
UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA


# Architetture Cloud a Supporto della Comunicazione per Scenari di Internet of Things

**Cloud Architectures for Communication Support in Internet of Things Scenarios**


**Relatore:**

**Chiar.mo Prof. Marco Picone**

**Correlatore:**

**Dott. Ing. Simone Cirani**

**Tesi di Laurea di:**

**MIRCO ROSA**



**ANNO ACCADEMICO 2015/2016**

A mio nonno Maurizio

# Ringraziamenti

Per prima cosa vorrei ringraziare il Prof. Marco Picone, il Dott. Ing. Simone Cirani e l'Ing. Luca Davoli, che hanno reso possibile la realizzazione del progetto e la stesura di questa tesi dimostrando sempre disponibilità, cortesia ed interesse.

Il grazie più grande va ai miei genitori, Franco e Mina, che mi sono sempre stati vicini supportandomi nei molti momenti difficili attraversati in questi anni e dandomi la possibilità di seguire le mie aspirazioni.

Un grazie particolare alle mie nonne Luisa e Giulia, che sono sempre state presenti e che mi hanno aiutato durante tutto questo percorso.

Ringrazio tutti i familiari vicini e lontani, che hanno sempre avuto un pensiero per me dimostrandomi simpatia ed affetto.

Infine ringrazio tutti i miei amici, in particolare Nicolò e la Compagnia, per tutti i bei momenti passati assieme e per aver reso speciale questa esperienza.

A tutti, grazie.

# Indice

# Introduzione

Fra tutte le evoluzioni che stanno interessando il mondo dell'Informatica in questi tempi quella relativa all'Internet of Things è sicuramente una delle più importanti e profonde: promette di modificare drasticamente la società ed il modo in cui viviamo, introducendo metodi di interazione e scambio di informazioni fra persone e cose mai visti prima; indirizzare lo sforzo della ricerca in questo ambito è quindi particolarmente auspicabile e l'esponenziale progresso tecnologico registrato negli ultimi anni rende facilmente accessibili strumenti potenti adatti alla causa, sia in termini hardware che software. Per questi motivi ci si è posti come obiettivo lo sviluppo di un'architettura software completa in grado di rendere fruibili informazioni di reti IoT locali ad utenti remoti connessi ad Internet, utilizzando protocolli prettamente dedicati all'Internet delle Cose e sfruttando le moderne tecnologie offerte dal Cloud in ambito di Computing, Storage e Messaging. Tra i molteplici fattori da tenere in considerazione in fase di progettazione abbiamo le prestazioni: l'obiettivo non sarà quindi soltanto sviluppare un sistema efficace e funzionalmente completo, ma operare scelte architetturali che permettano il mantenimento delle performance in scenari caratterizzati da grandi quantità di nodi e client. La tesi è suddivisa in quattro capitoli: il primo riguarda l'attuale stato dell'arte

in materia di Internet of Things e servizi Cloud, e descrive le tecnologie che riguarderanno più da vicino le tematiche affrontate nella tesi; il secondo offre una panoramica generale dell'Architettura, spiegando le funzioni dei vari componenti ed illustrando le funzionalità implementante. Il capitolo inerente la Google Cloud Platform tratta nel dettaglio i servizi Cloud che andremo ad utilizzare, esplorandone le funzionalità e motivando le scelte operate durante la progettazione; il quarto capitolo parla invece dell'implementazione, fornendo un'analisi minuziosa delle scelte operative fatte e di come sono stati trasposti nella pratica i concetti espressi nei capitoli precedenti. Infine nelle Conclusioni è riportato uno studio per quanto riguarda le prestazioni, e vengono presi in considerazione gli sviluppi futuri che potranno interessare il progetto.

# Capitolo 1

# Stato dell'Arte

## 1.1 Internet of Things

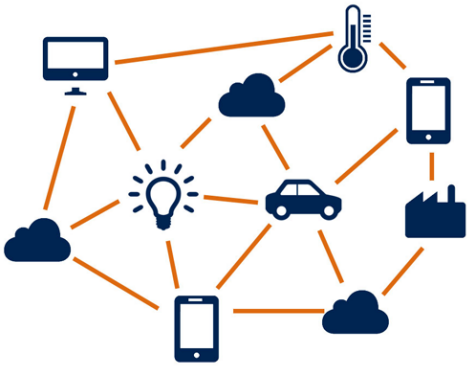

Figura 1.1: Scenario IoT

L'Internet of Things (Internet delle Cose) è una delle evoluzioni più importanti che sta interessando la Rete negli ultimi tempi ed è sicuramente quella che più promette di entrare con prepotenza nelle nostre vite: le "cose", come per esempio gli oggetti con cui abbiamo a che fare quotidianamente, da elementi passivi diventano intelligenti ("smart") cioè capaci di raccogliere ed elaborare informazioni sull'ambiente che li circonda, trasmetterle nella Rete ed aggregarle con quelle di altri Smart Objects per poi produrre l'output desiderato. Le applicazioni per questo tipo di interazione sono praticamente illimitate: domotica, robotica, biomedicina, industria automobilistica sono solo alcuni dei settori che

sfruttando l'Internet of Things potranno evolversi in direzioni fino a qualche anno fa impensabili, portando ad un livello superiore il concetto di comunicazione M2M (Machine to Machine) e M2H (Machine to Human). Gran parte del merito va attribuito al progresso tecnologico e all'imprinting dato alla ricerca negli ultimi decenni: dispositivi sempre più piccoli, potenti e a basso consumo energetico, protocolli e architetture di rete progettati per consentire ad un numero sempre maggiore di dispositivi di comunicare fra loro in modo immediato ed efficiente sono i principali fattori che permetteranno nei prossimi anni l'interconnessione intelligente di decine di miliardi di dispositivi in tutto il mondo. Una evoluzione così rapida e su larga scala non può però non destare perplessità su alcune tematiche critiche della gestione delle informazioni, quali sicurezza e privacy: essere costantemente connessi ed avere una disponibilità immediata delle informazioni è sicuramente un'ottima cosa, ma bisogna sempre tener presente che si ha a che fare con dati reali di persone reali e che lo sviluppo "prestazionale" deve necessariamente andare di pari passo con la tutela dei soggetti che andranno a fare uso della tecnologia. Fatte le dovute premesse, passiamo ora in rassegna i principali protocolli utilizzati dall'IoT.

### 1.1.1 Protocolli di Comunicazione

Un protocollo di comunicazione è un insieme di regole formalmente descritte, definite al fine di favorire la comunicazione tra una o più entità. I protocolli sono gestiti da organismi internazionali quali il World Wide Web Consortium (W3C), che hanno appunto il compito di fissare delle regole universali ed

evitare una diffusione di standard disomogenea e controproducente per lo sviluppo delle applicazioni.

### CoAP

CoAP (Constrained Application Protocol) è un protocollo applicativo sviluppato appositamente per consentire una gestione efficace ed efficiente di reti formate da nodi con capacità di calcolo e potenza energetica estremamente limitate; lo standard, la cui prima definizione risale al 2009, è stato oggetto negli ultimi anni di numerosissime migliorie ed espansioni che lo hanno portato ad essere uno dei punti di riferimento principali nell'evoluzione del settore dell'IoT. Le caratteristiche che rendono CoAP il protocollo ideale per l'Internet of Things sono molteplici:

- Modello REST (REpresentational State Transfer): è un modello client-server stateless a livelli che ha come principio fondamentale l'identificazione delle risorse tramite URI; essendo utilizzato anche da HTTP è possibile creare applicazioni in grado di interfacciarsi naturalmente con le strutture esistenti del World Wide Web.

- Ottimizzazione per grandi numeri di nodi: la struttura del protocollo è stata appositamente studiata per coordinare l'attività di un numero molto alto di nodi contemporaneamente, pur tenendo sempre conto delle limitate risorse hardware a disposizione.

- Facile integrazione con i sistemi esistenti: come anticipato in precedenza, CoAP condivide con HTTP la struttura REST e ciò rende possibile creare applicazioni cross-protocol con facilità.

- Discovery integrata: CoAP implementa nativamente un sistema di discovery, che permette di ottenere facilmente informazioni su nodi e servizi presenti nella stessa rete.

- Sicurezza: in scenario IoT la sicurezza è un fattore fondamentale, e CoAP implementa nel DTLS crittografia equivalente a quella di chiavi RSA a 3072-bit.

**6LoWPAN**

Il protocollo di rete IPv6 consente di gestire fino a $2^{128}$ indirizzi IP diversi, un numero estremamente superiore a quello del predecessore IPv4 capace di controllarne "solo" $2^{32}$: la necessità di un estensione del protocollo è stata dettata, come è facile intuire, dalla crescita esponenziale del numero di dispositivi che ci si aspetta di avere connessi alla rete nei prossimi anni. 6LoWPAN, acronimo di IPv6 over Low power Wireless Personal Area Networks, è una evoluzione del protocollo IPv6 studiata appositamente per consentire la comunicazione tramite protocollo IP a dispositivi dotati di capacità di calcolo e consumo di potenza estremamente ridotti, quali appunto gli Smart Objects. Le sue principali caratteristiche sono:

- Dimensione più grande dei pacchetti di trasmissione al fine di ridurre l'overhead

- Risoluzione gerarchica degli indirizzi

- Adaption Layer per garantire una maggior flessibilità sui diversi tipi di pacchetto

- Futuro supporto a Device Discovery e Service Discovery

**mDNS e DNS-SD**

mDNS, acronimo di multicast Domain Name System, è un sistema che si occupa della risoluzione dei nomi dei nodi in indirizzi IP e viceversa in reti che non sono dotate di un name server. E' un servizio zero-configuration, ovvero non richiede intervento umano o particolari configurazioni manuali, ed utilizza le stesse interfacce e gli stessi formati dei pacchetti utilizzati dai DNS classici; questo sistema è particolarmente interessante se coadiuvato dalla DNS-SD (DNS Service Discovery) che permette la creazione e il mantenimento automatico di reti dinamiche dove i nodi possono entrare ed uscire liberamente, con la possibilità di pubblicizzare servizi agli altri nodi presenti.

## 1.2 Piattaforme di Cloud Messaging

Per Cloud Messaging si intendono tutte quelle tecnologie e servizi che il Cloud offre per consentire lo scambio di messaggi fra diversi soggetti all'interno di una rete. Passiamo in rassegna le principali piattaforme di Cloud Messaging disponibili ad oggi, tenendo però presente che essendo un mercato in continua evoluzione in brevissimo tempo potrebbero esserci variazioni o nuovi competitor in grado di offrire servizi simili.

### 1.2.1 Google Cloud Messaging - Firebase

Google, tra le prime aziende ad offrire servizi di Cloud Storage e Cloud Computing agli sviluppatori, non poteva astenersi dall'avere fra le molteplici soluzioni offerte dalla Google Cloud Platform una soluzione per il Cloud Messaging. Google Cloud Messaging è una API molto potente e flessibile che permette di sviluppare facilmente applicazioni multipiattaforma: è perfettamente integrata con tutta la suite di servizi della Platform, e supporta la gestione di utenti lato Cloud (coadiuvata dalla Datastore API) consentendo il broadcast selettivo dei messaggi; per il backend sono utilizzabili moltissimi linguaggi di programmazione, mentre per il frontend sono disponibili API per Android (che supporta nativamente il GCM nei Play Services), iOS e WebApp Chrome. Un altro aspetto interessante del GCM riguarda il lato economico: non esiste un limite massimo di messaggi che possono essere inviati o ricevuti, anche se si dispone di un semplice account free; un fattore importante che stimola ancor di più l'utilizzo di questo sistema per la creazione di applicazioni cloud-oriented. Recentemente Google ha acquisito Firebase, piattaforma con caratteristiche simili al GCM che offre un servizio di Cloud Messaging ancora più flessibile e di facile implementazione che può comunque essere interfacciato con gli altri prodotti della Cloud Platform.

### 1.2.2 Apple Push Notification Service

Anche Apple fornisce ai propri sviluppatori un sistema di Cloud Messaging, l'Apple Push Notification Service: valido e semplice per quanto riguarda la comunicazione device-to-device ma restrittivo quando si tratta di voler

sviluppare un proprio backend custom. La politica Apple richiede una particolare certificazione per avere l'autorizzazione a creare e mettere in piedi il proprio backend di Cloud Messaging, cosa che spinge molto spesso gli sviluppatori ad affidarsi a servizi di terze parti (che forniscono API proprietarie e funzionanti solo sui loro APNs) o addirittura ad appoggiarsi al Google Cloud Messaging, che come detto in precedenza fornisce strumenti per il frontend anche per iOS.

### 1.2.3 Amazon Simple Notification Service

Anche Amazon negli ultimi anni ha impattato il mercato dei Web Services, ottenendo buoni risultati e ritagliandosi una considerevole fetta di mercato. L'offerta del colosso statunitense è l'Amazon Simple Notification Service (ASNs), che consente di inviare messaggi praticamente su qualsiasi piattaforma (anche Windows e FireOS), supporta l'invio di SMS, mette a disposizione API in molti linguaggi di programmazione e fornisce strumenti utili per monitorare il recapito dei messaggi.

## 1.3 Cloud Messaging per Internet of Things

Le piattaforme di Cloud Messaging sono progettate appositamente per inviare e ricevere un numero molto elevato di messaggi dalle dimensioni ridotte: questa caratteristica è perfetta per supportare le comunicazioni in scenari IoT, che tipicamente includono dispositivi always-on con un output di informazioni continuo (ad esempio sensori). Molto spesso cercare di connettere direttamente gli Smart Objects con il Cloud non è la soluzione migliore da

adottare: vincoli come capacità di calcolo ridotta e il dispendio energetico che può comportare un modulo di connettività alla rete cellulare mobile per l'accesso a internet ci spingono a cercare alternative in fase di progettazione.

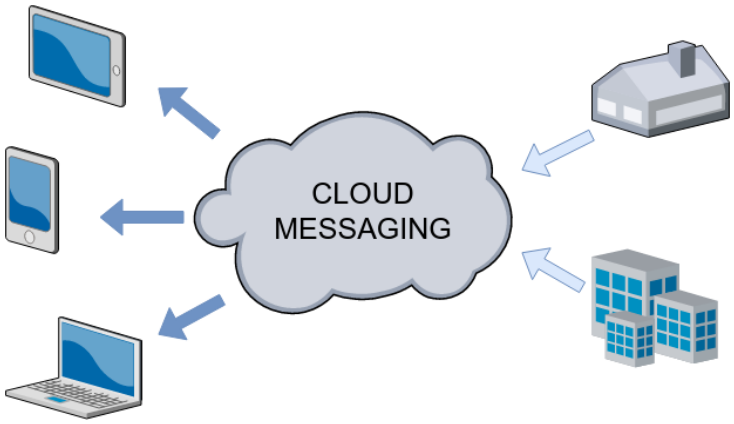


Figura 1.2: Cloud Messaging e IoT

Per questi motivi è conveniente valutare l'adozione di un livello aggiuntivo nell'architettura di rete: un nodo intermedio, con caratteristiche hardware e software adeguate, che sia in grado di gestire sia gli Smart Object entro il suo raggio di azione (per esempio all'interno di una rete locale) sia le comunicazioni con il Cloud. Questa soluzione è ottimale e ci permette inoltre di avere un maggior controllo sulle informazioni raccolte, consentendoci di effettuare una prima elaborazione in modo da fornire alla parte Cloud dei dati già preprocessati, rimuovendo anche eventuali problemi legati all'eterogeneità dei dispositivi utilizzati nella LAN. A seconda degli scenari sarà possibile (e consigliabile) aggiungere ulteriori livelli, in modo da avere una struttura modulare che garantisca una ripartizione del carico di lavoro bilanciata e intelligente. Infine l'infrastruttura Cloud dovrà essere progettata in modo da poter reindirizzare le informazioni ottenute, dopo averle eventualmente

elaborate o immagazzinate, utilizzando il Cloud Messaging verso i client che ne faranno richiesta; anche qui sarà necessario valutare le specifiche di progetto a seconda delle esigenze e soprattutto delle disponibilità economiche: il Cloud tipicamente è in grado di soddisfare qualsiasi requisito in termini capacità di calcolo e memoria, a patto di poter sopportare un sacrificio monetario corrispondente.

# Capitolo 2

# Architettura

In questo capitolo daremo una visione d'insieme dell'intera architettura e ci soffermeremo sulle funzionalità che questa ha da offrire.

## 2.1 Struttura

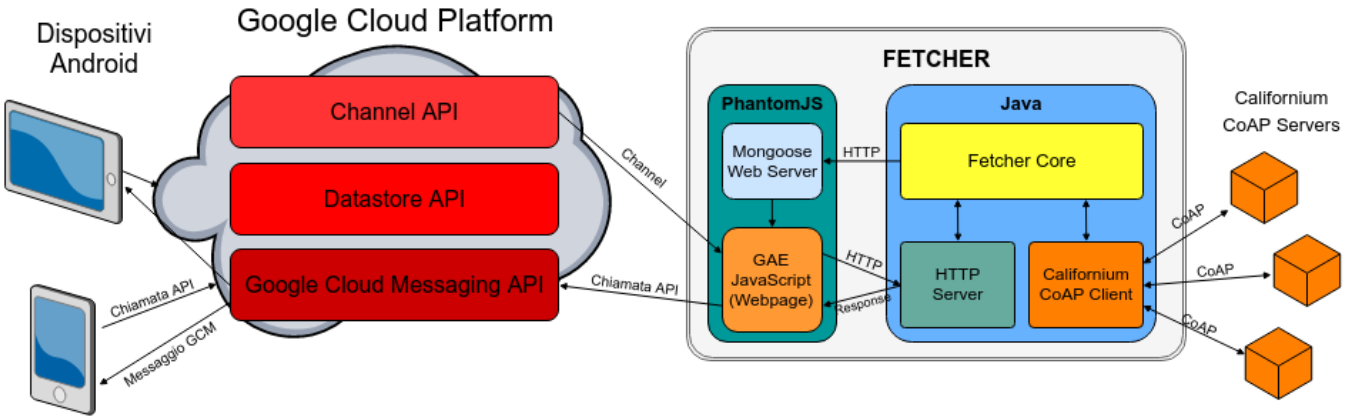


Figura 2.1: Architettura Completa

Nel diagramma in figura è rappresentata l'intera architettura sviluppata. Possiamo vedere che, in accordo con i concetti espressi nel capitolo prece-

dente, abbiamo un nodo intermedio (il Fetcher) che si occupa di gestire sia la rete IoT, utilizzando il framework Californium e la libreria JmDNS che vedremo tra poco, sia le comunicazioni con il Cloud avvalendosi dell'headless browser PhantomJS di cui daremo una descrizione più avanti. Vi è poi la parte del Google App Engine, che provvede a memorizzare ed elaborare le informazioni provenienti dal Fetcher, inviare notifiche ai client Android tramite Cloud Messaging e reindirizzare le richieste che gli stessi Client indirizzano alle risorse IoT.

## 2.2 Funzionalità

Vediamo ora quali sono le funzionalità implementate ed i loro meccanismi.

### 2.2.1 GET e POST

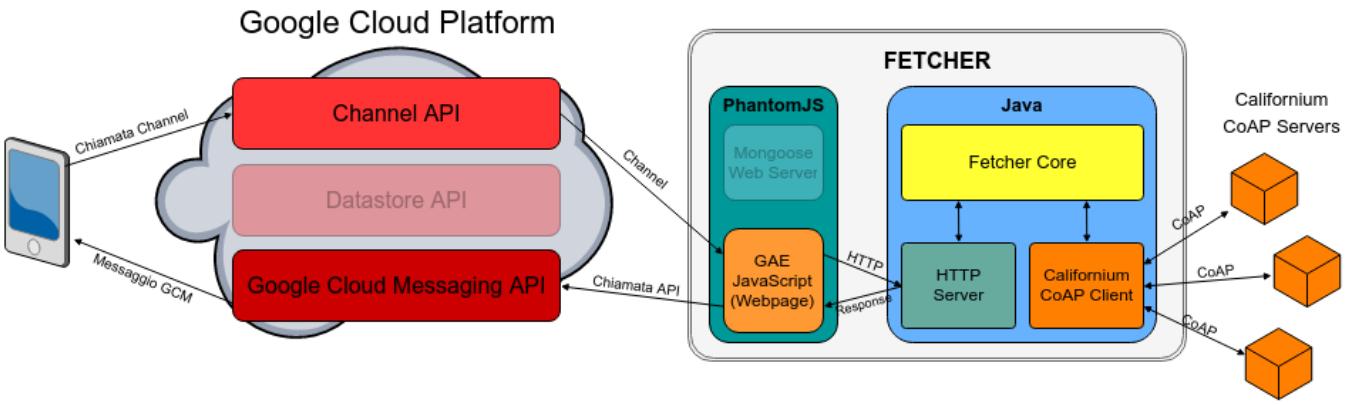


Figura 2.2: Percorso delle richieste GET e POST

I client che vogliono effettuare richieste ad una risorsa specifica nello scenario IoT devono prima di tutto rivolgersi al Cloud Engine, che si occuperà

di reindirizzarle Fetcher; a questo punto attraverso il Client Californium viene eseguita la corrispondente richiesta CoAP alla risorsa indicata in modo da ottenere il valore desiderato ed includerlo nel payload della risposta, che compierà il percorso a ritroso chiudendo tutte le operazioni aperte rimaste pendenti.

### 2.2.2 Observing

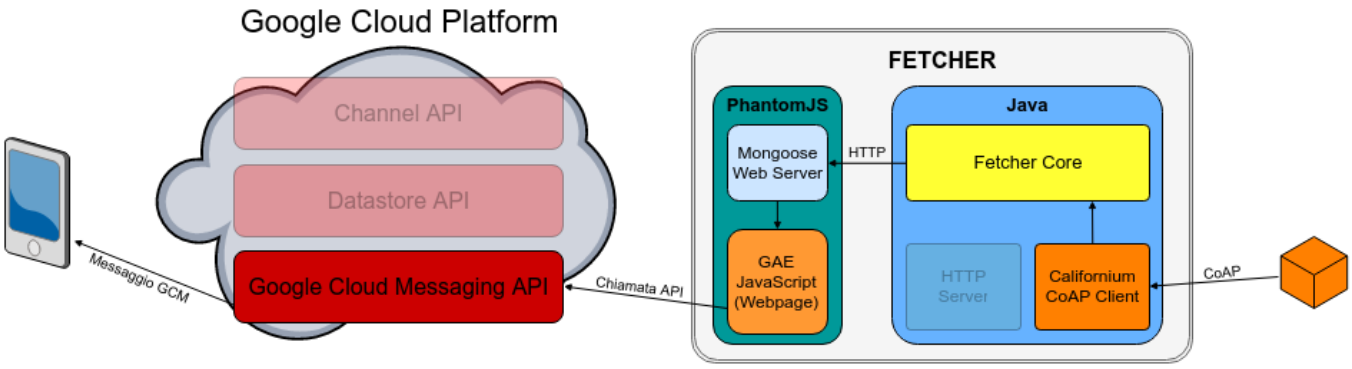


Figura 2.3: Percorso degli update derivanti da relazioni di Observing

La procedura di observing è molto simile a quella per l'esecuzione di una GET (che di fatto viene eseguita al momento della creazione della relazione), con la differenza che l'invio degli update futuri derivanti dalla risorsa verrà gestito da PhantomJS.

### 2.2.3 Sincronizzazione dello Scenario

I server CoAP pubblicizzano la loro presenza nella rete IoT e vengono identificati dalla Service Discovery del Fetcher, che ne memorizza struttura e caratteristiche: in questo modo si ha sempre una visione real-time della rete,

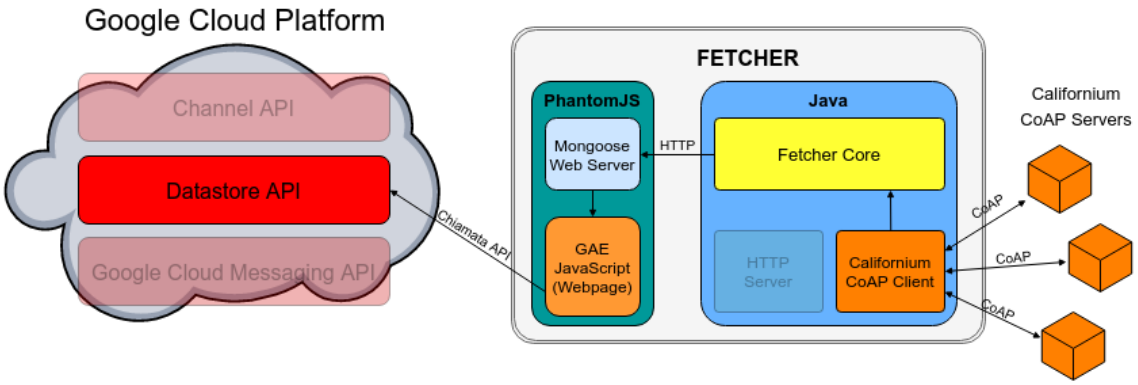


Figura 2.4: Sincronizzazione Fetcher-Cloud

i cui nodi possono variare dinamicamente nel tempo. Lo scenario locale è costantemente sincronizzato con il Cloud, che memorizza lo stato attuale sul Google Cloud Datastore rendendolo disponibile a qualsiasi client Android ne faccia richiesta.

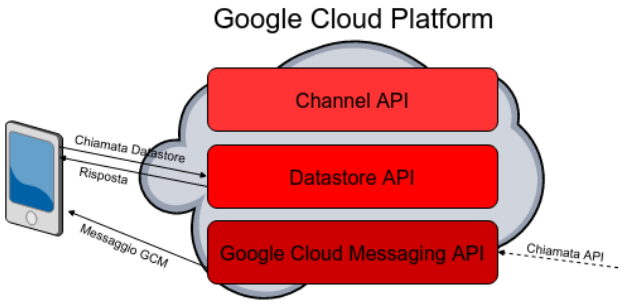


Figura 2.5: Sincronizzazione Android-Cloud

In questo caso il Cloud Messaging non viene utilizzato per trasportare i dati veri e propri, ma per notificare all'utente che è intervenuta una modifica e permettergli di effettuare una GET "mirata" al Datastore.

# Capitolo 3

# Google Cloud Platform

Google Cloud Platform come anticipato nei paragrafi precedenti è l'infrastruttura Cloud offerta da Google ed include servizi di ogni genere: computing, storage, big data, networking, data analysis, machine learning e molti altri uniti in un unico ecosistema e completamente compatibili fra loro. Ovviamente in questa tesi ne verranno utilizzati solamente alcuni, che saranno comunque sufficienti a creare una infrastruttura Cloud completamente funzionante che sia di supporto in uno scenario IoT.

## 3.1 Google Cloud Endpoints

E' la base su cui andremo a costruire la nostra architettura Cloud: Google ha concepito gli Endpoints come "unità operative" all'interno delle quali è possibile scrivere le proprie APIs scegliendo fra i molteplici linguaggi messi a disposizione e utilizzando a nostro piacimento tutti i servizi che la Cloud Platform ha da offrirci. La caratteristica fondamentale degli Endpoints è

però un'altra: sono studiati appositamente per interagire con client mobili, come per esempio app Android, iOS o WebApp, e la Platform scalerà automaticamente l'architettura in base al carico di lavoro senza la necessità di interventi manuali. Nel caso particolare di Android, che sarà quello preso in esame, le cose sono ancora più semplici: sarà sufficiente scrivere il codice delle nostre APIs in Android Studio (l'IDE fornito da Google per sviluppare applicazioni Android), e le librerie necessarie ad interfacciare l'app client con gli Endpoint nel cloud saranno compilate automaticamente al momento del deploy del modulo nella Platform. Oltre a queste principali funzionalità sono disponibili altri servizi complementari quali autenticazione degli utenti e delle singole chiamate alle APIs nonché strumenti di logging avanzati per monitorare il funzionamento e le prestazioni della nostra architettura.

### 3.1.1 Google Cloud Messaging

Come già detto nel capitolo precedente, Google Cloud Messaging è uno strumento potente e flessibile, con interessanti funzionalità di grouping dei messaggi, autenticazione degli utenti e supporto lato client per le maggiori piattaforme esistenti (Android, iOS e WebApp Chrome). Nel caso specifico di Android l'integrazione è totale in quanto i Google Play Services (presenti su tutti i dispositivi) supportano nativamente il GCM, e la compilazione automatica delle librerie fornita da Android Studio rende davvero immediato creare semplici sistemi funzionalmente completi lasciando allo sviluppatore il compito di dedicarsi principalmente della logica del programma, prevenendo tediosi problemi legati alla compatibilità.

```
Sender sender = new Sender(API_KEY);
List<RegistrationRecord> records =
   ofy().load().type(RegistrationRecord.class)
      .limit(10).list();
for (RegistrationRecord record : records) {
   Result result = sender.send(msg,
      record.getRegId(), 5);
   ...
}
```

### 3.1.2 Google Cloud Datastore

Il Google Cloud Datastore è un noSQL database offerto dalla piattaforma Google per lo storage dei dati. Come detto è un database non relazionale, ovvero un sistema di archiviazione che non impone uno schema fisso da seguire; questa caratteristica consente una scalabilità ed efficienza molto maggiore rispetto ai tradizionali database relazionali e permette ai servizi Cloud di distribuire e sincronizzare al meglio le informazioni sui vari Data Center, garantendo tempi di accesso bassissimi. Il risvolto della medaglia è che sarà lo sviluppatore a doversi occupare di consistenza e struttura dei dati, ma in scenari come il nostro nei quali non avremo a che fare con schemi fissi ed eccessivamente complessi è un compromesso più che accettabile. Le unità di dati sono chiamate Entità, possono essere organizzate in strutture ad albero e sono caratterizzate da:

- Tipo: il tipo di Entità, utilizzato spesso come parametro per le query.
- Proprietà: campi all'interno dei quali vengono memorizzate le informazioni, possono essere di vari tipi (stringa, intero, data, ecc) e una entità può averne un numero arbitrario. Come dicevamo prima può es-

sere importante, a seconda delle esigenze, impostare una struttura delle proprietà in modo da avere consistenza nel momento in cui si vada ad effettuare query sui dati.

- Identificatore: campo che identifica l'entità, può essere di tipo intero o stringa.

- Ancestor Path: tradotto letteralmente "percorso antenato" indica quali sono le entità antenate, ovvero i rami "superiori" dell'entità corrente.

- Chiave: la chiave dell'entità è la combinazione di Tipo, Identificatore e Ancestor Path. Questa composizione rende possibile avere entità con Nome e Tipo uguali in diverse parti della struttura, cosa che può tornare utile in particolari casistiche.

Entità (o gruppi di entità) possono essere aggiunti, rimossi, aggiornati o recuperati in modo del tutto similare ai database classici, ovvero mediante l'utilizzo di transazioni, sequenza di operazioni che soddisfa le proprietà ACID, e query, che devono necessariamente essere effettuate all'interno di uno scope ben preciso (come per esempio una entità padre).

```
Transaction transaction =
   datastore.beginTransaction();
...
datastore.put(server);
...
Query serverQuery = new
   Query("Server").setAncestor(scenario.getKey())
.setFilter(new
   FilterPredicate(Entity.KEY_RESERVED_PROPERTY,
   FilterOperator.EQUAL,
   KeyFactory.createKey(scenario.getKey(),
   "Server", serverName)));
```

```
...
ransaction.commit();
```

### 3.1.3 Google App Engine: Channel API

La Channel API è uno strumento ad alte prestazioni offerto dal Google App Engine per creare una connessione diretta e persistente (tramite web socket) fra il Cloud e dei client tipicamente scritti in JavaScript. La procedura per aprire il canale è piuttosto semplice:

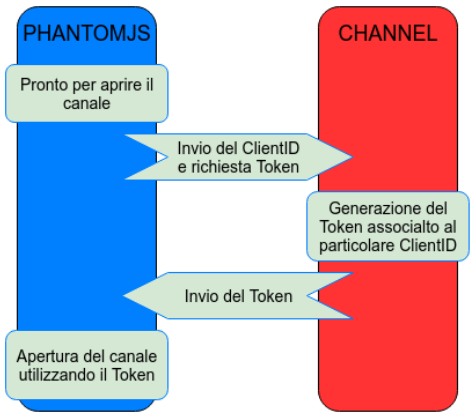

Figura 3.1: Creazione del Channel

- Il client richiede al Cloud l'apertura del canale inviando il proprio ClientID;
- La Channel API genera un token di identificazione univoca del canale (legato al ClientID ricevuto), e lo manda al client;
- Il client utilizza il token per aprire il canale e restare in ascolto.

```
public MyToken createChannel() {
    ChannelService channelService =
       ChannelServiceFactory.getChannelService();
       String channelKey = "TestChannel";
       String channelToken =
          channelService.createChannel(channelKey);
       MyToken token = new MyToken();
       token.setMyToken(channelToken);
       return token;
}
```

Come è facile intuire dalla procedura di creazione abbiamo a che fare con un canale unidirezionale, ovvero le informazioni possono essere unicamente inviate dal Cloud al client e non viceversa. I punti di forza di questo sistema sono principalmente due: il primo è l'assenza di polling, che riduce al minimo i tempi di attesa in caso di richieste; il secondo è che l'utilizzo dei Channel consente una comunicazione diretta fra Cloud e dispositivi che non hanno un indirizzo IP pubblico fisso: questo è un fattore fondamentale negli scenari IoT, dove si può avere a che fare per esempio con ambiti domestici dove tipicamente l'ISP assegna gli indirizzi IP in modo dinamico. Per completezza riportiamo anche il codice relativo alla richiesta di apertura del Channel da parte del client:

```
function createChannel() {
    gapi.client.channel.channelEndpoint
       .createChannel().execute(
      function(response) {
         if(!response.code) {
            console.log("Token: "+response.token);
            channel = new goog.appengine
               .Channel(response.token);
            socket = channel.open();
            ...
         }
      }
      );
}
```

# Capitolo 4

# Implementazione

In questo capitolo tratteremo dettagliatamente l'implementazione di tutte le componenti dell'architettura, analizzando le scelte operate durante lo sviluppo vero e proprio.

## 4.1 Scenario IoT

In questa sezione parleremo approfonditamente della parte relativa allo scenario IoT, analizzandone il codice e motivando le scelte operate in fase di progettazione.

### 4.1.1 Simulatore, server e risorse

Una delle caratteristiche principali delle reti IoT è l'elevato numero di nodi, e creare un ambiente fisico di testing per il nostro sistema avrebbe richiesto una notevole quantità di risorse sia in termini economici che di tempo: per questo motivo è stato scelto di sviluppare un simulatore software in grado di generare

una situazione che possa rispecchiare una ipotetica casistica reale. Prima però è necessaria una breve introduzione di alcuni strumenti fondamentali che saranno alla base della nostra architettura.

### Californium

Californium è un framework CoAP per lo sviluppo di applicazioni Java; il progetto è diviso a sua volta in 5 sotto-progetti e nella tesi verrà utilizzato quello principale, ossia il Californium Core. La piattaforma è completamente open source, il che ci ha permesso di apportare modifiche marginali al codice sorgente per soddisfare alcune esigenze progettuali di cui parleremo più avanti. Dopo un uso estensivo si può affermare che Californium è un framework solido e completo, con funzionalità interessanti quali:

- Creazione di client e server semplice ed estensibile, perfettamente integrata con l'OOP di Java.
- Strumenti completi per l'invio e la ricezione di richieste CoAP (token per i messaggi, helper per la gestione di URI e query string, ecc.).
- Supporto integrato per strutture ad albero di risorse da parte dei Server, con gestione ed aggiornamento automatico delle URI.
- Resource Discovery disponibile e abilitata di default nei server.

### PhantomJS - Mongoose Web Server

PhantomJS è un headless browser, ossia un software che ci permette di scrivere programmi JavaScript in grado di aprire, processare ed interagire con

pagine web senza necessariamente renderizzarne il contenuto a video. Offre inoltre la possibilità di integrare un semplice Web Server (Moongose) per consentire allo script di interagire con l'esterno tramite protocollo HTTP.

### JmDNS

JmDNS è una implementazione Java di mDNS (Vedi paragrafo 2.2.1) che può essere usata per registrazione e discovery di servizi all'interno di una rete locale. Offre la possibilità di configurare parametri di rete quali protocolli IP e porte di comunicazione predefinite per i vari servizi, oltre a supportare la definizione di domini e sottodomini per soddisfare particolari requisiti di progetto; non prevede invece la gestione automatica dei conflitti per quanto riguarda i nomi dei servizi, che deve essere implementata dallo sviluppatore in modo manuale.

### Risorse

Californium come detto in precedenza è in grado di gestire internamente alberi di risorse, ma se vogliamo aggiungere funzionalità accessorie alle singole componenti dobbiamo occuparci in prima persona di come la struttura del server viene coordinata. Per questo motivo si è deciso di estendere la classe CoapResource di Californium (che fornisce tutte le funzionalità CoAP di cui abbiamo bisogno) creando la classe astratta GenericCoapResource, i cui tratti salienti sono riportati di seguito:

```
public abstract class GenericCoapResource
   extends CoapResource{
```

```
protected
   GenericCoapResource(/*Parameters...*/) {
   super(name,true);
   this.server = server;
   setObservable(observable);
   setObserveType(Type.CON);
   for(String type: types)
      getAttributes().addResourceType(type);
      getAttributes().setTitle(name);
      if(observable)
         getAttributes().setObservable();
   }
   ...
   public boolean
      addChildRes(/*Parameters...*/) {
   GenericCoapResource resource = new
      ResourceFactory()
      .getResource(name,getURI(),type,
         server);
   ...
   if(!silent)
      server.notifyWellKnown();
   resource.setResParent(this);
   return resChildren.add(resource);
   }
}
```

Vengono memorizzate la risorsa genitore (null nel caso ci si trovi al primo livello) e le risorse figlie utilizzando metodi scritti ad hoc per l'aggiunta e la rimozione, ed è stato implementato un meccanismo di controllo del nome che impedisce di avere sullo stesso livello risorse omonime. E' stato inoltre necessario ottenere la URI della risorsa sotto forma di parametro nel costruttore, in quanto al momento della creazione dell'oggetto Californium non è ancora in grado di assegnarne uno. Infine è importante notare come nel momento in cui viene aggiunta o rimossa una risorsa venga notificato il well-known core

del server: questo passaggio si rende necessario in seguito a una modifica effettuata internamente sul framework, di cui parleremo della sezione dedicata.

La classe astratta appena descritta include le funzionalità comuni a tutte le risorse e può essere quindi estesa a nostro piacimento per creare classi (istanziabili dal simulatore) dotate di behaviour personalizzati. Per esempio:

```
public class TemperatureSensor extends
   GenericCoapResource {
   public TemperatureSensor(/*Parameters...*/) {
      super(/*Parameters...*/);
      Timer timer = new Timer();
      timer.schedule(new BehaviourTask(),
         MyPrefs.TEMP_INITIAL_DELAY,
         MyPrefs.TEMP_UPDATE_FREQUENCY);
   }
   //Data changing simulation
   private class BehaviourTask extends TimerTask {
      @Override
      public void run() {
         //Behaviour logic
      }
   }
   //It's a sensor, no POST implementation
   @Override
   public void handleGET(CoapExchange exchange) {
      ...
   }

}
```

Questa classe simula un sensore di temperatura: il codice è estremamente pulito, il focus è sul concentrato sul comportamento del sensore ed è possibile specificare come la risorsa deve comportarsi in caso di ricezione di una richiesta da parte del Fetcher.

### Server

Per rappresentare i server abbiamo adottato un metodo simile a quello precedente, estendendo la classe CoapServer offerta da Californium e creando la classe astratta GenericCoapServer, che implementa i metodi riguardanti la Service Discovery di JmDNS e la gestione avanzata delle risorse:

```
public abstract class GenericCoapServer extends
   CoapServer {
   protected GenericCoapServer(/*Parameters...*/)
      throws IOException {
      super(port);
      for(ResType resType :
         serverType.getResources()) {
         String resName = autoResName(resType);
         addResource(resName,resType);
      }
      ...
      this.start();
      register();
   }
   ...
   public void stopServer() throws IOException {
      unregister();
      this.stop();
   ...
   }
   public void unregister() throdebbaws IOException
      {
      regManager.unregisterService(name+":"+port);
   }
}
```

Anche qui sono stati implementati controlli del nome delle risorse in modo da non avere casi di omonimia.

La cosa più importante è però la presenza di metodi che consentono di intera-

gire con il sistema di Service Discovery richiamando le funzioni corrispondenti dell'oggetto ServerRegManager, riportato qui di seguito:

```
public class ServerRegManager {
...
    public void
       registerOnFirstNetworkInterface(/*Parameters...*/)
       {
    ServiceInfo service =
       ServiceInfo.create(/*Parameters...*/);
    jmdns.registerService(service);
}
```

Il ServerRegManager si occupa di registrare o deregistrare servizi sulla rete seguendo i parametri impostati per quanto riguarda porte, protocolli e domini; in aggiunta a ciò è stato necessario implementare un metodo di controllo dei nomi, in quanto non possono coesistere servizi omonimi nello stesso dominio.
Ancora una volta, è possibile estendere la classe astratta per creare server con le caratteristiche che più ci aggradano, massimizzando così il riutilizzo di codice.

```
public class Thermometer extends GenericCoapServer {
    ...
}
```

### Simulatore

Il simulatore è l'interfaccia di collegamento con l'utente, che ci consente di utilizzare le classi descritte sopra. Per la creazione degli oggetti è stato adottato il factory pattern, che per le risorse è implementato come segue:

```
public class ResourceFactory {
```

```
    public GenericCoapResource
       getResource(/*Parameters...*/) {
       switch (type) {
          case TEMPERATURE_SENSOR:
          return new
             TemperatureSensor(/*Parameters...*/);
          ...
       }
    }
}
```

E' stato utilizzato un Enum per definire i vari tipi di risorsa, e vengono passati come parametri l'URI del genitore, il server di appartenenza e se la risorsa è osservabile o meno; per quanto riguarda i server il metodo adottato è del tutto analogo. Per motivi di spazio non riporteremo qui la logica del simulatore vero e proprio (oltre 530 linee di codice), ma le sue funzionalità sono:

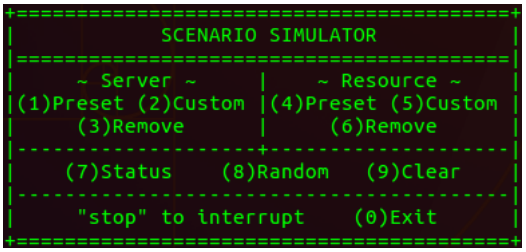

Figura 4.1: Simulatore

- Generazione di server e risorse in modo automatico, utilizzando pattern creati ad-hoc per rispecchiare una ipotetica situazione reale.
- Creazione di server personalizzati con una struttura di risorse a piacere.
- Aggiunta o rimozione dinamica di nodi all'interno della rete.
- Controllo per impedire la presenza di omonimie indesiderate.

### 4.1.2 Fetcher

Per Fetcher intendiamo il sistema che ha la funzione di nodo intermedio tra scenario IoT e Cloud: è uno degli elementi più importanti dell'intera

architettura, e ha il compito di:

- Gestire la Service Discovery.
- Mantenere in memoria l'elenco dei nodi presenti nella rete, la loro composizione e le relazioni di observing attive.
- Sincronizzare lo scenario con il Cloud.
- Gestire le richieste provenienti dai Client Android, occupandosi inoltre di inviare aggiornamenti in caso di observing delle risorse.

### Client CoAP

Il client CoAP gestisce tutte le comunicazioni con i nodi della rete locale, e si occupa di recuperare le informazioni sulla struttura delle risorse contenute nei server su ordine della Service Discovery che opera nel Fetcher Core. Le procedure per effettuare richieste GET e POST sono semplici e simili fra loro:

```
public ResponseDescriptor doCoapGet(FetcherResource
   resource) {
   this.setURI(resource.getCompleteURI());
   Request getRequest = new Request(Code.GET);
   getRequest.setURI(resource.getCompleteURI());
   CoapHandler handler = new CoapHandler() {
      @Override
      public void onLoad(CoapResponse response) {
         ResponseDescriptor desc = new
            Gson().fromJson(/*Parameters...*/);
         ...
      }
   };
   this.advanced(handler, getRequest);
}
```

La richiesta GET viene effettuata in modo asincrono e rimane "aperta", utilizzando un semaforo, fino a che non si riceve una risposta dal server; i dati ricevuti vengono poi incapsulati nell'oggetto ResponseDescriptor, un Java-Bean contenente campi come il valore richiesto e i parametri di identificazione della risorsa. Le richieste POST sono del tutto simili, con la differenza che il valore da comunicare è inserito nel payload e come risultato si riceverà un codice indicativo del successo/insuccesso della trasmissione.

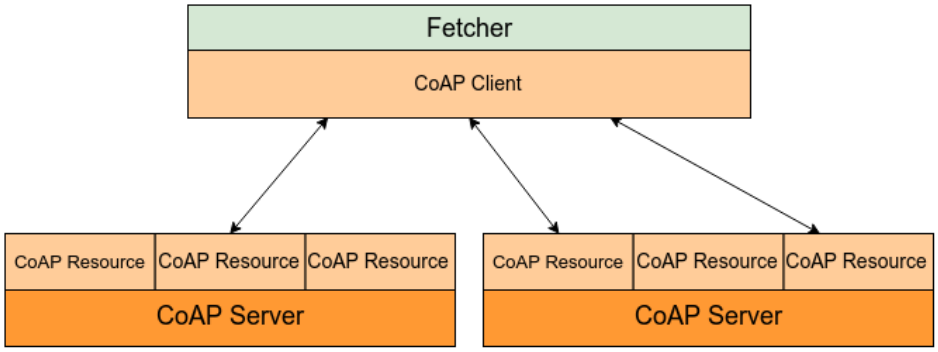


Figura 4.2: Comunicazione CoAP Client-Server

Discorso diverso sono invece le richieste di observing, che vengono pur sempre effettuate tramite GET, ma hanno come parametri aggiuntivi un flag specifico e un token da 8 byte identificativo della singola relazione:

```
public NewObservation doCoapObserve(FetcherResource
   resource) {
   Request getRequest = new Request(Code.GET);
   getRequest.setObserve();
   ...
   final byte[] token = new byte[8];
   for(int i=0; i<8; i++) {
      token[i]=(byte)((char)(sr.nextInt(26)+'a'));
   }
   getRequest.setToken(token);
```

```
    ...
    CoapObserveRelation relation =
       this.observe(getRequest, handler);
}
```

Nel nostro caso è stato conveniente generare casualmente il token utilizzando dei semplici caratteri alfabetici minuscoli, in quanto in fase di testing abbiamo avuto sempre a che fare con un numero relativamente limitato di nodi, ma il range di generazione è arbitrario ed estensibile a tutti i valori assumibili da un array di 8 byte. Inoltre, una volta instaurata la relazione di observing, questa viene salvata nell'oggetto CoapObserveRelation che verrà accluso alla lista di relazioni specifica della risorsa (vedi paragrafo successivo).

Infine il Client CoAP, non appena il Fetcher Core rileva l'ingresso di un nuovo server nella rete ha il compito di instaurare una relazione di observing ad una particolare risorsa, il "/.well-known/core", che contiene una lista completa di tutte le risorse disponibili nel nodo. Di default questa risorsa non è osservabile, ma solo consultabile tramite GET: la natura open source di Californium ci ha però permesso di apportare una piccola modifica al framework, rendendo possibile l'observing e consentendoci di avere notifica immediata di qualsiasi aggiunta o rimozione di risorse all'interno del server:

```
public FetcherServer doServerInit(String name,
   String address, int port) {
    ...
    getRequest.getOptions().clearUriPath()
    .clearUriQuery().setUriPath("/.well-known/core");
    getRequest.getOptions().setUriQuery("rt=*");
       //Takes all resource types

    CoapHandler handler = new CoapHandler() {
```

```
        @Override
        public void onLoad(CoapResponse response) {
            ArrayList<WebLink> webLinks = new
                ArrayList<>(LinkFormat
                .parse(response.getResponseText()));
            ...
        }
    };
    ...
}
```

Come possiamo vedere la richiesta di observing è caratterizzata dalla URI sopra citata e da un filtro, che per noi sarà semplicemente "rt=*" (quasiasi tipo di risorsa). Una volta instaurata la relazione, il server riceverà una notifica ed aggiornerà il proprio albero di risorse automaticamente (vedi paragrafo successivo).

### Fetcher Core

Il Fetcher Core è il componente che racchiude la maggior parte della logica del sistema: coordina le interazioni fra il Client CoAP e il Server HTTP, comunica con PhantomJS, gestisce la Service Discovery e mantiene in memoria lo stato dello scenario aggiornandolo quando necessario.
Prima di tutto occorre specificare come è organizzata la struttura dati che raffigura lo scenario: elementi della rete quali Server e Risorse sono rappresentati da oggetti che incorporano, oltre a informazioni come per esempio URI o porta, references a genitori e figli, permettendo quindi una modellizzazione corretta delle gerarchie ad albero. Ogni oggetto è inoltre responsabile per la propria gestione delle richieste e per le proprie relazioni di observing. La funzione di update invocata all'arrivo di notifiche dal "/.well-known/core"

ha un compito molto delicato, ed essendo molto lungo e complesso non ne verrà riportato il codice. Le risorse sono memorizzate in una lista, e quindi non organizzate gerarchicamente: l'algoritmo implementato ha il compito di ricostruire la struttura ad albero basandosi sulle URI, per poi aggiornare gli oggetti nel Fetcher aggiungendoli o rimuovendoli in modo adeguato; è importante notare inoltre come ogni nodo gestisca esclusivamente le operazioni nel suo "ramo" utilizzando meccanismi ricorsivi e delegando i compiti che non sono di sua competenza a chi di dovere. Per quanto riguarda invece la gestione delle richieste da parte delle risorse il metodo implementato fa uso estensivo dei Token per differenziare le tipologia di richiesta e per aggiornare correttamente la lista delle ObservingRelations attive sull'oggetto. Anche in questo caso la complessità dell'algoritmo è tale da non rendere possibile riportarne direttamente il codice in questa tesi.

La Service Discovery è invece gestita dal Fetcher nel modo seguente:

```
private void initDiscovery() {
    coAPServiceListener = new ServiceListener() {
        public void serviceResolved(ServiceEvent ev) {
            String name = ev.getName();
            String pattern = "(\\S+):(\\S+)";
            name=name.replaceAll(pattern, "$1 $2");
            String[] splitName = name.split("\\s");
            addSmartObject(/*Parameters...*/);
        }
        public void serviceRemoved(ServiceEvent ev) {
            //Same name pattern...
            removeSmartObject(/*Parameters...*/);
        }
        public void serviceAdded(ServiceEvent event) {
            JmDNSManager.getInstance().getJmDNS()
            .requestServiceInfo(/*Parameters...*/);
        }
```

```
    };
    regManager.addServiceListener(coAPServiceListener);
}
```

Questa funzione viene invocata allo startup e rileva l'ingresso o l'uscita di server dalla rete, eseguendo le prime operazioni di setup: di fondamentale importanza è la convenzione utilizzata per la nomenclatura, che ci permette di ottenere immediatamente informazioni quali il nome e la porta su cui opera il nodo.

L'aggiornamento del Cloud viene invece attuato inviando semplici richieste HTTP a PhantomJS, che si occuperà autonomamente di reindirizzarle in modo adeguato (vedi paragrafi successivi). Di seguito è riportato il metodo invocato nel caso in cui vengano aggiunti nuovi nodi:

```
public <T extends CloudObject> boolean
    addToCloud(String URI, T payload) {

    URL serverUrl = new
       URL("http://localhost:"+MyPrefs.PHANTOMJS_PORT+URI);
    HttpURLConnection urlConnection =
       (HttpURLConnection)serverUrl.openConnection();
    ...
    urlConnection.setRequestMethod("PUT");
    ...
    httpRequestBodyWriter.write(gson.toJson(payload));
    ...
}
```

Procedure del tutto simili a quella soprastante sono adottate per la rimozione di nodi e per l'invio di valori inerenti le relazioni di observing.

### Server HTTP

Il server HTTP ha la funzione di ricevere le chieste proveniente da PhantomJS, estrarre le informazioni dal payload ed incapsularle in un oggetto RequestDescriptor; una volta ricevuta la risposta sotto forma di ResponseDescriptor estrae da quest'ultimo i dati necessari e li rispedisce in risposta allo script concludendo la richiesta iniziale.

```
static class MyHandler implements HttpHandler {
...
   public void handle(HttpExchange t) throws
      IOException {
      switch (t.getRequestMethod()) {
         case "GET":
         RequestDescriptor getReqDescriptor = new
            RequestDescriptor(
            t.getRequestURI().toString());
         ...
         OutputStream getOs = t.getResponseBody();
         ResponseDescriptor payload =
            launcher.handleHTTPGet(getReqDescriptor);
         getOs.write(new
            Gson().toJson(payload).getBytes());
         getOs.close();
         break;
         /// POST request handling...
      }
   }
}
```

### PhantomJS

PhantomJS è il componente che consente la comunicazione con il Cloud. Ha la doppia funzione di:

- Caricare lo script JavaScript presente nella directory web del Google Cloud Engine, contenente il codice necessario per l'interazione con le APIs.

- Eseguire un'istanza del Web Server Mongoose, in modo da consentire la sincronizzazione dello scenario IoT con il Cloud e l'invio degli aggiornamenti da parte delle risorse in observing.

```
var page = require('webpage').create();
var server = require('webserver').create();
page.open("http://iot220999.appspot.com/",
   function(status) {});

var service = server.listen(11000, function
   (request, response) {
   var reqtype = request.method;
   switch (reqtype) {
      case "POST":
      var postBody = request.post;
      page.evaluateJavaScript('function(){gapi.client
      .messaging.messagingEndpoint.sendUpdate({\'value\':
         \''+requestJSCode+'\'}).execute();}');
      ...
      response.close();
      break;
      case "PUT":
      var putBody = request.post;
      var putUri =
         decodeURIComponent(request.url.substring(1));
      ...
      var script = "function(){gapi.client.datastore
      .datastoreEndpoint.handlePUT({\"uri\":
         \""+uri+"\",\"json\":
         \""+jsonObject+"\"}).execute();}";
      ...
      response.close();
      break;
```

```
        // DELETE code is similar...
    }
});
```

Per quanto riguarda il primo punto è sufficiente creare un oggetto di tipo "webpage" e aprire la pagina all'indirizzo dello script nel Cloud: in questo modo le APIs remote saranno in grado di effettuare richieste direttamente al Server HTTP integrato con il Fetcher. Il Web Server Moongoose ha invece la funzione di ricevere le richieste dal Fetcher Core e reindirizzarle al Cloud: per fare ciò viene utilizzata la funzione evaluateJavaScript() che consente di eseguire uno script custom direttamente sulla pagina aperta, permettendoci così di effettuare chiamate alle funzioni inserendo come parametri i dati contenuti nelle richieste ricevute. La scelta architetturale di adottare un sistema di questo tipo è stata dettata essenzialmente da due fattori. Il primo ha a che fare con la Channel API, che nel momento in cui si scrive consente di sviluppare client esclusivamente in Javascript: integrare un motore in grado di elaborare script direttamente all'interno del Fetcher sarebbe stato difficilmente realizzabile e laborioso; il secondo fattore invece riguarda direttamente l'architettura, che separando dalla logica del programma un compito delicato come la comunicazione locale-remoto ne ha guadagnato in termini di modularità e mantenibilità del codice.

## 4.2 Google Cloud e client Android

In quest'ultima sezione illustreremo la parte Cloud della nostra architettura, analizzando tutti gli aspetti presi in considerazione nello sviluppo delle

APIs e creando una App Android in grado di rendere fruibili agli utenti le informazioni offerte dallo scenario IoT.

### 4.2.1 JavaScript del Google App Engine

Prima di parlare delle singole APIs è importante spiegare come queste andranno ad interagire con la parte IoT: come anticipato precedentemente, sul Cloud è presente una web directory contenente il JavaScript che consente di avere accesso a tutte le funzionalità di cui abbiamo bisogno:

```
<script type="text/javascript">
function getFromSmartObject() { //Used by PhantomJS
    var xmlhttp = new XMLHttpRequest();
    var url = "http://localhost:10500/testserver";
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 && xmlhttp.status
            == 200) {
            gapi.client.messaging.messagingEndpoint
            .sendMessage({'message':
                xmlhttp.responseText})
            .execute();
        }
    }
    xmlhttp.open("GET", url, true);
    xmlhttp.send();
}
function onMessage(msg) { //Used by Channel API
    var xmlhttp = new XMLHttpRequest();
    var url =
       "http://localhost:10500/HTTPServer/"+msg.data;
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 && xmlhttp.status
            == 200) { //
            gapi.client.messaging.messagingEndpoint
            .sendUpdate({'value':
                xmlhttp.responseText})
            .execute();
```

```
            console.log("Response:
               "+xmlhttp.responseText);
        }
    }
    xmlhttp.open("GET", url, true);
    xmlhttp.send();
    return msg.data;
}
function init() {
    gapi.client.load('channel', apiVersion,
       channelCallback, apiRoot);
    // Same for other APIs...
}
```

Per quanto riguarda strettamente le APIs sono i metodi gapi.client.load() a rendere disponibili le funzionalità all'interno dell'istanza dello script: questo è sufficiente a soddisfare le nostre necessità, in quanto PhantomJS può fare uso a proprio piacimento delle funzioni del Cloud utilizzando evaluateJavaScript(). Abbiamo poi i metodi adibiti ad effettuare GET e POST direttamente al Server HTTP del Fetcher, che verranno invocati dalle APIs ogni qualvolta un client Android ne farà richiesta.

### 4.2.2 Channel

La Channel API è la parte della nostra architettura Cloud che si occupa della comunicazione Cloud-Fetcher. La sua implementazione nel back-end è piuttosto semplice, in quanto la creazione dei token e l'inizializzazione del canale è completamente automatizzata:

```
public class ChannelEndpoint {
    ...
    public void sendOnChannel(@Named("payload")
       String payload) {
```

```
        ChannelService channelService =
           ChannelServiceFactory.getChannelService();
        String channelKey = "TestChannel";
        channelService.sendMessage(new
           ChannelMessage(channelKey,payload));
    }
}
```

Sono necessari solo due metodi: uno per la creazione del Channel con la corrispondente generazione di un token identificativo, e un'altro per l'invio di dati sul canale (che verrà utilizzato dalle altre APIs presenti nell'architettura del Cloud).

### 4.2.3 Datastore

La Datastore API ha essenzialmente il compito di memorizzare la situazione dello scenario IoT (che invia aggiornamenti sulle modifiche in real time) e di renderla disponibile a tutti i client registrati; abbiamo già parlato nel capitolo precedente di caratteristiche e funzionalità del database non relazionale che lavora nel Datastore, ed ora vedremo come sono state impiegate.

#### handlePUT() e handleDELETE()

I metodi handlePUT() e handleDELETE() vengono invocati dal Fetcher tramite il JavaScript in esecuzione su PhantomJS:

```
public void handlePUT(/*Parameters...*/) {
    Transaction transaction =
       datastore.beginTransaction();
    try {
        ...
        switch (splitUri.length) {
            case 1:      //It's a Server
```

```
            CloudServer fetcherServer = new
               Gson().fromJson(json,
               CloudServer.class);
            objectName=fetcherServer.getName();
            addServer(/*Parameters...*/);
            break;
            ...
        }
        transaction.commit();
        } finally {
            if(transaction.isActive())
            transaction.rollback();
        else {
            endpoint.sendPUT(uri+"/"+objectName);
        }
    }
}
```

Per prima cosa viene inizializzata una transazione; successivamente, in base alla lunghezza della URI "radice", viene invocato il metodo adatto in grado di aggiungere o rimuovere entità dal Datastore e sincronizzare lo stato in memoria con quello reale. Se la transazione va a buon fine viene inviata una notifica ai client Android.

### Aggiunta e rimozione di elementi

Per aggiungere entità al Datastore sono stati sviluppati metodi specifici nel caso in cui si abbia a che fare con Scenari, Server o Risorse. Di seguito ne riportiamo un esempio:

```
    private void addResources(/*Parameters...*/) {
        Entity resource = new
           Entity(/*Parameters...*/);
        for(String type : fetcherResource.getTypes())
            resource.setProperty("Types",type);
        datastore.put(resource);
```

```
        for(CloudResource child :
            fetcherResource.getResChildren())
            addResources(child,resource); //Recursion
    }
```

Come è facile osservare occorre creare un oggetto Entity ed utilizzare il costruttore per specificare la chiave, per poi fare uso di setProperty() allo scopo di aggiungere i campi desiderati; infine viene eseguita l'operazione di PUT o DELETE vera e propria, e nel caso specifico delle risorse viene effettuata una chiamata ricorsiva per permettere la creazione (o rimozione) di strutture ad albero.

### Ricerca di elementi

Nel momento in cui dobbiamo effettuare operazioni di DELETE, o quando un client Android necessita di informazioni sullo scenario, sarà necessario operare ricerche fra le entità esistenti:

```
    private Entity findServer(String serverName,
        Entity scenario) {
        Query serverQuery = new Query("Server")
        .setAncestor(scenario.getKey())
        .setFilter(new
            FilterPredicate(Entity.KEY_RESERVED_PROPERTY,
            FilterOperator.EQUAL,
            KeyFactory.createKey(scenario.getKey(),
            "Server", serverName)));
        PreparedQuery pq =
            datastore.prepare(serverQuery);
        return pq.asSingleEntity();
    }
```

Niente di troppo difficile in questo caso: creiamo un oggetto Query, impostiamo i filtri desiderati e dopo l'invocazione di datastore.prepare(serverQuery)

otterremo come risultato una Entità (o una lista di Entità a seconda dei casi).

#### GET dei client Android

Quando i client Android hanno bisogno di informazioni sullo scenario possono ottenerle invocando direttamente il metodo più consono contenuto nella API:

```
public CloudServer getServer(@Named("uri")
   String uri) {
   Entity server = findServer(/*Parameters...*/);
   CloudServer updatedServer = new
      CloudServer(/*Parameters...*/);
   ...
   for(Entity resource :
      getServerResources(server))
      resources.add(new
         CloudResource(/*Parameters...*/);
   updatedServer.setResources(resources);
   return updatedServer;
}
```

Nell'esempio è riportata la funzione che restituisce lo stato di un Server: vengono effettuate diverse query al Datastore, che oltre alle proprietà del Server stesso forniscono una lista contenente tutte le Entità figlie; gli algoritmi utilizzati per Scenario e Risorse sono del tutto simili.

### 4.2.4 Cloud Messaging

L'ultima API che andiamo ad analizzare è quella dedicata al Cloud Messaging. Per riuscire ad interfacciarsi con i servizi Cloud, ciascun client Android dovrà effettuare una prima registrazione alla piattaforma: a questo scopo è stato implemento un Endpoint dedicato che si occupa di memorizzare la

lista degli utenti "iscritti" in apposite Entità del Datastore, lista che verrà successivamente utilizzata dal GCM per l'invio broadcast dei messaggi:

```
public void sendMessage(Message msg) throws
    IOException {
    Sender sender = new Sender(API_KEY);
    List<RegistrationRecord> records =
       ofy().load().type(RegistrationRecord.class)
          .limit(10).list();
    for (RegistrationRecord record : records) {
       Result result = sender.send(msg,
          record.getRegId(), 5);
       ...
    }
}
```

Il metodo sopra riportato ha il compito di inviare il messaggio passato come parametro ai primi 10 device registrati. Nel nostro progetto si è optato però per una diversificazione dei messaggi a seconda del loro utilizzo:

```
public void sendUpdate(@Named("value") String
    value) {
    sendMessage(new Message.Builder()
    .addData("type", IoTMessageType.DATA.toString())
    .addData("value", value)
    .build());
}
//Similar methods for other message types
```

Le tipologie di messaggio sono contenute all'interno di un Enum, che verrà utilizzato anche nelle App Client.

### 4.2.5 Client Android

L'intera struttura illustrata finora non avrebbe nessuna utilità senza un punto di contatto con gli utenti finali: la Google Cloud Platform comprensiva di

tutti i suoi servizi è supportata nativamente sui dispositivi Android attraverso i Play Services, ed è per questo motivo che la scelta della piattaforma client è ricaduta proprio sul più diffuso sistema operativo mobile in circolazione. Per prima cosa è importante fare una premessa sull'utilizzo del GCM. Nei paragrafi precedenti abbiamo evidenziato come il Cloud Messaging sia perfetto per scenari che richiedono l'invio di una grande quantità di messaggi dalle dimensioni ridotte: l'utilizzo di questo servizio non è stato quindi inteso come veicolo dei dati veri e propri, ma come un modo per notificare i client dei cambiamenti intervenuti e consentirgli di effettuare richieste mirate alle risorse presenti sul Cloud.

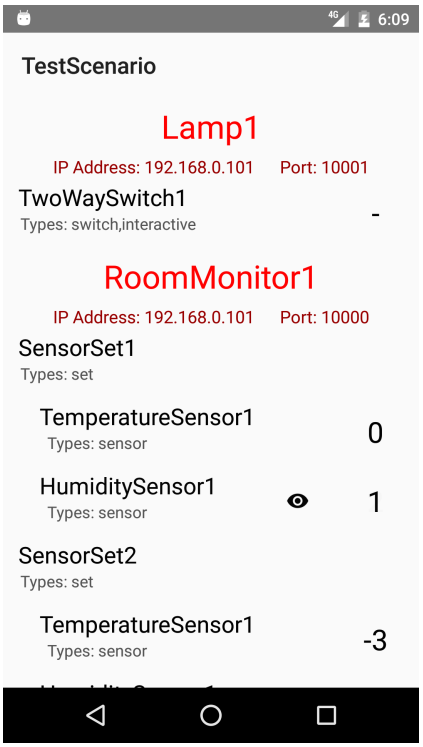

Figura 4.3: App Android

Vediamo ora come viene utilizzata la piattaforma di Cloud Messaging: l'App utilizza un WakefulBroadcastReceiver, che per ogni messaggio ricevuto dal GCM istanzierà un WakefulService:

```
...
protected void handleMessage(final Bundle extras) {
    new Handler(Looper.getMainLooper()).post(new
        Runnable() {
        @Override
        public void run() {
            Intent intent = null;
```

```
            switch (IoTMessageType.values()[Integer
              .valueOf(extras.getString("type"))]) {
               case PUT:
               intent = new Intent("PUT");
               intent.putExtra("uri",
                  extras.getString("uri"));
               intent.putExtra("payload",
                  extras.getString("payload"));
               break;
               //Same for other message types
            }
            getApplicationContext().sendBroadcast(intent);
        }
    });
}
```

Il metodo handleMessage() determina il tipo di messaggio ricevuto utilizzando il valore nel campo "type" visto nel paragrafo precedente, ed invia un Intent in broadcast: l'app implementa a sua volta Broadcast Receivers per ciascuna categoria, e gestisce i vari casi separatamente. Al momento opportuno vengono poi creati degli AsyncTasks, che utilizzando le librerie di comunicazione con il Cloud (generate automaticamente da Android Studio) effettuano le richieste necessarie:

```
public class GetServerAsyncTask extends
   AsyncTask<String, Void, CloudServer> {
   ...
   @Override
   protected CloudServer doInBackground(String...
      params) {
      if(datastoreService==null) {
         Datastore.Builder builder = new
            Datastore.Builder(/*Parameters...*/)
     .setRootUrl("https://iot220999.appspot.com/_ah/api/");
         datastoreService = builder.build();
      }
      ...
```

```
        CloudServer server =
            datastoreService.datastoreEndpoint()
        .getServer(params[0]).execute();
        return server;
    }
    @Override
    protected void onPostExecute(CloudServer param) {
        AppServer server = new
            AppServer(param,listAdapter);
        server.initializeResources(param.getResources());
        scenario.getServers().add(server);
    }
}
```

La richiesta vera e propria viene eseguita in background utilizzando un Thread diverso da quello della UI, in modo da non impattare negativamente sulla user experience. Nel momento in cui si riceve una risposta la struttura dati interna all'app viene aggiornata: ogni oggetto mantiene al suo interno una reference all'Adapter della RecyclerView utilizzata nella MainActivity, ed include funzioni ad hoc che aggiornano automaticamente l'interfaccia grafica.

Le richieste alle risorse possono essere effettuate in due modi:

- con un tap semplice viene effettuata una GET alla risorsa;
- con un tap prolungato viene attivato o disattivato l'observing, il cui stato è indicato dall'icona corrispondente.

Anche in questo caso le richieste al Cloud sono gestite da AsyncTasks dedicati che utilizzano le librerie client generate automaticamente dall'IDE:

```
@Override
protected String doInBackground(String... params) {
    if(channelService ==null) {
```

```
        Channel.Builder builder = new
           Channel.Builder(/*Parameters...*/)
        .setRootUrl("https://iot220999.appspot.com/_ah/api/");
        channelService =builder.build();
        }
        ...
        channelService.channelEndpoint()
        .sendOnChannel(params[0]).execute();
}
```

# Capitolo 5

# Conclusioni

Dopo aver sviluppato ed implementato tutti i componenti descritti nel capitolo 4 abbiamo a disposizione un'infrastruttura funzionalmente completa, che consente l'accesso alla struttura dinamica di una rete IoT locale e ai contenuti da essa offerti mediante l'utilizzo di una app Android connessa alla rete Internet, senza cioè il vincolo di doversi trovare all'interno della stessa LAN.

Le scelte fatte consentono inoltre una buona scalabilità, mantenendo (e per certi versi aumentando) le prestazioni in scenari contraddistinti da un alto numero di nodi e client. Per mettere alla prova il sistema è stato effettuato un test sul tempo di risposta delle richieste GET all'aumentare della "frequenza oraria" di queste ultime, e i risultati ottenuti sono riportati nel grafico sottostante:

| Numero GET | Latenza Media |
|---|---|
| 1-10 | $\sim$ 1000ms |
| 10-50 | $\sim$ 680ms |
| 50-100 | $\sim$ 570ms |
| 100-200 | $\sim$ 520ms |
| 200-300 | $\sim$ 490ms |
| 300-500 | $\sim$ 470ms |
| 500-1000 | $\sim$ 440ms |

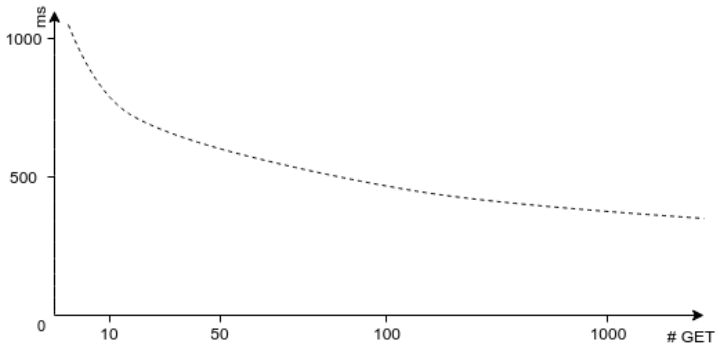


Figura 5.1: Tabella e diagramma delle prestazioni

Come possiamo vedere, inizialmente si ha una latenza relativamente alta, che però diminuisce costantemente all'aumentare del numero delle richieste: questo perché la Google Cloud Platform è progettata per gestire il carico di lavoro in modo intelligente, privilegiando ed offrendo tempi di risposta migliori alle istanze delle applicazioni con più traffico in entrata e uscita. Avremo quindi prestazioni migliori sulla piattaforma Cloud all'aumentare delle richieste da parte dei client Android, e la natura stessa del protocollo CoAP garantirà l'assenza di colli di bottiglia ed ottime performance della parte IoT anche in presenza di un numero di operazioni molto elevato.

Nonostante si disponga di una infrastruttura completamente funzionante, gli sviluppi attuabili sono molteplici:

- La natura prototipale della tesi ha dato relativamente poca importanza al fattore sicurezza e critografia: nel caso in cui si decidesse estendere l'utilizzo del sistema all'utente finale sarebbe d'obbligo occuparsi di questa parte, con la consapevolezza che comunque la base sviluppata offre possibilità concrete di implementare meccanismi a tutela della privacy.

- Per avere un riscontro concreto sulle prestazioni effettive della rete IoT sarebbe auspicabile svolgere test utilizzando dei dispositivi fisici: Arduino e Raspberry offrono soluzioni ideali per questi scenari, economiche ma comunque in grado di consentire l'esecuzione di programmi Java (i nostri Server) e completabili con componenti di sensoristica.

- Come illustrato nello stato dell'arte, la Google Cloud Platform consente di creare librerie anche per dispositivi non-Android: sarebbe quindi interessante diversificare la tipologia dei client in modo da sfruttare appieno l'estensibilità dell'infrastruttura creata.

# Bibliografia


[1] CoAP, RFC 7252 Constrained Application Protocol, http://coap.technology

[2] IETF, *The Constrained Application Protocol (CoAP)*, https://tools.ietf.org/html/rfc7252

[3] IETF, *Observing Resources in CoAP - draft-ietf-core-observe-16*, https://tools.ietf.org/html/draft-ietf-core-observe-16

[4] S. Cirani, *"Constrained Application Protocol," Lecture 8* (Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma)

[5] IETF, *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*, https://tools.ietf.org/html/rfc6282

[6] Multicast DNS, http://www.multicastdns.org/

[7] Apple, *Apple Push Notification Service*, https://developer.apple.com/

[8] Amazon, *Amazon Simple Notification Service (SNS)*, https://aws.amazon.com/it/sns/

[9] Google, *Google Cloud Platform Documentation*, `https://cloud.google.com/docs/`

[10] Eclipse Foundation, *Californium*, `http://www.eclipse.org/californium/`

[11] Jamie Mason, *PhantomJS Documentation*, `http://phantomjs.org/documentation/`

[12] Google, *Android Developers Documentation*, `https://developer.android.com`

[13] Google, *Android Open Source Project*, `https://source.android.com/`

[14] Google, *Gson Library*, `https://github.com/google/gson`