ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Tesi di laurea in
SISTEMI MOBILI M

SERVIZI DI DISCOVERY IN IOT: UNA SOLUZIONE MQTT-BASED PER GATEWAY KURA

Relatore:
Chiar.mo Prof.
PAOLO BELLAVISTA

Candidato:
CARLO ANTENUCCI

Correlatore:
Dott.
ALESSANDRO ZANNI

Correlatore:
Dott.
CARLO GIANNELLI

Anno Accademico 2014 – 2015
Sessione III

Indice

Introduzione

L'avanzamento tecnologico dovuto all'abbattimento dei costi, alla riduzione dell'hardware, ed allo sviluppo di nuovi protocolli sta portando sempre più oggetti ad accedere ad Internet, tanto che negli ultimi anni il numero dei dispositivi connessi ha superato quello degli abitanti della Terra. Questo trend non accenna a diminuire: secondo le stime di illustri compagnie del settore si prevede che nel giro di qualche anno ci saranno almeno tre dispositivi connessi per ogni persona del Pianeta.

Questo aumento non è riferito solo ai tradizionali mezzi di accesso ad Internet (PC, smartphone, tablet, etc.), ma a tutto ciò che ha a che fare con la vita quotidiana. Non sono rari i casi di città diventate smart per consentire una miglior gestione da parte dei governanti ed offrire servizi ai cittadini, di impianti domestici gestiti da remoto o auto-configurabili secondo le esigenze dei proprietari, di infrastrutture stradali che ricevono informazioni dai veicoli in transito in modo da consentire miglioramenti alla viabilità e segnalazione di problemi, o di dispositivi indossabili che consentono un monitoraggio continuativo dell'attività vitale delle persone.

L'avvento dell'Internet delle Cose può essere visto a tutti gli effetti come la più grande rivoluzione tecnologica degli ultimi anni, che porterà nelle mani degli utenti un elevatissimo numero di informazioni in grado di offrire innumerevoli benefici nella vita di ogni giorno.

Per rendere possibile l'accesso ai dati disponibili è necessario un meccanismo che consenta la scoperta e l'accesso ai nodi che offrono dei servizi di gestione delle informazioni: è impensabile che ogni dispositivo sappia a priori dove reperire i dati (o dove trasmetterli). Alcuni di questi meccanismi sono già disponibili in letteratura, ma presentano dei difetti che verrebbero ancor più accentuati se utilizzati con dispositivi dalle capacità computazionali limitate come quelli che fanno parte di IoT.

Risulta necessaria anche la presenza di nodi che si occupino del mantenimento e della distribuzione dei dati, ed è doveroso regolamentare l'accesso alle informazioni poiché queste potrebbero contenere dati personali o confidenziali.

Sono molte le aziende che hanno investito ingenti risorse per finanziare la ricerca e lo sviluppo di IoT. Benché una prima impressione sui cospicui investimenti possa risultare decisamente positiva per lo sviluppo e la diffusione di Internet of Things, analizzando la situazione da un punto di vista più business-oriented si potrebbe pensare che le aziende siano intenzionate a trarre il maggior profitto possibile; questo potrebbe cau-

sare la compromissione dell'idea di integrazione totale tra mondo reale e virtuale nel caso in cui un'azienda adottasse standard proprietari per imporsi a livello globale e tagliare fuori la concorrenza.

Per scongiurare questa ipotesi, aziende del calibro di IBM e Cisco stanno contribuendo a progetti open-source per la realizzazione di framework altamente configurabili e servizi con cui i dispositivi possono interagire. In questo contesto si inserisce anche un'azienda italiana: Eurotech. Leader mondiale nella realizzazione di sistemi embedded ed High Performance Computer, è uno dei membri fondatori dell'IoT Working Group della Eclipse Foundation, ed ha contribuito al rilascio di Kura, un progetto open-source che si propone di realizzare un gateway OSGi per IoT.

L'obiettivo di questo progetto di tesi è quello di realizzare un meccanismo che, basandosi su un protocollo di comunicazione leggero come MQTT, possa essere utilizzato da dispositivi con risorse limitate per la ricerca, l'accesso, e la trasmissione di dati a gateway Kura.

La soluzione proposta prevede la presenza di tre elementi di infrastruttura: i client, i gateway Kura ed un broker MQTT su cloud.

I client saranno interessati a trovare gateway nelle loro vicinanze. Per fare ciò dovranno individuare la più probabile località in cui si trovano, quindi annunceranno la loro presenza inviando al broker su cloud un messaggio da pubblicare in un topic location-specific. I client specificheranno nel messaggio le loro coordinate e le loro caratteristiche.

Il broker recapiterà il messaggio ricevuto a tutti i gateway presenti in quella località che si saranno precedentemente registrati al topic geografico.

I gateway, oltre a Kura, avranno in esecuzione un broker MQTT locale, e forniranno ai client l'accesso ad Internet mettendo a loro disposizione una rete WiFi; alla ricezione del messaggio di presenza il servizio in esecuzione sul gateway analizzerà le caratteristiche del client, e, se quest'ultimo risulterà essere utile alle applicazioni in esecuzione, gli invierà una risposta che dovrà contenere i parametri di configurazione per accedere alla rete WiFi e l'indirizzo locale a cui trovare il broker MQTT locale.

Quando un client riceverà risposte dai vari gateway interessati alle sue caratteristiche ne selezionerà uno analizzando i contenuti dei payload, configurerà la rete WiFi, stabilirà la connessione con il broker MQTT locale, ed inizierà a scambiare informazioni con il gateway in modo diretto.

Il primo capitolo di questa tesi illustrerà lo stato dell'arte definendo cosa si intende per Internet of Things, quali sono le sue caratteristiche principali e le sfide tecnologiche che dovranno essere superate. Successivamente verrà fatta una panoramica sui sistemi di discovery analizzando gli standard più utilizzati, verranno introdotti gli obiettivi che si intendono raggiungere, e saranno descritti i lavori correlati presenti in letteratura.

Nel secondo capitolo verranno introdotte le tecnologie che saranno utilizzate nella realizzazione del progetto: si descriveranno le caratteristiche principali e le architetture del protocollo di messaggistica MQTT e del framework OSGi per gateway IoT Kura.

Il terzo capitolo descriverà la fase di progettazione del servizio andando a realizzare una prima analisi dei requisiti volta a definire quali saranno i componenti che comporranno l'architettura del servizio e la loro

interazione. In seguito si introdurrà il formato dei messaggi che client e gateway dovranno condividere per poter comunicare e saranno analizzate le specifiche problematiche di ogni componente. In conclusione verrà illustrata quella che sarà l'architettura logica del servizio di discovery.

Nel quarto capitolo verrà descritta l'implementazione dei componenti analizzando nel dettaglio i meccanismi che andranno a comporli, mentre il capitolo 5 descriverà la procedura di configurazione del gateway Kura e di installazione del servizio, quindi verrà descritta la procedura di testing e saranno analizzati i risultati ottenuti.

Capitolo 1

Stato dell'arte

1.1 Internet of Things

1.1.1 Cosa si intende per Internet delle Cose

Il termine Internet of Things (IoT) è un neologismo, utilizzato per la prima volta nel 1999 da un gruppo di ricerca del MIT (Massachusetts Institute of Technology), di cui non esiste una definizione unica e precisa. Confrontandone alcune è possibile estrapolare quali siano i suoi maggiori argomenti di interesse.

La definizione data dall'IoT Council desrive Internet delle Cose come una visione, ponendo il focus sul fatto che lo sviluppo tecnologico stia portando sempre un maggior numero di oggetti, case e città a diventare smart, e che questo processo, come uno tsunami, non possa essere fermato. Il Consiglio si pone quindi l'obiettivo di seguire lo sviluppo e fare previsioni su cosa accadrà quando gli "smart objects" supereranno gli uomini:

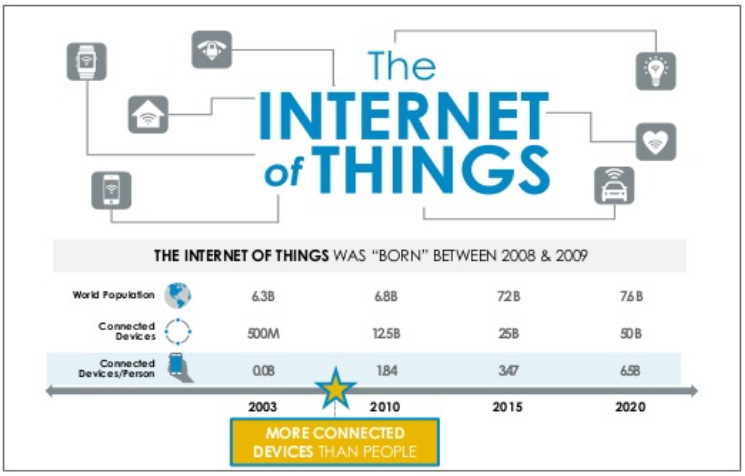
> *"The Internet of Things (IoT) is a vision. It is being built today. The stakeholders are known, the debate has yet to start. In hundreds of years our real needs have not changed. We want to be loved, feel safe, have fun, be relevant in work and friendship, be able to support our families and somehow play a role - however small - in the larger scheme of things. So what will really happen when things, homes and cities become smart? The result will probably be an tsunami of what at first looks like very small steps, small changes. The purpose of Council is to follow and forecast what will happen when smart objects surround us in smart homes, offices, streets, and cities."* [3]

Il gruppo IBSG (Internet Business Solutions Group) di Cisco, in una sua definizione, cerca invece di individuare una "data di nascita" per IoT, concentrando l'attenzione sul rapporto tra uomo e tecnologia, ed affermando che:

> *"IoT is simply the point in time when more "things or objects" were connected to the Internet than people."* [4]

Secondo uno studio realizzato dal gruppo, nel 2003 il rapporto tra i dispositivi connessi ad Internet (circa 500 milioni) e la popolazione mondiale (6,3 miliardi) era circa lo 0,08. Lo stesso studio, riproposto per il 2010, porta alla luce che il numero di dispositivi connessi ad Internet ha avuto una crescita esplosiva (circa 12,5 miliardi di unità). Tale incremento, dovuto alla diffusione dell'ubiquitous computing[1], porta il rapporto a superare l'unità (1,84 per l'esattezza). Stando a questi dati e secondo la definizione precedente, si può dire che IoT sia "nata" all'incirca tra il 2008 e il 2009 e, sempre secondo studi del gruppo IBSG di Cisco, si prevede che il numero di dispositivi continuerà ad aumentare fino a raggiungere i 50 miliardi nel 2020 (Figura 1.1).

Figura 1.1: Studio CISCO IBSG



Source: Cisco IBSG, April 2011

Il fatto che l'Internet of Things si sia sviluppata solo in tempi recenti è dovuto al fatto che il supporto tecnologico affinché oggetti quotidiani potessero accedere ad Internet non è banale. Basti pensare all'imponenza delle unità di calcolo negli anni '60 e '70, e al fatto che solo negli ultimi tempi si è raggiunto un livello tecnologico in grado di concentrare l'hardware necessario all'elaborazione e alla trasmissione delle informazioni in dispositivi di dimensioni ridotte.

La concentrazione dell'hardware, tuttavia, non è l'unico fattore che ha consentito lo sviluppo di IoT: hanno un ruolo centrale anche i miglioramenti dei protocolli di comunicazione e delle tecnologie di rappresentazione delle informazioni. Nonostante siano concepiti inizialmente per contesti differenti, i protocolli della famiglia 802, con opportuni perfezionamenti, si prestano ad essere utilizzati anche dai nodi che compongono l'Internet delle Cose. Per quanto riguarda la rappresentazione dei dati, invece, hanno un ruolo centrale XML e JSON.

Anche l'IIT (Istituto di Informatica e Telecomunicazioni) del CNR (Consiglio Nazionale delle Ricerche) ha dato una sua definizione di IoT dicendo che

[1] Ubiquitous Computing è un modello di interazione uomo-macchina in cui l'elaborazione delle informazioni è stata integrata all'interno di oggetti e attività di tutti i giorni: smartphone, orologi, etc.

> *"[...] indica il superamento dei classici limiti della rete che, fuoriuscendo dal mondo virtuale, si collega al mondo reale, al mondo degli oggetti. Tag e sensori infatti, associati ad un oggetto, possono identificarlo univocamente e raccogliere informazioni in tempo reale su parametri del suo ambiente [...]"*

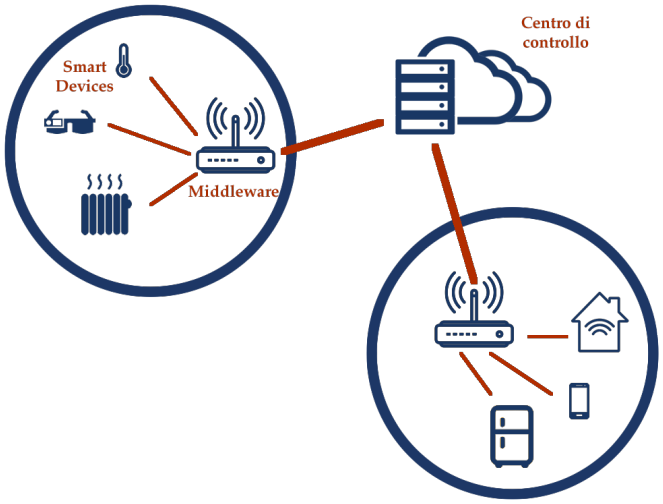
Questo punto di vista mette in luce la possibilità che i nodi IoT possano raccogliere sistematicamente informazioni in un determinato ambiente; questo significa che i nodi devono essere localizzati ed identificati per consentire la separazione di informazioni provenienti da ambienti diversi.

Partendo da queste definizioni si può quindi definire l'Internet of Things come l'interconnessione di dispositivi a basse prestazioni attraverso l'infrastruttura Internet esistente, in grado di coprire una grande varietà di protocolli, domini e applicazioni. IoT prevede inoltre di offrire un connettività avanzata di dispositivi, sistemi e servizi che vada oltre la semplice machine-to-machine communications (M2M).

1.1.2 Peculiarità di IoT

Essendo l'Internet of Things una rete di oggetti interconnessi in grado di comunicare tra loro e con il mondo esterno, si propone, di fatto, di realizzare un collegamento tra il modo reale e quello virtuale. Le soluzioni attuali sono altamente eterogenee, ma tutte si basano su una architettura a tre livelli: l'interfaccia con il mondo fisico, un livello intermedio di supporto e il centro di controllo.

Figura 1.2: Architettura di Internet of Things



Gli smart device realizzano l'interfaccia con il mondo fisico. Sono, come detto, dispositivi dalle ridotte capacità computazionali che, affinché possano accedere alla rete, devono essere dotati di interfacce di comunicazione. Essi sono caratterizzati da una elevata mobilità, possono essere identificati univocamente e localizzati, e interagiscono con l'uomo in modo bidirezionale. Tali end-point possono quindi essere inse-

riti all'interno di un determinato ambiente per raccogliere in modo sistematico dati e informazioni, con la possibilità di processare questi ultimi su nodi centralizzati dotati di maggior potenza computazionale.

Affinché tutto questo sia possibile è necessario che gli smart device siano in grado di comunicare tra di loro e con il mondo esterno. La maggior parte delle tecnologie che vengono utilizzate è di tipo wireless e nessuna di queste è stata realizzata appositamente per Internet of Things. I dispositivi possono essere dotati di connessione WiFi, Bluetooth, ZigBee, o, nel caso degli smartphone, di connessione dati cellulare (3/4G).

Mentre per quanto riguarda il livello di trasporto dello stack ISO/OSI ci si è basati su tecnologie esistenti, per quanto riguarda i protocolli a livello applicativo, dovendo fare i conti con la ridotta capacità di elaborazione degli end-device IoT, vengono utilizzati prevalentemente protocolli binari. Questi ultimi sono in grado di garantire un overhead molto più limitato rispetto a quello di protocolli testuali come HTTP e sono basati prevalentemente su pattern publish/subscribe garantendo il disaccoppiamento tra publisher (produttore del messaggio/evento) e subscriber (consumatore).

In IoT, a livello applicativo, c'è un'ampia varietà di protocolli dovuta al fatto che la connessione di differenti sensori e oggetti dà vita ad un ampio numero di possibili casi d'uso, a seconda del quale è più conveniente l'utilizzo di un protocollo rispetto ad un altro. I protocolli applicativi maggiormente utilizzati sono:

MQTT Message Queue Telemetry Transport: protocollo di comunicazione leggero in grado di garantire affidabilità anche in condizioni non ottimali (banda limitata e connessione instabile) basato su pattern publish/subscribe; necessita della presenza di un componente intermedio tra publisher e subscriber che si occuperà della distribuzione dei messaggi. A livello di trasporto utilizza TCP/IP.

CoAP Constrained Application Protocol: protocollo HTTP-like: si basa sul pattern request/response eliminando dal messaggio l'overhead che HTTP comporta ed utilizza messaggi in formato binario anziché testuale. A livello di trasporto si appoggia ad UDP.

XMPP eXtensible Messaging and Presence Protocol: protocollo che può operare sia con pattern publish/-subscribe che in modalità request/response progettato per sistemi di messaggistica istantanea e utilizzato anche per il controllo di presenza. È basato su XML, quindi molto verboso e pesante anche se generale e dinamico. Utilizza un modello client/server: il client invia il suo messaggio al server che si occuperà della distribuzione.

Il Middleware IoT realizza il layer intermedio offrendo servizi di supporto che consentono la completa integrazione di sistemi eterogenei e avrà il compito di dare organizzazione. Tra i principali servizi di supporto che il middleware dovrebbe mettere a disposizione ci sono:

Connettività e comunicazione: il middleware consente ai diversi dispositivi (eterogenei) della rete di interoperare. Spesso si preferisce far transitare ogni comunicazione per un nodo centrale evitando comunicazioni dirette tra gli smart device. In questo modo i dispositivi non dovranno fornire supporto alle comunicazioni introducendo un notevole vantaggio: smart device diversi che utilizzano protocolli diversi potranno comunicare senza dover necessariamente usare lo stesso protocollo o conoscerne

più di uno; sarà il middleware, conoscendo i dispositivi ed i protocolli utilizzati, ad occuparsi della traduzione dei messaggi.

Device Management: garantisce continua consapevolezza del suo contesto operativo[2]. Questo comporta che il middleware dovrà avere la completa gestione delle azioni e degli eventi degli smart device, in questo modo si renderà la gestione dei dispositivi e delle risorse indipendente dal sistema operativo e dalle applicazioni. Tra le principali funzioni di device management si possono considerare il discovery dinamico dei dispositivi, l'eliminazione di quelli difettosi, la loro localizzazione, etc.

Data collection, analysis e actuation: il middleware fornisce supporto alla memorizzazione dei dati ricevuti dalle entità che compongono la rete di dispositivi collegati ad esso, l'elaborazione degli stessi e la modifica allo stato di altri smart device in base alle informazioni raccolte. Questo compito risulta necessario poiché i dispositivi al disotto non dispongono di supporti adeguati alla memorizzazione e all'elaborazione dei dati; potrebbe essere oltremodo conveniente utilizzare un nodo intermedio per raccogliere dati (gateway IoT) e successivamente inviarli ad un cloud distribuito per la memorizzazione ed elaborazione.

Scalabilità: la presenza del middleware garantisce inoltre una migliore scalabilità: i dispositivi non dovranno far fronte alle richieste di un elevato numero di clienti, ma solo a quelle che il middleware stesso deciderà di inoltrare. Sarà il middleware, infatti a gestire nel miglior modo possibile le richieste dei clienti smistandole al dispositivo più adatto.

Sicurezza: rappresenta uno degli aspetti più critici di IoT poiché, oltre ai tradizionali problemi di sicurezza presenti nelle reti di calcolatori tradizionali bisogna far fronte anche a problematiche specifiche per le differenti tecnologie di smart device; a queste vanno aggiunte tutte le problematiche legate all'autenticazione e al riconoscimento.

1.1.3 Scenario attuale e aspettative di crescita

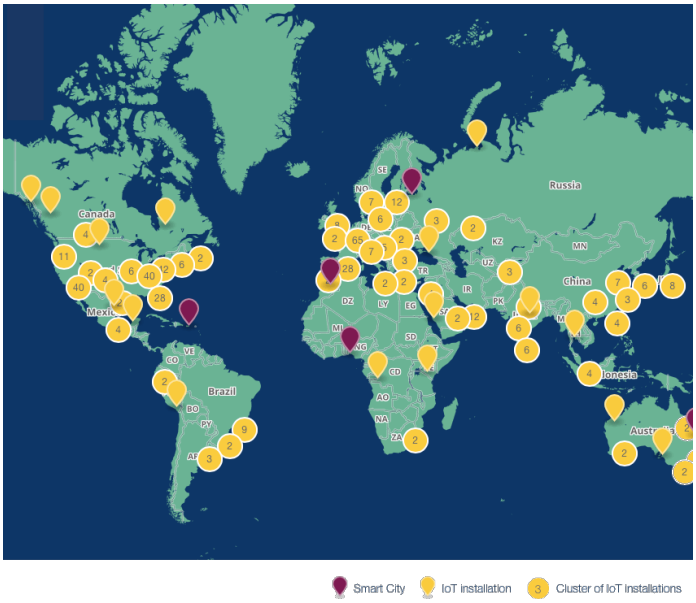
Secondo uno studio Gartner presentato in occasione dell'ITxpo 2015[5] i dispositivi smart connessi ad Internet sono quasi 5 miliardi, e con un aumento del 30% entro la fine del 2016 dovrebbero superare i 6 miliardi. Sempre secondo le stime dell'agenzia nel 2020 si dovrebbero avere più 20 miliardi di smart device connessi ad Internet.

Il trend è in forte crescita, grazie soprattutto alla varietà dei servizi offerti e degli scenari applicativi per cui IoT si presta. Tra questi spicca la realizzazione delle "Smart cities": città che utilizzano la tecnologia offerta da IoT con lo scopo di migliorare la vita degli abitanti. Gli smart device raccolgono dati e informazioni in tempo reale sullo stato degli edifici, le condizioni atmosferiche, la situazione del traffico, i consumi e le emissioni in determinate aree della città. L'elaborazione dei dati raccolti consentirà una gestione più efficiente della città che comporterà una riduzione dei costi di gestione.

Al mondo si contano attualmente quasi un centinaio di smart city, ma rappresentano solo una piccola parte delle installazioni IoT presenti al mondo.

[2] Per contesto operativo si intende lo stato di ogni dispositivo, le sue funzionalità, etc.

Figura 1.3: Installazioni IoT nel mondo[6]



Internet of Things, infatti, non è soltanto smart city: gli ambiti applicativi sono svariati come le smart home e smart building che, in modo simile alle smart city consistono nella raccolta di informazioni in ambito domestico o di edificio in un'ottica di risparmio energetico. La e-healt invece consente il monitoraggio real time di parametri vitali da remoto riducendo il ricorso all'ospedalizzazione sia a fini diagnostici che di cura, ma anche la localizzazione dei pazienti, ad esempio in caso di Alzheimer.

Un ulteriore ambito di applicazione sono le smart car: attraverso la connessione tra veicoli o con un'infrastruttura circostante si cerca di prevenire e/o evitare incidenti stradali, e fornire informazioni in tempo reale sullo stato del traffico; grazie a delle centraline che comunicano alle compagnie assicurative le informazioni sullo stato di guida al momento del sinistro (velocità, tipo di impatto, etc.), sono inoltre stati introdotti nuovi modelli assicurativi.

1.1.4 Problematiche e sfide tecnologiche

L'IoT Council dicendo che tutto il necessario per la realizzazione dell'Internet delle Cose è già disponibile e vada solo assemblato, si riferisce a quanto già detto sui protocolli e le tecnologie di rappresentazione dei dati.

Questo non significa che non ci siano sfide da superare, ma le problematiche sono altre: essendo il mercato di IoT in continua crescita le molte le aziende che stanno lavorando attivamente al suo sviluppo rappresentano una grande risorsa e, al contempo, una possibile minaccia all'obiettivo di integrare totalmente il mondo fisico e quello virtuale. L'assenza di veri e propri standard per IoT potrebbe portare uno o più producer ad applicarne di proprietari per imporsi nel mercato a livello globale causando il fallimento dell'obiettivo iniziale.

Secondo Mark O'Neill la mancanza di standard ben definiti rappresenta una delle maggiori vulnerabilità di IoT[7], poiché sta conducendo ad una eccessiva proliferazione di protocolli: MQTT, CoAP e XMPP sono solo i più utilizzati, ad essi vanno ad aggiungersi AMQP, DDS, DTLS e molti altri[8].

Un'altra problematica riconducibile all'assenza di standard è rappresentata dal fatto che IoT potrebbe raggiungere livelli di controllo troppo invasivi. Risulta quindi di fondamentale importanza una regolamentazione degli aspetti relativi alla privacy e alla sicurezza, come ad esempio la realizzazione di un livello di sicurezza che garantisca comunicazioni sicure[3] e gestione delle chiavi.

I progettisti, tuttavia, devono anche fare i conti con problematiche dipendenti dalle tecnologie a disposizione che potrebbero rallentare lo sviluppo. In primis devono tener conto delle limitate prestazioni dei terminali IoT e cercare sempre di far eseguire le operazioni più gravose all'esterno. Le ridotte capacità computazionali inoltre hanno portato all'utilizzo di sistemi di sicurezza che, a differenza della maggior parte dei protocolli tradizionali, non sono basati su eventi temporali, ma ad esempio su dei contatori che identificano gli eventi stessi.

Un altro problema legato alle tecnologie esistenti riguarda l'indirizzabilità: ad oggi il protocollo IPv4 non ha più disponibilità di indirizzi validi, per tanto il passaggio ad IPv6 (il protocollo è stato reso disponibile nel 2004 e si prevede che IPv4 continuerà ad essere utilizzato, in parallelo, fino al 2025) sarà una delle prime sfide tecnologiche che IoT dovrà affrontare.

Ad essa vanno ad aggiungersi tutti i problemi legati al discovery dei dispositivi e/o dei servizi: in IoT gran parte dei nodi sono mobili e quindi sarà necessario permettere l'inserimento dinamico degli oggetti in un contesto di esecuzione già operativo, possibilmente evitando la fase di configurazione iniziale. I nodi dovranno inoltre essere in grado di trovare servizi in grado di soddisfare le loro richieste, anche basati sulla loro posizione.

[3] In ambito IoT i classici sistemi di sicurezza (ad esempio SSL) potrebbero risultare troppo gravosi per i dispositivi.

1.2 Servizi di Discovery

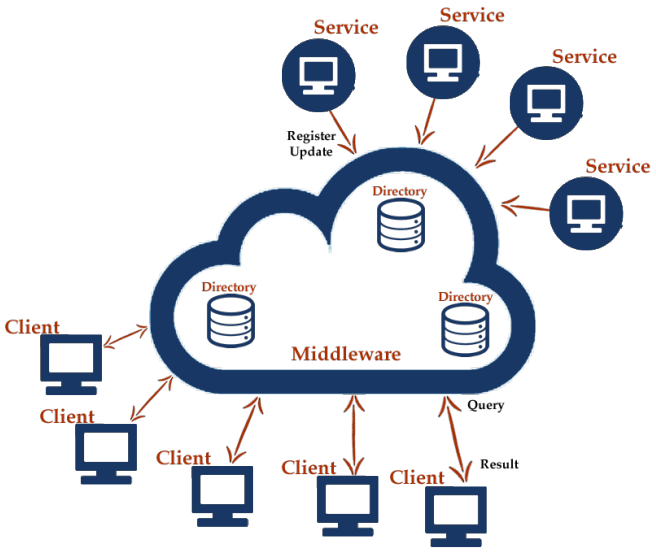
1.2.1 Elementi introduttivi

Introducendo IoT si è detto che risulta necessaria la presenza di un middleware che metta a disposizione dei nodi connessi una serie di servizi di supporto in grado di consentire l'integrazione di sistemi tra loro eterogenei.

Quando si parla di discovery si fa riferimento ad un particolare servizio offerto dal middleware che fornisce supporto alla localizzazione, alla configurazione e all'accesso a risorse e funzionalità messe a disposizione da un dispositivo e/o disponibili in una determinata località geografica. Questo supporto risulta utile in ambienti dinamici in cui è impensabile che un nodo conosca tutte le risorse di ogni località che visita.

In Figura 1.4 viene mostrato un modello generale per servizi di discovery: i servizi si registrano presso il middleware, mentre i clienti, per poter utilizzare i servizi, dovranno prima trovarli inviando una richiesta al livello di supporto che restituirà una lista dei servizi che ha a disposizione in grado di soddisfare le richieste del cliente.

Figura 1.4: Modello di middleware per servizi di discovery



In fase di registrazione il servizio deve specificare un nome che avrà il compito di descriverlo, ed alcune informazioni aggiuntive che forniranno maggiori dettagli sulle caratteristiche offerte. Le informazioni potranno essere aggiornate in un secondo momento attraverso dei meccanismi di update nel caso vengano apportate modifiche.

Quando il middleware riceve una richiesta di registrazione memorizzerà le informazioni del servizio in una struttura dati, che prende il nome di directory, insieme ad eventuali altre informazioni che descrivono il servizio. Tali strutture dati possono essere centralizzate (un'unica directory contenente tutti i servizi regi-

strati), oppure distribuite (ogni directory memorizzerà tutto il name space[4], creando quindi una duplicazione delle informazioni, o solo una porzione di esso).

Il nome del servizio e le query dei clienti hanno una sintassi simile: possono essere stringhe, un insieme di attributi, file XML, etc.

Alla ricezione delle richieste il middleware le confronterà con i nomi dei servizi registrati, restituendo al client una lista contenente quelli che fanno match – totale o parziale – con gli attributi richiesti dal cliente. Ricevuta la lista dei servizi disponibili (o il servizio migliore), il cliente potrà iniziare ad interagire con essi, sia in modo diretto che attraverso il middleware.

1.2.2 Problematiche introdotte dai sistemi mobili e dagli ambienti pervasivi

I servizi di discovery sono di fondamentale importanza quando si ha a che fare con sistemi mobili e distribuiti: nodi mobili che raggiungono una certa località non conoscono a priori i servizi offerti, dovranno quindi fare affidamento su un servizio di discovery per scoprirli e accedervi dinamicamente. In questi contesti, oltre alle caratteristiche generali che il sistema di discovery dovrà avere, bisogna tenere in considerazione anche gli ulteriori requisiti introdotti dalle caratteristiche dei dispositivi: mobilità, scalabilità, context awareness e minimizzazione delle comunicazioni di rete.

La mobilità è la caratteristica principale dei sistemi mobili. Durante la progettazione è fondamentale considerare che i nodi possano uscire da un determinato ambiente ed entrare in un altro. Il movimento del device causa principalmente tre problemi:

- aggiornamento dell'indirizzo e conseguente interruzione della comunicazione tra cliente e servitore
- possibile necessità di registrazione in una nuova directory nel middleware (questo processo prende il nome di directory handoff)
- possibile cambiamento del nome del dispositivo in modo che questo possa riflettere la sua nuova locazione.

Il middleware può gestire queste problematiche in due differenti modalità: notificando il movimento di un nodo a quelli ad esso collegati lasciando loro la gestione (questa soluzione accresce la complessità di clienti e servitori) o trattando la mobilità in modo trasparente a clienti e servitori; in questo caso ad aumentare è la complessità del middleware.

La scalabilità è la capacità che ha un sistema ad adattarsi ad un elevato numero di richieste. Questo è di per sé un problema generale dei servizi di discovery, ma quando si lavora in ambito mobile o in IoT la sua importanza aumenta. In questi contesti il sistema potrebbe avere a che fare con un elevato numero di richieste da parte di clienti che vogliono localizzare un servizio, e allo stesso tempo con servizi che vogliono registrarsi. Per fare in modo che queste richieste possano essere gestite con una buona qualità di servizio (QoS) non si potrà fare affidamento su un'unica directory centralizzata, ma sarà necessario partizionare lo

[4]Per name space, o spazio di nomi, si intende l'insieme dei nomi di tutti i servizi registrati presso un servizio di nomi.

spazio di nomi in sotto-spazi. La suddivisione più tipica è quella che prevede il partizionamento dell'area di copertura del servizio in più località, ad ognuna delle quali sarà associata una directory distribuita che si occuperà di registrare servizi e rispondere alle richieste dei clienti inoltrando, se necessario, la richiesta ad un'altra directory. In questo modo il sistema è in grado di gestire un maggior numero di clienti e di servizi, ma questo comporta anche il fatto che per soddisfare la richiesta di un cliente potrebbe essere necessario interrogare più directory, causando un aumento dei tempi di latenza.

Il context-awareness è una proprietà dei sistemi mobili che definisce l'abilità di rilevazione dei cambiamenti dell'ambiente di esecuzione, sia fisico che computazionale. Questa proprietà assume un ruolo centrale soprattutto in scenari di pervasive computing e incide sul servizio di discovery che dovrà fornire al cliente il servizio migliore che soddisfi le sue richieste tenendo in considerazione, oltre al contenuto dell'invocazione, anche altre caratteristiche non strettamente dipendenti dal tipo di servizio, come ad esempio la distanza tra cliente e servitore, il numero di richieste che il servitore ha in coda, ecc.

La minimizzazione delle comunicazioni risulta doverosa poiché queste rappresentano una delle maggiori fonti di consumo energetico[9]: i dispositivi mobili, specialmente in ambienti pervasivi dove le risorse dei device sono minime, hanno un'autonomia limitata. Riducendo le trasmissioni di rete si punta a massimizzare la durata delle batterie.

1.2.3 Protocolli standard

L'industria e la ricerca, da anni, svolgono studi approfonditi per la realizzazione di servizi di autoconfigurazione. Jini[10], Universal Plug and Play[11] ed il Service Location Protocol[12] sono i pionieri di questi servizi. Essi sono stati concepiti con l'idea di fornire un accesso semplificato alle risorse evitando all'utente il processo di configurazione, ma il reale potenziale di questi servizi viene messo in luce dal loro utilizzo in ambito distribuito e mobile[13] .

La mobilità consente ad un dispositivo di uscire da un contesto noto e precedentemente configurato e di entrare in ambienti nuovi e sconosciuti in cui i servizi di discovery, che possono essere visti come dei servizi di nomi locali, risultano fondamentali per la scoperta, l'autoconfigurazione, e l'accesso a servizi e risorse disponibili nella nuova località.

1.2.3.1 Jini (Apache River)

Jini è la soluzione di discovery principe in ambiente Java.

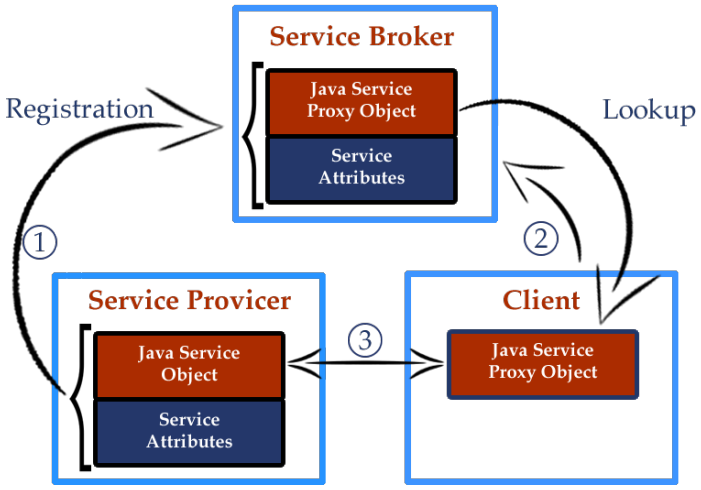
L'idea centrale di questa soluzione è la presenza di oggetti Java che offrono servizi, chiamati Service Provider. Questi si registreranno presso un oggetto di metalivello, il Service Broker, fornendogli un nome logico, una serie di attributi che descrivono il servizio ed un oggetto proxy (Java Service Proxy Object o JSPO[5]) che fungerà da stub per l'accesso alla risorsa.

[5]Il Java Service Proxy Object risulta essere più dinamico di una chiamata a procedura remota (RPC): consente di ottenere lo stub a runtime e di utilizzare il servizio ignorandone i dettagli.

Quando un cliente entra in una località effettuerà un lookup sul Service Broker per poter ottenere un determinato servizio, descrivendo ciò di cui si ha bisogno attraverso attributi. Il broker eseguirà una ricerca sui servizi registrati e, nel caso in cui dovesse trovarne uno che fa match con quanto specificato dal cliente risponderà alla sua richiesta con l'oggetto proxy.

Ottenuto il JSPO il cliente potrà conoscere l'interfaccia del servizio facendo reflection, attraverso attributi che la descrivono come in UPnP, o perché l'interfaccia è standardizzata. In Figura 1.5 è mostrata l'architettura di Jini.

Figura 1.5: Architettura di Jini



Affinché il servizio di discovery funzioni è necessario che client e servizi trovino il Service Broker. La ricerca avviene in due differenti modalità: nelle reti locali (LAN) si utilizza il Multicast Discovery: quando l'entità viene attivata utilizza il protocollo Multicast Request per cercare attivamente un broker. Trascorso un certo intervallo di tempo la ricerca attiva termina e l'entità si mette in ascolto di pacchetti Multicast Lookup Announce utilizzando il protocollo Multicast Announce Protocol. Se invece bisogna cercare un Service Broker in una rete geografica sarà necessario specificare l'indirizzo del broker a cui ci si vuole collegare, utilizzando l'Unicast Discovery[14].

Una particolarità di Jini riguarda la presenza di un meccanismo di lease che consente la realizzazione di un sistema di garbage collection distribuito: sia la registrazione di un servizio presso il broker che il JSPO hanno un periodo di validità terminato il quale, se la registrazione non è stata rinnovata, il servizio verrà cancellato e le risorse che occupava verranno liberate. L'utilità di questo meccanismo è quella di non dover esplicitare la rimozione del servizio o la chiusura della connessione verso di esso da parte di un client quando il nodo esce dalla località senza avere il tempo di chiudere la connessione (ad esempio in caso di perdita di connessione). Quando ciò avviene le tabelle del broker saranno parzialmente disallineate. Per questo motivo la definizione del lease ha un'importanza non trascurabile: un lease breve consente di mantenere stato consistente ma aumenta notevolmente il traffico di rete associato al rinnovo del lease, mentre lease più lunghi riducono il traffico di rete ma mantengono il sistema inconsistente per maggior tempo.

La presenza del Service Broker limita la scalabilità di Jini: in presenza di un elevato numero di client e Service Provider che inviano un elevato numero di richieste di lookup o di registrazione, il Service Broker potrebbe rappresentare un collo di bottiglia. Questo limite è tuttavia superabile grazie al fatto che ogni località potrà avere più di un Service Broker, e che i Service Provider possono registrarsi presso più broker garantendo in questo modo anche tolleranza ai guasti.

I Service Broker, inoltre, potranno registrare se stessi presso altri broker implementando nel Java Service Proxy Object la logica di business per accedere ai servizi della loro località.

1.2.3.2 Universal Plug and Play (UPnP)

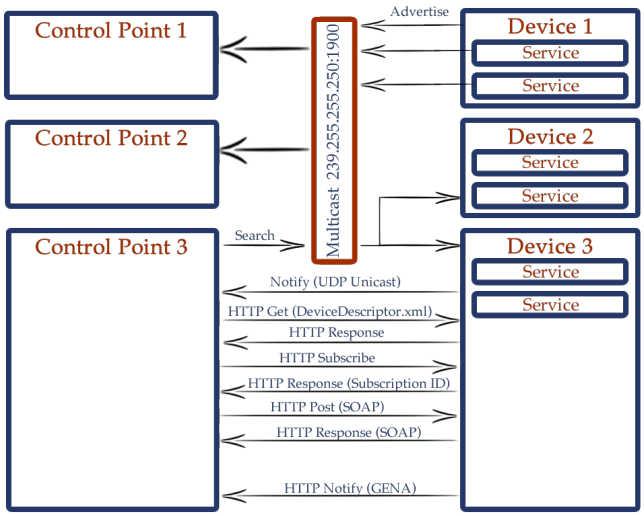
L'implementazione di Universal Plug and Play nasce dalla necessita di estendere il meccanismo Plug and Play di Microsoft, ampiamente utilizzato nei sistemi Windows, in ambienti distribuiti e si propone come uno strumento per individuare, identificare ed utilizzare in modo automatico e senza necessità di configurazioni risorse connesse alla rete. Ad oggi risulta essere la soluzione più utilizzata in ambito consumer e domestico.

UPnP si basa su UDP, TCP/IP, HTTP, XML, HTML e SOAP. Questi protocolli sono ampiamente diffusi e garantiscono indipendenza sia dal linguaggio utilizzato che, soprattutto, dal sistema operativo, ma possono risultare pesanti considerando che lo standard viene utilizzato prevalentemente in dispositivi con basse capacità computazionali.

La soluzione proposta da Universal Plug and Play supporta l'autoconfigurazione: se nella rete a cui viene collegato il dispositivo non c'è un DHCP server viene utilizzato il protocollo AutoIP per ottenere un indirizzo localmente valido. Il protocollo assegna al dispositivo un indirizzo IP reclamandolo da un range, a seguito di un controllo con ARP se questo risulta essere già occupato se ne estrae un altro.

In Figura 1.6 è illustrata una tipica interazione tra Control Point (i client che richiedono un servizio) e device che utilizzano UPnP per fare discovery:

Figura 1.6: Interazione UPnP tra Control Point e device



Ottenuto l'indirizzo, il dispositivo può iniziare a cercare risorse e servizi disponibili nella sua località. In questo caso non c'è necessità di avere un broker: UPnP, per localizzare risorse e dispositivi, utilizza il protocollo SSDP (Simple Service Discovery Protocol), basato su multicast UDP. Control Point e dispositivi che offrono servizi sono in ascolto sull'indirizzo `udp://239.255.255.250:1900`. Su questo indirizzo i device faranno advertising dei loro servizi; questi messaggi saranno memorizzati dai Control Point.

Quando i Control Point avranno bisogno di utilizzare una risorsa, analizzeranno gli advertisement ricevuti; nel caso in cui non trovino una risorsa che soddisfi le richieste verrà inviato un messaggio SSDP-Discovery (`search`) sull'indirizzo multicast. Il messaggio verrà ricevuto dai dispositivi che risponderanno (in unicast UDP al Control Point che ha inviato la richiesta) solo nel caso in cui possano offrire il servizio richiesto. La risposta conterrà l'URL di un file che descriverà il dispositivo.

Ricevuta la risposta dal device, il Control Point scaricherà la descrizione del dispositivo con una HTTP Request di tipo GET.

Nel file ottenuto saranno specificati un identificativo per il servizio, e tre URL per:

- accedere alle funzionalità del dispositivo, invocabili con messaggi SOAP sopra HTTP
- effettuare una sottoscrizione ad eventuali notifiche o eventi
- ottenere un file che conterrà le informazioni specifiche del servizio:
  - la sintassi per accedere al servizio offerto
  - la tabella di tutte le variabili di stato

Il Control Point, dopo aver invocato il servizio sul dispositivo con un messaggio SOAP, riceverà –al termine dell'esecuzione– un altro messaggio SOAP contenente i risultati dell'elaborazione.

Nel caso in cui un Control Point si dovesse registrare per ricevere notifiche, riceverà dal dispositivo un messaggio XML, formattato secondo lo standard GENA, contenente le variabili di stato che hanno subito una modifica rispetto all'ultima notifica ricevuta. Per ridurre la dimensione del messaggio il dispositivo tiene traccia di tutti i suoi sottoscrittori e dell'ultimo aggiornamento di stato che questi hanno ricevuto. Inoltre, per ottimizzare la comunicazione degli aggiornamenti, quando il dispositivo conferma la sottoscrizione invia al Control Point il suo Subscription ID e la tabella delle variabili di stato aggiornata all'istante della sottoscrizione.

La soluzione offerta da UPnP, come detto, è la più utilizzata in ambito consumer e domestico. Questo è dovuto al fatto che offre un servizio semplice e dinamico, utilizza protocolli già esistenti e ampiamente diffusi, è di facile realizzazione in ambienti con poca conoscenza pregressa e non ha elementi di infrastruttura. Quest'ultimo punto, tuttavia, può rappresentare anche un limite in termini di scalabilità: in presenza di un numero elevato di device e Control Point che interagiscono tra loro con multicast locali queste interazioni avranno un costo elevato.

Un altro punto a sfavore dei UPnP è senza dubbio l'utilizzo di protocolli e formati pesanti: il Device Descriptor ed il Service Descriptor, che conterranno rispettivamente la descrizione del device e dello specifico servizio, sono file XML, quindi ben strutturati e indipendenti dal linguaggio e dal sistema operativo, ma allo stesso tempo verbosi e pesanti da interpretare.

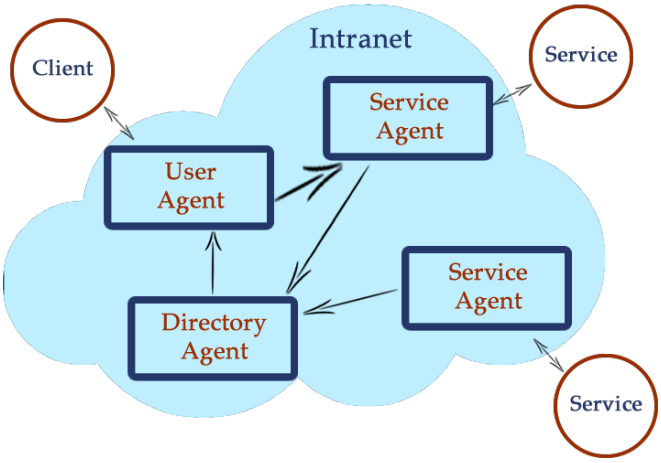
Questi file, inoltre, sono contenuti in una porzione di file system all'interno del dispositivo stesso. Affinché i device possano renderlo disponibile ai vari Control Point che vorranno scaricarlo dovranno avere al loro interno un server HTTP; quest'ultimo sarà utilizzato anche per l'interazione con i messaggi SOAP che il Control Point utilizzerà per interagire con il dispositivo.

1.2.3.3 Service Location Protocol (SLP)

Il Service Location Protocol è uno standard definito dalla Internet Engineering Task Force (IETF) in grado di gestire il discovery e l'advertising dei servizi senza specificare alcun protocollo per la comunicazione tra cliente e servitore. SLP ha la particolarità di essere totalmente dipendente dal protocollo IP, e basa il suo funzionamento sugli agenti che rappresentano servizi e clienti.

Come mostrato in Figura 1.7 servizi e client, non interagiscono in modo diretto, ma attraverso gli agenti che li rappresentano:

Figura 1.7: Service Location Protocol: architettura



Gli agenti previsti dal protocollo possono essere di tre tipologie:

Service Agent: rappresenta l'entry point per i servizi. Il servizio comunicherà all'agente la sua intenzione di registrarsi e quest'ultimo userà SLP per fare broadcast di advertisement dei suoi servizi e delle sue risorse.

User Agent: è il punto d'accesso del cliente: l'agente si occuperà di farà il lookup e interagire con i Service Agent.

Directory Agent: è una entità opzionale, sta in ascolto delle operazioni tra User Agent e Service Agent e manterrà una cache delle informazioni di naming già propagate.

L'idea alla base di questo standard è che sarà la risorsa a comunicare la disponibilità del servizio che offre, e non il cliente a cercarlo: un dispositivo che vuole mettere a disposizione un servizio dovrà comunicare questa decisione al Service Agent che periodicamente farà broadcast degli advertisement.

Lo User Agent non farà broadcast ma invierà la sua richiesta ad un indirizzo multicast su cui sono in ascolto tutti i Service Agent che risponderanno in unicast all'agente del client.

Potendo esserci più servizi in ascolto sul multicast una richiesta dello User Agent potrebbe essere ricevuta ed elaborata da più servizi che invieranno le loro risposte. Per questo motivo lo User Agent non inoltrerà la prima risposta che riceverà al client, ma attenderà per un certo periodo, terminato il quale analizzerà le risposte ricevute in modo da restituire al cliente quella ritenuta più appropriata[6].

I Directory Agent sono dei componenti opzionali e fungono, se presenti, da repository centralizzato. Fanno ascolto in broadcast, in questo modo memorizzano gli advertisement dei Service Agent, e potrebbero ricevere le richieste degli User Agent nel caso in cui questi non conoscano gli indirizzi dei Service Agent. In quest'ultimo caso i Discovery Agent inoltreranno la richiesta al Service Agent più adatto, e ottenuta la risposta la comunicheranno al richiedente.

Questa soluzione, rispetto alle precedenti, è molto meno diffusa probabilmente a causa della sua complessità dovuta principalmente ai tre elementi di infrastruttura, alla complessità dello User Agent (che ha anche il compito di selezionare la risposta migliore), e al fatto che il Discovery Agent in presenza di Service Agent multipli e disallineati ha un comportamento complesso. Un altro fattore che gioca contro SLP è il fatto che sono poche le reti che supportano il multicast a livello di più località e per questo motivo è utilizzabile solo in locale.

[6]La scelta della risposta migliore viene pilotata attraverso delle preferenze che il cliente può esprimere attraverso API standard.

1.3 Discovery in Internet of Things

1.3.1 Scenario attuale

Quando si ha a che fare con ambienti altamente dinamici, densi e distribuiti come IoT, la presenza di servizi di discovery in grado di trovare risorse e conoscere le loro funzionalità risulta fondamentale[15]. Tuttavia l'utilizzo dei servizi standard analizzati in precedenza potrebbe risultare eccessivamente costosa in termini di risorse o di tempo.

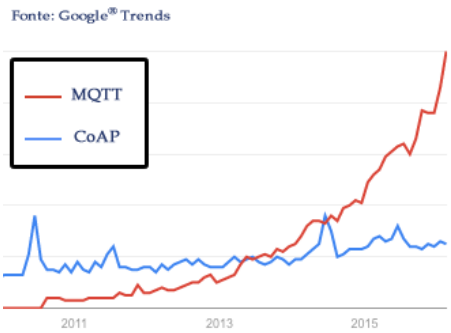
L'utilizzo di UPnP comporterebbe che ogni smart device abbia al suo interno un web server e scambi con i client messaggi verbosi e pesanti da elaborare. Se invece si decidesse di utilizzare Jini, in un ambiente così denso, per evitare problemi di scalabilità potrebbe essere necessaria la presenza di un elevatissimo numero di Service Broker che andrebbero a rendere ancora più densa la rete e rallenterebbero il processo di discovery nel caso in cui un dispositivo dovesse richiedere un servizio registrato su un Service Broker distante.

Queste problematiche stanno portando l'attenzione verso una nuova serie di servizi di discovery progettati ad-hoc per lavorare in ambienti altamente dinamici e in cui le capacità computazionali dei nodi sono molto limitate.

1.3.2 Obiettivi della tesi

Uno dei protocolli maggiormente utilizzati da IoT, che negli ultimi tempi sta avendo un grande successo nonostante sia tra i più recenti, è MQTT: stando agli andamenti di Google Trend l'interesse verso questo protocollo è addirittura quattro volte maggiore di quello per il più datato CoAP.

Figura 1.8: Google Trend: MQTT vs CoAP



L'importanza dei servizi di discovery nello sviluppo di IoT, e l'elevato interesse nei confronti di questo nuovo protocollo, nato per lavorare in situazioni proibitive, sono alla base di questo progetto di tesi che si pone l'obiettivo di realizzare un servizio che offra il supporto al discovery di gateway IoT basati su framework Kura, e che, grazie all'utilizzo di MQTT come sistema di messaggistica, possa essere utilizzato da dispositivi dalle ridotte capacità computazionali.

Il servizio che si andrà ad implementare dovrà:

- consentire ai dispositivi di trovare il gateway più vicino al dispositivo
- autoconfigurare lo smart device per accedere alla rete WiFi messa a disposizione dal gateway
- avviare la comunicazione diretta tra gateway e dispositivo

1.3.3 Lavori correlati

In letteratura sono presenti molti servizi di discovery progettati per lavorare con i dispositivi della Internet of Things. Di seguito sono esposte le soluzioni proposte.

1.3.3.1 Servizi di discovery distribuiti e P2P

Paganelli e Parlanti hanno presentato un sistema di discovery distribuito basato su un approccio peer-to-peer che sfrutta tecniche DHT[7][16]. La loro soluzione supporta attributi multipli e range query[8].

La proposta di Liu et al., invece, rappresenta un'architettura per il discovery di risorse distribuite (Distributed Resource Discovery, DRD)[17]. I nodi risorsa distribuiti comunicano tra loro sfruttando la rete P2P e gestiscono il meccanismo di registrazione Machine-to-Machine: l'approccio degli autori è stato quello di utilizzare un URI CoAP-based in cui l'univocità dei nomi è garantito dall'utilizzo di una funzione di hash sul MAC address del dispositivo. Attraverso la registrazione vengono memorizzate nella rete P2P molte informazioni relative alla risorsa (indirizzo IP, path e tipologia della risorsa, tipo di contenuto ed il nome dell'end point). I client che vogliono trovare una risorsa utilizzano dei Resource Discovery Component (RDC) che la cercheranno localmente e, nel caso in cui la ricerca non abbia esito positivo verrà estesa alla rete P2P.

Una terza soluzione peer-to-peer[18] prevede la realizzazione di un'architettura per il service discovery in cui i gateway IoT ne costituiscono la dorsale: i gateway tengono traccia di qualsiasi device entri o esca dal suo network aggiornando una lista di dispositivi memorizzata sul suo server CoAP. L'architettura è composta da due reti P2P sovrapposte: una per i servizi locali, ed una per quelli geografici.

1.3.3.2 Architettura centralizzata

Un esempio di architettura centralizzata è quella proposta da Jara et al.[19], basata su un'infrastruttura centralizzata (chiamata "Digcovery") che consente ai sensori di essere registrati, e su un servizio mobile che permette ai client di scoprire ed accedere ai sensori. L'architettura utilizza diverse "Digrectory", ognuna delle quali è associata ad un particolare dominio; le risorse si connetteranno alla Digrectory relativa al loro dominio. Il servizio mobile consente di sfruttare la geolocazione e la context-awareness durante il processo di discovery.

Una soluzione differente propone invece un framework service-oriented che sfrutta standard web (REST e JSON)[20]. Il framework è integrato in un'architettura centralizzata dove un registro centrale si occupa di indicizzare gli smart-object in base al loro dominio di appartenenza. La ricerca delle risorse avviene

[7] Distributed Hash Table, tipicamente utilizzate per sistemi di file-sharing peer-to-peer come Torrent o eMule.
[8] Per range query gli autori intendono richieste che rispecchino dei limiti specificando upper e lower bound.

interrogando il registro centralizzato che fornirà al cliente il riferimento diretto all'oggetto desiderato. Il problema principale di questa soluzione deriva dal fatto che è basata su domini applicativi e non considera che uno stesso oggetto può essere condiviso da più domini.

1.3.3.3 Servizi CoAP-based

CoAP[21] include un meccanismo di discovery che espone un web-service RESTful all'indirizzo `/.wellknown/core` che risponde ai client CoAP che richiedono un determinato servizio. I clienti ricevono molte informazioni, compresa una lista di tutte le risorse disponibili e un attributo che specifica il formato dei metadati delle risorse. Nonostante sia utile in molti scenari presenta alcune imperfezioni, come il fatto che non venga specificato come un client possa accedere al server ed annunciarsi come risorsa, o come un client possa consultare la directory delle risorse; inoltre l'utilizzo di una resource directory centralizzata e di CoAP può risentire di problemi di scalabilità.

Ishaq et al. illustrano le maggiori lacune dei sistemi IoT-based e propongono un meccanismo di auto-configurazione ed avvio automatico che utilizza CoAP e DNS[22]. La loro soluzione, traducendo il protocollo CoAP in HTTP, rende individuabile ogni sensore abilitato all'utilizzo di IPv6.

1.3.3.4 Discovery semantic-based

Alcune soluzioni di discovery propongono l'utilizzo di meccanismi basati sulla semantica e sulle ontologie. Zhou e Ma, ad esempio, hanno utilizzato un'ontologia che rappresenta sensori veicolari per valutare un algoritmo di web-service matching in grado di calcolare somiglianze e relatività semantiche al fine di trovare il miglior web-service disponibile[23].

Alam e Noll, invece, introducono un framework semantic-based che utilizza il concetto di advertisement da parte degli smart-object[24]. Questo meccanismo, secondo gli autori, rende la registrazione di servizi più semplice e, a sua volta, semplifica il processo di discovery. La registrazione conterrà i metadati dei servizi offerti (nome, id, end-point, posizione, etc.).

Un'altra proposta suggerisce l'utilizzo di un middleware che svolge il servizio di Service Directory usando tecnologie legate al semantic web sulle informazioni del contesto applicativo dedotte dai dati dei sensori[25].

Simon Mayer e Dominique Guinard, invece, analizzano il discovery dal punto di vista della Web of Things[26] ed utilizzano schemi di mapping multipli (chiamati "Discovery Strategies"). La loro proposta utilizza Microformats e Microdata con le tecnologie di semantic web su web service RESTful, e sfrutta JSON per garantire l'interoperabilità delle risorse. La maggiore limitazione di questo approccio è che le risorse devono essere connesse al web, altrimenti non possono essere individuabili.

1.3.3.5 Motore di ricerca ibrido per resource discovery

Ding et al. propongono una via alternativa al resource discovery: da una loro analisi si evince che vi è un lavoro molto limitato per quanto riguarda motori di ricerca in IoT[27], e le soluzioni esistenti non supportano ricerche multi-modali, come ad esempio quelle spazio-temporali, basate su valori o su keyword. La loro

proposta è suddivisa in tre layer: il primo monitora gli smart device e memorizza nello storage layer le informazioni che sensori e dispositivi producono. Lo storage layer è composto da diversi Raw-Data Storage, ognuno dei quali è in grado di gestire un elevato volume di informazioni. Il terzo livello, l'index layer, gestisce tre indicizzazioni: una per parole chiave, una spazio-temporale ed una per valori.

L'obiettivo principale di questa proposta è quello di realizzare un motore di ricerca per query multimodali che sia in grado di fornire dati generati da smart things in tempo reale, ma uno dei maggiori problemi è dovuto al fatto che il livello più basso della soluzione genera informazioni non strutturate e incentrate sui dati dei sensori anziché sui dispositivi.

1.3.3.6 Object Name Service e Domain Name Service

La soluzione proposta da Minbo, Zhu e Guangyu[28] realizza di un servizio di discovery per un sistema informativo agricolo utilizzando un Object Name Service (ONS). Ogni prodotto è dotato di un codice di monitoraggio, ed ONS provvede a fornire un mapping tra questo codice e l'indirizzo della risorsa IoT. Il servizio di lookup è composto da un risolutore del codice di monitoraggio ed un DNS che memorizza le informazioni rilevanti; poiché i prodotti seguono diverse filiere produttive gli autori hanno introdotto un servizio di discovery aggiuntivo che consente di mappare gli ID degli oggetti, o i loro codici, in una lista di server che offrono servizi IoT.

Il servizio presenta alcuni difetti evidenziati da Ishaq[22], come l'assenza di discovery automatico per i sensori, l'integrazione con il DNS, ed il fatto che dai web browser non si potesse accedere ai sensori in modo semplice; per attenuare il problema gli autori hanno proposto un meccanismo che abilitasse l'auto-configurazione e l'avvio automatico dei sensori sfruttando UPnP. L'utilizzo congiunto di UPnP e di OSN è utilizzato, inoltre, per il discovery di dispositivi integrati[29].

Capitolo 2

Tecnologie utilizzate

2.1 Message Queue Telemetry Transport

2.1.1 Cosa è MQTT

Il Message Queue Telemetry Transport (MQTT) è un protocollo di messaggistica leggero basato sul pattern publish-subscribe, utilizzato sopra il protocollo TCP/IP. È progettato per connessioni verso nodi remoti con scarsa potenza di calcolo in situazioni di "banda limitata", che dovrà quindi avere un "footprint limitato". La specifica MQTT[30] non definisce il significato di "banda limitata" e di "footprint limitato", quindi l'utilizzo del protocollo dipende dal contesto.

Esiste, inoltre, una seconda versione di questo protocollo chiamata MQTT-SN[31] (MQTT for Sensor Networks), progettata per dispositivi embedded connessi in una rete non-TCP/IP, come ad esempio ZigBee.

Originariamente sviluppato da IBM ed Eurotech, oggi il protocollo MQTT è il core del Progetto Eclipse Paho[1] ed è utilizzato, ad esempio, da Facebook nella sua app Messenger.

Nel 2013 la OASIS[2] ha adottato la specifica come protocollo standard per IoT con una particolare licenza che accetta solo cambiamenti che non vadano a modificarne il core.

Il modello publish-subscribe richiede la presenza di un message broker, il cui compito è quello di distribuire i messaggi a tutti i client che si sono registrati al topic di quel messaggio. La sottoscrizione ai topic consente di evitare l'ovvio problema che ogni client riceva ogni messaggio pubblicato sul broker da ogni altro client.

2.1.2 Architettura

Dal punto di vista architetturale MQTT è un protocollo message-oriented, in cui interagiscono due diverse tipologie di componenti: i client ed il broker. Ogni dispositivo che intende comunicare con un altro sfruttando il protocollo può essere visto come un client connesso via TCP ad un broker. Il compito di quest'ultimo è

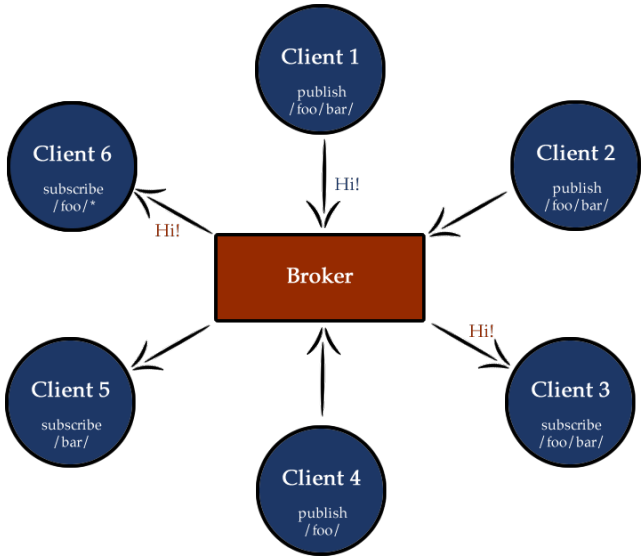
[1] http://eclipse.org/paho/

[2] La Organization for the Advancement of Structured Information Standards è un consorzio globale che lavora sullo sviluppo, la convergenza e l'adozione di standard per l'e-business e web services.

quello di distribuire i messaggi ricevuti su uno specifico argomento ai clienti che si erano sottoscritti a quel topic.

In Figura 2.1 è mostrata l'architettura del protocollo.

Figura 2.1: Architettura MQTT



Affinché un broker sia conforme alla specifica deve soddisfare le seguenti affermazioni:

1. il formato di tutti i pacchetti di controllo che il broker invia devono corrispondere ai formati descritti nella specifica ai capitoli 2 e 3;
2. segue le regole di matching dei topic descritte nella specifica al capitolo 4, sezione 7;
3. soddisfa tutti i requisiti di livello MUST definiti nella specifica, fatta eccezione per quelli relativi ai client;

Le implementazioni di broker MQTT sono numerose, ognuna con caratteristiche diverse. Alcune di esse realizzano anche funzionalità al disopra dello standard.

Tra tutte una tra le più popolari è Mosquitto, che recentemente è entrato a far parte dei progetti della Eclipse Foundation. Questo broker è open source, compatibile sia con MQTT che con MQTT-SN, è leggero, scritto in C –in modo tale da poter essere eseguito anche in ambienti che non hanno le risorse necessarie per eseguire una Java Virtual Machine– ed è uno dei broker più semplici da configurare.

L'attuale implementazione di Mosquitto ha un eseguibile di circa 120kB, ed un consumo di appena 3MB di RAM con 1000 client connessi. Offre inoltre la possibilità di fungere da bridge verso altri broker MQTT –comprese altre istanze di Mosquitto– consentendo la creazione di veri e propri network di broker

MQTT. Benché la configurazione di default accetti tutte le connessioni in entrata sulla porta 1883, è possibile configurare l'accesso con autenticazione.

Anche per i client esistono molte implementazioni –che devono soddisfare i requisiti 1 e 3 già visti per il broker[3]– e la Eclipse Foundation ha provveduto a raccoglierle in un progetto (Paho), all'interno del quale è possibile trovare librerie e tool in C e Java, oltre ad implementazioni in Lua, JavaScript, Python e C++ in fase di sviluppo.

Ogni messaggio all'interno del protocollo è una porzione discreta di dati totalmente opaca per il broker. Questi dati verranno pubblicati ad un certo indirizzo, chiamato topic, su cui i clienti possono effettuare sottoscrizioni. Tutti i messaggi pubblicati su un certo topic verranno recapitati solo ai client che si erano precedentemente registrati ad esso.

Il pattern publish-subscribe consente di disaccoppiare nello spazio mittente e destinatario dei messaggi: il publisher consegnerà il suo messaggio al broker, sarà quest'ultimo ad occuparsi della consegna a tutti i clienti che risultano connessi[4] e che hanno sottoscritto lo specifico topic su cui il mittente ha deciso di pubblicare il messaggio, senza la necessità che il produttore li conosca.

2.1.3 Caratteristiche

MQTT, nonostante sia un protocollo leggero e adatto a lavorare in condizioni limite, riesce ad offrire una serie di funzionalità che gli hanno permesso di affermarsi come uno degli standard di riferimento per quanto riguarda Internet of Things, tra cui:

- Topic matching
- Last Will and Testament
- Persistenza
- Sicurezza

2.1.3.1 Topic matching

In MQTT i topic sono strutturati gerarchicamente, come un file system. All'atto della pubblicazione di un messaggio il topic deve essere assoluto e privo di ambiguità, mentre in fase di sottoscrizione è possibile utilizzare delle wildcard per poter osservare la gerarchia completa: il carattere speciale "+" rappresenta un livello della gerarchia, mentre il carattere "#" rappresenta l'intero albero a partire da un certo punto della gerarchia, ad esempio, `a/+/d` fa match con `a/b/d` ma non con `a/b/c/d`, mentre `a/#/d` farà match anche con `a/b/c/d`.

[3] I requisiti di livello MUST che possono non essere soddisfatti sono, ovviamente, quelli specifici per i broker

[4] È possibile specificare ad MQTT di consegnare i messaggi anche a client che sottoscriveranno il topic in momento successivo, ottenendo anche un disaccoppiamento temporale.

2.1.3.2 Last Will and Testament

I client MQTT, in fase di connessione, possono registrare presso il broker una sorta di testamento. Questo messaggio verrà pubblicato dal broker –in un topic, anch'esso specificato nel messaggio di connessione– nel caso in cui il client si disconnetta in modo improvviso. Questi messaggi possono essere utilizzati per segnalare ai sottoscrittori di un determinato topic che un dispositivo non è più disponibile.

2.1.3.3 Persistenza

Il protocollo offre supporto a messaggi persistenti. Quando un client invia un messaggio al broker potrà specificare se desidera che quel messaggio venga mantenuto in modo persistente dal broker e in questo caso il broker provvederà a memorizzarlo. Verrà mantenuto in memoria solo l'ultimo messaggio –dichiarato persistente– ricevuto dal broker da un certo client in un determinato topic. Ad esempio, se i client inviano i seguenti messaggi persistenti:

```
Client 1    -> Message 1 on /foo/bar/  -> Broker
Client 1    -> Message 2 on /foo/bar/  -> Broker
Client 1    -> Message 3 on /foo/      -> Broker
Client 2    -> Message 4 on /foo/bar/  -> Broker
```

il broker memorizzerà solo i messaggi 2, 3 e 4.

Quando un client si sottoscriverà ad un topic, se il broker ha memorizzato messaggi persistenti su quell'indirizzo, questi gli saranno recapitati.

2.1.3.4 Sicurezza

I broker MQTT possono richiedere ai propri client di autenticarsi mediante username e password, e per garantire la privacy, la connessione TCP può essere criptata sfruttando canali SSL/TLS.

2.1.4 I messaggi

La specifica MQTT è suddivisa in tre sezioni: la prima analizza le tipologie di pacchetti, la seconda definisce i dettagli di ogni tipologia di pacchetto mentre la terza descrive come i pacchetti vengono trasferiti tra client e server.

2.1.4.1 Sintassi

Come detto MQTT è un protocollo message-oriented, questo comporta che ogni interazione tra client e broker avverrà mediante messaggi, ognuno dei quali avrà un header di dimensione fissa, uno di dimensione variabile, ed il payload.

L'header fisso è lungo due byte e segue lo schema riportato in Figura 2.2:

Figura 2.2: Fixed header

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

`Message Type` definirà il tipo di comando che si andrà ad inviare con il messaggio secondo l'enumerazione mostrata in Tabella 2.1:

Tabella 2.1: Message Type

Mnemonic	Enumeration	Description
Reserved	0	Reserved
`CONNECT`	1	Client request to connect to Server
`CONNACK`	2	Connect Acknowledgment
`PUBLISH`	3	Publish message
`PUBACK`	4	Publish Acknowledgment
`PUBREC`	5	Publish Received (assured delivery part 1)
`PUBREL`	6	Publish Release (assured delivery part 2)
`PUBCOMP`	7	Publish Completed (assured delivery part 3)
`SUBSCRIBE`	8	Client Subscribe request
`SUBACK`	9	Subscribe Acknowledgment
`UNSUBSCRIBE`	10	Client Unsubscribe request
`UNSUBACK`	11	Unsubscribe Acknowledgment
`PINGREQ`	12	PING Request
`PINGRESP`	13	PING Response
`DISCONNECT`	14	Client is Disconnecting
Reserved	15	Reserved

`DUP` rappresenta un flag con cui è possibile specificare se il messaggio che si sta inviando è un duplicato di un precedente pacchetto di tipo `PUBLISH`, `PUBREL`, `SUBSCRIBE` o `UNSUBSCRIBE` andato perso o non confermato dal relativo ACK. Per avere un acknowledge della consegna da parte del broker il messaggio deve essere trasmesso con livello di QoS maggiore di 0, così come il suo duplicato.

`QoS` specifica il livello di qualità di servizio richiesta per un messaggio di tipo `PUBLISH`. Il campo può assumere valore da 0 a 2 come mostrato in Figura 2.3:

Figura 2.3: Quality of Service

QoS value	bit 2	bit 1	Description		
0	0	0	At most once	Fire and Forget	<=1
1	0	1	At least once	Acknowledged delivery	>=1
2	1	0	Exactly once	Assured delivery	=1
3	1	1	Reserved		

`RETAIN` specifica se il messaggio di tipo `PUBLISH` deve essere mantenuto in memoria persistente e consegnato ai clienti che si sottoscriveranno al topic in un secondo momento (valore 1), o se la consegna deve avvenire solo a chi è sottoscritto al momento della pubblicazione. Un messaggio con flag `RETAIN` settato rimane disponibile anche al riavvio del broker.

Il secondo byte, occupato per intero dal campo `Remaining length` specifica il numero di byte residui all'interno del messaggio, ossia la dimensione totale dell'header variabile e del payload.

L'header variabile non è una prerogativa di tutti i messaggi MQTT, ma solo alcuni di essi che lo utilizzeranno per fornire informazioni aggiuntive. Questa porzione di messaggio, quando presente, è sempre compresa tra l'header fisso ed il payload; conterrà campi ben definiti dalla specifica che saranno presenti nel seguente ordine:

`Protocol Name` è presente nei messaggi `CONNECT`, specifica il nome del protocollo (MQIsdp) codificato in UTF.

`Protocol Version` è composto da 8 bit. Specifica la versione di protocollo utilizzata in un messaggio di tipo `CONNECT`

Flag per messaggi `CONNECT` ha dimensione 8 bit e contiene una serie di flag che consentono di specificare la presenza di particolari parametri di connessione all'interno del messaggio. La struttura di questo campo è mostrata in figura 2.4:

Figura 2.4: Connect flag

bit	7	6	5	4	3	2	1	0
	User Name Flag	Password Flag	Will Retain	Will QoS		Will Flag	Clean Session	Reserved

`Clean Session` se non è impostato, il broker conserverà le sottoscrizioni dei client dopo la loro disconnessione, in questo modo potrà ristabilire tutte le sottoscrizioni precedenti alla disconnessione in modo automatico. Oltre alle sottoscrizioni il broker memorizzerà anche i messaggi ricevuti su quel topic dopo la disconnessione e, quando il client tornerà online provvederà a recapitarglieli.

`Will` specifica la presenza del messaggio Last Will. Se questo flag è settato devono essere presenti anche i flag `Will QoS` e `Will Retain`, il payload, inoltre, deve contenere il `Will Topic` ed il `Will Message`.

`Will QoS` definisce il livello di QoS per il messaggio `Will`

`Will Retain` indica al broker se deve mantenere in memoria persistente il messaggio `Will`

`User Name` e **`Password`** specificano se nel payload sono presenti nome utente e password che il client utilizzerà per autenticarsi.

`Keep Alive` **timer** anch'esso è disponibile per messaggi di tipo `CONNECT`. È un campo di lunghezza 16 bit che definisce l'intervallo massimo tra i messaggi del client, trascorso il quale viene riconosciuto come disconnesso. Si misura in secondi, e può avere un valore massimo di 18 ore. Per rimanere connesso il client dovrà inviare un `PINGREQ`, a cui il risponderà con `PINGRESP`. Trascorso il Keep Alive il broker disconnetterà il client.

`Connect return code` contiene il codice che viene restituito nell'header variabile di un messaggio di tipo `CONNACK`. Ha lunghezza 1 byte e può assumere i seguenti valori:

- 0: connessione accettata
- 1: protocolli incompatibili
- 2: identificativo respinto
- 3: server non disponibile
- 4: autenticazione fallita
- 5: autorizzazione negata
- 6-255: non ancora in uso

`Topic name:` è un campo disponibile (ed obbligatorio) per i messaggi `PUBLISH`. Contiene l'identificativo del canale d'informazione su cui dovrà essere pubblicato il payload. È codificato in UTF.

Il payload è l'effettivo contenuto del messaggio. Anche questa porzione di messaggio, come l'header variabile, non è disponibile per tutte le tipologie di messaggi, ma solo per le seguenti:

`PUBLISH` contiene informazioni application-specific.

`CONNECT` è formato da una stringa UTF-8 obbligatoria (l'identificativo del client) ed altre che conterranno, se i relativi flag sono settati ad 1, nome utente, password, `Will Topic` e `Will Message`.

`SUBSCRIBE` in questo caso il payload è formato da una lista di topic a cui il cliente intende sottoscriversi, specificando per ognuno di essi il livello di QoS desiderato.

`SUBACK` contiene la lista di livelli QoS che gli amministratori del broker consentono di utilizzare.

2.1.4.2 L'identificatore di messaggio

Alle porzioni precedenti, per quanto riguarda messaggi di tipo `PUBLISH`, `PUBACK`, `PUBREC`, `PUBREL`, `PUBCOMP`, `SUBSCRIBE`, `SUBACK`, `UNSUBSCRIBE`, `UNSUBACK`, è obbligatorio, nel caso in cui QoS sia maggiore di 0, inserire anche il Message ID. Questo campo –di lunghezza 16 bit– ha il compito di identificare in maniera univoca ogni messaggio trasmesso da un client in una determinata direzione: un messaggio trasmesso ed uno ricevuto, anche se presentano lo stesso ID sono considerati totalmente differenti.

2.1.4.3 L'ordine dei messaggi

La consegna ordinata dei messaggi può essere condizionata da numerosi fattori, tra cui il numero di messaggi che possono lasciare il client senza che l'arrivo dei precedenti al broker sia confermato (in-flight messages), o dal fatto che il client sia single o multi-thread.

Per avere certezza che i messaggi vengano ricevuti in ordine è necessario attendere che la consegna sia portata a termine, aspettando il `PUBACK` o il `PUBREL` da parte del broker. Il numero di messaggi simultanei ha comunque effetto sulla garanzia che si può ottenere: se è possibile inviare un solo messaggio alla volta ogni flusso dovrà essere completato prima di avviare il successivo, quindi si avrà la garanzia che l'ordine sia rispettato, mentre nel caso in cui si inviino più messaggi simultaneamente si potrà fare affidamento solo sul QoS.

2.1.5 Livelli di Qualità di Servizio

MQTT consente ai client di specificare al broker il livello di qualità di servizio per ogni messaggio. Questo consente di definire come i messaggi dovranno essere consegnati. È fondamentale definire propriamente il livello di QoS, poiché determina come client e broker interagiranno. I livelli definiti dalla specifica sono tre:

Livello 0 - At most once: La consegna avviene secondo la strategia best effort del protocollo TCP/IP.

Non c'è nessuna conferma di ricezione del messaggio da parte del broker, quindi un client non potrà mai sapere se quel messaggio è giunto a destinazione o meno e per questo motivo non possono essere previste politiche di reinvio. Client e server cercheranno di consegnare il messaggio senza però attendere una "ricevuta" che ne accerti la riuscita.

Se da un punto di vista non viene assicurato che il messaggio venga consegnato al broker, da quello delle performance rappresenta la soluzione più veloce offerta da MQTT.

Figura 2.5: QoS 0: At Most Once

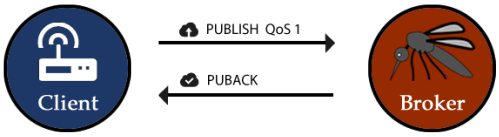
Livello 1 - At least once: Garantisce che il messaggio venga consegnato almeno una volta, ma non esclude che ci siano duplicati. Il broker comunica al client di aver ricevuto il messaggio facendone l'ACK attraverso il messaggio `PUBACK`; questi messaggi conterranno nell'header variabile un identificatore univoco.

La semantica di un messaggio `PUBLISH` con QoS di livello 1 prevede il salvataggio del messaggio da parte del client in memoria persistente[5], l'invio e, una volta ricevuto il `PUBACK`, la cancellazione del messaggio.

Nel caso in cui un messaggio non raggiunga il broker (o vada persa la `PUBACK` che viaggia nella direzione opposta), il messaggio verrà reinviato, specificando che il messaggio che si sta trasmettendo è un duplicato (flag `DUP` settato ad 1).

La ritrasmissione comporta la possibilità di una ricezione plurima da parte del broker di uno stesso messaggio, con conseguente consegna ai subscriber di messaggi duplicati: i messaggi verranno ritrasmessi sia nel caso di perdita del pacchetto che in caso di perdita dell'ACK. In questo secondo caso il mittente, pensando che il messaggio non sia arrivato a destinazione, lo ritrasmette ed il broker si troverà a dover inoltrare un duplicato del messaggio precedente.

Figura 2.6: QoS 1: At Least Once



Livello 2 - Exactly once: È il livello di QoS maggiore e assicura che i messaggi vengano consegnati una ed una sola volta. Questo livello impone un doppio scambio di ACK tra cliente e broker: il mittente, dopo aver salvato il messaggio in memoria persistente (se usata), lo invia al broker con il comando `PUBLISH`; il broker provvederà a salvare a sua volta il messaggio in memoria persistente, quindi ne confermerà la ricezione con il comando `PUBREC` senza iniziare la consegna ai sottoscrittori. Alla ricezione di `PUBREC` da parte del broker il client rimuoverà il messaggio dalla memoria persistente e risponderà con `PUBREL`. Questo messaggio comunicherà al broker che il client ha ricevuto l'ACK e che non invierà di nuovo quel messaggio. A questo punto il broker notificherà la ricezione del `PUBREL` con il pacchetto `PUBCOMP` ed inizierà a distribuire il messaggio ai sottoscrittori.

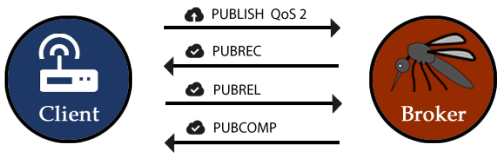
Affinché si sappia sempre a quale messaggio i differenti ACK fanno riferimento, questi dovranno contenere all'interno dell'header variabile l'identificativo del messaggio a cui si riferiscono.

È il livello massimo e quello che consente di avere la certezza che messaggi non arrivino duplicati, ma

[5]MQTT persistence consente la definizione e l'utilizzo di memoria persistente lato client. Benché questa feature non sia descritta nella specifica molte implementazioni di client la mettono a disposizione; essa consente di recuperare i messaggi che erano stati inviati e di cui non si era ricevuto l'ACK dal broker in seguito ad un crash improvviso dell'applicazione.

allo stesso tempo è quello dalle performance inferiori a causa del doppio scambio di ACK.

Figura 2.7: QoS 2: Exactly Once



2.1.6 Conclusioni

Le principali caratteristiche del protocollo MQTT sono:

- il pattern publish/subscribe che consente la distribuzione dei messaggi one-to-many ed il decoupling delle applicazioni
- il livello di trasporto del tutto indipendente dal contenuto del payload
- l'utilizzo del protocollo TCP/IP che fornisce la base per la connettività di rete
- i tre livelli di Quality of Service (QoS) per la consegna dei messaggi

  "At most once": consegna non assicurata

  "At least once": consegna assicurata con possibilità di messaggi duplicati

  "Exactly once": consegna assicurata e senza messaggi duplicati

- il piccolo overhead di trasporto
- la minimizzazione degli scambi di protocollo
- la riduzione del traffico di rete
- il meccanismo di notifica che segnala la disconnessione anomala di un client attraverso il messaggio Will

La versione 3.1 del protocollo introduce nuove funzionalità e ottimizzazioni, tra cui il supporto per UTF-8 e la possibilità di inviare i parametri d'accesso all'interno del messaggio `CONNECT`.

2.2 Kura: un framework OSGi per gateway IoT

2.2.1 I gateway IoT

Affinché dispositivi nati per altri obiettivi possano entrare a far parte di Internet of Things è necessario che questi abbiano un'interfaccia di rete che gli consenta di comunicare con gli altri dispositivi. La soluzione più semplice potrebbe essere quella di dotare i dispositivi di schede WiFi o Ethernet, ma queste soluzioni rischiano di essere troppo costose in termini di risorse. A questa problematica si aggiunge quella relativa al mantenimento e all'elaborazione delle informazioni raccolte dai dispositivi e numerosi altri servizi che risultano essere utili, se non indispensabili, ai device IoT ma che per la limitatezza delle risorse di questi ultimi non possono essere implementati direttamente sugli smart object.

Per far fronte a queste problematiche sono stati realizzati i gateway IoT, che, interposti tra gli smart device e la rete, fungono da punto di accesso ad Internet per quei device che non sono dotati di connettività IP-based ed offrono numerosi servizi.

Gli smartphone che consentono ad oggetti di uso quotidiano divenuti smart (orologi, braccialetti per il fitness etc.) di inviare loro i dati raccolti che verranno memorizzati nel telefono, elaborati da specifiche applicazioni ed eventualmente inviati su cloud, possono essere considerati dei gateway IoT. Le soluzioni realizzabili con gli smartphone tuttavia non sono le migliori quando si vuole monitorare un ambiente e non l'attività fisica o altri aspetti della quotidianità di una persona: se si ha necessità di realizzare una smart house, ad esempio, risulta migliore l'utilizzo di sistemi embedded da installare in modo fisso all'interno dell'abitazione; questi dispositivi sono in grado di analizzare in modo continuativo l'ambiente e regolare il comportamento degli impianti o degli elettrodomestici in autonomia, anche quando i proprietari –e quindi i loro smartphone– sono fuori dalla rete domestica.

Grazie alla diffusione di sistemi Linux-embedded con caratteristiche equiparabili a PC di fascia medio/alta dei primi anni 2000 come Raspberry Pi, Beagle Board, NVIDIA Jetson etc., sono molti i progetti di ricerca che preferiscono utilizzare delle soluzioni open source a dispetto dei gateway IoT forniti dai maggiori vendor del settore, e spesso sono proprio i vendor stessi a rilasciare (parzialmente o totalmente) i sorgenti delle loro soluzioni affinché la community possa trarne beneficio e/o contribuire ad estenderle e migliorarle.

2.2.2 Il framework OSGi[1]

Il modo più semplice per definire OSGi è dire che realizza un layer di supporto alla modularità per la piattaforma Java, dove per modularità si intende la suddivisione del codice in differenti parti logiche, ognuna delle quali rappresenta un diverso problema.

Java fornisce alcuni elementi di modularità sotto la forma object-oriented, ma questi non sono mai stati intesi come un supporto alla programmazione modulare coarse-grained. Nonostante questi elementi, la piattaforma Java presenta alcune limitazioni nella modularità, quali:

- il controllo della visibilità del codice
- il concetto di class path induce ad errori

- il limitato supporto alla gestione e al deploy

Tutte le applicazioni, anche le più semplici, potrebbero avere una di queste problematiche e quindi trarre benefici dalle caratteristiche di modularità offerte dal framework OSGi. Esso infatti può essere di aiuto in svariati contesti, ad esempio:

- risolve la `ClassNotFoundException` dovuta a problemi legati al class path: verifica che le dipendenze siano soddisfatte prima di eseguire il codice
- risolve gli errori a run-time dovuti all'utilizzo di versioni di librerie errate: controlla che il set di dipendenze sia consistente
- l'inconsistenza di tipo, quando si utilizzano classi appartenenti a moduli diversi, non sono più un problema poiché OSGi utilizza uno schema di class-loading gerarchico
- consente di scomporre l'applicazione in pacchetti JAR tra loro indipendenti; è possibile poi eseguire il deploy dei soli pacchetti di cui si ha bisogno
- consente di dichiarare, all'interno di ogni file JAR, quali porzioni di codice sono visibili dall'esterno (API pubbliche) e quali no (API non pubbliche), abilitando di fatto un nuovo livello di visibilità del codice
- offre supporto per l'esecuzione dinamica dei moduli (bundle)

2.2.2.1 L'architettura OSGi

La piattaforma di servizio OSGi è composta da due parti: il framework OSGi ed i servizi standard OSGi: il framework è l'ambiente di esecuzione che implementa ed offre le funzionalità di OSGi, mentre i servizi standard definiscono un insieme riutilizzabile di API per le attività più comuni.

Il framework riveste un ruolo centrale nella creazione di applicazioni OSGi-based e la specifica oltre a definirne il comportamento mette a disposizione una serie di API per la sua realizzazione, lasciando vari gradi di libertà a chiunque voglia implementarne una versione. Questo è dimostrato dalla presenza di svariate implementazioni del framework, tra cui Eclipse Equinox, Apache Felix e Knopflerfish.

Le diversità dei casi d'uso attesta il valore e la flessibilità offerta dal framework OSGi attraverso i tre livelli concettuali definiti nella specifica, ognuno dei quali dipende dal livello inferiore:

- Module layer
- Lifecycle layer
- Service layer

2.2.2.2 Il Module layer

Il Module layer si occupa del packaging e della condivisione del codice, definisce il concetto di modulo OSGi, chiamato bundle, che altro non è che un file JAR con dei metadati extra.

I bundle hanno al loro interno le proprie risorse e le proprie classi e, tipicamente, non sono delle intere applicazioni, ma dei moduli logici che combinati tra loro realizzano l'applicazione. I moduli hanno una maggiore potenza espressiva dei semplici JAR: nei bundle è dichiarato esplicitamente, all'interno del file di manifest, quali tra i package contenuti sono visibili all'esterno (exported packages) e da quali packages esterni il bundle dipende (imported e required packages).

Un ulteriore vantaggio deriva dal fatto che il framework OSGi può verificare la consistenza dei bundle automaticamente attraverso un processo denominato bundle-resolution, e coinvolge la corrispondenza tra exported, imported e required packages che garantisce la consistenza tra diversi bundle rispetto alle versioni ed altri vincoli.

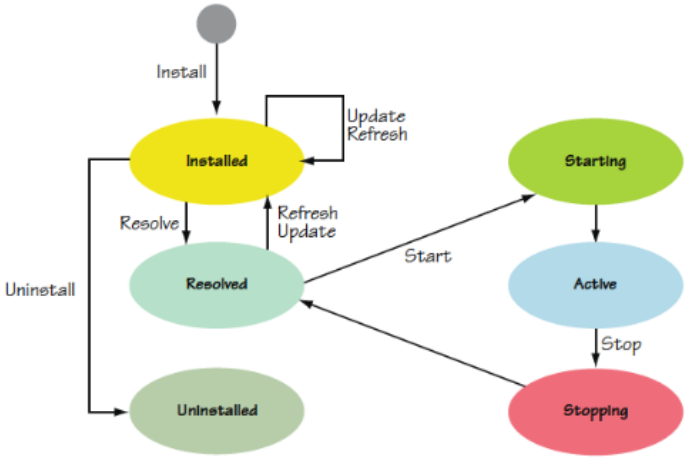
2.2.2.3 Il Lifecycle layer

Il lifecycle layer si occupa della gestione dei moduli a run-time e dell'accesso al framework OSGi sottostante. Definisce come i bundle vengono dinamicamente installati e gestiti dal framework OSGi; questo livello consente di raggiungere due obiettivi:

- definisce le operazioni sul ciclo di vita (i cui stadi sono mostrati in Figura 3.8) consentendo allo sviluppatore di amministrare, gestire e far evolvere l'applicazione in un modo ben definito; questo significa che i bundle possono essere tranquillamente aggiunti o rimossi dal framework senza la necessità di riavviare il processo dell'applicazione

- definisce, internamente all'applicazione, come i bundle ottengono l'accesso al loro contesto di esecuzione. Quest'ultimo fornisce loro un modo per interagire con il framework OSGi ed con i servizi che mette a disposizione a run-time; questo approccio al livello del ciclo di vita consente di realizzare applicazioni gestite dall'esterno e remotamente, applicazioni in grado di autogestirsi totalmente ed ogni loro combinazione

Figura 2.8: Ciclo di vita di un bundle OSGi



2.2.2.4 Il Service layer

Il service layer si occupa della comunicazione tra i vari moduli, e in particolare tra i componenti contenuti al loro interno. Supporta e favorisce un modello di programmazione flessibile, incorporando i concetti diffusi dal service-oriented computing (publish, find e bind), da ancor prima che questo divenisse popolare.

A differenza delle tradizionali Service-Oriented Architecture (SOA) in cui i servizi sono ampiamente legati ai servizi web, in OSGi i servizi sono locali alla virtual machine. Il Service Layer di OSGi promuove un approccio di sviluppo interface-based che favorisce la separazione tra interfacce ed implementazione. I servizi OSGi sono quindi delle interface Java che rappresentano un contratto tra il provider ed il cliente di quel determinato servizio, e questo rende il service layer leggero: i service provider saranno nient'altro che oggetti Java a cui si potrà accedere attraverso una invocazione diretta di metodo. In aggiunta il service layer espande il dinamismo bundle-based del lifecycle layer con il dinamismo service-based: i servizi offerti possono essere nascosti o mostrati in qualsiasi istante.

2.2.3 Eclipse Kura[2]

Kura è un progetto del gruppo di ricerca Eclipse IoT che fornisce una piattaforma open-source per la realizzazione di servizi per gateway IoT. Esso punta a realizzare un container OSGi per applicazioni Machine-to-Machine (M2M) su gateway di servizi. È a tutti gli effetti un application container che consente la gestione remota del gateway e fornisce un'ampia varietà di API che consentono agli sviluppatori di realizzare e distribuire la propria applicazione IoT.

L'implementazione di un dispositivo che funga da nodo sulla IoT è relativamente semplice. Le difficoltà aumentano quando bisogna realizzare un elevato numero di nodi che supportano numerose applicazioni. In questo contesto si inserisce Kura: la piattaforma è eseguita su una Java Virtual Machine e sfrutta OSGi per semplificare il processo di sviluppo e distribuzione di blocchi di codice riutilizzabili; le API che mette a

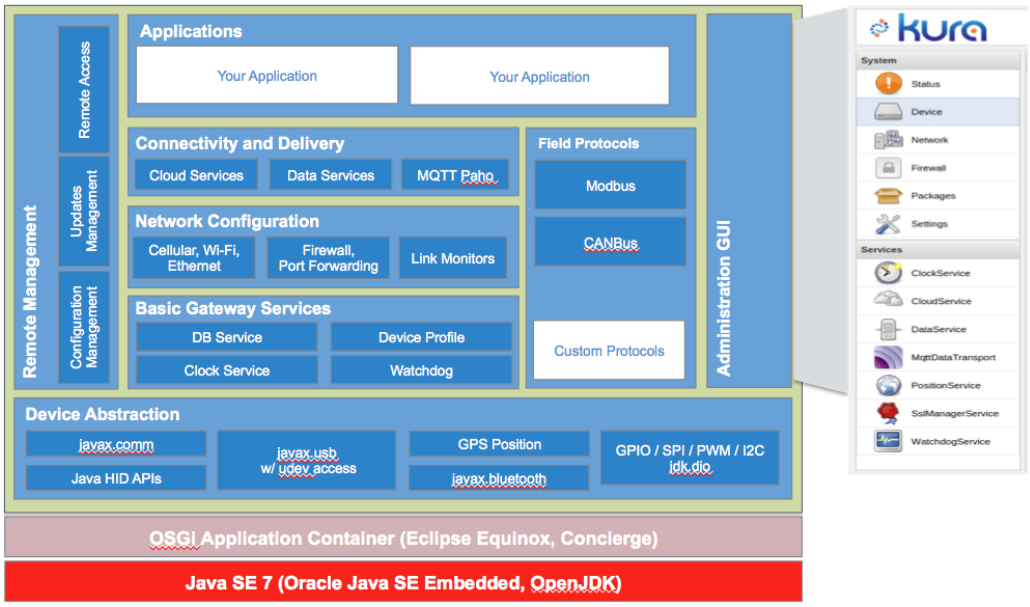
disposizione semplificano l'accesso all'hardware sottostante, tra cui porte seriali, GPS, USB, GPIO e I2C. Kura offre, inoltre, dei bundle che consentono la semplificazione di operazioni di gestione del gateway e della rete, ed altri che facilitano la comunicazione con il cloud.

Uno dei maggiori contributori del progetto è Eurotech –fornitore leader di tecnologie, prodotti e sistemi embedded– che ha reso disponibile un ampio sottoinsieme di Everywhere Software Framework[32]. Kura, come la tecnologia originale, è disponibile per tutti i tipi di dispositivi (da comuni pc fino al Raspberry Pi, passando per dispositivi indossabili, portatili rugged e console per veicoli): essendo basato su OSGi, e quindi su Java, può essere installato su qualsiasi dispositivo Linux-based fornendo un sistema gestibile da remoto, completo di tutti i servizi di cui le applicazioni hanno bisogno ed un'astrazione del dispositivo su cui Kura è in esecuzione in modo da consentire l'accesso all'hardware.

2.2.4 Architettura del framework Kura

Tutti i componenti di Kura sono progettati come servizi OSGi configurabili che espongono API e generano eventi. Per la maggior parte sono completamente realizzati in Java ma una parte di essi, avendo dipendenze dal sistema Linux sottostante, sono invocati attraverso la Java Native Interface[6].

Figura 2.9: Architettura del Framework Kura[33]



In Figura 2.9 è mostrata l'architettura del Framework Kura, che viene rilasciato con i seguenti servizi:

Input/Output: Kura fornisce servizi in grado di semplificare l'accesso ai dispositivi di input ed output. In particolare il Device Abstraction layer fornisce:

[6]La Java Native Interface, o JNI, consente ad una classe Java di interagire con il cosiddetto "codice nativo", specifico di un determinato sistema operativo.

- accesso alla porta seriale attraverso le API `javax.comm` 2.0 o OSGi I/O connection
- accesso alle porte e agli eventi USB sfruttando `javax.usb`, le API HID, ed estensioni personalizzate
- accesso all'interfaccia Bluetooth mediante le API `javax.bluetooth` o OSGi I/O connection
- un sistema di posizionamento per informazioni GPS ottenute da stream NMEA
- un Clock Service per la sincronizzazione del clock di sistema
- Kura API per accesso a GPIO/PWM/I2C/SPI

Servizi di Comunicazione: Kura mette a disposizione degli sviluppatori tre differenti soluzioni di comunicazione, tutte incentrate su MQTT:

`DataTransportService`: fornisce un accesso semplificato alle API offerte da Paho per l'utilizzo di MQTT, esentando lo sviluppatore dal dover considerare la complessità del layer di rete e del protocollo di pubblicazione. Il servizio consente ai clienti di connettersi al broker, inviare messaggi e ricevere quelli che saranno pubblicati nei topic a cui si sottoscriveranno in modo trasparente.

`DataService`: utilizza il `DataTransportReceiver`, a cui aggiunge funzionalità per la gestione della connessione con il broker, l'invio di messaggi con priorità ed un meccanismo di buffering per messaggi.

`CloudService`: estende le funzionalità del `DataService` permettendo di ottenere comunicazioni più complesse della comune publish-subscribe, come ad esempio request-response. Gestisce in multiplexing un singolo canale di comunicazione verso il broker, partizionando in modo opportuno i topic utilizzati da dispositivi ed applicazioni differenti.

`CloudService` fornisce, inoltre, un modello per la rappresentazione del payload che prende il nome di `KuraPayload`. Questo modello ha la stessa struttura dati, ma con nomi differenti, di `EdcPayload` – creata da Eurotech per comunicare con la piattaforma Everywhere Cloud – ed è composta da informazioni quali:

`setOn`: contiene l'istante in cui il messaggio è stato inviato a Kura.

`metrics`: è una struttura dati composta da tre campi: nome, valore e tipo di dato. Queste informazioni possono essere utili per effettuare una ricerca all'interno dei messaggi. Ogni payload può avere zero o più metrics.

`position`: è un campo opzionale che può contenere le informazioni geografiche associate al payload

`body`: può contenere informazioni aggiuntive di qualsiasi tipo. Il campo verrà memorizzato dalla piattaforma, ma non verrà utilizzato per analisi statistiche

Il servizio provvede anche alla serializzazione del payload con Google Proto Buf[7] in modo da limitare ulteriormente la banda necessaria all'invio.

[7] Protocol Buffer è un meccanismo estendibile ed indipendente dal linguaggio e dalla piattaforma che consente la serializzazione di una struttura dati in modo più veloce, rapido e semplice di XML.

I servizi di comunicazione illustrati sono senza dubbio utili, ma presentano alcuni difetti. Su tutti il fatto che ci si possa connettere ad un solo broker o che, a partire dall'ultima versione 1.3, le librerie Paho MQTT siano state inserite nel core bundle di Kura e non siano rese disponibili all'esterno causando, di fatto, l'impossibilità di utilizzare le librerie se non importandole nuovamente.

Remote Management: consente la gestione remota delle applicazioni IoT installate in Kura, compreso il loro deploy, aggiornamento e configurazione. Si basa sul `ConfigurationService` e sul `CloudService`

`ConfigurationService`: sfrutta `ConfigurationAdmin` e `MetaTypeService` definiti nella specifica OSGi per la gestione delle configurazioni dei bundle. Il servizio offre, in aggiunta a quanto già presente in OSGi, una semplificazione nell'accesso alle configurazioni correnti dei bundle che implementano l'interfaccia `ConfigurableComponent`, un servizio di snapshot con cui è possibile memorizzare le configurazioni di tutti i servizi in esecuzione nel container, la possibilità di eseguire un rollback ad uno snapshot precedente e la possibilità di accedere alle configurazioni da remoto utilizzando il `CloudService`.

Networking: fornisce API che consentono l'analisi e la configurazione delle interfacce di rete presenti nel gateway come Ethernet, WiFi e reti cellulari

Watchdog Service: registrando i componenti critici nel servizio watchdog consentirà il reset del sistema attraverso il watchdog hardware quando verranno rilevati dei problemi

Web administration interface: offre, all'interno del container, una console di gestione web-based per la configurazione del gateway

Con Kura, dunque, gli sviluppatori avranno a che fare con una infrastruttura standard per la gestione di eventi, che comprende il framework ed una serie di servizi di default. Un'applicazione viene distribuita come bundle OSGi e viene eseguita nel container unitamente ad altri componenti Kura.

Utilizzando la libreria Eclipse Paho, il framework fornisce un servizio di store and forward per tutte le applicazioni che, dopo aver acquisito localmente informazioni, desiderano inoltrare i dati ad un broker, o ad altri servizi cloud, sfruttando la leggerezza del protocollo MQTT.

Le applicazioni possono essere implementate da remoto come pacchetti OSGi e la loro configurazione può essere importata/esportata tramite un servizio di snapshot- Questo servizio può essere utilizzabile inoltre per configurare altri servizi Kura, compatibili con OSGi, che consentono di gestire le impostazioni di rete del gateway –DHCP, DNS, firewall, WiFi, routing–. Altri servizi offerti da Kura che possono essere utilizzati dai bundle sono:

position servizio GPS per geolocalizzare i gateway

click servizio che garantisce una buona sincronizzazione temporale

db utile per la memorizzazione di informazioni in un database SQL embedded

servizi di processo e watchdog garantiscono che tutto funzioni correttamente

Per comunicare con i dispositivi, le applicazioni Kura possono collegarsi all'infrastruttura dei device sfruttando le funzionalità di networking standard di Java o di protocolli opzionali come Modbus o CAN Bus.

Con i servizi OSGi per le comunicazioni seriali (USB e Bluetooth) che consentono l'astrazione dell'hardware, Kura consente l'accesso mobile ad un'ampia gamma di dispositivi. È garantita anche la possibilità di incorporare nel gateway dell'hardware addizionale, grazie alle API che consentono l'accesso ai dispositivi collegati via GPIO, I2C, PWM o SPI.

Il framework Kura, inoltre, offre un front-end web che consente allo sviluppatore/amministratore di attivare/disattivare da remoto tutti i bundle, o di mettere a disposizione una pagina di configurazione bundle-specific.

Unendo tutte queste caratteristiche, insieme al supporto degli strumenti di Eclipse, il gruppo di lavoro IoT della Eclipse Foundation punta a proporre Kura come scelta di riferimento per gli sviluppatori Java di applicazioni Enterprise e IoT/M2M.

Capitolo 3

Progettazione del servizio

Per fare sì che un gateway IoT che esegue un'istanza di Kura possa essere individuabile dovrà avere in esecuzione un bundle che metta a disposizione un servizio di discovery.

In questo progetto si è pensato di realizzarne uno che, grazie all'utilizzo di un protocollo leggero come MQTT, possa eseguire il lookup da qualsiasi dispositivo smart dotato di connessione ad Internet. Nello specifico si è scelto realizzare oltre al servizio per il gateway Kura, in esecuzione su Raspberry Pi B+, anche un client per smartphone Android che consente la ricerca del gateway più vicino, la connessione ad esso ed infine la trasmissione di dati relativi ai sensori di interesse delle applicazioni nel framework Kura.

3.1 Requisiti

Il servizio dovrà offrire le funzionalità di supporto chiave dei sistemi di discovery tradizionali:

- autoconfigurazione
- scoperta del gateway
- accesso al gateway

L'interazione tra client e server per il processo di discovery utilizzerà un broker MQTT in esecuzione su cloud; il sistema che si andrà a realizzare dovrà garantire:

- la maggiore granularità possibile, assicurando che i client –sfruttando la geolocazione– trovino solo i gateway realmente vicini
- la trasmissione di messaggi in un formato leggero, universale, e con il minor overhead possibile, in modo tale da non vanificare la leggerezza di MQTT con messaggi verbosi e che richiedano un parsing eccessivamente pesante.
- una buona scalabilità

La distanza dal client non sarà l'unico fattore discriminante nella scoperta dei gateway: gli smartphone riceveranno le informazioni di accesso al network solo dai gateway interessati dai sensori che mettono a disposizione.

Dopo che il client avrà trovato il gateway ed avrà ricevuto le informazioni necessarie per accedere al suo network dovrà stabilire la connessione, quindi risulterà conveniente che i due componenti comunichino in modo diretto.

3.2 Analisi dei requisiti

Per poter realizzare un servizio di discovery in grado di soddisfare i requisiti appena analizzati è stato utile suddividere il sistema, seguendo un approccio top-down, in sotto-problemi correlati alle entità in gioco: il gateway ed il cliente o, nello specifico, Kura ed Android.

Prima di scendere nel dettaglio delle problematiche delle singole entità, tuttavia, è bene analizzare l'intera struttura nel complesso e definire come i soggetti in gioco interagiscano tra loro.

3.2.1 Scenario

Volendo fare in modo che cliente e gateway comunichino attraverso canali differenti (uno per il discovery, in cui dovrà per forza di cose esserci un intermediario esterno, ed uno per la trasmissione diretta dei dati application-specific) sarà necessario prevedere un'architettura composta da tre componenti:

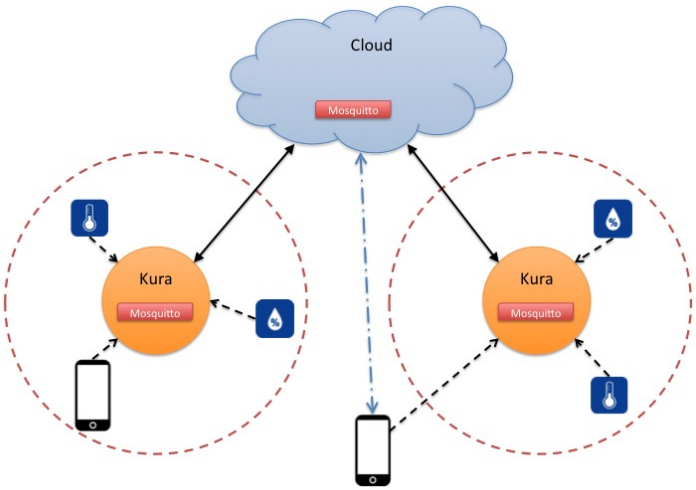
il cloud rappresenta l'entità remota, nota a client e gateway che farà da mediatore attraverso un'istanza di broker MQTT in esecuzione al suo interno.

i gateway possono essere più di uno, sono dispositivi che hanno in esecuzione il framework Kura, e mettono a disposizione dei client una serie di servizi. Fungono da nodo centrale di una sua sottorete. Poiché continuare a comunicare attraverso il cloud, dopo che il dispositivo ha effettuato l'accesso al network del gateway, può risultare poco conveniente rispetto ad una comunicazione diretta, i gateway metteranno a disposizione dei clienti un secondo broker MQTT all'interno del suo network. I gateway dovranno sempre essere connessi sia con il cloud che con il broker locale.

i device rappresentano i client del servizio di discovery; sono interessati a conoscere i gateway nelle loro vicinanze e possono essere più di uno. L'utilizzo di MQTT come protocollo di comunicazione fa si che qualsiasi dispositivo dotato di connessione ad Internet possa essere considerato un client. Saranno connessi con il cloud solo per il tempo necessario alla ricerca dei gateway nelle vicinanze.

In Figura 3.1 è mostrata l'architettura che avrà il sistema di discovery che si intende realizzare: alcuni gateway sono connessi al cloud e dei dispositivi (sensori, attuatori, smartphone, etc.) nei loro paraggi vogliono poter comunicare. Affinché i dispositivi possano interagire con i gateway dovranno in prima istanza stabilire una connessione con essi. Per fare ciò i client interrogheranno il cloud per ottenere le informazioni necessarie all'ingresso nel network del gateway.

Figura 3.1: Architettura del servizio di discovery



Per ridurre il numero di risultati ottenuti dal cloud, ad esempio scartando gateway troppo distanti, si è pensato di utilizzare dei topic geografici che riescano ad identificare un certo luogo in modo accurato e non ambiguo.

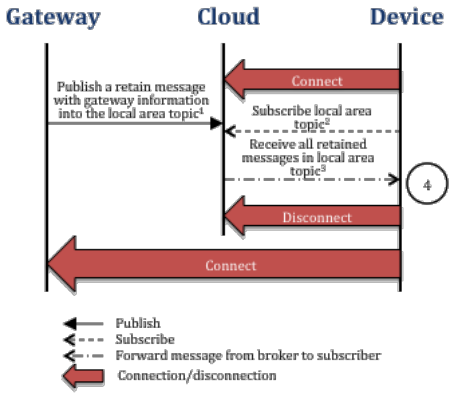
3.2.2 Possibili schemi

Da una prima analisi risultano possibili diverse soluzioni che consentono la realizzazione di un protocollo di discovery MQTT-based. Una prima soluzione può prevedere che siano i gateway ad annunciare la loro presenza in una determinata area geografica, in alternativa si potrebbe pensare che i dispositivi, all'ingresso nella località, inviino un messaggio di annuncio, mentre un'idea più articolata può prevedere che sia dispositivi che gateway annuncino la loro presenza. Di seguito verranno analizzati nel dettaglio questi schemi.

3.2.2.1 Gateway announce

La prima soluzione, illustrata in Figura 3.2, ha come idea generale la pubblicazione di un messaggio retained da parte del gateway. Tale messaggio conterrà le informazioni necessarie alla connessione alla rete locale e sarà pubblicato all'interno del topic geografico del gateway (1). I client, quando vorranno cercare un gateway stabiliranno la connessione e sottoscriveranno il topic geografico relativo alla loro posizione (2). In seguito alla sottoscrizione tutti i messaggi retained che il broker ha salvato all'interno di quel topic verranno consegnati al cliente (3) che dovrà provvedere ad analizzare le risposte e selezionare quello per cui potrebbe risultare più utile (4). Terminata questa fase il cliente si disconnetterà dal cloud e utilizzerà le informazioni contenute nei messaggi ricevuti per collegarsi alla rete del gateway.

Figura 3.2: Gateway announce



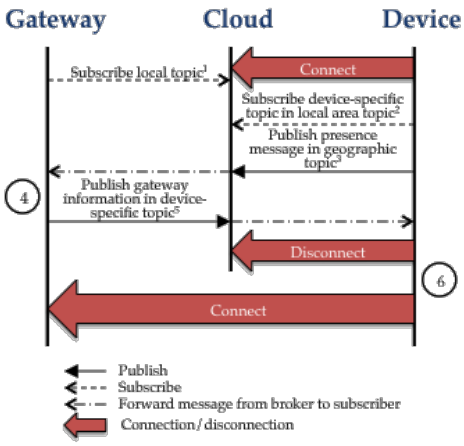
Un vantaggio introdotto da questa soluzione è sicuramente il throughput limitato poiché i gateway comunicheranno il retained message una volta soltanto, ma si corre il rischio di avere un disallineamento: nel caso in cui un gateway si dovesse disconnettere bisognerebbe fare in modo che il messaggio retained venga cancellato. Un'ulteriore punto a sfavore di questa soluzione risiede nel fatto che il client dovrà incaricarsi di scegliere il gateway. Questa operazione può comportare l'analisi di tutti i messaggi ricevuti che potrebbe risultare un lavoro eccessivamente gravoso per sistemi con risorse limitate e, in ogni caso, si traduce con un rallentamento nel processo di connessione al gateway.

3.2.2.2 Device announce

In questa soluzione si prevede che il gateway invii le proprie informazioni in modo diretto ai soli dispositivi considerati di interesse.

In fase di avvio ogni gateway si connetterà al cloud ed effettuerà la sottoscrizione al topic geografico relativo alla sua posizione (1). Allo stesso modo i dispositivi, dopo aver stabilito la connessione con il cloud, si sottoscriveranno ad un topic device-specific (2) –interno a quello geografico di loro interesse– quindi invieranno il loro messaggio di presenza sul topic geografico (3) che conterrà informazioni utili ai gateway per rispondere alla richiesta e valutare se sono interessati o meno a quanto il device ha da offrire. Il broker provvederà a propagare i messaggi ai gateway sottoscritti al topic, ed ognuno di questi, dopo aver valutato se il device può essere di loro interesse (4), può rispondere inviando sul topic device-specific con le informazioni necessarie per stabilire la connessione (5). Il client attenderà per un certo periodo la ricezione dei messaggi dai gateway nelle vicinanze interessati ad esso, quindi deciderà a quale connettersi (6).

Figura 3.3: Device announce

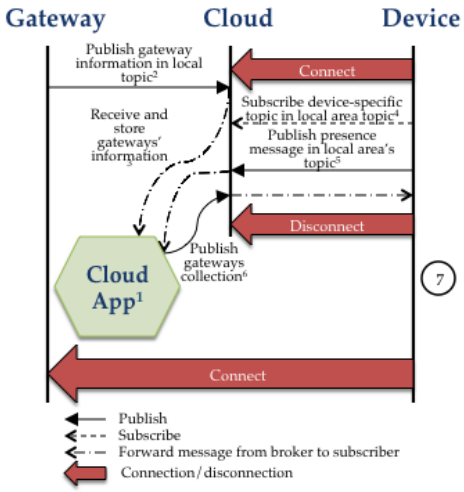


Rispetto all'approccio precedente il throughput risulterà più elevato: ogni gateway in un area geografica dovrà rispondere ad ogni cliente di suo interesse; tuttavia l'elaborazione più pesante verrà eseguita lato gateway, lasciando al dispositivo la sola scelta del gateway tra quelli vicini e realmente interessati ad esso.

3.2.2.3 Cloud application

La terza possibile soluzione risulta più semplice per quanto riguarda la realizzazione delle componenti per gateway e client, ma richiede la realizzazione di un terzo componente che si assume il compito di memorizzare le informazioni dei gateway in modo da inoltrarle ai clienti che le chiedono. Le informazioni inoltrate saranno in numero limitato e filtrate in modo che i client ricevano solo quelle relative ai gateway che sono interessati ai loro dati.

Figura 3.4: Cloud application



L'applicazione in esecuzione sul cloud (1), allo start-up, sottoscriverà tutta la gerarchia di topic. I gateway invieranno nel topic geografico le loro informazioni (2) che verranno acquisite e memorizzate dalla cloud app (3), organizzandole geograficamente. I client, dopo essersi connessi, sottoscriveranno un loro topic specifico (4), quindi pubblicheranno la loro presenza all'interno di quello geografico (5). Anche in questo caso sarà la cloud application a ricevere il messaggio e, sulla base della posizione del dispositivo e su ciò che mette a disposizione, selezionerà un certo numero di messaggi inviati precedentemente dai gateway da inoltrare al client (6) e quest'ultimo ne sceglierà uno e si connetterà ad esso.

Come già detto in questo modo si semplifica la realizzazione dei componenti che realizzano i clienti e i gateway, ma la complessità dell'applicazione in esecuzione su cloud risulta elevata ed introduce diverse problematiche finora assenti, come tutto ciò che comporta il tenere in memoria le informazioni dei gateway o il fatto che un nodo centralizzato incaricato di elaborare tutte le informazioni per un numero elevato di clienti può rappresentare un collo di bottiglia e quindi ridurre la scalabilità.

Stando a quanto detto, risulta che la soluzione più adatta sia la seconda proposta poiché non appesantisce il behaviour dei client con operazioni di valutazione troppo esose e, allo stesso tempo, non dovrebbe limitare la scalabilità. È vero che ci potrebbe essere un elevato flusso di messaggi tra dispositivi e gateway per il discovery, ma le comunicazioni interesseranno soltanto i dispositivi in una determinata area geografica: i gateway riceveranno richieste solo dai client nella loro area, mentre ai client verranno consegnate solo le risposte dei gateway in zona interessati ad essi.

3.3 Analisi del problema

L'analisi dei requisiti ha consentito di individuare tre diversi sotto-sitemi che entreranno in gioco all'interno del servizio di discovery: il cloud, il gateway e i client.

Per quanto riguarda il cloud l'unico requisito che dovrà soddisfare sarà quello di avere in esecuzione un'istanza di broker MQTT.

Poiché il gruppo di ricerca IoT della Eclipse Foundation mette a disposizione degli sviluppatori un server per facilitare lo sviluppo di applicazioni IoT con al suo interno, tra le altre cose, un broker MQTT che esegue la versione 1.3.1 di Mosquitto, si è pensato di appoggiarsi, almeno per la fase di realizzazione del servizio, a quest'ultimo in modo da spostare l'attenzione sull'effettiva implementazione del servizio. Il broker è raggiungibile all'indirizzo `tcp://iot.eclipse.org:1883`. Utilizza un bridge HTTP per mostrare all'indirizzo `http://eclipse.mqttbridge.com` la propria lista di topic e la sua configurazione non prevede l'autenticazione con username e password.

Per quanto riguarda gateway e client invece, risulterà necessario analizzare le specifiche problematiche di entrambi i componenti in modo separato; prima di questo è bene definire in modo non ambiguo quale sarà il formato dei messaggi che i componenti si scambieranno e quali saranno le informazioni che questi conterranno.

3.3.1 Il payload

La definizione di una rappresentazione dei dati non ambigua risulta necessaria affinché i messaggi scambiati possano essere interpretati in modo corretto sia dai clienti che dai gateway.

Onde evitare di incorrere nel problema della verbosità e della pesantezza nel parsing tipico di UPnP si è scelto di rappresentare i dati in formato JSON e, per generalizzare il più possibile, si è pensato di realizzare una struttura dati univoca che possa rappresentare allo stesso tempo annuncio di presenza da parte del client, risposta con le informazioni da parte del gateway, ed un ulteriore tipologia utilizzabile per la trasmissione dei dati application-specific. Tale struttura sarà composta da un campo che specificherà la tipologia di messaggio inviato, e da una stringa che conterrà l'effettivo contenuto del messaggio, anch'esso in formato JSON.

Questo secondo campo potrà essere di tre diverse tipologie:

`AnnounceContent` rappresenta il messaggio di annuncio inviato dal client. Esso conterrà

- l'ID del dispositivo
- le sue coordinate geografiche (latitudine e longitudine)
- la lista dei sensori che mette a disposizione dei gateway

`InfoGatewayContent` rappresenta le informazioni che il gateway invierà ai clienti che ritiene utili. Conterrà al suo interno:

- l'ID del gateway
- la distanza dal client

- l'URI del broker locale
- la lista dei sensori di cui vuole ricevere aggiornamenti
- tutti i parametri necessari alla connessione al suo network:
  - SSID
  - chiave d'accesso
  - tipo di sicurezza della rete WiFi

`SensorData` rappresenta i dati relativi ad una misurazione di un sensore. Questo oggetto conterrà tre campi:

- il timestamp dell'ultima misurazione
- la tipologia del sensore (rappresentata come stringa)
- i dati relativi alla misurazione

Un messaggio di questo tipo conterrà, a prescindere dal numero di sensori abilitati, una lista di uno o più `SensorData` in modo da trasmettere con un unico messaggio tutti i dati misurati, riducendo overhead e throughput.

Per consentire la corretta codifica/decodifica dei messaggi sarà utile fornire dei meccanismi che si occupino della serializzazione e della deserializzazione.

3.3.2 I topic

Un'altra possibile problematica riguarda i topic che verranno utilizzati per la comunicazione. Si è già detto che la comunicazione tramite il cloud utilizzerà dei canali location-specific, risulta quindi necessario definire come i topic verranno selezionati.

Una prima idea potrebbe essere quella di ottenere informazioni geopolitiche relative alla posizione. In questo modo si potrebbe ottenere una gerarchia di topic composta ad esempio da nazione, regione, provincia, città, etc. Per garantire una più elevata granularità in modo da contattare solo i gateway realmente vicini e non essere legati troppo a fattori geopolitici[1], si è pensato di aggiungere a queste informazioni un identificativo del luogo specifico in cui il dispositivo si trova. Un topic organizzato in questo modo è sicuramente human-readable, ma questo comporta dover ottenere le informazioni e codificarle opportunamente in modo che non ci sia ambiguità.

Google, attraverso i servizi Maps e Place, aiuta nella risoluzione del problema: è infatti possibile ottenere sia le informazioni geopolitiche che gli ID che il provider assegna ai Google Place. Analizzando le risposte ottenute per differenti richieste si è potuto notare, però, che in alcuni casi esse non coincidono: per determinate posizioni si riescono ad ottenere tutte le informazioni richieste, per altre solo alcune. Considerato che gateway e client saranno in località prossime, e quindi i loro dati non dovrebbero essere differenti, questo

[1] Se ad esempio il gateway è installato ad un incrocio ed è in ascolto soltanto su una via il device potrebbe non trovarlo poiché le sue coordinate lo posizionano nell'altra.

non è il problema maggiore. La difficoltà principale è dovuta alla differente codifica linguistica che il server applica: se si specifica il parametro `Locale` alla richiesta il server dovrebbe tradurre le informazioni, ma non sempre questo avviene. Dai test effettuati si è notato che, a seconda del dispositivo, del sistema operativo e della lingua impostata in quest'ultimo, anche specificando uno stesso valore di `Locale`, i risultati ottenuti sono differenti.

Questi problemi hanno portato ad applicare una soluzione differente che favorisce la non ambiguità a discapito della leggibilità: i Google Place racchiudono tutte le informazioni geopolitiche che si volevano utilizzare in precedenza, ed il loro ID è univoco e indipendente dalla lingua. Utilizzando questo attributo come topic si riesce a garantire al contempo univocità e granularità a livello di punto di interesse.

Oltre alla definizione dei topic per la comunicazione su cloud, risulta necessario stabilire su quali topic comunicheranno gateway e dispositivo quando saranno connessi al broker locale. Si è deciso di far sottoscrivere il gateway a due topic: uno per la ricezione dei dati dei sensori, ed uno su cui verranno pubblicati messaggi di servizio.

Per quanto riguarda il topic di ricezione dei dati il gateway sottoscriverà tutta la gerarchia `/sensor/#`, mentre i client pubblicheranno messaggi in `/sensor/ID/`. In questo modo si terranno separati i dati provenienti da più client ed il gateway potrà sempre sapere a quale dispositivo corrispondono quelle informazioni. Il topic dei messaggi di servizio, invece, verrà utilizzato dai client per pubblicare il loro Will Message in caso di disconnessione improvvisa. In questo caso, a meno che non si prevedano ulteriori messaggi di servizio, potrebbe non essere necessaria una gerarchia di topic: è sufficiente che il dispositivo memorizzi come Will Message il proprio ID, e il gateway saprebbe che gli ID ricevuti nei messaggi pubblicati nel topic `/notification/` sono quelli dei dispositivi che si sono disconnessi in modo anomalo.

3.3.3 Il gateway Kura

Il gateway IoT, come anticipato, sarà un Raspberry Pi B+ che eseguirà il framework Kura e realizzerà una rete WiFi a cui si collegheranno tutti i dispositivi che utilizzeranno il servizio di discovery. Il dispositivo accederà ad Internet utilizzando la connessione ethernet e la condividerà attraverso il WiFi a tutti i client. Eseguirà inoltre un'istanza di Mosquitto per soddisfare il requisito del broker locale.

Il bundle per il framework Kura che si andrà a realizzare dovrà essere visto come un bundle OSGi configurabile da parte dell'amministratore del gateway, che dovrà definire:

- il nome del gateway
- le sue coordinate
- il topic geografico da sottoscrivere
- le informazioni sui broker –sia locale che remoto–
- i sensori di suo interesse

- le informazioni sulla rete realizzata dal gateway
  - SSID
  - chiave d'accesso
  - tipo di sicurezza

I parametri relativi alle coordinate potrebbero essere ottenuti in modo automatico dal servizio, magari sfruttando altri servizi Kura, ma poiché non tutti i gateway potrebbero essere dotati di GPS e le coordinate ottenute mediante web-services (Google, Yahoo, Bing, Apple, etc.) non hanno un'elevata precisione si è preferito lasciare il compito all'amministratore. Lo stesso discorso può essere fatto per l'ID del Google Place da utilizzare come topic: si potrebbe ottenere una lista di possibili Places interrogando il web-service di Google con le coordinate, ma onde evitare che il sistema selezioni un Place errato si è preferito percorrere la strada della configurazione manuale.

Il componente dovrà essere dotato di due client MQTT: uno per le comunicazioni con il broker su cloud, e l'altro che verrà utilizzato per le comunicazioni attraverso il broker locale.

Tenendo in considerazione che Kura mette a disposizione tra i suoi servizi di default alcuni che semplificano l'utilizzo delle API della libreria Paho per l'utilizzo di MQTT, potrebbe sembrare ragionevole l'utilizzo di uno di questi service al fine di semplificare le operazioni. Sia il CloudService che il DataTransportService, tuttavia, presentano una grave limitazione –almeno per la realizzazione di questo servizio di discovery–: consentono l'utilizzo di un solo broker e poiché il servizio che si andrà a realizzare avrà bisogno di connettersi a due broker differenti allo stesso tempo, sarà necessario realizzare tutte le operazioni utilizzando in modo diretto le API offerte da Paho.

Il comportamento del bundle sarà relativamente semplice: in avvio caricherà l'ultima configurazione salvata utilizzando il servizio di snapshot offerto da Kura, quindi si connetterà sia al broker locale che a quello remoto –dove sottoscriverà il topic geografico. Alla ricezione di messaggi da parte dei clienti analizzerà il contenuto. Se il dispositivo sarà ritenuto utile gli verrà inviato un messaggio di tipo `InfoGateway` contenente le informazioni per accedere alla rete WiFi messa a disposizione. Quando il client sarà connesso il gateway riceverà i messaggi dal broker locale.

3.3.4 Il client Android

Il dispositivo Android, per poter essere abilitato alla ricerca di gateway Kura nelle vicinanze, dovrà avere installata una specifica applicazione che dovrà fornire determinate funzionalità:

Comunicazione utilizzando il protocollo MQTT è la parte centrale del servizio di discovery che si intende realizzare, quindi sarà necessario che l'applicazione Android riesca a comunicare attraverso MQTT per trovare i gateway più prossimi. Questa funzionalità è offerta dalle librerie Paho, che offrono delle implementazioni open-source di client MQTT in diversi linguaggi di programmazione, tra cui una specifica per Android che, essendo realizzata mediante Android Service, eseguirà in background e consentirà di mantenere attiva la connessione al broker, inviare e ricevere messaggi anche quando l'applicazione che la utilizza cede il controllo ad un'altra Activity.

La geolocazione risulta di fondamentale importanza poiché è attraverso questa che il dispositivo saprà dove cercare i gateway. Lo smartphone dovrà ottenere le sue coordinate GPS che saranno utilizzate per ottenere una lista di possibili località in cui il dispositivo potrebbe trovarsi.
Android e Google sono a disposizione degli sviluppatori tutti gli strumenti necessari, sia per ottenere le coordinate del dispositivo, sia per interagire con dei web-service che consentono di ottenere una lista di possibili località in cui lo smartphone potrebbe trovarsi.

Gestione del WiFi questa funzionalità risulta necessaria poiché è attraverso di essa che si andrà a realizzare il meccanismo di auto-configurazione: il client riceverà dal gateway una serie di informazioni che saranno utilizzate per configurare il servizio di gestione della connessione WiFi del dispositivo per accedere alla rete realizzata dal gateway.

Oltre alle tre feature appena elencate risulterà utile l'accesso ai sensori del dispositivi. Questo consentirà all'utente di selezionare quali sensori mettere a disposizione dei gateway, in modo tale che questi possano scartare gli smartphone che non offrono nulla di utile alle loro applicazioni e, una volta stabilita la connessione, accedere ai sensori stessi per comunicare i dati a Kura.

In questo caso non è necessario che l'applicazione disponga di due differenti client. Ne sarà sufficiente uno che alternerà periodi di connessione al broker su cloud, durante il processo di discovery, ed a quello locale quando invece avrà stabilita la connessione al gateway.

L'applicazione presenterà un'interfaccia grafica in cui si consentirà all'utente di selezionare i sensori da mettere a disposizione e di impostare il tempo di comunicazione dei dati dei sensori al gateway. Questo secondo parametro è stato introdotto al fine di ottimizzare il consumo energetico del dispositivo ed il traffico dati: utilizzando le classi offerte da Android per l'interazione con i sensori questi invieranno notifiche ad un listener con un nuova misurazione a frequenze elevate[2], ed inviare un messaggio ad ogni aggiornamento dei dati risulterebbe eccessivamente dispendioso. Inoltre, se il dispositivo dovesse mettere a disposizione più di un sensore, inviare un messaggio per ognuno di essi aumenterebbe il throughput. Si è quindi pensato di raccogliere i dati forniti dai sensori e di inviarli ad intervalli regolari in un unico messaggio.

In fase di connessione al broker il dispositivo dovrà fornire un proprio identificativo che dovrà essere univoco. Questo può risultare un problema, in particolare quando si stabilisce la connessione con il broker su cloud a cui connetteranno un maggior numero di client rispetto all'istanza locale al gateway: se si lascia la scelta dell'ID all'utente si va incontro al rischio che qualche dispositivo si veda negata la connessione a causa di identificativi duplicati. Per questo motivo si è deciso di acquisire in modo automatico l'IMEI dello smartphone e di utilizzarlo come ID univoco del dispositivo. L'IMEI verrà inoltre utilizzato in fase di sottoscrizione al topic: affinché le risposte dei gateway vengano ricevute solo dal dispositivo interessato i messaggi di configurazione verranno inviati sul topic "`PlaceID/IMEI`".

Il comportamento dell'applicazione risulta più articolato rispetto al bundle, e lavorando con servizi asincroni necessita di un meccanismo che sincronizzi l'esecuzione delle operazioni. Quando l'utente avvierà il processo di discovery l'applicazione dovrà ottenere le coordinate del dispositivo, con cui interrogherà il web-service Place di Google per ottenere la lista dei luoghi in cui potrebbe trovarsi lo smartphone. Dopo aver

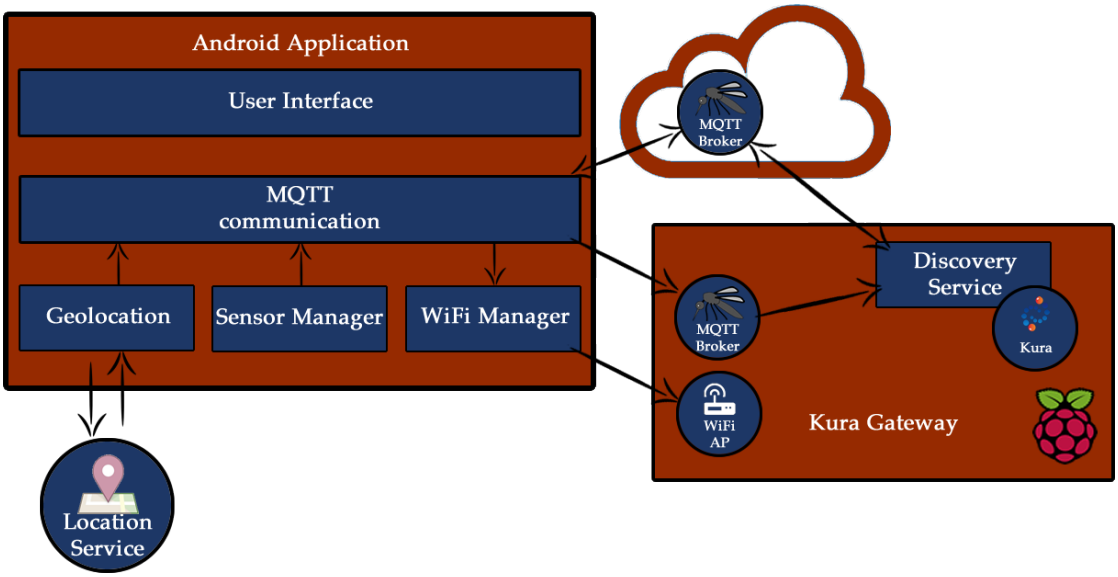
[2] Stando a quanto definito nel codice sorgente di Android Lollipop il delay minimo di aggiornamento è quello del sensore mentre quello massimo è di 200000µs (5Hz, quattro aggiornamenti al secondo)[34]

stabilito la connessione al broker su cloud sottoscriverà il suo topic device-specific nel Place di interesse ed attenderà per un intervallo di tempo che i gateway rispondano. Nel caso in cui nessun gateway sia presente o abbia risposto ripeterà le operazioni nel topic successivo, altrimenti selezionerà il gateway più vicino e cercherà di accedere al suo network per poi connettersi al broker locale ed iniziare la trasmissione dei dati.

3.4 Architettura logica

In Figura 3.5 è illustrata l'architettura dell'applicazione Android che si dovrà realizzare:

Figura 3.5: Architettura logica del servizio di discovery



Quando l'utente avrà avviato l'applicazione, selezionerà i sensori da mettere a disposizione dei gateway e darà inizio al processo di discovery. Il meccanismo di geolocazione recupererà le coordinate GPS del dispositivo e le invierà ad un web-service di localizzazione che restituirà una lista di possibili luoghi in cui potrebbe trovarsi lo smartphone.

Il servizio in esecuzione su Android pubblicherà sul cloud broker un messaggio contenente le sue coordinate ed i sensori che mette a disposizione nel topic relativo alla località con probabilità maggiore. Successivamente si metterà in attesa di ricevere risposte dai gateway.

Il broker recapiterà i messaggi a tutti i gateway sottoscritti a quel topic che analizzeranno il contenuto del messaggio. In questo modo potranno definire se il client risulterà loro utile, e in caso affermativo calcoleranno la distanza dallo smartphone ed invieranno sul topic device-specific un messaggio con le informazioni necessarie per connettersi al suo network e collegarsi con il broker locale.

Ricevuti i messaggi da tutti i gateway che lo ritengono utile, il dispositivo si disconnetterà dal cloud e creerà una nuova configurazione per accedere alla rete wireless realizzata dal Raspberry attraverso il servizio di gestione della connessione WiFi.

Terminato l'accesso al network l'applicazione si connetterà al broker locale in esecuzione sul gateway ed inizierà ad inviare i dati dei sensori richiesti in un topic device-specific all'interno del topic "`/sensor/`", che rappresenta la radice della gerarchia sottoscritta dal bundle Kura per la ricezione dei dati.

Capitolo 4

Implementazione del servizio

In questo capitolo verrà descritta l'implementazione del servizio. Sarà seguita la stessa procedura utilizzata in fase realizzativa, cercando di risolvere le varie problematiche relative ad ogni componente, ed in fine accorpare il tutto nel progetto finale.

4.1 Il bundle del servizio di Discovery

L'implementazione del servizio Kura prevede la realizzazione di un bundle configurabile che dovrà essere dotato di due client MQTT realizzati tramite le librerie Paho. Di seguito verrà mostrata nel dettaglio l'implementazione del modulo OSGi-Kura.

Il bundle che andrà a realizzare il servizio di discovery sarà configurabile, e consentirà all'amministratore del gateway di impostare le coordinate del dispositivo, il suo ID, il topic da sottoscrivere, i sensori a cui è interessato, e le informazioni sui broker (protocolli utilizzati, indirizzi e porte sulle quali sono in ascolto), oltre che le informazioni relative alla rete WiFi che il Raspberry mette a disposizione.

L'applicazione dovrà utilizzare due differenti client in modo simultaneo: uno sarà in ascolto sul cloud e l'altro sul broker. Come già anticipato in fase di analisi, i servizi messi a disposizione da Kura non consentono l'utilizzo di broker differenti in contemporanea, e per questo è stato necessario lavorare con i client messi realizzati dalle librerie Paho. Questo ha introdotto un ulteriore problema: dalla versione 1.3 di Kura le librerie sono contenute all'interno del core bundle del progetto e non sono rese accessibili dalle applicazioni, in questo modo la community cerca di spingere gli sviluppatori ad utilizzare i servizi offerti dal framework.

Un ulteriore aspetto del bundle che è stato realizzato riguarda l'utilizzo del servizio di configurazione e snapshot che Kura mette a disposizione. Si è fatto in modo che nel caso in cui fosse inserita una configurazione errata (URI dei broker non validi, campi vuoti, sicurezza del WiFi impostata in modo non corretto, etc.) il servizio facesse rollback all'ultima configurazione funzionante.

Di seguito verranno analizzate nel dettaglio queste caratteristiche e verranno mostrati i codici sorgenti delle classi realizzate.

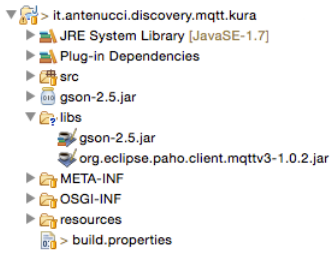
Per maggiori dettagli riguardo lo sviluppo di applicazioni Kura-based o l'utilizzo delle librerie Paho si può fare riferimento all'Appendice.

4.1.1 Creazione del progetto e configurazione del manifest

Per realizzare il bundle Kura per il servizio di discovery bisognerà innanzitutto creare un nuovo Plug-in Project e configurare il manifest in modo opportuno.

Dopo aver generato il progetto la prima cosa da fare sarà importare al suo interno le librerie Paho che saranno utilizzate per la comunicazione attraverso MQTT, e Gson per la serializzazione/deserializzazione dei payload. Queste verranno posizionate nella directory `libs` all'interno della root directory del progetto come mostrato in Figura 4.1:

Figura 4.1: Directory libs all'interno del progetto

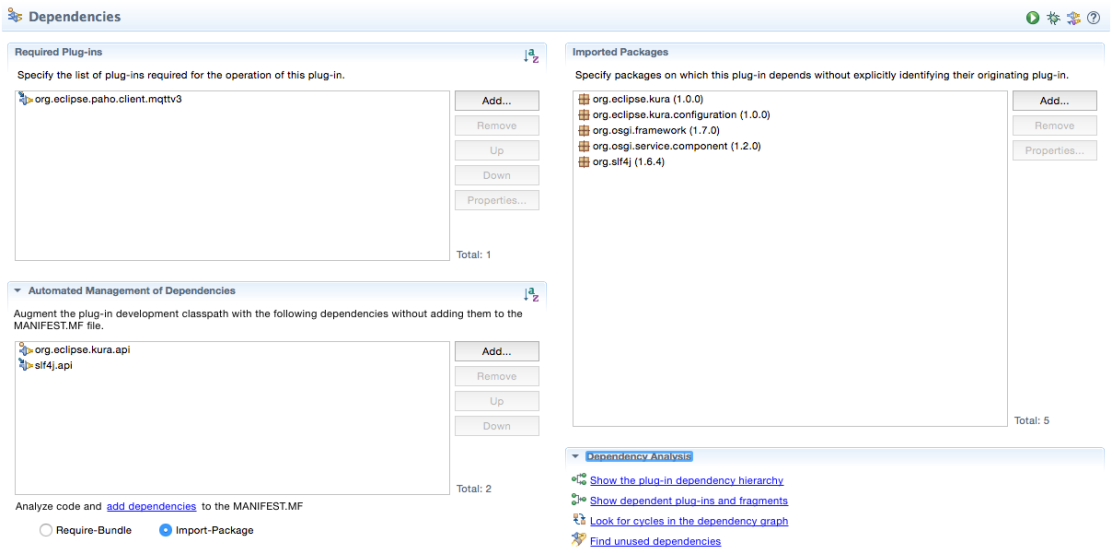


Gson sarà aggiunto al build path, come avviene solitamente per le applicazioni Java Standard, mentre le librerie Paho verranno importate come bundle all'interno del framework Kura.

Oltre alle librerie bisognerà importare anche le classi che realizzano i messaggi, i loro contenuti e le utility necessarie per la serializzazione/deserializzazione degli oggetti (descritte in Appendice). Queste sono state inserite in un apposito package all'interno del bundle.

Affinché i bundle OSGi riescano a risolvere le librerie sarà necessario specificare all'interno del Manifest dove possono essere trovate. Per questo motivo oltre che aggiungere le API di Kura ed il logger al meccanismo di risoluzione automatica sarà anche necessario inserire all'interno del manifest un riferimento alla libreria Gson aggiunta al build path e specificare che le librerie Paho saranno importate come bundle esterno.

Figura 4.2: Definizione delle dipendenze all'interno del manifest



In Figura 4.2 è mostrata la schermata di configurazione automatica delle dipendenze del manifest, mentre il Listato 4.1 mostra la struttura del manifest:

Listato 4.1: MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MqttDiscoveryService
Bundle-SymbolicName: it.antenucci.discovery.mqtt.kura
Bundle-Version: 1.0.0.qualifier
Bundle-Vendor: Carlo Antenucci
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Bundle-ClassPath: .,
 libs/gson-2.5.jar
Service-Component: OSGI-INF/component.xml
Import-Package: org.eclipse.kura;version="1.0.0",
 org.eclipse.kura.configuration;version="1.0.0",
 org.osgi.framework;version="1.7.0",
 org.osgi.service.component;version="1.2.0",
 org.slf4j;version="1.6.4"
Require-Bundle: org.eclipse.paho.client.mqttv3
```

4.1.2 Il ComponentDescriptor ed il MetaType file del bundle

Poiché si andrà a realizzare un servizio configurabile, il bundle avrà bisogno di un component descriptor in cui andranno specificati:

- quali sono i metodi di attivazione, disattivazione e aggiornamento
- la proprietà di tipo stringa `service.pid`
- la configuration policy (dovrà essere impostata a require)
- l'abilitazione e l'attivazione automatica del bundle
- la classe che implementa il bundle come provider

Il component descriptor è mostrato nel listato 4.2:

Listato 4.2: MqttDiscoveryService.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" activate="activate" configuration-
     policy="require" deactivate="deactivate" immediate="true" modified="updated" name="it.antenucci.
     discovery.mqtt.kura.MqttDiscoveryService">
   <implementation class="it.antenucci.discovery.mqtt.kura.MqttDiscoveryService"/>
   <property name="service.pid" type="String" value="it.antenucci.discovery.mqtt.kura.
        MqttDiscoveryService"/>
   <service>
      <provide interface="it.antenucci.discovery.mqtt.kura.MqttDiscoveryService"/>
   </service>
</scr:component>
```

Oltre al component descriptor nel bundle dovrà essere presente un file `MetaType`, che indicherà al `ConfigurationService` di Kura quali sono le proprietà configurabili del bundle e quali sono i suoi valori di default.

Le proprietà configurabili per il servizio di discovery sono quelle che andranno a comporre i messaggi di tipo InfoGateway ed alcuni dettagli aggiuntivi come indirizzo del broker su cloud e coordinate, ma saranno strutturate in modo leggermente più verboso in modo tale da facilitarne la validazione.

Di seguito sono elencate le proprietà configurabili del bundle:

- l'ID del gateway
- le coordinate:
  - latitudine
  - longitudine
- il topic location-specific
- I sensori richiesti

- l'URI di ogni broker suddiviso in
  - un campo per il protocollo
  - un campo per l'indirizzo
  - un campo per la porta
- i parametri della connessione WiFi
  - SSID
  - Chiave d'accesso
  - Tipo di sicurezza

4.1.3 Il package `kura.utils`

Durante la realizzazione del bundle ci si è resi conto che sarebbe stato utile realizzare delle classi che fornissero dei meccanismi di supporto alle operazioni, una classe che rappresentasse le proprietà del bundle, ed una che contenesse tutte le costanti necessarie all'esecuzione del servizio. L'implementazione di queste classi è mostrata in Appendice.

La classe `Constants` contiene tutte le costanti necessarie al funzionamento del servizio. È illustrata nel listato A.11 ed al suo interno sono definiti in modo statico:

- i nomi a cui saranno associate le proprietà all'interno dell'oggetto del parametro `Map<String name,Object object> properties` dei metodi `activate` e `upadated`
- i topic che saranno sottoscritti dal client connesso al broker locale
- due metodi consentiranno di ottenere:
  - l'ID di un sensore passando come parametro il nome
  - il nome del sensore passando come parametro l'ID

La classe `MqttDiscoveryProperties` racchiude al suo interno tutte le proprietà del bundle, un campo aggiuntivo nel quale verrà inserita una stringa di errore nel caso in cui la validazione delle proprietà non andasse a buon fine, ed un oggetto `InfoGatewayContent` che conterrà la parte statica del messaggio che i gateway invieranno ai clienti. La classe definirà inoltre tutti i metodi getter e setter per accedere ai campi al suo interno. È definita dal listato A.12 .

La classe `Utils` definisce al suo interno dei metodi statici di supporto al servizio di discovery come:

`calculateDistance()` utilizzata in fase di ricezione dei messaggi dal cloud. Calcola la distanza euclidea (in metri) tra client e gateway sulla base delle coordinate passate come parametri. Il risultato verrà inserito nel messaggio di risposta al cliente.

`validateURI()` richiamata in fase di validazione per verificare che l'URI sia stato inserito correttamente. La validazione avviene cercando di risolvere l'indirizzo fornito.

`validateProperties()` richiamata quando il framework aggiornerà la configurazione del bundle. Il metodo provvede a controllare se le proprietà sono state configurate adeguatamente. Verrà verificato che:

- non ci siano campi non settati (a parte la chiave di rete nel caso in cui la sicurezza non sia presente)
- che il campo relativo alla tipologia di sicurezza della rete WiFi abbia come valore una delle tre stringhe ammesse ("`WPA`", "`WEP`" e "`NONE`")
- che ci sia almeno un sensore selezionato
- che gli URI dei broker siano validi

Un ultimo controllo riguarda la presenza tra i problemi di configurazione del campo Device ID non impostato. Poiché questo campo potrà essere nullo solo al primo avvio (i valori di default saranno tutti nulli) o in caso di rollback alla configurazione iniziale, la sua presenza tra gli errori significherà che il servizio non è ancora stato configurato, quindi il campo `error` dell'oggetto `MqttDiscoveryProperties` –che verrà restituito dal metodo– conterrà una stringa che inviterà a configurare il servizio. In assenza di questo problema, se le proprietà supereranno tutti i controlli, il campo `error` rimarrà vuoto, mentre in presenza di errori verrà inserita al suo interno una stringa contenente tutte le problematiche riscontrate durante la validazione.

Durante il processo di validazione verrà anche riempito l'oggetto `InfoGatewayContent` contenuto in `MqttDiscoveryProperties`.

La classe `Utils` è definita nel listato A.13.

4.1.4 Il package `kura.listeners`

All'interno del package listeners sono contenute le due classi che implementano le interfacce `MqttCallback` per i due client dell'applicazione.

La classe `MqttCloudCallback` definisce due dei tre metodi esposti dall'interfaccia. Il metodo `connectionLost`, dopo aver notificato la disconnessione improvvisa attraverso il logger, provvederà a tentare una nuova connessione, mentre il metodo `messageArrived` realizzerà in un nuovo thread la deserializzazione del payload. Nel caso in cui il messaggio ricevuto sia di tipo `Announce` invocherà il metodo `sendInfoGateway` sul servizio, passando come parametro l'oggetto ottenuto deserializzando il payload del messaggio.

Listato 4.3: MqttCloudCallback

```
public class MqttCloudCallback implements MqttCallback {
   private final Logger LOG = LoggerFactory.getLogger(MqttDiscoveryService.class);
   private MqttDiscoveryService service;

   public MqttCloudCallback(MqttDiscoveryService service) {
      this.service = service;
   }

   @Override
   public void connectionLost(Throwable arg0) {
      LOG.warn("Cloud broker disconnected");
      service.connectBroker(BrokerType.CLOUD);
   }

   @Override
   public void deliveryComplete(IMqttDeliveryToken arg0) {}

   @Override
   public void messageArrived(String topic, final MqttMessage message) throws Exception {
      new Runnable(){
         @Override
         public void run(){
            Object o = MessageUtils.fromJson(new String(message.getPayload()));
            if (o instanceof AnnounceContent){
               service.sendInfoGateway((AnnounceContent)o);
            }
         }
      }.run();
   }
}
```

La classe `MqttLocalCallback` opererà in modo molto simile. La sola differenza tra le due classi risiede nel metodo `messageArrived`: in questo caso, dopo aver ricostruito il contenuto del messaggio, si provvederà a stampare le informazioni ricevute.

Se l'oggetto deserializzato sarà una lista di oggetti `SensorData` verrà mostrato a video il device che ha inviato il messaggio, e per ogni sensore il suo nome e le misurazioni comunicate. Se invece l'oggetto non è di tipo `List<SensorData>`, il messaggio è stato ricevuto nel topic "`/notification/`" e la classe comunicherà che il client con ID uguale al payload ricevuto si è disconnesso.

Nel listato 4.18 è mostrato il metodo `messageArrived` della classe `MqttLocalCallback`:

Listato 4.4: MqttLocalCallback.messageArrived()

```
   @Override
   public void messageArrived(final String topic, final MqttMessage message) throws Exception {
      new Runnable(){
         @Override
         public void run(){
            String msg = new String(message.getPayload());
            Object o = MessageUtils.fromJson(msg);
            if (o instanceof List<?> && ((List<?>) o).get(0) instanceof SensorData){
               List<SensorData> sensorData = (List<SensorData>)o;
               String[] split = topic.split("/");
               String devID = split[split.length-1];
```

```
                System.out.println("Sensor Data Received from " + devID + ":");
                for(SensorData data : sensorData){
                    System.out.println("Value of " + data.getType() + " at " + data.getTimestamp() + ":"
                        );
                    System.out.println("  X = " + data.getValue()[0]);
                    System.out.println("  Y = " + data.getValue()[1]);
                    System.out.println("  Z = " + data.getValue()[2]);
                }
            }
            else if(topic.equals(Constants.WILL_TOPIC)){
                System.out.println("Device " + msg + " disconnected.");
            }
        }
    }.run();
}
```

4.1.5 La classe `MqttDiscoveryService`

`MqttDiscoveryService` è la classe che implementa il vero e proprio servizio di discovery. Dovrà implementare l'interfaccia `ConfigurableComponent` per permettere al framework di capire che si tratta di un bundle configurabile. La classe definirà inoltre i metodi di attivazione, aggiornamento e disattivazione del servizio.

Il metodo di attivazione invocherà la validazione delle proprietà del bundle. Se l'oggetto restituito dal metodo della classe `Utils` conterrà una stringa di errore il servizio non verrà avviato, altrimenti verrà invocato `startService`.

Listato 4.5: MqttDiscoveryService.activate()

```
protected void activate(ComponentContext componentContext, Map<String, Object> properties){
    LOG.info("Bundle " + APP_ID + " has started with configuration!");
    this.properties = Utils.validateProperties(properties);
    if (this.properties.getErr() == "")
        startService();
    else{
        LOG.error("Please, configure the service properly.");
    }
}
```

Il metodo `startService` dapprima controllerà se il servizio è già in esecuzione, e in tal caso provvederà a fermarlo, quindi preparerà l'oggetto `MqttConnectionOption` che servirà a definire i parametri di connessione, e cercherà di stabilire la connessione ai due broker. I tentativi si ripeteranno finché entrambi i broker non saranno connessi, attendendo trenta secondi tra una prova non riuscita e l'altra.

Listato 4.6: MqttDiscoveryService.startService()

```
public void startService() {
    if(running){
        stopService();
    }
    connOpt = new MqttConnectOptions();
    connOpt.setCleanSession(true);
```

```
    connOpt.setConnectionTimeout(15); //15 seconds
    connOpt.setKeepAliveInterval(30); //30 seconds
    while(!running){
       try {
          cloud = new MqttAsyncClient(properties.getCloudBroker(), properties.getDevId(), null);
          cloud.setCallback(new MqttCloudCallback(this));
          connectBroker(BrokerType.CLOUD);
          local = new MqttAsyncClient(properties.getLocalBroker(), properties.getDevId(), null);
          local.setCallback(new MqttLocalCallback(this));
          connectBroker(BrokerType.LOCAL);
          if(cloud != null && local != null && cloud.isConnected() && local.isConnected())
             running = true;
       } catch (MqttException e) {
          LOG.warn("Something goes wrong connecting cloud broker. Check if the URI and retry.");
       }
       if(!running){
          LOG.warn("Something goes wrong. If the problem persist retry your configuration and your
               connection.\nRetry in 30 seconds...");
          try {
             Thread.sleep(30000);
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
       }
    }
}
```

I tentativi di connessione ai broker sono effettuati attraverso il metodo `connectBroker` specificando con quale dei due ci si intende collegare. Il metodo cercherà di connettersi al broker facendo tre tentativi, se riesce a stabilire la connessione sottoscriverà i topic di suo interesse ("`/sensor/#`" e "`/notification/`" per il broker locale, o il topic geografico per quello su cloud), altrimenti se il servizio è in funzione e l'altro client risulta connesso si attenderanno 30 secondi prima di effettuare altri tre tentativi. Se invece il servizio non è attivo e/o l'altro client non è connesso verrà invocato il metodo `startService` che provvederà a far ripartire il processo di connessione.

Listato 4.7: MqttDiscoveryService.connectBroker()

```
    IMqttAsyncClient cli;
    String msg = "Connecting to ";
    if(type==BrokerType.LOCAL){
       cli = local;
       msg += "local broker";
    }
    else{
       cli = cloud;
       msg += "cloud broker";
    }
    for(int i=0;i<3;i++){
       try{
          LOG.info(msg +", attempt " + (i+1));
          t = cli.connect(connOpt);
          t.waitForCompletion();
          break;
       }
       catch(MqttException e){
```

```
            if(i<3){
                LOG.error("Attempt " + (i+1) + " failed. Retry in 10 seconds");
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e1) {
                    e1.printStackTrace();
                }
            }
        }
    }
    if(cli == local){
        if(local.isConnected()){
            try{
                LOG.info("Local broker connected, subscribing to " + Constants.SENSOR_DATA_TOPIC + "
                     and " + Constants.WILL_TOPIC + "...");
                t = local.subscribe(Constants.SENSOR_DATA_TOPIC, 0);
                t.waitForCompletion();
                t = local.subscribe(Constants.WILL_TOPIC, 0);
                t.waitForCompletion();
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
        else{
            LOG.error("Unable to connect local broker.");
            if(running && cloud.isConnected()){
                LOG.error("Retry in 30 seconds.");
                try {
                    Thread.sleep(30000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                connectBroker(BrokerType.LOCAL);
            }
            else
                startService();
        }
    }
    else{
        if(cloud.isConnected()){
            try{
                cloudTopicSubscribed = properties.getCloudTopic();
                LOG.info("Cloud broker connected, subscribing to " + cloudTopicSubscribed + "...");
                t = cloud.subscribe(cloudTopicSubscribed, 0);
                t.waitForCompletion();
                if (cloud.isConnected())
                    return;
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
        else{
            LOG.error("Unable to connect cloud broker.");
            if(running && local.isConnected()){
                LOG.error("Retry in 30 seconds.");
                try {
```

```
                Thread.sleep(30000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            connectBroker(BrokerType.CLOUD);
        }
        else
            startService();
    }
  }
}
```

Il metodo `updated` è quello che si occuperà di aggiornare le proprietà del bundle. Ogni volta che l'amministratore modificherà i parametri questo metodo verrà invocato dal framework Kura. Al suo interno si effettuerà la validazione delle nuove proprietà. Se non dovessero esserci problemi verrà invocato il metodo `startService` per riavviare il servizio con la nuova configurazione, altrimenti si effettuerà il rollback alla configurazione precedente utilizzando il servizio `ConfigurationService`. In caso di rollback il metodo provvederà ad aggiungere al log una riga in cui comunicherà il fallimento dell'update specificando, che il servizio, nel caso in cui fosse attivo, continuerà a lavorare con la precedente configurazione.

Listato 4.8: MqttDiscoveryService.updated()

```
protected void updated(Map<String, Object> prop){
    LOG.info("Configuration of " + APP_ID + " updated!");
    properties = Utils.validateProperties(prop);
    if(properties.getErr() == ""){
        startService();
    }
    else{
        if(!properties.getErr().equals("Configure the service")){
            try {
                configurationService.rollback();
            } catch (KuraException e) {
                e.printStackTrace();
            }
            LOG.error(properties.getErr());
            if(running)
                LOG.warn("Service still working with previous configuration.");
            else
                LOG.error("Configure the service properly.");
        }
        LOG.error("Configure the service properly.");
    }
}
```

Quando il framework interromperà l'esecuzione del servizio lo farà invocando il metodo `deactivate`, che invocherà il metodo `stopService`, e scriverà nel log che il servizio è stato fermato.

Il metodo `stopService`, provvederà a eliminare le sottoscrizioni ai topic, disconnettere i client MQTT e liberare le risorse che stavano utilizzando. Questo metodo viene invocato anche da `startService` a seguito di un update: se il servizio è in esecuzione ed i client MQTT sono connessi, quando il me-

todo `updated` invocherà `startService` quest'ultimo prima di avviare la nuova connessione dovrà disconnettere i client.

Listato 4.9: MqttDiscoveryService.stopService()

```
private void stopService() {
   running = false;
   try {
      if (cloud != null){
         if ( cloud.isConnected()){
            cloud.unsubscribe(cloudTopicSubscribed);
            cloud.disconnect();
            cloud.close();
         }
         cloud = null;
      }
      if (local != null){
         if (local != null && local.isConnected()){
            local.unsubscribe(Constants.SENSOR_DATA_TOPIC);
            local.unsubscribe(Constants.WILL_TOPIC);
            local.disconnect();
            local.close();
         }
         local = null;
      }
   } catch (MqttException e) { e.printStackTrace(); }
}
```

L'ultimo metodo definito nella classe `MqttDiscoveryService` è quello che realizza la funzionalità di invio del messaggio di tipo `InfoGateway`. Questo metodo verrà invocato dalla classe `MqttCloudCallback`, e riceverà come parametro l'oggetto di tipo `AnnounceContent` ottenuto deserializzando il contenuto del messaggio. Il metodo creerà una lista di stringhe contenente tutti i sensori che appartengono sia alla lista contenuta nell'`AnnounceContent`, sia alla lista dei sensori che il gateway richiede. Se l'intersezione delle due liste è vuota il metodo termina, altrimenti completerà il riempimento del payload di tipo `InfoGatewayContent` –presente nell'oggetto `MqttDiscoveryProperties`– inserendo, oltre alla lista dei sensori appena riempita, la distanza euclidea tra i dispositivi calcolata con il metodo della classe `Utils`, quindi invierà il messaggio sul topic device-specific.

Listato 4.10: MqttDiscoveryService.sendInfoGateway()

```
public void sendInfoGateway(AnnounceContent announce) {
   List<String> requestedSensors = new ArrayList<String>();
   for (String s: announce.getAvailableSensors())
      if (properties.getSensors().contains(s))
         requestedSensors.add(s);
   if(requestedSensors.isEmpty()){
      LOG.info("Device " + announce.getDevId() + " not offers useful sensors.");
      return;
   }
   LOG.info("Sending InfoGW message to " + announce.getDevId());
   properties.getContent().setSensorsRequested(requestedSensors);
   double latD = announce.getLatitude();
   double lonD = announce.getLongitude();
```

```
        properties.getContent().setDistance(Utils.calculateDistance(properties.getLatGW(), properties.
            getLonGW(), latD, lonD));
        Message payload = new Message(MessageType.INFO_GW, MessageUtils.getJsonString(properties.
            getContent()));
        try {
            MqttMessage msg = new MqttMessage();
            msg.setPayload(MessageUtils.getJsonString(payload).getBytes());
            msg.setQos(0);
            msg.setRetained(false);
            cloud.publish(cloudTopicSubscribed+"/"+announce.getDevId(), msg);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
```

L'implementazione completa della classe `MqttDiscoveryService` è mostrata dal listato A.14.

4.1.6 Esportazione del bundle

Per esportare il bundle si possono seguire due differenti modalità: esportare il progetto come file `.jar` o creare un deployment package.

Vista la necessità di caricare all'interno del framework anche un bundle per le librerie Paho, trasferite dagli sviluppatori di Kura all'interno del core bundle e non accessibili in modo diretto, si è preferito utilizzare la seconda modalità.

L'utilizzo dei deployment package consente inoltre di poter rendere avviabile automaticamente il servizio al riavvio del framework Kura.

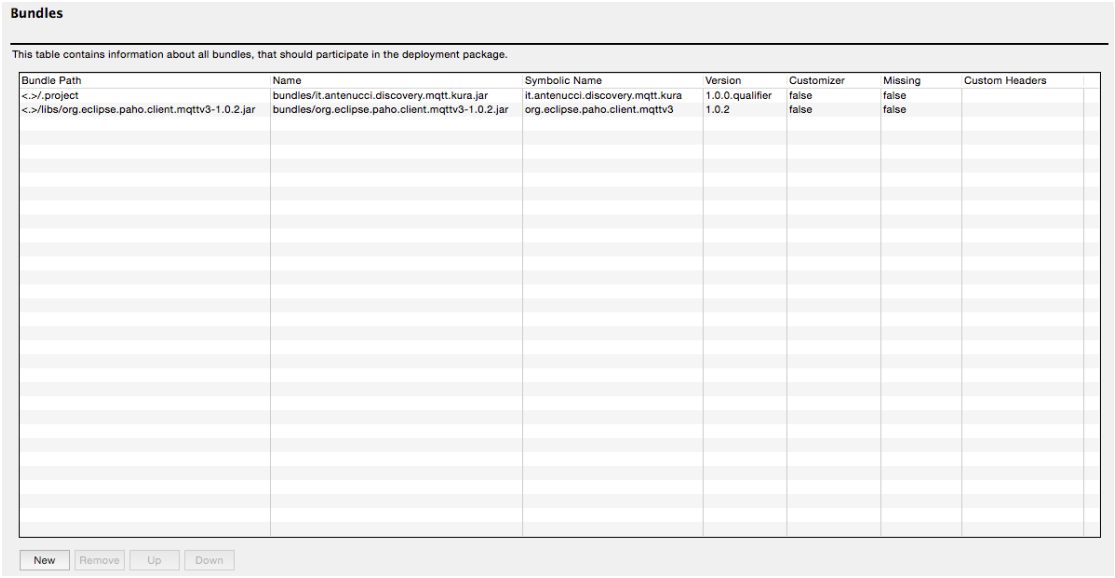
Per ottenere il deployment package bisognerà creare una nuova directory all'interno del progetto, in cui verrà inserito il Deployment Package Project. In questo file con estensione `.dpp`, dovranno essere specificati i percorsi dei bundle che si vogliono installare.

In questo caso specifico il Deployment Package Project `mqtt_discovery.dpp` conterrà due bundle:

- quello relativo al servizio di discovery, che punterà al file `.project` nella root directory del progetto del bundle
- le librerie Paho, che faranno riferimento alla file `.jar` copiato precedentemente nella directory `libs/`

La Figura 4.3 mostra la sezione Bundles all'interno dell'editor per i Deployment Package Project di Eclipse:

Figura 4.3: Bundle definiti nel file mqtt_discovery.dpp



Dopo aver salvato il file, facendo click con il tasto destro su di esso e selezionando Quick Build verrà generato all'interno della directory il deployment package `mqtt_discovery.dp`. Questo sarà il file che verrà utilizzato per l'installazione del bundle sul gateway.

4.2 L'applicazione Android

4.2.1 Realizzazione dei meccanismi necessari

Per poter risolvere al meglio il comportamento che ogni meccanismo dovrà avere si è pensato di iniziare la realizzazione dell'applicazione Android scomponendola in quattro applicazioni differenti, una per ogni funzionalità che andrà a comporre l'app finale.

Di seguito verrà analizzato lo sviluppo dei vari componenti, ognuno in esecuzione in modo indipendente dall'altro. Le classi che sono state sviluppate per la realizzazione dei seguenti meccanismi ed i relativi file XML sono presenti in Appendice.

4.2.1.1 Comunicazione MQTT

La comunicazione attraverso MQTT è la parte centrale del sistema di discovery, e verrà realizzata attraverso le librerie messe a disposizione da Eclipse Paho.

Queste ultime offrono l'implementazione di client MQTT open-source in svariati linguaggi di programmazione, tra cui anche uno per Android realizzato attraverso un `Service`. In questo modo il client fornisce una connessione al broker MQTT senza la necessità che l'applicazione sia attiva, rendendo possibile l'invio e la ricezione dei messaggi anche quando sarà in background.

Di seguito verrà descritto come utilizzare il client Android offerto da Eclipse Paho, iniziando dalla configurazione del progetto ed analizzando come il servizio lavora.

Poiché la comunicazione attraverso MQTT potrebbe essere uno dei colli di bottiglia del servizio di discovery, si è pensato di dotare questa applicazione di un meccanismo che consenta l'analisi dei tempi necessari per l'esecuzione delle varie operazioni.

Configurazione del progetto Android

Il primo passo per utilizzare il client MQTT fornito da Paho è quello di copiare le librerie `org.ecplise.paho.client.mqttv3` e `org.ecplise.paho.client.android.service` nella directory `libs` del progetto android; bisognerà quindi configurare il manifest con i giusti permessi dichiarando, oltre all'accesso ad Internet, anche che l'applicazione avrà un componente `Service` realizzato dalla classe `MqttService` definita nel package `org.eclipse.paho.android.service`.

Un ulteriore permesso dichiarato nel manifest consente all'applicazione di accedere alle informazioni relative allo smartphone in modo da ottenere il codice IMEI per poterlo utilizzare come ID univoco del client.

Il file manifest è mostrato nel listato A.15.

Application layout

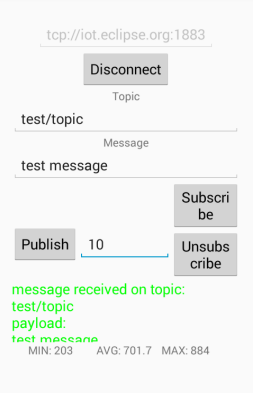
Nell'interfaccia dell'applicazione è stata utilizzata una `TextView` per mostrare i risultati delle operazioni richieste dall'utente (e i messaggi ricevuti), e quattro `EditText` per inserire l'URI del broker, il topic da sottoscrivere o in cui pubblicare messaggi, il payload degli stessi, ed il numero di messaggi da pubblicare. Sono stati inoltre inseriti quattro pulsanti con i quali si potrà:

- stabilire o chiudere una connessione con il broker
- pubblicare un messaggio in un determinato topic
- sottoscrivere un topic
- annullare la sottoscrizione al topic

Ai pulsanti e alle `EditText` sono stati aggiunti delle `TextView` in cui tenere traccia del tempo necessario per eseguire le operazioni richieste dall'utente.

In Figura 4.4 è mostrato il layout che avrà l'applicazione di esempio che può essere ottenuto con il codice XML mostrato nel listato A.16:

Figura 4.4: Android Mqtt Client layout



Utilizzo di MqttAndroidService

Come detto il servizio `MqttAndroidService` offre un'interazione in background con il broker, ma ogni applicazione che vorrà utilizzare MQTT per comunicare avrà bisogno di dichiarare un oggetto `MqttAndroidClient`. Questo oggetto è un'estensione di `BroadcastReceiver` che implementa l'interfaccia `MqttAsyncClient`. In questo modo sarà possibile per il servizio invocare operazioni sul client utilizzando gli `Intent` anche quando l'applicazione è in background.

Un altro oggetto è richiesto per definire i parametri di connessione. Con `MqttConnectOptions` è possibile specificare le modalità che il client utilizzerà per connettersi al broker (flag di clean session, il messaggio will, il timeout di connessione o l'intervallo di keep alive).

Altri due oggetti, invece, dovranno essere realizzati implementando le interfacce `MqttCallback` ed `IMqttActionListener`. La prima interfaccia definisce le callback invocate dal servizio MQTT quando il client perde la connessione, riceve l'`ack` dopo la consegna di un messaggio, o il client riceve un messaggio dal broker. L'interfaccia `IMqttActionListener` invece definisce due metodi che verranno invocati quando l'operazione che si è richiesta verrà completata con successo (`onSuccess`) o fallirà sollevando un'eccezione (`onFailure`).

MQTTActionListener

L'oggetto `MQTTActionListener` implementa l'interfaccia `IMqttActionListener` e sarà associato ad ogni operazione eseguita da `MqttClient`. `MQTTActionListener` verrà invocata dal `Service` quando un'operazione termina con successo o fallisce. L'esempio utilizza una semplice implementazione di questa classe che non farà altro che aggiornare la `TextView` dei risultati scrivendo quale operazione è stata eseguita, il suo esito, e aggiornando il tempo necessario per completare l'operazione.

Quest'ultima funzionalità consente di analizzare quanto tempo impiega il client a connettersi/disconnettersi o sottoscrivere un topic; nel caso di invio di messaggi questo metodo verrebbe invocato quando il client termina l'invio, ma ai fini di un'analisi accurata dell'interazione è risultato più interessante valutare il tempo necessario affinché il messaggio venisse pubblicato nel topic[1].

Ogni volta che verrà eseguita un'operazione con `MqttClient` sarà creata una nuova istanza di `MQTTActionListener` che all'atto della creazione memorizzerà l'istante in cui l'oggetto è pronto per essere utilizzato. Quando una delle callback verrà invocata, e quindi l'operazione (diversa dall'invio di messaggi) sarà terminata, si calcolerà il tempo che è trascorso dall'invio della richiesta alla fine della sua esecuzione sottraendo all'istante in cui il costruttore ha memorizzato la richiesta quello in cui è stata invocata la callback. L'implementazione della classe `ActionListener` è mostrata dal codice nel listato A.17 .

MQTTCallBack

Oltre che un `MQTTActionListener`, ogni `MqttClient` avrà bisogno di una classe che implementi l'interfaccia `MQTTCallback`, definita in `org.eclipse.paho.client.mqttv3`, che fornisce tre metodi invocati dal `Service` Android:

connectionLost invocato quando il client perde la connessione con il broker a seguito di un'eccezione (passata come parametro al metodo). In questo esempio il metodo scriverà nella `TextView` dei risultati, che il client non è più connesso con il broker riportando il problema che si è verificato. Inoltre provvederà a modificare il pulsante di connessione invocando il metodo messo a disposizione dall'`Activity`.
Questo metodo può essere utile nella realizzazione del servizio di discovery per riavviare la ricerca di un gateway quando si perde la connessione con quello a cui si era connessi.

deliveryComplete invocato quando il broker riceve il messaggio e lo pubblica nel topic. Nel caso in cui si stesse usando QoS = 1 il metodo viene invocato quando il client riceve il messaggio `PUBACK`, mentre se il QoS utilizzato è 2 l'invocazione avviene alla ricezione del messaggio `PUBCOMP`. Nell'esempio realizzato questo metodo viene utilizzato per notificare alla classe `Measurements` che il messaggio, il cui identificativo verrà recuperato dal parametro con interfaccia `IMqttToken`, è stato pubblicato dal broker.

messageArrived è forse il metodo più importante tra quelli definiti dall'interfaccia. È invocato quando un nuovo messaggio, pubblicato su un topic sottoscritto, viene consegnato al client. In questo esempio,

[1] Quando il messaggio è ricevuto dal broker e viene pubblicato, quest'ultimo invia al client un messaggio di tipo `PUBACK`, che verrà intercettato dal metodo `deliveryComplete` di `IMqttCallback`.

quando il client riceve un messaggio, la callback scrive nella `TextView` il payload del messaggio e il topic sul quale è stato ricevuto. Sviluppando il servizio di discovery questo metodo sarà utile per ricevere e memorizzare le informazioni dei gateway prima di scegliere a quale connettersi.

Il codice nel Listato A.18 realizza la classe `MQTTCallBack`.

`ButtonListener`

Per consentire l'interazione tra l'interfaccia utente ed i metodi definiti dalla classe `MqttClient` è stata definita una classe `ButtonListener` che implementa l'interfaccia `OnClickListener`. Questa classe, mostrata nel listato A.19 in Appendice, definisce un solo metodo (`onClick`) in cui vengono eseguite differenti operazioni a seconda del pulsante cliccato dall'utente. Indipendentemente dal pulsante, `onClick` invocherà il metodo `setMeasurements` della classe `Measurements`, che definirà il numero di operazioni che verranno effettuate al fine di poter calcolare il tempo medio di esecuzione[2].

In `ButtonListener` risulta necessario disporre dei riferimenti ottenuti nell'`Activity`, come gli elementi `EditText` usati per inserire il payload, il topic, o l'URI del broker. Per questo motivo vengono richiesti come parametri del costruttore il riferimento alla `MQTTActivity` ed il codice IMEI del dispositivo. Il costruttore provvederà a memorizzare i parametri e ad ottenere i riferimenti alle `View`.

Quando l'utente farà click sul pulsante `connect` il listener provvederà alla creazione id un nuovo `MqttAndroidClient` utilizzando il codice IMEI come identificativo. Ottenuto il nuovo client a quest'ultimo verrà associato un nuovo oggetto `MQTTCallBack`, creerà un nuovo oggetto `MqttConnectionOption` in cui sarà possibile definire i parametri di connessione, tra cui:

- intervallo di keep alive
- timeout di connessione
- flag di clean session
- topic e messaggio will

Terminata la configurazione delle opzioni verrà aperta la connessione invocando il metodo `connect` sull'oggetto client, passando come parametri l'oggetto `MqttConnectionOption` appena definito, la `View` che richiede la connessione, ed una nuova istanza di `MQTTActionListener`.

Nel caso in cui il client sia già connesso al broker, cliccando sul pulsante la connessione verrà chiusa.

Se invece il pulsante cliccato dall'utente è quello per l'invio dei messaggi, se il client è connesso, e topic e payload non sono vuoti, si controllerà quante volte l'utente vuole inviare il messaggio, si configurerà opportunamente la classe `Measurements`, e si terrà traccia del tempo di inizio delle operazioni di invio, quindi si creerà un oggetto `MqttMessage` che avrà come payload il testo inserito `nell'EditText`, gli si assegnerà QoS = 1 e si imposterà il flag `retained` a `false`. Una volta creato il messaggio questo verrà inviato al broker attraverso il metodo publish di `MqttClient`, dichiarando anche il topic (definito

[2] Verrà eseguita sempre una sola misurazione. L'unica eccezione riguarda l'invio di messaggi nel caso in cui l'utente specifichi un diverso valore nell'`EditText` del numero di messaggi da inviare.

nel relativo `EditText`), `l'Activity` e `l'ActionListener`. Invocando il metodo di invio si recupererà inoltre l'identificativo del messaggio per comunicare alla classe `Measurements` quando è iniziata l'operazione per quello specifico messaggio.

Per quanto riguarda il click sui pulsanti per sottoscrivere un topic o cancellare la sottoscrizione, la classe non farà altro che invocare il relativo metodo sul client, a condizione che quest'ultimo sia connesso e che `l'EditText` non sia vuoto.

MqttActivity

La classe `MqttActivity`, definita dal listato A.20, interagisce con l'interfaccia grafica e accede al servizio `TelephonyMagager` per ottenere il codice IMEI del dispositivo utilizzato per identificare univocamente il client. Al suo interno è definito un metodo privato per l'inizializzazione della UI che ottiene i riferimenti agli elementi del layout, e in avvio li disabilita tutti eccetto il pulsante per la connessione e il campo per l'inserimento dell'URI del broker.

La classe `MqttActivity` offre inoltre tre metodi pubblici:

updateUI invocato da `MqttActionListener` quando il processo connessione/disconnessione termina con successo, e da `MqttCallBack` nel caso in cui venga persa la connessione al broker. Questo metodo cambia il testo del pulsante di connessione da "Connect" a "Disconnect" nel caso in cui la connessione venga stabilita (o viceversa se il client si disconnette dal broker) ed abilita/disabilita tutti gli elementi del layout a seconda dello stato della connessione.

isConnected restituisce il valore della variabile connected, aggiornata dal metodo updateUI. È utilizzato per controllare lo stato della connessione in `ButtonListener`.

updateTime invocato dalla classe `Measurements` per mostrare all'utente il tempo minimo, massimo e medio necessario al client per eseguire le operazioni richieste.

Measurements

La classe `Measurements` viene utilizzata per tener traccia dei tempi necessari al client per interagire con il broker.

Il primo metodo da invocare per poter utilizzare la classe è `setActivity`, con cui viene memorizzata l'`Activity` che metterà a disposizione il metodo per mostrare i tempi nell'interfaccia.

Quando il client esegue una nuova operazione, qualunque essa sia, invocherà il metodo `setMeasurements`. In questo metodo, il numero di misurazioni passato come parametro viene memorizzato, viene vuotata la lista delle misurazioni, e viene messo a zero il contatore delle misurazioni aggiunte.

Al completamento dell'operazione, se non si tratta della pubblicazione di un messaggio, la classe `MQTTActionListener` invocherà il metodo `addTime` che aggiungerà alla lista la nuova misurazione e incrementerà il contatore. Se quest'ultimo risulterà uguale al numero di misurazioni impostato in precedenza si invocherà il metodo `updateTime` dell'activity, passando come parametri il tempo massimo, minimo e medio per eseguire l'operazione richiesta, ottenibili attraverso `getMaxTime`, `getMinTime`, e `getAvgTime` (che in questo caso coincideranno poiché l'operazione è singola).

Se invece l'operazione eseguita è l'invio del messaggio verranno utilizzati i metodi `startPublish` e `endPublish`. Il primo non farà altro che aggiungere ad uno `SparseArray`[3] di `Long` l'istante in cui è iniziata l'elaborazione di un certo messaggio, identificando il timestamp con l'identificativo del messaggio stesso. Il secondo metodo invocherà il metodo `addTime`, passando come parametro la differenza tra il timestamp ricevuto e quello memorizzato nell'array all'inizio dell'operazione, identificato dall'indice uguale all'ID del messaggio ricevuto. Terminate le operazioni svolte da `addTime` il metodo provvederà a rimuovere l'elemento appena utilizzato dall'array.

L'implementazione della classe è mostrata dal listato A.21 .

4.2.1.2 Geolocation

Al fine di ottenere i gateway Kura più prossimi, il dispositivo dovrà pubblicare un messaggio di presenza in un topic location-specific. Per questo risulta di fondamentale importanza per il sistema di discovery che si andrà a realizzare il processo di geolocazione del device.

Per poter cercare i gateway vicini al dispositivo è necessario ottenere informazioni geografiche. A tal proposito Android mette a disposizione almeno due servizi, da utilizzare insieme al servizio di localizzazione, accessibile mediante la classe `LocationManager`. Una prima soluzione è l'utilizzo del servizio `Geocoder`, messo a disposizione dal sistema operativo, mentre la seconda prevede l'interazione con un web service messo a disposizione da Google, utilizzando le Place Detection API.

Il `LocationManager` controlla le coordinate e periodicamente (o nel caso in cui il movimento del dispositivo abbia superato una certa soglia) invierà una notifica ad un listener. È inoltre possibile richiedere al servizio un singolo aggiornamento. Quest'ultima possibilità risulta particolarmente utile nel sistema di discovery che si andrà a realizzare: il device si geolocalizza e si connette al gateway più vicino, poi, finché sarà connesso a quest'ultimo, non avrà la necessità di aggiornare la posizione, ma potrà richiedere un nuovo aggiornamento singolo quando si disconnetterà per poter cercare un altro gateway Kura.

Ottenute le coordinate, per poter ottenere le informazioni geografiche relative alla posizione dello smartphone si potrà utilizzare `Geocoder` che, utilizzando le coordinate, consente di ottenere informazioni come Paese, regione, codice di avviamento postale, città, indirizzo, etc. Una pecca di questo servizio deriva dal fatto non è in grado di fornire informazioni riguardanti i punti di interesse vicini, come monumenti, scuole, o qualsiasi altra informazione che possa essere utilizzata come punto di riferimento per aumentare la precisione della ricerca.

Attraverso le Place Detection API, invece, è possibile interrogare un web service offerto da Google che, sulla base delle coordinate specificate come parametro, accedendo al database di Google Maps, restituirà una lista di probabili luoghi in cui potrebbe trovarsi il telefono. Il servizio è accessibile da Android utilizzando la libreria `GooglePlayServices` e, al contrario di `Geocoder`, non è in grado di accedere in modo diretto alle informazioni geopolitiche relative alla posizione del dispositivo, ma ai soli punti di interesse.

[3] Questa classe è simile ad un array di `Objects`, ma allo stesso tempo è simile anche ad una `HashMap`: è possibile avere una lista di indici non consecutivi, ad ognuno dei quali è associato un `Object`. Nel caso in esame ad ogni sensor type (un `Integer` definito dalla classe `SensorManager`) verrà associata la rappresentazione `JSON` (una `String`) dell'ultimo `SensorEvent` prodotto da quel sensore. L'utilizzo di questo oggetto è suggerito poiché risulta molto più memory efficient rispetto all'utilizzo di una `HashMap` per mappare `Integer` con oggetti.

Per questo motivo si è pensato di utilizzare entrambi i servizi in modo tale da recuperare le informazioni geopolitiche con `Geocoder`, e quelle relative ai punti di interesse più vicini sfruttando le Place Detection API[4].

Ottenere una key per Google Place Services

Per interagire con Google Places API è necessario identificare l'applicazione nella Google Developer Console per ottenere la chiave univoca che consentirà a Google di tenere traccia del numero di richieste inviate dall'applicazione al server[5].

Per ottenere la key come prima cosa sarà necessario ottenere la chiave privata del certificato digitale dell'applicazione. Questa è ottenibile, per un certificato di debug, semplicemente digitando i seguenti comandi nel terminale del sistema operativo:

```
#Linux or OSx (~/.android/ is the default location of debug keystore)
keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey
                                        -storepass android -keypass android
#Windows
keytool -list -v -keystore "%USERPROFILE%\.android\debug.keystore" -alias androiddebugkey
                                                    -storepass android -keypass android
```

Se invece l'applicazione è già in release e si vogliono aggiungere le Google API, per ottenere la chiave bisognerà utilizzare i seguenti comandi:

```
#Locate your release certificate
#   your_keystore_name is the fully-qualified path and name of keystore, including the extension.
keytool -list -keystore your_keystore_name
#Ask for the certificate informations
#   your_keystore_name is the fully-qualified path and name of keystore, including the extension.
#   your_alias_name is the name assigned to the certificate
keytool -list -v -keystore your_keystore_name -alias your_alias_name
```

La chiave corrisponderà alla linea SHA1:

[4]Se fosse stato possibile recuperare le coordinate utilizzate dalle Google Place APIs (ottenute in modo trasparente) per fornirle alla classe `Geocoder` –la classe restituisce le informazioni geografiche relative alla posizione del dispositivo–, si sarebbe potuto evitare l'utilizzo del servizio di sistema e del listener.

[5]Google impone un limite di 1000 richieste ogni 24 ore, 150000 se l'account sviluppatore ha la fatturazione abilitata.

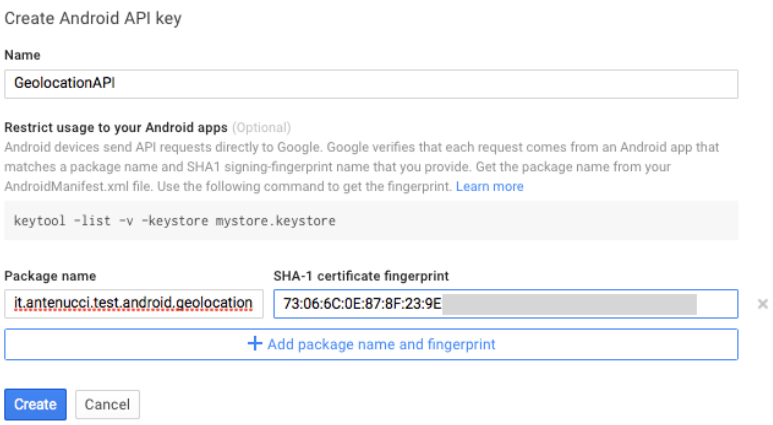
Figura 4.5: Debug key

```
MacBook-Pro-di-Carlo:~ carloantenucci$ keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey
-storepass android -keypass android
Nome alias: androiddebugkey
Data di creazione: 4-set-2015
Tipo di voce: PrivateKeyEntry
Lunghezza catena certificati: 1
Certificato[1]:
Proprietario: CN=Carlo Antenucci, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=IT
Autorità emittente: CN=Carlo Antenucci, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=IT
Numero di serie: 22b2994d
Valido da: Fri Sep 04 10:30:32 CEST 2015 a: Fri Jan 02 09:30:32 CET 2054
Impronte digitali certificato:
         MD5:  ED:B0:9B:C9:BC:EC:48:F7
         SHA1: 73:06:6C:0E:87:8F:23:9E
         SHA256: 4F:99:EB:56:1F:63:C9:
         Nome algoritmo firma: SHA256withRSA
         Versione: 3

Estensioni:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: C1 50 D5 0D 72 5F 99 DB   BA D2 47 87 22 5F 90 63  .P..r_....G."_.c
0010: A5 14 8E 25                                        ...%
]
]
```

Ottenuta la chiave, per registrare l'applicazione ed abilitare l'accesso alle Google API, bisognerà registrarla nella Google Developer Console. Dopo aver eseguito l'accesso in Google Developer Console e selezionato (o creato) il progetto nel quale si vogliono utilizzare le Google API, bisognerà entrare nel menù "APIs & auth"; nella pagina "APIs" si dovrà abilitare "Google Place APIs Android", quindi, nella pagina Credentials selezionare API key dopo aver cliccato sul pulsante "Add credentials" e selezionare "Android key".

Nella nuova pagina si dovrà quindi aggiungere un nome alla chiave e cliccare su "Add package name and fingerprint" , quindi inserire nelle text box apparse il nome del package dell'applicazione e la chiave privata ottenuta in precedenza (Figura 4.6), quindi cliccando su "Create" si otterrà la API key da utilizzare all'interno dell'applicazione.

Figura 4.6: Create API key



Importare le librerie e definire il manifest

Una volta abilitato l'utilizzo delle API, per potervi accedere sarà necessario importare la libreria `GooglePlayService` nel progetto Android. Questo è possibile scaricando le librerie utilizzando Android SDK Manager ed importando il progetto della libreria, posizionato in `$ANDROID_SDK_PATH/extras/google/google_play_services/libproject/`, all'interno del workspace.

Dopo aver aggiunto il progetto libreria ed aver creato il progetto che utilizzerà le API, cliccando su `Project/Properties`, nel box in basso all'interno del tab Android sarà possibile importare il progetto `google_play_service` come libreria. Si procederà, quindi, alla definizione dell'Android Manifest, in cui sarà necessario aggiungere i permessi relativi all'accesso alle informazioni di localizzazione (`android.permission.ACCESS_FINE_LOCATION`), quelle per l'utilizzo delle Google API (`com.google.android.providers.gsf.permission.READ_GSERVICES`) e, per poter interrogare il server, quelle per l'accesso ad Internet.

In aggiunta, attraverso due meta-tag, dovranno essere dichiarate la API key, e la versione dei Google Play Service da utilizzare.

Il codice del listato A.22 mostra l'Android Manifest completo.

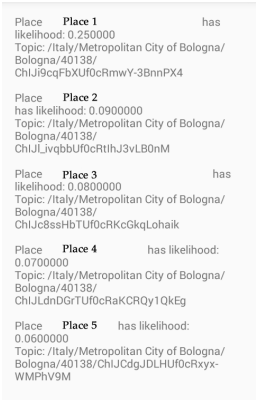
Application layout

Per questo esempio è stato utilizzato un layout con solo una `TextView` in cui mostrare le informazioni dei luoghi più vicini, ordinati per probabilità di verosimiglianza, secondo il formato "`Luogo: /mqtt/topic/according/to/position`".

Il file XML che definisce il layout è mostrato nel listato A.23 , e la Figura 4.7 mostra uno screenshot dell'applicazione:

La Figura 4.7 mostra il layout dell'applicazione:

Figura 4.7: Android Geolocation layout



GeolocationListener

La classe GeolocationListener è la classe che implementa le interfacce di tutti i listener necessari:

LocationListener definisce i metodi di callback che verranno invocati dal LocationService. Tra tali metodi quello di maggiore interesse è senza dubbio onLocationChanged che verrà invocato dal servizio quando avrà ottenuto, in questo esempio su richiesta dell'applicazione, la nuova posizione del dispositivo.
Quando il metodo di callback onLocationChanged viene invocato dal servizio di sistema, il topic viene ottenuto invocando il metodo privato getTopic passando come parametri latitudine e longitudine. Questo metodo utilizza le coordinate del dispositivo per ottenere le relative informazioni geografiche con le quali costruirà la struttura gerarchica del topic definita come /CountryName[/AdminArea][/SubAdminArea]/Locality/PostalCode[6].
Una volta creata la gerarchia viene invocato il metodo connect dell'oggetto GoogleApiClient che si connetterà con il web service. Quando la connessione sarà stabilita, il servizio GooglePlay invocherà il metodo onConnected definito nell'interfaccia ConnectionCallbacks.

ConnectionCallbacks definisce i metodi di callback invocati da GooglePlayService quando il client si sarà connesso/disconnesso al il web service. Quando il client avrà stabilito la connessione il metodo onConnected otterrà i PendingResult dal metodo getCurrentPlace definito dalla classe PlaceDetectionApi. Questo ritornerà una lista di PlacesLikelihood ognuno dei quali conterrà l'oggetto Place e la probabilità che il dispositivo si trovi realmente lì.
Scorrendo la lista, per ogni PlaceLikelihood, si andrà a concatenare alla TextView una nuova riga con il nome del punto di interesse ed il suo relativo topic, formato dalla gerarchia creata dal metodo getTopic e concatenando alla stringa ottenuta il place ID del punto di interesse in modo da identificarlo in modo univoco.

OnConnectionFailedListener definisce infine un metodo che verrà invocato da GooglePlayService quando la connessione al web service fallisce per qualche motivo.

Il costruttore dell'oggetto GeolocationListener dovrà avere come parametri l'Activity da utilizzare per ottenere il riferimento alla TextView. All'interno del costruttore vengono inoltre creati gli oggetti GoogleApiClient e Geocoder. Il codice che realizza la classe è mostrato dal listato A.24.

GeolocationActivity

La classe GeolocationActivity, come mostrato dal listato A.25, non fa nulla di speciale: tutte le principali operazioni sono eseguite all'interno della classe GeolocationListener.

Nell'activity è necessario soltanto ottenere il riferimento al LocationManager, creare un'istanza di GeolocationListener ed invocare il metodo updateLocation che richiede al LocationManager un aggiornamento singolo, basato su uno specifico provider della posizione (in questo esempio, per ottenere

[6]Nell'ottenere la gerarchia del topic si è notato che il campo AdminArea ritorna null; a volte questo avviene anche con il campo SubAdminArea.

una risposta più rapida, anche se meno precisa, si è localizzato il dispositivo utilizzando il provider di rete[7]) ed il listener che dovrà essere notificato.

Problemi riscontrati

Utilizzando l'applicazione su dispositivi differenti o con codifiche linguistiche diverse i risultati ottenuti da `Geocoder` non risultavano coincidenti, nonostante fosse esplicitato il parametro `Locale`. Lo stesso problema è stato notato invocando il web-service da browser. Per questo motivo si è abbandonata l'idea di utilizzare una gerarchia di topic che definisse in modo leggibile la posizione del dispositivo, e si è deciso di utilizzare come topic il solo Place ID. In questo modo si riesce a garantire al contempo univocità dei topic e del loro formato, indipendentemente dalla lingua o dal sistema utilizzato.

4.2.1.3 Connessione WiFi

Quando il dispositivo pubblica la sua presenza in un determinato topic geografico, dipendente dalla sua posizione; ogni gateway che sottoscrive il topic di quell'area analizza il messaggio pubblicato e, nel caso in cui il dispositivo offra l'accesso alle informazioni di sensori utili alle applicazioni in esecuzione, pubblicherà nel topic device-specific un messaggio contenente le informazioni necessarie per connettersi alla rete WiFi che mette a disposizione.

Quando il client riceve questo messaggio il servizio dovrà accedere al network programmaticamente.

Android mette a disposizione la classe `WifiConfiguration` in cui è possibile definire tutti i parametri necessari per stabile una connessione ad una particolare rete wireless: SSID, key, protocolli di sicurezza, etc. Una volta definiti i parametri l'oggetto `WifiConfiguration` può essere aggiunto al servizio di sistema di gestione delle connessioni WiFi, con cui si può interagire tramite la classe `WifiManager`. Questo servizio offre metodi per salvare le nuove configurazioni, connettere/disconnettere il dispositivo ad uno specifico network, rimuovere configurazioni e, ovviamente, abilitare/disabilitare la connessione WiFi.

I cambiamenti di stato del `WifiManager` possono essere intercettati attraverso un oggetto che estende la classe `BroadcastReceiver`. Questo intercetterà ogni cambiamento di stato e potrà essere utile nel servizio di discovery per stabilire una connessione al broker MQTT in esecuzione sul gateway dopo che il dispositivo sarà connesso alla sua rete.

Si è pensato che l'eccessivo tempo di autoconfigurazione del servizio di discovery potesse essere un problema e per questo motivo si è pensato di analizzare i tempi necessari a svolgere le operazioni.

Android Manifest e layout dell'applicazione

Per poter ricevere notifiche sui cambiamenti di stato del `WifiManager` e poter eseguire operazioni su di esso è necessario definire nel manifest del progetto degli specifici uses-permissions. In questo caso bisognerà aggiungere `ACCESS_WIFI_STATE` ed `ACCESS_NETWORK_STATE` per poter analizzare lo stato del WiFi, e per connettere/disconnettere il dispositivo ad una determinata rete si dovrà abilitare il permesso `CHANGE_WIFI_STATE`.

[7] È dunque necessario, affinché l'applicazione funzioni, abilitare nel dispositivo la geolocazione utilizzando almeno il provider di rete.

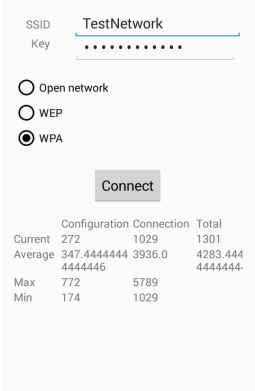
Si può inoltre definire all'interno del manifest il `BroadcastReceiver`, ma registrandolo all'interno del codice sarà possibile ricevere notifiche solo per uno specifico `Intent`: `android.net.conn.CONNECTIVITY_CHANGE`.

Il listato A.26 mostra l'Android Manifest di questo esempio.

Questa applicazione di test offrirà una via alternativa per connettere il dispositivo ad una determinata rete WiFi. Per fare ciò sarà necessario riempire un form con SSID, chiave di rete (se necessaria), e selezionare il protocollo di sicurezza utilizzato dalla rete. L'interfaccia avrà bisogno di due `EditText`, una per l'SSID, e l'altra per la chiave di rete, di un `RadioGroup` con tre `RadioButton` per selezionare il protocollo di sicurezza, e ovviamente di un `Button` per connettersi/disconnettersi.

Sono state aggiunte delle `TextView` aggiuntive che conterranno i tempi massimi, minimi e medi necessari al dispositivo per configurare una nuova rete e per connettersi ad essa, oltre che una misurazione dell'ultimo tempo di configurazione/connessione.

Figura 4.8: Android WiFi layout



È possibile ottenere il layout mostrato in Figura 4.8 utilizzando il file XML definito dal listato A.27 .

`WifiConnectionManager`

Questa classe offre alcune funzionalità che semplificano il processo di connessione e disconnessione ad una rete WiFi, definisce una classe `ConnectionTimeoutTask` (che estende `TimerTask`, utilizzata come timeout di connessione), e l'enumerazione dei protocolli di connessione.

Il costruttore necessita del riferimento a `WifiManager` in modo da memorizzarlo in una variabile interna per evitare di recuperarlo ogni volta richiedendolo all'oggetto `Context`.

Quando l'utente clicca sul pulsante di connessione il metodo `connect` viene invocato dall'`onClickListener` definito nell'`Activity` passando come parametro l'SSID della rete a cui ci si vuole connettere, la chiave, e il protocollo di sicurezza adottato. Il metodo abilita il WiFi, notifica alla classe `Measurements` l'inizio della procedura di configurazione e crea l'oggetto `WifiConfiguration` con cui si specificherà a

`WifiManager` come connettersi a quella determinata rete: per prima cosa viene settato il campo `SSID` del `WifiConfiguration` con l'SSID ricevuto come parametro, quindi il protocollo di sicurezza ricevuto viene utilizzato per configurare correttamente l'oggetto, aggiungendo specifici parametri per ogni tipologia di sicurezza. Nel caso in cui la rete a cui ci si vuole connettere sia WPA o WEP viene settato anche il campo `passphrase` con quella ricevuta come parametro.

Una volta configurato l'oggetto, questo viene utilizzato per aggiungere un nuovo network al `WifiManager` invocando il metodo `addNetwork()` e verrà notificato alla classe `Measurements` l'avvio della procedura di connessione (il cui inizio corrisponde al completamento della configurazione). Il metodo `addNetwork()` restituisce il network ID della configurazione appena inserita che verrà aggiunto ad una lista di interi contenente tutte le configurazioni create dall'applicazione. Tale lista consentirà di eliminare tutti i network aggiunti, lasciando inalterati quelli memorizzati nel sistema.

Il passo successivo sarà quello di iniziare il processo di connessione vero e proprio: per prima cosa si disconnetterà il `WifiManager`, quindi il network ID ottenuto in precedenza verrà abilitato sempre attraverso il `WifiManager`, ed in conclusione il servizio verrà riattivato.

Per evitare attese eccessivamente lunghe quando si verificano problemi di configurazione verrà creato un oggetto `Timer` per schedulare il `ConnectionTimeoutTask` dopo 30 secondi di attesa. Questo timer verrà cancellato dal metodo `stopTimer()` invocato dall'oggetto `WifiReceiver`.

Il metodo disconnect, a differenza del precedente, è molto semplice: disconnette il `WifiManager`, rimuove il network ID (salvato in un campo della classe quando viene invocato il metodo connect), e salva la configurazione di `WifiManager`[8]. Una volta terminate queste operazioni il metodo disabilita la connessione WiFi, rimuove il network ID dalla lista dei network, e setta il valore del campo `networkID` a -1.

La classe definisce anche un ulteriore metodo, utile alla rimozione di tutte le configurazioni aggiunte dall'applicazione: questo metodo scorre la lista, e per ogni elemento al suo interno invoca `removeNetwork()` sul `WifiManager` passando gli ID, salva la configurazione del servizio, e pulisce la lista.

Come detto, `WifiConnectionManager` definisce anche una classe che estende `TimerTask`. Questa classe contiene le operazioni da eseguire quando il timer avviato in fase di connessione di connessione termina, agendo da vero e proprio metodo di timeout. Quando questo accade forse il network non è configurato adeguatamente e di conseguenza il `WifiReceiver` non riceve nessuna invocazione. Per questo motivo quando va in esecuzione disconnette il `WifiManager`, rimuove il network, e salva la configurazione. Ultimate queste operazioni disabilita il WiFi e rimuove l'ID dalla lista delle configurazioni create, setta il valore del campo networkID a -1 e disabilita il timer.

Il metodo `stopTimer` controlla se l'oggetto `Timer` è `null`, in caso contrario lo cancella.

Il listato A.28 mostra la classe `WifiConnectionManager`:

WifiReceiver

Un importante componente di questo esempio è rappresentato dalla classe `WifiReceiver` che esten-

[8] Il metodo `saveConfiguration()` rende persistente la configurazione dei network. In altre parole, se si invoca questo metodo `WifiNetwork` salva la sua configurazione ed i network configurati al suo interno rimarranno disponibili anche in seguito al riavvio del dispositivo.

de BroadcastReceiver. Questa classe implementa metodi che verranno invocati da Android quando verrà eseguita una determinata azione. Per quanto riguarda i cambiamenti relativi alla connettività il WifiManager genererà un Intent con Action CONNECTIVITY_CHANGE, che verrà intercettata dal sistema, che a sua volta invocherà il metodo onReceive implementato in questa classe (se il receiver è registrato).

Il metodo verrà eseguito quando una Action identificata dalla stringa "android.net.conn.CONNECTIVITY_CHANGE" viene intercettata dal sistema operativo. Questo vuol dire che il dispositivo ha cambiato la sua connessione passando dal WiFi alla connessione dati dell'operatore o viceversa.

Il costruttore dell'oggetto WifiReceiver necessita di due oggetti come parametri: il WifiManager ed il ConnectivityManager. Questi parametri verranno memorizzati in dei campi interni alla classe in modo da non doverli ottenere nuovamente ogni volta che il metodo onReceive() va in esecuzione.

Quando onReceive() viene invocato, come prima cosa verifica se l'attributo SupplicantState di WifiManager è COMPLETED e se l'SSID a cui ci si è connessi è lo stesso definito dall'utente nell'interfaccia. Se questo controllo ha successo il processo di connessione è terminato e il dispositivo è connesso alla rete WiFi definita dall'utente. In caso contrario il client si è disconnesso, o magari ha stabilito la connessione con un altro network.

Se la connessione è andata a buon fine WifiReceiver in primo luogo notificherà alla classe Measurements il termine del processo di connessione, poi assegnerà alla stringa btnTxt il valore "Disconnect". Se il controllo non andrà a buon fine btnTxt assumerà valore "Connect". Nel caso in cui il dispositivo non si sia connesso alla rete specificata dall'utente il BroadcastReceiver invocherà il metodo deleteGatewaysNetwork() dell'oggetto WifiConnectionManager.

Terminato il controllo il receiver fermerà il timer della classe WifiConnectionManager, ed invocherà il metodo changeButtonTxt() definito nell'activity, che aggiornerà il testo del pulsante all'interno della UI con il valore della stringa btnTxt.

Il listato A.29 mostra cosa accade quando il sistema intercetta una Action di tipo CONNECTIVITY_CHANGE.

WifiActivity

L'Activity, onCreate() ottiene il riferimento al servizio WifiManager dal sistema e crea un nuovo WifiConnectionManager, ottiene i riferimenti alle View nella UI, ed associa al pulsante un nuovo onClickListener.

Al click sul pulsante, se il suo testo è "Connect" invocherà il metodo connect definito in WifiConnectionManager passando come parametri l'SSID, la chiave di rete e ID del RadioButton selezionato. Se invece il testo del pulsante è "Disconnect" il metodo non farà altro che invocare disconnect su WifiConnectionManager.

Un'altra operazione eseguita quando l'Activity viene creata è l'invocazione del metodo registerWifiReceiver, che nel caso in cui non ci sia già un WifiReceiver registrato creerà un nuovo IntentFilter aggiungendo ad esso l'Action identificata dalla stringa

"`android.net.conn.CONNECTIVITY_CHANGE`". Successivamente creerà un nuovo `WifiReceiver` e lo registrerà nel sistema insieme all'`IntentFilter` appena definito.

La classe definisce inoltre un metodo getter che consente di recuperare il valore dell'`EditText` SSID, per consentire al `WifiReceiver` di controllare se la rete a cui si è connesso il dispositivo è quella definita dall'utente, un metodo per aggiornare il testo del pulsante, ed uno per mostrare nell'interfaccia i tempi calcolati dalla classe `Measurements`.

Per evitare che il WifiReceiver sia invocato anche quando l'activity va in background, il metodo `onPause` eliminerà la registrazione del receiver, mentre il metodo `onResume` invocherà di nuovo `registerWifiReceiver`.

La classe WifiActivity è mostrata dal listato A.30.

`Measurements`

La classe `Measurements`, come nell'esempio di utilizzo delle librerie Paho per MQTT, offre dei metodi statici per tenere traccia dei tempi necessari all'applicazione per configurare e connettersi alla rete WiFi specificata dall'utente.

In questo caso i metodi invocati dall'applicazione sono tre:

`startConf` viene invocato quando il metodo connect di `WifiConnectionManager` inizia la sua esecuzione. Memorizza l'istante in cui inizia la configurazione della rete.

`startConn` viene invocato quando inizia la vera e propria procedura di connessione. Memorizza l'istante in cui viene invocato, calcola il tempo che è stato necessario per configurare la rete facendo la differenza dei timestamp, e aggiunge il risultato ad una lista in cui vengono memorizzati tutti i tempi di configurazione

`finish` viene invocato da `WifiReceiver` quando la connessione alla rete specificata dall'utente è stabilita con successo. Anche in questo caso viene calcolato il tempo necessario alla connessione andando a fare la differenza tra l'istante attuale e quello di inizio connessione, e viene aggiunto tale valore ad una seconda lista che tiene traccia dei tempi di connessione.
Dopo aver memorizzato il tempo, il metodo aggiorna le `TextView` dell'interfaccia, mostrando oltre ai tempi dell'ultima operazione (configurazione, connessione e totale) anche i tempi medi, quelli massimi e quelli minimi.

La classe definisce inoltre tre metodi con cui recuperare dalle liste il tempo medio, quello massimo e quello minimo.

Il listato A.31 mostra l'implementazione della classe di misurazione.

4.2.1.4 Sensor usage

Le applicazioni in esecuzione sul gateway potrebbero essere interessate a specifici sensori offerti dai vari device. Per questo motivo al fine di ridurre il numero di connessioni poco utili è importante cercare di filtrare i dispositivi in base ai sensori che possano fornire alle applicazioni informazioni che queste ultime ritengono interessanti: se un'applicazione in esecuzione su Kura offre soluzioni di domotica potrebbe non essere interessata ai dati di un giroscopio, e quindi un dispositivo che si connette a quel gateway potrebbe risultare inutile se non ha altri dati da offrire. È possibile realizzare questo filtro comunicando all'interno del messaggio di presentazione del device anche i sensori che mette a disposizione del gateway.

Per accedere ai sensori Android mette a disposizione un servizio di sistema, definito dalla classe `SensorManager`, che offre metodi per ottenere informazioni sui sensori disponibili e registrare dei listener su questi ultimi.

Ogni listener verrà notificato quando il sensore su cui è registrato aggiornerà i suoi dati con una nuova misurazione. L'utilizzo dei listener per inviare messaggi al broker, tuttavia, potrebbe non essere una soluzione ottimale: la minima frequenza di aggiornamento di un sensore è di circa quattro aggiornamenti ogni

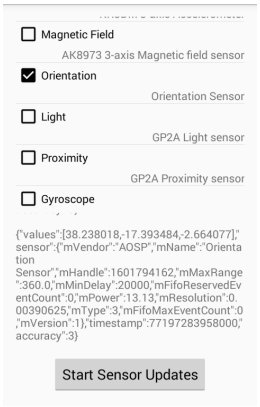
secondo, il che aumenterebbe il throughput, anche tenendo in considerazione che ogni dispositivo può mettere a disposizione un gran numero di sensori. Si è pensato che una soluzione migliore possa essere quella di utilizzare il `SensorListener` per aggiornare una lista di rappresentazioni `JSON` di `SensorEvent`[9], identificata dal tipo di sensore che genera l'aggiornamento, e creando un task schedulato periodicamente che per ogni sensore abilitato ottenga dalla lista l'ultimo aggiornamento e lo stampi nella `TextView` dell'interfaccia. Nel sistema di discovery che verrà realizzato questo task creerà un `MqttMessage` contenente tutti gli ultimi aggiornamenti e lo invierà al broker.

AndroidManifest e layout

Diversamente dai servizi di sistema visti in precedenza, l'accesso al `SensorManager` non necessita di alcuna uses-permission, quindi in questo caso l'Android Manifest sarà quello generato automaticamente da Eclipse.

Per quanto riguarda il layout, mostrato in Figura 4.9 e definito nel listato A.32 , viene utilizzato un componente `ListView` per mostrare tutti i sensori presenti nel dispositivo, ed una `TextView` in cui l'applicazione mostrerà periodicamente le rappresentazioni `JSON` dell'ultimo `SensorEvent` memorizzato da ogni sensore abilitato:

Figura 4.9: Android Sensor layout



Per mostrare tutti i sensori è stato inoltre realizzato un layout di riga personalizzato con cui si definisce che ogni riga della `ListView` sarà formata da due componenti: una `CheckBox` per selezionare se il tipo di sensore (il cui nome sarà stampato nella label del componente) sarà attivo o meno per l'applicazione, ed una `TextView` che conterrà il modello del sensore. Il layout di ogni riga della ListView è realizzato attraverso il file `sensor-row.xml` definito nel listato A.33.

[9]Un oggetto `SensorEvent` contiene tutte le informazioni relative all'aggiornamento: valori misurati, timestamp, precisione e, ovviamente il tipo di sensore che ha aggiornato i dati.

`SensorListener`

La classe `SensorListener` implementa l'interfaccia `SensorEventListener` che definisce i metodi che verranno invocati dal `SensorManager` quando un sensore esegue una nuova misurazione o modifica la sua precisione. In questo esempio quando il sensore ha ottenuto un nuovo dato, questo viene aggiunto ad uno `SparseArray`.

`SensorListener` fornisce inoltre tre metodi getter con cui ottenere gli ultimi aggiornamenti:

`getLastUpdate(int sensorType)` restituisce l'ultimo aggiornamento di uno specifico sensore

`getLastUpdatesAsString()` restituisce in un'unica stringa tutte le rappresentazioni `JSON` presenti nello `SparseArray`, separate dal carattere new line (`\n`)

`getLastUpdatesAsArray()` restituisce per intero lo `SparseArray` di stringhe

Listato 4.11: SensorListener.java

```
public class SensorListener implements SensorEventListener {

   private SparseArray<String> data = new SparseArray<String>();
   private Gson gson = new Gson();

   @Override
   public void onSensorChanged(SensorEvent event) {
      data.put(event.sensor.getType(), gson.toJson(event));
   }

   @Override
   public void onAccuracyChanged(Sensor sensor, int accuracy) {}

   public String getLastUpdate(int sensorType){
      System.out.println(""+data.get(sensorType, "No updates for " + SensorUtils.getSensorTypeAsString
           (sensorType)));
      String ret =  data.get(sensorType, "No updates for " + SensorUtils.getSensorTypeAsString(
           sensorType));
      data.remove(sensorType);
      return ret;
   }

   public String getLastUpdatesAsString(){
      String ret = "";
      for(int i = 0; i < data.size(); i++)
         ret += data.valueAt(i)+"\n";
      return ret;
   }

   public SparseArray<String> getLastUpdatesAsArray(){
      return data;
   }

   public void remove(int type) {
      data.remove(type);
   }
}
```

SensorAdapter

Per poter riempire la `ListView` in modo programmatico è stata realizzata la classe `SensorAdapter` che estende `ArrayAdapter<Sensor>`. Questa classe ridefinisce il metodo `getView`, che inserirà all'interno della `ListView` una nuova riga (che avrà il layout definito da `sensor_row.xml` ed i suoi componenti configurati opportunamente) per ogni sensore appartenente alla lista passata al costruttore come parametro.

Il listener `onCheckedChanged` associato alla `CheckBox` aggiungerà il sensore che verrà selezionato alla lista di quelli abilitati ed invocherà un metodo sull'`Activity` che si occuperà di registrare un listener. Se al contrario la `CheckBox` verrà deselezionata, la callback rimuoverà il sensore dalla lista di quelli abilitati e richiederà all'`Activity` di rimuovere il listener.

La classe fornisce inoltre due metodi pubblici: il primo restituisce la lista dei sensori abilitati, mentre il secondo consente di disabilitare tutti i sensori, e vuotare la lista di quelli selezionati.

Listato 4.12: SensorAdapter.java

```
public class SensorAdapter extends ArrayAdapter<Sensor>{
   private ArrayList<Sensor> enabledSensors = new ArrayList<Sensor>();
   private SensorActivity activity;

   public SensorAdapter(SensorActivity activity, int textViewResourceId, List<Sensor> list) {
        super(activity, textViewResourceId, list);
        this.activity = activity;
    }

   @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = (LayoutInflater) getContext()
             .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        convertView = inflater.inflate(R.layout.sensor_row, parent, false);
        final CheckBox type = (CheckBox)convertView.findViewById(R.id.checkBox1);
        final TextView name = (TextView)convertView.findViewById(R.id.textView1);
        final Sensor s = (Sensor)getItem(position);
        String sensorType = SensorUtils.getSensorTypeAsString(s.getType());
        type.setText(sensorType);
        type.setOnCheckedChangeListener(new OnCheckedChangeListener(){
         @Override
         public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
            if (isChecked){
               enabledSensors.add(s);
               activity.registerSensorListener(s);
               Log.i("CLICK ON "+type.getText().toString(), "Enabled");
            }
            else{
               enabledSensors.remove(s);
               activity.removeSensorListener(s);
               Log.i("CLICK ON "+type.getText().toString(), "Disabled");
            }
            if(enabledSensors.isEmpty() && activity.printTaskIsRunning() ){
               activity.stopPrintTask();
               activity.runOnUiThread(new Runnable(){
                  @Override
```

```
                public void run() {
                    ((TextView)activity.findViewById(R.id.updates)).append("\n\nAll Sensors Disabled"
                        );
                }
            });
        }
    }
    });
    name.setText(s.getName());
    return convertView;
 }

public ArrayList<Sensor> getEnabledSensors(){
    return enabledSensors;
}

public void removeEnabled() {
    ViewGroup v = (ViewGroup)activity.findViewById(R.id.sensorList);
    for(int i = 0; i < v.getChildCount(); i++)
        ((CheckBox)v.getChildAt(i).findViewById(R.id.checkBox1)).setChecked(false);
    enabledSensors.clear();
}
}
```

SensorUtils

La classe `SensorUtils`, definita nel listato A.34 , fornisce due metodi: il primo, ricevuto come parametro un intero che identifica il tipo sensore come definito da Android all'interno del `SensorManager`, restituisce una rappresentazione dello stesso sotto forma di stringa. Il secondo metodo esegue l'operazione inversa: data una rappresentazione in stringa del sensore restituisce il valore intero assegnatogli dal `SensorManager`.

SensorActivity

In questo esempio la classe `SensorActivity` utilizza il metodo `onCreate` per ottenere tutti i riferimenti agli elementi dalle UI, richiedere il `SensorManager`, creare un nuovo `SensorAdapter` da aggiungere alla `ListView`, ed assegnare al pulsante start/stop l'`onClickListener`.

Quando il pulsante viene cliccato, se non c'è alcuna azione schedulata, il listener controlla la lista dei sensori abilitati richiedendola al SensorAdapter. Se questa è vuota il metodo termina, altrimenti avvia una nuova schedulazione che manderà in esecuzione ad intervalli regolari (in questo caso ogni 5 secondi) il `PrintDataTask`, e cambierà il testo del pulsante. Al contrario, se c'è già una schedulazione attiva, l'utente avrà cliccato sul pulsante per fermarla; in questo caso verrà invocato il metodo `stopPrintTask()`, che dopo aver fermato l'esecuzione di `PrintDataTask` provvederà a modificare il testo del pulsante. Quest'ultimo metodo verrà invocato anche nel caso in cui tutti i sensori siano disabilitati durante la schedulazione.

La classe è definita dal listato A.35.

PrintDataTask

La classe `PrintDataTask` implementa l'interfaccia `Runnable` e realizza il task che verrà eseguito ad

intervalli regolari quando l'utente abilita l'aggiornamento. Quando viene eseguito, il metodo `run` scorrerà la lista dei sensori abilitati, per ognuno di essi richiederà al `SensorListener` l'ultimo aggiornamento ricevuto, e concatenerà la stringa `JSON` ricevuta alla `TextView`. Sarebbe stato possibile ottenere lo stesso risultato con la semplice invocazione del metodo `getLastUpdatesAsString` evitando di scorrere la lista dei sensori attivi.

La classe è implementata dal codice mostrato nel listato A.36.

4.2.2 Implementazione dell'applicazione finale

Terminata l'implementazione dei vari componenti che andranno a realizzare l'applicazione Android che consentirà la scoperta e la connessione automatica a gateway Kura, è stata iniziata l'implementazione dell'applicazione finale.

Durante lo sviluppo e la messa a punto dell'applicazione ci si è resi conto che alcuni dei componenti realizzati necessitavano di accorgimenti per poter interagire con gli altri o presentavano delle imperfezioni, e per questo motivo sono stati parzialmente modificati.

Un ulteriore problema emerso in fase di sviluppo riguarda l'interazione tra i vari componenti: essendo per la maggior parte asincroni è stato necessario introdurre un meccanismo che consentisse di sincronizzare le operazioni che ognuno di essi avrebbe svolto.

Di seguito verrà analizzata l'applicazione realizzata prestando particolare attenzione sull'implementazione dell'interazione dei vari componenti.

4.2.2.1 Creazione del progetto e configurazione del manifest

Dopo aver creato il nuovo progetto Android che realizzerà l'applicazione bisognerà come prima cosa configurarlo in modo appropriato affinché siano messe a disposizione le librerie di supporto e i componenti abbiano i permessi necessari per eseguire le loro operazioni.

Come prima cosa risulterà necessario preparare il build-path del progetto per fare in modo che tutte le librerie necessarie siano disponibili ai vari componenti. Bisognerà importare all'interno della directory `/libs` le librerie Gson e quelle Paho[10]. Sarà inoltre necessario dichiarare la dipendenza dell'applicazione dai progetti-libreria `appcompat_v7` e `google-play-services_lib`.

In aggiunta alle librerie bisognerà importare, come nel bundle Kura, un package contenente le classi dei messaggi e le loro utility.

Oltre alla preparazione del build path si dovrà configurare il manifest, aggiungendo tutti i permessi utilizzati nei precedenti esempi in modo da consentire l'accesso:

- ad Internet
- allo stato del dispositivo
- allo stato ed alle configurazioni di rete

[10]Oltre alle classiche libreria Java (`org.eclipse.paho.client.mqttv3`) anche la versione specifica per Android (`org.eclipse.phao.android.service`)

- al sistema di geolocazione
- ai Google Services

Bisognerà inoltre specificare i metadati per definire la versione e la chiave di accesso delle Google API, e la classe che implementerà il Service per MQTT. Il file manifest è mostrato nel listato A.37.

4.2.2.2 Application layout

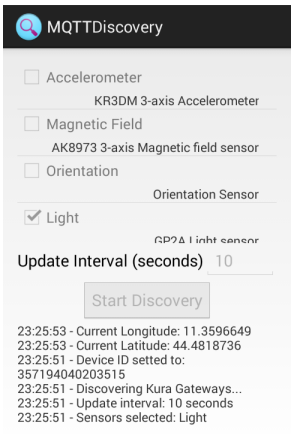
L'interfaccia grafica dell'applicazione presenterà la lista dei sensori disponibili sul dispositivo da cui l'utente potrà selezionare quali mettere a disposizione dei gateway. Il layout della lista è realizzato esattamente come nell'applicazione di esempio di accesso ai sensori vista nella sezione precedente.

L'utilizzatore dell'applicazione potrà inoltre definire l'intervallo di invio dei messaggi verso il broker locale attraverso un campo `EditText`, ed avviare il processo di discovery facendo click sull'apposito `Button`.

Nell'interfaccia sarà inoltre presente una `TextView` scorrevole che fungerà da log, in modo da tenere aggiornato l'utente sullo stato del processo di discovery.

Il layout dell'applicazione è mostrato dalla Figura 4.10 e può essere ottenuto dal codice XML mostrato nel listato A.38:

Figura 4.10: Layout dell'applicazione MQTTDiscovery



4.2.2.3 Il package `android.utils`

All'interno del package `android.utils` saranno presenti due classi: una (`Constants`) conterrà, come nel caso del bundle Kura, tutte le costanti utilizzate dall'applicazione come gli identificativi delle operazioni, i messaggi di errore, l'URI del broker MQTT su cloud, una soglia per determinare la probabilità minima di

appartenenza ad un determinato Google Place, i topic da sottoscrivere sul broker locale, ed alcuni timeout e costanti. La classe è mostrata nel listato A.39.

La seconda classe, invece, sarà la stessa classe `SensorUtils` utilizzata in precedenza e definita nel listato A.34.

4.2.2.4 Il package `android.listeners`

Il package `android.listeners` conterrà al suo interno le classi che realizzano i listener dei servizi utilizzati. Tutti i listener utilizzati nel processo di discovery (`GeolocationListener`, `MqttActionListener` ed `MqttCallBack`) estenderanno la classe `Observable`, e quando verrà creata un'istanza di queste classi registreranno come loro `Observer` l'istanza di `DiscoveryObserver` associata all'oggetto `Discovery`.

`GeolocationListener`

La classe `GeolocationListener` utilizzata dall'applicazione finale sarà differente da quella vista in precedenza. Come anticipato, testando l'applicazione di esempio si è notato che la gerarchia di topic presentava problemi con lingue e dispositivi differenti, per questo motivo si è deciso di utilizzare come topic il solo Google Place ID.

Questa scelta ha reso del tutto inutile l'utilizzo di `Geocoder` all'interno del `GeolocationListener` che quindi è stato ridefinito. La nuova versione, a differenza della precedente, non genererà più la gerarchia di topic, ma una volta ottenute le coordinate le assocerà all'oggetto `Discovery`, e avvierà la connessione del `GoogleApiClient`.

Quando la connessione sarà stabilita, ed il client avrà ricevuto la lista di possibili Place, la analizzerà e memorizzerà in due nuove liste il nome e l'ID dei Place con probabilità maggiore della soglia definita nella classe `Constants`.

Queste liste potranno essere ottenute attraverso i metodi `getTopics` e `getNames`.

Terminata l'analisi dei risultati forniti da Google il client verrà disconnesso e verrà notificato il `DiscoveryObserver`.

Listato 4.13: GeolocationListener.java

```
public class GeolocationListener extends Observable implements LocationListener, ConnectionCallbacks,
     OnConnectionFailedListener{

   private Discovery discovery;
   private GoogleApiClient mGoogleApiClient;
   private List<String>names = new ArrayList<String>();
   private List<String>topics = new ArrayList<String>();

   public GeolocationListener(DiscoveryActivity activity, Discovery discovery){
      this.discovery = discovery;
      addObserver(discovery.getDiscoveryObserver());
      mGoogleApiClient = new GoogleApiClient
            .Builder(activity)
            .addApi(Places.PLACE_DETECTION_API)
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
```

```
            .build();
    }

    @Override
    public void onLocationChanged(Location location) {
        setChanged();
        names.clear();
        topics.clear();
        discovery.setLatitude(location.getLatitude());
        discovery.setLongitude(location.getLongitude());
        mGoogleApiClient.connect();
    }

    @Override
    public void onConnectionFailed(ConnectionResult arg0) {}

    @Override
    public void onConnected(Bundle arg0) {
        PendingResult<PlaceLikelihoodBuffer> result = Places.PlaceDetectionApi
                .getCurrentPlace(mGoogleApiClient, null);
        result.setResultCallback(new ResultCallback<PlaceLikelihoodBuffer>() {
            @Override
            public void onResult(PlaceLikelihoodBuffer likelyPlaces) {
                for (PlaceLikelihood place : likelyPlaces) {
                    if (place.getLikelihood() > Constants.LIKELIHOOD_TRESHOLD){
                        names.add(getPlaceNameAsString(place.getPlace().getName()));
                        topics.add(place.getPlace().getId());
                    }
                }
                likelyPlaces.release();
                mGoogleApiClient.disconnect();
                if(topics.size()!=0)
                    notifyObservers(Constants.GEOLOCATION_DONE);
                else
                    notifyObservers(Constants.GEOLOCATION_ERROR);
            }
        });
    }

    private String getPlaceNameAsString(CharSequence name) {
        final StringBuilder sb = new StringBuilder(name.length());
        sb.append(name);
        return sb.toString();
    }

    public List<String> getTopics() {
        return topics;
    }

    public List<String> getNames() {
        return names;
    }

    @Override
    public void onConnectionSuspended(int arg0) {}

    @Override
    public void onStatusChanged(String provider, int status, Bundle extras) {}
```

```
    @Override
    public void onProviderEnabled(String provider) {}

    @Override
    public void onProviderDisabled(String provider) {}

}
```

MqttActionListener

La classe `MqttActionListener` verrà utilizzata per notificare al `DiscoveryObserver` quando una determinata operazione MQTT è stata completata, sia in caso di successo che di fallimento. La classe sarà definita come mostrato nel seguente snippet:

Listato 4.14: MqttActionListener.java

```
public class MQTTActionListener extends Observable implements IMqttActionListener {

    public enum Action{ CONNECT_C, CONNECT_GW, SUBSCRIBE, ANNOUNCE, DISCONNECT_C};
    private Action action;

    public MQTTActionListener(Action act, Discovery d) {
        action = act;
        addObserver(d.getDiscoveryObserver());
        setChanged();
    }

    @Override
    public void onFailure(IMqttToken arg0, Throwable arg1) {
        String msg = "";
        switch (action){
        case CONNECT_C: msg = "ERROR: Cloud" + Constants.MQTT_CONNECTION_ERROR + arg1.getMessage();
        break;
        case CONNECT_GW: msg = "ERROR: Gateway" + Constants.MQTT_CONNECTION_ERROR + arg1.getMessage();
        break;
        case SUBSCRIBE: msg = "ERROR: " + Constants.SUBSCRIPTION_ERROR + arg1.getMessage();
        break;
        case ANNOUNCE: msg = "ERROR: " + Constants.ANNOUNCE_NOT_PUBLISHED + arg1.getMessage();
        break;
        case DISCONNECT_C: msg = "ERROR:" + Constants.DISCONNECTION_ERROR + arg1.getMessage();
        break;
        }
        notifyObservers(msg);
    }

    @Override
    public void onSuccess(IMqttToken arg0) {
        switch (action){
        case CONNECT_C:
            notifyObservers(Constants.CLOUD_CONNECTED);
            break;
        case CONNECT_GW:
            notifyObservers(Constants.GATEWAY_CONNECTED);
            break;
        case SUBSCRIBE:
            notifyObservers(Constants.SUBSCRIPTION_CONFIRMED);
```

```
            break;
        case ANNOUNCE:
            notifyObservers(Constants.ANNOUNCE_PUBLISHED);
            break;
        case DISCONNECT_C:
            notifyObservers(Constants.CLOUD_DISCONNECTED);
            break;
        }
    }
}
```

MqttCallBack

`MqttCallBack` implementa l'interfaccia `MqttCallback` definita dalle librerie Paho, e definisce il comportamento dell'applicazione in seguito a determinati eventi legati ad MQTT:

- nel caso in cui l'applicazione dovesse perdere la connessione al broker MQTT, se è in esecuzione il task che invierà i messaggi su broker locale, bisognerà notificare il problema al `DiscoveryObserver` che provvederà a fermare lo scheduling della trasmissione.
- all'arrivo di un messaggio, si esegue in un nuovo thread la deserializzazione del payload; se si ottiene un oggetto di tipo `AnnounceContent` si provvederà a notificare il `DiscoveryObserver` inviandogli l'oggetto stesso.

Listato 4.15: MqttCallBack.java

```
public class MQTTCallBack extends Observable implements MqttCallback {

    private Discovery discovery;

    public MQTTCallBack(Discovery d) {
        this.discovery = d;
        addObserver(discovery.getDiscoveryObserver());
    }

    @Override
    public void connectionLost(Throwable arg0) {
        if (discovery.jobIsRunning()){
            setChanged();
            notifyObservers(Constants.MQTT_CONNECTION_LOST);
        }
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken arg0) {  }

    @Override
    public void messageArrived(String topic, final MqttMessage message) throws Exception {
        new Runnable(){
            @Override
            public void run(){
                Object o = MessageUtils.fromJson(new String(message.getPayload()));
                if ((o instanceof InfoGatewayContent)) {
                    setChanged();
```

```
                notifyObservers(o);
            }
        }
    }.run();
  }
}
```

SensorListener

La classe `SensorListener` sarà simile a quella vista in precedenza, ma con alcuni piccoli perfezionamenti dal punto di vista della pulizia e del riutilizzo del codice. Mentre in precedenza i dati venivano memorizzati in uno `SparseArray` di stringhe, questa volta verrà tenuta traccia del `SensorEvent` che scatena l'esecuzione del listener senza serializzarlo. Ad ogni nuovo aggiornamento verrà inserito il nuovo `SensorEvent` nello `SparseArray` identificandolo con il tipo di sensore che ha generato l'evento. In questo modo si andrà a sovrascrivere la misurazione precedente e, utilizzando la stessa istanza di `SensorListener` per ogni sensore, lo `SparseArray` conterrà sempre gli ultimi aggiornamenti ricevuti da ogni singolo sensore attivo.

La classe metterà a disposizione anche un metodo getter per ottenere lo `SparseArray`.

Listato 4.16: SensorListener.java

```
public class SensorListener implements SensorEventListener {
   private SparseArray<SensorEvent> data;

   public SensorListener(){
      data = new SparseArray<SensorEvent>();
   }

   @Override
   public void onSensorChanged(SensorEvent event) {
      data.put(event.sensor.getType(), event);
   }

   @Override
   public void onAccuracyChanged(Sensor sensor, int accuracy) {}

   public SparseArray<SensorEvent> getLastUpdatesAsArray(){
      return data;
   }
}
```

4.2.2.5 Il package `android.wifi`

Nel package `android.wifi` saranno contenute le due classi realizzate nell'esempio visto in precedenza: una fornirà i metodi per interagire con il servizio di sistema per la gestione del WiFi, e l'altra intercetterà i cambiamenti di stato della connessione in modo da capire quando il dispositivo è entrato nel network del gateway Kura.

`WifiConnectionManager`

Il comportamento della classe `WifiConnectionManager` sarà lo stesso della versione vista precedentemente anche se l'implementazione presenta alcuni dettagli differenti.

Una prima differenza è presente in fase di istanziazione dell'oggetto: nella nuova versione viene memorizzato il `DiscoveryObserver` e viene creata l'istanza di `WifiReceiver`, che in questo caso, verrà deregistrato una volta stabilita la connessione alla rete desiderata o terminato il timeout.

Il metodo `connect` segue la falsariga di quello della versione precedente. Le uniche differenze risiedono nel fatto che la nuova versione, prima di avviare la procedura di connessione, verifica se lo smartphone risulta essere già connesso alla rete del gateway, e in tal caso viene notificato il `DiscoveryObserver` senza fare altro. Una seconda differenza è riconducibile alla costruzione dell'oggetto `WifiConfiguration`: nella nuova implementazione viene realizzato in un metodo separato.

Sono stati portati all'interno del `WifiConnectionManager` anche tutti i metodi relativi alla registrazione/deregistrazione del `WifiReceiver`, oltre che un metodo aggiuntivo che consente al `WifiConnectionManager` di essere notificato quando la connessione alla rete WiFi del gateway risulta stabilita. Quando viene invocato questo metodo il `DiscoveryObserver` viene a sua volta notificato.

Le notifiche all'`Observer` vengono inviate anche quando entra in esecuzione il `ConnectionTimeoutTask` in caso di fallimento nello stabilire la connessione, o quando si invoca il metodo `disconnect`.

Listato 4.17: WifiConnectionManager.java

```
public class WifiConnectionManager extends Observable{
   private WifiManager manager;
   private int id_net = -1;
   private ArrayList<Integer> confs = new ArrayList<Integer>();
   private Timer timer;
   private String ssid;
   private WifiReceiver wifiReceiver;
   private DiscoveryActivity activity;
   private boolean receiverReady = false;

   public WifiConnectionManager(Discovery dis, WifiManager manager){
      this.manager = manager;
      addObserver(dis.getDiscoveryObserver());
      activity = dis.getDiscoveryActivity();
      wifiReceiver = new WifiReceiver(manager, this);
   }

   public void connect(String ssid, String passphrase, Security security){
      if (("\""+ssid+"\"").equals(manager.getConnectionInfo().getSSID())){
         setChanged();
         notifyObservers(Constants.WIFI_CONNECTED);
         return;
      }
      WifiConfiguration conf = setWifiConf(ssid, passphrase, security);
      if(!receiverReady)
         registerWifiReceiver();
      this.ssid = conf.SSID;
      manager.setWifiEnabled(true);
      id_net = manager.addNetwork(conf);
      confs.add(id_net);
```

```
    manager.disconnect();
    manager.enableNetwork(id_net, true);
    manager.reconnect();
    timer = new Timer();
    timer.schedule(new ConnectionTimeoutTask(), Constants.WIFI_TIMEOUT);
}

private WifiConfiguration setWifiConf(String ssid, String key, Security type){
    WifiConfiguration conf = new WifiConfiguration();
    conf.SSID = "\"" + ssid + "\"";
    switch(type){
    case OPEN:
        conf.status = WifiConfiguration.Status.ENABLED;
        conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.NONE);
        break;
    case WEP:
        if (isHexString(key))
            conf.wepKeys[0] = key;
        else
            conf.wepKeys[0] = "\"" + key + "\"";
        conf.wepTxKeyIndex = 0;
        conf.status = WifiConfiguration.Status.ENABLED;
        conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.NONE);
        conf.allowedProtocols.set(WifiConfiguration.Protocol.RSN);
        conf.allowedProtocols.set(WifiConfiguration.Protocol.WPA);
        conf.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.OPEN);
        conf.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.SHARED);
        conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.CCMP);
        conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.TKIP);
        conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.WEP40);
        conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.WEP104);
        break;
    case WPA:
        conf.preSharedKey = "\""+ key +"\"";
        conf.status = WifiConfiguration.Status.ENABLED;
        conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.TKIP);
        conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.CCMP);
        conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_PSK);
        conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.TKIP);
        conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.CCMP);
        conf.allowedProtocols.set(WifiConfiguration.Protocol.RSN);
        conf.allowedProtocols.set(WifiConfiguration.Protocol.WPA);
        break;
    default:
        return null;
    }
    return conf;
}

public void disconnect(){
    if(id_net != -1){
        manager.disconnect();
        manager.removeNetwork(id_net);
        manager.saveConfiguration();
        confs.remove((Object)id_net);
        id_net = -1;
        setChanged();
        notifyObservers(Constants.WIFI_DISCONNECTED);
    }
```

```
}

public void deleteGatewayNetworks(){
   for (int id : confs){
      manager.removeNetwork(id);
   }
   manager.saveConfiguration();
   confs.clear();
}

public void stopTimer(){
   if ((timer!=null)){
      timer.cancel();
      timer.purge();
      timer = null;
   }
}

private class ConnectionTimeoutTask extends TimerTask{
   @Override
   public void run() {
      if(receiverReady){
         unregisterReceiver();
         manager.disconnect();
         manager.removeNetwork(id_net);
         manager.saveConfiguration();
         manager.setWifiEnabled(false);
         confs.remove((Object)id_net);
         id_net = -1;
         setChanged();
         notifyObservers(Constants.WIFI_CONNECTION_ERROR);
         timer.cancel();
         timer.purge();
         timer = null;
      }
   }
}

private boolean isHexString(String key) {
   try {
      Long.parseLong(key, 16);
      return true;
   }
   catch (NumberFormatException ex) {
      return false;
   }
}

public String getSSID() {
   return ssid;
}

public void unregisterReceiver() {
   if(receiverReady){
      receiverReady = false;
      activity.unregisterReceiver(wifiReceiver);
   }
}
```

```
    public void connected() {
        stopTimer();
        unregisterReceiver();
        setChanged();
        notifyObservers(Constants.WIFI_CONNECTED);
    }

    private void registerWifiReceiver(){
        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
        ((Context)activity).registerReceiver(wifiReceiver, intentFilter);
        receiverReady=true;
    }
}
```

`WifiReceiver`

Il WifiReceiver è praticamente identico alla versione precedente. L'unica differenza nel suo comportamento dipende dalle operazioni che invocherà quando la connessione al network del gateway sarà stabilita. La nuova versione prima cancellerà la registrazione del `BroadcastReceiver` attraverso il `WifiConnectionManager`, in seguito farà arrivare al `DiscoveryObserver` una notifica relativa al completamento della connessione invocando, sempre sul `WifiConnectionManager`, il metodo `connected`.

Listato 4.18: WifiReceiver.java

```
public class WifiReceiver extends BroadcastReceiver {

    WifiManager wifiService;
    WifiConnectionManager wifiManager;

    public WifiReceiver(WifiManager wifiService, WifiConnectionManager wifiManager){
        super();
        this.wifiService = wifiService;
        this.wifiManager = wifiManager;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        if ((wifiService.getConnectionInfo().getSupplicantState().equals(SupplicantState.COMPLETED)) &&
                (wifiService.getConnectionInfo().getSSID().equals(wifiManager.getSSID()))){
            Log.i("WifiReceiver", "CONNECTED");
            wifiManager.unregisterReceiver();
            wifiManager.connected();
        }
    }
}
```

4.2.2.6 Il package `android.sensors`

Il package `android.sensors` contiene due classi: la prima definisce il task che realizzerà l'invio dei dati al broker locale, la seconda estenderà la classe `ArrayAdapter`, e sarà utilizzata per riempire la lista di sensori presente in interfaccia.

SendSensorDataTask

La classe `SendSensorDataTask` realizza il task per l'invio al gateway dei dati raccolti, che verrà schedulato ad intervalli regolari.

In fase di costruzione vengono memorizzati i riferimenti ricevuti come parametri, e viene creata una nuova istanza di `SensorListener` che verrà registrato per ogni sensore contenuto nella lista ricevuta con il messaggio `InfoGateway`.

Ad intervalli regolari il task recupererà dal `SensorListener` lo `SparseArray` di `SensorEvents`. Da ogni elemento al suo interno verrà creato un oggetto di tipo `SensorData`, che verrà a sua volta inserito in una lista. Terminato lo scorrimento dello `SparseArray` verrà creato l'oggetto `MqttMessage` da trasmettere, il cui payload sarà composto dalla serializzazione di un `Message` di tipo `SensorData` il cui contenutò sarà la serializzazione della lista appena riempita.

Il messaggio verrà quindi trasmesso sul topic "`/sensor/DEVICE_ID/`". Al termine dell'invio il task terminerà, ed attenderà di essere schedulato nuovamente.

La classe offre anche un metodo che verrà invocato quando si vorrà fermare l'esecuzione del task, e che provvederà a rimuovere la registrazione del listener rilasciando le risorse che questo stava utilizzando.

Listato 4.19: SendSensorDataTask.java

```
public class SendSensorDataTask implements Runnable{

   private MqttAndroidClient client;
   private SensorListener sensorListener;
   private SensorManager sensorManager;
   private String topic;

   public SendSensorDataTask(MqttAndroidClient client, SensorManager sensorManager, List<String>
        sensors) {
      this.client = client;
      this.sensorManager = sensorManager;
      this.sensorListener = new SensorListener();
      this.topic = Constants.SENSOR_DATA_TOPIC+client.getClientId();
      for (String s : sensors){
         Sensor sensor = sensorManager.getDefaultSensor(SensorUtils.getSensorTypeAsInt(s));
         sensorManager.registerListener(sensorListener,  sensor, SensorManager.SENSOR_DELAY_NORMAL);
      }
   }

   @Override
   public void run() {
      MqttMessage msg = new MqttMessage();
      SparseArray<SensorEvent> events = sensorListener.getLastUpdatesAsArray();
      List<SensorData> data = new ArrayList<SensorData>();
      for(int i = 0; i < events.size(); i++) {
            int key = events.keyAt(i);
            // get the object by the key.
            SensorEvent event = events.get(key);
            SensorData sensorData = new SensorData();
            sensorData.setTimestamp(event.timestamp);
            sensorData.setType(SensorUtils.getSensorTypeAsString(event.sensor.getType()));
            sensorData.setValue(event.values);
            data.add(sensorData);
```

```
        }
        Message payload = new Message(MessageType.SENSOR_DATA, MessageUtils.getJsonString(data));
        msg.setPayload(MessageUtils.getJsonString(payload).getBytes());
        msg.setQos(0);
        msg.setRetained(false);
        try {
            client.publish(topic, msg);
        } catch (MqttPersistenceException e) {
            e.printStackTrace();
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }

    public void stop(){
        sensorManager.unregisterListener(sensorListener);
    }

}
```

SensorAdapter

La classe `SensorAdapter` è simile a quella già mostrata nel precedente esempio di utilizzo dei sensori, con l'aggiunta di un controllo che disabiliterà la selezione di nuovi sensori nel caso in cui il processo di discovery sia già stato avviato. La classe metterà a disposizione anche un metodo per abilitare o disabilitare tutti gli elementi della `ListView` che andrà a riempire:

Listato 4.20: SensorAdapter.java

```
public class SensorAdapter extends ArrayAdapter<Sensor>{
    private ArrayList<Sensor> enabledSensors = new ArrayList<Sensor>();
    private boolean isListEnabled = true;
    private DiscoveryActivity activity;

    public SensorAdapter(DiscoveryActivity activity, int textViewResourceId, List<Sensor> list) {
        super(activity, textViewResourceId, list);
        this.activity = activity;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = (LayoutInflater) getContext()
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        convertView = inflater.inflate(R.layout.sensor_row, parent, false);
        final CheckBox type = (CheckBox)convertView.findViewById(R.id.checkBox1);
        final TextView name = (TextView)convertView.findViewById(R.id.textView1);
        final Sensor s = (Sensor)getItem(position);
        String sensorType = SensorUtils.getSensorTypeAsString(s.getType());
        type.setText(sensorType);
        type.setEnabled(isListEnabled);
        if(enabledSensors.contains(s))
            type.setChecked(true);
        type.setOnCheckedChangeListener(new OnCheckedChangeListener(){
            @Override
            public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
```

```
            if(isListEnabled){
               if ( (isChecked) && (!enabledSensors.contains(s))){
                  enabledSensors.add(s);
               }
               else{
                  enabledSensors.remove(s);
               }
            }
            else{
               Toast.makeText(activity, "Performing operation or already connected. You cannot select
                    sensors", Toast.LENGTH_SHORT)
               .show();
            }
         }
      });
      name.setText(s.getName());
      return convertView;
   }

   public ArrayList<Sensor> getEnabledSensors(){
      return enabledSensors;
   }

   public void removeEnabled() {
      ViewGroup v = (ViewGroup)activity.findViewById(R.id.sensor_list);
      for(int i = 0; i < v.getChildCount(); i++)
         ((CheckBox)v.getChildAt(i).findViewById(R.id.checkBox1)).setChecked(false);
      enabledSensors.clear();
   }

   public void setListEnabled(boolean val){
      isListEnabled = val;
   }

}
```

4.2.2.7 Il main package

Nel package principale dell'applicazione saranno contenute, oltre all'`Activity`, la classe che si occuperà della realizzazione vera e propria del servizio, e quella in cui sarà implementato il meccanismo di sincronizzazione delle operazioni.

`DiscoveryActivity`

La classe che realizza `l'Activity` è definita interamente nel listato A.40. Implementerà i tipici metodi che il sistema Android invocherà per la gestione del ciclo di vita, e in aggiunta, alcuni meccanismi di supporto.

All'avvio dell'applicazione verranno ottenuti i servizi di sistema per l'accesso ai sensori e alla gestione del WiFi, quindi si provvederà ad inizializzare l'interfaccia riempiendo la lista dei sensori e registrando il listener del pulsante. Al click sul `Button`, se il suo testo sarà "`Start Discovery`", si controllerà che il dispositivo sia dotato di connessione ad Internet ed in caso affermativo verrà avviato il processo di discovery.

Nel caso in cui il testo del pulsante sia "`Disconnect Gateway`" si provvederà a fermare l'esecuzione del `SendSensorDataTask` attraverso la classe `Discovery`.

Listato 4.21: DiscoveryActivity.java - Inizializzazione dell'interfaccia

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_discovery);
   sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
   connectivityManager = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
   initUI();
}

private void initUI() {
   fillSensorList();
   registerStartButton();
   status = (TextView)findViewById(R.id.status);
   status.setMovementMethod(new ScrollingMovementMethod());
   interval = (EditText)findViewById(R.id.interval);
}

private void fillSensorList() {
   //get ListView element
   sensorList = (ListView)findViewById(R.id.sensor_list);
   //creates sensor adapter and add it to sensorList
   sensorAdapter = new SensorAdapter(this, R.layout.sensor_row, sensorManager.getSensorList(Sensor.
        TYPE_ALL));
   sensorList.setAdapter(sensorAdapter);
}

private void registerStartButton() {
   //get Button element
   startButton = (Button)findViewById(R.id.start_discovery);
   startButton.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
         NetworkInfo activeNetworkInfo = connectivityManager.getActiveNetworkInfo();
         if(startButton.getText().toString().equals("Start Discovery")){
            if( activeNetworkInfo != null && activeNetworkInfo.isConnected())
               discovery = new Discovery(DiscoveryActivity.this);
            else
               appendStatus("Not connected to the internet. Unable to discover gateway.");
         }
         else
            if(discovery!=null)
               discovery.stop();
      }
   });
}
```

L'`Activity` metterà a disposizione due metodi per la disabilitazione e la riattivazione dell'interfaccia che verranno invocati rispettivamente quando verrà avviato il processo di discovery e quando questo terminerà, sia in caso di connessione al gateway, sia in caso di fallimento. Mentre il metodo di disabilitazione ogni volta che verrà invocato disattiverà ogni elemento dell'interfaccia, il comportamento del meccanismo di riattivazione sarà leggermente differente a seconda del successo o del fallimento del processo di discovery:

nel caso in cui lo smartphone fosse riuscito a connettersi ad un gateway verrà riabilitato solamente il pulsante il cui testo verrà cambiato in "`Disconnect Gateway`", se al contrario il dispositivo non fosse riuscito a trovare gateway o a connettersi, l'interfaccia verrà riattivata completamente.

Listato 4.22: DiscoveryActivity.java - Abilitazione e disattivazione dell'interfaccia

```
public void enableUI(final boolean discoveryDone) {
   runOnUiThread(new Runnable(){
      public void run(){
         if(!discoveryDone){
            for(int i = 0; i < sensorList.getChildCount(); i++){
               sensorList.getChildAt(i).findViewById(R.id.checkBox1).setEnabled(true);
            }
            sensorAdapter.setListEnabled(true);
            interval.setEnabled(true);
            startButton.setText("Start Discovery");
         }
         else
            startButton.setText("Disconnect Gateway");
         startButton.setEnabled(true);
      }
   });
}

public void disableUI() {
   runOnUiThread(new Runnable(){
      public void run(){
         for(int i = 0; i < sensorList.getChildCount(); i++){
            sensorList.getChildAt(i).findViewById(R.id.checkBox1).setEnabled(false);
         }
         sensorAdapter.setListEnabled(false);
         interval.setEnabled(false);
         startButton.setEnabled(false);
      }
   });
}
```

Altri due metodi messi a disposizione dalla classe `DiscoveryActivity` consentono l'accesso agli elementi della UI per l'aggiornamento dello stato del processo di discovery, e per mostrare dei `Toast` di avviso. Il primo metodo consente di aggiungere alla `TextView` di stato una nuova riga specificando l'orario dell'aggiornamento e l'operazione che è stata svolta:

Listato 4.23: DiscoveryActivity.java - Interazione con la User Interface

```
public void appendStatus(final String text){
   runOnUiThread(new Runnable(){
      public void run(){
         String date = new SimpleDateFormat("HH:mm:ss", Locale.ENGLISH).format(Calendar.getInstance
              ().getTime());
         status.setText(date + " - " + text + "\n" + status.getText().toString());
      }
   });
}

public void showToast(final String text) {
   runOnUiThread(new Runnable(){
      public void run(){
         Toast.makeText(DiscoveryActivity.this, text, Toast.LENGTH_SHORT).show();
```

```
        }
    });
```

Sono infine disponibili due metodi per ottenere i sensori che l'utente intende mettere a disposizione dei gateway ed il tempo di invio dei dati:

Listato 4.24: DiscoveryActivity.java - Restituzione delle scelte dell'utente

```
    public List<Sensor> getAvailableSensors() {
        return sensorAdapter.getEnabledSensors();
    }

    public long getInterval() {
        if(((TextView)findViewById(R.id.interval)).getText().toString().length()==0) return 0;
        return Integer.parseInt(((TextView)findViewById(R.id.interval)).getText().toString());
    }
}
```

`DiscoveryObserver`

Il meccanismo di sincronizzazione delle operazioni è realizzato attraverso la classe `DiscoveryObserver`. La classe riceverà notifiche dai vari listener, e a seconda del loro aggiornamento invocherà un metodo differente sull'oggetto `Discovery`.

In generale i listener trasmetteranno all'`Observer` dei valori costanti definiti all'interno della classe `Constants`. Tuttavia potrebbe capitare che l'oggetto sia di tipo differente: se l'oggetto inviato dall'`Observable` è di tipo stringa questo significherà che qualcosa non è andato a buon fine e si farà fallire il processo di discovery; se invece è un'istanza di `InfoGatewayContent`, questo verrà memorizzato all'interno di una lista che verrà consegnata all'oggetto `Discovery` una volta terminata l'attesa delle risposte dai gateway.

Terminata la geolocazione, dopo aver inviato al `Discovery` la lista dei topic geografici recuperata del `GeolocationListener`, `DiscoveryObserver` avvierà il processo di connessione al cloud.

Quando arriverà la conferma di connessione al cloud verrà invocato il metodo di sottoscrizione al primo topic della lista trasmessa in precedenza, e dopo aver ricevuto la conferma della sottoscrizione si provvederà a pubblicare il messaggio di tipo `Announce` sul topic.

Alla ricezione di questa conferma il comportamento del `DiscoveryObserver` è differente da quanto visto fino ad ora: invocherà il metodo `announcePublished`, e successivamente schedulerà il task `WaitingInfoGateway` per un'esecuzione ritardata di 30 secondi. In questo periodo accumulerà all'interno di una lista tutte le risposte provenienti dai gateway. Terminato il timeout, il task sarà schedulato, invierà all'oggetto `Discovery` tutte le informazioni ottenute dai gateway, e disattiverà il timer.

A questo punto il processo di `Discovery` potrà continuare disconnettendo il client dal cloud. A disconnessione avvenuta il `DiscoveryObserver` invocherà il metodo di connessione alla rete WiFi del gateway. Quando la connessione sarà stabilita `DiscoveryObserver` verrà notificato e potrà invocare su `Discovery` il metodo che avvierà il task application-specific.

Listato 4.25: DiscoveryObserver.java

```
/*
 * This class provides the synchronization of the discovey process steps
 */
public class DiscoveryObserver implements Observer  {
   /*
    * Task invoked when the waiting info gateway timer expires.
    */
   private class WaitingInfoGateway extends TimerTask{
      @Override
      public void run(){
         discovery.setInfoGw(infogw);
         timer.cancel();
         timer.purge();
         timer = null;
      }
   }

   private Timer timer;

   private Discovery discovery;
   private List<InfoGatewayContent> infogw;

   public DiscoveryObserver(Discovery d) {
      this.discovery = d;
   }

   /*
    * If the update object
    *       is a string, then something goes wrong and the object is the message error.
    *       is an instance of InfoGatewayContent the update method is invoked by MqttCallback.
    *                The object is added to the InfoGateway List.
    *       else is an integer defined in the Constants class. In this case will be invoked a method
       on
    *                Discovery class, according to the value of the object.
    */
   @Override
   public void update(Observable observable, Object data) {
      if(data instanceof String){
         discovery.fail((String)data);
         return;
      }
      if(data instanceof InfoGatewayContent && timer != null){
         infogw.add((InfoGatewayContent)data);
         return;
      }
      else{
         switch((int)data){
            case Constants.GEOLOCATION_DONE:
               discovery.setTopics(((GeolocationListener)observable).getTopics(), ((
                    GeolocationListener)observable).getNames());
               discovery.connectCloud();
               break;
            case Constants.CLOUD_CONNECTED:
               discovery.subscribeDeviceTopic(0);
               break;
```

```
            case Constants.SUBSCRIPTION_CONFIRMED:
               discovery.publishAnnounce();
               break;
            case Constants.ANNOUNCE_PUBLISHED:
               discovery.announcePublished();
               infogw = new ArrayList<InfoGatewayContent>();
               timer = new Timer();
               timer.schedule(new WaitingInfoGateway(), Constants.WAIT_TIME_FOR_GATEWAYS);
               break;
            case Constants.CLOUD_DISCONNECTED:
               discovery.connectGatewayWiFiNetwork();
               break;
            case Constants.WIFI_CONNECTED:
               discovery.connectLocalBroker();
               break;
            case Constants.GATEWAY_CONNECTED:
               discovery.doJob();
               break;
         }
      }
   }
}
```

Discovery

La classe `Discovery` è quella che si occupa dell'invocazione delle varie operazioni necessarie a realizzare il servizio di discovery.

In fase di inizializzazione verrà controllato che i parametri inseriti dall'utente soddisfino determinati requisiti (tempo di aggiornamento maggiore di 0 e almeno un sensore reso disponibile), e nel caso in cui non vengano riscontrati problemi il processo verrà avviato: si disabiliterà la UI, si otterrà l'IMEI del dispositivo da utilizzare come ID, e dopo aver aggiornato lo stato all'interno della `TextView` si provvederà a creare una nuova istanza di `WifiConnectionManager` e ad inizializzare la lista dei topic e dei nomi dei luoghi ad essi associati.

Concluse queste operazioni si richiederà al `LocationManager` un aggiornamento singolo (utilizzando il provider di rete per ottenere una risposta in tempi rapidi) da inviare a `GeolocationListener`.

Listato 4.26: Discovery.java - Costruttore

```
public Discovery(DiscoveryActivity activity){
   this.activity = activity;
   String msg = "";
   sensorUpdateInterval = activity.getInterval();
   //check parameters
   if(sensorUpdateInterval <= 0){
      msg = "Update interval must be upper than 0";
   }
   availableSensors = new ArrayList<String>();
   for (Sensor s : activity.getAvailableSensors())
      availableSensors.add(SensorUtils.getSensorTypeAsString(s.getType()));
   if(availableSensors.isEmpty()){
      if(msg!="")
         msg+="\n";
      msg += "No sensors selected";
   }
   if(msg!=""){
      //if not satisfied end.
      activity.showToast(msg);
      return;
   }
   else{
      //starting discovery process
      activity.disableUI();
      activity.appendStatus("Sensors selected: " + getAvailableSensorsAsString());
      activity.appendStatus("Update interval: " + activity.getInterval() + " seconds");
      activity.appendStatus("Discovering Kura Gateways...");
      //init
      discoveryObserver = new DiscoveryObserver(this);
      id = ((TelephonyManager)activity.getSystemService(Activity.TELEPHONY_SERVICE)).getDeviceId();
      activity.appendStatus("Device ID setted to: " + id);
      wifiManager = new WifiConnectionManager(this, (WifiManager)activity.getSystemService(Activity
           .WIFI_SERVICE));
      places = new ArrayList<String>();
      topics = new ArrayList<String>();
      //get location
      ((LocationManager)activity.getSystemService(Activity.LOCATION_SERVICE)).requestSingleUpdate(
            LocationManager.NETWORK_PROVIDER,
            new GeolocationListener(activity, this),
            null);
   }
}
```

Durante la sua esecuzione il `GeolocationListener` invocherà dei metodi setter per impostare i valori di latitudine e longitudine, quindi notificherà al `DiscoveryObserver` che la geolocazione del dispositivo è terminata. Il meccanismo di sincronizzazione utilizzerà un ulteriore metodo setter per trasmettere all'oggetto `Discovery` gli ID dei luoghi più probabili in cui si potrebbe trovare, ed i relativi nomi recuperandoli dal `GeolocationListener`.

Dopo aver settato la lista dei topic e dei nomi il `DiscoveryObserver` avvierà la procedura di connessione al broker su cloud. In questo metodo si preparerà il messaggio che dovrà essere trasmesso, si inizializzerà il client MQTT, e verrà richiesta la connessione al cloud broker.

Listato 4.27: Discovery.connectCloud()

```
/*
 * This method establishes the connection to the cloud. Is invoked by DiscoveryObserver
 * when the Geolocation finishes.
 */
public void connectCloud() {
   //preparing announce message
   AnnounceContent announce = new AnnounceContent(id, availableSensors, latitude, longitude);
   Message payload = new Message(MessageType.ANNOUNCE, MessageUtils.getJsonString(announce));
   mqttMessage.setPayload(MessageUtils.getJsonString(payload).getBytes());
   mqttMessage.setQos(0);
   mqttMessage.setRetained(false);
   //configuring client and callback
   client = new MqttAndroidClient(activity, Constants.CLOUD_URI, id);
   client.setCallback(new MQTTCallBack(this));
   //preparing connection option
   MqttConnectOptions connOpt = new MqttConnectOptions();
   connOpt.setCleanSession(true);
   connOpt.setConnectionTimeout(15); //15 seconds
   connOpt.setKeepAliveInterval(30); //30 seconds
   //connect
   activity.appendStatus("Connecting to cloud broker...");
   try {
      client.connect(connOpt, activity,
            new MQTTActionListener(Action.CONNECT_C, this));
   } catch (MqttException e) {
      e.printStackTrace();
      fail(e.getMessage());
   }
}
```

Stabilita la connessione, `DiscoveryObserver` invocherà il metodo di sottoscrizione del topic device-specific. Questo metodo richiede come parametro l'indice del topic da sottoscrivere. Se il parametro sarà maggiore del numero di topic trasmessi dal `GeolocationListener` vorrà dire che il servizio avrà inviato richieste su tutti i topic dei luoghi in cui potrebbe trovarsi e non ha ricevuto nessuna informazione dai gateway presenti in zona, e quindi il processo terminerà. Se invece l'indice è minore si provvederà a sottoscrivere il topic.

Quando il metodo viene invocato per la prima volta da `DiscoveryObserver` l'indice passato sarà 0. In questo caso si provvederà ad aggiornare lo stato nella `TextView` con la conferma di connessione, e ad impostare a zero il `topicCounter`, che verrà utilizzato per scorrere i topic sia dal metodo di sottoscrizione che da quello di pubblicazione. Una volta confermata la sottoscrizione del topic verrà invocato il metodo per la pubblicazione del messaggio.

Listato 4.28: Discovery.subscribeDeviceTopic(int counter)

```
/*
 * When the client is connected DiscoveryObserver invoke this method
 * for subscribing the device-specific topic.
 * The method is invoked also by chooseGateway method if there are no
 * responses from gateways
 */
public void subscribeDeviceTopic(int counter) {
   if (counter == 0){
      placesCounter=0;
      activity.appendStatus("Cloud broker connected!");
   }
   if( !(counter < places.size()) ){
      fail(Constants.NO_GATEWAY);
      return;
   }
   else{
      try {
         client.subscribe(topics.get(counter)+"/"+id, 0, activity,
               new MQTTActionListener(Action.SUBSCRIBE, this));
      } catch (MqttException e) {
         e.printStackTrace();
         fail(Constants.SUBSCRIPTION_ERROR + e.getMessage());
      }
   }
}

/*
 * Invoked by DiscoveryObserver when the subscription is done.
 */
public void publishAnnounce() {
   activity.appendStatus("Device specific topic subscribed. "
         + "(" + topics.get(placesCounter) + "/" + id + ")");
   try {
      client.publish(topics.get(placesCounter), mqttMessage, activity,
            new MQTTActionListener(Action.ANNOUNCE, this));
   } catch (MqttException e) {
      e.printStackTrace();
      fail(Constants.ANNOUNCE_NOT_PUBLISHED + e.getMessage());
   }
}
```

In seguito alla pubblicazione del messaggio il `DiscoveryObserver` invocherà il metodo `announcePublished` che provvederà ad aggiornare lo stato nell'`Activity`. Il servizio di discovery rimarrà in attesa per 30 secondi in modo da consentire a tutti i gateway all'interno della località di inviare le loro informazioni. Trascorso l'intervallo di attesa il `DiscoveryObserver` invocherà il metodo `setInfoGw` passando come parametro la lista di tutti i messaggi `InfoGateway` ricevuti. Questo metodo provvederà ad aggiornare la lista dei messaggi ricevuti dell'oggetto `Discovery`, quindi avvierà il metodo di scelta del gateway.

Quest'ultimo controllerà se la lista è vuota, ed in tal caso avvierà la sottoscrizione al topic successivo. Se invece sono stati ricevuti messaggi si selezionerà quello ricevuto dal gateway più vicino, che verrà memorizzato nella variabile `nearest` e rimosso dalla lista.

Listato 4.29: Discovery.java - ricezione delle informazioni e scelta del gateway più vicino

```
/*
 * Invoked when the announce message leaved the client. (It is QoS 0, so we cannot be
 * sure that it arrives to the broker).
 */
public void announcePublished() {
   activity.appendStatus("Announce published on \"" + places.get(placesCounter) + "\" topic.
        Waiting for responses from gateways...");
}

/*
 * Method invoked by DiscoveryObserver when the receive info-gateway timeout
 * expires. The collected messages will be passed as parameter.
 */
public void setInfoGw(List<InfoGatewayContent> infoGwCollected) {
   if(infoGwCollected.size() == 0)
      activity.appendStatus("No gateways registered in that topic.");
   else
      activity.appendStatus("Received " + infoGwCollected.size() + " gateways informations.");
   info=infoGwCollected;
   chooseGateway();
}

/*
 * This method is invoked when the client receives the info-gateway list
 * from DiscoveryObserver.
 */
private void chooseGateway() {
   if(info == null || info.isEmpty())
      subscribeDeviceTopic(placesCounter += 1);
   else{
      nearest = info.get(0);
      for(InfoGatewayContent m : info){
         if (m.getDistance()<nearest.getDistance())
            nearest=m;
      }
      info.remove(nearest);
      if(client != null)
         disconnectCloud();
      else
         connectGatewayWiFiNetwork();
   }
}
```

Selezionato il gateway più vicino allo smartphone, nel caso in cui il client sia già disconnesso dal cloud, si stabilirà la connessione alla rete WiFi del gateway invocando il metodo `connectGatewayWiFiNetwork`, che a sua volta invocherà la `connect` di `WifiConnectionManager`, passando come parametri le informazioni di rete contenute nel messaggio del gateway più vicino. Se invece il client risulterà connesso al cloud si provvederà ad interrompere la connessione e il metodo di accesso alla rete WiFi verrà invocato da `DiscoveryObserver` in seguito alla conferma della disconnessione.

Listato 4.30: Discovery.java - Disconnessione dal cloud e connessione alla rete WiFi del gateway

```
/*
 * Once the devices receives at least an info-gateway message the client disconnects the broker
 */
private void disconnectCloud() {
   try {
      client.disconnect(activity, new MQTTActionListener(Action.DISCONNECT_C, this));
      client = null;
   } catch (MqttException e) {
      e.printStackTrace();
   }
   activity.appendStatus("Cloud broker disconnected.");
}

/*
 * Invoked when the Cloud is disconnected by DiscoveryObserver
 */
public void connectGatewayWiFiNetwork() {
   activity.appendStatus("Connecting to " + nearest.getDeviceId() + " WiFi Network...");
   wifiManager.connect(nearest.getSSID(), nearest.getPassphrase(), nearest.getSecurity());
}
```

Alla ricezione della conferma di avvenuta connessione al network del gateway, l'ultima operazione da eseguire per considerare il processo completato è la connessione al broker locale mediante il metodo `connectLocalBroker`.

Listato 4.31: Discovery.connectLocalBroker()

```
/*
 * When the client is connected to the gateway WiFi network DiscoveryObserver
 * invokes this method to connect the local broker.
 */
public void connectLocalBroker() {
   activity.appendStatus("Connected to the device's WiFi network");
   client = new MqttAndroidClient(activity, nearest.getBrokerURI(), id);
   client.setCallback(new MQTTCallBack(this));
   //preparing connection option
   MqttConnectOptions connOpt = new MqttConnectOptions();
   connOpt.setCleanSession(true);
   connOpt.setConnectionTimeout(2000);
   connOpt.setWill(Constants.WILL_TOPIC, id.getBytes(), 0, false);
   connOpt.setKeepAliveInterval(500);
   try {
      client.connect(connOpt, activity,
            new MQTTActionListener(Action.CONNECT_GW, this));
   } catch (MqttException e) {
      e.printStackTrace();
      fail(e.getMessage());
   }
}
```

Quando il client MQTT sarà connesso al broker locale, `Discovery` avvierà su invocazione di `DiscoveryObserver` la schedulazione del task application-specific implementato nella classe `SendSensorDataTask` e riabiliterà il pulsante nella User Interface.

Listato 4.32: Discovery.doJob

```
/*
 * This method started when the client is connected to the gateway's local broker.
 */
public void doJob(){
   activity.appendStatus("Gateway's broker connected. Sending sensor data.");
   job = new SendSensorDataTask(client,
         (SensorManager)activity.getSystemService(Activity.SENSOR_SERVICE),
         nearest.getSensorsRequested());
   scheduleFuture = exec.scheduleAtFixedRate(job, 2, sensorUpdateInterval, TimeUnit.SECONDS);
   activity.enableUI(true);
}
```

Nel caso in cui uno dei precedenti passaggi fallisca, o se mentre si stanno inviando i dati sul broker locale viene persa la connessione, `DiscoveryObserver` verrà notificato, e riceverà come oggetto dell'update una stringa. In questo caso verrà invocato il metodo `fail` che controllerà il tipo di problema verificatosi e nel caso questo dipenda da problemi di connessione alla rete WiFi richiederà la scelta di un nuovo gateway. In caso contrario il processo di discovery terminerà, verrà aggiornato lo stato nell'`Activity` con il problema riscontrato e sarà riabilitata la User Interface.

Listato 4.33: Discovery.fail(String msg)

```
/*
 * This method started when the client is connected to the gateway's local broker.
 */
public void doJob(){
   activity.appendStatus("Gateway's broker connected. Sending sensor data.");
   job = new SendSensorDataTask(client,
         (SensorManager)activity.getSystemService(Activity.SENSOR_SERVICE),
         nearest.getSensorsRequested());
   scheduleFuture = exec.scheduleAtFixedRate(job, 2, sensorUpdateInterval, TimeUnit.SECONDS);
   activity.enableUI(true);
}
```

Dopo aver iniziato a comunicare al gateway i dati dei sensori, l'utente potrebbe voler sospendere l'invio e iniziare una nuova procedura di discovery. In questo caso facendo click sul pulsante verrà invocato il metodo `stop`. Questo provvederà a fermare la schedulazione del task application-specific, disconnetterà il client dal broker e il dispositivo dalla rete WiFi del gateway. Terminate queste operazioni, dopo aver aggiornato lo stato, riabiliterà l'interfaccia utente.

Listato 4.34: Discovery.stop()

```
/*
 * This method is invoked by DiscoveryActivity when the user clicks the
 * disconnection button.
 */
public void stop(){
   stopJob();
   if(client!= null){
      try {
         client.disconnect();
      } catch (MqttException e) {
         e.printStackTrace();
      }
      client = null;
   }
   wifiManager.disconnect();
   activity.appendStatus("Terminated.");
   activity.enableUI(false);
}

/*
 * This method stops the application-specific job.
 */
public void stopJob() {
   if(scheduleFuture != null)
      scheduleFuture.cancel(true);
   scheduleFuture = null;
}
```

La classe `Discovery`, definita nel listato A.39, è inoltre dotata di alcuni metodi getter che consentono di ottenere l'`Activity` a cui la classe fa riferimento, lo stato del task application-specific, il riferimento al `DiscoveryObserver`, e una stringa che elenca tutti i sensori abilitati da utilizzare per aggiornare lo stato all'avvio del processo.

Capitolo 5

Configurazione del gateway e test del sistema di discovery

5.1 Il sistema Raspberry-Kura

Il servizio di discovery consentirà a dispositivi mobili di trovare gateway Kura nelle loro vicinanze. I gateway, come detto in fase di analisi, saranno in esecuzione su Raspberry Pi, e questo comporta una preparazione del dispositivo. Il sistema embedded dovrà eseguire il framework Kura, un'istanza di broker MQTT (Mosquitto), e fungere da Access Point, condividendo la connessione cablata attraverso una rete WiFi. Prima di poter analizzare bundle in esecuzione, sarà quindi necessario configurare a dovere il Raspberry Pi.

Il primo step preliminare consiste nell'installazione del sistema operativo. In rete è possibile acquistare delle SD card contenenti una serie di sistemi operativi precaricati per Raspberry Pi, ma non tutte mettono a disposizione l'ultima versione di Raspbian[1] rilasciata (Jessie). Per questo motivo si è provveduto al download dell'immagine dal sito ufficiale[2], ed all'installazione su SD card.

Terminata questa operazione ed aggiornato il sistema operativo ed i pacchetti installati con i comandi

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

si potrà iniziare il vero e proprio processo di configurazione.

5.1.1 Configurazione del Raspberry Pi come Access Point Wi-Fi

Per poter configurare il Raspberry a fungere da Access Point sarà necessario che il dispositivo disponga di un adattatore WiFi e di una connessione ethernet, oltre che di alcuni software disponibili nel repository di Raspbian.

[1] Raspbian è un sistema operativo Debian-based ottimizzato per Raspberry Pi.
[2] `https://www.raspberrypi.org/downloads/raspbian/`

Dopo aver connesso il Raspberry alla rete locale ed ottenuto l'accesso ad Internet si potrà avviare la procedura di installazione delle applicazioni necessarie: hostapd ed isc-dhcp-server. Il primo è un demone che consente la realizzazione di Access Point e server di autenticazione, mentre il secondo realizzerà sul dispositivo un DHCP server che fornirà indirizzi IP in modo automatico a tutti i client che accederanno alla rete WiFi creata da hostapd. Per poter installare le applicazioni sarà sufficiente digitare da terminale i comandi

```
$ sudo apt-get update
$ sudo apt-get install hostapd isc-dhcp-server
```

Terminata l'installazione si andranno a configurare questi due servizi.

5.1.1.1 ISC DHCP server

Il Server DHCP è configurabile modificando due file. Il primo è il file di configurazione del demone DHCP, localizzato in `/etc/dhcp/dhcpd.conf`. In questo file bisognerà trovare e commentare le righe che definiscono il `domain-name`.

```
# option definitions common to all supported networks...
#option domain-name "example.org";
#option domain-name-servers ns1.example.org, ns2.example.org;
```

In modo simile bisognerà togliere il commento alla riga dove è presente il parametro `authoritative`; in questo modo si dichiarerà che il server in esecuzione su Raspberry sarà quello ufficiale per la rete che si andrà a configurare.

```
# If this DHCP server is the official DHCP server for the local
# network, the authoritative directive should be uncommented.
authoritative;
```

Quindi bisognerà aggiungere alla fine del file la configurazione specifica che si vorrà dare alla sottorete.

```
subnet 192.168.44.0 netmask 255.255.255.0 {
   range 192.168.44.10 192.168.44.100;
   option broadcast-address 192.168.44.255;
   option routers 192.168.44.1;
   default-lease-time 600;
   max-lease-time 7200;
   option domain-name "local";
   option domain-name-servers 8.8.8.8, 8.8.4.4;
}
```

Il secondo file che bisognerà editare sarà quello specifico di ISC localizzato in `/etc/default/isc-dhcp-server`, in cui si dovrà specificare su quale interfaccia rendere attivo il server DHCP. In questo caso la modifica consisterà nell'aggiungere l'interfaccia di rete WiFi, tipicamente `wlan0`, alla lista delle interfacce:

```
# On what interfaces should the DHCP server (dhcpd) serve DHCP requests?
#  Separate multiple interfaces with spaces, e.g. "eth0 eth1".
INTERFACES="wlan0"
```

5.1.1.2 hostapd

Apportate queste modifiche il server DHCP risulterà opportunamente configurato per assegnare gli indirizzi IP che vanno dal 192.168.44.10 al 192.168.44.100, e, dopo aver assegnato all'interfaccia WiFi un indirizzo IP statico, si potrà procedere con la configurazione di hostapd.

Per impostare l'IP statico sarà sufficiente modificare il file `/etc/network/interfaces` commentando quanto già presente per l'interfaccia `wlan0` ed aggiungendo una nuova definizione, come riportato di seguito:

```
iface wlan0 inet static
   address 192.168.44.1
   netmask 255.255.255.0
```

La configurazione di hostapd richiede la creazione di un file e la modifica di un altro. Quello che dovrà essere creato (`/etc/hostapd/hostapd.conf`) servirà a definire la configurazione del demone, specificando i driver dell'adattatore WiFi ed i parametri della rete che verrà creata:

```
interface=wlan0
driver=rtl871xdrv
ssid=Pi_AP
hw_mode=g
channel=7
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=0123456789
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

In `/etc/default/hostapd` dovrà essere specificato dove il demone potrà recuperare la configurazione modificando il file come segue:

```
# Uncomment and set DAEMON_CONF to the absolute path of a hostapd configuration
# file and hostapd will be started during system boot. An example configuration
# file can be found at /usr/share/doc/hostapd/examples/hostapd.conf.gz
#
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

5.1.1.3 Configurazione del NAT

Modificati questi file rimane da configurare il NAT per consentire ai client connessi alla rete WiFi di instradare i loro pacchetti sull'indirizzo ethernet in modo da accedere ad Internet. Questo è possibile aggiungendo al file `/etc/sysctl.conf` la riga `net.ipv4.ip_forward=1`, ed eseguendo i seguenti comandi:

```
$ sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
$ sudo iptables -A FORWARD -i eth0 -o wlan0 -m state --state RELATED,ESTABLISHED -j ACCEPT
$ sudo iptables -A FORWARD -i wlan0 -o eth0 -j ACCEPT
$ sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"
```

Infine bisognerà aggiungere in coda al file `/etc/network/interfaces` la seguente riga per consentire al sistema di recuperare le informazioni ad ogni riavvio del Raspberry:

```
up iptables-restore < /etc/iptables.ipv4.nat
```

Al termine di queste operazioni, riavviato il dispositivo, il Raspberry sarà a tutti gli effetti un Access Point WiFi a cui i client potranno connettersi per accedere ad Internet.

5.1.2 Installazione del broker MQTT

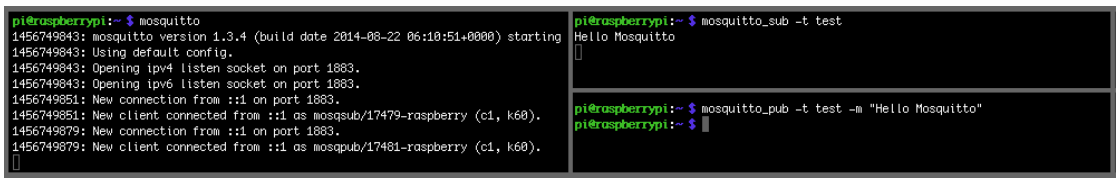
Come per i software necessari per la realizzazione dell'Access Point, anche Mosquitto è disponibile nel repository di Raspbian. In questo caso non è necessaria alcuna configurazione, e sarà sufficiente eseguire i seguenti comandi per avere il broker MQTT completamente funzionante sul proprio sistema:

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

Oltre al broker il pacchetto installato conterrà anche due client: uno per pubblicare messaggi, e l'altro per sottoscrivere topic e ricevere i messaggi pubblicati su di essi.

Per verificare che tutto funzioni adeguatamente è possibile lanciare tre diverse sessioni di terminale: una eseguirà il broker, una sottoscriverà un topic, e l'altra invierà un messaggio:

Figura 5.1: Mosquitto in funzione



5.1.3 Installazione di Kura

Per poter eseguire Kura su Raspberry Pi sarà necessario soddisfare tutte le dipendenze del framework.

Per prima cosa sarà necessario verificare la presenza di Java nel sistema digitando il comando:

```
$ java -version
```

Nel caso in cui non sia disponibile bisognerà provvedere all'installazione:

```
$ sudo apt-get install java
```

Terminata l'installazione di Java si potrà procedere con il download del pacchetto `deb` di Kura selezionando il link dalla pagina dei download del progetto[3], ed eseguendo il comando:

```
$ wget <link>
```

A download avvenuto sarà possibile installare il framework e soddisfare tutte le dipendenze in modo automatico con i seguenti comandi:

```
$ sudo apt-get update
$ sudo dpkg -i kura_*.deb
$ sudo apt-get install -f
```

Dopo aver riavviato il sistema al termine di queste operazioni Kura sarà installato e funzionante.

5.1.4 Installazione del deployment package

Per poter installare il deployment package del servizio di discovery su un framework Kura remoto esistono differenti possibilità: caricarli attraverso la view mToolkit in Eclipse, utilizzare l'interfaccia web di Kura, o caricare il deployment package all'interno del framework in modo che venga abilitato all'avvio.

Per eseguire il deployment secondo l'ultima modalità sarà necessario trasferire il deployment package nella directory `/opt/eclipse/kura/kura/packages`, e modificare il file `/opt/eclipse/kura/kura/dpa.properties` aggiungendo la seguente riga:

```
<package_name>=file\:/opt/eclipse/kura/kura/packages/<package_filename>.dp
```

Nel nostro caso `<package_name>` e `<package_filename>` corrisponderanno entrambi ad `mqtt_discovery`.

Completate queste operazioni il servizio di discovery sarà disponibile all'interno del framework dopo aver riavviato Kura.

[3] `https://eclipse.org/kura/downloads.php`

5.2 Testing del servizio di discovery lato Kura

Per poter valutare le performance e la scalabilità del sistema di discovery realizzato bisognerebbe analizzare il tempo necessario ad un gateway per rispondere ad un numero crescente di richieste proveniente da differenti clienti.

Supponendo che difficilmente le richieste arriveranno allo stesso gateway in contemporanea, si è pensato che un numero sufficientemente alto di client in grado di stressare il sistema, che rimanga al contempo realistico, si aggiri intorno alle 100 unità.

Si è inoltre deciso di analizzare il comportamento del bundle in condizioni di connettività differente.

5.2.1 Testing con JMeter

Poiché avviare in contemporanea richieste da più dispositivi risulta poco fattibile si è deciso di simulare lo scenario descritto utilizzando Apache JMeter[4], un'applicazione open source totalmente scritta in Java progettata per consentire l'esecuzione di load test e misurare le performance di applicazioni. Inizialmente fu pensato per testare Web Applications, ma successivamente è stato esteso con altre funzioni di test utilizzabili su applicazioni Java standalone.

JMeter può essere utilizzato per valutare le performance di risorse sia statiche che dinamiche, e può essere utilizzato per simulare un elevato numero di richieste ad un server/gruppo di server, in modo tale da testarne la resistenza o analizzarne le prestazioni complessive sotto diversi tipi di carico.

La duttilità di JMeter è data dai vari componenti messi a disposizione e dai plugin sviluppati da terze parti: combinandoli tra loro è possibile testare applicazioni diverse tra loro sotto diversi punti di vista.

Componenti e plugin possono essere suddivisi in gruppi. Quelli maggiormente interessanti per quanto riguarda lo studio in esame sono:

Sampler sono i componenti che eseguono il test. Ogni sampler genererà uno o più risultati, ognuno dei quali conterrà diverse informazioni (successo/fallimento del test, tempo di esecuzione, latenza, dimensione dei dati, ecc.)

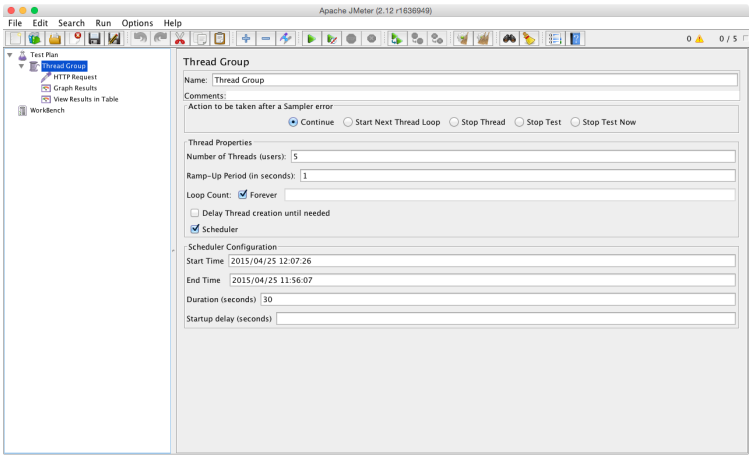
Listener consentono la visualizzazione dei risultati generati dai sampler. Forniscono diversi tipi di visualizzazione (riassunto, grafico, tabellare), e consentono il salvataggio dei risultati su file (in formato `CSV` o `XML`) e la lettura da questi ultimi

5.2.1.1 Utilizzare JMeter

L'utilizzo di JMeter risulta molto intuitivo: dopo aver scaricato e lanciato l'applicazione, per poter testare la latenza di un Web Server a seconda del numero di richieste, è sufficiente aggiungere al test plan un componente Thread Group, con cui è possibile configurare tutti i parametri relativi ai thread che invieranno richieste, come mostrato in Figura 5.2:

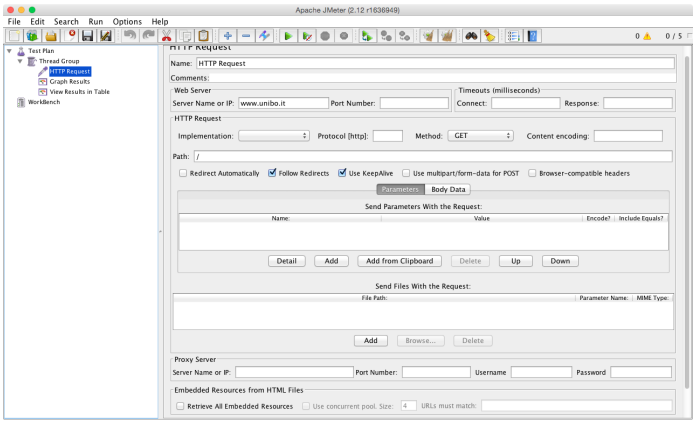
[4] http://jmeter.apache.org

Figura 5.2: JMeter - Thread Group



Dopo aver configurato opportunamente il Thread Group è possibile aggiungere a quest'ultimo un Sampler HTTP Request e configurarlo come mostrato in Figura 5.3:

Figura 5.3: JMeter - Sampler HTTP Request



A questo punto test plan è pronto e potrebbe essere lanciato, ma per poter ottenere e visualizzare i risultati è necessario aggiungere al Thread Group un listener che oltre alla visualizzazione consenta di salvare i risultati ottenuti in un file di testo. In questo esempio ne sono stati utilizzati due: il primo consente di ottenere una visualizzazione grafica dei risultati (Graph Results), mentre il secondo fornisce una visualizzazione tabellare che mostra le informazioni relative ad ogni HTTP Request inviata al server (View Results in Table).

5.2.1.2 jmeter-plugins

Il progetto jmeter-plugins[5] estende le funzionalità base di JMeter. Il progetto offre dei listener aggiuntivi in grado di generare nuovi tipi di grafici, consente la comunicazione inter-thread, offre la possibilità di caricare i risultati dei test online, e molto altro.

Questo progetto si è rivelato utile ai fini del testing del servizio di discovery grazie ad un plugin per l'analisi delle performance che riduce il consumo delle stesse rispetto a quello distribuito con JMeter. Per utilizzare quest'ultimo è necessario avere Apache Tomcat[6] in esecuzione sul server. Nel caso in esame l'esecuzione di un Application Server su un dispositivo con risorse limitate come Raspberry Pi andrebbe a ridurre quelle a disposizione di Kura. Al contrario, il plugin PerfMon distribuito nello Standard Set da jmeter-plugin insieme al proprio Server Agent –un'applicazione Java Standard che ogni secondo comunica al monitor gli aggiornamenti relativi all'utilizzo delle risorse sul server– ha un consumo di risorse nettamente inferiore a quelle utilizzate da Tomcat.

La differenza di utilizzo di risorse tra queste due possibili soluzioni è stato valutato utilizzando il comando `top` della shell Linux, ed è illustrato in Figura 5.4:

Figura 5.4: Consumo risorse

(a) Apache Tomcat

```
pi@raspberrypi ~ $ pgrep -f tomcat
4904
pi@raspberrypi ~ $ top

top - 18:20:47 up 25 min,  2 users,  load average: 0,45, 0,34, 0,25
Tasks:  72 total,   1 running,  71 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2,3 us,  1,0 sy,  0,0 ni, 96,6 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:    445740 total,   411980 used,    33760 free,    20828 buffers
KiB Swap:   102396 total,        0 used,   102396 free,   187456 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 2491 root      20   0  387m 142m  11m S   2,0 32,8   1:31.92 java
 5052 pi        20   0  4684 2456 2076 R   1,3  0,6   0:01.78 top
 4904 root      20   0  239m  43m  10m S   0,3  9,9   0:44.46 java
    1 root      20   0  2152 1352 1248 S   0,0  0,3   0:01.85 init
```

(b) PerfMon Server Agent

```
pi@raspberrypi ~ $ pgrep -f PerfMonAgent
6046
pi@raspberrypi ~ $ top

top - 18:24:32 up 29 min,  2 users,  load average: 0,24, 0,33, 0,26
Tasks:  75 total,   1 running,  74 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2,7 us,  0,3 sy,  0,0 ni, 96,7 id,  0,0 wa,  0,0 hi,  0,3 si,  0,0 st
KiB Mem:    445740 total,   384952 used,    60788 free,    22620 buffers
KiB Swap:   102396 total,        0 used,   102396 free,   181448 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 2491 root      20   0  387m 142m  11m S   1,3 32,8   1:35.62 java
 6076 pi        20   0  4684 2536 2156 R   1,0  0,6   0:00.54 top
 5173 root      20   0     0    0    0 S   0,3  0,0   0:00.29 kworker/0:1
 6046 root      20   0  203m  20m 9940 S   0,3  4,6   0:05.40 java
```

La valutazione è stata eseguita con il sistema a riposo e già in questo caso si può notare che Tomcat ha bisogno del doppio delle risorse di PerfMon. Quest'ultimo andando ad influire molto meno sulle risorse a

[5] http://www.jmeter-plugins.org
[6] http://tomcat.apache.org/

disposizione di Kura, risulta meno invasivo e consente un'analisi delle performance più accurata e dipendente quasi esclusivamente dal framework.

Un altro plugin che si utilizzerà in fase di testing sarà l'Ultimate Thread Group, che tra le feature che mette a disposizione consente di avviare i thread che rappresenteranno i clienti con un delay iniziale.

5.2.1.3 JMeter `MqttDiscoverySampler`

I sampler messi a disposizione da JMeter sono molto generici, e quelli presenti non si prestano per realizzare un testing adeguato al servizio di discovery.

Per favorire e semplificare la realizzazione di componenti ad-hoc per le differenti esigenze degli sviluppatori, JMeter mette a disposizione un set di librerie. Utilizzandole è stato possibile definire il nuovo `MqttDiscoverySampler`, semplicemente estendendo la classe `AbstractJavaSamplerClient` e ridefinendo i metodi `getDefaultParameters()` e `runTest()`.

Per poter operare con MQTT ed inviare messaggi compatibili con il formato che si utilizzerà nel servizio di discovery, sarà necessario importare le librerie Paho, Gson ed il package contenente i messaggi e le loro utility, oltre alle librerie JMeter.

Listato 5.1: MqttDiscoverySampler.java

```
public class MqttDiscoverySampler extends AbstractJavaSamplerClient implements Serializable {
   private static final long serialVersionUID = 1L;
   MqttMessage messageReceived;
   Semaphore mutex = new Semaphore(0);
   /* set up default arguments for the JMeter GUI */
   @Override
   public Arguments getDefaultParameters() {
      Arguments defaultParameters = new Arguments();
      defaultParameters.addArgument("CloudBroker", "tcp://iot.eclipse.org:1883");
      return defaultParameters;
   }

   @Override
   public SampleResult runTest(JavaSamplerContext context) {
      /* get parameters */
      SampleResult result = new SampleResult();
      String brokerAddress = context.getParameter( "CloudBroker" );
      String clientId = "client_" + new BigInteger(130, new SecureRandom()).toString(32);
      try {
         MqttConnectOptions connOpt = new MqttConnectOptions();
         connOpt.setCleanSession(true);
         connOpt.setConnectionTimeout(2000);
         connOpt.setKeepAliveInterval(500);
         MqttMessage msg = new MqttMessage();
         ArrayList<String> sensors = new ArrayList<String>();
         sensors.add("Accelerometer");
         sensors.add("Light");
         AnnounceContent announce = new AnnounceContent(clientId, sensors, 0, 0);
         Message payload = new Message(MessageType.ANNOUNCE, MessageUtils.getJsonString(announce));
         msg.setPayload(MessageUtils.getJsonString(payload).getBytes());
         msg.setQos(0);
         msg.setRetained(false);
```

```
            MqttAsyncClient mqtt = new MqttAsyncClient(brokerAddress, clientId, null);
            mqtt.setCallback(new MqttCallBack(this));
            mqtt.connect().waitForCompletion();
            mqtt.subscribe("TestPlace/"+clientId, 0).waitForCompletion();;
            result.sampleStart();
            long lat = System.currentTimeMillis();
            mqtt.publish("TestPlace", msg).waitForCompletion();
            mutex.acquire();
            result.sampleEnd();
            lat = System.currentTimeMillis() - lat;
            result.setBytes(messageReceived.getPayload().length + 2);
            result.setLatency(lat);
            result.setSuccessful( true );
            result.setResponseMessage( "Gateway discovered" );
            result.setResponseCodeOK(); /* 200 code */
            mqtt.disconnect().waitForCompletion();;
            mqtt.close();
            mqtt = null;
        } catch (Exception e) {
            result.setSuccessful( false );
            result.setResponseMessage( "Exception: " + e );
            /* get stack trace as a String to return as document data */
            java.io.StringWriter stringWriter = new java.io.StringWriter();
            e.printStackTrace( new java.io.PrintWriter( stringWriter ) );
            result.setDataType( org.apache.jmeter.samplers.SampleResult.TEXT );
            result.setResponseCode( "500" );;
        }
        return result;
    }

    public void setMessageReceived(MqttMessage msg) {
         messageReceived = msg;
         mutex.release();
    }

    class MqttCallBack implements MqttCallback{
        private MqttDiscoverySampler sampler;
        public MqttCallBack(MqttDiscoverySampler sampler) {
            this.sampler=sampler;
        }
        @Override
        public void connectionLost(Throwable arg0) {}

        @Override
        public void deliveryComplete(IMqttDeliveryToken arg0) {}

        @Override
        public void messageArrived(String arg0, MqttMessage arg1) throws Exception {
            sampler.setMessageReceived(arg1);
        }
    }
}
```

`getDefaultParameters()` definisce l'unico argomento del sampler configurabile dal pannello della GUI, e gli assegna un valore di default. Nel sampler realizzato il parametro si chiamerà `CloudBroker` e definirà l'URI del broker a cui connettersi.

`runTest()` rappresenta il vero e proprio comportamento del sampler e restituisce l'oggetto `SampleResult` che conterrà i risultati del test.

All'invocazione del metodo verranno inizializzati i parametri necessari alla simulazione ottenendo il valore del parametro `CloudBroker`, e generando un ID casuale da assegnare al client. Successivamente verrà definito un nuovo oggetto `MqttConnectOptions`, sarà generato un `MqttMessage` che conterrà un `AnnounceContent`, e verrà creato il client a cui sarà associato il meccanismo di callback. Terminata l'inizializzazione si connetterà il client e si sottoscriverà il topic device-specific.

A questo punto si avvierà il sampling e si invierà il messaggio al broker.

Ricevuta la conferma che il messaggio è stato inviato, il thread si sospenderà invocando il metodo `acquire` su un oggetto di tipo `Semaphore`. Sarà riattivato quando la callback `messageArrived` invocherà il metodo `setMessage`, che oltre a salvare il messaggio ricevuto rilascerà la risorsa `Semaphore` consentendo la ripresa dell'esecuzione del thread principale.

Quando quest'ultimo riprenderà la sua esecuzione terminerà il sampling, calcolerà la latenza e riempirà i campi dell'oggetto `result` da restituire terminata l'esecuzione.

Per poter rendere disponibile il nuovo sampler in JMeter sarà necessario esportare il progetto come archivio `.jar` e copiare quest'ultimo nella directory `<JMeterPath>/lib/ext`.

5.2.1.4 Test Plan

Il test plan JMeter sarà formato da cinque componenti:

Ultimate Thread Group (jmeter-plugins) simulerà i client che chiederanno al servizio di discovery i gateway disponibili

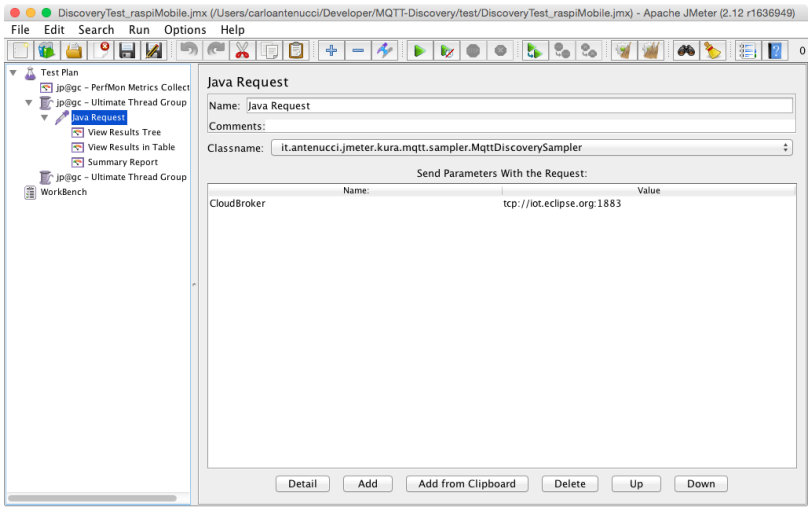
Java Request avvierà il sampler definito nella sezione precedente

View Result in Table raccoglierà i risultati in tabella e li salverà su un file

Summary Report visualizzerà un riassunto generale dei risultati ottenuti (numero di client simulati, tempo esecuzione medio, minimo e massimo, percentuale di errore ecc.). È stato utilizzato poiché consente di visualizzare i risultati temporanei del test quando quest'ultimo viene lanciato dalla riga di comando e non dalla GUI.

PerfMon Metrics Collector (jmeter-plugins) raccoglierà i dati inviati da PerfMon Server Agent, li mostrerà graficamente, e li salverà su file

Figura 5.5: JMeter Test Plan



Nei componenti verranno inserite delle stringhe al posto del numero dei client, che fungeranno da variabile e consentiranno l'automatizzazione del processo di simulazione utilizzando uno script bash.

5.2.1.5 Test launcher

La realizzazione di un test plan generico utilizzando variabili consente di evitare il tedioso lavoro di modificare di volta in volta il numero di client, lanciare il singolo test ed attendere la sua terminazione per ottenere i risultati: attraverso l'utilizzo dello script bash riportato in seguito sarà possibile automatizzare tutto il processo.

All'interno dello script si sostituirà in modo ricorsivo la variabile `_clients_` con il comando `sed`, salvando ad ogni iterazione un nuovo test plan con un numero di client sempre crescente. Bisognerà quindi lanciare la simulazione avviando JMeter e passando come parametro il file `jmx` appena generato.

Listato 5.2: testLauncher.sh

```
#!/bin/bash

jmeter="/Users/carloantenucci/Developer/apache-jmeter-2.12/bin/"
simPath="/Users/carloantenucci/Developer/MQTT-Discovery/test/"

cd $simPath
step=10
cli=0
i=0

while [ "$cli" -lt 100 ]; do
        if [ $i -lt 1 ]; then
```

```
                cli=1
        else
                ((cli=$i * $step))
        fi
        mkdir MQTT-Discovery-Test_${cli}
        sed 's/_clients_/'${cli}'/g' DiscoveryTest.jmx >
             ./MQTT-Discovery-Test_${cli}/DiscoveryTest_${cli}.jmx
        cd $jmeter
        echo "#######################################################"
        echo "# RUNNING TEST with ${cli} clients#                     "
        echo "#######################################################"
        ./jmeter -n -t ${simPath}MQTT-Discovery-Test_${cli}/DiscoveryTest_${cli}.jmx
        ((i=i+1))
        cd $simPath
done
```

5.2.2 Test del bundle

Volendo valutare come il bundle reagisce a diverse sollecitazioni si è pensato di analizzarne il comportamento simulando un numero crescente di client, ed andando ad analizzare i tempi necessari affinché questi ricevano una risposta dal gateway. Dall'analisi dei tempi di risposta si potrà capire se il sistema realizzato ha problemi di scalabilità.

I tempi di risposta non sono l'unico fattore che dovrà essere analizzato: dovendo lavorare con sistemi dalle prestazioni limitate sarà doveroso tenere d'occhio quanto l'aumento di client influirà sul consumo di risorse.

Poiché i dispositivi IoT-enabled possono utilizzare differenti tipologie di connessione, si è pensato di eseguire più volte il test cambiando di volta in volta la connessione utilizzata. I test sono stati effettuati con connettività WiFi a 100Mbps, e con reti mobili 4G, 3G e 2G. Far condividere una connessione mobile a più client sarebbe risultato poco realistico, e per questo si è deciso di mantenere lato client la connessione WiFi cambiando di volta in volta quella del Raspberry Pi.

Nelle seguenti tabelle sono mostrati i tempi di latenza ottenuti durante le varie simulazioni:

Tabella 5.1: Tempi di latenza

(a) Raspberry connesso in 2G

Client	Latenza Minima (ms)	Latenza Media (ms)	Latenza Massima (ms)
1	1225	1225,00	1225
10	1761	2508,20	4550
20	2600	2833,25	3777
30	1580	2515,90	4135
40	1964	2619,50	3477
50	2057	2977,38	3763
60	2517	4099,33	5705
70	2325	4064,20	6092
80	3132	4496,15	5780
90	2649	4259,47	5678
100	5563	5993,23	9027

(b) Raspberry connesso in 3G

Client	Latenza Minima (ms)	Latenza Media (ms)	Latenza Massima (ms)
1	368	368,00	368
10	661	706,90	815
20	938	983,85	1102
30	453	761,53	1365
40	852	1415,48	1527
50	1280	1583,52	2103
60	1428	1616,53	2089
70	1263	1616,09	1978
80	1563	1902,68	2035
90	1246	2020,43	2425
100	1177	1907,55	2510

(c) Raspberry connesso in 4G

Client	Latenza Minima (ms)	Latenza Media (ms)	Latenza Massima (ms)
1	448	448,00	448
10	335	662,80	789
20	533	734,15	1848
30	781	996,53	1115
40	378	923,80	1101
50	1299	1473,36	1634
60	1044	1398,60	1594
70	1168	1609,37	1801
80	1376	1639,35	2126
90	1610	2025,90	2754
100	1693	2072,70	2429

(d) Raspberry connesso in WiFi (100 Mbps)

Client	Latenza Minima (ms)	Latenza Media (ms)	Latenza Massima (ms)
1	351	351,00	351
10	355	584,80	904
20	361	745,85	2453
30	490	869,27	1236
40	340	897,80	1240
50	648	1174,18	1612
60	908	1144,20	1682
70	741	1329,04	1644
80	1151	1521,04	2238
90	475	1599,22	3657
100	933	1786,14	3632

Dai dati ottenuti si può notare che avviando la simulazione con Raspberry connesso in 2G i tempi di latenza vanno dal secondo abbondante ai nove, con tempi medi che oscillano tra i due e i sei secondi.

I tempi potrebbero risultare elevati, ma c'è da tenere in considerazione che la velocità di connessione offerta dal 2G è molto limitata rispetto agli attuali standard e in presenza di più client, in fase di ricezione, questa verrebbe ulteriormente ridotta tra i diversi messaggi in arrivo. Anche durante l'invio potrebbe esserci un ulteriore rallentamento dovuto al fatto che i messaggi in uscita, prima di poter essere inviati al broker, dovranno attendere il trasferimento dei messaggi precedenti.

Utilizzando connessioni più veloci si può notare infatti un netto miglioramento dei tempi: con connessione 3G e 4G i clienti riceveranno risposta mediamente in periodi che vanno dal mezzo secondo ai due,

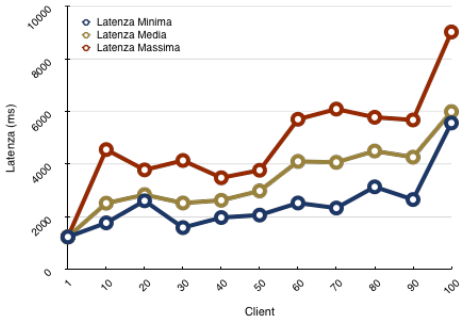
con picchi massimi di due secondi e mezzo, che nonostante siano nettamente superiori ai 400 millisecondi necessari in condizioni ottimali, rappresentano comunque un tempo accettabile.

Se si utilizza una connessione a banda larga, nonostante la velocità a disposizione sia circa dieci volte superiore, il miglioramento rispetto alle connessioni mobili di ultima generazione risulta molto limitato: il tempo medio in WiFi risulta sempre inferiore, ma la differenza con il 4G non supera mai i 500 millisecondi.

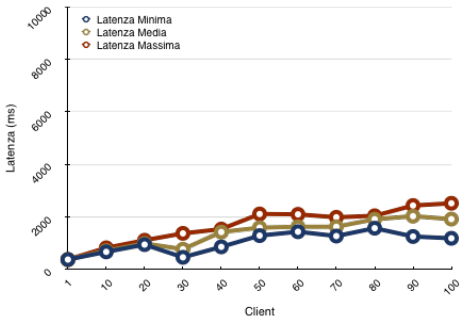
Di seguito sono mostrati i dati ottenuti in forma grafica:

Figura 5.6: Tempi di latenza

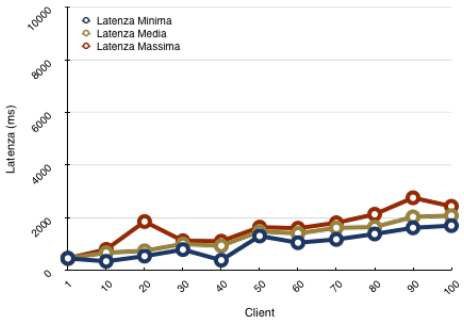
(a) Raspberry connesso in 2G



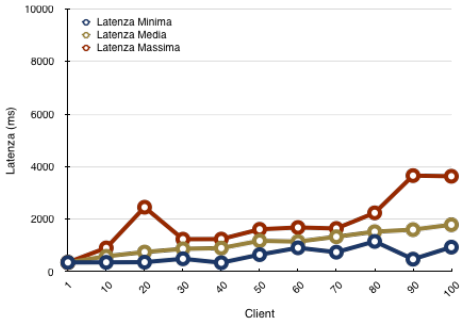
(b) Raspberry connesso in 3G



(c) Raspberry connesso in 4G



(d) Raspberry connesso in WiFi (100Mbps)



Se si analizza l'andamento al crescere dei client si può notare, tenendo in conto l'oscillazione del segnale della connessione, che le latenze medie crescono costantemente all'aumentare dei client.

Come si può notare dalle seguenti tabelle, dal punto di vista delle performance l'incremento di client comporta un maggiore utilizzo di CPU, ma non va ad intaccare l'utilizzo di memoria:

Tabella 5.2: Utilizzo risorse

(a) Raspberry connesso in 2G

Client	Utilizzo CPU medio (%)	Incremento CPU (%)	Utilizzo memoria medio* (%)	Incremento memoria (%)
1	1,53	4,04	38,69	0,01
10	2,70	9,00	38,10	0,02
20	4,21	32,65	38,82	0,01
30	6,77	39,80	38,51	0,62
40	6,25	61,00	38,88	0,02
50	6,96	71,29	38,71	0,03
60	7,94	73,00	38,73	0,02
70	9,35	75,76	38,88	0,02
80	10,18	70,71	38,72	0,04
90	10,96	77,78	38,73	0,04
100	12,15	70,30	38,72	0,03

(b) Raspberry connesso in 3G

Client	Utilizzo CPU medio (%)	Incremento CPU (%)	Utilizzo memoria medio* (%)	Incremento memoria (%)
1	2,12	4,12	37,91	0,01
10	2,70	18,18	37,89	0,02
20	3,98	20,42	37,93	0,03
30	4,53	32,00	37,94	0,01
40	6,13	55,00	37,96	0,01
50	7,11	77,00	37,88	0,03
60	7,72	83,00	37,87	0,01
70	9,86	80,98	37,90	0,03
80	10,02	98,00	38,09	0,02
90	11,25	97,03	38,07	0,05
100	12,34	81,00	37,90	0,04

(c) Raspberry connesso in 4G

Client	Utilizzo CPU medio (%)	Incremento CPU (%)	Utilizzo memoria medio* (%)	Incremento memoria (%)
1	1,42	3,06	35,72	0,02
10	3,21	13,27	35,77	0,08
20	3,96	31,00	35,90	0,11
30	4,68	28,57	37,77	0,04
40	5,90	61,00	36,25	0,21
50	7,28	65,66	43,01	0,01
60	7,80	84,85	37,74	0,02
70	9,28	93,05	37,85	0,02
80	10,57	96,00	37,78	0,02
90	11,42	76,77	37,84	0,03
100	12,54	97,00	37,79	0,03

*Memoria utilizzata globalmente dal Raspberry Pi

Client	Utilizzo CPU medio (%)	Incremento CPU (%)	Utilizzo memoria medio* (%)	Incremento memoria (%)
1	1,90	5,10	37,76	0,03
10	2,78	13,13	37,73	0,01
20	3,96	23,00	37,72	0,01
30	5,33	45,93	37,72	0,01
40	6,07	47,47	37,72	0,01
50	7,26	77,00	37,71	0,01
60	8,25	90,00	37,75	0,02
70	9,55	71,00	37,75	0,02
80	11,05	91,99	37,77	0,01
90	11,87	78,22	37,77	0,03
100	13,29	88,00	37,78	0,03

(d) Raspberry connesso in WiFi (100 Mbps)

*Memoria utilizzata globalmente dal Raspberry Pi

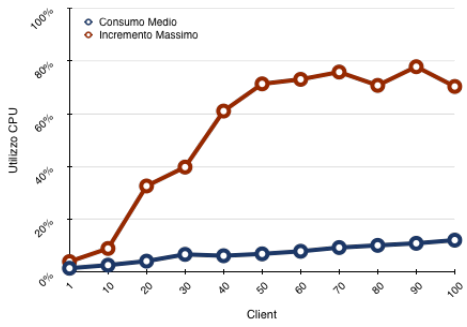
Benché gli elevati picchi di utilizzo della CPU possano far pensare che il sistema non riesca a reggere un numero di client superiore alle 40 unità, è bene notare che l'utilizzo medio si mantiene inferiore al 20% con qualsiasi tipologia di connessione.

Un altro aspetto interessante riguarda il fatto che, con una connessione più lenta (2G) i picchi di utilizzo risultino inferiori del 10%. Questo potrebbe dipendere dal fatto che la lentezza della connessione faccia arrivare i messaggi con un leggero ritardo. In questo modo il dispositivo riuscirebbe ad elaborarne alcuni mentre gli altri non sono ancora stati ricevuti alleggerendo il carico.

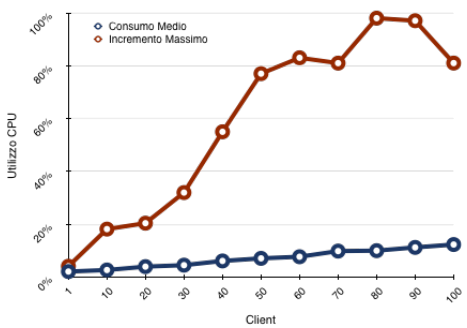
Di seguito si possono osservare i risultati in termini di utilizzo risorse (incremento ed utilizzo medio di CPU) in forma grafica:

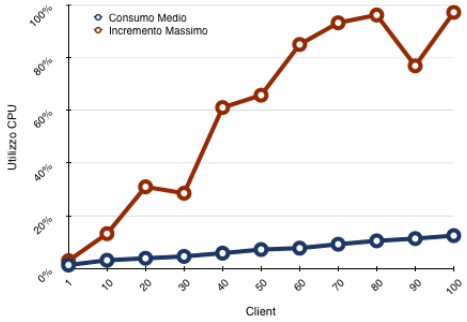
Figura 5.7: Utilizzo risorse

(a) Raspberry connesso in 2G

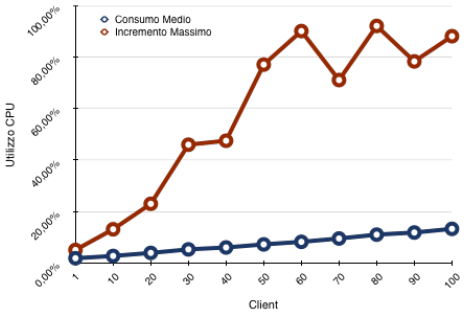


(b) Raspberry connesso in 3G





(c) Raspberry connesso in 4G



(d) Raspberry connesso in WiFi (100Mbps)

5.3 Testing dell'applicazione Android

5.3.1 Test dei componenti

Come accennato, per gli esempi di utilizzo del servizio MQTT offerto dalle librerie Paho e per quello di configurazione e connessione ad una rete WiFi sfruttando il `WifiManager`, si è tenuta traccia dei tempi necessari al dispositivo per completare le azioni richieste.

Oltre all'analisi dei tempi si è utilizzato PowerTutor[7] per valutare quanto l'utilizzo delle applicazioni incida sul consumo della batteria del dispositivo.

Di seguito saranno analizzati i risultati ottenuti per entrambi gli esempi, testati su un Samsung Nexus S, in modo da valutare quanto questi due elementi che andranno a comporre il servizio di discovery possano influenzare il servizio stesso.

5.3.1.1 MQTT

Al fine di avere un'idea delle tempistiche di interazione tra il servizio MQTT offerto da Eclipse Paho ed il broker sono stati realizzati alcuni test andando a modificare gli scenari: sono stati analizzati i tempi di comunicazione in presenza di connessione WiFi o mobile (3G o 2G), con broker locale o remoto, e variando il numero di messaggi inviati contemporaneamente per analizzare quanto incidano questi fattori.

Al fine di avere un numero di campioni che rendano le misurazioni il più possibile veritiere ogni operazione è stata eseguita per 20 volte. In questo modo sono state ottenute 20 misurazioni di tempo per connessione, invio di un singolo messaggio, sottoscrizioni, e cancellazione delle stesse. Per quanto riguarda l'invio di più messaggi contemporaneamente sono state ottenute 20 misurazioni di tempo di invio minimo, 20 di tempo medio e 20 di tempo massimo.

Nelle seguenti tabelle vengono riportati i tempi minimi massimi e medi di ogni operazione.

Tabella 5.3: Tempi di interazione con MQTT broker

	MIN (ms)	AVG (ms)	MAX (ms)
Connection	71	172,84	400
Subscription	13	65,79	142
Unsubscription	36	81,68	129
1 Message	36	93,73	158
5 Messages	142	149,46	155
10 Messages	220	227,58	264
20 Messages	386	421,52	488
50 Messages	1015	1051,72	1142

(a) Connessione WiFi, broker locale

[7] https://play.google.com/store/apps/details?id=edu.umich.PowerTutor

	MIN (ms)	AVG (ms)	MAX (ms)
Connection	74	144,05	269
Subscription	148	155,60	164
Unsubscription	148	154,35	169
1 Message	59	107,05	210
5 Messages	192	433,06	544
10 Messages	203	465,19	670
20 Messages	Error	Error	Error
50 Messages	Error	Error	Error

(b) Connessione WiFi, broker remoto

	MIN (ms)	AVG (ms)	MAX (ms)
Connection	370	466,37	969
Subscription	168	334,38	634
Unsubscription	155	265,79	549
1 Message	178	457,11	1038
5 Messages	405	704,46	827
10 Messages	343	886,74	1008
20 Messages	Error	Error	Error
50 Messages	Error	Error	Error

(c) Connessione 3G, broker remoto

	MIN (ms)	AVG (ms)	MAX (ms)
Connection	1171	1473,16	2253
Subscription	383	846,05	1551
Unsubscription	504	862,79	1291
1 Message	690	823,47	1112
5 Messages	866	1696,62	2020
10 Messages	947	1716,88	1924
20 Messages	Error	Error	Error
50 Messages	Error	Error	Error

(d) Connessione 2G, broker remoto

Come si può notare, e come era possibile immaginare, le prestazioni migliori si ottengono comunicando con un broker interno alla rete locale. Tuttavia sfruttando una connessione a banda larga è possibile ottenere all'incirca gli stessi tempi di interazione anche con un broker remoto, fatta eccezione per l'invio di più messaggi contemporaneamente. Se si utilizza invece una connessione mobile i tempi di interazione sono nettamente maggiori: con copertura 3G la comunicazione con il broker risulta leggermente più lenta, ma può essere ancora considerata accettabile, mentre se la rete non consente di andare oltre il 2G si può notare un notevole rallentamento nell'interazione con il broker.

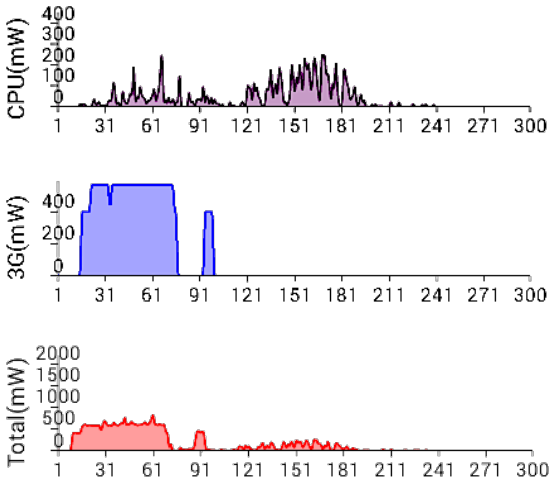
Dai risultati ottenuti si può constatare, inoltre, che in caso di connessione a broker locale è stato possibile inviare contemporaneamente fino a 50 messaggi, mentre lavorando con un broker remoto il numero massimo di invii contemporanei si riduce a 10: il client MQTT con un numero di messaggi maggiore solleverà una eccezione per un eccessivo numero di messaggi in pubblicazione. Si è ipotizzato che questa limitazione sia

dovuta alla dimensione del buffer dei messaggi in uscita, che in presenza di broker locale riesce a vuotarsi più velocemente consentendo l'invio di più messaggi.

Per analizzare il consumo di batteria l'applicazione è stata sottoposta ad un utilizzo intensivo sotto tutti i punti di vista: sono state effettuate un elevato numero di operazioni per cinque minuti sfruttando sia la connessione dati che quella a banda larga.

Utilizzando PowerTutor, si è potuto osservare che il consumo è risultato abbastanza elevato: l'applicazione ha assorbito una media di 207 mW con picchi di circa 1W, riconducibili principalmente all'utilizzo della connessione dati, come mostrato in Figura 5.8:

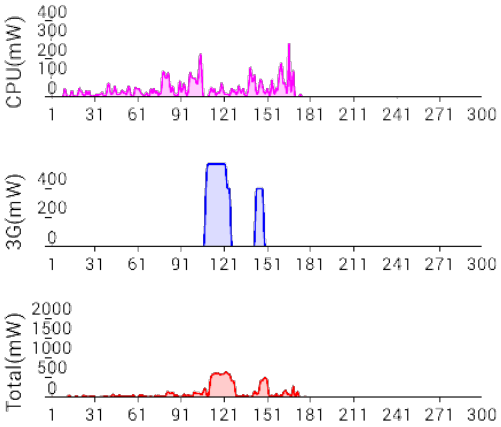
Figura 5.8: Utilizzo batteria



Poiché il consumo di batteria è risultato decisamente elevato, tenendo in considerazione che durante il test l'applicazione è stata sottoposta a continue richieste inviando messaggi a broker locali e remoti con connessione WiFi e 3G, si è deciso di eseguire un secondo test in cui si è cercato di replicare il comportamento del servizio di discovery: invio di un messaggio al broker remoto in 3G, ricezione di alcuni messaggi, connessione al broker locale, ed invio di messaggi ad intervalli regolari a quest'ultimo.

In questo caso l'assorbimento di potenza risulta ben diverso. Il consumo medio di batteria per un utilizzo di tre minuti è di circa 100 mW, e anche se può risultare ancora eccessivo questo non dovrebbe creare preoccupazioni. Come si può notare dai grafici in Figura 5.9 il fattore che più influenza il consumo è ancora una volta l'utilizzo della connettività mobile per comunicare con il broker remoto. In questa fase l'applicazione assorbe fino a 500mW di potenza, ma una volta stabilita la connessione al broker locale, inviando un messaggio in WiFi ogni 5 secondi, il consumo si riduce a meno di 50mW. Considerando che l'utilizzo della connessione mobile per la comunicazione tra broker e device è limitato all'invio del messaggio di presenza e alla ricezione dei parametri di connessione ai gateway, si può dire che l'assorbimento in potenza dell'applicazione può essere ritenuto accettabile.

Figura 5.9: Utilizzo batteria



5.3.1.2 WiFi

Si è deciso di analizzare anche i tempi necessari per configurare e stabilire una connessione ad una determinata rete WiFi, e anche in questo caso, si è valutato l'utilizzo di batteria da parte dell'applicazione. Di seguito sono riportati i tempi minimi, massimi e medi di configurazione e di connessione ad una rete WiFi, a seconda del protocollo di sicurezza adottato. Anche in questo caso per 20 diversi campioni.

Tabella 5.4: Tempi di configurazione e connessione ad una rete WiFi

	MIN (ms)	AVG (ms)	MAX (ms)
Configuration	12	17,65	29
Connection	1135	4158,70	5121
Total	1150	4176,35	5133

(a) Connessione ad Open Network

	MIN (ms)	AVG (ms)	MAX (ms)
Configuration	17	23,40	32
Connection	3096	4684,45	5004
Total	3116	4707,85	5027

(b) Connessione a WEP Network

	MIN (ms)	AVG (ms)	MAX (ms)
Configuration	141	196,65	848
Connection	1437	4773,50	4970,15
Total	1584	265,79	5709

(c) Connessione a WPA-PSK (TKIP) Network

	MIN (ms)	AVG (ms)	MAX (ms)
Configuration	116	172,90	244
Connection	4839	4928,55	5101,45
Total	4999	862,79	5419

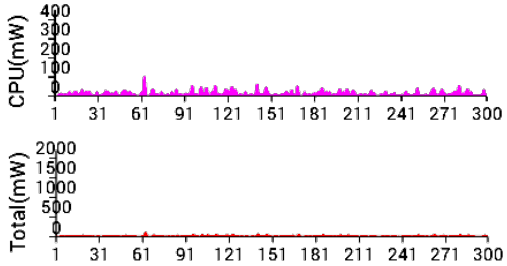
(d) Connessione a WPA2-PSK (EAP) Network

Dai dati riportati in tabella si evince che i tempi di configurazione per reti aperte e WEP sono simili, mentre la configurazione di una rete WPA-WPA2 è più lenta. Lo stesso discorso, anche se in proporzioni minori, può essere fatto per i tempi di connessione. Sono necessari mediamente 4,5 secondi per stabilire una connessione alla rete: ci si può connettere ad una rete aperta in quattro secondi abbondanti, serve mezzo secondo in più per accedere ad una rete WEP, mentre per stabilire una connessione ad una rete WPA-WPA2 sono necessari circa cinque secondi.

Questi risultati suggeriscono che la differenza nelle tempistiche di configurazione/connessione sono minime, per tanto sarà possibile utilizzare qualsiasi tipo di rete senza che questa influenzi eccessivamente il servizio di discovery.

Monitorando l'utilizzo della batteria con PowerTutor per tutta la durata del test si è potuto notare, come mostrato nei grafici riportati in Figura 5.10, che l'applicazione, pur connettendosi e disconnettendosi dalla rete WiFi con diverse configurazioni di sicurezza un elevato numero di volte, ha un consumo mediamente limitato (13mW). Si può quindi dire che, anche in questo caso, l'assorbimento di potenza da parte dell'applicazione risulta accettabile.

Figura 5.10: Utilizzo batteria



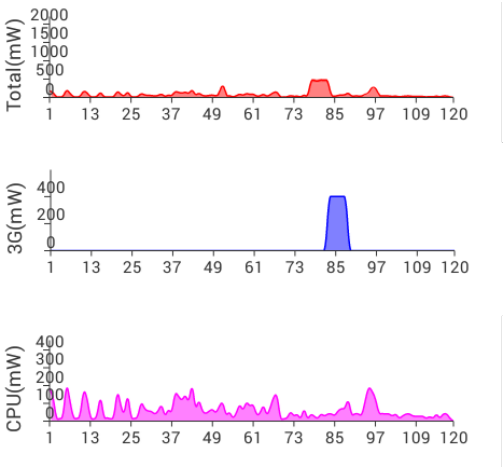
5.3.2 Test dell'applicazione

In considerazione del fatto che l'applicazione Android utilizzerà dei timeout, per il testing dell'applicazione si è pensato che l'analisi dei tempi necessari al discovery fosse superflua. Al contrario è di fondamentale importanza valutare quanto l'applicazione incida sul consumo delle risorse. Per questo motivo, anche in questo caso, si è quindi utilizzato PowerTutor per analizzare i consumi.

Dopo due minuti di simulazione in cui il dispositivo ha eseguito il discovery, si è connesso al gateway ed ha inviato i dati di tutti i sensori ogni dieci secondi, si è potuto notare che l'applicazione ha assorbito mediamente 90mW di potenza con un picco massimo di circa 500mW.

Come si può notare dai grafici in Figura 5.11 i risultati ottenuti nel secondo test di utilizzo di MQTT su Android sono stati confermati: l'assorbimento massimo si verifica quando c'è utilizzo della connessione dati. Terminata la fase di discovery i consumi rimangono comunque accettabili nonostante siano stati attivati tutti i sensori.

Figura 5.11: Utilizzo batteria



5.4 Considerazioni finali

Alla luce di quanto emerso dai differenti test realizzati si può dire che il bundle Kura, a fronte di un numero di client elevato ma comunque reale, scali abbastanza bene e riesca a rispondere in tempi utili a tutte le richieste ricevute.

Anche dal punto di vista dell'utilizzo di risorse ci si può ritenere soddisfatti: l'utilizzo di memoria è totalmente irrilevante, mentre i picchi di accesso alla CPU che si hanno durante le elaborazioni delle richieste non costituiscono un problema, considerando che l'utilizzo medio rimane comunque limitato.

Anche i test effettuati sull'applicazione Android possono essere ritenuti positivi: il consumo medio di batteria, mettendo a disposizione tutti i sensori e comunicando i dati ogni 10 secondi, rimane limitato ed ha picchi elevati soltanto durante l'utilizzo della connessione dati per inviare e ricevere messaggi, che in situazioni normali non dovrebbe impattare in modo considerevole.

L'utilizzo di un payload con i campi strettamente necessari al riconoscimento e all'autoconfigurazione dell'applicazione Android ha consentito di ottenere messaggi di discovery di dimensioni limitate: quelli che i gateway inviano ai client, anche nel caso vengano richiesti al dispositivo tutti i sensori, saranno di circa 550 Byte[8]. Lo stesso messaggio, utilizzando XML raddoppierebbe la sua dimensione.

Un'ulteriore ottimizzazione ha consentito di ridurre il traffico dati verso il gateway raccogliendo costantemente i dati dai sensori e inviandoli in un unico messaggio. In questo modo si è limitato il numero di messaggi da deserializzare all'interno del bundle, il throughput di rete e di conseguenza il consumo di batteria client-side.

[8] La lunghezza dei parametri di tipo stringa potrebbero far ridurre o aumentare la dimensione del payload.

Conclusioni

L'obiettivo del progetto di questa tesi era quello di realizzare un servizio di discovery che, grazie all'utilizzo del protocollo di MQTT, consentisse a dispositivi dalle limitate risorse computazionali di scoprire, connettersi e comunicare con gateway IoT che utilizzano il framework Kura.

Il progetto è stato portato a termine introducendo al contempo diverse ottimizzazioni per migliorare i tempi di risposta dei gateway ed il consumo di batteria lato client.

In fase implementativa sono state prese alcune decisioni che hanno portato ad un miglioramento delle performance delle due applicazioni consentendo la realizzazione di un servizio di discovery MQTT-based che fornisce supporto alla ricerca di gateway in prossimità dello smartphone, realizza un meccanismo di auto-configurazione per l'accesso del dispositivo alla rete WiFi messa a disposizione dal gateway, consente l'invio dei dati application-specific al broker MQTT locale, ed è facilmente configurabile attraverso l'interfaccia web messa a fornita da Kura.

Dall'analisi dei risultati ottenuti in fase di testing si può dire che il sistema realizzato non presenti problemi di scalabilità in condizioni di elevato carico ed abbia un limitato utilizzo di risorse. Questo ne consente l'esecuzione su sistemi con capacità computazionali limitate.

Al termine dell'implementazione sono stati realizzati due componenti: il servizio in esecuzione su Kura e l'applicazione Android.

Il service Kura riceve i messaggi di presenza dei client e, se interessato ai loro sensori, risponde con un messaggio contenente le informazioni per l'auto-configurazione dell'accesso alla rete WiFi. Il service sarà in esecuzione all'interno del framework Kura installato su un Raspberry Pi, che metterà a disposizione dei client un broker MQTT Mosquitto e fungerà da Access Point WiFi.

L'applicazione Android avrà il compito di trovare i gateway nelle vicinanze, connettersi ad uno di questi, ed inviare i dati dei sensori richiesti. L'applicazione realizzata avrà quattro diversi meccanismi per poter eseguire le operazioni richieste: sfrutterà le librerie Paho per realizzare il client MQTT da utilizzare in fase di comunicazione, otterrà le sue coordinate ed i possibili Google Place in cui potrebbe essere localizzato lo smartphone, configurerà l'accesso alla rete WiFi del gateway e si collegherà al broker locale utilizzando le informazioni ricevute in risposta al messaggio di presenza, e raccoglierà i dati dei sensori inviandoli al broker

locale del gateway da intervalli regolari.

In definitiva utilizzando il servizio realizzato sarà possibile trovare un gateway Kura nelle vicinanze del proprio smartphone, connettersi ad esso, e comunicargli i dati dei sensori del dispositivo in modo diretto utilizzando il protocollo MQTT che consente una comunicazione snella anche in condizioni di connettività limitata.

Benché quanto realizzato soddisfi gli obiettivi iniziali, il progetto non è che un punto di partenza. Il discovery in IoT deve tener conto di molti fattori, tra cui gli adattatori di rete disponibili nei vari dispositivi: non tutti gli smart device sono dotati di scheda WiFi o di connessione dati; una futura estensione potrebbe riguardare l'implementazione del servizio di discovery con standard di comunicazione differenti, come ad esempio Bluetooth e, utilizzando MQTT-SN, ZigBee. Queste soluzioni consentirebbero di estendere l'utilizzo del servizio anche a dispositivi dalle caratteristiche ancora più limitate di quelle viste in questo progetto e potrebbe essere interessante valutarne le performance.

Un altro sviluppo potrebbe riguardare l'auto-configurazione del bundle Kura in modo da esentare l'amministratore dal configurare tutti i parametri. Sarebbe utile, ad esempio, fare in modo che il gateway ad ogni avvio si geolocalizzi o configuri in automatico l'indirizzo del broker locale.

Inoltre potrebbe risultare interessante valutare le performance di una soluzione che preveda l'introduzione di un terzo elemento di architettura che operi in modo simile a quello proposto in fase di analisi, ma che agisca a livello di località riducendo eventuali problemi di scalabilità. Questa soluzione potrebbe prevedere che in caso di presenza dei nodi di località questi ultimi tengano traccia dei gateway nella loro area di copertura e inviino le informazioni ai client in un unico messaggio. Nel caso in cui una località sia sprovvista di nodo centrale, o che quest'ultimo sia momentaneamente guasto, il servizio potrebbe operare come descritto in questo lavoro.

Appendice

A.1 Messaggi ed utility generali

Come anticipato in fase di progettazione, si è pensato di realizzare una classe di messaggi che conterrà le differenti tipologie di contenuti. In questo modo il formato scambiato dai componenti sarà univoco e potrà essere ricostruito mediante funzioni di utilità che consentono la ricostruzione degli oggetti dai byte ricevuti.

La classe generica che verrà trasmessa sarà composta da un descrittore del tipo di dati inviati definito tramite l'enumerazione delle possibili tipologie, e dai dati stessi codificati come stringa JSON. La classe metterà a disposizione i metodi getter e setter per configurare ed ottenere il valore di ogni campo dell'oggetto e consentire al parser JSON di ricostruirlo.

Listato A.1: Message.java

```
public class Message {
   public enum MessageType{ANNOUNCE, INFO_GW, SENSOR_DATA};
   private MessageType type;
   private String content;
   public Message(MessageType type, String content){
      this.type = type;
      this.content = content;
   }

   public Message(){}

   public void setType(MessageType type){
      this.type = type;
   }

   public MessageType getType(){
      return type;
   }

   public String getContent(){
      return content;
   }

   public void setContent(String content){
      this.content = content;
   }
}
```

A.1.1 I contenuti

Il campo content della classe message potrà contenere tre differenti formati di contenuti:

`InfoGatewayContent` definisce il messaggio che i gateway invieranno ai client che riterranno utili. Conterrà gli attributi necessari ai client per connettere la rete WiFi generata dal gateway, e connettersi al broker locale: ID del gateway, distanza dal dispositivo che ha inviato il messaggio di presentazione, i sensori richiesti, i parametri per connettersi alla rete WiFi (SSID, chiave d'accesso e tipo di sicurezza) e l'URI del broker.

Listato A.2: InfoGatewayContent.java

```
public class InfoGatewayContent {

   public static enum Security{OPEN, WEP, WPA};

   private double distance;
   private String ssid;
   private String passphrase;
   private Security security;
   private String brokerURI;
   private List<String> sensorsRequested;
   private String deviceId;

   public InfoGatewayContent() {}

   public InfoGatewayContent(double distance, String ssid, String passphrase, Security security,
         String brokerURI, List<String> sensorRequested, String deviceId){
      this.distance = distance;
      this.ssid = ssid;
      this.passphrase = passphrase;
      this.security = security;
      this.brokerURI = brokerURI;
      this.sensorsRequested = sensorRequested;
      this.deviceId = deviceId;
   }

   public double getDistance() {
      return distance;
   }

   public void setDistance(double distance) {
      this.distance = distance;
   }

   public String getSSID() {
      return ssid;
   }

   public void setSSID(String ssid) {
      this.ssid = ssid;
   }

   public String getPassphrase() {
      return passphrase;
   }

   public void setPassphrase(String passphrase) {
```

```
        this.passphrase = passphrase;
    }

    public Security getSecurity(){
        return security;
    }

    public void setSecurity(Security security){
        this.security = security;
    }

    public String getBrokerURI() {
        return brokerURI;
    }

    public void setBrokerURI(String brokerURI) {
        this.brokerURI = brokerURI;
    }

    public List<String> getSensorsRequested() {
        return sensorsRequested;
    }

    public void setSensorsRequested(List<String> sensorsRequested) {
        this.sensorsRequested = sensorsRequested;
    }

    public String getDeviceId() {
        return deviceId;
    }

    public void setDeviceId(String deviceName) {
        this.deviceId = deviceName;
    }
}
```

`AnnounceContent` è la classe che definisce il contenuto di un messaggio con cui i client annunciano la loro presenza in una specifica località. La classe conterrà tutte le informazioni necessarie al gateway per capire se il dispositivo può risultare utile alle applicazioni in esecuzione o meno, ed eventualmente per contattarlo: l'identificativo, le coordinate geografiche e la lista di sensori messi a disposizione.

Listato A.3: AnnounceContent.java

```
public class AnnounceContent{

    private String devId;
    private List<String> availableSensors;
    private double latitude;
    private double longitude;

    public AnnounceContent(){}

    public AnnounceContent(String id, List<String> sensors, double latitude, double longitude){
        this.devId = id;
        this.availableSensors = sensors;
        this.latitude = latitude;
        this.longitude = longitude;
    }
```

```
    public List<String> getAvailableSensors() {
        return availableSensors;
    }

    public double getLatitude() {
        return latitude;
    }

    public double getLongitude() {
        return longitude;
    }

    public String getDevId() {
        return devId;
    }

    public void setDevId(String devId) {
        this.devId = devId;
    }
}
```

`SensorData` è invece una rappresentazione dei dati dei sensori. Consente di poter inviare le informazioni raccolte in un formato generico e indipendente dal dispositivo. Quando il cliente invierà dati in questo formato sarà trasmessa al cliente una lista di `SensorData`, in cui ogni elemento della lista conterrà nel campo `value` l'ultimo aggiornamento del sensore descritto dal campo `type`, ottenuto nell'istante indicato dal `timestamp`.

Listato A.4: SensorData.java

```
public class SensorData {
    private String type;
    private long timestamp;
    private float[] value = new float[3];

    public SensorData() {}

    public SensorData(String type, long timestamp, float[] value){
        this.type = type;
        this.timestamp = timestamp;
        this.value = value;
    }

    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }

    public long getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }
```

```
        public float[] getValue() {
            return value;
        }

        public void setValue(float[] value) {
            this.value = value;
        }
    }
```

A.1.2 Serializzazione e deserializzazione del payload

MQTT trasmette messaggi in formato binario, risulta quindi necessario che i payload siano espressi in un formato serializzabile, per questo motivo si è deciso di utilizzare una rappresentazione in formato JSON che offre un parsing più veloce ed un overhead minore rispetto ad XML. Le operazioni di serializzazione e deserializzazione dei dati sono realizzate dalla classe `MessageUtils` che, con l'ausilio delle librerie Gson di Google, offre due metodi statici:

`String getJsonString(Object o)` restituisce la rappresentazione dell'oggetto passato come parametro in formato JSON.

`Object fromJson(String json)` ricevuta in input una stringa JSON provvede alla ricostruzione del messaggio e ne restituisce il contenuto deserializzato. Esegue un primo parsing per ottenere l'oggetto di tipo Message, quindi a seconda del tipo di messaggio esegue un secondo parsing per la deserializzazione della stringa JSON che rappresenta il contenuto.

Listato A.5: MessageUtils.java

```
public class MessageUtils {
    protected static Gson gson = new Gson();

    public static Object fromJson(String json){
        Message m = gson.fromJson(json, Message.class);
        switch(m.getType()){
        case ANNOUNCE:
            return (AnnounceContent)gson.fromJson(m.getContent(), AnnounceContent.class);
        case INFO_GW:
            return (InfoGatewayContent) gson.fromJson(m.getContent(), InfoGatewayContent.class);
        case SENSOR_DATA:
            Type sensorDataType = new TypeToken<ArrayList<SensorData>>(){}.getType();
            return gson.fromJson(m.getContent(), sensorDataType);
        }
        return null;
    }

    public static String getJsonString(Object o) {
        return gson.toJson(o);
    }
}
```

A.2 Sviluppare con Kura

Di seguito verrà descritto come preparare l'ambiente di sviluppo per la realizzazione di applicazioni Kura; sarà inoltre mostrato come realizzare dei semplici bundle, come esportarli e come utilizzare le librerie Paho per comunicare attraverso il protocollo MQTT.

A.2.1 Preparazione dell'ambiente di sviluppo

La preparazione dell'ambiente di sviluppo è il primo passo da compiere per realizzare la propria applicazione Kura-based. Il punto di partenza è, ovviamente, l'installazione dell'ultima versione di Eclipse[9] su cui installare il plugin mToolkit per abilitare la connettività remota ad un device Kura-enabled.

Dopo aver installato Eclipse ed averlo dotato di mToolkit, per poter iniziare a sviluppare applicazioni Kura-based è necessario creare un workspace all'interno dell'IDE nel quale importare interamente il Kura Developer Workspace –disponibile nella sezione Downloads del sito del progetto: https://eclipse.org/kura/–.

Al termine dell'importazione saranno presenti nel workspace quattro progetti:

`org.eclipse.kura.api` il core delle API di Kura

`org.eclipse.kura.demo.heater` un progetto di esempio che può essere utilizzato come punto di partenza per la creazione di un primo bundle

`org.eclipse.kura.emulator` l'emulatore che consente l'esecuzione di Kura all'interno di Eclipse in ambienti Mac e Linux

`target-definition` un insieme di bundle necessari per soddisfare le dipendenze delle API e di Kura

I progetti importati presenteranno numerosi errori che verranno corretti in maniera automatica aprendo il file `kura-equinox_3.8.1.target` nel progetto `target-definition` e cliccando su Set as Target Platform. In questo modo si esegue un reset della piattaforma di destinazione, viene eseguito il rebuild dei progetti e gli errori vengono eliminati, consentendo allo sviluppatore di poter iniziare a realizzare le proprie applicazioni Kura-based.

A.2.2 Creare un bundle

Il miglior modo per apprendere come realizzare un bundle Kura-based è quello di "sporcarsi le mani" con un banale esempio: la realizzazione di un bundle Hello World che utilizza il logger di Kura.

Per iniziare a realizzare un bundle Kura-based, come in OSGi, è necessario definire un nuovo progetto per la realizzazione di plug-in disattivando la creazione automatica della classe `Activator`.

Terminato il processo di creazione si aprirà il manifest, nel quale dovrà essere specificato come verranno risolte le dipendenze. Aprendo il tab Dependencies, nel box Automated Management of Dependencies dovrà essere inserito come package da importare quello relativo ai servizi OSGi (`org.eclipse.osgi.services`) e, nel caso in esempio, al Simple Logger Facade for Java[10] utilizzato da Kura (`slf4j.api`).

[9] Nella documentazione presente su Github la community suggerisce la versione per sviluppatori Java EE.
[10] http://www.slf4j.org/apidocs/org/slf4j/Logger.html

Dopo aver salvato le modifiche apportate bisognerà creare la classe `HelloKura` definita nel listato che segue in cui il metodo `activate()` rappresenta l'entry point relativo all'attivazione del bundle, mentre il `deactivate()` è quello relativo alla disattivazione:

Listato A.6: HelloKura.java

```
public class HelloKura {
   private static final Logger s_logger = LoggerFactory.getLogger(HelloKura.class);
   private static final String APP_ID = "org.eclipse.kura.example.hello_kura";

   protected void activate(ComponentContext componentContext) {
      s_logger.info("Bundle " + APP_ID + " has started!");
      s_logger.info("\t" + APP_ID + ": Hello, Kura! :D");
   }

   protected void deactivate(ComponentContext componentContext) {
      s_logger.info("Bundle " + APP_ID + " has stopped!");
   }
}
```

Una volta terminata la realizzazione della classe `HelloKura` sarà necessario riaprire il manifest, e sempre nel box Automated Management of Dependencies, far analizzare il codice in modo da far aggiungere automaticamente le dipendenze.

Dopo aver realizzato la classe, perché questa sia eseguibile da Kura, bisognerà generare un component descriptor attraverso il wizard di Eclipse (New/Plug-in development/Component definition) assegnando al file `component.xml` che verrà generato il path di default seguito `/OSGI-INF`, e selezionando la classe `HelloKura` definita in precedenza. Terminato il processo si aprirà la view di modifica del descrittore in cui sarà sufficiente definire i metodi di attivazione e disattivazione del bundle, nel nostro caso `activate()` e `deactivate()`.

A.2.3 Bundle configurabile

Kura consente di configurare quei bundle progettati in modo tale da consentire questa operazione attraverso la sua interfaccia web.

Per fare ciò è necessario che oltre ai metodi visti nell'esempio precedente siano definiti all'interno del component descriptor anche il metodo che si occuperà di aggiornare le proprietà del bundle, ed una proprietà aggiuntiva: `service.pid`. Quest'ultima avrà come valore il nome della classe comprensiva del package.

Per specificare al framework che quello che si sta realizzando è un bundle configurabile sarà necessario che la classe che lo realizza implementi l'interfaccia `ConfigurableComponent`. Questa interfaccia non dichiara alcun metodo, ma funge da marker per tutti quei componenti che devono poter essere configurati a runtime sfruttando il servizio `ConfigurationService`. Quest'ultimo presuppone che le informazioni riguardanti la configurazione siano memorizzate in un file XML con lo stesso nome della classe che implementa il componente all'interno della directory `OSGI-INF/metatype`.

Per fare maggiore chiarezza, si andrà a realizzare un nuovo Hello World, questa volta con messaggio configurabile.

Come in precedenza, sarà necessario creare un nuovo Plug-in Project, aggiungendo al gestore automatico delle dipendenze, questa volta, anche le API di Kura (`org.eclipse.kura.api`).

Il seguente listato mostra la classe `HelloWorldConfigurable`:

Listato A.7: HelloKuraConfigurable.java

```
public class HelloKuraConfigurable implements ConfigurableComponent {
   private static final Logger s_logger = LoggerFactory.getLogger(HelloKuraConfigurable.class);
   private static final String APP_ID = "it.antenucci.kura.example.hello_kura_configurable.
        HelloKuraConfigurable";
   private Map<String, Object> properties;
   //Configurazione del nome delle proprietÃ
   private static final String PROP_MESSAGE_NAME = "message";
   private static final String PROP_BY_NAME = "by";

   protected void activate(ComponentContext componentContext) {
      s_logger.info("Bundle " + APP_ID + " has started!");
   }

   protected void activate(ComponentContext componentContext, Map<String, Object> properties) {
      s_logger.info("Bundle " + APP_ID + " has started with config!");
      updated(properties);
   }

   protected void deactivate(ComponentContext componentContext) {
      s_logger.info("Bundle " + APP_ID + " has stopped!");
   }

   public void updated(Map<String, Object> properties) {
      this.properties = properties;
      if(properties != null && !properties.isEmpty()) {
         Iterator<Entry<String, Object>> it = properties.entrySet().iterator();
         while (it.hasNext()) {
            Entry<String, Object> entry = it.next();
            s_logger.info("New property - " + entry.getKey() + " = " +
                  entry.getValue() + " of type " + entry.getValue().getClass().toString());
         }
      }
      printMessage();
   }

   private void printMessage() {
      String message = "\t" + APP_ID + ": " + properties.get(PROP_MESSAGE_NAME);
      if(properties!=null){
         if(properties.containsKey(PROP_MESSAGE_NAME))
            message += " by " + properties.get(PROP_BY_NAME);
         else{
            s_logger.info("Bundle " + APP_ID + ": property "+PROP_BY_NAME+" not setted");
            message += " by System";
         }
      }
      s_logger.info(message);
   }
}
```

Dopo aver terminato l'implementazione ed aver fatto aggiungere le dipendenze al manifest come nel precedente esempio, si può procedere alla creazione del component descriptor in cui sarà necessario

- definire i metodi activate, deactivate e modified (rispettivamente `activate`, `deactivate` e `updated`)
- aggiungere la proprietà di tipo stringa `service.pid` che avrà valore `<package>.HelloKuraConfigurable`
- impostare la configuration policy a require
- spuntare le checkbox relative ad abilitazione del bundle allo start e all'autoattivazione
- impostare nel tab Services la classe che implementa il bundle come provider

Dopo aver salvato le modifiche apportate al component descriptor si andrà a creare il file XML per la configurazione di default nella directory `OSGI-INF/metadata` definito come segue:

Listato A.8: HelloKuraConfigurable.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaData xmlns="http://www.osgi.org/xmlns/metatype/v1.2.0" localization="en_us">
    <OCD id="it.antenucci.kura.example.hello_kura_configurable.HelloKuraConfigurable"
        name="HelloKuraConfigurable"
        description="This is a metatype file for a simple hello world configurable example">

        <AD id="message"
            name="message"
            type="String"
            cardinality="0"
            required="true"
            default="Hello Kura!"
            description="Message configuration parameter"/>

        <AD id="by"
            name="by"
            type="String"
            cardinality="0"
            required="false"
            description="Who prints the message"/>

    </OCD>

    <Designate pid="it.antenucci.kura.example.hello_kura_configurable.HelloKuraConfigurable">
        <Object ocdref="it.antenucci.kura.example.hello_kura_configurable.HelloKuraConfigurable"/>
    </Designate>
</MetaData>
```

Ogni elemento `AD` rappresenta una entry delle proprietà, nel nostro caso la prima –obbligatoria– imposterà il messaggio che il bundle stamperà nel log, mentre la seconda consentirà di definire chi ha scritto il messaggio. Nel component descriptor si può infine notare il riferimento alla proprietà `pid`, definita all'interno del manifest.

L'ultima modifica da apportare affinché il bundle venga generato in modo corretto consiste nel selezionare all'interno del tab Build del manifest la directory metatype all'interno di `OSGI-INF`.

Una volta installato il bundle, nell'interfaccia web di Kura sarà possibile modificare la configurazione del servizio a runtime.

A.2.4 Esportazione di un bundle

Terminata la realizzazione di un bundle, per testarlo, installarlo o distribuirlo è necessario esportarlo. Questa operazione può essere eseguita in due differenti modalità: esportando il bundle OSGi, oppure creando un package di deployment.

A.2.4.1 Esportare un bundle OSGi

Un bundle OSGi non è altro che un plug-in OSGi stand-alone, cioè un archivio `.jar` contenente il codice Java, le risorse ed il manifest del bundle precedentemente realizzato. Il file generato, per poter essere eseguito, dovrà successivamente essere installato in un framework OSGi.

Il processo di esportazione e creazione del bundle è del tutto automatizzato: è sufficiente che lo sviluppatore selezioni il progetto del bundle e nel menù Export selezioni Deployable plug-ins and fragments dal sotto-menù Plug-in Development. Una volta terminato il processo, nella directory specificata, sarà presente il file `.jar` da installare nel framework.

A.2.4.2 Creare un Deployment Package

Un Deployment Package è un insieme di risorse raggruppate in un solo file che può essere distribuito in un framework OSGi attraverso il servizio Deployment Admin. Il package generato conterrà uno o più bundle, oggetti di configurazione, etc.

Come prima cosa sarà necessario creare un Deployment Package Project file (File/New/OSGi/Deployment Package Definition). A questo punto il modo più semplice di realizzare un deployment package è quello di aggiungere nel tab Bundles del file `.dpp` il bundle di cui si vuole eseguire il deployment aggiungendo il riferimento al file `.project` presente nella root directory del progetto, o in alternativa il file `.jar` esportato.

Questa seconda modalità può essere utile nel caso in cui si vogliano caricare nel framework delle librerie esterne in modo da renderle utilizzabili anche da eventuali altri bundle che verranno installati in seguito. Dopo aver salvato, avviando il Quick Build si genererà, nella stessa directory del file `.dpp`, un nuovo file avente lo stesso nome ma con estensione `.dp`. Questo sarà il deployment package vero e proprio e che verrà utilizzato per installare il bundle anche su sistemi di destinazione remoti.

A.3 Utilizzo delle librerie Paho

Le librerie Paho realizzano dei client MQTT in diversi linguaggi di programmazione, tra cui Java. Un client è composto principalmente da due classi: una realizzerà il client vero e proprio, mentre l'altra provvederà ad implementare l'interfaccia `MqttCallback`, che si occuperà della gestione degli eventi.

Le classi che realizzano il client mette a disposizione tutti i metodi necessari per interagire con il broker:

`connect` invia la richiesta di connessione al broker

`publish` invia un messaggio al broker su un determinato topic

`subscribe` richiede al broker la sottoscrizione al topic

`unsubscribe` provvede alla cancellazione della sottoscrizione

Esse possono operare in modo sincrono (`MqttClient`) o asincrono (`MqttAsyncClient`).

L'invocazione del costruttore richiede il passaggio di (almeno) due parametri in formato stringa in entrambe le sue versioni: l'URI del broker e l'identificativo che si vorrà assegnare al client.

Dopo la creazione dell'oggetto sarà possibile stabilire la connessione al broker utilizzando il metodo `connect` a cui è possibile passare come parametro un oggetto di tipo `MqttConnectOptions`, che andrà a definire quali saranno le opzioni di connessione, come ad esempio il keep alive interval, il connection timeout, o il flag di clean session.

Stabilita la connessione al broker si potrà quindi iniziare ad inviare messaggi e sottoscrivere topic.

Attraverso l'implementazione dell'interfaccia `MqttCallback`, che dovrà essere associata al client prima di stabilire la connessione, si realizzerà il comportamento che avrà il client in seguito alla ricezione di un messaggio, alla conferma della pubblicazione (in caso di QoS maggiore di 0), o in caso di perdita della connessione.

Nei seguenti listati è mostrato come utilizzare le librerie Paho per creare un client MQTT che invia un messaggio ed uno che lo riceve:

Listato A.9: MqttPublisher.java

```
public class MqttPublishSample {
    public static void main(String[] args) {
        String topic        = "MQTT Examples";
        String content      = "Message from MqttPublishSample";
        int qos             = 0;
        String broker       = "tcp://iot.eclipse.org:1883";
        String clientId     = "JavaSample";
        try {
            MqttClient sampleClient = new MqttClient(broker, clientId);
            MqttConnectOptions connOpts = new MqttConnectOptions();
            connOpts.setCleanSession(true);
            System.out.println("Connecting to broker: "+broker);
            sampleClient.connect(connOpts);
            System.out.println("Connected");
            System.out.println("Publishing message: "+content);
            MqttMessage message = new MqttMessage(content.getBytes());
            message.setQos(qos);
            sampleClient.publish(topic, message);
            System.out.println("Message published");
            sampleClient.disconnect();
            System.out.println("Disconnected");
            System.exit(0);
        }
        catch(MqttException me) {
            System.out.println("reason "+me.getReasonCode());
            System.out.println("msg "+me.getMessage());
            System.out.println("loc "+me.getLocalizedMessage());
            System.out.println("cause "+me.getCause());
            System.out.println("excep "+me);
```

```
            me.printStackTrace();
        }
    }
}
```

Listato A.10: MqttSubscriber.java

```
    public static void main(String[] args) {
        String topic        = "MQTT Examples";
        String content      = "Message from MqttPublishSample";
        int qos             = 0;
        String broker       = "tcp://iot.eclipse.org:1883";
        String clientId     = "JavaSample";
        try {
            MqttClient sampleClient = new MqttClient(broker, clientId);
            sampleClient.setCallback(new MqttCallback(){
                @Override
                public void connectionLost(Throwable arg0) {
                    System.out.println("Broker disconnected");
                }
                @Override
                public void deliveryComplete(IMqttDeliveryToken arg0) {
                    System.out.println("Message delivered to the broker");
                }
                @Override
                public void messageArrived(String topic, final MqttMessage message) throws Exception {
                    new Runnable(){
                        @Override
                        public void run(){
                            System.out.println("New message received: " new String(message));
                        }
                    }.run();
                }
            });
            MqttConnectOptions connOpts = new MqttConnectOptions();
            connOpts.setCleanSession(true);
            System.out.println("Connecting to broker: "+broker);
            sampleClient.connect(connOpts);
            System.out.println("Connected");
            System.out.println("Subscribing topic: "+topic);
            sampleClient.subscribe(topic, qos);
            while(true);
            sampleClient.disconnect();
            System.out.println("Disconnected");
            System.exit(0);
        }
        catch(MqttException me) {
            System.out.println("reason "+me.getReasonCode());
            System.out.println("msg "+me.getMessage());
            System.out.println("loc "+me.getLocalizedMessage());
            System.out.println("cause "+me.getCause());
            System.out.println("excep "+me);
            me.printStackTrace();
        }
    }
}
```

A.3.1 Le classi del bundle

A.3.1.1 Il package `kura.utils`

Di seguito sono mostrate le classi che fanno parte del package `kura.utils` non mostrate nel capitolo di implementazione.

Listato A.11: Constants.java

```
public class Constants {
   /*------------------*
    * PROPERTIES NAMES *
    *------------------*/
   public static final String DEVICE_ID = "device.id";
   public static final String PLACE_ID = "place.id";
   public static final String LATITUDE = "location.latitude";
   public static final String LONGITUDE = "location.longitude";
   public static final String CLOUD_BROKER_PROTOCOL = "broker.cloud.protocol";
   public static final String CLOUD_BROKER_URI = "broker.cloud.uri";
   public static final String CLOUD_BROKER_PORT = "broker.cloud.port";
   public static final String LOCAL_BROKER_PROTOCOL = "broker.local.protocol";
   public static final String LOCAL_BROKER_URI = "broker.local.uri";
   public static final String LOCAL_BROKER_PORT = "broker.local.port";
   public static final String WIFI_SSID = "wifi.ssid";
   public static final String WIFI_PASSPHRASE = "wifi.passphrase";
   public static final String WIFI_SECURITY = "wifi.security";
   public static final String SENSOR_PREFIX = "sensor";
   public static final String ACCELEROMETER = SENSOR_PREFIX + ".accelerometer";
   public static final String AMBIENT_TEMPERATURE = SENSOR_PREFIX + ".ambient.temperature";
   public static final String GAME_ROTATION_VECTOR = SENSOR_PREFIX + ".game.rotation.vector";
   public static final String GEOMAGNETIC_ROTATION_VECTOR = SENSOR_PREFIX + ".geomagnetic.rotation.
        vector";
   public static final String GRAVITY = SENSOR_PREFIX + ".gravity";
   public static final String GYROSCOPE = SENSOR_PREFIX + ".gyroscope";
   public static final String GYROSCOPE_UNCALIBRATED = SENSOR_PREFIX + ".gyroscope.uncalibrated";
   public static final String HEART_RATE = SENSOR_PREFIX + ".heart.rate";
   public static final String LIGHT = SENSOR_PREFIX + ".light";
   public static final String LINEAR_ACCELERATION = SENSOR_PREFIX + ".linear.acceleration";
   public static final String MAGNETIC_FIELD = SENSOR_PREFIX + ".magnetic.field";
   public static final String PRESSURE = SENSOR_PREFIX + ".pressure";
   public static final String PROXIMITY = SENSOR_PREFIX + ".proximity";
   public static final String RELATIVE_HUMIDITY = SENSOR_PREFIX + ".relative.humidity";
   public static final String ROTATION_VECTOR = SENSOR_PREFIX + ".rotation.vector";
   public static final String SIGNIFICANT_MOTION = SENSOR_PREFIX + ".significant.motion";
   public static final String STEP_COUNTER = SENSOR_PREFIX + ".step.counter";
   public static final String STEP_DETECTION = SENSOR_PREFIX + ".step.detection";

// /*--------*
//  * TOPICS *
//  *--------*/
   public static final String SENSOR_DATA_TOPIC = "/sensor/#";
   public static final String WILL_TOPIC = "/notify";

   /*----------------------*
    * SENSOR NAME FROM IDs *
    *----------------------*/
   public static String getSensorName(String id){
```

```
        switch(id){
            case ACCELEROMETER: return "Accelerometer";
            case AMBIENT_TEMPERATURE: return "Ambient Temperature";
            case GAME_ROTATION_VECTOR: return "Game Rotation Vector";
            case GEOMAGNETIC_ROTATION_VECTOR: return "Geomagnetic Rotation Vector";
            case GRAVITY: return "Gravity";
            case GYROSCOPE: return "Gyroscope";
            case GYROSCOPE_UNCALIBRATED: return "Gyroscope Uncalibrated";
            case HEART_RATE: return "Heart Rate";
            case LIGHT: return "Light";
            case LINEAR_ACCELERATION: return "Linear Acceleration";
            case MAGNETIC_FIELD: return "Magnetic Field";
            case PRESSURE: return "Pressure";
            case PROXIMITY: return "Proximity";
            case RELATIVE_HUMIDITY: return "Relative Humidity";
            case ROTATION_VECTOR: return "Rotation Vector";
            case SIGNIFICANT_MOTION: return "Significant Motion";
            case STEP_COUNTER: return "Step Counter";
            case STEP_DETECTION: return "Step Detection";
        }
        return null;
    }
    /*----------------------*
     * SENSOR ID FROM NAMEs *
     *----------------------*/
    public static String getSensorId(String name){
        switch(name){
            case "Accelerometer": return ACCELEROMETER;
            case "Ambient Temperature": return AMBIENT_TEMPERATURE;
            case "Game Rotation Vector": return GAME_ROTATION_VECTOR;
            case "Geomagnetic Rotation Vector": return GEOMAGNETIC_ROTATION_VECTOR;
            case "Gravity": return GRAVITY;
            case "Gyroscope": return GYROSCOPE;
            case "Gyroscope Uncalibrated": return GYROSCOPE_UNCALIBRATED;
            case "Heart Rate": return HEART_RATE;
            case "Light": return LIGHT;
            case "Linear Acceleration": return LINEAR_ACCELERATION;
            case "Magnetic Field": return MAGNETIC_FIELD;
            case "Pressure": return PRESSURE;
            case "Proximity": return PROXIMITY;
            case "Relative Humidity": return RELATIVE_HUMIDITY;
            case "Rotation Vector": return ROTATION_VECTOR;
            case "Significant Motion": return SIGNIFICANT_MOTION;
            case "Step Counter": return STEP_COUNTER;
            case "Step Detection": return STEP_DETECTION;
        }
        return null;
    }
}
```

Listato A.12: MqttDiscoveryProperties.java

```
public class MqttDiscoveryProperties {

   private String cloudTopic;
   private InfoGatewayContent content;
   private ArrayList<String> sensors;
   private double latGW;
   private double lonGW;
   private String cloudBroker = "";
   private String devId;
   private String localBroker = "";
   private String err = "";

   public String getCloudTopic() {
      return cloudTopic;
   }
   public void setCloudTopic(String cloudTopic) {
      this.cloudTopic = cloudTopic;
   }
   public InfoGatewayContent getContent() {
      return content;
   }
   public void setContent(InfoGatewayContent content) {
      this.content = content;
   }
   public ArrayList<String> getSensors() {
      return sensors;
   }
   public void setSensors(ArrayList<String> sensors) {
      this.sensors = sensors;
   }
   public double getLatGW() {
      return latGW;
   }
   public void setLatGW(double latGW) {
      this.latGW = latGW;
   }
   public double getLonGW() {
      return lonGW;
   }
   public void setLonGW(double lonGW) {
      this.lonGW = lonGW;
   }
   public String getCloudBroker() {
      return cloudBroker;
   }
   public void setCloudBroker(String cloudBroker) {
      this.cloudBroker = cloudBroker;
   }
   public String getDevId() {
      return devId;
   }
   public void setDevId(String devId) {
      this.devId = devId;
   }
```

```
    public String getLocalBroker() {
        return localBroker;
    }
    public void setLocalBroker(String localBroker) {
        this.localBroker = localBroker;
    }
    public String getErr() {
        return err;
    }
    public void setErr(String err) {
        this.err = err;
    }
}
```

Listato A.13: Utils.java

```
public class Utils {

    public static double calculateDistance(double lat1, double lng1, double lat2, double lng2) {
        double earthRadius = 6371000; //meters
        double dLat = Math.toRadians(lat2-lat1);
        double dLng = Math.toRadians(lng2-lng1);
        double a = Math.sin(dLat/2) * Math.sin(dLat/2) +
                Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
                Math.sin(dLng/2) * Math.sin(dLng/2);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
        float dist = (float) (earthRadius * c);
        return dist;
    }

    private static String validateURI(String string) {
        try {
            InetAddress.getByName(string);
            return string;
        } catch (UnknownHostException e) {
            return null;
        }
    }

    public static MqttDiscoveryProperties validateProperties(Map<String, Object> prop) {
        String err = "";
        MqttDiscoveryProperties properties = new MqttDiscoveryProperties();
        InfoGatewayContent content = new InfoGatewayContent();
        String ssid = (String)prop.get(Constants.WIFI_SSID);
        String security = (String)prop.get(Constants.WIFI_SECURITY);
        String pass = (String)prop.get(Constants.WIFI_PASSPHRASE);
        if (ssid == null)
            err += "SSID cannot be empty.\n";
        else
            content.setSSID(ssid);
        if(security == null || !(security.equalsIgnoreCase("WPA") || security.equalsIgnoreCase("WEP") ||
                security.equalsIgnoreCase("NONE")))
            err += "Wrong security. You can choose only WPA, WEP or NONE.\n";
        else{
            switch(security){
                case "WPA": content.setSecurity(Security.WPA); break;
                case "WEP": content.setSecurity(Security.WEP); break;
                case "NONE": content.setSecurity(Security.OPEN); break;
```

```
            }
            if ( !security.equals("NONE") && pass == null)
                err += "If security is not NONE, passphrase cannot be empty.\n";
            else
                content.setPassphrase(pass);
        }
        ArrayList<String> sensors = new ArrayList<String>();
        Iterator<Entry<String,Object>> it = prop.entrySet().iterator();
        while (it.hasNext()){
            Entry<String,Object> e = it.next();
            if(e.getKey().contains(Constants.SENSOR_PREFIX) && (boolean)e.getValue())
                sensors.add(Constants.getSensorName(e.getKey()));
        }
        if(sensors.isEmpty()){
            err += "Choose at least one sensor.\n";
        }
        content.setDeviceId((String)prop.get(Constants.DEVICE_ID));
        if (content.getDeviceId() == null)
            err += "Device ID cannot be empty.\n";
        String cloudBroker = Utils.validateURI((String)prop.get(Constants.CLOUD_BROKER_URI));
        if(cloudBroker == null)
            err += "Cloud broker URI is not valid.\n";
        else
            properties.setCloudBroker(prop.get(Constants.CLOUD_BROKER_PROTOCOL)+"://"+cloudBroker+":"+
                 prop.get(Constants.CLOUD_BROKER_PORT));
        String localBroker = Utils.validateURI((String)prop.get(Constants.LOCAL_BROKER_URI));
        if(localBroker == null)
            err += "Local broker URI is not valid.\n";
        else
            content.setBrokerURI(prop.get(Constants.LOCAL_BROKER_PROTOCOL)+"://"+localBroker+":"+prop.get
                 (Constants.LOCAL_BROKER_PORT));
        if(prop.get(Constants.LATITUDE) != null)
            properties.setLatGW((double) prop.get(Constants.LATITUDE));
        else
            properties.setLatGW(0);
        if(prop.get(Constants.LONGITUDE) != null)
            properties.setLonGW((double) prop.get(Constants.LONGITUDE));
        else
            properties.setLonGW(0);
        properties.setCloudTopic((String)prop.get(Constants.PLACE_ID));
        if (properties.getCloudTopic() == null)
            err += "Geographic topic cannot be null.\n";
        properties.setContent(content);
        properties.setDevId(content.getDeviceId());
        properties.setLocalBroker(content.getBrokerURI());
        properties.setSensors(sensors);
        if(!err.contains("Device ID"))
            properties.setErr(err);
        else
            properties.setErr("Configure the service");
        return properties;
    }
}
```

A.3.1.2 La classe MqttDiscoveryService

Listato A.14: MqttDiscoveryService.java

```
public class MqttDiscoveryService implements ConfigurableComponent{

   private final Logger LOG = LoggerFactory.getLogger(MqttDiscoveryService.class);
   private final String APP_ID = "MqttDiscoveryService";
   private boolean running = false;
   private MqttAsyncClient local, cloud;
   private ConfigurationService configurationService;
   private String cloudTopicSubscribed = "";
   private MqttConnectOptions connOpt;
   private MqttDiscoveryProperties properties;

   public enum BrokerType{LOCAL, CLOUD};

   protected void activate(ComponentContext componentContext, Map<String, Object> properties){
      LOG.info("Bundle " + APP_ID + " has started with configuration!");
      this.properties = Utils.validateProperties(properties);
      if (this.properties.getErr() == "")
         startService();
      else{
         LOG.error("Please, configure the service properly.");
      }
   }

   protected void deactivate(ComponentContext componentContext){
      stopService();
      LOG.info("Bundle " + APP_ID + " has stopped!");
   }

   protected void updated(Map<String, Object> prop){
      LOG.info("Configuration of " + APP_ID + " updated!");
      properties = Utils.validateProperties(prop);
      if(properties.getErr() == ""){
         startService();
      }
      else{
         if(!properties.getErr().equals("Configure the service")){
            try {
               configurationService.rollback();
            } catch (KuraException e) {
               e.printStackTrace();
            }
            LOG.error(properties.getErr());
            if(running)
               LOG.warn("Service still working with previous configuration.");
            else
               LOG.error("Configure the service properly.");
         }
         LOG.error("Configure the service properly.");
      }
   }

   public void startService() {
      if(running){
         stopService();
```

```
    }
    connOpt = new MqttConnectOptions();
    connOpt.setCleanSession(true);
    connOpt.setConnectionTimeout(15); //15 seconds
    connOpt.setKeepAliveInterval(30); //30 seconds
    while(!running){
       try {
          cloud = new MqttAsyncClient(properties.getCloudBroker(), properties.getDevId(), null);
          cloud.setCallback(new MqttCloudCallback(this));
          connectBroker(BrokerType.CLOUD);
          local = new MqttAsyncClient(properties.getLocalBroker(), properties.getDevId(), null);
          local.setCallback(new MqttLocalCallback(this));
          connectBroker(BrokerType.LOCAL);
          if(cloud != null && local != null && cloud.isConnected() && local.isConnected())
             running = true;
       } catch (MqttException e) {
          LOG.warn("Something goes wrong connecting cloud broker. Check if the URI and retry.");
       }
       if(!running){
          LOG.warn("Something goes wrong. If the problem persist retry your configuration and your
               connection.\nRetry in 30 seconds...");
          try {
             Thread.sleep(30000);
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
       }
    }
}

private void stopService() {
    running = false;
    try {
       if (cloud != null){
          if ( cloud.isConnected()){
             cloud.unsubscribe(cloudTopicSubscribed);
             cloud.disconnect();
             cloud.close();
          }
          cloud = null;
       }
       if (local != null){
          if (local != null && local.isConnected()){
             local.unsubscribe(Constants.SENSOR_DATA_TOPIC);
             local.unsubscribe(Constants.WILL_TOPIC);
             local.disconnect();
             local.close();
          }
          local = null;
       }
    } catch (MqttException e) { e.printStackTrace(); }
}

public void sendInfoGateway(AnnounceContent announce) {
    List<String> requestedSensors = new ArrayList<String>();
    for (String s: announce.getAvailableSensors())
       if (properties.getSensors().contains(s))
          requestedSensors.add(s);
    if(requestedSensors.isEmpty()){
```

```
            LOG.info("Device " + announce.getDevId() + " not offers useful sensors.");
            return;
        }
        LOG.info("Sending InfoGW message to " + announce.getDevId());
        properties.getContent().setSensorsRequested(requestedSensors);
        double latD = announce.getLatitude();
        double lonD = announce.getLongitude();
        properties.getContent().setDistance(Utils.calculateDistance(properties.getLatGW(), properties.
             getLonGW(), latD, lonD));
        Message payload = new Message(MessageType.INFO_GW, MessageUtils.getJsonString(properties.
             getContent()));
        try {
            MqttMessage msg = new MqttMessage();
            msg.setPayload(MessageUtils.getJsonString(payload).getBytes());
            msg.setQos(0);
            msg.setRetained(false);
            cloud.publish(cloudTopicSubscribed+"/"+announce.getDevId(), msg);
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }

    public void connectBroker(BrokerType type) {
        IMqttToken t;
        IMqttAsyncClient cli;
        String msg = "Connecting to ";
        if(type==BrokerType.LOCAL){
            cli = local;
            msg += "local broker";
        }
        else{
            cli = cloud;
            msg += "cloud broker";
        }
        for(int i=0;i<3;i++){
            try{
                LOG.info(msg +", attempt " + (i+1));
                t = cli.connect(connOpt);
                t.waitForCompletion();
                break;
            }
            catch(MqttException e){
                if(i<3){
                    LOG.error("Attempt " + (i+1) + " failed. Retry in 10 seconds");
                    try {
                        Thread.sleep(10000);
                    } catch (InterruptedException e1) {
                        e1.printStackTrace();
                    }
                }
            }
        }
        if(cli == local){
            if(local.isConnected()){
                try{
                    LOG.info("Local broker connected, subscribing to " + Constants.SENSOR_DATA_TOPIC + "
                         and " + Constants.WILL_TOPIC + "...");
                    t = local.subscribe(Constants.SENSOR_DATA_TOPIC, 0);
                    t.waitForCompletion();
```

```
                t = local.subscribe(Constants.WILL_TOPIC, 0);
                t.waitForCompletion();
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
        else{
            LOG.error("Unable to connect local broker.");
            if(running && cloud.isConnected()){
                LOG.error("Retry in 30 seconds.");
                try {
                    Thread.sleep(30000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                connectBroker(BrokerType.LOCAL);
            }
            else
                startService();
        }
    }
    else{
        if(cloud.isConnected()){
            try{
                cloudTopicSubscribed = properties.getCloudTopic();
                LOG.info("Cloud broker connected, subscribing to " + cloudTopicSubscribed + "...");
                t = cloud.subscribe(cloudTopicSubscribed, 0);
                t.waitForCompletion();
                if (cloud.isConnected())
                    return;
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
        else{
            LOG.error("Unable to connect cloud broker.");
            if(running && local.isConnected()){
                LOG.error("Retry in 30 seconds.");
                try {
                    Thread.sleep(30000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                connectBroker(BrokerType.CLOUD);
            }
            else
                startService();
        }
    }
  }
}
```

A.4 I meccanismi Android

Di seguito sono mostrati i codici sorgenti delle classi e dei file XML non mostrati nel capitolo di implementazione. I listati sono suddivisi in differenti sezioni, una per ogni meccanismo a cui se ne aggiungerà una per l'applicazione finale.

A.4.1 Comunicazione MQTT

Listato A.15: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.antenucci.test.android.mqtt"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="9"
        android:targetSdkVersion="21" />

    <!-- imei permission -->
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <!-- mqtt client permission -->
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@+style/AppTheme" >
        <activity
            android:name=".MQTTActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name="org.eclipse.paho.android.service.MqttService"></service>

        <meta-data
        android:name="it.antenucci.test.android.mqtt"
        android:value="@string/app_id" />
    </application>

</manifest>
```

Listato A.16: activity_mqtt.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingBottom="@dimen/activity_vertical_margin"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
tools:context="it.antenucci.test.android.mqtt.MQTTActivity" >

<Button
    android:id="@+id/connect"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/broker_addr"
    android:layout_centerHorizontal="true"
    android:text="Connect" />

<TextView
    android:id="@+id/lbl_topic"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/connect"
    android:layout_centerHorizontal="true"
    android:text="Topic" />

<EditText
    android:id="@+id/txt_topic"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/txt_msg"
    android:layout_alignRight="@+id/subscribe"
    android:layout_below="@+id/lbl_topic"
    android:ems="10"
    android:text="/topic/test" >

    <requestFocus />
</EditText>

<TextView
    android:id="@+id/lbl_msg"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/txt_topic"
    android:layout_centerHorizontal="true"
    android:text="Message" />

<EditText
    android:id="@+id/txt_msg"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/publish"
    android:layout_alignRight="@+id/subscribe"
    android:layout_below="@+id/lbl_msg"
    android:ems="10"
    android:text="test message" />

<EditText
    android:id="@+id/broker_addr"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="15dp"
        android:text="tcp://iot.eclipse.org:1883" />

    <Button
        android:id="@+id/unsubscribe"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/res"
        android:layout_below="@+id/subscribe"
        android:layout_toRightOf="@+id/connect"
        android:text="Unsubscribe" />

    <TextView
        android:id="@+id/res"
        android:layout_width="wrap_content"
        android:layout_height="80dp"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/unsubscribe"
        android:text="Result"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <EditText
        android:id="@+id/nmsg"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/publish"
        android:layout_alignBottom="@+id/publish"
        android:layout_toLeftOf="@+id/subscribe"
        android:layout_toRightOf="@+id/publish"
        android:ems="10"
        android:inputType="number" />

    <Button
        android:id="@+id/publish"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/unsubscribe"
        android:layout_alignParentLeft="true"
        android:layout_marginBottom="20dp"
        android:text="Publish" />

    <Button
        android:id="@+id/subscribe"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/unsubscribe"
        android:layout_alignRight="@+id/res"
        android:layout_below="@+id/txt_msg"
        android:text="Subscribe" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/res"
        android:layout_centerHorizontal="true"
        android:layout_marginBottom="120dp" >
```

```
        <TextView
            android:id="@+id/textViewMin"
            android:layout_width="86dp"
            android:layout_height="wrap_content"
            android:text="MIN" />

        <TextView
            android:id="@+id/textViewAvg"
            android:layout_width="82dp"
            android:layout_height="wrap_content"
            android:text="AVG" />

        <TextView
            android:id="@+id/textViewMax"
            android:layout_width="78dp"
            android:layout_height="wrap_content"
            android:text="MAX" />
    </LinearLayout>

</RelativeLayout>
```

Listato A.17: MqttActionListener.java

```
public class MQTTActionListener implements IMqttActionListener {

   public enum Action{ CONNECT, PUBLISH, SUBSCRIBE, DISCONNECT, UNSUBSCRIBE};
   MQTTActivity activity;
   TextView res;
   Action action;
   Long start;

   public MQTTActionListener(MQTTActivity activity, Action act) {
      start = System.currentTimeMillis();
      this.activity = activity;
      res = (TextView)activity.findViewById(R.id.res);
      action = act;
   }

   @Override
   public void onFailure(IMqttToken arg0, Throwable arg1) {
      long performingTime = System.currentTimeMillis() - start;
      res.setText("");
      res.setTextColor(Color.parseColor("#FF0000"));
      String result = "";
      switch (action){
         case CONNECT: result = "Connection failed: " + arg1.getMessage();
            break;
         case DISCONNECT: result = "Disconnection failed: " + arg1.getMessage();
            break;
         case SUBSCRIBE: result = "Subscription to failed: " + arg1.getMessage();
            break;
         case PUBLISH: result = "Publish failed: " + arg1.getMessage();
            break;
         case UNSUBSCRIBE: result = "Unsubscription to failed: " + arg1.getMessage();
            break;
      }
      Log.i("FAILURE", result.split(" ")[0]);
```

```
            res.setText(result);
            if(action != Action.PUBLISH){
                Measurements.addTime(performingTime);
            }
        }

        @Override
        public void onSuccess(IMqttToken arg0) {
            long performingTime = System.currentTimeMillis() - start;
            res.setText("");
            res.setTextColor(Color.parseColor("#00FF00"));
            String result = "";
            switch (action){
                case CONNECT:
                    result = "Connection established";
                    activity.updateUI(Action.CONNECT);
                    break;
                case DISCONNECT:
                    result = "Disconnection confirmed";
                    activity.updateUI(Action.DISCONNECT);
                    break;
                case SUBSCRIBE:
                    result = "Subscription confirmed";
                    break;
                case UNSUBSCRIBE:
                    result = "Unsubscription confirmed";
                    break;
                case PUBLISH:
                    result = "Publication confirmed";
                    break;
            }
            Log.i("SUCCESS", result.split(" ")[0]);
            res.setText(result);
            if(action != Action.PUBLISH){
                Measurements.addTime(performingTime);
            }
        }
    }
```

Listato A.18: MqttCallBack.java

```
public class MQTTCallBack implements MqttCallback {
    MQTTActivity activity;
    TextView res;

    public MQTTCallBack(MQTTActivity ctx) {
        this.activity = ctx;
        res = (TextView)activity.findViewById(R.id.res);
    }

    @Override
    public void connectionLost(Throwable arg0) {
        if(arg0 != null){
            res.setText("");
            res.setTextColor(Color.parseColor("#FF0000"));
            res.setText("Connection lost: " + arg0.toString());
            activity.updateUI(Action.DISCONNECT);
        }
```

```
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken arg0) {
        Measurements.endPublish(arg0.getMessageId(), System.currentTimeMillis());
              res.setText("");
              res.setTextColor(Color.parseColor("#00FF00"));
              res.setText("message "+ arg0.getMessageId() +" delivered");
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        res.setText("");
        res.setTextColor(Color.parseColor("#00FF00"));
        res.setText("message received on topic:\n" + topic +
              "\npayload:\n"+new String(message.getPayload()));
    }
}
```

Listato A.19: ButtonListener.java

```
public class ButtonListener implements OnClickListener{
    EditText topic, message, host, nmsg;
    TextView results;
    MQTTActivity activity;
    MqttAndroidClient client;
    MqttConnectOptions connOpt;
    String imei;

    public ButtonListener(MQTTActivity activity, String imei){
        this.activity = activity;
        this.topic = (EditText)activity.findViewById(R.id.txt_topic);
        this.message = (EditText)activity.findViewById(R.id.txt_msg);
        this.host = (EditText)activity.findViewById(R.id.broker_addr);
        this.results = (TextView)activity.findViewById(R.id.res);
        this.nmsg = (EditText)activity.findViewById(R.id.nmsg);
        this.imei = imei;
        Measurements.setActivity(activity);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()){
        case  R.id.connect:
            Measurements.setMeasurements(1);
            try {
                if(!activity.isConnected()){
                    client = new MqttAndroidClient(activity, host.getText().toString(), imei);
                    client.setCallback(new MQTTCallBack(activity));
                    connOpt = new MqttConnectOptions();
                    connOpt.setCleanSession(true);
                    connOpt.setConnectionTimeout(500);
                    connOpt.setWill("notify", (imei + " lose connection").getBytes(), 0, false);
                    connOpt.setKeepAliveInterval(500);
                    client.connect(connOpt, R.id.connect,
                          new MQTTActionListener(activity, Action.CONNECT));
                }
                else{
```

```
            client.disconnect(R.id.connect,
                  new MQTTActionListener(activity, Action.DISCONNECT));
            }
         } catch (MqttException e) {
            e.printStackTrace();
         }
         break;
      case  R.id.publish:
         if(client.isConnected() && message.getText().length() > 0 && topic.getText().length() > 0){
            try {
               int numMessages = 1;
               if(!nmsg.getText().toString().isEmpty())
                  if(Integer.parseInt(nmsg.getText().toString()) > 1)
                     numMessages = Integer.parseInt(nmsg.getText().toString());
               Measurements.setMeasurements(numMessages);
               Long start = System.currentTimeMillis();
               MqttMessage msg = new MqttMessage(message.getText().toString().getBytes());
               msg.setQos(1);
               msg.setRetained(false);
               for(int i = 0; i < numMessages; i++){
                  int msgId = client.publish(topic.getText().toString(), msg, activity,
                        new MQTTActionListener(activity, Action.PUBLISH)).getMessageId();
                  Measurements.startPublish(msgId, start);
               }
            } catch (MqttPersistenceException e) {
               e.printStackTrace();
            } catch (MqttException e) {
               e.printStackTrace();
            }
         }
         else{
            results.setText("");
            results.setTextColor(Color.parseColor("#FF0000"));
            results.setText("ERROR: client not connected or topic and/or payload is empty.");
         }
         break;
      case  R.id.subscribe:
         if(client.isConnected() && topic.length() > 0){
            Measurements.setMeasurements(1);
            try {
               client.subscribe(topic.getText().toString(), 0, activity,
                     new MQTTActionListener(activity, Action.SUBSCRIBE));
            } catch (MqttException e) {
               e.printStackTrace();
            }
         }
         else{
            results.setText("");
            results.setTextColor(Color.parseColor("#FF0000"));
            results.setText("ERROR: client not connected or topic is empty.");
         }
         break;
      case  R.id.unsubscribe:
         if(client.isConnected() && topic.length() > 0){
            Measurements.setMeasurements(1);
            try {
               client.unsubscribe(topic.getText().toString(), activity,
                     new MQTTActionListener(activity, Action.UNSUBSCRIBE));
            } catch (MqttException e) {
```

```
                e.printStackTrace();
            }
        }
        else{
            results.setText("");
            results.setTextColor(Color.parseColor("#FF0000"));
            results.setText("ERROR: client not connected or topic is empty.");
        }
        break;
    }
  }
}
```

Listato A.20: MqttActivity.java

```
public class MQTTActivity extends Activity{

    private Button pub, sub, conn, unsub;
    private EditText topic, message, host;
    private TextView max, min, avg;
    private ButtonListener buttonListener;
    private String imei;
    private boolean connected;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_mqtt);
        imei = ((TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE)).getDeviceId();
        initUI();
    }

    private void initUI(){
        // get UI elements
        pub = (Button)findViewById(R.id.publish);
        sub = (Button)findViewById(R.id.subscribe);
        unsub = (Button)findViewById(R.id.unsubscribe);
        conn = (Button)findViewById(R.id.connect);
        host = (EditText)findViewById(R.id.broker_addr);
        topic = (EditText)findViewById(R.id.txt_topic);
        message = (EditText)findViewById(R.id.txt_msg);
        max = (TextView)findViewById(R.id.textViewMax);
        min = (TextView)findViewById(R.id.textViewMin);
        avg = (TextView)findViewById(R.id.textViewAvg);
        // create a new button listener
        buttonListener = new ButtonListener(this, imei);
        // assign button listener to all buttons
        conn.setOnClickListener(buttonListener);
        pub.setOnClickListener(buttonListener);
        sub.setOnClickListener(buttonListener);
        unsub.setOnClickListener(buttonListener);
        updateUI(Action.DISCONNECT);
    }

    public void updateUI(Action action) {
        if(action == Action.CONNECT){
            connected = true;
            conn.setText("Disconnect");
```

```
            pub.setEnabled(true);
            sub.setEnabled(true);
            unsub.setEnabled(true);
            topic.setEnabled(true);
            message.setEnabled(true);
            host.setEnabled(false);
        }
        else{
            connected = false;
            conn.setText("Connect");
            pub.setEnabled(false);
            sub.setEnabled(false);
            unsub.setEnabled(false);
            topic.setEnabled(false);
            topic.setText("");
            message.setEnabled(false);
            message.setText("");
            host.setEnabled(true);
        }
    }

    public boolean isConnected() {
        return connected;
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    public void updateTiming(long max, long min, double avg) {
        this.max.setText("MAX: "+ max);
        this.min.setText("MIN: "+ min);
        this.avg.setText("AVG: "+ avg);
    }
}
```

Listato A.21: Measurements.java

```
public class Measurements {
    private static List<Long> times = new ArrayList<Long>();
    private static SparseArray<Long> startTimes = new SparseArray<Long>();
    private static int measurementsNum = 0;
    private static int counter = 0;
    private static MQTTActivity activity;

    public static void setActivity(MQTTActivity act){
        activity = act;
    }

    public static void setMeasurements(int num){
        measurementsNum = num;
        counter = 0;
        times.clear();
    }

    public static void addTime(long time){
        times.add(time);
```

```
        Log.i("Time added", time+"");
        counter++;
        if(counter == measurementsNum){
            activity.updateTiming(getMaxTime(), getMinTime(), getAvgTime());
            times.clear();
            counter = 0;
            measurementsNum = 0;
        }
    }

    public static void startPublish(int msgId, long startTime){
        startTimes.put(msgId, startTime);
    }

    public static void endPublish(int msgId, long endTime){
        addTime(endTime-startTimes.get(msgId));
        startTimes.remove(msgId);
    }

    private static double getAvgTime(){
        double avg = 0;
        for (long t : times)
            avg = avg + t;
        avg = avg/times.size();
        return avg;
    }

    private static long getMaxTime(){
        long max = times.get(0);
        for(long t : times)
            if (t > max)
                max = t;
        return max;
    }


    private static long getMinTime(){
        long min = times.get(0);
        for(long t : times)
            if (t < min)
                min = t;
        return min;
    }
}
```

A.4.2 Geolocazione

Listato A.22: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.antenucci.test.android.geolocation.playservice"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="9"
```

```
        android:targetSdkVersion="21" />

    <uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="it.antenucci.test.android.geolocation.playservice.GeolocationActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

        <meta-data
            android:name="com.google.android.gms.version"
            android:value="@integer/google_play_services_version" />
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="AIzaSyDE6hLUdKweg-mXHydqGlO9wR9WHEOCiTQ" />
    </application>

</manifest>
```

Listato A.23: activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="it.antenucci.test.android.geolocation.playservice.GeolocationActivity" >

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >
    </TextView>

</RelativeLayout>
```

Listato A.24: GeolocationListener.java

```
public class GeolocationListener implements LocationListener, ConnectionCallbacks,
     OnConnectionFailedListener {

   private String topic;
   private TextView tv;
   private Activity activity;
   private GoogleApiClient mGoogleApiClient;
   private Geocoder geocoder;

   public GeolocationListener(Activity activity){
      this.activity = activity;
      tv = (TextView)activity.findViewById(R.id.textView);
      mGoogleApiClient = new GoogleApiClient
            .Builder(activity)
            .addApi(Places.PLACE_DETECTION_API)
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .build();
      geocoder = new Geocoder(activity, Locale.ENGLISH);
   }

   @Override
   public void onLocationChanged(Location location) {
      topic = getTopic(location.getLatitude(), location.getLongitude());
      mGoogleApiClient.connect();
   }

   @Override
   public void onConnected(Bundle arg0) {
      final String LOG_TAG = "ON_CONNECTED";
      Log.i(LOG_TAG, "google api client is connected, getting results");
      PendingResult<PlaceLikelihoodBuffer> result = Places.PlaceDetectionApi
            .getCurrentPlace(mGoogleApiClient, null);
      result.setResultCallback(new ResultCallback<PlaceLikelihoodBuffer>() {
         @Override
         public void onResult(PlaceLikelihoodBuffer likelyPlaces) {
            Log.i(LOG_TAG, "printing " + likelyPlaces.getCount() + " results");
            tv.setText("");
            for (PlaceLikelihood placeLikelihood : likelyPlaces) {
               String text = String.format(Locale.ENGLISH, "Place 'ʼs' has likelihood: %g",
                     placeLikelihood.getPlace().getName(),
                     placeLikelihood.getLikelihood());
               Log.i(LOG_TAG, text);
               Log.i(LOG_TAG, "\nTopic: " + topic + "/" + placeLikelihood.getPlace().getId() + "\n\n")
                    ;
               tv.append(text + "\nTopic: " + topic + "/" + placeLikelihood.getPlace().getId() + "\n\n
                    ");
            }
            likelyPlaces.release();
            mGoogleApiClient.disconnect();
         }
      });
   }

   private String getTopic(double latitude, double longitude) {
```

```
        String LOG_TAG = "getAddress()";
        Log.i(LOG_TAG , "Creating the topic hierarcy with available informations");
        try {
           List<Address> addresses = geocoder.getFromLocation(latitude ,longitude, 1);
           Address obj = addresses.get(0);
           String topic = "";
           topic += "/" + obj.getCountryName(); //nome nazione  *1
           if (obj.getAdminArea() != null)
              topic += "/" + obj.getAdminArea(); //regione
           if(obj.getSubAdminArea() != null)
              topic += "/" + obj.getSubAdminArea(); //provincia
           topic += "/" + obj.getLocality(); //cittÃ
           topic += "/" + obj.getPostalCode(); //CAP
           return topic;

        } catch (IOException e) {
           // TODO Auto-generated catch block
           e.printStackTrace();
           Toast.makeText(activity, e.getMessage(), Toast.LENGTH_SHORT).show();
           return "";
        }
     }

     @Override
     public void onConnectionFailed(ConnectionResult arg0) {}

     @Override
     public void onConnectionSuspended(int arg0) {}

     @Override
     public void onStatusChanged(String provider, int status, Bundle extras) {}

     @Override
     public void onProviderEnabled(String provider) {}

     @Override
     public void onProviderDisabled(String provider) {}

}
```

Listato A.25: GeolocationActivity.java

```
1  public class GeolocationActivity extends Activity{
2
3     private LocationManager lm;
4     private GeolocationListener geolocationListener;
5
6     @Override
7     protected void onCreate(Bundle savedInstanceState) {
8        super.onCreate(savedInstanceState);
9        setContentView(R.layout.activity_main);
10       super.onCreate(savedInstanceState);
11       lm = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
12       // Define a listener that responds to location updates
13       geolocationListener = new GeolocationListener(this);
14
15       updateLocation();
16    }
```

```
17
18      @Override
19      public boolean onCreateOptionsMenu(Menu menu) {
20          // Inflate the menu; this adds items to the action bar if it is present.
21          getMenuInflater().inflate(R.menu.main, menu);
22          return true;
23      }
24
25      @Override
26      public boolean onOptionsItemSelected(MenuItem item) {
27          // Handle action bar item clicks here. The action bar will
28          // automatically handle clicks on the Home/Up button, so long
29          // as you specify a parent activity in AndroidManifest.xml.
30          int id = item.getItemId();
31          if (id == R.id.action_settings) {
32              return true;
33          }
34          return super.onOptionsItemSelected(item);
35      }
36
37
38      private void updateLocation(){
39          lm.requestSingleUpdate(
40                  LocationManager.NETWORK_PROVIDER,
41                  geolocationListener,
42                  null);
43      }
44  }
```

A.4.3 Connessione WiFi

Listato A.26: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.antenucci.test.android.wifi"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="21" />

    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" >
    </uses-permission>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" >
    </uses-permission>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" >
    </uses-permission>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".WifiActivity"
```

```
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listato A.27: activity_wifi.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="it.antenucci.test.android.wifi.WifiActivity" >
    <TextView
        android:id="@+id/lblSSID"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_marginLeft="15dp"
        android:layout_marginTop="15dp"
        android:text="SSID" />
    <EditText
        android:id="@+id/ssid"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/lblSSID"
        android:layout_alignParentRight="true"
        android:ems="10"
        android:text="TestNetwork" >
        <requestFocus />
    </EditText>
    <TextView
        android:id="@+id/lblKey"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/lblSSID"
        android:layout_below="@+id/ssid"
        android:text="Key" />
    <EditText
        android:id="@+id/key"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/ssid"
        android:layout_below="@+id/lblSSID"
        android:ems="10"
        android:inputType="textPassword"
        android:text="0123456789" />
    <RadioGroup
        android:id="@+id/radioSecurity"
        android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/key"
    android:layout_marginTop="15dp" >
    <RadioButton
        android:id="@+id/open"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="Open network" />
    <RadioButton
        android:id="@+id/wep"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="WEP" />
    <RadioButton
        android:id="@+id/wpa"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="WPA" />
</RadioGroup>
<Button
    android:id="@+id/execute"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/radioSecurity"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="20dp"
    android:text="Connect" />
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/execute"
    android:layout_marginTop="14dp"
    android:orientation="vertical" >
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent" >
        <TextView
            android:id="@+id/Meas"
            android:layout_width="60dp"
            android:layout_height="wrap_content"
            android:text="" />
        <TextView
            android:id="@+id/Conf"
            android:layout_width="90dp"
            android:layout_height="wrap_content"
            android:text="Configuration" />
        <TextView
            android:id="@+id/Conn"
            android:layout_width="80dp"
            android:layout_height="wrap_content"
            android:text="Connection" />
        <TextView
            android:id="@+id/Tot"
            android:layout_width="70dp"
```

```
        android:layout_height="wrap_content"
        android:text="Total" />
</LinearLayout>
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/Current"
        android:layout_width="60dp"
        android:layout_height="wrap_content"
        android:text="Current" />
    <TextView
        android:id="@+id/CurrentConf"
        android:layout_width="90dp"
        android:layout_height="wrap_content"
        android:text="TextView" />
    <TextView
        android:id="@+id/CurrentConn"
        android:layout_width="80dp"
        android:layout_height="wrap_content"
        android:text="TextView" />
    <TextView
        android:id="@+id/CurrentTot"
        android:layout_width="70dp"
        android:layout_height="wrap_content"
        android:text="TextView" />
</LinearLayout>
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/Avg"
        android:layout_width="60dp"
        android:layout_height="wrap_content"
        android:text="Average" />
    <TextView
        android:id="@+id/AVGConf"
        android:layout_width="90dp"
        android:layout_height="wrap_content"
        android:text="TextView" />
    <TextView
        android:id="@+id/AVGConn"
        android:layout_width="80dp"
        android:layout_height="wrap_content"
        android:text="TextView" />
    <TextView
        android:id="@+id/AVGTot"
        android:layout_width="70dp"
        android:layout_height="wrap_content"
        android:text="TextView" />
</LinearLayout>
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/Max"
        android:layout_width="60dp"
        android:layout_height="wrap_content"
        android:text="Max" />
```

```
            <TextView
                android:id="@+id/MaxConf"
                android:layout_width="90dp"
                android:layout_height="wrap_content"
                android:text="TextView" />
            <TextView
                android:id="@+id/MaxConn"
                android:layout_width="80dp"
                android:layout_height="wrap_content"
                android:text="TextView" />
        </LinearLayout>
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="match_parent" >
            <TextView
                android:id="@+id/Min"
                android:layout_width="60dp"
                android:layout_height="wrap_content"
                android:text="Min" />
            <TextView
                android:id="@+id/MinConf"
                android:layout_width="90dp"
                android:layout_height="wrap_content"
                android:text="TextView" />
            <TextView
                android:id="@+id/MinConn"
                android:layout_width="80dp"
                android:layout_height="wrap_content"
                android:text="TextView" />
        </LinearLayout>
    </LinearLayout>
</RelativeLayout>
```

Listato A.28: WifiConnectionManager.java

```
public class WifiConnectionManager {

   public static enum Security{OPEN, WEP, WPA};
   private WifiManager manager;
   private int id_net = -1;
   private ArrayList<Integer> confs = new ArrayList<Integer>();
   private Timer timer;
   private WifiActivity activity;

   public WifiConnectionManager(WifiActivity act, WifiManager manager){
      this.activity = act;
      this.manager = manager;
   }

   public void connect(String ssid, String key, Security type){
      manager.setWifiEnabled(true);
      Measurements.startConf();
      WifiConfiguration conf = new WifiConfiguration();
      conf.SSID = "\"" + ssid + "\"";
      switch(type){
      case OPEN:
         conf.status = WifiConfiguration.Status.ENABLED;
         conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.NONE);
```

```
            break;
        case WEP:
            if (isHexString(key))
                conf.wepKeys[0] = key;
            else
                conf.wepKeys[0] = "\"" + key + "\"";
            conf.wepTxKeyIndex = 0;
            conf.status = WifiConfiguration.Status.ENABLED;
            conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.NONE);
            conf.allowedProtocols.set(WifiConfiguration.Protocol.RSN);
            conf.allowedProtocols.set(WifiConfiguration.Protocol.WPA);
            conf.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.OPEN);
            conf.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.SHARED);
            conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.CCMP);
            conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.TKIP);
            conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.WEP40);
            conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.WEP104);
            break;
        case WPA:
            conf.preSharedKey = "\""+ key +"\"";
            conf.status = WifiConfiguration.Status.ENABLED;
            conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.TKIP);
            conf.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.CCMP);
            conf.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_PSK);
            conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.TKIP);
            conf.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.CCMP);
            conf.allowedProtocols.set(WifiConfiguration.Protocol.RSN);
            conf.allowedProtocols.set(WifiConfiguration.Protocol.WPA);
            break;
        default:
            return;
        }
        id_net = manager.addNetwork(conf);
        confs.add(id_net);
        Measurements.startConn();
        manager.disconnect();
        manager.enableNetwork(id_net, true);
        manager.reconnect();
        //if in 30 secs not connected notify to user
        timer = new Timer();
        timer.schedule(new ConnectionTimeoutTask(), 30000);

    }

    private boolean isHexString(String key) {
        try {
            Long.parseLong(key, 16);
            return true;
        }
        catch (NumberFormatException ex) {
            return false;
        }
    }

    public void disconnect(){
        manager.disconnect();
        manager.removeNetwork(id_net);
        manager.saveConfiguration();
        confs.remove((Object)id_net);
```

```
        id_net = -1;
    }

    public void deleteGatewayNetworks(){
        for (int id : confs){
            manager.removeNetwork(id);
        }
        manager.saveConfiguration();
        confs.clear();
    }

    public void stopTimer(){
        if ((timer!=null))
            timer.cancel();
    }

    private class ConnectionTimeoutTask extends TimerTask{
        @Override
        public void run() {
            Measurements.finish();
            Log.i("CONNECTION PROBLEM", "UNABLE TO CONNECT SPECIFIED NETWORK");
            manager.disconnect();
            manager.removeNetwork(id_net);
            manager.saveConfiguration();
            manager.setWifiEnabled(false);
            confs.remove((Object)id_net);
            id_net = -1;
            Measurements.fail();
            timer.cancel();
            activity.runOnUiThread(new Runnable(){
                @Override
                public void run() {
                    Toast.makeText((activity), "UNABLE TO CONNECT SPECIFIED NETWORK", Toast.LENGTH_SHORT).
                        show();
                }

            });
        }
    }
```

Listato A.29: WifiReceiver.java

```
public class WifiReceiver extends BroadcastReceiver {

    WifiManager wifiService;
    WifiConnectionManager wifiManager;

    public WifiReceiver(WifiManager wifiService, WifiConnectionManager wifiManager){
        super();
        this.wifiService = wifiService;
        this.wifiManager = wifiManager;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i("ACTION", intent.getAction());
        Log.i("STATE", ""+wifiService.getConnectionInfo().getSupplicantState().toString());
        String btnTxt = "";
```

```
        if ((wifiService.getConnectionInfo().getSupplicantState().equals(SupplicantState.COMPLETED)) &&
              (wifiService.getConnectionInfo().getSSID().equals("\""+((WifiActivity)context).getSSID()+"
                   \""))){
           Measurements.finish();
           btnTxt = "Disconnect";
           Log.i("WifiReceiver", "CONNECTED");
           Toast.makeText(((WifiActivity)context), "CONNECTED", Toast.LENGTH_SHORT).show();
        }
        else {
           btnTxt = "Connect";
           if(!wifiService.getConnectionInfo().getSupplicantState().equals(SupplicantState.DISCONNECTED)
                ){
              Log.i("WifiReceiver", "NOT CONNECTED TO THE SPECIFIED NETWORK");
              Toast.makeText(((WifiActivity)context), "NOT CONNECTED TO THE SPECIFIED NETWORK", Toast.
                   LENGTH_SHORT).show();
           }
           else{
              Log.i("WifiReceiver", "DISCONNECTED");
              Toast.makeText(((WifiActivity)context), "DISCONNECTED", Toast.LENGTH_SHORT).show();
           }
           wifiManager.deleteGatewayNetworks();
        }
        wifiManager.stopTimer();
        ((WifiActivity)context).changeButtonTxt(btnTxt);
     }
```

Listato A.30: WifiActivity.java

```
public class WifiActivity extends Activity {
   private WifiConnectionManager wifiManager;
   private Button btn;
   private EditText ssid;
   private EditText key;
   private RadioGroup radio;
   private WifiManager wifiService;
   private WifiReceiver wr;
   private TextView currConn, currConf, avgConn, avgConf;
   private TextView minConn, minConf, maxConn, maxConf;
   private TextView currTot, avgTot;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_wifi);
      wifiService = (WifiManager)getSystemService(Context.WIFI_SERVICE);
      wifiManager = new WifiConnectionManager(this, wifiService);
      Measurements.setActivity(this);
      if(wr == null)
         registerWifiReceiver();
      btn = (Button)findViewById(R.id.execute);
      ssid = (EditText)findViewById(R.id.ssid);
      key = (EditText)findViewById(R.id.key);
      radio = (RadioGroup)findViewById(R.id.radioSecurity);
      currConf = (TextView)findViewById(R.id.CurrentConf);
      currConn = (TextView)findViewById(R.id.CurrentConn);
      currTot = (TextView)findViewById(R.id.CurrentTot);
      avgConf = (TextView)findViewById(R.id.AVGConf);
      avgConn = (TextView)findViewById(R.id.AVGConn);
```

```
    avgTot = (TextView)findViewById(R.id.AVGTot);
    maxConf = (TextView)findViewById(R.id.MaxConf);
    maxConn = (TextView)findViewById(R.id.MaxConn);
    minConf = (TextView)findViewById(R.id.MinConf);
    minConn = (TextView)findViewById(R.id.MinConn);
    btn.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            if (((Button)v).getText().toString().equalsIgnoreCase("connect")){
                Security security = null;
                switch(radio.getCheckedRadioButtonId()){
                case R.id.open:
                    security = Security.OPEN;
                    break;
                case R.id.wep:
                    security = Security.WEP;
                    break;
                case R.id.wpa:
                    security = Security.WPA;
                    break;
                }
                wifiManager.connect(ssid.getText().toString(), key.getText().toString(), security);
            }
            else{
                wifiManager.disconnect();
            }
        }
    });
}

@Override
protected void onResume() {
    super.onResume();
    registerWifiReceiver();
}

@Override
protected void onPause() {
    super.onPause();
    if(wr != null){
        unregisterReceiver(wr);
        wr = null;
    }
}

public void changeButtonTxt(final String text){
    this.runOnUiThread(new Runnable(){
        @Override
        public void run() {
            if(!btn.getText().toString().equals(text)){
                btn.setText(text);
            }
        }
    });
}

public String getSSID() {
    return ssid.getText().toString();
}
```

```
    private void registerWifiReceiver(){
        if(wr==null){
            IntentFilter intentFilter = new IntentFilter();
            intentFilter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
            wr = new WifiReceiver(wifiService, wifiManager);
            registerReceiver(wr, intentFilter);
        }
    }

    public void printTiming(final long confNow, final long connNow, final double confAVG,
                final double connAVG, final long confMax, final long connMax,
                final long confMin, final long connMin) {
        runOnUiThread(new Runnable() {

            @Override
            public void run() {
                currConf.setText(""+confNow);
                currConn.setText(""+connNow);
                currTot.setText(""+(confNow+connNow));
                avgConf.setText(""+confAVG);
                avgConn.setText(""+connAVG);
                avgTot.setText(""+(confAVG+connAVG));
                maxConf.setText(""+confMax);
                maxConn.setText(""+connMax);
                minConf.setText(""+confMin);
                minConn.setText(""+connMin);
            }
        });
    }
}
```

Listato A.31: Measurements.java

```
public class Measurements {
    public static long connTime =0 , confTime =0;
    private static List<Long> connections = new ArrayList<Long>();
    private static List<Long> configurations = new ArrayList<Long>();
    private static WifiActivity activity;

    public static void setActivity(WifiActivity act){
        activity = act;
        configurations.clear();
        connections.clear();
    }

    public static void startConf(){
        confTime = System.currentTimeMillis();
    }

    public static void startConn(){
        connTime = System.currentTimeMillis();
        confTime = connTime - confTime;
        configurations.add(confTime);
    }

    public static void finish(){
        if(connTime == 0)
```

```
            return;
        connTime = System.currentTimeMillis() - connTime;
        connections.add(connTime);
        activity.printTiming(confTime, connTime, getAvg(configurations), getAvg(connections),
                    getMax(configurations), getMax(connections), getMin(configurations), getMin(
                        connections));
        confTime = 0;
        connTime = 0;
    }

    private static double getAvg(List<Long> list) {
        if(list.isEmpty())
            return 0;
        double avg = 0;
        for(Long t : list)
            avg = avg + t;
        avg = avg / list.size();
        return avg;
    }

    private static long getMax(List<Long> list){
        if(list.isEmpty())
            return 0;
        long max = list.get(0);
        for (long t: list)
        if(t>max)
            max = t;
        return max;
    }

    private static long getMin(List<Long> list){
        if(list.isEmpty())
            return 0;
        long min = list.get(0);
        for (long t: list)
        if(t<min)
            min = t;
        return min;
    }

    public static void fail() {
        if (configurations.size()>connections.size())
            configurations.remove(configurations.size()-1);
        confTime = 0;
        connTime = 0;
    }
}
```

A.4.4 Utilizzo sensori

Listato A.32: activity_sensor.xml

```
1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2      xmlns:tools="http://schemas.android.com/tools"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent"
5      android:paddingBottom="@dimen/activity_vertical_margin"
```

```
6       android:paddingLeft="@dimen/activity_horizontal_margin"
7       android:paddingRight="@dimen/activity_horizontal_margin"
8       android:paddingTop="@dimen/activity_vertical_margin"
9       tools:context="it.antenucci.test.android.sensors.SensorActivity" >
10
11      <ListView
12          android:id="@+id/sensorList"
13          android:layout_width="match_parent"
14          android:layout_height="wrap_content"
15          android:layout_above="@+id/updates"
16          android:layout_alignParentTop="true"
17          android:layout_centerHorizontal="true" >
18      </ListView>
19
20      <TextView
21          android:id="@+id/updates"
22          android:layout_width="wrap_content"
23          android:layout_height="180dp"
24          android:layout_above="@+id/startupdate"
25          android:layout_alignParentLeft="true"
26          android:layout_alignParentRight="true"
27          android:gravity="bottom"
28          android:maxLines="12"
29          android:scrollbars="vertical" />
30
31      <Button
32          android:id="@+id/startupdate"
33          android:layout_width="wrap_content"
34          android:layout_height="wrap_content"
35          android:layout_alignParentBottom="true"
36          android:layout_centerHorizontal="true"
37          android:text="Start Updating Sensors" />
38
39  </RelativeLayout>
```

Listato A.33: sensor_row.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <CheckBox
        android:id="@+id/checkBox1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="CheckBox" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_vertical|right"
        android:text="TextView" />
</LinearLayout>
```

Listato A.34: SensorUtils.java

```
public class SensorUtils {
   public static String getSensorTypeAsString(int type) {
      switch(type){
      case Sensor.TYPE_ACCELEROMETER:
         return "Accelerometer";
      case Sensor.TYPE_AMBIENT_TEMPERATURE:
         return "Ambient Temperature";
      case Sensor.TYPE_GAME_ROTATION_VECTOR:
         return "Game Rotation Vector";
      case Sensor.TYPE_GEOMAGNETIC_ROTATION_VECTOR:
         return "Geomagnetic Rotation Vector";
      case Sensor.TYPE_GRAVITY:
         return "Gravity";
      case Sensor.TYPE_GYROSCOPE:
         return "Gyroscope";
      case Sensor.TYPE_GYROSCOPE_UNCALIBRATED:
         return "Gyroscope Uncalibrated";
      case Sensor.TYPE_HEART_RATE:
         return "Heart Rate";
      case Sensor.TYPE_LIGHT:
         return "Light";
      case Sensor.TYPE_LINEAR_ACCELERATION:
         return "Linear Acceleration";
      case Sensor.TYPE_MAGNETIC_FIELD:
         return "Magnetic Field";
      case Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED:
         return "Magnetic Field Uncalibrated";
      case Sensor.TYPE_ORIENTATION:
         return "Orientation";
      case Sensor.TYPE_PRESSURE:
         return "Pressure";
      case Sensor.TYPE_PROXIMITY:
         return "Proximity";
      case Sensor.TYPE_RELATIVE_HUMIDITY:
         return "Relative Humidity";
      case Sensor.TYPE_ROTATION_VECTOR:
         return "Rotation Vector";
      case Sensor.TYPE_SIGNIFICANT_MOTION:
         return "Significant Motion";
      case Sensor.TYPE_STEP_COUNTER:
         return "Step Counter";
      case Sensor.TYPE_STEP_DETECTOR:
         return "Step Detector";
      }
      return null;
   }

   public static int getSensorTypeAsInt(String type) {
      switch(type){
      case "Accelerometer":
         return Sensor.TYPE_ACCELEROMETER;
      case "Ambient Temperature":
         return Sensor.TYPE_AMBIENT_TEMPERATURE;
      case "Game Rotation Vector":
         return Sensor.TYPE_GAME_ROTATION_VECTOR;
      case "Geomagnetic Rotation Vector":
         return Sensor.TYPE_GEOMAGNETIC_ROTATION_VECTOR;
```

```
        case "Gravity":
            return Sensor.TYPE_GRAVITY;
        case "Gyroscope":
            return Sensor.TYPE_GYROSCOPE;
        case "Gyroscope Uncalibrated":
            return Sensor.TYPE_GYROSCOPE_UNCALIBRATED;
        case "Heart Rate":
            return Sensor.TYPE_HEART_RATE;
        case "Light":
            return Sensor.TYPE_LIGHT;
        case "Linear Acceleration":
            return Sensor.TYPE_LINEAR_ACCELERATION;
        case "Magnetic Field":
            return Sensor.TYPE_MAGNETIC_FIELD;
        case "Magnetic Field Uncalibrated":
            return Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED;
        case "Orientation":
            return Sensor.TYPE_ORIENTATION;
        case "Pressure":
            return Sensor.TYPE_PRESSURE;
        case "Proximity":
            return Sensor.TYPE_PROXIMITY;
        case "Relative Humidity":
            return Sensor.TYPE_RELATIVE_HUMIDITY;
        case "Rotation Vector":
            return Sensor.TYPE_ROTATION_VECTOR;
        case "Significant Motion":
            return Sensor.TYPE_SIGNIFICANT_MOTION;
        case "Step Counter":
            return Sensor.TYPE_STEP_COUNTER;
        case "Step Detector":
            return Sensor.TYPE_STEP_DETECTOR;
        }
        return -1;
    }
}
```

Listato A.35: SensorActivity.java

```
public class SensorActivity extends Activity {

    SensorManager sensorManager;
    ListView sensorList;
    TextView updates;
    Button startUpdates;
    SensorAdapter sensorAdapter;
    SensorListener sensorListener = new SensorListener();
    ScheduledThreadPoolExecutor exec = new ScheduledThreadPoolExecutor(1);
    long period = 5; // the period between successive executions
    ScheduledFuture<?> scheduleFuture;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sensor);
        //get layout components
        sensorList = (ListView)findViewById(R.id.sensorList);
        updates = (TextView)findViewById(R.id.updates);
```

```
    updates.setMovementMethod(new ScrollingMovementMethod());
    startUpdates = (Button)findViewById(R.id.startupdate);
    //get sensor manager
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    //creates sensor adapter and add it to sensorList
    sensorAdapter = new SensorAdapter(this, R.layout.sensor_row, sensorManager.getSensorList(Sensor.
        TYPE_ALL));
    sensorList.setAdapter(sensorAdapter);
    startUpdates.setOnClickListener(new OnClickListener(){
       @Override
       public void onClick(View v) {
          if (scheduleFuture == null){
             List<Sensor> enabledSensors = sensorAdapter.getEnabledSensors();
             if(enabledSensors.isEmpty()){
                updates.append("\n    NO SENSORS ENABLED\n");
                ((Button)v).setText("Start Sensor Updates");
                return;
             }
             scheduleFuture = exec.scheduleAtFixedRate(new PrintDataTask(SensorActivity.this), 2,
                                         period, TimeUnit.SECONDS);
             ((Button)v).setText("Stop Sensor Updates");
             Log.i("SensorActivity","Sensor update started");
          }
          else{
             stopPrintTask();
          }
       }

    });
}

@Override
protected void onStop() {
    for(Sensor sensor : sensorManager.getSensorList(Sensor.TYPE_ALL)){
       sensorManager.unregisterListener(sensorListener, sensor);
    }
    if(scheduleFuture != null)
       scheduleFuture.cancel(true);
    scheduleFuture = null;
    sensorAdapter.removeEnabled();
    super.onStop();
}

@Override
protected void onPause() {
    for(Sensor sensor : sensorManager.getSensorList(Sensor.TYPE_ALL)){
       sensorManager.unregisterListener(sensorListener, sensor);
    }
    if(scheduleFuture != null)
       scheduleFuture.cancel(true);
    scheduleFuture = null;
    sensorAdapter.removeEnabled();
    super.onPause();
}

@Override
protected void onResume() {
    startUpdates.setText("Start Sensor Updates");
    super.onResume();
```

```
    }

    public void registerSensorListener(Sensor sensor){
        sensorManager.registerListener(sensorListener, sensor, SensorManager.SENSOR_DELAY_NORMAL);
    }

    public void removeSensorListener(Sensor sensor){
        sensorManager.unregisterListener(sensorListener, sensor);
        sensorListener.remove(sensor.getType());
    }

    public SensorListener getSensorListener(){
        return sensorListener;
    }

    public SensorAdapter getSensorAdapter(){
        return sensorAdapter;
    }

    public boolean printTaskIsRunning() {
        return scheduleFuture != null;
    }

    public void stopPrintTask() {
        if(scheduleFuture != null)
            scheduleFuture.cancel(true);
        scheduleFuture = null;
        startUpdates.setText("Start Sensor Updates");
        Log.i("SensorActivity","Sensor update stopped");
    }

}
```

Listato A.36: PrintDataTask.java

```
public class PrintDataTask implements Runnable {
   SensorActivity activity;
   SensorAdapter sensorAdapter;
   SensorListener sensorListener;
   TextView updates;

   public PrintDataTask(SensorActivity act){
      activity = act;
      sensorAdapter = act.getSensorAdapter();
      sensorListener = act.getSensorListener();
      updates = (TextView)act.findViewById(R.id.updates);
   }

   @Override
   public void run() {
      for (Sensor sensor : sensorAdapter.getEnabledSensors()){
         final String lastup = sensorListener.getLastUpdate(sensor.getType());
         Log.i(SensorUtils.getSensorTypeAsString(sensor.getType()).toUpperCase(Locale.ENGLISH), lastup
              );
         activity.runOnUiThread(new Runnable(){
            @Override
            public void run() {
               updates.append("\n"+lastup+"\n");
            }
         });
      }
   }
}
```

A.5 Android MQTTDiscovery

Listato A.37: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.antenucci.discovery.mqtt.android"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="23" />

    <!-- imei permission -->
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <!-- internet permissions -->
    <uses-permission android:name="android.permission.INTERNET" />
    <!-- geolocation permissions -->
    <uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <!-- mqtt client permission -->
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <!-- wifi permissions -->
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="it.antenucci.discovery.mqtt.android.DiscoveryActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!-- MQTT service -->
         <service android:name="org.eclipse.paho.android.service.MqttService">

         </service>
        <!-- google apis metadata -->
        <meta-data
            android:name="com.google.android.gms.version"
            android:value="@integer/google_play_services_version" />
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="AIzaSyDE6hLUdKweg-mXHydqGlO9wR9WHEOCiTQ" />
    </application>

</manifest>
```

Listato A.38: AndroidManifest.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="it.antenucci.discovery.mqtt.android.DiscoveryActivity" >

    <ListView
        android:id="@+id/sensor_list"
        android:layout_width="match_parent"
        android:layout_height="fill_parent"
        android:layout_above="@+id/update_time_line"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" >

    </ListView>

    <LinearLayout
        android:id="@+id/update_time_line"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:layout_above="@+id/start_discovery"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/label_interval"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/interval_label"
            android:textAppearance="?android:attr/textAppearanceMedium" />

        <EditText
            android:id="@+id/interval"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:ems="10"
            android:inputType="number"
            android:text="@string/interval_value" >
        </EditText>
    </LinearLayout>

    <Button
        android:id="@+id/start_discovery"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_above="@+id/status"
        android:text="@string/start_buton"/>

    <TextView
```

```
        android:id="@+id/status"
        android:layout_width="match_parent"
        android:layout_height="140dp"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:maxLines="8"
        android:scrollbars="horizontal|vertical" />

</RelativeLayout>
```

Listato A.39: Constants.java

```
public class Constants {
   //likelihood treshold
   public static final float LIKELIHOOD_TRESHOLD = (float) 0.05;

   //cloud URI
   public static final String CLOUD_URI = "tcp://iot.eclipse.org:1883";

   //announce retry
   public static final int RETRY_ATTEMPT = 3;
   public static final int RETRY_INTERVAL = 10 * 1000; // 10 seconds
   //waiting time for gateways responses
   public static final int WAIT_TIME_FOR_GATEWAYS = 30 * 1000; // 30 seconds
   //wifi timeout
   public static final int WIFI_TIMEOUT = 15 * 1000; //15 seconds

   // DiscoveryObserver error messages
   public static final String GEOLOCATION_ERROR = "Unable to get device location.";
   public static final String MQTT_CONNECTION_ERROR = " broker not connected: ";
   public static final String SUBSCRIPTION_ERROR = "Unable to perform subscription: ";
   public static final String ANNOUNCE_NOT_PUBLISHED = "Unable to publish announce: ";
   public static final String NO_GATEWAY = "No responses from gateways";
   public static final String WIFI_CONNECTION_ERROR = "WiFi connection failed. Trying with another
        gateway...";
   public static final String DATA_NOT_PUBLISHED = "Unable to publish sensor data: ";
   public static final String WIFI_DISCONNECTED = "Disconnected from gateway's WiFi";
   public static final String DISCONNECTION_ERROR = "Disconnection error";
   public static final Object MQTT_CONNECTION_LOST = "Connection to the broker lost";

   //discovery observer constants
   public static final int GEOLOCATION_DONE = 1;
   public static final int CLOUD_CONNECTED = 2;
   public static final int SUBSCRIPTION_CONFIRMED = 3;
   public static final int ANNOUNCE_SENT = 4;
   public static final int INFO_GW_RECEIVED = 5;
   public static final int WIFI_CONNECTED = 6;
   public static final int GATEWAY_CONNECTED = 7;
   public static final int ANNOUNCE_PUBLISHED = 8;
   public static final int CLOUD_DISCONNECTED = 9;

   //sensor data topic
   public static final String SENSOR_DATA_TOPIC = "/sensor/";
   //will topic for local broker
   public static final String WILL_TOPIC = "/notify";
}
```

Listato A.40: DiscoveryActivity.java

```
public class DiscoveryActivity extends Activity {

   private TextView status;
   private EditText interval;
   private ListView sensorList;
   private Button startButton;
   private SensorManager sensorManager;
   private SensorAdapter sensorAdapter;
   private Discovery discovery;
   private ConnectivityManager connectivityManager;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_discovery);
      sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
      connectivityManager = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
      initUI();
   }

   private void initUI() {
      fillSensorList();
      registerStartButton();
      status = (TextView)findViewById(R.id.status);
      status.setMovementMethod(new ScrollingMovementMethod());
      interval = (EditText)findViewById(R.id.interval);
   }

   private void fillSensorList() {
      //get ListView element
      sensorList = (ListView)findViewById(R.id.sensor_list);
      //creates sensor adapter and add it to sensorList
      sensorAdapter = new SensorAdapter(this, R.layout.sensor_row, sensorManager.getSensorList(Sensor.
          TYPE_ALL));
      sensorList.setAdapter(sensorAdapter);
   }

   private void registerStartButton() {
      //get Button element
      startButton = (Button)findViewById(R.id.start_discovery);
      startButton.setOnClickListener(new OnClickListener() {
         @Override
         public void onClick(View v) {
            NetworkInfo activeNetworkInfo = connectivityManager.getActiveNetworkInfo();
            if(startButton.getText().toString().equals("Start Discovery")){
               if( activeNetworkInfo != null && activeNetworkInfo.isConnected())
                  discovery = new Discovery(DiscoveryActivity.this);
               else
                  appendStatus("Not connected to the internet. Unable to discover gateway.");
            }
            else
               if(discovery!=null)
                  discovery.stop();
         }
      });
   }
```

```
    public void enableUI(final boolean discoveryDone) {
        runOnUiThread(new Runnable(){
            public void run(){
                if(!discoveryDone){
                    for(int i = 0; i < sensorList.getChildCount(); i++){
                        sensorList.getChildAt(i).findViewById(R.id.checkBox1).setEnabled(true);
                    }
                    sensorAdapter.setListEnabled(true);
                    interval.setEnabled(true);
                    startButton.setText("Start Discovery");
                }
                else
                    startButton.setText("Disconnect Gateway");
                startButton.setEnabled(true);
            }
        });
    }

    public void disableUI() {
        runOnUiThread(new Runnable(){
            public void run(){
                for(int i = 0; i < sensorList.getChildCount(); i++){
                    sensorList.getChildAt(i).findViewById(R.id.checkBox1).setEnabled(false);
                }
                sensorAdapter.setListEnabled(false);
                interval.setEnabled(false);
                startButton.setEnabled(false);
            }
        });
    }


    public void appendStatus(final String text){
        runOnUiThread(new Runnable(){
            public void run(){
                String date = new SimpleDateFormat("HH:mm:ss", Locale.ENGLISH).format(Calendar.getInstance
                    ().getTime());
                status.setText(date + " - " + text + "\n" + status.getText().toString());
            }
        });
    }

    public void showToast(final String text) {
        runOnUiThread(new Runnable(){
            public void run(){
                Toast.makeText(DiscoveryActivity.this, text, Toast.LENGTH_SHORT).show();
            }
        });
    }

    public List<Sensor> getAvailableSensors() {
        return sensorAdapter.getEnabledSensors();
    }

    public long getInterval() {
        if(((TextView)findViewById(R.id.interval)).getText().toString().length()==0) return 0;
        return Integer.parseInt(((TextView)findViewById(R.id.interval)).getText().toString());
    }
}
```

Listato A.41: Discovery.java

```
public class Discovery {
   private String id;
   private List<String> availableSensors;
   private List<String> places;
   private List<String> topics;
   private double latitude;
   private double longitude;
   private DiscoveryActivity activity;
   private MqttAndroidClient client;
   private MqttMessage  mqttMessage = new MqttMessage();
   private WifiConnectionManager wifiManager;
   private InfoGatewayContent nearest;

   private ScheduledThreadPoolExecutor exec = new ScheduledThreadPoolExecutor(1);
   private ScheduledFuture<?> scheduleFuture;

   List<InfoGatewayContent> info = new ArrayList<InfoGatewayContent>();

   private long sensorUpdateInterval;
   private int placesCounter;
   private DiscoveryObserver discoveryObserver;
   private SendSensorDataTask job;

   public Discovery(DiscoveryActivity activity){
      this.activity = activity;
      String msg = "";
      sensorUpdateInterval = activity.getInterval();
      //check parameters
      if(sensorUpdateInterval <= 0){
         msg = "Update interval must be upper than 0";
      }
      availableSensors = new ArrayList<String>();
      for (Sensor s : activity.getAvailableSensors())
         availableSensors.add(SensorUtils.getSensorTypeAsString(s.getType()));
      if(availableSensors.isEmpty()){
         if(msg!="")
            msg+="\n";
         msg += "No sensors selected";
      }
      if(msg!=""){
         //if not satisfied end.
         activity.showToast(msg);
         return;
      }
      else{
         //starting discovery process
         activity.disableUI();
         activity.appendStatus("Sensors selected: " + getAvailableSensorsAsString());
         activity.appendStatus("Update interval: " + activity.getInterval() + " seconds");
         activity.appendStatus("Discovering Kura Gateways...");
         //init
         discoveryObserver = new DiscoveryObserver(this);
         id = ((TelephonyManager)activity.getSystemService(Activity.TELEPHONY_SERVICE)).getDeviceId();
         activity.appendStatus("Device ID setted to: " + id);
         wifiManager = new WifiConnectionManager(this, (WifiManager)activity.getSystemService(Activity
```

```
            .WIFI_SERVICE));
        places = new ArrayList<String>();
        topics = new ArrayList<String>();
        //get location
        ((LocationManager)activity.getSystemService(Activity.LOCATION_SERVICE)).requestSingleUpdate(
              LocationManager.NETWORK_PROVIDER,
              new GeolocationListener(activity, this),
              null);
    }
}

/*
 * Setter method for latitude and longitude. Invoked by GeolocationListener
 */
public void setLatitude(double latitude) {
    this.latitude = latitude;
    activity.appendStatus("Current Latitude: " + latitude);
}

public void setLongitude(double longitude) {
    this.longitude = longitude;
    activity.appendStatus("Current Longitude: " + longitude);
}

/*
 * Setter method for topics and place names. Invoked by DiscoveryObserver when
 * the geolocation process ends.
 */
public void setTopics(List<String>topics, List<String> names) {
    activity.appendStatus("Received " + topics.size() + " possible places for the device");
    this.topics = topics;
    this.places = names;
}

/*
 * This method establishes the connection to the cloud. Is invoked by DiscoveryObserver
 * when the Geolocation finishes.
 */
public void connectCloud() {
    //preparing announce message
    AnnounceContent announce = new AnnounceContent(id, availableSensors, latitude, longitude);
    Message payload = new Message(MessageType.ANNOUNCE, MessageUtils.getJsonString(announce));
    mqttMessage.setPayload(MessageUtils.getJsonString(payload).getBytes());
    mqttMessage.setQos(0);
    mqttMessage.setRetained(false);
    //configuring client and callback
    client = new MqttAndroidClient(activity, Constants.CLOUD_URI, id);
    client.setCallback(new MQTTCallBack(this));
    //preparing connection option
    MqttConnectOptions connOpt = new MqttConnectOptions();
    connOpt.setCleanSession(true);
    connOpt.setConnectionTimeout(15); //15 seconds
    connOpt.setKeepAliveInterval(30); //30 seconds
    //connect
    activity.appendStatus("Connecting to cloud broker...");
    try {
        client.connect(connOpt, activity,
              new MQTTActionListener(Action.CONNECT_C, this));
    } catch (MqttException e) {
```

```
            e.printStackTrace();
            fail(e.getMessage());
        }
    }

    /*
     * When the client is connected DiscoveryObserver invoke this method
     * for subscribing the device-specific topic.
     * The method is invoked also by chooseGateway method if there are no
     * responses from gateways
     */
    public void subscribeDeviceTopic(int counter) {
        if (counter == 0){
            placesCounter=0;
            activity.appendStatus("Cloud broker connected!");
        }
        if( !(counter < places.size()) ){
            fail(Constants.NO_GATEWAY);
            return;
        }
        else{
            try {
                client.subscribe(topics.get(counter)+"/"+id, 0, activity,
                        new MQTTActionListener(Action.SUBSCRIBE, this));
            } catch (MqttException e) {
                e.printStackTrace();
                fail(Constants.SUBSCRIPTION_ERROR + e.getMessage());
            }
        }
    }

    /*
     * Invoked by DiscoveryObserver when the subscription is done.
     */
    public void publishAnnounce() {
        activity.appendStatus("Device specific topic subscribed. "
                + "(" + topics.get(placesCounter) + "/" + id + ")");
        try {
            client.publish(topics.get(placesCounter), mqttMessage, activity,
                    new MQTTActionListener(Action.ANNOUNCE, this));
        } catch (MqttException e) {
            e.printStackTrace();
            fail(Constants.ANNOUNCE_NOT_PUBLISHED + e.getMessage());
        }
    }


    /*
     * Invoked when the announce message leaved the client. (It is QoS 0, so we cannot be
     * sure that it arrives to the broker).
     */
    public void announcePublished() {
        activity.appendStatus("Announce published on \"" + places.get(placesCounter) + "\" topic.
                Waiting for responses from gateways...");
    }

    /*
     * Method invoked by DiscoveryObserver when the receive info-gateway timeout
     * expires. The collected messages will be passed as parameter.
```

```
 */
public void setInfoGw(List<InfoGatewayContent> infoGwCollected) {
   if(infoGwCollected.size() == 0)
      activity.appendStatus("No gateways registered in that topic.");
   else
      activity.appendStatus("Received " + infoGwCollected.size() + " gateways informations.");
   info=infoGwCollected;
   chooseGateway();
}

/*
 * This method is invoked when the client receives the info-gateway list
 * from DiscoveryObserver.
 */
private void chooseGateway() {
   if(info == null || info.isEmpty())
      subscribeDeviceTopic(placesCounter += 1);
   else{
      nearest = info.get(0);
      for(InfoGatewayContent m : info){
         if (m.getDistance()<nearest.getDistance())
            nearest=m;
      }
      info.remove(nearest);
      if(client != null)
         disconnectCloud();
      else
         connectGatewayWiFiNetwork();
   }
}

/*
 * Once the devices receives at least an info-gateway message the client disconnects the broker
 */
private void disconnectCloud() {
   try {
      client.disconnect(activity, new MQTTActionListener(Action.DISCONNECT_C, this));
      client = null;
   } catch (MqttException e) {
      e.printStackTrace();
   }
   activity.appendStatus("Cloud broker disconnected.");
}

/*
 * Invoked when the Cloud is disconnected by DiscoveryObserver
 */
public void connectGatewayWiFiNetwork() {
   activity.appendStatus("Connecting to " + nearest.getDeviceId() + " WiFi Network...");
   wifiManager.connect(nearest.getSSID(), nearest.getPassphrase(), nearest.getSecurity());
}

/*
 * When the client is connected to the gateway WiFi network DiscoveryObserver
 * invokes this method to connect the local broker.
 */
public void connectLocalBroker() {
   activity.appendStatus("Connected to the device's WiFi network");
   client = new MqttAndroidClient(activity, nearest.getBrokerURI(), id);
```

```
        client.setCallback(new MQTTCallBack(this));
        //preparing connection option
        MqttConnectOptions connOpt = new MqttConnectOptions();
        connOpt.setCleanSession(true);
        connOpt.setConnectionTimeout(2000);
        connOpt.setWill(Constants.WILL_TOPIC, id.getBytes(), 0, false);
        connOpt.setKeepAliveInterval(500);
        try {
            client.connect(connOpt, activity,
                  new MQTTActionListener(Action.CONNECT_GW, this));
        } catch (MqttException e) {
            e.printStackTrace();
            fail(e.getMessage());
        }
    }

    /*
     * This method started when the client is connected to the gateway's local broker.
     */
    public void doJob(){
        activity.appendStatus("Gateway's broker connected. Sending sensor data.");
        job = new SendSensorDataTask(client,
              (SensorManager)activity.getSystemService(Activity.SENSOR_SERVICE),
              nearest.getSensorsRequested());
        scheduleFuture = exec.scheduleAtFixedRate(job, 2, sensorUpdateInterval, TimeUnit.SECONDS);
        activity.enableUI(true);
    }

    /*
     * If some problem occurs this method will be invoked. If the problem is relative to WiFi
        connection
     * or announce publication, first we try to choose another gateway, then if there are no more
        gateways
     * we stop the process
     */
    public void fail(String msg) {
        activity.appendStatus(msg);
        if (msg.equals(Constants.WIFI_CONNECTION_ERROR)){
            chooseGateway();
            return;
        }
        if(client!= null){
            try {
                client.disconnect();
            } catch (MqttException e) {
                e.printStackTrace();
            }
            client = null;
        }
        wifiManager.disconnect();
        activity.enableUI(false);
    }

    /*
     * This method is invoked by DiscoveryActivity when the user clicks the
     * disconnection button.
     */
    public void stop(){
        stopJob();
```

```
        if(client!= null){
            try {
                client.disconnect();
            } catch (MqttException e) {
                e.printStackTrace();
            }
            client = null;
        }
        wifiManager.disconnect();
        activity.appendStatus("Terminated.");
        activity.enableUI(false);
    }

    /*
     * This method stops the application-specific job.
     */
    public void stopJob() {
        if(scheduleFuture != null)
            scheduleFuture.cancel(true);
        scheduleFuture = null;
    }

    public boolean jobIsRunning() {
        return scheduleFuture!=null;
    }

    public DiscoveryActivity getDiscoveryActivity() {
        return activity;
    }

    public String getAvailableSensorsAsString(){
        String sensors = "";
        for (String s : availableSensors){
            sensors += s;
            if (availableSensors.indexOf(s)!=availableSensors.size()-1)
                sensors += ", ";
        }
        return sensors;
    }

    public DiscoveryObserver getDiscoveryObserver() {
        return discoveryObserver;
    }
}
```

Bibliografia

[1] Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[2] Eclipse IoT. Kura. http://eclipse.org/kura.

[3] Internet of Things Council. http://www.iotcouncil.org.

[4] Dave Evans. The internet of things the internet of things - how the next evolution of the internet is changing everything. http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf, April 2011.

[5] Gartner. Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015. http://www.gartner.com/newsroom/id/3165317, November 2015.

[6] Internet of Things World Forum. Global iot deployment. http://www.iotwf.com/iotwf2015/deployment-map.

[7] Mark O'Neill. Internet of things - top ten concerns. http://www.scmagazineuk.com/internet-of-things--top-ten-concerns/article/339217/.

[8] Postscapes Tracking the Internet of Things. Iot protocols. http://postscapes.com/internet-of-things-protocols.

[9] Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1548–1557 vol.3, 2001.

[10] Apache. Apache River (Jini). https://river.apache.org/.

[11] Open Interconnect Consortium. Universal Plug and Play (UPnP). http://www.upnp.org/.

[12] Internet Engineering Task Force. Service Location Protocol (SLP). https://datatracker.ietf.org/wg/svrloc/documents/.

[13] Sumi Helal. Standards for service discovery and delivery. *Pervasive Computing, IEEE*, 1(3):95–100, July 2002.

[14] Apache. Jini discovery & join specification. `https://river.apache.org/doc/specs/html/discovery-spec.html`.

[15] Soumya Kanti Datta, Rui Pedro Ferreira Da Costa, and Christian Bonnet. Resource discovery in internet of things: Current trends and future standardization aspects. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 542–547, Dec 2015.

[16] Federica Paganelli and David Parlanti. A dht-based discovery service for the internet of things. *Journal of Computer Networks and Communications*, 2012, 2012.

[17] Meirong Liu, Teemu Leppanen, Erkki Harjula, Zhonghong Ou, Mika Ylianttila, and Timo Ojala. Distributed resource discovery in the machine-to-machine applications. In *Mobile Ad-Hoc and Sensor Systems (MASS), 2013 IEEE 10th International Conference on*, pages 411–412. IEEE, 2013.

[18] Simone Cirani, Luca Davoli, Giorgio Ferrari, Rémy Léone, Paolo Medagliani, Marco Picone, and Luca Veltri. A scalable and self-configuring architecture for service discovery in the internet of things. *Internet of Things Journal, IEEE*, 1(5):508–521, 2014.

[19] Antonio J Jara, Pablo Lopez, David Fernandez, Jose F Castillo, Miguel A Zamora, and Antonio F Skarmeta. Mobile digcovery: discovering and interacting with the world through the internet of things. *Personal and ubiquitous computing*, 18(2):323–338, 2014.

[20] Giancarlo Fortino, Marco Lackovic, Wilma Russo, and Paolo Trunfio. A discovery service for smart objects over an agent-based middleware. In *Internet and Distributed Computing Systems*, pages 281–293. Springer, 2013.

[21] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[22] Isam Ishaq, Jeroen Hoebeke, Jen Rossey, Eli De Poorter, Ingrid Moerman, and Piet Demeester. Facilitating sensor deployment, discovery and resource access using embedded web services. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 717–724. IEEE, 2012.

[23] Ming Zhou and Yan Ma. A web service discovery computational method for iot system. In *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, volume 3, pages 1009–1012. IEEE, 2012.

[24] Sarfraz Alam and Josef Noll. A semantic enhanced service proxy framework for internet of things. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 488–495. IEEE, 2010.

[25] Chang Ho Yun, Yong Woo Lee, and Hae Sun Jung. An evaluation of semantic service discovery of a u-city middleware. In *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, volume 1, pages 600–603. IEEE, 2010.

[26] Simon Mayer and Dominique Guinard. An extensible discovery service for smart things. In *Proceedings of the Second International Workshop on Web of Things*, page 7. ACM, 2011.

[27] Zhiming Ding, Xu Gao, Limin Guo, and Qi Yang. A hybrid search engine framework for the internet of things based on spatial-temporal, value-based, and keyword-based conditions. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 17–25. IEEE, 2012.

[28] Li Minbo, Zhu Zhu, and Chen Guangyu. Information service system of agriculture iot. *AUTOMATIKA: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije*, 54(4):415–426, 2014.

[29] J. Mitsugi, Y. Sato, M. Ozawa, and S. Suzuki. An integrated device and service discovery with upnp and ons to facilitate the composition of smart home applications. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 400–404, March 2014.

[30] Andrew Banks and Rahul Gupta. Mqtt version 3.1. 1. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`, October 2014.

[31] Andy Stanford-Clark and Hong Linh Truong. Mqtt for sensor networks (mqtt-s) protocol specification. `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`, 2008.

[32] Eurotech. Kura: Un gateway Java per l'Internet of Things. `http://www.eurotech.com/it/sala+stampa/news/?700`.

[33] Eclipse IoT. Kura documentation: Introduction. `http://eclipse.github.io/kura/doc/intro.html`.

[34] Android. Android SensorManager.getDelay(int rate). `http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/5.1.1_r1/android/hardware/SensorManager.java#SensorManager.getDelay%28int%29`.