POLITECNICO DI TORINO

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA

Tesi di Laurea Magistrale

Studio, realizzazione e validazione di un sistema partecipativo di monitoraggio ambientale

Relatori

Prof. Filippo GANDINO

Candidato

Cosmin Daniel SOLOMON

Marzo 2020

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Argomento della tesi

Nell'ambito di "smart city" e "smart buildings", diventa sempre più importante raccogliere efficacemente dati ambientali (e.g., di inquinamento, rumore, atmosferici), in maniera distribuita e pervasiva. Esistono diversi approcci alternativi per la trasmissione dei dati caratterizzati da diversi vantaggi e svantaggi.
Fra le possibili soluzioni vi è quella di utilizzare una rete di dispositivi di monitoraggio, dotati di accesso a Internet tramite una connessione Wi-Fi oppure una connessione dati, per mezzo di una scheda SIM. Un limite di questo approccio è l'assenza di reti Wi-Fi in molti punti del territorio. Inoltre per entrambe le tecnologie ci sarebbero dei costi di gestione a carico del gestore del sistema. L'ammontare sarebbe strettamente dipendente dal numero di dispositivi di monitoraggio. Infatti ciascun dispositivo dovrebbe essere dotato di una propria SIM, e tenendo conto del fatto che lo scopo è quello di installare un numero elevato di dispositivi, per ottenere un monitoraggio capillare, il costo dei piani dati e della loro gestione potrebbe essere considerevole.

Una possibilità innovativa, realizzata nel contesto di questa tesi, è rappresentata da un sistema di monitoraggio ambientale partecipativo che implica il coinvolgimento dei cittadini nella fase di trasmissione e raccolta delle rilevazioni ambientali. Il monitoraggio ambientale è il processo che consiste nell'osservare, misurare e immagazzinare misure di grandezze fisiche che caratterizzano un determinato ambiente per un periodo di tempo sufficientemente lungo al fine di poter generare statistiche. Il sistema che implementa tale processo attraverso dispositivi hardware e applicazioni software è chiamato sistema di monitoraggio ambientale [1].
Quando il cittadino è parte attiva di tale sistema, condividendo i propri strumenti hardware e software per la raccolta di dati ambientali, il sistema di monitoraggio è

di tipo partecipativo [2].

Il sistema partecipativo di raccolta e trasmissione, oggetto di questa tesi, è costituito da tre componenti principali:

- il dispositivo di monitoraggio, in grado di raccogliere dei dati ambientali, pre-elaborarli, salvarli e trasmetterli via Bluetooth;
- il componente centrale, ovvero l'applicazione per smartphone Android, in grado di rilevare dispositivi nel proprio range di comunicazione, di raccogliere i dati e di inoltrarli al terzo componente;
- un server remoto in grado di raccogliere i dati trasmessi dagli smartphone Android e di memorizzarli, organizzarli e visualizzarli.

1.2 Sviluppo della tesi

Partendo dai requisiti della proposta di tesi, il primo passo è stato lo studio preliminare della fattibilità del sistema utilizzando i dispositivi di monitoraggio già disponibili, i dispositivi Android e la tecnologia Bluetooth per la fase di scambio dati. Questo studio preliminare mi ha portato poi ad approfondire i dettagli tecnici e a ricercare i meccanismi possibili per consentire la comunicazione sfruttando la tecnologia Bluetooth tenendo conto delle opportunità offerte e dei limiti imposti dai diversi dispositivi e sistemi operativi coinvolti. Una volta definite le procedure necessarie da un punto di vista concettuale sono passato all'implementazione, concentrandomi dapprima sullo scambio di dati generici e aggiungendo poi la parte necessaria per lo scambio dei dati specifici raccolti dal dispositivo di monitoraggio tramite i sensori. Una parte consistente è stata rappresentata dallo studio e sviluppo dell'applicazione Android e nella fase finale una piccola frazione di tempo è stata dedicata anche allo sviluppo della struttura del server remoto per l'immagazzinamento dei dati. Infine vari test sono stati messi in atto sia nelle fasi intermedie che a sistema completato.

1.3 Organizzazione della tesi

La parte seguente di questa tesi è organizzato nel seguente modo:

- il capitolo 2 descrive i dispositivi e i loro sistemi operativi, le tecnologie e le librerie che sono state necessarie per lo sviluppo del sistema
- il capitolo 3 descrive a livello concettuale la struttura del progetto e spiega di fatto come funziona il sistema realizzato

- il capitolo 4 presenta i dettagli implementativi del sistema, riportando porzioni di codice sorgente con relativi commenti dei punti fondamentali per comprenderne gli effetti
- il capitolo 5 presenta l'implementazione di un'estensione del sistema, pensata per la raccolta dei dati in movimento, disponendo di un proprio dispositivo di monitoraggio
- il capitolo 6 si concentra sui test effettuati e sulla valutazione del sistema, presentando alcuni risultati
- segue infine una breve conclusione

Capitolo 2

Dispositivi e tecnologie del sistema

2.1 Dispositivo di monitoraggio: Raspberry Pi

Il dispositivo utilizzato per il monitoraggio dei dati ambientali è il Raspberry Pi [3]. Si tratta di un single-board computer di piccole dimensioni e a basso costo in grado di supportare diversi sistemi operativi; in particolare è adatto per sistemi operativi basati su GNU/Linux, come Raspbian(versione derivata dal più famoso Debian). Esistono diversi modelli e per ogni modello ci sono varie versioni hardware. In particolare i modelli più recenti integrano un modulo Bluetooth, necessario per il funzionamento del sistema implementato in questa tesi. Il dispositivo dispone inoltre di ingressi USB, HDMI, Jack audio da 3.5 mm ed eventualmente la porta RJ45, che ne rendono possibile l'utilizzo come se fosse un normale PC. Il Raspberry Pi non dispone di un hard disk o unità a stato solido ma utilizza una scheda di memoria micro SD dalla quale esegue il boot del sistema operativo ed è utilizzata come memoria non volatile.
Il Raspberry Pi dispone di alcuni connettori GPIO che permettono di collegare ad esso vari dispositivi, come per esempio sensori di vario genere. Nello specifico sono stati collegati sensori in grado di rilevare la temperatura, l'umidità e la concentrazione delle polveri sottili nell'aria. Non disponendo di una batteria interna, deve essere alimentato da una fonte esterna come potrebbe essere un caricatore USB del telefono.

Figura 2.1: Raspberry Pi 3 Model B [4]

2.2 Tecnologia Bluetooth

Il Bluetooth è una tecnologia standardizzata di trasmissione dati a corto raggio senza fili. Rappresenta un metodo economico e sicuro per lo scambio di informazioni utilizzando uno specifico range di frequenze radio(da 2400 a 2483.5 MHz), suddiviso in 79 canali(40 nella versione a basso consumo) per ridurre le interferenze, e permette di individuare i dispositivi coperti dal segnale radio presenti in uno spazio limitato. Qualsiasi tipo di dispositivo il cui hardware e software soddisfino le specifiche dello standard Bluetooth è in grado di comunicare con un dispositivo di diversa natura che implementa le stesse specifiche. Pertanto questa tecnologia è stata adottata da tantissimi tipi diversi di dispositivi come telefoni, PC, cuffie e auricolari, mouse, tastiere, stampanti, autoradio e molti altri consentendo lo sviluppo di numerose applicazioni e interazioni senza la necessità di collegamenti cablati[5].

A partire dal 1999, anno in cui è stata rilasciata la prima versione dello standard

Bluetooth, l'organizzazione Bluetooth Special Interest Group (SIG) ha continuato lo sviluppo della tecnologia rilasciando Bluetooth 2.0 nel 2004 seguito dalla versione 3.0 nel 2009, che ha permesso velocità di trasferimento molto più elevate. Molte innovazioni sono state però portate con la versione 4.0 nel 2010 in quanto includeva le specifiche della nuova versione radio low energy, che ha contribuito a una sempre maggiore diffusione e adozione della tecnologia Bluetooth. Sono stati poi rilasciati nuovi aggiornamenti con nuove funzionalità per il mondo Internet of Things con le versioni 4.1 del 2013 e 4.2 del 2014, che ha anche migliorato aspetti come la velocità e la privacy. Infine nel 2016 è stata rilasciata l'ultima specifica(Bluetooth 5) che ha portato un range quadriplicato, velocità raddoppiata e un notevole incremento della capacità di broadcasting dei dati [6].

Il range del Bluetooth, inteso come la massima distanza possibile tra due dispositivi affinché questi siano in grado di comunicare, dipende da molti fattori:

- specifiche dello strato fisico della tecnologia wireless
- potenza di trasmissione
- guadagno dell'antenna di trasmissione e ricezione
- ambiente(esterno, casa/ufficio)

In base a questi fattori, il range varia da qualche metro a qualche decina o centinaia di metri [7].

Come emerge dalle righe precedenti esistono due diverse versioni radio: Bluetooth "classico" e Bluetooth Low Energy(BLE). Il Bluetooth classico, noto anche come Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR), è particolarmente indicato per trasferimenti continui di dati con una velocità di trasmissione compresa tra 1 Mb/s e 3 Mb/s. Tutte le versioni rilasciate fino alla 4.0 esclusa, appartenevano solo a questa categoria. Per coprire però nuove esigenze tecnologiche, è stata rilasciata la specifica Bluetooth Low Energy(presentato inizialmente come Bluetooth Smart); quest'ultima ha un ridotto consumo energetico ma anche una velocità di trasferimento inferiore, compresa tra 125 Kb/s e 2 Mb/s [8].
È importante sottolineare che la specifica Low Energy non è semplicemente una versione a basso consumo del Bluetooth classico ma è una nuova specifica progettata per un insieme diverso di applicazioni. Le due specifiche hanno infatti caratteristiche differenti. Il Bluetooth classico è utilizzato per streaming audio o altro genere di trasferimenti che richiedono un throughput elevato, anche se richiede più tempo per stabilire la connessione rispetto alla versione BLE. Il Bluetooth Low Energy supporta invece connessioni e disconnessioni rapide per inviare piccoli pacchetti di dato. Quindi risulta particolarmente utile per quelle applicazioni in

cui è necessario inviare saltuariamente piccole quantità di dati mentre per buona parte del tempo non vi è alcun scambio di dati tra le parti interessate. Quando ciò avviene l'antenna può lavorare in modalità a basso consumo(sleep mode) e questo permette di risparmiare molta energia. Quindi il Bluetooth Low Energy è adatto per scambiare informazioni di stato mentre il Bluetooth classico è adeguato per streaming di contenuti multimediali o dati grezzi [9]. Nel contesto di questa tesi entrambe le specifiche sono state utilizzate. Il Bluetooth Low Energy è alla base dei beacon Bluetooth descritti in una sezione successiva ed è utilizzato dal dispositivo di monitoraggio(Raspberry Pi) per segnalare la propria presenza mentre il Bluetooth classico è stato utilizzato per inviare i dati raccolti dal Raspberry Pi al dispositivo Android.

Nell'ambito dei telefoni cellulari, il Bluetooth si è diffuso fin dalla sua prima versione. Infatti nel 2000 è stato presentato il primo telefono dotato di Bluetooth, rivisto e messo in commercio l'anno successivo. La prima adozione della versione Bluetooth Low Energy è invece avvenuta con lo smartphone iPhone 4S di Apple nel 2011 mentre Google ha iniziato il supporto nativo BLE in Android nel 2013 a partire dalla versione 4.3 del sistema operativo [6]. Attualmente, come suggerito dal sito ufficiale [10] basandosi sui dati di ABI Research, il 100% dei nuovi telefoni e tablet sono dotati della tecnologia Bluetooth.

2.3 Bluetooth RFCOMM

Come altri standard wireless, anche lo standard Bluetooth è suddiviso in diversi strati protocollari, ognuno dei quali è responsabile di coprire una parte della comunicazione. Le specifiche definite dall'organizzazione Bluetooth SIG coprono la parte fisica e data-link dello stack protocollare e sono normalmente integrate direttamente nel modulo radio Bluetooth del dispositivo. Sono stati invece definiti altri protocolli per gestire la connessione e il canale logico della comunicazione end-to-end a pacchetti. Le loro implementazioni sono in genere integrate nei sistemi operativi oppure in moduli software installabili [11]. Di questa categoria sono di particolare rilevanza i protocolli L2CAP ed RFCOMM.
RFCOMM(Radio frequency communication) è un protocollo di trasporto a flusso di dati(concettualmente simile a TCP nel mondo IP) pensato per trasferimenti affidabili. La specifica Bluetooth RFCOMM emula una comunicazione seriale RS-232 appoggiandosi al protocollo L2CAP. L'architettura prevista da RFCOMM è di tipo client - server, quindi uno dei due estremi della comunicazione dovrà fungere da server e attendere le connessioni mentre il client dovrà conoscere l'indirizzo fisico Bluetooth del server e iniziare la connessione. RFCOMM dispone di un meccanismo per il controllo di flusso che consente di evitare o ridurre la quantità di

pacchetti da ritrasmettere. Inoltre garantisce l'arrivo in ordine e senza duplicati dei pacchetti trasmessi sfruttando i servizi messi a disposizione dal protocollo L2CAP sottostante, imponendo la creazione di canali di massima affidabilità [12].

2.4 Beacon BLE

Quando si parla di beacon ci si riferisce normalmente a un dispositivo hardware wireless di piccole dimensioni alimentato a batteria e a basso consumo in grado di trasmettere messaggi di piccole dimensioni utilizzando il protocollo Bluetooth Low Energy. In genere viene utilizzato per comunicare informazioni ai dispositivi presenti nei suoi dintorni. Il tipo di informazioni trasmesse sono spesso collegamenti(URL) a messaggi promozionali(Proximity Marketing), dati ambientali come la pressione dell'aria, la temperatura o l'umidità di un ambiente interno ma chiaramente esistono beacon di altre svariate tipologie. Dal punto di vista software i beacon sono dei messaggi inviati periodicamente in broadcast e quindi qualsiasi dispositivo in ascolto è in grado di riceverli [13].
Vari protocolli sono stati definiti per implementare il formato dei beacon BLE ognuno dei quali dà la possibilità di inoltrare messaggi con uno specifico formato e di specificarne il contenuto arbitrario. I principali sono iBeacon, Altbeacon, Eddystone, e URI Beacon. Quello utilizzato in questa tesi è il protocollo Eddystone ideato da Google e descritto nella sezione successiva.
I dispositivi Raspberry Pi che dispongono di un modulo Bluetooth integrato o aggiunto sono in grado di comportarsi come dei beacon hardware e quindi di comunicare in broadcast messaggi di uno specifico formato.

2.5 Eddystone

Nel 2015 Google ha rilasciato Eddystone [14], un protocollo che definisce il formato per i messaggi beacon BLE. Come emerge dalle specifiche ufficiali, "Eddystone è completamente conforme alle specifiche del core Bluetooth ed è semplice da implementare su una vasta gamma di dispositivi BLE esistenti" [15]. Inoltre risulta essere facile da utilizzare sia con i dispositivi Android che iOS in quanto è ben integrato con le rispettive API che consentono di rilevare i beacon. Ci sono quattro formati possibili:

- Eddystone-UID
- Eddystone-URL
- Eddystone-TLM
- Eddystone-EID

Il formato Eddystone-UID è pensato per contenere un identificativo univoco(beacon ID) di 16 byte, suddiviso logicamente in una porzione da 10 byte per il namespace e una da 6 byte per l'istanza. Normalmente si può sfruttare l'identificativo inoltrato in broascast dal dispositivo BLE per risalire alle informazioni correlate in un server remoto. In questo progetto non si è sfruttata tale peculiarità ma i 16 byte sono stati considerati come un unico blocco che semplicemente identificano un qualsiasi dispositivo Raspberry Pi che è parte del sistema di monitoraggio.

Il frame Eddystone-URL serve invece a comunicare un URL in un formato compresso ai dispositivi vicini che potranno così accedere a contenuti web che sono legati al ambiente circostante; rappresenta quindi il formato utilizzato per il web fisico.

Il frame Eddystone-TLM ha l'obiettivo di consentire al dispositivo beacon stesso di comunicare informazioni sul proprio stato, come il voltaggio o il livello della batteria o il numero di pacchetti trasmessi.

Infine il tipo Eddystone-EID comunica un identificatore effimero cifrato(che cambia in maniera periodica) e quindi potrà essere decifrato remotamente solo dal servizio con cui è stato registrato.

2.6 Android

2.6.1 Sistema operativo

Android è un sistema operativo sviluppato da Google insieme ad altre compagnie(Open Handset Alliance) e rilasciato nella sua prima versione nel 2008. È stato pensato e progettato fin da subito per dispositivi mobili, in particolare smartphone e tablet anche se oggi esistono versioni con interfacce specifiche per Smart TV, sistemi di infotainment delle automobili e altri accessori tecnologici. Android è un sistema operativo open source basato sul kernel Linux anche se alcune parti sono state rimosse, sostituite o aggiunte per andare incontro ad alcune esigenze dei dispositivi mobili come il consumo energetico e le risorse hardware limitate che hanno caratterizzato soprattutto le prime generazioni di smartphone. Il codice sorgente è aperto a tutti ma alcuni servizi e app di Google(come il Play Store) sono proprietari e il loro codice non è a sorgente aperta. Nonostante non sia obbligatorio preinstallare le app di Google sul proprio sistema, molti produttori le integrano in quanto contribuiscono a migliorarne l'esperienza utente. Al di sopra del kernel è stato creato lo stack software di Android ricco di librerie, che permettono agli sviluppatori di creare applicazioni di ogni genere. Il modo più comune per distribuire le applicazioni sono gli app store. Il più diffuso è Google

Play Store che mette a disposizione più di un milione di app [16]. A novembre 2019 Android risulta essere il sistema operativo più diffuso al mondo con una quota di mercato di circa 76% seguito da iOS con quasi il 23% [17].

Nel corso degli anni il sistema operativo è stato evoluto e sono state rilasciate nuove versioni del sistema stesso e delle API fornite. Ciascuna versione ha introdotto importanti funzionalità e potenzialità per gli utenti e per gli sviluppatori oppure hanno reso più efficienti determinati meccanismi già esistenti. Il framework API messo a disposizione dalla piattaforma Android consente di interagire con il sistema Android sottostante. Il framework API è costituito da un insieme di packages e classi, elementi e attributi XML per il manifest file(descritto più avanti), permessi e autorizzazioni che l'applicazione può richiedere al sistema o all'utente e un insieme di messaggi(chiamati intents) che possono essere scambiati tra applicazioni diverse o tra componenti della stessa applicazione. Quando il framework viene aggiornato, le nuove API rimangono compatibili con le precedenti. Infatti i cambiamenti consistono spesso in aggiunta di nuovi metodi e le parti che vengono aggiornate portano alla deprecazione di quelle precedenti che però continuano ad essere disponibili per le vecchie applicazioni. Capita invece raramente che vengano modificate o rimosse le parti già esistenti se non in quei casi quando è necessario rendere più robusto e sicuro il sistema [18]. Di seguito sono elencate le principali versioni rilasciate fin dalla nascita con il relativo nome in codice e il corrispondente livello di API. Ciascuna versione della piattaforma Android supporta uno specifico livello di API anche se ciò include chiaramente il supporto per tutti i livelli precedenti.

Versione(nome)	**Livello API**	**Anno di rilascio**
1.0(/)	1	2008
2.0(Eclair)	5	2009
3.0(Honeycomb)	11	2011
4.0(Ice Cream Sandwich)	14	2011
5.0(Lollipop)	21	2014
6.0(Marshmallow)	23	2015
7.0(Nougat)	24	2016
8.0(Oreo)	26	2017
9.0(Pie)	28	2018

Table 2.1: Principali versioni del sistema operativo Android

Per indicare che una funzionalità è disponibile per un insieme di versioni successive a partire da una specifica, si usa aggiungere il segno '+' dopo il numero; ad esempio con Android 5.0+ si indicano la versione 5.0 e tutte quelle successive. È importante sottolineare che quando un'applicazione viene creata dagli sviluppatori, si deve specificare la versione minima che il sistema operativo deve avere per poterla installare ed eseguire. Nel contesto di questa tesi la versione minima è stata impostata ad Android 5.0(API 21). Impostando la versione minima a tale valore significa che l'applicazione potrà essere eseguita su almeno 89,3% dei dispositivi a livello mondiale(secondo le stime fornite da Google in seguito ai dati raccolti nella prima settimana di maggio 2019 [19]). Nel seguente grafico si può visualizzare la distribuzione delle varie versioni del sistema operativo nei dispositivi Android:

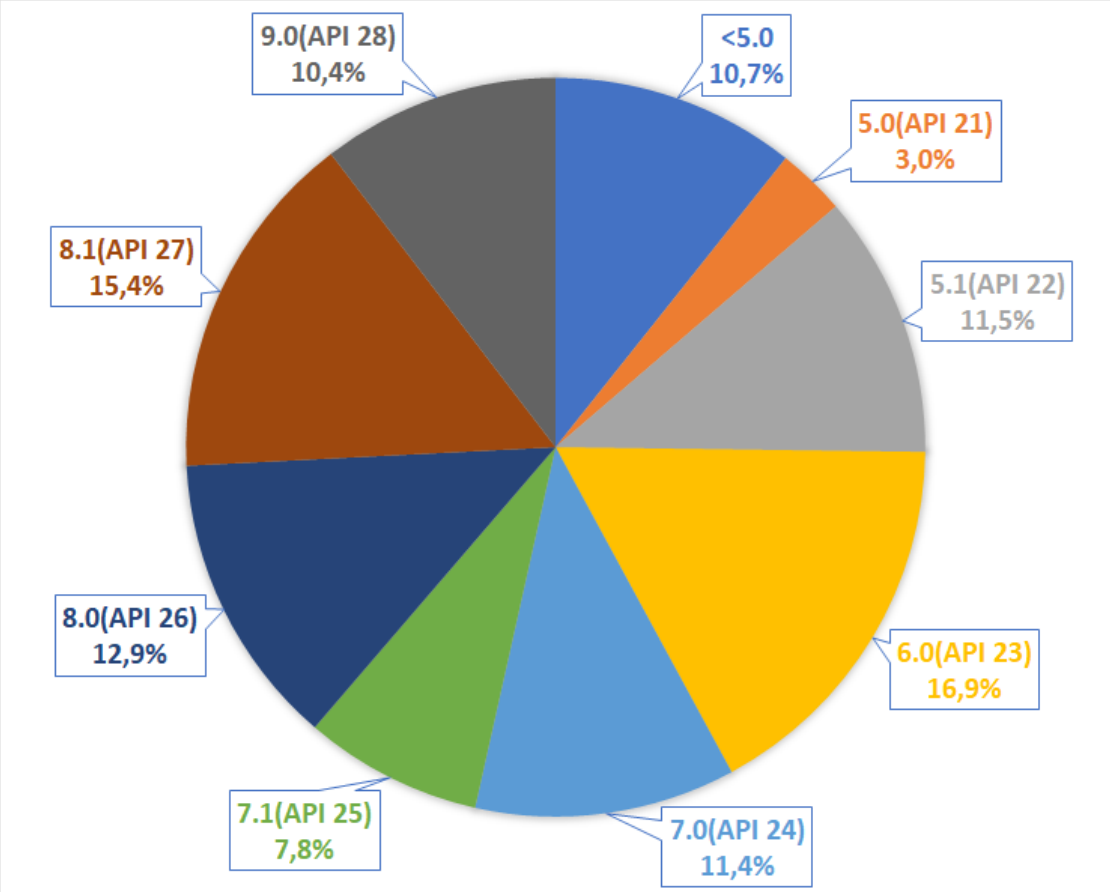

Figura 2.2: Distribuzione versioni Android a maggio 2019 [19]

2.6.2 Sicurezza e permessi

Nel momento in cui un'app Android viene installata viene generato un nuovo utente a cui l'app viene associata. In questo modo, trattandosi di un ambiente UNIX-style, un'applicazione non può accedere ai file di un'altra in quanto in base ai permessi concessi dal filesystem, un utente non può leggere o modificare i file di un'applicazione di un altro utente. Per garantire la separazione in fase di esecuzione, un'applicazione Android viene eseguita all'interno di una macchina virtuale(ART - Android RunTime). Per ciascuna applicazione in esecuzione viene creato un processo separato con una propria istanza di tale macchina virtuale.
Ci sono alcune regole che devono essere rispettate per evitare che il sistema operativo uccida il processo all'interno del quale l'applicazione viene eseguita. Prima di tutto ciascuna applicazione deve dichiarare le risorse hardware che utilizzerà, come il Bluetooth, la fotocamera o altri sensori del dispositivo. Dal punto di vista della sicurezza per avere accesso ad alcune risorse o ai dati privati dell'utente come potrebbero essere la posizione, i contatti o altri dati personali è necessario chiedere i permessi all'utente stesso attraverso opportune finestre di dialogo. Tali permessi sono chiamati "pericolosi" e si distinguono dai permessi "normali" che invece non richiedono l'approvazione dell'utente ma vengono concessi in automatico dal sistema in quanto non mettono a rischio la privacy dell'utente.

2.6.3 Manifest file

Un'applicazione Android deve includere il manifest file(denominato AndroidManifest.xml) che è un documento in formato XML che contiene informazioni descrittive e può essere visto come una sorta di contratto tra l'applicazione e il sistema Android. È all'interno di questo documento che bisogna dichiarare le risorse hardware richieste dall'app, i permessi di cui si necessita ma anche tutte le componenti Android(Activity, Service, BroadcastReceiver e ContentProvider) che ne fanno parte. Inoltre deve definire il minimo livello di API supportato attraverso il campo "minSdkVersion" ma contiene anche il campo "targetSdkVersion" che indica la massima versione su cui è stata testata e che non necessita di attivare la modalità compatibilità, attivata invece dal sistema quando quest'ultimo è di una versione successiva rispetto a quella target. Le informazioni contenute in questo documento sono utilizzate nella fase di compilazione del progetto ma anche dal sistema operativo e dai market di distribuzione per stabilire quali dispositivi possono installare la relativa applicazione.

2.6.4 Componenti di un'app Android

Un'applicazione può essere costituita da uno o più componenti [20]. Esistono quattro tipi di componenti:

- Activity: è l'unico componente che permette all'utente di interagire con l'applicazione in quanto dispone di un'interfaccia grafica. Lo scopo di questo componente è di consentire all'utente di eseguire uno specifico compito e quindi normalmente un'app è costituita da tante activity, ognuna indipendente dalle altre e dotata della propria interfaccia grafica. L'interazione tra le activity costituisce il flusso logico dell'applicazione. L'interazione dell'utente con l'applicazione può iniziare in differenti punti del flusso e quindi in differenti activity a differenza dei programmi desktop dove il lancio del programma corrisponde con l'esecuzione un metodo "main" predefinito.

- Service(servizio): è utilizzato per eseguire un'operazione di lunga durata in background(come il download di un file o il playback di un file audio). Non permette all'utente di interagire direttamente con esso attraverso un'interfaccia grafica, ma il componente che lo ha lanciato può comunicare con il servizio tramite un'interfaccia client-server.

- ContentProvider: è il componente attraverso il quale un'applicazione espone dei dati alle altre applicazioni all'interno del sistema. Quindi quando un'app vuole condividere dei dati persistenti a cui ha accesso, utilizza un ContentProvider che le applicazioni esterne possono interrogare o modificare, se consentito, specificando gli URI che identificano i dati esposti.

- BroadcastReceiver: rimane in attesa di messaggi che segnalano il verificarsi di determinati eventi e informa l'applicazione, anche quando quest'ultima non è in esecuzione. I messaggi possono essere generati dal sistema operativo(per esempio, cambiamento di stato della rete, boot del sistema completato, livello della batteria basso o altri eventi di sistema) oppure da altre applicazioni(download di un file completato).

Nelle seguenti righe vengono elencati maggiori dettagli relativi al componente Service. In particolare un servizio può essere di diverso tipo [21]:

- Background: una volta fatto avviare da un altro componente, il servizio sarà eseguito in background senza ritornare un risultato al componente chiamante. Inoltre il suo ciclo di vita è indipendente dal ciclo di vita del componente chiamante. Esso continua ad esistere anche se il componente chiamante è stato terminato.

- Bound: permette ai componenti di interagire con il servizio attraverso un'interfaccia client - server. Il suo ciclo di vita è legato a quello dei componenti associati a tale servizio. Nel momento in cui nessun componente sarà più associato, il servizio verrà terminato.

- Foreground: quando è in esecuzione viene notificato all'utente attraverso una notifica e quindi l'utente è informato del fatto che l'applicazione sta eseguendo delle operazioni in background per mezzo di un servizio. Inoltre la notifica può contenere dei comandi che possono consentire all'utente di controllare il funzionamento del servizio(ad esempio controllare il playback di una traccia musicale).

A partire dalla versione Android 8.0 (API level 26) sono stati imposti dei limiti per l'esecuzione dei servizi in background in quanto consumano risorse del sistema impattando in maniera negativa sull'esperienza utente. Per cercare di migliorare l'esperienza utente e allungare la durata della batteria, i servizi di tipo background di un'app vengono arrestati da sistema dopo alcuni minuti dal momento in cui l'applicazione non è più in foreground. I servizi di tipo Bound e Foreground non sono soggetti a questi limiti.
Sempre a partire da Android 8.0 non è più permesso che un'app che è in background lanci un servizio di tipo background ma può solo creare un servizio di tipo foreground con l'obbligo quindi di informare l'utente tramite una notifica quando il servizio è in esecuzione.

2.6.5 Intent

Un intent è una sorta di messaggio asincrono, contenente la descrizione di un'azione da eseguire, che permette di attivare un componente di un'applicazione e rappresenta un modo standard per la comunicazione tra componenti della stessa applicazione o di applicazioni differenti. Gli intents vengono raggruppati in due categorie:

- Espliciti: indicano il componente specifico, normalmente presente all'interno della stessa applicazione, che dovrà essere attivato

- Impliciti: indicano il tipo di componente che dovrà essere attivato, specificando l'azione da eseguire e lasciando al sistema il compito di trovare il componente che sia in grado di gestirla, sfruttando i filtri(intents filters) che i componenti dichiarano nel manifest file

Un esempio di intent esplicito è quello creato da una Activity che vuole far partire un servizio specifico all'interno della stessa applicazione; può farlo invocando il metodo "startService(...)" e passargli un intent(contenente il nome del servizio) che attiverà il componente. Un intent implicito è ad esempio quello che viene generato da una Activity, per richiedere a un'altra applicazione, che gestisce le telefonate, di effettuare una chiamata(al numero passato nell'intent) in seguito a un'azione dell'utente.
L'intent dà la possibilità di passare al componente che verrà lanciato una quantità limitata di dati a patto che siano serializzabili. Gli intents possono anche essere

originati dal sistema operativo in seguito a specifici eventi ed essere consegnati alle applicazioni che hanno richiesto di essere notificate per mezzo del BroadcastReceiver.

2.6.6 Persistenza locale dei dati

Un'applicazione può salvare dei dati in maniera permanente nella propria porzione privata del file system. Una soluzione efficace e leggera fornita dallo stack Android è il database SQLite per salvare localmente dati strutturati relazionali e per eseguire operazioni CRUD in linguaggio SQL.
Per facilitare l'utilizzo del database agli sviluppatori e ridurre le possibilità di commettere errori, Android mette a disposizione Room [22]. Si tratta di uno strato sopra SQLite che gestisce in automatico le operazioni di basso livello per accedere al database permettendo allo sviluppatore di concentrarsi sulla struttura dei dati e sulle operazioni da effettuare su di essi. Room prevede un'architettura basata su tre elementi:

- Room Database: è il componente principale che può essere visto come il punto di accesso al database sottostante
- Data Access Objects(DAO): è l'oggetto che fornisce i metodi per accedere al database
- Entity: rappresenta una qualsiasi tabella del database

Nel momento in cui l'applicazione vuole accedere al database sottostante, dovrà creare un'istanza del Room Database, che sarà utilizzata a sua volta per ottenere un'istanza del DAO. Per mezzo del DAO si potranno visualizzare o modificare i dati all'interno del database modellato dalle Entity.
Le operazioni possibili che si possono eseguire sul database corrispondo ai metodi offerti dal DAO. Quindi gli sviluppatori dovranno soltanto definire, seguendo le convenzioni del linguaggio di programmazione utilizzato, il tipo di operazione da effettuare(Insert, Update , Delete oppure Select con relativa query) e il corrispondente codice SQL sarà generato automaticamente in fase di compilazione. Sempre durante questa fase sarà anche verificata la correttezza delle query SQL specificate, evitando quindi di avere errori a run-time. Inoltre è automatica la conversione da query SQL agli oggetti che contengono i dati. Seguendo questo approccio, si riduce la possibilità di errore in quanto lo sviluppatore dovrà scrivere meno codice di basso livello e il codice SQL si dovrà specificare solo nel caso delle query.

2.6.7 WorkManager

WorkManager è una API che consente di definire un compito che sarà dato in gestione al sistema operativo con la garanzia che verrà eseguito(una volta sola o

periodicamente), in maniera asincrona, anche nei casi in cui l'applicazione verrà terminata oppure il dispositivo sarà riavviato [23]. Si ha la possibilità di specificare i vincoli che dovranno essere soddisfatti affinché l'WorkManager possa avviare l'esecuzione del compito. Per esempio si può imporre che un determinato compito venga eseguito quando la connessione alla rete Internet sarà disponibile. Nel caso in cui uno dei vincoli venisse a mancare durante l'esecuzione del compito è possibile richiedere di ritentare l'esecuzione dell'intero compito successivamente(quando i vincoli saranno nuovamente soddisfatti) imponendo anche un ritardo variabile in maniera lineare o esponenziale. Questo significa, che dopo l'interruzione del compito, il tempo che intercorre tra un tentativo e il successivo aumenta in maniera lineare o esponenziale all'aumento del numero di tentativi.
Il compito si definisce implementando una classe che estende "Worker" andando ad aggiungere le istruzioni specifiche che dovranno essere eseguite all'interno del metodo "doWork", che alla fine di tutto ritornerà l'esito. È quindi il codice aggiunto dallo sviluppatore a stabilire se le operazioni richieste sono andate a buon fine o meno, ritornando opportunamente uno dei possibili risultati standard al WorkManager. In base al risultato che riceve, quest'ultimo stabilirà se dovrà essere ritentato in futuro.

2.6.8 Android Beacon Library

Android Beacon Library è una libreria Android open source che consente di rilevare la presenza di beacon BLE in prossimità del dispositivo [24]. Attraverso i meccanismi forniti dalla libreria è possibile ricevere delle notifiche nel momento in cui un beacon diventa visibile oppure quando esce dal campo di visibilità. Per poter utilizzare la libreria è richiesto che il dispositivo sia dotato del chipset Bluetooth Low Energy e abbia una versione di Android uguale o superiore a 4.3. La libreria può essere configurata in maniera tale da rilevare vari formati di beacon tra cui il formato Eddystone, descritto in precedenza.
La libreria fornisce inoltre un meccanismo per attivare in background l'applicazione creata dallo sviluppatore nel momento in cui viene rilevato un beacon con le caratteristiche richieste. Infatti in seguito alla rilevazione, viene generato un intent destinato all'applicazione che quindi sarà attivata e potrà eseguire il codice desiderato. Questa feature è particolarmente utile per consentire il rilevamento dei beacon nel lungo periodo anche sulle versioni di Android 8.0+ in cui non sarebbe possibile avere un servizio sempre attivo in background per effettuare delle scansioni periodiche.
La rilevazione dei beacon è realizzata impostando dei filtri Bluetooth hardware ed eseguendo delle scansioni software periodicamente. La prima volta che un nuovo beacon del tipo desiderato compare nel range del dispositivo, viene rilevato entro pochi secondi grazie ai filtri Bluetooth hardware. Questo tempo si allunga per i

rilevamenti successivi dello stesso tipo nel caso in cui si rimane nelle immediate vicinanze del beacon già rilevato, in quanto i filtri hardware saranno disattivati. Il rilevamento del primo beacon può inoltre essere più lento nel caso in cui i filtri hardware non siano disponibili o non funzionino. Le rilevazioni veloci appena descritte, lavorano in combinazione con il funzionamento di base che consiste nell'eseguire una scansione software una volta ogni N minuti. L'intervallo N tra una scansione e un'altra è configurabile, anche se questo va chiaramente ad influire sulla batteria e quindi non sono consigliati periodi di tempo troppo brevi. Ad esempio per le versioni precedenti ad Android 8.0, il valore di default di N impostato dalla libreria è pari a cinque minuti ed il funzionamento è realizzato per mezzo di un servizio sempre attivo in background. Per Android 8+ si usa lo JobScheduler per eseguire le scansioni periodiche e ciò implica un periodo di tempo di circa quindici minuti tra una scansione e la successiva.
Supponendo quindi che il dispositivo sia in grado di sfruttare entrambi i meccanismi, il tempo necessario per rilevare il primo beacon di uno specifico tipo quando si entra nel suo campo d'azione è di circa cinque secondi. Avendo individuato un beacon per individuare un secondo beacon(proveniente dallo stesso o da un dispositivo hardware diverso) è necessario attendere la scansione software che nel caso peggiore corrisponde a cinque minuti per le versioni 5.0 - 7.x e a quindici minuti(a volte anche qualche minuto in più) per le versioni 8.0+.
La libreria è in grado di rilevare i beacon anche dopo che il dispositivo è stato riavviato oppure dopo che l'applicazione è stata terminata. Il modo in cui questa funzionalità viene implementata dipende dalla versione di Android:

- versioni 4.3 - 7.x: per mezzo dell'AlarmManager viene inviato un intent al servizio per attivarlo. In aggiunta si sfruttano dei BroadcastReceiver per ascoltare e reagire agli eventi che segnalano boot del sistema completato oppure collegamento e scollegamento del dispositivo al caricabatterie.

- versioni 8+: si utilizza lo JobScheduler per eseguire delle scansioni periodiche anche se l'app non è in esecuzione.

2.6.9 Loopj Library

Loopj è una libreria Android per effettuare delle operazioni HTTP sincrone o asincrone e gestire le risposte. È nota anche come "Android Asynchronous Http Client" e fornisce varie funzionalità che semplificano l'interazione con la rete; per esempio riprova ad inviare in automatico le richieste nel caso in cui non siano andate a buon fine a causa di connessioni mobili instabili o altri problemi di rete. Le richieste Http vengono eseguite in automatico in un thread separato per non bloccare il thread principale(UI thread) dell'app. A differenza di altre librerie, come potrebbe essere Volley di Google, che fornisce funzionalità simili, la libreria

Loopj è molto più leggera e aggiunge un overhead di soli 90 kb all'applicazione. Alcune delle altre funzionalità fornite dalla libreria sono la gestione delle ridirezioni circolari e relative oppure il salvataggio persistente dei cookie [25].

2.6.10 IDE: Android Studio

Android Studio è un ambiente di sviluppo inegrato(IDE) multipiattaforma per creare applicazioni per il sistema operativo Android. Esso è basato su IntelliJ IDEA e fornisce un insieme di utili strumenti a supporto degli sviluppatori che contribuiscono a facilitare la creazione e l'evoluzione del progetto e ad aumentare la produttività [26]. Tra le principali caratteristiche ci sono:

- un editor di testo rapido nel fornire suggerimenti per i linguaggi Java, Kotlin e C++
- un emulatore che consente di testare le app senza la necessità di avere un dispositivo fisico
- differenti viste del progetto. Particolarmente utile risulta essere la vista "Android" che permette allo sviluppatore di visualizzare solo i file e le risorse principali dell'applicazione e che corrispondono normalmente agli unici file che vengono modificati durante lo sviluppo
- un sistema integrato per gestire e automatizzare il processo di build tramite l'uso di Gradle che è affiancato da un plugin che aggiunge funzionalità specifiche per Android e che è in grado di generare differenti APK in base alle caratteristiche del dispositivo target, come potrebbe essere la densità dello schermo
- supporto e integrazione per i più usati sistemi di controllo versione come Git, GitHub, CVS, Mercurial, Subversion, e Google Cloud Source Repositories
- una finestra denominata "Logcat" che consente di osservare tutti i messaggi di log del dispositivo connesso e può essere utilizzato anche per il debug delle app

2.6.11 Sviluppo e distribuzione di un'app

Il codice sorgente delle app Android può essere sviluppato utilizzando i linguaggi Kotlin, Java e C++ insieme al kit di sviluppo di Android(SDK). Nel momento in cui si vuole testare o rilasciare l'applicazione, il codice sorgente insieme alle risorse e ai dati vengono compilati e viene prodotto un unico pacchetto compresso con estensione .apk chiamato Android Package(APK). Tale archivio può essere

distribuito agli utenti attraverso i vari app marketplace ma anche attraverso siti web o via email. È infatti sufficiente scaricare l'APK per poter installare la relativa applicazione sul proprio dispositivo [20].

2.7 Server remoto

Il server remoto è il componente del sistema che rimane sempre in attesa di connessioni da parte dei dispositivi mobili, ricevendo da essi i dati da salvare nel database gestito dal server stesso. Di seguito vengono descritte le due principali caratteristiche del server stesso ovvero l'architettura utilizzata e il tipo di database.

2.7.1 Servizio web REST

Un servizio web(web service) è un componente applicativo per sistemi distribuiti basato su protocolli aperti come HTTP e altri standard Web. A differenza di un'applicazione web con la quale normalmente un utente interagisce per mezzo del browser, è accessibile da applicazioni client senza interfaccia grafica con le quali vengono scambiati dati in formato XML o JSON. Per mezzo degli standard Web è quindi possibile far comunicare applicazioni sviluppate in diversi linguaggi di programmazione in esecuzione su sistemi operativi differenti. Uno degli approcci possibili per la creazione di un servizio web è quello di seguire l'insieme delle linee guida di REST.

REST(Representational State Transfer) è un'architettura per sistemi distribuiti client-server di cui la più utilizzata implementazione è basata sul protocollo HTTP. L'interazione tra client e server(o meglio web service) avviene attraverso scambio di messaggi HTTP(richieste e risposte). Ciascun messaggio è indirizzato ad una specifica risorsa. Una risorsa non è altro che un qualsiasi tipo di informazione presente lato server ed è identificata univocamente da un identificatore globale(URI). Quando si interagisce con una risorsa viene trasferito solo un suo stato mentre la risorsa in sé rimane sul server. Un servizio REST definisce diverse operazioni possibili che possono essere eseguite sulle risorse a partire dalle cosiddette operazioni CRUD(Create, Read, Update, Delete) corrispondenti ai principali verbi HTTP(POST, GET, PUT, DELETE). Nelle implementazioni REST basate su HTTP, ogni richiesta eseguita su una risorsa è indipendente dalle altre.
Un servizio web REST espone un'interfaccia costituita da delle URI alle quali sono associate delle risorse su cui si possono svolgere le operazioni consentite dal servizio. I client possono quindi svolgere tali operazioni per mezzo di messaggi HTTP indirizzati alle rispettive URI. Quando il container del servizio riceve la richiesta, invoca automaticamente il metodo corrispondente, scritto nel linguaggio di programmazione utilizzato per lo sviluppo del servizio. Alcuni metodi si aspettano

di ricevere dei dati come parametri. Le informazioni conformi alle specifiche del servizio(normalmente in formato XML o JSON) e contenute nei messaggi HTTP vengono automaticamente convertite nei rispettivi oggetti e passati come parametri ai metodi invocati.
Trattandosi di un ambiente distribuito è opportuno informare il client sull'esito dell'operazione richiesta inviandogli un messaggio a operazione conclusa. I servizi web REST sfruttano i codici di ritorno HTTP standard e quindi per ogni richiesta vengono stabiliti i codici HTTP che saranno ritornati in base alle specifiche applicative.

2.7.2 Deployment di un servizio web

Un servizio web viene depositato e lanciato all'interno di un ambiente di esecuzione chiamato web container, a sua volta ospitato da un server web in grado di gestire le richieste HTTP provenienti da altri elaboratori. Il web container è responsabile del ciclo di vita del servizio, di fare il mapping tra URL e servizio e di autorizzare le richieste. Un esempio di web container che consente di eseguire dei moduli web è Apache Tomcat.

2.7.3 DBMS MySQL

MySQL è un sistema di gestione di basi di dati relazionale(che supporta il linguaggio SQL) di proprietà di Oracle Corporation disponibile però anche con licenza a sorgente aperta(GPL) [27].
È supportato dalla maggioranza dei sistemi operativi e dai più noti linguaggi di programmazione, come C, C++, PHP o Java che è stato quello utilizzato in questo caso.
MySQL è utilizzato da un elevato numero di piccole, medie e grandi aziende in quanto risulta essere molto scalabile, adatto sia per applicazioni con un volume di dati limitato che per quelle con volumi molto elevati. Altre caratteristiche che lo rendono tra i più utilizzati DBMS al mondo sono le prestazioni elevate, la costante disponibilità, la considerevole protezione dei dati e l'ottimo supporto per le transazioni.

Capitolo 3

Progetto del sistema

All'interno di questo capitolo viene descritto come è stato progettato il sistema senza addentrarsi nei particolari implementativi che saranno invece presentati nel capitolo successivo. In particolare ognuna delle seguenti tre sezioni si focalizza sul compito del rispettivo componente e sulla sua interazione con gli altri componenti che costituiscono il sistema.
Nel seguente diagramma viene mostrato il sistema nel suo insieme. Le tre parti del sistema sono il dispositivo di monitoraggio(Raspberry Pi e sensori), il dispositivo Android(normalmente uno smartphone) e il server remoto(web service e DB). Osservando il diagramma da sinistra si nota il Raspberry Pi al quale sono connessi dei sensori che misurano con continuità grandezze ambientali come temperatura, umidità e particolato atmosferico. La comunicazione tra il dispositivo di monitoraggio e il dispositivo Android avviene per mezzo della tecnologia Bluetooth mentre viene usata la rete Internet per il caricamento dei dati sul server remoto.

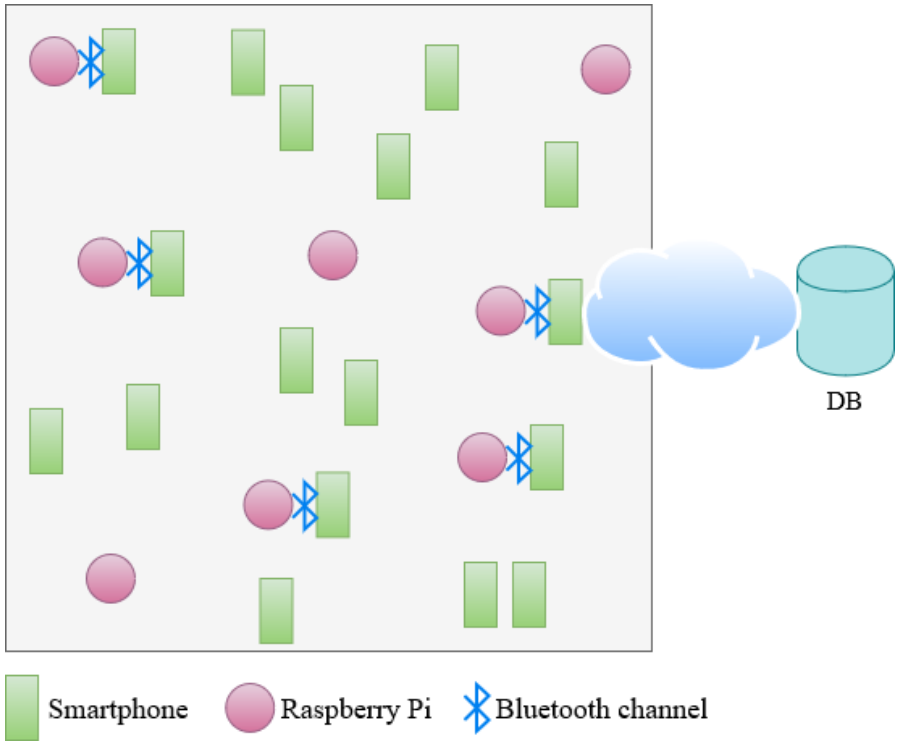

Figura 3.1: Sistema completo di monitoraggio ambientale

3.1 Dispositivo di monitoraggio: Raspberry Pi

Il Raspberry Pi è pensato per trovarsi all'esterno(normalmente in un ambiente cittadino, come potrebbe essere la fermata di un bus) e deve svolgere i seguenti compiti:

- accumulare i dati ambientali dai sensori
- segnalare la propria presenza inviando beacon BLE
- attendere connessioni Bluetooth Rfcomm
- inviare i dati accumulati allo smartphone connesso

3.1.1 Accumulare i dati ambientali dai sensori

Il Raspberry Pi dispone di una serie di connettori GPIO ai quali sono connessi i sensori che misurano alcune grandezze fisiche.
Un sensore(DHT22) è utilizzato per rilevare la temperatura e l'umidità presenti nell'ambiente con una periodicità di circa due secondi.

Quattro sensori(HPMA115S0-XXX Honeywell), tutti dello stesso tipo, servono a percepire la qualità dell'aria misurando il particolato atmosferico(noto anche come polveri sottili). In particolare viene misurata la concentrazione(in µg/m^3) delle particelle PM10 (diametro inferiore a 10µm) e PM2.5 (diametro inferiore a 2.5µm) con una periodicità di circa un secondo.
All'avvio del dispositivo di monitoraggio viene lanciato un primo programma che ha il compito di leggere i dati dai sensori appena descritti, e di salvarli all'interno di un file dal quale potranno essere prelevati successivamente da un secondo programma(anch'esso lanciato all'avvio del sistema) che avrà il compito di inviarli ad un dispositivo Android.

3.1.2 Segnalare la propria presenza inviando beacon BLE

Il Raspberry Pi deve segnalare continuamente la propria presenza per far sì che qualsiasi dispositivo Android eventualmente presente nelle immediate vicinanze possa accorgersi della sua presenza e quindi avviare il processo di connessione.
La segnalazione della propria presenza viene realizzata attraverso un tool di sistema che consente di avviare al lancio del sistema operativo l'invio periodico in broadcast di beacon BLE in formato Eddystone. Per la precisione viene utilizzato il formato Eddystone-UID. Il valore dell'identificatore univoco(UID) è stato deciso a priori e ha il solo scopo di evitare che l'applicazione Android reagisca di fronte a qualsiasi beacon; in questo modo l'applicazione Android avvierà il processo di connessione con il Raspberry Pi solo nel caso in cui abbia rilevato un beacon di tipo Eddystone-UID contenente la sequenza univoca stabilita a priori.

3.1.3 Attendere connessioni Bluetooth Rfcomm

Nel momento in cui un dispositivo Android rileva un beacon BLE, come descritto nella sottosezione precedente, tenta di avviare una connessione Bluetooth RFCOMM con il dispositivo di monitoraggio. Quest'ultimo dovrà quindi rimanere in attesa di connessioni iniziate da eventuali client.
L'attesa è realizzata per mezzo di un programma che implementa un server Bluetooth, il quale riceve la richiesta e stabilisce un canale di comunicazione bidirezionale RFCOMM con il client. Si tratta di un server di tipo iterativo che quindi serve un client alla volta. Pertanto all'arrivo della richiesta di connessione, se il server è disponibile la accetta immediatamente, altrimenti la mantiene in coda fino a quando le richieste pendenti non siano state completate.
Il fatto di utilizzare un server iterativo non rappresenta un limite. Infatti nel caso ideale lo scopo del sistema è quello di trasferire sul server remoto tutti i dati ambientali rilevati dal Raspberry Pi, a partire da quelli più recenti(rispetto al momento in cui viene stabilita la connessione) a quelli più vecchi che non sono stati

ancora inoltrati. Ciò significa che è sufficiente che un solo dispositivo Android alla volta si connetta al Raspberry Pi, potendo così sfruttare tutta la banda disponibile. Nel caso in cui un secondo dispositivo Android voglia stabilire una connessione, verrà messo in attesa e non appena sarà accettato, inizierà a ricevere i dati che nel frattempo si sono accumulati tra l'istante in cui il client precedente ha stabilito la connessione e l'istante in cui il secondo client è stato accettato, più eventuali dati precedenti non ancora trasferiti, sfruttando a sua volta tutta la banda disponibile. Supponendo inoltre un'elevata partecipazione e connessioni frequenti, la quantità di dati che si accumula tra una connessione e la successiva è bassa e quindi distribuirla su più client con l'utilizzo di un server concorrente non porterebbe alcun vantaggio.

3.1.4 Inviare i dati accumulati allo smartphone connesso

Dopo aver stabilito la connessione, il server in esecuzione sul Raspberry Pi inizia a trasmettere i dati al dispositivo Android, iniziando da quelli più recenti presenti nei file contenenti le misure ambientali, fino a quelli più vecchi che non siano già stati inoltrati in precedenza. Per tenere traccia dei blocchi di dato che sono ancora da trasferire, viene fatto uso di un file di supporto il quale per ogni riga contiene due istanti temporali(timestamp) di inizio e fine blocco. Le misure comprese tra i due istanti saranno quindi prelevate e inviate. Dato che un blocco potrebbe includere un numero elevato di misure, viene inoltrato un pezzo del blocco(chunk) alla volta. Terminato l'inoltro di un chunk, il server attende conferma di avvenuto trasferimento da parte del client, prima di procedere con il successivo, continuando fino alla fine del blocco, se la connessione non si dovesse interrompere in anticipo. In caso di fallimento dell'invio di un chunk a causa dell'interruzione della connessione oppure della mancata ricezione della conferma da parte del client dopo un tempo finito, la connessione viene chiusa dal server che torna disponibile per nuovi client. Normalmente il fallimento può avvenire a causa di problemi durante l'invio, prima che l'intero chunk sia stato ricevuto. L'altra rara possibilità è che il client abbia ricevuto effettivamente l'intero chunk ma nel brevissimo lasso di tempo tra l'istante nel quale l'applicazione esegue l'invio della risposta e l'istante nel quale il modulo hardware Bluetooth inoltra il pacchetto, il client sia uscito dal raggio d'azione del Raspberry Pi.
In assenza di problemi di trasmissione, dopo aver inoltrato tutti i blocchi con successo, il server chiude la connessione con il client e torna in attesa di nuove connessioni oppure serve quelle pendenti.

3.2 Smartphone e applicazione Android HelpEnvironmentNow

Un qualsiasi smartphone(o tablet) che dispone del chipset Bluetooth e di una versione di Android uguale o superiore a 5.0 può essere parte del sistema installando l'applicazione HelpEnvironmentNow e accettando i permessi richiesti per accedere alla posizione del dispositivo. È anche necessario mantenere sempre(o comunque il più possibile) il Bluetooth attivo. Secondo varie ricerche, come evidenziato in [28], avere il Bluetooth sempre attivo sul proprio dispositivo non comporta un consumo significativo di energia nei lunghi periodi di inutilizzo. L'applicazione è quindi pensata per essere eseguita in maniera autonoma in background senza che sia richiesto l'intervento dell'utente. Le sue funzioni sono le seguenti:

- accorgersi della presenza di un Raspberry Pi
- connettersi al Raspberry Pi e ricevere le rilevazioni
- salvare le misure ricevute in una base di dati locale
- inoltrare i dati prelevati

3.2.1 Accorgersi della presenza di un Raspberry Pi

Un Raspberry Pi segnala la propria presenza inviando con continuità in broadcast dei beacon Bluetooth Low Energy in formato Eddystone. L'applicazione sfrutta delle scansioni Bluetooth filtrate a livello hardware quando possibile, ed esegue delle scansioni software periodiche in background in maniera da poter rilevare la presenza dei beacon desiderati. Le scansioni sono state implementate per via dalla libreria `Android Beacon Library` che con poche righe di codice consente di attivare questo funzionamento anche quando l'applicazione rimane sempre in background oppure nei casi in cui l'applicazione non è in esecuzione in quanto non le è permesso eseguire dei servizi indefinitamente. La libreria utilizza quindi dei meccanismi differenti in base alla versione del sistema operativo e del dispositivo per effettuare le scansioni.
Nel momento in cui viene rilevato un beacon BLE di tipo Eddystone con l'UID desiderato(deciso a priori e definito nel codice dell'applicazione) l'applicazione reagisce, estrae dal beacon l'indirizzo Bluetooth fisico e attiva un servizio temporaneo che avrà il compito di connettersi al dispositivo di monitoraggio.

3.2.2 Connettersi al Raspberry Pi e ricevere le rilevazioni

Il servizio temporaneo, avviato in seguito alla rilevazione di un Raspberry Pi in prossimità, come primo passo tenta di ottenere la posizione corrente attraverso il

sistema di geolocalizzazione dello smartphone, in quanto sarà utilizzata durante la fase di ricevimento dei dati per attribuirla alle misure ambientali dato che il Raspberry Pi non dispone della posizione geografica in cui è situato.
L'operazione successiva consiste nel connettersi al dispositivo di monitoraggio tramite protocollo Bluetooth RFCOMM. Se la connessione non ha successo immediatamente, vengono eseguiti ulteriori tentativi. Se la fase di connessione si conclude con successo, il servizio inizia a ricevere il flusso di oggetti JSON, contenenti le rilevazioni ambientali. Dopo ciascun oggetto JSON ricevuto per intero, il dispositivo Android inoltra conferma al dispositivo di monitoraggio. La dimensione di un oggetto JSON è stabilita dal dispositivo di monitoraggio ed è contenuta per far sì che il server Bluetooth riceva frequentemente conferma dei dati inoltrati.

3.2.3 Salvare le misure ricevute in una base di dati locale

Tra un'iterazione e l'altra del processo di lettura descritto nella sottosezione precedente, si estraggono dall'oggetto JSON le singole rilevazioni ambientali. A ciascuna misura ambientale deve essere associata la posizione geografica(latitudine, longitudine, altitudine) ottenuta precedentemente. La posizione geografica dovrebbe infatti corrispondere in maniera approssimativa alla posizione del dispositivo di monitoraggio. A questo punto le rilevazioni ambientali, arricchite con la posizione, vengono salvate nella base di dati dell'app prima di proseguire ciclicamente con il ricevimento dei blocchi rimanenti. Salvando le misure ambientali nella base di dati locale man mano che si ricevono, consente di ottenere la scalabilità dell'applicazione. Infatti anche se si devono ricevere un numero totale di misure molto elevato, è possibile farlo senza avere la necessità di strutture dati capienti da allocare nella memoria heap che essendo di dimensioni limitate rappresenterebbe un limite su alcuni dispositivi.

3.2.4 Inoltrare i dati prelevati

Dopo aver chiuso la connessione Bluetooth con il server, il servizio temporaneo imposta ed esegue una richiesta di schedulazione specificando un compito che dovrà essere eseguito non appena la connessione a Internet risulterà essere disponibile, sfruttando le potenzialità offerte dalle API WorkManager. Il servizio temporaneo termina appena dopo aver inoltrato la richiesta, che sarà soddisfatta dal sistema operativo appena possibile.
Il compito specificato nella richiesta di schedulazione consiste nell'andare a leggere dal database locale le misure ambientali presenti e inoltrarle al server remoto tramite la connessione a Internet attiva. Dato che il numero di rilevazioni da inviare potrebbe essere elevato, si prosegue in maniera ciclica: per ogni iterazione

si estrae un insieme di misure, si costruisce un oggetto JSON che le contenga e lo si inoltra al server remoto come corpo di una richiesta HTTP. Nel caso in cui si dovesse ricevere conferma che la richiesta sia stata gestita con successo da parte del server remoto, le misure che sono state trasmesse verrebbero rimosse dallo storage locale.
Può verificarsi che l'esecuzione del compito non abbia successo ad esempio perché la connessione si potrebbe interrompere o il server potrebbe non rispondere; in tal caso il compito verrebbe fermato in automatico e la sua esecuzione sarebbe ritentata, aumentando in modo esponenziale il tempo tra un tentativo e l'altro in caso di ripetuto fallimento.

3.3 Server remoto per la centralizzazione dei dati ambientali

Il server remoto svolge la funzione di raccolta e inserimento in un database MySQL dei dati ambientali provenienti da tutti i client Android. Per non concedere accesso diretto alla base dati dall'esterno in quanto fonte di problemi di sicurezza e gestionali, si interpone tra il client e la base dati un servizio web. Inserendo quindi uno strato intermedio, si rende indipendente il client dalla struttura della base di dati; per il client sarà soltanto necessario conoscere l'interfaccia del servizio web e il formato JSON accettato. Inoltre in questa maniera si aumenta la flessibilità in quanto in qualsiasi momento si può cambiare la configurazione o il tipo di DBMS senza dover modificare l'applicazione client.
Quindi il server remoto si aspetta di ricevere richieste HTTPS all'URL che identifica il servizio web e che contengono al loro interno il blocco di informazioni ambientali nel formato JSON stabilito, dal quale saranno estratte e inserite nel database.

3.3.1 Web service

Il servizio web è basato su architettura REST e dispone di un'interfaccia costituita da un solo metodo che riceve come parametro l'oggetto JSON contenente i dati ambientali. Il compito del servizio è quello di estrarre i dati ricevuti, preparare la stringa SQL e inserirli nella base di dati. Nel caso in cui tutta l'operazione abbia successo, viene ritornato al client un messaggio HTTP con codice 204 per indicare che la richiesta è stata soddisfatta e che può quindi procedere con eventuali altre richieste.

Capitolo 4

Implementazione

Questo capitolo presenta l'implementazione delle funzionalità descritte nel capitolo precedente mettendo in evidenza le sezioni di codice più rilevanti.
La prima sezione si concentra sulla comunicazione Bluetooth del dispositivo di monitoraggio con un client Android.
La seconda sezione descrive la struttura dell'applicazione mentre la terza sezione descrive il servizio web.

4.1 Raspberry Pi

Per segnalare la presenza del dispositivo di monitoraggio e per gestire la comunicazione con un client sono stati creati rispettivamente degli script Python ed è stato sviluppato un server Bluetooth.
Il server è stato sviluppato in due versioni distinte. In un primo momento ho prodotto una soluzione utilizzando il linguaggio C mentre nella fase finale della tesi ho utilizzato il linguaggio Python per produrre una nuova versione, differente sotto molti aspetti rispetto alla prima versione e alla quale ci si riferisce con la dicitura `versione finale` nelle sezioni successive.
Durante la fase di sviluppo sono stati utilizzati due Raspberry Pi differenti, entrambi dotati di modulo Bluetooth 4.1: Raspberry Pi 3 Model B e Raspberry Pi Zero W. Sul Raspberry Pi 3 è stato installato il sistema operativo Raspbian Stretch derivato dal più noto Debian(distribuzione completamente costituita da software libero basata su GNU/Linux) ed è stata utilizzata l'implementazione dello stack Bluetooth denominata Bluez, alla versione 5.37. Il Raspberry Pi Zero mi è stato predisposto con la distribuzione Arch Linux e versione 5.52 di Bluez.

4.1.1 Configurazione iniziale

Al fine di abilitare specifiche funzionalità legate al Bluetooth, sono necessarie alcune azioni elencate di seguito.
Il dispositivo Raspberry Pi dovrà trasferire i dati ambientali tramite connessione Bluetooth RFCOMM al dispositivo Android. Prima di stabilire la connessione, il protocollo Bluetooth richiede che i due dispositivi siano associati e ciò avviene in seguito al processo di pairing [13]. Normalmente questo richiede l'intervento dell'utente per l'accettazione. Lo scopo è però quello di rendere automatico questo processo e stabilire la connessione senza che venga richiesto il consenso dell'utente. Affinché questo sia possibile bisogna prima di tutto rendere sempre visibile il Raspberry Pi come dispositivo Bluetooth; questo si realizza modificando il file `/etc/bluetooth/main.conf` abilitando la voce `DiscoverableTimeout 0` che di base è commentata e quindi non avrebbe effetto.
Un'altra azione necessaria è quella di lanciare il seguente comando Bash:

```
1 echo -e 'discoverable on\npairable on \nagent on \ndefault-agent \
    nquit' | bluetoothctl
```

Listing 4.1: Comando Bash per abilitare pairing automatico

Questo comando sfrutta il programma interattivo `bluetoothctl` che internamente comunica con il daemon `bluetoothd` che fornisce accesso alle interfacce Bluetooth del dispositivo. In particolare viene registrato un `agent` che avrà il compito di gestire il pairing tra dispositivi e autorizzare le connessioni in arrivo.

4.1.2 Esecuzione automatica al boot

Per consentire l'avvio automatico del sistema di monitoraggio al boot sul Raspberry Pi si è sfruttato il gestore di sistema `systemd` che dà la possibilità di aggiungere un comando o un programma come un servizio del sistema operativo.
Sono stati definiti dei file con estensione `.service` che consentono di specificare gli script e i moduli Python che dovranno essere eseguiti come servizi all'avvio. Nel seguente blocco di codice è visualizzato il file `.service` relativo allo script Python che inoltra dei beacon BLE:

```
1 [Unit]
2 Description=Advertise beacon BLE
3 After=bluetooth.target
4
5 [Service]
6 ExecStart=/usr/bin/python /home/alarm/sensing_station_platform/bt/
    broadcast_eddy.py
7 Restart=always
8
9 TimeoutStopSec=30
```

```
10
11 [Install]
12 WantedBy=multi-user.target
```

Listing 4.2: File .service per avviare il servizio che annuncia beacon BLE

Lo stesso meccanismo è stato utilizzato per avviare il server Bluetooth.

4.1.3 Invio continuo di beacon BLE

Nella sottosezione precedente si è fatto riferimento allo script Python eseguito come servizio che annuncia beacon BLE in broadcast. L'obiettivo viene raggiunto per mezzo della libreria `Scapy` che tra le sue moltissime funzionalità di rete, include la possibilità di annunciare in broadcast diversi formati di beacon Bluetooth Low Energy tra cui anche il formato Eddystone da me utilizzato.
Il contenuto dello script Python è il seguente:

```
1  from scapy.layers import bluetooth
2  from scapy.contrib import eddystone
3  from scapy import compat
4
5  # open a HCI socket to device hci0
6  bt = bluetooth.BluetoothHCISocket(0)
7
8  # set the Eddystone frame
9  ns = compat.hex_bytes('a8844da2d40a11e9bb65')
10 ins = compat.hex_bytes('2a2ae2dbcce4')
11 edy_UID = eddystone.Eddystone_UID(tx_power=0, namespace=ns, instance=
       ins,)
12 edy_frame = eddystone.Eddystone_Frame() / edy_UID
13 bt.sr(edy_frame.build_set_advertising_data())
14
15 # start advertising
16 bt.sr(bluetooth.HCI_Hdr()/
17       bluetooth.HCI_Command_Hdr()/
18       bluetooth.HCI_Cmd_LE_Set_Advertise_Enable(enable=True))
```

Listing 4.3: Script per inoltrare beacon BLE in broadcast

Lo script definisce l'identificatore univoco di 16 byte in formato esadecimale(suddivisi in due parti da 10 e 6 byte come previsto dallo standard) e imposta il formato `Eddystone_UID` durante la creazione del frame Eddystone. Tramite l'ultimo metodo `bt.sr(...)` viene avviato il broadcast periodico di beacon contenenti l'identificatore specificato.

4.1.4 Server bluetooth - versione finale

Per gestire la comunicazione con un client Android, è stato sviluppato un programma in linguaggio Python che implementa un server Bluetooth RFCOMM. Il suo compito è quello di accettare le richieste di connessione e inviare le misure ambientali dopo averle prelevate dai file che le contengono. Prima di descrivere il flusso del server, è importante comprendere come vengono salvate le misure rilevate.
Il gruppo di ricerca ha sviluppato dei programmi in linguaggio Python che ricevono dai sensori di temperatura/umidità e di particolato atmosferico le letture delle rispettive grandezze fisiche e le salvano all'interno di due file in formato CSV che vengono generati all'inizio di ogni nuovo giorno e sono destinati a contenere tutte le misure delle ventiquattro ore successive. Per ogni giorno saranno quindi disponibili due file con nomi unici corrispondenti alla rispettiva data convertita in stringa.
Il server Bluetooth RFCOMM, dopo aver stabilito la connessione con il client, dovrà prelevare i dati dai file CSV e trasferirli. Le rilevazioni ambientali devono essere inviate in ordine decrescente cioè partendo da quelle più recenti(rispetto all'istante in cui viene stabilita la connessione) fino a quelle più datate. Inoltre le rilevazioni che sono già state inoltrate con successo in una precedente connessione, non saranno più inviate.
Per ogni misura rilevata da un sensore viene generata una riga contenente il numero del sensore logico(`sensorID`), l'istante temporale in cui è stata rilevata(`timestamp`), il valore della misura(`data`) e due campi che indicano la posizione geografica che però vengono ignorati dal server Bluetooth, in quanto saranno assegnati successivamente dal client.

1	sensorID	timestamp	data	geohash	altitude
2	200	1579651200	36	abcdefghIJ	265
3	201	1579651200	50	abcdefghij	265
4	202	1579651200	40	abcdefghij	265
5	203	1579651200	56	abcdefghij	265
6	204	1579651200	37	abcdefghIJ	265
7	205	1579651200	52	abcdefghij	265
8	206	1579651200	33	abcdefghij	265
9	207	1579651200	46	abcdefghij	265
10	200	1579651201	36	abcdefghIJ	265
11	201	1579651201	50	abcdefghij	265
12	202	1579651201	40	abcdefghij	265
13	203	1579651201	56	abcdefghij	265
14	204	1579651201	38	abcdefghIJ	265
15	205	1579651201	53	abcdefghij	265
16	206	1579651201	34	abcdefghij	265
17	207	1579651201	48	abcdefghij	265
18	200	1579651202	35	abcdefghIJ	265
19	201	1579651202	49	abcdefghIJ	265

Figura 4.1: Esempio di file CSV contenente le rilevazioni ambientali

Per tenere traccia delle rilevazioni che non sono state ancora inoltrate utilizzo un file accessorio `blocks.csv` che per ogni riga contiene due istanti temporali che indicano l'inizio e la fine di un intervallo di rilevazioni ancora da trasferire.

start_timestamp	end_timestamp
1579705320	1579651200
1579726800	1579719600
1579749900	1579748400
0	1579750800

Figura 4.2: Esempio del file blocks.csv

Le righe del file sono ordinate a partire dagli intervalli con le rilevazioni più vecchie fino all'intervallo con le rilevazioni più recenti. Il campo `start_timestamp` dell'ultima riga del file(che fa riferimento al blocco di dati più recente) contiene per convenzione il valore `0` per indicare che il server dovrà considerare come istante iniziale quello corrispondente all'istante in cui viene stabilita la connessione con un

client.
Per maggiore chiarezza nella seguente figura viene mostrata una panoramica sulla memorizzazione delle rilevazioni ambientali e del modo in cui si tiene traccia di quelle da trasmettere:

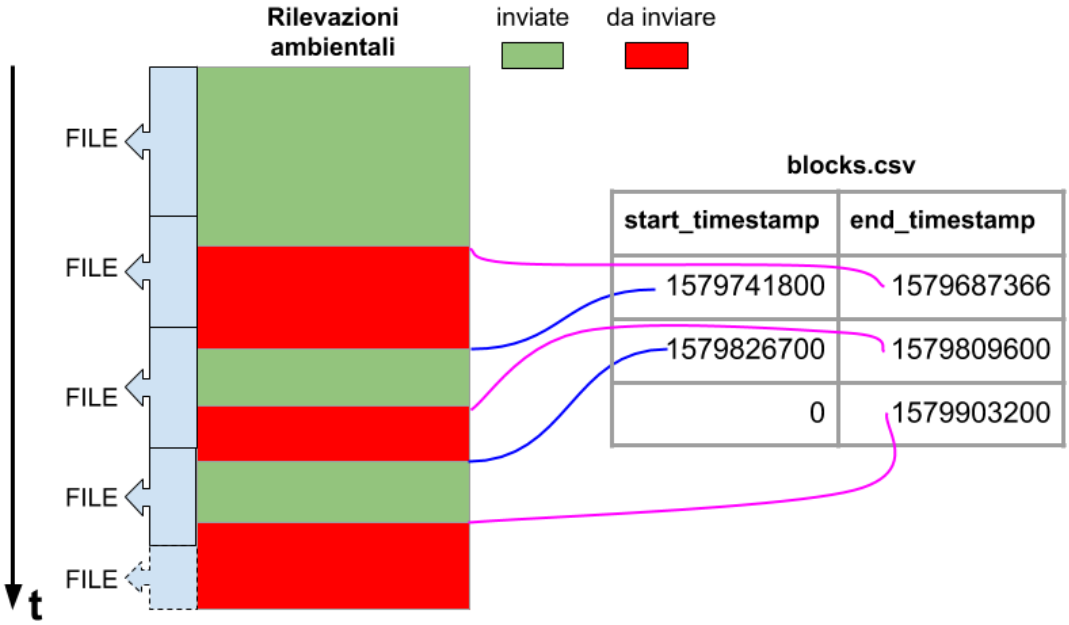

Figura 4.3: Panoramica dell'organizzazione delle rilevazioni ambientali

Tenendo conto del modo in cui sono memorizzate le rilevazioni ambientali, nel momento in cui si stabilisce la connessione con un client, il server leggerà il file `blocks.csv` e inizierà l'invio del blocco più recente che potrebbe includere misure memorizzate in più file distinti. Inoltre siccome un blocco potrebbe contenere un numero elevato di rilevazioni, viene inviato a piccoli pezzi(chunk).
Tutte le rilevazioni presenti in un chunk vengono convertite e inviate in formato JSON al client.

```
{
    "m": [
        { "sensorID": 20, "timestamp": 1234567890, "data": 23.4 },
        { "sensorID": 25, "timestamp": 1234567890, "data": 47.4 },
        { "sensorID": 12, "timestamp": 1234567890, "data": 25 },
        { "sensorID": 13, "timestamp": 1234567890, "data": 6 },
        { "sensorID": 20, "timestamp": 1234567891, "data": 23.4 }
    ]
}
```

Listing 4.4: Esempio di un JSON inviato al client

Nel seguente diagramma è visualizzato il flusso del server dopo aver stabilito la connessione con un client. Per semplicità è omesso il caso in cui l'invio di un chunk dovesse fallire. In tal caso infatti tutti e tre i cicli verrebbero interrotti e si passerebbe direttamente ad aggiornare il file che tiene traccia delle rilevazioni ancora da inoltrare.

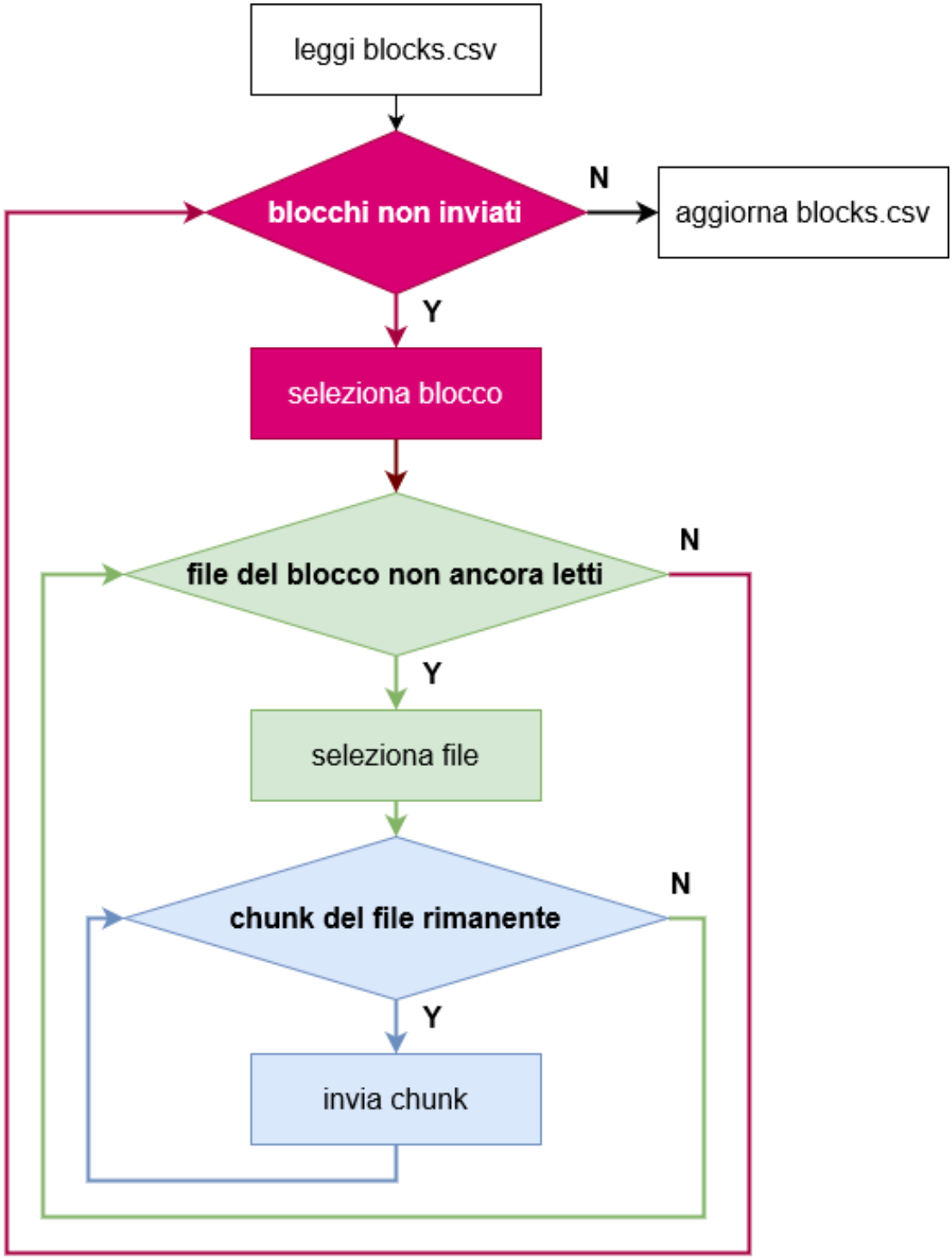

Figura 4.4: Flusso del server

La versione finale del server appena descritto l'ho sviluppata per adattarla al nuovo sistema di memorizzazione delle misure basata su file CSV. Ho scelto il linguaggio Python per lo sviluppo del server sia per uniformarlo con i programmi che sono stati prodotti dal gruppo di ricerca e che creano e scrivono le misure nei file sia perché vi sono molte librerie che facilitano l'accesso ai dati in formato CSV. Questo sistema è più adeguato anche per alcuni casi particolari, come potrebbe essere quello di un dispositivo di monitoraggio che rimane per lunghi periodi di tempo isolato. Inoltre utilizzando il formato JSON per lo scambio dei dai risulta più semplice e più facile da modificare in futuro.

4.1.5 Server bluetooth - prima versione

Per completezza in questa sottosezione viene presentata la prima versione del server Bluetooth che ho sviluppato in linguaggio C. Questa versione del server è stata testata sul Raspberry Pi 3. Le due principali differenze rispetto alla versione finale, presentata nel capitolo precedente, sono il sistema utilizzato per memorizzare le misure ambientali e il formato in cui vengono inviate al client. Per quanto riguarda il primo punto, in questo caso le misure non vengono memorizzate e lette dai file ma si fa uso di un'area di memoria condivisa. Per quanto riguarda il secondo punto, non viene utilizzato il formato JSON per inviare le misure ma si inoltra un flusso di byte seguendo una sequenza prestabilita e nota al client. Entrambi i punti saranno sviluppati più nel dettaglio nelle prossime righe.
Un programma sviluppato dal gruppo di ricerca e denominato `weather_station.c` è stato il punto di partenza dello sviluppo. Esso conteneva la parte di codice che legge con regolarità i dati dai sensori. Pertanto ho avuto il compito di aggiungere la parte necessaria per memorizzare ogni singola lettura all'interno di un'area di memoria condivisa e per gestire la comunicazione con lo smartphone via Bluetooth. Il main del programma era stato creato per generare in partenza dei processi figli che avessero il compito di leggere i dati dal sensore di temperatura ed umidità(DHT22) ogni due secondi e da quelli per il particolato atmosferico(HPMA115S0-XXX Honeywell) ogni secondo. Dopo aver generato i processi figli, il main avvia un ciclo infinito all'interno del quale controlla che i processi figli siano ancora attivi.
A questa struttura di partenza ho aggiunto la generazione di un ulteriore processo responsabile di svolgere la funzione di server Bluetooth RFCOMM e quindi di rimanere in attesa di connessioni da parte dei dispositivi Android.
Per poter accumulare i dati letti dai sensori viene creata nelle prime fasi del programma un'area di memoria condivisa. L'area condivisa è necessaria proprio perché ci sono vari processi che ricevono le misure dai sensori e le scrivono nella memoria condivisa dalla quale il processo del server Bluetooth RFCOMM andrà poi a leggere quando richiesto. Per gestire l'accesso concorrente alla risorsa condivisa è utilizzato un semaforo che permette a un solo processo per volta di accederci.

L'area di memoria condivisa è strutturata come segue: due campi indicano il numero di misure provenienti rispettivamente dal sensore DHT22 e dai sensori del particolato atmosferico; seguono due vettori di tipo char che rappresentano effettivamente i due buffer all'interno dei quali scrivere le misure rilevate dalle due tipologie sensori e infine ci sono ulteriori due campi che sono i puntatori alla prossima locazione libera all'interno dei due buffer. Di seguito viene mostrata tale struttura in linguaggio C che viene mappata nell'area di memoria condivisa:

```
struct shm_area {
    size_t current_reads_dht; // this field increments each time a
    new read is done until MAX_READS_DHT
    size_t current_reads_pm; // this field increments each time a new
     read is done until MAX_READS_PM
    char data_dht[DHT_DATA_SIZE];
    char data_pm[PM_DATA_SIZE];
    char *current_dht_data_ptr;
    char *current_pm_data_ptr;
};
```

Listing 4.5: Struttura in C della memoria condivisa

I due buffer vengono scritti in maniera circolare: quando sono pieni si ricomincia a scrivere dall'inizio e quindi i dati salvati da più tempo andranno persi in quanto saranno sovrascritti dall'arrivo di quelli nuovi.

Il blocco di codice seguente contiene la funzione che crea l'area di memoria condivisa, la inizializza opportunamente e crea il semaforo:

```
void create_shared_concurrent_dht_area() {
    const char * shm_name  = "/dht";
    int shm_fd;

    shm_fd = shm_open(shm_name, O_CREAT | O_RDWR, 0666); // 0666 read
    -write permissionss
    if (shm_fd == -1) {
        perror("Shared memory segment failed\n");
        system("reboot");
    }
    ftruncate(shm_fd, shm_size);
    shm_ptr = mmap(0, shm_size, PROT_READ | PROT_WRITE, MAP_SHARED,
    shm_fd, 0);
    if (shm_ptr == MAP_FAILED) {
        perror("Map failed\n");
        system("reboot");
    }
    shm_ptr->current_reads_dht = 0;
    shm_ptr->current_reads_pm = 0;
```

```
18      shm_ptr->current_dht_data_ptr = shm_ptr->data_dht; // points to
        the start of data_dht area
19      shm_ptr->current_pm_data_ptr = shm_ptr->data_pm; // points to the
         start of data_pm area
20      last_read_dht_ptr = shm_ptr->data_dht + ((MAX_READS_DHT - 1) *
        DHT_READ_SIZE); // points to the last free space for dht reads
21      last_read_pm_ptr = shm_ptr->data_pm + ((MAX_READS_PM - 1) *
        PM_READ_SIZE); // points to the last free space for pm reads
22      sem_id = sem_open("/sem_dht", O_CREAT, S_IRUSR | S_IWUSR, 1);
23  }
```

Listing 4.6: Funzione in C per inizializzare la memoria condivisa

La variabile `shm_ptr` è un puntatore a una `struct shm_area` definita globalmente così da poter essere utilizzata da tutti i processi figli che avranno la necessità di accedere all'area di memoria condivisa. Lo stesso vale anche per `sem_id` che è un puntatore a una variabile di tipo `sem_t` e viene utilizzato da un processo per ottenere l'ingresso nella sezione critica e per segnalare l'uscita dalla stessa.
Nel caso in cui si dovessero verificare errori in fase di creazione e inizializzazione della memoria condivisa, il programma non potrebbe funzionare e quindi il sistema verrebbe riavviato. Considerato che questa funzione viene eseguita al lancio del programma, il riavvio non provoca la perdita di alcun dato.

Successivamente viene visualizzato il blocco di codice che memorizza i dati relativi al particolato atmosferico nella memoria condivisa come una sequenza di byte. Tale blocco di codice è eseguito dal processo che comunica con il sensore del particolato atmosferico. Normalmente ci sono quattro processi di questo genere, uno per ognuno dei quattro sensori.

```
1  sprintf(message, "%ld%0*d%0*d%0*d%0*d", t, PM_VALUE_SIZE,
       write_buffer[0], PM_VALUE_SIZE, write_buffer[1], SENSOR_SN_SIZE,
       Sensor_SN[index], SENSOR_SN_SIZE, Sensor_SN[index+1]);
2  sem_wait(sem_id);
3       memcpy(shm_ptr->current_pm_data_ptr, message, PM_READ_SIZE);
4       if(shm_ptr->current_reads_pm < MAX_READS_PM)
5           shm_ptr->current_reads_pm++;
6       if(shm_ptr->current_pm_data_ptr < last_read_pm_ptr)
7           shm_ptr->current_pm_data_ptr += PM_READ_SIZE;
8       else // circular buffer, when the buffer is full it is
        overwritten
9           shm_ptr->current_pm_data_ptr = shm_ptr->data_pm; // points to
         the start of pm shm area for data_pm
10 sem_post(sem_id);
```

Listing 4.7: Memorizzazione delle misure PM nell'area condivisa

Come si può notare dalla funzione `sprintf`, per ogni valore di PM10 e PM2.5 viene associato anche un istante temporale(come Unix timestamp) e il numero del

corrispondente sensore logico. Prima di procedere con la scrittura nella memoria condivisa viene eseguita una chiamata a `sem_wait` per ottenere l'accesso, si inserisce quindi la sequenza di byte ed eventualmente si incrementa il numero di letture presenti nel buffer nel caso in cui non abbia già raggiunto la dimensione massima. Il processo che comunica con il sensore DHT22, esegue un blocco di codice simile a quello appena descritto.

Nel momento in cui si riceve una connessione da parte di un client, i dati accumulati nella memoria condivisa vengono inviati come flusso di byte. In particolare vengono per prima inoltrati i metadati in maniera che il client sia in grado di sapere quanti byte dovrà leggere. Dopodiché si prosegue con l'invio dei dati relativi alla temperatura ed umidità. Lo stesso ordine si applica poi per le informazioni sul particolato atmosferico. La struttura della sequenza che viene trasmessa è organizzata come segue:

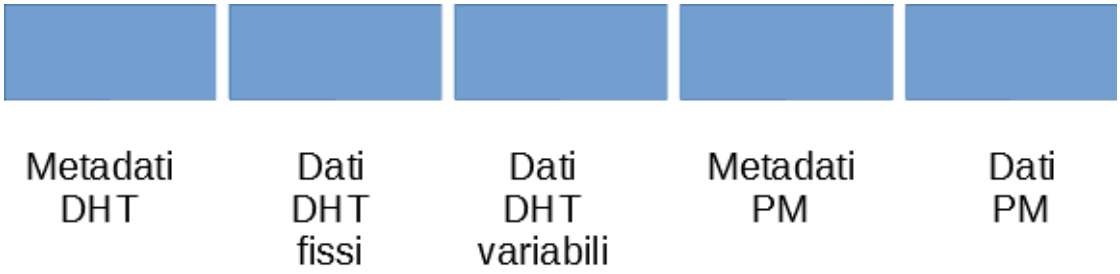

Figura 4.5: Struttura del flusso di byte inviato dal Raspberry Pi

DHT fa riferimento al sensore di temperatura ed umidità mentre PM si riferisce al sensore per il particolato atmosferico.

I metadati servono ad indicare la dimensione dei dati ambientali. Andando più nello specifico la struttura dei metadati DHT è la seguente:

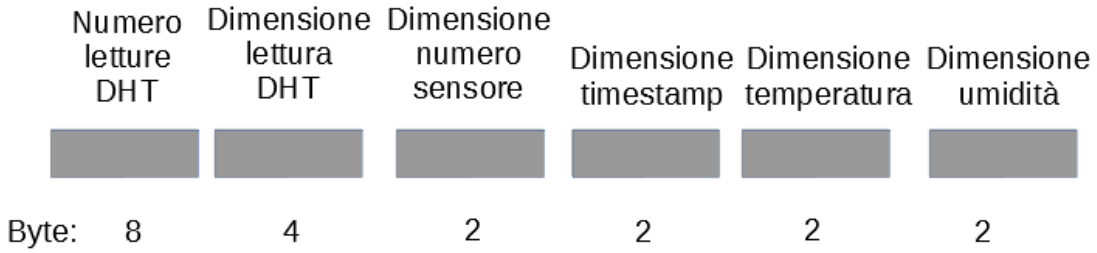

Figura 4.6: Struttura dei metadati DHT

Ciascun campo che compone i metadati, contiene tanti caratteri quanti sono i

byte del campo stesso. Per esempio il campo `Dimensione timestamp` è composto da due caratteri. Il valore dei due caratteri saranno due cifre che servono ad indicare la dimensione del timestamp; nello specifico i due caratteri avranno valore `10` in quanto il timestamp Unix è rappresentato da una sequenza di dieci cifre. Considerazioni analoghe valgono per gli altri campi `Dimensione X`. Per quanto riguarda invece il primo campo `Numero letture DHT`, esso è composto da otto caratteri e il suo valore indica il numero di letture DHT che saranno presenti nei dati DHT variabili. Per esempio, se nel momento in cui il Raspberry Pi stabilisce la connessione con un client, sono state memorizzate 275 letture di temperatura ed umidità, il campo conterrà il valore `00000275`.

Dopo i metadati seguono i dati DHT che si dividono in una parte fissa e una variabile. I dati DHT fissi sono semplicemente due numeri che identificano rispettivamente il sensore logico per la temperatura e il sensore logico per l'umidità. La parte variabile contiene invece tutte le letture memorizzate dal sensore DHT. Ciascuna lettura DHT ha il seguente formato:

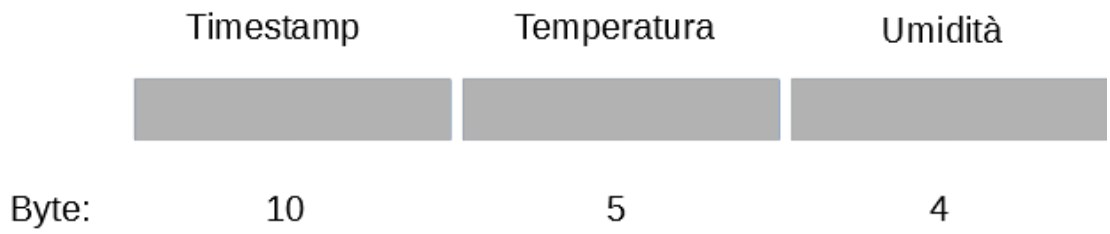

Figura 4.7: Struttura singola lettura DHT

La dimensione complessiva dei dati variabili DHT è quindi facilmente calcolabile per il client, moltiplicando il valore contenuto nel primo campo dei metadati DHT per il valore contenuto nel secondo campo dei metadati DHT(che è la somma delle dimensioni dei campi della Fig. 4.3).
La struttura dei metadati PM è molto simile a quella DHT e non viene riportata. Per i dati PM non ci sono dei dati fissi da poter inviare una volta sola. Questo è dovuto al fatto che mentre per la temperatura ed umidità viene utilizzato un solo sensore, e quindi è sufficiente inviare una volta sola il numero del sensore logico per la temperatura e per l'umidità, per il particolato vengono usati quattro sensori e quindi è necessario associare a ciascuna lettura il numero del relativo sensore logico.

Il blocco di codice relativo al server Bluetooth RFCOMM che inoltra i dati seguendo la convenzione appena descritta, è visualizzato di seguito. Per facilitarne la lettura, sono state rimosse le istruzioni relative ai casi anomali.

```
1 while(1) {
```

```
2       client = accept(server_socket, NULL, NULL);
3
4       sem_wait(sem_id);
5       if(shm_ptr->current_reads_dht > 0 && shm_ptr->current_reads_pm >
        0) {
6           /* send dht meta-data to the client */
7           result = send_dht_meta_data(shm_ptr->current_reads_dht,
        client);
8           if(result != -1) {
9               /* send dht fixed data to client */
10              result = send_dht_fixed_data(client);
11              if(result != -1) {
12                  /* send dht variable data to client */
13                  result = send_variable_data(shm_ptr->
        current_reads_dht, client, shm_ptr->data_dht, DHT_READ_SIZE);
14                  if(result != -1) {
15                      /* send pm meta data to client */
16                      result = send_pm_meta_data(shm_ptr->
        current_reads_pm, client);
17                      if(result != -1) {
18                          /* send pm variable data to client */
19                          result = send_variable_data(shm_ptr->
        current_reads_pm, client, shm_ptr->data_pm, PM_READ_SIZE);
20                          if(result != -1) {
21                              if(wait_ack(client) == 1) {
22                                  printf("ACK ricevuto\n");
23                                  reset_shared_buffers(shm_ptr);
24                              }
25                          }
26                      }
27                  }
28              }
29          }
30      }
31      sem_post(sem_id);
32      shutdown(client, SHUT_RDWR);
33      close(client);
34  }
```

Listing 4.8: Server Bluetooth RFCOMM in C

Per la comunicazione tra il server e il client sono stati utilizzati i socket Bluetooth di tipo RFCOMM. Anche se il codice della creazione del socket non è riportato, è utile specificare che il livello di sicurezza del socket Bluetooth è stato impostato a `MEDIUM` per permettere il pairing automatico quando si riceve una richiesta di connessione; questa impostazione deve essere accompagnata dalle azioni elencate in precedenza nella sottosezione `Configurazione iniziale`. Tornando al codice riportato si può osservare che il server rimane in attesa di connessioni, in quanto la

funzione `accept` è bloccante e si procede non appena si stabilisce la connessione con un client Bluetooth. Siccome è necessario accedere alla memoria condivisa all'interno della quale ci sono i dati ambientali si esegue una `sem_wait` e quando si ottiene l'accesso si prosegue con la lettura e l'invio dei dati al client. Anche se di fatto viene inviato un flusso continuo di byte, sono state definite diverse funzioni ognuna responsabile di inoltrare uno specifico blocco. Se in qualsiasi punto della procedura di invio si dovesse verificare un errore, l'operazione non andrebbe a buon fine e il server tornerebbe in ascolto, rimanendo bloccato sulla funzione `accept` pronto a servire un nuovo client. Sia in caso di invio con successo che in caso di fallimento, viene rilasciata la memoria condivisa tramite la chiamata a `sem_post` e chiusa la connessione con il client.

4.2 Applicazione Android

L'applicazione Android chiamata HelpEnvironmentNow è stata sviluppata in linguaggio Java utilizzando l'ambiente di sviluppo integrato Android Studio. Considerato il fatto che non richiede l'interazione con l'utente per funzionare, dispone di una sola Activity mentre il componente Android che gestisce la connessione e lo scambio dati con il Raspberry Pi è il servizio denominato ClassicService.
Nelle pagine che seguono si descrivono più nel dettaglio le componenti e le classi Java dell'applicazione con la visualizzazione dei blocchi di codice più rilevanti. La versione dell'applicazione qui presentata è quella compatibile con la versione finale del server Bluetooth. In precedenza avevo invece sviluppato una soluzione compatibile con la prima versione del server.
Prima di proseguire può essere utile visualizzare i package e le classi Java che compongono l'app:

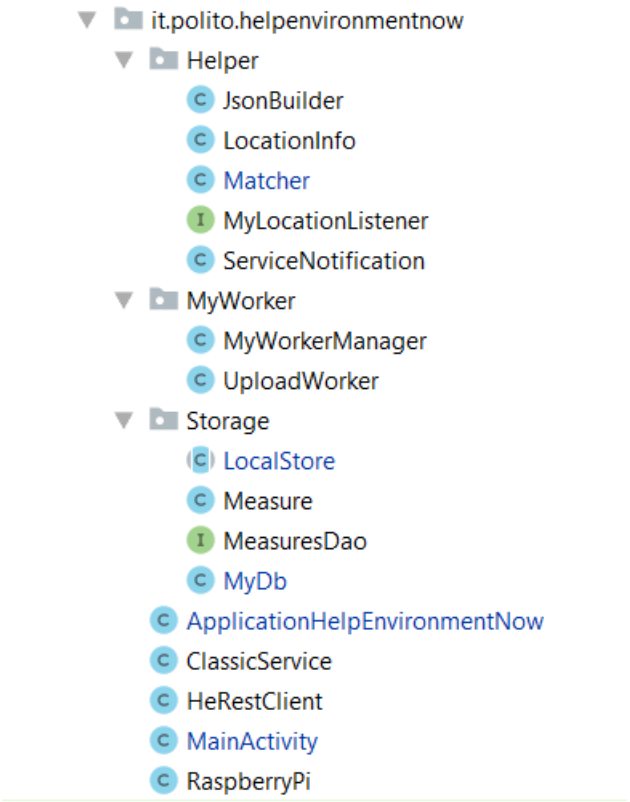

Figura 4.8: Classi e interfacce dell'app

Le cinque classi dell'applicazione che implementano le funzionalità principali sono ApplicationHelpEnvironmentNow, ClassicService, HeRestClient, MainActivity e RaspberryPi. Oltre a richiamare dei loro metodi interni, sfruttano le classi e le interfacce presenti nei tre package Helper, MyWorker e Storage. Essi contengono i metodi per svolgere compiti specifici di più basso livello e sono particolarmente utili per separare i compiti e aumentare la leggibilità delle classi principali.

4.2.1 Il manifest file dell'applicazione

Il manifest file visualizzato di seguito contiene solo i permessi e le componenti dell'app aggiunte manualmente. In realtà il file completo è costituito da molte più righe. Infatti alcune voci vengono aggiunte in automatico dall'ambiente di sviluppo Android Studio nella fase di creazione del progetto, come per esempio il valore dei campi `minSdkVersion` e `targetSdkVersion` il cui significato è stato spiegato nella sottosezione `Manifest file`. Altro contenuto è invece iniettato nel file XML al momento dell'aggiunta di librerie tra le dipendenze del file build.gradle(app). Per esempio l'utilizzo della libreria `Android Beacon Library` ha comportato l'aggiunta

di alcuni permessi come `android.permission.BLUETOOTH` oppure di alcune componenti come broadcast receiver o servizi.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="it.polito.helpenvironmentnow">

    <uses-permission android:name="android.permission.
    ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.
    FOREGROUND_SERVICE" />
    <uses-permission android:name="android.permission.
    ACCESS_FINE_LOCATION" />

    <application
        android:name=".ApplicationHelpEnvironmentNow"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:theme="@style/AppTheme">
        <activity android:name=".ConfigActivity"></activity>
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:launchMode="singleInstance">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
    LAUNCHER" />
            </intent-filter>
        </activity>

        <service android:name=".ClassicService" />

    </application>

</manifest>
```

Listing 4.9: File AndroidManifest.xml dell'app HelpEnvironmentNow

4.2.2 Le dipendenze esterne

Il file `build.gradle(app)` del progetto Android contiene tra le proprie informazioni le dipendenze, ovvero i riferimenti alle librerie esterne con le rispettive versioni che sono state aggiunte al progetto, come visualizzato di seguito dove sono state

elencate solo le dipendenze aggiunte manualmente, omettendo quelle aggiunte automaticamente da Android Studio:

```
dependencies {
    implementation 'org.altbeacon:android-beacon-library:2.16.2'
    implementation 'com.loopj.android:android-async-http:1.4.10'
    implementation 'com.google.android.gms:play-services-location
    :17.0.0'
    implementation 'com.github.drfonfon:android-geohash:0.22'
    implementation "androidx.work:work-runtime:2.2.0"
    implementation "androidx.room:room-runtime:2.2.1"
}
```

4.2.3 MainActivity

L'applicazione durante il suo funzionamento ha la necessità di utilizzare la posizione del dispositivo Android e quindi di conseguenza l'utente deve concedere i permessi ritenuti pericolosi per la sua privacy. Di fatto si necessita del permesso `ACCESS_FINE_LOCATION` ritenuto `dangerous` dal sistema operativo. Pertanto è necessario che l'utente fornisca i permessi richiesti. Ci sono due modi per farlo, forzati dal sistema in base alla versione del sistema operativo tenuto conto che il valore di `targetSdkVersion` risulta essere impostato a 28 nel manifest file(dove 28 indica il livello di API corrispondente ad Android 9 Pie).
Nel caso in cui la versione di Android sia inferiore ad Android 6.0, il sistema chiederà in automatico i permessi necessari all'utente in fase di installazione dell'applicazione. Quindi l'utente deve concedere il permesso per accedere alla posizione del dispositivo per poter proseguire con l'installazione che altrimenti verrebbe annullata. Dopo questa fase non sarà più richiesto il suo intervento.
Se invece la versione del sistema operativo è uguale o superiore ad Android 6.0, i permessi non saranno richiesti in fase di installazione ma a run-time. Pertanto dopo aver concluso l'installazione è necessario che l'utente la mandi in esecuzione premendo sulla relativa icona. A questo punto sullo schermo apparirà la MainActivity che è l'unica Activity presente e ha il solo scopo di chiedere i permessi all'utente nel momento in cui l'applicazione viene avviata per la prima volta, e successivamente mostra solo una breve descrizione all'utente di come l'applicazione sia in grado di funzionare in maniera autonoma mantenendo il Bluetooth attivo.
L'applicazione necessita del permesso per accedere alla posizione del dispositivo per due motivi. Il primo motivo riguarda la scansione Bluetooth Low Energy; infatti la piattaforma Android richiede tale permesso affinché un'app possa eseguire delle scansioni BLE in quanto a partire dai beacon BLE rilevati durante la scansione è possibile risalire alla posizione del dispositivo e quindi localizzare l'utente. La

seconda ragione consiste nella necessità di accedere alla posizione del dispositivo Android prima di ricevere i dati da un Raspberry Pi in maniera da poter associare i dati raccolti a una posizione geografica prima di inviare le rilevazioni al server remoto.

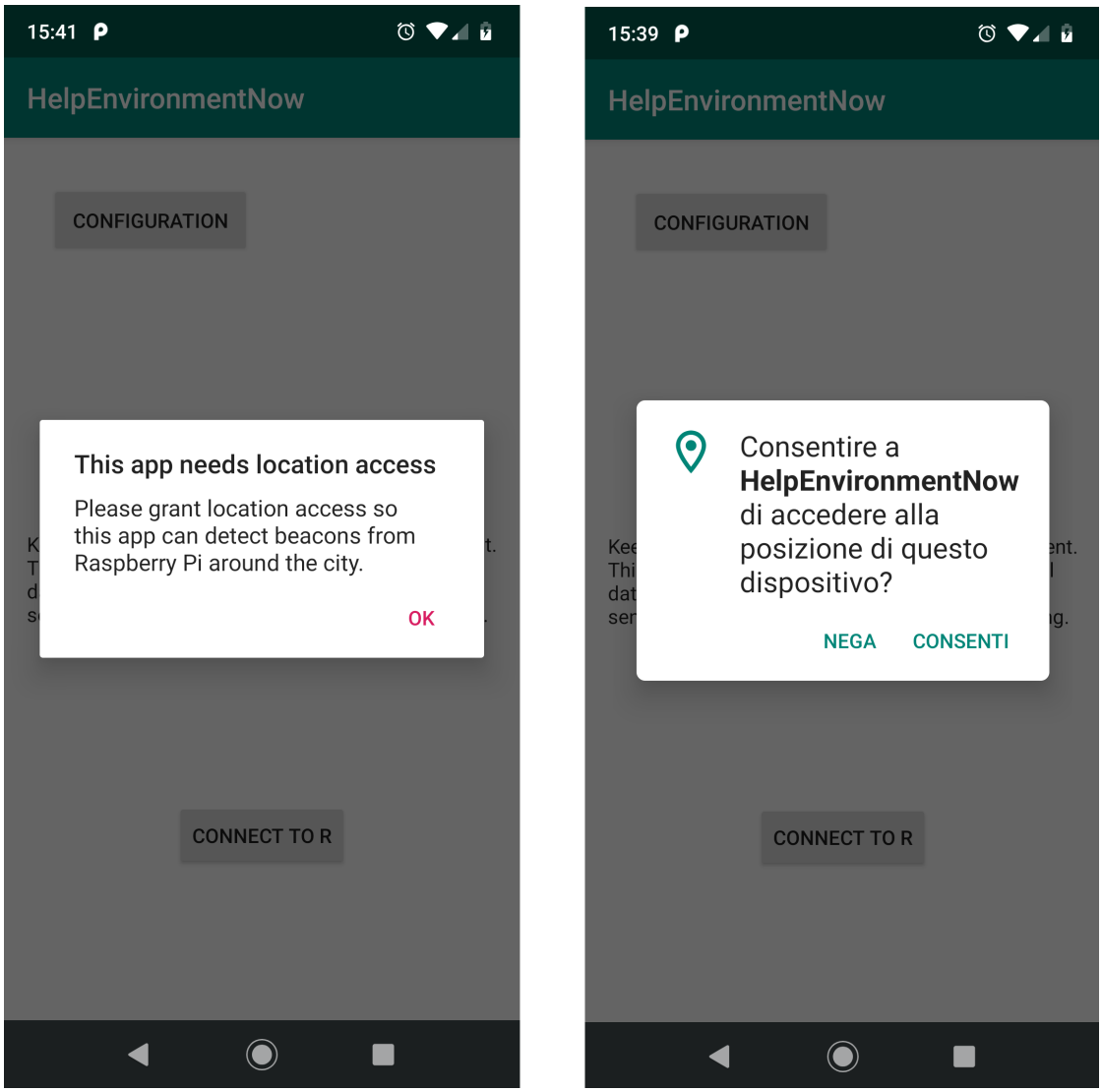

Figura 4.9: Richiesta permessi al primo lancio per Android 6.0+

4.2.4 Flusso dell'app

Il seguente diagramma illustra in quale sequenza sono eseguite le principali classi dell'applicazione in seguito alla rilevazione di un beacon proveniente da un dispositivo di monitoraggio.

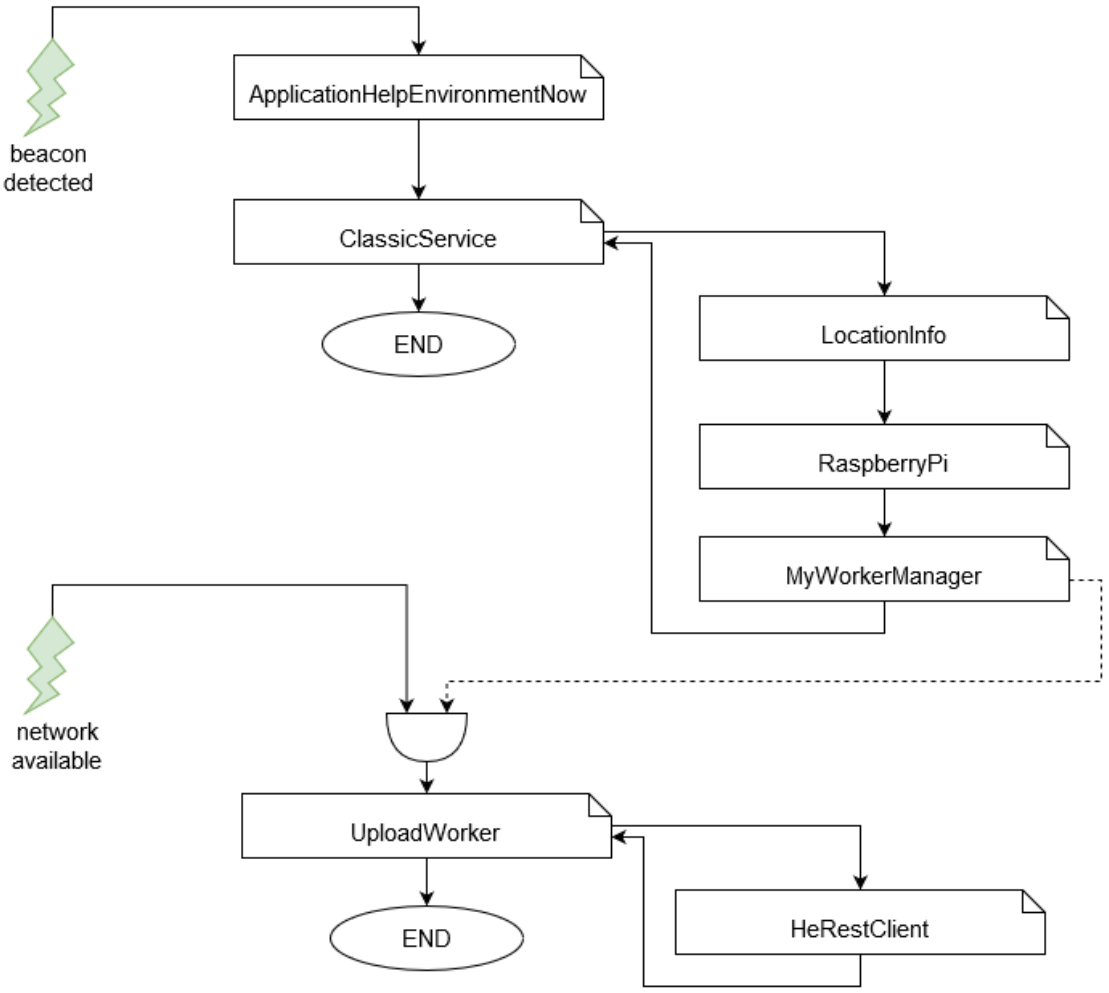

Figura 4.10: Flusso dell'invocazione delle classi nell'app

4.2.5 Classe ApplicationHelpEnvironmentNow

Per ogni applicazione Android in esecuzione, all'interno del suo processo viene istanziato un unico oggetto di classe `Application` o di una sua sottoclasse, come in questo caso; infatti ApplicationHelpEnvironmentNow estende la classe `Application` avendo così la possibilità di ridefinire il comportamento di default. Inoltre la classe ApplicationHelpEnvironmentNow implementa alcune interfacce della libreria `Android Beacon Library`. Questo meccanismo è suggerito dalla documentazione della libreria [24] appena citata per consentire che l'applicazione venga lanciata dal sistema attraverso un intent quando viene rilevato un beacon dalle caratteristiche imposte, nelle vicinanze. Di seguito viene presentato il codice di cui sopra:

```
1  public class ApplicationHelpEnvironmentNow extends Application
       implements BootstrapNotifier, RangeNotifier, Configuration.
       Provider {
2
3      private static final String namespaceId="0xa8844da2d40a11e9bb65";
4      private static final String instanceId = "0x2a2ae2dbcce4";
5      private RegionBootstrap regionBootstrap;
6      private BeaconManager beaconManager;
7      private Region region;
8      private Identifier myBeaconNamespaceId, myBeaconInstanceId;
9
10     @Override
11     public void onCreate() {
12         super.onCreate();
13         Log.d(TAG, "App started up");
14         initializeMovementMode();
15         beaconManager = BeaconManager.getInstanceForApplication(this)
       ;
16         beaconManager.setRegionStatePersistenceEnabled(false);
17         beaconManager.getBeaconParsers().add(new BeaconParser().
       setBeaconLayout(BeaconParser.EDDYSTONE_UID_LAYOUT));
18         // wake up the app when any beacon is seen (you can specify
       specific id filters in the parameters below)
19         myBeaconNamespaceId = Identifier.parse(namespaceId);
20         myBeaconInstanceId = Identifier.parse(instanceId);
21         region = new Region("it.polito.helpenvironmentnow.
       boostrapRegion", myBeaconNamespaceId, myBeaconInstanceId, null);
22         regionBootstrap = new RegionBootstrap(this, region);
23     }
24
25     @Override
26     public void didDetermineStateForRegion(int state, Region arg1) {
27         if(state == INSIDE) {
28             try {
29                 beaconManager.addRangeNotifier(this);
30                 beaconManager.startRangingBeaconsInRegion(region);
31             } catch (RemoteException e) {
32                 e.printStackTrace();
33             }
34         }
35     }
```

Listing 4.10: Parte 1 classe ApplicationHelpEnvironmentNow.java

La callback `onCreate()` della classe `Application` è stata ridefinita per impostare il tipo di beacon da rilevare(Eddystone-UID) contenente lo specifico identificatore univoco(lo stesso annunciato dal Raspberry Pi). Inoltre istanziando l'oggetto di tipo `RegionBootstrap` viene attivato il sistema di rilevazione del beacon che permetterà di eseguire scansioni in background.

La callback `didDetermineStateForRegion(...)` viene utilizzata per avviare il processo di ranging che sostanzialmente oltre a identificare la presenza dei beacon, consente anche di accedere alle informazioni contenute nel pacchetto dati.

Successivamente la classe ridefinisce anche il metodo della libreria precedentemente citata che verrà invocato nel momento in cui il beacon richiesto è stato rilevato:

```
@Override
public void didRangeBeaconsInRegion(Collection<Beacon> beacons,
    Region region) {
     if(beacons.size() > 0) {
          for (Beacon beacon : beacons) {
               if (beacon.getServiceUuid() == 0xfeaa && beacon.
    getBeaconTypeCode() == 0x00) {
                    // This is a Eddystone-UID frame
                    Identifier detectedNamespaceId = beacon.getId1();
                    Identifier detectedInstanceId = beacon.getId2();
                    if(myBeaconNamespaceId.equals(detectedNamespaceId) &&
     myBeaconInstanceId.equals(detectedInstanceId)) {
                         String remoteMacAddress = beacon.
    getBluetoothAddress();
                         if(remoteMacAddress != null) {
                              Intent intent = new Intent(this,
    ClassicService.class);
                              intent.putExtra("remoteMacAddress",
    remoteMacAddress);
                              ContextCompat.startForegroundService(this,
    intent);
                              break;
                         }
                    }
               }
          }
     }
}
```

Listing 4.11: Parte 2 classe ApplicationHelpEnvironmentNow.java

Questa callback riceve come parametro una collezione di pacchetti che inglobano i beacon rilevati. Nel caso in cui uno dei beacon corrisponde a quello desiderato si estrae dal pacchetto dati, l'indirizzo MAC Bluetooth del dispositivo Raspberry Pi che ha annunciato tale pacchetto. L'ultimo compito di questo metodo, prima di terminare, è quello di avviare il servizio(passandogli l'indirizzo MAC appena ottenuto) che dovrà connettersi al dispositivo di monitoraggio per prelevare i dati ambientali.

4.2.6 Servizio ClassicService

La classe ClassicService estende la classe Android `IntentService`, definendo quindi un servizio che esegue i seguenti compiti:

- ottenimento della posizione corrente
- lettura delle rilevazioni ambientali dal Raspberry Pi
- memorizzazione delle rilevazioni nel database locale
- richiesta di caricamento dei dati sul server remoto

Il servizio ClassicService è di tipo `foreground` e quindi quando sarà in esecuzione una notifica comparirà nella barra delle notifiche per segnalarlo. Questa verrà rimossa in automatico non appena il servizio avrà terminato la sua esecuzione. Quando il servizio viene lanciato, la callback `onHandleIntent(...)` viene invocata ed esegue le operazioni necessarie per raggiungere gli obiettivi elencati. Il suo codice sorgente si presenta come segue:

```
1  @Override
2  protected void onHandleIntent(Intent intent) {
3
4      LocationInfo.getCurrentLocation(getApplicationContext(), this);
5      /* Wait until the device current location is returned. When
       location is ready, locationCompleted(...)
6       * is called and sets curLocationReady to true and so the while
       cycle will be interrupted
7       * and the field curLocation will contain latitude, longitude,
       altitude */
8      while(!curLocationReady) {
9          int WAIT_LOCATION_MS = 500;
10         SystemClock.sleep(WAIT_LOCATION_MS);
11     }
12
13     // Open the db connection. It will be used inside Raspberry Pi
       object
14     MyDb myDb = new MyDb(getApplicationContext());
15
16     /* the remote device is the the Raspberry Pi */
17     String remoteDeviceMacAddress = intent.getStringExtra("
       remoteMacAddress");
18     RaspberryPi rPi = new RaspberryPi();
19     long insertions = rPi.connectAndRead(remoteDeviceMacAddress,
       curLocation, myDb);
20     // Close the db connection as it is not used anymore
21     myDb.closeDb();
22     if(insertions > 0) {
```

```
23          /* I enqueue a work that the Worker Manager will execute when
        network became available */
24          MyWorkerManager.enqueueNetworkWorker(getApplicationContext())
        ;
25      }
26      Log.d("ClassicService", "Service Completed");
27  }
```

Listing 4.12: Callback onHandleIntent del servizio MainService

Dato che questo servizio viene lanciato in seguito alla rilevazione di un beacon si cerca di ottenere la posizione del dispositivo Android(latitudine, longitudine e altitudine) che sarà associata ai dati ambientali che verranno raccolti. È bene notare che l'altitudine rilevata è espressa in metri sopra l'ellissoide di riferimento WGS 84 e non sopra il livello del mare. Il codice più di basso livello che tenta di ottenere la posizione è contenuto nella classe LocationInfo. Il servizio si limita a invocarne il metodo statico `getCurrentLocation` e aspettare il risultato. Internamente al metodo `getCurrentLocation` vi è una chiamata asincrona ed è questa la ragione per qui viene utilizzato un ciclo while per attendere il suo completamento(che normalmente avviene entro un secondo). La variabile `curLocationReady` viene messa a `true` nel momento in cui la richiesta di posizione è stata completata e di conseguenza il ciclo while viene interrotto e si prosegue oltre.
La lettura delle rilevazioni ambientali e la loro memorizzazione nel dispositivo si realizza chiamando il metodo `connectAndRead(...)` della classe RaspberryPi, descritta più avanti.
Nel caso in cui sia stato possibile ricevere alcune rilevazioni ambientali, viene programmato un compito invocando `enqueueNetworkWorker(...)` che caricherà i dati raccolti sul server remoto quando la rete risulterà disponibile. Il servizio ClassicService, conclusa l'esecuzione di tutto il codice contenuto dentro `onHandleIntent`, termina e sarà nuovamente eseguito quando l'applicazione sarà risvegliata da un nuovo beacon.

4.2.7 Classe RaspberryPi

Nelle sezioni precedenti è stata citata la classe RaspberryPi che include il codice per gestire la comunicazione Bluetooth con un dispositivo RaspberryPi. Il metodo pubblico che può quindi essere chiamato dall'esterno è `connectAndRead` che, come suggerisce il nome, dà la possibilità di connettersi ad un dispositivo di monitoraggio Raspberry Pi(il cui indirizzo MAC Bluetooth deve essere fornito come parametro) e di leggere i dati ambientali raccolti.

```
1  public long connectAndRead(String remoteDeviceMacAddress, Location
       location, MyDb myDb) {
2
```

```
3       bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
4       if(bluetoothAdapter != null)
5           connectAndReadFromRaspberry(remoteDeviceMacAddress, location,
        myDb);
6
7       return totalInsertions;
8   }
```

Listing 4.13: Metodo connectAndRead della classe RaspberryPi.java

Il metodo ritorna il numero di rilevazioni lette dal Rasperry Pi ed è composto da poche righe di codice in quanto il blocco di codice che effettua la connessione e la lettura dei dati è incluso nel metodo privato `connectAndReadFromRaspberry`:

```
1   private void connectAndReadFromRaspberry(String
        remoteDeviceMacAddress, Location location, MyDb myDb) {
2
3       BluetoothSocket socket = getBluetoothSocketByReflection(
        remoteDeviceMacAddress);
4       if(socket != null) {
5           int attempt = 1;
6           if (bluetoothAdapter.isDiscovering())
7               bluetoothAdapter.cancelDiscovery();
8           while (attempt <= MAX_CONNECTION_ATTEMPTS && !socket.
        isConnected()) {
9               try {
10                  Log.d(TAG, "Socket connect() attempt:" + attempt);
11                  socket.connect();
12              } catch (IOException e) {
13                  Log.d(TAG, "Socket connect() failed!");
14                  e.printStackTrace();
15                  if (attempt < MAX_CONNECTION_ATTEMPTS)
16                      SystemClock.sleep(BLUETOOTH_MSECONDS_SLEEP); //
        sleep before retry to connect
17              }
18              attempt++;
19          }
20          if(socket.isConnected()) {
21              try {
22                  InputStream socketInputStream = socket.getInputStream
        ();
23                  OutputStream socketOutputStream = socket.
        getOutputStream();
24                  readMatchSaveChunks(socketInputStream,
        socketOutputStream, location, myDb);
25              } catch (IOException e) {
26                  Log.e(TAG, "socket IOException during
        readMatchSaveChunks");
27                  e.printStackTrace();
28              } finally {
```

```
29                  try {
30                      Log.d(TAG, "I close connected socket.");
31                      socket.close();
32                  } catch (IOException e) {
33                      e.printStackTrace();
34                  }
35              }
36          }
37      }
38  }
```

Listing 4.14: Metodo connectAndReadFromRaspberry della classe RaspberryPi.java

Come prima operazione viene creato l'oggetto BluetoothSocket ma non attraverso l'API standard di Android che richiederebbe l'utilizzo del servizio SDP(Service Discovery Protocol) attivo sul dispositivo remoto. Infatti SDP oltre ad essere stato sostituito dalla versione 5 di Bluez(e quindi non disponibile sulla versione di Raspbian utilizzata), impone di specificare l'UUID del servizio per poter risalire al canale RFCOMM del dispositivo remoto a cui connettersi. Per evitare l'uso di SDP e quindi per aprire una connessione diretta ad una porta RFCOMM specificando direttamente il numero del canale, è stato sfruttato il metodo `createInsecureRfcommSocket(int channel)`, che come descritto in [13] non è visibile nella documentazione pubblica ma fa parte del codice sorgente della piattaforma Android. Per richiamare quindi questo metodo è necessario fare uso della Reflection di Java per ottenere un suo riferimento a runtime, come indicato alla riga 7 del codice seguente:

```
1  private BluetoothSocket getBluetoothSocketByReflection(String
       remoteDeviceMacAddress) {
2      BluetoothDevice remoteDevice;
3      BluetoothSocket socket = null;
4      if(BluetoothAdapter.checkBluetoothAddress(remoteDeviceMacAddress)
       ) {
5          remoteDevice = bluetoothAdapter.getRemoteDevice(
       remoteDeviceMacAddress);
6          try {
7              socket = (BluetoothSocket) BluetoothDevice.class.
       getMethod("createInsecureRfcommSocket", int.class).invoke(
       remoteDevice, 1);
8          } catch (IllegalAccessException e) {
9              e.printStackTrace();
10         } catch (InvocationTargetException e) {
11             e.printStackTrace();
12         } catch (NoSuchMethodException e) {
13             e.printStackTrace();
14         }
15     }
```

```
16
17      return socket;
18  }
```

Listing 4.15: Meotdo privato getBluetoothSocketByReflection della classe RaspberryPi.java

Nonostante esista anche il metodo `createRfcommSocket(int channel)`, è stata scelta la versione `Insecure` in quanto permette il pairing Bluetooth automatico e una connessione cifrata, accettando però il fatto che il canale di comunicazione non avrà una chiave autenticata, e quindi potrebbe essere soggetto ad attacchi man-in-the-middle. L'alternativa più sicura(con chiave autenticata) sarebbe stata quella di richiedere ad ogni connessione con un Raspberry Pi, il consenso dell'utente, e questo renderebbe chiaramente molto meno efficace il funzionamento dell'applicazione in un contesto reale.

Chiarito il modo in cui viene creato il socket Bluetooth, il passo successivo consiste nell'aprire la connessione, ritentando più volte nel caso in cui non sia possibile connettersi al primo tentativo. Si procede con la lettura dei dati in maniera ciclica, leggendo ad ogni iterazione un intero oggetto JSON, salvando nel database locale tutte le misure ricevute(dopo aver associato loro la posizione geografica) e inviando conferma al Raspberry Pi; queste operazioni vengono eseguite dal metodo `readMatchSaveChunks(...)`. Il parsing del flusso di dati viene effettuato per mezzo della classe standard `JsonReader` che consente di estrarre le singole rilevazioni ambientali che vengono accumulate e inserite nel database Sqlite locale sfruttando la libreria `Room Persistence Library` per inserire in maniera efficiente tutti i dati in un'unica operazione.

4.2.8 WorkManager e classe UploadWorker

Dopo che il servizio ha terminato di ricevere dati dal dispositivo di monitoraggio, accoda una richiesta per l'WorkManager, che eseguirà il compito passato come parametro. A tal proposito ho definito una classe con un metodo wrapper che ha lo scopo di impostare le caratteristiche della richiesta. Oltre a specificare quando il task dovrà essere eseguito tramite il vincolo `NetworkType.CONNECTED`, ho impostato il criterio di backoff di tipo esponenziale, ovvero nel caso in cui il task non possa essere eseguito con successo la prima volta, verrà ritentato più volte aumentando in maniera esponenziale il tempo che intercorre tra una richiesta e la successiva. Nel caso in cui sia già stata passata al WorkManager una richiesta dello stesso tipo, questa sarà sostituita da quella nuova in quanto è sufficiente una sola richiesta per consentire il caricamento di tutti i record salvati fino a quel momento nel database. Qui viene mostrato il codice appena descritto:

```
public static void enqueueNetworkWorker(Context context) {
    final String WORK_TAG = "uploadSensorsData";
    // Create a Constraints object that defines when the task should run
    Constraints constraints = new Constraints.Builder().setRequiredNetworkType(NetworkType.CONNECTED).build();
    // A WorkRequest for non-repeating work
    OneTimeWorkRequest uploadWorkRequest = new OneTimeWorkRequest.Builder(UploadWorker.class).
            setBackoffCriteria(BackoffPolicy.EXPONENTIAL, 2, TimeUnit.MINUTES).
                    setConstraints(constraints).build();
    WorkManager.getInstance(context).enqueueUniqueWork(WORK_TAG, ExistingWorkPolicy.REPLACE, uploadWorkRequest);
}
```

Listing 4.16: Meotdo enqueueNetworkWorker della classe MyWorker.java

Nonostante ciò potranno ugualmente esistere due richieste per lo stesso tipo di compito nel sistema nel caso in cui la richiesta venga accodata mentre una sia già in esecuzione. Per evitare che i due compiti vengano eseguiti in concorrenza, ho configurato un solo thread per l'WorkManager, ridefinendo il seguente metodo nella classe ApplicationHelpEnvironmentNow:

```
@Override
public Configuration getWorkManagerConfiguration() {
    return new Configuration.Builder().setExecutor(Executors.newSingleThreadExecutor()).build();
}
```

Listing 4.17: Configurazione singolo thread per WorkManager

La classe UploadWorker estende la classe `Worker` e quindi definisce il compito che dovrà essere eseguito nel momento in cui saranno soddisfatte le condizioni imposte nella richiesta per l'WorkManager. Le classi che estendono Worker dispongono di una callback `doWork()` che verrà invocata, per eseguire il codice definito dallo sviluppatore. In questo caso il codice consiste in un ciclo che estrae dal database una parte dei record, le converte in formato JSON e le invia al server remoto, rimanendo in attesa dell'esito. Se tutto dovesse andare a buon fine i record inoltrati vengono rimossi dalla tabella e si prosegue con quelli rimanenti. Il metodo è illustrato di seguito:

```
@Override
public Result doWork() {
    Log.d(TAG, "doWork() called");
    boolean sendResult;
    if(Looper.myLooper() == null) {
        Log.d(TAG, "Looper prepare called");
```

```
7           Looper.prepare();
8       }
9       MyDb myDb = new MyDb(context);
10      JsonBuilder jsonBuilder = new JsonBuilder();
11      HeRestClient heRestClient = new HeRestClient(context);
12
13      long totMeasures = myDb.getTotalMeasures();
14      Log.d(TAG,"tot measures: " + totMeasures);
15
16      while(totMeasures > 0) {
17          List<Measure> measures = myDb.getSomeMeasures();
18          Log.d(TAG,"List size: " + measures.size());
19
20          JSONObject jsonObject = jsonBuilder.buildDataBlock(measures);
21          if(jsonObject == null) {
22              releaseResources(myDb);
23              return Result.retry();
24          }
25          sendResult = heRestClient.sendToServerWithResult(jsonObject.
    toString());
26          if(!sendResult) {
27              releaseResources(myDb);
28              return Result.retry();
29          }
30          myDb.deleteMeasures(measures);
31          totMeasures = myDb.getTotalMeasures();
32      }
33
34      releaseResources(myDb);
35      return Result.success();
36  }
```

Listing 4.18: Meotdo doWork della classe UploadWorker.java

Un esempio di formato JSON è mostrato qui sotto. Chiaramente nei casi reali, l'array `data` contiene un numero molto maggiore di record; qui ne sono presenti solo due a scopo illustrativo.

```
1  {
2      "data": [
3          {
4              "s": 20, "t": 1234567892, "d": 23.4, "g": "abcdefghijkl",
           "a": 302.5
5          },
6          {
7              "s": 21, "t": 1234567892, "d": 53.9, "g": "abcdefghijkl",
           "a": 302.5
8          }
9      ]
10 }
```

Listing 4.19: Esempio di formato JSON inviato al server

4.2.9 Classe HeRestClient

La classe HeRestClient contiene il codice che serve a inoltrare l'oggetto JSON al server remoto. Per semplificare la comunicazione con il servizio web di architettura REST ho sfruttato la libreria `Loopj`. Ho usato la classe `SynchHttpClient` che consente di inoltrare una richiesta HTTP e aspettare in maniera sincrona la risposta, prima di procedere con l'istruzione successiva. Il protocollo effettivamente utilizzato è HTTPS con autenticazione tramite username e password per accedere all'API del servizio web. Nel caso in cui la richiesta non abbia successo la prima volta, vengono effettuati fino a cinque tentativi; tale comportamento è fornito dalla libreria stessa. Nel caso in cui alla fine di tutti i tentativi la richiesta non abbia avuto esito positivo viene richiesto al WorkManager di ritentare l'esecuzione in un secondo momento. Di seguito viene mostrato il codice del metodo che inoltra l'oggetto JSON al web service:

```
1  public boolean sendToServerWithResult(final String dataBlock) {
2      AsyncHttpResponseHandler responseHandler = new
       AsyncHttpResponseHandler() {
3          @Override
4          public void onSuccess(int statusCode, Header[] headers, byte
       [] responseBody) {
5              Log.d("Client", "Worker PUT SUCCESS:"+statusCode);
6              sendResult = true;
7          }
8
9          @Override
10         public void onFailure(int statusCode, Header[] headers, byte
       [] responseBody, Throwable error) {
11             Log.d("Client", "Worker PUT FAIL:"+statusCode);
12             sendResult = false;
13         }
14     };
15     StringEntity entity = new StringEntity(dataBlock,
       StandardCharsets.UTF_8);
16     restClient.put(context, HE_WEB_SERVICE_URL, entity, "application/
       json", responseHandler);
17
18     return sendResult;
19 }
```

Listing 4.20: Metodo sendToServerWithResult della classe HeRestClient.java

4.3 Web service HelpEnvironment

Il servizio web HelpEnvironment di architettura REST è stato sviluppato in linguaggio Java(jdk e jre versione 1.8.0_181) utilizzando l'implementazione Jersey dell'API JAX-RS della piattaforma Java Enterprise Edition(Java EE). Il servizio web è stato testato localmente ed ospitato da un server Apache Tomcat v7.0. In questo caso Apache Tomcat è stato configurato in maniera da forzare l'utilizzo del protocollo HTTPS per poter accedere all'API del servizio. Il servizio web può essere depositato anche su altri server(container) che supportano Java EE ma dovranno essere configurati opportunamente per imporre l'utilizzo di HTTPS, autenticazione base tramite username e password ed altre eventuali politiche di sicurezza.

Il servizio web è stato sviluppato utilizzando l'ambiente di sviluppo Eclipse Jee. Le classi che lo costituiscono sono visualizzate di seguito:

Figura 4.11: Classi del servizio HelpEnvironment

Le classi DataBlock e Measure contengono i campi corrispondenti ai campi del formato JSON usato per la trasmissione dei dati ambientali. In questa maniera quando il container web Apache Tomcat riceverà una richiesta HTTPS, convertirà automaticamente il corpo della richiesta in un oggetto della classe DataBlock; questo processo di conversione dal documento JSON all'oggetto Java corrispondente è noto come unmarshalling.
La classe HelpEnvironmentResources dichiara e implementa i metodi corrispondenti alle richieste HTTP consentite dal servizio web mentre la classe HeService contiene il codice specifico che viene invocato per soddisfare la richiesta e quindi serve a separare l'implementazione interna del servizio dai metodi di interfaccia(l'API del servizio web).
Infine la classe MySqlDatabase è semplicemente una classe wrapper che consente di ottenere una connessione al database MySQL.

4.3.1 API del servizio web

L'API del servizio web è rappresentata dalla classe HelpEnvrionmentResources.java costituita da un solo metodo che consente di inserire i dati ambientali ricevuti nel

database. Il metodo `insertNewMeasures` viene invocato automaticamente quando il container web riceve una richiesta HTTPS di tipo PUT. L'URL della richiesta dovrà essere come segue: `https://<IP_ADDRESS>:<PORT>/HelpEnvironment/helpenvironment/he/measures`. La prima parte dell'URL identifica l'host su cui il servizio è attivo. La stringa `HelpEnvironment/` fa riferimento al modulo web che contiene il servizio web mentre `helpenvironment/` identifica in questo caso il servizio stesso. `he/` si riferisce alla risorsa principale(`root`) del servizio mentre `measures` identifica il metodo `insertNewMeasures`.
Ogni volta che il container web riceve una richiesta come descritto, istanzia un nuovo oggetto della classe HelpEnvrionmentResources su cui invoca il metodo `insertNewMeasures` passandogli come parametro un oggetto della classe DataBlock contenente tutti i dati ricevuti nel corpo del messaggio HTTPS. Il metodo si limita a chiamare `insertMeasures` dell'oggetto della classe HeService e nel caso in cui non si verificano eccezioni esegue `return` che porterà alla generazione di una risposta HTTP con codice 204 destinata al client per informarlo del successo dell'operazione.

```
1  @Path("/he")
2  public class HelpEnvrionmentResources {
3
4      private HeService heService = new HeService();
5
6      @PUT
7      @Path("/measures")
8      @Consumes({MediaType.APPLICATION_JSON})
9      public void insertNewMeasures(DataBlock data) {
10         heService.insertMeasures(data);
11
12         return; // HTTP 204 will be returned to the client if no
       exception in insertNewMeasures
13     }
14
15 }
```

Listing 4.21: Classe HelpEnvrionmentResources

4.3.2 Servizio HeService

Il servizio HeService fornisce un solo metodo che estrae le misure dall'oggetto della classe DataBlock per inserirle in una tabella del database. La tabella ha la seguente struttura:

- id(chiave primaria di tipo intero che si auto-incrementa)
- id logico del sensore

- timestamp
- misura
- latitudine e longitudine codificati in una stringa geohash
- altitudine

La base di dati e la struttura delle tabelle(tra cui anche quella appena descritta) erano già state definite e generate dal gruppo di ricerca mentre io ho solo prodotto il codice Java per consentire al servizio web di accedere al database e inserire i dati. Siccome la quantità di record da inserire normalmente è elevata, essi non vengono inseriti uno per uno ma in blocchi, attraverso l'uso di `prepared statement`, per ottenere prestazioni migliori.

Capitolo 5

Estensione per rilevamenti mobili

Sfruttando le tecnologie e i concetti del sistema di monitoraggio visto nei capitoli precedenti, è stata sviluppata una versione differente del sistema, pensata per la raccolta di dati in movimento. Questa estensione richiede all'utente di possedere un proprio dispositivo di monitoraggio Raspberry Pi, con un software leggermente differente e una diversa versione dell'applicazione Android che consente di ottenere rilevazioni ambientali mentre ci si sposta insieme al Raspberry Pi. Infatti la versione partecipativa del sistema presentato finora non richiedeva all'utente di avere il controllo sui dispositivi di monitoraggio, che erano invece pensati per essere installati in dei punti fissi del territorio monitorato.
Questa versione è invece pensata sia per raccogliere dati in maniera statica, quando ad esempio il Raspberry Pi è fermo a casa dell'utente, sia in maniera dinamica quando il Raspberry Pi si sposta insieme allo smartphone dell'utente. In particolare la modalità dinamica deve essere attivata volontariamente dall'utente ogniqualvolta il Raspberry Pi venga spostato in maniera significativa.
Il Raspberry Pi può quindi lavorare in modalità statica oppure in modalità dinamica. Se il dispositivo è in modalità statica ci rimane fino a quando un client Android non si connette ad esso in modalità dinamica; appena la connessione viene stabilita, il Raspberry Pi passa in modalità dinamica. Nel momento in cui la connessione viene chiusa il client invia l'ultima posizione geografica al Raspberry Pi che tornerà in modalità statica e fino alla successiva connessione con un client, tutte le rilevazioni ambientali saranno associate alla posizione ricevuta alla chiusura della connessione. Quindi in modalità statica il Raspberry Pi associa già una posizione geografica alle rilevazioni ambientali; in modalità dinamica invece il Raspberry Pi inoltra in tempo reale le rilevazioni al client connesso che avrà il compito di associare a ciascuna rilevazione la corrispondente posizione geografica.

5.1 Panoramica dell'implementazione sul Raspberry Pi

In questa nuova versione, durante il suo funzionamento, il Raspberry Pi accumula le rilevazioni ambientali e quindi le salva nei file in formato CSV, aggiungendo un campo che serve ad indicare se la rilevazione è avvenuta durante la modalità statica o durante la modalità dinamica. Quindi a differenza della versione partecipativa dei capitoli precedenti, in questa versione sul Raspberry Pi sono presenti due server in ascolto su due canali distinti. Entrambi i server sono stati sviluppati in linguaggio Python. Un primo server `bt_static.py` ha il compito di inoltrare a un client i dati che sono stati accumulati in modalità statica mentre un secondo server `bt_dynamic.py` ha il compito di stabilire la connessione con un client, cambiare la modalità del Raspberry Pi da statica a dinamica e gestire la comunicazione con il client inviandogli le misure in tempo reale ogniqualvolta il client lo richieda; infine quando il client chiude la connessione, il server cambia la modalità del Raspberry Pi da dinamica a statica aggiornando la posizione geografica. La modalità di funzionamento del Raspberry Pi viene registrata all'interno di un file `pi_mode.csv` che viene scritto dal programma server `bt_dynamic.py` ogni volta che è richiesto un cambio di modalità generato da una richiesta di connessione al server stesso da parte di un client. I servizi(`dht.py` e `sensing.py`) che hanno invece il compito di accumulare i dati ambientali all'interno di file CSV, leggono lo stesso file ogni volta che vengono lanciati per stabilire se attribuire il valore `statico` o `dinamico` alle rilevazioni successive. Infatti ogni volta che la modalità del Raspberry Pi viene cambiata, i servizi precedentemente citati vengono fermati e lanciati nuovamente. Riassumendo, il server `bt_static.py` è molto simile nella sua struttura e nel suo funzionamento al server Bluetooth descritto nei capitoli precedenti mentre il server `bt_dynamic.py` oltre a gestire una comunicazione continua con il client, ha anche il compito di modificare e gestire la modalità di funzionamento del Raspberry Pi scrivendo il file e riavviando i servizi `dht.py` e `sensing.py`.

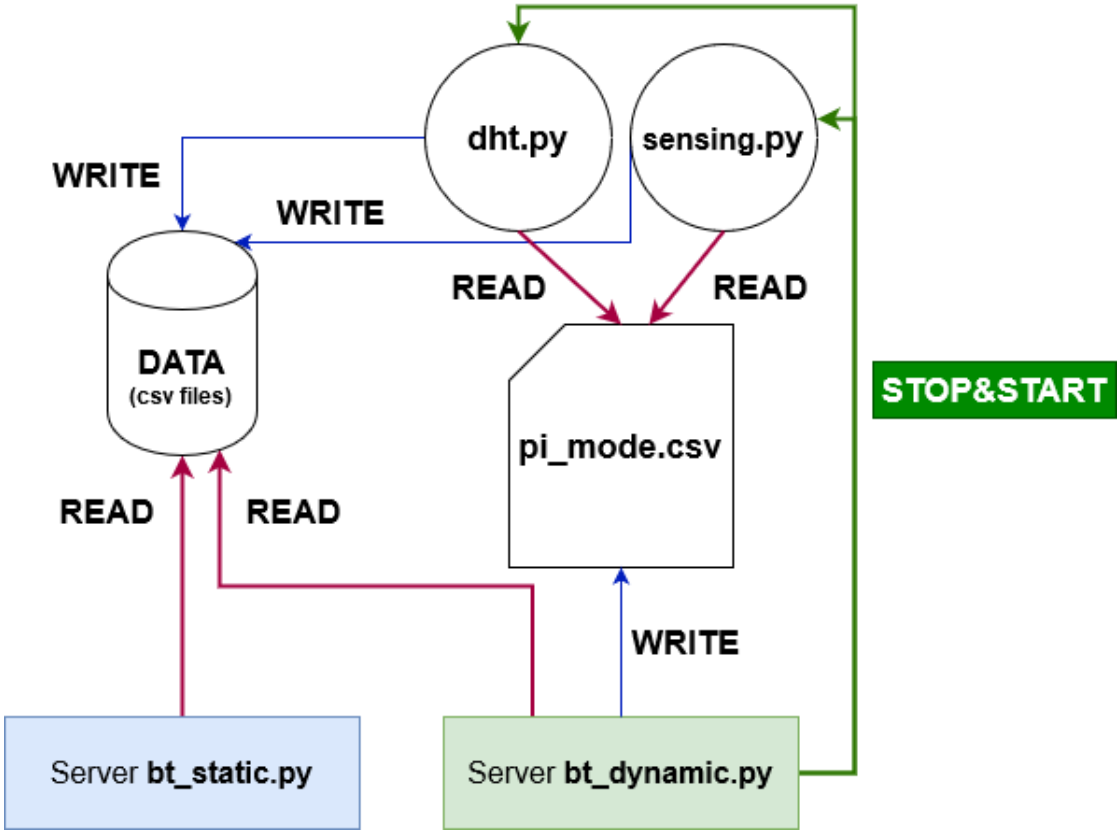

Figura 5.1: Panoramica versione mobilità Raspberry Pi

5.1.1 Server per la modalità dinamica

Il server per la modalità dinamica(implementato nello script `bt_dynamic.py`) dopo aver stabilito la connessione con un client e aver cambiato la modalità del Raspberry Pi, mantiene aperta la connessione fino a quando il client non richiede la terminazione della connessione. Quindi è il client Android ad avere il controllo sulla durata della connessione. Il compito del server è quello di rimanere in attesa di comandi da parte del client e di eseguirli. In particolare sono previsti due possibili tipi di comandi inviati dal client al server:

- GET: per richiedere le ultime rilevazioni che non siano già state richieste in precedenza e in ogni caso rilevate dopo aver stabilito la connessione
- SET: per comunicare al server l'ultima posizione geografica e richiedere conferma per la chiusura della connessione

I comandi consistono in due oggetti in formato JSON.

Una tipica comunicazione è composta da una serie di richieste `GET` con una periodicità di qualche decina di secondi e una richiesta `SET` in seguito alla quale viene cambiata la modalità del Raspberry Pi. Infine il server invia conferma al client e la connessione viene chiusa.

5.2 Implementazione dell'app

L'app Android per la raccolta di dati in mobilità è differente rispetto a quella per il sistema partecipativo e questo è dovuto al fatto che l'utente deve avere la possibilità di eseguire una scansione e selezionare il Raspberry Pi a cui connettersi. Nelle seguenti immagini viene mostrato un tipico processo di connessione.

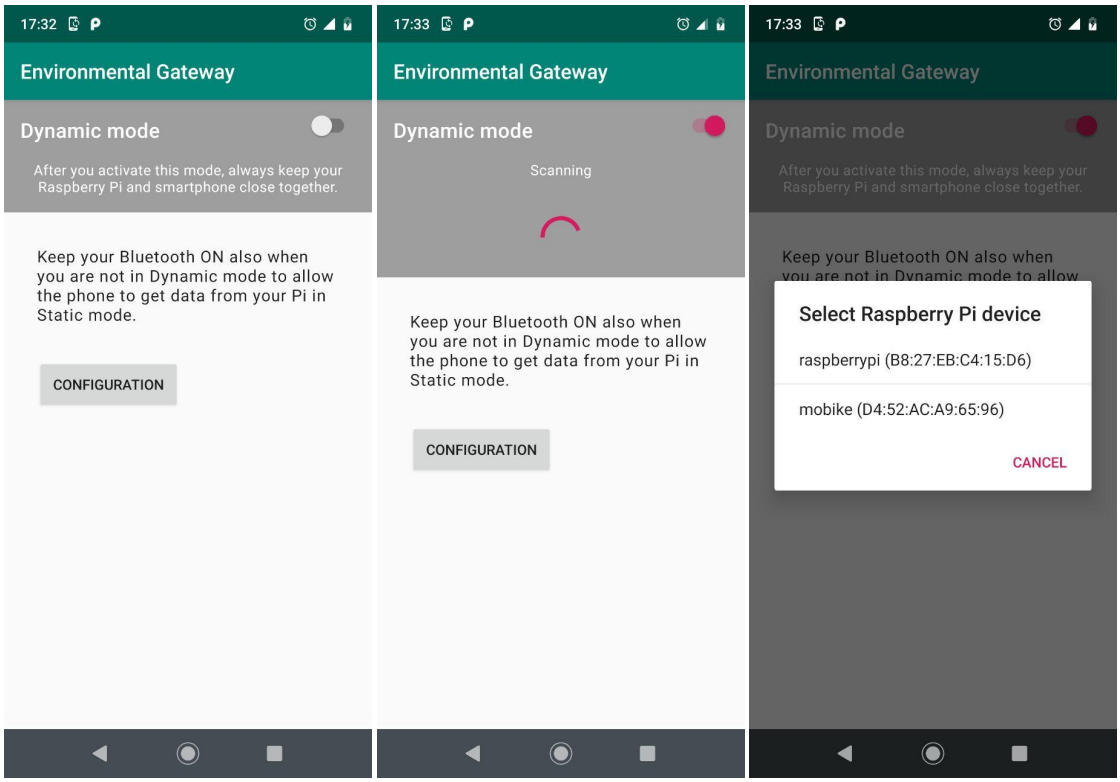

Figura 5.2: App versione mobilità - scansione

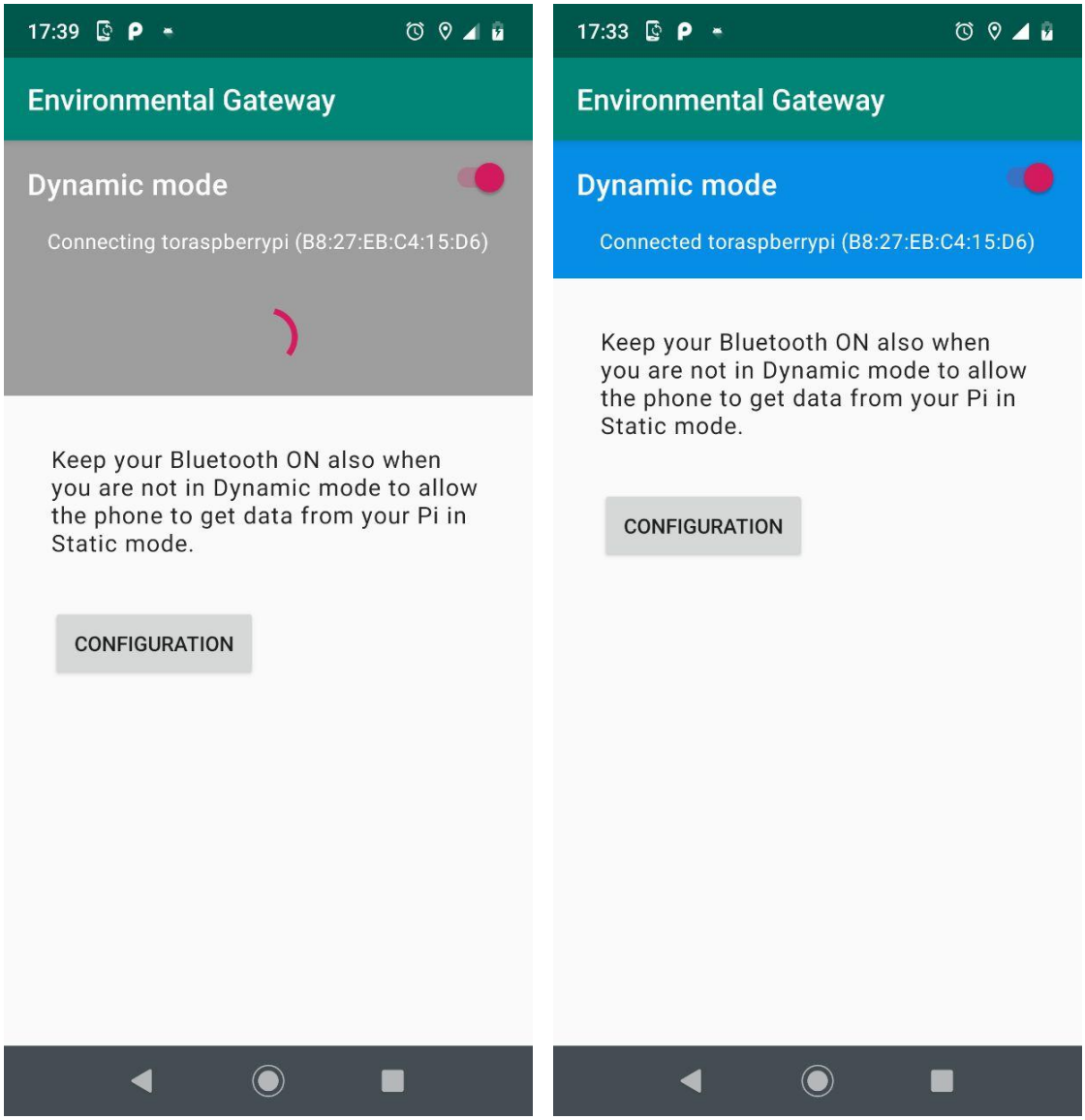

Figura 5.3: App versione mobilità - connessione

Dal momento in cui l'utente seleziona il Raspberry Pi a cui connettersi, viene lanciato un servizio Android che ogni 20 secondi(valore facilmente modificabile da codice) invia un comando `GET` al server e legge quindi le rilevazioni ambientali. Nel momento in cui l'utente decide di chiudere la connessione con il client, il servizio ottiene l'ultima posizione geografica del dispositivo e la comunica al Raspberry Pi con il comando `SET`. Il servizio attende conferma e termina.
L'app contiene anche un altro servizio, molto simile nella sua struttura e nel suo funzionamento a quello descritto per la versione partecipativa, che viene avviato

in automatico nel momento in cui viene rilevato un beacon e che ha lo scopo di leggere le rilevazioni ottenute in modalità statica dal dispositivo di monitoraggio.

Capitolo 6

Testing e valutazione

Questo capitolo descrive i test effettuati a sistema completato e alcune valutazioni. In sintesi, i risultati riportati confermano, come era atteso, che il comportamento dell'applicazione HelpEnvironmentNow è in parte influenzato dalla versione del sistema operativo e dal dispositivo Android, inteso come marca e modello. In particolare il tempo di rilevamento dei beacon e il continuo funzionamento dell'app in background sono stati i due aspetti analizzati tramite prove sperimentali.
Infine l'ultima sezione indica una stima della quantità di bytes trasferiti dall'applicazione per mezzo della connessione dati oppure Wi-Fi.

6.1 Dispositivi utilizzati per il test

Per eseguire i test sono stati utilizzati i seguenti smartphone:

	Dispositivo 1	**Dispositivo 2**	**Dispositivo 3**
Marca	Motorola	Samsung	Samsung
Modello	Moto G7 Plus	Galaxy A5 SM-A500FU	Galaxy J5(2016) SM-J510FN
Versione Android	9	6.0.1	7.1.1
Livello API	API 28	API 23	API 25
Batteria	3000 mAh	2300 mAh	3100 mAh

Table 6.1: Smartphone Android utilizzati

6.2 Tempo di rilevamento dei beacon

La documentazione della libreria Android Beacon Library annunciava già in partenza differenze per quanto riguarda il tempo di rilevamento dei beacon in base alle differenti versioni del sistema operativo. I test da me effettuati confermano queste differenze con l'aggiunta di qualche nota particolare.
I dispositivi 2 e 3 impiegano normalmente meno di trenta secondi ad individuare un nuovo beacon mentre sono necessari circa cinque minuti per rilevare il secondo beacon proveniente dallo stesso Raspberry Pi rimanendo nel suo campo d'azione. Dai primi test il dispositivo 1 era invece in grado di rilevare il beacon solo con la scansione software e quindi in media il tempo di rilevamento era di otto minuti, non riuscendo a sfruttare le scansioni filtrate gestite direttamente dal chipset Bluetooth.

	Dispositivo 1	**Dispositivo 2**	**Dispositivo 3**
Tempo primo rilevamento	mediamente 8 min	<= 30 s	<= 30 s
Tempo secondo rilevamento	circa 15 min	circa 5 min	circa 5 min

Table 6.2: Tempi di rilevamento da parte dello smartphone

Dopo aver effettuato varie ricerche, ho individuato che la fonte del problema è riconducibile alle impostazioni utente relative alle scansioni Bluetooth, come indicato dalla fonte ufficiale [29]. In particolare per consentire la scansione Bluetooth a livello di sistema in background(scansioni hardware filtrate) è richiesto che sia attiva la voce `Bluetooth scanning`. Nonostante alcuni dispositivi o versioni di Android siano in grado di sfruttare le scansioni filtrate anche se l'opzione è disattivata, per altre è necessario attivare l'opzione `Bluetooth scanning`; per attivare l'opzione è necessario andare in `Settings > Security and Location > Location > Scanning`. In questo modo l'applicazione sarà in grado di rilevare la comparsa di un nuovo beacon in background entro pochi secondi senza dover aspettare la scansione software periodica.

6.3 Funzionamento dell'app in background

Per verificare che l'applicazione HelpEnvironmentNow funzioni autonomamente in background per lungo tempo, ho mantenuto il Bluetooth attivo sui dispositivi a mia disposizione per alcuni giorni, accendendo di tanto in tanto il Raspberry Pi e verificando l'arrivo dei dati sul database in esecuzione sul PC locale. Le verifiche

da me effettuate hanno confermato il funzionamento dell'app nel tempo.

La libreria Android Beacon Library fornisce inoltre dei meccanismi per garantire il rilevamento dei beacon in background anche in seguito ad arresti dell'app oppure dopo il riavvio del dispositivo.
Terminando l'applicazione tramite l'app switcher, sui dispositivi 2 e 3 l'app viene rilanciata in background entro pochi secondi mentre sul dispositivo 1 l'app riparte dopo qualche minuto. L'unica azione che non le permette di ripartire è quando l'utente esegue un arresto forzato tramite `Impostazioni > Applicazioni` premendo su `Arresto forzato`. In tal caso l'applicazione ripartirà solo su esplicita richiesta dell'utente, avviandola tramite l'apposita icona del launcher.
Riavviando il dispositivo, lasciando il Bluetooth attivo prima di spegnere il dispositivo, su tutti i dispositivi testati l'applicazione è ripartita in automatico.

6.4 Quantità di dati trasferiti

Dopo che un dispositivo Android ha raccolto i dati da un dispositivo di monitoraggio, li invia in un formato JSON al server remoto. Un esempio di tale formato è stato visualizzato precedentemente(sottosezione `WorkManager e classe UploadWorker`). L'oggetto JSON viene aggiunto come contenuto di un messaggio https indirizzato al server.

Facendo uso del software Wireshark ho eseguito delle catture di rete per effettuare delle stime della quantità di dati trasmessi dall'applicazione Android al server.
Prima e dopo l'inoltro del messaggio http, viene stabilita e chiusa la connessione TCP ed effettuata la negoziazione TLS(handshake). Questo comporta uno scambio di messaggi per una dimensione complessiva di circa 3 KB; il valore può variare leggermente in base alla configurazione di sicurezza utilizzata.
L'oggetto JSON che viene trasmesso ha una dimensione che dipende dal numero di rilevazioni `N` che il dispositivo Android invia al server in una sola connessione e che attualmente può essere al massimo `30000`. La dimensione è pertanto calcolabile tramite la seguente formula:

$$dimensioneJSON = [N * 65]byte \tag{6.1}$$

nella quale 65 indica la dimensione media di una rilevazione ambientale in formato JSON.
Bisogna infine considerare che i dati trasmessi vengono inviati in pacchetti IP differenti, normalmente dalla dimensione massima di 1500 byte ciascuno di cui 40 byte di header IP e TCP, e i restanti 1460 byte di dato.

Tenendo conto delle dimensioni precedenti, per dare un'idea della quantità di dati che l'app Android potrebbe inoltrare, viene di seguito calcolata la quantità di byte trasferiti corrispondenti a 30000 rilevazioni inviate con una sola connessione al server:

$$dimensioneJSON_{max} = 30000 * 65 = 1950000byte \tag{6.2}$$

Per inviare 1950000 byte sono necessari 1336 pacchetti IP per una dimensione totale di circa 2000000 byte. A questi bisogna aggiungere circa 3 KB per l'apertura e la chiusura della connessione, circa 1 KB per gli header HTTP e qualche decina di KB per i messaggi TCP ACK. La dimensione totale risulta quindi essere di circa 2MB.

Allo stato attuale un Raspberry Pi esegue al massimo una lettura di temperatura ed umidità ogni due secondi e quattro letture di particolato atmosferico ogni secondo. In un'ora può accumulare al massimo 3600 rilevazioni di temperatura ed umidità e 28800 rilevazioni di particolato atmosferico. Supponendo ad esempio che un tipico dispositivo Android partecipante raccolga in media circa tre ore di rilevazioni al giorno dai vari dispositivi di monitoraggio distribuiti in una città, e approssimando per eccesso, invierà circa 7 MB, alla luce dei calcoli precedenti. Questo valore stimato può chiaramente variare notevolmente sia in positivo che in negativo in base al singolo caso.

6.5 Consumo della batteria

Per valutare il consumo della batteria dello smartphone da parte dell'app, è stata eseguita una prova della durata di sei ore, durante la quale i tre smartphone testati sono rimasti nel raggio d'azione del Raspberry Pi, con il quale hanno avuto connessioni frequenti. Per tutta la durata gli smartphone hanno avuto il Bluetooth attivo e sono stati connessi alla rete Wi-Fi per mezzo della quale i dati sono stati inoltrati al web service. Il consumo complessivo da attribuire all'app è composto dalle seguenti tre componenti:

- consumo dovuto all'utilizzo del sistema da parte dell'app
- consumo dovuto all'utilizzo del Bluetooth
- consumo dovuto all'utilizzo della rete dati o Wi-Fi

Nel seguente grafico viene mostrato il consumo complessivo rilevato per ciascun dispositivo, dovuto solo alle prime due componenti, mentre la terza componente risulta difficile da stimare in quanto la rete dati o Wi-Fi sono utilizzate da tanti altri servizi e applicazioni.

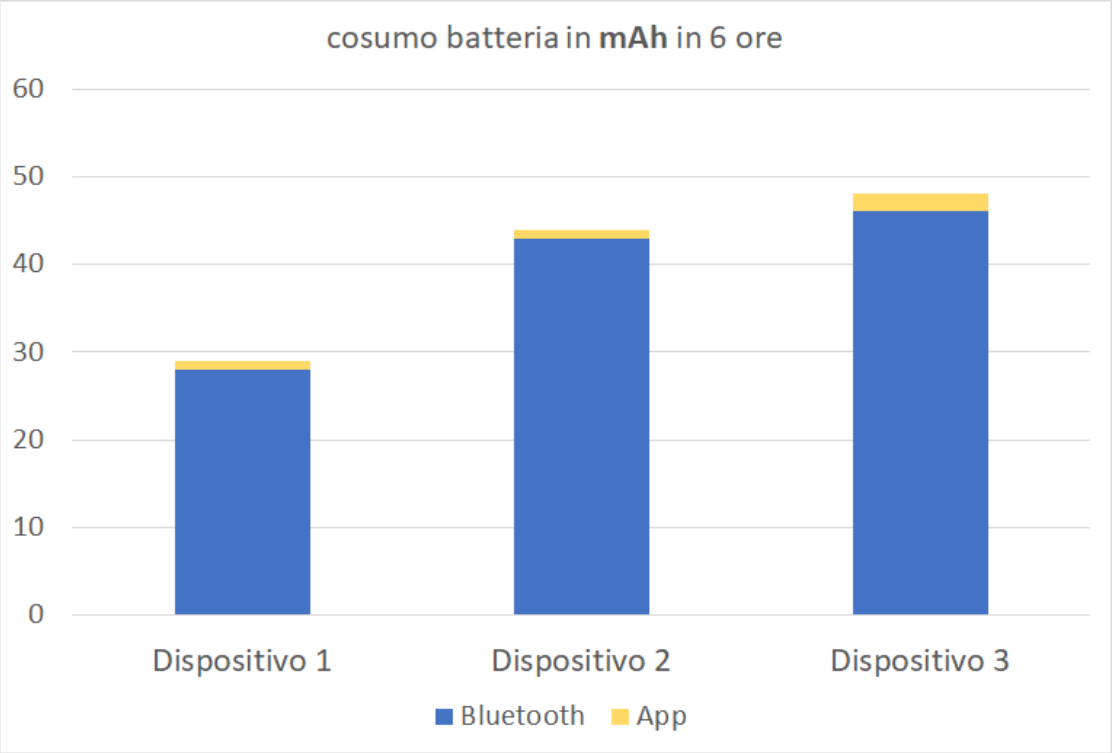

Figura 6.1: Consumo della batteria in mAh

In tutti e tre i casi è possibile osservare che la maggior parte del consumo è da attribuire alle frequenti connessioni Bluetooth che si sono verificate durante il test. In un contesto reale, la probabilità che uno smartphone esegua dei trasferimenti Bluetooth così frequenti è bassa, pertanto anche il conseguente consumo energetico sarebbe inferiore ai valori ottenuti.
Tenendo conto della capacità della batteria di ciascun smartphone(presente nella sezione 6.1), il seguente grafico mostra i mAh sottratti al totale disponibile, in percentuale.

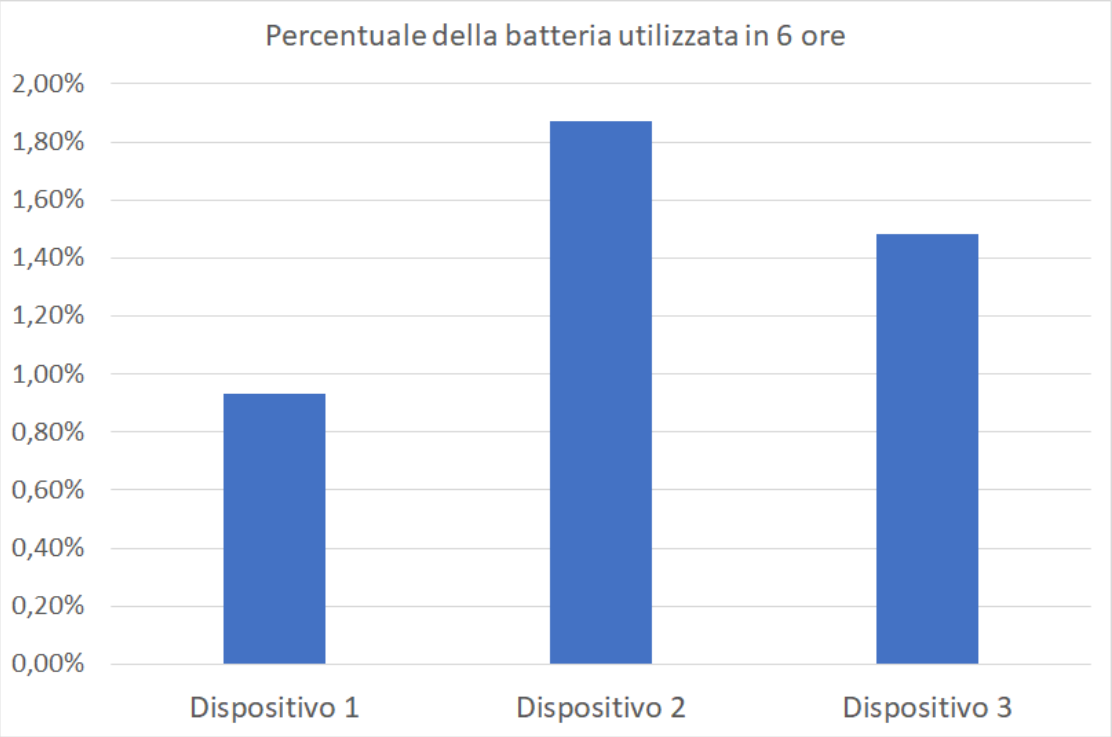

Figura 6.2: Consumo della batteria in percentuale

Conclusioni

Negli ultimi anni è aumentato sempre di più l'interesse per lo studio e l'osservazione delle grandezze fisiche legate all'ambiente e all'inquinamento ambientale. In particolar modo in contesti cittadini la qualità dell'aria risulta essere un fattore rilevante in quanto strettamente legato alle condizioni di salute dei cittadini. Il livello della qualità dell'aria è soggetto a regolamentazione ed è tenuto sotto osservazione dalle autorità che impongono restrizioni per la circolazione dei veicoli quando viene superato un valore limite. Nonostante le stazioni di monitoraggio forniscano rilevazioni ambientali accurate, esse sono situate solo in alcuni punti del territorio e quindi risulta difficile monitorare tutte le aree in modo pervasivo. Diventa quindi sempre più importante raccogliere dati ambientali con elevata frequenza e in maniera quanto più possibile capillare.

Il sistema di monitoraggio ambientale partecipativo sviluppato in questa tesi rappresenta un modo efficace per raccogliere informazioni da un insieme di dispositivi di monitoraggio senza connessione a Internet distribuiti densamente all'interno di un'area più o meno estesa. I dispositivi mobili con sistema operativo Android sono molto diffusi ai giorni nostri tra i cittadini e quindi possono essere sfruttati, con il consenso degli utenti, come gateway per i dispositivi di monitoraggio in maniera da centralizzare le informazioni raccolte su una base di dati remota accessibile tramite la rete Internet, a cui gli smartphone Android hanno accesso. L'applicazione HelpEnvironmentNow, in grado di funzionare autonomamente, è il nodo centrale del sistema e risulta essere poco dispendiosa sia in termini di consumo della batteria che in termini di consumo del traffico dati. Inoltre anche gli utenti che non dispongono di un piano dati possono utilizzare il proprio dispositivo per raccogliere informazioni che saranno trasferite sul database remoto non appena il dispositivo sarà connesso a una rete Wi-Fi.

Il sistema sviluppato si concentra sulla raccolta di dati ambientali ma le tecniche utilizzate, come l'annuncio di beacon Bluetooth Low Energy per segnalare la presenza dei dispositivi di monitoraggio e lo scambio locale dei dati automatizzato tramite Bluetooth, potrebbero essere sfruttate anche in altri contesti modificando

la parte di codice sorgente relativa ai dati che si vogliono trasmettere.

Nell'ipotesi che il sistema venga utilizzato in un caso reale, sarà necessario configurare opportunamente il server che raccoglie le informazioni ed eventualmente modificare il processo di autenticazione dei client in modo da garantire un adeguato livello di sicurezza.

Un possibile sviluppo futuro dell'applicazione partecipativa è legata alla visualizzazione della situazione ambientale sotto forma di grafici e statistiche in una determinata area monitorata, su richiesta dell'utente. In questo modo l'app oltre ad avere la funzione di raccolta dati, darà anche la possibilità all'utente di ottenere informazioni utili sull'ambiente circostante.

Bibliografia

[1] *Monitoraggio ambientale*. URL: https://www.higeco.com/monitoraggio-ambientale/.

[2] *Il progetto R-MAP, rete di monitoraggio ambientale partecipativo*. URL: https://www.arpae.it/cms3/documenti/_cerca_doc/ecoscienza/ecoscienza2015_6/patruno_desavino_es2015_6.pdf.

[3] *RaspberryPi*. URL: https://it.wikipedia.org/wiki/Raspberry_Pi.

[4] *Raspberry Pi 3 Model B*. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[5] *Bluetooth*. URL: https://it.wikipedia.org/wiki/Bluetooth.

[6] *The history of the Bluetooth SIG*. URL: https://www.bluetooth.com/about-us/our-history/.

[7] *Understanding Bluetooth Range*. URL: https://www.bluetooth.com/bluetooth-technology/range/.

[8] *Radio Versions*. URL: https://www.bluetooth.com/bluetooth-technology/radio-versions/.

[9] *The pros and cons of Bluetooth Low Energy*. URL: https://www.electronicsweekly.com/news/design/communications/pros-cons-bluetooth-low-energy-2014-10/.

[10] *The IoT for Everyone. Everywhere.* URL: https://www.bluetooth.com/markets/phone-pc/.

[11] *List of Bluetooth protocols*. URL: https://en.wikipedia.org/wiki/List_of_Bluetooth_protocols#Radio_frequency_communication_(RFCOMM).

[12] *RFCOMM with TS 07.10*. URL: http://www.dcs.ed.ac.uk/home/slipb/Bluetooth%20VCC%20Models/Documents/bluetooth_f1.pdf.

[13] Fabio Collini, Matteo Bonifazi, Alessandro Martellucci, and Stefano Sanna. *Android | Programmazione avanzata*. 2nd ed. LSWR, 2015. ISBN: 978-88-6895-072-9.

[14] *Eddystone(Google).* URL: https://en.wikipedia.org/wiki/Eddystone_(Google).

[15] *Eddystone.* URL: https://github.com/google/eddystone.

[16] *Android è per tutti.* URL: https://www.android.com/intl/it_it/everyone/.

[17] *Mobile Operating System Market Share Worldwide.* URL: https://gs.statcounter.com/os-market-share/mobile/worldwide.

[18] *What is API Level?* URL: https://developer.android.com/guide/topics/manifest/uses-sdk-element.

[19] *Distribution dashboard.* URL: https://developer.android.com/about/dashboards.

[20] *Application Fundamentals.* URL: https://developer.android.com/guide/components/fundamentals.

[21] *Services overview.* URL: https://developer.android.com/guide/components/services.

[22] *Save data in a local database using Room.* URL: https://developer.android.com/training/data-storage/room.

[23] *Schedule tasks with WorkManager.* URL: https://developer.android.com/topic/libraries/architecture/workmanager.

[24] *Android Beacon Library.* URL: https://altbeacon.github.io/android-beacon-library/index.html.

[25] *Android Asynchronous Http Client.* URL: https://android-async-http.github.io/android-async-http/.

[26] *Meet Android Studio.* URL: https://developer.android.com/studio/intro.

[27] *Tutorial MySQL per principianti.* URL: https://www.ionos.it/digitalguide/server/know-how/imparare-a-utilizzare-mysql-guida-per-principianti/.

[28] *Does Bluetooth drain smartphone battery?* URL: https://senion.com/insights/bluetooth-drain-battery/.

[29] *Bluetooth Low Energy.* URL: https://source.android.com/devices/bluetooth/ble.