UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola
di Ingegneria

Corso di Laurea Triennale in
Ingegneria Informatica

Confronto prestazionale tra i protocolli MQTT e HTTP per applicazioni per dispositivi mobili

---

Performance comparison between MQTT and HTTP protocols for mobile applications

Relatore:
Prof. Alessandro Fantechi

Correlatore:
Ing. Walter Nunziati

Candidato:
Steven Alexander Salazar Molina

Anno Accademico 2017-2018

Indice

Introduzione

Col passare del tempo si parla sempre più spesso del mondo Internet of Things, come sappiamo tutti al giorno d'oggi le reti IoT sono ormai una realtà che è già cominciata e continueranno ad espandersi nei prossimi anni. Uno studio fatto da *Statista-The Statistics Portal* [1] prevede, come si osserva in Figura 1, che nel 2025 il numero di dispositivi IoT connessi a Internet supererà leggermente i 75 miliardi, e dunque risulta giustificato realizzare degli studi sulle prestazioni che si ottengono dai diversi protocolli che permettono ai dispositivi di comunicare.

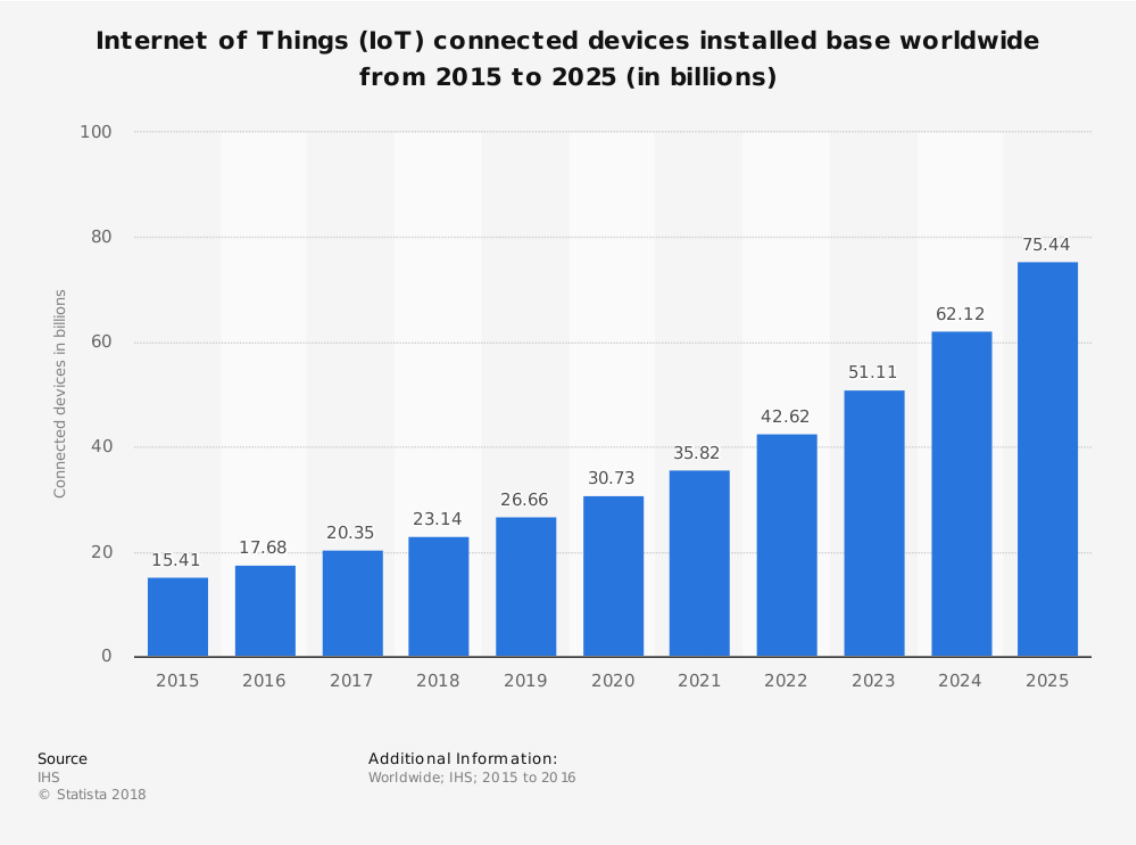


Figura 1: Numero di dispositivi IoT connessi a Internet dal 2015 al 2025.

In particolare, sappiamo che IoT utilizza le infrastrutture, i protocolli e le tecnologie già esistenti. Tuttavia, è importante notare che, poichè questi protocolli non sono stati pensati a lavorare nel mondo IoT, potrebbero implicare un utilizzo non efficiente delle risorse o

potrebbero anche non soddisfare i requisiti di cui le reti IoT hanno bisogno e per questo motivo, risulta necessario introdurre delle alternative [2].
Così come l'internet che conosciamo, le reti IoT lavorano con il modello TCP/IP. Il diagramma in Figura 2 mostra appunto alcuni protocolli che sono già stati introdotti.

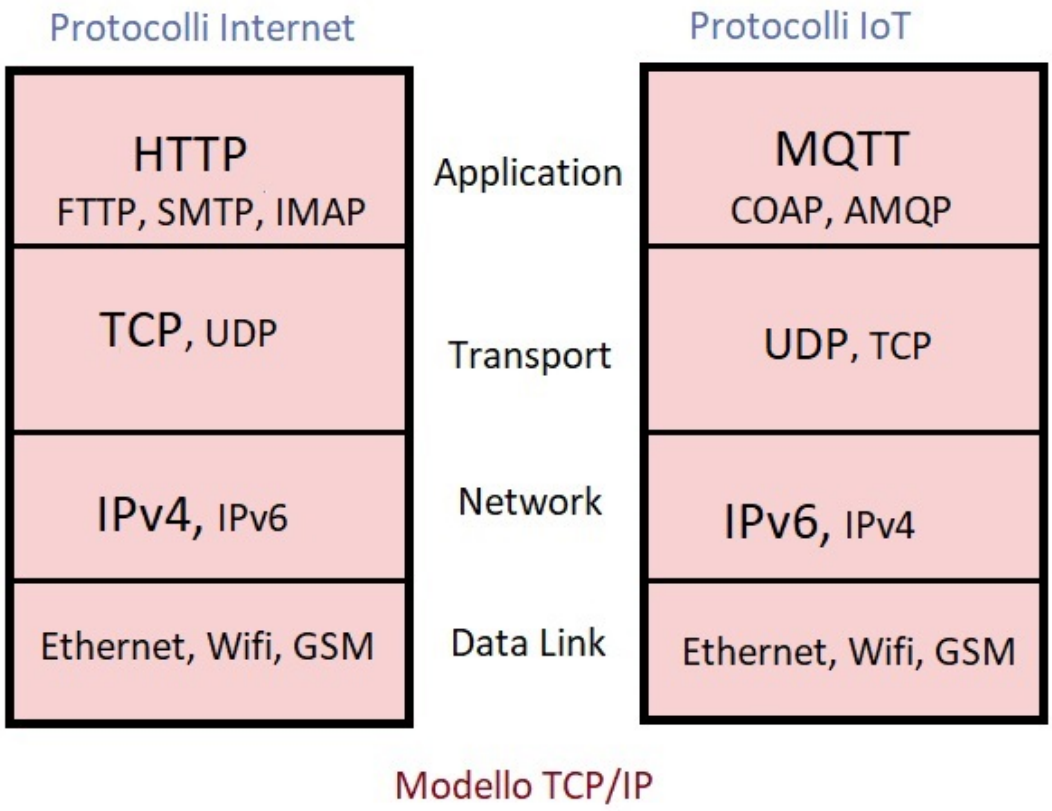


Figura 2: Internet and IOT Protocols.

Nel diagramma la dimensione della fonte utilizzata per i nomi dei protocolli è proporzionale al loro utilizzo, una vista generale mostra che lo sviluppo negli ultimi anni per entrambe le parti è aumentato nell'ultimo livello e il livello più basso, ovvero livello Application e Data Link. Entrando nel particolare, a sinistra osserviamo come il livello Application HTTP è presentato come quello più usato tra i diversi protocolli, osserviamo anche TCP viene comunemente più usato e che IPv4 risulta ancora molto comune nella navigazione normale su internet. A destra invece, viene mostrato MQTT come quello più utilizzato al livello Application, UDP al livello Trasporto e finalmente quando si parla del livello Networking IPv6 è molto più usato.
Il motivo per cui a livello Networking il protocollo IPv6 è molto più usato, è principalmente dovuto all'esaurimento degli indirizzi IPv4. Per quanto riguarda il livello Trasporto il protocollo UDP risulta più usato dato che nel mondo IoT gran parte dei dispositivi sono dei sensori che hanno come priorità la velocità di trasmissione, però in alcuni casi è comunque richiesto un controllo degli errori e del flusso in modo da garantire un certo grado di affidabilità e si utilizza, dunque, anche il protocollo TCP. Per quanto riguarda il livello Application

invece, il motivo principale per cui i protocolli presenti nella pila che corrispondente ai dispositivi IoT sono tutti diversi è perché i protocolli a sinistra non essendo stati pensati per lavorare su dispositivi con risorse limitate protrebbero portare ad un loro utilizzo non ottimale, come verrà spiegato nel primo capitolo.
È importante notare che inizialmente, tra i protocolli a sinistra, HTTP era semplicemente visto come un altro protocollo, ma oggigiorno è il protocollo dominante nel suo livello che viene utilizzato anche per realizzare le funzioni degli altri protocolli come ad esempio il trasferimento di files, la comunicazione tramite posta elettronica, etc. Per quanto riguarda il mondo IoT si prevede che si segua un pattern simile con il protocollo MQTT dato che il suo utilizzo negli ultimi anni sta sempre aumentando.

Questo lavoro di tesi ha come obiettivo realizzare un confronto prestazionale tra i protocolli *HTTP* e *MQTT*, questo confronto è basato sulle prestazioni ottenute dalla loro implemetanzione in un caso tipico in cui è necessario trasmettere informazioni con frequenza molto elevata. In particolare, come caso di studio pratico è stata realizzata un'applicazione che trasmette la posizione GPS di un dispositivo con intervalli di 1-2 secondi e riceve le posizioni di altri dispositivi in modo da rendere possibile la loro visualizzazione su una mappa, e dunque, nell'utilizo prolungato dell'applicazione la trasmissione/ricezione di dati e l'utilizzo di batteria saranno notevoli. Per questo motivo, risulta necessaria l'ottimizzazione dell'applicazione in modo che i dispositivi siano in grado di utilizzarla anche quando le risorse sono limitate.

Nel primo capitolo vengono introdotti 3 protocolli, il protocollo TCP su cui HTTP e MQTT sono basati, il protocollo HTTP presentato come servizio *REST* e i possibili problemi che potrebbe presentare ed infine il protocollo MQTT con un esempio del suo utilizzo e come sono strutturati i messaggi scambiati tra i client MQTT.
Il secondo capitolo presenta il caso di studio, la sua progettazione, le librerie utilizzate e l'implementazione dell'applicazione per dispositivi mobili; si presenta inoltre anche la parte server sia per un client MQTT che per uno HTTP.
Il terzo capitolo introduce le tecnologie utilizzate per la realizzazione del Server sia come Web Server che come Broker MQTT, si presenta inoltre la possibilità di creare un server che goda di portabilità.
Nel quarto capitolo viene fatta un'introduzione su quali saranno i test eseguiti sui dispositivi android, questi test affronterano i due aspetti più importanti da tenere in considerazione: la batteria del dispositivo e i dati consumati.
Nel quinto capitolo si metteno a confronto i risultati ottenuti dai diversi test, mostrando sempre un particolare interesse per la batteria e i dati cosumati.

Il progetto di tesi è stato svolto in collaborazione con l'azienda Magenta srl, che ha fornito il supporto al caso di studio e alle tecnologie utilizzate. Le attività sono state svolte nell'ambito del progetto SY4.0, co-finanziato da Regione Toscana nell'ambito del bando "POR CREO 2014/2020 – Azione 1.1.5 - Progetti strategici di ricerca e sviluppo".

Capitolo 1

Protocolli di comunicazione tra processi

In questo capitolo vengono presentati tre protocolli importanti per il nostro caso di studio, prima di descrivere il loro funzionamento o come questi protocolli vengono utilizzati è utile ricordare che la comunicazione tra processi può avvenire in diversi modi ma seguendo sempre un'implementazione del modello ISO/OSI (Figura 1.1), in particolare, faremmo un accenno sul protocollo TCP che troviamo nel livello trasporto, i protocolli HTTP e MQTT che si trovano invece nel livello applicazione [3].

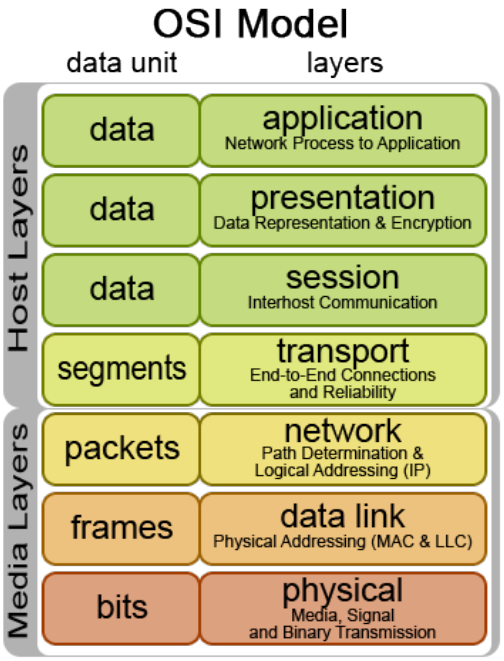


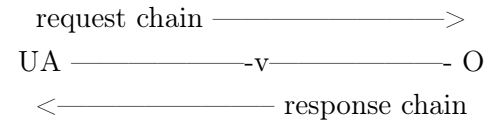
Figura 1.1: Modello ISO/OSI

1.1 Protocollo TCP

TCP è un protocollo di livello trasporto che permette lo scambio di pacchetti tra due host, la caratteristica più importante di questo protocollo è l'affidabilità, ovvero una volta che i due host stabiliscono una connessione tutti i pacchetti arrivano a destinazione, questo però non significa che i pacchetti vengano trasmessi solo una volta, l'affidabilità è possibile grazie al meccanismo di ritrasmissione veloce che si presenta quando un pacchetto viene perso.
Tuttavia, TCP presenta delle possibili limitazioni, che però possono essere risolte in diversi modi, una delle più note si trova proprio nel primo passo che questo protocollo realizza, ovvero, la creazione della connessione.
In particolare, quando un host ha bisogno di stabilire molte connessioni con diversi client è necessario che disponga di una gran capacità di memoria e banda, in diversi casi però questo non è possibile, e diventa quindi una criticità da tenere in considerazione.

1.2 Protocollo HTTP

HTTP è un protocollo di livello applicazione ed è basato sul protocollo TCP, è basato anche sulla modalità di comunicazione *request/response*, un client invia una richiesta al server nella forma di *request method*, URI e versione del protocollo, seguita da un messaggio MIME-like che contiene i modificatori della richiesta, informazioni dei client, e possibilmente un body che contiene dati da trasmettere al server. Il server risponde con uno stato, includendo la versione del protocollo del messaggio e un codice di successo o fallimento, seguito da un messaggio MIME-like contenente informazioni riguardanti il server, meta-informazioni, e possibilmente un body.
Comunemente la comunicazione HTTP è iniziata da un User Agent (UA) e consiste in una richiesta che deve essere applicata a una risorsa nella parte server, nel caso più semplice questo avviene con un'unica connessione v tra UA e Origin Server (O) [4].



Data la sua semplicità nell'utilizzo come *RESTful Service* è diventato uno dei protocolli che ha avuto maggiore diffusione negli ultimi anni.
In un'architettura REST (Representational State Transfer) [5] l'informazione e le funzionalità vengono viste come risorse, l'accesso alle risorse avviene attraverso l'utilizzo delle URIs (Uniform Resource Identifiers).
La limitazione presentata da TCP nel paragrafo precedente viene risolta dallo stile di architettura REST dato che questo vincola a stabilire connessioni *stateless*, ovvero connessioni in cui non si tiene memoria dello stato corrente. In questo caso dunque, una volta che la connessione è stata stabilita e la risorsa richiesta è stata messa a disposizione del client, la connessione viene chiusa in modo da permettere al server di stabilire più connessioni con più client.

1.2.1 Esempio di funzionamento

In questo paragrafo viene mostrato un esempio di utilizzo di HTTP come servizio REST [12].

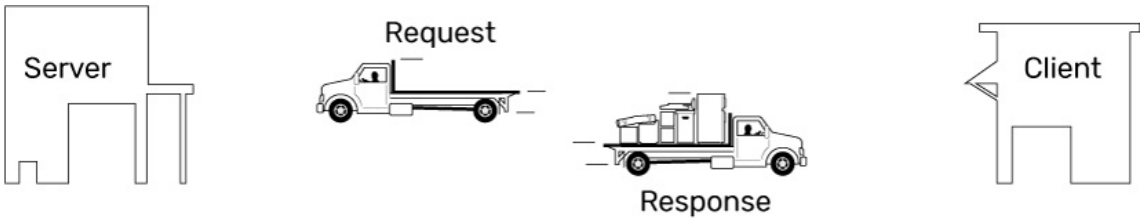


Figura 1.2: Esempio HTTP

Supponiamo di avere due entità, come in Figura 1.2, un client e un server che hanno già stabilito una connessione.
Il client ha bisogno di lavorare con una risorsa e per prenderla manda una *richiesta* (camion vuoto) al server.
Il server dispone di spazio sufficiente per accettare più richieste (camion), e dunque, procede accettando la richiesta del client, dopodiché il server manda una *risposta* (camion con la risorsa richiesta) al client.
Una volta che il client ha ricevuto la risposta la connessione viene chiusa.

1.2.2 Problematiche

Nonostante la semplicità con cui funziona HTTP (come servizio REST) è importante notare che ci sono dei problemi che si potrebbero presentare sia dalla parte server che client:

- il client ogni volta che ha bisogno di comunicare con il server deve stabilire la connessione, e dunque la fase di setup di TCP viene ripetuta più volte. In un caso in cui la frequenza di invio/richieste di dati è alta, la fase di connessione sarà da tenere in considerazione non solo per il tempo di CPU (ovvero batteria utilizzata) ma anche per il consumo di dati che si avrà dato che i messaggi di creazione della connessione TCP verranno ritrasmessi tante volte quanti sono i messaggi HTTP.

- Inoltre, nella situazione in cui un client debba disporre di una risorsa elaborata da altri client è necessario che si effettui il polling periodico al server, ovvero ogni certo intervallo di tempo il client deve mandare una richiesta per verificare se la risorsa è stata modificata, dunque il numero di dati in download aumenterà all'aumentare dei client e sarà sempre da tenere in considerazione.

- Il problema che si potrebbe presentare dalla parte server è che dato l'elevato numero di pacchetti in trasmissione, la probabilità di rischio di congestione aumenta notevolmente all'aumentare del numero di client connessi.

1.3 Protocollo MQTT

Message Queueing Telemetry Transport (MQTT) [6] è un protocollo basato su una connessione TCP, semplice e leggero pensato per lavorare su un'architettura *Publish/Subscribe*, è stato progettato da Andy Stanford-Clark della IBM e Arlen Nipper di Cirrus Link nel 1999 [8] per dispositivi con risorse limitate. I principi con cui è stato progettato sono dunque: minimizzare la banda utilizzata da un dispositivo e minimizzare i requisiti di risorse dei dispositivi, garantendo sempre un certo grado di affidabilità [10].
Col passare del tempo piano piano siamo diventati sempre più connessi tra di noi, questo aumento di connettività non è stato solo tra persone ma anche tra dispositivi, l'Internet of Things è ormai una realtà e le reti di devices che sono ora capaci di comunicare tra di loro vengono usate sempre di più. C'è stata dunque una necessità di avere a disposizione un protocollo che segua il modello *publish/subscribe* in maniera leggera e che permetta di avere uno scambio di messaggi bidirezionale [9].
A differenza del protocollo HTTP, MQTT è basato su un'architettura diversa in cui non si parla di server che accetta le *richieste* dei client e restituisce *risposte*.
Un server MQTT viene comunemente chiamato *broker* [11] e il suo compito è di ricevere tutti i messaggi, filtrarli, determinare quali client sono sottoscritti ad ogni messaggio e inoltrare questi messaggi ai client corrispondenti. Il broker ha anche il compito di mantenere le sessioni dei clienti persistenti, includendo sottoscrizioni e messaggi persi; il broker inoltre, se richiesto, permette anche di avere un sistema di autenticazione.

1.3.1 Esempio di funzionamento

Analogamente al *Paragrafo 1.2.1*, è possibile mostrare il funzionamento della comunicazione attraverso il protocollo MQTT con un esempio [12].

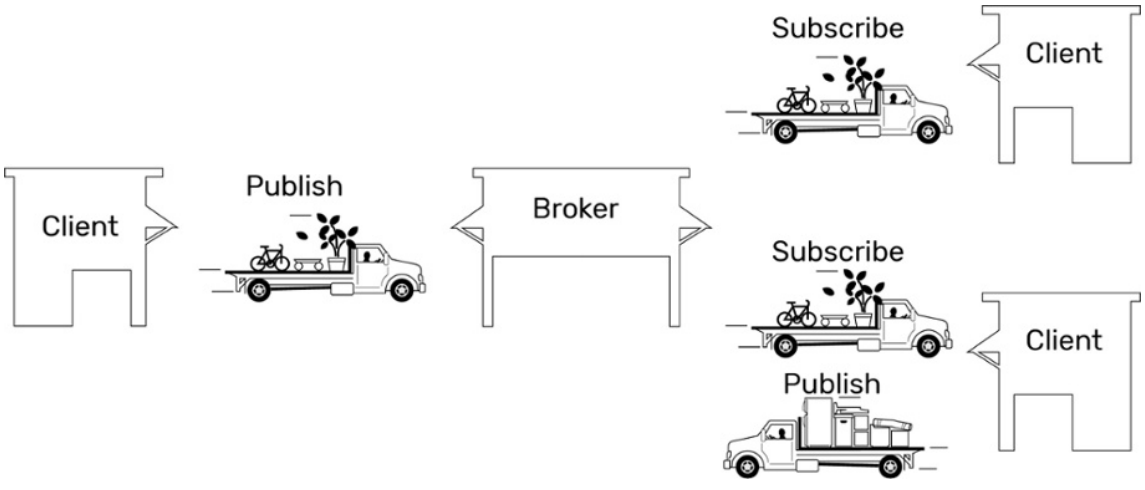


Figura 1.3: Esempio MQTT

Supponiamo di avere 3 client, come in Figura 1.3, che hanno già stabilito una connessione con il broker.
Supponiamo inoltre che il client a sinistra sia un programma che gira su un sensore il cui unico compito è di mandare dati, questo ad esempio, riesce a leggere quali sono gli oggetti che stanno in un giardino di una casa, data l'architettura di publish/subscribe questo dovrà essere sottoscritto ad un topic in cui pubblicherà le letture che effettua. Il topic verrà identificato attraverso la stringa `casa/giardino/`.
I client a destra invece, sono un'applicazione su un dispositivo mobile e un programma su un computer all'interno della casa dotato di una telecamera.
Il client che ha accesso alla telecamera oltre a mostrare i dati che arrivano al topic `casa/giardino/`, pubblicherà gli oggetti che riesce a percepire all'interno della casa, al topic `casa/interno/`.
Il client sul dispositivo mobile invece ha l'unico compito di mostrare gli oggetti della casa a seconda delle necessità dell'utente.
Il compito del server (broker) in questo caso, è semplicemente di mantenere la connessione ed inoltrare i messaggi ai corrispondenti client sottoscritti ai topic dei messaggi, in questo modo per un client non esiste più il vincolo di richiedere una risorsa (o di verificare se la risorsa è disponibile) ma questo verrà notificato attraverso la pubblicazione sul topic a cui il client è sottoscritto.
Ad un certo istante di tempo succede che il sensore a sinistra pubblica sul topic `casa/giardino/` ciò che riesce a percepire, ad esempio una stringa *JSON* in cui descrive gli oggetti, abbiamo poi gli altri due client che sono sottoscritti proprio a questo topic, il broker quindi procede inoltrando il messaggio a questi due, supponiamo anche che il computer con la telecamera pubblica anch'esso ogni tot secondi ciò che riesce a percepire, ma dal momento che nessun client è interessato al topic `casa/interno/` questo messaggio viene semplicemente ignorato.

1.3.2 Struttura dei messaggi

MQTT permette di configurare i messaggi tra i client in molti modi diversi, i messaggi devono essere configurati ad ogni pubblicazione.
Come abbiamo già visto, in un'architettura *publish/subscribe* un client per mandare un messaggio deve assumere il ruolo di *publisher*, per farlo è sufficiente creare un oggetto MqttMessage (come vedremmo nel *Capitolo 2*) e impostare gli attributi nel modo desiderato per poi procedere con l'invio del messaggio attraverso la chiamata al metodo *publish*.

In Figura 1.4 osserviamo la struttura di un messaggio MQTT [13], questo ha diversi attributi che possono essere configurati a seconda delle necessità:

- L'attributo packetId permette di identificare univocamente un messaggio, questo è rilevante solo nel caso in cui l'attributo qos è maggiore di zero perché permetterà di identificare e ritrasmettere i messaggi persi.

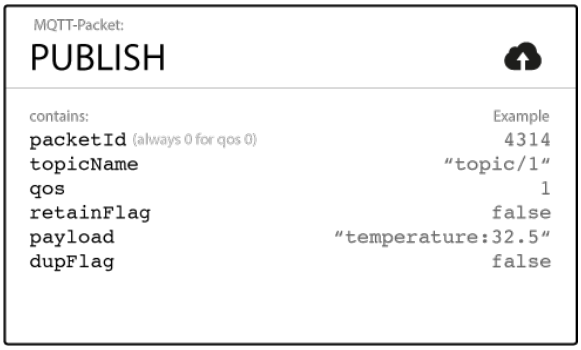


Figura 1.4: Struttura messaggio MQTT

- TopicName invece, è una semplice stringa strutturata gerarchicamente attraverso l'utilizzo degli slash (/) come delimitatori. Ad esempio, "casa/giardino/" e "casa/interno/" indicano due possibili topic che stanno sotto un topic radice "casa/", per ulteriori informazioni sui topic ci si riferisce a [14].

- qos indica la Qualità del Servizio, questo può prendere valori interi da 0 a 2.
  Il livello scelto determina che tipo di garanzia ha un messaggio di raggiungere il destinatario (client o broker), per maggiori informazioni è possibile consultare [15].

- L'attributo retainFlag permette di definire se il messaggio deve essere salvato dal broker come ultimo valore utile per un topic specifico. Se un messaggio contiene questo attributo con valore *true*, un nuovo client che si sottoscriva allo stesso topic riceverà il messaggio anche se il publisher non è più connesso al broker o non invia più messaggi.

- Il campo payLoad è il contenuto utile del messaggio, ovvero la parte che interessa di più ai client che ricevono il messaggio. È possibile inviare immagini, testo in qualsiasi codifica, dati criptati e virtualmente ogni tipo di dato in binario.

- Il flag DUP indica se il messaggio che si sta trasmettendo è un duplicato ed è stato inviato nuovamente perché il destinatario (client o broker) non ha risposto con l'ACK del messaggio. Si può intuire quindi, che questo flag è rilevante solo nel caso di QoS 1 o 2, normalmente questo meccanismo di ritrasmissione viene gestito all'interno della libreria utilizzata [15].

Quando un client invia un messaggio a un broker MQTT, il broker riceve il messaggio, legge i campi (con eccezione del payload), risponde con l'ACK (in base al livello di QoS) e processa il messaggio inoltrandolo ai client sottoscritti al topic.
Dunque, è utile notare che il client che pubblica un messaggio ha la sola responsabilità di confermare il corretto arrivo al broker, il broker invece è responsabile del corretto arrivo ai client sottoscritti al topic del messaggio. Cioè i client che pubblicano non ottengono alcun *feedback* se qualcuno è interessato ai messaggi pubblicati o quanti client hanno ricevuto il messaggio dal broker.

Capitolo 2

Caso di studio

In questo capitolo viene introdotto il caso di studio su cui i seguenti capitoli si basano, verrà presentato il motivo del caso di studio, una soluzione per la sua realizzazione, le librerie utilizzate, l'implementazione del client e una bozza di come dovrebbe essere strutturato il server.

2.1 Introduzione

Com'è già stato anticipato nell'*Introduzione* della tesi, un caso tipico in cui la trasmissione/ricezione di dati avviene con frequenza molto elevata sono le applicazioni che hanno come obiettivo rendere possibile la visualizzazione della posizione di altri dispositivi attraverso una mappa in real-time.
In quesa situazione diventa critico ottimizzare il più possibile l'utilizzo delle risorse in modo da evitare il loro consumo eccessivo (per esempio di banda e batteria). Dunque, l'applicazione deve tenere in considerazione l'utilizzo delle risorse ma effettuare comunque le seguenti operazione:

- Determinare e comunicare al server la posizione attuale del dispositivo con intervalli di 1-2 secondi.
- Ricevere e rendere visibile la posizione degli altri client.

2.2 Progettazione

Come abbiamo appena detto, il client deve essere in grado di inviare la propria posizione, ricevere la posizione degli altri e aggiornare la mappa in modo da renderle visibili. Il client dunque deve interagire con gli altri e per fare questo è stato scelto di utilizzare un approccio Client-Server, per connetterci al server è necessario avere degli attributi che permettano di identificare un host esterno (server), ed è anche necessario che il client abbia un codice univoco che permetta di notificare agli altri la sua posizione.
In particolare, nel caso di un client MQTT è necessario avere un oggetto *mqttAndroidClient* che ci permetta di utilizzare il protocollo e i suoi metodi, ed è dunque necessario anche avere una variabile che contenga il topic in cui si pubblicano i messaggi e il topic a cui ci si

sottoscrive per ricevere la posizione degli altri dispostivi.
Nel caso di un client HTTP invece, la comunicazione con il server risulta più semplice dal momento che si devono eseguire solo due task, uno che richieda ad ogni secondo la posizione degli altri e un altro che mandi la propria posizione ogni 1-2 secondi. È utile notare anche che le chiamate HTTP risulterebbero ancora più semplici con l'utilizzo di una libreria terza che permetta di eseguire richieste HTTP senza la creazione manuale dei due task.
In Figura 2.1 si osserva dunque il diagramma UML che rappresenta come deve essere strutturato il client.

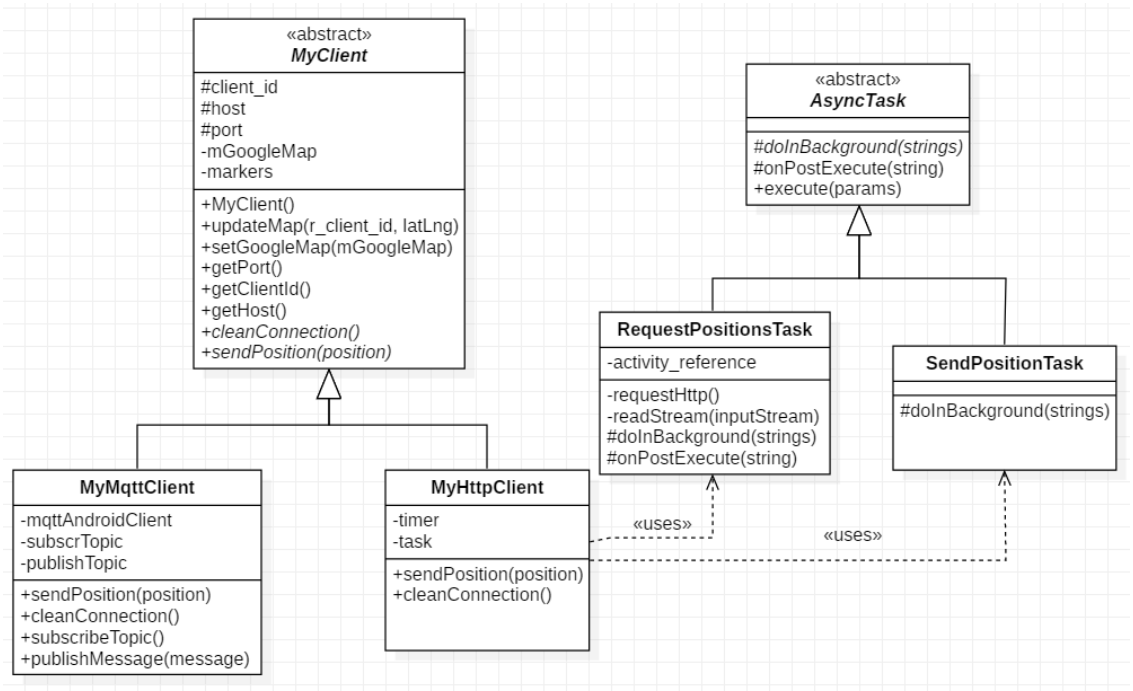


Figura 2.1: Diagramma UML del client MQTT e HTTP

Una volta che abbiamo già stabilito la struttura del client è possibile pensare a come dovrebbe essere strutturata l'applicazione.
Innanzitutto è necessario un Activity[1] che permetta di leggere l'input (il protocollo desiderato, il client_id, l'host server e la porta su cui fare le richieste); dopo aver letto l'input, dobbiamo rendere visibile la mappa con un altro Activity che avrà anche il compito di creare il client concreto che riceverà la posizione degli altri e manderà la propria posizione attraverso la chiamata al metodo *sendPosition(position)*.
In Figura 2.2 osserviamo il Diagramma UML dell'applicazione complessiva.

2.3 Librerie

Per la realizzazione dell'applicazione sono state utilizzate diverse librerie, in alcuni casi è stato possibile scegliere tra più opzioni e in altri casi la scelta della libreria da utilizzare è

[1]Un Activity in android java è una classe che permette di interagire con l'utente attraverso degli oggetti di interfaccia grafica, dunque, un Object Activity contiene UI Objects che permettono di leggere l'input dell'utente e di mostrare l'output all'utente.

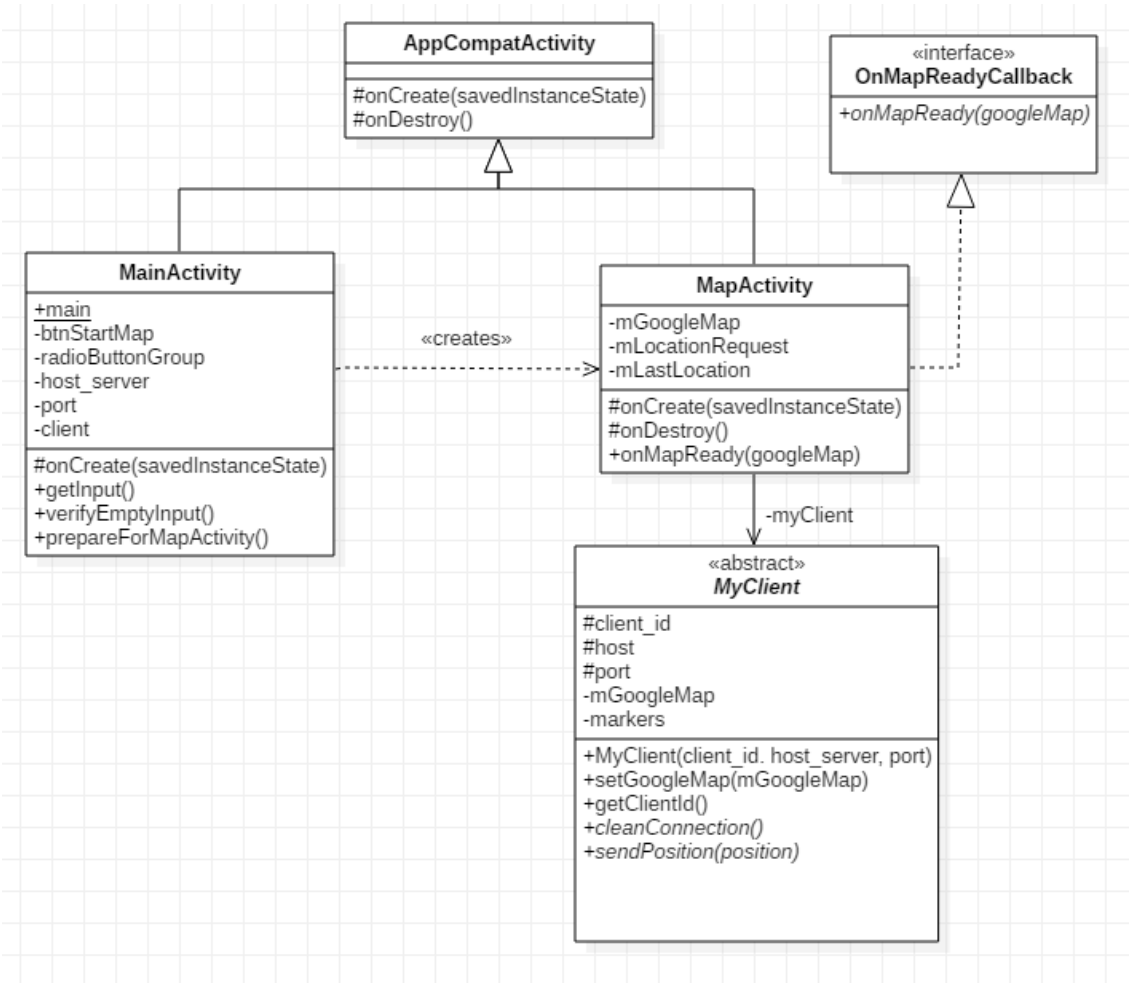


Figura 2.2: Diagramma UML dell'applicazione

stata fatta in base ai suggerimenti di *Android Developer* [17]; in questa sezione verranno mostrate le scelte fatte.

Per la realizzazione di una Mappa su cui mostrare e richiedere la propria posizione è stata presa in considerazione la guida [18] basata sulle informazioni trovate nella pagina di Android Developer [17].

Per il client che invierà la propria posizione abbiamo diverse scelte, ad esempio per quanto riguarda il client HTTP è stato possibile utilizzare una delle libreria già esistenti che semplificano a lungo termine la realizzazione di richieste HTTP, tra le più note abbiamo: OkHttp, Retrofit, Picasso, Volley, etc.

Nonostante le risposte date in *Best library to make HTTP calls*[2] e i risultati visti in *AsyncTask vs Volley vs Retrofit*[3], è stata scelta la libreria HttpUrlConnection, dato che rappresenta l'utilizzo più semplice possibile senza librerie esterne, le richieste concorrenti HTTP da realizzare sono limitate a 2 nel caso peggiore e dato che la semplicità all'effettuare queste

[2]Migliori libreria per effettuare chiamate HTTP con Java Android `https://www.quora.com/What-is-the-best-library-to-make-HTTP-calls-from-Java-Android`.

[3] Risultati che mostrano la performance tra AsyncTask, Volley e Retrofit `http://instructure.github.io/blog/2013/12/09/volley-vs-retrofit/`.

richieste è uno dei punti di forza di HTTP (come vedremmo nel paragrafo dell'implementazione).
Per quanto riguarda il caso del client MQTT, in generale è possibile scegliere tra più opzioni ma dal momento che l'obiettivo del nostro caso di studio è la realizzazione di un'applicazione Java Android, la scelta di una libreria MQTT è limitata ad una sola (senza perdere la caratteristica Open Source[4]), Eclipse Paho[5].

2.4 Implementazione del caso di studio

Per l'implementazione dell'applicazione sono state prese in considerazioni diverse guide, nelle seguenti sezioni verranno mostrati frammenti di codice con i rispettivi riferimenti.

2.4.1 Client

Com'è già stato leggermente anticipato nel *Paragrafo 2.3*, per la creazione dell'applicazione Android che farà da client sono stati presi in considerazione tre importanti riferimenti:

- Una guida per capire come utilizzare GoogleMaps in un'applicazione [18].
- Una guida in cui si spiega come utilizzare la libreria Eclipse Paho [19] e il progetto open source (in cui si utilizza la libreria) preso da `https://github.com/bytehala/android-mqtt-quickstart`.
- Una guida su come realizzare chiamate HTTP [20] e gli esempi presi da `https://stackoverflow.com/questions/8654876/`.

Ma prima di iniziare a lavorare su questi 3 punti è stato scelto di costruire il client astratto come mostrato in Figura 2.1 e il MainActivity come mostrato in Figura 2.2.
Il codice del client astratto è disponibile nell'Appendice A, Sezione *Codice in Java per la realizzazione dell'applicazione.*

2.4.1.1 GoogleLocation API

Per quanto riguarda la visualizzazione della mappa, seguendo lo schema in Figura 2.2 oltre agli attributi presenti nella nostra guida [18] abbiamo bisogno semplicemente di un riferimento ad un MyClient object, e attraverso il metodo *sendPosition(position)* manderemo la propria posizione al server, per quanto riguarda la richiesta della posizione degli altri client invece, questo verrà fatto dal client concreto (MQTT o HTTP). Il codice è disponibile nell'Appendice A, Sezione *Codice per l'implementazione di GoogleLocation API*

[4]Confronto tra le diverse librerie MQTT come client disponibili a: `https://en.wikipedia.org/wiki/Comparison_of_MQTT_Implementations`.
[5] Maggiori informazioni su Eclipse Paho sono disponibili al seguente link `https://www.eclipse.org/paho/`.

2.4.1.2 Client HTTP

Per quanto riguarda la creazione del client HTTP, è necessario creare una classe che estenda la classe astratta *MyClient.java*, definita all'inizio del *Paragrafo 2.4.1*, in questo modo si rende possibile la istanziazione di questo client nel MapActivity.
Come abbiamo già anticipato, per comunicare con il server attraverso il protocollo HTTP, è necessario creare due task in modo che eseguano le richieste in background, senza bloccare l'interfaccia grafica. Per questo motivo, seguendo il diagramma UML definito nel *Paragrafo 2.2* oltre alla classe *MyHttpClient.java* verranno mostrate due classi: *RequestPositionsTask.java* e *SendPositionTask*. Il codice è dispnoibile nell'Appendice A, Sezione *Codice per la realizzazione del client HTTP*.

2.4.1.3 Client MQTT

Così come nel caso del client HTTP, anche qui è necessario creare una classe che estenda *MyClient.java* in modo da rendere possibile il suo utilizzo nel *MapActivity.java*. Il codice è disponibile nell'Appendice A, Sezione *Codice per la realizzazione del client MQTT*. Nelle immagini seguenti si mostra l'aspetto dell'applicazione realizzata, si osserva dunque che l'interfaccia grafica della mappa con la posizione degli altri client è la stessa sia per il caso MQTT che HTTP:
In Figura 2.3 osserviamo che per utilizzare l'applicazione con il protocollo HTTP dobbiamo inviare/richiedere i messaggi sulla porta 8080, per il caso MQTT invece la porta scelta è 1884.

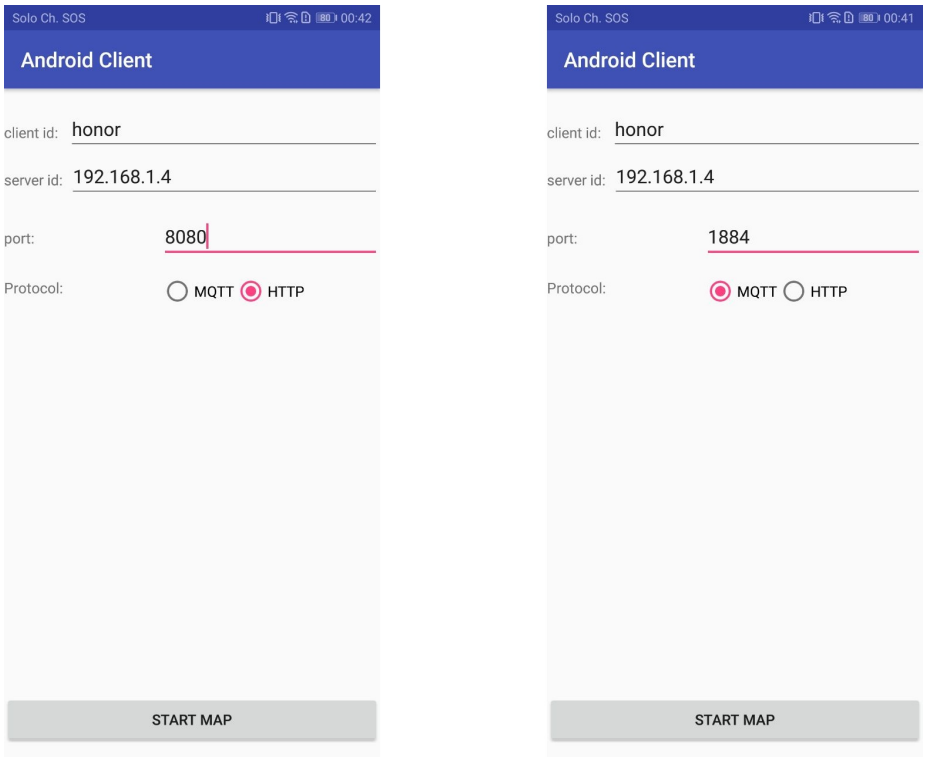


Figura 2.3: Activity principale, a sinistra un esempio di connessione con un server HTTP e a destra con uno MQTT.

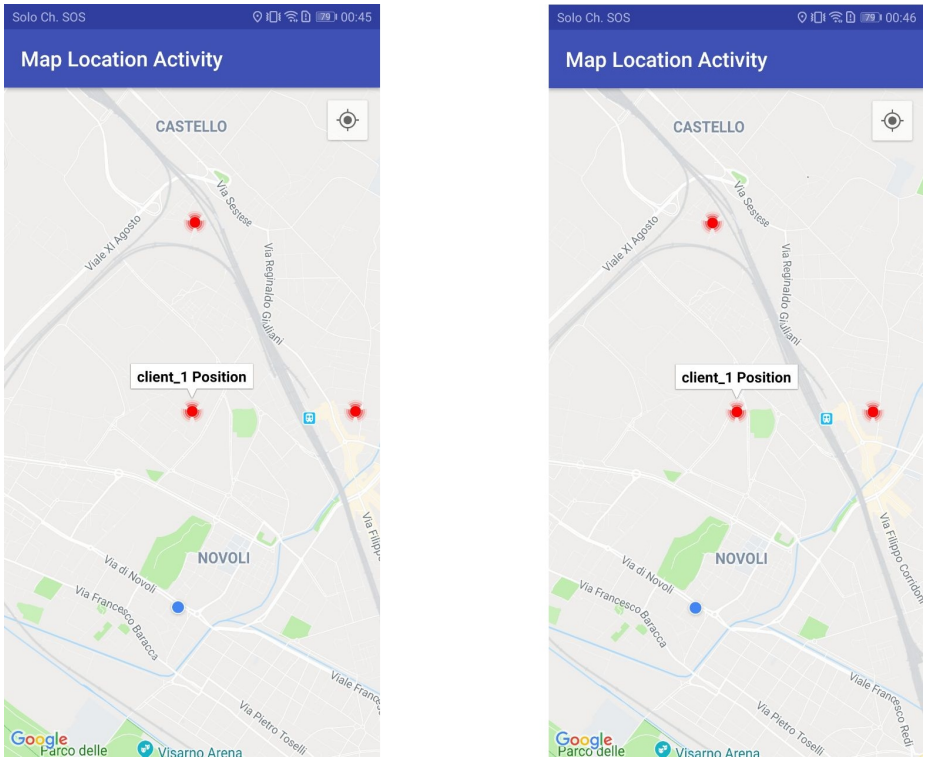


Figura 2.4: Map activity in due istanti di tempo diversi.

La Figura 2.4 mostra invece lo stato della mappa in due istanti di tempo diversi, a sinistra lo stato corrispondente al tempo t e a destra al tempo $t + \delta$, nell'istante di tempo t si hanno tre client che trasmettono la propria posizione al server, nell'istante di tempo $t + \delta$ due dei tre client sono rimasti nella stessa posizione, si osserva dunque che il *client 1* ha cambiato leggeremente la sua posizione e l'ha mandata al server, nel caso HTTP questa nuova posizione è stata semplicemente aggiunta al database e viene dunque scaricata dopo la richiesta del nostro *client honor*, nel caso MQTT invece il *client honor* viene notificato di questa variazione grazie al fatto che si è sottoscritto al topic `test_posizione/#`.

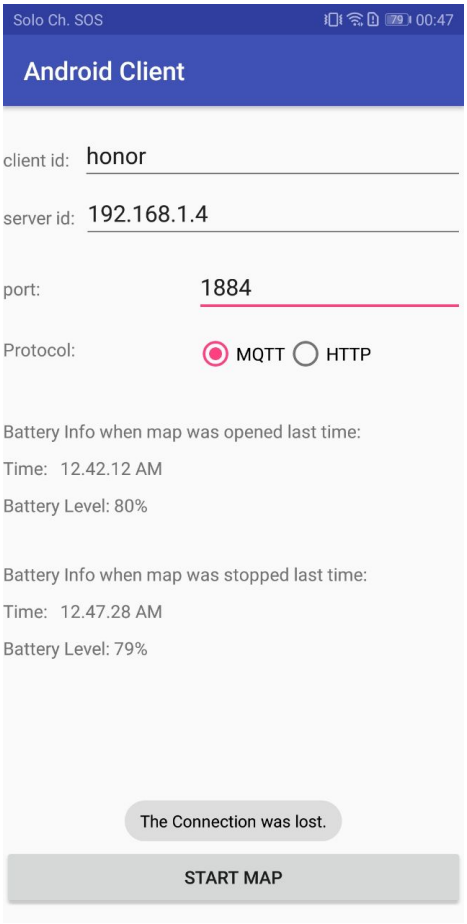


Figura 2.5: Activity principale dopo la chiusura della mappa.

In Figura 2.5 si mostra l'activity principale dopo la chiusura della mappa, nel caso MQTT viene mostrato un *toast message* che ci avvisa che la connessione è stata chiusa/persa, questo oltre ad offrire i campi per stabilire una nuova connessione mostra anche informazioni riguardanti il consumo della batteria durante l'utilizzo della mappa e uno dei due protocolli per inviare la propria posizione e ricevere la posizione degli altri.

2.4.2 Server HTTP

Seguendo lo schema in Figura 2.1 e l'implementazione nel *Paragrafo 2.4.1.2* ci si accorge che il server HTTP deve mettere a disposizione due operazioni tramite delle URL:

- Un URL `http://server_ip:port/get-all-position` che restituisce la posizione di tutti i client attraverso una string JSON.
- Un URL `http://server_ip:port/clients/positions/` su cui si invia la propria posizione come una string JSON.

Dal momento che la realizzazione manuale di un server HTTP non è così semplice e che esistono diversi tool che permettono di semplificare i passi, è stato scelto di utilizzare uno di questi come verrà mostrato nella *Sezione 3.2*. Tuttavia, in questa sezione verrà mostrato

ciò di cui il server deve disporre e come deve rispondere.

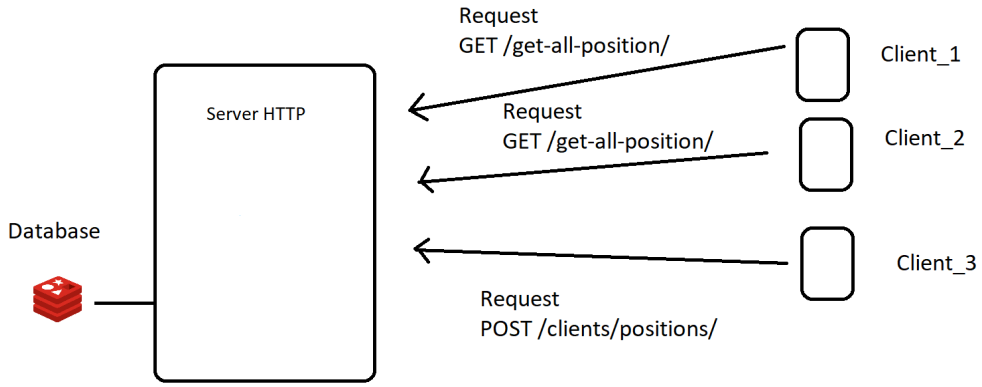


Figura 2.6: Server HTTP

In Figura 2.6 si osserva una possibile situazione in cui potrebbe trovarsi il server in un determinato istante di tempo: a destra abbiamo dei client che richiedono la posizione degli altri e a questi viene restituita una stringa JSON della forma seguente:

```
[
    {
        "id": "client_1",
        "name": "{
                \"client_id\": \"client_1\",
                \"latitude\": 43.79,
                \"longitude\": 11.226
                }"  },
    {
        "id": "browser",
        "name": "{
                \"client_id\": \"client_2\",
                \"latitude\": 43.78,
                \"longitude\": 11.226
                }"  },
    {
        "id": "browser",
        "name": "{
                \"client_id\": \"client_3\",
                \"latitude\": 43.81,
                \"longitude\": 11.242
                }"  }
]
```

Dall'altra parte invece abbiamo dei client che inviano la propria posizione attraverso una *POST request* verso l'url `http:server_ip:port/clients/positions` con attributo:

```
json =  {
        "client_id": "client_3",
        "latitude": 43.80,
        "longitude": 11.24
        }
```

2.4.3 Broker MQTT

Anche nel caso del broker MQTT abbiamo bisogno di due operazioni che il broker deve mettere a disposizione, il broker deve:

- Permettere ai client di sottoscriversi ad un determinato topic.
- Permettere ai client di pubblicare messaggi su un determinato topic.

A questo punto però, ci si accorge che queste due operazioni vengono ripetute in qualsiasi altro caso di studio e dunque torna comodo utilizzare un qualsiasi broker MQTT già esistente.

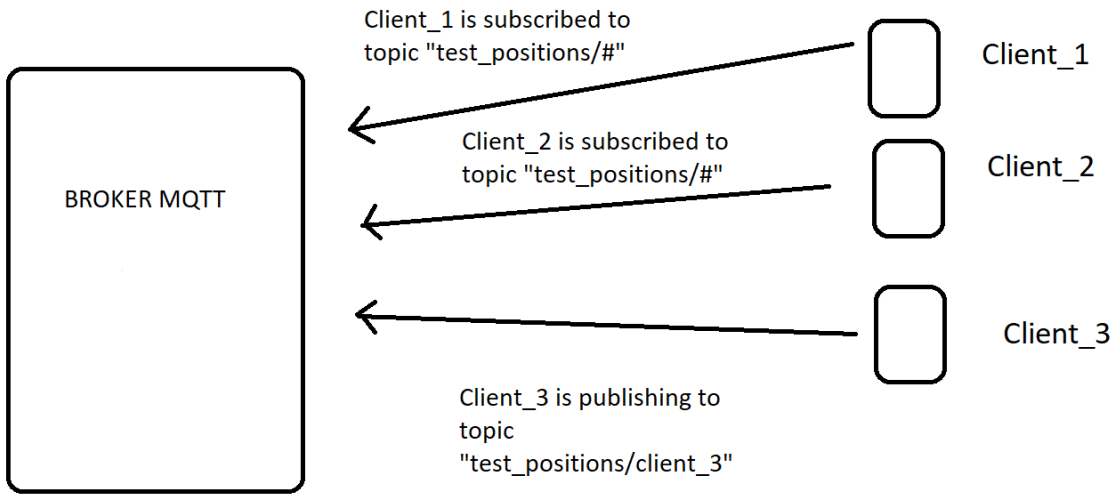


Figura 2.7: Broker MQTT

In Figura 2.7 si osserva un possibile funzionamento del nostro broker MQTT, abbiamo dunque un client con id *client_3* che pubblica la seguente stringa JSON sul topic `test_positions/client_3`:

```
json =  {
        "client_id": "client_3",
        "latitude": 43.80,
        "longitude": 11.24
        }
```

Questa stringa verrà inoltrata dal broker verso i client_1 e client_2 dato che si sono sottoscritti a ricevere la posizione di tutti.
Tra i diversi broker per il nostro caso di studio è stato scelto di utilizzare *Mosquitto* come broker MQTT, data la sua caratteristica *open source*.

Capitolo 3

Tecnologie

In questo capitolo verranno presentate le diverse tecnologie con cui è stato realizzata la parte server del caso di studio. Come abbiamo già anticipato nel *Capitolo 2* costruire un server HTTP non è così semplice come realizzare la parte client HTTP, in particolare per il nostro server HTTP è stato scelto di utilizzare uno dei diversi framework che semplificano la costruzione di questo.
Per il server MQTT invece è stato pensato di utilizzare uno già esistente, tra i più noti e che mantengono la caratteristica *open source* è stato scelto *Mosquitto*.
Nella sezione finale di questo capitolo viene prestata la piattaforma Docker, che permette di rendere possibile la portabilità del nostro server (sia HTTP che MQTT).

3.1 Web Service con Spring Boot

Dal momento che la creazione di un Web Service richiede l'implementazione manuale di un Web server come Tomcat (o altri), è stato scelto di utilizzare uno dei framework già esistenti che permette di semplificare i passi da realizzare.
Spring è il framework più famoso quando si parla di applicazione Java Enterprise Edition (EE), la sua caratteristica principale è la *dependency injection* che risulta in una *inversione delle responsabilità* [21].
Tra i diversi vantaggi che Spring offre, il più rilevante è il fatto di poter mettere insieme diversi tools e API nel progetto java mantenendo tutto collegato attraverso Spring.
Per capire il funzionamento della sua caratteristica principale proviamo a considerare un esempio:
Supponiamo di creare un'interfaccia Vehicle, e diverse classi che la implementano Car, Bus e Bike.
Per utilizzare le funzionalità di un Car abbiamo bisogno di dichiarare vehicle come un Car

```
Vehicle vehicle = new Car();
```

Il problema si presenta quando vogliamo utilizzare le funzionalità di un Bike, in questo caso risulterebbe necessario modificare vehicle come:
`Vehicle vehicle = new Bike();`
Cosa che potrebbe peggiorare quando l'applicazione cresce e l'utilizzo di vehicle aumenta.
Grazie a Spring è sufficiente dichiarare vehicle come: `Vehicle vehicle;`
In questo caso, Spring fornisce un grado di flessibilità nell'istanziare vehicle come Car, Bus o Bike con una semplice modifica in un file di configurazione senza toccare il codice in cui si utilizzano le funzionalità.
Tornando al nostro caso di studio, Spring mette a disposizione un modulo per lo sviluppo web.
Un possibile problema di Spring è che non è pensato per una persona inesperta, ma per uno che ha già realizzato altre applicazioni attraverso l'uso delle dependency e sa come configurare l'applicazione.
Questo possibile problema è stato risolto grazie a **Spring Boot**, attraverso una pre-configurazione, che semplifica la creazione di applicazioni Spring.

3.1.1 Implementazione

Seguendo la guida su come creare un servizio Restful con Spring Boot [22], come primo passo, dobbiamo creare un bozza attraverso lo inizializzatore di Spring Boot[1] Una volta creata l'applicazione Spring dobbiamo creare una representation class per il nostro caso, cioè un client che dispone di un id e contiene la sua posizione, per semplicità è stato scelto di inserire la posizione del client come una stringa JSON che contiene id, latitudine e longitudine (cioè si prende come String name ciò che ci viene passato quando un client effettua la richiesta POST vista nel capitolo precedente).
Su Spring, gli endpoints REST sono semplicemente dei MVC controllers, cioè per gestire ciò che succede quando un client effettua una richiesta su una url (ad esempio `http://server_ip:port/get-all-positions/`) si deve creare un controller che restituisce una risorsa.
Osserviamo nel Controller che abbiamo bisogno di un referimento ad un oggetto che permette di eseguire delle operazione sui dati salvati, e ovviamente per avere a disposizione questi dati salvati bisogna averli messi in una struttura dati o meglio ancora in un Database.
Dal momento che ci interessa semplicemente di salvare una stringa (valore) che corrisponde ad un client_id (chiave) possiamo utilizzare un database non relazionale basato sul modello di chiave-valore.

[1] Spring Boot mette a disposizione di tutti uno inizializzatore che crea un'applicazione pre configurata in modo da avere un web service pronto per l'utilizzo con un semplice click, `https://start.spring.io/`.

ClientsController.java

```
@Controller
public class ClientsController {

    // riferimento ad un oggetto ReactiveRedisOperations che ci permette di
    ↪  inserire elementi nel database Redis o di leggerli
    private final ReactiveRedisOperations<String, Client> clientOps;

    ClientsController(ReactiveRedisOperations<String, Client> coffeeOps) {
        this.clientOps = coffeeOps;
    }

    @GetMapping("/get-all-position")
    @ResponseBody
    public Flux<Client> all() {
      System.out.println("sending all positions");
        return (clientOps.keys("*")
                .flatMap(clientOps.opsForValue()::get));
    }

    public void addClientPosition(String id, double lat, double lon) {
    Flux.just("{ \"client_id\": \""+id+"\", \"latitude\": "+lat+",
    ↪  \"longitude\": "+ lon+" }")
        .map(name -> new Client(id, name))
        .flatMap(coffee -> clientOps.opsForValue().set(coffee.getId(),
        ↪  coffee)).subscribe();
  }

  @PostMapping("/clients/positions")
  @ResponseBody
  public String registerLocationForClient(@RequestParam(name="json",
  ↪  required=true) String json) {

    JSONObject jsonText;
    try {
      jsonText = new JSONObject(json);
        System.out.println("recived from ClientID:
        ↪  "+jsonText.getString("client_id")+ "\n"+jsonText);
        addClientPosition(jsonText.getString("client_id"),
        ↪  jsonText.getDouble("latitude"),
        ↪  jsonText.getDouble("longitude"));
    } catch (JSONException e1) {
      e1.printStackTrace();
    }
    return "Success";
  }
}
```

Client.java

```java
@RedisHash("Client")
public class Client implements Serializable {

    private String id;
    // Stringa json che contiene la posizione del client
    private String name;

    public Client() {
    }
    public Client(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Client{" + "id='" + id + '\'' + ", name='" + name + '\'' +
        ↪  '}';
    }
}
```

3.2 DataBase Redis

Come abbiamo detto nel *Paragrafo 3.1*, Spring fornisce supporto su diversi tools e API grazie alla sua caratteristica di supporto alla dependency injection. Redis è un database NoSQL [23] supportato da Spring; Redis offre la possibilità l'utilizzo di strutture dati come strings, hashes, lists, sets, etc.
Redis permette anche di salvare i dati in maniera persistente attraverso un flag di configurazione.
Seguendo la guida che Spring mette a disposizione [24], per utilizzare Redis nel nostro Web Service è stato necessario aggiungere le dependencies:

pom.xml

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

E aggiungere un file di configurazione:

ClientConfiguration.java

```
@Configuration
public class ClientConfiguration {
    @Bean
    ReactiveRedisOperations<String, Client>
     ↪  redisOperations(ReactiveRedisConnectionFactory factory) {

        Jackson2JsonRedisSerializer<Client> serializer = new
         ↪  Jackson2JsonRedisSerializer<>(Client.class);

        RedisSerializationContext.RedisSerializationContextBuilder<String,
         ↪  Client> builder =
                RedisSerializationContext.newSerializationContext(new
                 ↪  StringRedisSerializer());

        RedisSerializationContext<String, Client> context =
         ↪  builder.value(serializer).build();

        return new ReactiveRedisTemplate<>(factory, context);
    }
}
```

3.3 Mosquitto MQTT Broker

Riprendendo l'argomento della *Sezione 2.4.3*, dal momento che le operazioni di pubblicazione sui topic e sottoscrizione ai topic sono due operazioni che generalmente vengono ripetute tutte le volte, torna utile utilizzare uno dei broker MQTT già esistenti.
Tra i più famosi è stato scelto Mosquitto [25].
Eclipse Mosquitto è un broker open source che implementa MQTT nelle versioni 3.1 e 3.1.1, Mosquitto è leggero e adatto per utilizzo su tutti i dispositivi. Mosquitto è disponibile su diversi sistemi operativi, nonostante il procedimento di installazione dipenda dall'ambiente di lavoro, nella *Sezione 3.4* verrà mostrato come superare questa limitazione.

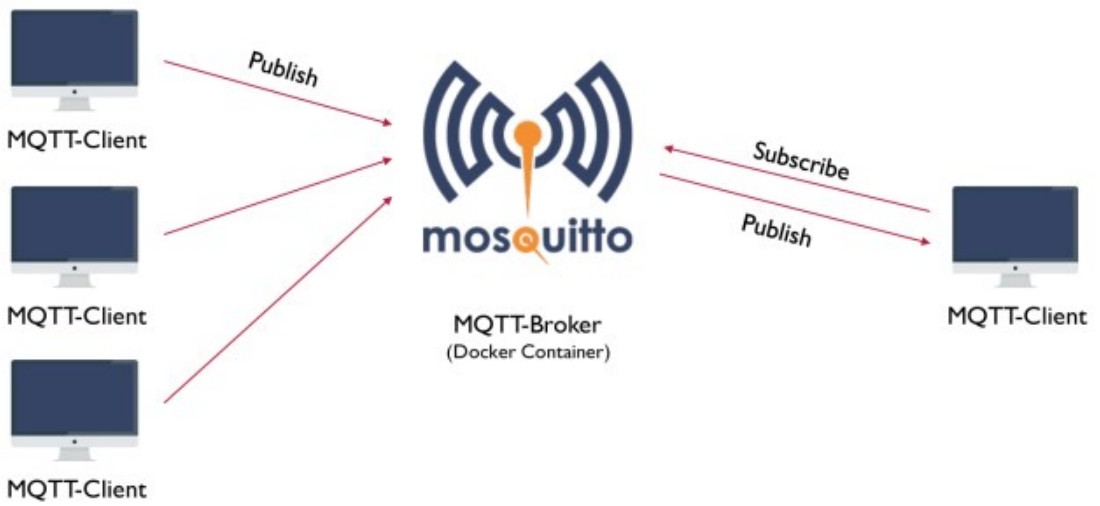


Figura 3.1: Mosquitto come broker MQTT.

Nella Figura 3.1 osserviamo come il funzionamento di Mosquitto in un altro caso di studio (come ad esempio quello mostrato nel riferimento [26]) sia uguale al broker mostrato in Figura 2.7.

3.4 Docker

Docker è un programma che permette di realizzare una virtualizzazione a livello di sistema operativo, comunemente conosciuto come *Docker containerization* [27].
Docker viene maggiormente usato per eseguire package chiamati *containers*. I containers sono isolati tra di loro, e gestiscono i loro propri tools, librerie e file di configurazione. Però possono comunicare tra di loro attraverso canali ben definiti. I container sono molto più leggeri delle macchine virtuali e vengono creati da delle *immagini* che specificano il contenuto preciso che questi possiedono e la loro configurazione. Un'immagine è come una cattura di una configurazione ben precisa di un programma in cui si aggiungono anche file esterni di cui questo programma ha bisogno, ma normalmente le immagini vengono combinate con altre immagini standard presenti in repositories pubblici.

3.4.1 Containerizzazione

Per permettere che il nostro Web Service possa esssere eseguito su qualsiasi sistema operativo dobbiamo realizzare la sua containerizzazione; dal momento che il Web Service utilizza Redis come database servirebbe anche la containerizzazione di quest'ultimo, ma tenendo in considerazione che molte immagini di tecnologie vengono messe in repository pubblici e che attraverso le porte di un server è possibile comunicare con altri container, dobbiamo semplicemente creare l'immagine dell'applicazione web in java enterprise.

Seguendo la guida di Spring Boot su come creare una docker image [28] dobbiamo:

- Generare il file eseguibile (jar) della nostra applicazione java enterprise.
- Creare un file DockerFile che definisce ciò che sarà all'interno del container, come l'accesso risorse (si può dichiarare quale porta del server collegare al container, l'accesso al disco, l'utilizzo di file esterni, etc.) Dopo aver definito tutto ciò che ci serve in questo file possiamo aspettarci che il comportamento del container e la nostra app sia lo stesso[2].

Una volta che abbiamo questi due file la guida di Spring Boot ci suggerisce di utilizzare Maven o Gradle per costruire l'immagine, tuttavia è possibile farlo senza ulteriori installazioni eseguendo il seguente comando nel command-line di Docker[3]:

```
docker build -t mywebservice --build-arg jar_file=spring_boot.jar .
```

DockerFile per la costruzione dell'immagine

DockerFile

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG jar_file
COPY ${jar_file} app.jar
ENTRYPOINT
 ↪  ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Una volta creata l'immagine del nostro web service per creare i container, ovvero per avere un server che funzioni come web server e come broker, è sufficiente eseguire i seguenti comandi sull'interfaccia di Docker:

```
docker run -d -ti -p 1883:1883 -p 9001:9001 toke/mosquitto

docker run -d -p 6379:6379 redis

docker run -d -p 8080:8080 mywebservice
```

[2]Dettagli sui DockerFile possono essere trovati su `https://docs.docker.com/get-started/part2/`.

[3]Docker mette a disposizione dell'utente una CLI, nella Sezione *Set build-time variables (–build-arg)* in `https://docs.docker.com/engine/reference/commandline/build/` troviamo dettagli sul comando per costruire un'immagine.

Dove il primo comando serve per scaricare ed istanziare il broker MQTT, questa immagine viene scaricata dal repository pubblico `https://hub.docker.com/r/toke/mosquitto/`, non è stata presa in considerazione l'immagine ufficiale di Eclipse-Mosquitto, dato che una volta creato il container questo non accettava le connessioni e diversi post suggerivano di utilizzare *toke/mosquitto*.
Il secondo comando scarica l'immagine di Redis dal suo repository pubblico:
`https://hub.docker.com/_/redis/` e istanzia il container.
Il terzo invece crea il container della nostra immagine costruita precedentemente esponendo il servizio sulla porta 8080.

Capitolo 4

Test su un dispositivo mobile

Una volta conclusa la fase di implementazione del client e il server per il nostro caso di studio, dobbiamo confrontare le performance ottenute con i due protocolli.
Per realizzare un confronto delle prestazioni ottenute abbiamo bisogno di stabilire i modi con cui misurare le caratteristiche di nostro interesse, dal momento che la differenza sostanziale dell'utilizzo dei due protocolli sta proprio nell'effettuare il polling periodico al server, dunque nella quantità di istruzioni eseguite e dati scambiati, i due fattori da tenere in considerazione sono il consumo della batteria e la quantità di dati scambiati.
In questo capitolo verranno introdotte le modalità che sono state utilizzate per misurare il consumo della batteria, i dati inviati dal dispositivo (upload), i dati scaricati dal dispositivo (download) e il consumo di banda nella rete.
Inoltre, si mostrerà anche quali sono state le modalità dei test effettuati e i risultati ottenuti.

4.1 Test sul consumo di batteria

Per effettuare la misurazione della batteria utilizzata dall'applicazione sono stati presi in considerazione i seguenti post:

- *How much battery does my app android consumes on user devices?*
  `https://stackoverflow.com/questions/15900094`

- *How can we calculate how much battery life my user defined application is utilising*
  `https://stackoverflow.com/questions/22343774`

In questi posts viene spiegato che al momento non è possibile sapere esattamente quanta batteria sta utilizzando un'applicazione dato che l'utilizzo delle risorse normalmente viene considerato come una cosa esterna all'applicazione (altre applicazioni potrebbero utilizzare le stesse risorse e dunque una misurazione che considera la somma della quantità di batteria utilizzata dall'applicazione e dalle risorse potrebbe essere non precisa) e che dunque il miglior modo per farlo è utilizzare dei tool esterni o le statistiche che Android stesso mette a disposizione nelle impostazioni dei dispositivi.

4.1.1 Modalità dei test

I test sono stati effettuati su un dispositivo *Honor 9 lite* con[1]:

- Sistema operativo: Android 8.0.0 (Oreo).
- Chipset: HiSilicon Kirin 659.
- CPU: Octa-core (4x2.36 GHz Cortex-A53 & 4x1.7 GHz Cortex-A53).
- Battery: 3000 mAh battery.

Si sono considerati 16 test per ogni protocollo con un utilizzo dell'applicazione di 5 ore sul telefono in uno stato in cui le altre applicazioni erano state chiuse, è importante notare che il consumo di batteria da parte dei servizi delle applicazioni come Google Play Services, Messenger, IU Sistema, Tastiera, Gmail, etc. sono stati considerati come consumo aggiuntivo alla nostra applicazione dato che non è stato possibile chiuderli dal momento che (ad esempio) la mappa utilizzava Google Play Services, il sistema operativo manteneva attivo il Servizio Huawei Home, etc. Tuttavia, il consumo di batteria di queste applicazioni non ha interferito nella nostra misurazione dato che i servizi erano attivi sia durante l'utilizzo del protocollo MQTT e che HTTP, inoltre, come si osserva in Figura 4.1, il consumo della batteria di queste applicazioni (ad eccezione di Play Services per la visualizzazione della mappa) è notevolmente inferiore rispetto a quello dell'applicazione realizzata.
È importante notare che Android nella versione 8.0.0 oltre ad offrire la possibilità di vedere la percentuale della batteria utilizzata permette di visualizzare un'analisi del consumo delle risorse da parte di un'applicazione. In Figura 4.2 osserviamo una valutazione di quanto ha consumato l'applicazione [2] durante il suo uso in due test, a sinistra il caso HTTP e a destra MQTT.

4.1.2 Risultati MQTT

Nelle Tabelle 4.1 e 4.2 osserviamo i risultati ottenuti in 16 test, questi risultati sono stati presi manualmente con gli intervalli mostrati nella prima colonna. Osserviamo dunque che il primo test ha dato un risultato leggermente più elevato rispetto agli altri, tuttavia dal momento che negli altri casi la differenza tra i risultati è praticamente nulla possiamo comunque valutare il consumo della batteria come la media aritmetica ottenuta dai test.

4.1.3 Risultati HTTP

Nelle Tabelle 4.3 e 4.4 invece, osserviamo i risultati nello stesso numero di test ma con il protocollo HTTP, così come nel caso MQTT questi risultati sono stati presi manualmente con intervalli mostrati nella prima colonna. Così come nel caso MQTT, anche in questi test il primo ha dato un risultato leggermente maggiore rispetto agli altri ma questo è comunque accettabile dato che negli altri test la variazione è minima.

[1] Specifiche prese da `https://www.gsmarena.com/huawei_honor_9_lite-8962.php`

[2] Per accedere a queste informazioni è sufficiente entrare nelle impostazioni, cercare le applicazioni installate e selezionare l'applicazione desiderata.

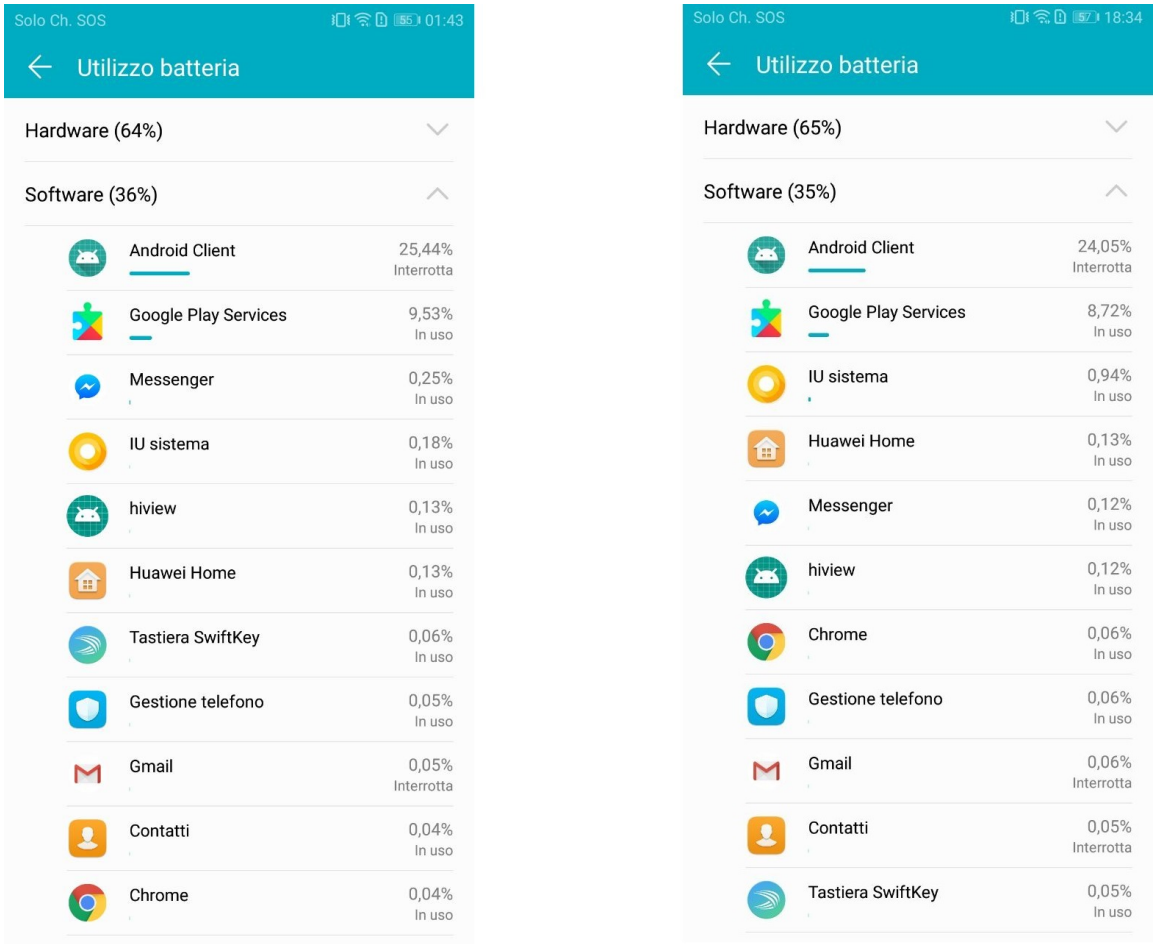


Figura 4.1: Batteria consumata in due test, a sinistra il test con HTTP e a destra il test con MQTT

4.1.4 Wireshark

Wireshark è un programma *open source* che permette di analizzare i pacchetti nella rete. Per la nostra analisi prestazionale verrà utilizzato per misurare quanti byte si inviano al server, e come si utilizza la banda della rete con entrambi i protocolli.
Tra le diverse opzioni che Wireshark offre tramite la sua interfaccia grafica ne sono state prese in considerazione due, così come si evidenzia in Figura 4.3.
L'opzione nel menu *statistiche/conversazioni* permette di visualizzare quanti pacchetti e byte si trasmettono in una conversazione sia a livello IP che a livello TCP.
Mentre l'altra opzione permette invece di visualizzare graficamente il rate di trasmissione, ovvero quanti byte ad ogni secondo si trasmettono. Questa opzione offre inoltre di aggiungere un filtro di visualizzazione sul grafico I/O: per il nostro caso di studio siamo interessati ai filtri sulla porta 1884[3] e sulla 8080.

[3]Dal momento che Mosquitto è disponibile anche su Windows, è stato scelto di utilizzare la porta 1884 per comunicare con Mosquitto sulla macchina virtuale creata da Docker.

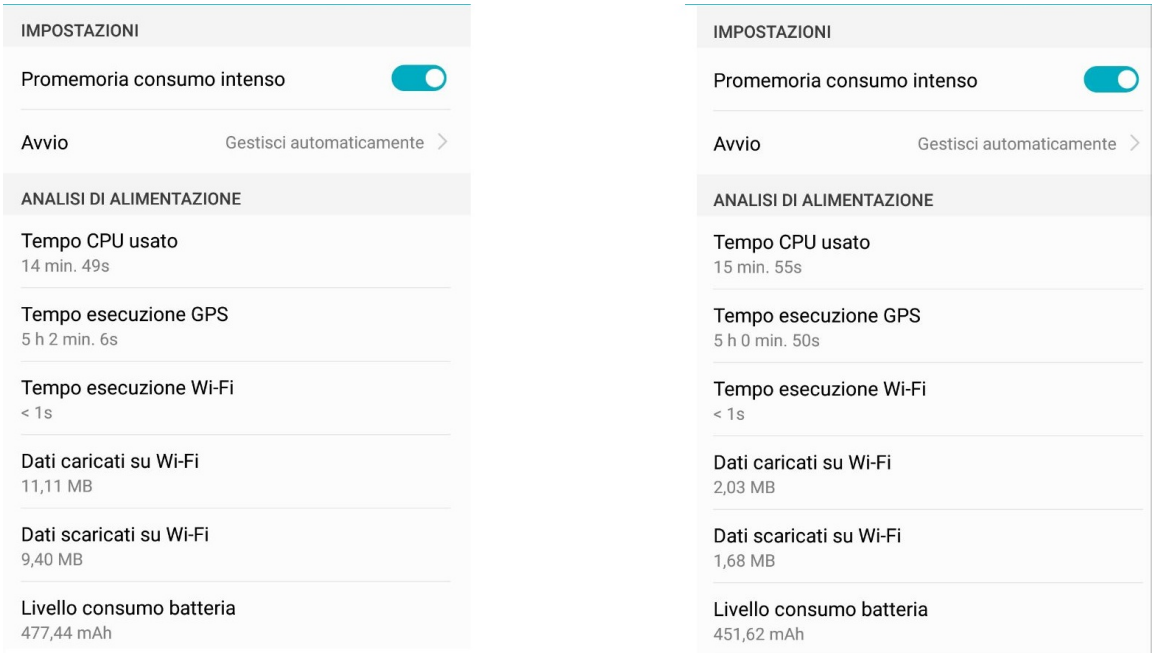


Figura 4.2: Informazioni sull'utilizzo delle risorse dall'applicazione, a sinitra il caso di studio con HTTP e a destra il caso di studio con MQTT.

4.2 Test sul consumo di banda

Per quanto riguarda il consumo di banda, le misurazioni dei dati mandati/scaricati e della banda nell'utilizzo dell'applicazione, si sono effettuate attraverso il programma *Wireshark* che verrà spiegato nella *Sezione 4.2.1*.
Come abbiamo detto precedentemente, la particolarità del client HTTP è di effettuare il polling periodico al server, ovvero di richiedere delle informazioni ad ogni intervallo di tempo. Questa caratteristica avrà dunque un impatto sulla quantità di dati trasferiti sia in upload e che in download, però dal momento che la lunghezza della stringa JSON che viene restituita al client in download, ovvero la quantità di byte scaricati, dipende sostanzialmente dal numero di client che sono connessi al server, una sua misurazione dipenderà anche da questo.
E dunque, la maniera più efficace per confrontare i due protocolli è considerare il caso in cui entrambi scaricano la posizione dello stesso numero di client. Inoltre, dato che il numero di byte scaricati aumenta all'aumentare dei client è sufficiente considerare il caso in cui si ha un unico client che manda la propria posizione e la scarica nel caso HTTP o la riceve dal broker nel MQTT.

Minuti	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8
10	1	0	0	0	0	1	1	1
30	3	2	2	2	1	2	2	2
50	5	5	5	5	4	5	5	5
60	8	6	6	6	6	6	6	6
70	9	7	8	8	7	7	7	7
80	12	9	9	9	8	9	9	9
90	13	11	11	11	10	10	11	11
100	15	12	12	12	11	12	12	12
110	16	14	13	13	13	13	13	13
120	18	15	15	15	14	14	15	15
130	20	17	16	16	16	16	16	16
140	22	18	18	17	17	17	18	18
150	23	20	19	19	19	19	19	19
160	25	21	21	21	20	20	20	21
170	26	22	22	22	22	22	21	22
180	28	24	23	24	23	23	23	24
190	29	26	25	25	25	25	25	25
200	31	27	26	27	26	26	26	27
210	32	28	28	28	28	27	28	28
220	33	30	30	30	29	29	29	30
230	35	31	31	31	31	30	31	31
240	37	33	33	33	32	32	32	33
250	38	34	34	34	34	33	34	34
260	40	36	35	36	35	35	35	36
270	42	37	37	37	37	46	37	37
280	43	39	39	39	38	38	39	39
290	45	40	40	40	39	39	40	40
300	46	42	42	42	41	41	42	42

Tabella 4.1: Consumo complessivo della batteria con il protocollo MQTT

Minuti	Test 9	Test 10	Test 11	Test 12	Test 13	Test 14	Test 15	Test 16
10	0	1	1	0	0	0	0	1
30	2	4	4	2	2	2	2	4
50	5	7	6	5	5	5	4	7
60	6	8	8	6	6	6	6	8
70	8	10	9	8	8	8	7	10
80	9	11	11	9	9	9	9	11
90	11	13	12	11	11	11	11	13
100	12	14	13	12	12	12	12	14
110	14	16	15	14	14	13	13	16
120	15	17	16	15	15	15	15	17
130	17	18	18	17	17	16	16	19
140	18	20	19	18	18	17	17	20
150	20	22	20	20	20	19	19	22
160	21	23	22	21	21	20	21	23
170	23	25	23	23	23	22	22	24
180	24	26	25	24	24	23	24	26
190	26	28	26	26	26	25	25	27
200	27	29	28	27	27	26	27	29
210	29	31	29	29	29	28	28	30
220	30	32	31	30	30	29	30	32
230	32	34	32	32	32	31	31	33
240	33	35	34	33	33	32	33	35
250	35	37	35	35	35	34	34	36
260	37	38	37	37	36	35	36	38
270	38	40	38	38	38	37	37	39
280	40	41	40	40	39	38	39	41
290	41	43	41	42	41	40	40	42
300	43	44	43	43	43	41	42	44

Tabella 4.2: Consumo complessivo della batteria con il protocollo MQTT

Minuti	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8
10	1	1	1	1	1	1	1	2
30	4	4	4	4	4	4	4	5
50	7	7	7	7	7	7	7	8
60	9	9	8	8	8	8	8	9
70	10	10	10	10	10	10	10	11
80	12	12	11	11	11	11	11	12
90	13	13	13	13	13	13	13	14
100	15	15	14	14	14	14	14	15
110	16	16	16	16	16	16	16	17
120	18	18	17	17	17	17	17	19
130	19	19	19	19	19	19	19	20
140	21	21	20	21	20	20	21	22
150	22	22	22	22	22	22	22	23
160	24	23	23	24	23	23	23	25
170	25	25	25	25	25	25	25	26
180	27	26	26	27	26	27	27	28
190	29	28	28	28	28	28	29	29
200	30	29	30	30	29	29	30	31
210	32	31	31	31	31	31	32	33
220	33	32	33	33	32	32	33	34
230	35	34	34	34	34	34	35	36
240	37	36	36	36	35	35	37	37
250	38	37	37	37	37	37	38	39
260	40	39	39	39	39	38	40	41
270	41	41	41	41	40	40	41	42
280	43	42	42	42	42	42	43	44
290	45	43	43	44	43	43	44	45
300	47	45	45	46	45	45	46	47

Tabella 4.3: Consumo complessivo della batteria con il protocollo HTTP

Minuti	Test 9	Test 10	Test 11	Test 12	Test 13	Test 14	Test 15	Test 16
10	0	1	1	0	1	0	1	1
30	2	4	4	2	3	2	4	4
50	5	7	7	5	6	5	7	6
60	6	8	8	6	8	6	9	8
70	8	10	10	8	9	8	10	9
80	9	11	11	9	11	9	12	11
90	11	13	13	11	12	11	13	12
100	12	14	14	12	14	12	14	14
110	14	16	16	14	15	14	16	15
120	15	17	18	15	17	15	17	17
130	17	19	19	17	18	17	19	19
140	18	20	21	18	20	18	20	20
150	20	22	22	20	22	20	22	21
160	21	23	24	21	23	21	23	22
170	23	25	25	23	25	23	25	24
180	24	26	27	24	26	25	26	25
190	26	28	29	26	28	26	28	27
200	27	30	30	27	29	28	29	28
210	29	31	32	29	31	29	31	30
220	31	33	33	30	32	30	32	31
230	32	34	34	32	34	32	34	33
240	34	36	37	33	36	34	35	34
250	35	37	38	35	37	36	37	36
260	37	39	40	36	39	38	39	37
270	38	41	42	38	41	39	40	39
280	40	42	43	39	42	40	42	40
290	41	44	45	41	44	42	43	42
300	43	45	47	43	46	43	45	43

Tabella 4.4: Consumo complessivo della batteria con il protocollo HTTP

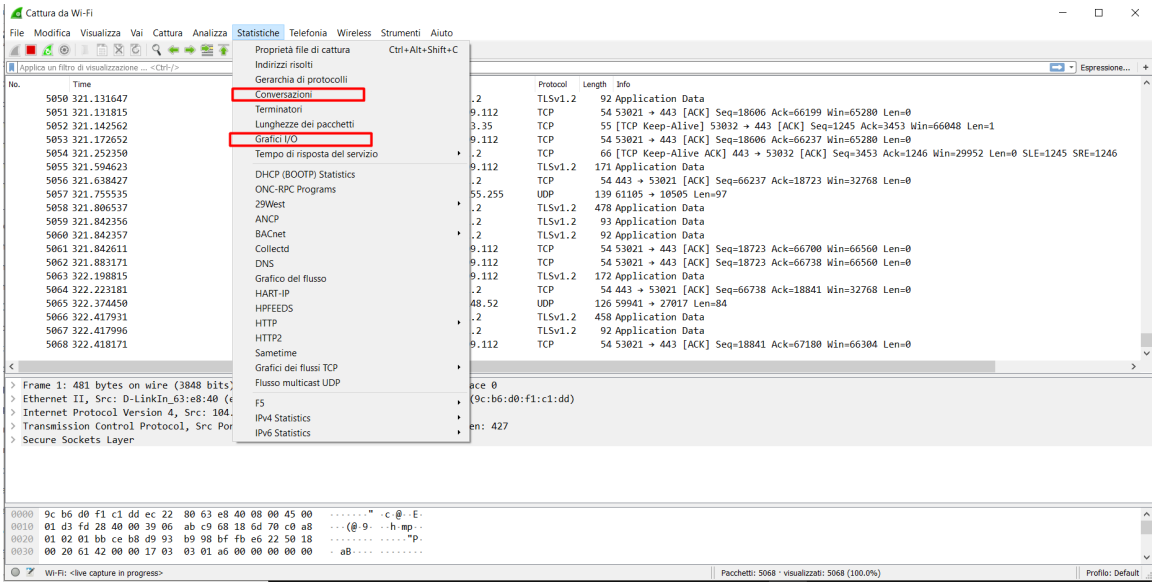


Figura 4.3: Interfaccia grafica di Wireshark, in colore rosso si evidenziano le due statistiche prese in considerazione.

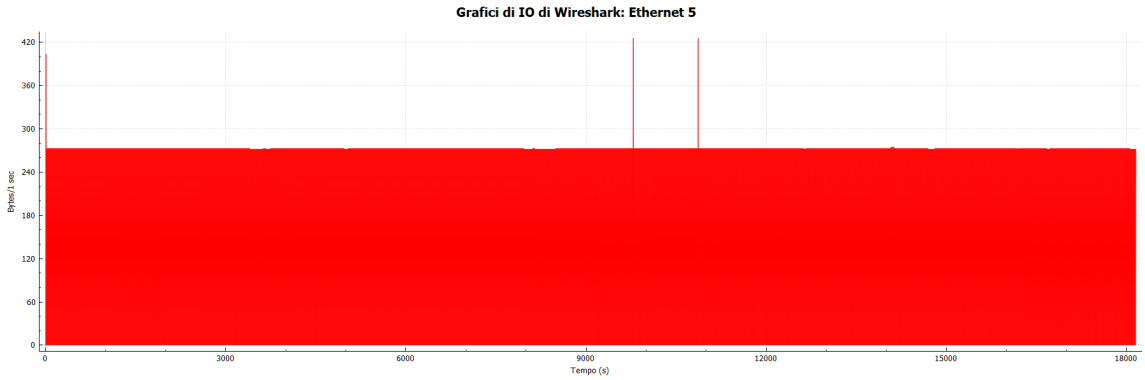


Figura 4.4: Grafico IO ottenuto durante l'utilizzo del protocollo MQTT durante 5 ore.

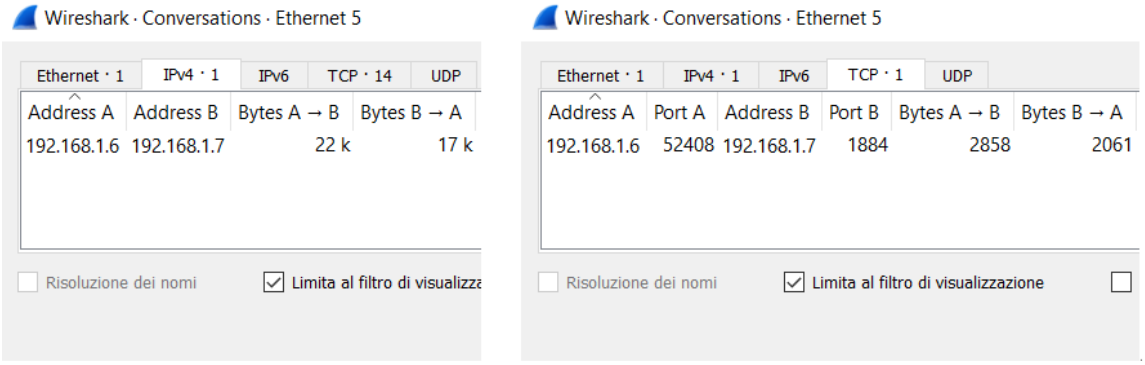


Figura 4.5: A sinistra informazioni sulla conversazione tra due host a livello IP e a destra tra due processi a livello TCP.

4.2.1 Consumo di banda

Il grafico in Figura 4.4 mostra il numero di byte trasmessi ad ogni secondo durante l'utilizzo dell'applicazione per 5 ore con il protocollo MQTT; in altre parole, questo grafico mostra quanta banda occupa l'applicazione con il protocollo MQTT, si osserva che il rate è abbastanza costante sull'intervallo di osservazione, questo è dovuto proprio dal fatto che si scaricano e si mandano dati con un singolo client. Nel Capitolo 5 verrà mostrato un confronto tra i grafici di questo tipo.
Nella Figura 4.5 si osserva invece come sono stati misurati i bytes in trasmissione col passare del tempo, i risultati ottenuti nella stessa modalità di test (descritta in *Sezione 4.1.1*) verranno mostrati nelle due seguenti sezioni.

4.2.2 Risultati consumo dati MQTT

Nelle Tabelle 4.5 e 4.6 vengono mostrati i dati (in Kilobytes) trasmessi dall'applicazione durante un utilizzo di 5 ore. Così come nel test per misurare la batteria consumata anche in questo caso si sono presi in considerazione 16 test con invertalli mostrati nella prima colonna delle tabelle. È importante notare che i risultati in download non stati presi in considerazione dato che all'aumentare dei client questo valore non può che aumentare e dunque è sufficiente tenere in considerazione solo il numero di byte in upload.

4.2.3 Risultati consumo dati HTTP

I risultati del consumo di dati con il protocollo HTTP vengono mostrati nelle Tabelle 4.7 e 4.8, i dati nelle tabelle sono stati ottenuti seguendo la stessa modalità di test con il protocollo MQTT. Osserviamo dunque che come ci aspettavamo il numero di byte aumenta notevolmente arrivando a raggiungere i 13 MB in un utilizzo di 5 ore, questi risultati verranno messi a confronto con il protocollo MQTT nel seguente capitolo.

Minuti	Dati trasmessi misurati in KB							
	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8
10	90	84	82	99	84	90	89	79
30	288	249	247	258	234	125	249	234
50	318	412	413	424	395	293	410	388
60	348	495	496	507	474	376	493	464
70	398	570	574	589	565	458	577	543
80	430	658	656	672	651	541	659	621
90	474	735	734	743	734	622	741	700
100	525	814	833	825	812	699	827	780
110	583	899	897	909	888	779	902	856
120	654	982	978	992	971	861	985	935
130	718	1062	1064	1075	1050	944	1062	1012
140	768	1141	1146	1150	1129	1027	1146	1090
150	819	1229	1225	1237	1207	1109	1229	1167
160	876	1274	1303	1325	1290	1192	1302	1244
170	930	1356	1388	1404	1371	1275	1392	1322
180	983	1441	1470	1570	1451	1356	1473	1401
190	1037	1533	1556	1570	1530	1439	1559	1479
200	1087	1600	1636	1653	1612	1516	1635	1562
210	1136	1665	1720	1730	1692	1600	1717	1650
220	1200	1741	1804	1813	1771	1682	1800	1720
230	1247	1824	1888	1895	1850	1763	1882	1798
240	1297	1912	1964	1978	1931	1843	1966	1871
250	1351	1984	2046	2060	2010	1926	2049	1950
260	1402	2067	2129	2144	2092	2009	2125	2029
270	1456	2134	2211	2227	2173	2092	2215	2109
280	1507	2221	2295	2309	2242	2173	2296	2181
290	1559	2297	2382	2391	2320	2257	2377	2377
300	1613	2370	2458	2468	2401	2342	2457	2457

Tabella 4.5: Quantità di bytes trasmessi dall'applicazione con il protocollo MQTT.

Minuti	Dati trasmessi misurati in KB							
	Test 9	Test 10	Test 11	Test 12	Test 13	Test 14	Test 15	Test 16
10	87	78	77	77	90	75	86	80
30	241	229	236	232	243	231	238	236
50	390	389	395	387	398	386	395	391
60	467	469	474	464	478	464	472	469
70	547	548	558	541	557	542	550	547
80	626	627	635	617	635	618	628	625
90	705	701	713	695	713	696	696	702
100	776	778	777	775	791	774	785	776
110	859	858	860	852	855	853	860	849
120	935	935	938	930	946	930	938	927
130	1017	1014	1012	1007	1026	1008	1015	1006
140	1096	1093	1087	1086	1104	1087	1095	1085
150	1170	1173	1167	1163	1184	1167	1173	1165
160	1254	1252	1250	1243	1243	1245	1252	1243
170	1334	1331	1328	1319	1334	1326	1329	1324
180	1409	1411	1407	1398	1411	1404	1407	1401
190	1487	1491	1487	1477	1489	1483	1485	1480
200	1566	1570	1570	1558	1566	1561	1566	1555
210	1645	1649	1650	1636	1643	1640	1645	1616
220	1724	1718	1728	1715	1724	1718	1723	1711
230	1803	1797	1806	1793	1801	1797	1801	1798
240	1882	1877	1877	1868	1880	1885	1879	1886
250	1962	1959	1954	1950	1960	1949	1959	1945
260	2049	2036	2032	2026	2036	2029	2036	2022
270	2120	2115	2110	2099	2112	2107	2115	2100
280	2186	2189	2190	2176	2199	2185	2194	2179
290	2265	2271	2267	2256	2277	2262	2271	2256
300	2349	2349	2351	2341	2354	2348	2354	2338

Tabella 4.6: Quantità di bytes trasmessi dall'applicazione con il protocollo MQTT.

Minuti	Dati trasmessi misurati in KB							
	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8
10	608	516	499	458	398	408	457	426
30	1495	1356	1290	1348	1323	1307	1106	1297
50	2406	2200	2157	2244	2224	2179	1963	2174
60	2848	2596	2599	2690	2675	2630	2401	2612
70	3261	3020	3023	3125	3128	3086	2823	3052
80	3711	3409	3487	3556	3581	3534	3192	3490
90	4155	3857	3919	4025	4034	3990	3516	3905
100	4606	4278	4364	4555	4555	4439	3851	4340
110	5061	4721	4721	4901	4953	4890	4093	4785
120	5478	5164	5246	5356	5413	5338	4541	5221
130	5933	5593	5653	5797	5867	5778	4980	5660
140	6389	6043	6095	6252	6284	6200	5385	6100
150	6835	6468	6531	6697	6737	6660	5773	6540
160	7279	6908	6952	7156	7191	7087	6230	6951
170	7730	7349	7363	7602	7641	7538	6687	7390
180	8181	7798	7800	8055	8094	8008	7144	7827
190	8627	8185	8234	8505	8500	8475	7583	8270
200	9077	8610	8673	8965	8953	8939	7894	8706
210	9517	9043	9095	9412	9406	9377	8268	9146
220	9977	9486	9521	9856	9852	9826	8699	9565
230	10326	9914	9934	10256	10298	10275	9074	9999
240	10744	10383	10344	10700	10745	10724	9404	10423
250	11194	10790	10787	11123	11159	11173	9792	10859
260	11607	11300	11189	11663	11643	11622	10249	11285
270	12007	11744	11600	12063	12043	12071	10706	11711
280	12553	12159	12024	12547	12590	12520	11003	12137
290	12997	12600	12465	13044	13060	13008	11580	12566
300	13400	13002	12998	13564	13624	13418	11917	13016

Tabella 4.7: Quantità di bytes trasmessi dall'applicazione con il protocollo HTTP.

Minuti	Dati trasmessi misurati in KB							
	Test 9	Test 10	Test 11	Test 12	Test 13	Test 14	Test 15	16
10	457	493	412	425	461	452	433	449
30	1330	1331	1298	1260	1317	1319	1294	1293
50	2193	2204	2170	2135	2196	2182	2158	2166
60	2631	2620	2596	2568	2631	2601	2664	2565
70	3047	3055	2999	3000	3065	3030	3096	3003
80	3490	3493	3427	3432	3500	3471	3494	3434
90	3928	3932	3869	3869	3928	3901	3917	3872
100	4367	4369	4288	4290	4353	4340	4351	4313
110	4785	4812	4748	4742	4782	4776	4773	4740
120	5223	5244	5173	5202	5221	5215	5213	5157
130	5663	5683	5604	5633	5651	5646	5637	5605
140	6117	6123	6043	6058	6060	6083	6070	6120
150	6548	6548	6470	6497	6493	6516	6505	6529
160	6979	6982	6909	6932	6927	6952	6938	6936
170	7425	7409	7331	7401	7357	7382	7386	7364
180	7848	7844	7766	7846	7808	7813	7811	7999
190	8281	8284	8188	8218	8239	8248	8251	8233
200	8719	8725	8651	8656	8667	8682	8684	8660
210	9173	9156	9084	9087	9098	9118	9119	9091
220	9599	9597	9522	9528	9529	9554	9548	9520
230	10056	10093	9950	9975	9969	9938	9984	9953
240	10513	10594	10362	10400	10430	10390	10417	10402
250	10970	11043	10874	10825	10891	10842	10850	10851
260	11427	11539	11235	11250	11356	11277	11302	11300
270	11884	12032	11698	11675	11826	11727	11754	11749
280	12341	12528	12110	12105	12287	12180	12187	12198
290	12790	13025	12522	12430	12737	12636	12617	12648
300	13260	13525	12934	12840	13190	13092	13050	13100

Tabella 4.8: Quantità di bytes trasmessi dall'applicazione con il protocollo HTTP.

Capitolo 5

Confronto prestazionale

In questa parte finale si riprendono i concetti su come avviene la comunicazione attraverso i due protocolli e si mettono a confronto i risultati ottenuti nel Capitolo 4. Dunque, nei seguenti paragrafi per ogni fattore di studio verranno mostrate le aspettative teoriche, un grafico che rende visibile la differenza dei risultati ottenuti con i due protocolli.

5.1 Consumo di batteria

Dopo aver effettuato i test e ottenuto i risultati mostrati nel *Capitolo 4* si potrebbe pensare che i test effettuati non siano quelli più appropriati per un confronto prestazionale tra i due protocolli dal momento che si sono considerati situazioni in cui è presente un unico client che invia e riceve la propria posizione e non situazioni più complesse in cui sono presenti più client. In realtà però, la situazione considerata è quella più interessante dato che si sta cercando di confrontare il caso in cui entrambi i protocolli potrebbero funzionare bene, ovvero il caso in cui entrambi potrebbero dare risultati molto diversi anche quando la situazione è la più semplice possibile. Per mostrare che la situazione considerata nel Capitolo 4 è quella più appropriato possiamo osservare i seguenti risultati:

Numero di client	MQTT	HTTP
1	15%	17%
3	16%	19%
5	17%	19%
10	17%	20%

Tabella 5.1: Risultati sul consumo di batteria all'aumentare dei client in un test di due ore.

In Tabella 5.1 si osserva che all'aumentare dei client la differenza dell'uso della batteria tende ad aumentare con entrambi i protocolli ma l'incremento nel caso HTTP è leggermente più veloce rispetto al caso MQTT.
Riprendendo quanto detto nel Capitolo 1, sappiamo che la struttura architetturale su cui il

protocollo MQTT è basato ci permette di ridurre il numero di istruzioni eseguite rispetto all'HTTP dato che nel caso MQTT non è necessario ristabilire la connessione ogni volta che si vuole inviare/richiedere dati, e dunque a minor numero di istruzioni eseguite minore dovrebbe essere l'impiego della CPU e successivamente si dovrebbe consumare una minore percentuale di batteria nel utilizzo prolungato.
Dunque, considerando i risultati mostrati nel capitolo precedente è possibile mostrare graficamente la differenza ottenuta con i due protocolli.

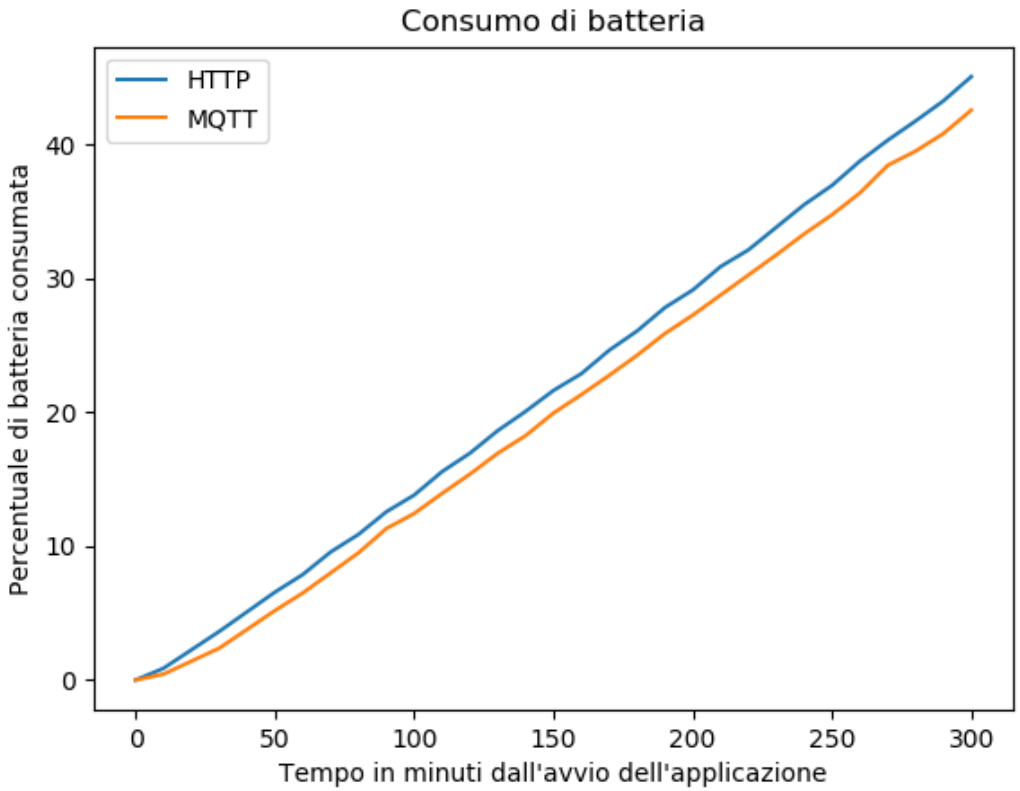


Figura 5.1: Batteria consumata in 5 test con un utilizzo di 5 ore per ogni test.

La Figura 5.1 mostra graficamente ciò che abbiamo ottenuto nella *Sezione 4.1*, si osserva in effetti che la retta blu rappresenta il valor medio della batteria consumata con il protocollo HTTP in tempo di 5 ore, e la retta in colore arancione il valor medio della batteria consumata con il protocollo MQTT nelle stesse modalità di test. Queste due rette tendono ad essere abbastanza stabili dati diversi motivi che verranno spiegati nei seguenti paragrafi.
Dal momento che si parla del consumo di una risorsa limitata (non ricaricabile durante l'esecuzione del test) questa funzione sia per MQTT che per HTTP non può che essere una funzione monotona crescente e giustamente in figura non si osservano picchi.
Inoltre, nel caso HTTP questa funzione è abbastanza stabile dal momento che si utilizza l'architettura REST dove per ogni richiesta viene stabilita la connessione e successivamente viene chiusa (caratteristica *stateless* delle architetture REST). Nel caso MQTT invece, dal momento che la connessione è unica e verso il broker, l'andamento del consumo della bat-

teria non può che rimanere stabile.
È importante osservare che la pendenza della retta del consumo con MQTT è leggermente meno elevata rispetto a quella del HTTP, in effetti in Figura 5.2 osserviamo come la funzione definita come BatteryUsage(HTTP)-BatteryUsage(MQTT), tende a crescere leggermente con il tempo. In questo modo dunque sappiamo che aumentando la durata dei test la distanza tra la funzione BatteryUsage(HTTP) e BatteryUsage(MQTT) tenderà a crescere.

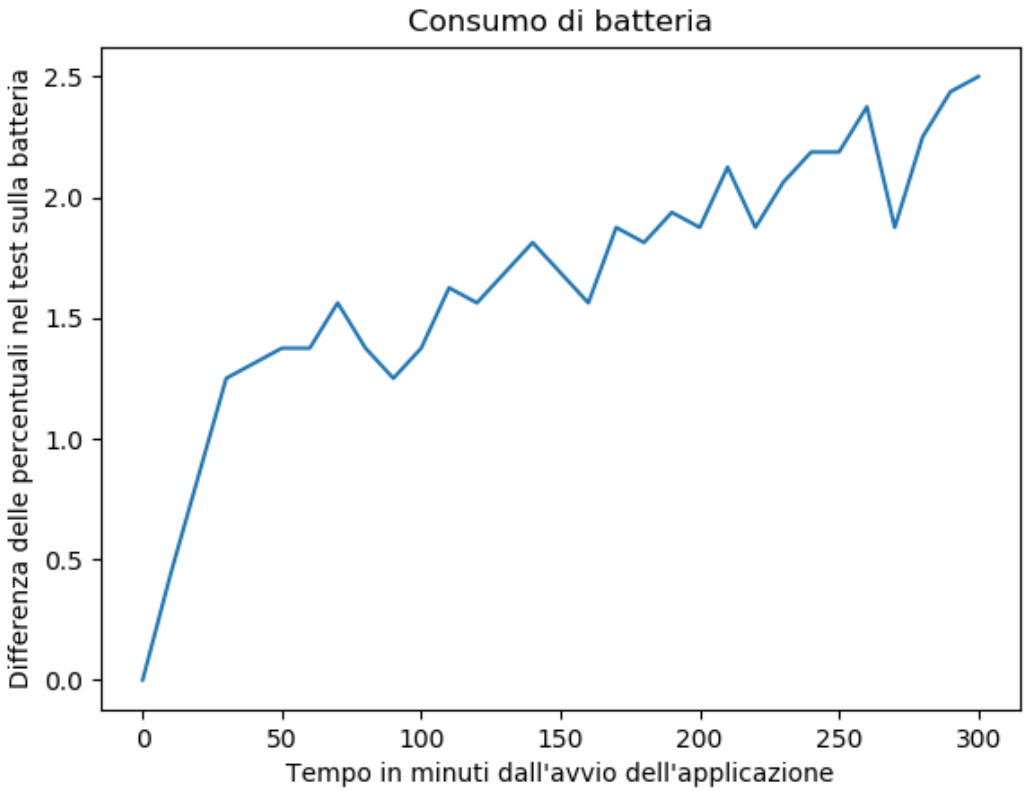


Figura 5.2: Differenza tra i consumi di batteria delle due funzioni mostrate in Figura 5.1.

Il motivo per cui il grafico in Figura 5.2 contiene dei picchi è dovuto al fatto che, nonostante si misuri una percentuale del consumo di una risorsa, si sta considerando la differenza di due funzioni, ovvero in alcuni istanti di tempo è possibile che la funzione BatteryUsage(HTTP) assuma valori non troppo distanti come ad esempio 1-2%, e per questo motivo una piccola variazione è plausibile.

5.2 Consumo dei dati

Per quanto riguarda il consumo dei dati, sappiamo che da una parte il protocollo HTTP ogni volta che vuole richiedere qualcosa al server deve stabilire la connessione, inviare il messaggio e chiudere la connessione, dunque ad ogni secondo di tempo vengono seguiti questi 3 passi

e ogni 1-2 secondi vengono seguiti nuovamente per mandare la propria posizione. Dall'altra parte invece il protocollo MQTT non ha bisogno di stabilire e chiudere la connessione tutte le volte, è compito del broker mantenere la connessione con il client e dunque l'unico possibile aumento dei dati è dato dall'aumentare dei client che pubblicano sotto il topic a cui ci si sottoscrive, ma questo succede anche nel caso HTTP dato che questi messaggi vengono scaricati come parte della stringa JSON e quindi la differenza sostanziale sono i messaggi di creazione e chiusura della connessione.
Riprendendo i risultati mostrati nel *Capitolo 4* possiamo creare un grafico che rende visibile la quantità di byte che ogni protocollo ha consumato in 5 ore.

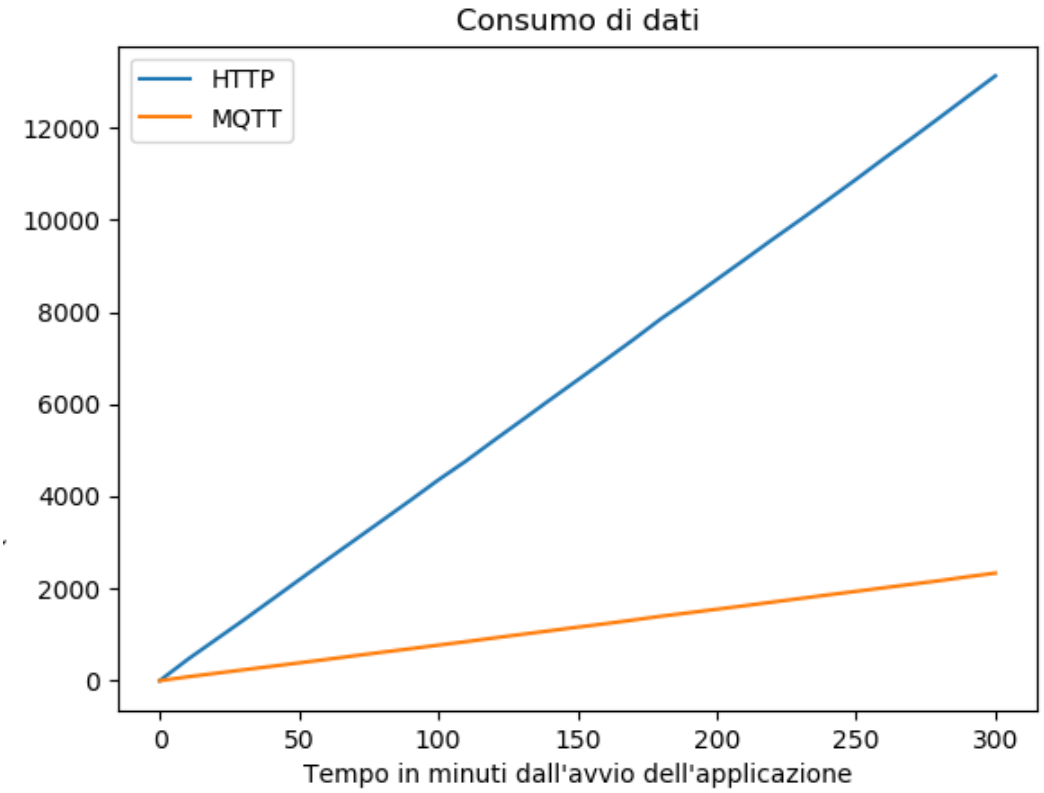


Figura 5.3: Consumo dei dati in MB con i due protocolli.

In Figura 5.3 notiamo che anche nel caso più semplice in cui si ha un solo client che manda e riceve la propria posizione la differenza tra le due funzioni è enorme, in 16 test con un utilizzo di 5 ore l'applicazione realizzata che comunica con HTTP ha consumato tra 12 e 13 Megabytes. mentre nel caso MQTT il consumo è stato soltanto circa 2 Megabytes.

Oltre a notare il consumo dei dati del client è utile notare quanto si occupa la rete con ogni protocollo, ovvero il rate con cui si manda e riceve dati nella rete, per visualizzare questo possiamo utilizzare una delle statistiche offerte da Wireshark.

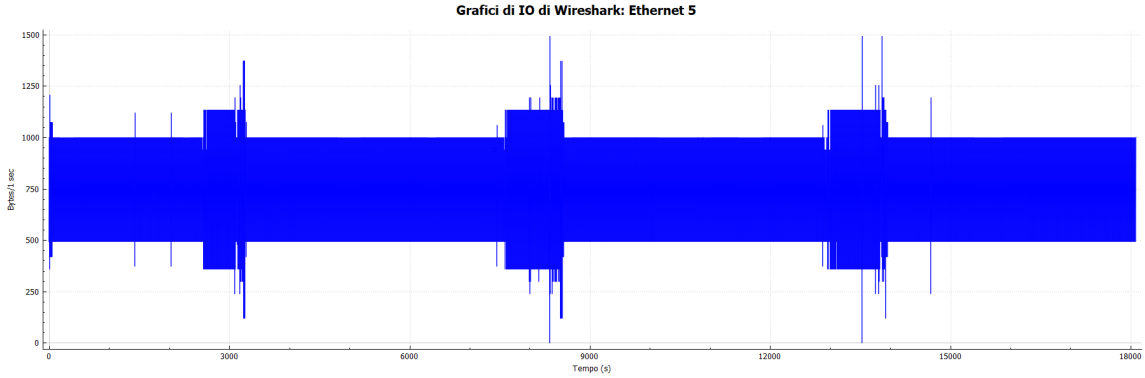


Figura 5.4: Grafico I/O che mostra il rate in bytes/seconds con cui viaggiano i messaggi nella rete, caso HTTP.

In Figura 5.4 osserviamo che l'applicazione occupa la rete con 500-1000 bytes per ogni secondo della comunicazione con eccezione di alcuni istanti di tempo in cui possono essere successe due cose: la perdita di un pacchetto che viene poi ritrasmesso con un suo corrispondente aumento dei messaggi nella rete o il ritardo di pacchetti che vengono ad aumentare la misurazione in intervalli di tempi successivi. Si osserva infatti che il grafico è abbastanza simetrico dato che queste due possibili problemi aumentano e diminuiscono il rate nello stesso modo.

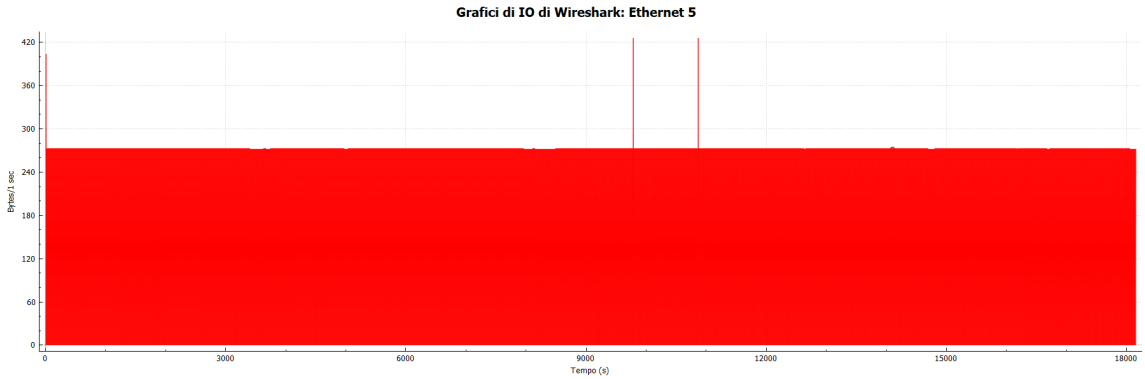


Figura 5.5: Grafico I/O che mostra il rate in bytes/seconds con cui viaggiano i messaggi nella rete, Ccaso MQTT.

In Figura 5.5 si mostra invece il grafico I/O che corrisponde al protocollo MQTT, osserviamo che l'occupazione di banda nel caso MQTT è generalmente tra 240 e 260 bytes/sec con eccezione di 3 istanti di tempo in cui molto probabilmente un pacchetto a livello tcp è stato perso ed è stata richiesta la sua ritrasmissione mostrando appunto un rate leggermente minore al doppio di quello normale. Si osserva inoltre che il rate nel caso MQTT può arrivare circa a 0, questo è dovuto dal fatto che il client MQTT non effettua il polling,

ovvero non invia richieste al server ogni secondo dato che è compito del broker mantenere la connessione con i client.

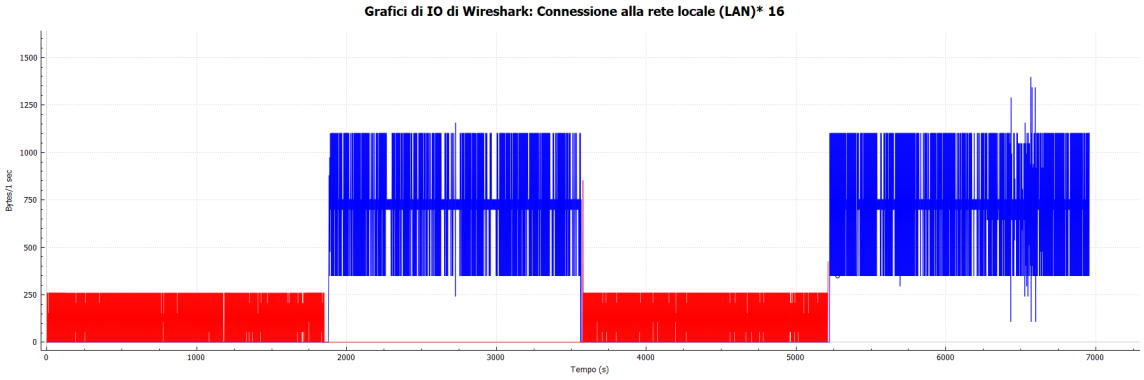


Figura 5.6: Grafico I/O che mostra il rate in bytes/seconds con cui viaggiano i messaggi nella rete, Ccaso MQTT e HTTP insieme.

In Figura 5.6 osserviamo invece l'occupazione di banda alternando il protocollo MQTT con HTTP con intervalli di 30 minuti per ciascuno, si osserva infatti lo stesso comportamento dei due grafici precedenti, il protocollo MQTT ha rate di circa 250 bytes/sec e il protocollo HTTP ha invece un rate che varia tra 400-1100 bytes/sec che è un intervallo leggermente maggiore rispetto alla Figura 5.4 ma comunque accettabile per il confronto. si osserva inoltre, che nell'ultimo intervallo di tempo HTTP presenta di nuovo i picchi osservati nel suo grafico.

Conclusioni

Considerando quanto è stato mostrato nel Paragrafo 5.1, possimamo concludere che le nostre aspettative teoriche siano state soddisfatte anche se la differenza tra il consumo dei due protocolli non è molto elevata, come si osserva in Figura 5.1. È importante notare però, che come abbiamo mostrato in Figura 5.2. all'aumentare del tempo la differenza tra le due funzioni tende a crescere, e dunque, HTTP tende a consumire più batteria rispetto al protocollo MQTT; e per questo motivo possiamo anche concludere che in una giornata lavorativa in cui si potrebbe pensare di utilizzare l'applicazione per più di 8 ore, con il protocollo HTTP si utilizerebbe più batteria rispeto al protcollo MQTT. E per questo motivo, possiamo finalmente concludere che la scelta di utilizzare il protocollo HTTP per permettere a dispositivi IoT di comunicare, risulta non ottimale dal momento che questi hanno risorse limitate e un risparmio energetico è più che necessario.

Per quanto riguarda invece il consumo dei dati, come abbiamo mostrato nel Paragrafo 5.2, possiamo concludere che le nostre aspettative teoriche siano state soddisfatte anche in questo caso, nonostante il protocollo HTTP sia semplice da implementare e ci sia un'ampia varietà di framework e librerie che lo rendono ancora più semplice e facile da utilizzare, presenta comunque un grave problema al dover utilizzare una gran quantità di dati. Per capire che questo problema, comparato con il consumo di batteria è molto più grave, è utile mostrare i costi reali che si avrebbero con entrambi i protocolli.
Prendendo in considerazione i costi che si trovano nel sito ufficiale di uno dei più grandi distributori di schede SIM per dispositivi IoT, Things Mobile [29], possiamo mostrare quale sarebbe il risparmio monetario con il protocollo MQTT.
Sia 0.10 €/MB la tariffa a consumo più ecconomica, offerta da Things Mobile, possiamo valutare quali sarebbero i costi nel nostro caso di studio.
In un utilizzo di 5 ore HTTP consuma tra 12 e 13 megabytes e dunque il costo, **per dispostivo**, sarebbe almeno di:

$$12\text{MB} * 0.1€/\text{MB} = 1.2\ €$$

Per quanto riguarda il caso MQTT invece, il costo per dispositivo in un utilizzo di 5 ore sarebbe di:

$$2\text{MB} * 0.1€/\text{MB} = 0.2\ €$$

E dunque, risulta evidente che rispetto al caso HTTP, MQTT offre non solo un minor consumo di batteria ma anche un minor consumi di dati e dunque anche un minor costo in euro per ogni dispositivo con accesso a internet che si vuole utilizzare.
Date queste due conclusioni risultano evidenti i motivi per cui nella Figura 2, HTTP non è presente tra le alternative di protocolli per dispisitivi IoT, e si utilizza più comunemente il protocollo MQTT.

Appendice A

Implementazione del caso di studio

Codice in Java per la realizzazione dell'applicazione

MainActivity.java

```
public class MainActivity extends AppCompatActivity {

    // in alcuni casi è utile avere un riferimento al main activity
    // in questo caso l'activity deve modificare l'informazione
    // della batteria quando la mappa viene chiusa
    public static MainActivity main;

    Button btnStartMap;
    RadioGroup radio_group;
    String client;
    String host;
    int port;
    // Questo RelativeLayout verrà usato per mostrare
    // l'informazione sulla batteria (cioè dopo il click su Start Map)
    RelativeLayout layoutInfo;
    // Metodo usato per visualizzare la percentuale della batteria
    public static int getBatteryPercentage(Context context) {
        // preso da StackOverFlow
        IntentFilter iFilter = new
        ↪  IntentFilter(Intent.ACTION_BATTERY_CHANGED);
        Intent batteryStatus = context.registerReceiver(null, iFilter);

        int level = batteryStatus != null ?
        ↪  batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) : -1;
        int scale = batteryStatus != null ?
        ↪  batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1) : -1;

        float batteryPct = level / (float) scale;

        return (int) (batteryPct * 100);
    }
```

MainActivity.java

```
// Metodo usato per mostrare l'ora attuale
private static String getReminingTime() {
    // preso da StackOverFlow
    String delegate = "hh.mm.ss aaa";
    return (String)
     ↪  DateFormat.format(delegate,Calendar.getInstance().getTime());
}

// Questo metodo viene chiamato da MapsActivity in modo
// da aggiornare lo stato della batteria e il tempo in cui
// la Mappa è stata chiusa
public void updateCurrentBatteryInfo(){
    TextView batteryLevel = (TextView) findViewById(R.id.BatteryLevelView2);
    TextView battertTime = (TextView) findViewById(R.id.BatteryTimeView2);

    batteryLevel.setText("Battery Level:
     ↪  "+getBatteryPercentage(MainActivity.this)+"%");

    battertTime.setText("Time:   "+getReminingTime());
}
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    btnStartMap = (Button) findViewById(R.id.btnStartMap);
    radio_group = (RadioGroup) findViewById(R.id.radioGroup);
    prepareForMapActivity();
}

// questo metodo prepara l'activity per la Mappa
// contiene il metodo getInput() e verifiyInput()
public void prepareForMapActivity(){
btnStartMap.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // inizio lettura dell'input
            String protocol;
            int selectedId = radio_group.getCheckedRadioButtonId();

            if(selectedId == R.id.radioMQTT){
                protocol = "MQTT";
            }else{
                protocol = "HTTP";
            }
            String l_client = ((EditText)findViewById(R.id.clientId))
            .getText().toString().trim();
```

MainActivity.java

```
                if(!l_client.equals(""))
                    client= l_client;
                else {
                    Toast.makeText(MainActivity.this, "Please insert a valid
                    ↪  client id", Toast.LENGTH_LONG).show();
                    return;}
                String l_host  = ((EditText)findViewById(R.id.serverId))
                .getText().toString().trim();
                if(!l_host.equals(""))
                    host = l_host;
                else{
                    Toast.makeText(MainActivity.this, "Please insert a valid
                    ↪  host", Toast.LENGTH_LONG).show();
                    return;}
                String l_port = ((EditText)findViewById(R.id.port))
                .getText().toString().trim();
                if(!l_port.equals(""))
                    port = Integer.valueOf(l_port);
                else{
                    Toast.makeText(MainActivity.this, "Please insert a valid
                    ↪  port", Toast.LENGTH_LONG).show();
                    return;}
                // end reading input
                // ora passo i valori letti al MapActivity
                Intent i = new Intent(MainActivity.this,
                ↪  MapsActivity.class);
                i.putExtra("protocol", protocol);
                i.putExtra("client_id",client);
                i.putExtra("host", host);
                i.putExtra("port",port);

                // mostriamo informazioni sulla batteria prima di avviare la
                ↪  mappa
                layoutInfo = (RelativeLayout)
                ↪  findViewById(R.id.RelativeInfoView);
                layoutInfo.setVisibility(View.VISIBLE);
                TextView batteryLevel = (TextView)
                ↪  findViewById(R.id.BatteryLevelView);
                TextView battertTime = (TextView)
                ↪  findViewById(R.id.BatteryTimeView);

                batteryLevel.setText("Battery Level:
                ↪  "+getBatteryPercentage(MainActivity.this)+"%");
                battertTime.setText("Time:    "+getReminingTime());
                startActivity(i);
            }
        });
    }
}
```

MyClient.java

```
public abstract class MyClient {
    // Variabili utilizzate per connettersi al server
    protected int port;
    protected String client_id;
    protected String host;

    // Un riferimento per interagire con GoogleMap API
    private GoogleMap mGoogleMap;

    // Gli altri client verranno mostrati come markers
    // con delle icone prese da
    ↪ https://github.com/googlemaps/js-marker-clusterer/
    private HashMap<String, Marker> clients;
    MyClient(final String client_id, final String host, final int port,
    ↪ GoogleMap mGoogleMap){
        this.clients = new HashMap<>();
        this.host=host;
        this.client_id=client_id;
        this.port=port;
        this.mGoogleMap=mGoogleMap;
    }
    // Dal momento che mostrare un client sulla mappa è una cosa che
    ↪ entrambi i client (http e mqtt)
    // faranno, torna comodo inserire il metodo qui
    public void updateMap(String r_client_id, LatLng latLng) {
        // se il client che stiamo aggiornando è già presente nella nostra
        ↪ lista di clients
        // lo togliamo in modo da riaggiungerlo con una nuova posizione
        if(clients.containsKey(r_client_id) &&
        ↪ clients.get(r_client_id)!=null) {
            clients.get(r_client_id).remove();
        }
        MarkerOptions markerOptions = new MarkerOptions();
        markerOptions.position(latLng);
        markerOptions.title(r_client_id + " Position");
        markerOptions.icon(BitmapDescriptorFactory.fromAsset("m3.png"));
        Marker m = mGoogleMap.addMarker(markerOptions);
        clients.put(r_client_id,m);
    }
    public void setGoogleMap(GoogleMap mGoogleMap){
    ↪ this.mGoogleMap=mGoogleMap; }
    public int getPort() { return port; }
    public String getClientId() { return client_id; }
    public String getHost() { return host;    }
    // Tutti i client devono pulire la connessione
    abstract public void cleanConnection()throws Exception;
    // Tutti i client devono inviare la propria posizione
    abstract public void sendPosition(String position);
}
```

Codice per l'implementazione di GoogleLocation API

MapsActivity.java

```
public class MapsActivity extends AppCompatActivity
        implements OnMapReadyCallback {

    // attributi necessari per usare GoogleAPI
    // parte di codice generata in automatico da AndroidStudio
    GoogleMap mGoogleMap;
    SupportMapFragment mapFrag;

    // questa variabile è quella più interessante per il nostro caso
    // permette di ricevere la posizione del dispositivo ogni tot secondi
    LocationRequest mLocationRequest;
    FusedLocationProviderClient mFusedLocationClient;
    Location mLastLocation;

    // riferimento al client in modo da inviare messaggi
    // quando l'oggetto viene creato, questo in automatico
    // inizia a ricevere i messaggi dal server
    MyClient client;

    // Quando l'applicazione viene minimizzata è inutile continuare
    // a essere notificati della posizione, per cui firstTime
    // ci permette di non ricevere più questo messaggio che contiene la
    ↪  posizione
    boolean firstTime=false;
    // quando si riprende l'app è utile poter ricevere i messaggi nuovamente
    boolean removedLocationUpdated=false;

    @Override
    protected void onDestroy() {
        try {
            // quando questo activity viene chiuso
            // la connessione deve essere pulita
            client.cleanConnection();
        }catch(Exception e){
            Log.d("EXCEPTION DISCONNECTING", e.getMessage());
        }
        // e si aggiorna l'info sulla batteria
        MainActivity.main.updateCurrentBatteryInfo();
        super.onDestroy();
    }
```

MapsActivity.java

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_maps);
    Bundle extras = getIntent().getExtras();
    if(extras !=null) {
        // riprendo i messaggi inviati dal MainActivity
        String which = extras.getString("protocol");
        String client_id = extras.getString("client_id");
        String host = extras.getString("host");
        int port = extras.getInt("port");

        // si crea il client scelto dall'utente
        if(which.equals("HTTP")){
            client = new MyHttpClient(client_id, host, port, mGoogleMap);
        }else if (which.equals("MQTT")){
            client = new MyMqttClient(client_id, host, port, mGoogleMap);
        }
    }
    getSupportActionBar().setTitle("Map Location Activity");
    mFusedLocationClient =
    ↪  LocationServices.getFusedLocationProviderClient(this);

    mapFrag = (SupportMapFragment)
    ↪  getSupportFragmentManager().findFragmentById(R.id.map);
    mapFrag.getMapAsync(this);
    // quando l'applicazione entra in funzione è utile che lo schermo
    // rimanga accesso anche quando non lo si tocca
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
}
@Override
public void onMapReady(GoogleMap googleMap) {
    mGoogleMap = googleMap;
    client.setGoogleMap(mGoogleMap);
    mLocationRequest = new LocationRequest();
    // si richiede la posizione ogni 2 secondi
    mLocationRequest.setInterval(2000);
    // in alcuni casi il sistema operativo potrebbe darci la posizione
    // più frequentemente se altre app la richiedono
    // per cui è necessario impostare un massimo di frequenza
    mLocationRequest.setFastestInterval(1000);
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    mGoogleMap.getUiSettings().setMapToolbarEnabled(false);
```

MapsActivity.java

```
    if (android.os.Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_FINE_LOCATION)
                == PackageManager.PERMISSION_GRANTED) {
            //Location Permission already granted
            mFusedLocationClient.requestLocationUpdates(mLocationRequest,
             ↪  mLocationCallback, Looper.myLooper());
            mGoogleMap.setMyLocationEnabled(true);
        } else { checkLocationPermission(); }
    } else {
        mFusedLocationClient.requestLocationUpdates(mLocationRequest,
         ↪  mLocationCallback, Looper.myLooper());
        mGoogleMap.setMyLocationEnabled(true);
    }
}
// la gestione delle risposte che contengono la posizione va fatta
 ↪  attraverso una Callback
LocationCallback mLocationCallback = new LocationCallback() {
    @Override
    public void onLocationResult(LocationResult locationResult) {
        List<Location> locationList = locationResult.getLocations();
        if (locationList.size() > 0) {
            // Quando la posizione è pronta per essere utilizzata la si
             ↪  prende
            // Si trova nell'ultima posizione disponibile
            Location location = locationList.get(locationList.size() - 1);
            mLastLocation = location;
            // si crea un oggetto latLng in modo da permettere di muovere la
             ↪  vista dello schermo
            LatLng latLng = new LatLng(location.getLatitude(),
             ↪  location.getLongitude());
            // si prepara la stringa JSON da inviare
            String string_json = "{ \"client_id\":
             ↪  \""+client.getClientId()+"\", \"latitude\":
             ↪  "+latLng.latitude+", \"longitude\": "+latLng.longitude+"}";
            // la stringa con la posizone viene inviata al server MQTT o
             ↪  HTTP
            client.sendPosition(string_json);
            if(!firstTime) { // la prima volta facciamo uno zoom alla propia
             ↪  posizione
            mGoogleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(
            latLng, 11));
                firstTime=true;
            }
        }
    }
};
}
```

Codice per la realizzazione del client HTTP

MyHttpClient.java

```
public class MyHttpClient extends MyClient {

    private Timer timer;
    private TimerTask task;

    public MyHttpClient(final String client_id, final String host,
                        final int port, GoogleMap mGoogleMap) {
        super(client_id, host, port, mGoogleMap);

        // Una volta creato il Client, bisogna fare il polling al server in
        ↪  modo da richiedere la posizione degli altri client.
        final Handler handler = new Handler();
         timer = new Timer();
         task = new TimerTask() {
            @Override
            public void run() {
                handler.post(new Runnable() {
                    public void run() {
                        // richiedo la posizione a
                        // http://host_server:8080/get-all-position/
                        new RequestPositionsTask(MyHttpClient.this).execute(
                            "http://"+MyHttpClient.this.host+":"+
                            MyHttpClient.this.port+"/get-all-position");
                    }
                });
            }
        };
         // si richiede la posizione ogni 1000 ms
        timer.schedule(task, 0, 1000);
    }

    // Nello stesso modo della ricezione della posizione degli altri client
    // quando si invia una richiesta HTTP si utilizza un Asyntask
    @Override
    public void sendPosition(String position) {
        new SendPositionTask().execute(
        "http://"+host+":"+port+"/clients/positions",position);
    }
```

MyHtppClient.java

```
// Dal momento che non si deve bloccare l'interfaccia grafica
// mandando richieste HTTP, è obbligatorio usare una delle alternative tra
↪  Threads, Services o AsyncTasks
// in questo caso la ricezione delle posizioni viene fatta tramite un
↪  AsyncTask (data la sua semplicità)
private static class RequestPositionsTask extends  AsyncTask<String, Void,
↪  String>{
    // Una volta ricevuto il messaggio dal server bisogna mostrare le
    ↪  posizioni sulla mappa
    // per cui, è necessario un riferimento al client
    private WeakReference<MyHttpClient> activity_reference;

    RequestPositionsTask(MyHttpClient client){
        activity_reference = new WeakReference<>(client);
    }

    private String requestHttp(String url_to_download){
        String result="";
        HttpURLConnection httpURLConnection = null;
        try {
            // si crea l'url a cui ci si connetterà
            URL url = new URL(url_to_download);
            // si crea la connessione
            httpURLConnection = (HttpURLConnection) url.openConnection();
            // si abilita lo streaming http senza buffering quando la
            ↪  lunghezza del messaggio non è nota
            // 0 in modo da usare un valore di default
            httpURLConnection.setChunkedStreamingMode(0);
            httpURLConnection.setReadTimeout(10000);
            httpURLConnection.setConnectTimeout(15000);
            httpURLConnection.setRequestMethod("GET");
            httpURLConnection.setDoInput(true);
            InputStream in = httpURLConnection.getInputStream();
            // si legge l'input ricevuto dal server (la stringa json con le
            ↪  posizioni)
            result = readStream(in);
        }catch(MalformedURLException mal){
            Log.d("*** FROM HTTP**", "MAL FORMED" + mal.getMessage());
        }catch(IOException io){
            Log.d("*** FROM HTTP**", "IO-" + io.getMessage());
        }
        finally{
            if(httpURLConnection!=null)
                httpURLConnection.disconnect();
        }
        return result;
    }
```

MyHttpClient.java

```
@Override
protected String doInBackground(String... strings) {
    // strings[0] contiene l'url su cui mandare la richiesta
    return requestHttp(strings[0]);
}
@Override
protected void onPostExecute(final String s) {
    // una volta che la risposta è pronta bisogna aggiornare la mappa
    MyHttpClient client = activity_reference.get();
    JSONArray json;
    try{
        json = new JSONArray(s);
        for (int i = 0; i < json.length(); i++) {
            JSONObject json_from_get = new JSONObject(json.getString(i));
            String id = json_from_get.getString("id");
            // ci interessa di far visualizzare solo la posizione degli
            ↪  altri
            if(!id.equals(client.client_id)){
                JSONObject j_latlong = new
                ↪  JSONObject(json_from_get.getString("name"));

                // bisogna modificare leggermente la posizione dato che
                ↪  l'icona dei marker non è centrata
                // ha come riferimento la parte sinistra superiore
                LatLng latLng = new
                ↪  LatLng(j_latlong.getDouble("latitude")-0.00000534,
                ↪  j_latlong.getDouble("longitude")+0.0000008);
                // aggiorno il client con id 'id' e la posizione 'latLng'
                client.updateMap(id, latLng);
            }
        }
     }catch(JSONException mal){
        Log.d("*** ASYNC RECEIVE", "JSON EXCEPTION "+mal.getMessage());
    }
    client=null;
    super.onPostExecute(s);
}
```

MyHttpClient.java

```
    // metodo utilizzato per leggere la risposta di un server HTTP
    private String readStream(InputStream in) {
    // preso da stackoverflow, il link è in una delle pagine precedenti
        BufferedReader reader = null;
        StringBuffer response = new StringBuffer();
        InputStreamReader input2 = null;
        try {
            input2 = new InputStreamReader(in);
            reader = new BufferedReader(input2);
            String line = "";
            // si legge riga per riga e le inseriamo nello StringBuffer
            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                    input2.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        // si ritorna la riposta pronta per essere utilizzata
        return response.toString();
    }
}


// Una volta che la si sta per chiudere l'applicazione
// bisogna fermare il polling
@Override
public void cleanConnection() throws Exception {
    if(timer!=null){
        if(task!=null)
            task.cancel();
        timer.cancel();
    }
}
```

MyHttpClient.java

```
    // Analogamente a come abbiamo fatto per ricevere messaggi
    // bisogna fare lo stesso per l'invio dei dati (tutte e due sono
    ↪  richieste HTTP)
    private static class SendPositionTask extends AsyncTask<String, Void,
    ↪  Void> {
    // @param strings: string[0] contiene l'URL a cui fare le richieste HTTP
    //  string[1] contiene il messaggio che si vuole trasmettere
        @Override
        protected Void doInBackground(String... strings) {
            HttpURLConnection urlConnection = null;
            try {
                OutputStream out = null;
                URL url = new URL(strings[0]);
                urlConnection= (HttpURLConnection) url.openConnection();
                urlConnection.setDoOutput(true);
                urlConnection.setChunkedStreamingMode(0);
                urlConnection.setReadTimeout(10000);
                urlConnection.setConnectTimeout(15000);
                // in questo caso è desiderabile avere un minimo di
                ↪  sicurezza non mostrando direttamente ciò che si
                ↪  trasmette
                urlConnection.setRequestMethod("POST");
                // è utile sapere anche che cosa ha risposto il server,
                ↪  ovvero il codice di risposta
                urlConnection.setDoInput(true);
                out = new
                ↪  BufferedOutputStream(urlConnection.getOutputStream());
                BufferedWriter writer = new BufferedWriter(new
                ↪  OutputStreamWriter(out, "UTF-8"));
                // invio del messaggio (stringa json che contiene la
                ↪  posizione) da trasmettere
                writer.write("json="+strings[1]);
                writer.flush();
                writer.close();
                out.close();
                urlConnection.connect();
            }catch (MalformedURLException mal){
                Log.d("*** FROM HTTP CLIENT**",
                ↪  "MalFormed"+mal.getMessage());
            }catch (IOException io){
                Log.d("*** FROM HTTP CLIENT**",  "IO_ " +io.getMessage());
            }finally{
                if(urlConnection != null) urlConnection.disconnect();
            }
            return null;
        }
    }
}
```

Codice per la realizzazione del client MQTT

MyMqttClient.java

```
public class MyMqttClient extends MyClient {

    // client MQTT preso dalla libreria PAHO/Eclipse
    private MqttAndroidClient mqttAndroidClient;
    // topic a cui ci si sottoscrive
    private final String subscriptionTopic;
    // topic in cui si pubblicano i messaggi
    private final String publishTopic;

    Context context;

    // l'invio della posizione è semplicemente una pubblicazione
    @Override
    public void sendPosition(String position) {
        publishMessage(position);
    }

    public void addToHistory(String message) {
        Log.d("*** from MQTT ***", message);
        Toast.makeText(context, message, Toast.LENGTH_LONG).show();
    }

    @Override
    public void cleanConnection() throws MqttException {
        mqttAndroidClient.disconnect(context, new IMqttActionListener() {
            @Override
            public void onSuccess(IMqttToken asyncActionToken) {
                Log.d("*** FROM MQTT ***", "SUCCESS ON DISCONNECTING");
            }
            @Override
            public void onFailure(IMqttToken asyncActionToken, Throwable
             ↪  exception) {
              Log.d("*** FROM MQTT ***", "FAILED ON DISCONNECTING - "+
               ↪  exception.getMessage());
            }
        });
    }
```

MyMqttClient.java

```
public MyMqttClient(final String client_id, final String host, int port,
↪  GoogleMap mGoogleMap) {
    super(client_id, host, port, mGoogleMap);
    // quando ci si sottoscrive ad un topic/# si riceve tutti i messaggi
    // sotto topic/
    subscriptionTopic = "test_posizione/#";
    // i messaggi invece vengono pubblicati sotto i propri topic
    ↪  test_posizione/client_id
    publishTopic = "test_posizione/"+client_id;
    // context torna utile per interagire con l'activity e mostrare dei
    ↪  toast
    this.context=MainActivity.main.getBaseContext();

    mqttAndroidClient = new MqttAndroidClient(context, "tcp://" + host + ":"
    ↪  + port, client_id);
    try {
        MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
        // si prova la riconnessione
        mqttConnectOptions.setAutomaticReconnect(true);
        // non si pulisce la sessione una volta chiusa la connessione
        // ovvero siamo interessati ad essere sottoscritti ancora
        // dopo la disconessione
        mqttConnectOptions.setCleanSession(false);

        // per gestire come reagire quando ci si connette è necessario
        // utilizzare una callBack
        mqttAndroidClient.setCallback(new MqttCallbackExtended() {
            @Override
            public void connectComplete(boolean reconnect, String serverURI)
            ↪  {
                if(reconnect){
                    addToHistory("Reconnected to : " + serverURI);
                    subscribeToTopic();
                }else{
                    addToHistory("Connected to: " + serverURI);
                }
            }
            // quando la connessione viene persa si mostra semplicemente un
            ↪  messaggio
            @Override
            public void connectionLost(Throwable cause) {
                addToHistory("The Connection was lost.");
                if(cause!=null)
                    Log.d("connection lost", cause.getMessage());
            }
            @Override
            public void deliveryComplete(IMqttDeliveryToken token) {
            }
```

MyMqttClient.java

```
    @Override
    public void messageArrived(String topic, MqttMessage message) throws
    ↪  Exception {
        // dal momento che ci si sottoscrive solo ad un topic, il parametro
        ↪  topic è inutile
        String str_json = new String(message.getPayload());
        try {
            JSONObject json = new JSONObject(str_json);
            String r_client_id = json.getString("client_id");
            if(r_client_id.trim().equals(""))
                throw new JSONException("Attribute client_id not found");
            if(!r_client_id.equals(client_id)){
                // dato che gli altri client sono dei marker
                // è necessario modificare leggermente la posizione
                // perché il centro del marker parte dalla parte sinistra
                ↪  superiore
                double lat = json.getDouble("latitude")-0.00000534;
                double lon = json.getDouble("longitude")+0.0000008;
                // addToHistory("Updating map for " +r_client_id+ " with
                ↪  lat:" +lat + " and long:" + lon);
                updateMap(r_client_id, new LatLng(lat,lon));
            }
        }catch(JSONException e){
            addToHistory("received not a valid json :" +str_json);
        }

    }

});
// ci si prova a connettere
Toast.makeText(context,"Connecting to "+host, Toast.LENGTH_SHORT).show();
mqttAndroidClient.connect(mqttConnectOptions, context, new
↪  IMqttActionListener() {

    @Override
    public void onFailure(IMqttToken asyncActionToken, Throwable exception)
    ↪  {
        Log.d("*** FROM MQTT CLIENT***","exception:"+
        ↪  exception.getMessage());
        addToHistory("Failed to connect to: " + host);
    }
```

MyMqttClient.java

```
            @Override
            public void onSuccess(IMqttToken asyncActionToken) {

                DisconnectedBufferOptions dscntedBufferOptions = new
                ↪  DisconnectedBufferOptions();
                dscntedBufferOptions.setBufferEnabled(true);
                dscntedBufferOptions.setBufferSize(100);
                // dato che l'invio dei messaggi sono tanti non si dovrebbe
                ↪  avere un buffer persistente
                dscntedBufferOptions.setPersistBuffer(false);
                // non siamo interessati ai messaggi vecchi
                dscntedBufferOptions.setDeleteOldestMessages(false);
                mqttAndroidClient.setBufferOpts(dscntedBufferOptions);
                // ci si sottoscrive al topic
                subscribeToTopic();
            }
        });
    }catch (MqttException ex){
        Log.d("Exception:", ex.getMessage());
    }catch (Exception e){
        Log.d("Exception2",e.getMessage());
    }catch (Error e3){
        Log.d("Exception4", e3.getMessage());
    }catch (Throwable e2){
        Log.d("Exception3", e2.getMessage());
      }

}
public void publishMessage(String publishMessage){
    try {
        // viene instanziato un oggetto MqttMessage in modo da pubblicarlo
        // con semplicità
        MqttMessage message = new MqttMessage();
        // il payload viene caricato (messaggio con la posizione)
        message.setPayload(publishMessage.getBytes());
        // si imposta il flag retained come true
        // dato che potrebbe essere utile che il broker inoltri
        // l'ultima posizione del client anche ai nuovo arrivati
        message.setRetained(true);
        mqttAndroidClient.publish(publishTopic, message);
   } catch (MqttException e) {
        System.err.println("Error Publishing: " + e.getMessage());
        e.printStackTrace();
    }
}
```

MyMqttClient.java

```
    public void subscribeToTopic(){
        try {
            // la qualità del servizio viene scelta 0
            // dato che è inutile essere sicuri della corretta consegna del
            ↪ messaggio dal momento che la frequenza di invio è alta
            mqttAndroidClient.subscribe(subscriptionTopic, 0, context, new
            ↪ IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    addToHistory("Subscribed!");
                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable
                ↪ exception) {
                    addToHistory("Failed to subscribe");
                }
            });
        } catch (MqttException ex){
            System.err.println("Exception while subscribing");
            ex.printStackTrace();
        }
    }
}
```

Il codice completo dell'applicazione è disponibile nel seguente repository su github: `https://github.com/StevenSalazarM/Performance_comparison_http_mqtt`

Inoltre, è disponibile anche il codice per la realizzazione del server HTTP: `https://github.com/StevenSalazarM/Example_WebServer_SpringBoot`

Riferimenti

[1] Dispositivi IoT connessi a Internet dal 2015 al 2025.
https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide

[2] Guide to IOT Networking and Messaging Protocols.
http://www.steves-internet-guide.com/iot-messaging-protocols/

[3] ISO/OSI model.
https://gendersec.tacticaltech.org/wiki/index.php/Networking_concepts

[4] Standard HTTP 1.1 Overall operation.
https://tools.ietf.org/html/rfc2616

[5] Restful web services.
https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

[6] MQTT v3.1.1 standard.
http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[7] MQTT official site.
http://mqtt.org/faq

[8] History about MQ Telemetry Transport from Wikipedia.
https://en.wikipedia.org/wiki/MQTT

[9] A background history, MQTT movitation.
https://github.com/mqtt/mqtt.github.io/wiki/history

[10] MQTT official site.
http://mqtt.org/faq

[11] Broker MQTT.
https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment

[12] Request/Response vs Publish/Subscribe.
https://blog.opto22.com/optoblog/request-response-vs-pub-sub-part-1

[13] Message Attributes.
https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe

[14] MQTT Topics & Best Practices.
https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices

[15] Quality of Service 0, 1 and 2.
https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels

[16] Retained Messages.
https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages

[17] Android Developer.
https://developer.android.com/guide/

[18] GoogleMaps Tutorial.
https://github.com/priyankapakhale/GoogleMapsNearbyPlacesDemo

[19] Guide about how to use the MQTT calls through Eclipse/Paho Library.
https://www.eclipse.org/paho/clients/java/

[20] HttpURLConnection's guide from Android Developer.
https://developer.android.com/reference/java/net/HttpURLConnection

[21] Spring Framework.
https://www.quora.com/What-is-The-Spring-Framework/answer/Manu-Mishra-5

[22] Building a RESTful Web Service with Spring Boot Actuator.
https://spring.io/guides/gs/actuator-service/

[23] Introduction to Redis.
https://redis.io/topics/introduction

[24] Accessing Data Reactively with Redis.
https://spring.io/guides/gs/spring-data-reactive-redis/

[25] Mosquitto Broker MQTT.
https://mosquitto.org/

[26] An example about how Mosquitto works.
https://blog.mi.hdm-stuttgart.de/index.php/2017/08/31/iot-with-the-raspberry-pi-final-application-part-3/

[27] Docker.
https://en.wikipedia.org/wiki/Docker_(software)

[28] Spring Boot with Docker.
https://spring.io/guides/gs/spring-boot-docker/

[29] Things Mobile, operatore per dispositivi IoT.
https://www.thingsmobile.com/it/business/piani