Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

CMDeBike:

un'architettura basata sul Web of Things per il monitoraggio della mobilità con ebike

Relatore:
Chiar.mo Prof.
Marco Di Felice

Presentata da:
Adamo Fapohunda

Correlatori:
Dott. Luca Sciullo
Ing. Paolo Borrelli

Sessione III
Anno Accademico 2019/2020

Dedicato a Nicoletta

Abstract

In questo elaborato di tesi si è partiti dall'analisi del mondo dell'*Internet of Things*, del *Web of Things* e dei sistemi di tracciamento remoto per elaborare una soluzione (CMDeBike) per la raccolta e la visualizzazione dei dati prodotti da *ebike*.

La progettazione e l'implementazione sono state svolte in due fasi: la prima fase ha visto lo sviluppo di un *edge device* dotato di descrizione semantica; la seconda fase è stata volta allo sviluppo di un insieme di servizi che forniscono la possibilità di cercare/trovare gli *edge device*, salvarne i dati secondo *policy* configurabili a *runtime* e visualizzare tramite interfaccia grafica i risultati dei salvataggi.

L'architettura ottenuta rispetta lo standard W3C WoT e risulta essere scalabile, configurabile ed estendibile. Inoltre, costituisce una base per l'introduzione nel mondo del W3C WoT di nuovi pattern per la gestione del salvataggio e dell'aggregazione dei dati.

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

Il settore dell'*Internet of Things* nasce all'inizio degli anni 2000 e da allora ha visto una crescita esponenziale. Di fatto, si è passati dai 15,41 miliardi di dispositivi connessi nel 2015 ai 30,73 del 2020 [56]. Il mercato globale dell'IoT è stato valutato a 826,25 milioni di dollari nel 2019 e si prevede che raggiungerà i 3.281,55 milioni di dollari entro il 2027 [38].

Una delle applicazioni dell'IoT in forte crescita è il settore delle biciclette elettriche o *ebike*. Paesi come la Cina vantano oltre 200 milioni di *ebike* utilizzate ogni giorno [80] [42] e paesi come la Svizzera ne promuovono l'utilizzo come mezzo di trasporto più ecologico [79].

Sebbene l'utilizzo di tecnologie IoT comporti ovvi vantaggi, pone di fronte a una serie di criticità. L'assenza di interoperabilità, dovuta alla frammentazione degli standard indotti dall'utilizzo di soluzioni proprietarie, è sicuramente la più impattante di queste in termini di costi di sviluppo.

Diversi enti si sono attivati per tentare di risolvere la problematica, tra tutti spicca il lavoro portato avanti dal *World Wide Web Consortium* per il *Web of Things*. Questo si propone di utilizzare standard e tecnologie Web, già ampiamente diffuse e consolidate, integrandole con una serie di componenti architetturali atte all'integrazione di differenti piattaforme e domini applicativi.

Lo scopo del lavoro di questa tesi è la progettazione e l'implementazione

di un sistema di raccolta e visualizzazione di dati prodotti da dispositivi *ebike* (GPS, antifurto, ecc...). Il progetto nasce da una collaborazione tra l'Università degli studi di Bologna e l'azienda Costruzioni Motori Diesel (CMD). Quest'ultima è specializzata nella prototipazione, nello sviluppo e nella produzione di parti meccaniche e assemblaggio di componenti nel settore automobilistico con particolare attenzione allo sviluppo di *software* ed elettronica per le *Engine Control Unit* (ECU).

Per garantire interoperabilità, scalabilità ed espandibilità del sistema, è stata adottata un'architettura secondo i principi dello standard W3C WoT.

Il sistema che è stato sviluppato si è dimostrato essere particolarmente efficace in fase di *testing* per quanto riguarda la scalabilità e si ritiene che possa essere un prodotto innovativo nel suo settore per la grande flessibilità che viene offerta nella gestione del salvataggio dei dati dei dispositivi.

In questo elaborato sono descritte le varie fasi di studio, progettazione e realizzazione dell'applicazione. Nella prima parte, che abbraccia i capitoli 2, 3 e 4, viene fatta una panoramica sullo stato dell'arte dell'IoT con particolare attenzione allo standard W3C WoT, per poi passare in rassegna una serie di soluzioni dedicate al mondo del tracciamento remoto di veicoli. La seconda parte, invece, è dedicata all'illustrazione dell'applicazione realizzata. Viene prima descritta la fase di progettazione (capitolo 5), in seguito la fase di implementazione (capitolo 6) per poi passare ai test di validazione del sistema (capitolo 7). Infine, vengono esposte le conclusioni e gli sviluppi futuri (capitolo 8).

Parte I

Stato dell'arte

Capitolo 2

Panoramica generale

2.1 Internet of Things

Internet è un sistema globale di reti di computer interconnesse che utilizzano la *suite* di protocolli *Internet* (TCP/IP) per servire miliardi di utenti in tutto il mondo. La forma principale di comunicazione dell'attuale *Internet* è tra esseri umani (*human-to-human*). L'*Internet of Things* (IoT) permette di estendere le capacità di *Internet* per abilitare la comunicazione anche tra macchine [17].

L'IoT è un paradigma relativamente recente nell'ambito delle scienze dell'informazione; formalmente è costituito dalle due parole Internet e Thing. Per *Internet* si intende un sistema globale di computer interconnessi che usano la *suite* di protocolli standard TCP/IP per servire miliardi di utenti in tutto il mondo [49]. Il punto di forza dell'IoT risiede nell'idea di proiettare una visione di infrastruttura globale che rende ogni oggetto e ogni persona connessa in ogni luogo e in ogni momento per avere la possibilità di comunicazione *human-to-human*, *human-to-machine* e *machine-to-machine*.

La definizione di IoT risulta attualmente ancora in evoluzione a partire dalla prima proposta da Kevin Ashton [17] nel 1999 che si riferiva all'IoT come un insieme di oggetti univocamente identificabili connessi attraverso la tecnologia radio-frequency identification (RFID). Si ritiene che possa essere

più affine al significato attuale di IoT la definizione proposta in [34];

Definizione 2.1.1. *L'Internet of Things è un sistema di oggetti fisici che possono essere trovati, monitorati e controllati attraverso dispositivi elettronici che comunicano a diversi livelli di interfacce di rete ed eventualmente che possono essere connessi alla rete Internet.*

L'obiettivo che l'IoT si prefigge è quello di unificare gli oggetti del mondo sotto un'infrastruttura comune offrendo la possibilità di controllare gli oggetti che sono presenti nell'ambiente e ricevendo informazioni da essi.

Definiamo *smart thing*, o semplicemente *Thing*, un oggetto che viene arricchito con una o più delle seguenti *feature* [34]:

- sensori (temperatura, luminosità, movimento, ecc...);
- attuatori (display, motori, casse, ecc...);
- possibilità di effettuare computazione ossia eseguire programmi o seguire certe logiche;
- interfaccia di comunicazione (*wired* o *wirless*).

Tali *Thing* hanno come scopo quello di connettersi, di raccogliere informazioni dal mondo che le circonda, di analizzarne la complessità e di reagire anche senza un intervento umano. Con l'evoluzione tecnologica, le capacità di tali dispositivi aumentano esponenzialmente sia in termini di *storage* che in termini di potere computazionale contro una dimensione che diventa sempre più ridotta.

Come mostrato in Figura 2.1, lo sviluppo dell'IoT è iniziato con il bisogno delle aziende logistiche di tracciare i prodotti, la soluzione è stata l'utilizzo della tecnologia *Radio-frequency identification* (RFID) che ha permesso loro di essere più efficienti e di ridurre l'errore umano. In seguito, proprio grazie alla nascita di nuovi sensori *wireless* e alla miniaturizzazione delle unità di calcolo, le capacità dei *device* sono state esponenzialmente estese [10].

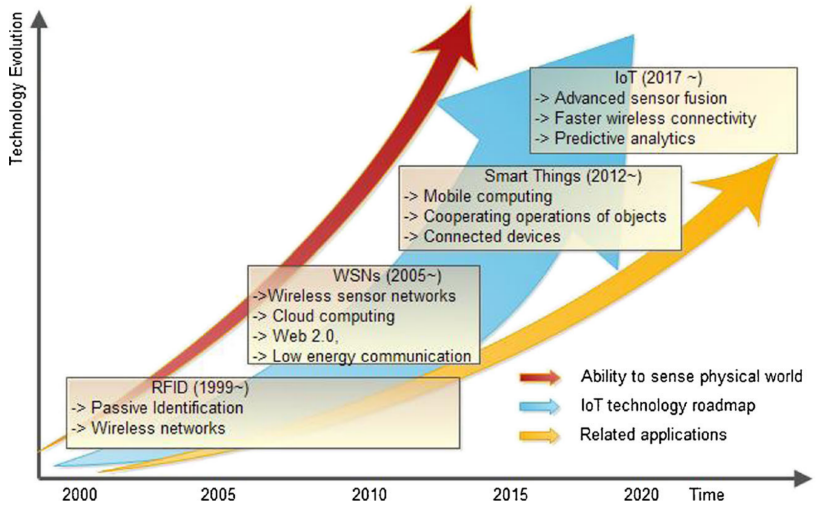


Figura 2.1: Evoluzione dell'IoT [10].

Concretamente, l'IoT comprende un vasto numero di tecnologie differenti come le reti di sensori *wireless* (WSNs), *barcodes*, sensori intelligenti, RFID, NFC, *cloud computing*, ecc...

L'appartenenza di un oggetto all'Iot non implica necessariamente che la *Thing* stessa sia direttamente connessa a *Internet*. Tuttavia, altre applicazioni potrebbero accedervi utilizzando reti come metodi Auto-ID (Automatic Identification come bar code, QR code, NFC e RFID tag), short-range radio (Bluetooth, ZigBee, ecc...) o la rete Wi-Fi [34].

Per un'implementazione efficace dell'IoT si possono definire alcuni prerequisiti [49]:

- richiesta dinamica di risorse;
- richieste real-time;
- *availability* delle applicazioni;
- protezione dei dati e privacy dell'utente;
- applicazioni efficienti in termini di consumi;
- esecuzioni delle applicazioni vicino all'utente finale;

- accesso a un sistema *cloud* aperto e interoperabile.

L'IoT presenta applicazioni in diversi ambiti della vita di tutti i giorni tra cui: il design di *smart cities* o *smart homes*, la previsione di disastri naturali, la gestione di impianti industriali, fino ad arrivare ad applicazioni in ambito medico che per il monitoraggio dello stato di salute dei pazienti.

Per godere dei grandi benefici delle numerose applicazioni dell'Iot è necessario affrontare diversi punti critici tra i quali [17]:

- Interoperabilità e necessità di uno standard: gestire un ecosistema globale di *Thing* ad oggi risulta pressoché impossibile perché molti produttori di dispositivi utilizzano tecnologie diverse. È opportuno trovare uno standard per l'IoT o, come proposto nel Web of Things (Sezione 2.2), definire un singolo *application layer protocol* universale per la comunicazione tra *device* e applicazioni [34].

- Gestione dei nomi e delle identità: dato che l'IoT idealmente connette miliardi di dispositivi, diventa necessaria la definizione di un meccanismo robusto ed efficiente per l'identificazione univoca dei dispositivi.

- Riservatezza delle informazioni e sicurezza di rete: si presenta la necessità di impedire l'accesso non autorizzato alle informazioni relative ai diversi dispositivi e di gestire la sicurezza e l'efficienza dei canali di trasmissione che siano cablati o *wireless*.

- Scalabilità: formalmente definita come "la capacità di un sistema o di una rete di gestire la crescita di un *environment* senza effetti sulle performance" [3] è uno dei maggiori punti critici vista la complessità dell'*environment* IoT.

- Gestione dei consumi nell'IoT: la grande quantità di dispositivi connessi alla rete e l'elevata velocità di trasferimento dei dati comportano elevati consumi. Per contrastare questo fenomeno è necessario rendere i dispositivi stessi sempre più efficienti dal punto di vista energetico.

- Big Data e Cloud Computing: la grande quantità di dati raccolti dai numerosi sensori in *environment* grandi e disomogenei fa dell'IoT un buon esempio di Big Data. La rete IoT è tipicamente dotata di una quantità di memoria limitata, pertanto l'integrazione con il *cloud computing* è un elemento fondamentale, su cui vi è grande attenzione e ricerca attiva, per i vantaggi che può portare in termini di scalabilità, *storage* delle risorse e possibilità di visualizzazione dei dati [3].

Nelle sezioni che seguono verrà approfondito il principale punto di criticità dell'IoT ossia la presenza di una moltitudine di standard 2.1.1. Infine, verrà presentata una delle soluzioni, ossia il Web of Things 2.2.

2.1.1 Varietà negli standard

Come evidenziato nella sezione precedente, è fondamentale il raggiungimento di uno standard nei principali aspetti dell'IoT per poter avere un crescita effettiva dell'utilizzo e della competitività dei dispositivi. I punti chiave dei tentativi di regolamentazione sono:

- design di *policy* e di architetture distribuite;
- garanzia di privacy e di protezione degli utenti;
- realizzazione di reti affidabili e sicure;
- implementazione di standard;
- esplorazione di nuove tecnologie.

In tutto il mondo ci sono stati tentativi e sforzi per stilare diversi standard per le tecnologie IoT a partire dall'*European Telecommunications Sandards Institute* (ETSI) e dall'*European Committee for Electro-technical Standardization* (CEN/CENELEC) che hanno realizzato più standard per tecnologie RFID, WSN, ecc.. , passando per gli Stati Uniti in cui l'*American National Standards Institute* (ANSI) che si è occupata della gestione di standard IoT,

fino ad arrivare in Cina dove molta ricerca viene svolta da *China Communications Standards Association* e dalla *China Electronics Standardization Institute* (CESI) riguardo alle tecnologie RFID e UHF (ultra high frequency). Oltre agli sforzi locali, anche diversi enti di standardizzazione internazionali come *International Telecommunication Union* (ITU), *Electronic Product Code global* (EPCglobal), *International Electro-technical Commission* (IEC), *International Organization for Standardization* (ISO) e *Institute of Electrical and Electronics Engineers* (IEEE) stanno definendo un insieme di standard riguardanti le tecnologie IoT[10].

2.1.2 Service-oriented architecture

Come evidenziato nelle sezioni precedenti, il design di un'architettura IoT deve essere orientato a ottenere un sistema estensibile, scalabile, adattivo, decentralizzato, interoperabile e in grado di gestire tecnologie eterogenee. Basandosi su questi requisiti, l'architettura più diffusa risulta essere la *service-oriented architecture* (SoA) che permette di decomporre sistemi complessi e monolitici in applicazioni che costituiscono un ecosistema di componenti semplici e ben definite [9][10].

Un'architettura IoT SoA generica consiste di quattro livelli: *Sensing layer*, *Network layer*, *Service layer*, *Interface layer* [10]. Questi diversi livelli permettono di scomporre un sistema complesso come quello di un'architettura IoT in un insieme di semplici sotto-strutture eterogenee che possono essere riusate e manutenute individualmente. In Figura 2.2 è mostrato un esempio di architettura SoA applicata in un contesto di IoT.

Sensing layer

Il *Sensing layer* è costituito da tag e sensori che permettono la raccolta di informazioni dall'ambiente e lo scambio di dati tra differenti dispositivi. Ogni oggetto nella rete possiede una *digital entity* che garantisce che tutti i dispositivi siano identificabili in modo univoco all'interno del *digital domain*.

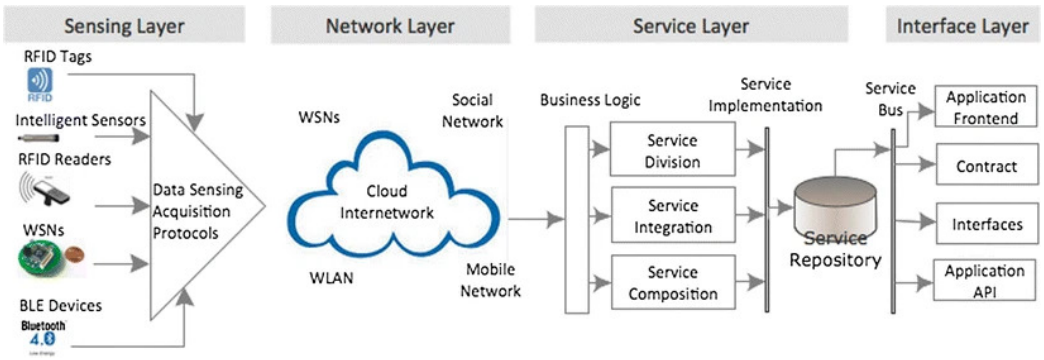


Figura 2.2: Architettura Service-oriented per l'IoT [10].

La tecnica di assegnare un identificatore univoco a un oggetto è chiamata *Universal Unique Identifier* (UUID).

I sensori che appartengono alla rete possono essere un numero molto elevato in un sistema complesso, da ciò deriva che ogni singolo sensore deve essere limitato nei costi, nelle dimensioni e nel consumo di risorse. Inoltre, un'altra caratteristica imprescindibile dei sensori è la capacità di essere individuabili all'interno del *digital environment* e di poter comunicare le rilevazioni che vengono fatte sull'ambiente fisico alle applicazioni in grado di gestirle.

Per quanto riguarda la comunicazione, i diversi dispositivi hardware possono comunicare in molti modi distinti tra cui i più comuni sono *WLAN*, *ZigBee* e *Bluetooth*.

Network layer

Il *Network layer* ha lo scopo di connettere le *Thing*. In questo modo *Thing* diverse possono condividere i dati con gli altri dispositivi della rete e, a questo stesso livello applicativo, risulta possibile aggregare i dati che verranno trasmessi alle unità di altro livello che contengono la logica per poter prendere decisioni. I principali requisiti di questo livello sono:

- tecnologie di network management;
- efficienza della rete;

- requisiti di QoS;
- tecnologie per la ricerca dei dispositivi;
- *processing* dei dati e dei segnali;
- sicurezza e privacy.

Service layer

Il *Service layer* ha il compito creare e gestire i servizi richiesti da utenti o applicazioni. I servizi nel *Service layer* sono eseguiti direttamente sulla rete per localizzare efficacemente nuovi servizi e recuperare dinamicamente i metadati su di essi.

Nel *Service layer* vengono svolte attività come lo *storage* delle informazioni, la gestione dei dati, i database di ontologie e i motori di ricerca. Tali operazioni vengono svolte dalle seguenti tipologie di servizi:

- **Service discovery**: permette di trovare gli oggetti che forniscano le informazioni necessarie.
- **Service composition**: permette l'interazione tra le *Thing* che sono connesse alla rete, inoltre offre la possibilità di schedulare o ricreare servizi per ottenerne di più efficaci e affidabili.
- **Trustworthiness managemen**: permette di capire come l'informazione ottenuta dai diversi servizi debba essere processata.
- **Service API**: permette l'interazione tra i diversi servizi.

2.2 Web of Things

Uno degli approcci per risolvere alcune problematicità del contesto dell'IoT è il *Web of Things* che riutilizza gli standard e le tecnologie Web esistenti per portare gli oggetti fisici del mondo all'interno web. Questo costituisce il

mezzo di comunicazione più diffuso su *Internet* in quanto *web service* che si appoggiano su di esso, si sono dimostrati essere indispensabili per la creazione di applicazioni interoperabili [83]. Ed è proprio lo sviluppo di *web server embedded* incredibilmente leggeri che permettono alle *smart things* che li implementano di essere astratte come *web service* e integrate senza problemi nel Web esistente. I *web server embedded* consentono la comunicazione tra *smart thing* utilizzando i linguaggi standard del Web; tali server differiscono da quelli tradizionalmente utilizzati perché, per essere inseriti all'interno delle *Thing*, devono essere leggeri e avere una bassa complessità fornendo allo stesso tempo più funzionalità possibili [83]. L'utilizzo del web, dunque, permette di mettere le *Thing* in comunicazione a un livello più alto, quello applicativo, al contrario dei diversi standard come *Zigbee*, *Bluetooth*, *IEEE 802.15.4* , *low-power WiFi* e *6LoWPAN* che sono in grado di fornire integrazione solo al *network layer*. [33]

Il WoT viene definito come un ecosistema di servizi che vengono orchestrati per renderli sempre più intelligenti e *human-centric* [83]. Il WoT presenta alcune differenze con una visione tradizionale di Web perché, includendo i servizi delle *smart thing*, viene aggiunta una componente di complessità che deriva dalla possibile intermittenza di connessione nella *Thing* e dal fatto che i *web server embedded* non sono potenti come quelli tradizionali e non sempre sono implementabili su dispositivi con limitato potere computazionale.

Il WoT, nonostante queste limitazioni, presenta diversi punti di forza che lo rendono una buona soluzione per la costruzione di sistemi IoT tra i quali:

- integrazione diretta in molti dispositivi data la possibilità di costruire *web server* dalle dimensioni di pochi KB;

- astrazione rispetto agli standard dei diversi produttori grazie all'utilizzo del Web (soluzione *one-for-all*);

- risulta semplice ampliare delle applicazioni già esistenti sul Web con l'utilizzo di dispositivi fisici che vengono astratti come *web service*.

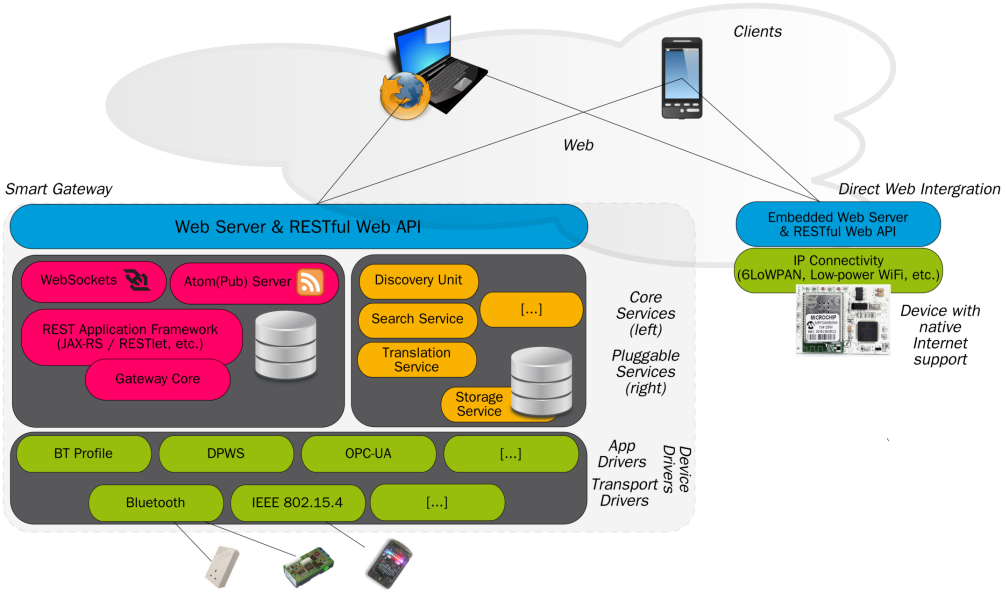


Figura 2.3: Integrazione diretta e indiretta delle *Thing* all'interno del Web rispettivamente attraverso *embedded web server* e *smart gateway* [33].

2.2.1 Integrazione delle Smart Thing nel Web

Una delle criticità presentate precedentemente è il fatto che dispositivi con risorse limitate non sempre possono ospitare un *web server*, motivo per cui esistono due modalità differenti che permettono l'integrazione degli oggetti fisici nel Web [83], tali modalità sono mostrate in Figura 2.3 e sono definite come segue:

- **Direct integration**: lo standard Wot richiede al livello applicativo la possibilità di connettività e interoperabilità e quindi il *web server* integrato nella *Thing* deve permettere la comunicazione attraverso gli standard del web. Per comunicare sul Web, tutto deve essere indirizzabile e quindi avere un indirizzo IP o essere IP-enabled quando si connette a Internet. Le *Thing* così definite sono direttamente inegrabili nel Web.

- **Indirect integration**: non tutti i dispositivi possiedono le risorse che permettono l'integrazione diretta di un *web server*. Per tali dispositivi

è possibile costruire uno *smart gateway* che si situa tra le *Thing* e il Web e che astrae i protocolli proprietari con cui comunicano le *Thing* fornendo protocolli Web o semplici *application programming interface* API che rendono le *Thing* accessibili dal web.

2.2.2 Web service paradigms

Integrare un *server* nelle *Thing* in maniera diretta o indiretta permette che queste siano accessibili sul Web attraverso semplici pagine sia statiche che dinamiche. Per il World Wide Web Consortium (W3C) un *web service* deve supportare l'interoperabilità anche nel caso *machine-to-machine* e vengono definiti due paradigmi principali: *Web service* arbitrari e *REST-compliant Web service*. Tali paradigmi possono essere adattati in un contesto di *smart things* attraverso le architetture WS-* e RESTful che verranno presentate nelle sezioni che seguono come descritte in [83].

Architettura WS-*

Con architettura WS-* si fa riferimento ai *web service* che utilizzano messaggi SOAP (Simple Object Access Protocol) con payload XML (Extensible Markup Language) e un protocollo di trasporto basato su HTTP. Tale architettura è particolarmente diffusa negli ambienti che implementano comunicazione *machine-to-machine*. Le tecnologie principali utilizzate sono:

- SOAP: protocollo basato su XML per lo scambio di informazioni tra applicazioni su HTTP;
- WSDL: permette di fornire agli utenti informazioni su come utilizzare il servizio attraverso una definizione tramite un linguaggio basato su XML;
- UDDI: *framework* di registrazione basato su XML che permette di descrivere e fare ricerca dei servizi Web nel mondo;

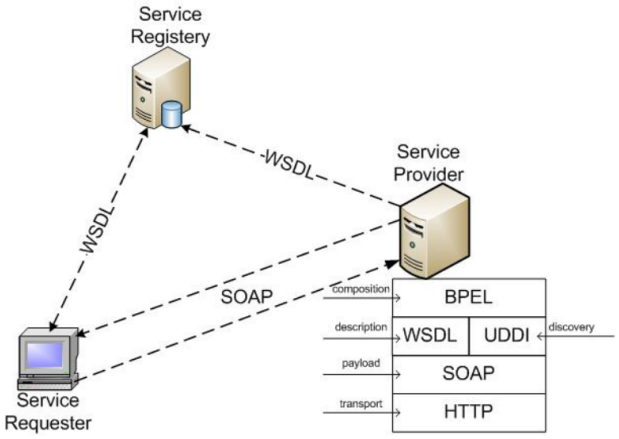


Figura 2.4: Workflow e Protocl Stack di architettura WS* [83].

- BPEL: permette la definizione di una notazione per la specifica dei comportamenti basandosi sulle interazioni tra servizi web.

In Figura 2.4 è possibile vedere un'implementazione del *workflow* di un'architettura WS-* e il relativo *Protocol Stack*. Per utilizzare il WS-* per le *smart thing* è stata proposta un'architettura *Service-Oriented Device Architecture* (SODA) che permette di integrare dispositivi fisici di diversi costruttori in un sistema distribuito in cui tutti i sensori e gli attuatori sono esposti come *web service*. In alternativa è possibile definire un *bus adapter* incaricato della comprensione e della comunicazione con standard proprietari, ma che viene allo stesso tempo esposto come un servizio uniforme *Service-Oriented Architecture* (SOA).

Architettura RESTful

L'architettura *REpresentational State Transfer* (REST) è l'architettura più utilizzata all'interno del Web e quella preferita in un contesto WoT per via della bassa complessità, delle interazioni *stateless* e del *loose-coupling*. Queste proprietà la rendono particolarmente efficace per l'integrazione in *web server* con risorse limitate oltre che facilitare la composizione di tali *service*. L'architettura REST si basa su quattro principi:

1. **Identificazione delle risorse**: nel Web tale identificazione avviene tramite *Uniform Resource Identifiers (URI)*.

2. **Interfaccia uniforme**: le risorse devono essere disponibili attraverso un'interfaccia uniforme con una semantica di interazione ben definita, tale interfaccia è fornita da *Hypertext Transfer Protocol (HTTP)* che viene utilizzato come protocollo applicativo.

3. **Messaggi auto-descrittivi**: le risorse sono disaccoppiate dalle loro rappresentazioni in modo tale che sia possibile utilizzare liberamente una varietà di formati di dati per descrivere le risorse stesse, a condizione che i formati di rappresentazione siano concordati e comprensibili dagli *endpoint*. I messaggi possono essere di diversi formati HTML, XML, JSON, ecc..., dove i metadati sulla risorsa possono essere utilizzati per controllare la memorizzazione nella cache, rilevare errori di trasmissione, negoziare il formato di rappresentazione ed eseguire l'autenticazione o il controllo dell'accesso tra gli *endpoint*.

4. **Operazioni stateless**: le operazioni devono essere tali per cui tutte le informazioni che servono per la richiesta siano parte della richiesta stessa.

2.2.3 Architettura REST del *Web of Things*

Nella sezione che segue viene presentata un'architettura per il *Web of Things* che si basa sulla proposta da Guinard in [34][33]. Guinard identifica quattro livelli per l'architettura WoT che vengono definiti sopra al *network layer*. Tali livelli aggiungono funzionalità al *network layer* e sono separati ed indipendenti. Per questo motivo ad ogni livello può seguire direttamente il livello applicativo senza la necessità di implementarli tutti e quattro. Una rappresentazione grafica è presente in Figura 2.5 dove i livelli sono definiti come segue:

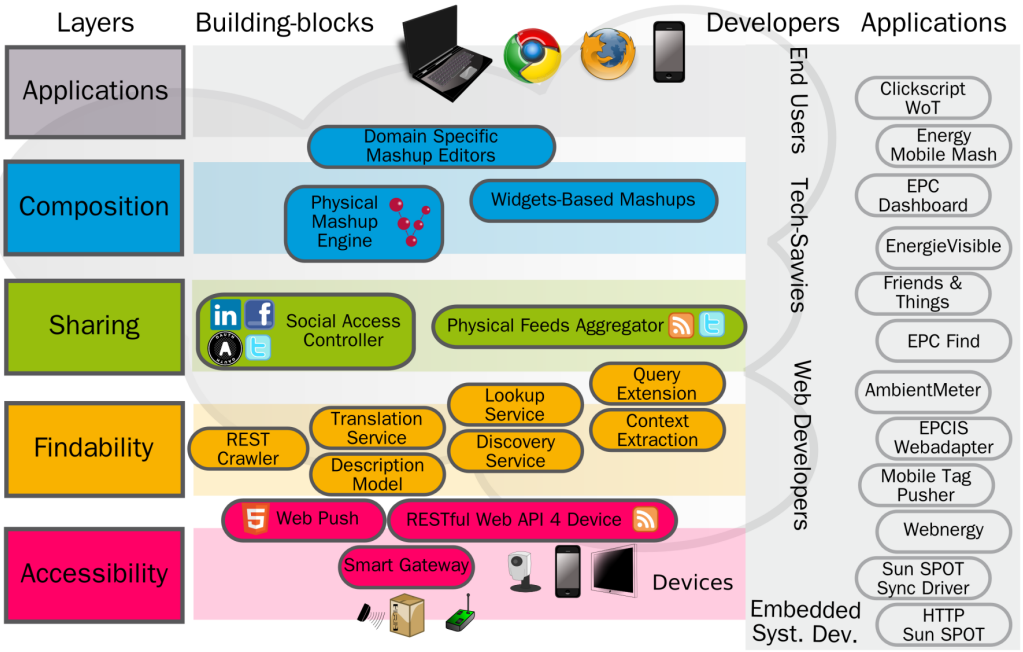


Figura 2.5: Architettura del *Web of Things* basato su REST presentato da Guinard in [34][33].

- **Accessibility Layer**: l'obiettivo di questo livello è quello di trasformare una *Thing* in una *web thing* con la quale gli altri dispositivi e le altre applicazioni possono comunicare. A questo livello vengono implementate le *WoT thing* e vengono arricchite con la possibilità di esporre RESTful APi utilizzando HTTP e JSON. Inoltre, viene descritto come i servizi del Web possono ottenere una connessione *real-time* o *event-driven* con la *Thing* attraverso *web-socket*.

- **Findability Layer**: l'obiettivo di questo livello è quello di permettere alle *web application* che desiderano utilizzare la *Thing* di capire di che tipologia di *Thing* si tratta, quali servizi può offrire, come trovarla, ecc ... In questo livello viene proposto un protocollo basato su HTTP che permette di fornire modelli per i dati, una sintassi per il payload, una semantica da seguire per descrivere le *Thing* e i *service*. Implementando questo livello risulta possibile per le *Thing* essere trovate in modo

automatico e utilizzate da altre applicazioni WoT.

- **Sharing Layer**: questo livello permette di implementare la sicurezza e l'efficienza di una condivisione dei dati sul Web implementando un meccanismo di condivisione che si basa sulle RESTful API.

- **Composition Layer**: i dati raccolti risultano eterogenei e quindi, l'obiettivo di questo livello, è quello di fornire integrazione e aggregazione attraverso un ecosistema di Web strumenti come, ad esempio, software per effettuare analisi sui dati o piattaforme di *mashup*.

2.2.4 Composizione dei servizi web

L'obiettivo finale del WoT è quello di ottenere *smart thing* che siano in grado di collaborare in maniera autonoma. Il problema fondamentale dell'IoT risiede nella diversità degli standard adottati e quindi, per costruire un'applicazione, lo sviluppatore ha la necessità di comprendere e adattarsi a tutti i diversi standard dei dispositivi oppure di selezionare solo dispositivi che adottano lo stesso standard. Entrambi gli scenari sono molto limitanti perché non permettono di scalare il sistema in un contesto diverso da quello per cui è stato progettato. Nel WoT tutte le *Thing* vengono astratte come come risorse Web indirizzabili, ricercabili e accessibili dal Web permettendo così agli sviluppatori di integrare le risorse delle *smart thing* all'interno di *web service* esistenti per creare una *mashup*, ossia un'applicazione o un servizio creato dalla composizione di diversi *web service*.

In [35] vengono presentate due tipologie di mashup:

- **Physical-Virtual Mashup**: sono applicazioni composte sia da *web service* offerti da *smart thing* che da *web service* tradizionali che forniscono una presentazione informativa dei dati raccolti. Di fatto, nonostante utilizzando le richieste HTTP sia possibile per l'utente fare *query* per recuperare informazioni sullo stato delle *Thing*, questo non è sufficiente per rispondere ai requisiti dell'utente, ma sono necessari ser-

vizi che forniscano un'interfaccia uniforme che permetta di comunicare con essi.

- **Phisical-Physical Mashup**: in questo caso si tratta di *mashup* i cui *web service* sono forniti solamente dalle *smart thing* in maniera diretta o indiretta.

2.2.5 Criticità nel WoT

Eterogeneità e scalabilità

Nonostante il WoT sia in grado di limitare i problemi di eterogeneità presenti nell'IoT fornendo una soluzione per la gestione degli standard differenti, ancora non è in grado di gestire in maniera uniforme le diverse necessità degli utenti. Infatti, a seconda della tipologia di applicazione che si vuole costruire, ci sono differenti specifiche in termini di *data quality*, *spatial resolution* o *sampling rate*. Si può pensare, per esempio, a quanto siano diversi i requisiti di un sistema di *Thing* che richiede un monitoraggio *real-time* da uno che prevede l'analisi delle misurazioni nel tempo.

Inoltre, per quanto concerne alla scalabilità, nonostante sia possibile gestire in maniera semplice un numero crescente di dispositivi nell'*environment* WoT, non risulta possibile per dispositivi dalle risorse limitate soddisfare le richieste di grandi numeri di applicazioni. Per questo motivo risulta necessario fare design ad hoc di sistemi che permettano la gestione delle richieste.

Sicurezza e privacy

La sicurezza sul Web è stata per molto tempo studiata e perfezionata, nonostante ciò, non sempre è possibile implementare tali tecnologie per la sicurezza all'interno di dispositivi *embedded*.

Ricerca e discovery

Gli approcci utilizzati nel Web per costruire *search engine* si basano sull'assunzione che i contenuti cambiano poco frequentemente e che quindi è possibile aggiornare un index a una bassa frequenza. Tale assunzione non è valida in un contesto WoT in cui i sensori producono dati che vengono aggiornati a distanza di pochi minuti o addirittura secondi e in cui i dati risultano significativi solo per un breve istante temporale. In generale esistono due approcci per costruire un sistema di ricerca per il WoT:

- Push approach: gli output dei sensori sono inseriti nel sistema di ricerca che li utilizza per risolvere le *query*. Questo metodo risulta però effettivamente implementabile solo in sistemi con pochi dispositivi.

- Pull approach: quando il sistema di ricerca riceve una *query* da parte dell'utente, vengono richiesti i dati rilevanti. Questo sistema risulta maggiormente scalabile rispetto al precedente, ma risulta meno accurato nei risultati restituiti all'utente e meno efficiente in termini di tempo.

2.3 W3C Web of Things

Il World Wide Web Consortium (W3C) *Web of Things* (WoT) ha come obiettivi quello di permettere l'usabilità delle piattaforme IoT e di permettere l'interoperabilità di queste con altri domini applicativi fornendo diversi componenti standard complementari (es Metadata e API) [78]. Il prodotto del lavoro di ricerca sono diversi documenti sia normativi che informativi.

I documenti normativi specificano le linee guida che devono essere seguite per soddisfare i requisiti dello standard. In particolare, sono stati redatti 4 documenti:

- **WoT Architecture**: documento che definisce l'architettura astratta per i singoli blocchi che compongono il W3C *Web of Things* e come

questi interagiscono tra loro. Si basa su una serie di requisiti derivati da casi d'uso reali per più domini applicativi.

- **WoT Thing Description**: documento che descrive un modello formale e una rappresentazione comune per un *Web of Things (WoT) Thing Description*. La *Thing Description* (TD) descrive i metadati e le interfacce delle *Thing*, dove per *Thing* si intende un'astrazione di una entità fisica o virtuale con la quale è possibile interagire e che a sua volta partecipa alla rete WoT. Viene dunque definita un'ontologia per la descrizione semantica delle *Thing* con focus su dati, modelli di interazione e metadati per la gestione della sicurezza e della comunicazione.

- **WoT Discovery**: mentre con il documento *WoT Thing Description* viene affrontato il problema dell'interoperabilità tramite la descrizione semantica, resta aperta la problematica di come ottenere la *Thing Description*. Il documento *WoT Discovery* espone il processo per il recupero della TD tramite l'utilizzo di un servizio dedicato: *Thing Directory Service*. A differenza dei precedenti documenti, al momento questo è ancora in stato di sviluppo.

- **WoT Profile**: il *WoT Architecture* e il *WoT Thing Description* definiscono un meccanismo di descrizione e un formato per descrivere una moltitudine di *device* differenti che lavorano su vari protocolli. Essendo in formato estremamente flessibile, potrebbe far ritornare il problema dell'interoperabilità. Il documento *WoT Profile* definisce meccanismi generici per la definizione di *WoT Thing Description* in modo non ambiguo e le regole per la definizione dei modelli dei dati. Anche in questo caso il documento risulta essere ancora in stato di sviluppo.

I documenti informativi sono di supporto all'utente per meglio comprendere le specifiche dei documenti normativi. In particolare, sono stati redatti 3 documenti:

- **WoT Scripting API**: Dcumento che descrive un'API (*application programming interface*) che rappresenta l'interfaccia WoT che consente agli script di trovare, utilizzare, ed esporre *Thing* definite da interazioni WoT specificate dallo script stesso.

- **WoT Binding Templates**: Documento che descrive design pattern ed estensioni al vocabolario della *WoT Thing Description* per consentirle di essere utilizzata con differenti protocolli o *payload* su differenti standard.

- **WoT Security and Privacy Guidelines**: Documento che definisce i requisiti di sicurezza generali per un sistema *Web of Things* (WoT) astraendo un "*threat model*". Il *WoT threat model* definisce le principali parti interessate alla sicurezza, le risorse rilevanti per la sicurezza, i possibili aggressori, le superfici di attacco e infine le minacce per un sistema WoT. Utilizzando questo modello generico come guida, dovrebbe essere possibile selezionare un insieme specifico di obiettivi di sicurezza per un'implementazione concreta del sistema WoT.

Capitolo 3

W3C WoT Architecture

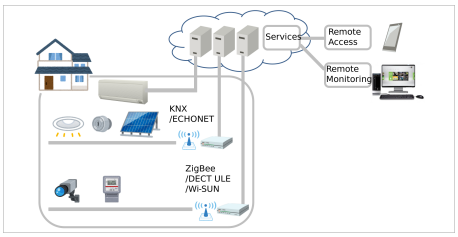
Come già accennato nel capitolo precedente, il *W3C Web of Things* (WoT) ha lo scopo di consentire l'interoperabilità tra piattaforme IoT e domini applicativi. Nel complesso, l'obiettivo del WoT è preservare e integrare gli standard e le soluzioni IoT esistenti. Per raggiungere l'obiettivo il W3C ha elaborato un'architettura astratta che a partire da requisiti definisce una struttura concettuale di base che può essere utilizzata in una varietà di scenari di *deployment* concreti.

In questo capitolo vengono illustrati alcuni casi d'uso (Sezione 3.1), dai quali vengono estratti i requisiti funzionali e tecnici di un'architettura WoT (Sezione 3.2). In seguito vengono presentate le specifiche (Sezione 3.3) di tale architettura e i blocchi che la costituiscono (Sezione 3.4). Infine, viene presentata un'implementazione dell'architettura WoT (Sezione 3.5).

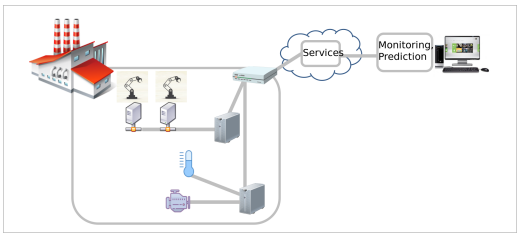
3.1 Casi d'uso

L'architettura astratta del WoT permette di definire un *framework* concettuale che può essere mappato su una varietà di scenari concreti. I casi d'uso presi in considerazione del W3C sono i seguenti [74]:

- **Settore consumer**: un esempio di questo caso d'uso sono le *Smart Home* (Figura 3.1a) in cui i *gateway* fanno da punto di interconnessione



(a) Esempio di architettura smart home



(b) Esempio di architettura smart factory

Figura 3.1: Esempi di casi d'uso nel W3C WoT [74].

tra gli *edge device* (sensori, telecamere, attuatori, ecc...) e i servizi che attraverso *Internet* li monitorano/gestiscono.

- **Settore industriale**: nel caso dell'ambito industriale sono possibili diverse applicazioni tra cui, per esempio, le *Smart Factory* (Figura 3.1b) che permettono di controllare il sistema produttivo attraverso servizi *cloud* di monitoraggio per la previsione di guasti e il rilevamento di anomalie.

- **Trasporti e logistica**: monitoraggio dei veicoli, dei costi del carburante, delle esigenze di manutenzione per ottimizzare l'utilizzo di una flotta di veicoli. Monitoraggio della qualità e delle condizioni dei beni trasportati, particolarmente utile per garantire l'integrità della catena del freddo.

- **Risorse energetiche**: monitoraggio di piattaforme petrolifere e del trasporto delle risorse energetiche per migliorare la sicurezza per i lavoratori e ridurre i danni ambientali.

- **Agricoltura**: il controllo dei nutrienti del terreno, automazione dell'irrigazione, ecc... per migliorare la qualità dei prodotti agricoli.

- **Salute**: la raccolta e l'analisi di dati clinici per raccogliere statistiche in aree di nuova esplorazione. Oppure strumenti di monitoraggio remoto di pazienti per ridurre al minimo il rischio di recidive.

- **Monitoraggio dell'ambiente**: il monitoraggio dei parametri di inquinamento dell'aria e dell'acqua, i livelli di radioattività, la quantità di polveri sottili, ecc...

- **Smart cities**: il monitoraggio dei ponti, delle infrastrutture, dei canali, ecc... per analizzare le condizioni dei materiali, il deterioramento al fine di effettuare lavori di riparazioni e mantenimento più mirati e di prevenire danni significativi.

Vengono anche introdotti una serie di pattern di *deployment* che illustrano come le *thing* interagiscono con *controller*, altri dispositivi, agenti e *server* [72]. Il pattern più semplice è il *Device Controller* che modella l'interazione nel caso di un *device* controllato da un utente (es. il telecomando del condizionatore). Per l'interazione tra due *thing* vi è pattern *Thing-to-Thing* (es. sensore di temperature e condizionatore). A seguire, aumentanto il grado di complessità vengono esposi i pattern *Edge Device*, *Digital Twin* e *Cross-domain Collaboration*. In particolare, un *Digital Twin* è una rappresentazione virtuale di uno o più dispositivi che risiede in un *server cloud* o in un *edge device*, mentre il *cross-domain collaboration* definisce come sistemi di domini differenti (come smart cities e smart factories) possano essere uniti all'interno di un unico sistema.

3.2 Requisiti

3.2.1 Requisiti funzionali

A partire dai casi d'uso vengono poi definite le proprietà richieste che una architettura WoT deve avere [73].

Principi generali

- **Interazione**: l'architettura WoT dovrebbe consentire l'interazione reciproca di diversi ecosistemi utilizzando il Web;

- **Comunicazione**: l'architettura dovrebbe essere basata su API RESTful e consentire di utilizzare più formati tra i *payload* comunemente usati nel Web;
- **Flessibilità**: dovrebbe essere in grado di coprire la grande varietà di configurazioni e dispositivi presenti nel WoT;
- **Compatibilità**: il WoT dovrebbe poter permettere l'interazione tra architetture IoT che utilizzano standard diversi;
- **Scalabilità**: il WoT dovrebbe essere in grado di scalare su numeri molto alti di dispositivi;
- **Interoperabilità**: il WoT dovrebbe permettere l'utilizzo di dispositivi e sistemi *cloud* differenti.

Funzionalità di una *Thing*

- lettura e aggiornamento delle informazioni di stato;
- *subscribe/unsubscribe* alle notifiche o a eventi che segnalino il cambiamento delle informazioni di stato della Thing;
- invocazioni di funzioni che causino l'attivazione degli attuatori oppure che scatenino processi di calcolo.

Requisiti di un'architettura WoT

- **Ricerca delle Thing**: l'architettura WoT dovrebbe permettere una ricerca semantica delle *Thing* sulla base degli attributi o delle funzionalità basandosi su un unico vocabolario.
- **Descrizione delle Thing**: l'architettura WoT dovrebbe permettere la descrizione degli attributi delle *Thing* come nome, descrizione, versione, formato, link a altre *Thing* relazionate e metadati informativi.

- **Network**: l'architettura WoT dovrebbe supportare l'utilizzo in contemporanea di più protocolli, sia quelli comunemente utilizzati su *Internet*, che quelli comunemente usati nelle reti locali.

- **Deployment**: l'architettura WoT dovrebbe supportare dispositivi con capacità differenti, dagli *edge device* alle *virtual thing* in *cloud*. Inoltre, dovrebbe supportare più livelli di gerarchie delle *thing* con entità intermedie come *gateway* e *proxy*.

- **Application**: l'architettura WoT dovrebbe descrivere applicazioni per un numero ampio di *device* differenti come *edge device*, *gateway*, *cloud device* e *UI/UX device* usando lo standard Web.

- **Legacy adoption**: l'architettura WoT dovrebbe permettere la gestione di protocolli IP e non-IP legacy e l'utilizzo in maniera trasparente dei protocolli IP esistenti.

3.2.2 Requisiti tecnici

Mentre i *Common Pattern* definiscono l'architettura astratta, in questa sezione verranno elencati i requisiti tecnici derivanti dall'architettura astratta.

Device

Per quanto riguarda i dispositivi, l'accesso ai *device* avviene tramite *Thing Description (TD)*, ossia una descrizione dell'interfaccia e delle funzionalità della *Thing*. La *TD* è costituita dalle seguenti informazioni:

- ***General Metadata***: contengono l'identificativo del *device* (URI), alcune informazioni sul *device* fisico e altre informazioni interpretabili da persone;

- ***Information Model***: definisce gli attributi del device, le sue impostazioni, le funzionalità che offre, i protocolli di comunicazione;

- ***Security Information***: include informazioni riguardo all'autenticazione, alle autorizzazioni e a come comunicare in maniera sicura.

Applicazioni

Le applicazioni devono essere in grado di generare e utilizzare reti e programmi basate sulle TD. Quest'ultime devono poter essere recuperate e processate dalle applicazioni stesse.

Digital Twins

I *Digital Twins* devono generare la *program interface* internamente basandosi sui metadati. Inoltre devono rappresentare i *virtual device* usando le stesse *program interface*. Un *twin* deve produrre ed esporre la descrizione del *virtual device*.

I *Digital Twins* devono avere nuovi identificatori rispetto ai *device* originali a loro gemelli e possono offrire diversi meccanismi di trasporto, sicurezza e impostazioni.

Discovery

Affinché le TD dei *device* siano accessibili ad altri *device* e applicazioni, occorre un modo per condividerle. Le *Directory* possono soddisfare questo requisito fornendo funzionalità per consentire ai dispositivi stessi o agli utenti di registrare le TD.

Security

Le informazioni per l'autenticazione, l'autorizzazione e la crittografia del *payload* devono essere descritte nella TD dei *device*. L'architettura dovrebbe supportare più meccanismi di sicurezza.

3.3 Architettura WoT

3.3.1 Introduzione

Per affrontare i casi d'uso precedentemente esposti e per soddisfare i requisiti da questi ricavati, il *Web of Things* (WoT) si basa sul concetto di *Web Thing* che può essere utilizzato dai cosiddetti *Consumer*. Per *Web Thing* o semplicemente *Thing* si intende un'astrazione di un'entità fisica o virtuale descritta da una *Thing Description* (TD). Per *Consumer* si intende un'entità in grado di processare una TD e di interagire con le *Thing*.

Una TD è la rappresentazione esterna di una *Thing*. Questa è specifica di una istanza, ossia descrive una singola *Thing* e non tipologie di *Thing*. Il formato di rappresentazione della TD è JSON-LD che può essere processato sia da classiche librerie JSON che da un processore JSON-LD dato che l'*information model* sottostante è basato su grafi. Oltre alla TD, potrebbero essere aggiunte altre rappresentazioni della *Thing* come interfacce grafiche HTML, immagini dell'entità fisica, ecc... Perché una *Thing* sia definita come tale, la TD deve essere conforme agli standard WoT e quindi deve avere un formato *machine-undersandable* in modo da permetterne la ricerca, l'interpretazione e la possibilità di fare azioni da parte dei *Consumer*. In Figura 3.2 è possibile visualizzare una rappresentazione dell'interazione tra *Thing* e *Consumer* attraverso la *Wot Thing Description*.

Una *Thing* può essere, oltre che un'astrazione di un'entità fisica, anche l'astrazione di un'entità virtuale ossia una composizione di più *Thing*, in questo caso viene detta *Intermediary*. Un modo per poterle implementare

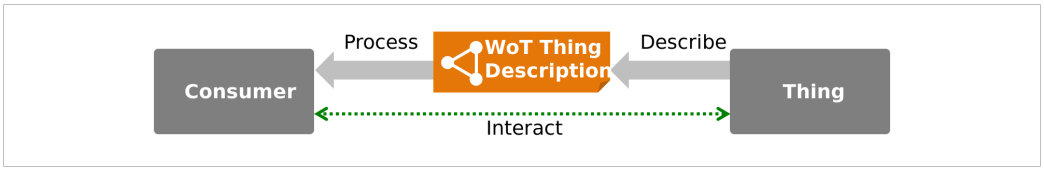


Figura 3.2: Interazione tra *Thing* e *Consumer* attraverso *WoT Thing Description* [75].

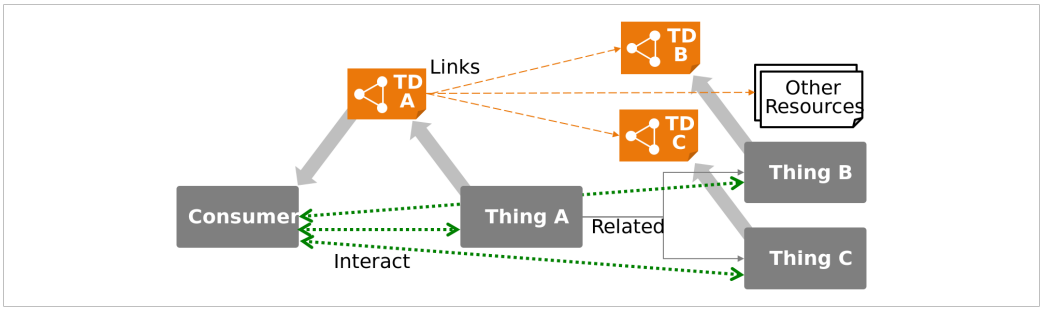


Figura 3.3: *Thing* collegate tramite *link* [75]

è quello di fornire una singola TD che contiene un sovrainsieme di tutte le *capability* delle Thing che la compongono. Quindi la TD principale è costruita collegando le *sotto-Thing* in una struttura gerarchica. Risulta inoltre possibile inserire nelle TD *link* che fanno riferimento non solo alle strutture annidate, ma anche ad altre *Thing* che presentano una certa relazione con quella descritta dalla TD oppure ad altre tipologie di risorse come manuali, cataloghi, file CSD, UI grafiche, ecc...

Grazie a questa tipologia di *Web linking*, il *Web of Things* diventa navigabile sia per gli umani che per le macchine. In Figura 3.3 sono mostrate una serie di *Thing* collegate tra loro tramite *linking*.

I concetti dell'architettura W3C WoT sono applicabili a diversi livelli delle applicazioni IoT, in particolare dal *device level* al *cloud level*. Questa integrazione unificata su più livelli di astrazione permette diverse modalità di comunicazione tra cui *Thing-to-Thing*, *Thing-to-Gateway*, *Thing-to-Cloud*, *Gateway-to-Cloud* e *Cloud-to-Cloud*. In Figura 3.4 è riportato un esempio di una possibile implementazione di un'architettura WoT che combina i diversi livelli.

3.3.2 Web Thing

Una *Web Thing* ha quattro caratteristiche che devono essere definite:

- **Behaviour**: definisce una raccolta dei comportamenti autonomi della *Thing* e degli *handler* per l'*Interaction Affordance*.

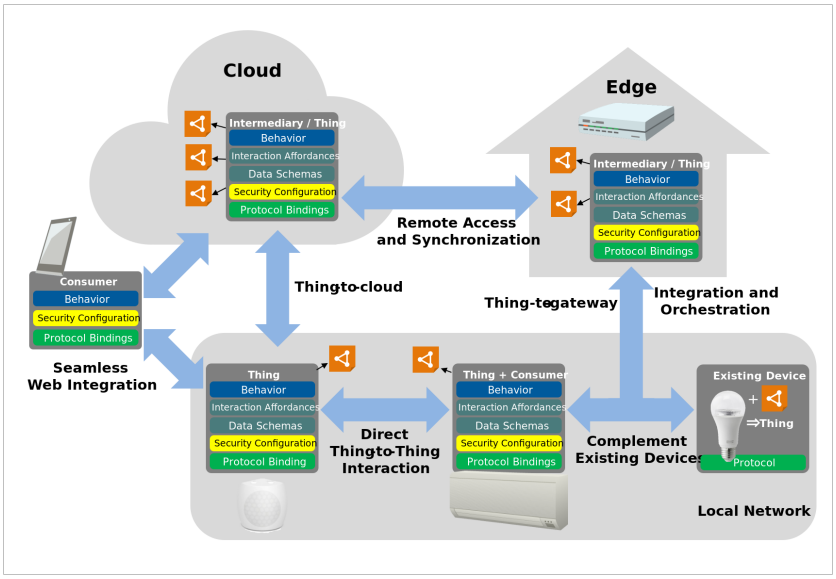


Figura 3.4: Esempio di architettura astratta del W3C WoT [75].

- **Interaction Affordance**: fornisce un modello che specifica come i *Consumer* possano interagire con le *Thing* attraverso operazioni astratte, ovvero senza informazioni riguardo al protocollo di rete o al formato dei dati. Un *Interaction Affordance* generico è dato dalla navigazione tramite *link*, ma il W3C specifica tre tipologie di *Interaction Affordance*:
  - **Property**: permette di esporre lo stato di una *Thing*, tale stato deve essere *readalble* e, opzionalmente, può essere anche *writeable*;
  - **Action**: permette di invocare sulle *Thing* delle funzioni che manipolano direttamente lo stato o che lo manipolano attraverso logiche interne alla *Thing*;
  - **Event**: permette di trasferire in maniera asincrona i dati dalla *Thing* al *Consumer*, ma, invece che trasmettere lo stato, vengono trasferiti eventi. Tali eventi possono essere causati da condizioni che non sono espresse nella *Property*.

  In questo componente va specificato anche il Data Schema, ossia la

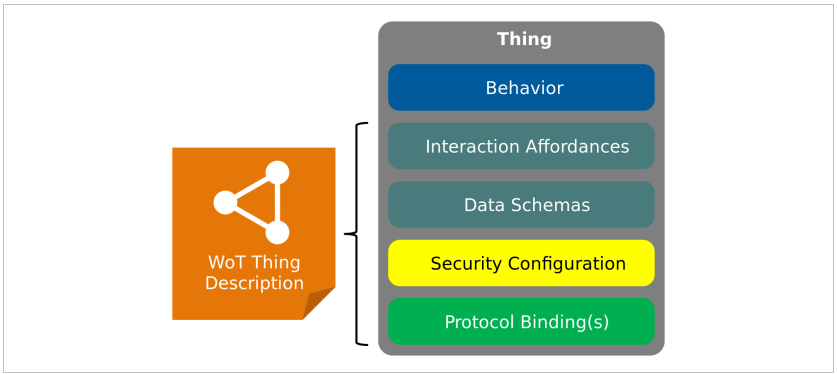


Figura 3.5: Architettura di una *Thing* [76]

descrizione dell'*Information Model* inteso come la struttura del *payload* e i dati che devono essere passati tra *Thing* e *Consumer* durante un'interazione.

- **Security Configuration**: rappresenta il meccanismo di controllo degli accessi all'*Interaction Affordance*.
- **Protocol Binding**: aggiunge informazioni che permettono di mappare ogni *Interaction Affordance* su un messaggio concreto in un protocollo specifico come HTTP, COAP o MQTT.

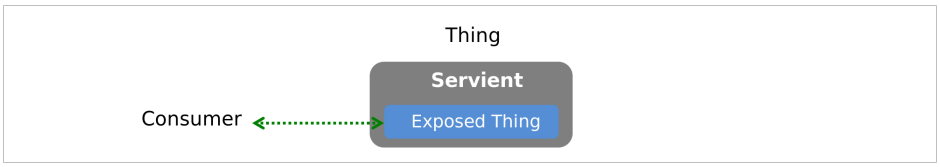
In Figura 3.5 è possibile vedere una rappresentazione grafica degli aspetti architetturali di una *Thing* che sono stati descritti.

3.3.3 Servient

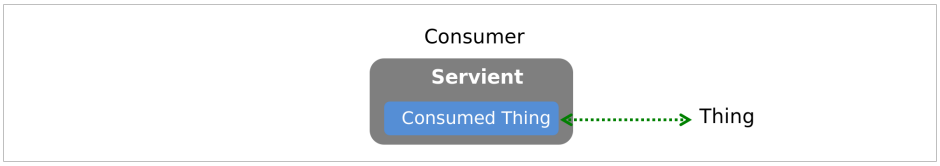
Quando i componenti dell'architettura astratta WoT vengono implementati come uno *stack software*, lo *stack* viene chiamato *Servient*. Di seguito sono elencate le implementazioni come *Servient* di una *Thing*, un *Consumer* e un *Intermediatary* presentate in [76].

- **Implementazione di una *Thing* come Servient**: il *Servient* contiene una rappresentazione della *Thing* chiamata *Exposed Thing* e rende la sua interfaccia WoT disponibile per i *Consumer*. Un *Exposed Thing*

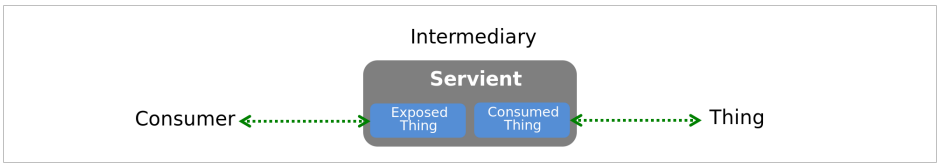
è infatti una rappresentazione software di una *Thing* che fornisce un'interfaccia delle *Interaction Affordance* messe a disposizione dalla *Thing* stessa.



- **Implementazione di un *Consumer* come *Servient***: il Servient deve creare una rappresentazione della *Thing* chiamata *Consumed Thing* che permette di comunicare con le *Exposed Thing*. Una *Consumed Thing* è una rappresentazione *software* di una *Thing* consumata da un *Consumer* e viene generata da questo dopo aver processato la TD.

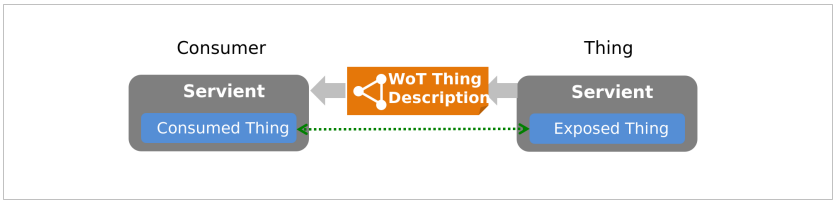


- **Implementazione di un *Intermediatary* come *Servient***: dato che l'*Intermediatary* svolge sia il ruolo di *Thing* che quello di *Consumer*, il *Servient* conterrà sia *Exposed Thing* che *Consumed Thing*

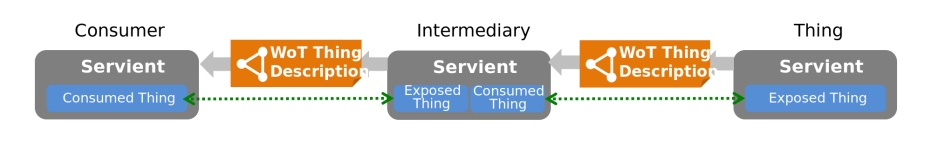


La comunicazione tra *Servient* può avvenire in due modalità distinte:

- **Comunicazione diretta**: quando la *Thing* e il *Consumer* hanno lo stesso protocollo di rete e sono accessibili tra loro.



- **Comunicazione indiretta**: quando la *Thing* e il *Consumer* utilizzano protocolli diversi o si trovano su reti diverse. Possono essere messi in comunicazione tramite l'ausilio degli *Intermediary*.



3.4 Elementi principali

Vengono ora passati in rassegna gli elementi principali (*WoT Building Blocks*) che consentono di implementare l'architettura astratta.

3.4.1 WoT Thing Description

La specifica *WoT Thing Description* (TD) definisce l'*Information Model* basandosi su un vocabolario semantico e su una rappresentazione JSON. Una TD fornisce i metadati relativi a una *Thing* in modo che siano comprensibili sia da persone che da macchine. Sia il modello che la rappresentazione del TD sono allineati con i Linked Data, in questo modo le varie implementazioni possono scegliere di usare JSON-LD e i database a grafo per consentire l'elaborazione semantica dei metadati.

In generale in una TD sono contenuti i metadati riguardanti:

- informazioni generali (nome, ID, descrizione, ecc...);
- relazioni, definite attraverso *linking*, con altre *Thing*;
- *Interaction Affordance*;
- *Public Security Metadata*;
- *Protocol Binding*.

La Thing Description (TD) può essere vista come il punto di accesso ad una Thing per conoscere i servizi a disposizione e le risorse correlate. Può essere paragonata all'*index.html* in un sito Web.

La TD di una *Thing* viene creata e ospitata dalla *Thing* stessa e recuperata durante il processo di *discovery*. Tuttavia, la TD può anche essere ospitata esternamente se la *Thing* in questione ha risorse limitate o quando un dispositivo esistente viene adattato per diventare parte del WoT.

Per facilitare la gestione dei dispositivi, un pattern comune è quello di registrare le TD con una directory e fare in modo che i *Consumer* la memorizzino (*caching*) e vengano notificati in caso di cambiamenti.

3.4.2 WoT Binding Templates

L'IoT utilizza un'ampia varietà di protocolli per l'accesso ai dispositivi, perché ovviamente non esiste un unico approccio per ogni situazione. Dunque, la sfida maggiore del WoT è sicuramente quella di riuscire ad integrare le varie piattaforme IoT (OCF, 4 oneM2M, 5 Mozilla IoT, ecc...) e i dispositivi che non seguono uno standard preciso, ma che forniscono un'interfaccia su un protocollo di rete.

Le specifiche per il *Wot Binding Templates* sono un documento non normativo che fornisce una collezione di metadati utili come linee guida su come far interagire piattaforme IoT differenti. Per ogni piattaforma IoT deve essere creato un *Binding Template* che poi può essere riutilizzato da tutte le TD della piattaforma. I *Consumer* del sistema devono essere in grado di implementare i *Protocol Binding* richiesti attraverso un *Protocol Stack* che viene configurato a seconda delle informazioni fornite dalla TD (Figura 3.6)

I metadati che costituiscono il *Protocol Binding* abbracciano cinque dimensioni:

- **IoT Platform**: permette di gestire l'introduzione di modifiche proprietarie a livello applicativo, come HTTP *header* specifici o opzioni di CoAP, in un'applicazione IoT;

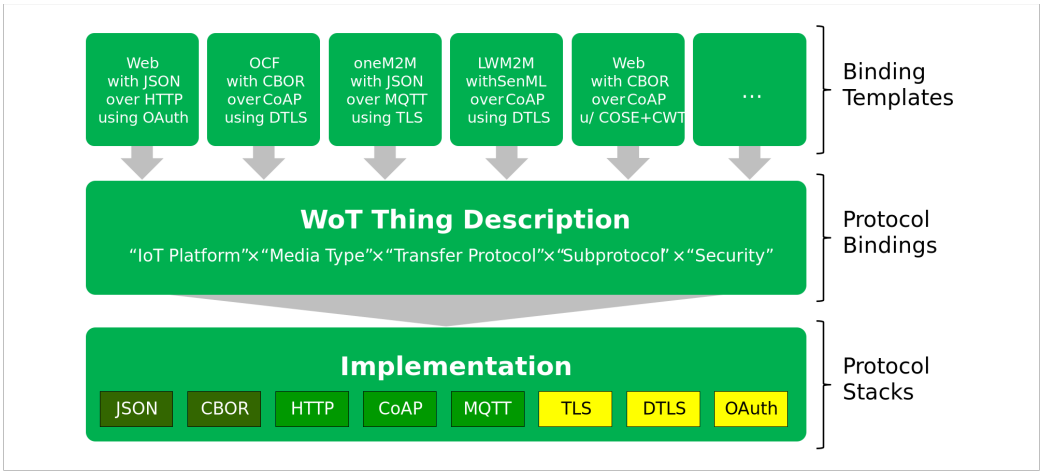


Figura 3.6: Dai *Binding Template* per la piattaforma IoT al *Protocl Binding* implementato dai *Consumer* [76]

- **Media Type**: permette l'identificazione delle diverse tipologie di formati utilizzati nello scambio dei dati;
- **Transfer Protocol**: protocolli standard utilizzati a livello applicativo senza opzioni specifiche dell'applicazione;
- **Subprotocol**: dato che i *Transfer Protocol* possono avere possibili estensioni e che tali estensioni non possono essere identificate dallo schema URI, queste devono essere dichiarate in maniera esplicita. Un esempio di *subprotocol* è il meccanismo di *long polling* per HTTP;
- **Security**: ai diversi livelli dello stack di comunicazione possono essere utilizzati diversi meccanismi di sicurezza e spesso contemporaneamente.

3.4.3 WoT Scripting API

Nonostante non sia necessario per rispettare lo standard, la raccolta *WoT Scripting API* fornisce una collezione di *ECMAScript-based API* simili a quelle dei Web browser. Queste specifiche definiscono la struttura e gli algoritmi dell'interfaccia di programmazione che permette agli script di ricercare,

consumare, produrre ed esporre *WoT Thing Description*. Il *runtime* del *WoT Scripting API* si occupa di creare le istanze degli oggetti locali che fungono da interfaccia per altre *Thing* e le *Interaction Affordances* (Properties, Actions, and Events).

3.4.4 WoT Security and Privacy Guidelines

Questa specifica fornisce delle linee guida per la sicurezza e la gestione della privacy di ognuno dei *building block*. Tali specifiche possono fornire un'indicazione sulla gestione della sicurezza nel sistema, ma la sicurezza deve essere valutata in ogni specifico caso di architetture WoT.

3.5 Architettura astratta del Servient

Come illustrato nella sezione precedente, il Servient è uno stack software che implementa i *building block* WoT. In particolare, permette di esporre e/o consumare *Thing*. A seconda del *Protocol Binding*, può svolgere un ruolo sia di server che di client. Da quì il nome Servient. In Figura 3.7 è possibile vedere l'implementazione di un Servient costituita da:

- Behaviour Implementation;
- Scripting WoT Runtime;
- Protocol Stack Implementation;
- System API.

Behaviour implementation

Il *behaviour* permette di descrivere il comportamento di una *Thing* che includono:

- comportamenti autonomi della *Thing* come la gestione dei la lettura dei sensori o i cicli di controllo degli attuatori;

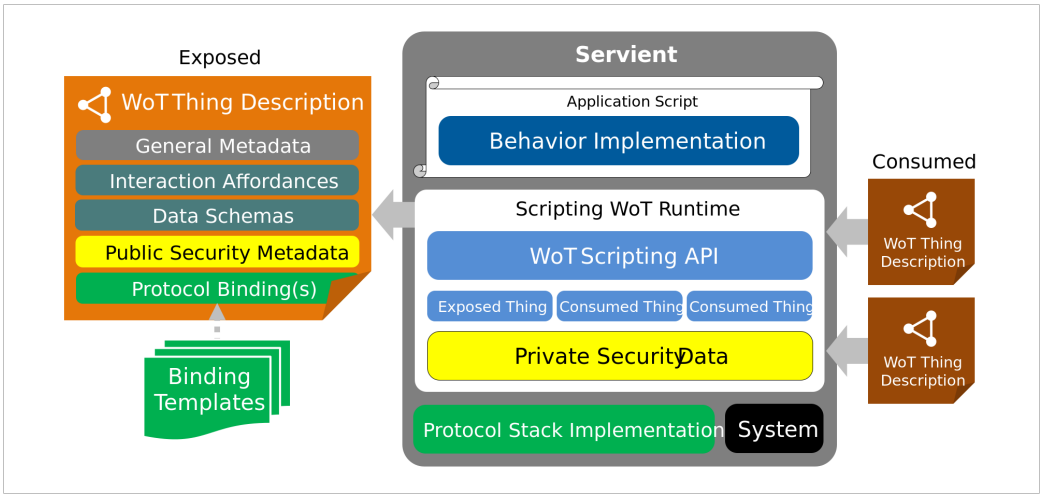


Figura 3.7: Implementazione di un Servient con WoT Scripting API (opzionali) [76]

- i gestori delle *Interaction Affordance* ossia le azioni concrete da eseguire nel momento in cui bisogna attivare un'*affordance*;
- il *consumer behaviour* che consiste nel controllo della *Thing* o nella creazione di un *mashup*;
- *intermediary behaviour* che viene svolto nel momento in cui il Servient è un proxy per la *Thing* o una composizione di *Thing*.

L'implementazione di un *behaviour* in un *Servient* definisce quali *Thing*, *Consumer* e *Intermediary* si trovano nel componente.

Il *behaviour* può essere definito in un qualunque linguaggio di programmazione, l'unico requisito è che l'*Interaction Affordance* sia visibile esternamente tramite interfaccia WoT.

WoT Runtime

Il *WoT Runtime* corrisponde all'implementazione dell'astrazione *Thing* e del suo *Interaction Model*. Il *WoT Runtime* fornisce l'*environment* per l'implementazione del *behaviour* e quindi la possibilità di esporre e/o consumare

Thing. Da ciò deriva che il *WoT Runtime* deve essere in grado di recuperare, processare, organizzate e soddisfare le WoT *Thing Description*. Deve inoltre interfacciarsi con l'implementazione del protocol stack del Servient, dato che permette di disaccoppiare l'implementazione del *behavior* dal *Protocol Binding*. Infine, solitamente si interfaccia anche con il sistema fisico sottostante (sensori e attuatori dell'hardware locale).

Ogni WoT runtime è costituito dai seguenti elementi:

- Un'interfaccia *application-facing* (es. API) per implementare il *behaviour*. Il WoT Scripting API (componente opzionale) definisce delle API in ECMAScript [27] che specificano l'interfaccia tra l'implementazione del *behavior* e lo script del *runtime*.

- La possibilità di creare un'istanza software della *Thing* basandosi sulla sua TD che fornisca un'interfaccia per l'implementazione del *behaviour*. Vengono identificate due possibili astrazioni delle *Thing*:

  - *Exposed Thing abstraction*: rappresenta una *Thing* che locale accessibile dall'esterno tramite lo stack di implementazione del Servient. Il behaviour per tali *Thing* permette un controllo completo della *Thing* esposta definendone i metadati, l'*Interaction Affordance* e i comportamenti autonomi.

  - *Consumed Thing abstraction*: rappresenta una *Thing* remota che deve essere acceduta dal *Consumer*. L'implementazione del *behaviour* è ristretto alla lettura dei metadati e all'attivazione dell'*Interaction Affordance* come specificato nella TD corrispondente.

- Il *Private Security Data* offre la possibilità di autorizzare, proteggere l'identità e la confidenzialità delle interazioni. Viene gestito dal WoT Runtime, ma è inaccessibile in maniera diretta dalle applicazioni.

Protocol Stack Implementation

Il *Protocol Stack* implementa l'interfaccia WoT delle *Thing* esposte e viene utilizzato dai *Consumer* per accedere all'interfaccia WoT delle *Thing* remote (tramite *Consumed Things*). Produce i veri e propri messaggi interagire nella rete su un certo protocollo. I *Servient* possono implementare più protocolli e quindi supportare protocolli multipli per consentire l'interazione con diverse piattaforme IoT.

System API

L'implementazione di un *WoT Runtime* può fornire l'accesso all'hardware locale o alle API di sistema per implementare dei *behavior* attraverso l'astrazione della Thing, come se fossero accessibili tramite protocollo di comunicazione. In questo caso, il WoT Runtime dovrebbe consentire di istanziare una *Consumed Thing* che internamente si interfaccia con API di sistema invece che con lo stack protocollare.

Un dispositivo può anche essere fisicamente esterno a un Servient, ma connesso tramite un protocollo proprietario o un protocollo non idoneo come interfaccia WoT. In questo caso, il *WoT Runtime* può accedere a dispositivi *legacy* con tali protocolli (es. ECHONET Lite, BACnet, X10, I2C, SPI, ecc.) tramite API proprietarie, ma li espone all'implementazione del *behavior* tramite l'astrazione della *Thing*.

Un *Servient* può quindi fungere da *gateway* per i dispositivi *legacy*. Questo dovrebbe essere fatto solo se il dispositivo non può essere descritto direttamente utilizzando una *WoT Thing Description*.

Capitolo 4

Sistemi di remote tracking

4.1 Sistemi di tracciamento remoto

Uno dei campi in cui l'IoT ha portato innovazione, è sicuramente quello del *remote tracking* dei veicoli. Come accennato nella Sezione 2.1, la necessità di ottimizzare la logistica ha reso essenziale il poter determinare dove si trova un veicolo in dato momento, quali tragitti ha percorso e in che stato si trova. I primi sistemi di tracciamento per la gestione delle flotte erano a tracciamento passivo, ossia il raccoglimento dei dati di GPS e sensori veniva effettuato a conclusione del viaggio e non in tempo reale. I sistemi in tempo reale, invece, permettono di acquisire i dati mentre questi vengono prodotti. Sono stati sviluppati diversi sistemi che permettono l'invio in tempo reale dei dati del veicolo tramite reti cellulare o satelliti e l'acquisizione di questi da parte dei data center [44]. Di seguito vengono passate in rassegna alcune soluzioni aziendali per il *tracking* dei veicoli.

4.1.1 Targa Telematics

Targa Telematics [36] è un'azienda italiana che grazie all'installazione di un dispositivo (*black box*) permette di trasformare veicoli tradizionali in veicoli *smart*. Offre diversi strumenti di gestione come: *Corporate Car Sharing* per la condivisione delle auto aziendali al fine di ottimizzare l'utilizzo; *fleet*

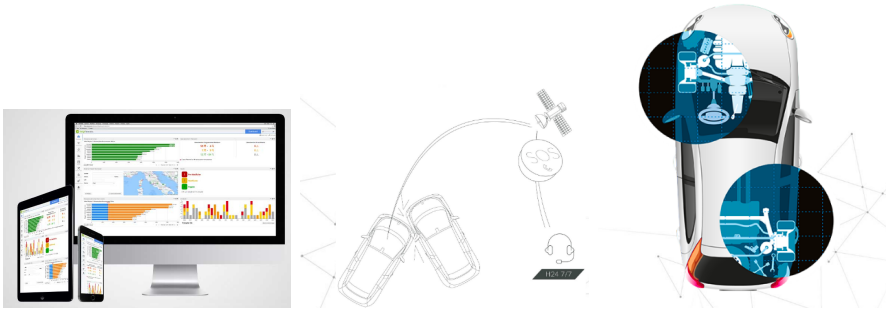


Figura 4.1: Panoramica prodotti e funzionalità Targa Telematics [36].

managment per la raccolta dei dati provenienti dai mezzi al fine di migliorare l'efficienza e diminuire i costi; diagnostica telematica per monitorare a distanza e in tempo reale livello del carburante, stato dei freni ed eventuali anomalie rilevate dalle centraline a bordo vettura (Figura 4.1).

4.1.2 Frotcom

Frotcom [29] è un'azienda portoghese fornitrice di sistemi di tracciamento veicolare e di gestione di flotte (Figura 4.2).

Offre diverse funzionalità come la possibilità di localizzare tramite GPS i veicoli della flotta, di recuperare i dati dai sensori in tempo reale, di pianificare ed ottimizzare i percorsi, gestione di allarmi e immobilizzazione remota del veicolo.

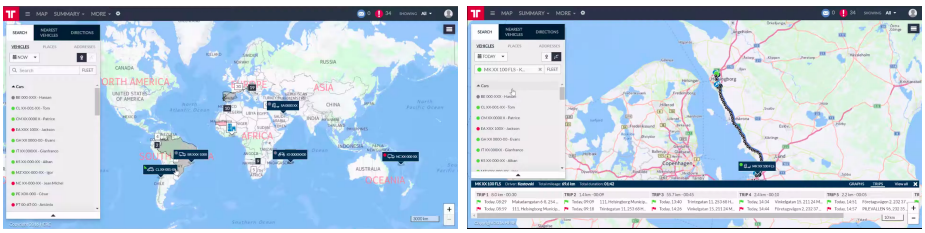


Figura 4.2: Viste applicazione Web Frotcom [29].

4.2 Tracciamento remoto di ebike

Oltre al mondo del *remote tracking* industriale, le tecnologie IoT hanno pervaso anche un settore relativamente nuovo: il settore delle biciclette elettriche o *ebike*. Per *ebike* si intende "biciclette dotate di un motore elettrico per fornire al ciclista assistenza alla pedalata" [43]. L'adozione delle *ebike* è un trend in continua crescita: paesi come la Cina vantano oltre 200 milioni di *ebike* utilizzate ogni giorno [80] [42] e paesi come la Svizzera ne promuovono l'utilizzo come mezzo di trasporto più ecologico [79]. La ragione di tale incremento è dovuto al fatto che le *ebike* si dimostrano essere un valido mezzo di trasporto urbano consentendo a chiunque di poterne fare uso minimizzando i tempi di spostamento [62]. L'integrazione di tecnologie IoT nelle *ebike* ha aperto un ventaglio di possibilità che i maggiori costruttori del settore hanno tentato di implementare. A seguire vengono passate in rassegna alcune delle principali soluzioni.

4.2.1 Sitael

Sitael [22] è un'azienda italiana che realizza soluzioni sia per il settore aerospaziale che per quello dell'*Internet of Things*. Recentemente è entrata nel settore delle *ebike* con il sistema ESB. ESB è un sistema nato per integrare *ebike*, *cloud computing* e *smartphone* (Figura 4.3a). A seconda dell'elettronica installata, è in grado di recuperare dalle *ebike* informazioni sulla localizzazione (GPS) e sula diagnostica per poi inviarle ai server dove queste vengono elaborate per sintetizzare informazioni utili sulla manutenzione, sul controllo del veicolo o sulla eventuale gestione di una flotta di veicoli (Figura 4.3b).

L'applicazione mobile permette di recuperare le informazioni in tempo reale per calcolare la vita della batteria disponibile e lo stato di usura dei componenti.

I veicoli equipaggiati con il sistema integrano moduli GPS e GPRS per poter controllare i parametri da remoto. Inoltre la presenza di tali moduli

(a) Panoramica sistema Sitael ESB [22].

(b) Panoramica funzionalità dei diversi prodotti Sitael ESB [22].

offre la possibilità di attivare/disattivare un allarme dallo smartphone e ricevere eventuali avvisi ogni qual volta la bici viene spostata dalla sua posizione originaria (grazie all'accelerometro).

4.2.2 Bosch

Bosch[12] produce un'ampia gamma di motori elettrici per tutte le esigenze corredati da ciclocomputer collegabili allo smartphone.

Uno tra questi è lo *SmartphoneHub* [14]. Si tratta di un computer di bordo integrabile con la maggior parte dei veicoli motorizzati dalla casa. Possiede un display da 1.5" LCD in grado di mostrare informazioni quali velocità e stato della batteria. Inoltre, è predisposto per ospitare uno smartphone che con la relativa applicazione consente di accedere a funzionalità più avanzate tramite connessione *Bluetooth*. Infine, quando connesso ad uno *smartphone*, permette di recuperare informazioni sul veicolo tramite l'assistente vocale (Amazon Alexa o Google Assistant).

Per gli utenti più esigenti, Bosch mette a disposizione due ciclocomputer (Kiox[13] e Nyon[13]) che uniti all'applicazione eBike Connect [25] (Figura **??**) consentono di tenere sotto controllo tutti i parametri sia del ciclista che della bicicletta stessa. A differenza dello SmartphoneHub offrono un display a colori di dimensioni sufficienti a consentire la funzione di navigazione.

Le principali funzionalità messe a disposizione dall'applicazione includono la registrazione dell'attività con sincronizzazione automatica sui sistemi *cloud*

Figura 4.4: Display C Yamaha montato su una ebike [20].

e *virtual coach* che suggerisce al ciclista se sta andando sopra o sotto la sua velocità media e infine la possibilità di aggiornare il firmware della centralina del motore.

4.2.3 Yamaha

Come Bosh anche Yamaha correda le proprie *ebike* con computer di bordo *smart*. Ad esempio il Display C[20] è un ciclocomputer dotato di display TFT a colori da 2,8 pollici associabile a tutti i motori prodotti dalla casa a partire dal 2019 (Figura 4.4). Offre la possibilità di visualizzare le principali informazioni sul viaggio e sul veicolo. Dispone di connettività *Bluetooth Low Energy* (LE) tramite la quale riesce a fornire funzionalità aggiuntive. La casa giapponese non sviluppa un'applicazione proprietaria, ma dichiara di essere pienamente compatibile con le principali applicazioni di tracking come *Komoot* e *Wellfit e-kit connection*.

4.2.4 Gocycle

Gocycle [31] è un'azienda inglese che produce biciclette elettriche dedicate all'utilizzo prettamente urbano. A differenza di Bosh e Yamaha, utilizza lo

Figura 4.5: Gocycle: *ebike* e applicazione [31].

smartphone dell'utente come ciclo-computer.

L'applicazione (GocycleConnect [32]), oltre alle classiche funzionalità di visualizzazione dei dati permette di avere funzionalità avanzate come: personalizzazione del livello di assistenza del motore e lo sforzo del ciclista grazie ad un sensore di coppia; impostazioni del cambio automatico; invio dei log al centro assistenza in caso di problemi; aggiornamento del firmware; possibilità di bloccare l'assistenza del motore della bici; assitente vocale per la gestione delle notifiche.

4.2.5 Shimano

Shimano[65] produce una vasta gamma di motori dedicati sia alla mobilità urbana che all'utilizzo fuoristrada. In particolare la gamma di motori *STEPS* viene utilizzata dalle principali case costruttrici di *ebike*[64]. Nonostante tutti i motori vengono forniti con ciclocomputer avente display integrato, la casa giapponese ha reso disponibile due applicazioni per *smartphone*, E-TUBE PROJECT [23] ed E-TUBE RIDE [24], che si integrano con i suoi sistemi. La prima ha il solo scopo di aggiornare il firmware e di regolare le impostazioni del cambio elettronico e del motore tramite connessione *Bluetooth*. La seconda consente di creare *dashboard* personalizzate per la visualizzazione dei parametri della *ebike* e integra un sistema per la gestione delle notifiche con *gestures*.

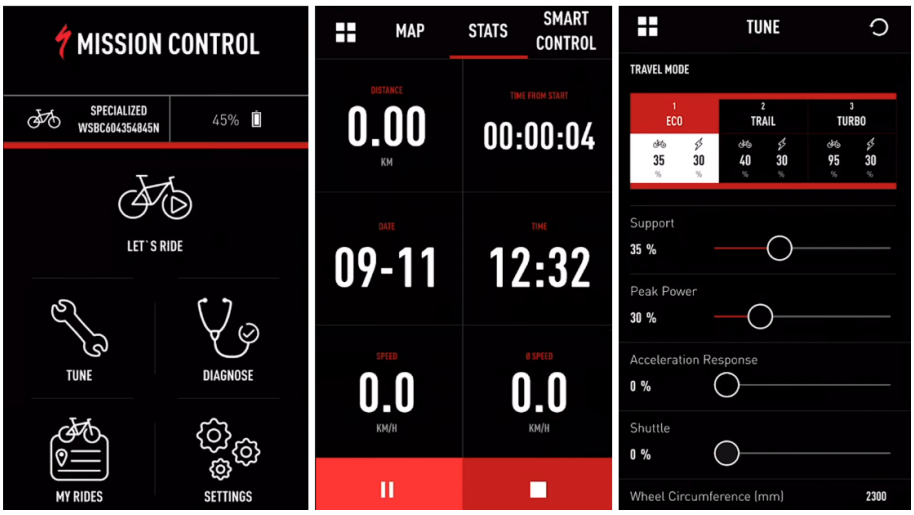


Figura 4.6: Schermata impostazioni applicazione Mission Control di Specialized [67].

4.2.6 Specialized

Specialized è una casa costruttrice di biciclette statunitense attiva dal 1974 [67]. Anch'essa offre una vasta gamma di *ebike* per tutti gli usi. A differenza delle concorrenti ha posto particolare accento sulla personalizzazione delle caratteristiche del motore ottenibile tramite l'applicazione Mission Control[51].

Questa offre un minuzioso sistema di personalizzazione e diagnostica del motore. Come mostrato in Figura 4.6 è possibile impostare: il picco di potenza erogata dal motore per ogni modalità (da eco a sport); la risposta in accelerazione, ossia quanto velocemente il motore risponde alla spinta del ciclista; lo shuttle, ossia il livello di assistenza del motore in base alla cadenza di pedalata; supporto, ossia la proporzione di assistenza in base alla spinta del ciclista per ogni modalità.

Infine, mette a disposizione dell'utente una *feature* per il controllo del consumo della batteria in relazione all'utilizzo. Tramite la funzione di *Smart Control*, un algoritmo controlla l'output del motore in base alla durata e alla distanza del tragitto impostato al fine di garantire sufficiente autonomia della

batteria.

4.2.7 VanMoof

VanMoof [70] è una casa produttrice di *ebike* olandese nata nel 2009. I veicoli da loro prodotti sono studiati per l'utilizzo urbano (Figura 4.7). Le biciclette sono in grado di rilevare eventuali tentativi di furto e sono dotate di dissuasore acustico e GPS. Di fatto, l'azienda si offre di recuperare gratuitamente il veicolo in caso di furto.

L'applicazione permette di visualizzare i parametri della bicicletta ed inoltre offre delle funzionalità *smart* che la distinguono dalla concorrenza come: attivazione/disattivazione del sistema antifurto; visualizzare la posizione della bicicletta; regolazione del cambio elettronico.

Figura 4.7: Principali modelli prodotti da VanMoof [70].

4.3 Funzionalità comuni

Dalla ricerca svolta si evince come la maggior parte dei sistemi dedicati alle *ebike* fa uso dello *smartphone* come ciclo-computer e come strumento per l'interfacciamento ai sistemi *cloud*, mentre le soluzioni *enterprice* tendono ad utilizzare dispositivi dedicati che si interfacciano con l'elettronica del veicolo da un lato e con i sistemi *cloud* dall'altro. Nonostante le soluzioni risultino essere molto diverse, è comunque possibile estrapolare le principali funzionalità comuni. Queste sono riassumibili con:

- la possibilità di monitorare in tempo reale la posizione e gli spostamenti dei veicoli;
- la possibilità di monitorare in tempo reale i parametri del veicolo (es. stato della batteria);
- una funzionalità di antifurto con *tracking remoto* via GPS;
- la possibilità di reperire informazioni sui veicoli tramite assistente vocale.

Parte II

Progetto CMDeBike

Capitolo 5

Progettazione

In questo capitolo è illustrata una panoramica della fase di progettazione del sistema sviluppato. Verrà definito il contesto nel quale si colloca il progetto con i relativi obiettivi. A seguire vi sarà la descrizione generale della soluzione architetturale. Infine, sarà esposta la rassegna dettagliata dei singoli componenti architetturali.

5.1 Obiettivi

Lo scopo del lavoro di tesi è la realizzazione di un sistema in grado di recuperare, salvare e visualizzare i dati prodotti da una bicicletta elettrica. L'esigenza nasce dalla recente acquisizione da parte di Costruzioni Motori Diesel (CMD) [18] dell'azienda Italmoto [41] produttrice di *ebike*. In particolare, l'obiettivo della collaborazione con l'azienda è quello di costruire un prototipo utilizzando come base una *ebike* prodotta da ItalMoto [69] e un Raspberry Pi [15] accoppiato ad una scheda LTE [48] per la connettività ad Internet. L'intenzione dell'azienda è quella di sviluppare un prodotto che sia versatile ed adattabile per poter rendere una comune *ebike* un dispositivo *smart*. Contestualmente è richiesto che l'utente finale sia in grado di visualizzare i dati generati dal *device/ebike*.

5.2 Requisiti

Il sistema deve fornire le seguenti funzionalità:

- **Registrazione**
  L'utente deve potersi registrare inserendo i seguenti dati:
  - email;
  - numero di telefono;
  - password;
  - *International Mobile Equipment Identity* (IMEI).

  Il codice IMEI viene utilizzato come identificatore della singola *ebike* e viene fornito dal costruttore. Un utente deve poter registrare più di una *ebike* durante la fase di registrazione.

- **Autenticazione**
  L'utente deve poter accedere al sistema utilizzando le credenziali create nella fase di autenticazione (email e password).

- **Data Presentation**
  L'utente, per ogni *ebike* da lui registrata, deve poter visualizzare:
  - la posizione corrente;
  - lo stato della batteria;
  - i tragitti percorsi con relativo stato dei sensori disponibili;
  - gli eventi di allarme lanciati.

- **Data Storage**
  I servizi *cloud* devono salvare tutte le informazioni provenienti dai sensori installati sulla *ebike*:
  - accelerometro;
  - giroscopio;

  - stato batteria;
  - *Global Positioning System* (GPS).

- **Antifurto**
  L'antifurto, quando attivato, deve segnalare (tramite SMS) un eventuale tentativo di furto.

- **Assistente vocale**
  Deve essere possibile richiedere informazioni sul veicolo tramite assistente vocale (*Amazon Alexa* [4]). In particolare lo stato di carica della batteria.

5.3 Architettura

In Figura 5.1 è rappresentata una panoramica dell'architettura. L'*edge device* (1) rappresenta una *ebike* connessa al sistema. Questo interagisce con la *Thing Description Directory* (2) che ne tiene traccia e con l'*Aggregator* (3) che ne legge i dati. Qust'ultimo si occupa di fare da intermediario tra il *Persistor* (4) e il servizio di *Backend* (5). È il componente che si occupa di recuperare ad intervalli regolari i dati dagli *edge device*, formattarli in un formato prestabilito e inviarli al *Persistor* per essere salvati sul *database*. Il *Backend* fornisce gli *endpoint* utili per la visualizzazione dei dati da parte della *Dashboard* (6). Inoltre, consente di comunicare direttamente con l'*edge device*. Infine, l'assistente vocale *Alexa* (7) ospitato dai servizi *AWS*, comunica con il *Backend* per recuperare informazioni utili sugli *edge device*.

Nella sezione che segue vengono passati in rassegna tutti i componenti architetturali sopra citati con un particolare focus sui dettagli che li caratterizzano.

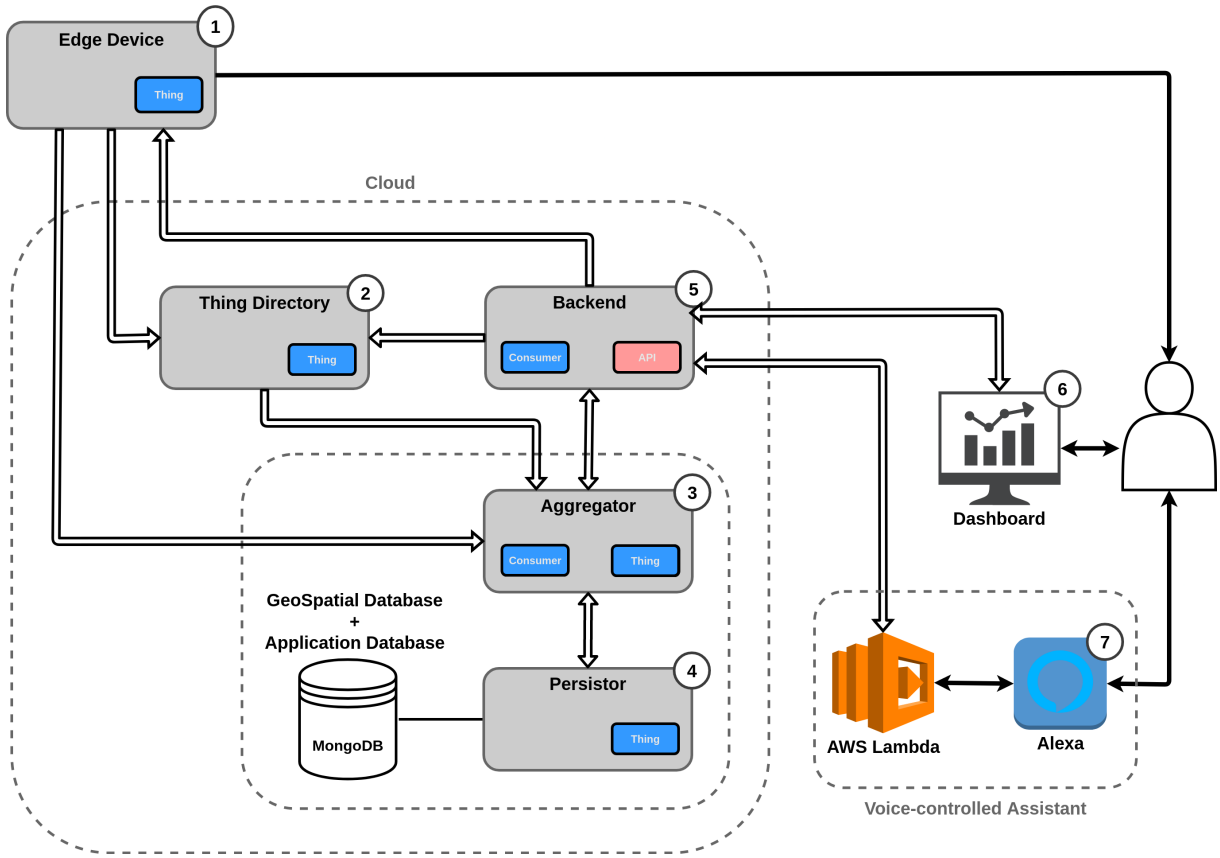


Figura 5.1: Panoramica dell'architettura che mostra le connessioni tra le componenti principali.

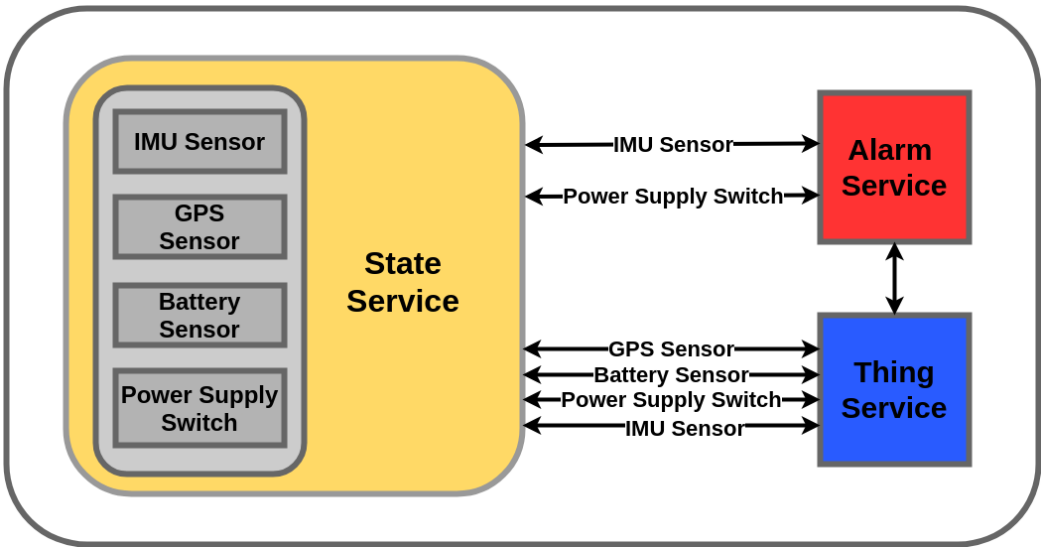


Figura 5.2: Panoramica dell'architettura dell'edge device con i relativi componenti. In giallo il componente che si interfaccia direttamente con l'*hardware*. In blu il componente che espone la WoT Thing. In rosso il servizio che si occupa di rilevare un tentativo di furto e inviare un SMS.

5.3.1 Componenti

Edge Device

L'*edge device* è il componente che viene fisicamente montato sulle *ebike*. È composto da tre elementi:

- **State Service**: Si tratta del componente che si interfaccia direttamente con i sensori. In particolare, legge i valori della piattaforma inerziale (nota anche come inertial measurement unit, o IMU), del GPS, della tensione della batteria (tramite un convertitore analogico-digitale) e i cambiamenti di stato del tasto di accensione/spegnimento. In Figura 5.2 è rappresentato in giallo.

- **Alarm Service**: È il componente che si occupa di leggere il sensore IMU, calcolarne il modulo e, se questo dovesse superare un certo valore

soglia, di inviare un SMS al numero configurato e inoltrare un *alert* al Thing Service. In Figura 5.2 è rappresentato in rosso.

- **Thing Service**: Si tratta del componente che espone lo stato dei sensori dell'edge device. In Figura 5.2 è rappresentato in blu. È un *Servient* la cui TD abbreviata è esposta nel listato 5.1. In particolare, sulla base del lavoro pubblicato in [45], la descrizione semantica delle proprietà è stata definita combinando diverse ontologie: Vehicle Signal Specification (VSS) [71], SOSA/SSN [63] e WoT.

  La *Thing* è definita come una classe *vsso:Vehicle* le cui *proprietà* sono sottoclassi di *vsso:Branch*. Queste a loro volta saranno oggetti che generano un certo numero di segnali (*vsso:hasSignal*). I segnali sono classi *vsso:ObservableSignal* e quest'ultimi sono sottoclassi di *sosa:ObservableProperty*. Un esempio è mostrato nel listato 5.2.

  Non tutte le *property* rispettano il pattern in quanto non tutte hanno un corrispettivo nell'ontologia VSS. Sono escluse *TheftProtection* ed *AliveMessageInterval*. Per la stessa ragione, sono stati escluse *action* ed *event*, anche se il pattern resta valido in quanto si sarebbe potuto definire le *property* di una *action* come vsso:ActuableSignal e mappare questi come sottoclassi di sosa:ActuableProperty.

  Le *action* messe a disposizione sono *SetPhoneNumber* per poter impostare il numero di telefono per gli SMS, *Update* per avviare la procedura di update da remoto.

  Infine, gli *event* generati dalla *Thing* sono *alive* e *alarm*. Il primo è utile alla TDD per verificare che un device sia effettivamente *online* e il secondo viene lanciato in caso venga rilevato un tentativo di furto.

La comunicazione tra i diversi componenti avviene tramite *message passing*. Ogni sensore viene letto dallo *State Service* e il risultato viene reso disponibile su uno specifico *socket*. Tutti i sensori, ad accezione della piattaforma inerziale, vengono acceduti tramite pattern *Request/Resplay*. Quando una richiesta di lettura di una *property* arriva sulla *WoT Thing*, questa la

inoltrerà un allo *State Service* sul *socket* corrispondente alla risorsa e quest'ultimo replicherà con un messaggio di risposta. Data la necessità di dover leggere costantemente il sensore IMU e data la necessità di condividere il risultato sia con l'*Alarm Service* che con il *Thing Service*, l'accesso al sensore è stato gestito tramite pattern *Publish/Subscribe*. Ciclicamente lo *State Service* legge il valore della IMU e pubblica gli aggiornamenti per i servizi sottoscritti sulla risorsa.

```
{
  "@context": [
    "https://www.w3.org/2019/wot/td/v1",
    { . . . }
  ],
  "@type": [
    "Thing",
    "vsso:Vehicle"
  ],
  "id": "868822040060674",
  "title": "EB-00",
  "description": "First cmd ebike prototype",
    . . .
  "properties": {
    "CurrentLocation": { . . . },
    "AngularVelocity": { . . . },
    "Acceleration": { . . . },
    "TheftProtection": { . . . },
    "BatteryTension": { . . . },
    "AliveMessageInterval": { . . . }
  },
  "actions": {
    "Update": { . . . },
    "SetPhoneNumber": { . . . }
  },
  "events": {
    "AlarmTriggered": { . . . },
    "Alive": { . . . }
  }
}
```

Listing 5.1: Thing Description dell'edge device.

```
"CurrentLocation": {
  "type": "object",
  "@type": "vsso:CurrentLocation",
  "sosa:madeBySensor": "vsso:GPS",
  "properties": {
    "vsso:hasSignal": {
      "@type": "sosa:Result",
      "properties": {
        "Speed": {
          "title": "Speed",
          "@type": "vsso:VehicleSpeed",
          "unit": "qudt:Speed",
          "type": "number"
        },
        "Longitude": {
          "title": "Longitude",
          "@type": "vsso:Longitude",
          "description": " Current longitude of vehicle, as reported by GPS"
              ,
          "unit": "qudt:AngleUnit",
          "type": "number"
        },
        "Latitude": {
          "title": "Latitude",
          "@type": "vsso:Latitude",
          "description": "Current latitude of vehicle, as reported by GPS.",
          "unit": "qudt:AngleUnit",
          "type": "number"
        }
      }
    },
    "sosa:ResultTime": {
      "type": "string",
      "@type": "xsd:DateTime"
    }
  }
```

Listing 5.2: Property *CurrentLocation* dell'*edge device*.

Thing Description Directory

Per poter interagire con gli *edge device* è necessario poterne recuperare le *Thing Description* (TD). Il componente che si occupa di tenere traccia dei *device* effettivamente *online* è la *Thing Description Directory* (TDD).

Questa, dunque, implementa un *Servient* per risolvere il problema della *findability* delle Thing. Si tratta di un componente che segue lo standard del W3C [77] ma si discosta leggermente da questo per integrare logiche utili alla modellazione del sistema.

Le *property* definite dalla TD della TDD sono:

- **searchJSONPath**: Come specificato dallo standard, permette di effettuare la ricerca sintattica tramite JSONPath. Gli unici oggetti (*edge device*) gestiti sono quelli *online*. Di fatto, se un oggetto risulta essere *offline*, non verrà catturato dalla *query*. Le richieste vengono effettuate tramite GET avente un JSONPath valido come parametro. Nel caso in cui il JSONPath risulti non valido, viene ritornato un errore 400 (Bad Request). Lo standard prevede anche altre due tipologie di errore, 401 per mancata autenticazione e 403 per permessi insufficienti, ma non avendo implementato meccanismi di autenticazione e di accesso, questi sono stati esclusi.

- **checkInterval**: Quando un *edge device* entra nel sistema, la TDD si registra all'evento *alive* generato dal *device*. Se il tempo trascorso dalla ricezione dell'ultimo evento è maggiore di un certo valore soglia, allora lo stato del *device* viene impostato ad *offline*. La *property checkInterval* consente di impostare la frequenza con la quale viene effettuato il controllo sullo stato dei *device*. Tale valore costituirà la soglia oltre la quale avviene il cambiamento di stato. La richiesta viene effettuata tramite PUT avente come *body* il valore da impostare. L'esito sarà 200 in caso di successo e verrà sovrascritto il valore di default.

Lo standard, inoltre, prevede quattro *action*: *createTD*, *updateTD*, *updatePartialTD* e *deleteTD*, per descrivere le operazioni REST di base per la creazione, l'aggiornamento e la rimozione delle risorse di tipo TD. Non essendo utili ai fini del prototipo del progetto, si è deciso di utilizzare la sola *action* di creazione della risorsa *createTD*. Questa viene invocata dagli *edge device* al momento dell'avvio tramite una richiesta POST contenente nel

corpo l'intera TD. Se l'operazione va a buon fine, la TD viene registrata e viene ritornato lo stato 200. In caso la TD sia già stata registrata, questa viene sovrascritta con la nuova. Non viene gestito il caso di *Thing* anonime, ossia sprovviste di identificatore univoco.

Infine, per quanto riguarda gli eventi, lo standard prevede un solo evento *registration* che segnala la creazione/registrazione di una nuova *Thing* e/o eventuali cambiamenti di stato della risorsa (update/delete). Dato che vi è la necessità di gestire la situazione in cui le *Thing* possono essere *offline*, si è optato per l'introduzione di due eventi:

- **newThing**: si tratta dell'evento generato quando una *Thing* viene registrata. Il dato emesso è il solo identificatore univoco della *Thing* che ha causato l'evento.

- **newThingStatus**: si tratta dell'evento generato quando una *Thing* cambia di stato da *offline* ad *online* e viceversa. Il dato emesso è formato dalla coppia *<id, stato>* dove l'*id* è identificatore univoco della *Thing* e lo stato è la stringa *online/offline*.

A differenza di quanto specificato dal Notification API dello standard, non è stata integrato il supporto per l'*event filtering* in quanto si è deciso di optare per l'utilizzo di due eventi separati.

Aggregator

L'*Aggregator* è il componente che si occupa di gestire le logiche per il salvataggio dei dati e l'interfacciamento con il servizio di *Backend*. All'avvio consuma la TD della TDD e si sottoscrive agli *event newThing* e *newThingStatus* generati da questa. Quando un *edge device* entra nel sistema, l'*Aggregator* viene informato dalla TDD e a quel punto inizia il processo di salvataggio dei. Il componente è studiato per essere configurabile, di fatto, è possibile configurare sia quali *property* di un *edge device* salvare, sia le *policy* da utilizzare per il salvataggio. Per ogni *edge device* che si desidera

```
const x: Exp = {
    type: 'Pow',
    elem: {type: 'Val', val: "AngularVelocity.vsso:hasSignal.x"},
    exponent: {type: 'Num', val: 2}
}
const y: Exp = {
    type: 'Pow',
    elem: {type: 'Val', val: "AngularVelocity.vsso:hasSignal.y"},
    exponent: {type: 'Num', val: 2}
}
const z: Exp = {
    type: 'Pow',
    elem: {type: 'Val', val: "AngularVelocity.vsso:hasSignal.z"},
    exponent: {type: 'Num', val: 2}
}

const mapping: Statement[] = [
    {

    type: 'Ass',
    left: {type: 'Var', val: "properties.AngularVelocityMag"},
    right: {type: 'Sqrt', elem: {
        type: 'Plus',
        left: x,
        right: {type: 'Plus', left: y, right: z}}}
   }

]
```

Figura 5.3: Esempio di pre-aggregazione che associa i valori letti dall'accelerometro sui tre assi, alla property *AnguarVelocityMag* calcolandone il modulo.

gestire, è necessario specificare una configurazione definita mediante un linguaggio *ad hoc* dove è possibile descrivere la computazione desiderata sotto forma di albero. In particolare, la configurazione è data da tre elementi: il *ComputeNode*, l'*EventNode* e lo *State*.

Il *ComputeNode* permette di specificare la *policy* relativa al campionamento dei dati. Questo può essere di diversi tipi:

- **EmptyNode**: Rappresenta un nodo privo di computazione. Si tratta del nodo foglia dell'albero.

- **ActionNode**: Rappresenta il nodo che permette di invocare le *action* su una *Thing*. In particolare, viene utilizzato per invocare le *action* del *Persistor*.

```
Exp := Null | Num | Val | Exp Plus Exp | Exp Mult Exp
     | Exp Div Exp | Sqrt Exp Exp | Pow Exp Exp
```

Figura 5.4: Grammatica delle espressioni.

- **LoopNode**: Rappresenta il nodo che permette di eseguire un *ComputeNode* ad intervalli di tempo regolari. Inoltre permette di specificare come i dati letti da una *Thing* (*edge device*) debbano essere mappati su una *feature* GeoJSON [61]. La scelta dello standard GeoJSON è stata fatta considerando il dominio applicativo in quanto i dati salvati hanno bisogno di essere identificati spazialmente e temporalmente.

  Oltre al semplice *mapping*, l'*Aggregator* consente di fare operazioni di preaggregazione mediante le espressioni matematiche di base definite dalla grammatica in Figura 5.4. Un esempio di preaggregazione è presentato in Figura 5.3 dove i valori letti dall'accelerometro sui tre assi vengono mappati sulla property AnguarVelocityMag calcolandone il modulo. In Figura 5.5 è mostrato il risultato.

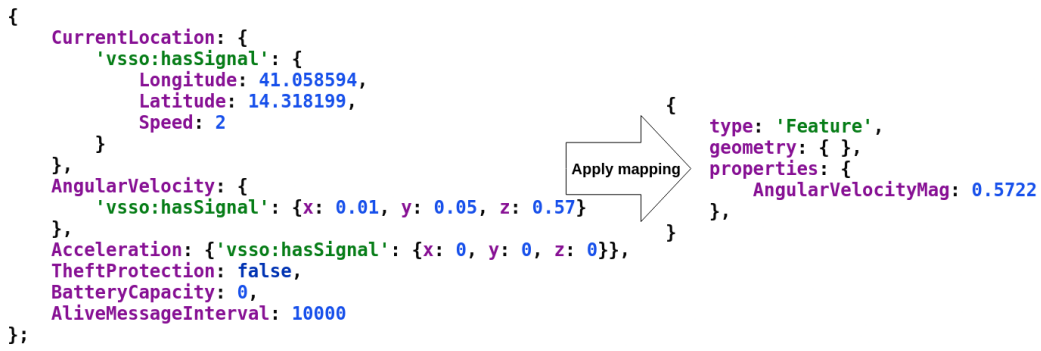


Figura 5.5: Risultato della preaggregazione definita in Figura 5.3 a partire dai dati letti dall'*edge device*. È possibile notare come l'oggetto *feature* creato presenti solo il modulo della velocità angolare.

- **ConditionNode**: Si tratta dell'unico nodo in grado di creare *branch* nella definizione dell'albero. In particolare permette di definire una condizione e due *ComputeNode*, uno per ogni flusso possibile (true/false). La valutazione del nodo consiste nella valutazione della condizione e, a seconda del risultato (true/false), viene valutato il corrispondente *ComputeNode*. La grammatica che definisce una condizione, è mostrata in Figura 5.6.

```
Condition := Bool | Exp > Exp | Exp < Exp
    | Exp == Exp | Condition == Condition
    | Exp != Exp | Condition != Condition
    | Condition && Condition | Condition || Condition
```

Figura 5.6: Grammatica delle condizioni.

L'**EventNode** permette di specificare le *policy* relative alla gestione degli eventi generati da uno specifico *edge device*. Questo conterrà l'insieme degli eventi a cui si è interessati. La valutazione di un *EventNode* consiste nella sottoscrizione all'evento specificato e all'esecuzione del corrispondente *ActionNode* quando questo viene generato.

Infine lo **State** rappresenta lo spazio di memoria nel quale gli *ActionNode* recuperano le variabili e salvano eventuali risultati. È inoltre possibile definire le variabili utili per iniziare la computazione. Se, ad esempio, si volesse far partire la computazione con una variabile IMEI, sarebbe sufficiente creare un oggetto contente una proprietà IMEI e il suo valore. In sintassi *ECMAScript* si potrebbe scrivere "{IMEI: 123456789}" dove IMEI sarà il nome della variabile di Stato acceduta dall'*ActionNode* e "123456789" il suo valore. Una variabile può essere recuperata o scritta specificando lo *string path* dell'oggetto all'interno dello State.

Un esempio di *ComputeNode* espresso in sintassi *ECMAScript* è presente in Figura 5.7.

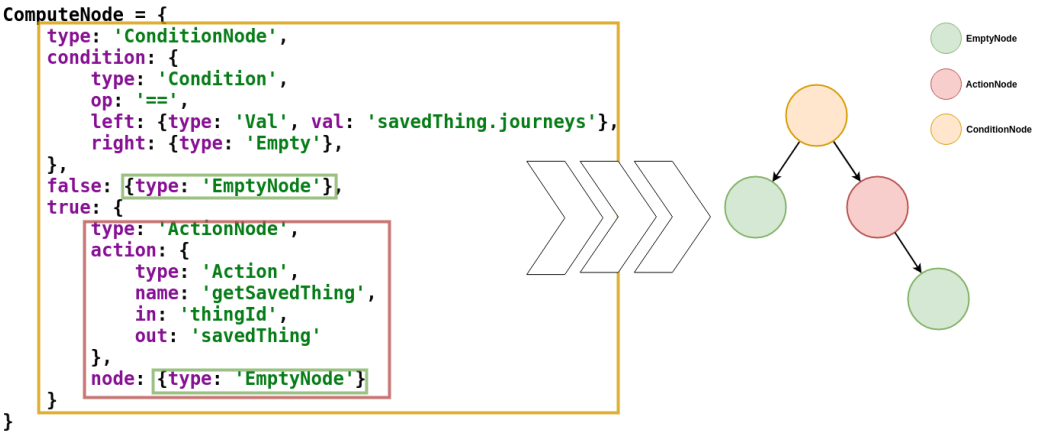


Figura 5.7: Esempio di *ComputeNode*. La radice è un *ConditionNode*, nel caso in cui la valutazione della condizione sia falsa (false), viene interrotta computazione con un *EmptyNode* (verde). In caso risulti vera, viene valutato un *ActionNode* (rosso). In particolare viene eseguita la *action getSavedThing* che prende in input la variabile *thingId* precedentemente specificata nello *State*.

Persistor

Si tratta della thing che comunica direttamente con il *database* e lo espone sotto forma di *WoT Thing*. Per capire quali caratteristiche dovesse avere il *Servient*, si è partiti da un'analisi del dominio applicativo e dei requisiti. Si vuole:

- poter salvare i dati prodotti dai sensori di un *edge device*;
- avere un accesso rapido ai dati facenti parte di un singolo tragitto;
- poter salvare gli eventi generati da un *edge device*;
- poter salvare informazioni sugli utenti

Le entità coinvolte sono dunque:

- **Utente**: L'entità utente possiede informazioni di base utili per identificare questo all'interno del sistema. In particolare avrà:
  - id: un identificativo univoco;
  - nome;
  - password.
- **Thing**: L'entità *Thing* permette di identificare un *edge device*. Inoltre, deve contenere informazioni sulla frequenza di campionamento con la quale i dati devono essere salvati. In particolare possiede:
  - id: identificativo univoco della *Thing*;
  - name: il nome con il quale viene visualizzata una *Thing*;
  - update_freq: la frequenza di campionamento espressa in secondi;
  - lastFeature: l'identificativo dell'ultima *feature* salvata;
  - lastJourney: l'identificativo dell'ultimo viaggio salvato;
  - userId: l'identificativo dell'utente che possiede la l'*edge device*;

  - imei: il codice IMEI dell'*edge device*.

  Dato che si vuole poter recuperare gli *edge device* a partire dal codice IMEI, tale campo costituisce un indice.

- **Feature**: Gli *edge device* possono essere equipaggiati con differenti sensori ma i dati da questi prodotti hanno senso nel contesto applicativo solo se corredati da *timestamp* e coordinate geografiche. Per rendere il modello dei dati condivisibile con future altre applicazione e, al contempo, per renderlo il più estensibile possibile senza impattare negativamente sulle performance in fase di recupero dei dati, si è deciso di mappare le proprietà esposte dagli *edge device* su oggetti GeoJSON come mostrato nel listato 5.3.

```
1  {
2    "type": "Feature",
3    "geometry": {
4      "type": "Point",
5      "coordinates": [44.488465, 11.341348]
6    },
7    "properties": {
8      "time": 2020-12-21T10:26:07.165+00:00,
9          . . .
10   }
11 }
```

Listing 5.3: Esempio GeoJSON con annesse proprietà temporali.

Gli oggetti vengono dunque rappresentati dalla entità *Feature*. In particolare, per poter salvare una *feature* occorrono almeno:

  - id: identificativo univoco;
  - coordinate: i valori di longitudine e latitudine espressi in gradi decimali;
  - time: il timestamp (UTC) relativo al momento in cui il dato viene salvato.

Le *feature* dovranno essere indicizzate sia sulle coordinate spaziali che sulle coordinate temporali per permettere di avere prestazioni ottimali in entrambi i contesti di utilizzo.

- **Viaggio**: È l'entità che modella i possibili viaggi che l'utente può effettuare dove per viaggio si intende l'insieme di *feature* raccolte durante un certa tratta. È composta da:
  - id: identificativo univoco;
  - features: lista degli identificativi delle *feature* salvate durante il viaggio;
  - thingId: identificativo della *Thing* a cui è associato il viaggio;
  - startDate: *timestamp* del momento di inizio del viaggio;
  - endDate: *timestamp* del momento di fine del viaggio.

  Gli attributi di inizio e fine viaggio devono essere indici per filtrare temporalmente i viaggi stessi.

- **Evento**: È l'entità che modella gli eventi generati da un *edge device*. In particolare possiede:
  - id: identificativo univoco;
  - thingId: identificativo dell'*edge device* che lo ha generato;
  - nome: nome dell'evento;
  - date: *timestamp* relativo al momento di ricezione dell'evento.

Per garantire il massimo della versatilità in caso si dovesse aggiungere nuove property alle feature, si è deciso di optare per un database NoSQL, in particolare MongoDB. Tutte le entità sopra descritte vengono salvate come *document* in una *collection* predefinita.

Per poter interagire con le entità, il *Servient Persistor* espone una serie di *action* specifiche dedicate al salvataggio e una generica per la lettura (executeQuery). La scelta è dovuta alla necessità di sapere a priori quale dato

si sta salvando. Per la fase di recupero dei dati invece, è possibile comporre *query* arbitrariamente complesse da utilizzare come parametro della *action*. Queste verranno inoltrate al database sottostante.

Backend

Il *Backend* è il componete che espone le API utilizzate dai servizi di *frontend* (*Dashboard* e *Alexa*). La gestione delle chiamate avviene tramite controller specifici costruiti per rispecchiare le operazioni possibili sulle entità gestite dal sistema (*user*, *Things*, ecc..). La logica di *business* è interamente delegata ai *service provider* del Backend stesso. In particolare vi è un *service* che ingloba il *Servient* utile per poter consumare le TD di *Aggregator* e TDD. La prima è necessaria per poter invocare le *action* per il recupero e il salvataggio dei dati, mentre la seconda è necessaria per poter invocare le *action* di uno specifico *edge device*. Nel caso in cui, ad esempio, si voglia poter impostare il numero di telefono di un *edge device*, è possibile invocare la relativa API sul *Backend*. La chiamata scatena in un'invocazione della *property* searchJSONPath della TDD al fine di recuperare la TD dell'*edge device* e infine accedere alla *action SetPhoneNumber* di questo.

Dashboard

La *Dashboard* è il componente che consente la visualizzazione dei dati da parte dell'utente. Questa comunica esclusivamente con il Backend. Sulla base di quanto fatto dalla concorrenza [29] 4.3b, si è deciso di suddividere le principali funzionalità in schermate navigabili tramite *navbar*: Mappa, Viaggi, Grafici, Eventi, Account e Veicoli.

La modellazione è stata fatta utilizzando il software per realizzazione di *wireframe* Balsamiq [11]. Esempi delle schermate sono presenti in Figura 5.8.

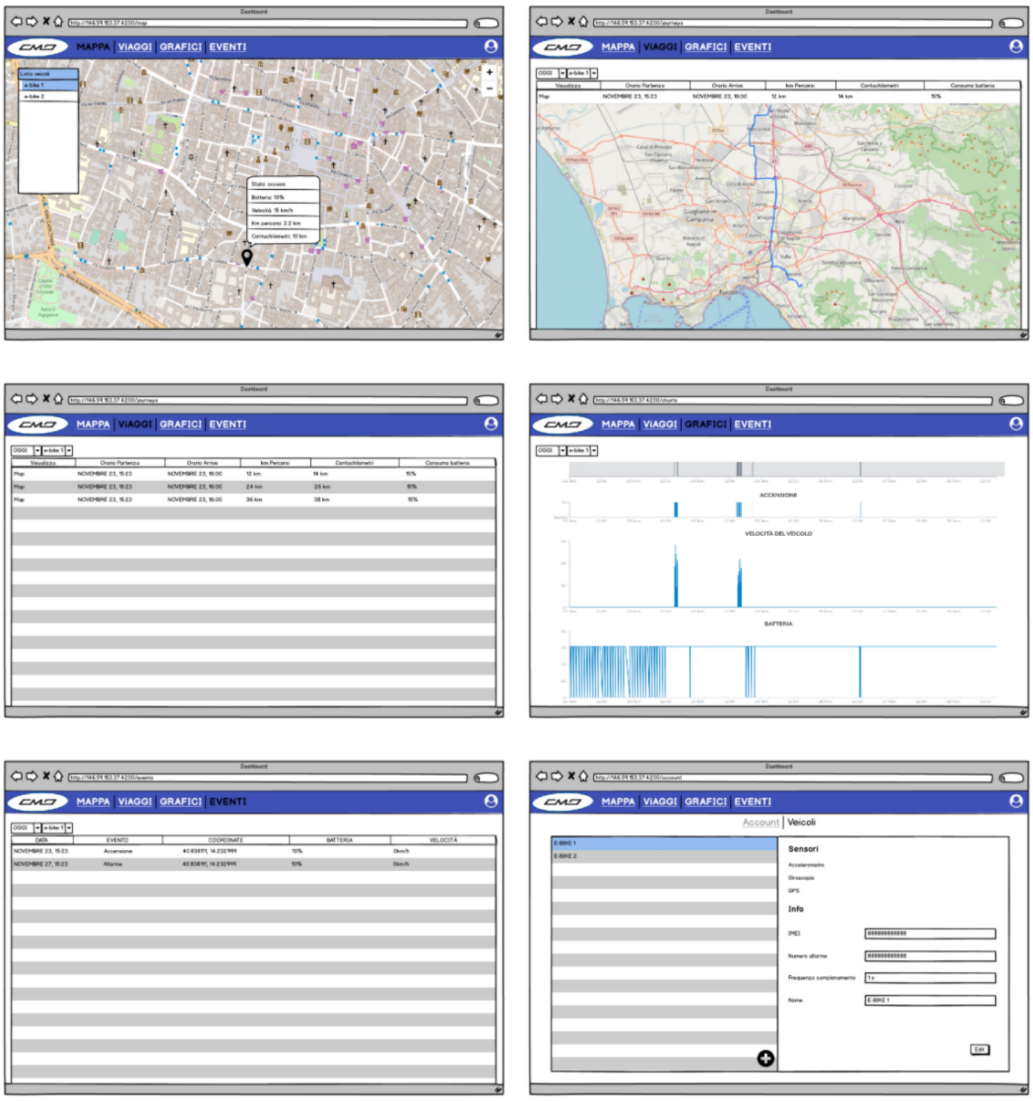


Figura 5.8: *Wireframe* delle principali schermate della *Dashboard* modellate con Balsamiq.

Skill Alexa

Amazon Alexa, o anche solo Alexa, è un assistente virtuale intelligente sviluppato da Amazon [5]. È capace di gestire l'interazione vocale con gli utenti tramite un sofisticato interprete di linguaggio naturale. Alcune funzioni di base sono native, mentre altre possono essere aggiunte tramite

il meccanismo delle *skill*. Dal punto di vista del programmatore, una *skill* non è altro che un servizio che si interfaccia con il motore di riconoscimeto del linguaggio naturale di Alexa per poter definire che esperienza deve può avere un certo utente che interagisce con la skill. Il codice delle *skill* può essere ospitato da un server oppure può essere eseguito sui servizi Amazon (AWS Lambda). Per semplicità si è scelto di avvalersi della seconda opzione. La *skill* Alexa, dunque, comunica con il servizio di *Backend* per fornire un'interfaccia vocale all'utente.

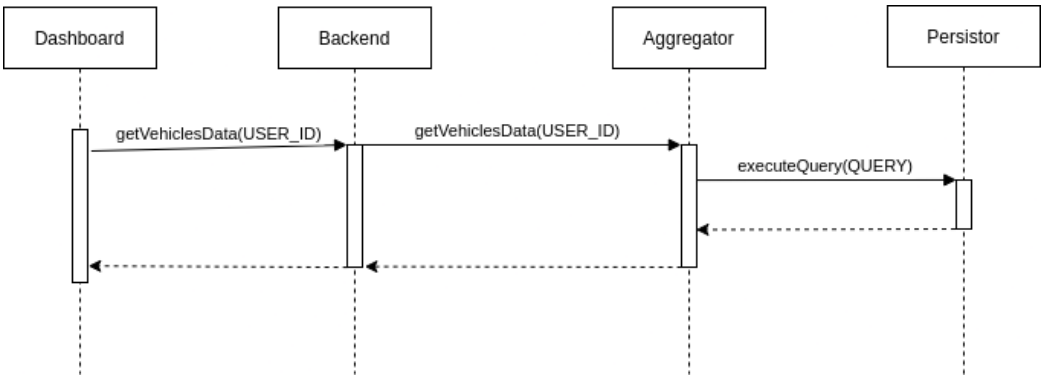


Figura 5.9: Sequenza di azioni che descrive la relazione tra i servizi *Dashboard*, *Backend*, *Aggregator* e *Persistor* nel caso della visualizzazione dello stato dei veicoli posseduti da uno specifico utente.

5.3.2 Flussi

- **Visualizzazione stato veicoli**
  L'utente vuole poter visualizzare sulla mappa l'ultima posizione nota del veicolo con relativo stato dei sensori.

  1. L'utente accede alla *landing page* della *Dashboard* contenete la mappa.
  2. La *Dashboard* effettua una chiamata al *Backend* contenente lo *userId* per recuperare l'ultimo punto dei veicoli posseduti dall'utente.
  3. Il *Backend* invoca la *action getVehiclesData* dell'*Aggregator* passando lo *userId* come parametro.
  4. L'*Aggregator* costruisce una *query* per recuperare le informazioni richieste invocando la *action executeQuery* del *Persistor*. La *query* permette di recuperare tutte le *Thing* appartenenti all'utente con lo *userId* specificato.
  5. Il *Persistor* esegue la query sul database e ne ritorna il risultato.

  Il diagramma di sequenza in Figura 5.9 mostra l'intero processo.

- **Visualizzazione lista viaggi effettuati da un veicolo**
  L'utente vuole poter visualizzare la lista dei viaggi effettuati da uno specifico veicolo.

  1. L'utente accede alla sezione viaggi della *Dashboard*.
  2. La *Dashboard* richiede al *Backend* la lista dei veicoli appartenenti all'utente.
  3. Il *Backend* invoca la *action getVehicles* dell'*Aggregator* utilizzando lo *userId* come parametro.
  4. L'*Aggregator* compone la *query* da utilizzare come parametro per invocare la *executeQuery* del *Persistor*.
  5. L'utente seleziona il *device* di interesse e l'intervallo temporale desiderato.
  6. La *Dashboard* fa una richiesta al *Backend* per ottenere la lista dei viaggi specificando il range temporale (UTC) e l'id del veicolo.
  7. Il *Backend* invoca la *action getJourneys* dell'*Aggregator* con i dati forniti dalla *Dashboard*.
  8. L'*Aggregator* compone la *query* specificando l'entità di interesse (*journey*) con il filtro temporale e l'id dell'*edge device*. Infine invoca la *executeQuery* del *Persistor*.

  Il diagramma di sequenza in Figura 5.10 mostra l'intero processo.

- **Arrivo nuovo *edge device* nel sistema**

  1. L'*edge device* si registra alla TDD inviando la propria TD.
  2. La TDD notifica l'*Aggregator* generando un *event newThing* contenente l'identificativo dell'*edge device* (codice IMEI).
  3. L'*Aggregator* recupera la TD del *device* appena entrato nel sistema facendo una ricerca sintattica sulla property *searchJSONPath* della TDD.

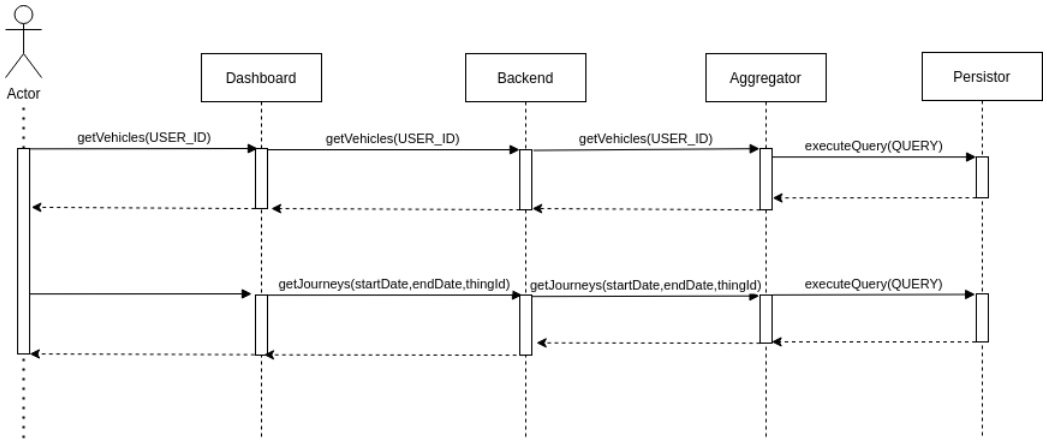


Figura 5.10: Sequenza di azioni che descrive la relazione tra l'utente e i servizi *Dashboard*, *Backend*, *Aggregator* e *Persistor* nel caso della visualizzazione della lista dei viaggi effettuati con uno specifico veicolo.

4. L'*Aggregator* recupera la configurazione dell'*edge device* e lancia la valutazione dei corrispondenti nodi.
5. Ad intervalli regolari, a seconda della frequenza di campionamento impostata, l'*Aggregator* legge le *property* dell'*edge device*, le mappa su un oggetto GeoJSON e infine esegue la *policy* definita nel *ComputeNode*.
6. L'esecuzione del *ComputeNode* scatenerà una serie di invocazioni alle *action* messe a disposizione dal *Persistor*.

Il diagramma di sequenza in Figura 5.11 mostra l'intero processo.

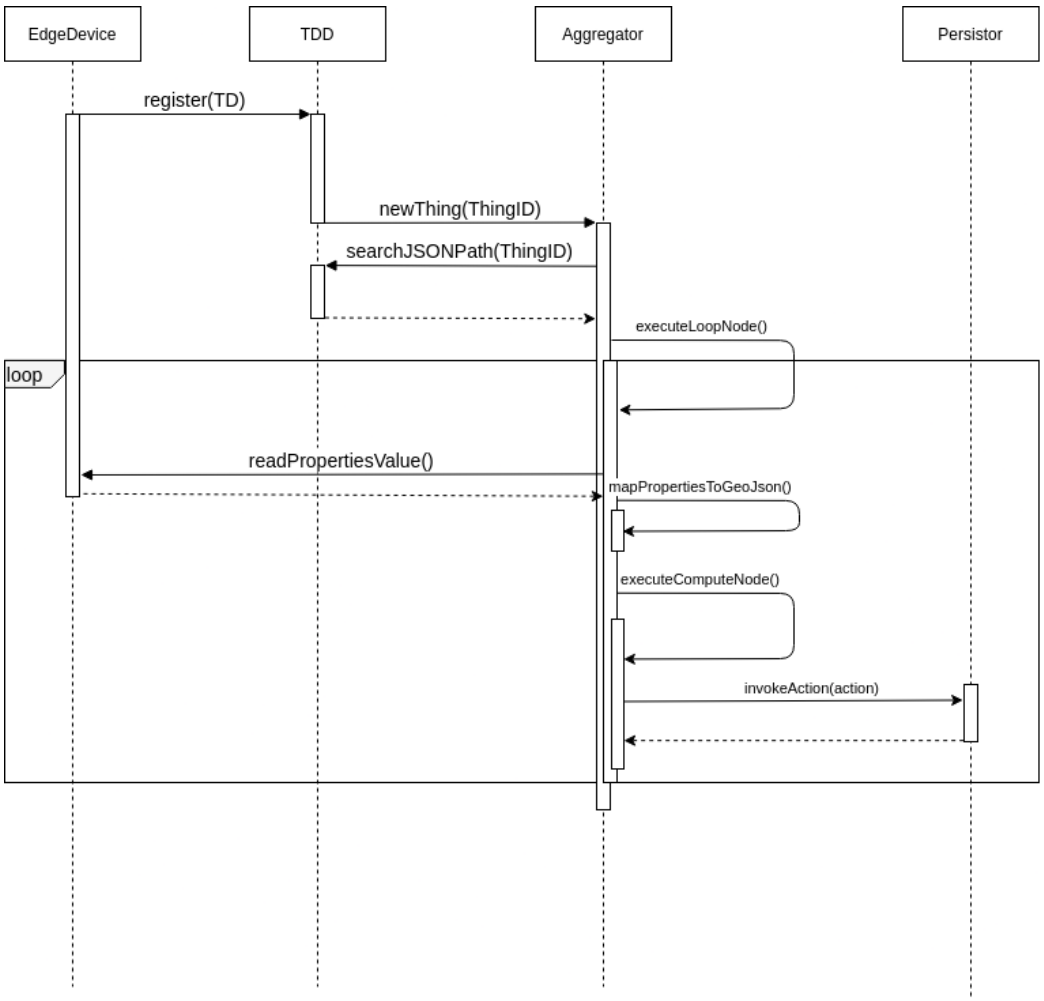


Figura 5.11: Sequenza di azioni che descrivono le interazioni tra l'*edge device* e i servizi TDD, *Aggregator* e *Persistor* nel caso in cui l'*edge device* registri per la prima volta la propria TD.

Capitolo 6

Implementazione

In questo capitole vengono illustrate le tecnologie utilizzate con un focus specifico sull'implementazione dei singoli componenti descritti nel capitolo precedente. Verranno inoltre citate le principali problematiche e le soluzioni trovate.

6.1 Tecnologie

ZeroMQ

ZeroMQ [84] è stato utilizzato per implementare la comunicazione tramite *message passing* nell'*edge device*. Si tratta di una libreria *open-source* sviluppata in C e pensata per applicazioni distribuite e concorrenti. Permette di avere una coda dei messaggi, ma, a differenza dei *message-oriented middleware*, può funzionare senza la necessità di avere un *message broker*. Difatti lo "Zero" nel nome sta per zero broker.

ZeroMQ mette a disposizione dei *socket* rappresentanti connessioni *many-to-many* tra *endpoint*. Il collegamento tra questi può avvenire utilizzando diversi pattern come ad esempio *request-replay* e *publish-subscribe*. Ogni pattern definisce una particolare topologia di rete. Ad esempio *request-replay* definisce un *service bus*, *publish-subscribe* definisce un *data distribution tree*,

ecc... Tutti i messaggi che transitano dai *socket* sono atomici e vengono trattati come dati *Binary Large OBject* (BLOB).

Inoltre, la libreria fornisce API per circa venti linguaggi differenti. All'interno del progetto sono state utilizzate le API per Javascript/TypeScript e Python.

NestJS

NestJS è stato utilizzato per l'implementazione del *Backend*. Si tratta di un *framework open-source* per la creazione di applicazioni *server-side* altamente efficienti e scalabili [21]. Si basa su Node.js ed è interamente sviluppato in TypeScript. Utilizza *framework* server HTTP come Express o Fastify, ma a differenza di questi, definisce un'architettura chiara e semplice basata su pochi componenti (*controller*, *module* e *provider*) e già impostata al momento della creazione di un progetto (tramite client integrato). Questo permette di suddividere applicazioni complesse in microservizi riducendo i tempi di ingegnerizzazione. I componenti architetturali principali di NestJS sono:

- **Module**: utilizzato per organizzare il codice e suddividere le funzionalità in unità logiche riutilizzabili. Le classi TypeScript utilizzate come moduli sono decorate con il decoratore *@Module* che fornisce metadati che Nest utilizza per organizzare la struttura dell'applicazione. Ogni applicazione deve avere almeno un modulo (root) che Nest utilizza per costruire il grafo applicativo (Figura 6.1). Quest'ultimo viene utilizzato per creare le strutture dati che consentono di risolvere le dipendenze di e tra *module* e *provider* (*Dependency Injection*).

- **Providers**: anche chiamati *service*, sono un modo per astrarre la complessità e la logica di business. Questi possono essere iniettati nei *controller* o in altri provider. Un *service* in Nest.js è una classe TypeScript decorata con il decoratore *@Injectable*. Per il progetto sono stati creati

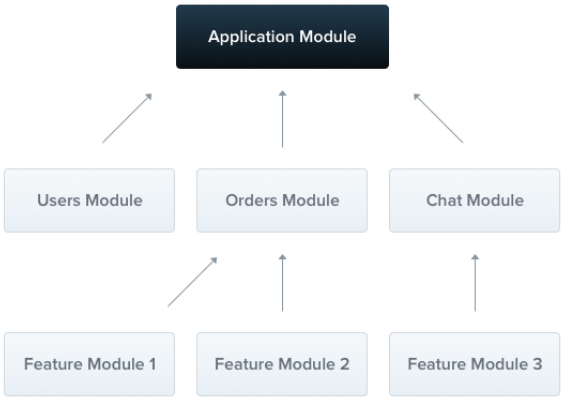


Figura 6.1: Esempio di application graph di NestJS [52].

dei *service* che ospitano i *Servient* utilizzati per consumare le TD e per interagire con le Wot Thing.

- **Controllers**: sono i componenti responsabili della gestione delle richieste (Figura 6.2). Il loro compito è quello di ritornare al chiamante le risposte appropriate. Il meccanismo di *routing* seleziona quale *controller* deve ricevere determinate richieste. Spesso, ogni *controller* ha più di un *path* e *path* diversi possono eseguire azioni diverse. Un *controller* è definibile come una classe TypeScript decorata con il decoratore *@Controller*. Al decoratore viene specificato come input quale *path* si desidera che il *controller* gestisca.

Angular

Angular [6] è un *framework front-end open source* per la creazione di applicazioni Web *single page* dinamiche e moderne. Nasce a partire da AngularJS, ma a differenza di questo è interamente scritto in Typescript. È sviluppato principalmente da Google e la prima versione (Angular 2) risale al 2016 [7].

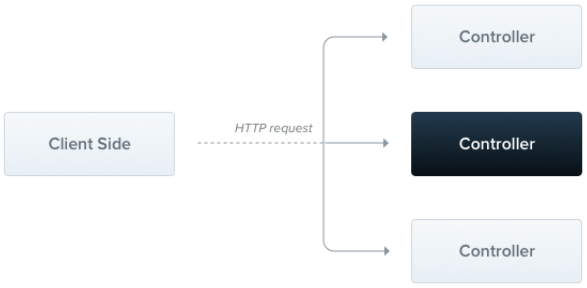


Figura 6.2: Esempio richiesta da parte di un *client* verso i *controller* [19].

L'architettura di Angular si basa sul concetto di *component* che sono organizzati in *NgModule*. Questi ultimi permettono di raccogliere il codice in insiemi funzionali. Un'applicazione Angular è definita dal suo insieme di *NgModules*. Ogni applicazione deve avere almeno un modulo (root).

I *component*, invece, permettono di definire le viste. Queste non sono altro che i componenti grafici che Angular può selezionare e modificare in base alla logica del programma. I *component* fanno uso dei *service* per inglobare la logica di business e la comunicazioni con servizi esterni (es. *Backend*). I *sevice provider* possono essere iniettati nei *component* come dipendenze, il che rende codice modulare, riusabile ed efficiente.

I *module*, i *component* e i *service* sono semplicemente classi che vengono decorate con opportuni decoratori. Questi servono a fornire ad Angular i metadati per associare i *component* ai rispettivi *template* oppure per fornire le informazioni necessarie a fare *injecting* dei *service* nei *component*.

Angular è stato utilizzato all'interno del progetto per realizzare il componente della Dashboard.

Mongoose

Mongoose [53] è una libreria Object Data Modeling (ODM) per Node.js che semplifica la comunicazione con il database No-SQL MongoDB. Offre diverse funzionalità come la gestione delle relazioni tra i dati, la validazione dello schema e la traduzione degli oggetti nel codice nella loro rappresentazione in MongoDB e viceversa. In particolare, ogni oggetto è gestito tramite uno schema e ogni schema è associato ad una *collection* di MongoDB definendone la forma dei documenti ivi contenuti. Ogni chiave nello schema definisce una proprietà del relativo *document*. Nello schema è anche possibile definire i tipi delle proprietà e la libreria si occuperà di effettuare il *cast*. Oltre allo schema, Mongoose mette a disposizione dei costruttori *higher-order* (*Model*) per creare istanze dei *document* e fornisce un'interfaccia per interrogare e aggiornare il database.

In definitiva, Mongoose permette di creare astrazioni dei modelli che rendono semplice l'utilizzo di *collection* e *document* dato che è possibile utilizzarli come normali oggetti all'interno del codice. Nel progetto è stato utilizzato per implementare la comunicazione tra il *Persistor* e MongoDB.

Docker

Docker è uno strumento *software open-source* per la pacchettizzazione di un'applicazione e delle sue dipendenze in un *container* virtuale che può essere eseguito direttamente su un server [82]. In Docker i *container* non sono altro che le immagini in esecuzione sul *Docker Engine*. Un'immagine è un pacchetto eseguibile che include tutto il necessario per l'esecuzione di una certa applicazione: codice, runtime, strumenti di sistema, librerie di sistema, ecc...

A differenza dei classici sistemi di virtualizzazione, i *container* vengono eseguiti direttamente sul sistema operativo (SO). Questo permette di scalare le applicazioni in modo semplice e veloce evitando l'*overhead* di un SO per ogni applicazione (Figura 6.3).

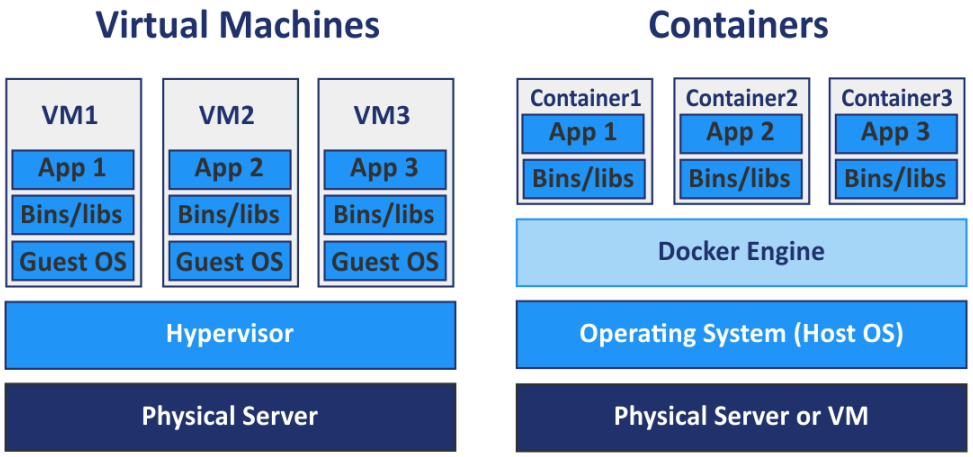


Figura 6.3: Docker Containers vs. Virtual Machines [47].

Inoltre, esistono diversi strumenti, dai più ai meno sofisticati, per l'orchestrazione dei servizi e la gestione di *cluster* di container. Avendo a disposizione una sola macchina, per il progetto si è scelto di utilizzare Docker Compose [57]. Tuttavia sono stati creati i file di configurazione per Kubernetes [46] grazie al tool *kompose convert*. Tale configurazione è stata infine testata su minikube [81]. Ogni servizio all'interno del sistema è stato quindi pacchettizzato come *container* Docker ed eseguito come un'applicazione *multicontainer* grazie a Docker Compose.

Eclipse Thingweb node-wot

Tutti i servizi necessitano di poter interagire con le *WoT Thing*:

- il *Backend* deve poter consumare la TD della TDD, dell'*Aggregator* e degli *edge device*;
- l'*Aggregator* deve poter consumare la TD degli *edge device* e del *Persistor* per il salvataggio e deve esporre le *action* per la configurazione delle *policy* e per il recupero dei dati;

- il *Persistor* deve esporre le *action* per il salvataggio e il recupero dei dati;
- la TDD deve consentire la ricerca sintattica e la registrazione delle TD degli *edge device*;
- gli *edge device* devono esporre le property relative ai sensori.

Per l'implementazione dei *Servient* necessari è stata utilizzata *Eclipse Thingweb node-wot* [26]. Si tratta di una libreria sviluppata dalla *Eclipse Foundation* sulla base dello standatd WoT del W3C. L'implementazione è su Node.js e supporta diversi *protocol binding* come HTTP/HTTPS, CoAP/-CoAPS, MQTT e Websocket (solo server). Permette di essere usata come normale dipendenza di progetti Node.js oppure tramite *client* da riga di comando per l'esecuzione di script. Per il progetto si è optato per l'utilizzo come dipendenza.

6.2 Componenti

Edge Device

Come già illustrato in nella Sezione 5.3.1, l'*edge device* è composto dai servizi *Thing Service*, *Alarm Service* e *State Service*.

Il *Thing Service* è stato sviluppato con la libreria di node-wot in TypeScript, mentre i restanti due *service* sono stati implementati in Python. La scelta di Python è dovuta al grande quantitativo di esempi già esistenti per la comunicazione a basso livello con i sensori. L'utilizzo di TypeScript, invece, è stata una scelta obbligata dovuta al fatto che al momento node-wot è l'unica libreria che implementa un *Servient* secondo lo standard del W3C. L'utilizzo di due linguaggi differenti ha reso necessario un meccanismo di *message passing* tra i diversi servizi. A tal scopo si è fatto uso della libreria ZeroMQ.

Lo *State Service* è stato implementato utilizzando un *thread* per ogni sensore. In particolare è stata implementata una classe *Worker* (Listato 6.1)

che permette eseguire il codice di un *DataProducer* in un ciclo. Il *DataProducer* permette di legare le funzioni specifiche per l'interazione con i sensori al meccanismo di *message passing* per lo scambio dei dati. Di fatto, per essere istanziato necessita del contesto di ZeroMQ e della funzione che verrà utilizzata per la lettura del sensore. Sono state definite due sottoclassi di *DataProducer*, una implementa il pattern *publish-subscribe* (*Publisher*) e l'altra implementa il pattern *request-replay* (*Responder*). Un esempio di utilizzo di *DataProducer* è presente nel listato 6.2.

```
1  class Worker:
2      producer: DataProducer
3      name: str
4
5      def __init__(self, name: str, producer: DataProducer):
6          self.name = name
7          self.producer = producer
8          self.thread: threading.Thread
9
10     def start(self) -> None:
11         def produce_data(producer: DataProducer):
12             while True:
13                 producer.produce()
14
15         self.thread = threading.Thread(target=produce_data, name=self.name,
               args=(self.producer,))
16         self.thread.daemon = True
17         self.thread.start()
18
19     def is_alive(self) -> bool:
20         return self.thread.is_alive()
```

Listing 6.1: Classe Worker dello State Service dell'*edge device* .

```
imei_fun = compose(
    lambda module_info: module_info.imei if module_info else "",
    lambda _: sim_module.get_info().or_else(None))

Worker("IMEI", Responder(context, imei_fun))
```

Listing 6.2: Esempio di utilizzo di un DataProducer Responder. In particolare è stato utilizzato il sensore sim_module che si occupa della comunicazione sull'interfaccia seriale. La funzione in input al Responder è la composizione di due funzioni anonime. Viene letto il sensore e successivamente viene controllato il risultato.

Inoltre, per ogni sensore presente è stata definita una classe *ad hoc* che astrae il protocollo comunicazione e le relative complessità. Un'eccezione è stata fatta per il sensore che si occupa di leggere il GPS. Questo utilizza un modulo SIM7600 che permette anche l'invio degli SMS, ma tutti i comandi viaggiano sulla porta seriale tramite protocollo AT. Per gestire la problematica è stata creata una classe per il SIM7600 che contiene un *lock* per l'accesso concorrente alla seriale. Questa viene infine iniettata nelle due classi che ne fanno uso.

L'*Alarm Service* è stato implementato con un *thread* che legge costantemente i valori prodotti dalla piattaforma inerziale (tramite ZeroMQ) e calcola il modulo dell'accelerazione. Se questo supera un certo valore soglia, allora provvede ad inviare un SMS e a pubblicare un messaggio di allerta.

Il *Servient* del *Thing Service* è stato implementato con node-wot. Inizialmente il protocol binding utilizzato era HTTP, ma a causa di un bug nella libreria di node-wot [1] si è passati al protocollo CoAP.

Per quanto riguarda la connessione ad Internet, questa è stata ottenuta mediante l'utilizzo di driver che permettono di vedere il modulo LTE SIM7600 come una normale scheda di rete. Dato che il *Thing Service* all'avvio ha la necessità di recuperare l'indirizzo della scheda di rete per poter creare i *link/form* della TD, è stato creato un *network namespace* separato tramite *ip-netns* [40] all'interno del quale è presente la sola scheda di rete LTE e la rete locale (per la comunicazione su ZeroMQ).

Infine, tutti i servizi vengono avviati al momento dell'accensione del Raspberry Pi mediante l'utilizzo di *systemd* e una serie di script *bash*.

Thing Description Directory

La TDD è stata implementata con la libreria node-wot. Per l'implementazione della ricerca sintattica è stata utilizzata la libreria JSONPath [8]. Essendo un componente prototipale, si è deciso di non interfacciare la TDD con un database, ma di mantenere tutte le *Thing* registrate in *memory*.

Inoltre, sono state fatte delle modifiche alla gestione degli eventi aggiungendo al *Servient* il *binding* per il protocollo CoAP oltre a quello per HTTP per il *bug* esposto nella sezione precedente. La TDD, dunque, si sottoscrive agli eventi generati dagli *edge device* usando CoAP. Anche i sottoscrittori degli eventi generati dalla TDD (Aggregator) utilizzeranno il binding CoAP. L'utilizzo di tale protocollo ha impedito l'invio della TD delle *Thing* con eventi a causa del superamento della Maximum Transmission Unit (MTU) che è di soli 1024 *byte* per il *payload* [60]. Sarà dunque il sottoscrittore degli eventi *newThing* e *newThingStatus* a dover leggere la property *searchJSONPath* per ottenere la TD desiderata.

Aggregator

Come descritto nella sezione 5.3.1, l'*Aggregator* è il *Servient* che da un lato si interfaccia con il *Persistor* e la TDD (consumandone la TD) e dall'altro espone una *action* per poter definire una *policy* per il salvataggio dei dati per ogni singolo *edge device*.

La definizione di una *policy* è resa possibile dall'implementazione di un interprete per il linguaggio all'interno dall'*Aggregator* stesso. Il linguaggio permette di definire un albero di computazione e, per ogni tipo di nodo di tale albero, è associata un'apposita funzione di valutazione. Nel codice del *Servient* è presente un *handler* (*newThingHandler*) specifico che viene invocato quando la TDD emette un evento. Al *newThingHandler* viene passata in input una configurazione contenente lo stato iniziale, il codice

relativo alla gestione degli eventi e il codice relativo alla gestione dei dati da salvare, questi ultimi espressi nel linguaggio dell'*Aggregator*. Infine, sarà il *newThingHandler* a legare l'interpretazione del codice all'invocazione delle *action* sul *Persistor* tramite una serie di chiusure che verranno utilizzate dalle funzioni di valutazione dei nodi. Il codice del *newThingHandler* è mostrato nel Listato 6.3.

```
1  public async newThingHandler(thing: ConsumedThing, configuration:
       Configuration) {
2
3      let {computeNode, state, eventNode} = configuration
4
5      const setter = async (path, val: Promise<any>) => {
6          const value = await val
7          state = set(state, value, path);
8      }
9      const getter = (x) => {
10         return parseFloat(x) ? x : get(state, x)
11     }
12     const invoker: (name: string, parameter: any) => Promise<any> = async (
           name: string, parameter: any) => {
13         return await this.persistor.invokeAction(name, parameter)
14             .catch(err => {
15                 logger.error(`Unable to invokeAction: ${name} with ${JSON.
                       stringify(parameter)} ERR: ${err}`);
16                 throw err;
17             });
18     }
19     const mapData: (dataMapping: { mapping: Statement[]; out: string }) =>
           Promise<void> =
20         (dataMapping: { mapping: Statement[], out: string }) => {
21             return thing.readAllProperties()
22                 .then(prop => {
23                     let x: [string, WoT.PropertyValueMap, Statement[]];
24                     x = [thing.id, prop, dataMapping.mapping]
25                     return x;
26                 })
27                 .then(prop => setter(dataMapping.out, this.
                       mapPropertiesToGeoJson(prop)))
28         }
29     const loopFunction = (loop: () => void) => {
30         let interval = 10000;
31         const savedInterval = get(state, 'savedThing.update_freq');
32         if (savedInterval) {
```

```
33              interval = savedInterval * 1000;
34          }
35          let timeout = setTimeout(loop, interval)
36          this.saveThingHandlers.set(thing.id, timeout)
37          set(state, timeout, "timer");
38      }
39      const subscriber = (e: ThingEvent) => {
40          set(state, {eventName: e.eventName, thingImei: thing.id}, "
                alarmEventInput");
41          thing.subscribeEvent(e.eventName, async () => {
42              evalNode(e.eventCode, getter, setter, invoker, null, null)
43                  .catch(err => thing.unsubscribeEvent(e.eventName))
44          }).catch(err => {
45              logger.error(`Unable to subscribe: ${err}`);
46          });
47      }
48
49      evalEvent(eventNode, subscriber);
50
51      evalNode(computeNode, getter, setter, invoker, mapData, loopFunction);
52
53  }
```

Listing 6.3: Handler *newThingHandler* che si occupa di chiamare le funzioni di valutazione dei nodi. In particolare viene chiamata la valutazione dei nodi *EventNode* e la valutazione dei nodi *ComputeNode*. Le chiusure getter e setter vengono utilizzate dall'interprete per accedere alle variabili di definite nello stato (sia in lettura che in scrittura). La chiusura mapData legge le properties dagli *edge device* (*readAllProperties*) e converte il risultato in un oggetto GeoJSON secondo quanto specificato nel mapping. La chiusura invoker viene utilizzata per invocare i le *action* sul Persistor. La chiusura *loopFunction* viene utilizzata dall'interprete per accedere all'update frequency della Thing. Infine, la chiusura subscriber viene utilizzata per la sottoscrizione agli eventi.

Tutti gli elementi del linguaggio sono stati definiti come tuple taggate. Ad esempio un *ComputeNode* che si occupa di invocare le *action* sarà costituito dalla tupla "{ type: 'ActionNode', action: Action, node: ComputeNode }". L'interprete è stato realizzato utilizzando uno *switch case* sulla proprietà *type* per emulare il costrutto di *pattern matching* normalmente presente nei linguaggi funzionali.

Persistor

Il *Persistor* è il *Servient* che funge da interfaccia con il database. Anche quest'ultimo è stato implementato con la libreria node-wot oltre che Mongoose per l'interfacciamento con MongoDB. In particolare per la gestione delle entità in fase di salvataggio sono stati definiti metodi di istanza, metodi statici, modello, schema e tipo. Ad esempio, per l'entità *user* è stato definito il metodo *addThing* che preso un *document user* lo modifica aggiungendo l'identificatore di una *thing* e il metodo statico *findOneOrCreate* che prende un *model* e ritorna il *record* se esiste, altrimenti ne crea uno. I metodi di stanza e statici sono stati aggiunti alle entità tramite lo schema come mostrato nel listato 6.4.

```
1  import {Schema} from 'mongoose';
2  import {findOneOrCreate} from "./user.statics";
3  import {addThing} from "./user.methods";
4
5  const UserSchema = new Schema({
6      email: {
7          type: String,
8          unique: true,
9          required: 'Please enter user email',
10     },
11     password: {
12         type: String,
13         required: 'Please enter user password',
14     },
15     things: [{ type: Schema.Types.ObjectId, ref: 'thing' }]
16 })
17
18 UserSchema.statics.findOneOrCreate = findOneOrCreate
19 UserSchema.methods.addThing = addThing
20
21 export default UserSchema;
```

Listing 6.4: Esempio di schema relativo all'*user*. Sono stati aggiunti i metodi *addThing* (metodo di istanza) e *findOneOrCreate* (metodo statico). Inoltre è presente un controllo espresso mediante chiave *required* che permette di richiedere espressamente che il campo sia riempito in fase di salvataggio. In caso negativo, viene ritornato il messaggio di errore specificato.

Backend

Il servizio di *Backend* è stato implementato con NestJS. Per l'interazione con *Aggregator*, TDD ed *edge device*, è stato creato un *service* apposito che si occupa di istanziare un *WoT Servient* con i relativi *protocol binding* (Listato 6.5). Si tratta dell'unico componente con dipendenze diretta da node-wot.

```
@Injectable()
export class ServientService {
  private _helper: Helpers;
  private _wot: WoT;

  constructor() {
    const servient = new Servient();
    servient.addClientFactory(new HttpClientFactory());
    servient.addClientFactory(new CoapClientFactory());
    servient.start().then((WoT) => {
      this._wot = WoT;
      this._helper = new Helpers(servient);
    });
  }

  consume(td: ThingDescription): Observable<ConsumedThing> {
    return from(this._wot.consume(td));
  }

  get wot(): WoT {
    return this._wot;
  }

  get helper(): Helpers {
    return this._helper;
  }
}
```

Listing 6.5: Classe ServientService. All'avvio dell'applicativo viene invocato il costruttore del servizio che si occupa di istanziare il Servient.

Il *ServientService* viene iniettato come dipendenza in altri due servizi:

- *TdiClientService*: si occupa di consumare la TD della TDD. Espone un metodo *getThingById* che legge la proprietà *searchJSONPath* della TDD per recuperare la TD della *Thing* desiderata;

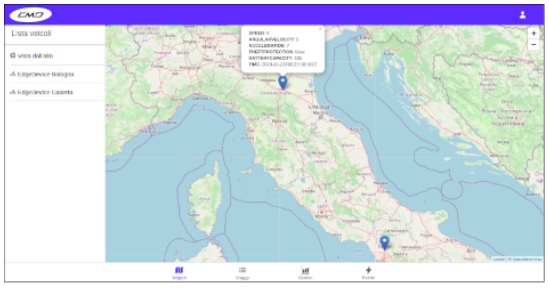
- *EdgeDeviceClientService*: si occupa di consumare la TD degli *edge device* . Espone due metodi, *setPhoneNumberById* e *updateThingById*. Il primo recupera la TD della *Thing* (*getThingById*) e successivamente invoca l'*action SetPhoneNumber* dell'*edge device* in esame. Il secondo, in modo similare, recupera la TD e invoca la *action Update*.

Dashboard

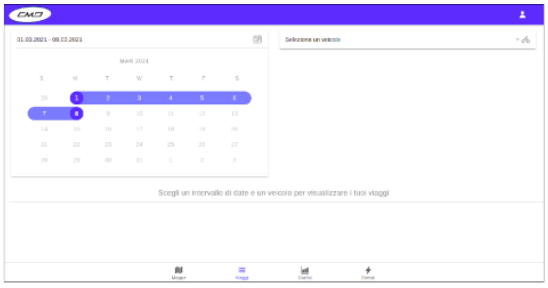
La *Dashboard* è stata sviluppata in Angular 11 e rispecchia le schermate dei *wireframe* esposti nella sezione 5.3.1. Rispetto a questi sono stati fatti piccoli cambiamenti per facilitare la visualizzazione da smartphone. Di fatto quasi tutti i componenti sono stati portati su *Ionic* [39] il che permette di avere una *Web app* compilabile anche per i dispositivi mobili. Non sono state implementate le schermate utente. Una panoramica dell'applicativo con le principali schermate è presente in Figura 6.4.

Skill Alexa

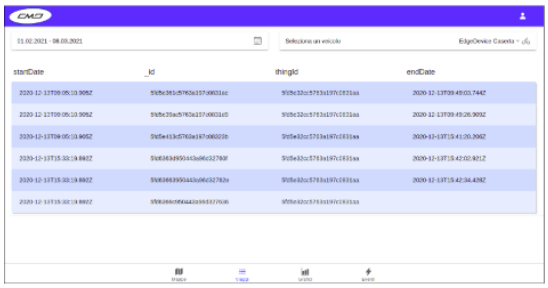
Per implementare la *skill Alexa* ci si è avvalsi della *developer console* messa a disposizione da Amazon. Il nome di invocazione per poter attivare la skill è "cmd ebike". Oltre agli *intent* definiti di default, è stato aggiunto un ulteriore *intent* per richiedere lo stato della batteria passando come paramento (*slot*) il nome del veicolo scelto. Ai fini del progetto, questo è un componente per puro scopo dimostrativo. Non è stata gestita l'autenticazione degli utenti e la *skill* è utilizzabile dalla sola console di sviluppo in quanto non pubblicata.



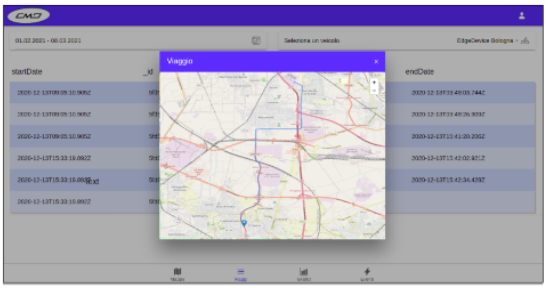
a) Sezione Mappe.



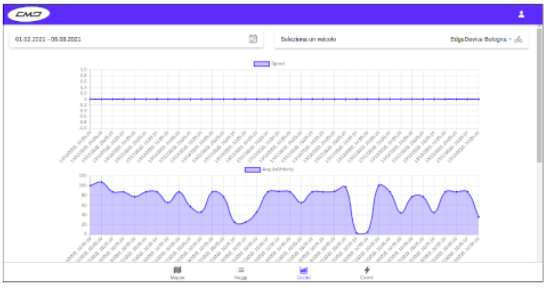
b) Sezione Viaggi: Selezione intervallo temporale.



c) Sezione Viaggi: Lista viaggi.



d) Sezione Viaggi: Particolare viaggio selezionato.



e) Sezione Grafici.

f) Sezione Eventi.

Figura 6.4: Esempio schermate *Dashboard* in Ionic.

Capitolo 7

Validazione

7.1 Servizi

7.1.1 Configurazione

Per la validazione dei servizi si è cercato di replicare uno scenario di utilizzo reale. Questo prevede l'impiego di un numero costante di *edge device*. Inoltre viene simulato un certo numero di utenti sulla *Dashboard*, inizialmente essi sono il 10% dei *device* connessi. Quindi, utilizzando 100 device ci saranno 10 utenti attivi sulla *Dashboard*. Il test così descritto rappresenta sicuramente uno scenario peggiore del caso reale in quanto normalmente i *device* possono essere *offline*.

Per quanto riguarda la configurazione dell'*Aggregator*, è stata utilizzata la stessa *policy* per tutti i *device* e la stessa frequenza di campionamento (30s). La *policy* in questione è descritta brevemente come segue:

- Le *property* vengono lette e mappate su *feature* GeoJSON prendendo tutte le proprietà dell'*edge device* e calcolando il modulo dell'accelerometro e del giroscopio.
- La *feature* ottenuta viene salvata sul database e, se il valore dell'antifurto è passato dall'essere acceso all'essere spento, allora viene creato un nuovo *document journey* e tutte le *feature* salvate da quel momento

in poi verranno referenziate in quel *journey*. Viceversa, il *journey* viene chiuso e tutte le *feature* successive saranno salvate senza comparire in un *journey*.

Gli eventi *alert* generati dall'antifurto sono stati esclusi dai test. Inoltre, data l'impossibilità di poter emulare un numero adeguato di *edge device*, si scelto di estromettere la comunicazione con questi. Di fatto l'*Aggregator* non legge gli *edge device*, ma accede ad una variabile in memoria. Per quanto riguarda la TDD, invece, è stata inibita la sottoscrizione agli eventi e il conseguente controllo sullo stato dei *device*. Infine, è stato creato uno script che invoca la registrazione sulla TDD di un numero configurabile di *device* distribuiti casualmente su un intervallo temporale di 30 minuti.

Per simulare gli utenti sulla *Dashboard* si è optato per l'invocazione sul *Backend* delle stesse chiamate che questa effettua nei principali flussi utente. Il software utilizzato a tal scopo è Gatling [30]. Lo strumento consente mettere sotto stress un sistema mediante *script* in Scala per definire chiamate, frequenza e distribuzione di utenti nel tempo.

La macchina utilizzata per i test è un *virtual private server* (VPS) ospitato sulla piattaforma *OVHcloud* [37]. È dotata di una CPU a 4 core da 2.4 Ghz e 8Gb di memoria RAM.

Infine la metrica utilizzata per la valutazione del sistema consiste nel verificare che il tempo che intercorre tra un salvataggio di un dato e quello successivo sia congruo. Si assume che, se il sistema è in buono stato di efficienza allora la distanza tra i punti tenderà a coincidere con la frequenza di campionamento (30s). L'analisi è stata effettuata con l'utilizzo di un Notebook Jupyter [59] (Python) e la libreria Pandas [58] per la connessione al database, il recupero dei dati e il calcolo della media delle distanze.

7.1.2 Risultati

Il primo test effettuato è uno stress test pensato per capire quale sia il limite di *device* che il sistema può effettivamente gestire senza degradare nelle performance.

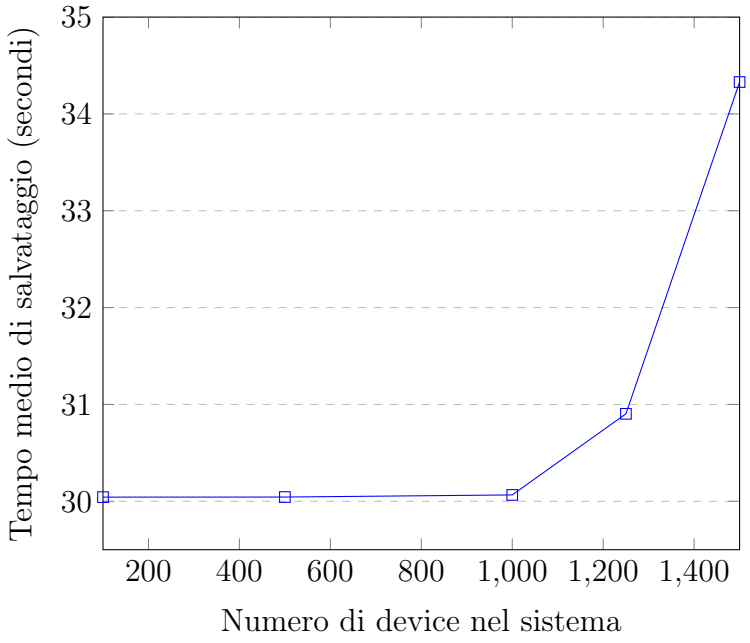


Figura 7.1: Tempi medi di salvataggio delle *feature* GeoJSON. Per ogni punto del grafico sono stati immessi nel sistema il corrispondete numero di *device*. Il test è stato effettuato per la durata di 12h per ogni insieme di *device*. Il risultato è stato calcolato come la media delle distanze tra le feature salvate nel database.

Come si evince dai risultati in Figura 7.1, oltre i 1250 *device* le performance degradano molto velocemente. Si passa da una media di 30.90 secondi (1250 *device*) a una media di 34.33 secondi (1500 *device*). Inoltre si è notato come sia nel caso dei 1250 *device* che in quello dei 1500, il ritardo tenda ad accumularsi con il tempo.

Il grafico in Figura 7.2 mostra l'andamento del fenomeno misurato a *step* di 1 ora con 1500 *device*. Pertanto si può concludere che il massimo numero

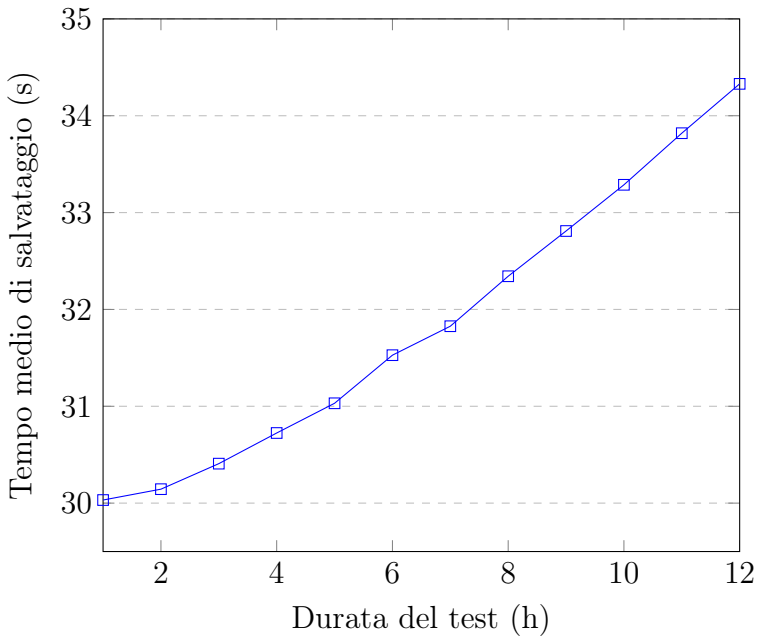


Figura 7.2: Tempi medi di salvataggio delle *feature* GeoJSON all'aumentare del tempo (1500 *device* per 12h).

di *device* che il sistema è in grado di gestire adeguatamente è circa 1200.

Ci si è chiesti, dunque, quale fosse il collo di bottiglia tra tutti i servizi del sistema. Dall'analisi effettuata sull'utilizzo della CPU, il *Persistor* risulta essere il servizio che permane su un utilizzo quasi costante del 100% della CPU su un solo *core*. Pertanto, è stato effettuato un test con Gatling per capire quali fossero i suoi limiti. È stato creato uno script di configurazione in Scala replicando le stesse chiamate che l'*Aggregator* effettua per ogni *Thing*. In particolare, sono state utilizzate la *saveFeature*, *saveJourney*, *closeJourney*, *updateThingLastFeature* e *updateThingLastJourney*.

Come mostrato dai risultati in Figura 7.3, il *Persistor* non riesce a gestire un volume di richieste superiore alle 250 al secondo. Nonostante il caso di test rappresenti comunque un *lower bound* alle performance in quanto normalmente il numero di chiamate effettuate per ogni *device* va dalle 2 alle 4 contro le 5 costanti del test, si è deciso di sperimentare il comportamento all'aumentare dei nodi *Persistor*.

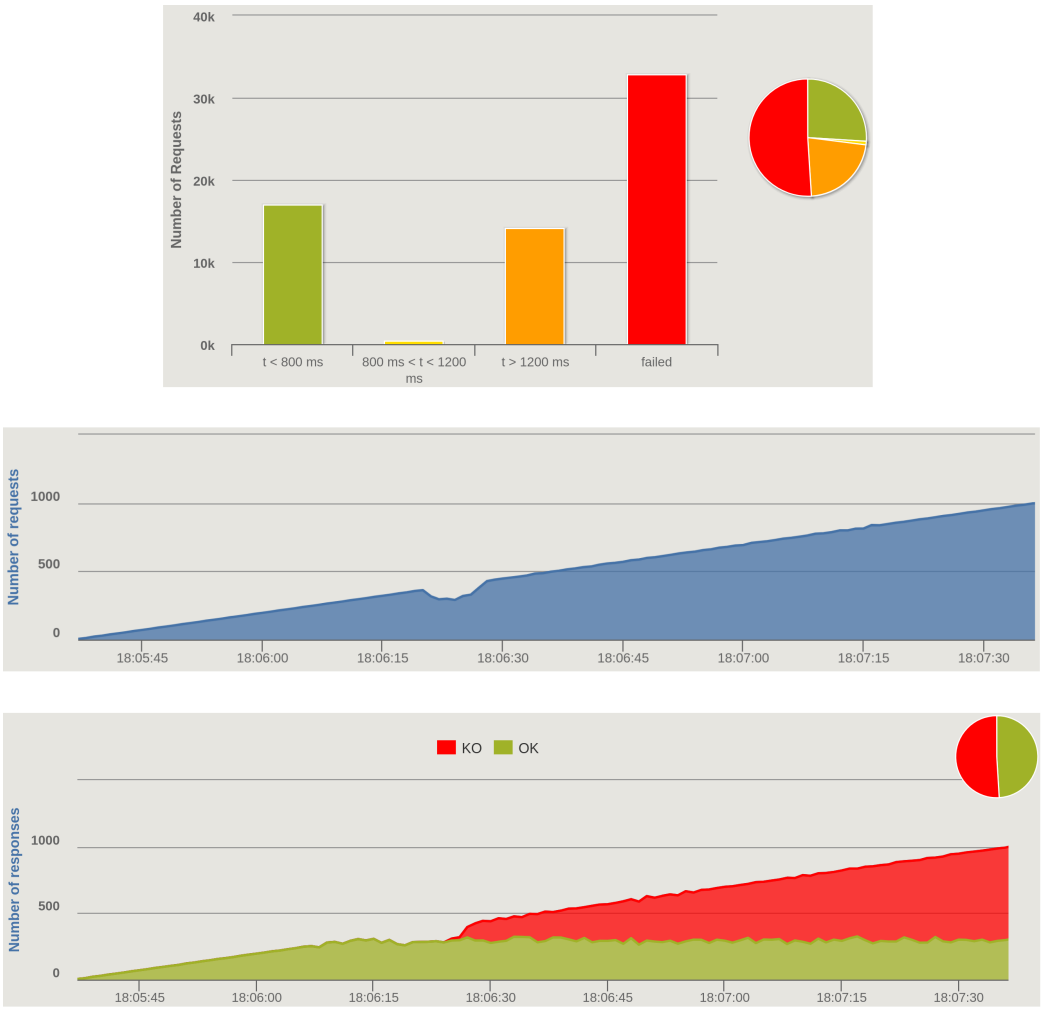


Figura 7.3: Risultati del test sul *Persistor*. Il test è stato effettuato per la durata di due minuti inviando un numero crescente di richieste al secondo. Il primo grafico mostra i tempi di risposta delle richieste. Il secondo mostra il numero di richieste effettuate nel tempo. L'ultimo grafico mostra quante richieste sono state soddisfatte, ossia il numero di risposte andate a buon fine.

Si è dunque introdotto nel sistema un *load balancer* realizzato con Nginx [55] e opportunamente convertito in container Docker. La politica di gestione è di tipo *round-robin*. Sono stati effettuati altri tre test con 2, 3 e 4 quattro nodi. I risultati sono presentati rispettivamente nelle Figure 7.4, 7.5 e 7.6.

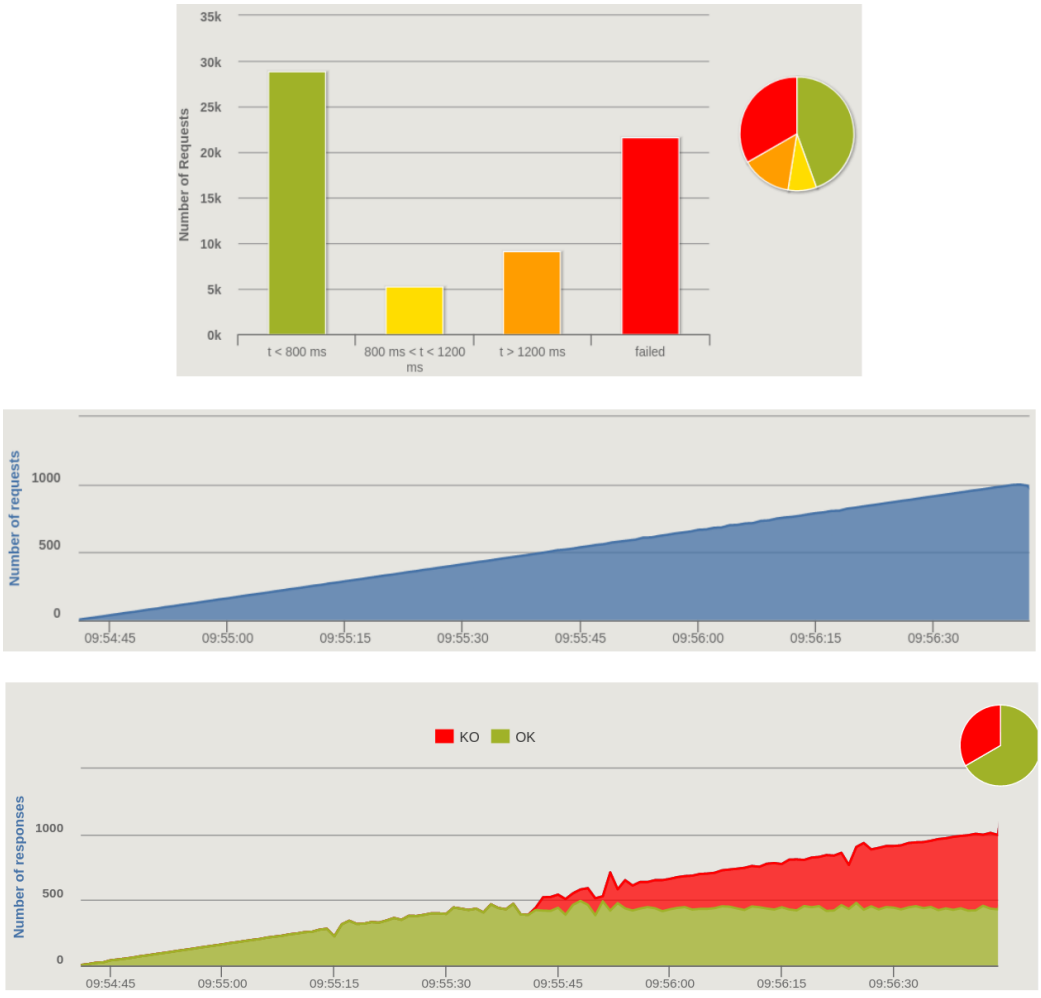


Figura 7.4: Risultati del test sul *Persistor* utilizzando 2 nodi.

Figura 7.5: Risultati del test sul *Persistor* utilizzando 3 nodi.

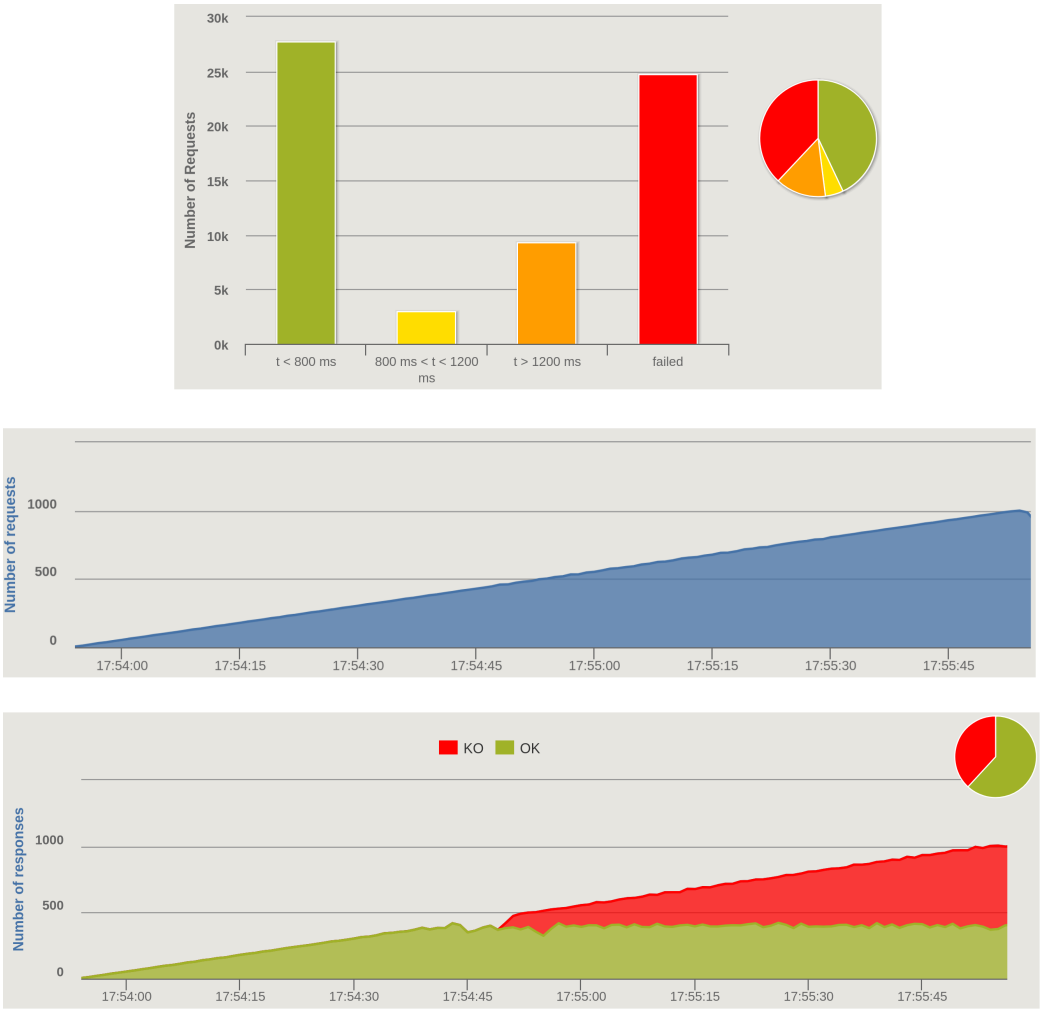


Figura 7.6: Risultati del test sul *Persistor* utilizzando 4 nodi.

È possibile notare come il miglior risultato sull'hardware disponibile sia stato ottenuto utilizzando due nodi *Persistor*. Sono dunque stati ripetuti i test relativi al salvataggio dei dati utilizzando la configurazione con due *Persistor*. Dai risultati esposti in Figura 7.7 si evince come il sistema resti stabile fino a 2000 *device*, con un leggero peggioramento intorno ai 2500 e un peggioramento netto a 3000. In un contesto d'uso reale, dove i *device* vanno normalmente offline anche per diverse ore al giorno (ad esempio per la ricarica della batteria) è possibile assumere che il sistema possa sostenere 2500 dispositivi.

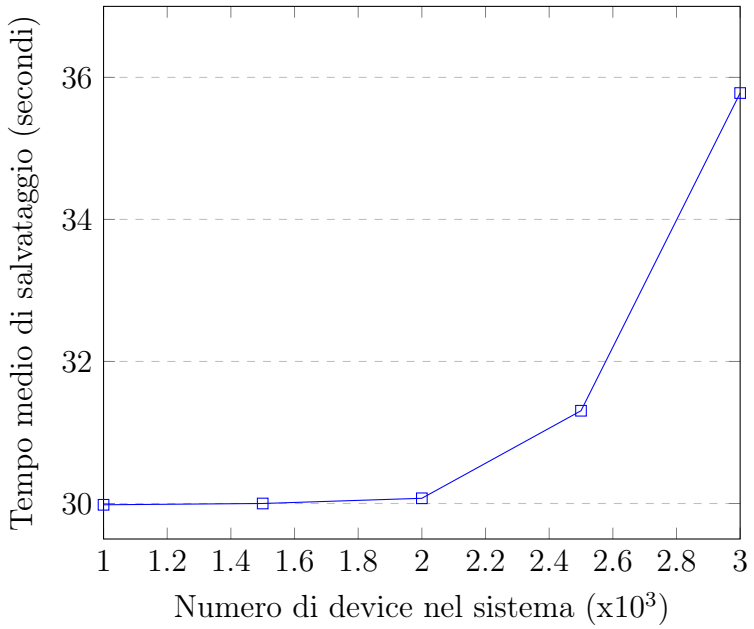


Figura 7.7: Tempi medi di salvataggio delle *feature* GeoJSON nel sistema con 2 nodi *Persistor*. Per ogni punto del grafico sono stati immessi nel sistema il corrispondete numero di *device*. Il test è stato effettuato per la durata di 12h per ogni insieme di *device*. Il risultato è stato calcolato come la media delle distanze in termini di tempo tra le *features* salvate nel database.

Infine, partendo dai risultati precedenti, è stato inserito l'elemento degli utenti. Dato che il sistema si è dimostrato stabile con 200 utenti (il 10% di 2000), si è deciso di aumentarne il numero sino a 1000 (50%). Dai risultati

mostrat in Figura 7.8 è possibile concludere che il sistema è in grado di sostenere 2000 *device* e 1000 utenti costantemente senza che le prestazioni degradino.

Figura 7.8: Risultati del test sul *Persistor*. Il test è stato effettuato per la durata di 190 minuti, di cui i primi 10 sono stati utilizzati per aumentare gradualmente il numero di utenti sino ad arrivare a 1000. Il primo grafico mostra i tempi di risposta delle richieste. Il secondo mostra il numero di richieste effettuate nel tempo. L'ultimo grafico mostra quante richieste sono state soddisfatte, ossia il numero di risposte ritornate.

7.2 Consumi Edge Device

7.2.1 Configurazione

In questa sezione vengono esposti brevemente tutti i componenti *hardware* utilizzati per lo sviluppo dell'*edge device*.

- **Raspberry Pi**: si tratta di un *single-board computer* sviluppato dalla Raspberry Pi Foundation a partire dal 2012. La peculiarità del *device* è l'essere stato sviluppato per permettere la creazione di prototipi. Di fatto presenta un processore su architettura ARM e 40 pin in configurazione 2x20 per General Purpose Input/Output (anche noto come GPIO), Serial Peripheral Interface (SPI), integrated circuit (IC), Universal Asynchronous Receiver-Transmitter (UART) e alimentazione a 3,3 e 5 Volt. Inoltre vanta un discreto numero di distribuzioni su base Linux pronte all'uso. Vengono prodotti diversi modelli che si differenziano per potenza del processore, dimensione della RAM e opzioni di connettività. In particolare sono stati utilizzati un Raspberry Pi 3 Model B [15] e un Raspberry Pi Zero W [16] per l'esecuzione dell'applicativo dell'*edge device*.

- **MCP3008** [50]: si tratta di un convertitore analogico-digitale con a disposizione otto canali. Il tutto è montato su uno *shield* inserito sul GPIO del Raspberry Pi 3 con il quale è possibile comunicare tramite Serial Peripheral Interface (SPI). Viene utilizzato per leggere la tensione della batteria.

- **ADS1115** [2]: si tratta di un convertitore analogico-digitale con a disposizione quattro canali. Comunica su bus *Inter-Integrated Circuit* (I^2C). Saldato direttamente al Raspberry Pi Zero è utilizzato per leggere la tensione della batteria.

- **Waveshare SIM7600E-H 4G HAT** [48]/**Waveshare SIM7000E NB-IoT HAT** [66]: sono moduli utilizzati per abilitare il Raspberry

Pi all'invio/ricezione di SMS, alla navigazione internet e infine alla ricezione del posizionamento GPS. In particolare la connessione è gestita tramite interfaccia USB mediante l'utilizzo di appositi driver [28]. La lettura del GPS, invece, è gestita tramite la porta seriale del Raspberry.

- **MPU6050** [54]: Il sensore MPU-6050 prodotto da InvenSense, contiene in un singolo integrato, un accelerometro MEMS a 3 assi ed un giroscopio MEMS a 3 assi. Il sensore comunica tramite bus *Inter-Integrated Circuit* (I^2C).

7.2.2 Risultati

Per poter misurare il consumo energetico è stato utilizzato un tester USB marchiato Muker [68]. L'applicazione è stata installata su un Raspberry Pi 3 sul quale sono stati disabilitati il modulo WiFi e Bluetooth. È stata esclusa la configurazione con Raspberry Pi Zero in quanto non ancora disponibile al momento dei test.

Si è deciso di misurare i consumi all'aumentare della frequenza di campionamento. La frequenza di generazione dell'evento *Alive* è stata fissata a 60 secondi ed è mantenuta costante per tutti i test. Per semplicità è stato escluso l'evento *Alarm* in quanto essendo raro, non avrebbe comunque impattato sui risultati.

Tutti i test sono stati effettuati per un minimo di 3 ore sino ad un massimo di 8 ore e ripetuti cinque volte per ogni frequenza di campionamento per cercare di ridurre i fattori di consumo legati all'intensità del segnale GPS e LTE che la board riceve. Il grafico in Figura 7.9 mostra i risultati ottenuti per le frequenze 1s, 15s, 30s e 60s. È possibile notare come per frequenze superiori ai 30s non vi è sostanziale differenza nei consumi che si attestano poco sotto i 2 W/h.

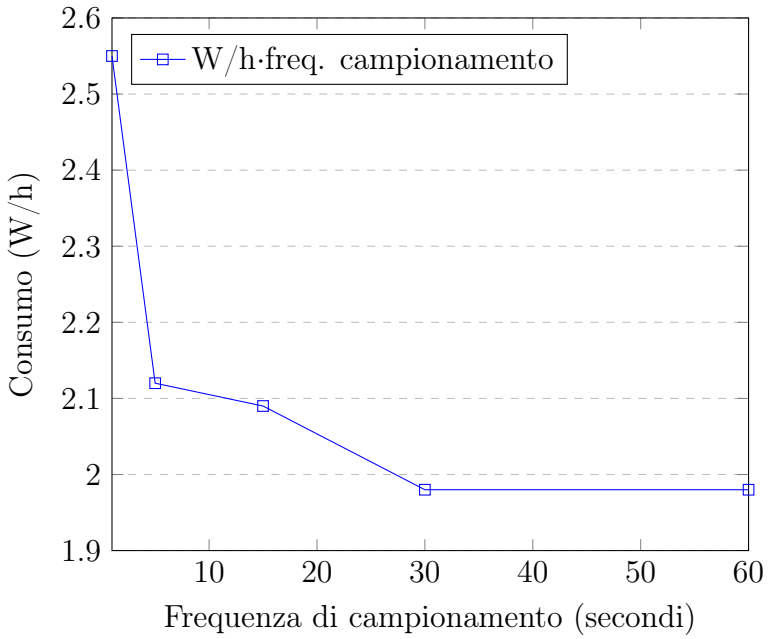


Figura 7.9: Consumi espressi in W/h dell'*edge device* al variare della frequenza di campionamento. Le frequenze prese in considerazione sono 1s, 15s, 30s e 60s.

Capitolo 8

Conclusioni e sviluppi futuri

In questo elaborato è stata presentata la realizzazione di un'applicazione per la raccolta e la gestione dei dati prodotti da veicoli *ebike* secondo l'architettura WoT proposta dal W3C.

Prima di partire con la progettazione è stato necessario un periodo di studio preliminare per approfondire le criticità del mondo IoT in relazione ai sistemi di tracciamento remoto. Successivamente è stato redatto un primo documento con i requisiti condivisi e approvati dall'azienda CMD. Si è poi passati all'elaborazione delle prime architetture. Queste sono risultate essere poco flessibili e incapaci di affrontare problemi complessi come quelli della *findability* delle *Thing*. Dunque è stato svolto un ulteriore lavoro di studio delle tecnologie WoT per poi adottarle per la progettazione dell'architettura finale.

Lo sviluppo è avvenuto in due fasi: nella prima fase sono state fornite da CMD le componenti *hardware* per l'implementazione dell'*edge device*; nella seconda fase si è passati all'implementazione di versioni minimali dei diversi servizi che compongono la parte *cloud* dell'architettura.

Il lavoro si è rivelato essere molto costoso in termini di tempo per via della complessità dell'architettura e delle problematiche riscontrate durante lo sviluppo. Di fatto, al momento, tutti i componenti sono da ritenersi prototipi e non prodotti finiti. Non è stata gestita la sicurezza, non è stata implementata

la parte relativa alla gestione dei dispositivi e dell'utente nella *Dashboard*, il linguaggio utilizzato per la configurazione delle *policy* nell'*Aggregator* necessità di uno strumento per la generazione in quanto estremamente verboso e infine resta da definire dove debba essere salvata la *policy*.

Partendo proprio dalle parti mancanti, è possibile iniziare a stilare requisiti per eventuali sviluppi futuri. Il sistema pensato per le *query* sul database è estremamente versatile e potrebbe essere usato per aggiungere diversi livelli di personalizzazione alla *Dashboard*, come ad esempio la possibilità di aggregare i dati per intervalli temporali. Si potrebbe migliorare le caratteristiche di scalabilità del sistema integrando un ulteriore *load balancer* per la TDD che possa gestire ulteriori *Aggregator*. Come già accennato, si potrebbe costruire un *tool* per la generazione del codice per il linguaggio dell'*Aggregator*, anche utilizzando strumenti grafici come NodeRED. Si potrebbe espandere il linguaggio dell'*Aggregator* per renderlo del tutto indipendente dal *Persistor*. Questo permetterebbe di avere un sistema in cui i vari *Persistor* si registrano alla TDD e l'utente tramite *query* sintattiche seleziona le *action* da inserire nel codice interpretato dall'*Aggregator*. Inoltre, si potrebbe utilizzare il *bus* CAN (Controller Area Network) per permettere all'*edge device* di esporre i parametri di funzionamento del motore. Di fatto, l'azienda prevede l'integrazione del sistema su motori marini ibridi, utilizzati su imbarcazioni commerciali (taxi e barche di raccolta rifiuti a Venezia). Infine, non deve essere trascurata la manutenzione per tenere il sistema allineato con gli aggiornamenti dello standard W3C WoT.

In conclusione, si ritiene che il lavoro proposto in questo elaborato possa essere utilizzato come punto di partenza per lo sviluppo di un ecosistema per la gestione di flotte di *ebike* o di altri tipi di veicoli. Il sistema, infatti, presenta un'architettura che mostra diversi punti di forza: permette una minuziosa personalizzazione dei parametri da monitorare del singolo veicolo; permette di decidere sotto quali condizioni e come salvare i dati; permette di gestire anche diverse tipologie di veicoli. Inoltre, attraverso i test effettuati è stato dimostrato che il sistema risulti essere scalabile all'aumentare dei

dispositivi rendendolo così utilizzabile in contesti reali. Infine, si ritiene che il lavoro svolto nella suddivisione e pacchettizzazione delle diverse componenti possa permettere uno sviluppo semplice e modulare al fine di ottenere un prodotto finito.

Bibliografia

Libri

[33] Dominique Guinard. *A Web of Things Application Architecture - Integrating the Real-World into the Web.*

[34] Dominique D. Guinard e Vlad M. Trifa. *Building the Web of Things.* 2016.

[43] Marilyn Johnson e Geoff Rose. *Safety implications of e-bikes.* 15/02. 2015.

Risorse Online

[1] *[http/core] ERR_STREAM_WRITE_AFTER_END error after the first event · Issue #323 · eclipse/thingweb.node-wot.* `https://github.com/eclipse/thingweb.node-wot/issues/323`. (Accessed on 02/28/2021).

[2] *ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier ID: 1085 - $14.95 : Adafruit Industries, Unique & fun DIY electronics and kits.* `https://www.adafruit.com/product/1085`. (Accessed on 03/01/2021).

[4] *Amazon Alexa Official Site: What is Alexa?* `https://developer.amazon.com/it-IT/alexa`. (Accessed on 03/01/2021).

[5] *Amazon Alexa Voice Assistant | Alexa Developer Official Site*. https://developer.amazon.com/en-US/alexa. (Accessed on 03/02/2021).

[6] *Angular - Introduction to the Angular Docs*. https://angular.io/docs. (Accessed on 02/28/2021).

[7] *AngularJS: Angular, version 2: proprioception-reinforcement*. https://blog.angularjs.org/2016/09/angular2-final.html. (Accessed on 02/28/2021).

[8] *astronautlabs/jsonpath: Query and manipulate JavaScript objects with JSONPath expressions. Robust JSONPath engine for Node.js.* https://github.com/astronautlabs/jsonpath\#readme. (Accessed on 02/28/2021).

[11] *Balsamiq. Rapid, Effective and Fun Wireframing Software | Balsamiq*. https://balsamiq.com/. (Accessed on 03/01/2021).

[12] Bosch. https://www.bosch-ebike.com/en/products/drive-unit/.

[13] Bosch Kiox. https://www.bosch-ebike.com/it/prodotti/kiox/.

[14] Bosch SmartphoneHub. https://www.bosch-ebike.com/us/products/smartphonehub/.

[15] *Buy a Raspberry Pi 3 Model B – Raspberry Pi*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. (Accessed on 02/20/2021).

[16] *Buy a Raspberry Pi Zero W – Raspberry Pi*. https://www.raspberrypi.org/products/raspberry-pi-zero-w/. (Accessed on 02/20/2021).

[18] *CMD Engine - a Loncin Company - Costruzioni Motori Diesel - Progettazione e realizzazione di motori e parti meccaniche per l'industria automobilistica, la nautica, l'aviazione, sistemi di microgenerazione - Produzione motori marini diesel, elettrici e ibridi, general aviation, aerei ultraleggeri, pirogassificatore CHP - FNM, Avio, ECO20X - Basilicata, Atella, Campania, Caserta*. http://www.cmdengine.com/?lang=en. (Accessed on 03/01/2021).

[19] *Controllers | NestJS - A progressive Node.js framework.* https://docs.nestjs.com/controllers. (Accessed on 03/06/2021).

[20] Display C. https://global.yamaha-motor.com/business/e-bike-systems/it/products/display-c/.

[21] *Documentation | NestJS - A progressive Node.js framework.* https://docs.nestjs.com/. (Accessed on 02/28/2021).

[22] *e-Bike Management System for e-Mobility | SITAEL S.p.A.* https://www.sitael.com/internet-of-things/emobility/e-bike-management-system/. (Accessed on 08/21/2020).

[23] E-TUBE PROJECT. https://shimano-steps.com/e-bikes/italy/it/product-information/city-trekking.

[24] E-TUBE RIDE. https://shimano-steps.com/e-bikes/italy/it/product-information/city-trekking.

[25] eBike Connect. https://play.google.com/store/apps/details?id=com.bosch.ebike&hl=en.

[26] *eclipse/thingweb.node-wot: thingweb.node-wot.* https://github.com/eclipse/thingweb.node-wot. (Accessed on 02/28/2021).

[27] *ECMAScript 2021 Language Specification.* https://tc39.es/ecma262/. (Accessed on 02/06/2021).

[28] *File:SIM7X00-Driver.7z - Waveshare Wiki.* https://www.waveshare.com/wiki/File:SIM7X00-Driver.7z. (Accessed on 02/25/2021).

[29] Frotcom. https://www.frotcom.com/it-IT/software-gestione-flotte-aziendali-GPS.

[30] *Gatling Open-Source Load Testing – For DevOps and CI/CD.* https://gatling.io/. (Accessed on 02/25/2021).

[31] Gocycle Website. https://gocycle.com/it/.

[32] GocycleConnect. https://play.google.com/store/apps/details?id=com.gocycle.Gocycle&hl=en.

[36] *Home - Targa Telematics*. https://www.targatelematics.com/en/. (Accessed on 08/21/2020).

[37] *Hosting Internet, Cloud e server dedicati | OVHcloud*. https://www.ovh.it/?xtor=SEC-13-GOO-[IT-2019_S_Hosting_Offensive(Dynamic)]-[380860853739]-. (Accessed on 02/25/2021).

[38] *Internet of Things (IoT) Market | Size | Scope | Trends | Forecast*. https://www.verifiedmarketresearch.com/product/global-internet-of-things-iot-market-size-and-forecast-to-2026/. (Accessed on 01/02/2021).

[39] *Ionic - Cross-Platform Mobile App Development*. https://ionicframework.com/. (Accessed on 03/02/2021).

[40] *ip-netns(8) - Linux manual page*. https://man7.org/linux/man-pages/man8/ip-netns.8.html. (Accessed on 02/28/2021).

[41] *Italmoto – Moto, Scooter and E-bike Made in Italy*. https://www.italmoto.com/en/. (Accessed on 03/01/2021).

[46] *Kubernetes*. https://kubernetes.io/. (Accessed on 02/28/2021).

[47] *Kubernetes vs Docker*. https://www.nakivo.com/blog/docker-vs-kubernetes/. (Accessed on 03/06/2021).

[48] *LTE CAT4/4G/LTE-TDD/LTE-FDD/UMTS/HSPA+/GSM/GPRS/EDGE HAT for Raspberry Pi, SIM7600E-H*. https://www.waveshare.com/sim7600e-h-4g-hat.htm. (Accessed on 02/20/2021).

[50] *MCP3008 - Analog to Digital Converters*. https://www.microchip.com/wwwproducts/en/MCP3008. (Accessed on 02/25/2021).

[51] Mission Control App. https://specialized.picturepark.com/Go/jF5UJso9/V/59535/1.

[52] *Modules | NestJS - A progressive Node.js framework*. https://docs.nestjs.com/modules. (Accessed on 03/06/2021).

[53] *Mongoose ODM v5.11.18*. https://mongoosejs.com/. (Accessed on 02/28/2021).

[54] *MPU-6050 | TDK*. https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/. (Accessed on 02/25/2021).

[55] *NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy*. https://www.nginx.com/. (Accessed on 03/01/2021).

[56] *Number of IoT devices 2015-2025 | Statista*. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. (Accessed on 01/02/2021).

[57] *Overview of Docker Compose | Docker Documentation*. https://docs.docker.com/compose/. (Accessed on 02/28/2021).

[58] *pandas documentation — pandas 1.2.2 documentation*. https://pandas.pydata.org/pandas-docs/stable/index.html. (Accessed on 02/25/2021).

[59] *Project Jupyter | Home*. https://jupyter.org/. (Accessed on 02/25/2021).

[60] *RFC 7252 - The Constrained Application Protocol (CoAP)*. https://tools.ietf.org/html/rfc7252. (Accessed on 02/08/2021).

[61] *RFC 7946 - The GeoJSON Format*. https://tools.ietf.org/html/rfc7946. (Accessed on 03/01/2021).

[63] *Semantic Sensor Network Ontology*. https://www.w3.org/TR/vocab-ssn/. (Accessed on 02/23/2021).

[64] Shimano Steps line. https://shimano-steps.com/e-bikes/italy/it/product-information/city-trekking.

[65] Shimano Website. https://shimano-steps.com/e-bikes/europe/en.

[66] *SIM7000E NB-IoT HAT - Waveshare Wiki*. https://www.waveshare.com/wiki/SIM7000E_NB-IoT_HAT. (Accessed on 03/08/2021).

[67] Specialized Bicycle Components. https://www.specialized.com/.

[68] *Test of USB safety tester J7-t*. https://lygte-info.dk/review/USBmeter\%20safety\%20tester\%20J7-t\%20UK.html. (Accessed on 02/25/2021).

[69] *Tiquattro EB - La Bici Elettrica Fat di Italmoto*. https://www.italmoto.com/en/models/tiquattro-eb/. (Accessed on 03/01/2021).

[70] VanMoof Website. https://www.vanmoof.com/shop/en_it/.

[71] *VSSo*. http://automotive.eurecom.fr/vsso. (Accessed on 02/22/2021).

[72] *Web of Things (WoT) Architecture: common patterns*. https://www.w3.org/TR/wot-architecture/\#sec-common-usecase-patterns. (Accessed on 03/05/2021).

[73] *Web of Things (WoT) Architecture: requirements*. https://www.w3.org/TR/wot-architecture/\#sec-requirements. (Accessed on 03/05/2021).

[74] *Web of Things (WoT) Architecture: use cases*. https://www.w3.org/TR/wot-architecture/\#sec-use-cases. (Accessed on 01/06/2021).

[75] *Web of Things (WoT) Architecture: use cases*. https://www.w3.org/TR/wot-architecture//\#sec-wot-architecture. (Accessed on 01/06/2021).

[76] *Web of Things (WoT) Architecture: use cases*. https://www.w3.org/TR/wot-architecture//\#sec-building-blocks. (Accessed on 01/06/2021).

[77] *Web of Things (WoT) Discovery*. https://www.w3.org/TR/wot-discovery/\#exploration-directory-api. (Accessed on 02/21/2021).

[78] *Web of Things Working Group Charter*. https://www.w3.org/2016/12/wot-wg-2016.html. (Accessed on 01/02/2021).

[81] *Welcome! | minikube*. https://minikube.sigs.k8s.io/docs/. (Accessed on 02/28/2021).

[82] *What is a Container? | App Containerization | Docker*. `https://www.docker.com/resources/what-container`. (Accessed on 02/28/2021).

[84] *ZeroMQ*. `https://zeromq.org/`. (Accessed on 02/27/2021).

Articoli

[3] Zainab H Ali, Hesham A Ali e Mahmoud M Badawy. «Internet of Things (IoT): definitions, challenges and recent research directions». In: *International Journal of Computer Applications* 128.1 (2015), pp. 37–47.

[9] Luigi Atzori, Antonio Iera e Giacomo Morabito. «The internet of things: A survey». In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[10] Luigi Atzori, Antonio Lera e Giacomo Morabito. «The Internet of Things: a survey». In: *Tp chí Nghiên cu dân tc* (dic. 2018).

[17] Kriti Chopra, Kunal Gupta e Annu Lambora. «Future internet: the Internet of Things-a literature review». In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)* (2019), pp. 135–139.

[35] Dominique Guinard e Vlad Trifa. «Towards the web of things: Web mashups for embedded devices». In: *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain* (2009).

[42] Shuguang Ji et al. «Electric vehicles in China: emissions and health impacts». In: *Environmental science & technology* 46.4 (2012), pp. 2018–2024.

[44] Kanchan Kamble. «Smart Vehicle Tracking System». In: *International Journal of Distributed and Parallel systems* 3 (lug. 2012), pp. 91–98.

[45] B. Klotz et al. «A Car as a Semantic Web Thing: Motivation and Demonstration». In: *2018 Global Internet of Things Summit (GIoTS)* (2018), pp. 1–6.

[49] Somayya Madakam, R Ramaswamy e Siddharth Tripathi. «Internet of Things (IoT): A Literature Review». In: *Journal of Computer and Communications* 3 (apr. 2015), pp. 164–173.

[62] Jan-Philipp Sander e Stefanie Marker. «Comparison of Rides on an Electric and a Conventional Bicycle in a Naturalistic Cycling Study». In: *International Cycling Safety Conference 2016* ().

[79] T Weber, G Scaramuzza e K-U Schmitt. «Evaluation of e-bike accidents in Switzerland». In: *Accident Analysis & Prevention* 73 (2014), pp. 47–52.

[80] Jonathan Weinert et al. «The future of electric two-wheelers and electric vehicles in China». In: *Energy Policy* 36.7 (2008), pp. 2544–2555.

[83] Deze Zeng, Song Guo e Zixue Cheng. «The web of things: A survey». In: *JCM* 6.6 (2011), pp. 424–438.

Ringraziamenti

Ringrazio il mio relatore Marco Di Felice, il correlatore Luca Sciullo e l'azienda CMD tutta. Non di meno tutte le persone che mi hanno accompagnato nel percorso, dalla famiglia ai colleghi. In particolare Filippo, Gabriele, Michele, Patrizia e Laura.