Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

STUDIO E INTEGRAZIONE DI UN AMBIENTE WEB OF THINGS PER IL MONITORAGGIO STRUTTURALE CON FRAMEWORK DI INTEROPERABILITÀ INDUSTRIALE ARROWHEAD

Relatore:
Dott.
FEDERICO MONTORI

Presentata da:
ALESSIO DI DIO

Correlatore:
Dott.
LUCA SCIULLO

Sessione II
Anno Accademico 2019/2020

Sommario

Ogni giorno viene generata una grandissima mole di dati che è possibile rilevare, gestire e studiare attraverso l'uso di sensori di diverso tipo. Tutti questi tipi di sensori sono componenti intelligenti che, pur facendo parte dell'Internet of Things, hanno problemi di interoperabilità dovuti ad esigenze di mercato, ingegneristiche e storiche. Questo limita enormemente le potenzialità dell'Internet of Things, un settore in piena crescita, per cui sono in via di sviluppo una serie di tecnologie che permettono la comunicazione tra Smart Things nonostante la diversa struttura interna. Ad oggi esistono alcune soluzioni che però necessitano di essere migliorate ed aggiornate per poterle rendere utilizzabili su larga scala. Con questa tesi dimostrerò che è possibile realizzare un framework che permette la comunicazione tra Smart Things e smartphone e, più in generale, con il Web dimostrandone il corretto funzionamento attraverso un caso d'uso reale relativo al monitoraggio strutturale.

Indice

Elenco delle figure

Introduzione

La diffusione e la crescita esponenziale del numero di dispositivi connessi ad Internet permette di risolvere problematiche che prima risultavano difficili da risolvere da parte dell'uomo, ad esempio il monitoraggio strutturale. Ad oggi l'uomo deve fissare una verifica periodica della struttura da monitorare ed è possibile quindi che alcune verifiche siano totalmente inutili mentre magari ne sia necessaria una d'urgenza nell'intercorrere tra una visita e l'altra. Questo fa si che alcune volte l'uomo faccia degli sforzi inutili mentre altre volte sembra che sia indifferente davanti ai problemi. Scavando più a fondo però è possibile notare che molto spesso gli errori non vengono fatto dall'uomo ma sono causati da un difetto del sistema di monitoraggio. Attraverso l'utilizzo dell'Internet of Things è possibile installare dei sensori nell'edificio di interesse in modo tale da monitorarlo continuamente ed in tempo reale ed agire con la verifica dell'edificio "in presenza" solamente quando i sensori rilevano dei valori anomali. I sensori necessari per questo tipo di monitoraggio sono principalmente accelerometri e giroscopi che installati in determinati punti strategici della struttura permettono di rilevare al meglio il suo stato di "salute". Lo scopo di questa tesi è quindi duplice, da una parte permettere l'interazione in tempo reale tra i sensori e l'uomo e dall'altro gestire i dati ricevuti dai sensori per monitorare l'edificio. Per realizzare il primo punto è stato utilizzato uno standard recentemente proposto dal W3C, il Web of Things, mettendolo in comunicazione con il framework di interoperabilità industriale Arrowhead. Per dimostrare il corretto funzionamento nella comunicazione tra i due sistemi è stata sviluppata un'applicazione mobile che permette inoltre il monitoraggio di un edificio situato a Bologna in zona Lazzaretto. La tesi è divisa in 5 capitoli:

- Capitolo 1, Stato dell'Arte: in questo capitolo andremo a vedere la letteratura relativa all'Internet of Things ed ai problemi di integrazione con altri sistemi dello stesso tipo, vedremo inoltre una soluzione proposta emergente, il Web of Things, una tecnologia che astrae i dispositivi fisici permettendo l'interoperabilità tra qualunque tipo di dispositivo discutendo su alcuni progetti che lo utilizzano. Vedremo inoltre l'architettura e funzionamento del framework di interoperabilità industriale Arrowhead dimostrando come sia possibile l'interoperabilità tra i due sistemi.

- Capitolo 2, Architettura: in questo capitolo vedremo, dal punto di vista architetturale, come sia possibile realizzare l'integrazione tra il WoT ed Arrowhead. Inoltre vedremo l'architettura dell'applicazione mobile che verrà realizzata per dimostrare la corretta integrazione tra

WoT ed Arrowhead e che fornirà una possibile soluzione al problema del monitoraggio strutturale.

- Capitolo 3, Implementazione: in questo capitolo verranno descritte le componenti sviluppate per la corretta realizzazione del progetto, verranno inoltre mostrati e spiegati gli snippet di codice più importanti.
- Capitolo 4, Risultati: in questo capitolo verranno mostrati i risultati prodotti e verranno discussi.
- Capitolo 5, Conclusioni: in questo capitolo verrà fatto un riepilogo di quanto sviluppato nel corso di questa tesi e verranno suggeriti sviluppi futuri per il miglioramento del progetto.

Capitolo 1

Stato dell'Arte

1.1 IoT e Problemi di Integrazione

Al giorno d'oggi, ognuno di noi è circondato da sensori: li troviamo all'interno dei nostri smartphone, nelle automobili, nelle nostre case. Alcuni sensori posso essere collegati in rete in modo tale da poter comunicare tra loro ma ad oggi non c'è un framework "universale" che permetta a qualsiasi sensore di comunicare con qualunque altro. La necessità di far si che vari oggetti totalmente diversi ma appartenenti al mondo del IoT possano comunicare tra di loro nasce dal fatto che avendo oggetti (o thing) di diverso tipo è possibile ottenere informazioni di diverso tipo ma legate tra loro in modo tale da poterle incrociare per ottenere ulteriori nuove informazioni che possono portare un soggetto a comportarsi in modo diverso. Degli esempi possono essere le sveglie che suonano prima a causa del traffico, le scarpe da ginnastica che trasmettono valori relativi all'attività fisica, un ombrello che avverte nel caso in cui è prevista pioggia [2]. Questi sono gli esempi più banali di una tecnologia che sta portando alla nascita di una nuova metodologia di pensiero non più basata sul controllo umano ma basata su un controllo affidato a sistemi informatici, principalmente sensori, che costruiscono reti permettendo un minor sforzo umano, una maggiore efficienza e una maggiore rapidità d'azione quando si è davanti ad un'anomalia o ad un problema. Stanno nascendo vari paradigmi emergenti che provano a risolvere questo problema come ad esempio il paradigma C-IoT che viene usato nell'architettura SenSquare [4] o il paradigma del Web of Things, proposto dal W3C, che permette l'interoperabilità di qualunque tipo di thing. Spesso, quando si realizza un software che si basa sul mondo IoT, ci si rende conto che è necessario un paradigma orientato ai servizi in modo tale da poter soddisfare i bisogni dei consumatori dovuti al fatto che i dati devono essere accessibili in qualunque momento

in cui l'utente voglia ottenerli. In questi casi una buona opzione è quella di utilizzare una struttura basata su un cloud che può essere globale o locale. La scelta sul cloud dipende dalla tipologia di dati e su come devono essere manipolati gli stessi perchè nel caso in cui si utilizzi un cloud globale si riduce la capacità di interoperabilità del sistema mentre se si utilizza un sistema con cloud locale si garantisce una maggiore sicurezza e una migliore interoperabilità [3]. La più grande difficoltà nella realizzazione di questi sistemi consiste quindi nel rendere strumenti autonomi interoperabili, questo è dovuto al fatto che vengono utilizzati protocolli di rete a bassa potenza come ZigBee, ZWave o Bluetooth, protocolli di rete standard come Ethernet e WiFi e collegamenti cablati. Una soluzione per risolvere i problemi di interoperabilità può essere quella di risolverli alla radice attraverso una standardizzazione hardware ma questo ad oggi non è ancora avvenuto. Vari produttori commerciali hanno sviluppato strumenti che supportano più protocolli di rete attraverso l'assemblaggio di più componenti hardware ma ad oggi la soluzione migliore sembra essere quella di risolvere il problema dal punto di vista software creando applicazioni che non si basino sul protocollo di rete utilizzato. Per questo motivo sono stati proposti vari protocolli di rete di livello 7 (Applicazione) come CoAP (Constrained Application Protocol), MQTT (Message Queue Telemetry Transport), XMPP (Extensible Messaging and Presence Protocol) e HTTP (HyperText Transfer Protocol). Ciascuno di essi ha delle caratteristiche uniche sia architetturali sia per quanto riguarda la messaggistica, utili per diversi tipi di applicazioni in ambito IoT che dipendono anche dalla potenza energetica e di rete richiesti. Le prime applicazioni in ambito IoT hanno largamente utilizzato i protocolli CoAP e XMPP, solo successivamente si passò all'utilizzo delle API REST, da qui si capì che una buona strategia poteva essere quella di utilizzare delle Thing come fossero servizi adottando un approccio orientato ai servizi. Oggi non abbiamo più un solo computer nelle nostre case e ne portiamo sempre uno con noi sotto forma di smartphone e nonostante questa evidenza ad oggi le applicazioni IoT spesso non sono compatibili con i sistemi Android eliminando così la possibilità di poter gestire i propri strumenti con un tocco sullo smartphone.

1.2 Web of Things

Il Web of Things (o WoT) descrive un insieme di standard del W3C per risolvere i problemi di interoperabilità di oggetti smart di diverso tipo. Può essere usato in molti domini applicativi come ad esempio in ambito domestico. Ci sono molti oggetti della casa che trarrebbero beneficio nell'essere

connessi: luci ed aria condizionata potrebbero accendersi o meno basandosi sul fatto se in camera è presente qualcuno o meno, le tapparelle possono essere chiuse automaticamente in base alle condizioni meteorologiche e alla presenza di persone, il consumo di energia e di altre risorse può essere ottimizzato in base a modelli di utilizzo e previsioni, si potrebbe accedere alla casa da remoto attraverso assistenti vocali e rendere la propria abitazione domotica. In ambito industriale un esempio sono le Smart Factories, quest'ultime richiedono un monitoraggio dei macchinari di produzione dei prodotti fabbricati, traendo quindi vantaggio dalla capacità di prevedere guasti delle macchine e dalla scoperta anticipata delle anomalie per prevenire costosi tempi di inattività e sforzi di manutenzione. Inoltre, il monitoraggio delle apparecchiature di produzione collegate e dell'ambiente nello stabilimento di produzione permetterebbe di riconoscere la presenza di gas velenosi, rumore o calore eccessivi aumentando la sicurezza dei lavoratori e riducendo i rischi di incidenti o infortuni. In ambito di trasporti e logistica sarebbe possibile monitorare il traffico, il consumo di carburante e la manutenzione del veicolo oltre a poter garantire una migliore rotta e una migliore qualità delle merci trasportate. Come è possibile interagire con i nostri oggetti smart? Per farlo è necessario utilizzare dispositivi che prendono il nome di controllori remoti che possono inviare messaggi all'oggetto smart.

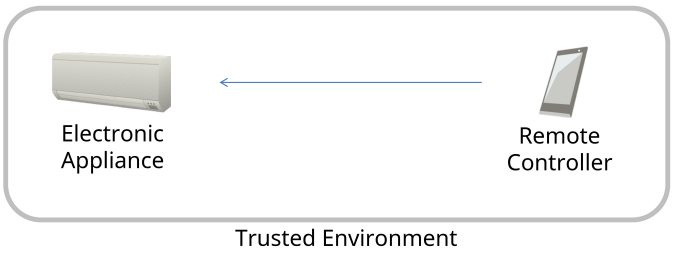

Figura 1.1: Interazione oggetto smart - dispositivo di controllo
Fonte: W3C Architecture [7]

L'interazione può avvenire anche tramite due oggetti smart che, interagendo tra di loro, avviano ad esempio un condizionatore quando la temperatura supera una determinata soglia. Nell'esempio sottostante vediamo come un sensore di temperatura possa interagire con il condizionatore avviandolo quando ritiene che la temperatura sia troppo alta.

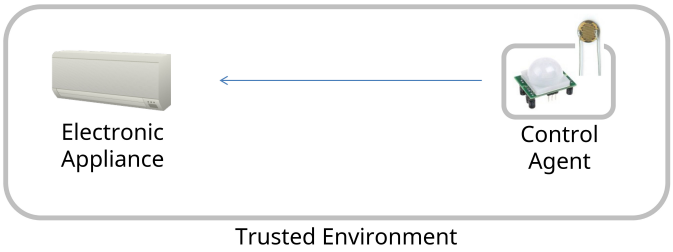

Figura 1.2: Interazione oggetto smart - sensore smart
Fonte: W3C Architecture [7]

Un'altra tecnica di interazione con i dispositivi smart consiste nell'utilizzare un dispositivo autonomo remoto, come ad esempio uno smartphone che, quando collegato ad una determinata rete ottiene automaticamente i permessi di accesso per l'interazione del dispositivo smart mentre se si trova all'esterno della rete ha necessità di usare delle credenziali per interagire.

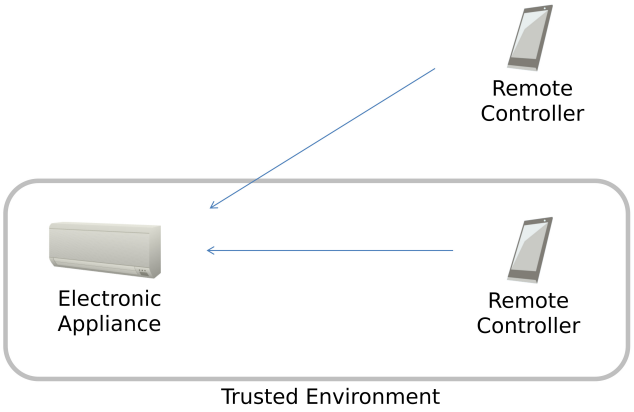

Figura 1.3: Interazione oggetto smart - dispositivo remoto
Fonte: W3C Architecture [7]

Il WoT si basa su dei principi comuni che sono:

- L'architettura WoT deve consentire l'interoperabilità tra diversi ecosistemi usando le tecnologie web.
- L'architettura WoT deve utilizzare API RESTful.
- L'architettuta WoT deve permettere l'utilizzo dei formati più comunemente diffusi sul web.

- Flessibilità: a causa dell'esistenza di un'ampia varietà di dispositivi smart totalmente diversi, il WoT deve essere in grado di essere compatibile con ognuno di essi.

- Compatibilità: a causa dell'esistenza di molti sistemi e standard IoT, il WoT deve essere compatibile con essi costruendo un livello di astrazione più alto in modo da essere compatibile con i sistemi e gli standard IoT attuali e futuri.

- Scalabilità: le applicazioni WoT devono riuscire a rendere scalabili i sistemi IoT che usano migliaia di dispositivi smart in modo tale che questi dispositivi possono usufruire tutti delle stesse funzionalità anche se sono stati creati da aziende produttrici diverse.

- Interoperabilità: il WoT deve fornire l'interoperabilità tra le aziende produttrici di dispositivi smart e il cloud in modo tale che un qualunque dispositivo smart possa essere collegato ad un qualunque servizio di cloud.

I suddetti dispositivi smart in ambito WoT prendono il nome di Web Thing. Ogni Web Thing ha delle funzionalità comuni che sono:

- Capacità di lettura dello stato della Thing.

- Capacità di aggiornamento delle proprietà della Thing.

- Capacità di ricezione di notifiche quando si ha una modifica di una qualunque proprietà della Thing.

- Capacità di invocare funzioni con parametri di input e output capaci di effettuare calcoli che modifichino le proprietà della Thing.

- Capacità di ricezione di notifiche quando si verificano determinati eventi.

L'architettura WoT consente ai client di conoscere attributi, funzionalità, e punti di accesso della Thing e di poter cercare informazioni in base ai suoi attributi e funzionalità. Inoltre supporta un meccanismo di descrizione che permette di descrivere le proprietà e le funzioni delle Thing in modo tale che non siano leggibili solo dall'uomo ma anche dalla macchina. In ambito WoT la Thing è un'astrazione del dispositivo fisico e viene rappresentata attraverso la sua descrizione che prende il nome di Thing Description (TD), quest'ultima permette al client, che prende il nome di Consumer, di scoprire ed interpretare le capacità della Thing e di adattarsi alle diverse possibili

implementazioni della stessa consentendo così l'interoperabilità tra diverse piattaforme IoT, ecosistemi e standard.

Figura 1.4: Interazione Thing - Consumer
Fonte: W3C Architecture [7]

Una Thing può anche essere l'astrazione di un'entità virtuale cioè della composizione di una o più Things (ad esempio, una stanza composta da diversi sensori e attuatori). Un'opzione per la composizione è fornire una Thing Description unica che contenga l'insieme delle funzionalità dell'entità virtuale. Nel caso in cui la composizione risulti complessa è possibile realizzare una Thing Description formata da una struttura gerarchica di Things in cui la TD serve come punto di ingresso e contiene i metadati delle Things contenute.

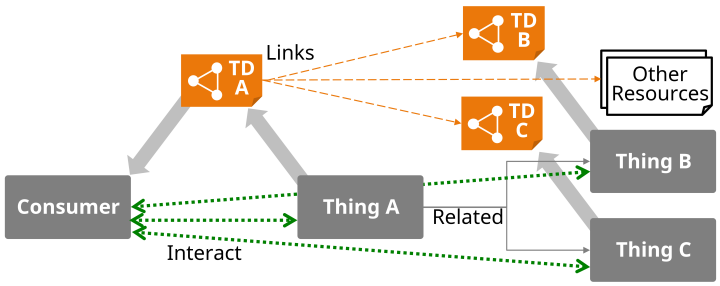

Figura 1.5: Interazione Entità virtuale - Consumer
Fonte: W3C Architecture [7]

Una Thing ha quattro aspetti architetturali importanti: comportamento, Interaction Affordances, configurazione di sicurezza e Protocol Bindings. Il comportamento della Thing include sia il comportamento autonomo sia gli handlers per le Interaction Affordances. Le Interaction Affordances forniscono un modello su come i Consumer possono interagire con la Thing attraverso operazioni astratte ma senza riferimento a uno specifico protocollo di rete o codifica dei dati. Il Protocol Bindings inserisce i dettagli aggiuntivi necessari per mappare ciascuna Interaction Affordance ai messaggi concreti di un

determinato protocollo; in generale, è possibile utilizzare diversi protocolli concreti per supportare diversi sottoinsiemi di Interaction Affordances, anche all'interno di una singola Things. La configurazione di sicurezza di una Thing rappresenta i meccanismi utilizzati per controllare l'accesso agli Interaction Affordances e la gestione dei relativi metadati di sicurezza pubblica e dati di sicurezza privata.

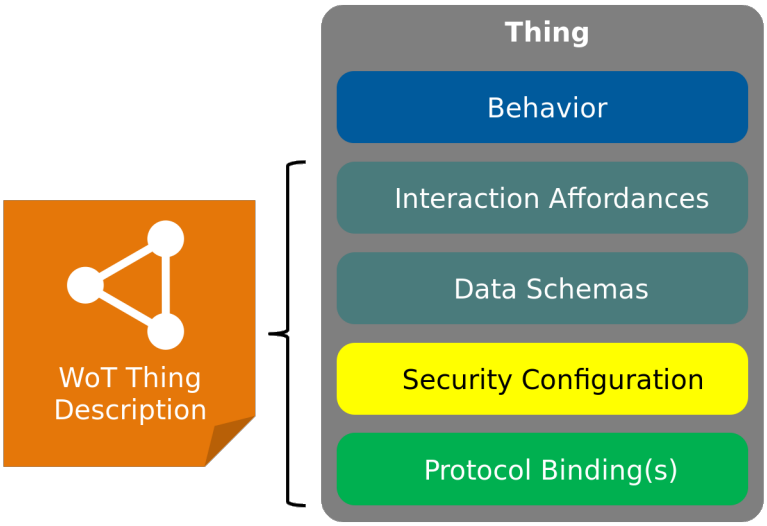

Figura 1.6: Strutttura interna Thing Description
Fonte: W3C Architecture [7]

Le Interaction Affordances possono essere di tre tipi:

- Proprietà: rappresentano lo stato della Thing, devono essere leggibili e possono essere scrivibili, inoltre possono essere osservabili cioè è possibile ricevere una notifica quando si ha un cambiamento della proprietà. Un esempio di proprietà sono i valori di un sensore, i risultati di calcoli.

- Azioni: permettono di invocare azioni della Thing. Un'azione può anche manipolare più proprietà cambiandone lo stato sia interno che fisico nel tempo. Esempi di azioni sono accensione e spegnimento di una lampadina, modifica della luminosità.

- Eventi: descrivono azioni che vengono eseguite successivamente ad un determinato avvenimento come ad esempio l'avvio di un allarme dopo aver recepito un movimento.

La Thing Description, descrivendo la Thing, può contenere tutte le informazioni sopra esposte e viene rappresentata sotto forma di JSON-LD.

EXAMPLE 2: Thing Description with TD Context Extension for semantic annotations

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        { "saref": "https://w3id.org/saref#" }
    ],
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "@type": "saref:LightSwitch",
    "securityDefinitions": {"basic_sc": {
        "scheme": "basic",
        "in": "header"
    }},
    "security": ["basic_sc"],
    "properties": {
        "status": {
            "@type": "saref:OnOffState",
            "type": "string",
            "forms": [{
                "href": "https://mylamp.example.com/status"
            }]
        }
    },
    "actions": {
        "toggle": {
            "@type": "saref:ToggleCommand",
            "forms": [{
                "href": "https://mylamp.example.com/toggle"
            }]
        }
    },
    "events": {
        "overheating": {
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh"
            }]
        }
    }
}
```

Figura 1.7: Esempio JSON Thing Description
Fonte: W3C Architecture [7]

1.3 Arrowhead

Arrowhead è un framework orientato ai servizi il cui obiettivo è supportare in modo efficiente lo sviluppo, la diffusione e il funzionamento di sistemi cooperativi interconnessi. Gli elementi costruttivi del framework sono sistemi che forniscono e consumano servizi e che cooperano come sistemi di sistemi. I sistemi più importanti sono Service Registry, Authorization e Orchestrator e sono chiamati Core Services. Questi possono essere utilizzati da qualsiasi sistema che segue le linee guida del framework Arrowhead [6]. L'Authorization è un sistema che contiene due database: il primo descrive quali applicazioni di sistema possono consumare un determinato servizio, il secondo descrive quali cloud locali possono consumare i servizi di un determinato cloud. Attraverso questo sistema è possibile quindi definire quali componenti hanno determinate autorizzazioni rispetto ad altri componenti. Questo sistema fornisce 2 Core services: AuthorizationControl e TokenGeneration, il primo fornisce 2 interfacce per la ricerca delle autorizzazioni:

- Intra-Cloud Authorization: definisce i permessi tra un sistema consumatore e un fornitore nello stesso cloud locale per un servizio specifico.
- Inter-Cloud Authorization: definisce i permessi affinché un Cloud esterno utilizzi un Servizio specifico dal Cloud Locale.

Il secondo genera i token utilizzati per gestire le autorizzazioni.
L'Orchestrator è il sistema che fornisce il collegamento tra sistemi, il suo scopo principale è fornire ai sistemi le informazioni riguardo dove devono connettersi. Il risultato include regole che diranno al Sistema, che agisce come consumatore del servizio, a quale/i Sistema/i del fornitore di servizi deve connettersi e come. Tali regole di orchestrazione includono:

- Informazioni riguardo l'accessibilità ad un determinato fornitore di servizi, ad esempio fornendo indirizzo di rete e porta.
- Dettagli dell'istanza del servizio che riguardano il fornitore, ad esempio URL e metadati.
- Informazioni relative ai permessi, ad esempio token di accesso e firma.
- Informazioni aggiuntive necessarie per stabilire la connessione.

Queste regole vengono fornite al sistema che farà da consumatore in 2 modi: possono essere richiesti dal sistema stesso attraverso una richiesta "pull" o l'Orchestrator stesso può aggiornare il sistema quando è necessario attraverso una richiesta "push". Nella versione di Arrowhead 4.0 viene implementato

solo il metodo "pull" per cui l'Orchestrator negozierà con gli altri sistemi core mentre cerca di facilitare la nuova richiesta di servizio. Questo diventa necessario in due casi:

- Quando è richiesta la responsabilità per tutti i sistemi nel cloud locale: in questo caso le connessioni non possono essere stabilite senza la conoscenza, l'approvazione e gli eventi di orchestrazione registrati dei Core Systems.
- Ragioni di QoS e gestione delle risorse: le connessioni peer-to-peer ad hoc non possono essere consentite in determinate reti gestite e scenari di distribuzione, ogni tentativo di connessione deve essere adeguatamente autorizzato e le sue aspettative di QoS gestite.

In questi casi, quando l'orchestrator è l'unico punto di ingresso per stabilire nuove connessioni all'interno del Cloud locale, i sistemi non hanno nessuna possibilità di saltare nessuno dei loop di controllo con i Core Systems. Quando questi problemi di sicurezza e protezione non sono presenti, il processo di orchestrazione viene ridotto e queste interazioni tra i sistemi principali potrebbero essere limitate. Quindi l'orchestrator fornisce due servizi principali e può consumarne molti altri, ma almeno due, questo dipende dalla versione. Questa figura illustra le interfacce obbligatorie e opzionali di questo sistema.

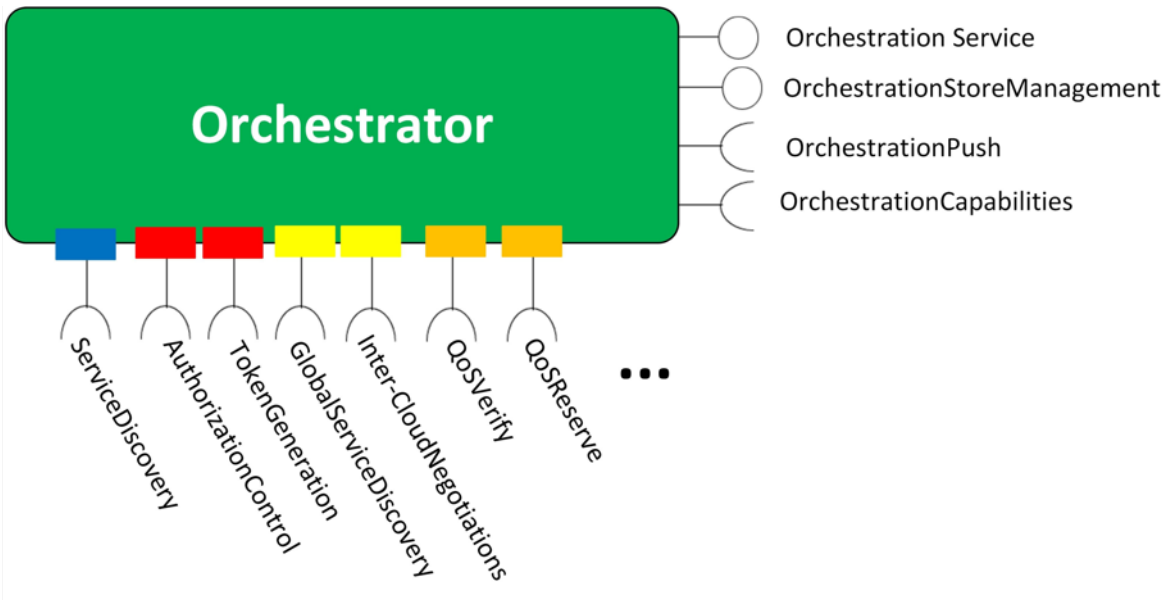

Figura 1.8: Interfacce obbligatorie e opzionali Orchestrator
Fonte: W3C Architecture [7]

Nell'immagine sovrastante vediamo che i Servizi forniti dall'Orchestrator sono:

- Orchestration Service
- OrchestrationStoreManagement Service

Mentre i Servizi consumati possono variare, a seconda dell'istanziazione/installazione del sistema. Ad esempio, l'orchestrator può utilizzare i servizi di:

- ServiceDiscovery Service dal ServiceRegistry
- AuthorizationControl Service dall'Authorization System
- TokenGeneration Service dall'Authorization System
- GlobalServiceDiscovery dal Gatekeeper
- Inter-CloudNegotiations dal Gatekeeper
- QoSVerify dal QoS Manager
- QoSReserve dal QoS Manager

L'orchestrator utilizza principalmente servizi da altri sistemi core per soddisfare la sua funzionalità primaria cioè fornire obiettivi di connessione per sistemi in modo sicuro. Durante questo processo detto di "orchestrazione" l'Orchestrator facilita la creazione della richiesta di servizio da un sistema o elabora un "push" della "coreografia" a livello di sistema di sistemi (SoS) dal Plant Description Engine (detto "Choreographer"). In quest'ultimo caso, l'Orchestrator System utilizza OrchestrationPush dai sistemi applicativi interessati per fornire loro un set rinnovato di regole di connessione. All'interno dell'Orchestrator, è presente un database che acquisisce i collegamenti in fase di progettazione tra i sistemi, chiamato Orchestration Store. Gli operatori del Cloud e di altri strumenti di progettazione di System-of-Systems ("SoS Choreographers") possono modificare le regole memorizzate nell'Orchestration Store, mentre altri sistemi generici no. Il servizio ServiceDiscovery viene utilizzato per pubblicare il servizio di Orchestrazione nel Service Registry. Questo servizio viene utilizzato anche per interrogare il Service Registry e recuperare informazioni su altri sistemi. I servizi del sistema di Authorization possono essere utilizzati per verificare il controllo degli accessi e implementare altre attività amministrative relative alla sicurezza mentre i servizi del QoS Manager possono essere utilizzati per gestire accordi e configurazioni di

qualità del servizio a livello di dispositivo, rete e servizio. Orchestrator può essere utilizzato in due modi: il primo utilizza regole predefinite provenienti dal database di Orchestrator Store per trovare i provider appropriati per il consumatore, la seconda opzione è "l'orchestrazione dinamica", in questo caso il servizio principale ricerca l'intero cloud locale per trovare i provider corrispondenti.
Il Service Registry è un servizio che fornisce un database che colleziona tutti i servizi attualmente offerti all'interno del cloud locale. Gli scopi di questo sistema sono:

- Permettere agli altri sistemi di registrare quali Servizi offrono al momento, rendendo l'annuncio disponibile ad altri sistemi sulla rete dando anche la possibilità di aggiornare o rimuovere gli annunci, se necessario.

- Permettere agli altri sistemi di utilizzare la funzionalità di ricerca del Service Registry per trovare i Public Core System Service offerti nella rete, altrimenti sarebbe necessario utilizzare l'Orchestrator.

Il Service Registry fornisce un Core Service, il Service Discovery. Ci sono due scenari di casi d'uso connessi al Service Registry:

- Registrazione e deregistrazione dei servizi,

- Ricerca attraverso interrogazione del Service Registry.

Il metodo di registrazione permette di registrare i servizi che conterranno vari metadati e un endpoint fisico. I vari parametri rappresentano le informazioni sull'endpoint che dovrebbero essere registrate.
Il metodo di deregistrazione permette di eliminare l'istanza di un servizio precedentemente registrato nel Service Registry, il parametro d'istanza rappresenta le informazioni sull'endpoint che dovrebbero essere rimosse.
Il metodo query permette di trovare e tradurre il nome simbolico di un servizio nell'indirizzo di un endpoint fisico, ad esempio un indirizzo IP e una porta. Il parametro query viene utilizzato per richiedere un sottoinsieme di tutti i servizi registrati che soddisfano la richiesta dell'utente del servizio. L'elenco restituito contiene gli endpoint dei servizi che hanno soddisfatto la query.
Il Service Registry può essere protetto utilizzando il protocollo HTTPS. Se viene avviato in modalità protetta, verifica se il sistema possiede un certificato di identità X.509 appropriato e se tale certificato è conforme a Arrowhead. La struttura del certificato e le linee guida per la creazione garantiscono che:

- Il sistema è correttamente avviato nel cloud locale,

- Il sistema appartiene a questo Cloud locale,
- Il sistema ha quindi automaticamente il diritto di registrare i propri servizi nel registro.

Se questi criteri vengono soddisfatti, il messaggio di registrazione o rimozione del sistema viene elaborato. Un sistema può eliminare o modificare solo le voci che lo contengono come fornitore di servizi.

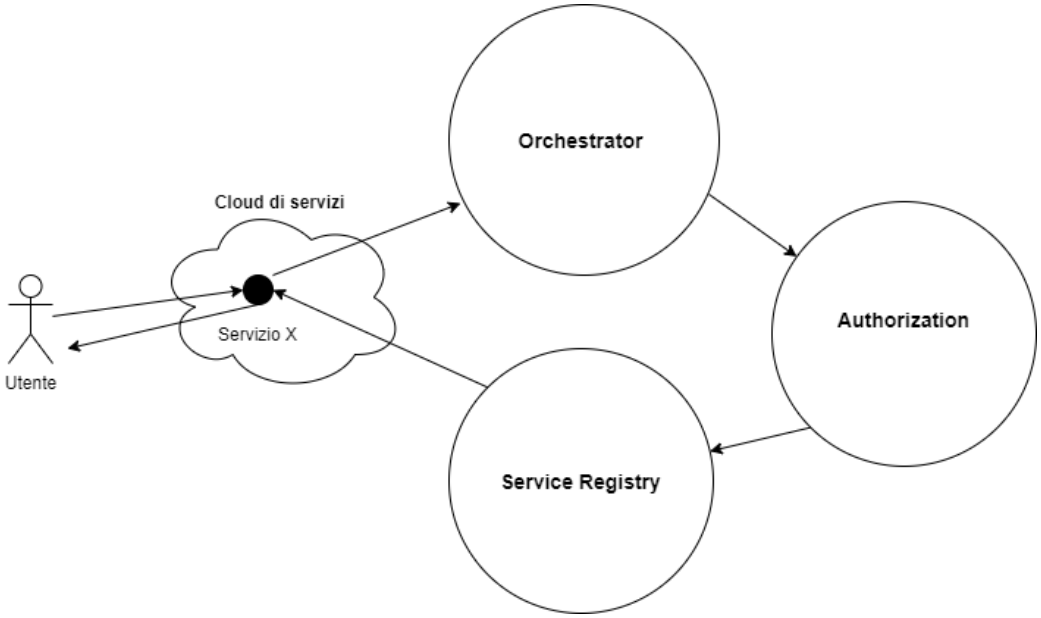

Figura 1.9: Interazione tra i Core Services durante la richiesta di utilizzo di un servizio da parte di un utente

Nella figura sovrastante è possibile vedere un esempio in cui un utente richiede l'utilizzo di un servizio X presente all'interno di un cloud locale contenuto nel Service Registry. Quando l'utente invierà la richiesta all'Arrowhead Service Registry per l'utilizzo del servizio verrà eseguito il processo di "orchestrazione" effettuato dall'Orchestrator in cui verranno verificati i permessi necessari per usufruire del servizio richiesto. Per effettuare la verifica l'Orchestrator comunicherà con l'Authorization ottenendo i permessi necessari per usare il servizio ed i permessi posseduti dall'utente, successivamente l'esito dei controlli verrà comunicato al Service Registry che darà il verdetto all'utente, per cui se l'utente avrà i permessi necessari il servizio sarà disponibile per l'utilizzo altrimenti l'utilizzo gli verrà negato.

1.4 Motivazioni

Ciò che mi ha spinto a realizzare questo progetto è stata la necessità molto visibile nell'ambito della ricerca di avere un framework che potesse integrare i vantaggi del Web of Thing con quelli di Arrowhead realizzando un framework che, oltre a rendere interoperabile l'utilizzo delle Thing all'interno di Arrowhead, potesse essere utilizzato per il monitoraggio strutturale in modo tale da poter verificare la "salute" degli edifici evitando inutili manutenzioni nei momenti inopportuni e notificando all'uomo quando c'è una reale necessità di manutenzioni e quando l'edificio potrebbe essere un reale pericolo per la salute dell'uomo. Esistono altri progetti che permettono l'interoperabilità di Web Thing come il WoT Store, un framework che abilita la scoperta semantica di applicazioni per il WoT e che permette di installarle su Thing compatibili con la Thing Description attraverso un processo automatico che non implica la necessità di ulteriori operazioni di configurazione o codifica [5]. Il WoT Store inoltre offre un catalogo di applicazioni che implementano il comportamento della Thing facendo si che l'utente può interrogare lo store e scaricare le applicazioni compatibili con le sue Things. Un altro esempio è questo lavoro in cui gli autori hanno sviluppato un misuratore smart, cioè un sensore IoT che può essere integrato come servizio del framework Arrowhead [3] o il CATSWoTS, un Social Context-aware formato da Web Things basato sull'idea di un sistema incentrato sull'utente in cui i profili di ogni Thing siano mantenuti a livello centralizzato e anche a livello distribuito in cui l'aspetto social permette alle Things di ottenere informazioni dalle altre Things [1]. A livello tecnico, dopo aver conosciuto i suddetti lavori mi sono reso conto che nessuno di essi porta vantaggi riguardo l'usabilità, risultando quindi lavori sperimentali utilizzabili solo da pochi utenti escludendo gli addetti ai lavori. Il mio progetto, a differenza di quelli citati, essendo utilizzabile da tutti i possessori di uno smartphone Android ha una copertura di utenti molto elevata considerando che nel mercato degli smartphone gli smartphone Android ricoprono la fetta più grossa. Inoltre grazie alla facilità d'uso l'utente con pochi tocchi sullo schermo potrà ottenere tutte le informazioni che desidera.

Capitolo 2

Architettura

2.1 Integrazione WoT e Arrowhead

Nonostante il grande potenziale del Web of Thing non tutti i dispositivi, soprattutto quelli più datati, possono usufruirne. Immaginando un ecosistema formato da un insieme di Web Things che collezionano i dati ottenuti da dei sensori, i consumatori potrebbero accedere a questi dati in 2 modi:

- Attraverso un dispositivo WoT-enabled che quindi ha la possibilità di interagire con altre Web Things,

- Attraverso un sistema che permetta l'interoperabilità tra WoT ed altri framework, come ad esempio Arrowhead.

Una possibilità è quindi quella di far comunicare il mondo del Web of Thing con il framework Arrowhead, il problema che si pone però è che il Web of Thing a causa della sua struttura non può esporre i suoi servizi direttamente con il framework Arrowhead a causa del fatto che l'interazione con il Web of Things avviene esclusivamente attraverso il consumo dell'endpoint della Web Thing, per questo motivo per realizzare l'interoperabilità tra il Web of Things e il framework Arrowhead è necessario un intermediario che faccia in modo che i servizi offerti dalla Web Thing possano essere utilizzati indirettamente, facendo in modo che il client per usufruire dei servizi forniti dalla Web Thing non comunichi direttamente con essa ma i risultati gli vengano forniti da un soggetto terzo. Di seguito è mostrata graficamente l'architettura utilizzata per la realizzazione del framework necessario all'interoperabilità WoT-Arrowhead.

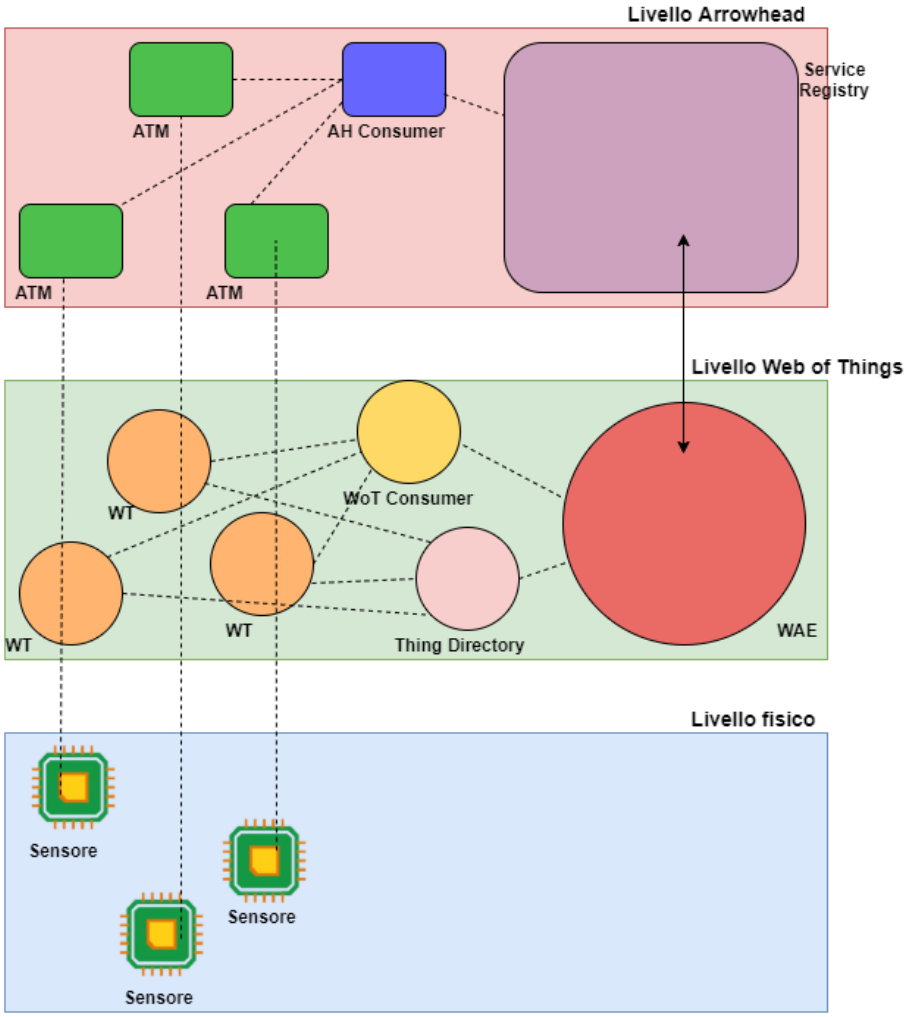

Figura 2.1: Rappresentazione grafica architettura WoT-Arrowhead

L'immagine mostra una struttura formata da tre livelli:

- Livello fisico: è il livello di cui fanno parte le componenti reali quindi i sensori o qualunque oggetto smart che abbia la possibilità di comunicare attraverso un qualunque protocollo.
- Livello Web of Things: in questo livello troviamo le Web Thing che rappresentano le astrazioni delle componenti reali presenti nel livello fisico e che comunicano direttamente con quest'ultimo. Le Web Thing dopo essere state prodotte ed esposte vengono inserite all'interno della

Thing Directory e rese inoltre disponibili all'utilizzo da parte dei consumatori facenti parte del mondo del Web of Things, infine, una volta inserite all'interno della Thing Directory, verranno gestite dal WoT Arrowhead Enabler che si occuperà di pubblicarle come servizi dell'Arrowhead framework e di renderle utilizzabili anche dai consumatori WoT-enabled.

- Livello Arrowhead: in questo livello troviamo gli Arrowhead Thing Mirror, che rappresentano l'anello di congiunzione tra il mondo WoT ed il mondo Arrowhead, infatti partendo dalla Web Thing viene creata una copia sotto forma di web server HTTP che comunica direttamente con la Web Thing e che permette ai consumatori del framework Arrowhead di utilizzare indirettamente le Web Thing attraverso il loro ATM tramite la pubblicazione sotto forma di servizi all'interno dell'Arrowhead Service Registry.

Ho sviluppato quindi alcuni componenti che permettono alle Thing di comunicare con Arrowhead attraverso un componente che fa da proxy della Thing ed un componente che ha il compito di ricercare nuove Thing, creare e registrare il proxy su Arrowead, queste componenti sono:

- Arrowhead Thing Mirror (ATM): questa componente costituisce il server proxy che rappresenta la Web Thing rendendo quindi la Web Thing utilizzabile come se fosse un servizio Web HTTP. Ciò che fa l'ATM è creare un server HTTP per ogni Web Thing mettendolo in ascolto ad un determinato indirizzo IP ed una determinata porta, dopodichè il client HTTP potrà utilizzare la Web Thing raggiungendo l'URL del suo server proxy ed aggiungendo la parte query dell'URL in base alla richiesta che vorrà fare alla Web Thing che dipende dalle sue necessità, a quel punto l'ATM inoltrerà la richiesta ricevuta alla Web Thing che la elaborerà e ritornerà il risultato all'ATM che lo mostrerà al client HTTP. Un'altra funzione effettuata dall'ATM consiste nel registrare il server HTTP come servizio del framework Arrowhead inserendo come endpoint l'indirizzo IP e la porta del server HTTP.

- Thing Directory (TD): la Thing Directory è una componente che contiene gli endpoint di tutte quelle Web Thing che devono essere inserite all'interno dell'Arrowhead framework. Una Web Thing può richiedere di essere inserita all'interno dell'Arrowhead framework richiamando l'azione aggiungi della Thing Directory o, in alternativa, può essere inserita di default nella lista delle Web Thing conosciute dalla TD.

- WoT Arrowhead Enabler (WAE): questa componente contiene la lista delle Web Thing conosciute e due azioni: query che ritorna al client la lista di tutti i servizi presenti su Arrowhead e ricerca che cerca, all'interno della Thing Directory, endpoint che non conosce e, nel caso in cui li trova, per ognuno di essi crea un ATM assegnandogli un proprio URL e registrandolo all'interno del framework Arrowhead, infine quando il WAE verrà spento i servizi da lui registrati verranno deregistrati in quanto non più disponibili.

- WoT Client: il WoT Client è un componente che, ad intervalli regolari, richiama l'azione di ricerca del WAE in modo tale da verificare se sono state aggiunte nuove Web Thing alla Thing Directory per poter, nel caso in cui siano presenti nuove Web Thing, creare i relativi ATM e registrarli su Arrowhead.

2.2 Interazione tra le componenti

Le varie componenti precedentemente descritte interagiscono tra di loro dal momento in cui nasce una Web Thing all'utilizzo da parte di un client HTTP, pur essendo componenti che utilizzano linguaggi e protocolli diversi la comunicazione avviene grazie al meccanismo di mirroring svolto da parte dell'Arrowhead Thing Mirror che funge da vero e proprio "ponte" tra l'ambiente di sviluppo relativo al Web of Thing e l'ambiente di sviluppo basato sui servizi del framework Arrowhead. Nel grafico non è presente il WotClient in quanto il suo compito si limita a consumare il WoT Arrowhead Enabler e richiamarne l'azione di ricerca ad intervalli regolari per cui sia a livello di interazioni che di costi è stato considerato trascurabile.

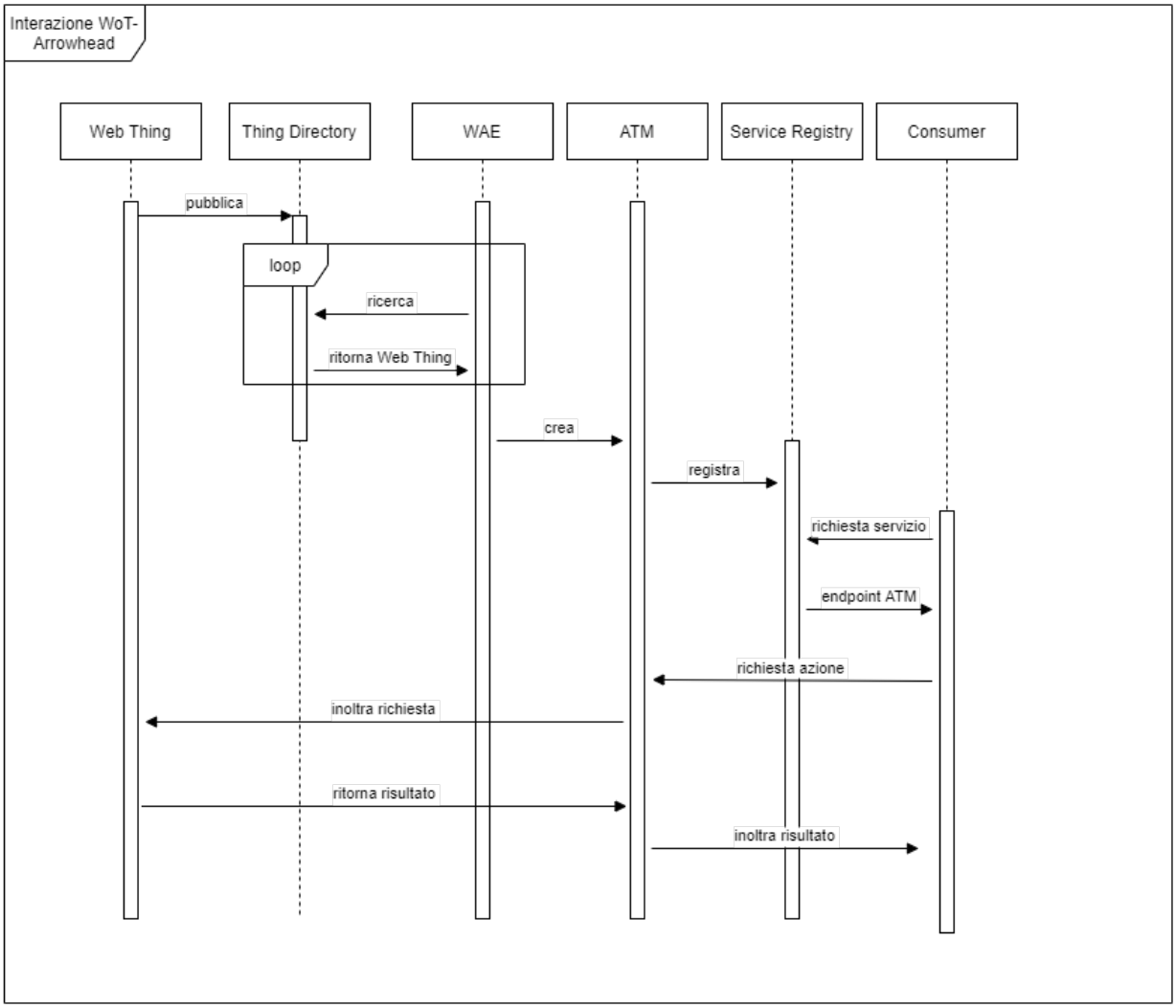

Figura 2.2: Diagramma di sequenza: Interazione WoT-Arrowhead

Il diagramma di sequenza sovrastante rappresenta le interazioni che intercorrono tra la nascita di una Web Thing ed il suo utilizzo da parte di un qualunque consumer HTTP.
Inizialmente la Web Thing viene esposta ottenendo un suo endpoint che pubblica all'interno della Thing Directory, sia per le Web Thing che per il WAE risulta molto semplice interagire con la Thing Directory in quanto ha un endpoint fisso ed il suo unico scopo è quello ricevere endpoint, verificare se li possiede, altrimenti aggiungerli alla lista dei suoi endpoint. Il compito del WAE consiste nel verificare se la Thing Directory ha aggiunto nuovi endpoint non ancora conosciuti dal WAE e quindi che non possiedono un ATM, per

questo motivo è presente un loop in cui il WAE consuma la TD per leggere la sua lista di Web Thing conosciute e, nel caso in cui ne trovasse qualcuna che lui non conosce creerebbe un ATM con un suo indirizzo IP ed una sua porta, il WAE possiede inoltre un altro compito che viene sfruttato unicamente da altre Web Thing e consiste nell'invocazione dell'azione query, questa azione ritorna al chiamate la lista dei servizi presenti su Arrowhead. Dopo essere stato creato, l'ATM consuma la Web Thing di cui ha creato il server proxy lasciandolo in ascolto di richieste da parte di client HTTP e registra il suo URL formato da indirizzo IP e porta come servizio Arrowhead all'interno del Service Registry. Infine abbiamo il consumer, lui invierà una richiesta al servizio registrato sul Service Registry in base alle sue necessità ed otterrà l'URL dell'ATM collegato. Una volta ottenuto l'URL richiederà l'azione di cui ha bisogno all'ATM che inoltrerà la richiesta alla Web Thing a cui fa riferimento. A questo punto la Web Thing produrrà il risultato e lo fornirà all'ATM che, a sua volta, lo inoltrerà al consumer.

2.3 WoT App

E' stata sviluppata un'app per smartphone Android che permette di interfacciarsi con le Web Thing registrate come servizi sul framework Arrowhead, inoltre quest'app contiene una funzione di monitoraggio che permette di monitorare lo stato di un edificio realmente esistente attraverso sei sensori, di cui tre accelerometri e tre giroscopi, registrati come tre Web Thing. Grazie ai sensori l'app effettua delle rilevazioni e registra tutti i valori ottenuti all'interno di un database. Grazie all'ampia mole di dati raccolti, costruisce dei grafici che mostrano l'andamento dei valori rilevati. Tramite la pagina relativa alle impostazioni dell'app è possibile decidere l'intervallo che intercorre tra una rilevazione ed un'altra e la soglia che non deve essere superata da ogni sensore, nel caso in cui la soglia viene superata l'utente riceve una notifica che lo informa del superamento della soglia prefissata. Per ottenere i valori da parte del sensore quindi possiamo o accederci manualmente selezionando il sensore di interesse e la proprietà dalla relativa pagina o abilitare la rilevazione automatica nelle impostazioni in cui ogni X secondi prefissati dall'utente l'app effettuerà, su tutti i sensori, una rilevazione delle proprietà selezionate dall'utente.
I componenti che vengono utilizzati per la realizzazione dell'app sono:

- Database: è stato creato un database contenete una tabella chiamata Sensore che contiene al suo interno i dati relativi alla misurazione e a quale sensore ha effettuato tale misurazione,

- Lista dei servizi: viene realizzata una lista contentente i servizi. Per popolarla viene fatta una richiesta HTTP ad Arrowhead in modo da ottenere i servizi sotto forma di JSON e gestirli al meglio,

- Menù a tendina personalizzato: viene utilizzato per selezionare, dalla pagina della Web Thing, le azioni da eseguire o le proprietà da visualizzare. È stato personalizzato nel suo comportamento interno in modo tale che eseguisse le sue funzioni anche quando l'utente seleziona una voce uguale a quella precedentemente selezionata, questo è stato necessario perchè di default questo componente effettua il suo comportamento solo quando viene selezionata una voce diversa da quella precedentemente selezionata.

- Grafico a linea: viene realizzato un grafico a linea che mostra l'andamento del parametro preso in oggetto nel tempo.

- Intent service: viene realizzato un Intent service per permettere il monitoraggio in background, effettuandolo sia mentre l'utente sta utilizzando l'app facendo qualche altra operazione, sia quando l'app viene messa in background.

Una delle operazioni più importanti dell'app è il monitoraggio che è direttamente collegato all'operazione di rilevazione. Questa operazione è illustrata nel diagramma di sequenza sottostante.

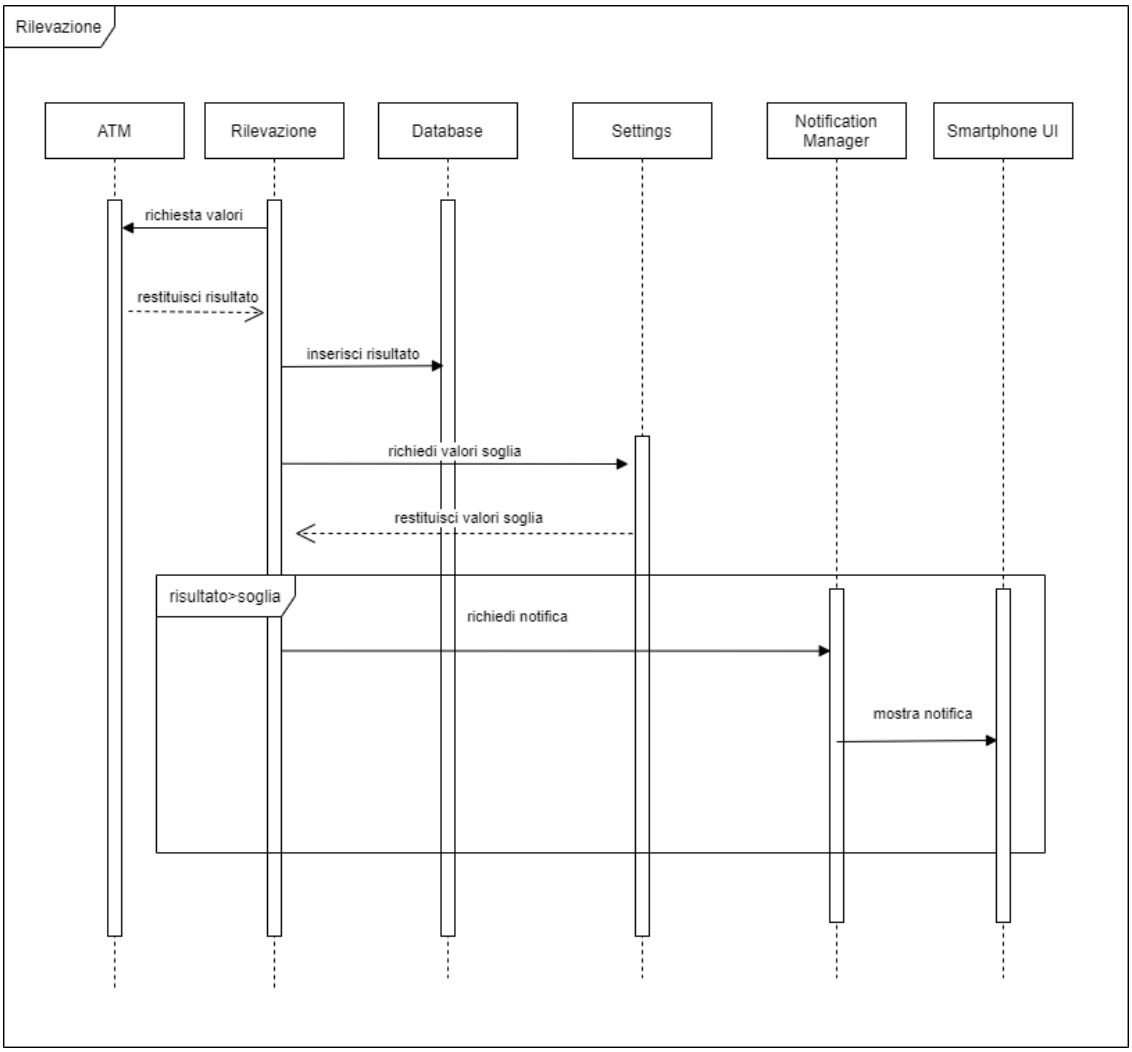

Figura 2.3: Diagramma di sequenza: processo di rilevazione

Da questo diagramma di sequenza si evincono le interazioni che intercorrono nel momento in cui viene avviata la procedura di monitoraggio che si può concludere in due modi diversi, in base al tipo di risultato ottenuto dalla rilevazione.
Inizialmente la classe Rilevazione, che ha il compito di ottenere i dati dal sensore, fa una richiesta HTTP GET all'ATM relativo al sensore in esame, dopo aver ottenuto i risultati, la classe Rilevazione li inserisce all'interno del database, successivamente richiede i valori soglia relativi ai parametri di cui è attivo il monitoraggio ed infine, una volte ottenuti i valori soglia dalle Impostazioni, confronta i risultati con le soglie e, nel caso in cui la soglia è inferiore al risultato richiede alla classe che gestisce le notifiche di creare una

notifica in modo tale da informare l'utente che la soglia per quel determinato parametro è stata superata. Se viceversa la soglia è maggiore del risultato ottenuto l'algoritmo si interrompe dopo aver inserito i risultati sul database. É essenziale che i risultati vengano immediatamente inseriti all'interno del database perchè successivamente potrebbero essere utilizzati per creare un grafico a linee che mostri l'andamento dei valori del sensore al passare del tempo.

Capitolo 3

Implementazione

In questo capitolo verrà descritta l'implementazione di tutte le componenti necessarie al fine del corretto funzionamento del sistema sopra descritto dal punto di vista architetturale, verranno mostrati snippet di codice e ne verrà fornita una spiegazione dettagliata sul loro utilizzo.
Per la realizzazione del progetto sono stati utilizzati i linguaggi di programmazione Java e JavaScript. Java è stato utilizzato per la realizzazione dell'applicazione mobile in Android mentre JavaScript, attraverso la runtime Node JS, è stato utilizzato per la realizzazione delle componenti legate al Web of Things: WAE, ATM, TD e WoTClient. Ho deciso di utilizzare Node JS perchè permette una facile gestione dei server e degli eventi generati sul framework. Per quanto riguarda le librerie utilizzate per realizzare l'applicazione mobile ho deciso di utilizzare la libreria Volley per gestire l'utilizzo delle API REST, il framework Room per gestire la persistenza dei dati e la libreria MPAndroidChart[1] per la gestione dei grafici. Per quanto riguarda le librerie utilizzate per realizzare il WAE ho utilizzato il pacchetto HTTP per gestire il server ATM, il pacchetto URL per la gestione degli eventi in base all'URL, il pacchetto Child Process[2] per la creazione dei vari ATM, il pacchetto Process[3] per gestire la registrazione e deregistrazione dei servizi sul Service Registry ed il pacchetto Axios[4] per la gestione delle API REST.
Per la realizzazione del WAE sono stati sviluppati 3 componenti: una Thing Directory cioè una Web Thing che ha il compito di memorizzare gli endpoint delle Web Thing presenti, un WotClient cioè una Web Thing il cui unico compito è quello di richiamare l'azione ricerca del WAE, ed il WAE che contiene al suo interno un ATM (Arrowhead Thing Mirror) cioè un server proxy che

[1]https://github.com/PhilJay/MPAndroidChart
[2]https://nodejs.org/api/childprocess.html
[3]https://nodejs.org/api/process.html
[4]https://www.npmjs.com/package/axios

permette di usufruire dei servizi forniti dalla Web Thing attraverso chiamate REST.

3.1 Thing Directory

```
WoT.produce({
    title: "td",
    description: "td",
    "@context": ["https://www.w3.og/2019/wot/td/v1", { "iot":
    "http://example.org/iot" }],
    "securityDefinitions": { "nosec_sc": { "scheme": "nosec"
   }},
    "security": "nosec_sc",
    properties: {
        listaThing: {
            type: "string",
            description: "lista thing conosciute",
            observable: true
      }
    },
    actions: {
      aggiungi: {
        description: "aggiungi thing alla lista thing
   conosciute",
        uriVariables: {
            indirizzo: { "type": "string" }
        }
      }
    },
    events: {
    }
})
```

Listing 3.1: Produzione Thing Directory

Come già detto, la TD è una Web Thing che possiede una proprietà chiamata ”Lista Thing” che raccoglie, sotto forma di stringa, tutte le Web Thing che conosce inserendole sotto forma di endpoint separati da un’andata a capo. Questa Web Thing possiede inoltre un’azione chiamata ”aggiungi” che richiede in input un parametro chiamato ”indirizzo” che ha il compito di verificare se l’indirizzo ricevuto in input (che rappresenta un endpoint di una Web Thing) è già presente nella lista delle Web Thing che la TD conosce, di seguito il frammento di codice relativo.

```
thing.setActionHandler("aggiungi", async (params,options) =>
   {
```

```
2           let l=await thing.readProperty("listaThing").then((
    val)=>{
3             return val;
4           });
5           if(l!= null){
6             var lista=l.split(/\r?\n/);
7             if(!lista.includes(options.uriVariables.indirizzo
    )){
8               await thing.writeProperty("listaThing",l+"\n"+
    options.uriVariables.indirizzo);
9             }
10          }
11          else{
12            await thing.writeProperty("listaThing",options.
    uriVariables.indirizzo);
13          }
14        });
```

Listing 3.2: Azione Aggiungi

In questo snippet vediamo il comportamento dell'azione "aggiungi". Ciò che fa è inizialmente leggere la sua stessa proprietà, lista thing, per verificare se sono presenti endpoint o meno. Nel caso in cui la lista sia vuota significa che si sta registrando il primo endpoint e che quindi sicuramente quest'ultimo non è già presente all'interno della lista; nel caso in cui la lista non è vuota viene verificato se l'endpoint ricevuto in input non è presente tra quelli contenuti dalla lista delle things, se questa affermazione risulta vera l'indirizzo ricevuto viene aggiunto alla proprietà Lista Thing della Thing Directory. A causa del fatto che nel mio progetto vengono prese in esame delle Web Thing realmente esistenti, quando la TD viene esposta viene effettuata una richiesta HTTP GET ad un indirizzo che contiene gli endpoint delle suddette Web Things e, una volta ottenuti, vengono inseriti all'interno della proprietà Lista Thing della TD. Di seguito lo snippet relativo.

```
1 axios.get("http://137.204.143.89:8080/")
2           .then(async function (response) {
3             var lt=response.data[0]+"\n";
4             for(var i=1;i<response.data.length;i++){
5               lt+=response.data[i]+"\n";
6             }
7             await thing.writeProperty("listaThing",lt);
8             await thing.readProperty("listaThing").then((val)
    =>{
9               return console.log("val:",val);
10            });
```

```
11              })
```

Listing 3.3: Richiesta GET ai sensori reali

3.2 WoT Arrowhead Enabler

Il WoT Arrowhead Enabler (WAE) è una Web Thing che possiede una property chiamata Lista Thing che contiene la lista delle Web Thing da lui conosciute e due actions: query e ricerca, la prima quando viene invocata dal consumatore ritorna la lista dei servizi registrati sul Service Registry di Arrowhead, la seconda ottiene la lista degli endpoint delle Web Thing da registrare come servizi tramite l'interazione con la Thing Directory, verifica se ha già registrato gli endpoint ottenuti e se risulta che alcuni endpoint non sono stati registrati crea, per ognuno di essi, un ATM di cui registra l'URL come servizio sull'Arrowhead Service Registry ed infine aggiorna la sua proprietà contenente la lista di Web Thing conosciute. Quando il WAE termina la sua esecuzione cancella i servizi precedentemente registrati relativi agli ATM.

```
1  WoT.produce({
2      title: "wae",
3      description: "wae",
4      "@context": ["https://www.w3.org/2019/wot/td/v1", { "iot"
       : "http://example.org/iot" }],
5      "securityDefinitions": { "nosec_sc": { "scheme": "nosec"
       }},
6      "security": "nosec_sc",
7      properties: {
8          listaThing: {
9              type: "string",
10             description: "lista thing conosciute",
11             observable: true
12       }
13     },
14     actions: {
15       query: {
16         description: "ottieni lista dei servizi ArrowHead"
17       },
18       ricerca:{
19         description:"cerca nuove thing basandoti sulla lista
       di web thing gia' conosciute"
20       }
21     },
22     events: {
23     }
```

```
24 })
```

Listing 3.4: Produzione WAE

L'azione *query* ritorna al consumatore la lista di tutti i servizi Arrowhead. La lista dei servizi viene ottenuta attraverso una richiesta HTTP GET sotto forma di array JSON.

```
1  thing.setActionHandler("query", (params) => {
2            return axios.get('http://137.204.57.93:8443/
     serviceregistry/mgmt/')
3            .then((response)=>{
4              return response.data;
5            })
6            .catch((error)=>{
7              console.log("Errore: ");
8              console.log(error);
9            });
10         });
```

Listing 3.5: Azione Query

L'azione *ricerca* inizialmente legge la proprietà del WAE chiamata Lista Thing in modo tale da ottenere gli endpoint delle Web Thing che già conosce, successivamente consuma la Thing Directory e ne legge la sua proprietà Lista Thing in modo tale da ottenere la lista di tutte le Web Thing connesse, dopodichè verifica se tra la lista di endpoint ottenuta dalla TD ne sono presenti alcune che non si trovano nella lista degli endpoint del WAE, se la risposta è affermativa viene eseguito il metodo ATM, altrimenti non succede nulla.

```
1    thing.setActionHandler("ricerca",async (params)=>{
2            let l=await thing.readProperty("listaThing").then((
     val)=>{
3              return val;
4            });
5            let b;
6            await WoTHelpers.fetch("http://localhost:8085/td").
     then(async (td) => {
7                try {
8                    let thing = await WoT.consume(td);
9                    b=await thing.readProperty("listaThing");
10               }catch(err){
11                 console.error("Script error:", err);
12               }
13           }).catch((err) => { console.error("Fetch error:",
     err);
14           });
15             var s=b.split(/\r?\n/);
16             var lista;
```

```
17              if(l==null){
18                ATM(s[0]);
19                await thing.writeProperty("listaThing",s[0]);
20              }
21              for(var i=0;i<s.length;i++){
22                l=await thing.readProperty("listaThing").then((
    val)=>{
23                  return val;
24                });
25                lista=l.split(/\r?\n/);
26                if(!lista.includes(s[i])){
27                    ATM(s[i]);
28                    await thing.writeProperty("listaThing",l+"\
    n"+s[i]);
29                }
30              }
31          });
```

Listing 3.6: Azione Ricerca

Il metodo ATM riceve in input l'endpoint della Web Thing di cui deve creare l'ATM, una volta richiamato questo metodo consuma la Web Thing ottenuta e successivamente crea un server HTTP mettendolo in ascolto ad un indirizzo IP prefissato con porta variabile, il valore della porta viene incrementato di uno per ogni nuovo ATM creato mentre l'indirizzo IP viene mantenuto. Il comportamento del server creato dipende da ciò che contiene il campo query dell'URL, infatti questo campo può contenere uno fra questi 3 parametri:

- Action: se l'URL contiene il parametro action il server invocherà l'azione contenuta nel suddetto parametro sulla suddetta Web Thing, se l'invocazione ritorna un risultato questo viene inviato come risposta sotto forma di oggetto JSON, sia nel caso in cui il risultato sia un oggetto JSON di sua natura sia nel caso in cui non lo sia, in questa seconda eventualità sarà lo stesso ATM a tradurre il risultato in un oggetto JSON in modo tale da ritornare al client come risposta un oggetto JSON anche in questo caso. Nel caso in cui invece l'invocazione dell'azione per un qualunque motivo non ritorni un risultato, verrà stampato a schermo un messaggio di errore.

- Property: se l'URL contiene il parametro property il server leggerà la proprietà contenuta nel suddetto parametro sulla suddetta Web Thing, se la lettura del parametro avviene correttamente il risultato viene inviato come risposta sotto forma di stringa JSON aggiungendo il nome della Web Thing che ha elaborato il risultato, sia nel caso in cui il risultato sia un oggetto JSON di sua natura sia nel caso in cui non lo sia, in

questa seconda eventualità sarà lo stesso ATM a tradurre il risultato in un oggetto JSON in modo tale da ritornare al client come risposta una stringa JSON anche in questo caso. In questo ATM troviamo controlli specifici riguardo determinate proprietà presenti nei sensori utilizzati per il monitoraggio strutturale, per questo motivo è stato inserito un controllo specifico per verificare se era stata richiesta la lettura di uno dei parametri riferiti ai suddetti sensori, questo è dovuto al fatto che questi sensori ritornano come risultato un array di valori per cui la gestione del JSON per quanto riguarda l'inserimento del nome della Web Thing che ha elaborato il risultato è stata modificata in modo tale da non ostacolare la gestione dei valori rilevati. Nel caso in cui invece la lettura della property per un qualunque motivo non ritorni un risultato, verrà stampato a schermo un messaggio di errore.

- Event: se l'URL contiene il parametro event il server sottometterà la Web Thing all'evento indicato all'interno del suddetto parametro e tornerà al client il risultato prodotto dalla Web Thing

```
function ATM(s){
WoTHelpers.fetch(s).then(async (td) => {
    try {
        let thing = await WoT.consume(td);
        console.info("==========");
        console.info(td);
        console.info("==========");
        let server=http.createServer(async (req,resp)=>{
          const queryObject = url.parse(req.url,true).query
 ;
          if(queryObject['action']!=undefined){
            var ris="";
            var r;
            let val=await thing.invokeAction(String(
 queryObject['action']));
            if(val==undefined){
              resp.end("Risultato azione "+String(
 queryObject['action'])+" ASSENTE");
            }
            else{
              if(typeof val==='object'){
                resp.writeHead(200, {'Content-Type': '
 application/json'});
                var presente=false;
                var valk=Object.keys(val);
                resp.write(val);
                resp.end();
              }
```

```
25                else{
26                  val='{"'+String(queryObject['action'])+'":"
  '+val+'"}';
27                  resp.end(val);
28                }
29              }
30            }
31            if(queryObject['property']!=undefined){
32              var ris="";
33              let val=await thing.readProperty(String(
  queryObject['property']));
34              if(val==undefined){
35                resp.end("Risultato proprieta' "+String(
  queryObject['property'])+" ASSENTE");
36              }
37              else{
38                if(typeof val==='object'){
39                  resp.writeHead(200, {'Content-Type': '
  application/json'});
40                  if(String(queryObject['property'])=="
  AccelerationSamples"||String(queryObject['property'])=="
  GyroscopeSamples"){
41                    val["0"].nome=td.title;
42                  }
43                  else{
44                    val.nome=td.title;
45                  }
46                  resp.write(JSON.stringify(val));
47                  resp.end();
48                  var presente=false;
49                  var valk=Object.keys(val);
50                }
51                else{
52                  var risposta='{"'+String(queryObject['
  property'])+'":"'+val+'","nome":'+'"'+td.title+'"}';
53                  resp.end(risposta);
54                }
55
56              }
57            }
58            if(queryObject['event']!=undefined){
59              thing.subscribeEvent(String(queryObject['event'
  ]),(val)=>{
60                resp.writeHead(200, {'Content-Type': '
  application/json'});
61                if(val==undefined){
62                  resp.end("Risultato evento "+String(
  queryObject['event'])+" ASSENTE");
63                }
```

```
                else{
                  resp.end("Risultato evento "+String(
    queryObject['event'])+" : "+val);
                }
              });
            }
          }).listen(port,ip,(error)=>{
              if(error){
                return console.log("Errore: ",error);
              }
              else{
                return console.log("Il server funziona all'
    indirizzo: ",server.address().address+":"+server.address()
    .port);
              }
          });
          registra(JSON.stringify(td),ip,port);
          port++;
      }catch(err){
        console.error("Script error:", err);
      }
  }).catch((err) => { console.error("Errore durante la Fetch:
    ", err);
  });
}
```

Listing 3.7: Funzione ATM

Dopo aver messo il server in ascolto viene invocata la funziona registra ed incrementato il numero di porta che sarà la porta utilizzata per il prossimo ATM. Il metodo registra ottiene in input la Thing Description della Web Thing sotto forma di stringa con sintassi JSON, l'indirizzo IP ed il numero di porta; il suo compito consiste nell'effettuare una richiesta HTTP POST per registrare un nuovo servizio su Arrowhead fornendo come Service Definition e System Name il nome della Web Thing, come indirizzo e come porta i parametri ricevuti in input e come metadata la Thing Description della Web Thing sotto forma di stringa con sintassi JSON. Infine, se la richiesta va a buon fine, viene inserito l'ID del servizio appena creato in un array formato da tutti gli ID dei servizi creati che verrà successivamente utilizzato per deregistrare i servizi quando verrà terminata l'esecuzione del WAE.

```
function registra(thing,ip,port){
let thing0=JSON.parse(thing);
axios.post('http://137.204.57.93:8443/serviceregistry/mgmt'
  , {
      serviceDefinition: thing0.title,
      providerSystem:{
        systemName: thing0.title,
```

```
          address: ip,
          port: port
        },
        secure: 'NOT_SECURE',
        metadata: {
          additionalProp1: thing
        },
        interfaces:['HTTPS-INSECURE-JSON']
      })
      .then(function (response) {
        id.push(response.data.id);
      })
      .catch(function (error) {
        console.log("Errore");
      });
}
```

Listing 3.8: Funzione Registra

Il metodo deregistra è un metodo asincrono che esegue un ciclo per un numero di volte pari al numero dei servizi registrati dal WAE, all'interno del ciclo viene effettuata una richiesta HTTP DELETE all'Arrowhead Service Registry che permette di rimuovere dal Service Registry i servizi precedentemente registrati dal WAE.
Infine, quando il WAE viene esposto viene creato un processo figlio utilizzando l'NPM Child Process che esegue il WoT Client. Il suo compito è quello di consumare il WAE ogni N secondi ed invocare l'azione di ricerca in modo tale da vedere se nel tempo trascorso tra un'invocazione e l'altra sono state inserite nuove Web Thing all'interno della Thing Directory. Dall'immagine del WoTClient è possibile osservare come il WAE abbia un endpoint fisso e ben preciso, questo è necessario in quanto altrimenti non sarebbe possibile da parte del WoTClient consumarlo.

```
  WoTHelpers.fetch("coap://localhost:5684/wae").then(async (
   td) => {
    try {
        let thing = await WoT.consume(td);
        await thing.invokeAction("ricerca");
    }catch(err){
      console.error("Script error:", err);
    }
}).catch((err) => { console.error("Fetch error:", err);
});
```

Listing 3.9: Comportamento WoT Client

3.3 WoT App

Per la realizzazione dell'applicazione Android chiamata WoT App sono stati sviluppati vari componenti fra cui:

- Un database che colleziona i dati ottenuti dalle rilevazioni effettuate dai sensori,
- una Recycler View che effettua una richiesta GET ottenendo un oggetto JSON contente al suo interno tutti i servizi registrati sull'Arrowhead Service Registry utilizzabili da smartphone,
- un'Activity che permette di interagire con la Web Thing selezionando, tramite Spinner, le proprietà da visualizzare o le azioni da eseguire e che attraverso un bottone permette di visualizzare lo storico relativo alla proprietà visualizzata,
- un'Activity che raccoglie lo storico delle rilevazioni effettuate e ne mostra l'andamento attraverso un grafico,
- un'Activity dedicata alle impostazioni personalizzabili dall'utente in cui quest'ultimo può decidere quali parametri monitorare ed ogni quanti secondi effettuare una rilevazione,
- un'Intent Service che si occupa di effettuare le rilevazioni dai sensori tramite una richiesta HTTP GET all'ATM relativo ad ogni sensore e, una volta ricevuto il valore rilevato, lo confronta con i valori soglia scelti nelle impostazioni verificando se la soglia è minore del valore rilevato, in questo caso viene richiamato un broadcast receiver che creerà una notifica,
- un Broadcast Receiver che avverte l'utente che sono stati rilevati dati anomali,
- un Notification Helper che permette la creazione del canale di notifica e della notifica in sè.

3.4 Il Database

È stato realizzato un database utilizzato il framework Room con all'interno una tabella chiamata Sensore le cui colonne sono:

- ID: rappresenta l'ID univoco della rilevazione effettuata sul sensore, è la chiave primaria della tabella,

- Data: rappresenta la data in cui è stata effettuata la rilevazione,
- Nome: rappresenta il nome del sensore da cui è stata fatta la rilevazione,
- Proprietà: rappresenta la proprietà rilevata dal sensore, come ad esempio accelerometro o giroscopio,
- Valore composto: rappresenta il valore ottenuto dai sensori triassiali,
- Valore: rappresenta il valore ottenuto dai sensori

```
  @Entity
public class Sensore {
    @PrimaryKey(autoGenerate = true)
    private int id;
    @TypeConverters(DateConverter.class)
    @NonNull
    private Date data;
    private String nome;
    private String proprieta;
    @Embedded(prefix = "valcomp")
    private ValComposto vc;
    private String valore;
```

Listing 3.10: Definizione tabella Sensore

Per interagire con la tabella del Database è stata realizzata una DAO chiamata DaoSensore che contiene 3 query:

- La prima query, chiamata Inserisci Valore permette di inserire una nuova rilevazione all'interno del database,
- La query Filtra permette di selezionare unicamente le rilevazioni di una determinata proprietà effettuate da un determinato sensore,
- La query Filtra Tutto permette di selezionare tutte le rilevazioni presenti all'interno della tabella

```
@androidx.room.Dao
public interface DaoSensore {
    @Insert
    public void InserisciValore(Sensore valore);
    @Query("SELECT * FROM Sensore WHERE upper(nome)=upper(:
   nome) AND proprieta=:comando")
    public List<Sensore> filtra(String comando,String nome);
    @Query("SELECT * FROM Sensore")
    public List<Sensore> filtraTutto();
}
```

Listing 3.11: Definizione DaoSensore

3.5 Main Activity

La Main Activity rappresenta la homepage cioè la schermata che si raggiunge non appena si avvia l'app, al suo interno troviamo l'istanziazione della toolbar contenete il menù per raggiungere la pagina relativa alle impostazioni, l'istanziazione del database, l'avvio dell'Intent Service relativo alla rilevazione ed un controllo che verifica se è la prima volta che viene avviata l'app, nel caso in cui questo controllo ha come risultato vero viene avviata l'activity relativa alle impostazioni in modo tale da inserire le preferenze dell'utente prima di iniziare ad usare veramente l'applicazione. Infine troviamo un pulsante che permette di raggiungere l'activity Visualizza Servizi. Infine, nella onDestroy dell'activity è stato inserita l'operazione di chiusura dell'Intent service in esecuzione.

```
@Override
    public void onDestroy() {
        super.onDestroy();
        Log.v("MYAPP","Destroy main");
        stopService(new Intent(MainActivity.this, Rilevazione
    .class));
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tb=findViewById(R.id.toolbar);
        setSupportActionBar(tb);
        db= Room.databaseBuilder(getApplicationContext(), com
    .example.wot2.Database.class,"appdb").build();
        Intent is=new Intent(this, Rilevazione.class);
        ContextCompat.startForegroundService(this,is);
        sharedPreferences = getSharedPreferences(sharedPrefs,
     MODE_PRIVATE);
        boolean isFirstTime=sharedPreferences.getBoolean("
    isFirstTime",true);
        if(isFirstTime){
            startActivity(new Intent(this,SettingsActivity.
    class));
        }
        final Button visualizza=findViewById(R.id.Visualizza)
    ;
        visualizza.setOnClickListener(new View.
    OnClickListener() {
            @Override
            public void onClick(View view) {
```

```
26                  Intent i=new Intent(getApplicationContext(),
    VisualizzaServizi.class);
27                  startActivity(i);
28              }
29          });
30      }
```

Listing 3.12: Codice onCreate e onDestroy Main Activity

3.6 Settings Activity

Questa activity permette all'utente di gestire le sue preferenze riguardo le soglie che non devono essere superate quando viene rilevato un determinato valore e riguardo l'intervallo che deve intercorrere tra una rilevazione ed un'altra tramite le Shared Preferences. La scelta dei parametri da monitorare viene fatta attraverso un Switch Button che può avere valore ON oppure OFF, nel caso in cui è ON verrà mostrata la soglia attuale ed un bottone che permette di avviare un Alert per modificare la soglia. È inoltre presente un bottone per modificare l'intervallo di rilevazione per cui premendolo verrà avviato un Alert che permetterà di modificare tale intervallo espresso in secondi.

```
1  public AlertDialog creaAlert(String titolo, final String
       messaggio, final String campo,final String risultato,
       EditText et, final int defVal, final TextView r) {
2           builder = new AlertDialog.Builder(this);
3           builder.setTitle(titolo);
4           et = new EditText(this);
5           et.setInputType(InputType.TYPE_CLASS_NUMBER);
6           builder.setView(et);
7           final EditText finalEt = et;
8           builder.setPositiveButton("Salva", new
       DialogInterface.OnClickListener() {
9               @Override
10              public void onClick(DialogInterface dialog, int
       which) {
11                  int p = Integer.parseInt(finalEt.getText().
       toString());
12                  sharedPreferences.edit().putInt(campo, p).
       apply();
13                  r.setText(messaggio + sharedPreferences.
       getInt(campo, defVal));
14                  Toast.makeText(getApplicationContext(),
       risultato, Toast.LENGTH_SHORT).show();
15              }
16          });
```

```
17          builder.setNegativeButton("Annulla", new
    DialogInterface.OnClickListener() {
18              @Override
19              public void onClick(DialogInterface dialog, int
    which) {
20                  dialog.dismiss();
21              }
22          });
23          return builder.create();
24      }
```

Listing 3.13: Codice per la gestione dell'AlertDialog

Il metodo sopra mostrato permette la creazione dinamica di un Alert in base ai parametri ricevuti in input che sono:

- Titolo dell'AlertDialog,
- Messaggio contenuto nell'AlertDialog,
- Campo da modificare,
- Risultato che rappresenta il messaggio da stampare dopo aver interagito con l'Alert
- Edit Text usata per modificare il campo in oggetto,
- Valore predefinito del campo in oggetto,
- Text View da modificare dopo aver concluso l'utilizzo dell'Alert Dialog.

3.7 Activity Visualizza Servizi

Quest'activity contiene una Recycler View di Card View che mostra all'utente i servizi disponibili sul Service Registry di Arrowhead con cui è possibile interagire. Per ottenerli è necessaria l'interazione con degli URL attraverso chiamate REST per questo motivo viene utilizzato il framework Volley. All'avvio dell'activity viene effettuata una richiesta HTTP GET che permette di ottenere un oggetto JSON contente un array JSON formato da servizi, per ogni servizio è necessario verificare se quest'ultimo è interagibile dall'app, per fare questo si controlla il campo Metadata del servizio verificando se al suo interno ha un oggetto JSON che rappresenta la Web Thing con cui l'utente dovrà interagire; se all'interno dei metadati è presente l'oggetto JSON il servizio viene aggiunto alla lista di servizi che popolerà la Recycler View. Viene gestita inoltre la possibilità di cliccare su ogni card per avviare un'ulteriore

activity che mostra con maggiori dettagli il servizio selezionato in quanto la card mostra solamente indirizzo IP e nome del servizio. Di seguito viene riportato uno snippet di codice che mostra la richiesta GET che permette alla Recycler View di popolarsi in quanto risulta essere l'operazione più importante fatta da questa activity. Dallo snippet sarà visibile la metodologia usata per la gestione dei servizi attraverso cui viene scelto quale servizio inserire nella Recycler View. È presente un ciclo che esamina tutti i servizi ottenuti dall'Arrowhead Service Registry e, dopo aver ottenuto le prime informazioni da conservare come ID del servizio, nome, indirizzo e porta, viene verificato se è presente un oggetto JSON all'interno del campo metadata, se è presente il servizio viene conservato altrimenti no. Infine è presente un controllo che verifica se sono stati conservati servizi o meno e nel caso in cui non è stato conservato nessun servizio viene mostrato un messaggio specifico all'utente.

```
String url="http://137.204.57.93:8443/serviceregistry/mgmt/";
        JsonObjectRequest o= new JsonObjectRequest(Request.
    Method.GET, url, null, new Response.Listener<JSONObject>()
     {
            @Override
            public void onResponse(JSONObject response) {
                try {
                    JSONArray jsonArray=response.getJSONArray
    ("data");
                    for(int i=0;i<jsonArray.length();i++){
                        JSONObject dato=jsonArray.
    getJSONObject(i);
                        int id=dato.getInt("id");
                        JSONObject serviceDefinition=dato.
    getJSONObject("serviceDefinition");
                        int idSD=serviceDefinition.getInt("id
    ");
                        String sd=serviceDefinition.getString
    ("serviceDefinition");
                        JSONObject provider=dato.
    getJSONObject("provider");
                        String address=provider.getString("
    address");
                        int port=provider.getInt("port");
                        try{
                            JSONObject metadata=dato.
    getJSONObject("metadata");
                            prop=metadata.getString("
    additionalProp1");
                            if(!prop.equals("")){
                                try{
                                    JSONObject td=new
    JSONObject(prop);
```

```
22                                      tutti.add(new Servizio(
    address,port,sd,prop));
23                                  }catch (JSONException e) {}
24                              }
25                              Log.v("a","ID: "+id+"servizio: "+
    address+":"+port+", "+sd+", "+prop);
26                          }catch (JSONException e) {
27                              Log.e("Errore","errore json: ",e)
    ;
28                              prop="";
29                              Log.v("a","ID: "+id+"servizio: "+
    address+":"+port+", "+sd+", "+prop);
30                          }
31                      }
32                      if(tutti.size()==0){
33                          tvs.setText("Non e' presente nessun
    servizio");
34                      }
35                      rvAdapter=new AdapterServizi(
    VisualizzaServizi.this,tutti);
36                      rv.setAdapter(rvAdapter);
37                      rvAdapter.setOnItemClickListener(
    VisualizzaServizi.this);
38                  } catch (JSONException e) {
39                      e.printStackTrace();
40                  }
41              }
42          }, new Response.ErrorListener() {
43              @Override
44              public void onErrorResponse(VolleyError error) {
45                  error.printStackTrace();
46              }
47          });
48          queue.add(o);
```

Listing 3.14: Codice creazione richiesta Volley GET

3.8 Activity Servizio Espanso

È possibile accedere a questa Activity premendo su una delle card mostrate dalla Recycler View, al suo interno troveremo due spinner personalizzati che permetteranno di visualizzare una determinata proprietà o eseguire una determinata azione della Web Thing a cui si riferiscono. Gli spinner sono stati personalizzati creando una classe apposita chiamata mSpinner che sovrascrive il metodo setSection, metodo utilizzato quando si seleziona un elemento dello spinner, in modo tale che anche se l'utente seleziona l'elemento che

risultava essere già selezionato verrà comunque eseguito il comportamento contenuto nel metodo onItemSelected dello Spinner.

```
@Override public void
    setSelection(int position, boolean animate)
    {
        boolean sameSelected = position ==
   getSelectedItemPosition();
        super.setSelection(position, animate);
        if (sameSelected) {
            getOnItemSelectedListener().onItemSelected(this,
   getSelectedView(), position, getSelectedItemId());
        }
    }
```

Listing 3.15: Codice mSpinner

L'activity Servizio Espanso ottiene dagli Extra dell'Intent che porta al suo avvio le informazioni relative al nome, indirizzo, porta dell'ATM e Thing Description della Web Thing. Grazie alla TD riesce ad ottenere le proprietà e le azioni eseguibili e, dopo che l'utente seleziona l'azione o la proprietà di suo interesse esegue un metodo chiamato Interagisci, questo metodo effettua una richiesta HTTP GET usando il framework Volley all'indirizzo ed alla porta precedentemente citati indicando nel campo query dell'URL il nome della property o della action. Visto che la nostra app non sa esattamente cosa otterrà come risultato della richiesta è stato sviluppato un metodo chiamato containssObject contenente un algoritmo ricorsivo che estrapola i dati nel caso in cui siano contenuti in oggetti nidificati dentro altri oggetti. A causa del fatto che la mia app comunicherà anche con sensori reali e di cui sono conosciute le caratteristiche è stato inserito un controllo che permette la gestione dei dati ottenuti dai suddetti sensori, questo è stato necessario a causa del fatto che essendo sensori triassiali forniscono 3 valori (x,y,z) per cui è richiesta una gestione diversa rispetto ad un sensore che fornisce un solo valore. Per questo motivo è stata realizzata la classe "Valore Composto" che contiene 3 interi chiamati x,y,z e l'orario di rilevazione. Infine, dopo aver ottenuti i dati, questi vengono visualizzati a video e registrati all'interno del database. Di seguito viene mostrato il metodo ricorsivo containssObject.

```
public ArrayList<Valore> containssObject(JSONObject o){
        Iterator<String> iter = o.keys();
        while(iter.hasNext()) {
            String key = iter.next();
            try {
                containssObject(o.getJSONObject(key));
            } catch (JSONException e) {
                while (iter.hasNext()) {
                    Log.v("APP","verificando "+key);
```

```
10                      try {
11                          f.add(new Valore(key, o.getString(key
    )));
12                          Log.v("APP","aggiunto "+key);
13                           key=iter.next();
14                          if ( ! iter.hasNext()) {
15                              Log.v("APP","verificando "+key);
16                              f.add(new Valore(key, o.getString
    (key)));
17                              Log.v("APP","aggiunto "+key);
18                          }
19                      } catch (JSONException ex) {
20                          ex.printStackTrace();
21                      }
22                  }
23              }
24          }
25          return f;
26  }
```

Listing 3.16: Codice metodo containssObject

Questo metodo ricorsivo riceve in input un oggetto JSON ed itera per tutte le sue chiavi verificando se qualche chiave contiene come valore un oggetto JSON a sua volta, in questo caso abbiamo la chiamata ricorsiva al metodo passando l'oggetto JSON come parametro altrimenti viene aggiunto il valore contenuto nella relativa chiave ad un ArrayList di valori che rappresentano il risultato della rilevazione.
Dopo aver interagito col servizio che rappresenta la Web Thing viene mostrato un pulsante che permette di visualizzare lo storico rispetto al dato ottenuto, premendo sul pulsante viene quindi avviata l'activity Storico.

3.9 Activity Storico

Quest'activity ha il compito di mostrare la serie storica dei valori relativi ad una determinata proprietà o azione rilevati rappresentando inoltre attraverso un grafico a linee la variazione dei valori ottenuti al variare del tempo. Per mostrare lo storico dei valori rilevati viene usata una Nested Scroll View popolata attraverso una query al database che permette di ottenere unicamente i dati di nostro interesse; con quegli stessi dati viene popolato il grafico a linee realizzato con la libreria MPAndroidChart. Per rappresentare in un unico grafico i valori x,y,z ottenuti dai sensori triassiali viene costruito il vettore dei valori ed attraverso la sua formula vengono ottenuti i relativi valori. Di seguito lo snippet di codice che permette la creazione del grafico a linee.

```
for(int k=0;k<tutti.size();k++){
            Date d=tutti.get(k).getData();
            DateFormat dateFormat = new SimpleDateFormat("dd/
    MM/yyyy");
            String strDate = dateFormat.format(d);
            labelsName.add(strDate);
            pos.add(new Entry(k, (float) Math.sqrt(
                    (Float.parseFloat(tutti.get(k).getVc().
    getX())*Float.parseFloat(tutti.get(k).getVc().getX()))+
                    (Float.parseFloat(tutti.get(k).getVc().
    getY())*Float.parseFloat(tutti.get(k).getY()))+
                    (Float.parseFloat(tutti.get(k).getVc().
    getZ())*Float.parseFloat(tutti.get(k).getVc().getZ()))
            ));
        }
        LineDataSet lineDataSet= new LineDataSet(pos,"
    Variazione "+c);
        dataSets.add(lineDataSet);
        LineData data=new LineData(dataSets);
        lineChart.setData(data);
        Description description= new Description();
        description.setText("Data");
        lineDataSet.setLineWidth(3f);
        lineDataSet.setValueTextSize(10f);
        lineChart.setDragEnabled(true);
        lineChart.setScaleEnabled(false);
        lineChart.setDescription(description);
        lineChart.setDrawBorders(true);
        lineChart.setNoDataText("Grafico assente per mancanza
     di rilevazioni");
        XAxis xAxis=lineChart.getXAxis();
        xAxis.setValueFormatter(new IndexAxisValueFormatter(
    labelsName));
        xAxis.setPosition(XAxis.XAxisPosition.TOP);
        xAxis.setDrawGridLines(false);
        xAxis.setDrawAxisLine(false);
        xAxis.setLabelCount(labelsName.size());
        xAxis.setLabelRotationAngle(270);
        lineChart.animateY(2000);
        lineChart.invalidate();
```

Listing 3.17: Codice metodo containssObject

Dallo snippet vediamo come la prima operazione da fare sia quella di costruire il vettore delle 3 serie di valori attraverso la formula $\sqrt{x^2 + y^2 + z^2}$.
Dopo aver ottenuto i valori viene costruito il grafico mettendo nell'asse delle X il tempo e nell'asse delle Y i valori calcolati.

3.10 Intent Service Rilevazione

L'Intent Service Rilevazione viene avviato in background quando viene avviata l'app ed ha il compito di effettuare una rilevazione ogni N secondi definiti dall'utente e verificare se i valori rilevati siano superiori alle soglie prefissate dall'utente. Quando questo Service viene lanciato viene creata una notifica permanente che avverte l'utente che l'app è in esecuzione in background e viene eseguito il suo comportamento che è racchiuso in un loop infinito che si ferma per il tempo definito dall'utente prima di eseguire un'altra iterazione. Inizialmente, dalle Shared Preferences, viene ottenuto l'intervallo tra una rilevazione e l'altra, successivamente tramite Volley viene effettuata una richiesta HTTP GET sincrona utilizzando un Request Future. Questa richiesta permette di ottenere tutti i servizi presenti sul Service Registry di Arrowhead e, come avviene per la Recycler View, viene applicato il medesimo algoritmo per capire se l'app può interagire o meno con i servizi che trova, l'unica aggiunta che è stata fatta consiste nell'avere due array list: uno che contiene gli indirizzi e l'altro che contiene le porte, entrambi sono dati necessari per poter successivamente interagire col servizio. Dopodichè ottiene dalle Shared Preferences il valore degli Switch Button relativi al monitoraggio e, per ogni Switch che vale true viene eseguito il metodo rilevazione, infine viene applicato uno sleep al system con una durata pari al valore definito dall'utente.
Il metodo rilevazione riceve in input tre parametri: il nome della proprietà che sta rilevando, l'indirizzo e la porta a cui deve fare la richiesta. Una volta richiamato questo metodo effettua una richiesta HTTP GET attraverso la libreria Volley inserendo come URL l'indirizzo, la porta e la proprietà ottenute in input, successivamente effettua le stesse operazioni fatte dal metodo "Interagisci" presente nell'activity chiamata Servizio Espanso, l'unica operazione aggiunta al metodo rilevazione è il controllo della soglia necessario per il monitoraggio per cui dopo aver ottenuto i valori x,y,z dal sensore triassiale viene calcolato il vettore e viene confrontato il suo valore con quello della soglia che è stato precedentemente ottenuto dalle Shared Preferences, se il valore del vettore è maggiore alla soglia viene creato un intent che invia una richiesta broadcast ad un Broadcast Receiver che ha il compito di creare la notifica.

```
1 public class CreaCanale extends Application {
2     public static final String CHANNEL_ID = "canal1ID";
3     @Override
4     public void onCreate() {
5         super.onCreate();
6         createNotificationChannel();
7     }
```

```
8       private void createNotificationChannel() {
9           if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
10              NotificationChannel serviceChannel = new
    NotificationChannel(
11                      CHANNEL_ID,
12                      CHANNEL_ID,
13                      NotificationManager.IMPORTANCE_DEFAULT
14              );
15              NotificationManager manager = getSystemService(
    NotificationManager.class);
16              manager.createNotificationChannel(serviceChannel)
    ;
17          }
18      }
19  }
```

Listing 3.18: Codice creazione notifica permanente

3.11 Creazione delle notifiche

Per la creazione delle notifiche sono stati realizzati due componenti: un Broadcast Receiver chiamato Rilevazione Receiver ed un Notification Helper. Il Notification Helper ha il compito di creare il canale di notifica e la notifica in sè.

```
1   public class NotificationHelper extends ContextWrapper {
2       public static final String canale1ID = "Rilevazione";
3       private NotificationManagerCompat NM;
4
5       public NotificationHelper(Context base) {
6           super(base);
7           createChannel();
8       }
9
10      public void createChannel() {
11          NotificationChannel canale1 = new NotificationChannel
    (canale1ID, canale1ID,
12                  NotificationManager.IMPORTANCE_DEFAULT);
13          canale1.enableLights(true);
14          canale1.enableVibration(true);
15          canale1.setLightColor(R.color.colorPrimary);
16          canale1.setLockscreenVisibility(Notification.
    VISIBILITY_PRIVATE);
17          getManager().createNotificationChannel(canale1);
18      }
19
20      public NotificationManagerCompat getManager() {
```

```
21          if (NM == null) {
22              NM = NotificationManagerCompat.from(
    getApplicationContext());
23          }
24          return NM;
25      }
26
27      public NotificationCompat.Builder
    getCanale1NotificationMonitoriaggio(String titolo, String
    messaggio) {
28          return new NotificationCompat.Builder(this,canale1ID)
29                  .setSmallIcon(R.drawable.ic_importante)
30                  .setContentTitle(titolo)
31                  .setContentText(messaggio)
32                  .setStyle(new NotificationCompat.BigTextStyle
    ()
33                          .bigText(messaggio));
34      }
35  }
```

Listing 3.19: Codice creazione notifica

Come è possibile vedere dallo snippet sovrastante, inizialmente viene creato il canale per la notifica impostando l'importanza, il colore del LED di notifica, la vibrazione e mostrando la notifica anche nel caso in cui lo smartphone abbia lo schermo bloccato, dopodichè viene creata la notifica. Attraverso il Notification Compat Builder viene creata una notifica in cui viene impostata l'icona, il titolo e il contenuto; questi ultimi 2 campi vengono forniti come parametri in input quando viene richiamato il Notification Compat Builder. La chiamata verso il Notification Compat Builder è a cura del Broadcast Receiver chiamato Rilevazione Receiver, questo viene richiamato in fase di rilevazione se viene soddisfatta la condizione sopra menzionata ed il suo compito è quello di ottenere il tipo di notifica che deve creare estrapolando questa informazione dagli Extra dell'Intent che l'ha richiamato e successivamente, in base al tipo, creerà una notifica impostando il titolo, il messaggio e l'ID.

```
1   public class RilevazioneReceiver extends BroadcastReceiver {
2       @Override
3       public void onReceive(Context context, Intent intent) {
4           String tipo=intent.getStringExtra("Rilevazione");
5           if(tipo.equals("Acceleration")){
6               Log.v("notifica","ACCELERATION");
7               int notificaID=5;
8               NotificationHelper NH = new NotificationHelper(
    context);
9               NotificationCompat.Builder nc = NH.
    getCanale1NotificationMonitoriaggio("Soglia "+tipo+"
    superata!",
```

```
10                      "Attenzione! Hai superato la soglia
    impostata!");
11              NotificationManagerCompat
    notificationManagerCompat=NotificationManagerCompat.from(
    context);
12              notificationManagerCompat.notify(notificaID, nc.
    build());
13          }
14      }
15  }
```

Listing 3.20: Codice invocazione Notification Helper

Lo snippet sovrastante mostra un esempio di creazione della notifica: dopo aver ottenuto il tipo dagli Extra dell'Intent se il tipo è "Acceleration" viene creata una notifica con ID pari a 5 e con il messaggio visibile nello snippet.

Capitolo 4

Risultati

In questo capitolo verranno mostrati i risultati prodotti dal software sviluppato sia per quanto riguarda il WAE sia per quanto riguarda la WoT App. Seguiranno immagini rappresentanti ciò che viene mostrato in output all'utente di diverso tipo: per quanto riguarda il WAE vedremo cosa comunica sul Prompt dei comandi dopo che viene lanciato e quando viene spento, per quanto riguarda la WoT App vedremo le varie schermate dell'applicazione e tutto ciò che è possibile fare con esse.

4.1 WAE

Quando viene lanciato, come già detto nel capitolo relativo all'implementazione, per prima cosa consuma la Thing Directory in modo tale da ottenere gli endpoint delle Web Thing di cui è necessario creare un ATM e registrarlo come servizio, successivamente, dopo aver creato l'ATM, lo registra come servizio e una volta registrati tutti gli ATM viene stampato a video un messaggio in cui viene confermata la corretta registrazione del servizio e ne viene indicato l'ID necessario per l'utilizzo.
Quando viene terminato il processo del WAE è necessario deregistrare i servizi precedentemente registrati questo perchè altrimenti sull'Arrowhead Service Registry troveremo dei servizi che in realtà non è possibile utilizzare. Una volta terminato il processo il WAE invocherà quindi il metodo Deregistra per tutti i servizi precedentemente registrati e stamperà a video un messaggio di corretta rimozione del servizio.

4.2 WoT App

Quando viene avviata l'app per la prima volta l'utente verrà rinviato nella pagina relativa alle impostazioni in modo tale da poter inserire le sue preferenze riguardo i parametri da monitorare e l'intervallo di rilevazione.

Figura 4.1: Schermata relativa alle impostazioni che è possibile personalizzare all'interno dell'app

Per tutti gli altri avvii dell'app l'utente verrà rinviato alla Main Activity che contiene un bottone attraverso cui è possibile accedere alla lista dei servizi con cui è possibile interagire.

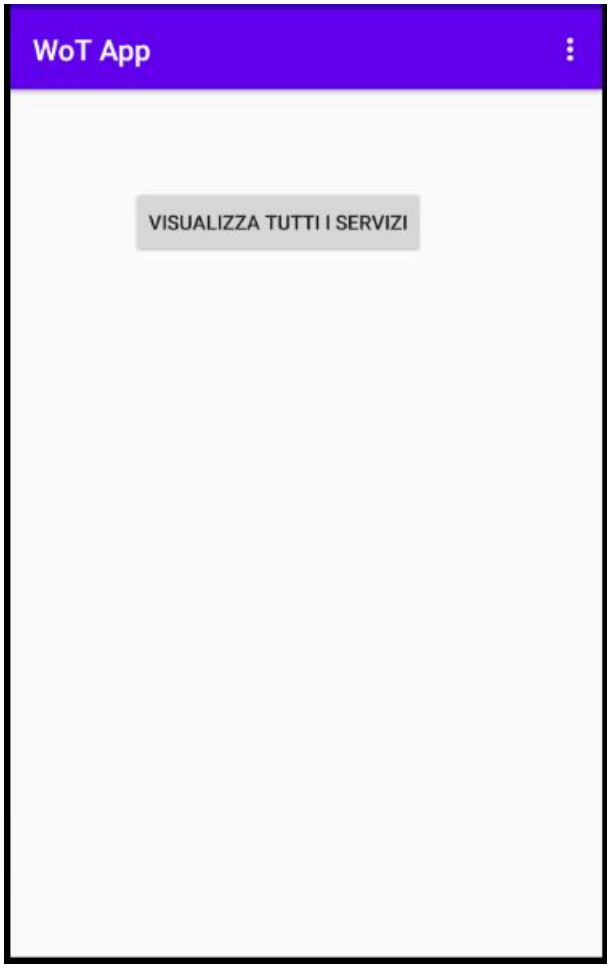

Figura 4.2: Schermata Main Activity

Come è possibile vedere dalla toolbar dell'applicazione è sempre visibile il menù che porta alle impostazioni rappresentato dai tre pallini verticali, inoltre quando l'app viene avviata viene mostrata una notifica permanente che avverte l'utente che l'applicazione è in uso anche in background.

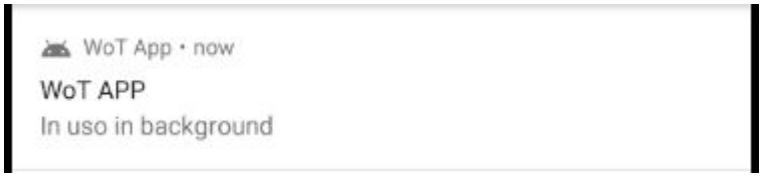

Figura 4.3: Notifica che avverte l'utente che l'app opera in background

Dopo aver premuto il bottone presente nella Main Activity l'utente si troverà davanti alla lista dei servizi, potrà decidere di quale servizio usufruire premendoci sopra.

Figura 4.4: Schermata Lista Servizi

In questo esempio vengono mostrati tutti i servizi offerti dalle Web Thing rappresentanti dei sensori reali il cui funzionamento non è legato all'applicazione, per questo motivo tra tutta la lista dei servizi sopra mostrati l'utente può interagire con tre di essi: testserial-1-measure, testserial-2-measure, testserial-3-measure. Queste tre Web Things contengono al loro interno un accelerometro ed un giroscopio, entrambi sensori triassiali, che rilevano le condizioni dell'edificio in cui sono installate.
Dopo aver premuto su uno dei tre servizi sopra menzionati l'utente si troverà davanti ad una schermata che permette l'interazione con lo stesso.

Figura 4.5: Schermata in cui l'utente può interagire col servizio

Il servizio sopra mostrato fornisce la possibilità di visualizzare le proprietà della Web Thing ma non di eseguire azioni su di essa, questo perchè questa Web Thing non possiede alcuna azione. Dopo aver selezionato una delle proprietà da visualizzare, l'app mostrerà il risultato in basso ed apparirà un tasto che permette di visualizzare lo storico relativo alla proprietà selezionata.

Figura 4.6: Schermata rappresentante i risultati della rilevazione

Dall'immagine è possibile notare che la lettura della proprietà "Acceleration" ha fornito come risultato l'orario di rilevamento di tale proprietà da parte del sensore ed i valori rilevati, inoltre è apparso il tasto che permette di visualizzare lo storico per quel determinato parametro.
Una volta premuto il bottone l'utente verrà rimandato in una schermata contente una Nested Scroll View contenente tutti i risultati ottenuti per quella determinata proprietà ed un grafico a linee rappresentante la variazione dei valori ottenuti nel tempo. A causa del fatto che i valori rilevati provengono da sensori triassiali avremo tre valori corrispondenti alle coordinate x,y,z. Per questo motivo sia per la rappresentazione del grafico che per il calcolo del valore usato nel monitoraggio viene realizzato il vettore, come è già stato spiegato nel capitolo relativo all'implementazione.

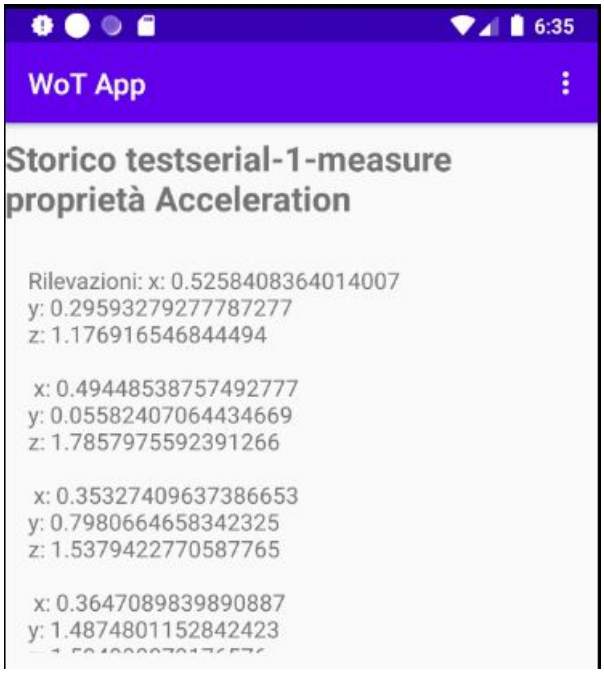

Figura 4.7: Schermata contenente lo storico delle rilevazioni

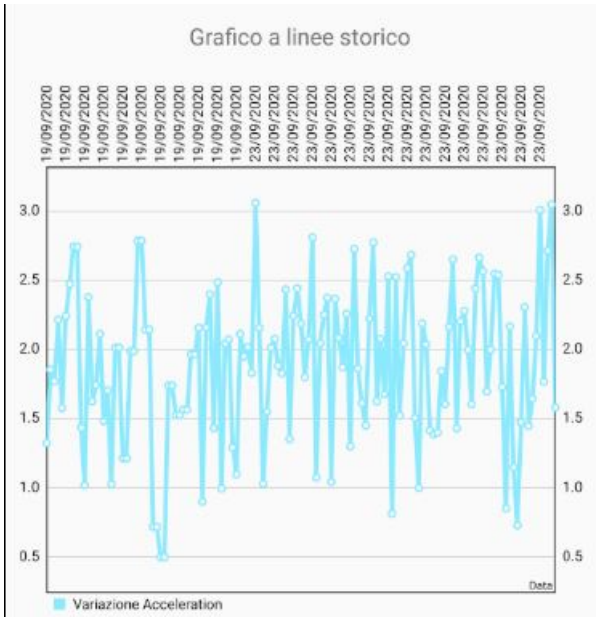

Figura 4.8: Esempio di un grafico relativo alla proprietà Acceleration di un sensore

Infine, quando viene rilevato un parametro il cui vettore è superiore alla soglia prefissata viene inviata una notifica che informa l'utente che la soglia relativa a quel parametro è stata superata.

Figura 4.9: Notifica che avverte l'utente riguardo il superamento della soglia

Capitolo 5

Conclusioni

In questa tesi ho realizzato un framework che permette l'interoperabilità tra dispositivi intelligenti attraverso il mondo del Web of Things ed il framework Arrowhead permettendone l'utilizzo attraverso un'applicazione mobile che ho usato per dimostrare il corretto funzionamento tra l'interoperabilità dei due sistemi, la possibilità di usufruirne attraverso un'applicazione mobile ed un caso d'uso reale relativo al monitoraggio strutturale. Questo elaborato dimostra inoltre come possa essere facilmente possibile la diffusione su larga scala di questa tecnologia in quanto tutto ciò di cui ha bisogno è uno smartphone Android. I risultati sono promettenti in quanto l'applicazione in futuro potrebbe essere aggiornata in modo tale da permettere facilmente l'interazione con altre Web Things sotto forma di servizi. Inoltre la funzionalità di monitoraggio applicata ai sensori inseriti nell'edificio utilizzato per la realizzazione del progetto potrebbe essere riutilizzata per altre costruzioni diverse dagli edifici quali ponti e strade in modo tale da poter agire preventivamente al crollo degli stessi.

5.1 Stato attuale e limiti del progetto

Ad oggi il framework è utilizzabile solo per alcuni tipi di Web Things che hanno una determinata struttura interna quindi è possibile che provando a gestire nuove Web Things sia necessario aggiornare il codice in modo tale gestirle nel modo corretto, inoltre potrebbe essere necessario implementare e migliorare le funzionalità relative alla sicurezza.

5.2 Sviluppi futuri

Ad oggi la mia applicazione non è pubblicata in nessuno store digitale, un obiettivo potrebbe essere quello di renderla disponibile su Play Store e, un domani, svilupparne la versione per smartphone con sistema operativo iOS. Il framework inoltre è molto flessibile quindi un domani potrebbe essere espanso gestendo qualunque altro tipo di Web Thing.

Ringraziamenti

Il momento dei ringraziamenti è sempre il più arduo, è quel momento in cui ci spogliamo del nostro ego e mostriamo a tutti che un uomo da solo non vale niente e che solo attraverso il lavoro di squadra è possibile ottenere risultati. Vorrei ringraziare tutti coloro che mi sono stati vicino in quest'avventura da fuori sede durata 3 anni: i miei amici, i miei genitori, i miei nonni, zii e cugini, per me fate tutti parte di una grande famiglia, mi siete stati vicini nei momenti belli e anche nei momenti più brutti e bui, sapevate quando avevo bisogno di un consiglio o di essere spronato senza bisogno di chiederlo. Un ringraziamento speciale va al mio compagno d'avventura Antonino Zocco, abbiamo iniziato questo percorso insieme 3 anni fa, abbiamo vissuto insieme e dormito insieme per questi 3 anni ed è stata un'esperienza che rifarei con tutto il cuore. Ci siamo divertiti, abbiamo faticato ma abbiamo sempre fatto tutto con il sorriso e nei momenti più tristi ci siamo sempre fatti forza a vicenda, mi hai insegnato tanto e spero di aver fatto altrettanto. Grazie di tutto fratello ti voglio bene. Voglio ringraziare un altro fratello conosciuto solo all'ultimo anno di Università: Lorenzo Benvenuti, ti avrò detto mille volte che avrei preferito conoscerti ed averti come coinquilino fin dal primo anno. Grazie per tutte le chiacchierate, i dibattiti e tutte le esperienze che abbiamo condiviso, compresa la quarantena, di cui sono sicuro che senza di te sarebbe stata molto più dura. Infine ringrazio il dott. Federico Montori ed il dott. Luca Sciullo per avermi reso partecipe a questo grande e ambizioso progetto, grazie per aver creduto in me, spero di aver soddisfatto le vostre aspettative. Un ulteriore ringraziamento va al dott. Federico Montori in quanto è sempre stato disponibile a darmi consigli riguardo la magistrale, grazie a lei ho scelto di rimanere a Bologna e sono sicuro che in futuro, se ne avrò bisogno, potrò contare su di lei.

Bibliografia

[1] Sabeen Javaid, Hammad Afzal, Fahim Arif, Naima Iltaf, Haider Abbas, and Waseem Iqbal. Catswots: Context aware trustworthy social web of things system. *Sensors*, 19(14):3076, Jul 2019.

[2] Adrian McEwen and Hakim Cassimally. *L'Internet delle cose*. Number 9. Apogeo Editore, 2014.

[3] F. Montori, T. S. Cinotti, and D. Brunelli. Smart energy services integrated within the arrowhead communication framework. In *2016 International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*, pages 558–563, 2016.

[4] Federico Montori, Luca Bedogni, and Luciano Bononi. A collaborative internet of things architecture for smart cities and environmental monitoring. *IEEE Internet of Things Journal*, 5(2):592–605, 2017.

[5] L. Sciullo, C. Aguzzi, M. Di Felice, and T. S. Cinotti. Wot store: Enabling things and applications discovery for the w3c web of things. In *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–8, 2019.

[6] Pal Varga, Fredrik Blomstedt, Luis Lino Ferreira, Jens Eliasson, Mats Johansson, Jerker Delsing, and Iker Martínez de Soria. Making system of systems interoperable – the core components of the arrowhead framework. *Journal of Network and Computer Applications*, 81:85 – 95, 2017.

[7] World Wide Web Consortium (W3C). Web of things architecture. `https://www.w3.org/TR/wot-architecture/`.