

**Università degli Studi di Verona**

---

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica

**Sincronizzazione dei dati tramite protocolli  
HTTP e MQTT**

Candidato:

**Francesco Carnielli**

Matricola VR087968

Relatore:

**Prof. Damiano Carra**



## **Sommario**

Scopo del lavoro di questa tesi è individuare ed analizzare i protocolli di rete in grado di mantenere i dati sincronizzati tra client e server. A seguito di un evento, gli interlocutori devono essere avvisati in tempo utile, ottimizzando le risorse disponibili ed aggiornando le eventuali risorse.

Nel dettaglio si analizzeranno i protocolli HTTP e MQTT, la loro evoluzione, progettazione e gli scenari d'uso. Verranno presentati due applicativi sviluppati presso Supermercato24, i requisiti iniziali, la loro risoluzione e le scelte tecniche intraprese.

Infine si valuteranno, tramite benchmark, la bontà di questi protocolli con i loro pro e contro, illustrando possibili conseguenze future.

# **Ringraziamenti**

Il lavoro di tesi svolto e presentato in questo elaborato non avrebbe mai potuto essere portato avanti e finito senza il supporto di tutte quelle persone che hanno dedicato parte del loro tempo e delle loro energie per aiutarmi e per ascoltarmi.

In primo luogo vorrei ringraziare tutti i ragazzi di Supermercato24, in particolar modo Marco Risi, che in questi due anni di lavoro diurno e notturno l'hanno resa l'azienda che è oggi.

Ringrazio di cuore tutti gli amici della mia compagnia per avermi costantemente spronato e sostenuto.

Ringrazio infine la mia famiglia per il sostegno fornito in questi anni.



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Background e riferimenti</b>	<b>4</b>
2.1	Paradigmi di comunicazione . . . . .	5
2.1.1	Sincrono . . . . .	5
2.1.2	Asincrono . . . . .	6
2.2	HTTP . . . . .	6
2.2.1	WebSocket . . . . .	8
2.3	MQTT . . . . .	9
2.3.1	QoS . . . . .	11
2.4	Sicurezza . . . . .	11
<b>3</b>	<b>Soluzione adottata</b>	<b>15</b>
3.1	Supermercato24 . . . . .	15
3.1.1	Panoramica tecnica . . . . .	17
3.1.2	Struttura del Sistema . . . . .	17
3.1.3	Struttura del Database . . . . .	19
3.1.4	Struttura dell'Architettura . . . . .	22
3.2	Socksberry . . . . .	22
3.2.1	Analisi iniziale . . . . .	23
3.2.2	Implementazione . . . . .	23
3.2.3	Difficoltà affrontate . . . . .	25
3.2.4	Conclusioni . . . . .	26
3.3	SmIoT . . . . .	26
3.3.1	Analisi iniziale . . . . .	26

3.3.2	Implementazione . . . . .	27
3.3.3	Difficoltà affrontate . . . . .	29
3.3.4	Conclusioni . . . . .	29
3.4	Analisi Competitor . . . . .	30
<b>4</b>	<b>Valutazione sperimentale</b>	<b>32</b>
4.1	Confronto quantitativo . . . . .	32
4.1.1	Creazione della connessione . . . . .	34
4.1.2	Mantenimento della connessione . . . . .	35
4.1.3	Ricezione dati . . . . .	35
4.2	Analisi pacchetti . . . . .	37
4.2.1	Creazione della connessione . . . . .	37
4.2.2	Sottoscrizione al topic . . . . .	38
4.2.3	Mantenimento della connessione . . . . .	38
4.2.4	Ricezione dati . . . . .	38
4.2.5	Chiusura della connessione . . . . .	39
<b>5</b>	<b>Conclusioni</b>	<b>41</b>
<b>A</b>	<b>Sorgenti</b>	<b>44</b>
A.1	Socksberry Controller . . . . .	44
A.2	Socksberry View . . . . .	45
A.3	SmIoT Controller . . . . .	47
<b>B</b>	<b>Test</b>	<b>53</b>
B.1	Grafici risultati . . . . .	53
B.1.1	Creazione della connessione . . . . .	53
B.1.2	Ricezione dati . . . . .	55
B.2	Dettaglio dei pacchetti . . . . .	57
B.2.1	Sottoscrizione al topic . . . . .	57
B.2.2	Ricezione dati . . . . .	58
<b>Bibliografia</b>		<b>60</b>



# Capitolo 1

## Introduzione

INTERNET e le tecnologie informatiche digitali più generali, hanno permesso un'ampia base di cambiamenti nell'organizzazione delle attività economiche così profonde da giustificare il titolo di rivoluzione, definita sotto l'aspetto tecnologico e socio-culturale. L'informazione può essere più facilmente collezionata, salvata, processata, resa disponibile, comunicata ed utilizzata. La riduzione del costo di comunicazione comporterà un aumento del traffico, maggior accesso alle informazioni, autonomia personale nella scelta delle decisioni ed ultimamente nella dispersione delle attività economiche [1].

Di riflesso, i protocolli e le strutture con cui queste interconnessioni vengono stabilite stanno subendo un'incessante evoluzione, mantenendo la rete accessibile a tutti come mezzo libero di informazione e comunicazione. La costante necessità di avere tutte le informazioni immediatamente disponibili, tempestivamente aggiornate ed il rapido aumento dei dispositivi connessi, che sono sia consumatori sia produttori di dati, richiede diversi approcci per adeguarsi a queste nuove esigenze. Il traffico dati generato si raddoppia approssimativamente ogni anno. Questo dato rappresenta una crescita estremamente rapida, maggiore di qualsiasi altro servizio di comunicazione. Se la mole di informazioni transitata su Internet continua a raddoppiare ogni anno, potrebbe essere necessaria un'altra forma della *Legge di Moore* [2].

Mantenere i dati sincronizzati attraverso diversi dispositivi connessi è diventato un pattern comune tra tutti i moderni software con funzionalità di rete. Ogni utente si aspetta, come UX (*User Experience*), di aprire la propria casella di posta con il cellulare e vedere le stesse e-mail viste precedentemente su desktop. Alla ricezione di

una nuova e-mail, dovrà ricevere il giusto avviso sui propri dispositivi. In un servizio di messaggistica istantanea, ci si aspetta di ricevere un messaggio il più velocemente possibile e non al prossimo caricamento dell’interfaccia. Modificando un file condiviso, questo deve essere aggiornato necessariamente con le ultime modifiche, per evitare di lavorare su una revisione già obsoleta. Con l’eterogeneità dei dispositivi e la ricerca di standard comuni, questo paradigma può non sembrare così semplice. La crescente mole di dati rappresenta un problema per le strutture sottostanti. Perciò diventa fondamentale trovare il giusto compromesso tra le richieste di design e la stabilità dell’implementazione.

In questo elaborato verranno descritti i principali protocolli nello scambio e sincronizzazione delle informazioni in Internet, la differenza tra loro, alcuni *TestCase* e la loro progettazione in un ambiente dinamico e in costante evoluzione come può essere una *Start-Up*. Più precisamente sono stati analizzati i protocolli HTTP e MQTT. Il primo, *standard de facto*, per la navigazione attraverso pagine web e il secondo progettato per venire incontro alle esigenze dei nuovi dispositivi smart. Questi protocolli si differenziano sia per i principi di realizzazione sia per un differente approccio logico nel loro utilizzo.

Si spiegherà come si è riuscito a mantenere i dati sincronizzati, a seguito di un *trigger*, con diversi dispositivi connessi che condividono la stessa utenza o utilizzano la stessa risorsa. Si cercherà di risolvere il problema delle chiamate di aggiornamento di ciascun *client* e i vari conflitti emersi. Inoltre è stato verificato che un’inadeguata analisi dei requisiti o un approccio non validato da test può provocare difficoltà non trascurabili ai propri utenti [3].

Nei capitoli successivi verrà ampliato quanto brevemente indicato nell’introduzione. Questi sono pertanto così strutturati: nel capitolo 2 verranno descritti i principi e le caratteristiche tecniche di questi protocolli, nel capitolo 3 verranno presentate le soluzioni adottate e i problemi risolti, il capitolo 4 illustrerà una valutazione sperimentale, infine nel capitolo 5 sono riassunte le conclusioni dell’intero lavoro.



# Capitolo 2

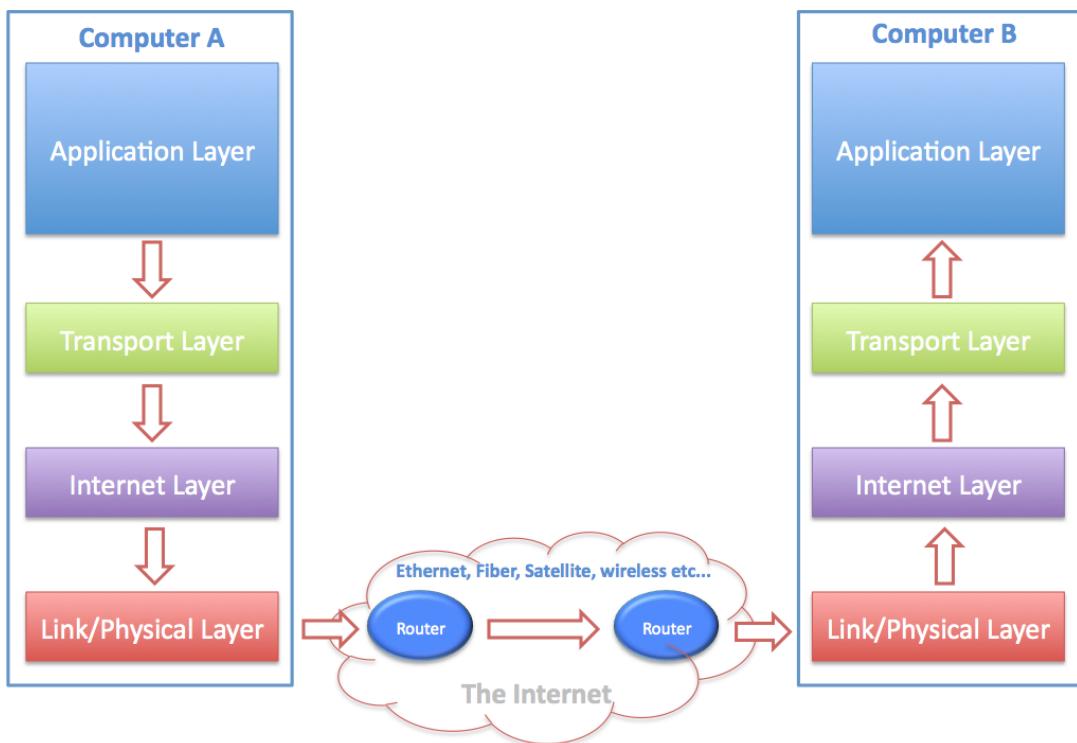
## Background e riferimenti

ON il termine “Internet” ci si riferisce alla rete pubblica formata da dispositivi (nodi) autonomi ed eterogenei distribuiti in tutto il mondo basata su una suite di protocolli standard.

Lo sviluppo di un collegamento tra più nodi è stato motivato originariamente da possibili applicazioni militari, quali collegamenti tra centinaia di università e installazioni governative tramite linee telefoniche. Per questo motivo la capacità di collegare tra loro diverse reti in maniera semplice è stata sin dall'inizio uno dei principali requisiti di progetto. Un altro obiettivo era che la rete potesse sopravvivere alla perdita dell'hardware di *subnet* senza interrompere le conversazioni in corso. Le connessioni dovrebbero rimanere intatte per tutto il tempo in cui il computer sorgente e quello destinazione fossero in funzione, anche se alcuni dei computer o delle linee di trasmissione tra essi fossero andati improvvisamente in avaria. Inoltre era necessaria un'architettura flessibile, poiché erano previste applicazioni con richieste divergenti, che spaziavano dal trasferimento di file alla trasmissione della voce in tempo reale. Questi requisiti hanno portato alla scelta di una rete a comunicazione di pacchetto basata su uno strato senza connessione. Il suo scopo è quello di consentire agli host di mandare pacchetti in qualsiasi rete e farli viaggiare in modo indipendente l'uno dall'altro fino alla destinazione. Potrebbero persino arrivare con un ordine diverso da quello originario e in questo caso (se è richiesta una consegna in sequenza) è compito degli strati superiori ordinare gli strati. Lo strato internet definisce un formato ufficiale per i pacchetti e un protocollo, il cui scopo è quello di consegnare i pacchetti alla destinazione corretta. L'instradamento dei pacchetti è

chiaramente il problema più importante, assieme alla necessità di garantire l'assenza di congestioni. Questa architettura è poi diventata nota con il nome del modello di riferimento TCP/IP, in funzione dei due più importanti protocolli in essa definiti: il TCP (*Transmission Control Protocol*) e IP (*Internet Protocol*) [4].

Su questa architettura, organizzata secondo una logica a livelli sovrapposti, si trovano tutte le applicazioni. Tuttavia, anche nello strato applicativo c'è l'esigenza di protocolli di supporto che permettono ai software di funzionare. Nelle successive sezioni verranno introdotti questi protocolli del livello applicativo: HTTP e MQTT.



**Figura 2.1:** Modello dello stack TCP/IP a layer sovrapposti

## 2.1 Paradigmi di comunicazione

### 2.1.1 Sincrono

Le architetture dei grandi sistemi informativi sono, ancora oggi, nella quasi totalità dei casi, costruite su piattaforme *client/server*. In questo paradigma due host interagiscono tra loro: il primo richiede dati e il secondo li fornisce. Questo tipo di comunicazione è denominato **sincrono** e segue questo flusso:

1. il *client* invia una richiesta

2. il *server* elabora la richiesta e invia la risposta

Durante tutto il periodo di interazione, gli interlocutori risultano impegnati esclusivamente nella transazione e quindi impossibilitati ad eseguire altre operazioni. Se il *server* dovesse risultare impegnato nel rispondere, l'esperienza per il *client* sarebbe imprevista. Le entità che possono interagire in un dato istante sono solo due, il che rende più difficoltosa la realizzazione di applicazioni che richiedono comunicazioni 1:N o N:N. Nonostante i limiti, questo modello ha riscosso un incredibile successo. Ciò è da imputare sia alla semplicità del modello stesso sia alla sua adattabilità a moltissimi ambiti applicativi.

Un'applicazione di consumo delle risorse basata su questo modello, interroga costantemente un servizio per aggiornare i propri valori. Perché un'applicazione del genere risulti utile, deve effettuare una serie di interrogazioni con un'alta frequenza per garantire un aggiornamento dei valori con un ritardo massimo stabilito. Minore è il ritardo massimo accettabile, maggiore è la frequenza delle interrogazioni e di conseguenza lo spreco di risorse (se la risorsa non è cambiata) [5].

### 2.1.2 Asincrono

Un paradigma di comunicazione dove esiste un totale disaccoppiamento tra produttore e consumatore dell'informazione è denominato **asincrono**. I consumatori (*subscriber*) possono esprimere il loro interesse verso un sottoinsieme di risorse ed essere successivamente notificati dell'aggiornamento generato da un produttore (*publisher*). Tale comunicazione asincrona viene identificata come *publish/subscribe*. L'introduzione di questo nuovo modello all'interno di applicazioni esistenti ha permesso di migliorare l'uso delle risorse da parte delle applicazioni stesse, lasciando inalterato il modo di operare degli utenti.

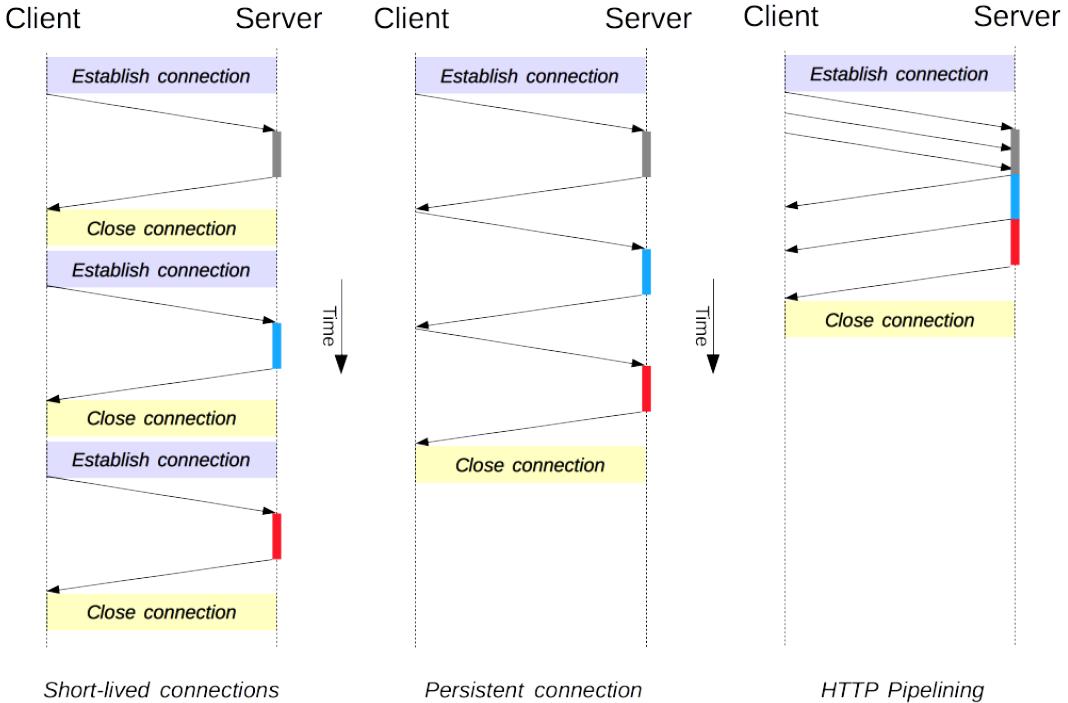
## 2.2 HTTP

Uno dei protocolli del livello applicativo più utilizzati in Internet è HTTP (*HyperText Transfer Protocol*). Si basa sul pattern sincrono richiesta/risposta (*client/server*): il *client* effettua una richiesta e il *server* restituisce una risposta. Vi sono quindi due tipi di messaggi: messaggi di richiesta e messaggi di risposta.

Grazie agli strati sottostanti, HTTP non deve occuparsi di come avvengono le connessioni, ma definire una ben definita sintassi fra i vari utilizzatori. A sua volta, questo protocollo viene utilizzato come base su cui disporre ulteriori strutture e/o architetture, dove SOAP, RCP e RESTful sono alcuni esempi.

Il paradigma richiesta/risposta permette un'unica direzionalità nel flusso dati, definito nello standard HTTP/1.0, nel quale alla ricezione della risposta, la connessione viene conclusa [6]. Se si volesse interrogare successivamente il servizio, sarebbe necessario creare una nuova connessione. Solo nella versione aggiornata dello standard HTTP/1.1, viene data la possibilità di non chiudere immediatamente il collegamento [7]. Queste connessioni persistenti *KeepAlive* possono essere lasciate aperte un numero prefissato di secondi ed utilizzate in seguito per altre richieste. Mantenendo aperta la comunicazione, è possibile inviare richieste multiple senza aspettare la risposta di ognuna, con un meccanismo di pipeline; il *server* dovrà inviare le risposte nello stesso ordine di ricezione [8].

Nella nuova versione dello standard HTTP/2, è permessa la comunicazione preventiva tramite meccanismo di Server Push [9]. Viene data la possibilità di soddisfare più richieste contemporanee che sono necessarie per concludere la richiesta originale. Questo è possibile solamente quando il *server* riesce a stabilire a priori le successive richieste, come possono essere le immagini allegate ad una pagina web.



**Figura 2.2:** Modello delle diverse tipologie di connessioni sincrone HTTP

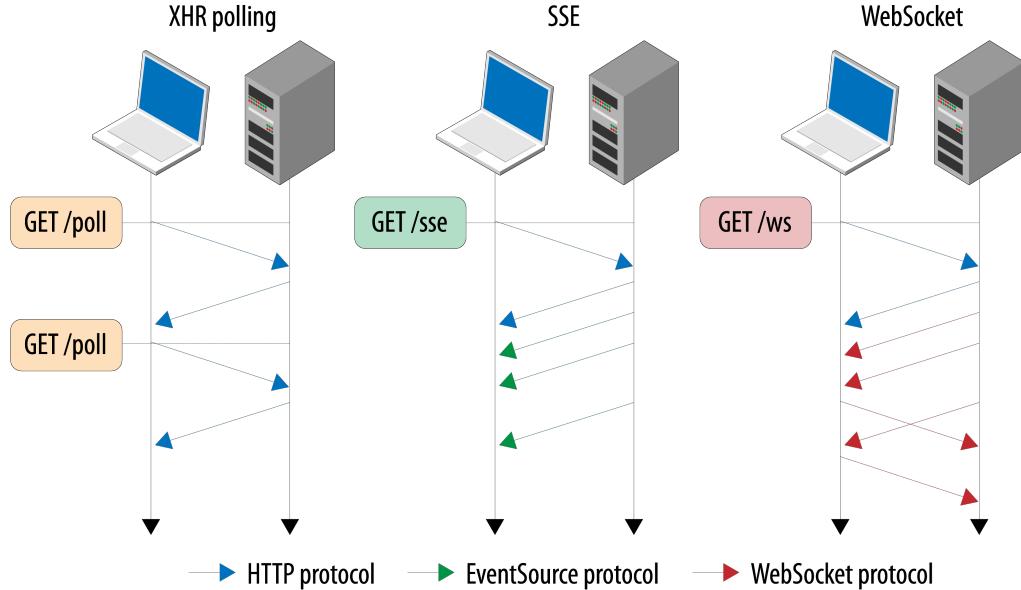
### 2.2.1 WebSocket

Nonostante la possibilità di tenere aperta la connessione tra i due interlocutori, il paradigma sincrono richiesta/risposta rimane invariato e non è mai il *server* che invia informazioni al *client* senza un'esplicita richiesta iniziale. Il *client*, quindi, può accorgersi di un certo cambiamento solamente inviando richieste multiple XMLHttpRequest o <iframe>s in un determinato lasso di tempo con lunghi periodi di polling e aprendo più connessioni [10].

Per venire incontro a questa esigenza è stato introdotto il **WebSocket**, una tecnologia che fornisce una comunicazione bidirezionale *full-duplex* tra i due interlocutori utilizzando una singola connessione TCP [11]. Infatti il **WebSocket** è un protocollo indipendente, la cui unica correlazione con l'HTTP è nel modo in cui fa l'*handshake* durante una richiesta *Upgrade* verso il *server*. La comunicazione viene mantenuta aperta da entrambe le parti e chiusa solo tramite richieste esplicita o problemi di rete. Lo scambio di informazioni si trasforma da sincrono ad asincrono, dove gli interlocutori possono inviare richieste anche se non interrogati.

Di particolare nota sono i SSE (*Server-sent events*). Una tecnologia non ancora supportata da tutti i browser, che permette l'invio di notifiche Push asincrone

da parte del *server* a seguito di una richiesta esplicita iniziale [12]. Il *client* non chiude immediatamente la comunicazione e rimane in attesa senza intervenire delle successive notifiche.



**Figura 2.3:** Diversi approcci del pattern di comunicazione sincrona rispetto asincrona

## 2.3 MQTT

La crescente diffusione di dispositivi smart collegati tra di loro attraverso la rete secondo un meccanismo M2M (*Machine to Machine*), denota un trend definito come IoT (*Internet of Things*) [13].

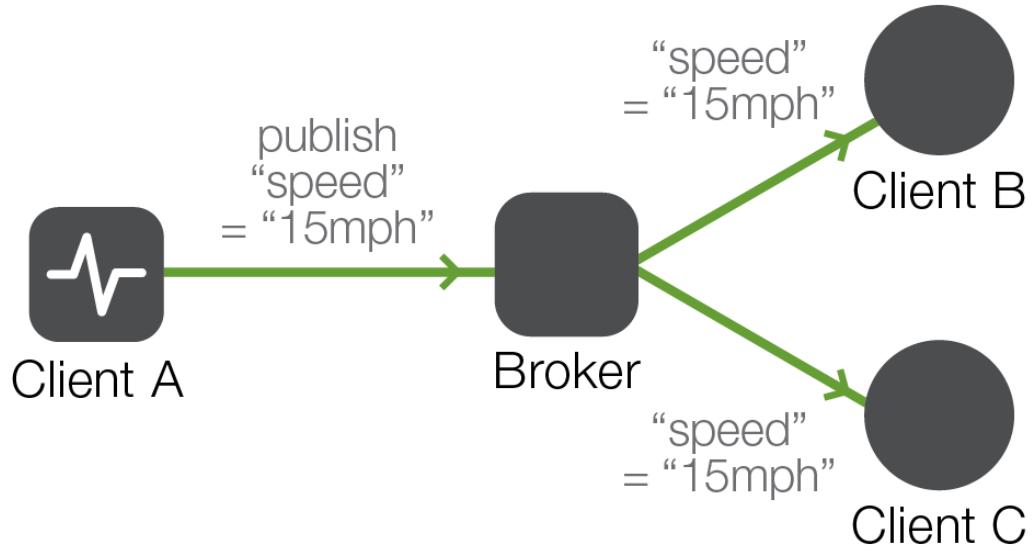
L'IoT rappresenta la più dirompente rivoluzione tecnologica della nostra vita. Con un numero di *device* connessi a Internet stimato tra i 50 e 100 miliardi entro il 2020, stiamo vivendo un cambiamento di paradigma in cui oggetti di uso quotidiano diventano interconnessi ed intelligenti [14].

Per questi dispositivi sono emerse differenti sfide:

- le connessioni devono avere un basso impatto sulle prestazioni del sistema, soprattutto considerando la scarsa potenza computazionale disponibile
- la banda può essere limitata, non sempre disponibile o di scarsa qualità
- le notifiche dovranno essere tempestive e causate da qualche evento

Con queste caratteristiche, è stato definito un nuovo protocollo MQTT (*MQ Telemetry Transport*). Nato nel 1999 da Andy Stanford-Clark di IBM e Arlen Nipper di Arcom, è diventato uno standard pubblico nel Marzo del 2013 sponsorizzato da IBM [15]. Si tratta di un protocollo di messaggistica estremamente semplice e leggero posizionato in cima allo stack TCP/IP. Una variante del protocollo, definita MQTT-SN, è stata sviluppata per funzionare su reti non TCP/IP come *ZigBee* [16, 17]. Si basa sul pattern *publish/subscribe (producer/consumer)*: comunicazione di messaggi tra diversi processi, oggetti o altri agenti.

Il funzionamento logico di una comunicazione si basa sul concetto di comunicazione asincrona tra due o più player, i quali dialogano attraverso un tramite (*broker*). Il *publisher* di un messaggio (*producer*) pubblica il proprio messaggio al *broker*, potendo anche specificare il *topic* del messaggio. I *subscriber* si rivolgono a loro volta al *broker* abbonandosi nella ricezione di messaggi o solo di alcuni *topics*. Il *broker* inoltra ogni messaggio inviato da un *publisher* a tutti i *subscriber* interessati a quel messaggio. La connessione non viene necessariamente chiusa a messaggio ricevuto o inviato ed i partecipanti non devono richiedere nuovi messaggi ad intervalli di tempo, ma sono notificati immediatamente.



**Figura 2.4:** Modello del dispatch di un messaggio da parte del broker a diversi subscriber

### 2.3.1 QoS

MQTT fornisce la possibilità di garantire che il messaggio sia effettivamente stato consegnato attraverso un meccanismo denominato **QoS** (*Quality of Service*). Con l'ipotesi che la banda può essere limitata o inaffidabile, i *client* possono negoziare il giusto livello di qualità del servizio in base alle loro esigenze:

**QoS 0 – at most once** il messaggio non deve essere ammesso dal destinatario o salvato e rinviato dal mittente (stessa garanzia prevista dal protocollo TCP)

**QoS 1 – at least once** viene garantito che il messaggio sia stato consegnato almeno una volta ed è possibile che lo stesso messaggio venga recapitato più volte

**QoS 2 - exactly once** viene garantito che il messaggio verrà consegnato esattamente una volta

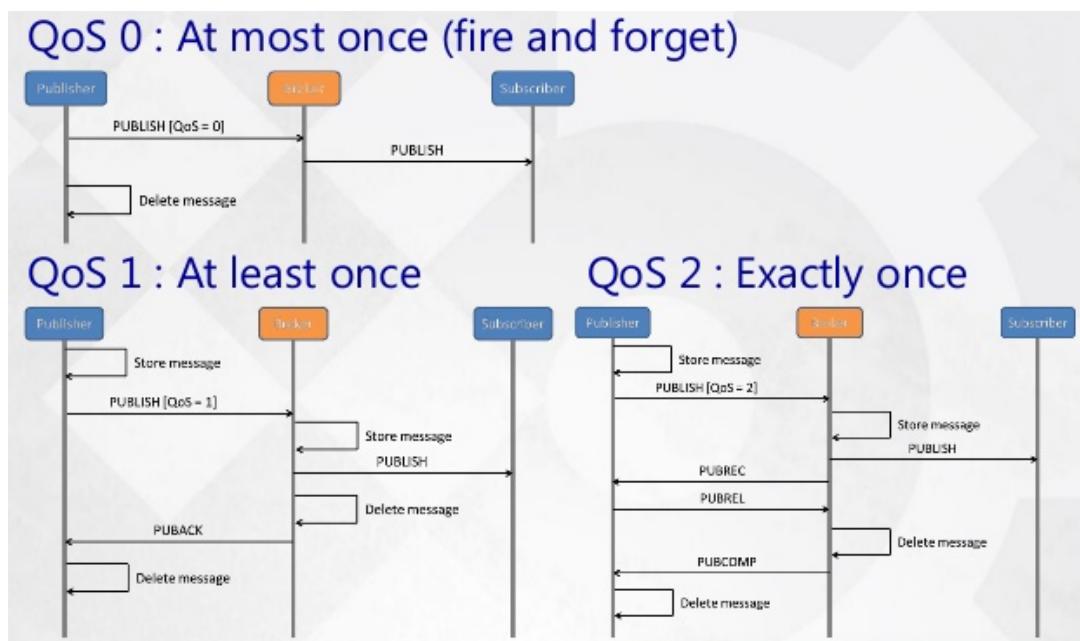


Figura 2.5: I vari livelli di qualità del servizio disponibili tramite MQTT

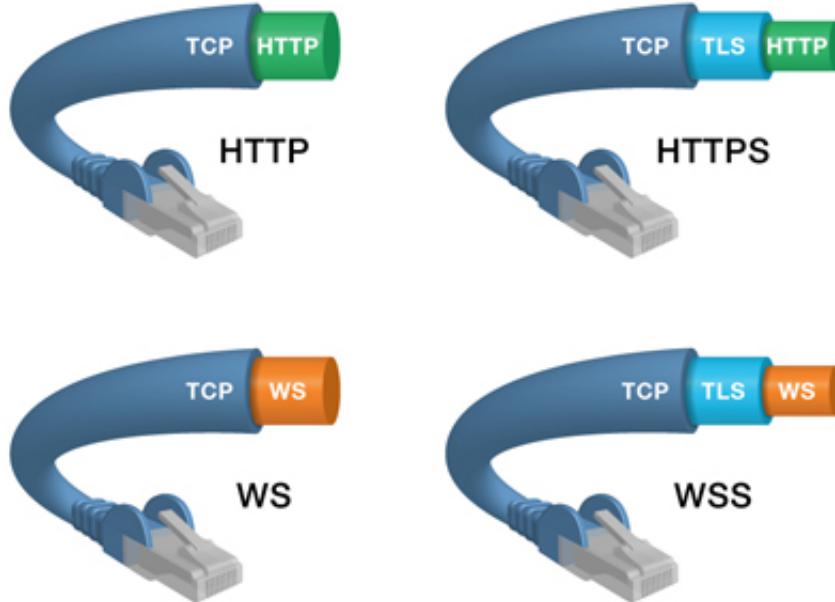
## 2.4 Sicurezza

Entrambi i protocolli sono stati sviluppati al livello applicativo sopra lo stack TCP / IP. Viene garantita la confidenzialità della comunicazione evitando intercettazioni o ma-

nomissioni dei messaggi tramite il protocollo TLS attraverso il layer del trasporto TCP, risultando trasparente per gli strati superiori [18].

I dati necessari per instaurare questa comunicazione protetta (*handshake*) vengono negoziati prima che il primo byte dell'applicativo venga inviato. La volontà di cifrare questi dati deve essere necessariamente specificata all'inizio della connessione, per permettere lo scambio delle chiavi necessarie a garantire la giusta riservatezza, comportando un leggero sovraccarico (*overhead*) computazionale per tutta la durata della comunicazione.

Cifrando i dati dell'applicativo con le chiavi scambiate inizialmente tra i due interlocutori (*Alice* e *Bob*), un terzo personaggio (*Charlie*), difficilmente riuscirebbe a comprendere questa comunicazione.



**Figura 2.6:** Astrazione della connessione tramite TLS attraverso il protocollo TCP

Le porte di default del *server* per accettare questo tipo di connessioni differiscono rispetto a quelle prive di cifratura e sono:

Protocollo	Porta	Sicuro
HTTP	<b>80</b>	<b>X</b>
HTTPS	<b>443</b>	<b>✓</b>
WS	<b>80</b>	<b>X</b>
WSS	<b>443</b>	<b>✓</b>
MQTT	<b>1883</b>	<b>X</b>
MQTTS	<b>8883</b>	<b>✓</b>

**Tabella 2.1:** Elenco dei protocolli con le loro porte e relativa cifratura



# Capitolo 3

## Soluzione adottata

L’OBIETTIVO del lavoro della tesi è quello di confrontare tra loro differenti tecniche e soluzioni di sincronizzazione dati attraverso Internet. I due sistemi proposti sono stati progettati per migliorare la UX delle applicazioni di Supermercato24 attualmente in produzione. Nel mondo e-commerce si vuole ottimizzare l’esperienza d’acquisto tra i dispositivi dell’utente, sia dal lato tecnico sia dal lato del design. Inoltre, per gestire gli ordini in arrivo si vuole aggiornare le metriche qualitative del servizio ad ogni evento.

### 3.1 Supermercato24

Ordini la spesa online e ti arriva a casa in un’ora. Supermercato24 è la *Start-Up* che mette in contatto chi vuole acquistare prodotti online dai supermercati e i “*personal shopper*”, persone che fanno la spesa per conto del cliente e la consegnano a domicilio. Nata a Verona nel settembre 2014 da un’idea di Enrico Pandian, oggi conta 30 dipendenti e più di settecento fattorini dislocati nelle 16 province italiane servite. Utilizzare il servizio di Supermercato24 è piuttosto semplice, basta inserire il proprio codice di avviamento postale sul sito. A quel punto il portale elenca i supermercati convenzionati più vicini. Una volta selezionati i prodotti e inviato l’ordine, un fattorino che lavora con Supermercato24 fa la spesa al posto del cliente e la consegna entro un’ora [19].

Il modello di business si basa su tre fattori: costo di consegna della spesa (4.90 €), accordi con le grandi marche e sovrapprezzo sul prodotto acquistato. Il mercato

del *grocery* in Italia vale circa 180 miliardi, ma è attivo per lo più offline, con un’incidenza potenziale legata all’e-commerce dell’1%. Una fetta da oltre un miliardo e mezzo di euro, che potrebbe far gola a tanti [20].

Supermercato24 è attiva in 16 province e 300 comuni, esegue circa 500 consegne al giorno e serve tramite i suoi “*shopper*” oltre 40mila clienti, facendo leva su più di 30 insegne. In catalogo ha circa 90mila articoli, il tasso di crescita mese su mese è del 15% (a giugno le consegne hanno toccato quota 300mila), ha attualmente 250mila utenti e ha chiuso il 2016 con un fatturato di circa 6 milioni di euro, a fronte dei 750mila del 2015 [21].

Nel 2016 è stato siglato un accordo tra Supermercato24 e Samsung, per portare il portale e-commerce all’interno della linea frighi smart IoT dell’azienda sudcoreana. La soluzione si chiama Samsung Family Hub e unisce domotica e commercio elettronico, ed è stato presentato all’IFA di Berlino a Settembre [22]. Attraverso uno schermo esterno, incorporato nell’anta del frigo, è possibile ordinare la spesa con il proprio account. Nel futuro sarà possibile comprare i prodotti, non appena esauriti, attraverso telecamere e sensori interni.



**Figura 3.1:** Presentazione della linea Samsung Family Hub all’IFA del Settembre 2016

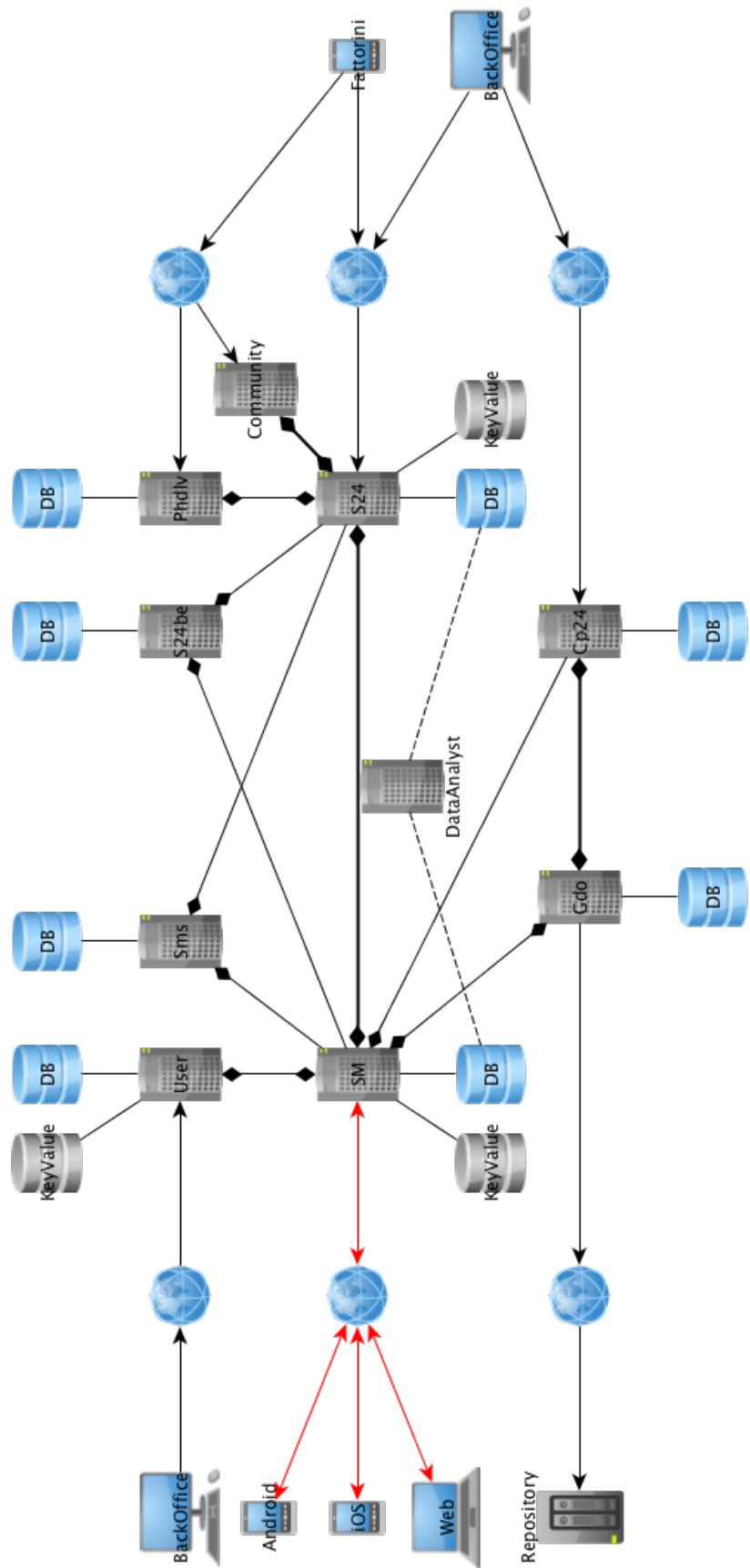
### 3.1.1 Panoramica tecnica

La piattaforma di Supermercato24 è costituita da due Applicativi monolitici scritti in PHP 5.5 con il framework *Laravel*. Otto Microservizi gestiscono le varie integrazioni e namespace. Il Database utilizzato è MySQL 5.6 in *cluster* su due *server* in *High Availability* che si occupa di gestire catalogo/utenze/negozi/ordini. Viene utilizzato il KeyValue Redis 2.8 per gestire: i *token* di sessione degli utenti, i carrelli non ancora confermati e i messaggi asincroni. I Client sono scritti in: Javascript con il framework *Angular* per la parte *browser*, Java per la parte *Android* e Objective-C per la parte *iOS*. È disponibile un ambiente di *staging* per controllare il corretto funzionamento della piattaforma avvalorato da *Unit Test*.

Per il lavoro della tesi è stato utilizzato RaspberryPi 3 come *server* stand-alone e NodeJS 7 come *back-end* che si occupa della gestione delle connessioni dei *socket*. Nelle prossime sezioni entrerò nel dettaglio dell'architettura appena descritta, evidenziando i problemi emersi e le soluzioni adottate.

### 3.1.2 Struttura del Sistema

Supermercato24 si compone di diversi componenti suddivisi con scopi specifici, i quali dialogano, se previsto, tramite API KEY o WebHook. Il progetto *Sm* si occupa di fornire diverse API a tutti i *client* per sfogliare il catalogo dell'e-commerce. La comunicazione è fatta tramite chiamate HTTP e nel lavoro di tesi si cercherà di sincronizzarla in tempo reale (evidenziata in rosso nella figura 3.2). I clienti possono usufruire del servizio con il proprio browser o utilizzare la relativa app e sono accreditati dal progetto *User* tramite *token* di sessione salvato nel KeyValue. I prodotti sono catalogati per mezzo del progetto *Cp24*, il quale fornisce un backoffice agli operatori e lavora in maniera sinergica con il progetto *Gdo* per aggiornare i prezzi raccolti da repository esterni. Ad ogni ordine creato c'è uno scambio di informazioni verso il progetto *S24*, il quale fa transitare l'ordine al fattorino migliore e offre un backoffice di controllo ai relativi responsabili. I fattorini si abilitano nella piattaforma con il progetto *Community* e ricevono comunicazioni tramite il progetto *Sms*. Ogni fattorino può chiamare il proprio cliente tramite il progetto *Phdlv* che si frappone come proxy telefonico tra lo shopper e il cliente. Infine il progetto *DataAnalyst* confronta i Database dei vari progetti per estrarre metriche e analisi di business.



**Figura 3.2:** Modello della struttura di Supermercato24

### 3.1.3 Struttura del Database

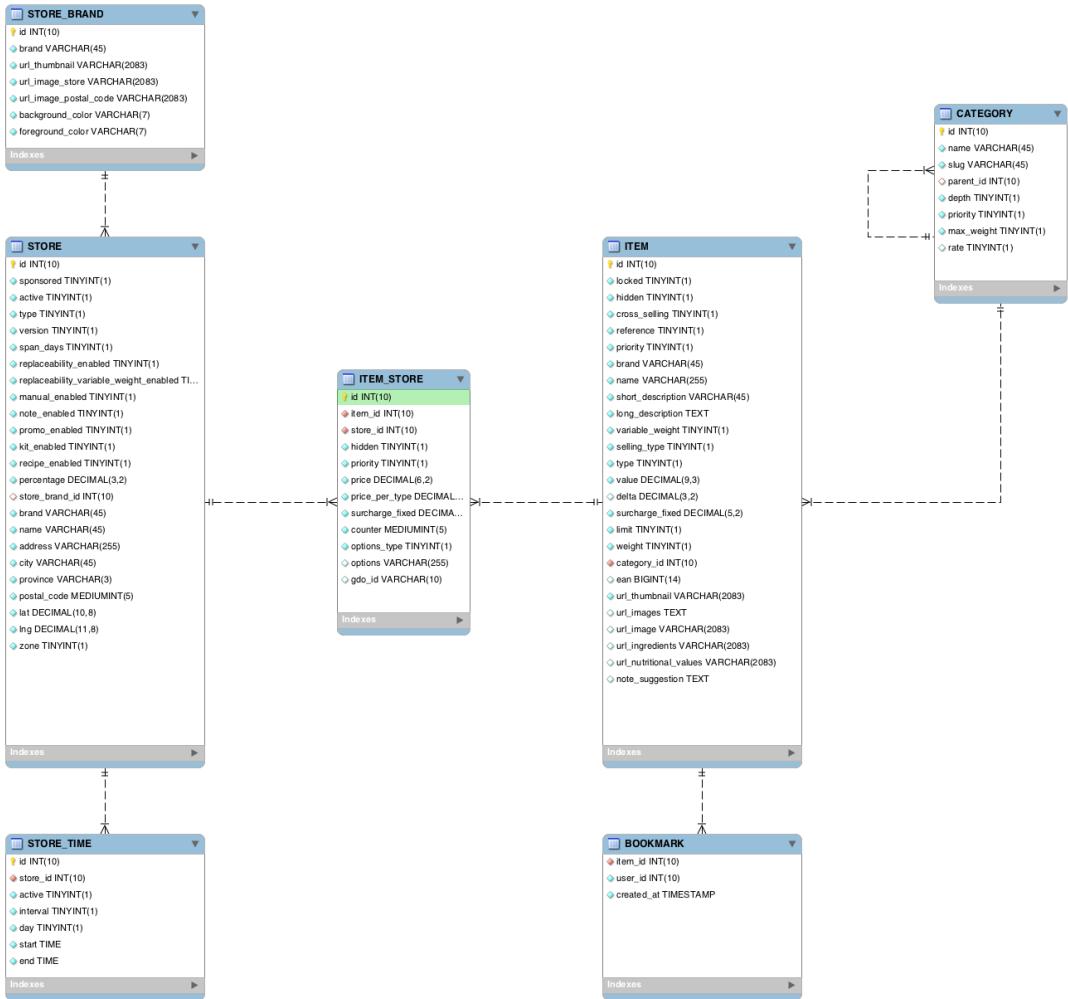
I prodotti all'interno dell'e-commerce sono organizzati secondo una relazione binaria N:N tra la tabella *Item* e la tabella *Store*. La tabella *Item* contiene tutti i dettagli di un prodotto quali nome, marchio, EAN, immagini e grammatura. Ogni prodotto può essere salvato nella tabella *Bookmark* come preferito da parte del cliente. Tutti i prodotti sono catalogati attraverso una categoria di terzo livello univoca (foglia) tramite una relazione N:1. La tabella *Category* contiene un'alberatura ricorsiva di elementi collegati tra di loro, per definire una struttura a livelli sovrapposti.

La tabella *Store* contiene tutti i dettagli dei negozi disponibili nella piattaforma. Ogni negozio ha a disposizione i propri orari di apertura nella tabella *StoreTime* tramite una relazione 1:N e sono catalogati per insegna nella tabella *StoreBrand* tramite una relazione N:1.

Infine la tabella *ItemStore* contiene i prezzi dei prodotti presenti in determinati supermercati, dove alcuni sono aggiornati quotidianamente tramite accordi commerciali con la GDO.

Quando un cliente effettua un ordine, lo stato di ogni prodotto viene salvato nella tabella *OrderDetail* in relazione N:1 con la tabella *Order*.

La maggior parte delle tabelle del Database utilizzano il motore InnoDB per usufruire delle proprietà ACID tramite *Foreign Key*. La tipologia di *locking*, a livello di riga, permette di aggiornare (in scrittura) il catalogo giornalmente senza causare disservizi ai clienti che stanno navigando sul sito (in lettura).



**Figura 3.3:** Modello E-R della struttura del Database

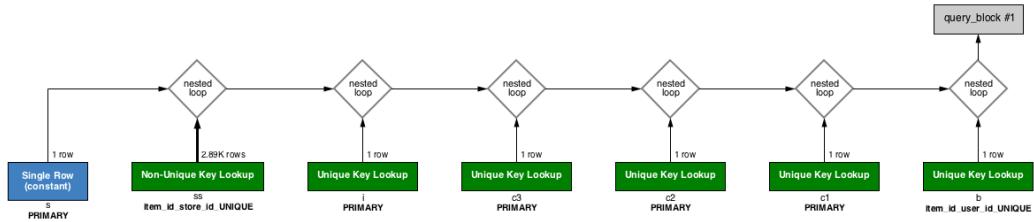
Dato l'identificativo del negozio, sono necessarie sei *Join* per ottenere tutti i dettagli di un prodotto, combinando le tuple delle relazioni appena definite e controllando tutti i rispettivi *flag* di attivazione e ordinamento.

```

SELECT *
FROM ITEM i
    JOIN ITEM_STORE ss ON ss.item_id = i.id AND ss.hidden = 0
    JOIN STORE s ON s.id = ss.store_id AND s.active = 1
    JOIN CATEGORY c3 ON c3.id = i.category_id AND c3.depth = 2
    JOIN CATEGORY c2 ON c2.id = c3.parent_id AND c2.depth = 1
    JOIN CATEGORY c1 ON c1.id = c2.parent_id AND c1.depth = 0
    LEFT JOIN BOOKMARK b ON b.item_id = i.id AND b.user_id = 32600
WHERE
    i.hidden = 0 AND ss.store_id = 5030;

```

**Listing 3.1:** Query originaria per ottenere le informazioni di un prodotto



**Figura 3.4:** Execution Plan della query originaria per ottenere le informazioni di un prodotto

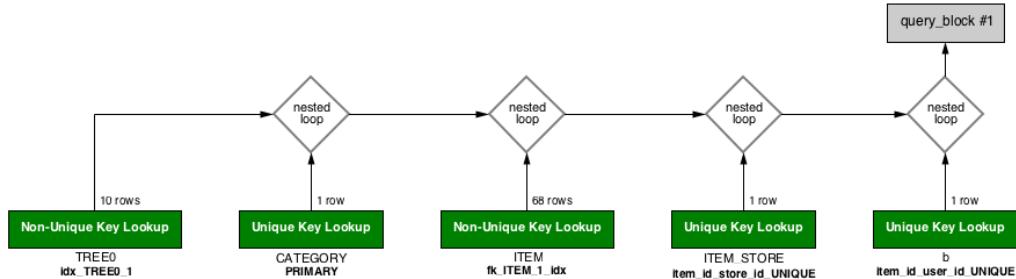
Queste combinazioni sono state successivamente raccolte in una *View* e raggruppate in una tabella temporanea volatile MEMORY compilata giornalmente ordinata con i rispettivi *flag* di attivazione.

```

SELECT *
FROM PRODUCT p
LEFT JOIN BOOKMARK b ON b.item_id = p.item_id AND b.user_id = 32600
WHERE
    store_id = 5030;

```

**Listing 3.2:** Query ottimizzata per ottenere le informazioni di un prodotto



**Figura 3.5:** Execution Plan della query ottimizzata per ottenere le informazioni di un prodotto

Come si evince dal confronto delle due immagini, la normalizzazione delle tabelle ha ridotto le relazioni necessarie per ottenere una o più risorse dal sistema. Con il progetto della sezione 3.3 si cercherà di rendere questo modello accessibile in tempo reale. A seguito di un determinato evento, come l'aggiunta di un preferito, la creazione di un ordine o l'aggiornamento del catalogo, sarà compito dell'applicativo propagare tali dati al giusto utente.

### 3.1.4 Struttura dell'Architettura

Tutte le risorse sono organizzate secondo l'architettura RESTful, quindi consumabili con il metodo HTTP GET [23].

Il carrello è una risorsa salvata temporaneamente tramite KeyValue, secondo l'associazione N:N Negozio:Cliente con la struttura HASH. Questa associazione fornisce la chiave necessaria per accedere al carrello, il cui valore è la quantità del prodotto scelta dal cliente.

```
redis> HSET store_id_user_id item_store_id 1.2
(integer) 1
redis> HGET store_id_user_id item_store_id
"1.2"
redis> HINCRBYFLOAT store_id_user_id item_store_id 0.4
"1.6"
redis> HINCRBYFLOAT store_id_user_id item_store_id -0.6
"1"
redis> HDEL store_id_user_id item_store_id
(integer) 1
redis> HGET store_id_user_id item_store_id
(nil)
```

**Listing 3.3:** Elenco dell'operazioni effettuate nel carrello da parte del cliente

Ogni modifica della quantità di un prodotto avviene tramite metodo HTTP PUT, mentre l'eliminazione tramite metodo HTTP DELETE. Ogni azione ha una complessità temporale  $O(1)$ .

Quando il cliente ha completato il proprio carrello, risulterà possibile effettuare l'ordine tramite metodo HTTP POST. Salvati l'indirizzo di spedizione, l'orario di consegna e il metodo di pagamento appropriato questi dati vengono recapitati al fattorino migliore. Con il progetto della sezione 3.2 si cercherà di notificare in tempo reale la creazione di un ordine tramite *dashboard*. Queste informazioni serviranno per avvertire gli operatori e misurare l'efficienza del servizio.

## 3.2 Socksberry

Nella precedente sezione 3.1.4 è emersa la necessità di monitorare internamente il numero di ordini creati, consegnati, la loro distribuzione geografica e il fatturato totale in tempo reale. A questi requisiti si sono aggiunti le informazioni riguardanti i clienti e un indice qualitativo del nostro servizio di consegna BOR (*Bad Order*

*Rate*). Questo progetto è stato rinominato “Socksberry” dall'unione di “Websocket” e “RaspberryPi”.

### 3.2.1 Analisi iniziale

Le metriche KPI (*Key Performance Indicator*) da analizzare e mostrare sono:

- totale transato quotidiano
- conteggio degli ordini quotidiani
- conteggio degli ordini consegnati quotidiani
- conteggio degli ordini eliminati per inefficienza quotidiani
- conteggio degli accessi e registrazioni quotidiani
- latitudine e longitudine dell'indirizzo del cliente per ogni consegna

Per ogni voce occorre poi estrarre:

- valore record assoluto
- valore complessivo
- differenza percentuale rispetto al mese precedente

Tutti i valori dovranno essere aggiornati e calcolati in tempo reale senza appesantire il carico del *server*. L'output verrà mostrato su un monitor, tv o comunque dispositivi interni non smart tramite un browser.

### 3.2.2 Implementazione

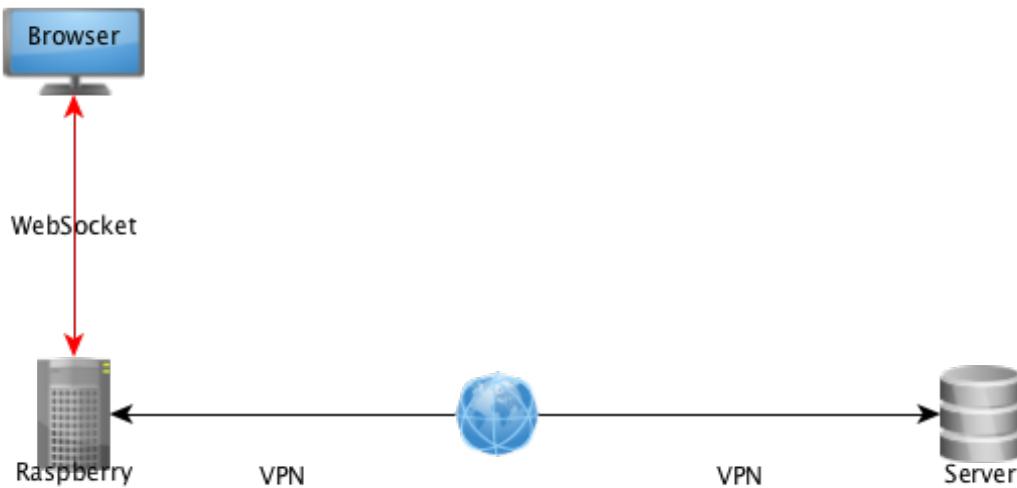
Per poter estrarre tutti i dati e mantenerli congruenti si è preferito dividere il flusso in due fasi:

1. fase di bootstrap, effettuata tramite query al Database
2. fase di update, ad ogni ordine, del modello precedente tramite KeyValue

Successivamente, si è preferito non esporre questo servizio sui *server* di produzione per motivi di scalabilità e sicurezza. Non è prevista nessuna policy di autenticazione, poiché viene garantita dalla comunicazione attraverso VPN con chiave RSA.

Si è preferito utilizzare un RaspberryPi come elaboratore dati (“*controller*”), il quale si connette al *server* tramite una connessione VPN incapsulando le varie comunicazioni TCP verso il Database e verso il KeyValue instaurate da un applicativo NodeJS. Il software sviluppato, appena avviato, si connette al Database per ottenere i dati necessari (fase di “bootstrap”) e successivamente instaura una comunicazione *publish/subscribe* (fase di “update”) per ricevere gli aggiornamenti del modello. Il programma è riportato in appendice A.1.

RaspberryPi 3 è un *single-board computer* economico e portatile, costruito attorno ad un SoC (*System-on-a-chip*). Ospita un sistema operativo basato su kernel Linux e connesso alla rete aziendale tramite Wi-Fi. È stato installato: la piattaforma NodeJS per l'applicativo, Chromium come browser eseguibile a schermo intero da terminale e OpenVPN come terminatore della VPN.



**Figura 3.6:** Modello delle connessioni del progetto Socksberry

Per mostrare questi dati è stato sviluppato un applicativo Javascript (“*view*”) collegato tramite HTTP WebSocket come descritto nella sezione 2.2.1. La connessione HTTP asincrona permette di ricevere i dati solo quando disponibili, evitando

aggiornamenti in un intervallo di tempo programmato. Il programma è riportato in appendice A.2.

Si vengono a creare così due *broker* distinti: il *KeyValue* che pubblica gli eventi del *server* al *controller* tramite connessione asincrona TCP e il *controller* che pubblica gli eventi del *KeyValue* alla *view* tramite connessione asincrona HTTP.



**Figura 3.7:** Foto del progetto Socksberry completato e operativo

### 3.2.3 Difficoltà affrontate

Inizialmente la fase di bootstrap dell'applicativo era associata alla prima connessione da parte di un *client* e isolata da altri interlocutori. Quindi per ogni *client* connesso veniva effettuata una query iniziale e una *subscription* dei *topics* all'interno dello *scope* del socket. Si è poi reso necessario condividere questo modello tra tutti i socket, migliorando le performance e diminuendo il carico complessivo.

La comunicazione asincrona creata tramite WebSocket viene utilizzata prevalentemente in un'unica direzione: dal *controller* verso la *view*. È stata proposta l'adozione dei SSE, ma le notifiche non venivano inviate in tempo utile derivato dal delay progettuale.

La connessione da parte di un *client* tramite WebSocket potrebbe venire effettuata mentre il *controller* è in ancora in fase di bootstrap non ricevendo alcun

dato e rimanendo inutilmente in attesa. Abbiamo previsto, quindi, una fase “init” anche per la *view*, aspettando la fase utile di scambio dati e attivando la grafica della *dashboard* asincronamente.

### 3.2.4 Conclusioni

Si è scelto di non utilizzare il protocollo MQTT tra il *controller* e la *view* perché:

- la batteria non è un fattore critico
- la banda non è un fattore critico
- la potenza computazionale non era condivisa con risorse critiche
- trattandosi di uno strumento di controllo interno non erano necessarie ottimizzazioni

## 3.3 SmIoT

Nella precedente sezione 3.1.3 è emersa la necessità di notificare a tutti i dispositivi connessi ogni azione intrapresa da un cliente in tempo reale. Lo scopo di questo lavoro è valutare la migliore implementazione disponibile e compatibile con lo stack attuale. Questo progetto è stato rinominato “SmIoT” dall’unione di “Supermercato” e “IoT”.

### 3.3.1 Analisi iniziale

I requisiti iniziali sono:

- sincronizzazione inserimento/modifica/eliminazione di un prodotto nel carrello
- notificare inserimento/eliminazione di un prodotto preferito
- notificare la transazione dello stato dell’ordine
- sincronizzare gli slot orari di ogni supermercato

I possibili sviluppi dall’implementazione potrebbero essere:

- creazione di una chat per l’assistenza clienti
- utilizzare notifiche push a fini di marketing
- fornire informazioni dettagliate sull’arrivo del fattorino
- analizzare i clienti connessi in tempo reale

Tutti i valori sono associati per utente e devono essere isolati ai suoi dispositivi. L’utilizzo del servizio da parte dei dispositivi mobile, condiviso tra app e sito responsive, si attesta sul 50%, mentre gli ordini fatti solo da app sono circa il 30%. Con questi dati si è scelto di non sottovalutare le difficoltà della UX dei *client* non desktop.

### 3.3.2 Implementazione

Attualmente tutti i requisiti sono gestiti tramite chiamate RESTful descritte nella sezione 3.1.4. Il cambio di paradigma, verso una tipologia asincrona, dovrebbe migliorare l’esperienza generale del cliente aumentando la soddisfazione.

Grazie all’esperienza maturata con il precedente progetto della sezione 3.2, inizialmente era stato considerato l’utilizzo del `WebSocket`. Il primo mockup, infatti, utilizzava un’implementazione simile, ma i risultati, soprattutto verso i dispositivi smart, non erano abbastanza soddisfacenti. La possibile degradazione della banda e il sovraccarico delle connessioni non garantivano la scalabilità e la qualità prevista inizialmente.

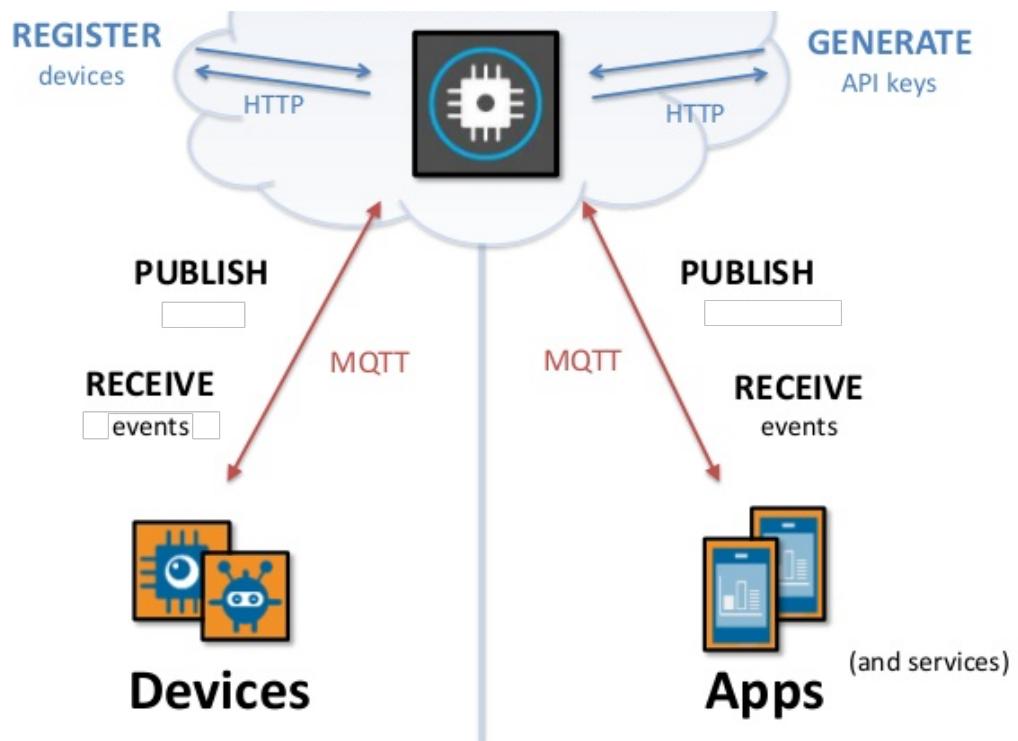
Si è dovuto cercare un protocollo alternativo che potesse garantire la stabilità necessaria per questi dispositivi. Sono stati analizzati diversi protocolli quali: AMQP, STOMP e MQTT. Il protocollo MQTT, descritto nella sezione 2.3, teoricamente poteva soddisfare i requisiti iniziali.

È stato sviluppato un applicativo `NodeJS` (“*broker*”) sul *server* di produzione connesso con il `KeyValue` tramite una comunicazione *publish/subscribe* per ricevere gli aggiornamenti di ogni utente. Il programma è riportato in appendice A.3.

Ogni *client* una volta connesso e autenticato al sito, effettua l’*handshake* verso il *broker* tramite il *token* di sessione che lo identifica all’interno del sistema. Tramite protocollo MQTT viene concordato il giusto livello di QoS in base alla sua connessione.

Alla notifica di un evento, viene effettuato il *dispatch* da parte del *broker* all'utente appropriato, seguendo questo flow:

1. connessione del dispositivo **A** del cliente Foo
2. connessione del dispositivo **B** del cliente Foo
3. connessione del dispositivo **C** del cliente Foo
4. il dispositivo **A** aggiunge/modifica/elimina un prodotto/risorsa chiamata PUT/DELETE
5. il controller Php effettua l'operazione del prodotto/risorsa e “pubblica” il relativo evento
6. disconnessione del dispositivo **C** del cliente Foo
7. il KeyValue consegna il messaggio al broker
8. il broker spedisce il messaggio al dispositivo **B**
9. il dispositivo **B** conferma il messaggio ed aggiorna le proprie risorse



**Figura 3.8:** Modello delle connessioni del progetto SmIoT

La possibilità di riutilizzare la stessa connessione MQTT per encapsulare le chiamate RESTful è ancora in fase embrionale. Con questa possibilità si potrebbe sfruttare la comunicazione già instaurata invece di crearne una nuova per intraprendere l’azione.

### 3.3.3 Difficoltà affrontate

Data la complessità dell’operazione, la migrazione è attualmente in fase di test; controllando la compatibilità con tutti i dispositivi. Per evitare problemi di regressione, il modello dati non è stato variato completamente, permettendo di ripristinare le vecchie funzionalità dinamicamente.

Non è ancora possibile cambiare il QoS una volta connesso: il passaggio dal Wi-Fi al 3G non fa aumentare il livello di qualità del servizio. Si è scelto di effettuare nuovamente la connessione negoziando la qualità del servizio corretta.

Grazie ai ping *KeepAlive* inviati dai vari dispositivi si possono interrompere le connessioni in esubero o fallite.

Attualmente l’1% degli ordini mensili viene confermato in contemporanea tra due o più dispositivi e solo il primo viene confermato. Si è scelto di applicare un *mutex* alla risorsa condivisa “carrello” e lanciare un errore al tentativo di creazione contemporanea. La sincronizzazione di questi così poco influenti non è stata ritenuta necessaria.

### 3.3.4 Conclusioni

Dall’analisi dei dati, risulta che circa il 10% degli ordini viene effettuato con una combinazione di più dispositivi. Alcuni esempi potrebbero essere:

- creazione della spesa durante il lavoro e relativa conferma dell’ordine a casa
- apertura dell’e-mail “conferma account” dal browser a seguito di una registrazione tramite app
- apertura delle e-mail CRM (*Customer relationship management*) dal browser

Sincronizzando i dati tra questi *device* non è più necessario, da parte dell’utente, ricaricare il proprio applicativo per notare le modifiche, diminuendo anche le

chiamate verso il *server*. Inoltre l'introduzione graduale di queste feature, grazie al protocollo appropriato, non comporta nessun degrado della qualità dell'esperienza utente.

### 3.4 Analisi Competitor

Durante la fase progettuale sono stati analizzati i rispettivi competitor e altri servizi simili per osservare come hanno affrontato e risolto questo problema.

Recentemente, gli sviluppatori della chat Messenger di Facebook, per salvaguardare il consumo della batteria e ridurre le latenza dei messaggi, hanno iniziato a migrare il loro codebase verso il protocollo MQTT [24]. Noti fornitori di servizi enterprise, quali AWS e IBM, hanno già sviluppato software *ad hoc* basati su questo protocollo [25, 26].

Amazon ha preferito rimanere al protocollo HTTP e tramite chiamate AJAX asincrone, propaga le operazioni sul carrello.

Al contrario, Instacart, fornitore di servizi in stile Supermercato24, ha scelto di utilizzare il `WebSocket` come principale metodo di sincronizzazione dati. Usando il servizio Firebase (acquistato da Google), riesce ad esternalizzare questo task, mantenendo il carrello aggiornato [27]. Firebase è un servizio SaaS ((Software as a service)) che permette il salvataggio e la sincronizzazione dei dati verso *client* multipli. Attraverso `WebSocket` propaga la risorsa aggiornata ad ogni cambiamento.

Si è ipotizzato che la differenza dell'implementazione tra Amazon e Instacart (e quindi Supermercato24) sia dovuto all'utilizzo medio delle due piattaforme da parte della clientela. Se su Amazon il carrello si attesta su pochi SKU (*Stock Keeping Unit*) e quindi con un minor tempo d'acquisto, per gli e-commerce di categoria *food-and-drinks* gli SKU sono mediamente maggiori, con conseguente aumento del tempo necessario per concludere l'ordine [28]. Se su Supermercato24 il tempo medio di completamento di un ordine è di 40 minuti, è dunque necessario sincronizzare tempestivamente il carrello per poterlo concludere successivamente senza perdere dati tra i dispositivi.



# Capitolo 4

## Valutazione sperimentale

I seguito verranno confrontate le soluzioni proposte per mantenere i dati sincronizzati. Nonostante la differenza del protocollo usato, entrambe condividono la stessa architettura sottostante e lo stesso pattern asincrono.

Verranno usati due approcci per validare la giusta implementazione. Il primo approccio sarà quantitativo: in base al numero di messaggi inviati/ricevuti si confronteranno il consumo delle risorse e la capacità massima di invio/ricezione. Il secondo approccio sarà qualitativo: verranno analizzati i pacchetti inviati/ricevuti descrivendo la sintassi di ognuno.

### 4.1 Confronto quantitativo

Si è analizzato il consumo medio di risorse in un dispositivo Android in diversi ambienti e la velocità di ricezione dei messaggi [29].

Si sono usate le seguenti specifiche:

- cellulare modello Nexus 5 con Android 6 completamente carico
- analisi dettagliata del dispositivo tramite software *adb* e *Battery Historian* [30, 31]
- connessioni senza compressione e senza cifratura
- *back-end* installato localmente emulando i dati di produzione
- libreria Socket.io@1.7.2 per il protocollo HTTP

- libreria Mqtt@3.1.1 per il protocollo MQTT

Il flow seguito è il seguente:

1. collegamento del cellulare tramite cavo USB con la modalità sviluppatore attivata
2. attivazione del demone di debugging `./adb devices;`
3. eliminazione dei precedenti dump
  - `./adb shell dumpsys procstats --reset;`
  - `./adb shell dumpsys batterystats --reset;`
4. lancio di 1024 benchmark in base al protocollo e alla funzionalità testata
5. download del dump `./adb bugreport > bugreport.txt;`
6. analisi del dump tramite *Battery Historian* con applicativo *Docker*
7. ripetizione degli step 3-6 per 10 campioni
8. chiusa della modalità debug `./adb kill-server;`

Le principali metriche osservate in ordine d'importanza sono:

**consumo della batteria** come requisito progettuale e principale indicatore della bontà sviluppata

**bandwidth** utilizzato, per garantire un maggior traffico dati

**CPU lato applicativo** per controllare l'utilizzo del protocollo da parte dell'applicazione

**CPU lato sistema** per controllare le *system calls* necessarie a mantenere la connessione

Mentre si è deciso di omettere:

**tempo in foreground** dato che la sincronizzazione utilizza *subroutine*

**CPU lato applicativo** del *server*, a favore della performance dei *client*

**latenza dei messaggi** poiché dai primi esami risultava trascurabile

### 4.1.1 Creazione della connessione

Si è iniziato ad osservare il costo della connessione iniziale con entrambi i protocolli. Verrà analizzato il consumo di batteria medio per l'instaurazione della comunicazione.

- MQTT invia una serie di parametri iniziali quali: *ClientId*, versione del protocollo, eventuale *payload* di autenticazione e il metodo di QoS proposto dal client
- HTTP invia una serie di headers opzionali per identificare correttamente la risorsa che si vuole ottenere. Utilizzando la porta HTTP, viene effettuata una richiesta *Upgrade* per effettuare uno switch del protocollo

Nella sezione 4.2.1 è possibile notare nel dettaglio l'invio di questi pacchetti. Si è analizzato la percentuale di batteria consumata, velocità dati e uso delle risorse in rapporto a un'ora di utilizzo Wi-Fi. Nell'appendice B.1.1 sono apprezzabili i grafici di questi test.

	HTTP	deviazione std	MQTT	deviazione std
% utilizzo batteria / ora	<b>1.55</b>	0.08	1.70	0.09
KBs / ora	<b>26649.28</b>	2702.90	10770.50	1263.60
User CPU time / ora	8.62	0.27	<b>7.00</b>	0.55
System CPU time / ora	3.45	0.23	<b>2.41</b>	0.18

**Tabella 4.1:** Metriche osservate in un'ora di tentativi di connessione

La tabella mostra come l'iniziale richiesta tramite HTTP consumi meno risorse, data la ridotta quantità di dati inviata. Poiché MQTT si fa carico di effettuare da subito la sottoscrizione ai *topic* necessari, richiede due risposte dal parte del server. Nel dettaglio, i pacchetti 4.2.1 e 4.2.2 sono sequenziali. MQTT conferma la *subscription* ad ogni riconnessione tramite il flag *cleanstart=true*, così facendo sono sempre necessarie due richieste per connessione, ciò depone a favore del protocollo HTTP. Disabilitando questa opzione si possono ottenere dei risultati simili per entrambi i protocolli, ma il *server* dovrebbe salvare questa *subscription* per un tempo determinato a priori.

#### 4.1.2 Mantenimento della connessione

Stabilire tempestivamente se la connessione verso il *server* (e viceversa) è ancora disponibile, senza aspettare un lungo timeout TCP / IP, aumenta le probabilità che il successivo pacchetto dati venga spedito (o ricevuto) e in caso negativo, è compito del *client* cercare di ristabilire la comunicazione.

- MQTT fornisce nativamente il metodo PINGREQ che invia un ping per sapere se la connessione è ancora valida
- HTTP non propone questa funzionalità, ma può essere integrata programmaticamente tramite un meccanismo di *polling*

Sono stati effettuati diversi test per ogni timeout con un campione di dieci minuti.

	% utilizzo batteria / ora			
	3G		Wi-Fi	
intervallo secondi <i>KeepAlive</i>	HTTP	MQTT	HTTP	MQTT
60	1.11368	<b>0.72280</b>	0.15778	<b>0.00994</b>
120	0.48512	<b>0.31856</b>	0.08713	<b>0.00417</b>
240	0.33092	<b>0.15842</b>	0.02836	<b>0.00169</b>
480	0.08078	<b>0.07806</b>	0.00763	<b>0.00051</b>

**Tabella 4.2:** Consumo della batteria in entrambi gli ambienti per mantenere la connessione

Il ping nativo 4.2.3 MQTT è più conveniente in entrambi gli ambienti rispetto all'emulazione programmatica tramite *pooling*. Normalmente il costo della creazione della connessione è trascurabile, ma non il suo mantenimento nel tempo. Se si considera che il 50% di traffico proviene da mobile e che ci vogliono 40 minuti per completare un ordine, con un timeout di 240 secondi si riesce a risparmiare il 48% di batteria per ordine rispetto al protocollo HTTP.

#### 4.1.3 Ricezione dati

La ricezione dei dati da parte dei *client*, a seguito di un evento, è il principale utilizzo di questo pattern. I messaggi dovrebbero essere recapitati, indipendentemente dalla qualità e dalla congestione della linea.

Sono stati inviati 1024 messaggi (1 byte ciascuno) il più velocemente possibile da parte del *broker*. È stato abilitato il flag QoS 1, inibiti i ping *KeepAlive* e si deliberatamente degradato la rete con un elevato rumore da parte di altri software installati.

	3G		Wi-Fi	
	HTTP	MQTT	HTTP	MQTT
messaggi / ora	1462	<b>160032</b>	3382	<b>263068</b>
messaggi ricevuti	237 / 1024	<b>1024 / 1024</b>	521 / 1024	<b>1024 / 1024</b>

**Tabella 4.3:** Numero di messaggi ricevuti da parte del client

Grazie al QoS nativo MQTT, si fa carico di controllare che tutti i messaggi siano effettivamente recapitati. Data la criticità nella sincronizzazione dei dati, l'affidabilità della ricezione è un requisito iniziale. Se un messaggio non è stato ricevuto, viene rispedito dal *server* il prima possibile.

Non è stato previsto nessun meccanismo di *queue* per discriminare se un messaggio è in ritardo, analizzando solo l'attendibilità dell'implementazione. Si potrebbe aggiungere un identificativo incrementale per ogni risorsa scambiata e scartare gli aggiornamenti obsoleti.

Si è analizzato la percentuale di batteria consumata, velocità dati e uso delle risorse in rapporto a un'ora di utilizzo Wi-Fi. Nell'appendice B.1.2 sono apprezzabili i grafici di questi test.

	HTTP	deviazione std	MQTT	deviazione std
% utilizzo batteria / ora	4.12	0.25	<b>1.11</b>	0.31
KBs / ora	15777.10	1888.67	<b>23333.21</b>	2720.71
User CPU time / ora	21.17	1.56	<b>3.72</b>	0.79
System CPU time / ora	3.33	0.17	<b>2.46</b>	0.45

**Tabella 4.4:** Metriche osservate in un'ora di ricezione dati

Dalla tabella si può evincere la leggerezza del protocollo MQTT confrontando il tempo speso da parte della CPU nello stato dell'applicazione. Nello specifico è stato

adottato la stessa codifica *application/json; charset=utf-8* osservabile nella sezione 4.2.4.

## 4.2 Analisi pacchetti

Tramite software *Wireshark* si è analizzato i principali pacchetti scambiati, mostrando prima il pacchetto HTTP e poi MQTT.

Si sono usate le seguenti specifiche:

- analisi dettagliata dei pacchetti tramite software *Wireshark*
- connessioni senza compressione e senza cifratura
- *back-end* installato localmente emulando i dati di produzione

### 4.2.1 Creazione della connessione

Tramite protocollo HTTP viene effettuato l'*handshake* necessario per creare il *WebSocket*. Tra l'indirizzo *0x0040* e *0x0070* si può notare il cambio di protocollo, come evidenziato nella sezione 2.2.1. MQTT fornisce tutti i parametri necessari, si noti il token di sessione per autenticare la richiesta nel byte all'indirizzo *0x0060*.

```

0000  02 00 00 00 45 00 00 e3 3e f2 40 00 40 06 00 00  ....E...>.@.@
0010  7f 00 00 01 7f 00 00 01 0b b8 dc 7f 01 d3 94 ac  .....
0020  8d de 93 69 80 18 31 c3 fe d7 00 00 01 01 08 0a  ...i..1.....
0030  33 3c a5 2b 33 3c a5 2a 48 54 54 50 2f 31 2e 31  3<.+3<.*HTTP/1.1
0040  20 31 30 31 20 53 77 69 74 63 68 69 6e 67 20 50  101 Switching P
0050  72 6f 74 6f 63 6f 6c 73 0d 0a 55 70 67 72 61 64  rotocols..Upgrad
0060  65 3a 20 77 65 62 73 6f 63 6b 65 74 0d 0a 43 6f  e: websocket..Co
0070  6e 6e 65 63 74 69 6f 6e 3a 20 55 70 67 72 61 64  nnection: Upgrad
0080  65 0d 0a 53 65 63 2d 57 65 62 53 6f 63 6b 65 74  e..Sec-WebSocket
0090  2d 41 63 63 65 70 74 3a 20 66 63 72 6f 6f 2b 47  -Accept: fcroo+G
00a0  33 50 30 62 71 32 52 66 53 38 45 5a 39 6f 37 35  3P0bq2Rfs8EZ9o75
00b0  72 6d 4b 45 3d 0d 0a 53 65 63 2d 57 65 62 53 6f  rmKE=..Sec-WebSo
00c0  63 6b 65 74 2d 45 78 74 65 6e 73 69 6f 6e 73 3a  cket-Extensions:
00d0  20 70 65 72 6d 65 73 73 61 67 65 2d 64 65 66 6c  permessage-defl
00e0  61 74 65 0d 0a 0d 0a  atem.....

```

**Listing 4.1:** HTTP Upgrade Request per creare il socket WS

```

0000  02 00 00 00 45 00 00 6c fe c2 40 00 40 06 00 00  ....E..l..@.@
0010  7f 00 00 01 7f 00 00 01 de 5b 07 5b 56 1f e3 58  ....[. [V..X

```

```

0020 b5 ad d1 73 80 18 31 d7 fe 60 00 00 01 01 08 0a ...s..1..`.....
0030 33 65 87 64 33 65 87 62 10 36 00 04 4d 51 54 54 3e.d3e.b.6..MQTT
0040 04 c2 00 14 00 20 30 39 31 65 31 30 38 38 31 36 ..... 091e108816
0050 33 66 34 64 65 61 39 30 35 66 64 32 32 32 61 64 3f4dea905fd222ad
0060 34 61 66 36 61 66 00 06 66 6f 6f 62 61 72 00 00 4af6af..foobar..

```

**Listing 4.2:** MQTT Creazione socket

### 4.2.2 Sottoscrizione al topic

La sottoscrizione al topic HTTP viene effettuata in Javascript e viene spezzata in due richieste TCP. MQTT permette la sottoscrizione simultanea a più argomenti contemporaneamente con una singola richiesta. Questo pacchetto viene inviato immediatamente dopo la creazione della connessione. Il dettaglio dei pacchetti sono apprezzabili nell'appendice B.2.1.

### 4.2.3 Mantenimento della connessione

Il probe *KeepAlive* in HTTP serve solo per controllare che la connessione sia ancora attiva, mentre MQTT si aspetta anche una risposta da parte il *server*.

```

0000 02 00 00 00 45 00 00 37 13 b4 40 00 40 06 00 00 .....E..7..@.0...
0010 7f 00 00 01 7f 00 00 01 0b b8 dc 7f 01 d3 98 5d .....].....
0020 8d de 93 a7 80 18 31 c1 fe 2b 00 00 01 01 08 0a .....1..+.....
0030 33 3d 06 83 33 3d 06 83 81 01 33 3=..3=....3

```

**Listing 4.3:** HTTP Probe KeepAlive

```

0000 02 00 00 00 45 00 00 36 26 3c 40 00 40 06 00 00 .....E..6&<@.0...
0010 7f 00 00 01 7f 00 00 01 de 5b 07 5b 56 1f e3 9c .....[. [V...
0020 b5 ad d1 7c 80 18 31 d7 fe 2a 00 00 01 01 08 0a ...|..1..*.....
0030 33 65 e0 da 33 65 92 d7 c0 00 3e..3e....

```

**Listing 4.4:** MQTT Probe KeepAlive

### 4.2.4 Ricezione dati

La serializzazione dei dati avviene attraverso il formato JSON. I dettagli dei pacchetti sono apprezzabili nell'appendice B.2.2.

#### 4.2.5 Chiusura della connessione

Per entrambi i protocolli, la chiusura della connessione (senza interruzione forzata) viene semplicemente notificata al *server*.

```
0000  02 00 00 00 45 00 00 38 28 69 40 00 40 06 00 00 ....E..8(i@.0...
0010  7f 00 00 01 7f 00 00 01 0b b8 dc 7f 01 d3 99 54 .....T
0020  8d de 93 c2 80 18 31 c1 fe 2c 00 00 01 01 08 0a .....1.,.....
0030  33 3d f1 42 33 3d f1 42 88 02 03 e9            3=.B3=.B....
```

**Listing 4.5:** HTTP Chiusura della connessione

```
0000  02 00 00 00 45 00 00 36 8f 40 40 00 40 06 00 00 ....E..6.@@.0...
0010  7f 00 00 01 7f 00 00 01 de 5b 07 5b 56 1f e3 9e .....[. [V...
0020  b5 ad d2 59 80 18 31 d0 fe 2a 00 00 01 01 08 0a ...Y..1..*.....
0030  33 65 f9 2a 33 65 eb 75 e0 00            3e.*3e.u..
```

**Listing 4.6:** MQTT Chiusura della connessione



# Capitolo 5

## Conclusioni

INCRONIZZARE notevoli quantità di dati in tempo reale è un aspetto critico per molte piattaforme online. L'utilizzo di una connessione persistente implica l'uso continuo della connessione dati, *parsing* dello stream input e *encoding* dello stream output. Compiti non indifferenti a seguito della stima tra i 50 e 100 miliardi dei dispositivi connessi entro il 2020. Fenomeni di *spinlock* o di *polling* per ottenere un risorsa aggiornata, dovrebbero essere abbandonati a favore di una comunicazione asincrona tra gli interlocutori.

Il lavoro di tesi si è sviluppato con un'introduzione ai paradigmi di comunicazione e background dei protocolli nel capitolo 2. Nel capitolo 3 sono state proposte due soluzioni implementative diverse con una panoramica tecnica dei software attuali presso l'azienda Supermercato24. Queste applicazioni sono state poi valutate tramite test nel capitolo 4.

Le prove effettuate hanno confermato come sia estremamente complesso costruire un modello che caratterizzi al meglio il risparmio di risorse nella creazione e nel mantenimento della connessione. Tra queste variabili, di cui bisogna tenere necessariamente conto, c'è la reale possibilità che i pacchetti dati possano non arrivare a destinazione. Di conseguenza la qualità del servizio deve essere una prerogativa irrinunciabile.

HTTP, ampiamente utilizzato nella rete, nonostante sia avido di risorse garantisce un buon punto di partenza. L'introduzione del WebSocket che implementa la comunicazione asincrona, ha permesso a questo protocollo di mantenersi ancora *standard de facto* in Internet. La nuova versione di questo protocollo aprirà le porte

a nuovi scenari, come già sta facendo Google per la modifica dei documenti in tempo reale in *Drive*.

MQTT, di contro, è stato progettato per soddisfare da subito queste esigenze offrendo una serie di metodi nativi sgravando lo sviluppatore da questo compito. La garanzia della consegna del messaggio prevista su tre livelli a secondo dalla criticità e la leggerezza dello stream dati, lo pone come un'alternativa più che valida.

Dalle verifiche sperimentali sono emerse le differenze tra i due protocolli sulle risorse utilizzate. In un'ora di test il consumo di batteria con HTTP è il triplo rispetto a MQTT. Considerato il tempo trascorso dalla CPU eseguendo il codice nello *user space*, denota la leggerezza del secondo protocollo.

Grazie a questa analisi è stato approvata la sperimentazione del protocollo MQTT come vettore nella sincronizzazione dati lato e-commerce. Un altro problema attuale in Supermercato24 è il consumo significativo del gestionale dei fattorini. La comunicazione della posizione geografica tramite chiamate AJAX, esaurisce velocemente la batteria dei dispositivi; problema comune ad altri servizi simili come Deliveroo, Foodora e Just Eat. L'utilizzo di *power bank* a supporto diventa quindi indispensabile per poter affrontare più ordini. Concluso con successo il test del progetto 3.3 verrà valutata in futuro questa migrazione. Uno sviluppo interessante sarebbe confrontare questa possibile evoluzione ed i benefici ottenuti. Un'altra opportunità potrebbe essere l'approfondimento di questi i test con l'attivazione della compressione e della crittografia, sacrificando la potenza computazionale a favore della connessione dati.

Concludendo, i protocolli proposti rappresentano soltanto un piccolo insieme. L'affidabilità e la compatibilità restano gli ambiti in cui la ricerca dovrà muoversi in preparazione della crescita esponenziale nell'ambiente IoT.



# Appendice A

## Sorgenti

### A.1 Socksberry Controller

```
var nsp = io.of('/socksberry').use(function(socket, next) {  
  
    next();  
});  
  
/*  
 * pub/sub pattern  
 */  
nsp.on('connection', function(socket) {  
  
    /*  
     * subscribe  
     */  
    var sub = cached(my); // KeyValue broker  
  
    /**  
     * subscribe local object as broker topic as callback  
     *  
     * @param {String} name - model name  
     * @param {Object} object - model object  
     * @param {Integer} counter - switch between KeyValue Database  
     */  
    modelsLoop(function(name, object, counter) {  
  
        sub.select(counter, function() {  
  
            sub.subscribe(name);  
        });  
    });  
  
    /**
```

```

* message from KeyValue broker
*/
sub.on('message', function(channel, message) {

    if (models[channel] === undefined) { // missing rigth channel
        return; // next
    }
    var cmd = JSON.parse(message);

    models[channel].update(cmd.event.split('.'), cmd.data,
        function(err, res) { // update local model

            if (res !== null) {

                var out = {
                    status: true,
                    data: res
                }; // stringify data, instead of send out directly
                nsp.to(socket.id).emit(channel, JSON.stringify(out));
            }
        });
});

/***
 * event from WebSocket broker
*/
socket.on('disconnect', function() {

    /**
     * unsubscribe local object from broker topic as callback
     *
     * @param {String} name - model name
     */
    modelsLoop(function(name) {

        sub.unsubscribe(name);
    });
}).on('init', function() {

    nsp.to(socket.id).emit('init', JSON.stringify(staticOut));
});
});

```

**Listing A.1:** Doppio broker HTTP: KeyValue e WebSocket

## A.2 Socksberry View

```
// define socket connection handler
```

```

var manager, socket;

// in case of network errors, init data with static (empty) model
app.setOfflineModel = function() {
    app.user = _data.user;
    app.s24 = _data.s24;
    app.offline = true;
};

// After manager has installed correctly the connection with server,
// init socket and initialize event callback
app.initSocket = function() {

    socket = manager.socket('/socksberry');

    // define socket io callback
    socket.on('error', function(data) {

        console.error(data);
        app.offline = true;

    }).on('disconnect', function() {

        console.log('disconnect');
        app.offline = true;

    }).on('connect', function() {

        console.log('connect');
        socket.emit('init');
        app.offline = true;
        app.placeholderMap = false;

    }).on('user', function(data) {

        app.user = JSON.parse(data).data;

    }).on('s24', function(data) {

        app.s24 = JSON.parse(data).data;

    }).on('init', function(data) {

        var data = JSON.parse(data).data;
        app.s24 = data.s24;
        app.user = data.user;
        app.offline = false;

    });

};

```

```

};

// init Socket IO
app.init = function() {

    var url = config.protocol + '://' + config.ip + ':' + config.port;
    var opts = {
        reconnectionDelay: 10000
    };
    manager = io.Manager(url, opts);
    // save reference, for debug purpose
    // window.manager = manager;

    // static (offline) initialization
    app.setOfflineModel();
    app.placeholderMap = true;
    // init socket
    app.initSocket();
};

// See https://github.com/Polymer/polymer/issues/1381
window.addEventListener('WebComponentsReady', function() {
    // imports are loaded and elements have been registered
    // init data
    app.init();
});

```

**Listing A.2:** HTTP WebSocket subscriber

### A.3 SmIoT Controller

```

/**
 * move consumer outside from server.
 * Otherwise duplicated messages are sent from every client, using dispatcher
 */
consumerInterface.on('message_buffer', function(channel, message) {

    var channelString = channel.toString();
    var funnels = channelString.split('/');
    var userId = funnels[0];

    if (dispatcher.emit(userId, funnels.slice(1).join('/'), message)) { // had user
        // pass
    } else { // remove subscriber
        consumerInterface.unsubscribe(channelString);
    }
});

```

```

/**
 * transform net stream into mqtt client
 */
serverInterface.on('connection', function(stream) {

    stream.setKeepAlive(options.keepAlive);
    stream.setNoDelay(options.noDelay);
    stream.setTimeout(options.timeout);
    var client = mqttConnection(stream);

    /**
     * connection between broker interface and server interface
     *
     * @function publisher
     * @param {Buffer} channel - client topic
     * @param {Buffer} message - client message
     * @param {Boolean} [pingreq] - check if this eventListener is alive
     */
    var publisher = function(channel, message, pingreq) {

        if (pingreq) {
            return;
        }

        if (client.authorized && client.topics[client.topicPrefix + channel]) {
            client.publish({
                qos: returnCode.SUBSCRIPTION_QOS_0,
                topic: channel,
                payload: message
            });
        }
    };

    client.once('connect', function(packet) {

        /**
         * <pre>
         * 'version': the protocol version string
         * 'versionNum': the protocol version number
         * 'keepalive': the client's keepalive period
         * 'clientId': the client's ID
         * 'will': an object with the following keys:
         *   'topic': the client's will topic
         *   'payload': the will message
         *   'retain': will retain flag
         *   'qos': will qos level
         *   'clean': clean start flag
         *   'username': v3.1 username
         *   'password': v3.1 password
         */
    });
}

```

```

* </pre>
*/
stream.pause(); // pauses the reading of data

var response = {
  returnCode: returnCode.CONNECTION_REFUSED_SERVER_UNAVAILABLE
};

var userToken = packet.username;
if (!userToken) {
  response.returnCode = returnCode.CONNECTION_REFUSED_NOT_AUTHORIZED;
  stream.resume();
  client.connack(response);
  return;
}

brokerInterface.get(userToken, function(err, userId) {

  if (err) {
    // pass
  } else if (userId) { // client init

    dispatcher.on(userId, publisher);

    client.authorized = true;
    client.topics = {};
    client.clientId = packet.clientId;
    client.userId = userId;
    client.userToken = userToken;
    client.topicPrefix = userId + '/';

    response.returnCode = returnCode.CONNECTION_ACCEPTED;
  } else {
    response.returnCode = returnCode.CONNECTION_REFUSED_BAD_CREDENTIALS;
  }

  stream.resume();
  client.connack(response);
});

}).on('subscribe', function(packet) {

  var response = {
    messageId: packet.messageId,
    granted: []
  };

  var channels = [];
  var notGranted = [];
  var subscriptions = {};

```

```

    for (var i = 0, ii = packet.subscriptions.length; i < ii; ++i) {
        var topic = packet.subscriptions[i].topic;
        notGranted[i] = returnCode.SUBSCRIPTION_FAILURE;

        if (client.authorized) {
            response.granted[i] = returnCode.SUBSCRIPTION_QOS_0;
            topic = client.topicPrefix + topic;

            if (subscriptions[topic] === undefined) { // unique
                subscriptions[topic] = true;
                channels[i] = topic;
            }
        }
    }

    if (client.authorized === false) {
        response.granted = notGranted;
        client.suback(response);
        return;
    }

    // https://github.com/NodeRedis/node_redis/issues/1188
    consumerInterface.subscribe(channels, function(err, latestChannel) {

        if (err) {
            response.granted = notGranted;
        } else if (latestChannel) {
            Object.assign(client.topics, subscriptions);
        } else {
            response.granted = notGranted;
        }
        client.suback(response);
    });
}

).on('unsubscribe', function(packet) {

    var response = {
        messageId: packet.messageId
    };

    if (!client.authorized) {
        client.unsuback(response);
        return;
    }

    var unsubscriptions = [];
    for (var i = 0, ii = packet.unsubscriptions.length; i < ii; ++i) {
        var topic = client.topicPrefix + packet.unsubscriptions[i];

        if (client.topics[topic]) {

```

```

        unsubscriptions.push(topic);
        delete (client.topics[topic]);
    }
}

client.unsuback(response);
}).on('pingreq', function() {

    if (client.authorized) {
        client.pingresp();
    } else {
        client.destroy();
    }
}).once('close', function() {

    if (client.stream.destroyed === false) {
        client.destroy();
    }

    if (!client.authorized) {
        return;
    }

    client.authorized = false;
    dispatcher.removeListener(client.userId, publisher);
    var unsubscriptions = Object.keys(client.topics);
    if (!unsubscriptions) {
        return; // pass
    }

    if (!dispatcher.emit(client.userId, [], true)) {
        consumerInterface.unsubscribe(unsubscriptions, function(err) {

            if (err) {
                // pass
            } else {
                client.topics = {};
            }
        });
    }
}).once('disconnect', function() {

    client.destroy();
}).once('error', function(err) {

    client.destroy();
});

stream.once('timeout', function() {

```

```
    client.destroy();
});
});
```

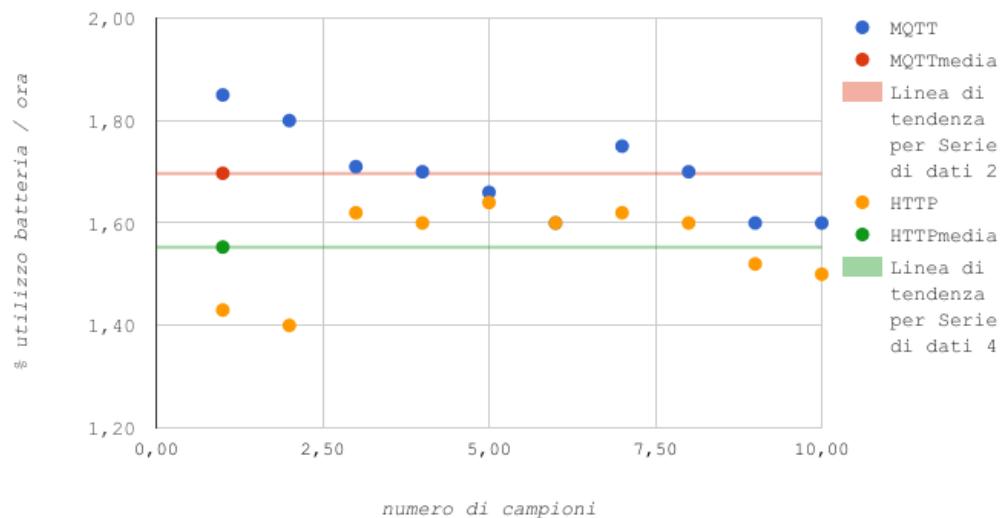
**Listing A.3:** Dispatcher broker MQTT

# Appendice B

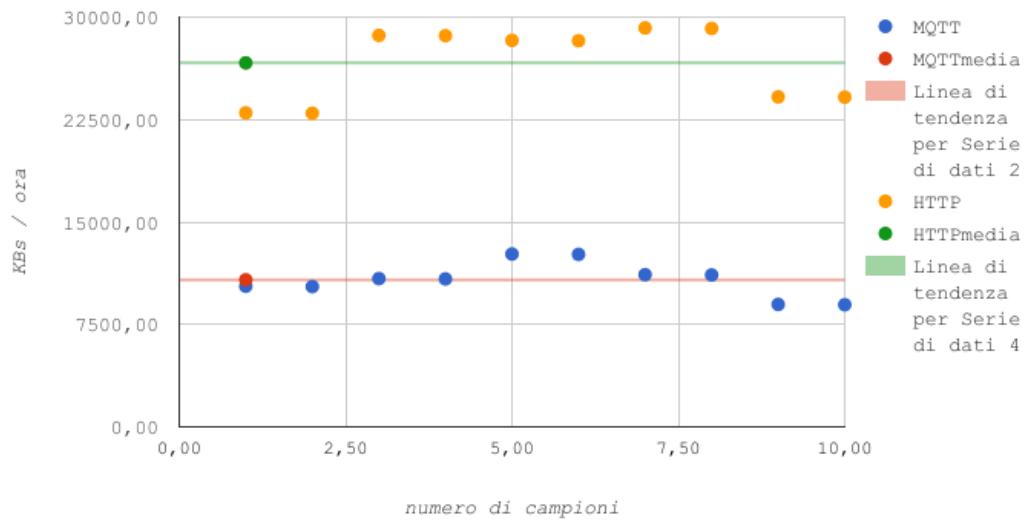
## Test

### B.1 Grafici risultati

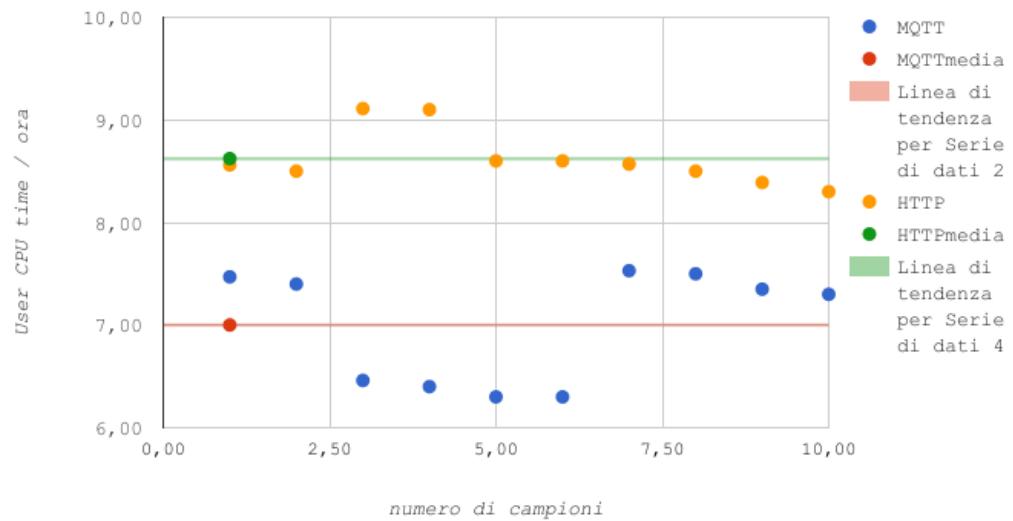
#### B.1.1 Creazione della connessione



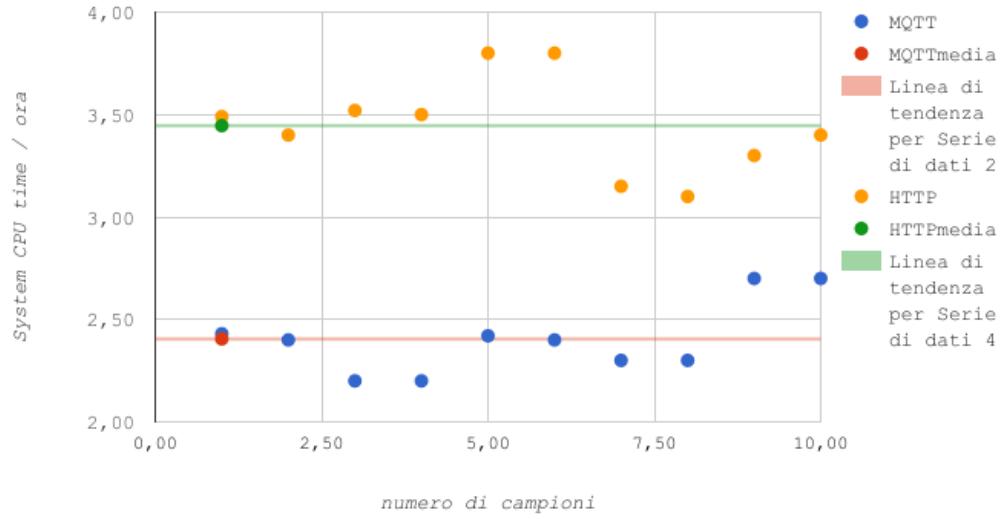
**Figura B.1:** % utilizzo batteria / ora nella creazione della connessione



**Figura B.2:** KBS / ora nella creazione della connessione

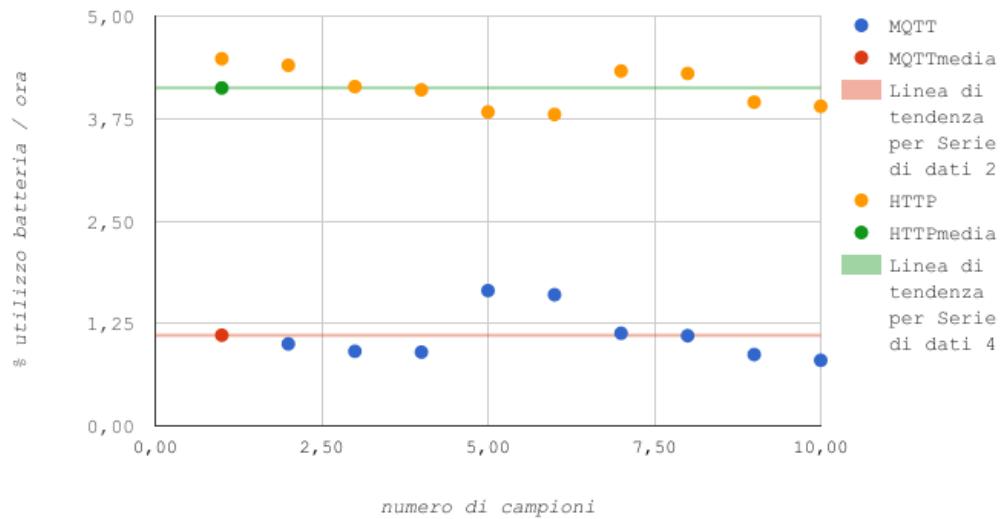


**Figura B.3:** User CPU time / ora nella creazione della connessione

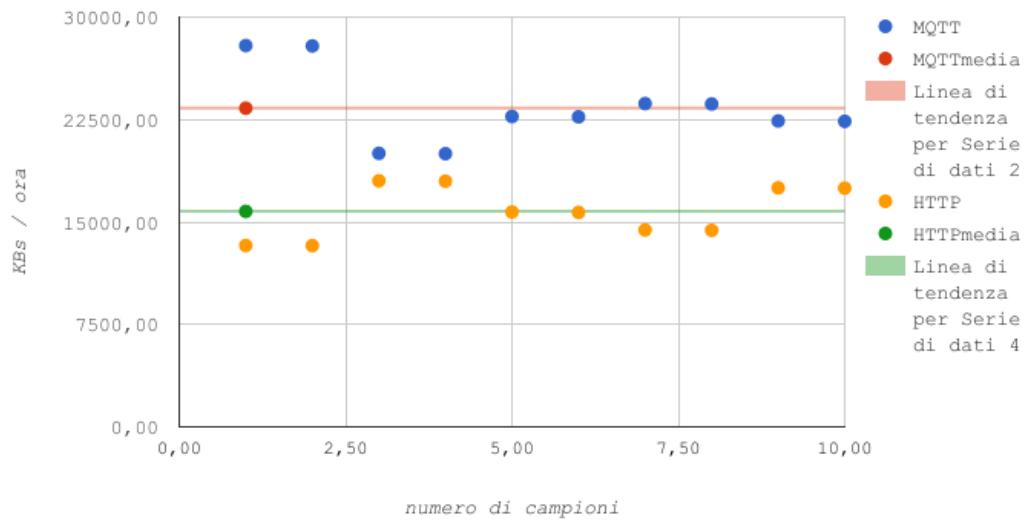


**Figura B.4:** System CPU time / ora nella creazione della connessione

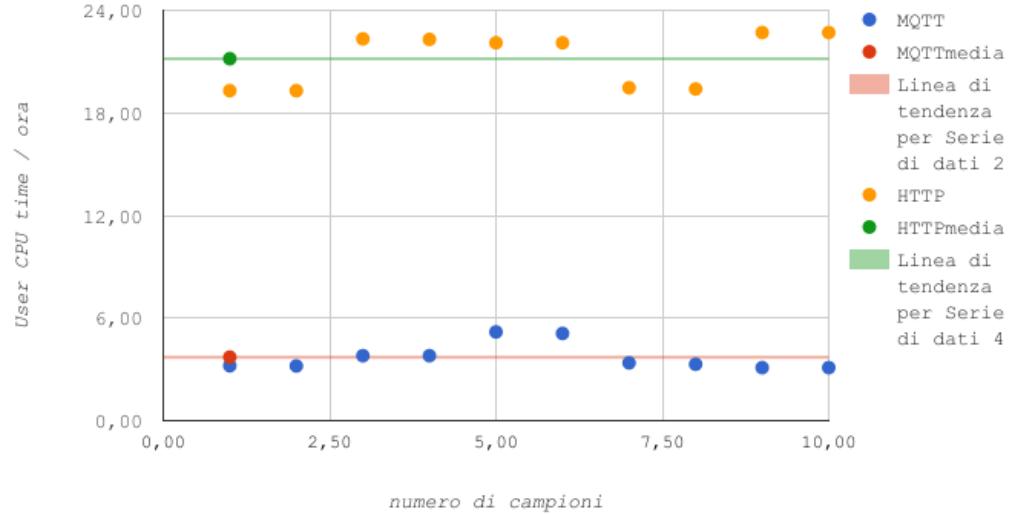
### B.1.2 Ricezione dati



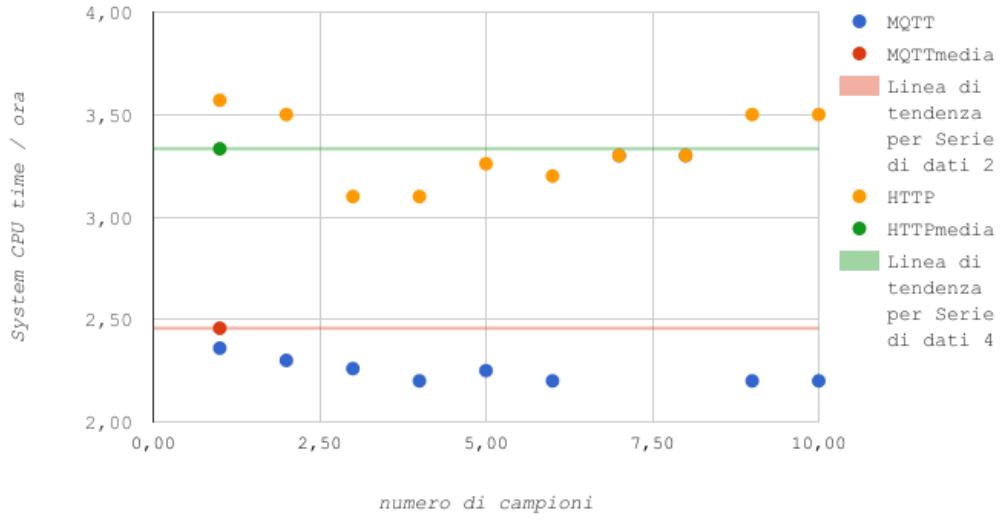
**Figura B.5:** % utilizzo batteria / ora nella ricezione dei dati



**Figura B.6:** KBS / ora nella ricezione dei dati



**Figura B.7:** User CPU time / ora nella ricezione dei dati



**Figura B.8:** System CPU time / ora nella ricezione dei dati

## B.2 Dettaglio dei pacchetti

### B.2.1 Sottoscrizione al topic

```

0000 02 00 00 00 45 00 02 4b 1d 79 40 00 40 06 00 00 ....E..K.y@.@
0010 7f 00 00 01 7f 00 00 01 dc 75 0b b8 47 fe 66 15 .....u..G.f.
0020 76 23 7f d6 80 18 31 3c 00 40 00 00 01 01 08 0a v#....1<.@.....
0030 33 3c a5 6d 33 3c a5 27 50 4f 53 54 20 2f 73 6f 3<.m3<.'POST /so
0040 63 6b 65 74 2e 69 6f 2f 3f 45 49 4f 3d 33 26 74 cket.io/?EIO=3&t
0050 72 61 6e 73 70 6f 72 74 3d 70 6f 6c 6c 69 6e 67 ransport=polling
0060 26 74 3d 4c 65 71 32 37 65 52 26 73 69 64 3d 58 &t=Leq27eR&sid=X
0070 64 74 62 59 78 68 6c 61 76 4c 6d 4d 51 71 51 41 dtbYxhlavLmMQqQA
0080 41 41 46 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f AAF HTTP/1.1..Ho
0090 73 74 3a 20 31 32 37 2e 30 2e 30 2e 31 3a 33 30 st: 127.0.0.1:30
00a0 30 30 0d 0a 43 6f 6e 6e 65 63 74 69 6f 6e 3a 20 00..Connection:
00b0 6b 65 65 70 2d 61 6c 69 76 65 0d 0a 43 6f 6e 74 keep-alive..Cont
00c0 65 6e 74 2d 4c 65 6e 67 74 68 3a 20 31 36 0d 0a ent-Length: 16..
00d0 41 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a 4f 72 69 Accept: */*..Ori
00e0 67 69 6e 3a 20 68 74 74 70 3a 2f 2f 31 32 37 2e gin: http://127.
00f0 30 2e 30 2e 31 3a 33 30 30 30 0d 0a 55 73 65 72 0.0.1:3000..User
0100 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c 61 2f -Agent: Mozilla/
0110 35 2e 30 20 28 4d 61 63 69 6e 74 6f 73 68 3b 20 5.0 (Macintosh;
0120 49 6e 74 65 6c 20 4d 61 63 20 4f 53 20 58 20 31 Intel Mac OS X 1
0130 30 5f 31 32 5f 33 29 20 41 70 70 6c 65 57 65 62 0_12_3) AppleWeb
0140 4b 69 74 2f 35 33 37 2e 33 36 20 28 4b 48 54 4d Kit/537.36 (KHTML
0150 4c 2c 20 6c 69 6b 65 20 47 65 63 6b 6f 29 20 43 L, like Gecko) C
0160 68 72 6f 6d 65 2f 35 38 2e 30 2e 33 30 30 34 2e hrome/58.0.3004.

```

```

0170 33 20 53 61 66 61 72 69 2f 35 33 37 2e 33 36 0d 3 Safari/537.36.
0180 0a 43 6f 6e 74 65 6e 74 2d 74 79 70 65 3a 20 74 .Content-type: t
0190 65 78 74 2f 70 6c 61 69 6e 3b 63 68 61 72 73 65 ext/plain;charse
01a0 74 3d 55 54 46 2d 38 0d 0a 44 4e 54 3a 20 31 0d t=UTF-8..DNT: 1.
01b0 0a 52 65 66 65 72 65 72 3a 20 68 74 74 70 3a 2f .Referer: http:/
01c0 2f 31 32 37 2e 30 2e 30 2e 31 3a 33 30 30 30 2f /127.0.0.1:3000/
01d0 0d 0a 41 63 63 65 70 74 2d 45 6e 63 6f 64 69 6e ..Accept-Encodin
01e0 67 3a 20 67 7a 69 70 2c 20 64 65 66 6c 61 74 65 g: gzip, deflate
01f0 2c 20 62 72 0d 0a 41 63 63 65 70 74 2d 4c 61 6e , br..Accept-Lan
0200 67 75 61 67 65 3a 20 69 74 2d 49 54 2c 69 74 3b guage: it-IT,it;
0210 71 3d 30 2e 38 2c 65 6e 2d 55 53 3b 71 3d 30 2e q=0.8,en-US;q=0.
0220 36 2c 65 6e 3b 71 3d 30 2e 34 0d 0a 43 6f 6f 6b 6,en;q=0.4..Cook
0230 69 65 3a 20 69 6f 3d 58 64 74 62 59 78 68 6c 61 ie: io=XdtbYxhla
0240 76 4c 6d 4d 51 71 51 41 41 46 0d 0a 0d 0a vLmMQqQAAAF.....

```

```

0000 02 00 00 00 45 00 00 44 c6 90 40 00 40 06 00 00 ....E..D..@.0...
0010 7f 00 00 01 7f 00 00 01 dc 75 0b b8 47 fe 68 2c .....u..G.h,
0020 76 23 7f d6 80 18 31 3c fe 38 00 00 01 01 08 0a v#....1<.8.....
0030 33 3c a5 6d 33 3c a5 27 31 33 3a 34 30 2f 73 6f 3<.m3<.'13:40/so
0040 63 6b 73 62 65 72 72 79 cksberry

```

**Listing B.1:** HTTP Sottoscrizione al topic "socksberry"

```

0000 02 00 00 00 45 00 00 40 6c ed 40 00 40 06 00 00 ....E..@1.@.0...
0010 7f 00 00 01 7f 00 00 01 de 5b 07 5b 56 1f e3 90 .....[. [V...
0020 b5 ad d1 77 80 18 31 d7 fe 34 00 00 01 01 08 0a ...w..1..4.....
0030 33 65 92 d6 33 65 87 65 82 0a 00 01 00 05 73 6d 3e..3e.e.....sm
0040 69 6f 74 00 iot.

```

**Listing B.2:** MQTT Sottoscrizione al topic "smiot"

### B.2.2 Ricezione dati

```

0000 02 00 00 00 45 00 01 22 ba 6d 40 00 40 06 00 00 ....E.."m@.0...
0010 7f 00 00 01 7f 00 00 01 0b b8 dc 7f 01 d3 98 63 .....c
0020 8d de 93 b0 80 18 31 c1 ff 16 00 00 01 01 08 0a .....1.....
0030 33 3d b1 8c 33 3d 67 ec 81 7e 00 ea 34 32 2f 73 3=..3=g..~..42/s
0040 6f 63 6b 73 62 65 72 72 79 2c 5b 22 75 73 65 72 ocksberry,[ "user
0050 22 2c 22 7b 5c 22 73 74 61 74 75 73 5c 22 3a 74 ", {"\status":t
0060 72 75 65 2c 5c 22 64 61 74 61 5c 22 3a 7b 5c 22 rue,\data\:{\
0070 72 65 67 69 73 74 72 61 74 69 6f 6e 5c 22 3a 7b registration\:{\
0080 5c 22 64 61 69 6c 79 5c 22 3a 30 2c 5c 22 64 61 \\"daily\":0,\\"da
0090 69 6c 79 5f 72 65 63 6f 72 64 5c 22 3a 31 35 39 ily_record\":159
00a0 2c 5c 22 74 6f 74 61 6c 5c 22 3a 33 31 37 2c 5c ,\\"total\":317,\\
00b0 22 76 61 72 69 61 74 69 6f 6e 5c 22 3a 2d 31 30 "variation\":-10
00c0 30 7d 2c 5c 22 73 65 73 73 69 6f 6e 5c 22 3a 7b 0},\\"session\":{
00d0 5c 22 64 61 69 6c 79 5c 22 3a 32 2c 5c 22 64 61 \\"daily\":2,\\"da
00e0 69 6c 79 5f 72 65 63 6f 72 64 5c 22 3a 31 31 38 ily_record\":118
00f0 33 2c 5c 22 74 6f 74 61 6c 5c 22 3a 32 35 39 35 3,\\"total\":2595

```

```

0100  31 2c 5c 22 76 61 72 69 61 74 69 6f 6e 5c 22 3a  1,"variation":1
0110  2d 39 38 2e 37 33 33 39 31 30 31 30 37 36 31 32 -98.733910107612
0120  33 7d 7d 7d 22 5d

```

**Listing B.3:** HTTP Payload dati inviato dal server

```

0000  02 00 00 00 45 00 01 05 11 c0 40 00 40 06 00 00 ....E.....@.@
0010  7f 00 00 01 7f 00 00 01 07 5b de 5b b5 ad d1 88 .....[. [...]
0020  56 1f e3 9e 80 18 31 d5 fe f9 00 00 01 01 08 0a V....1.....
0030  33 65 eb 75 33 65 eb 75 7b 5c 22 73 74 61 74 75 3e.u3e.u{"statu
0040  73 5c 22 3a 74 72 75 65 2c 5c 22 64 61 74 61 5c s":true,"data\
0050  22 3a 7b 5c 22 72 65 67 69 73 74 72 61 74 69 6f ":"{"registratio
0060  6e 5c 22 3a 7b 5c 22 64 61 69 6c 79 5c 22 3a 30 n": {"daily":0
0070  2c 5c 22 64 61 69 6c 79 5f 72 65 63 6f 72 64 5c , "daily_record\
0080  22 3a 31 35 39 2c 5c 22 74 6f 74 61 6c 5c 22 3a ":"159, "total":1
0090  33 31 37 2c 5c 22 76 61 72 69 61 74 69 6f 6e 5c 317, "variation\
00a0  22 3a 2d 31 30 30 7d 2c 5c 22 73 65 73 73 69 6f "-100}, "session":2
00b0  6e 5c 22 3a 7b 5c 22 64 61 69 6c 79 5c 22 3a 32 n": {"daily":2
00c0  2c 5c 22 64 61 69 6c 79 5f 72 65 63 6f 72 64 5c , "daily_record\
00d0  22 3a 31 31 38 33 2c 5c 22 74 6f 74 61 6c 5c 22 ":"1183, "total":1
00e0  3a 32 35 39 35 31 2c 5c 22 76 61 72 69 61 74 69 :25951, "variation":1
00f0  6f 6e 5c 22 3a 2d 39 38 2e 37 33 33 39 31 30 31 on": -98.7339101
0100  30 37 36 31 32 33 7d 7d 7d

```

**Listing B.4:** MQTT Payload dati inviato dal server

# Bibliografia

- [1] M. P. Feldman. *The Internet revolution and the geography of innovation*. Vol. 54. International Social Science Journal, mar. 2002, pp. 47–56. URL: <http://onlinelibrary.wiley.com/doi/10.1111/1468-2451.00358/full>.
- [2] A. M. Odlyzko K. G. Coffman. *Internet Growth: Is There a “Moore’s Law” for Data Traffic?* Vol. 4. Springer US, mar. 2002, pp. 47–93. URL: [http://link.springer.com/chapter/10.1007/978-1-4615-0005-6\\_3](http://link.springer.com/chapter/10.1007/978-1-4615-0005-6_3).
- [3] A. Grant. *CPU spin*. Ott. 2015. URL: <https://www.facebook.com/arig/posts/10105815276466163>.
- [4] A. S. Tanenbaum. *Reti di calcolatori*. quarta. Pearson Education IT, 2003, pp. 40–44. ISBN: 9788871921822. URL: <https://books.google.it/books?id=n5m6VFpZYgUC>.
- [5] F. Papale. «Un Meccanismo efficiente per l’implementazione del modello content-based Publish-Subscribe su sistemi topic-based». Tesi di dott. Laurea Magistrale in Ingegneria Informatica: Università degli Studi di Roma, 2009. Cap. 1.
- [6] *RFC1945*. Rapp. tecn. NTWG, mag. 1996. URL: <https://tools.ietf.org/html/rfc1945#section-1.3>.
- [7] *RFC2068*. Rapp. tecn. NTWG, gen. 1997. URL: <https://tools.ietf.org/html/rfc2068#section-8.1>.
- [8] *RFC7230*. Rapp. tecn. IETF, giu. 2014. URL: <https://tools.ietf.org/html/rfc7230#section-6.3.2>.
- [9] *RFC7540*. Rapp. tecn. IETF, mag. 2015. URL: <https://tools.ietf.org/html/rfc7540#section-8.2>.

- [10] *RFC6202*. Rapp. tecn. IETF, apr. 2011. URL: <https://tools.ietf.org/html/rfc6202#section-1>.
- [11] *RFC6455*. Rapp. tecn. IETF, dic. 2011. URL: <https://tools.ietf.org/html/rfc6455#section-1.1>.
- [12] *Server-Sent Events*. Feb. 2015. URL: <https://www.w3.org/TR/eventsource>.
- [13] *RFC7452*. Rapp. tecn. IAB, mar. 2015. URL: <https://tools.ietf.org/html/rfc7452#section-2>.
- [14] M. Boussard M. A. Feki F. Kawsar. *The Internet of Things: The Next Technological Revolution*. Vol. 46. IEEE, feb. 2013, pp. 24–25. URL: <http://ieeexplore.ieee.org/document/6457383/#full-text-section>.
- [15] *MQTT*. URL: <http://mqtt.org/faq>.
- [16] H. L. Truong A. Stanford-Clark. *MQTT For Sensor Networks*. Rapp. tecn. Nov. 2013. URL: [http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN\\_spec\\_v1.2.pdf](http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf).
- [17] M. Risi. «Confronto di tecniche di localizzazione per le WSN basate su RSSI». Tesi di dott. Laurea Triennale in Ingegneria Informatica: Università degli Studi di Pavia, 2009. Cap. 5.
- [18] *RFC5246*. Rapp. tecn. NTWG, ago. 2008. URL: <https://tools.ietf.org/html/rfc5246#section-1>.
- [19] *Supermercato24*. URL: <https://www.supermercato24.it>.
- [20] F. Marino. *Supermercato24, la startup che cresce portando la spesa a casa in un'ora*. Lug. 2016. URL: [http://www.economyup.it/startup/4732\\_supermercato24-la-startup-che-cresce-portando-la-spesa-a-casa-in-un-ora-nonostante-amazon.htm](http://www.economyup.it/startup/4732_supermercato24-la-startup-che-cresce-portando-la-spesa-a-casa-in-un-ora-nonostante-amazon.htm).
- [21] S. Biagio. *Supermercato24, 3 milioni di euro per la startup della spesa online*. Lug. 2016. URL: <http://www.ilsole24ore.com/art/tecnologie/2016-07-18/supermercato24-3-milioni-euro-la-startup-spesa-online-122342.shtml>.

- [22] A. Nisi. *Il frigorifero intelligente che ordina la spesa. Accordo Samsung – Supermercato24*. Set. 2016. URL: <http://thefoodmakers.startupitalia.eu/56627-20160908-samsung-supermercato24-e-commerce>.
- [23] A. Thomas. «Architectural Styles and the Design of Network-based Software Architectures». Tesi di dott. Information e Computer Science: University of California, 2000. Cap. 5. URL: [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm).
- [24] *Building Facebook Messenger*. Ago. 2011. URL: <https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920>.
- [25] *IBM bluemix*. URL: <https://www.ibm.com/developerworks/cloud/library/cl-mqtt-bluemix-iot-node-red-app>.
- [26] *AWS IoT – Cloud Services for Connected Devices*. Ott. 2015. URL: <https://aws.amazon.com/it/blogs/aws/aws-iot-cloud-services-for-connected-devices>.
- [27] R. Patton. *How Google will use Firebase to supercharge its cloud computing*. Nov. 2014. URL: <https://www.wired.com/2014/11/why-google-acquired-firebase>.
- [28] T. Iaconis K. Bird. *Instacart: Competitive Analysis and Potential Product Roadmap*. Apr. 2015. URL: <http://launchingtechventures.blogspot.it/2015/04/instacart-competitive-analysis-and.html>.
- [29] N. Stephend. *Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android*. Mag. 2012. URL: <http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https>.
- [30] *Adb*. URL: <https://developer.android.com/studio/profile/battery-historian.html>.
- [31] *Battery Historian*. URL: <https://github.com/google/battery-historian>.