;
  len_tgvl=parseResp(line);
  fflush(stdout);
  key=search("rsp");
  fflush(stdout);
  for(i=key.index;i<key.index+key.l;i++){
    if(strcmp(tgvl[i].tval,wantType)==0){
      fflush(stdout);
      return 0;
    }
    else{
      if(strcmp(tgvl[i].tval,"stat")==0){
        exit(1);
      }
      else{
        if(strcmp(tgvl[i].tval,"err")==0){
          continue;
        }
        else{
          exit(1);}
      }
    }
  }
  return -1;
}
```

Figura 44 "real_time.c": definizione della funzione waitResp()

In questa funzione *search*(), ci si affiderà ad una nuova struttura denominata *key* e costituita da due campi di tipo intero, il primo adibito allo storage dell'indice per il quale si verifica la corrispondenza tra il *tag* passato come parametro e i *tag* salvati nell'array di strutture *chiave-valore*, mentre il secondo campo conterrà un numero intero indicante il numero di corrispondenze trovate. Solo a questo punto si può procede con il confronto tra i *valori* delle chiavi selezionate nella struttura key e il *valore* della chiave che era stato passato come parametro alla funzione *waitResp*().

```
srch search(char *str){ //str è il tag da cercare
  int i,j;
  srch key={0,0};
  for(i=0;i<len_tgvl;i++){
    if((strcmp(tgvl[i].tag,str))==0)
    {
      key.index=i;
      key.l=1;
      j=i+1;
      while((j<len_tgvl)&&(strcmp(tgvl[j].tag,str))==0)
      {
        key.l++;
        j++;
      }
      return key;
    }
  }
  printf("search: non son entrato nell'if quindi torno key={0,0}\n");
  return key;
}
```

Figura 45 "real_time.c": definizione della funzione search()

Parallelamente a tutte queste azioni si è cercato di implementare una soluzione alternativa nel caso in cui non vi fossero corrispondenze o insorgessero alcuni errori. Ciò è stato è possibile attraverso le *exit*() o attraverso il semplice ma efficace strumento di debug (non riportato nelle figure per non appesantire l'immagine), rappresentato dalla stampa sul terminale dopo ogni istruzione, così da prendere visione del nuovo quadro che si veniva a delineare.

Come ultima funzione da descrivere in questo paragrafo, ci si può soffermare sulla *qsort*(). Questa è una funzione personalizzabile della libreria C che permette di riordinare il vettore di strutture *tgvl,* i cui campi sono stati riempiti dalle funzioni precedenti, secondo l'ordine alfabetico delle *chiavi* della struttura. Il problema sta nel fatto che, nel caso delle caratteristiche e dei descrittori, vengono acquisiti più *tag* dello stesso tipo perché riferiti a caratteristiche o descrittori differenti ma appartenenti allo stesso servizio, quindi è fondamentale poter riassegnare ogni coppia *chiave-valore* alla specifica sede di appartenenza. Per questo motivo, tra i parametri

attuali, verrà inserita la funzione *compare_keys()* che permette di ordinare i *tag* non solo secondo l'ordine alfabetico ma, nel caso in cui due di essi siano uguali, anche secondo l'indice di acquisizione temporale specificato nel campo *int index* della struttura *tgvl*. In questo modo sarà sempre possibile rintracciare le appartenenze corrette.

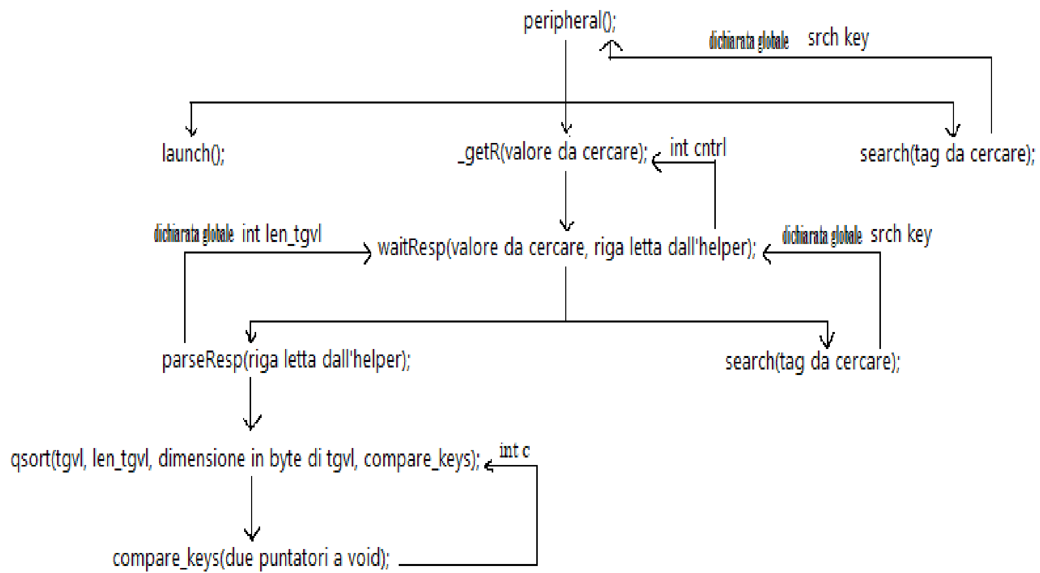

Figura 46 "real_time.c": ricostruzione delle chiamate di funzione e dei valori di ritorno

3.3 Funzioni del programma “*real_time.c*”

Illustrato il cuore del procedimento, si può analizzare il problema da un’ottica più elevata e spiegare quindi le altre funzioni il cui svolgimento si basa su quello delle funzioni viste nel precedente paragrafo. Seguendo indicativamente l’ordine delle funzioni presente nel *main()*, si incontra:

Peripheral(): contiene al suo interno la funzione *launch*() che, non a caso, viene eseguita per prima. Quest’ultima contiene tutte le istruzioni necessarie per un corretto avvio del processo figlio, le quali sono state esaminate precedentemente. Quindi, verranno inviate al file bluepy-helper.c, le stringhe costituite in primis dal comando *conn* seguito dal MAC Address del dispositivo al quale ci si deve connettere e al tipo di indirizzo che, come già evidenziato, può essere *random* o *public*. Si entra quindi in un ciclo infinito il cui ruolo è quello di tentare una connessione fintanto che il valore corrispondente al tag *state* risulta essere la stringa di valore *tryconn*.

```
do{
  _getR("stat"); //cerca il valore
  key=search("state"); //cerca il tag
  for(i=key.index;i<key.index+key.l;i++){
    if(strcmp(tgvl[i].tval,"tryconn")==0){
      continue;
      }
    else{
      if(strcmp(tgvl[i].tval,"conn")!=0){
        exit (1);
        }
      else{
        controllo=1;
        break;
        }
       }
    }
    if(controllo!=0)
      break;
  }while (1);
}
```
Figura 47 "real_time.c": alcune istruzioni della funzione peripheral()

Quindi si prosegue stabilendo la connessione con la periferica, verificando che in corrispondenza del tag indicato con *stat* ci sia il valore pari a *conn*, condizione che permette di uscire dal ciclo. In quest'ultimo stato, l'host può accedere alla maggior parte delle funzioni del sensore per le quali si hanno i permessi.

Dunque, una volta impostata la connessione, la funzione restituisce il controllo al *main*() che può successivamente chiamare la funzione *log*in(), la quale permette la stampa a video del messaggio di connessione avvenuta, affiancato dall'ora e dalla data. Altrimenti, se la connessione non può essere stabilita, il programma esce.

Sensor() : questa funzione deve il suo nome a quello della classe omonima presente nel file receive.py, in quanto, effettivamente, ne esegue gli stessi compiti. Lo scopo finale di tutte le istruzioni eseguite tra il receive.py e il btle.py è quello di memorizzare l'handle corrispondente al Client Characteristic Configuration nella variabile *cccd* e successivamente di memorizzare l'handle corrispondente all'UUID che identifica la caratteristica con i dati utili, nella variabile *ctrl*. Questi due handle sono sempre gli stessi quando ci si connette ad uno stesso dispositivo, motivo per il quale si era ipotizzato di farci stampare a terminale, direttamente dal Python, gli handle specifici del mouse, evitando di implementare queste funzioni ma poi, esaminando meglio il codice Python, le istruzioni non sembravano troppo infattibili, per cui si è scelto di provare ad editare tutte le funzioni.

Effettivamente, la scelta di implementarle piuttosto che farsi stampare dei valori specifici, è risultata corretta in quanto, per continuare a sperimentare, è stata gentilmente concessa la possibilità di connettersi da remoto ad una Raspberry posizionata nelle vicinanze del sensore vero e proprio, per cui,

attraverso la generalità delle funzioni implementate, è stato possibile ricavare anche gli handle per il sensore.

Di fatto, attivando la connessione Bluetooth da remoto tra i due dispositivi, abbiamo provato a visualizzare i dati veri e propri captati dal sensore invece che quelli relativi al mouse.

Tornando alla funzione *sensor*() del programma .c implementato, riceve come parametro uno dei tre elementi dell'enumerazione rappresentanti il servizio in uso (*EMG*, *GYRO*, *SYSTEM*), dichiarati tra le variabili globali. Il primo servizio personalizzato è il servizio di elettromiografia, il secondo è il servizio di piattaforma inerziale, il terzo è il servizio di configurazione del sistema e telemetria.

GetServiceByUuid(): Dopo aver stabilito una connessione, il flusso di dati viene inviato tramite il collegamento BLE. Come parametro, la funzione riceve l'UUID del servizio in uso, precedentemente generato, che identificherà in maniera univoca l'*EMG* o il *GYRO* o, ancora, il *SYSTEM*, e il quale verrà inviato all'helper assieme al comando *svcs*. La risposta da parte dell'helper sarà successivamente sottoposta all'intero ciclo di analisi descritto in precedenza: le coppie di *chiave-valore* verranno salvate nel vettore di strutture *tgvl*, quindi, se alla chiave *rsp* corrisponde il valore *find*, vorrà dire che il servizio è presente e dunque si può passare alla ricerca, più in dettaglio, di un riscontro con la parola di codice corrispondente al tag *hstart*. Questa ricerca serve per individuare il *valore* associato a tale tag che rappresenta l'handle iniziale per quel servizio specifico. L'indice indicante il primo riscontro e la lunghezza delle occorrenze, che avrà valore 1, verranno salvati nell'altra struttura, quella i cui campi sono costituiti dai due interi. Solo a questo punto, il valore trovato potrà essere assegnato al campo riservato per l'handle iniziale dell'arrey di strutture *svc*, sfruttando

l'efficienza della funzione *strtol*() che permette la conversione dal tipo stringa al tipo *long int*. Dunque si eseguiranno le stesse operazioni di ricerca per il tag *hend* e, il suo valore verrà assegnato, come per *hstart*, al rispettivo campo dell'array, e sancirà l'altro estremo degli handle delle caratteristiche offerte dal servizio stesso.

```
typedef struct{
  char *uuid;
  uint16_t hndStart;
  uint16_t hndEnd;
}service;
service svc[100];
```

Figura 48 "real_time.c": array di strutture rappresentante il servizio

```
void getservicebyuuid(char* uuid){
  long ret;
  char *ptr;
  fprintf(fpw,"svcs %s\n",uuid);
  fflush(fpw);
  _getR("find"); //"find" è il value associato al tag "resp" e sta per TROVATO!
  int i;
  key=search("hstart");
  if(key.l==1){
     ret=strtol(tgvl[key.index].tval,&ptr,16);
      svc->hndStart=ret;
   printf("getservice: svc->hndStart = %d\n",svc->hndStart);
     }
   key=search("hend");
    if(key.l==1){
      ret=strtol(tgvl[key.index].tval,&ptr,16);
      svc->hndEnd=ret;
     printf("getservice: svc->hndEnd = %d\n",svc->hndEnd);
      }
}
```

Figura 49 "real_time.c": istruzioni della funzione getServicebyuuid()

Inserendo sia nella finzione *parseResp*() che nella funzione *getServiceByUuid*() alcuni *printf*() ed eseguendo quanto visto fin ora, si possono evidenziare i risultati ottenuti dalle funzioni indicate.

La prima funzione, evidenzia in primis il campo *index* dell'array di strutture *tgvl*, quindi mostra, per un elemento del suddetto record, i diversi campi *tag* e i rispettivi *valori*. La seconda funzione espone due dei campi del

servizio, permettendo di sottolineare come l'handle iniziale sia 18 (valore decimale della stringa esadecimale 12) e quello finale sia 24 (valore decimale della stringa esadecimale 18). Ciò significa che le caratteristiche che questo servizio contiene, avranno valori di handle associati a questo intervallo.

```
// output in fase di sperimentazione

parseResp: 0 tgvl[0].tag=*rsp* tgvl[0].tval=*find*
parseResp: 1 tgvl[1].tag=*hstart* tgvl[1].tval=*12*
parseResp: 2 tgvl[2].tag=*hend* tgvl[2].tval=*18*
getservice: svc->hndStart = 18
getservice: svc->hndEnd = 24
```

Figura 50 "real_time.c": output delle funzioni parseResp() e getService()

GetCharacteristics(): Inviando all'helper la parola di codice *char*, necessaria per indicare che ci si sta riferendo alle caratteristiche, e i valori di inizio e fine degli handle del servizio in cui queste si trovano, si riesce ad individuare, grazie alla funzione *_getR*(), la presenza del valore *find* in corrispondenza della ricerca del tag *rsp*. Anche in questo caso, le corrispondenze verranno salvate nei campi di una nuova struttura specifica per le caratteristiche, che comprenderanno quindi l'UUID, l'handle, il campo riservato alle proprietà della caratteristica (che racchiude le operazioni e le procedure possibili su di essa) e infine il valore dell'attributo.

Andando a richiamare la funzione *search*() in successione per i campi appena indicati, utilizzando come parametri i tag previsti dal protocollo di comunicazione, cioè *uuid*, *hnd*, *props*, *vhnd*, verrà ogni volta aggiornato il vettore *key*. Si otterranno, quindi, più occorrenze dello stesso *tag*, il cui *valore* verrà assegnato, lettura dopo lettura, ai successivi elementi che costituiscono l'array *chr*.

```
typedef struct{
  char* uuid;
  uint16_t handle;
  uint16_t properties;
  uint16_t valHandle;
}characteristics;
characteristics chr[100];
```

Figura 51 "real_time.c": array di strutture rappresentanti le caratteristiche

A questo punto è necessario estrapolare gli handle che contengono le informazioni utili. Questi si troveranno in corrispondenza degli UUID caratteristici, e sono riconoscibili perché simili a quelli del servizio in uso.

È in questa fase che torna utile l'ordinamento eseguito in precedenza nella funzione *parseResp*(), possibile attraverso la chiamata alla funzione *qsort*() dove, come precedentemente indicato, l'ordinamento è stato reso stabile dall'impostazione di due chiavi di ricerca, ossia da una principale che riordina alfabeticamente gli handle e una secondaria, che è utilizzata nel caso in cui le chiavi confrontate risultino uguali, e che riordina secondo il valore dell'*index* con cui le suddette chiavi sono state processate dalla funzione *parseResp*. In questo modo, come precedentemente spiegato, non si perde la correlazione tra i vari campi e la caratteristica di cui fanno parte.

Grazie a questo stratagemma si potrà ricercare tra le caratteristiche, quella il cui UUID coincide con quello del servizio e il cui carattere finale risulta essere 1: questo identificherà la prima caratteristica il cui handle è l'oggetto di ricerca e, grazie alla costruzione ben ordinata della struttura delle caratteristiche secondo la quale ad un certo campo UUID corrisponde il suo preciso handle, si è potuto memorizzare quest'ultimo nella variabile *ctrl*. Stesso discorso è stato adottato nel caso si stesse ricercando tra le caratteristiche quella il cui UUID coincidesse con quello del servizio e il cui carattere finale risultasse 2.

Eseguendo i confronti appena visti, si è potuto memorizzare questo secondo handle nella variabile *ctrl*, non sovrascrivendo il valore memorizzato precedentemente ma facendo sì che, a seconda dell'occorrenza di uno dei due UUID, fosse selezionato il percorso specifico di assegnazione.

I due UUID a cui si fa riferimento possono essere resi visibili con lo stesso procedimento visto in precedenza per i servizi, ossia stampando a video i risultati della *parseResp*(), ottenuti senza includere la fase di ordinamento, così da poter cogliere più facilmente l'ordine di acquisizione.

```
// output in fase di sperimentazione pre-ordinamento

parseResp: 0 tgvl[0].tag=*rsp* tgvl[0].tval=*find*
parseResp: 1 tgvl[1].tag=*hnd* tgvl[1].tval=*13*
parseResp: 2 tgvl[2].tag=*props* tgvl[2].tval=*10*
parseResp: 3 tgvl[3].tag=*vhnd* tgvl[3].tval=*14*
parseResp: 4 tgvl[4].tag=*uuid* tgvl[4].tval=*cef92ff1-4179-5abe-9ab4-f41a779bd20f*
parseResp: 5 tgvl[5].tag=*hnd* tgvl[5].tval=*17*
parseResp: 6 tgvl[6].tag=*props* tgvl[6].tval=*8*
parseResp: 7 tgvl[7].tag=*vhnd* tgvl[7].tval=*18*
parseResp: 8 tgvl[8].tag=*uuid* tgvl[8].tval=*cef92ff2-4179-5abe-9ab4-f41a779bd20f*
```

Figura 52 "real_time.c": output della funzione parseResp() per le caratteristiche

Si evidenzia, come anticipato nella parte riguardante i servizi, che i valori degli handle caratteristici sono contenuti all'interno del range dei valori del servizio stesso. Questi saranno trasformati da stringhe esadecimali in cifre decimali nella fase di assegnazione al campo della struttura *chr*.

```
void getCharacteristics(services serv){
  int i,j=0;
  long ret;
  char *ptr;
  if(svc->uuid==0)
    fprintf(fpw,"char %X %X\n", svc->hndStart, svc->hndEnd);
  else
    fprintf(fpw,"char %X %X %s\n", svc->hndStart, svc->hndEnd, svc->uuid);
  fflush(fpw);
  _getR("find");
  srch key=search("hnd");
  for(i=key.index;i<key.index+key.l;i++){
    ret=strtol(tgvl[i].tval,&ptr,16);
    chr[j].handle=ret;
    j++;
  }
  key=search("props");
  j=0;
  for(i=key.index;i<key.index+key.l;i++){
    ret=strtol(tgvl[i].tval,&ptr,16);
    chr[j].properties=ret;
    j++;
 }
  key=search("vhnd");
  j=0;
  for(i=key.index;i<key.index+key.l;i++){
    ret=strtol(tgvl[i].tval,&ptr,16);
    chr[j].valHandle=ret;
    j++;
  }
```

Figura 53 "real_time.c": alcune istruzioni della funzione getCharacteristics()

Quindi, nella variabile *ctrl* verrà salvato il valore dell'handle di uno dei due UUID evidenziati, cioè 14, o meglio la corrispondente cifra 20 per l'occorrenza del primo UUID caratteristico e 18, o meglio la corrispondente cifra 24 per il secondo.

GetDescriptors(): i descrittori delle caratteristiche GATT, vengono utilizzati principalmente per fornire al client i metadati per la Descriptor Declaration, cioè forniscono tutte quelle informazioni aggiuntive sulla caratteristica e sul suo valore.

Il tipo di UUID 0x2902 è quello che corrisponde alla Client Characteristic Configuration Descriptor (CCCD), responsabile dell'attivazione e della

disattivazione delle notifiche per la segnalazione di nuovi dati pronti da trasmettere dal server al client.

```
void getDescriptors(services serv){
char* cccdUUID="00002902-0000-1000-8000-00805f9b34fb";
int i,j=0;
long ret;
char *ptr;
fprintf(fpw,"desc %X %X\n",svc->hndStart, svc->hndEnd);
fflush(fpw);
_getR("desc");
srch key=search("hnd");
for(i=key.index;i<key.index+key.l;i++){
  ret=strtol(tgvl[i].tval,&ptr,16);
  des[j].hnd=ret;
  j++;
}
j=0;
key=search("uuid");
for(i=key.index;i<key.index+key.l;i++){
  int byte=strlen(tgvl[i].tval);
  des[j].uuid=(char*)malloc((byte+1)*sizeof(char));
  strcpy(des[j].uuid,tgvl[i].tval);
  j++;
}
for(i=0;i<key.l;i++){
   if(strcmp(des[i].uuid,cccdUUID)==0){
     cccd[serv]=des[i].hnd;
     printf("getdes: cccd[%d]=%d\n",serv,cccd[serv]);
   }
   else{
     if(strcmp(des[i].uuid,uuid1)==0){
       ctrl[serv]=des[i].hnd;
       printf("getdes: ctrl[%d]=%d\n",serv,ctrl[serv]);
     }
```

Figura 54 "real_time.c": prima parte della funzione getDescriptors

Per questa funzione, il processo di implementazione a livello di codice è simile a quello visto per le caratteristiche ma limitato ai soli due campi che compongono l'elemento, seguendo le indicazioni fornite dal btle.py. Ovviamente ciò si tradurrà in un nuovo record di strutture costituito da due soli campi.

Arrivati a questo livello, siamo stati in grado di compiere il passo fondamentale al fine di abilitare le notifiche e cioè di poter prendere visione, in maniera real-time, dello streaming dei dati.

Perciò è stato necessario memorizzare nella variabile *cccd* il valore dell'handle relativo al UUID 0x2902, il quale è apprezzabile solo in questa fase in cui vengono evidenziati tutti gli UUID dei vari strati di informazione assieme ai rispettivi handle.

```
for(i=0;i<key.l;i++){
   if(strcmp(des[i].uuid,cccdUUID)==0){
     cccd[serv]=des[i].hnd;
     printf("getdes: cccd[%d]=%d\n",serv,cccd[serv]);
   }
```

Figura 55 "real_time.c": seconda parte delle istruzioni

```
// output in fase di sperimentazione

parseResp: 0 tgvl[0].tag=*rsp* tgvl[0].tval=*desc*
parseResp: 1 tgvl[1].tag=*hnd* tgvl[1].tval=*12*
parseResp: 2 tgvl[2].tag=*uuid* tgvl[2].tval=*00002800-0000-1000-8000-00805f9b34fb*
parseResp: 3 tgvl[3].tag=*hnd* tgvl[3].tval=*13*
parseResp: 4 tgvl[4].tag=*uuid* tgvl[4].tval=*00002803-0000-1000-8000-00805f9b34fb*
parseResp: 5 tgvl[5].tag=*hnd* tgvl[5].tval=*14*
parseResp: 6 tgvl[6].tag=*uuid* tgvl[6].tval=*cef92ff1-4179-5abe-9ab4-f41a779bd20f*
parseResp: 7 tgvl[7].tag=*hnd* tgvl[7].tval=*15*
parseResp: 8 tgvl[8].tag=*uuid* tgvl[8].tval=*00002902-0000-1000-8000-00805f9b34fb*
parseResp: 9 tgvl[9].tag=*hnd* tgvl[9].tval=*16*
parseResp: 10 tgvl[10].tag=*uuid* tgvl[10].tval=*00002901-0000-1000-8000-00805f9b34fb*
parseResp: 11 tgvl[11].tag=*hnd* tgvl[11].tval=*17*
parseResp: 12 tgvl[12].tag=*uuid* tgvl[12].tval=*00002803-0000-1000-8000-00805f9b34fb*
parseResp: 13 tgvl[13].tag=*hnd* tgvl[13].tval=*18*
parseResp: 14 tgvl[14].tag=*uuid* tgvl[14].tval=*cef92ff2-4179-5abe-9ab4-f41a779bd20f*
```

Figura 56 "real_time.c": output della funzione parseResp() per i descrittori

Per comprendere meglio il quadro in cui si sta lavorando, si può sottolineare che ogni servizio è caratterizzato dal suo unico UUID e fornirà un certo numero di caratteristiche: la prima, quella il cui UUID corrisponde con 0x2800 (GATT Primary Service Declaration) identifica il servizio in uso e, come notato nella descrizione della *getServiceByUuid*() il suo handle iniziale vale 12; la successiva rappresenta la prima caratteristica del servizio ed è riconoscibile dal suo UUID pari a 0x2803 il cui nome ufficiale fornito dal SIG è Characteristic Declaration Attribute. Quindi, ad ogni occorrenza di questo valore di UUID, corrisponde una nuova caratteristica del servizio stesso, e tra ogni ricorrenza del 0x2803, saranno specificati altri UUID corrispondenti ai descrittori che ciascuna caratteristica presenta: ad

esempio 0x2901 (GATT Characteristic User Descriptor) è un UUID che può essere presente o meno in una caratteristica e che permette una descrizione human reading. Oltre a questi, saranno presenti gli UUID specifici e personalizzati del servizio in uso, che conterranno i dati utili, quelli che il sensore invia tramite Bluetooth al dispositivo centrale.

Eseguendo progressivamente il codice sui tre servizi, si andrà a selezionare uno specifico UUID per ogni servizio con le proprie caratteristiche e i relativi descrittori.

Dall'analisi dell'output riportato in figura, salta all'occhio che solo la prima caratteristica è fornita del descrittore 0x2902 il cui handle è 15. Questo si traduce nella possibilità di ricevere notifiche solo dall'handle con UUID personalizzato 1, ovvero il 14.

3.4 Abilitazione alla ricezione delle notifiche

Gli ultimi step dell'applicazione prevedono i passi fondamentali di scrittura della richiesta di abilitazione, da parte del client, sulle caratteristiche del server.

La funzione *writeChar*() richiamata dal *main(),* permette di comunicare all'helper che si intende eseguire una richiesta di scrittura. Una volta ricevuta tale richiesta per un dato attributo, lo stack del protocollo del server controlla i permessi dell'attributo e, se questo è consentito in scrittura, memorizza il valore da scrivere ed esegue qualsiasi elaborazione specifica sulla caratteristica corrispondente all'handle memorizzato nella variabile *ctrl:* l'handle di controllo ricopre, in quest'ultima parte finale, un ruolo fondamentale insieme all'handle della caratteristica del CCCD.

```
login("CONNECT\n");
sensor(serv);
char *comm="CC08060A0000";
writeChar(EMG,comm);
login("ENABLE");
char *enab="0100";
writeEnable(EMG,enab);
```

Figura 57 "real_time.c": alcune istruzioni della funzione main()

Osservando la chiamata alla funzione *writeChar*(), i parametri attuali sono costituiti sia dall'elemento dell'enumerazione rappresentante il servizio in uso che da una stringa di comando costituita da sei byte nella loro rappresentazione esadecimale e che specificheranno allo stack del server, come controllare i dati: infatti, essendo il servizio di tipo personalizzato e non standard, questo passaggio è necessario per configurare i canali elettromiografici da utilizzare. Ad esempio, il secondo byte 08 specifica la frequenza di campionamento utilizzata dal convertitore analogico-digitale di cui il sensore è munito. Se l'operazione va a buon fine, si abilita il sensore e a schermo apparirà la scritta "ENABLE" ad indicare che i dati possono essere processati. Dunque, per abilitare la ricezione Bluetooth di tali dati, occorre andare a modificare il valore 0000 presente in corrispondenza dell'handle 15, attraverso le istruzioni contenute nella funzione *writeEnable*(). Quest'ultima, infatti, permette l'invio all'helper dei comandi: *wr* per indicare l'operazione di Write Request; *cccd* cioè l'handle del CCCD; la stringa *enab* inizializzata nel *main()* con il valore standard fornito dal SIG per abilitare la ricezione di *ntfy*, ovvero delle notifiche che si stanno cercando.

```
void writeEnable(services serv,char *enab){
fprintf(fpw,"wr %X %s\n", cccd[serv], enab);  //write request
fflush(fpw);
_getR("wr");
}
```

Figura 58 "real_time.c": definizione della funzione writeEnable()

Le notifiche sono pacchetti che includono l'handle del valore della caratteristica contenente i dati utili (come anticipato e verificabile in figura, quella con l'handle 14) insieme al suo valore corrente. Per prenderne visione, sarà infine necessario leggere le risposte fornite dall'bluepy-helper a questi ultimi due comandi impartiti.

```
rsp=$ntfyhnd=h14d=b003F44A63F87FD3FD23A3FE4D03FDAB4407A5314
rsp=$ntfyhnd=h14d=b014238B743C5D243B0EC42588340BC793EBE1B0A
rsp=$ntfyhnd=h14d=b023CE4603C6F8F3DB3583EC9773F46803F882C00
rsp=$ntfyhnd=h14d=b033FCF4F3FE1DA3FDA794078D042350843C62100
rsp=$ntfyhnd=h14d=b0443B67242601A40C4DE3EC5443CE9FC3C7394
```

Figura 59 "real_time.c": output della riga di risposta letta dall'helper

La funzione *getNotifications*() è adibita, appunto, alla lettura delle varie righe di risposta dell'helper, alla loro analisi attraverso la funzione già esaminata *parseResp*() e alla associazione di ogni stringa di notifica col preciso istante in cui è mandata in output che, grazie alla sincronizzazione dei clock del server e del client, coinciderà con quella di campionamento.

```
void getNotifications(services serv){
  time_t now;
  size_t size=0, n;
  char *line;
  while((n=getline(&line,&size,fpr))!=-1){
    len_tgvl=parseResp(line);
    key=search("d");
    now=time(NULL);
    for(int i=key.index;i<key.index+key.l;i++){
      if(serv==0)
      printf("EMG %s >  %s\n",asctime(localtime(&now)),tgvl[i].tval);
      else{
        if(serv==1)
          printf("GYRO %s >  %s\n",asctime(localtime(&now)),tgvl[i].tval);
        else
          printf("SYSTEM %s >  %s\n",asctime(localtime(&now)),tgvl[i].tval);
      }
    }
    continue;
  }
}
```

Figura 60 "real_time.c": definizione della funzione getNotifications()

Il *main()*, rimane dunque in ascolto delle notifiche per poi terminare il programma *real_time.c* con le chiamate alle funzioni che disabilitano il sensore e disconnettono il collegamento BLE tra i dispositivi nonché chiudono le pipes attraverso i rispettivi file pointers, sia in scrittura che in lettura.

```
login("DISCONNECT");
disconnect();
```

Figura 61) "real_time.c": alcune istruzioni della funzione main()

```
void disconnect(){
fprintf(fpw,"disc\n");
fflush(fpw);
_getR("stat");
fprintf(fpw,"quite\n");
fflush(fpw);
fclose(fpw);
fclose(fpr);
}
```

Figura 62 "real_time.c": definizione della funzione disconnect()

Dunque, come ultima azione, una volta accertatisi del corretto funzionamento del codice *real_time.c*, sono stati eliminati tutti i *printf*()

inseriti in fase di sviluppo come strumento di debugging, lasciando esclusivamente quelli funzionali alla visualizzazione, in output, degli handle iniziale e finale del servizio in uso, il *printf*() relativo all'handle delle variabili *cccd* e *ctrl* (questi due *printf*() in realtà dovrebbero essere omessi in un funzionamento reale, ma in questa sede verranno lasciati per sottolineare il corretto riscontro dei valori) e, quindi, il *printf*() della funzione *getNotifications*(), al quale viene indicato di stampare il *valore* corrispondente a tutte le occorrenze del tag *d* memorizzate nell'array di strutture *tgvl->tval* e conteggiate dall'array key.l.

In definitiva, eseguendo da terminale il programma, si otterrà il risultato desiderato.

```
Tue Dec  1 12:15:18 2020
 CONNECT

getservice: svc->hndStart = 18
getservice: svc->hndEnd = 24
getdes: ctrl[0]=20
getdes: cccd[0]=21
getdes: ctrl[0]=24

Tue Dec  1 12:15:20 2020
 ENABLE

EMG Tue Dec  1 12:15:20 2020
 >  003D18A83C78B23D812E3C4E2F3CE4A83E447314

EMG Tue Dec  1 12:15:20 2020
 >  0A412F35430B4143F74643260B41AEC43FE012

EMG Tue Dec  1 12:15:20 2020
 >  173FE0E540011A412B7C4308AC43F9CD432C0E

EMG Tue Dec  1 12:15:21 2020
 >  243FA8F43FE7783FE0483FFBCF4121D1430108

EMG Tue Dec  1 12:15:21 2020
 >  313F0C023F65D13FA8FC3FE6FB3FE0EF3FFBAF

EMG Tue Dec  1 12:15:21 2020
 >  3E3CDBB93E39643F097C3F64053FA75B3FE52D

EMG Tue Dec  1 12:15:21 2020
 >  4B3DF79D3C712B3CD0BF3E2EF33F03A13F61F8

EMG Tue Dec  1 12:15:21 2020
 >  5841D3FB4012F23E01EF3C78153CD0993E2F1A

EMG Tue Dec  1 12:15:21 2020
 >  6543FA33434D3E41D9854019533E071B3C77B3

EMG Tue Dec  1 12:15:21 2020
 >  7240F94042DCB343F8884352DA41E288402590

EMG Tue Dec  1 12:15:21 2020
 >  7F3FE46A3FF0F840EF7D42D3A943F688435735

EMG Tue Dec  1 12:15:21 2020
 >  8C3F9F683FE1613FDFA83FEAAC40E5EE42CC2E

EMG Tue Dec  1 12:15:21 2020
 >  993EF57E3F5B8C3F9D8C3FDF5F3FDF543FE941

EMG Tue Dec  1 12:15:21 2020
 >  A63CAD793E0D493EF56D3F5BAE3F9CBA3FE117

EMG Tue Dec  1 12:15:22 2020
 >  B33E3CD33C8AFD3CA9183E07C53EF31B3F5BEB

EMG Tue Dec  1 12:15:22 2020
 >  C0420E434062B63E46433C8A263C9FBB3E0189
```

Figura 63 "real_time.c": output finale con in evidenza l'evento di connessione, il range degli handle del servizio EMG, i valori dei descrittori, l'evento di abilitazione e i dati utili del servizio EMG al passare dei secondi

CAPITOLO 4

CONCLUSIONI

L'obiettivo fissato, che consisteva nel togliere lo strato software del codice Python e di sostituirlo con un programma in linguaggio C in grado di ricevere tutti i dati del sensore, by-passando la scrittura su file di tali dati, è stato raggiunto. Rimuovendo il doppio passaggio tra i linguaggi di programmazione che era presente nell'applicazione prima del precedente lavoro, siamo stati in grado di aumentare l'efficienza, la repentinità di risposta nonché di evitare l'utilizzo della memoria di massa. Il salto dal linguaggio C e C++ compiuto verso un'acquisizione operata dal linguaggio Python e seguito da un ulteriore salto per tornare nuovamente ad altri sorgenti con estensione .c o .cpp, è stato evitato garantendo una maggiore stabilità dettata da un minor numero di strati software. Il Python, che come sottolineato è un linguaggio interpretato e il cui funzionamento è subordinato ad un programma scritto in C, è stato escluso attraverso la scrittura del programma real_time.c, precedentemente descritto passo dopo passo. Il risultato finale è che il nuovo codice real_time.c può essere integrato con la restante applicazione che regola il funzionamento del sensore. I dati utili rilevati da quest'ultimo, quindi, verranno radiotrasmessi al dispositivo client attraverso il canale BLE che, come visto, non può prescindere dal bluepy-helper.c, in quanto l'helper è in grado di multiplexare gli ingressi provenienti sia dal canale Bluetooth LE che quelli impartiti dal nostro programma, al fine di permettere una fase di ricezione istantanea. Dunque, il real_time.c, per come è stato editato, permette di visualizzare a video, senza passaggi intermedi, i pacchetti di notifica inviati dal sensore.

SITOGRAFIA

1. https://www.mdpi.com/2079-9292/9/6/934/htm
2. https://jlk.fjfi.cvut.cz/arch/manpages/man/python-bluepy.1.en#THE_PERIPHERAL_CLASS
3. https://docs.python.it/html/lib/lib.html
4. http://ianharvey.github.io/bluepy-doc/
5. http://www.punto-bit.com/cgi-bin/main.pl?fileART=14&la=it
6. https://learn.adafruit.com/reverse-engineering-a-bluetooth-low-energy-light-bulb/control-with-bluez
7. https://www.opensourceforu.com/2015/06/linux-without-wires-the-basics-of-bluetooth/#:~:text=The%20host%20consists%20of%20the,such%20as%20APIs%20and%20profiles.
8. https://people.csail.mit.edu/albert/bluez-intro/index.html
9. http://joost.damad.be/2013/08/experiments-with-bluetooth-low-energy.html
10. https://vitolavecchia.altervista.org/caratteristiche-tecniche-fondamentali-bluetooth-low-energy/
11. https://developer.gnome.org/glib/stable/glib-IO-Channels.html#g-io-channel-unix-new
12. http://www.science.unitn.it/~fiorella/guidac/txt/guidac.txt
13. http://lia.deis.unibo.it/Courses/sola0506-info/lucidi/3-ProcessiUnix(2x).pdf
14. http://ninuzzo.freehostia.com/ant/it/teoria/so/pipe.html
15. https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html#:~:text=Characteristic%20Descriptors,after%20the%20characteristic%20value%20attribute.

16. http://magistri.altervista.org/SISTEMI/terza/Guidaallinguaggioc.pdf
17. http://lia.deis.unibo.it/Courses/SOA0405/lucidi/7u.comunicazioneUnix4.pdf
18. https://www.st.com/resource/en/programming_manual/dm00698052-bluetooth-le-stack-v3x-programming-guidelines-stmicroelectronics.pdf