

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura
Laurea in Ingegneria Elettronica e delle Telecomunicazioni

Gestione di messaggi MQTT multi-topic per applicazioni smart-city con dispositivi Raspberry

Tesi in
Software per le Telecomunicazioni e Laboratorio T-1

Studente:
Andrea Castronovo

Relatore:
Prof. Daniele Tarchi

Correlatore:
Prof.ssa Carla Raffaelli

Sessione di Dicembre
Anno accademico 2020–2021

A te che hai sempre creduto in me,
a te che mi hai dato tanti valori.
Oggi saresti orgogliosa del mio traguardo.

A mia nonna Caterina.

Abstract

Grazie all'impiego diffuso delle nuove tecnologie della comunicazione e mobilità, si cerca di sviluppare un insieme di strategie urbanistiche tese all'ottimizzazione dei servizi pubblici così da mettere in relazione le infrastrutture con il capitale umano e intellettuale di chi le abita. In un mondo in cui le moderne tecnologie prendono sempre più spazio è naturale pensare che queste siano sempre più presenti nei luoghi in cui trascorriamo la nostra esistenza. Nasce così il concetto di "smart-city" (città intelligente) introdotto in questo contesto come un dispositivo strategico utile a sottolineare la crescente importanza delle tecnologie dell'informazione e della comunicazione, ottimizzando le soluzioni per la mobilità e la sicurezza. Una delle tante tecnologie utilizzate per le smart-cities e che ha avuto una crescita esponenziale nell'ultimo decennio è l'Internet of Things (Internet delle cose), con esso si definisce un insieme di dispositivi tutti interconnessi tra loro con l'intento di scambiarsi dati attraverso Internet.

Con le precedenti premesse si cercano protocolli in grado di ottimizzare l'avvento di tutte queste tecnologie, è stato così introdotto nel '99 il protocollo MQTT in grado di gestire messaggi leggeri in ambienti dove si richiede basso impatto e banda limitata; circostanze caratteristiche delle smart-city e dell'IoT.

Il progetto di tesi consiste nello sviluppo di un'architettura Publisher-Subscriber ideata per utenti deboli veicolari, con l'intento di migliorare la sicurezza stradale. Nella trattazione verranno spiegati lo scenario applicativo e le tecnologie utilizzate per la realizzazione di questo progetto, quindi Android per la realizzazione della parte Publisher con l'acquisizione dei dati da sensori e invio tramite protocollo MQTT al broker centrale, quest'ultimo realizzato con un Raspberry Pi che ha l'obiettivo di gestire i messaggi in arrivo e ripartenza.

Introduzione

Motivazioni e obiettivi

L'aumento esponenziale dell'uso di nuove tecnologie della comunicazione, quindi l'accrescimento dell'importanza delle tecnologie nella vita quotidiana di ogni essere umano, crea la necessità di ottimizzare soluzioni per la mobilità e la sicurezza.

L'indipendenza energetica e la connettività ad una rete costituita da oggetti sono due delle più importanti richieste da soddisfare per poter rimanere al passo coi tempi rendendo queste tecnologie applicabili al contesto in cui devono lavorare.

Per questi motivi si è voluto progettare un architettura che soddisfacendo le richieste energetiche e di connettività, si ponga l'intento di ottimizzare la sicurezza stradale utilizzando tecnologie che siano in grado di stare al passo con l'evoluzione tecnologica cercando di adattarsi a un ambiente così innovativo come quello delle città-intelligenti.

Metodi e strumenti

Per raggiungere lo scopo finale si utilizza un Raspberry con funzione di nodo centrale dell'architettura (broker) per la gestione dei messaggi inviati/ricevuti, il suo obbiettivo è quindi quello di raccogliere, coordinare e indirizzare i dati che attraversano questa architettura con lo scopo di interconnettere tutti i dispositivi mobili fra loro. Si utilizza inoltre un device Android per la raccolta dei dati ambientali (pressione, umidità, temperatura, ecc...) e umani (contapassi, cardiofrequenzimetro) con l'obbiettivo che vengano inviati tramite protocollo MQTT al nodo centrale. Grazie a questo protocollo si è reso possibile il passaggio dei dati acquisiti dai sensori al broker centrale, nello specifico è stato possibile ricevere e leggere sul Raspberry i dati elaborati dall'app Android potendosi iscrivere ai diversi topic.

Organizzazione

La tesi è stata organizzata nel seguente modo:

- Nel primo capitolo si vuole descrivere lo scenario in cui il progetto di tesi è stato sviluppato, quindi si vuole dare una descrizione generale dell'ambiente in cui si espande l'architettura illustrando il concetto delle smart-cities e descrivendo una delle tecnologie più usate al suo interno: Internet of Things.
- Nel secondo capitolo si parla delle caratteristiche generali e peculiarità introducendo le tecnologie utilizzate per la realizzazione del progetto quali: l'utilizzo di Raspberry come broker centrale per l'indirizzamento dei messaggi, Android per lo sviluppo dell'applicazione utile al raccoglimento e invio dei dati. Si parlerà inoltre del protocollo MQTT utilizzato dalle due tecnologie.
- Nel terzo capitolo ci si sofferma sulla parte di progetto riguardante Android, quindi sullo sviluppo dell'applicazione per la raccolta dei dati dai sensori volti alla loro visualizzazione e invio tramite protocollo MQTT al broker centrale.
- Nel quarto capitolo ci si sofferma nella parte di progetto riguardante Raspberry, quindi si descrive la configurazione come broker centrale e si parla della recezione dei messaggi dall'applicazione Android.

Indice

Abstract	v
Introduzione	vii
1 Scenario Applicativo	1
1.1 Smart-City	1
1.2 Internet of Things	4
1.2.1 Descrizione di IoT	4
1.2.2 Architettura IoT	5
1.2.3 L'uso di MQTT nell'IoT	7
2 Tecnologie utilizzate	9
2.1 Android	9
2.1.1 Architettura della piattaforma	9
2.1.2 Ambiente di sviluppo: Android Studio	11
2.1.3 Motivazioni	13
2.2 Raspberry Pi	13
2.2.1 Hardware	13
2.2.2 Software	14
2.2.3 Motivazioni	14
2.3 MQTT	15
2.3.1 Introduzione	15
2.3.2 Publish/Subscribe	16
2.3.3 Struttura pacchetto del messaggio MQTT	17
2.3.4 Motivazioni	19
3 Android	20
3.1 Sensori	20
3.1.1 Panoramica dei sensori	20
3.1.2 Acquisizione e Stampa	21
3.2 Passaggio dati fra Activity	32

INDICE

3.2.1	Panoramica passaggio dati	32
3.2.2	Invio	34
3.2.3	Ricezione	35
3.3	Invio dati con protocollo MQTT	37
3.3.1	Panoramica libreria Paho	37
3.3.2	Installazione	37
3.3.3	Connessione	38
3.3.4	Opzioni	39
3.3.5	Pubblicazione	40
4	Raspberry	42
4.1	Installazione	42
4.2	Configurazione	43
4.3	Avvio del servizio	44
4.4	Visualizzazione dati	45
	Conclusioni	49
	Ringraziamenti	52

Capitolo 1

Scenario Applicativo

1.1 Smart-City

Per smart-city s'intende un'area urbana dove, grazie all'utilizzo dell'innovazione tecnologica e più nello specifico delle tecnologie digitali, è possibile perfezionare le infrastrutture pubbliche e i servizi offerti ai cittadini rendendoli più efficienti. Quindi per rendere una città intelligente (smart) si parte dalla digital transformation e dall'utilizzo della tecnologia IoT - Internet of Things nelle diverse sfere della Pubblica Amministrazione: trasporti pubblici e mobilità; gestione e distribuzione dell'energia; illuminazione pubblica; sicurezza urbana; gestione e monitoraggio ambientale; gestione dei rifiuti; manutenzione e ottimizzazione degli edifici pubblici; sistemi di comunicazione e informazione e altri servizi di pubblica utilità. Quindi con il termine smart-city si fa riferimento a una città intelligente, ma soprattutto a una città sostenibile, efficiente e innovativa, che sia nello specifico in grado di garantire un'elevata qualità di vita ai suoi cittadini grazie all'utilizzo di soluzioni e sistemi tecnologici connessi e integrati tra loro.[1]

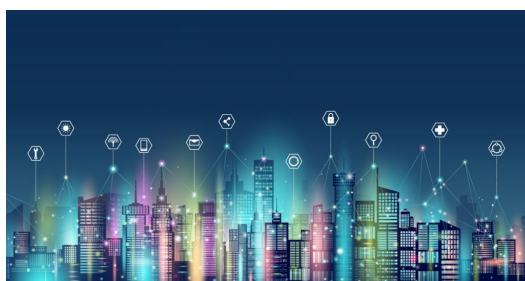


Figura 1.1: Smart-city [1]

CAPITOLO 1. SCENARIO APPLICATIVO

Quando l'Unione Europea parla di Smart Cities include 6 dimensioni[2]:

- Smart People – i cittadini vanno coinvolti e resi partecipi.
- Smart Governance – l'amministrazione deve dare centralità al capitale umano, alle risorse ambientali, alle relazioni e ai beni della comunità.
- Smart Economy – l'economia e il commercio urbano devono essere rivolti all'aumento della produttività e dell'occupazione all'interno della città attraverso l'innovazione tecnologica. Un'economia basata sulla partecipazione e sulla collaborazione e che punta su ricerca e innovazione.
- Smart Living – il livello di comfort e benessere che deve essere garantito ai cittadini legato ad aspetti come la salute, l'educazione, la sicurezza, la cultura ecc. sono anch'essi di prioritaria importanza.
- Smart Mobility – le soluzioni di mobilità intelligente, dall'e-mobility alla sharing mobility ad altre forme di mobility management, devono guardare a come diminuire i costi, diminuire l'impatto ambientale e ottimizzare il risparmio energetico.
- Smart Environment – sviluppo sostenibile, basso impatto ambientale ed efficienza energetica sono aspetti prioritari della città del futuro.



Figura 1.2: Le 6 dimensioni incluse dall'European Commission [1]

Quindi una Smart City è una città dove il capitale infrastrutturale, legato alle infrastrutture tecnologiche, insieme al capitale umano, al capitale ambientale e sociale, vengono massimizzati così che possano essere interconnesse le infrastrutture che la città offre a servizio dei cittadini con il capitale di chi le abita [1]. Tra i punti fondamentali e le caratteristiche principali di una città intelligente, troviamo infatti:

1. Partecipazione e responsabilità condivisa: informazione e comunicazione sono fondamentali per permettere ai cittadini di interagire, dialogare e partecipare allo sviluppo della città e alle decisioni dell'amministrazione locale. Per questo motivo la città intelligente deve potersi appoggiare a un sistema informatico che permetta a chiunque di inviare in tempo reale una segnalazione su un problema o una richiesta. Il primo fondamento della smart city è quindi di essere inclusiva e massimizzare il capitale umano e sociale anche attraverso azioni volte a promuovere lo sviluppo delle attività e del commercio in città.
2. Smart building o edifici intelligenti: gli edifici di nuova costruzione o che subiscono un intervento di riqualificazione devono rispondere a precisi standard di efficienza energetica e smartness. Lo smart building è il tassello fondamentale per la costruzione di una smart city e fa parte del capitale infrastrutturale e sociale.
3. Efficienza energetica e sostenibilità ambientale: per gestire in modo efficiente l'energia e ottenere risultati di risparmio energetico bisogna puntare alla creazione di una smart grid o di sistemi di smart energy. Una città intelligente deve anche puntare sull'uso delle energie rinnovabili e su sistemi intelligenti di gestione dei rifiuti in un'ottica di economia circolare. Importante anche il ruolo delle aree verdi e dei parchi, perché anche il capitale ambientale va ottimizzato e reso efficiente.
4. Sicurezza Integrata: in una città intelligente, la sicurezza è un aspetto importantissimo. Sicurezza significa minore criminalità e maggiore attenzione alle aree critiche come le periferie ad esempio. L'utilizzo di tecnologie innovative e sistemi di sicurezza sempre più interconnessi e integrati permette di raggiungere risultati importanti in questo ambito.
5. Trasporto e mobilità: smart mobility, e-mobility ma anche soluzioni di smart parking. Perché una città sia efficiente, più vivibile e

intelligente, bisogna andare verso soluzioni che snelliscano il traffico e riducano l'inquinamento.

Per far funzionare le smart cities, alla base ci sono delle tecnologie abilitanti in grado di supportare e facilitare il processo di trasformazione delle città. Tra queste tecnologie, vengono individuate come prioritarie: il 5G, **Internet of things (IoT)**, Analisi dei “big data”, Intelligenza Artificiale (AI), ecc...

1.2 Internet of Things

1.2.1 Descrizione di IoT

Internet of Things, letteralmente Internet delle cose o meglio tradotto Internet degli Oggetti, si riferisce a una branca del servizio internet che permette di interconnettere più oggetti fra loro; in particolare questo servizio consiste nella creazione di una rete fra oggetti fisici (auto, moto, sensori, attuatori, device, ecc...) che permette l'interscambio di dati e informazioni fra i diversi dispositivi. In particolare l'idea di rendere qualsiasi oggetto connesso ad internet si estende anche agli essere umani, da cui nasce il concetto di Internet of Everything, così che internet possa essere usato addirittura per connettere persone e per avere un insieme di dati da gestire in maniera diretta, veloce e sicura perché un obbiettivo primario è quello di rendere più efficiente il servizio offerto, ed al giorno d'oggi questo è possibile soltanto avendo una connessione veloce e stabile.

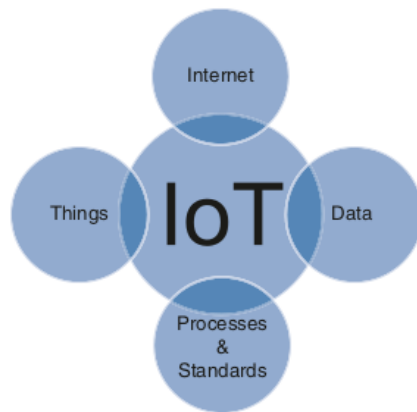


Figura 1.3: IoT - definizione [3]

L'Internet of Things è ampiamente diffuso in molti ambiti differenti, dall'agricoltura ai trasporti al settore automobilistico fino alle città dove

possiamo appunto riprendere il concetto di città-intelligenti. Tra i più importanti troviamo quello industriale ed energetico, è infatti grazie anche all'introduzione dell'IoT che si può parlare di industria4.0, sviluppo del settore industriale con l'introduzione di nuove tecnologie all'avanguardia per migliorare le condizioni di lavoro, creare nuovi modelli di business e aumentare la produttività migliorando così anche la qualità dei prodotti.

1.2.2 Architettura IoT

L'IoT dovrebbe essere in grado di interconnettere un gran numero di oggetti eterogenei attraverso Internet, quindi c'è un bisogno critico di avere un'architettura flessibile a strati. Siccome ancora non si converge verso un'architettura comune, ne esistono di più tipi come illustrato nella Fig.1.4.

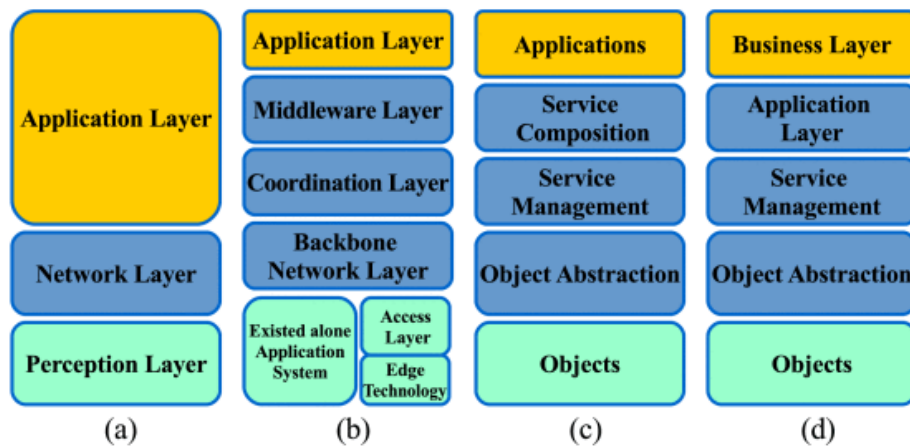


Figura 1.4: IoT - architetture. (a) Three-layer. (b) Middle-ware based. (c) SOA based. (d) Five-layer. [4]

La più comune è il modello (d) Five-layer:

1. Objects Layer

Questo è il primo livello che rappresenta sia i sensori fisici dell'IoT che hanno l'obiettivo di raccogliere ed elaborare informazioni come la posizione, temperatura, peso, movimento, vibrazione, ecc... sia gli attuatori che hanno il compito di eseguire diverse funzionalità. Questo livello, detto anche perception-layer (percezione), digitalizza e trasferisce i dati acquisiti all' Object Abstraction layer, il livello sovrastante, attraverso canali sicuri. I grandi dati creati dall' IoT vengono inizializzati in questo strato.

2. Object Abstraction Layer

Questo livello ha il compito di prendere i dati acquisiti dal livello primario (Object layer) e di trasferirli attraverso canali protetti al layer superiore: Service Management. I dati possono essere trasferiti attraverso varie tecnologie come RFID, 5G, WiFi, Bluetooth Low Energy, infrarossi, ecc. Altre funzioni, come l'archiviazione o il processamento dati vengono gestiti a questo strato.

3. Service Management Layer

Questo livello ha il compito di abbinare un servizio, tramite protocolli di rete cablata prendendo decisioni in autonomia, a chi l'ha richiesto in base ad indirizzi e nomi. Questo layer consente di poter lavorare su applicazioni IoT senza la necessità di avere una piattaforma hardware specifica.

4. Application Layer

Il livello applicativo fornisce i servizi richiesti dai clienti, ad esempio dati come misurazioni di temperatura e umidità dell'aria che sono stati acquisiti e processati dai livelli precedentemente descritti. L'importanza di questo livello per l'IoT è che ha la capacità di fornire servizi intelligenti di alta qualità per soddisfare le esigenze dei clienti, è proprio in questo strato che gli utenti finali possono interagire attraverso un dispositivo per ricercare i dati d'interesse.

5. Business Layer

Questo livello ha il compito di gestire le attività e i servizi complessivi del sistema IoT. Le responsabilità di questo livello sono di costruire un modello di business, grafici, diagrammi di flusso, ed altri tipi, sulla base dei dati ricevuti dal livello Applicazione. Dovrebbe anche progettare, analizzare, implementare, valutare, monitorare e sviluppare elementi relativi al sistema IoT. Il Business Layer consente di supportare i processi decisionali basati sull'analisi dei Big Data. Inoltre, questo livello confronta l'output di ogni livello con l'output previsto per migliorare i servizi e mantenere la privacy degli utenti.

L'architettura serve a comprendere il funzionamento logico dell'Internet of Things, ma per comprendere meglio il significato e le funzionalità reali dell'IoT è necessario discutere ed approfondire gli elementi costitutivi che vanno dall'identificazione di chi ha richiesto l'informazione e soprattutto quale dato è stato richiesto, fino alla semantica che ha la capacità di estrarre conoscenze in modo intelligente per fornire i servizi richiesti.[4]



Figura 1.5: IoT - elementi [4]

L'identificazione serve ad abbinare i servizi alla domanda in maniera univoca, sono presenti diversi metodi d'identificazione ma tutti si eguagliano nell'indirizzamento degli oggetti per distinguere l'ID (nome specifico del dispositivo) dall'indirizzo (indirizzo all'interno di una rete di comunicazione). Il rilevamento raccoglie dati da oggetti correlati all'interno della rete per inviarli al cloud così che questi possano essere interpretati e analizzati. La comunicazione interconnette un insieme di oggetti eterogenei per fornire servizi intelligenti specifici, per farlo vengono utilizzate tecnologie come RFID, Wifi o 5g. La computazione è l'elemento di calcolo (es. microcontrollori, microprocessori, SOC, FPGA) che rappresenta il "cervello" e la capacità computazionale dell'IoT, per implementare questo elemento e quindi per eseguire le applicazioni IoT vengono utilizzate diverse piattaforme hardware tra cui ad esempio Raspberry. La semantica nell'IoT si riferisce alla capacità di estrarre conoscenza in modo intelligente da diverse macchine per fornire i servizi richiesti.[4]

Per gestire in maniera efficiente l'intera architettura IoT sono stati creati vari protocolli in grado di semplificare il lavoro dei programmatori di strutture IoT, in generale esistono quattro grandi categorie dei protocolli, vale a dire: protocolli applicativi come **MQTT**, protocolli di rilevamento dei servizi, protocolli di infrastruttura e altri protocolli influenti. Tuttavia, non tutti questi protocolli devono essere raggruppati insieme per fornire una determinata applicazione IoT.

1.2.3 L'uso di MQTT nell'IoT

Dalla necessità di un protocollo di comunicazione affidabile per i sistemi IoT nasce MQTT che è un protocollo applicativo di messaggistica sviluppato nel 1999 per mano dell'organizzazione OASIS (Organization for the Advancement of Structure Information Standards). Questo protocollo è un servizio di messaggistica di tipo publisher-subscriber che è in grado di trasmettere dati su reti a bassa larghezza di banda o inaffidabili con un consumo di energia molto basso, ideale per essere implementato in

dispositivi da risorse limitate. Caratteristiche come l'implementazione leggera, aperta e facile rendono MQTT un protocollo di comunicazione ideale per gli ambienti con vincoli come l'IoT [5].

Facendo riferimento agli scenari di applicazione dell'IoT è possibile vedere come MQTT sia diffuso in moltissimi di questi diventando motivo di ricerca per lo sviluppo di nuovi progetti all'interno di aziende di importanza mondiale [6]: nel settore automotive Volkswagen per creare una connessione fra le auto [7] e BMW per applicazioni di car-sharing [8], mentre per il settore smart-home l'IBM per il monitoraggio e controllo dell'energia [9].

Capitolo 2

Tecnologie utilizzate

2.1 Android

Android è un sistema operativo sviluppato da Google per un'ampia gamma di dispositivi mobili, nasce nel 2003 con rilascio della prima versione ufficiale nel settembre del 2008. Questo software è stato creato per sistemi embedded (smartphone, tablet, orologio, auto, ecc...) si tratta di una piattaforma open source quindi presenta un codice sorgente che può essere visualizzato, scaricato e migliorato senza la necessità di richiedere consensi poiché non presenta diritti d'autore ad eccezione di un piccolo numero di pacchetti. L'utilità dei dispositivi android risiede nel fatto di avere a disposizione uno svariato numero di componenti hardware tra cui GPS, sensori di temperatura, umidità, pressione e anche sensori gestiti a livello software tramite librerie che dai dati acquisiti elaborano dati in uscita; è importante sapere però che la presenza di questi sensori su ogni device è dipendente dal modello stesso del dispositivo.

2.1.1 Architettura della piattaforma

Android è quindi un software basato principalmente su *kernel Linux*, esso infatti rappresenta il primo livello della pila software su cui si basa l'architettura come mostrato nella figura 2.1. Grazie all'utilizzo di un kernel così noto è possibile sfruttare le principali funzionalità di sicurezza consentendo così agli sviluppatori un lavoro meno complesso pur mantenendo la qualità del progetto.

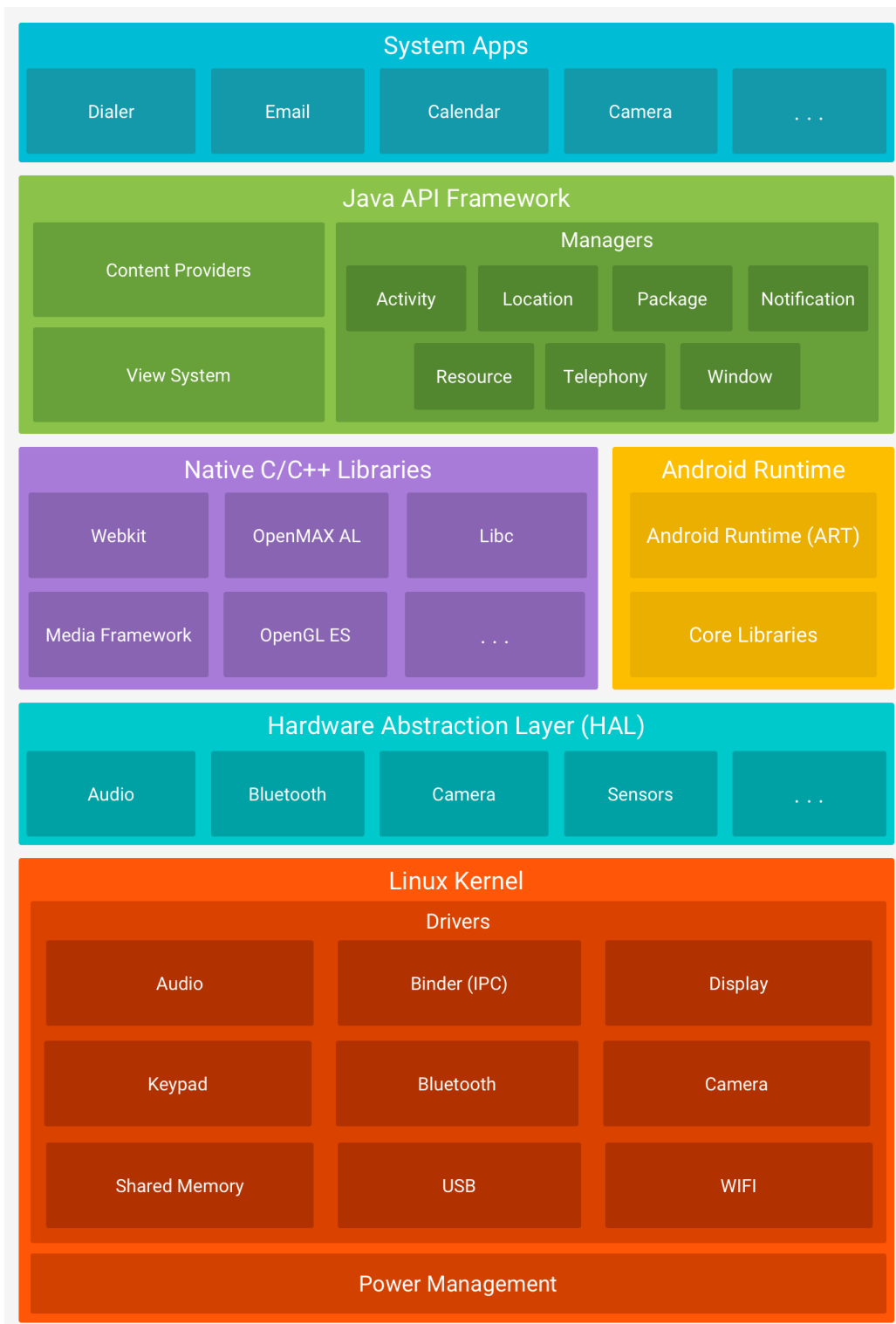


Figura 2.1: Stack software Android [10]

Come già detto e come mostrato in figura 2.1 il livello primario, alla base dell'architettura è quello Linux.

Il secondo strato viene denominato *Hardware Abstraction Layer* (HAL) e fornisce un'astrazione dell'hardware definendo interfacce standard che hanno il compito di esporre le capacità fisiche del dispositivo al framework API Java di livello superiore. Non esiste un modulo specifico per l'HAL bensì esso è costituito da più moduli di libreria ognuno in grado di interagire con un tipo specifico di componente hardware come il bluetooth, la fotocamera, il modulo per i sensori o altri. Per comprendere meglio il funzionamento di questo layer, quando ad esempio un'API del framework effettua una chiamata per accedere al sensore di pressione presente sul dispositivo, il sistema Android carica il modulo della libreria sensori.

Il livello successivo è costituito da due blocchi:

- *Android Runtime (ART)*: software in grado di fornire i servizi utili all'esecuzione di un programma, è stato scritto per poter eseguire macchine virtuali sui dispositivi poiché ogni applicazione viene eseguita nel proprio processo con una propria istanza di Android Runtime.
- *Librerie Native C/C++*: poiché Android utilizza codice nativo per eseguire e gestire i servizi, come ad esempio i servizi principali quali ART e HAL, si richiedono quindi le librerie native scritte in C e C++.

Proseguendo nell'esposizione troviamo il quarto livello, *Java API Framework*, siccome l'intero set di funzionalità del sistema operativo Android è disponibile tramite API scritte in Java con questo livello è possibile utilizzare le risorse e quindi gli elementi costitutivi necessari per creare app Android. L'ultimo strato, ma non meno importante, è il *System Apps* che comprende le app di base (e-mail, messaggi SMS, calendari, ecc...) ed esse hanno due funzioni: sia come app per gli utenti sia come app per gli sviluppatori così che se una nuova applicazione di terzi deve utilizzare una funzione che un app di sistema ha già implementata al suo interno, l'app di terzi può decidere di utilizzare la funzione dall'app di sistema senza dover nuovamente implementare la funzione all'interno dell'app stessa.^[10]

2.1.2 Ambiente di sviluppo: Android Studio

Android Studio è l'ambiente di sviluppo integrato (IDE) ufficiale per lo sviluppo di app Android, basato su IntelliJ IDEA.

CAPITOLO 2. TECNOLOGIE UTILIZZATE

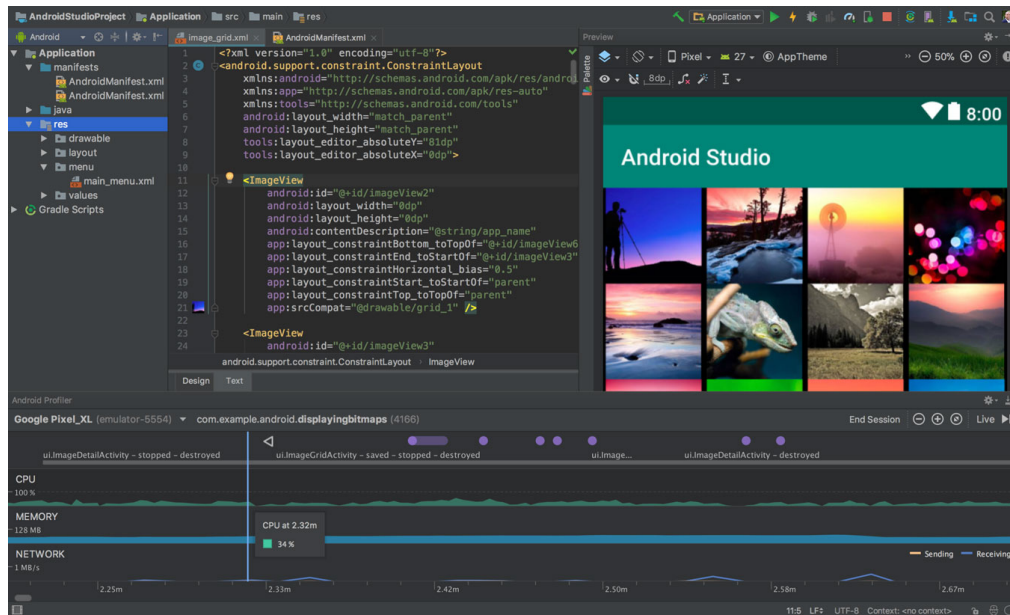


Figura 2.2: Editor Android Studio [11]

Ogni progetto in Android Studio presenta un insieme di file suddivisi principalmente in due gruppi: Application e Gradle Scripts; sotto Gradle Scripts vi sono tutti i file di build del progetto, mentre Application si divide in:

- manifest, contenente il file AndroidManifest.xml che serve a descrivere le informazioni essenziali sull'app di progetto utili per il build Android, il sistema operativo e Google Play.
- java, contenente appunto i file sorgente in linguaggio Java utili alla programmazione della struttura dell'app.
- res, contenente tutto quello che non è codice sorgente come layout XML per la programmazione della parte grafica, immagini, ecc.

Android Studio utilizza Gradle come base del sistema di compilazione, quindi il file Gradle serve a gestire diverse funzionalità aggiuntive quali la creazione di più APK per la stessa app senza la necessità di dover modificare i diversi file sorgente. Inoltre Android Studio fornisce un servizio di debugging real time per la simulazione dell'applicazione di progetto, con la possibilità di utilizzare device virtuali implementabili grazie alla virtualizzazione della macchina.[11]

2.1.3 Motivazioni

Principalmente i due sistemi operativi più diffusi al mondo sono iOS e Android, quest'ultimo però presenta dei vantaggi in un ambiente di progettazione come quello proposto da questo elaborato di tesi. Innanzitutto Android è un software open source con una miriade di risorse applicabili per ogni tipo di progetto, come librerie per tool specifici del dispositivo. Inoltre android garantisce un'ottimizzazione in termini di consumi, necessaria e fondamentale per applicazioni IoT. Riguardo allo sviluppo dell'applicazione sono sempre disponibili un elevato numero di forum e addirittura siti Android per sviluppatori di applicazioni contenenti manuali e spiegazioni su qualsiasi libreria utile per il proprio progetto. Infine, riportando qualche dato: grazie alla licenza gratuita di Android si stima che i produttori possano realizzare dispositivi mobili a prezzi inferiori consentendo alle persone in tutto il mondo di accedere alla tecnologia degli smartphone; sempre grazie all'open source viene data la libertà di progetto ai produttori di device Android, avendo così più di 24 mila dispositivi diversi in commercio con quasi 1300 brand produttori di dispositivi mobili con sistema operativo Android ed oltre 1 milione di app; l'accrescimento dello sviluppo Android consente di avere un incremento dei posti di lavoro disponibili, l'industria dei dispositivi mobili apporta più di 3000 miliardi di dollari al PIL globale in 236 paesi [12, 13].

2.2 Raspberry Pi

Il Raspberry Pi è un computer a scheda singola prodotto nel Regno Unito dalla Raspberry Pi Foundation e distribuito nel 2012, progettata per ospitare sistemi operativi basati su Kernel Linux o RISC OS.[14]

Nella trattazione di questo capitolo si farà riferimento al modello utilizzato per lo sviluppo del progetto: Raspberry Pi3B+.

2.2.1 Hardware

Il progetto si basa su un system-on-a-chip di fabbricazione Broadcom (BCM2837B0) con processore Cortex-A53 (ARMv8) da 1,4GHz a 64 bit. La scheda inoltre è dotata di una memoria SDRAM da 1GB (LPDDR2), di una GPU videoCore IV e di una porta MicroSD per caricare il sistema operativo e il salvataggio dati come equivalente di disco fisso non volatile.

Per la gestione I/O sono presenti 4 porte fisiche USB 2.0, un ingresso Gigabit Ethernet (con portata massima 300 Mbps), 40 pin di GPIO per la connessione con periferiche esterne, e una porta HDMI Full-size.

La scheda è alimentata con una potenza d'ingresso 5V/2.5A DC che possono essere fornite tramite l'alimentatore che viene fornito in dotazione o tramite GPIO; inoltre supporta la modalità Power-over-Ethernet (PoE) con apposita HAT aggiuntiva.

Per la connettività presenta connessione wireless IEEE 802.11.b/g/n/ac da 2.4 GHz e 5 GHz e Bluetooth 4.2, Bluetooth Low Energy.[15]

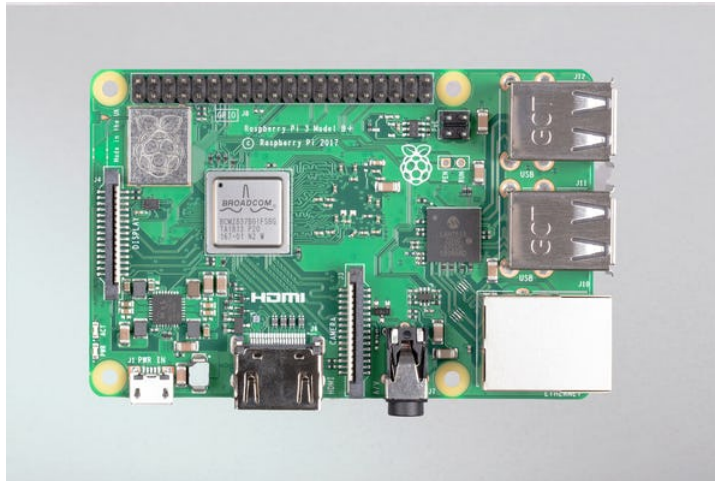


Figura 2.3: Raspberry Pi3B+ [15]

2.2.2 Software

Raspberry Pi supporta ufficialmente sistemi operativi basati su Kernel Linux, per il normale utilizzo viene consigliato il Raspberry Pi OS come sistema operativo gratuito Debian ed ottimizzato per l'hardware di Raspberry Pi.

Per l'installazione del sistema operativo viene utilizzata la scheda MicroSD da implementare all'avvio del dispositivo con installato al suo interno Raspberry Pi OS, mentre per l'avvio è necessario disporre di un alimentatore micro-USB 2.5A.[15]

2.2.3 Motivazioni

Dalle descrizioni precedenti si evince la potenza di Raspberry in termini di consumo energetico, prestazioni di calcolo e varietà di possibili collegamenti di periferiche per la realizzazione di svariati progetti; altro vantaggio di questa scheda system-on-a-chip è l'elevato rapporto qualità prezzo.

2.3 MQTT

Come si è visto nella sezione inerente all'Internet of Things (1.2) si evince la necessità di avere un protocollo di comunicazione efficiente e affidabile per realizzare la sua implementazione. Al giorno d'oggi esistono diversi protocolli di messaggistica tra cui scegliere per implementare le varie esigenze dell'IoT; nel seguito della trattazione verrà descritto il protocollo utilizzato per la realizzazione del progetto di tesi: il protocollo MQTT (Message Queue Telemetry Transport).

2.3.1 Introduzione

Questo è un protocollo di messaggistica applicativo, inventato dal Dr. Andy Stanford-Clark di IBM e Arlen Nipper di Arcom nel 1999 e standardizzato (ISO/IEC PRF 20922) nel 2013 per mano dell'organizzazione OASIS per client-server. Inventato principalmente per la riduzione al minimo della larghezza di banda della rete e dei requisiti delle risorse del dispositivo garantendo però una consegna affidabile con un consumo di energia fortemente ridotto. Il protocollo funziona su protocolli di rete TCP/IP che forniscono connessioni ordinate, senza perdite e bidirezionali, ed è un protocollo di tipo publish/subscribe che utilizza il numero di porta registrato IANA 8883 per le connessioni SSL/TSL e il numero di porta 1883 per le connessioni non TLS[5]; quest'ultima sarà quella utilizzata nel progetto di tesi.

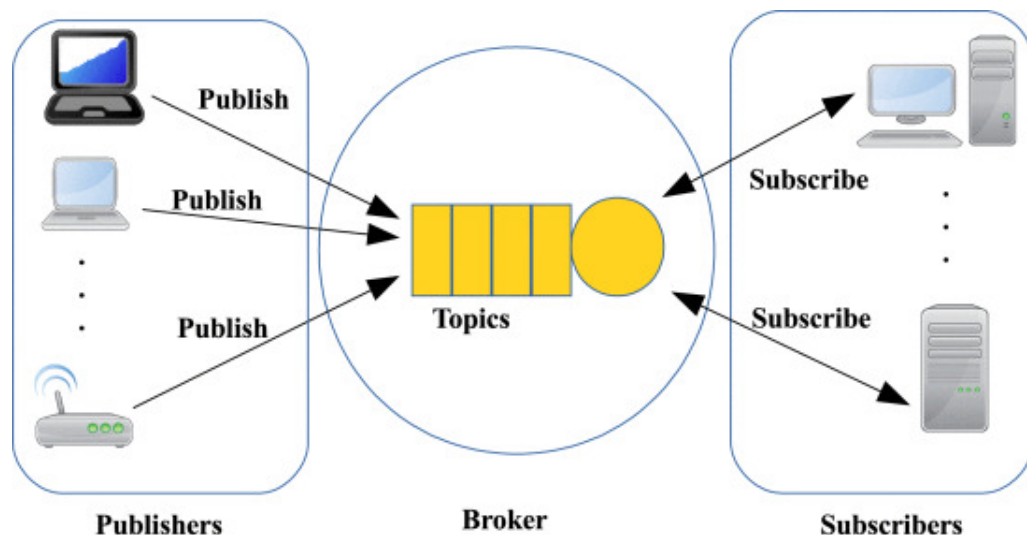


Figura 2.4: Architettura MQTT [4]

MQTT ha tre componenti costitutivi: Publisher (un MQTT client) che nel caso in oggetto è rappresentato da Android, broker (un MQTT Server) realizzato da Raspberry, e Subscriber (un MQTT client) che sarà sviluppato sempre dall'app Android.

2.3.2 Publish/Subscribe

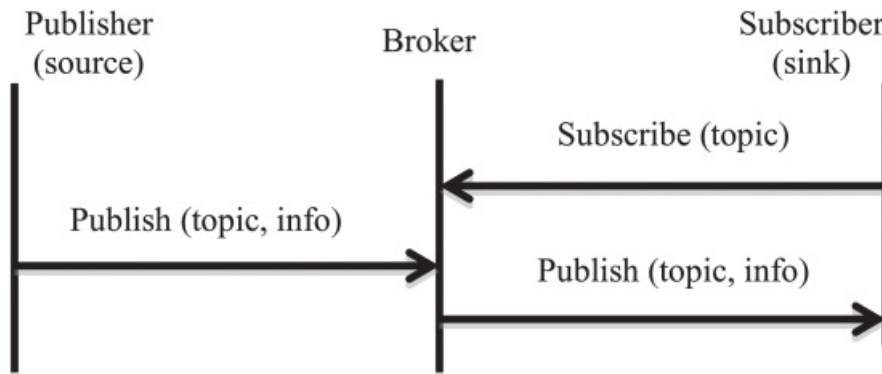


Figura 2.5: Publish/Subscribe in MQTT [4]

Un client (publisher o subscriber) è un dispositivo che utilizza il protocollo MQTT, esso è responsabile dell'apertura della connessione di rete al server, della creazione di messaggi da pubblicare, della pubblicazione dei messaggi dell'applicazione sul server, della sottoscrizione per richiedere i messaggi dell'applicazione che è interessato a ricevere, dell'annullamento dell'iscrizione per rimuovere una richiesta di messaggi dell'applicazione e della chiusura della connessione di rete al server. Il messaggio trasportato dal protocollo attraverso la rete contiene dati di *payload*, una *QoS*, una raccolta di proprietà e un nome di argomento (*topic*). Il server MQTT (broker) è un programma o altro dispositivo che si potrebbe paragonare ad un ufficio postale tra editori e abbonati che quindi è responsabile dell'accettazione delle connessioni di rete dai client, dell'accettazione dei messaggi dell'applicazione pubblicati dai client, dell'elaborazione delle richieste di sottoscrizione e annullamento della sottoscrizione dai client, dell'invio dei messaggi dell'applicazione ai client in base alle sue sottoscrizioni e della chiusura della connessione di rete dal client.[5]

L'utilizzo di un architettura di tipo Publish/Subscribe presenta dei vantaggi quali: il disaccoppiamento spaziale, dove i diversi client indipendentemente dalla loro funzione non hanno la necessità di conoscersi

a vicenda ma hanno bisogno solo di sapere a priori l'indirizzo IP e la porta del broker avendo così la possibilità di collegare un gran numero di client al server; il disaccoppiamento temporale, permettendo così di non far funzionare i diversi client publisher e subscriber nello stesso istante salvando i messaggi inviati dal primo; asincronia, lavorando in modo asincrono il protocollo permette l'invio di più messaggi negli stessi istanti di tempo attraverso funzioni di *CallBack*.

2.3.3 Struttura pacchetto del messaggio MQTT

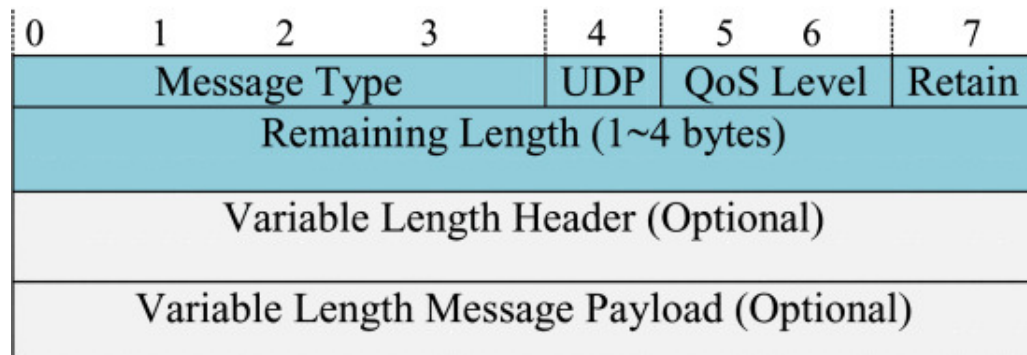


Figura 2.6: Formato messaggio MQTT [4]

Message Type

I primi due byte del messaggio sono header fissi, i primi 4 bit presentano il *Message Type* che indica una varietà di messaggi[4]:

- CONNECT. richiesta del client di connettersi al broker.
- CONNACK. risposta del broker ad una richiesta di connessione mandata dal client.
- PUBLISH. richiesta al broker di pubblicare un messaggio su uno specifico topic da parte del Publisher.
- PUBACK. risposta del Broker al seguito della pubblicazione di un messaggio.
- SUBSCRIBE. richiesta da parte del client al Broker di iscriversi ad un topic.
- SUBACK. risposta del Broker al seguito di una richiesta di Subscribe da parte del client.

- UNSUBSCRIBE. richiesta da parte del client al Broker per disiscriversi da un topic.
- DISCONNECT. richiesta del client di disconnettersi dal Broker.

UDP

Il successivo bit rappresenta il flag UDP che indica se il messaggio è duplicato e che quindi il destinatario potrebbe averlo già ricevuto in precedenza.[4]

QoS (Quality of Service)

I seguenti due bit rappresentano i flag di QoS utili alla garanzia del recapito dei messaggi di pubblicazione definendo la qualità dei messaggi pubblicati e ricevuti.

MQTT dispone di tre livelli QoS, facendo degli esempi[5]:

- QoS 0. conosciuta anche come "consegna al più una volta", ed è l'invio in tempo reale dei dati del sensore di pressione, umidità o altro, a un'applicazione di lettura remota, che può essere interpretata dal Raspberry nel caso in questione, dove non importa se la connessione all'applicazione che legge i dati dal sensore viene persa per un po'.
- QoS 1. rinominata "consegna almeno una volta", dove appunto l'invio dei messaggi è garantito almeno una volta; potrebbe verificarsi duplicazione del messaggio.
- QoS 2. indicata anche come "consegna esattamente una volta", in questa modalità si assicura che i messaggi siano consegnati esattamente una volta e non di più; in questo caso si osserva un piccolo sovraccarico di trasporto e gli scambi di protocollo sono ridotti al minimo per ridurre il traffico di rete.

Retain

L'ultimo bit del primo Byte rappresenta il flag di conservazione ed informa il server di conservare l'ultimo messaggio di pubblicazione (Publisher) ricevuto su un dato topic e di inviarlo ai nuovi abbonati (Subscriber) come primo messaggio al momento della loro richiesta di sottoscrizione a quel topic.[4] È importante sapere che affinché questo meccanismo funzioni il client deve essere connesso al broker in maniera persistente e il QoS deve essere maggiore di 0.

Remaining Length

Il secondo Byte del messaggio mostra la lunghezza rimanente del messaggio, ovvero definisce il numero di byte nel pacchetto che non sono stati usati.^[4]

Variable Length Header

Campo opzionale nel pacchetto messaggio MQTT, ed ha appunto una lunghezza variabile utile ad alcuni tipi di messaggio che necessitano di inviare informazioni ulteriori addizionali al broker.

Variable Length Message Payload

Quest'ultimo campo, anch'esso variabile e opzionale, corrisponde ai bit d'informazione che riconoscono il contenuto del messaggio pubblicato dal client Publisher sul broker. Se il messaggio è di tipo Connect allora il payload conserva le credenziali (Username e Password) utili a stabilire la corretta connessione con il broker, invece se il messaggio è di tipo Publish allora questo campo contiene il dato vero e proprio inviato.

2.3.4 Motivazioni

Come ampiamente mostrato in questo capitolo il protocollo MQTT è leggero per la comunicazione con dispositivi con risorse limitate, ha una enorme fattibilità per le reti incentrate sull'IoT, ed infine è fortemente affidabile per connessione con limitate larghezze di banda tipiche dell'Internet of Things.

Capitolo 3

Android

Nei prossimi capitoli verrà utilizzato un approccio pratico, in particolare per questo capitolo verranno descritti i passi affrontati per lo sviluppo dell'architettura realizzata con dispositivo Android tramite la programmazione su ambiente di sviluppo dedicato: Android Studio.

Ricordando che lo scopo principale dell'intero progetto è di ottimizzare le soluzioni per la mobilità e la sicurezza realizzando un architettura Publisher/Subscriber da inserire nel contesto IoT, possiamo definire l'obiettivo di questa applicazione che è quello di diventare un client per l'architettura potendo quindi richiedere una connessione al broker centrale ed inviare/ricevere messaggi MQTT;

in particolare, il lavoro svolto con questo elaborato di tesi è stato quello di realizzare la parte Publish gestendo l'acquisizione dei dati dai sensori se presenti sul dispositivo per poi permettere la loro visualizzazione a video sulla schermata principale e tramite il passaggio di questi ultimi attraverso le varie activity abilitare anche l'invio di questi dati tramite MQTT al broker centrale.

3.1 Sensori

3.1.1 Panoramica dei sensori

Gran parte dei dispositivi Android ha la disponibilità di sensori integrati fisici e/o software che misurano il movimento, l'orientamento e varie condizioni ambientali fornendo dati già grezzi e con elevata precisione e accuratezza; nel progetto in questione vengono acquisiti i dati dai seguenti sensori: temperatura ambiente, pressione, umidità, posizione, altitudine e contapassi.

È possibile accedere ai sensori ed ottenere i dati attraverso l'**Android sensor framework** purché sia incluso all'interno del progetto il package **android.hardware** contenente le seguenti classi ed interfacce:

- **SensorManager**: questa classe serve a creare un'istanza del servizio del sensore, fornendo vari metodi per accedere ai sensori, settare e annullare la registrazione dei *listener* di eventi del sensore.
- **Sensor**: con questa classe è possibile creare un'istanza per uno specifico sensore, fornendo metodi specifici per il tipo di sensore.
- **SensorEvent**: il sistema utilizza questa classe per inizializzare un oggetto evento sensore così che fornisca informazioni sull'evento includendo informazioni come i dati grezzi, il tipo di sensore che ha generato l'evento o l'accuratezza del dato.
- **SensorEventListener**: con questa interfaccia è possibile creare due metodi di *callback* per ottenere notifiche di evento del sensore quando quest'ultimo cambia valore o precisione.

Nel progetto di tesi vengono utilizzate queste API relative ai sensori per eseguire due attività di base: identificazione per verificarne la presenza sul dispositivo che ospita l'applicazione, siccome la presenza del sensore è dipendente dal dispositivo; monitoraggio degli eventi del sensore così che sia possibile acquisire il relativo dato quando questo subisce una variazione.^[16]

3.1.2 Acquisizione e Stampa

Per gestire l'acquisizione dei sensori, indipendentemente dal tipo di sensore bisogna inizializzare il **Sensor Manager** così che sia possibile istanziare il servizio del sensore per poi utilizzare i vari metodi che fornisce la classe; quindi bisogna per prima importare la suddetta libreria.

```
import android.hardware.SensorManager;
...
public class SensorInfo ... implements SensorEventListener{

    SensorManager sensorManager;
    ...
    protected void onCreate(Bundle savedInstanceState) {
        ...
        sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    }
    ...
}
```

In questo modo è stato istanziato il *sensorManager* per poi essere utilizzato per ottenere il servizio; successivamente in base al tipo di sensore si potranno utilizzare i vari metodi.

Contapassi

Come detto precedentemente bisogna istanziare un oggetto della classe *Sensor* per ogni sensore che si vuole utilizzare.

```
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
...
public class SensorInfo ... {

    Sensor pedometer;
    boolean isStepSensorPresent = false;
    TextView pedometerValue;
    ...
    protected void onCreate(Bundle savedInstanceState) {
        pedometerValue = findViewById(R.id.pedometer_text_view);
        ...
        if ((sensorManager.getDefaultSensor(Sensor.TYPE_STEP_DETECTOR)) !=
            null) {
            //StepDetector sensor is present
            pedometer =
                sensorManager.getDefaultSensor(Sensor.TYPE_STEP_DETECTOR);
            isStepSensorPresent = true;
            pedometerValue.setText("0.0");
            //Start listen event for pedometer
            sensorManager.registerListener(this, pedometer,
                SensorManager.SENSOR_DELAY_NORMAL);
        } else {
            //StepDetector sensor is not present
            pedometerValue.setText("Not Present");
            isStepSensorPresent = false;
        }
    }
    ...
}
```

Per primo si importano nuovamente le librerie necessarie all'utilizzo dei package elencati precedentemente, successivamente si istanzia il *Sensor* per il sensore contapassi e una variabile *boolean* utile a salvare il risultato della richiesta di presenza del sensore sul dispositivo; in questo modo la variabile logica verrà settata a *true* soltanto se il sensore è presente fisicamente sul dispositivo. In quest'ultimo caso sarà possibile ottenere il valore del

contapassi ogni qualvolta che il sensore cambia il proprio valore purché si avvii il *.registerListener* per il contapassi, perciò si è deciso di inizializzare il valore a 0 passi. Altrimenti se il sensore non è fisicamente presente sul dispositivo viene settata la *TextView* come "Not Present" così che sia possibile visualizzare all'utente tramite la schermata principale dell'applicazione che il sensore non è presente. Viene utilizzato il tipo *TYPE_STEP_DETECTOR* per avere un conteggio dei passi sequenziale.

Per la visualizzazione e l'acquisizione del dato ogni qualvolta che il valore varia si deve richiedere il dato all'interno della funzione *onSensorChanged* così che sia possibile verificare quale sensore abbia richiesto l'evento; quindi nel caso in questione il contapassi.

```
public class SensorInfo ... {

    int stepDetect = 0;
    ...
    public void onSensorChanged(SensorEvent sensorEvent) {
        if(sensorEvent.sensor == pedometer){
            stepDetect = (int) (stepDetect + sensorEvent.values[0]);
            pedometerValue.setText(String.valueOf(stepDetect));
        }
        ...
    }
}
```

Viene inizializzato un contatore *stepDetect* utile al salvataggio del conteggio dei passi totali così che possa essere incrementato ad ogni rilevamento di un nuovo numero di passi. Infine con *setText* è possibile settare la *TextView* collegata all'icona del contapassi presente nella schermata principale.

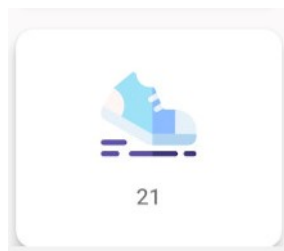


Figura 3.1: Valore contapassi

Posizione - Altitudine

Per l'acquisizione della posizione del dispositivo la gestione è un po' più complessa rispetto al contapassi, questo perché entrano in gioco criteri di privacy e sicurezza che devono essere rispettati per poter acquisire

latitudine e longitudine. Per primo è importante abilitare i permessi per l'acquisizione della posizione sul file *Manifest.xml*

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION" />
```

e anche le *dependencies* nel file *Gradle (Module)*

```
dependencies {
    implementation
        'com.google.android.gms:play-services-location:18.0.0'
    ...
}
```

A questo punto è possibile partire con la vera e propria acquisizione, per prima vanno importati i package che contengono le librerie utili al rilevamento della posizione. Successivamente all'interno della classe dell'activity vengono dichiarati i *float* utili alla raccolta dei valori corrispondenti a latitudine e longitudine, e il valore logico *locationPermission* che ha il compito di segnalare la risposta dell'utente alla richiesta dei permessi GPS. Inoltre deve essere dichiarato l'identificativo per la richiesta dei permessi che deve essere maggiore di 0.

```
import com.google.android.gms.location.LocationCallback;
import com.google.android.gms.location.LocationRequest;
import com.google.android.gms.location.LocationResult;
import com.google.android.gms.location.LocationServices;

public class SensorInfo ... {

    private static final int REQUEST_CODE_LOCATION_PERMISSION = 1;
    boolean locationPermission = false;
    float latitude;
    float longitude;
    ...
    protected void onCreate(Bundle savedInstanceState) {
        gpsValue = findViewById(R.id.textView_position);
        altitudeValue = findViewById(R.id.altitude_text_view);
        ...
    }
}
```

Successivamente si è creata una funzione *accessLocation()* che viene chiamata all'interno del metodo *onResume()* così che all'avvio e riavvio da RAM dell'applicazione sia possibile richiedere i permessi se negati e/o la posizione se i permessi sono garantiti. All'interno della funzione ci si interroga se i permessi sono abilitati a priori, da impostazioni

dell'applicazione, in caso di esito positivo viene settato il *boolean* a valore logico *true* per segnalare l'avvenuto ottenimento dei permessi e in seguito viene chiamata una seconda funzione *getCurrentLocation()* per ottenere la posizione corrente.

```
protected void onResume() {
    super.onResume();
    accessLocation(); //Location
}

private void accessLocation(){
    if (ContextCompat.checkSelfPermission(getApplicationContext(),
        Manifest.permission.ACCESS_FINE_LOCATION) !=
        PackageManager.PERMISSION_GRANTED) {
        locationPermission = false;
        ActivityCompat.requestPermissions(SensorsInfo.this, new
            String[]{Manifest.permission.ACCESS_FINE_LOCATION},
            REQUEST_CODE_LOCATION_PERMISSION);
    } else {
        locationPermission = true;
        getCurrentLocation();
    }
}
```

Invece nel caso in cui l'interrogazione abbia esito negativo il valore logico viene settato a *false* e successivamente vengono richiesti i permessi a runtime all'utente. Il risultato del nuovo interrogatorio viene revisionato su una nuova funzione: *onRequestPermissionsResult*

```
public void onRequestPermissionsResult(int requestCode, @NonNull
    String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions,
        grantResults);

    if (requestCode == REQUEST_CODE_LOCATION_PERMISSION &&
        grantResults.length > 0) {
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            getCurrentLocation();
        } else {
            altitudeValue.setText("Not Present \n without location");
        }
    }
}
```

Qui di nuovo verifichiamo l'esito del nuovo interrogatorio posto all'utente, in caso di esito negativo viene settata la *TextView* del sensore altitudine segnalando a video l'impossibilità di calcolare il valore del sensore senza il suo consenso GPS, in quanto questo dato viene ottenuto tramite una funzione del

package *location*. In caso di esito positivo allora viene richiamata la funzione *getCurrentLocation()*.

```
private void getCurrentLocation() {
    LocationRequest locationRequest = new LocationRequest();
    locationRequest.setInterval(10000);
    locationRequest.setFastestInterval(3000);
    locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);

    if (ActivityCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION) !=
        PackageManager.PERMISSION_GRANTED &&
        ActivityCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_COARSE_LOCATION) !=
        PackageManager.PERMISSION_GRANTED) {
        //NewRequestPermission
        return;
    }
    LocationServices.getFusedLocationProviderClient(SensorsInfo.this)
        .requestLocationUpdates(locationRequest, new LocationCallback() {
            @Override
            public void onLocationResult(@NonNull LocationResult
                locationResult) {
                super.onLocationResult(locationResult);
                super.onLocationResult(locationResult);
                LocationServices
                    .getFusedLocationProviderClient(SensorsInfo.this)
                    .removeLocationUpdates(this);
                if(locationResult != null &&
                    locationResult.getLocations().size() > 0){
                    int latestLocationIndex =
                        locationResult.getLocations().size() - 1;
                    latitude = (float) locationResult.getLocations()
                        .get(latestLocationIndex).getLatitude();
                    longitude = (float) locationResult.getLocations()
                        .get(latestLocationIndex).getLongitude();
                    gpsValue.setText(String.format("Latitu: %.2f\nLongit:
                        %.2f", latitude, longitude));
                    altitudeValueSensor = (float)
                        locationResult.getLocations().get(latestLocationIndex)
                        .getAltitude();
                    altitudeValue.setText(String.format("%.2f m",
                        altitudeValueSensor));
                }
            }
        }, Looper.getMainLooper());
}
```

Con questa funzione si acquisiscono i dati inerenti al sensore GPS per

ottenere la posizione e da questa si ricava l'altitudine del dispositivo. In particolare per ottenere questi risultati si entra nell' *onLocationResult()* dove ogni qualvolta che la posizione varia questa viene aggiornata grazie all'avvio del *Looper.getMainLooper()* che permette l'avvio di un *thread* parallelo all'esecuzione del programma. Infine con l'utilizzo dei *.setText()*, metodi delle *TextView* collegate alle *CardView* del file *.xml*, i dati aggiornati vengono visualizzati nella schermata principale.

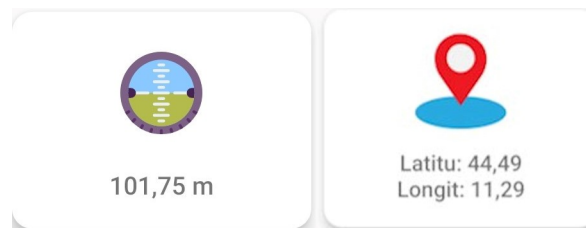


Figura 3.2: Altitudine Posizione

Temperatura, Umidità e Pressione

Per l'acquisizione dei restanti sensori ci si rifà in gran parte al modello d'acquisizione per il contapassi, l'unica differenza risiede nel fatto che questa applicazione è già stata sviluppata in precedenza da un collega realizzando l'acquisizione dei dati da sensori esterni tramite un modulo bluetooth collegato all'applicazione. Per questo motivo si è deciso di utilizzare un approccio di tipo mutuamente esclusivo per questi sensori, ovvero l'acquisizione viene gestita soltanto nel momento in cui il modulo non risulta connesso all'applicazione e pertanto non risulta possibile acquisire i dati da sensori esterni; al contrario il contapassi non era un sensore implementato esternamente pertanto non necessita della mutua esclusione.

Ricordando che come primo passo bisogna implementare l'inizializzazione del **Sensor Manager**, possiamo poi iniziare l'acquisizione definendo i sensori.

```
public class SensorInfo ... {
    Sensor temperature;
    Sensor pressureSensor;
    Sensor humiditySensor;
    boolean isAmbientTempPresent = false;
    boolean isPressureSensorPresent = false;
    boolean isHumiditySensorPresent = false;
    float tempValueSensor = 0;
    float pressureValueSensor = 0;
    float humidityValueSensor = 0;
```

```
protected void onCreate(Bundle savedInstanceState) {  
    tempValue = findViewById(R.id.textView_temp);  
    humidityValue = findViewById(R.id.textView_brightness);  
    pressureValue = findViewById(R.id.pressure_text_view);  
  
    setSensor();  
}  
}
```

Per l'inizializzazione, quindi per l'interrogazione della presenza dei sensori sul dispositivo è stata creata una funzione *setSensor()* così che per ogni sensore che si voglia aggiungere in futuro non si ha la necessità di richiamare tutto il codice più volte ma solo di aggiungere l'inizializzazione all'interno della funzione rendendo il codice più leggibile.

```
private void setSensor() {  
    //Ambient temperature sensor  
    if ((sensorManager.getDefaultSensor(Sensor.TYPE_AMBIENT_TEMPERATURE))  
        != null) {  
        //AmbientTemperature sensor is present  
        temperature = sensorManager  
            .getDefaultSensor(Sensor.TYPE_AMBIENT_TEMPERATURE);  
        isAmbientTempPresent = true;  
    } else {  
        //AmbientTemperature sensor is not present  
        tempValue.setText("Not Present");  
        isAmbientTempPresent = false;  
    }  
  
    //Pressure sensor  
    if ((sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE)) != null) {  
        //Pressure sensor is present  
        pressureSensor =  
            sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);  
        isPressureSensorPresent = true;  
    } else {  
        //Pressure sensor is not present  
        pressureValue.setText("Not Present");  
        isPressureSensorPresent = false;  
    }  
  
    //Humidity sensor  
    if ((sensorManager.getDefaultSensor(Sensor.TYPE_RELATIVE_HUMIDITY))  
        != null) {  
        //Humidity sensor is present  
        humiditySensor =  
            sensorManager.getDefaultSensor(Sensor.TYPE_RELATIVE_HUMIDITY);  
        isHumiditySensorPresent = true;  
    }  
}
```

```

    } else {
        //Humidity sensor is not present
        humidityValue.setText("Not Present");
        isHumiditySensorPresent = false;
    }
}

```

Possiamo notare che l'interrogazione è molto simile a quella utilizzata per il contapassi, l'unica differenza risiede nella mancanza dell'avvio del *.registerListener* all'interno del ciclo "if-else". Questo avviene perché a differenza del contapassi gli altri sensori devono funzionare soltanto quando i sensori esterni, quindi il modulo, non sono collegati. Siccome l'avvio/stop del *Listener* di eventi deve avvenire più volte all'interno del codice, sono state create due semplici funzioni che richiamino l'intero codice del *.registerListener* per ogni sensore.

```

//Function to unregister listener of sensor that i choose
private void unregisterListenerSensor(){
    if (isAmbientTempPresent) {
        sensorManager.unregisterListener(this, temperature);
    }
    if (isPressureSensorPresent) {
        sensorManager.unregisterListener(this, pressureSensor);
    }
    if (isHumiditySensorPresent) {
        sensorManager.unregisterListener(this, humiditySensor);
    }
}

//Function to unregister listener of sensor that i choose
private void registerListenerSensor(){
    if(isAmbientTempPresent){
        //Start listen event for temperature
        sensorManager.registerListener(this, temperature,
            SensorManager.SENSOR_DELAY_NORMAL);
    }
    if(isPressureSensorPresent){
        //Start listen event for pressure
        sensorManager.registerListener(this, pressureSensor,
            SensorManager.SENSOR_DELAY_NORMAL);
    }
    if(isHumiditySensorPresent){
        //Start listen event for humidity
        sensorManager.registerListener(this, humiditySensor,
            SensorManager.SENSOR_DELAY_NORMAL);
    }
}

```

Ora è possibile intuire in quali punti dovranno essere richiamate queste funzioni, in particolare dovrà essere stoppato il *register* di questi sensori quando il modulo bluetooth viene collegato all'applicazione; viceversa quando scollegato. Per gestire questo le due funzioni create sono state propriamente richiamate all'interno del codice creato dal mio collega che gestisce la connessione/disconnessione del modulo; in particolare in questa sezione di codice viene usata una variabile logica (*boolean*) per capire se il modulo è connesso o no. Quindi la variabile *connectedToGatt* viene settata a valore logico alto soltanto quando il modulo è connesso, viceversa a valore logico basso quando disconnesso. Questo discorso è molto utile poiché questi sensori non necessitano di funzionare anche quando l'applicazione viene messa in pausa, cosa invece utile per il contapassi, pertanto si avrà il bisogno di stoppare il *register* quando l'app viene messa in pausa e viceversa quando riavviata; per fare questo vengono modificate le funzioni di Android (*onPause()* per l'uscita momentanea dall'app e *onResume()* per il riavvio).

```
protected void onResume() {
    super.onResume();
    //Restart register
    if (!connectedToGatt) {
        registerListenerSensor();
    }
}

protected void onPause() {
    super.onPause();
    if (!connectedToGatt) {
        unregisterListenerSensor();
    }
}
```

A questo punto basterà implementare l'ascolto del cambiamento di valore per ogni sensore, allo stesso modo di come è stato implementato per il contapassi, gestendo quindi la visualizzazione del valore nella schermata principale.

```
public void onSensorChanged(SensorEvent sensorEvent) {
    if(sensorEvent.sensor == pedometer){
        stepDetect = (int) (stepDetect + sensorEvent.values[0]);
        pedometerValue.setText(String.valueOf(stepDetect));
    }
    else if(sensorEvent.sensor == temperature){
        tempValueSensor = sensorEvent.values[0];
        tempValue.setText(String.format("%.2f C", tempValueSensor));
    }
    else if(sensorEvent.sensor == pressureSensor){
        pressureValueSensor = sensorEvent.values[0];
    }
}
```



```
        pressureValue.setText(String.format("%.2f hPa",
            pressureValueSensor));
    }
    else if(sensorEvent.sensor == humiditySensor){
        humidityValueSensor = sensorEvent.values[0];
        humidityValue.setText(String.format("%.2f %",
            humidityValueSensor));
    }
}
```

Ricordando che la presenza o meno di ogni sensore è fortemente dipendente dal modello del dispositivo e che l'utilizzo del modulo serve proprio a completare questa mancanza, il risultato della rappresentazione della schermata principale è visualizzato nella figura

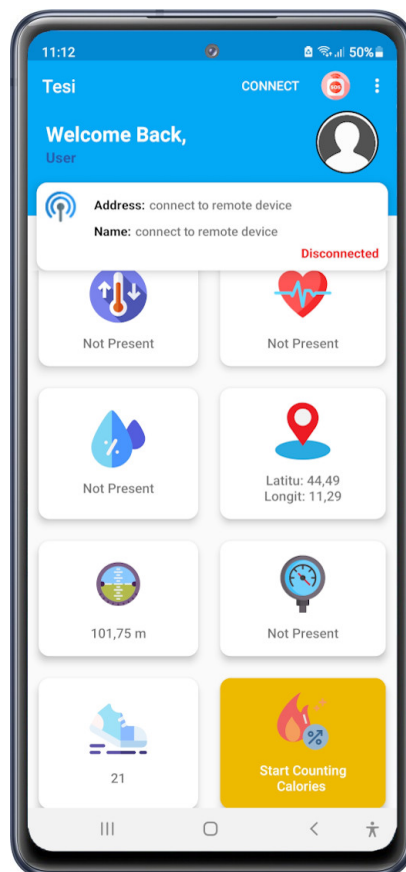


Figura 3.3: Schermata principale per Samsung Galaxy S20FE

3.2 Passaggio dati fra Activity

Dopo aver acquisito e stampato i valori dei sensori presenti sul dispositivo abbiamo la necessità di inviare questi dati al broker centrale dell'architettura Publish/Subscribe tramite protocollo MQTT. Nella versione precedente, realizzata dal collega, i dati venivano inviati su richiesta dell'utente tramite la pressione di un bottone *Send Data To Cloud*

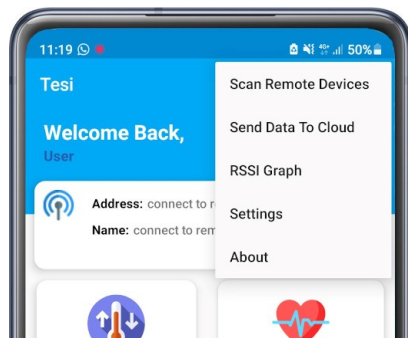


Figura 3.4: Opzioni menù

pertanto si è deciso di utilizzare questa funzione aggiungendo l'invio dei dati precedentemente descritti; ovviamente permane l'obbligo dell'invio dei dati soltanto se il modulo non è collegato, altrimenti rimane ovvio che hanno la precedenza i dati acquisiti da sensori esterni. Nel momento in cui si preme *Send Data To Cloud* l'applicazione avvierà un servizio per entrare quindi in una nuova activity: **MqttService.java**; siccome ci si trova in una nuova classe i dati precedentemente acquisiti dai sensori non risulteranno visibili pertanto si è preferito passare i dati da un activity a un servizio invece di implementare nuovamente tutta l'acquisizione descritta nella sezione precedente, così da avere un codice più snello e non ridondante.

È importante sapere che quando si entra nella selezione menù (:) a basso livello si entra in una porzione di codice, in particolare una funzione (*onOptionsItemSelected(@NonNull MenuItem item)*) all'interno della classe *SensorInfo*, nella quale ci si interroga su quale bottone è stato premuto dall'utente per avviare le varie attività o servizi come nel caso d'interesse.

3.2.1 Panoramica passaggio dati

Nel caso in questione si deve gestire un passaggio di dati fra un activity (*SensorInfo*) e un servizio (*MqttService*), per gestire questi passaggi esistono dei metodi della classe *Intent* purché si implementi il package

android.content.Intent .Nel caso in questione si gestisce l'invio dei dati utilizzando il metodo **putExtra(String name, DataType value)** dove il valore che si vuole passare al nuovo *Intent* è rappresentato da "value" e dichiarato con il proprio tipo di dato, mentre con "name" viene indicato il nome del dato extra purché sia preceduto dal proprio package definito nel file *AndroidManifest.xml* sul quale si sta lavorando. Quindi per utilizzare questo metodo bisogna prima inizializzare e definire l'*Intent* che verrà collegato al servizio che si vorrà avviare[17]; per fare un esempio

```
public class Example {
    Intent exampleIntent;
    int data = 10;
    String name = package_name + "data";

    protected void onCreate(Bundle savedInstanceState){
        exampleIntent = new Intent(this, secondClassToBeCalled.class);
        exampleIntent.putExtra(name, data);
    }
}
```

In questo modo verrà passato il valore *data* con il proprio nome definito da *name*. Successivamente basterà quindi avviare il servizio.

```
startService(exampleIntent);
```

È stato implementato l'avvio di un servizio e non di un activity così che l'invio dei dati possa avvenire in background.

La recezione all'interno del servizio viene invece gestita dal metodo **getDataTypeExtra(String name, DataType defaultValue)** con "DataType" il tipo di dato che si è passato, quindi *int* per l'esempio precedente, e con *defaultValue* il valore che si vuole passare alla variabile nel caso in cui la recezione non sia andata a buon fine per un qualunque motivo[17]. Quindi continuando l'esempio precedente i dati verranno ricevuti e potranno essere gestiti all'interno di una particolare funzione di Android: **onStartCommand(Intent intent, int flags, int startId)**, che verrà chiamata all'avvio del servizio.

```
public class secondClassToBeCalled extends Service{
    String name = package_name + "data";
    public void onCreate(){
        ...
    }
    public int onStartCommand(Intent intent, int flags, int startId) {
        int data_received = intent.getIntExtra(name, 999);
    }
}
```

3.2.2 Invio

Nel caso d'interesse, quindi nell'activity *SensorInfo* per gestire l'invio dei dati si implementa quanto descritto precedentemente all'interno della funzione che gestisce la selezione menù, così che venga avviato l'invio e il servizio al momento della pressione del bottone *Send Data To Cloud*.

```
public class SensorsInfo ...{
    Intent mqttService;
    public boolean onOptionsItemSelected(@NonNull MenuItem item) {
        switch (item.getItemId()) {
            ...
            case (R.id.action_mqtt):
                mqttService = new Intent(this, MqttService.class);
                mqttService
                    .putExtra(StaticResources.EXTRA_CONNECTED_TO_GATT,
                        connectedToGatt);
                mqttService
                    .putExtra(StaticResources.EXTRA_LOCATION_PERMISSION,
                        locationPermission);
                if(isStepSensorPresent){
                    mqttService
                        .putExtra(StaticResources.EXTRA_PEDOMETER_VALUE_SENSOR,
                            stepDetect);
                }
                if(!connectedToGatt) {
                    if (isAmbientTempPresent) {
                        mqttService
                            .putExtra(StaticResources.EXTRA_TEMP_VALUE_SENSOR,
                                tempValueSensor);
                    }
                    if (isHumiditySensorPresent) {
                        mqttService
                            .putExtra(StaticResources.EXTRA_HUMIDITY_VALUE_SENSOR,
                                humidityValueSensor);
                    }
                    if (locationPermission) {
                        mqttService
                            .putExtra(StaticResources.EXTRA_LATITUDE_VALUE_SENSOR,
                                latitude);
                        mqttService
                            .putExtra(StaticResources.EXTRA_LONGITUDE_VALUE_SENSOR,
                                longitude);
                        mqttService
                            .putExtra(StaticResources.EXTRA_ALTITUDE_VALUE_SENSOR,
                                altitudeValueSensor);
                    }
                }
            }
        }
    }
}
```

```

        if (isPressureSensorPresent) {
            mqttService
                .putExtra(StaticResources.EXTRA_PRESSURE_VALUE_SENSOR,
                    pressureValueSensor);
        }
    }
    try {
        startService(mqttService);
    } catch (IllegalStateException | SecurityException e) {
        e.printStackTrace();
    }
    return true;
}
}
}

```

In questo modo si può notare che il contapassi viene inviato indipendentemente dalla connessione del modulo ovviamente se presente sul dispositivo, al contrario gli altri sensori vengono inviati soltanto se *connectedToGatt* risulta a valore logico basso; si può notare che questo avviene anche per la posizione (latitudine, longitudine) perché al contrario della visualizzazione l'invio della posizione se il modulo è connesso corrisponde al valore dettato dal modulo bluetooth; per questi sensori l'invio dipende giustamente anche dalla loro presenza fisica sul dispositivo o dall'ottenimento dei permessi dall'utente.

3.2.3 Ricezione

Per la ricezione è stata scritta la porzione di codice all'interno del servizio dedicato, quindi si entra all'interno del servizio *MqttService* e in particolare si utilizza il modello precedentemente descritto.

```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    boolean connectedToGatt = intent
        .getBooleanExtra(StaticResources.EXTRA_CONNECTED_TO_GATT, false);
    boolean locationPermission = intent
        .getBooleanExtra(StaticResources.EXTRA_LOCATION_PERMISSION, false);
    if (locationPermission) {
        latitude = intent
            .getFloatExtra(StaticResources.EXTRA_LATITUDE_VALUE_SENSOR,
                360);
        longitude = intent
            .getFloatExtra(StaticResources.EXTRA_LONGITUDE_VALUE_SENSOR,
                360);
    }
}

```

```
        altitudeValueSensor = intent
        .getFloatExtra(StaticResources.EXTRA_ALTITUDE_VALUE_SENSOR,
            9999);
    }
    float tempValueSensor = intent
    .getFloatExtra(StaticResources.EXTRA_TEMP_VALUE_SENSOR, -999);
    float humidityValueSensor = intent
    .getFloatExtra(StaticResources.EXTRA_HUMIDITY_VALUE_SENSOR, -1);
    float pressureValueSensor = intent
    .getFloatExtra(StaticResources.EXTRA_PRESSURE_VALUE_SENSOR, 0);
    int stepDetect = intent
    .getIntExtra(StaticResources.EXTRA_PEDOMETER_VALUE_SENSOR, -1);
```

Si può notare che la posizione e di conseguenza l'altitudine devono essere ricevuti soltanto se *locationPermission* è a valore logico alto, ovvero se l'utente ha concesso i permessi; diversamente per gli altri sensori. Ogni sensore presenta il suo *defaultValue* utile a capire se l'invio/recezione è andata a buon fine.

- *connectedToGatt* = false
I dati vengono inviati per ogni caso d'errore.
- *locationPermission* = false
La posizione viene inviata solo se concesso.
- *latitudine/longitudine* = 360
La latitudine massima è 90° mentre la longitudine massima è 180° pertanto si avrà così un flag d'errore.
- *altitudine* = 9999
Il valore massimo è 8850mt (Monte Everest) così che sia riconoscibile l'errore.
- *temperatura* = -999
Valore impossibile nella realtà fisica.
- *umidità* = -1
Il dato acquisito è in percentuale (0-100).
- *pressione* = 0
Impossibile raggiungere pressione assoluta a 0.
- *contapassi* = -1
Il valore dei passi parte a contare da 0.

3.3 Invio dati con protocollo MQTT

Per concludere il progetto questi dati acquisiti devono essere inviati al server, quindi il broker centrale (Raspberry), tramite il protocollo MQTT così che sia possibile elaborarli.

3.3.1 Panoramica libreria Paho

Il servizio Paho Android è una libreria client MQTT scritta da Eclipse Foundation in Java per lo sviluppo di applicazioni su Android. Il progetto Paho è stato creato per fornire implementazioni open source affidabili di protocolli di messaggistica aperti e standard destinati ad applicazioni nuove, esistenti ed emergenti per Internet of Things (IoT).[\[18\]](#)

La connessione MQTT è incapsulata all'interno di un servizio Android che viene eseguito in background, mantenendolo attivo quando l'applicazione passa da un'activity all'altra. Questo livello di astrazione è necessario per poter ricevere messaggi MQTT in modo affidabile.

3.3.2 Installazione

Per l'installazione delle librerie è necessario implementarle nel file *build.gradle* (*Module*) aggiungendole come dependencies.

```
repositories {  
    maven {  
        url "https://repo.eclipse.org/content/repositories/paho-releases/"  
    }  
    maven {  
        url "https://repo.eclipse.org/content/repositories/paho-snapshots/"  
    }  
}  
  
dependencies {  
    implementation  
        'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'  
    implementation  
        'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.0'  
}
```

La prima parte aggiunge il repository di rilascio di Paho alla configurazione di gradle, in modo che Gradle sia in grado di trovare il pacchetto JAR del servizio Android Paho; inoltre viene implementata l'ultima versione di Snapshots. La seconda parte aggiunge il servizio Paho Android e client come dipendenza all'applicazione.[\[19\]](#)

3.3.3 Connessione

A questo punto è necessario instaurare una connessione con il broker, per prima cosa il servizio deve essere dichiarato nell'*AndroidManifest.xml* inserendo anche i seguenti permessi:

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
...
<application
    <service
        android:name="org.eclipse.paho.android.service.MqttService" />
</application>
```

Successivamente all'interno del servizio MqttService viene gestito il codice utile a stabilire la connessione, in particolare viene dichiarato il cliente che nel caso in questione risulterà essere un *Publisher*. Poi viene definito il *ServerUri* ovvero l'indirizzo IP del broker (Raspberry) all'interno della rete nella quale si sviluppa l'architettura, seguito dalla porta usata dal protocollo. Inoltre si definisce il tipo di memoria. A questo punto è possibile istanziare l'ID per poi inserirlo nell'inizializzazione del cliente come connessione asincrona: **new MqttAsyncClient()**.

```
public class MqttService extends Service {
    MqttAsyncClient client;
    private final String serverUri = "tcp://192.168.1.2:1883";
    private MemoryPersistence persistence;
    public void onCreate() {
        super.onCreate();
        String clientId = MqttClient.generateClientId();

        try {
            client = new MqttAsyncClient(serverUri, clientId, persistence);
            IMqttToken token = client.connect();
            client.setCallback(new MqttCallback() {
                @Override
                public void connectionLost(Throwable cause) {
                    Log.e(TAG, "Connection Lost");
                }

                @Override
                public void messageArrived(String topic, MqttMessage
                    message) throws Exception {
                    Log.e(TAG, "Message arrived");
                }
            });
        }
    }
}
```



```
        @Override
        public void deliveryComplete(IMqttDeliveryToken token) {
            Log.e(TAG, "Delivery Complete");
        }
    });
} catch (MqttException ex){
    ex.printStackTrace();
}
}
```

3.3.4 Opzioni

Per la connessione è possibile settare una serie di opzioni utili a rendere quest'ultima più sicura e precisa, in particolare nel progetto di tesi viene settato il nome utente e la password, la versione del protocollo MQTT utilizzato, il tipo di connessione e la riconnessione automatica.

```
public class MqttService extends Service {
    private final String user = "andrea";
    private final String pwd = "1234";

    public void onCreate() {
        super.onCreate();
        String clientId = MqttClient.generateClientId();

        try {
            client = new MqttAsyncClient(serverUri, clientId, persistance);
            MqttConnectOptions mqttConnectOptions = new
                MqttConnectOptions();
            mqttConnectOptions.setMqttVersion
                (MqttConnectOptions.MQTT_VERSION_3_1_1);
            mqttConnectOptions.setCleanSession(true);
            mqttConnectOptions.setAutomaticReconnect(true);
            mqttConnectOptions.setUserName(user);
            mqttConnectOptions.setPassword(pwd.toCharArray());
            IMqttToken token = client.connect(mqttConnectOptions);

            ...
        } catch (MqttException ex){
            ex.printStackTrace();
        }
    }
}
```

3.3.5 Pubblicazione

Come ultimo passo rimane la pubblicazione dei messaggi, in particolare per questo passaggio è stata creata in precedenza dal mio collega una funzione che gestisce l'invio dei dati così da poter richiamare la funzione tutte le volte che si voglia pubblicare un messaggio rendendo quindi il codice molto più snello.

```
void pub(String topic, String payload, int QoS){
    byte[] encodedPayload;
    try {
        encodedPayload = payload.getBytes(StandardCharsets.UTF_8);
        MqttMessage message = new MqttMessage(encodedPayload);
        message.setQos(QoS);
        client.publish(topic, message);
        Toast.makeText(this.getContext(), "Data sent",
            Toast.LENGTH_SHORT).show();
    } catch (MqttException e) {
        e.printStackTrace();
    }
}
```

Per tale funzione bisogna passare in ingresso il *topic* del messaggio, la stringa *payload* contenente il valore quindi il messaggio che si vuole inviare e il *QoS*.

A questo punto è possibile richiamare la funzione ogni qualvolta che si vuole inviare il dato, quindi si richiamerà per ogni dato ottenuto dai sensori; siccome i dati vengono ricevuti dall'activity precedente nella funzione *onStartCommand()*, come descritto nella sezione precedente, allora sarà in questa funzione che pubblicheremo tutti i messaggi ognuno con il suo topic differenziandoli dai topic dei dati ricevuti dal modulo bluetooth così che sia visibile correttamente come sia stato acquisito il dato. In particolare si inviano i seguenti topic, sempre se vi è la presenza fisica del sensore:

- contapassi - "sensorsDevice/pedometer"
- temperatura - "sensorsDevice/temp"
- umidità - "sensorsDevice/humidity"
- pressione - "sensorsDevice/pressure"
- altitudine - "sensorsDevice/altitude"
- latitudine - "sensorsDevice/latitude"
- longitudine - "sensorsDevice/longitude"

```
public int onStartCommand(Intent intent, int flags, int startId) {7
    if(stepDetect != -1){
        String stepDetectString = String.valueOf(stepDetect);
        pub(StaticResources.PEDOMETER_SENSOR_TOPIC, stepDetectString,
            StaticResources.QOS_0);
    }
    if(connectedToGatt){
        //Send data acquired from bluetooth
        ...
    }else{
        if(tempValueSensor != -999){
            String tempValueSensorString = String.valueOf(tempValueSensor);
            pub(StaticResources.TEMP_SENSOR_TOPIC, tempValueSensorString,
                StaticResources.QOS_0);
        }
        if(humidityValueSensor != -1){
            String humidityValueSensorString =
                String.valueOf(humidityValueSensor);
            pub(StaticResources.HUMIDITY_SENSOR_TOPIC,
                humidityValueSensorString, StaticResources.QOS_0);
        }
        if(pressureValueSensor != 0){
            String pressureValueSensorString =
                String.valueOf(pressureValueSensor);
            pub(StaticResources.PRESSURE_SENSOR_TOPIC,
                pressureValueSensorString, StaticResources.QOS_0);
        }
        if(locationPermission && altitudeValueSensor != 9999){
            String altitudeValueSensorString =
                String.valueOf(altitudeValueSensor);
            pub(StaticResources.ALTITUDE_SENSOR_TOPIC,
                altitudeValueSensorString, StaticResources.QOS_0);
        }
        if(locationPermission && latitude != 360 && longitude != 360){
            String positionString = String.valueOf(latitude);
            pub(StaticResources.LATITUDE_TOPIC, positionString,
                StaticResources.QOS_0);
            positionString = String.valueOf(longitude);
            pub(StaticResources.LONGITUDE_TOPIC, positionString,
                StaticResources.QOS_0);
        }
    }
}
```

Capitolo 4

Raspberry

Questo capitolo descrive tutti i passaggi utili a realizzare la parte *server* dell'architettura di progetto, quindi sarà incentrato sul broker centrale dell'architettura Publish/Subscribe realizzato con dispositivo Raspberry introdotto al capitolo 2 (Tecnologie utilizzate). In particolare verranno descritti tutti i passaggi utili alla configurazione e alla visualizzazione dei messaggi ricevuti dal client publisher Android.

4.1 Installazione

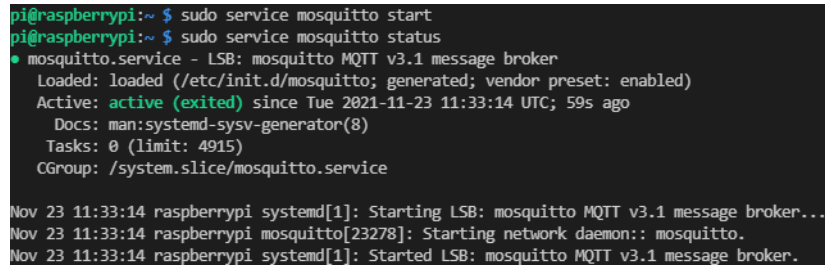
Per la realizzazione del server sono stati utilizzati i pacchetti Mosquitto di Eclipse creando un broker di messaggi ideato dalla Eclipse Foundation che implementa il protocollo MQTT in versione 5.0, 3.1.1 (quella utilizzata) e 3.1. [20] Questo servizio è fortemente utile poiché implementabile nel sistema operativo oggetto di questa tesi, il *Raspberry Pi OS*, fornendo anche una serie di librerie in C.

Una volta che il Raspberry Pi3B+ è stato alimentato e collegato tramite il proprio cavo HDMI ad un monitor o tramite SSH ad un portatile esterno, è possibile procedere con l'installazione dei pacchetti **Mosquitto** tramite linea di comando.

```
sudo apt-get update
sudo apt-get install mosquitto
sudo apt-get install mosquitto-clients
```

A questo punto se tutto è andato correttamente a buon fine i pacchetti risulteranno installati pertanto sarà possibile utilizzare le risorse di Mosquitto. Ad esempio sarà possibile avviare, stoppare e visualizzare lo stato del servizio.

```
sudo service mosquitto start
sudo service mosquitto status
```



```
pi@raspberrypi:~$ sudo service mosquitto start
pi@raspberrypi:~$ sudo service mosquitto status
● mosquitto.service - LSB: mosquitto MQTT v3.1 message broker
   Loaded: loaded (/etc/init.d/mosquitto; generated; vendor preset: enabled)
   Active: active (exited) since Tue 2021-11-23 11:33:14 UTC; 59s ago
     Docs: man:systemd-sysv-generator(8)
    Tasks: 0 (limit: 4915)
   CGroup: /system.slice/mosquitto.service

Nov 23 11:33:14 raspberrypi systemd[1]: Starting LSB: mosquitto MQTT v3.1 message broker...
Nov 23 11:33:14 raspberrypi mosquitto[23278]: Starting network daemon:: mosquitto.
Nov 23 11:33:14 raspberrypi systemd[1]: Started LSB: mosquitto MQTT v3.1 message broker.
```

Figura 4.1: Stato del servizio dopo l'avvio

```
sudo service mosquitto stop
```

4.2 Configurazione

Dopo aver stoppato il servizio è possibile ora configurare il file di configurazione utile ad avviare correttamente il servizio, come verrà descritto successivamente. Come primo passo è utile entrare all'interno della cartella di destinazione e successivamente digitare il comando per creare il file.

```
cd /etc/mosquitto
sudo nano fileConfig.conf
```

A questo punto verrà aperto il file di configurazione dove bisognerà inserire le varie opzioni, in particolare verrà settata la porta (1883) che dovrà essere utilizzata dal protocollo e le configurazioni per definire che la connessione è permessa solo con credenziali dicendo correttamente il percorso e il file contenente le credenziali.

```
listener 1883
allow_anonymous false
password_file /etc/mosquitto/password
```

A questo punto basterà premere *Ctrl+x*, digitare *y* e premere *Invio*; in questo modo il file verrà salvato. Per vedere se il file è stato creato e scritto correttamente si può digitare su linea di comando i seguenti comandi;

```
ls
cat fileConfig.conf
```

quello che si dovrà avere risulterà come rappresentato in figura.

```
pi@raspberrypi:/etc/mosquitto $ sudo nano fileConfig.conf
pi@raspberrypi:/etc/mosquitto $ ls
ca_certificates certs conf.d fileConfig.conf
pi@raspberrypi:/etc/mosquitto $ cat fileConfig.conf
listener 1883
allow_anonymous false
password_file /etc/mosquitto/password
```

Figura 4.2: Visualizzazione del fileConfig.conf

A questo punto sarà necessario settare il nome utente e la password, ricordando dal capitolo precedente che l'*username* dovrà essere **"andrea"** mentre la password che verrà richiesta due volte per conferma dopo aver avviato il comando dovrà essere **"1234"**; quindi basterà digitare il comando *mosquitto_passwd* seguito dal nome del file di password e il nome utente.

```
sudo mosquitto_passwd -c password andrea
```

È importante che il nome del file di password sia uguale a quello scritto come percorso all'interno del fileConfig.conf, ovvero password.

4.3 Avvio del servizio

A questo punto è possibile riavviare il servizio, in particolare sarà necessario avviarlo con annesso il file di configurazione.

```
sudo service mosquitto start
mosquitto -c fileConfig.conf
```

A questo punto sarà possibile vedere tutte le connessioni che avvengono con il broker.

```
pi@raspberrypi:/etc/mosquitto $ sudo service mosquitto start
pi@raspberrypi:/etc/mosquitto $ mosquitto -c fileConfig.conf
1637670736: mosquitto version 1.4.10 (build date Tue, 26 Oct 2021 22:24:15 +0200) starting
1637670736: Config loaded from fileConfig.conf.
1637670736: Opening ipv4 listen socket on port 1883.
1637670736: Opening ipv6 listen socket on port 1883.
1637670938: New connection from 192.168.1.7 on port 1883.
1637670938: New client connected from 192.168.1.7 as paho5473316237077 (c1, k60, u'andrea').
[]
```

Figura 4.3: Visualizzazione delle connessioni al broker

4.4 Visualizzazione dati

Come ultimo passo rimane da visualizzare i dati inviati dall'applicazione Android al broker, è importante ricordare che l'invio dei dati è fortemente dipendente dalla presenza fisica dei sensori sul dispositivo. Come abbiamo visto nel capitolo 3 (Android) infatti i dati all'interno del servizio `MqttService` vengono ricevuti soltanto se il sensore è presente, quindi in caso contrario le varie variabili assumono i valori di default evitando di essere inviati. All'interno del broker è possibile iscriversi ad ogni topic ma se il dato non viene inviato giustamente non verrà ricevuto e visualizzato, rimanendo quindi in ascolto fino a che non si decide di terminare il processo.

```
pi@raspberrypi:~ $ mosquitto_sub -t "sensorsDevice/temp" -u "andrea" -P "1234"
```

Figura 4.4: Visualizzazione del topic `sensorsDevice/temp`

La stessa cosa avviene se ci si sottoscrive ad un topic differente da quelli inviati.

```
pi@raspberrypi:~ $ mosquitto_sub -t "Topic_null" -u "andrea" -P "1234"
```

Figura 4.5: Visualizzazione di un topic inventato

Possiamo quindi notare che il comando da terminale per iscriversi ad un topic è definito da `mosquitto_sub` definendo il topic al quale ci si vuole iscrivere;

```
mosquitto_sub -t "TOPIC" -u "andrea" -P "1234"
```

è importante inserire il nome utente e le credenziali per poter accedere al servizio, altrimenti in caso contrario il comando verrà negato ritentando la richiesta finché non si forza il servizio a terminare premendo `Ctrl+c`.

```
pi@raspberrypi:~ $ mosquitto_sub -t "sensorsDevice/altitude"
Connection Refused: not authorised.
Connection Refused: not authorised.
Connection Refused: not authorised.
Connection Refused: not authorised.
```

Figura 4.6: Comando negato per autenticazioni errate

CAPITOLO 4. RASPBERRY

Rimane quindi la possibilità di sottoscrivere a diversi topic così da visualizzare tutti i messaggi inviati, per fare questo è necessario aprire più terminali ed in particolare uno per ogni topic.

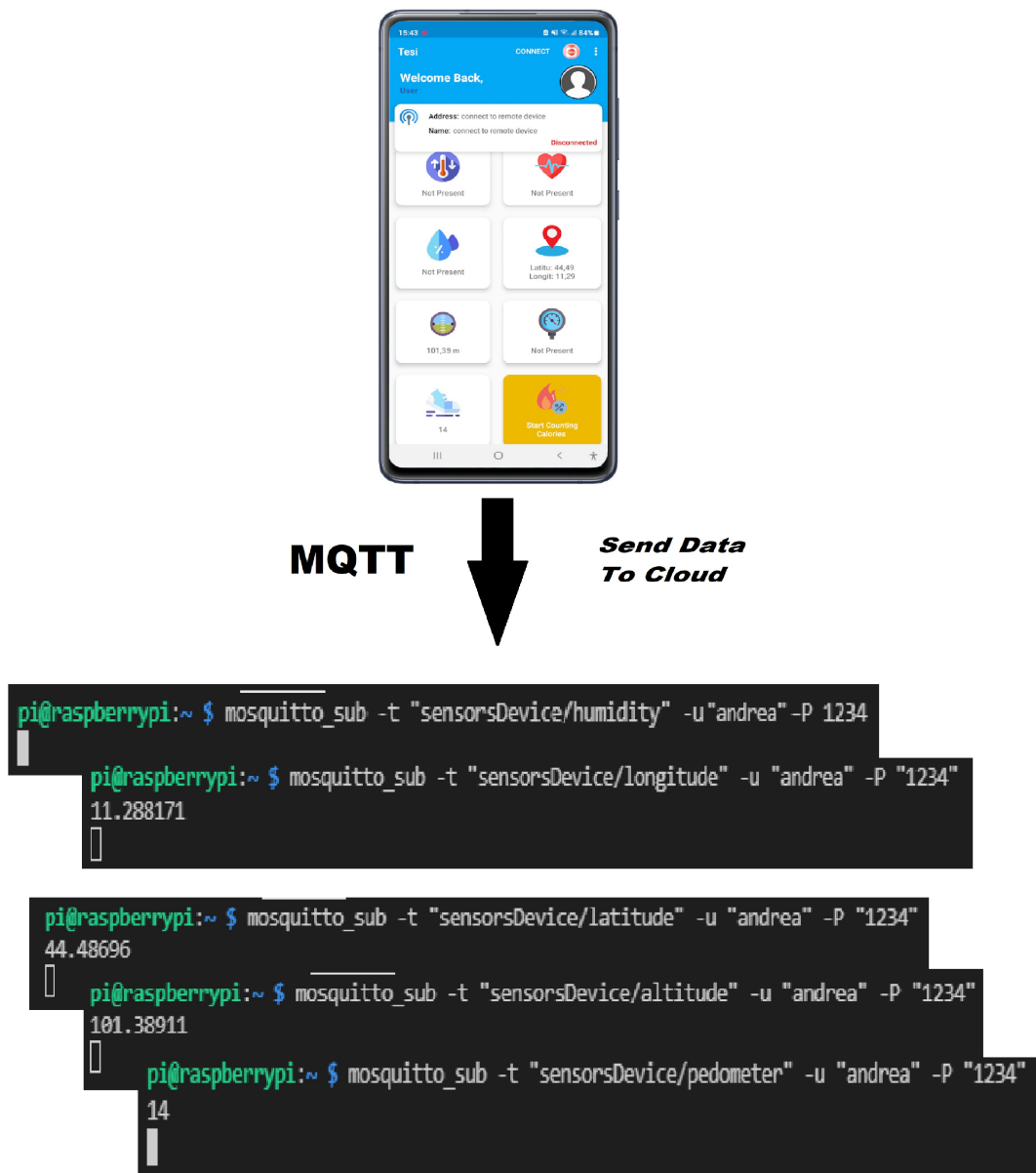
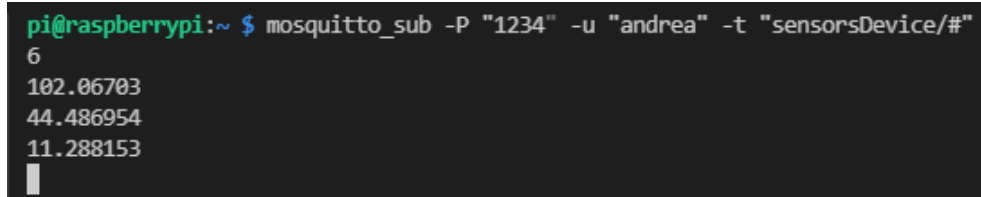


Figura 4.7: Visualizzazione multi-topic dei dati Android

Vi è inoltre un' ulteriore possibilità ovvero quella di iscriversi al topic primario, in questo modo il broker invierà a chi ha deciso di sottoscrivere a questo topic tutti i dati ricevuti. Per rendere possibile questa ulteriore configurazione basterà utilizzare il comando *mosquitto_sub* definendo come topic **sensorsDevice/#**.

A terminal window on a Raspberry Pi. The prompt is 'pi@raspberrypi:~ \$'. The command entered is 'mosquitto_sub -P "1234" -u "andrea" -t "sensorsDevice/#"'. The output shows four lines of data: '6', '102.06703', '44.486954', and '11.288153'. A cursor is visible at the end of the last line.

```
pi@raspberrypi:~ $ mosquitto_sub -P "1234" -u "andrea" -t "sensorsDevice/#"  
6  
102.06703  
44.486954  
11.288153  
█
```

Figura 4.8: Visualizzazione topic primario

Conclusioni

In questo elaborato di tesi è stato realizzato un sistema client-server per applicazioni IoT implementabili in scenari smart-city con l'intento di migliorare la sicurezza e la mobilità, sfruttando il protocollo di messaggistica MQTT. L'architettura è basata su publish/subscribe, in particolare con questo progetto di tesi viene realizzata la parte di publisher con l'applicazione Android acquisendo i dati dai sensori presenti sul dispositivo e inviandoli con protocollo MQTT al broker centrale; la parte di subscriber viene invece gestita direttamente all'interno del broker centrale tramite le librerie mosquitto fornite da Eclipse.

Questo progetto risulta essere adattabile a possibili sviluppi futuri, in particolare sarà implementato il lato subscriber direttamente all'interno dell'applicazione Android così che tutti gli utenti possano pubblicare i loro dati ma contemporaneamente vedere quelli degli altri utenti. Poiché l'assenza di alcuni sensori sul dispositivo porta inefficienza, oltre all'implementazione del modulo bluetooth già sviluppata, si potrebbe realizzare l'acquisizione dei dati mancanti direttamente da internet tramite la posizione acquisita. Inoltre sarebbe possibile gestire l'acquisizione dei dati accelerometro e giroscopio così da creare una soglia oltre la quale l'applicazione invia sempre tramite protocollo MQTT un messaggio di alta priorità così da poter migliorare la sicurezza stradale degli utenti deboli veicolari.

Bibliografia

- [1] (2020) Smart city: cos'è e come funziona. LUMI4INNOVATION. [Online]. Available: <https://www.lumi4innovation.it/smart-city-cosè-come-funziona-caratteristiche-ed-esempi-in-italia/>
- [2] Smart cities. European Commission. [Online]. Available: https://ec.europa.eu/info/eu-regional-and-urban-development/topics/cities-and-urban-development/city-initiatives/smart-cities_en
- [3] R. Ammar and S. Samer, *Internet of Things From Hype to Reality*. Springer, 2019.
- [4] A. M. M. M. M.A., *Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications*. IEEE, 2015.
- [5] B. Mishra and A. Kertesz, *The Use of MQTT in M2M and IoT Systems: A Survey*. IEEE, 2020.
- [6] (2020) Mqtt: Use cases. OASIS. [Online]. Available: <https://mqtt.org/use-cases/#automotive>
- [7] (2021) Emq helps saic volkswagen building iov platform. EMQ Technologies Co. [Online]. Available: <https://www.emqx.io/blog/emqx-in-volkswagen-iov>
- [8] (2021) Car-sharing application relies on hivemq for reliable connectivity. HiveMQ GmbH. [Online]. Available: <https://www.hivemq.com/case-studies/bmw-mobility-services/>
- [9] (2021) Telemetry use case: Home energy monitoring and control. IBM. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.pro.doc/q002790_.htm
- [10] *Platform Architecture*, Google, 2020. [Online]. Available: [https://developer.android.com/guide/platform#:~:text=Android%](https://developer.android.com/guide/platform#:~:text=Android%20OS,version,API%20level)

20is%20an%20open%20source,components%20of%20the%20Android%20platform.

- [11] *Meet Android Studio*, Google, 2021. [Online]. Available: <https://developer.android.com/studio/intro>
- [12] *Android è per tutti*. Android. [Online]. Available: https://www.android.com/intl/it_it/everyone/
- [13] GSMA, *The Mobile Economy*. GSMA, 2015.
- [14] (2021) Raspberry pi documentation. Raspberry Pi Foundation. [Online]. Available: <https://www.raspberrypi.org/documentation/>
- [15] (2021) Raspberry pi 3 model b+. Raspberry Pi Foundation. [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/>
- [16] *Sensors Overview*, Google, 2021. [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_overview
- [17] *Intent*, Google, 2021. [Online]. Available: <https://developer.android.com/reference/android/content/Intent>
- [18] *Eclipse Paho Android Service*, Eclipse Foundation. [Online]. Available: <https://www.eclipse.org/paho/index.php?page=clients/android/index.php>
- [19] *Paho Android Service - MQTT Client Library Encyclopedia*, HiveMQ. [Online]. Available: <https://www.hivemq.com/blog/mqtt-client-library-encyclopedia-paho-android-service/>
- [20] *Mosquitto documentation*, Eclipse Foundation. [Online]. Available: <https://mosquitto.org/documentation/>

Ringraziamenti

Vorrei ringraziare il Prof.re Daniele Tarchi per l'opportunità fornita nel redigere questa tesi di laurea in un argomento innovativo e così interessante, desidero ringraziarlo anche per l'attenzione e il tempo dedicatomi.

Vorrei continuare ringraziando i miei genitori Delia, Antonio e mio fratello Marco per avermi sostenuto e per avermi concesso la possibilità di affrontare questo percorso utile per il mio futuro.

Inoltre desidero soprattutto ringraziare Susi per avermi supportato e sopportato durante tutto questo lungo percorso, per essermi stata accanto durante tutti i momenti di difficoltà e per avermi incoraggiato a continuare, grazie. Concludo ringraziando anche i miei amici e il mio coinquilino Federico per le tante ore di studio affrontate insieme.