Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Studio e implementazione di un Servient Android per W3C Web of Things

Relatore:
Dott.
FEDERICO MONTORI

Presentata da:
SILVANO CARRADORI

Correlatore:
Dott.
LUCA SCIULLO

Sessione II
Anno Accademico 2020/2021

Abstract

La rapida crescita dell'*Internet of Things* ha portato alla creazione di una vastità di tecnologie e dispositivi differenti non sempre comunicanti fra di loro. Sulla base di questo problema nasce l'idea del *Web of Things*, che utilizza i principi del web per far convivere diverse tecnologie già funzionanti e rodate, senza crearne di nuove. Lo standard promosso dal W3C nel 2014 ne descrive l'architettura e le componenti. Tuttavia, fra le decine di implementazioni ufficialmente riconosciute dal W3C, nessuna è dedicata al mondo Android. Questa tesi consiste nella realizzazione di un WoT *Servient* per dispositivi Android conforme alle specifiche definite dal W3C. Fra le principali funzioni, il *Servient* permette di consumare *Things*, importando direttamente le *Thing Descriptions* o indicando un riferimento, e permette di interagire con esse controllando un determinato cambiamento di stato o invocando delle azioni. È possibile esporre delle *Things*, con una dedizione particolare all'esposizione di risorse e sensori interni del dispositivo utilizzato con relative *Interaction Affordances*, il tutto utilizzando i quattro *Binding Templates* implementati (HTTP, WebSocket, CoAP, MQTT).

Indice

Abstract **2**

1 Introduzione **13**

2 Stato dell'arte **17**

2.1 Internet of Things 17

2.1.1 Architettura 18

2.1.2 Dispositivi 19

2.1.3 Connessione dei dispositivi alla rete 20

2.1.4 Protocolli 21

2.2 Web of Things 23

2.2.1 Nascita del Web of Things 23

2.2.2 Web of Things prima del W3C 24

2.2.3 W3C's Web of Things 26

3 AndroidWotServient **31**

3.1 Obiettivo 31

3.2 Requisiti 32

3.3 Architettura 32

3.3.1 App Android 33

3.3.2 Servient Core 35

4 Implementazione **39**

4.1 Struttura e dettagli del codice 39

4.1.1 Modulo "wot-servient" 40
4.1.2 Modulo "app" 43
4.2 Tecnologie utilizzate 46
4.2.1 Java 46
4.2.2 JSON e Libreria Jackson 48
4.2.3 Esposizione Sensori 48
4.2.4 Settings 49
4.2.5 Database Room 50
4.2.6 Librerie Protocolli 50

5 Validazione 53
5.1 Caso d'uso e ambiti di utilizzo 53
5.2 Implementazione 55

6 Conclusioni 61
6.1 Limiti del progetto e possibili sviluppi 61

Bibliografia 62

Elenco delle figure

2.1 I due principali modelli architetturali dell'IoT [12] 18
2.2 Indipendent integration [5] 21
2.3 Network integration [5] 22
2.4 Hybrid integration [5] 22
2.5 Architettura Web of Things [3] 25
2.6 Thing description [19] 27
2.7 Binding template [18] 28

3.1 Architettura AndroidWotServient 33

4.1 Struttura del codice 40
4.2 Ways to add TD 45
4.3 Add TD from URL 45
4.4 Add TD from Discovery 45
4.5 ConsumedThing details 45
4.6 Dialog Sensors 47
4.7 Thing Exposed 47
4.8 ExposedThing details 47
4.9 Thing Property URLs 47

5.1 Scenario caso d'uso 54
5.2 Location trigger 56
5.3 Sensor trigger 56
5.4 Plain Text trigger 56

5.5 Scelta delle Azioni 56
5.6 Scenario caso d'uso 57

Capitolo 1

Introduzione

L'*Internet of Things*, ramo dell'informatica nato nel 1999 [2], ha come concetto di base quello di connettere i sensori presenti negli oggetti (*Things*) alla rete Internet per farli comunicare. I campi di applicazione sono molti a livello domotico (le *Smart Home*), a livello cittadino (*Smart Cities*), a livello industriale, tutti in continua evoluzione grazie alla rapida crescita dei dispositivi nel mondo [15], ai sensori sempre più presenti in questi dispositivi e all'evoluzione delle reti.
La prima grande evoluzione del concetto di *Internet of Things* fu quella del *Web of Things*, necessaria a causa dell'elevata frammentazione dei protocolli di comunicazione e strutture dati utilizzate che metteva un freno all'interoperabilità fra le diverse piattaforme IoT. Basato sui principali concetti del web, in particolare sul paradigma architetturale REST, il *Web of Things* diventò presto uno standard che, piuttosto di creare nuove tecnologie che avrebbero ulteriormente frammentato questo settore, definisce delle interfacce che permettono di utilizzare le tecnologie web già esistenti e consolidate [17]. Lo standard fu promosso dal W3C (*World Wide Web Consortium*) che, a partire dal 2014, ne accelerò lo sviluppo [13].
Scopo di questa tesi è lo sviluppo di un WoT *Servient* per dispositivi Android, seguendo le indicazioni fornite dal W3C [17]. Il *Servient* ha la possibilità sia di esporre i sensori interni del dispositivo, sia di consumare *Things* esterne.

L'elaborato è strutturato nel modo seguente. Nel capitolo 2 vengono illustrati l'attuale stato dell'arte dell'*Internet of Things*, la nascita del primo concetto di *Web of Things*, fino alla creazione dello standard del W3C con una panoramica su dispositivi e protocolli utilizzati. Nel capitolo 3 si descrive il WoT *Servient* per Android sviluppato, analizzando requisiti tecnici e funzionali dell'architettura. Il capitolo 4 analizza il processo di implementazione, le varie tecnologie utilizzate e le scelte progettuali. Infine, nel capitolo 5 il progetto viene messo sul campo mostrando un caso d'uso che è possibile realizzare.

Capitolo 2

Stato dell'arte

2.1 Internet of Things

Con *Internet of Things* ci si riferisce ad un insieme di dispositivi capaci di inviare e ricevere informazioni attraverso Internet e più in generale le reti. Dalla sua nascita nel 1999 quando fu coniato il termine da Kevin Ashton [2], il concetto di IoT ha iniziato a farsi sempre più preponderante grazie, soprattutto, all'incremento considerevole dei dispositivi connessi nel mondo[10].
La nostra società è da sempre stata basata sul concetto di Cosa (*Thing*), il fatto che adesso esse abbiano la possibilità di comunicare fa ampliare a dismisura i campi di applicazione e ha portato ad inserire nel concetto di IoT una serie di settori fra loro differenti. Concetti quali *Smart Home*, *Smart City* hanno, e avranno, nuove possibilità di interfacciarsi creando, quindi, nuovi campi di applicazione[4]. Se Internet è stato, fin dalla sua creazione, basato su un ambiente unico (generalmente la casa o l'ufficio), negli ultimi anni il baricentro si è indubbiamente spostato per strada, nei negozi, o comunque non più rilegato ad un singolo luogo. L'obiettivo è quindi quello di far diventare sempre più Things *web-present* incorporando in esse *web-server* o comunque esponendo la loro *web-presence* in un *web-server*[8], mantenendo la capacità di interconnettere milioni, o miliardi, di dispositivi eterogenei[7].

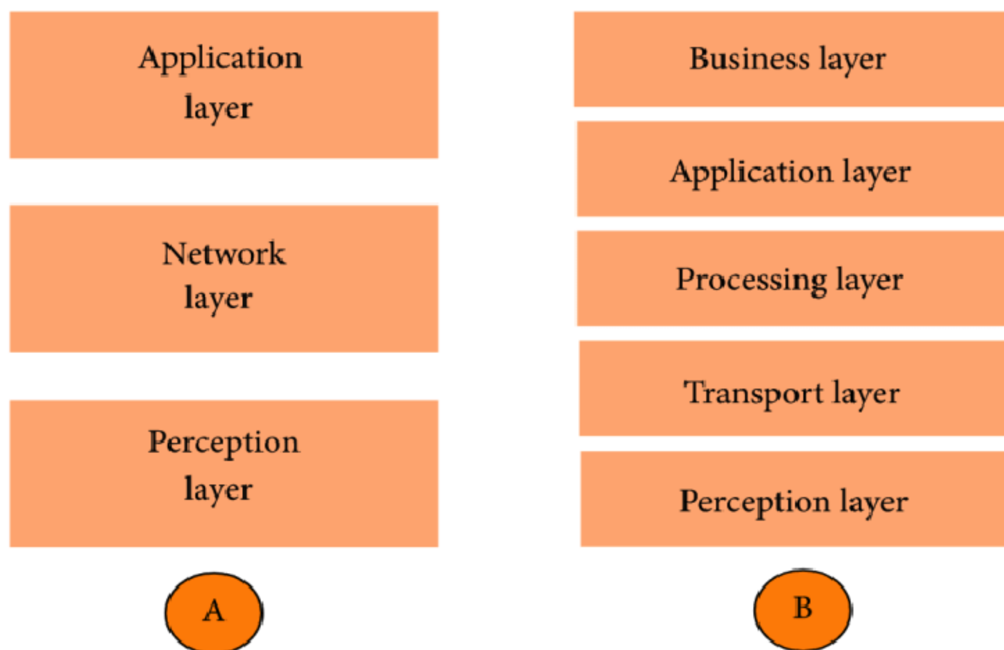

Figura 2.1: I due principali modelli architetturali dell'IoT [12]

2.1.1 Architettura

L'enorme differenza fra i diversi dispositivi che comprendiamo nella famiglia dell'IoT ha fatto nascere il bisogno di un'architettura a strati flessibile. Il modello base scelto è quello a 3 *layers* (Figura 2.1), così composti:

- **Perception Layer** Si occupa di raccogliere, tramite sensori e attuatori, dati come la localizzazione, l'umidità, il movimento, ecc. Successivamente processa questi dati e li passa al *layer* successivo.

- **Network Layer** Questo *layer* è il più complesso, si occupa di trasmettere all'*Application Layer* i dati ricevuti dall'*Objects Layer* utilizzando diverse tecnologie (RFID, Wi-Fi, Bluetooth, Rete cellulare) e protocolli (ZigBee, LoRa, ecc.).

- **Application Layer** Questo *layer* fornisce, a chi li richiede, i dati provenienti dai sensori della *Thing* (arrivati dal *Network Layer*). Può fornire lo stato di una *Thing*, può eseguire un'azione, può scatenare un

evento. E lo fa cercando di fornire servizi di alta qualità per incontrare le esigenze di chi lo richiede (*Smart Home*, trasporti, automazione industriale, Salute, ecc.).

Nonostante molti ricercatori si siano accordati sull'utilizzo del modello a 3 *layers* [9], essa diventò non sufficiente a causa dello sviluppo dell'IoT. Pertanto, venne proposta un'architettura a 5 *layers* (Figura 2.1), che aggiunge il *Business Layer*, il cui scopo è la gestione delle applicazioni IoT e della privacy dell'utente, e il *Processing Layer*, responsabile della memorizzazione e dell'analisi delle informazioni raccolte dal *Perception Layer*.

2.1.2 Dispositivi

Col passare del tempo il parco dispositivi che possiamo definire *Smart* continua ad aumentare, pensiamo ad esempio al concetto di *Smart Home*, che fino a qualche tempo fa era sconosciuto, ad oggi lo viviamo quotidianamente. A supportare questa evoluzione non basta la sola presenza dei dispositivi fisici, ma è necessario uno sviluppo complessivo dell'intero settore e soprattutto delle tecnologie alla base. Da una parte la potenza di calcolo e i sempre più presenti sensori all'interno dei dispositivi di uso comune, dall'altra lo sviluppo delle reti divenute sempre più accessibili, performanti e stabili.
Oggigiorno è di uso comune possedere e interagire con *Things*, che possono essere veri e propri ambienti (*Smart Building*, *Smart Cities*), oggetti di uso comune (elettrodomestici, lampadine, prese elettriche). A volte anche inconsapevolmente, se prendiamo come esempio un comunissimo smartphone di fascia bassa, contiene al suo interno diversi sensori (accelerometro, sensore di prossimità, sensore di luminosità, Localizzazione, ecc.) pronti a diventare parte attiva dell'ecosistema IoT.
I casi d'uso sono molteplici, nell'uso casalingo (*Smart Home*, domotica), industriale (controllo dei macchinari, comunicazione fra essi), dei trasporti (monitoraggio dei parametri dei trasporti urbani), della sanità [1]. Quelli citati

sono solo alcuni dei casi d'uso possibili, il limite dei dispositivi e dei sistemi di comunicazione tende sempre di più a scomparire.

2.1.3 Connessione dei dispositivi alla rete

Aspetto di base dell'IoT è, chiaramente, la connessione dei dispositivi fisici alla rete per poter comunicare con altri sistemi. Nonostante il concetto sia abbastanza semplice, il numero di *Things* ad oggi presenti e la loro eterogeneità rendono l'esecuzione di tale processo molto complesso. Se una stampante possiede risorse stabili (connessione alla corrente, connessione alla rete), lo stesso non possiamo dire per un piccolissimo sensore di temperatura che funziona a batteria e comunica con il protocollo Zigbee. Quest'ultimo infatti deve gestire le poche risorse disponibili nel miglior modo possibile evitando sprechi, sarebbe controproducente collegarlo direttamente alla rete e quindi caricarlo di tutti gli oneri che ne derivano.
Se prendiamo un insieme di computer connessi alla rete possiamo notare che sono trattati ugualmente, le loro funzionalità dipendono da come li utilizziamo. Una *Thing* dell'IoT, invece, è soggetta a diverse condizioni (Consumo di energia, meccanismi di sicurezza adottati, larghezza di banda disponibile) che ne determinano le funzionalità[5]. Per creare un'interoperabilità fra *Things*, o gruppi di esse, si determina, quindi, un sistema simile alle WSN (*Wireless Sensors Network*) integrate in diversi modi.

- **Independent integration (Figura 2.2)** Con questo tipo di integrazione i dispositivi sono connessi, appunto, indipendentemente e direttamente alla rete. Il dispositivo non deve, quindi, passare per un nodo intermedio per connettersi con altri dispositivi. Nonostante questo, fornire un indirizzo IP ad ogni dispositivo potrebbe non essere una scelta ottimale, soprattutto per dispositivi a poca potenza che devono risparmiare risorse.

- **Network based integration (Figura 2.3)** Nell'integrazione basata sulla rete il dispositivo non ha necessità di connettersi ad internet auto-

Figura 2.2: Indipendent integration [5]

nomamente, ma si connette ad un *gateway* che si occuperà di svolgere questo ruolo. Quindi la comunicazione fra due diverse *Things* di due diverse WSN dovrà passare per il nodo padre (*Gateway*) della WSN.

- **Hybrid integration (Figura 2.4)** Questo tipo di integrazione acquisisce e rielabora le caratteristiche migliori delle precedenti.

2.1.4 Protocolli

Per fornire una comunicazione efficiente nei dispositivi con risorse limitate, sono stati introdotti diversi protocolli IoT, ciascuno con le proprie caratteristiche e adatto a particolari esigenze, condizioni di rete e dimensione delle informazioni da scambiare [11]. Fra i principali protocolli abbiamo:

- **Constrained Application Protocol (CoAP)** Progettato per facilitare lo scambio di informazioni *machine- to-machine* (M2M). Fornisce un meccanismo di interazione asincrono basato sul concetto richiesta/risposta, mantenendo i concetti basilari del Web, come gli *URIs* [16].

- **Message Queuing Telemetry Transport (MQTT)** È uno standard open di OASIS (*Organization for the Ad- vancement of Structured Information Standards*) particolarmente utilizzato per gli scambi

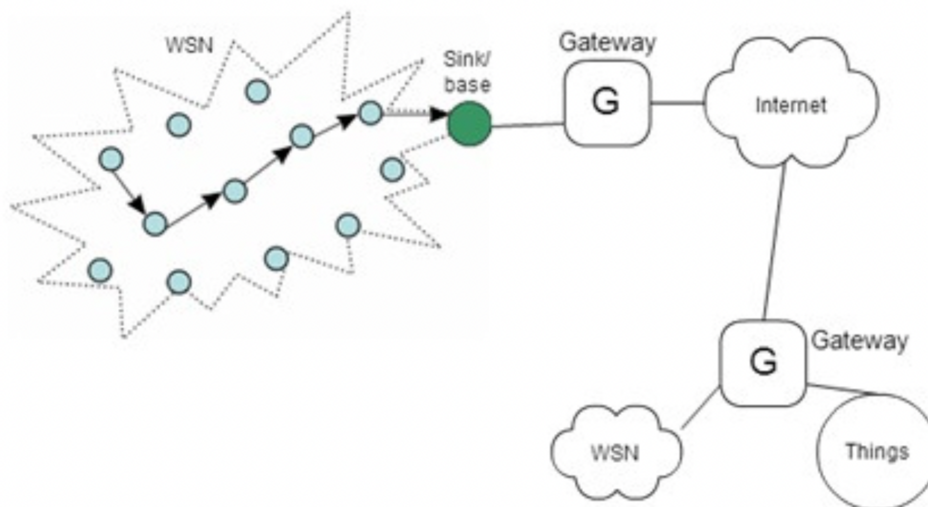

Figura 2.3: Network integration [5]

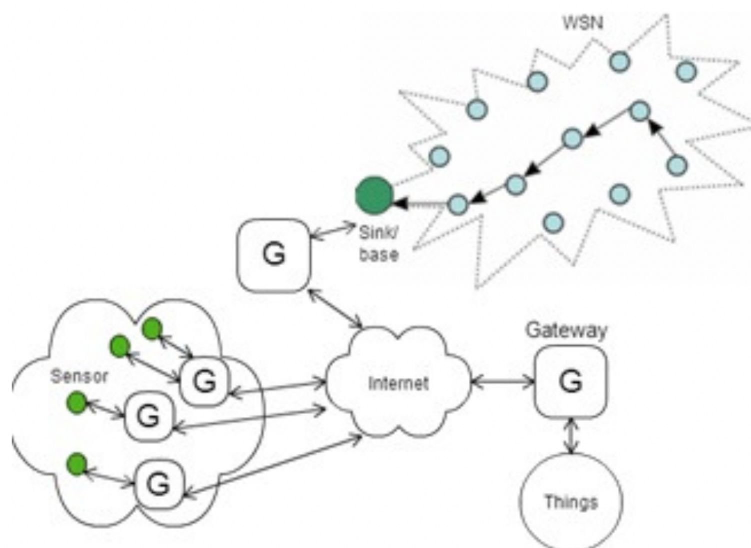

Figura 2.4: Hybrid integration [5]

di informazioni non troppo pesanti, si basa sul meccanismo di pubblicazione/sottoscrizione che consente uno scambio rapido anche negli ambienti wireless con poca banda a disposizione [11].

- **WebSocket** Inizialmente progettato da IETF (*Internet Engineering Task Force*) per fornire un tipo di comunicazione *full-duplex* fra browser web e server, dove entrambe le parti possono comunicare simultaneamente. Nonostante sia stato ideato per sostituire le tecnologie HTTP esistenti, è stato applicato nell'ambito dell'IoT per le comunicazioni in tempo reale [6].

- **Hyper Text Transfer Protocol (HTTP)** Originariamente sviluppato da Tim Berners-Lee, è principalmente un protocollo di messaggistica web. Supporta l'architettura web *RESTful* di richiesta/risposta e, analogamente a CoAP, utilizza l'URI (*Universal Resource Identifier*) come identificatore per le risorse.

2.2 Web of Things

2.2.1 Nascita del Web of Things

Il *Web of Things* è la prima grande evoluzione del concetto di IoT. L'idea è quella di rendere disponibile una *Thing* come una risorsa Web, che è quindi più facile da utilizzare visto che si usano i già diffusi protocolli Web.
Come spiegato nel capitolo precedente, l'interoperabilità fra diversi dispositivi dell'IoT è un requisito fondamentale, con diversi fattori da tenere in considerazione (la sicurezza, la compatibilità, la gestione degli errori, affidabilità), che rendono quindi necessaria una standardizzazione. La creazione di un nuovo protocollo nato esclusivamente per l'IoT che tenesse conto di tutti questi fattori sembrerebbe la scelta più ovvia, tuttavia si è optato per l'utilizzo di tecnologie già esistenti e rodate, bypassando tutte le criticità, già risolte, della creazione di una tecnologia ex novo (in particolare sicurezza e

diffusione). Nel risultate *Web of Things* le risorse sono disponibili tramite meccanismi web, utilizzando gli *URIs* come identificatori. [20]

2.2.2 Web of Things prima del W3C

Si discusse molto di come il *Web of Things* sarebbe potuto diventare uno strumento molto potente per integrare risorse fisiche come risorse web, definendo aspetti tecnici dell'implementazione[20]. Era necessario utilizzare un concetto già molto diffuso nell'architettura Web, quello di API REST (*Representational state transfer*) che consente tramite URI (*Uniform Resource Identifier*) di identificare singolarmente una risorsa e poi fornisce il supporto per un insieme molto diversificato di interazioni (indicati spesso come *basic verbs* perché sono fondamentali per la creazione di una varietà di scenari) [20]:

- **PUT** Crea una nuova risorsa con un nome specificato. Ciò richiede i diritti di accesso adeguati e le informazioni necessarie per la creazione.
- **GET** Restituisce la rappresentazione della risorsa richiesta.
- **POST** È possibile aggiornare una risorsa con nuove informazioni. Un aggiornamento può modificare lo stato di una risorsa esistente ottenendo la risorsa aggiornata utilizzando lo stesso URI.
- **DELETE** Elimina la risorsa specificata cancellandola completamente o rimuovendo il riferimento ad essa.

Insieme ad un'adeguata rappresentazione delle risorse (HTML e XML), questi principi REST, seppur basici e mai messi in pratica in questi termini, furono necessari per massimizzare l'interoperabilità nell'*Internet of Things*.

A livello architetturale (Figura 2.5, il *Web of Things* presenta 4 *layers*:

- **Access** Trasforma una *Thing* connessa (*Networked Thing*) in una *Web Thing*, a questo punto è possibile interfacciarsi con essa tramite i protocolli web.

- **Find** Espone la Thing rendendola rintracciabile utilizzando le rappresentazioni standard del web.
- **Share** Gestisce i dati trasmessi su Internet garantendo sicurezza e celerità.
- **Compose** Consente l'interoperabilità fra la nuova risorsa web creata e tutte le altre.

È presente anche un quinto *layer*, ma non viene di solito indicato come tale in quanto è solamente il metodo di connessione delle *Things* ad Internet.

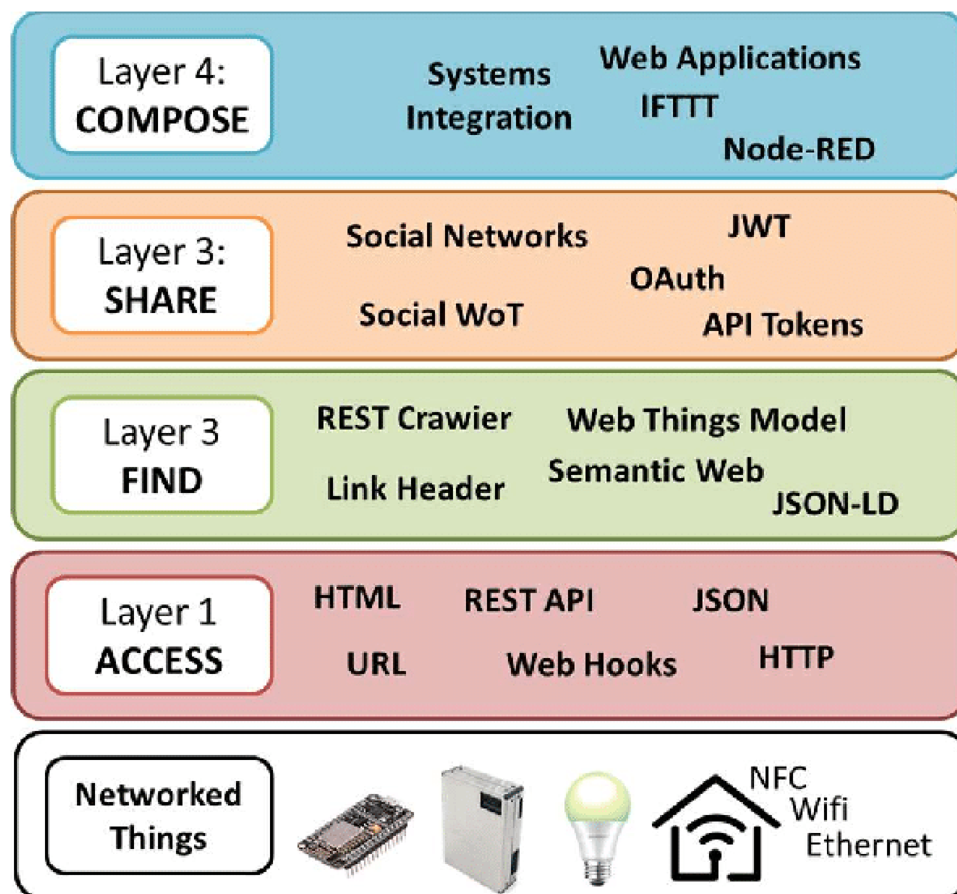

Figura 2.5: Architettura Web of Things [3]

2.2.3 W3C's Web of Things

Obiettivo del W3C (*World Wide Web Consortium*), organizzazione che si occupa di progettare standard, è cercare di standardizzare il concetto di WoT, all'epoca c'erano molte versioni e proposte differenti. "The W3C WoT architecture is designed to describe what exists rather than to prescribe what to implement." [17].
Come detto, alla base c'è il concetto di *Thing*, che è un'astrazione di un'entità fisica o virtuale descritta tramite una *Thing Description* [19], uno dei due *building block* necessari all'implementazione dell'architettura del W3C's WoT [17]. Ogni *Thing* appartenente al W3C's WoT deve avere almeno una Thing Description che, tramite un formato standardizzato processato sotto forma di JSON, fornisce le informazioni in modo che siano leggibili dall'uomo e comprensibili per la macchina. Una Thing Description (Figura 2.6) comprende, oltre ai metadata descrittivi (ID, nome, descrizione, ecc.) anche quelli relativi all'interazione (*Interaction Affordance*), alla sicurezza (*Security Metadata*) e alla comunicazione (*Protocol Binding*).

Interaction Affordance

Le *Interaction Affordances* costituiscono la parte operativa di una *Thing Description*, infatti sono dei metadata che descrivono come un utilizzatore può interagire con la *Thing*. Esistono sostanzialmente tre tipi di *Affordance* (Proprietà, Eventi e Azioni), anche se in aggiunta c'è anche la *Navigation Affordance* sviluppata tramite i *Web Link*.

- **Proprietà** Una proprietà è una *Interaction Affordance* che espone lo stato di una *Thing*. Opzionalmente possono fungere anche da risultato di qualche computazione o da parametri di configurazione, ma, dato che il tipo di utilizzo non viene scelto *by design*, si possono riassumere nelle forme *Read-Only* e *Read-Write*. Oltre a queste informazioni, sono

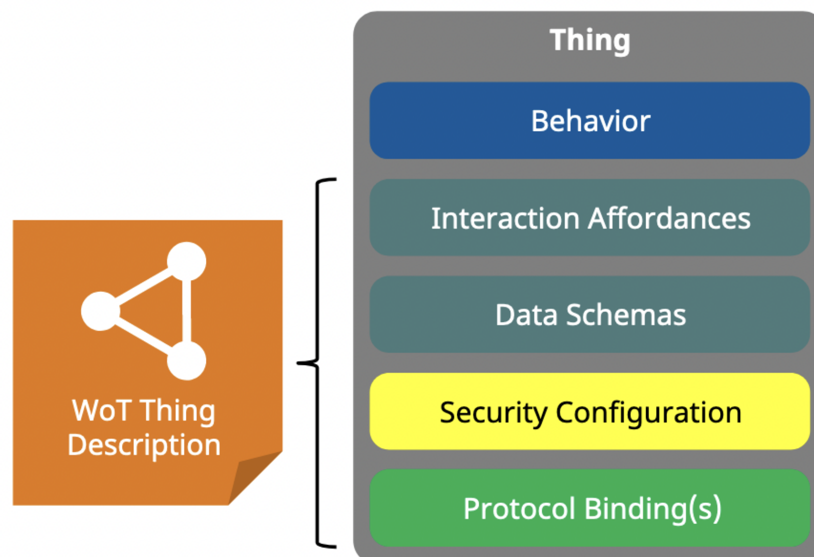

Figura 2.6: Thing description [19]

specificati nome (necessario), ID (necessario), data-schema (opzionale) e tipo di dato (opzionale).

- **Azioni** Sono funzioni che possono cambiare lo stato interno di una *Thing*, eseguire azioni su altre *Things* (anche contemporaneamente) o, comunque, eseguire funzioni non direttamente collegate all'ambiente della *Thing*. Così come le proprietà, anche le azioni sono composte da informazioni quali nome, ID, ecc.

- **Eventi** Gli eventi sono cambiamenti di stato che avvengono in modo asincrono dalla *Thing* al *Consumer*, gli eventi quindi non comunicano stati.

Binding Templates [18]

Altro elemento costitutivo del W3C's WoT, insieme alle *Thing Descriptions*, sono i *binding templates* (Figura 2.7). Infatti il *Web of Things* na-

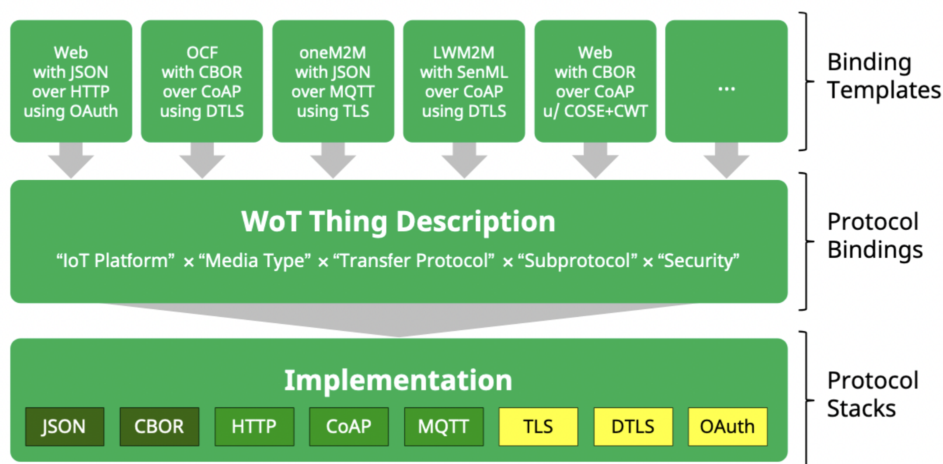

Figura 2.7: Binding template [18]

sce proprio dall'esigenza di abilitare l'interoperabilità fra diverse piattaforme IoT, dal momento che non esiste un protocollo appropriato per tutti i contesti. Lo scopo è, quindi, quello di creare un template che descrive gli elementi necessari e opzionali per implementare un sistema di comunicazione (tipicamente formato da protocollo, formato dati e sicurezza). I *binding templates* sono creati una sola volta per ogni piattaforma IoT e possono essere riutilizzati in tutte le *Thing Descriptions* di quella piattaforma. Il *Consumer* che andrà poi ad utilizzare una determinata *Thing Description* si occuperà di implementare i *Protocol Binding* corrispondenti.

Gli elementi presenti in un *Binding Template* sono:

- **Piattaforma IoT** Vengono introdotte modifiche proprietarie all'*Application Layer* (opzioni per CoAP o HTTP *headers* specifici).

- **Tipo di supporto** Viene specificato il tipo di supporto utilizzato per lo scambio dati. Si possono anche utilizzare i *Media Type Parameters* per specificare ulteriormente.

- **Protocollo di trasferimento** Il *Transfer Protocol* specifica il protocollo utilizzato per la comunicazione. Tipicamente viene utilizzato lo

schema dell'URI per identificarlo.

- **Subprotocol** Utilizzato per specificare le estensioni di alcuni protocolli. I *Forms* possono contenere le informazioni necessarie all'identificazione.
- **Sicurezza** Sono i meccanismi di sicurezza che possono essere applicati ai differenti *layers* dello *stack* di comunicazione.

Capitolo 3

AndroidWotServient

Questo capitolo tratta del progetto AndroidWotServient, un *Servient* WoT per dispositivi Android basato sulle ultime specifiche del W3C. Non esiste allo stato attuale [21] un'implementazione riconosciuta per Android. Nelle seguenti sezioni vengono definiti obiettivi, requisiti tecnici e funzionali, trattando dei principali componenti che costituiscono l'architettura.

3.1 Obiettivo

Il progetto sviluppato è un WoT *Servient* per dispositivi Android, sviluppato in Java, conforme all'ultimo aggiornamento delle specifiche del W3C, datato 9 Aprile 2020[17]. Attualmente circa una decina di implementazioni sono state ufficialmente riconosciute dal W3C, ma nessuna per Android. Questo progetto trae ispirazione da SANE WoT Servient [14] sviluppata in Java 11, che a sua volta è fortemente ispirata a node-wot [4] sviluppata in NodeJs. In particolare, è stata presa in esame la struttura interna del *core* e, nonostante non sia compatibile con Android e la differenza di versione Java (Java 8 vs Java 11), si è cercato di trasportare, ove possibile, alcuni componenti adattandoli.
Oltre alle funzionalità di base di un WoT *Servient*, sono state aggiunte l'im-

portazione da **file** di *Thing Description* e di configurazioni per il *protocol binding*, oltre al *Discovery* di *Thing Description* con MQTT.

3.2 Requisiti

Di seguito sono elencati i requisiti che il progetto fornisce.

- **Esposizione sensori interni del dispositivo Android** Deve essere possibile esporre i sensori interni, inclusa la Localizzazione, del dispositivo Android utilizzato tramite la creazione di una *Thing Description.*
- **Personalizzazione Thing Description** L'utente deve poter indicare Nome e ID della *Thing Description*, oltre alla scelta dei sensori da utilizzare. Deve anche poter togliere l'esposizione della *Thing* in questione.
- **Scelta e modifica Binding Templates** L'utente deve poter scegliere quali *Binding Templates* utilizzare, fra quelli implementati, per l'esposizione e il consumo di una *Thing.*
- **Importare una Thing Description per consumare la Thing** Per il consumo della *Thing* si deve poter importare una *Thing Description* da *file*, da URL o tramite *Discovery*
- **Possibilità di consumare una Thing tramite Interaction Affordances** Dopo l'importazione della *Thing Description* deve essere possibile visualizzare le proprietà e invocare le azioni.

3.3 Architettura

In questa sezione verrà analizzata l'architettura del progetto.
Il progetto sostanzialmente si divide in due macro blocchi, il *core* del WoT *Servient* e l'applicazione Android.
Nel *core* sono stati implementati i due *Building Blocks* necessari per il suo

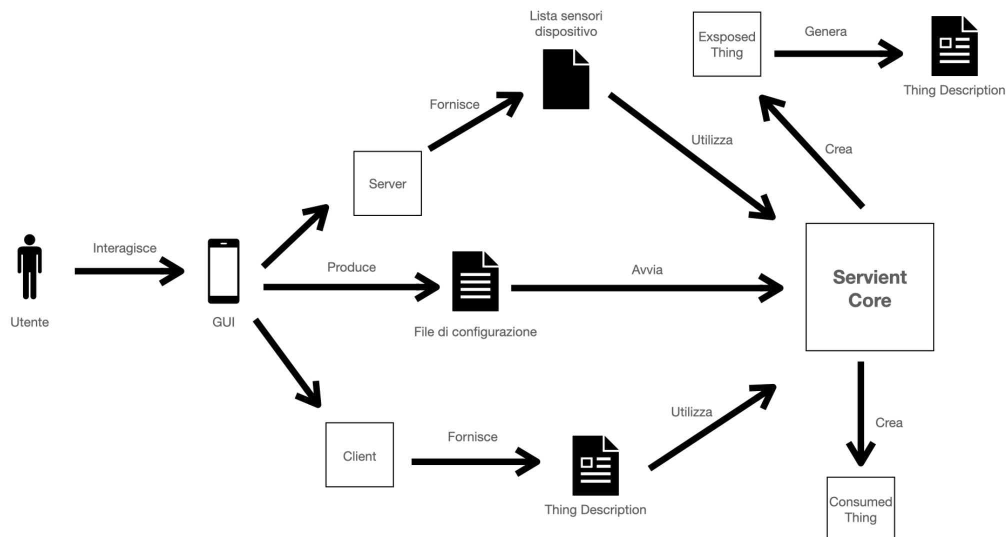

Figura 3.1: Architettura AndroidWotServient

funzionamento (*Thing Description* e *Binding Template*) oltre alle interfacce necessarie alla creazione di tutte le articolazioni di una *Thing* (*Consumed* ed *Exposed*). L'architettura del progetto è stata pensata per implementare agilmente *Binding Templates* senza dover modificare il *core*. A questo proposito, come *Binding Templates* sono stati implementati HTTP, WebSocket, CoAP e MQTT.
Per quanto riguarda l'applicazione Android, essa utilizza ciò che è stato implementato nel *core* per creare effettivamente il WoT *Servient* e fornire all'utente un'interfaccia grafica (GUI) che consente di accedere a tutte le funzionalità.

3.3.1 App Android

L'utente interagisce con il *Servient* esclusivamente tramite Android GUI:

- tramite il **Client**, la cui principale funzione è quella di consumare una o più *Things* esposte già su qualche server;

- tramite il **Server**, che consente di far diventare il dispositivo Android un web server esponendo i suoi sensori interni.

Client

Requisito fondamentale per poter utilizzare il Client è avere una *Thing Description*, o un riferimento ad essa. Quando non sono presenti *Consumed Things* (un particolare tipo di *Thing* che non contiene elementi atti all'esposizione) l'unica azione possibile è quella per aggiungere una *Thing Description*. Questo è possibile farlo in tre modalità distinte:

- **da file**, dove l'utente può scegliere un *file* in JSON contenente la *Thing Description*;
- **da URI**, dove si può inserire un URL (HTTP o CoAP) che punta ad una *Thing Description*;
- **dal Discovery**) Viene avviato il *Discovery* (MQTT) delle *Thing Descriptions*, fra quelle trovate si può selezionare quale importare.

Dopo aver importato la *Thing Description*, viene generata la corrispondente *Consumed Thing* (Figura 3.1). Il tipo di interazione dipende molto dagli elementi presenti nella *Thing Description*, in linea di massima è possibile visualizzare i principali dettagli, eliminare la *Consumed Thing* in questione, visualizzare proprietà ed interagire con le azioni.

- Le proprietà si suddividono principalmente in due categorie, *observable* e *non-observable*. Per quelle *observable*, il valore della proprietà si aggiorna autonomamente quando necessario, mentre quelle *non-observable*, l'aggiornamento deve essere eseguito manualmente dall'utente.
- Le azioni possono essere invocate tramite i protocolli specificati nella *Thing Description*. Possono modificare valori delle stesse proprietà oppure scatenare azioni esterne alla *Thing* attuale.

Server

Il compito principale del Server è quello di generare le *Exposed Things*, cioè di esporre i sensori interni del dispositivo generando le *Thing Descriptions* corrispondenti. Sono richieste alcune informazioni per procedere alla creazione della *Thing Description*:

- **Nome della Thing** È possibile inserire manualmente il nome oppure scegliere la generazione automatica. In quest'ultimo caso verrà dato il nome di default "Android Device";
- **ID della Thing** È possibile generarlo automaticamente oppure manualmente inserendo del testo. L'ID specificato avrà poi ripercussioni sugli URI generati dai protocolli implementati.
- **Sensori da esporre** Il sistema rileva i sensori presenti nel dispositivo e fa scegliere all'utente quali esporre. Fra questi sono presenti anche risorse che non sono propriamente dei sensori (Localizzazione, Batteria, ecc.).

Una volta inserite tutte le informazioni richieste, sarà invocato il *Servient Core* che genererà la corrispondente *Exposed Thing* (Figura 3.1) basandosi sui *Binding Templates* implementati.

3.3.2 Servient Core

È il *Core* del programma che svolge differenti funzioni a seconda delle richieste. Richiede all'avvio un file di configurazione dove sono presenti i dati di configurazione per tutti i protocolli. I suoi compiti sono i seguenti:

- avviare/terminare un'istanza del *Servient*
- gestire il *Fetch/Discovery* delle *Thing Description*
- consumare una *Thing Description* creando così una *Consumed Thing*

- generazione *Exposed Thing* prendendo in *input* una *Thing* (generata tramite il *Thing Builder*)
- gestire l'avvicendamento dei *Binding Templates* implementati, scegliendo quello più opportuno per l'operazione che si sta svolgendo
- fornire un'interfaccia per la creazione di futuri *Binding Templates* (sia Client che Server)

Binding Templates

I *Binding Templates* di questo progetto sono stati implementati come *packages* indipendenti del *core* per poter essere facilmente rimossi o aggiunti. Ne sono stati implementati quattro (HTTP, WebSocket, CoAP e MQTT). Tramite la sezione *Settings* dell'app è possibile aggiungere da *file* e gestire tutte le configurazioni dei protocolli sia per Client che per Server, oltre alla disattivazione delle notifiche. Per il Client è possibile attivare/disattivare uno per uno i protocolli, mentre per il server si possono indicare, per ogni protocollo, tutti i dati di configurazione. Per HTTP, WebSocket e CoAP questi dati di configurazione sono la porta e l'*host*, per MQTT invece è possibile indicare anche indirizzo, *clientId*, *username* e *password* del *broker* di riferimento e il nome del *topic* dove saranno inserite tutte le *Thing Descriptions* (quest'ultimo dato è necessario per la funzione *Discovery*).

Capitolo 4

Implementazione

Per la realizzazione del progetto è stato necessario utilizzare diverse tecnologie e componenti. Questo è dovuto all'eterogeneità fra le varie parti del programma, in particolare il *binding* dei protocolli. Di seguito saranno descritte la struttura del codice e le tecnologie utilizzate. Il progetto è *open source* ed è possibile raggiungere la *repository* tramite questo *link*: `https://github.com/UniBO-PRISMLab/android-wot-servient`

4.1 Struttura e dettagli del codice

Come spiegato nel precedente capitolo, il progetto si divide sostanzialmente in due moduli, il *Servient Core* e l'applicazione Android. Nel *Core* (modulo denominato "wot-servient") è presente l'implementazione necessaria a far funzionare un *Servient* per il *Web of Things*, nel modulo "app" invece è presente la vera e propria applicazione Android che sfrutta ciò che è stato creato nel *Core* per creare effettivamente il WoT *Servient* e fornire all'utente un'interfaccia grafica (GUI) che consente di accedere a tutte le funzionalità. Data la vastità del progetto non è possibile descrivere la struttura di tutti i componenti presenti, tuttavia si può specificare come gran parte di essi fungano da *utilities* per funzionalità più grandi.

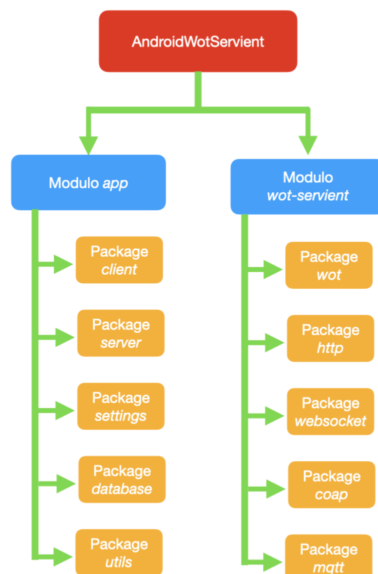

Figura 4.1: Struttura del codice

4.1.1 Modulo "wot-servient"

In questo modulo è presente il *package* di default "wot", che contiene il nucleo del progetto, insieme a un package per ogni Binding Template implementato.

Package "wot"

Contiene interfacce (e implementazioni) per le macro funzionalità del progetto, oltre a quelle necessarie al binding dei protocolli. Fra queste, abbiamo:

- **Thing e Interaction Affordances** Sono presenti le tre classi per la definizione di tutti i possibili stati che una *Thing* può assumere.
  - La classe *Thing* contiene tutte le variabili necessarie al contenimento delle informazioni quali ID, nome, descrizioni, *forms*, azioni, proprietà, ecc. Ad ognuna di esse è stata aggiunta una *an-*

notation tramite la libreria Jackson (Sezione 4.2.2) necessarie poi convertire una *Thing Description* in *Thing*.

- La classe *ConsumedThing* è una sottoclasse di *Thing*. Presenta funzioni come getProperties() (per lettura delle proprietà) e getClient() (per prendere il *ProtocolClient* predisposto). Parallela alla classe *ConsumedThing* ci sono le tre implementazioni *Consumed* per le *Interaction Affordances* (*ConsumedProperty*, *ConsumedAction* e *ConsumedEvent*).
- Anche *ExposedThing* è una sottoclasse di *Thing*, in essa sono presenti metodi per aggiunta/rimozione di una proprietà/azione e per la loro esposizione. Sono presenti anche qua le relative *Interaction Affordances* (*ExposedProperty*, *ExposedAction*, *ExposedEvent*).

- **Interfaccia WoT** L'interfaccia WoT fornisce i metodi per consumare, esporre, fare il *fetch* e il *discover* delle *Things*. È implementata dalla classe *DefaultWot* che possiede le seguenti funzioni:

  - **discover()** Avvia il *Discovery* che ritornerà tutte le *Things* disponibili;
  - **produce()** Prende in *input* una *Thing* e ritorna una *Exposed Thing*;
  - **consume()** Prende in *input* una *Thing* e ritorna una *Consumed Thing*;
  - **fetch()** Prende in *input* un URI che rimanda ad una *Thing Description* e ritorna la *Thing* corrispondente;
  - **destroy()** Spegne il *Servient* e smette di esporre tutte le *Things*;

- **Classe Servient** Questa è la classe che svolge il lavoro effettivo. Infatti la stessa classe *DefaultWot* non fa altro che chiamare tutti i metodi implementati qua. Di seguito una panoramica dei principali metodi:

- **start()** Cerca tutti i protocolli, Client e Server, che il programma supporta e li inizializza, rendendo così pronto il *Servient* ad agire.
- **clientOnly()** Una variazione della funzione *start()* dove vengono avviati solo i protocolli Client.
- **serverOnly()** Una variazione della funzione *start()* dove vengono avviati solo i protocolli Server.
- **expose()** Tutti i *ProtocolServer* inizializzati espongono la *Thing* che corrisponde all'*id* passato come parametro. Prima di essere esposta, una *Thing* deve essere aggiunta al *Servient* tramite la funzione *addThing()*.
- **destroy()** Tutti i *ProtocolServer* inizializzati eseguono il *destroy* della *Thing* che corrisponde all'*id* passato come parametro. Dopo non sarà più possibile interagire con essa
- **shutdown()** Fa il *destroy* di tutti i protocolli avviati.

Protocol Binding

Le interfacce necessarie al *binding* dei protocolli sono state strutturate in modo tale da rendere il più possibile semplice l'aggiunta di un nuovo protocollo senza dover modificare il *Core* del progetto. Ne sono presenti due, una per il *binding* dei protocolli Server e una per i protocolli Client.

- **Interfaccia ProtocolClient** Definisce come bisogna interagire con una *Thing* tramite uno specifico protocollo (HTTP, CoAP, WebSocket, MQTT). Fra le principali funzioni abbiamo:

  - **readResource()** Legge la risorsa definita nel *form* passato come parametro. Può essere una *Thing Description* o una *Thing Property*;
  - **writeResource()** Scrive il contenuto passato come parametro nella risorsa definita nel parametro *form*, per esempio una *Thing Property*;

- **invokeResource()** Invoca la risorsa definita nel *form*, ad esempio una *Thing Action*;
- **observeResource()** Crea un *Observable* per la risorsa passata come parametro, ad esempio una *Thing Property* di tipo *observable*;
- **discover()** Avvia il processo di *Discovery*;

- **Interfaccia ProtocolServer** Definisce come esporre una *Thing* affinché si possa interagire con essa tramite il protocollo in questione. È un'interfaccia molto semplice, poiché contiene solamente le funzioni di accensione/spegnimento per un protocollo e per *expose*/*destroy* di una *Thing*;

4.1.2 Modulo "app"

La prima scelta fatta per quanto riguarda la struttura del codice è stata la suddivisione in due *packages* della componente Client e di quella Server. Il *Core* permette di istanziare il *Servient* con le funzioni Client e Server contemporaneamente, tuttavia si è deciso di istanziarli separatamente (uno solo Client e uno solo Server). La scelta è dovuta principalmente al fatto che modificando la configurazione di un protocollo bisogna riavviare il *Servient*, quindi se dividiamo Client e Server sarà necessario riavviarne solo una parte. Si sono quindi creati 4 *packages*, uno Client, uno Server, uno per le impostazioni (Sezione 4.2.4) e uno per il *Database* (Room, Sezione 4.2.5).

Client

È composto da un *fragment ClientFragment* dell'*activity* principale dedicato e dalla classe Client, nella quale viene istanziato il *Servient* tramite il metodo *clientOnly()*. Per fare questo è stato necessario, chiaramente, inserire un *file* di configurazione in JSON. Sono quindi interrogate le *SharedPreferences* (dove sono salvate le configurazioni dei protocolli) per prendere le

informazioni necessarie alla generazione del *file* di configurazione. Come si può notare dall'esempio, in *client-factories* sono presenti i *packages* delle implementazioni dei protocolli client. Inoltre, essendo un *Servient* Client, non è necessario impostare le configurazioni per i protocolli perché nella *Thing Description* vengono fornite tutte le informazioni necessarie per consumare una risorsa. Fa eccezione MQTT in quanto per funzionare ha bisogno di accedere ad un *broker* dove avverranno poi le *subscriptions* ai *topics*.

```
{
  "wot": {
    "servient": {
      "servers": [],
      "client-factories": [
        "com.example.wot_servient.http.client.MqttProtocolClientFactory"
      ],
      "mqtt": {
        "broker": "tcp://test.mosquitto.org",
        "bind-port": 1883,
        "client-id": "3dft6747c",
        "username": "testWoTServient",
        "password": "testWoTServient",
        "all-TD-topic": "AndroidWotServientAllTDTopic"
      }
    }
  }
}
```

Subito dopo l'avvio del *Servient*, se non sono presenti *Consumed Things*, come si può vedere in figura 4.2, l'unica interazione possibile è quella con il *Button* per aggiungere una *Thing Description*. Si aprirà quindi un menù con tre possibili scelte per inserire una *Thing Description* (Figure 4.3 e 4.4). Dopo aver importato la *Thing Description*, viene generata una *Consumed Thing* e inserita nella *Recycler View* insieme a tutte le altre. Adesso è possibile cliccare sulla riga corrispondente alla *Consumed Thing* in questione e si aprirà un'*activity* dove è possibile eliminare la *Consumed Thing* e visualizzare proprietà ed azioni corrispondenti (Figura 4.5).

Server

Il *package* "server" è strutturato in modo speculare a quello del Client. Anche qua abbiamo un *fragment* dedicato *ServerFragment* e la classe Server che istanzia il *Servient serverOnly*. In aggiunta qua è presente la gestione dei

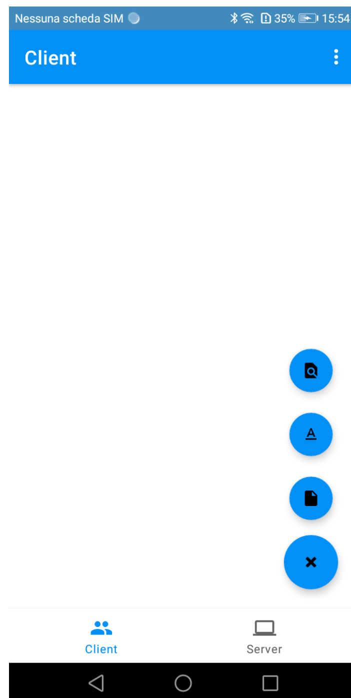

Figura 4.2: Ways to add TD

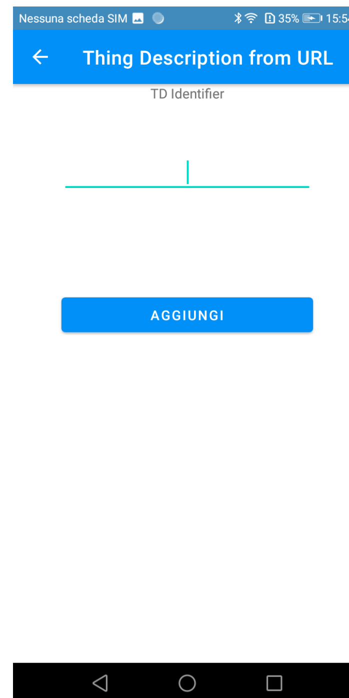

Figura 4.3: Add TD from URL

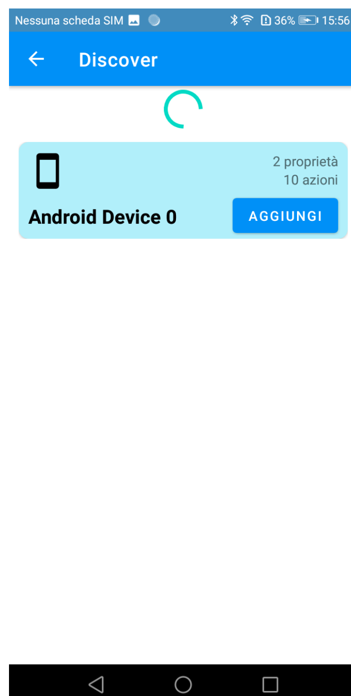

Figura 4.4: Add TD from Discovery

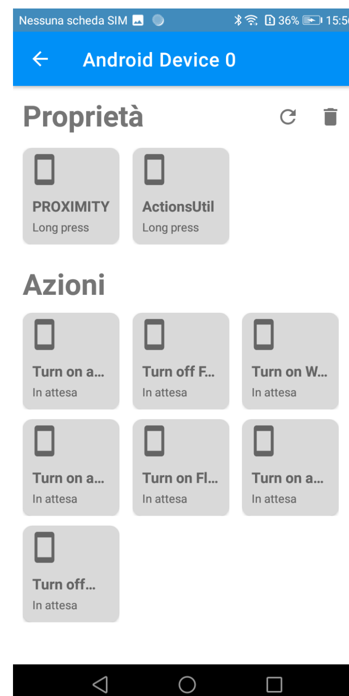

Figura 4.5: ConsumedThing details

sensori del dispositivo (Sezione 4.2.3) tramite la classe DeviceAsThing che converte i sensori scelti dall'utente in *Thing Properties* di una *Thing* completamente dedicata al dispositivo.
Per quanto riguarda il *file* di configurazione, bisogna inserire le configurazioni per tutti i protocolli Server presenti.
A livello di GUI, inizialmente si presenta con un *Button* che avvia le funzionalità Server, generando una *Persistent Notification* indicante lo stato attuale del Server (disattivabile nelle Impostazioni). Cliccando il *Button* in basso si aprirà un *Dialog*, contenente un form (Figura 4.6), che consente all'utente di esporre una *Thing* specificando alcune informazioni.
Dopo aver esposto i sensori viene quindi generato un *item* sulla *Recycler View* (Figura 4.7) che, cliccato, crea un'*activity* specifica per quella *Exposed Thing* dove è possibile visualizzare tutte le informazioni oppure rimuoverla (Figure 4.8 e 4.9).

4.2 Tecnologie utilizzate

4.2.1 Java

Il linguaggio di programmazione utilizzato è Java, nello specifico la versione 8. Si è scelto di utilizzare Java rispetto a Kotlin per diversi motivi fra cui l'ampia disponibilità di librerie e di documentazione online, senza contare che rimane comunque il linguaggio più diffuso su Android. Inoltre il progetto di base era scritto in Java 11, quindi è stato più facile l'adattamento di alcuni componenti già esistenti, nonostante molte librerie non siano compatibili con Android. Altro aspetto importante è il paradigma *Object Oriented* che garantisce una struttura comprensibile, permettendo una più facile gestione e manutenzione di progetti complessi come questo.

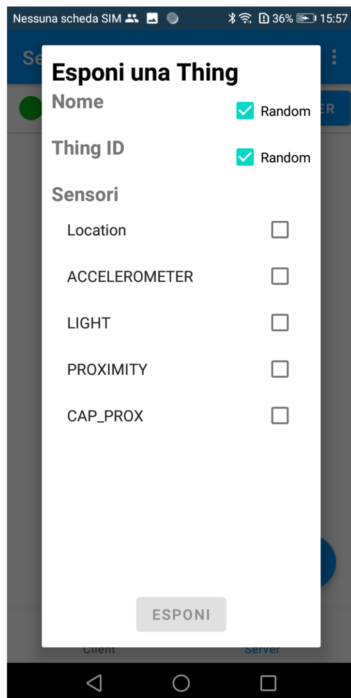

Figura 4.6: Dialog Sensors

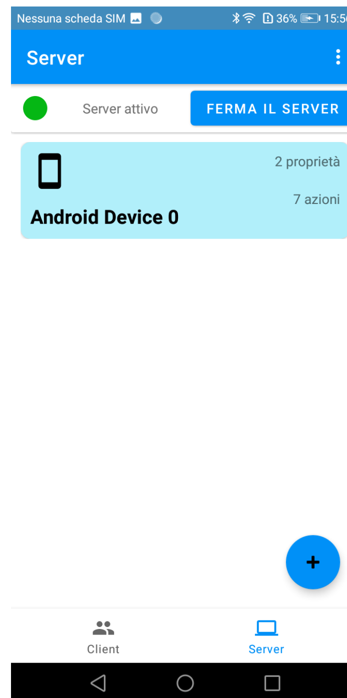

Figura 4.7: Thing Exposed

Figura 4.8: ExposedThing details

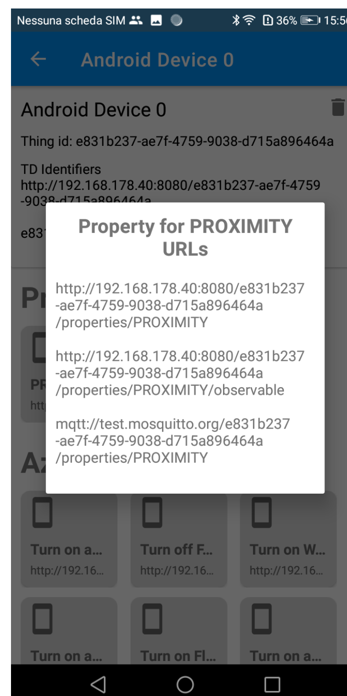

Figura 4.9: Thing Property URLs

4.2.2 JSON e Libreria Jackson

Come formato dati per i *files* di configurazione è stato scelto il JSON. Il progetto originale utilizzava HOCON (*Human-Optimized Config Object Notation*) che garantisce maggiore leggibilità. Il core ha mantenuto il supporto a questo tipo di formato dati, ma si è scelto di utilizzare il JSON perché è già utilizzato per le *Thing Descriptions* ed è anche il tipo di metadato più utilizzato per la trasmissione di messaggi nel *Web of Things*.
Il linguaggio Java ha nativamente il supporto al formato dati JSON tramite la libreria *org.json*, tuttavia manipolazione, serializzazione e deserializzazazione sono semplificati dall'utilizzo della libreria Jackson. Quest'ultima è stata utilizzata per la generazione dei file di configurazione e per la serializzazione delle *Thing Descriptions* in oggetti *Thing*.

4.2.3 Esposizione Sensori

L'SDK di Android fornisce la libreria per la gestione dei sensori integrata (*Sensor Framework*) tramite la quale è possibile verificare i sensori presenti nel dispositivo e riceverne gli aggiornamenti.
Alcuni di questi sensori sono *hardware-based*, altri sono *software-based*. Quelli *hardware-based* sono integrati fisicamente all'interno del dispositivo (ad esempio l'accelerometro o il giroscopio), mentre quelli *software* sono generati dal sistema operativo (ad esempio l'orientamento). Il *SensorManager* non fa differenza fra le due tipologie e fornisce un'interazione comune qualunque sia il sensore scelto.
Dopo aver individuato i sensori presenti si possono visualizzare i dati relativi, ma per ricevere gli aggiornamenti in tempo reale è necessario registrare un *listener* (*SensorEventListener*) che è composto da due funzioni principali:

- **onAccuracyChanged(Sensor sensor, int accuracy)** Viene invocata quando cambia la precisione del sensore registrato. A differenza di *onSensorChanged()*, questo viene chiamato solo quando la precisione cambia.

- **onSensorChanged(SensorEvent event)** Invocata quando c'è un nuovo *SensorEvent* che contiene tutte le informazioni sul sensore (es. il tipo, il produttore) e sull'evento scatenato (es. il *timestamp* o la precisione). Il *SensorEvent* contiene anche un *array* con tutti i nuovi valori riscontrati per ogni asse. La grandezza dell'*array* dipende dal sensore in questione, ad esempio per l'accelerometro ha grandezza 3 (un valore per ogni asse), mentre per il sensore di luminosità ha grandezza 1.

Se un sensore è esposto nel WoT, ad ogni chiamata di *onSensorChanged()* viene aggiornata la relativa *Thing Property* (il tutto serializzato tramite formato JSON, esempio 4.1).


```
{
  "name": "LIGHT",
  "id": 0,
  "vendor": "MTK",
  "stringType": "android.sensor.light",
  "fifoMaxEventCount": 0,
  "fifoReservedEventCount": 0,
  "highestDirectReportRateLevel": 0,
  "maxDelay": 1000000,
  "maximumRange": 65535,
  "minDelay": 0,
  "power": 0.0010000000474974513,
  "reportingMode": 1,
  "resolution": 1,
  "type": 5,
  "version": 1,
  "isAdditionalInfoSupported": false,
  "isDynamicSensor": false,
  "isWakeUpSensor": false,
  "event": {
    "timestamp": 341113304423976,
    "accuracy": 3,
    "values": {
      "0": 6
    }
  }
}
```


Listing 4.1: SensorEvent serializzato in JSON

4.2.4 Settings

La sezione delle impostazioni è stata realizzata, come da linee guida Android, tramite la *AndroidX Preference Library* che permette di gestire il tutto tramite i *files* XML inserendo direttamente le impostazioni nelle *SharedPreferences*. Bisogna solamente occuparsi della gestione al cambiamento di una

preference. In questo caso bisogna attivare un *listener* sulle *SharedPreferences* che ci informerà ad ogni cambiamento. Nel progetto, ad esempio, quando viene attivato/disattivato un protocollo viene automaticamente riavviato il *Servient*.

4.2.5 Database Room

Per il salvataggio delle *Consumed Things* nella parte Client del progetto viene utilizzato il *database* locale *Room* che fornisce un livello di astrazione su *SQLite* per consentire un accesso fluido al database sfruttando tutta la potenza di *SQLite*. In particolare, ogni qualvolta viene inserita una *Thing Description*, essa viene salvata nel *database* per evitare di dover reinserire le *Thing Descriptions* ad ogni riavvio.

4.2.6 Librerie Protocolli

Ad eccezione della parte Client di HTTP, per la quale si è utilizzata la libreria Java integrata *URLConnection*, ogni protocollo necessita di una libreria esterna.

- **HTTP server** Per la parte Server di HTTP si è utilizzata la libreria *SparkJava* che fornisce gli strumenti per avviare un *web server* HTTP, con la possibilità di definire la porta, l'*host* e i vari percorsi di indirizzamento.

- **CoAP** È stata utilizzata la libreria *Californium*. Essa non è nata come libreria per Android, tuttavia ha tutte le caratteristiche necessarie al nostro scopo. Come da definizione, è dedicata principalmente ai piccoli dispositivi IoT ma è possibile utilizzarla anche con dispositivi più potenti. Supporta tutte le *features* di CoAP.

- **WebSocket** Per quanto riguarda WebSocket, l'implementazione del progetto originale risulta compatibile con Android. Infatti è bastato

fare solamente il *downgrade* del linguaggio da Java 11 a Java 8 e modificare l'invio dei messaggi per la comunicazione. La libreria utilizzata fa parte del framework *Netty* che permette di sviluppare facilmente *Network Applications*.

- **MQTT** È stato il protocollo più semplice da implementare, in quanto non era necessario creare un *web Server*, ma interagire con un *Broker* già creato altrove. Questa caratteristica lo contraddistingue perchè è l'unico fra i protocolli implementati a potersi connettere dall'esterno. La libreria utilizzata è *Paho Android Service*.

Capitolo 5

Validazione

In questo capitolo verrà fornito un esempio di validazione del progetto AndroidWotServient. Si mostrerà un possibile caso d'uso relativo ad integrazione fra *Smart Home* e dispositivo Android, descrivendo i possibili ambiti di utilizzo, i vantaggi che ne derivano e l'implementazione sviluppata.

5.1 Caso d'uso e ambiti di utilizzo

Nel caso d'uso che andremo ad analizzare viene utilizzato il WoT *Servient* Android per automatizzare alcune delle azioni che svolgiamo quotidianamente su dispositivi *Smart*. In particolare prenderemo in esame uno smartphone Android e tre fra i più comuni dispositivi di una *Smart Home* (*Smart Bulb*, *Smart Socket* e *Smart Thermostat*). L'obiettivo è quello di scatenare delle azioni su questi dispositivi in base al luogo in cui ci troviamo, in questo caso prenderemo in considerazione il luogo di lavoro come trigger per l'automazione. I dispositivi utilizzati sono i seguenti:

- **Smartphone Android** Dispositivo che espone la proprietà *location* e l'azione *turnSilentModeOn()* che attiva la modalità Silenziosa;
- **Smart Bulb/Smart Socket** Ha la proprietà *isTurnedOn* (indica l'attuale stato della lampadina) e le azioni *turnOn()* e *turnOff()*;

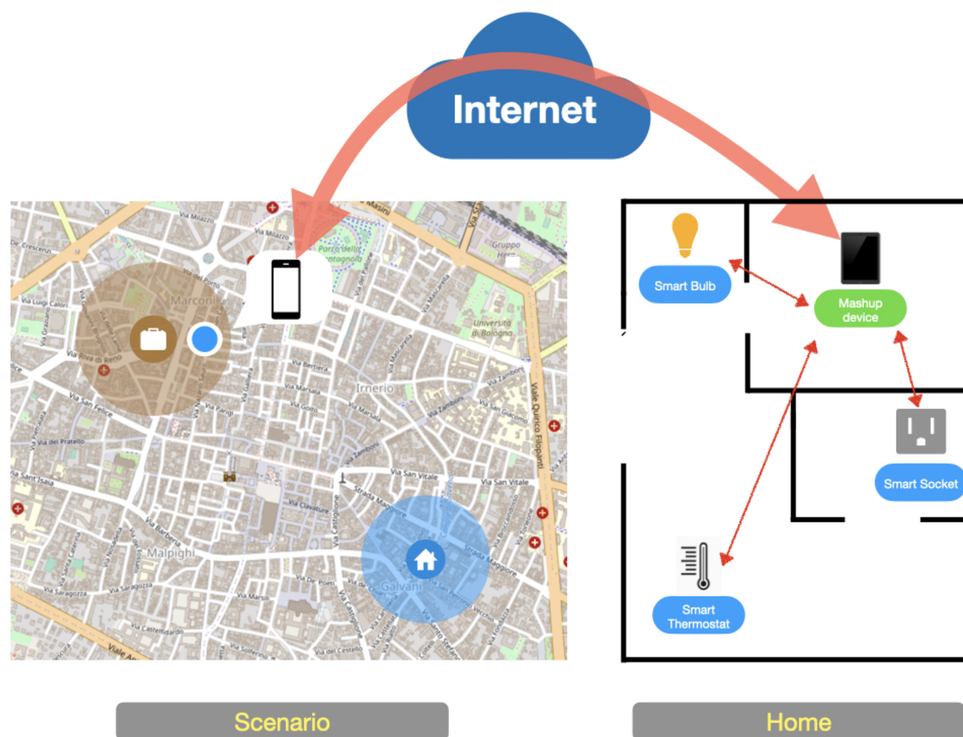

Figura 5.1: Scenario caso d'uso

- **Smart Thermostat** Ha una proprietà relativa all'attuale temperatura impostata e un'azione *setTemperature()* che imposta la temperatura con il valore passato come parametro;

Quindi lo scenario (Figura 5.1) è il seguente: quando la proprietà *location* indica che il dispositivo si trova sul luogo di lavoro, allora esegui le azioni *turnOn()* su *Smart Bulb* e *Smart Socket*, *setTemperature(18)* sullo *Smart Thermostat* e *turnSilentModeOn()* sul dispositivo Android.
Questo caso d'uso e simili automatizzano delle operazioni che svolgiamo quotidianamente, portando anche diversi vantaggi fra i quali:

- evitare gli sprechi energetici perché consente di spegnere automaticamente i dispositivi non utilizzati;

- migliore gestione della climatizzazione, perché consente di impostare una temperatura più bassa quando siamo fuori casa;

Oltre ai dispositivi già citati deve essere presente anche un dispositivo di *Mashup* che consente di comunicare con l'esterno dell'ambiente domestico, ad esempio un tablet sempre presente in casa. In esso saranno presenti le quattro *Consumed Things* relative ai dispositivi in questione e una funzione di AndroidWoTServient appositamente sviluppata per poter eseguire questo tipo di automazione.

5.2 Implementazione

Per l'implementazione di questo caso d'uso è stata sviluppata appositamente una funzionalità di AndroidWotServient che consente di invocare una o più azioni al verificarsi di una data condizione. Questa funzionalità sarà eseguita sul dispositivo di *Mashup*, dove sono presenti le *Consumed Things* relative ai quattro dispositivi (Smartphone, *Smart Socket*, *Smart Bulb*, *Smart Thermostat*). È presente un form dove inserire tutti i dati relativi all'automazione. Prima di tutto bisogna scegliere una *Thing Property* fra quelle delle *Consumed Things* e il *trigger* da invocare, ne esistono tre tipi:

- **Localizzazione (Figura 5.2)** Consente di specificare Latitudine e Longitudine (è possibile anche ricercare tramite indirizzo), per indicare la localizzazione, e la distanza (in metri) per indicare il raggio entro cui il *trigger* deve scatenarsi;

- **Sensore (Figura 5.3)** Consente di specificare il valore esatto per ciascuno degli assi di un sensore (non è necessario inserirli tutti), quando il sensori avrà esattamente quei valori allora sarà scatenato il *trigger*;

- **Plain text (Figura 5.4)** È possibile inserire direttamente in *plain text* il valore per far scattare il *trigger*.

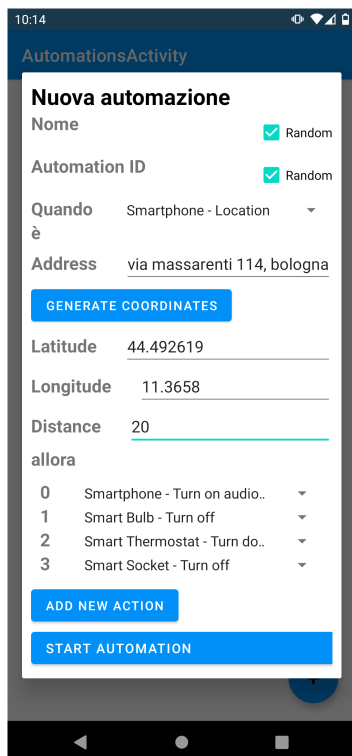

Figura 5.2: Location trigger

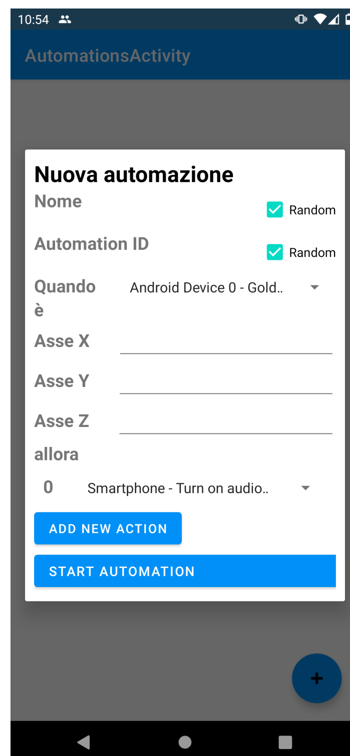

Figura 5.3: Sensor trigger

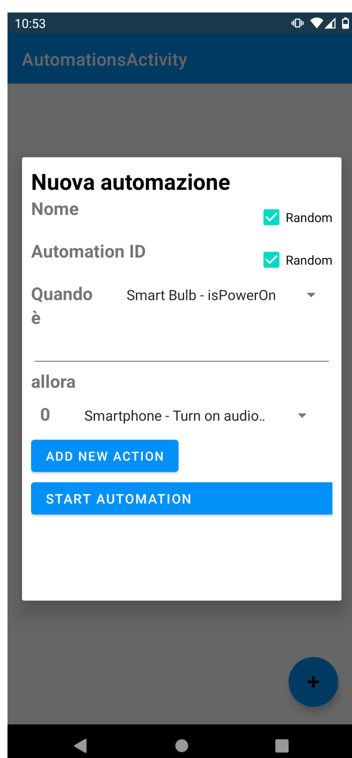

Figura 5.4: Plain Text trigger

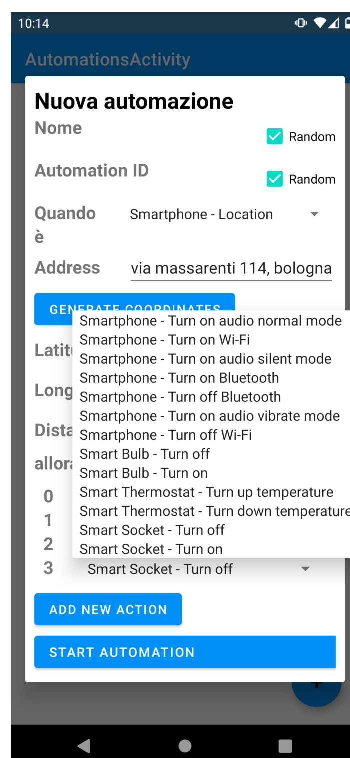

Figura 5.5: Scelta delle Azioni

In questo caso d'uso sceglieremo come *Consumed Thing* lo Smartphone e come *trigger* la Localizzazione. A questo punto bisogna inserire le coordinate (Latitudine e longitudine) oppure inserire l'indirizzo e procedere con la generazione automatica. Per completare la condizione va inserita anche la lunghezza (in metri) del raggio entro cui far partire il *trigger*. Ultima informazione necessaria a creare l'automazione è la scelta delle azioni da invocare nel momento in cui il trigger viene attivato. Le azioni presenti sono quelle delle *Consumed Things*, nel nostro caso sceglieremo *turnSilentModeOn()* per lo Smartphone, *turnOff()* per *Smart Bulb* e *Smart Socket*, *setTemperature(18)* per *Smart Thermostat*. A questo punto l'automazione è attiva e al verificarsi della condizione verranno invocate le azioni scelte.

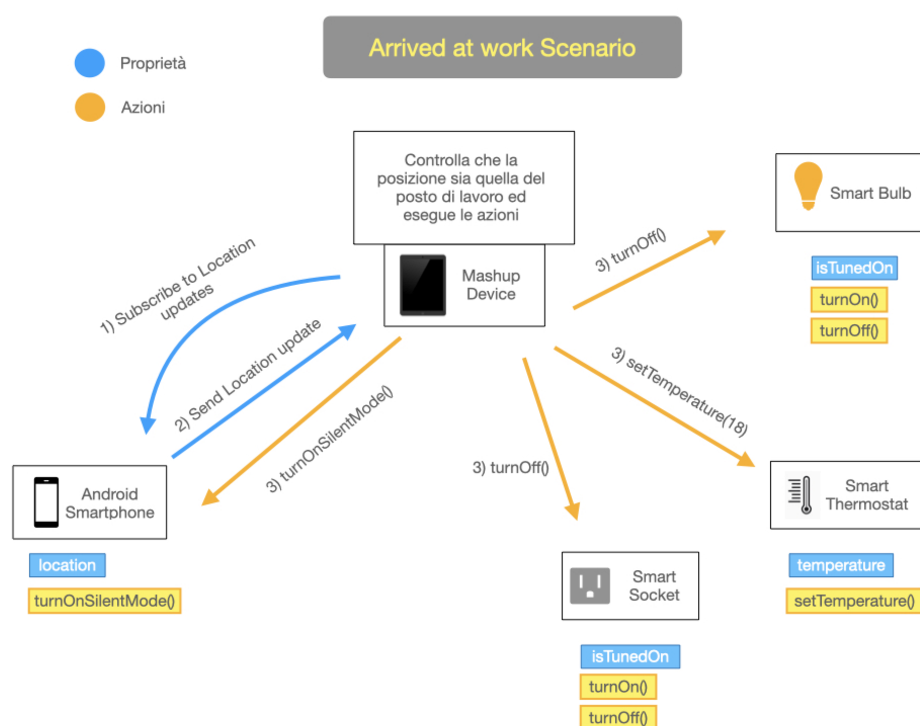

Figura 5.6: Scenario caso d'uso

In figura 5.6 è evidenziato come avviene il flusso dei dati e di comunicazione subito dopo che l'Automazione è stata creata. Il *Mashup device* si

sottoscrive agli aggiornamenti relativi alla Localizzazione dello Smartphone, quindi il dispositivo continua ad aggiornare la *Thing Property location.* Il *Mashup device* riceve il cambiamento della *Property*, controlla che sia nel raggio scelto in precedenza e invoca le tre azioni scelte.
Come protocolli di comunicazione, attualmente l'unico che comunica verso l'esterno è MQTT in quanto ha la necessità di connettersi ad un *broker.* Nel nostro caso d'uso infatti Smartphone e *Mashup* device sono entrambi connessi allo stesso *broker*, quindi la comunicazione può avvenire attraverso Internet. Gli altri dispositivi invece possono utilizzare qualunque protocollo, perché devono solo comunicare localmente con il *Mashup device.* Potenzialmente il *Mashup* può essere eseguito in cloud, per fare questo è però necessario che i dispositivi coinvolti siano dotati di un IP pubblico.

Capitolo 6

Conclusioni

In questa tesi è stato inizialmente introdotto il paradigma dell'*Internet of Things*, facendo un excursus sui campi di applicazione e tencologie utilizzate fino ad arrivare al problema dell'interoperabilità e quindi all'ideazione del *Web of Things* e alla sua implementazione. Si è presentato il progetto di tesi AndroidWotServient, un WoT *Servient* per dispositivi Android basato sulle specifiche del W3C descrivendo la fase di progettazione architetturale e implementativa. È stata quindi necessaria una fase di studio del *Web of Things* in generale, in particolare delle specifiche fornite dal W3C, e, come detto nell'elaborato, della struttura interna di SANE WoT *Servient* [14]. Il progetto è stato poi messo sul campo presentando un caso d'uso che automatizza alcune delle azioni che svolgiamo quotidianamente su dispositivi Smart.

6.1 Limiti del progetto e possibili sviluppi

Lo sviluppo del progetto è stato molto complicato, in particolare la strutturazione del codice per semplificare l'implementazione di nuove funzionalità future. Il progetto in Java 11 ha fornito una notevole mano d'aiuto, tuttavia il *downgrade* a Java 8 e la non compatibilità di gran parte delle librerie hanno reso il lavoro non semplice. Inoltre è stata anche sviluppata da zero l'interfaccia grafica, che ora permette facilmente di importare *Consumed Things* e

di esporre i sensori interni del dispositivo utilizzato.
Fra i possibili sviluppi del progetto abbiamo:

- **Protocolli** È possibile facilmente aggiungerne di nuovi in quanto il progetto è stato strutturato proprio per non dover modificare il *Core* ad ogni integrazione di *Binding Templates*;

- **Sicurezza** Bisogna evolvere gli attuali protocolli per utilizzare i meccanismi di sicurezza appropriati (HTTPS, CoAPS, ecc.);

- **IP pubblico** Per quanto riguarda i protocolli HTTP, WebSocket e CoAP è possibile esporre le Thing Descriptions tramite IP pubblico per permettere la comunicazione non solo in locale. MQTT invece, avendo la necessità di utilizzare un broker, ha già questa possibilità.

- **Broker locale MQTT** È possibile migliorare il supporto ad MQTT permettendo l'utilizzo di un broker locale. Quindi creando un broker direttamente in AndroidWoTServient da esporre nella rete locale o verso l'esterno.

- **GUI** Nonostante quella attuale rispetti i requisiti indicati, è possibile migliorare la GUI sviluppando funzionalità aggiuntive massimizzando l'utilizzo dei dati già presenti;

- **WoT Scripting API** È possibile esplorare e sviluppare il *Building Block* opzionale *WoT Scripting API*;

- **W3C** È necessario continuare a tenere d'occhio le specifiche del W3C, che sono spesso soggette a cambiamenti, adattando il progetto.

Bibliografia

[1] Kenneth Li-Minn Ang e Jasmine Kah Phooi Seng. «Application specific internet of things (ASIoTs): Taxonomy, applications, use case and future directions». In: *IEEE Access* 7 (2019), pp. 56577–56590.

[2] Kevin Ashton et al. «That 'internet of things' thing». In: *RFID journal* 22.7 (2009), pp. 97–114.

[3] «Building the Web of Things: Book.webofthings.io». In: ().

[4] «Eclipse Thingweb node-wot». In: (). URL: `https://github.com/eclipse/thingweb.node-wot`.

[5] Mahmoud Elkhodr, Seyed Shahrestani e Hon Cheung. «The Internet of Things: New Interoperability, Management and Security Challenges». In: *International Journal of Network Security I& Its Applications* 8.2 (mar. 2016), pp. 85–102. ISSN: 0974-9330. DOI: `10.5121/ijnsa.2016.8206`. URL: `http://dx.doi.org/10.5121/ijnsa.2016.8206`.

[6] Ian Fette e Alexey Melnikov. *The websocket protocol*. 2011.

[7] Ala Al-Fuqaha et al. «Internet of things: A survey on enabling technologies, protocols, and applications». In: *IEEE communications surveys & tutorials* 17.4 (2015), pp. 2347–2376.

[8] Tim Kindberg et al. «People, places, things: Web presence for the real world». In: *Mobile Networks and Applications* 7.5 (2002), pp. 365–376.

[9] Rwan Mahmoud et al. «Internet of things (IoT) security: Current status, challenges and prospective measures». In: (2015), pp. 336–341. DOI: `10.1109/ICITST.2015.7412116`.

[10] Federico Montori, L. Bedogni e L. Bononi. «A Collaborative Internet of Things Architecture for Smart Cities and Environmental Monitoring». In: *IEEE Internet of Things Journal* 5 (2018), pp. 592–605.

[11] Dae-Hyeok Mun, Minh Le Dinh e Young-Woo Kwon. «An Assessment of Internet of Things Protocols for Resource-Constrained Applications». In: 1 (2016), pp. 555–560. DOI: `10.1109/COMPSAC.2016.51`.

[12] Mohd Muntjir, Mohd Rahul e Hesham Alhumiany. «An Analysis of Internet of Things(IoT): Novel Architectures, Modern Applications,Security Aspects and Future Scope with Latest Case Studies». In: *Building Services Engineering Research and Technology* 6 (giu. 2017).

[13] Dave Raggett. «The web of things: Challenges and opportunities». In: *Computer* 48.5 (2015), pp. 26–32.

[14] «SANE WoT Servient». In: (). URL: `https://github.com/sane-city/wot-servient`.

[15] Satista. «Internet of Things (IoT) connected devices instal- led base worldwide from 2015 to 2025». In: (2016). URL: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`. [Last update 27-November-2016].

[16] Zach Shelby, Klaus Hartke e Carsten Bormann. «The constrained application protocol (CoAP)». In: (2014).

[17] W3C. «WoT Architecture». In: (2020). URL: `https://www.w3.org/TR/wot-architecture/`. [Last update 9-April-2020].

[18] W3C. «WoT Binding Templates». In: (2020). URL: `https://www.w3.org/TR/wot-binding-templates/`. [Last update 30-January-2020].

[19] W3C. «WoT Thing Description». In: (2020). URL: `https://www.w3.org/TR/wot-thing-description/`. [Last update 23-June-2020].

[20] Erik Wilde. «Putting things to REST». In: (2007).

[21] «WoT Implementations». In: (). URL: https://www.w3.org/WoT/developers/.