Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Laurea Magistrale in Informatica

TITOLO DELLA TESI

Relatore:
Chiar.mo Prof.
Marco Di Felice

Presentata da:
Daniele Rossi

Sessione III
Anno Accademico 2018/2019

Questa è la Dedica*:*
ognuno può scrivere quello che vuole,
anche nulla . . .

Indice

Elenco delle figure

Elenco dei frammenti di codice

Introduzione

Nonostante la sua tutt'altro che recente nascita, solo negli ultimi anni il settore dell'IoT ha assistito ad una importante espansione che, molto probabilmente, sarà destinata a durare per i prossimi anni avvenire. Se nel 2015, il numero di dispositivi connessi era di 15.41 miliardi, attualmente se ne contano circa 30, con una proiezione di 75.44 miliardi entro il 2025[1].

Tra le principali critiche mosse al settore dell'Internet of Thing (IoT) possiamo sicuramente includere la difficile interoperabilità tra i numerosissimi prodotti esistenti che possiedono protocolli e formati differenti e che spesso non sono in grado di dialogare tra di loro.

Al giorno d'oggi, il campo dell'IoT è riuscito a compiere passi molto importanti per gestire al meglio il limite dell'interoperabilità, ovvero è stata introdotta la tecnologia 5G per fornire connessioni che siano sempre più affidabili e sicure. I fattori in comune che hanno portato all'avvicinamento di questi due campi sono sicuramente la necessità di nuove applicazioni per il futuro dell'IoT e la costante evoluzione del settore della tecnologia 5G[2].

Grazie all'applicazione e allo sviluppo del paradigma del Web of Things (WoT), si è cercato di gestire questo importante limite. Nello specifico, è stato proprio grazie all'utilizzo di standard e tecnologie web già ampiamente consolidate ed adatte alla costruzione di applicazioni fortemente scalabili, che è stato possibile affrontare il problema della difficile interoperabilità tra i prodotti esistenti all'interno del mondo dell' IoT. La standardizzazione promossa fin dal 2015 dal World Wide Consortium (W3C) ha cercato di ridurre la vasta frammentazione presente all'interno dell'IoT concentrandosi su componenti complementari che permettono di far inte-

grare tra di loro differenti piattaforme e domini applicativi. Al sapiente lavoro di standardizzazione partecipano non solo i membri del W3C ma diversi collaboratori provenienti da importanti aziende tecnologiche a livello mondiale.

Lo scopo delle seguente tesi è la progettazione e l'implementazione dello stack software WoT Servient per sistemi embedded, seguendo con precisione l'architettura proposta del W3C. La piattaforma si compone di un'interfaccia a linea di comando attraverso la quale l'utente è in grado di definire Thing Descriptions, le quali verranno usate come basi per la creazione di script eseguibili dai sistemi di questo tipo. I microcontrollori, in questo modo, assumeranno il ruolo di Thing ed esporranno le loro funzionalità agli utenti con cui interagiranno.

In questo elaborato sono esposte le varie tappe di studio e di lavoro svolto per la realizzazione di questo stack software.

Nella prima parte (capitoli 1 e 2) viene esposto lo stato dell'arte. Il capitolo 1 contiene una panoramica sull'IoT con cenni storici, la sua evoluzione, le sue caratteristiche e le sue criticità. Nel capitolo 2 si analizzerà il W3C WoT, illustrandone l'utilità e l'architettura del W3C WoT, mostrando i suoi casi d'uso, gli elementi principali e l'implementazione del Servient.

La seconda parte dell'elaborato (capitoli 3, 4 e 5) si focalizzerà sul sistema sviluppato. Nel capitolo 3 verrà presentata l'idea che ha dato origine a questo progetto, i requisiti funzionali e tecnici e la sua architettura. Il capitolo 4 riguarderà la presentazione del dettaglio delle tecnologie e delle librerie adottate ed i moduli software realizzati. Infine nel capitolo 5 si presenterà un caso d'uso su cui si è validata l'implementazione del sistema oggetto di questo lavoro.

Parte 1

Stato dell'arte

Capitolo 1

Internet of Things

Con il termine Internet of Things (IoT) si indica un paradigma in cui la rete Internet viene integrata con oggetti fisici. Nell'IoT, la rete Internet non è più una semplice collezione di contenuti multimediali, ma viene estesa al mondo fisico, real-time, trasformandosi in una rete di oggetti di ogni genere e dimensione: auto, smartphone, videocamere, applicazioni per la casa, giocattoli, strumenti medici, sistemi industriali, animali, persone, edifici ecc. Tutti questi oggetti sono connessi tra loro e comunicano scambiandosi informazioni. Attraverso l'integrazione con la rete Internet, essi vengono "potenziati", acquisiscono intelligenza e nuove funzionalità così da essere in grado di svolgere nuovi compiti. Gli oggetti appartenenti a questa nuova classe prendono il nome di *smart things* o semplicemente *Things*.

Una *smart thing*, quindi, è un oggetto fisico che viene potenziato digitalmente tramite una o più delle seguenti caratteristiche:

- sensori (temperatura, luce, movimento, ecc.)
- attuatori (schermi, motori, ecc.)
- computazione (capacità di eseguire programmi e logica)
- interfacce di comunicazione (cablate o wireless)

Unendo le parole e i concetti di "Internet" e "Things", si ottiene la nuova tipologia di rete Internet globale costituita da oggetti fisici in grado di fare sensing, attuare azioni, eseguire calcoli, comunicare e scambiare informazioni, che è proprio l'Internet of Things.

Il concetto dell'Internet of Things come lo conosciamo oggi è il risultato di un lungo processo di raffinazione e espansione che è stato portato avanti nel corso di decenni.

La prima comparsa del termine risale al 1999 e la sua paternità è da attribuire a Kevin Ashton, cofondatore e direttore esecutivo di Auto-ID Center (consorzio di ricerca con sede al Massachusetts Institute of Technology), il quale, all'epoca, si riferiva solo ed esclusivamente a quei dispositivi connessi che potevano essere identificati in modo univoco tramite la tecnologia Radio-Frequency IDentification (RFID)[3].

Con il passare del tempo, grazie anche allo sviluppo delle reti di sensori wireless (WSNs), le capacità sensoristiche e computazionali dei dispositivi sono aumentate, facendoli diventare sempre più intelligenti e autonomi. L'aumento della loro complessità ha fatto sì che anche le reti divenissero sempre più sofisticate e performanti, intaccando l'intera infrastruttura IoT e rendendo il nostro grado di interazione con gli oggetti sempre più fluido, intuitivo e immersivo.

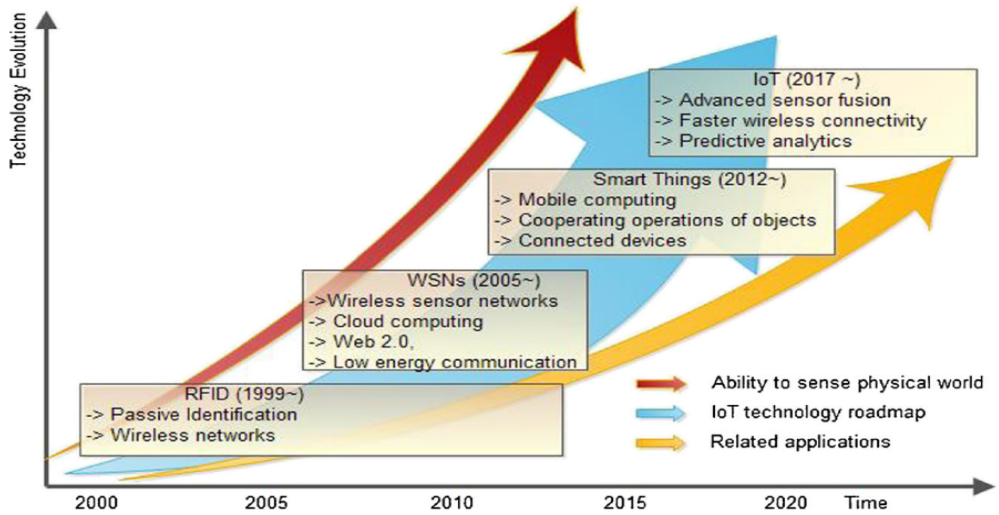


Figura 1.1: Evoluzione dell'IoT

Al giorno d'oggi, le Thing possono variare da semplici prodotti muniti di Auto-ID tag (codici a barre, codici QR, NFC e RFID tag) così da poter essere monitorati durante i loro spostamenti, fino a sistemi ad alta complessità come ad esempio un'automobile, un edifico o persino un'intera città. L'obiettivo dell'IoT è quello di rendere le Thing connesse ovunque e in qualsiasi momento, con chiunque e con qualunque cosa. Per questo motivo ad ogni Thing viene attribuito un identificatore univoco universale (UUID) che le permette di essere individuata e acceduta univocamente all'interno della rete.

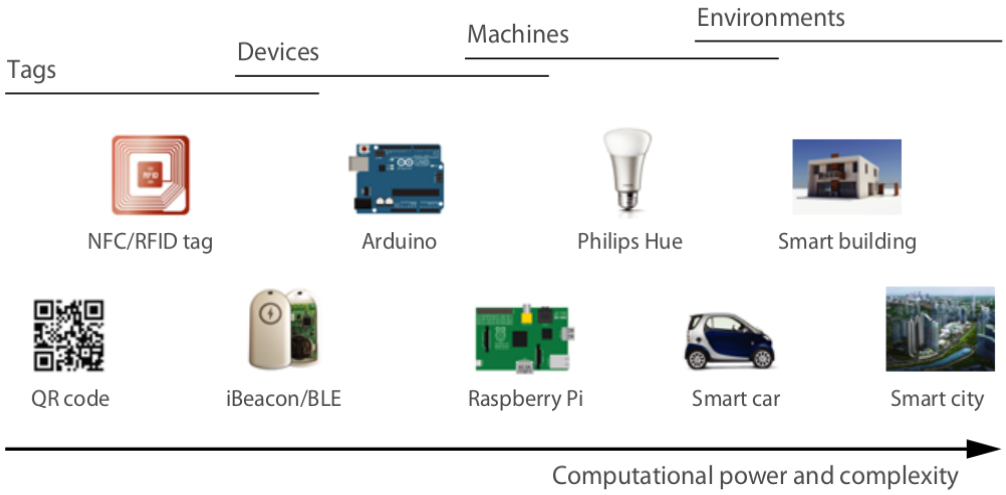


Figura 1.2: Tipologie di Thing

1.1 Architettura dell'IoT

Un requisito fondamentale del paradigma IoT è quello di rendere le Thing presenti in rete interconnesse tra loro, in grado di comunicare e scambiare informazioni. Il suo soddisfacimento, però, viene complicato dalla miriade di Thing presenti in commercio, ognuna con uno specifico hardware e uno specifico software che, il più delle volte, le rendono incompatibili e incapaci di comunicare. Oltre a questo, si aggiunge il fatto che Thing diverse possono essere utilizzate per uno

stesso scenario poiché, ad esempio, eseguono la stessa funzione. Quindi deve essere presente un sistema di regole che stabilisca in che modo le diverse tecnologie si relazionino le une con le altre e quale configurazione scegliere nel deployment dei differenti scenari. Questo sistema di regole prende il nome di architettura dell'IoT.

L'architettura dell'IoT è costiuita da quattro livelli, ognuno dei quali specifica una diversa funzionalità[2]:

1. **Livello di Sensing**
   Fanno parte di questo livello i sistemi hardware e i loro sensori. Questi ultimi sono in grado di collezionare dati provenienti dall'ambiente esterno, come ad esempio: temperatura, umidità, qualità dell'aria, velocità, rumore, pressione, movimento ecc. In alcuni casi possono avere una minima capacità di storage al fine di memorizzare un certo numero di misurazioni. Queste misurazioni, poi, sono convertite in segnali comprensibili alla Thing a esso collegata.
   Nel determinare il livello di sensing di un'applicazione IoT bisogna tener conto di diversi fattori, tra cui il costo, la dimensione, le risorse e il consumo di energia dei sensori, la politica di deployment, l'eterogeneità, la tipologia di comunicazione e di rete (es. ZigBee, Z-Wave, Bluetooth, WiFi, Ethernet ecc).
2. **Livello di Rete**
   In questo livello viene definita la configurazione della rete all'interno della quale sono connesse le Thing. In primo luogo, la tecnologia di rete deve essere tale da supportare la comunicazione machine-to-machine (M2M), altrimenti le Thing non potrebbero comunicare e scambiare dati tra loro. Si deve mantenere un Quality of Service (QoS) standard, in modo tale che le prestazioni della rete non crollino a causa dell'elevato numero di Thing connesse o dell'elevato traffico. Deve essere sempre possibile scoprire automaticamente e mappare le nuove Thing che si connettono alla rete. E infine, si deve garantire la sicurezza dei dati che viaggiano in rete, soprattutto perché essi possono contenere informazioni confidenziali, sia che si tratti di applicazioni per privati, sia di applicazioni aziendali. Questo ultimo punto è il più complesso da gestire per due motivi principali: il primo, è che l'aggiunta di un livello

di sicurezza aumenta la complessità dell'intera applicazione e il secondo, è che l'implementazione del protocollo di sicurezza al livello di Thing viene limitata fortemente dalla ridotta capacità computazionale che hanno alcune di esse. Per cui o si fornisce una sicurezza di base o la si sposta al livello applicativo.

3. **Livello di Servizio**
   Fanno parte di questo livello le tecnologie middleware. Esse offrono tutta una serie di servizi (es. API) che vengono eseguiti direttamente in rete e che si posizionano ad un livello superiore rispetto alle Thing. Infatti, questa tipologia di servizi sono dedicati alla raccolta, gestione e analisi dei dati prodotti dalle Thing (es. dati che provengono dalle attività di sensing), al fine di fornire strumenti di controllo ali utenti o alle applicazioni.
4. **Livello di Applicazione**
   Il livello di applicazione è l'ultimo livello dell'architettura dell'IoT, si trova al livello più elevato di astrazione ed è rappresentato dall'interfaccia dell'intero sistema IoT a cui l'utente può accedere, sulla quale sono esposti i metodi di interazione con il sistema sottostante (costituito da servizi, rete e sensori). L'interfaccia permette all'utente di accedere ai servizi forniti dalle differenti Thing presenti nel sistema IoT, cosa che non potrebbe fare in assenza, dato che, il più delle volte, esse appartengono a fornitori diversi e di conseguenza possono non rispettare gli stessi protocolli e standard. Quindi, in conclusione, l'interfaccia rende compatibili tra loro Thing eterogenee e fornisce meccanismi che facilitano la loro gestione e la loro interconnessione.

1.2 Interoperabilità nell'IoT

Il livello di applicazione dell'infrastruttura dell'IoT ci ha introdotto il concetto di compatibilità tra Thing. Soltanto Thing compatibili tra loro sono in grado di comunicare e scambiare informazioni. Ora la domanda che ci si pone è la seguente: come si fa a rendere Thing diverse compatibili tra loro? Ed è dalla ricerca della risposta a questa domanda che nasce il problema dell'interoperabilità.

L'associazione internazionale IEEE (Institute of Electrical and Electronics Engineers) ha definito l'interoperabilità come *"la capacità di due o più sistemi o componenti di scambiarsi informazioni e di saper interpretare e utilizzare le informazioni scambiate"*[4].

Esistono diversi livelli di interoperabilità[5], all'incirca uno per ogni livello dell'interfaccia dell'IoT. Ora di seguito li andremo ad esaminare.

- **Interoperabilità di dispositivo**
  L'interoperabilità al livello di dispositivo riguarda la capacità di rendere dispositivi eterogenei (con hardware, software, protocolli e standard di comunicazione differenti) capaci di scambiare informazioni e così di fare in modo che ogni genere di dispositivo possa essere integrato in una qualsiasi piattaforma IoT.
  Si distinguono due categorie principali di dispositivi: high-end e low-end. I dispositivi high-end (es. Raspberry Pi o smartphone) hanno sufficienti risorse e capacità computazionali da poter essere considerati a tutti gli effetti dei mini-calcolatori, mentre i dispositivi low-end (es. RFID-tag, sensori, attuatori, Arduino), al contrario, hanno ridotte capacità computazionali e risorse, in termini di durata della batteria, velocità del processore, RAM e protocolli di comunicazione e per entrambe le categorie, ogni caratteristica varia da dispositivo a dispositivo.
- **Interoperabilità di rete**
  Si sale di complessità, ora non si considerano più i dispositivi come entità isolate ma come entità interconnesse in rete. Nell'interoperabilità di rete non si fa riferimento ad un'unica rete ma a più reti (rete di reti), dove ad ognuna delle quali è connesso un dispositivo diverso. L'obiettivo è quello di far comunicare tra loro sistemi connessi a reti diverse (ad esempio tramite gateway) e fare in modo che lo scambio di messaggi sia il più possibile senza interruzioni. L'interoperabilità di rete, quindi, si dovrà occupare della gestione degli indirizzi di rete, del routing, del QoS, dell'ottimizzazione delle risorse, della sicurezza ecc.

- **Interoperabilità sintattica**

  L'interoperabilità sintattica definisce le regole da seguire circa il formato e la struttura dei dati scambiati tra sistemi eterogenei. Per fare in modo che i dati che viaggiano in rete siano interpretabili da due o più sistemi differenti, è necessario che essi concordino una grammatica comune per definire le regole sintattiche di codifica e decodifica dei messaggi scambiati. Nel caso in cui, invece, le regole di codifica del sistema mittente siano incompatibili con quelle di decodifica del sistema destinatario, si avrà un mismatch nel parse tree del processo di decodifica del messaggio, risultando, di fatto, incomprensibile al destinatario.

  Esempi concreti di schemi sintattici sono le API RESTful e WSDL o ad un livello più basso, i formati di serializzazione JSON e XML.

- **Interoperabilità semantica**

  Al livello successivo dell'interoperabilità sintattica, c'è l'interoperabilità semantica, essa viene definita dal World Wide Web Consortium (W3C) come l'attivazione di agenti, servizi e applicazioni con il fine di fare in modo che i dati e la conoscenza vengano scambiati in modo significativo (con un significato di senso compiuto), sia dentro che fuori dal Web. Il Web of Things affronta il problema tramite la condivisione di API da parte dei sistemi interconnessi. In questo modo essi sanno qual è il significato dei dati che ricevono e quali sono le operazioni da eseguire su di essi poiché, appunto, seguono lo schema definito dall'API in comune.

  L'interoperabilità semantica svolge anche un secondo compito: quello di rendere compatibili tra loro gli schemi del formato, definito dall'interoperabilità sintattica e del significato dei dati.

- **Interoperabilità di piattaforma**

  Al livello più alto si posiziona l'interoperabilità di piattaforma, che si occupa di rendere compliant piattaforme software differenti al fine di poter integrare i dati provenienti da ciascuna di esse.

  L'interoperabilità di piattaforma nasce per definire le linee guida per l'uso dei diversi sistemi operativi, linguaggi di programmazione, strutture dati, archi-

tetture e meccanismi di accesso propri delle Thing e dei dati da esse prodotti. Esistono diversi sistemi operativi specifici per dispositivi IoT, come ad esempio Contiki[1], RIOT[2], TinyOS[3] e OpenWSN[4], tutti con diverse versioni disponibili. Dall'altra parte, anche i fornitori di piattaforme IoT come Apple HomeKit[5], Android Things[6], Amazon AWS IoT[7] e IBM Watson[8] hanno i propri sistemi operativi, linguaggi di programmazione e strutture dati.
Per concludere, si può dire che è proprio grazie all'interoperabilità di piattaforma che è possibile sviluppare applicazioni multi-piattaforma a cui l'utente si può interfacciare e gestire i dati provenienti da ognuna di esse.

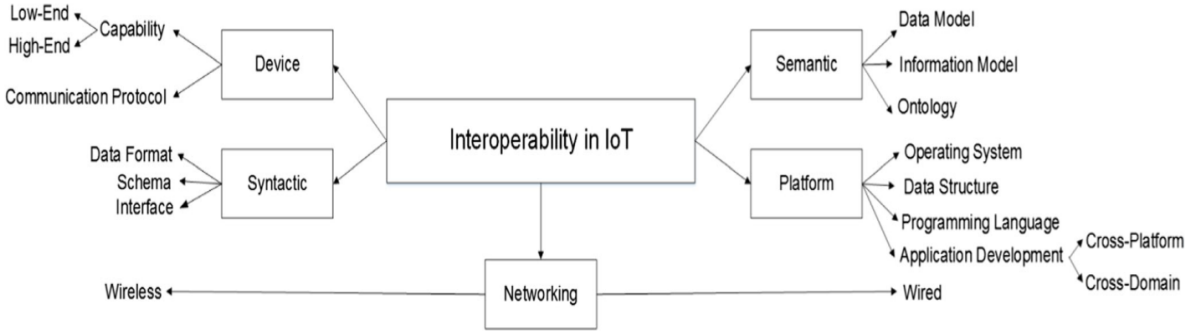


Figura 1.3: Tipologie di Interoperabilità

Senza interoperabilità, gli sviluppatori avranno l'obbligo di adattare le loro applicazioni alle specifiche delle piattaforme che andranno ad utilizzare, costringendoli in questo modo a rimanere confinati all'interno della stessa o delle stesse tipologie di piattaforme compatibili con i loro use case. Per di più, le funzionalità di queste piattaforme sono spesso limitate ai pochi utilizzi pensati e proposti dal loro produttore.

[1] https://www.contiki-ng.org/
[2] https://www.riot-os.org/
[3] http://www.tinyos.net/
[4] http://www.openwsn.org/
[5] https://www.apple.com/it/shop/accessories/all-accessories/homekit
[6] https://developer.android.com/things
[7] https://aws.amazon.com/iot/
[8] https://www.ibm.com/cloud/watson-iot-platform

L'importanza dell'interoperabilità nell'IoT è stata enfatizzata sia dall'accademia che dal settore industriale. La strada che si è scelto di percorrere è quella della standardizzazione, cioè la definizione di standard che rappresentino la base di riferimento da seguire per la realizzazione di tecnologie fra loro compatibili.

A questo punto si può procedere in due modi: il primo è quello di definire da zero nuovi standard ad-hoc, specifici per le tecnologie del mondo dell'IoT, mentre il secondo è quello di sfruttare standard già esistenti e adattarli a questo mondo. Il metodo scelto dal Web of Things è proprio quest'ultimo, con l'idea di utilizzare gli standard, i protocolli e i blueprint del Web.

Capitolo 2

W3C Web of Things

Il World Wide Web Consortium (W3C) è una comunità internazionale i cui membri, uno staff a tempo pieno e il pubblico, lavorano insieme per sviluppare gli standard web[7]. Nel 2015, il W3C ha istituito un Working Group per lavorare alla standardizzazione del WoT. Lo scopo del gruppo è stato, ed è tuttora, quello di contrastare la frammentazione dell'IoT attraverso una serie di componenti standard (meta-dati, API, ecc.) che consentono una facile integrazione tra piattaforme IoT e domini applicativi[8].

Il lavoro del W3C è stato reso necessario anche dal fatto che, nel corso degli anni, non si è riusciti a definire uno standard o un protocollo universale abbastanza forte da indirizzare i produttori e gli sviluppatori di sistemi IoT verso un'unica direzione, nonostante le continue proposte da parte di diversi organismi di standardizzazione, alleanze industriali e venditori in generale.

Il Web of Things parte da una Thing fisica e la virtualizza, cioè ne crea una copia esatta virtuale (*digital twin*) accessibile dall'utente tramite le tecnologie del web. L'utente accede direttamente alla copia virtuale senza che si debba preoccupare della posizione fisica della Thing stessa o di quali protocolli siano stati utilizzati per accedervi.

2.1 Architettura del WoT

Di seguito si andrà a descrivere l'architettura del WoT e le sue componenti. Nel farlo si farà riferimento alle ultime pubblicazioni (*draft*) del W3C pubblicate nel 2020[9], tenendo presente però che sono documenti in continua evoluzione (ed in parte incompleti) e possono subire dei cambiamenti importanti sia da un punto di vista strutturale che di contenuto.

A grandi linee, il fine dell'architettura del WoT è quello di descrivere ciò che esiste piuttosto di dire cosa deve essere implementato, non viene definito nessuna tecnologia o implementazione concerta. Per questo motivo si descrive un'architettura astratta, basata su un insieme di requisiti derivanti da vari casi d'uso (use-case), ognuno per un diverso dominio applicativo. Questi requisiti sono classificati in base all'ambito di applicazione in diverse "categorie" chiamate *building blocks*, che possono essere viste come i pilastri che ogni piattaforma che si basa sul Web of Things deve rispettare (processo di standardizzazione). L'architettura del WoT li descrive uno per uno e definisce il modo in cui sono correlati. I building blocks, poi, possono essere mappati su una varietà di deployment concreti.

Le varie sezioni dell'architettura possono essere normative o informative e per ognuna lo si andrà a specificare.

2.1.1 Casi d'uso

Questa sezione è informativa.

Di seguito si andranno ad elencare i casi d'uso presi in esame dal W3C come punti di partenza per derivare l'architettura astratta.

- Consumer (utente o consumatore finale)
  - applicazioni di Smart Home (es. domotica)
- Industriale
  - monitoraggio e previsione di possibili guasti e anomalie dei macchinari
  - applicazioni di Smart Factory

- Trasporti e logistica
  - monitoraggio di veicoli, di costi del carburante e di manutenzione degli stessi
  - tracciamento di spedizioni per garantire la loro condizione e qualità
- Utilità
  - lettura automatica di contatori residenziali, commerciali e industriali
  - monitoraggio delle condizioni e della produttività delle risorse rinnovabili
- Assicurazioni
  - monitoraggio dei rischi aziendali al fine di ridurre gli incidenti ed aumentare la sicurezza sul lavoro
  - monitoraggio delle previsioni del meteo al fine di re-instradare il traffico per evitare danni provocati dalla grandine o dalla caduta degli alberi
- Agricoltura
  - monitoraggio delle condizioni del terreno e definizione di piani ottimali per l'irrigazione e la concimazione
  - monitoraggio degli andamenti della produzione agricola
- Sanità
  - raccolta ed analisi di dati dagli studi clinici
  - controllo remoto di pazienti dopo il ricovero in ospedale
- Monitoraggio ambientale
  - monitoraggio dei livelli di inquinamento dell'aria, dell'acqua, della radioattività, ozono, temperatura, umidità ecc...
- Smart Cities
  - monitoraggio di ponti, dighe, canali per certificare le condizioni dei materiali e il loro deterioramento

  - costruzione di parcheggi intelligenti (*smart parking*) al fine di ottimizzare e monitorare la gestione degli spazi disponibili ed automatizzare i pagamenti e le prenotazioni
  - controllo intelligente dei semafori
  - monitoraggio ed ottimizzazione della gestione dei rifiuti
- Smart Buildings
  - monitoraggio del consumo energetico negli edifici
  - raccolta di feedback sulla soddisfazione degli inquilini riguardo all'abitabilità dei loro edifici
- Automobili
  - monitoraggio dello stato operativo ed ottimizzazione della manutenzione
  - miglioramento della sicurezza del conducente grazie ad un sistema di notifiche preventive sulle strade più pericolose e sulle condizioni del traffico

Oltre ai casi d'uso appena discussi, il W3C definisce anche dei pattern per illustrare come i dispositivi e le Thing interagiscono con i controller, con altri dispositivi, con agenti e server. Sono specificati sia pattern semplici come *Device Controllers* e *Thing-to-Thing*, sia pattern più complessi come *Digital Twins* e *Cross-domain Collaboration*. Questi ultimi due casi sono sicuramente i più interessanti: infatti il primo identifica una rappresentazione virtuale di uno o più dispositivi, mentre il secondo è una dimostrazione di come si possano connettere tra loro numerosi sistemi, che utilizzano protocolli o standard differenti, in un unico ecosistema.

2.1.2 Thing

Questa sezione è normativa.

Al centro dell'architettura del WoT c'è il concetto di Thing o Web Thing. Una Thing è un'astrazione di un'entità fisica (es. un dispositivo o una stanza) o virtuale (composizione di una o più Thing, es. una stanza con diversi sensori e attuatori, ognuno dei quali è a sua volta una Thing), descritta tramite un'insieme di meta-dati che seguono un determinato formato e vocabolario standard. Questo insieme di meta-dati prende il nome di *Thing Description* (TD) ed è basato sul formato JSON (JavaScript Object Notation).

La Thing Description rappresenta una vera e propria interfaccia che viene processata dai *Consumers* per poter accedere alla Thing stessa. Un Consumer può essere sia una persona fisica, sia un dispositivo elettronico, sia un'altra Thing. Nel momento in cui un Consumer andrà a visualizzare la Thing Description, esso potrà accedere a tutte le funzionalità che la Thing implementa.

Una Thing, per essere definita tale, da un lato DEVE avere almeno una TD e dall'altro, NON è possibile che più Thing condividano la stessa TD.

Il verbo utilizzato per indicare l'azione di visualizzazione della TD così da poter essere processata dai Consumers è *esporre*, mentre il verbo che indica l'azione effettuata dai Consumers di accedere alla TD e di eseguire le funzionalità della Thing è *consumare*. Perciò, è possibile consumare solo Thing che sono state esposte e una Thing esposta prende il nome di ExposedThing, mentre una Thing consumata prende il nome di ConsumedThing.

Figura 2.1: Interazione tra Consumer e Thing

Una Thing dovrà essere hostata in rete, in modo tale che potrà essere acceduta tramite un URI (Uniform Resource Identifier) ed internamente dovrà implementare uno stack software che, tramite l'interfaccia web, ne permetterà l'interazione.

Un tipico scenario in tal senso è rappresentato da un server HTTP in esecuzione su un dispositivo embedded (es. Arduino) con sensori e attuatori che implementa lo stack sotware ed espone la sua descrizione astratta sul web.

Può accadere che la rete locale dove è collegata la Thing non sia raggiungibile dalla rete Internet. Questo può accadere per colpa di dispositivi quali NAT (Network Address Translation) o firewall. Per rimediare a questa situazione il W3C WoT introduce l'entità di intermediario che si posiziona tra Thing e Consumer.

L'intermediario agisce come un proxy per la Thing ed ha una sua TD sulla quale è linkata la TD della Thing a cui fa riferimento. Dal lato del Consumer, il fatto che ci sia o meno un intermediario non cambia. La sua presenza è resa trasparente dal fatto che anche lui ha una TD, per cui risulta indistinguibile dal normale concetto di Thing.

Gli intermediari possono estendere le funzionalità di una Thing oppure possono creare una Thing virtuale componendo più Thing concrete.

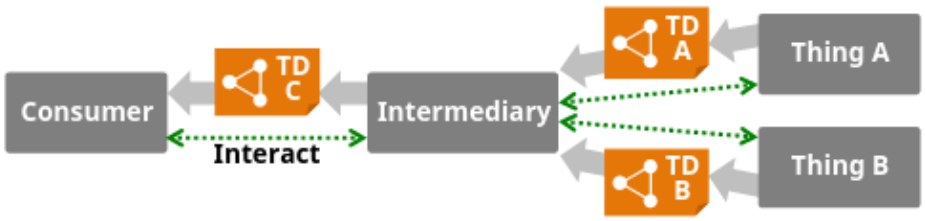


Figura 2.2: Intermediario

Successivamente si andranno ad analizzare i quattro Building Blocks, necessari al fine di implementare sistemi che risultino conformi all'architettura del WoT.

2.1.3 Thing Description

Questa sezione è informativa.

La WoT Thing Description[10] è un building block fondamentale e può essere considerata come l'entry point di una Thing (come lo è, ad esempio, la pagina *index.html* di un sito Web). La TD definisce due componenti: un modello di informazione basato su un vocabolario semantico che deve essere rispettato dalla Thing corrispondente e, come si è detto precedentemente, una rappresentazione serializzata basata sul formato dei dati JSON. Sia il modello di informazione che il formato di rappresentazione sono allineati con i Linked Data[11], in questo modo le varie implementazioni possono scegliere di utilizzare il formato JSON-LD[12] e i database a grafo per sfruttare al massimo l'elaborazione dei meta-dati.

E' stato scelto il formato JSON per due motivi principali: il primo è che è ampiamente diffuso nelle applicazioni Web, pe cui non va a incidere sulla curva di apprendimento dei sistemi WoT, il secondo è che è comprensibile dai dispositivi informatici, per cui è particolarmente adatto per applicazioni che implementano comunicazioni machine-to-machine (M2M). Per ora, il formato JSON-LD offre un buon compromesso tra semantica comprensibile alla macchina e usabilità per gli sviluppatori.

La TD ha cinque componenti principali: i meta-dati testuali, dei quali ne abbiamo già parlato in precedenza, che descrivono la Thing stessa (es. nome della Thing, ID, descrizioni ecc.), un insieme di *Interaction Affordances* che indicano le modalità con cui si può interagire con la Thing e quali sono le sue funzionalità, una serie di schemi sintattici sui dati per fare in modo che le Thing li comprenda, i *Web links*, utilizzati per esprimere qualsiasi relazione, sia formale che informale, ad altre Thing o ad altri documenti presenti sul web e infine le specifiche sulla sicurezza.

Per quanto riguarda le Interaction Affordances, il W3C ne definisce tre tipi: *Proprietà*, *Azioni* ed *Eventi*.

- **Proprietà**

  Una proprietà espone lo stato di una Thing (può essere vista come una variabile dei linguaggi di programmazione). Ad una proprietà, si deve sempre poter accedere in lettura, mentre l'accesso in scrittura è opzionale. E' inoltre possibile rendere una proprietà osservabile: in questo caso al Consumer verrà

notificato ogni cambiamento di stato della Thing con un messaggio push. Esempi di proprietà sono i valori misurati di un sensore (es, temperatura, umidità ecc.), lo stato di un attuatore o i parametri di configurazione.

- **Azione**
  Un'azione consente di invocare una funzione sulla Thing. La sua invocazione può produrre come effetto un cambiato di uno o più stati di una Thing. Esempi di azioni sono la regolazione della luminosità di una lampadina o la stampa di un documento.
- **Eventi**
  Un evento corrisponde ad una condizione, la quale, nel momento in cui si verifica, innesca un invio asincrono di messaggi push verso il Consumer. Non viene comunicato lo stato ma vengono comunicati i cambiamenti di stato di una Thing. Solitamente, per poter ricevere i messaggi in seguito al verificarsi di un evento, un Consumer deve prima sottoscriversi all'evento stesso. Di contro, un Consumer può anche decidere di cancellarsi da un evento per non riceverne più i messaggi.
  Un esempio di un evento è l'innesco di un allarme.

Di norma, per ogni Interaction Affordance si espone un endpoint (un URI), accedendo al quale si può visionare il valore di uno stato della Thing per le proprietà, invocare una funzione per le azioni e sottoscriversi ad un evento.

Le Thing, generalmente, salvano la propria TD in una directory che agisce da cache, velocizzandone così l'accesso e l'utilizzo da parte dei Consumers e facilitando la gestione dei sensori e attuatori connessi.

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

Figura 2.3: Thing Description

2.1.4 Binding Templates

Questa sezione è informativa.

In una piattaforma IoT può essere presente qualsiasi tipologia di dispositivi, ognuno con un differente set di componenti e di protocolli di comunicazione. Questo perché non esiste un singolo protocollo che vada bene per qualsiasi scenario ed è per questo motivo che il Web of Things si sta focalizzando sullo studio di come rendere dispositivi diversi interoperabili.

L'obiettivo dei Binding Templates[13] è quello di fornire un insieme di meta-dati e blueprint di comunicazione che specificano il modo in cui si deve interagire con le differenti piattaforme IoT (es. OCF[9], oneM2M[10], Mozilla IoT[11], ecc.).

Per ogni tipologia di piattaforma IoT, i Binding Templates illustrano quali siano i protocolli a cui meglio si adatta e per ogni protocollo hanno associati i meta-data che devono essere aggiunti alla TD. Ovviamente, si fa riferimento soltanto a quei protocolli che supportano le tecnologie del Web come ad esempio HTTP, COAP o MQTT. Nella scelta del protocollo di comunicazione si deve tener conto del tipo di architettura sottostante e di quali metodi espone. Esempi di pattern architetturali sono Restful e PubSub, mentre esempi di metodi ad essi collegati sono GET, PUT, POST, DELETE, PUBLISH e SUBSCRIBE.

I Consumers che consumano una TD devono implementare il corrispondente Protocol Binding per poter accedere alle funzionalità della Thing e invocarne le funzioni, andando ad includere lo stack tecnologico necessario e la sua configurazione in base alle informazioni trovate nella TD stessa.

[9] `https://openconnectivity.org/`
[10] `http://www.onem2m.org/`
[11] `https://iot.mozilla.org/`

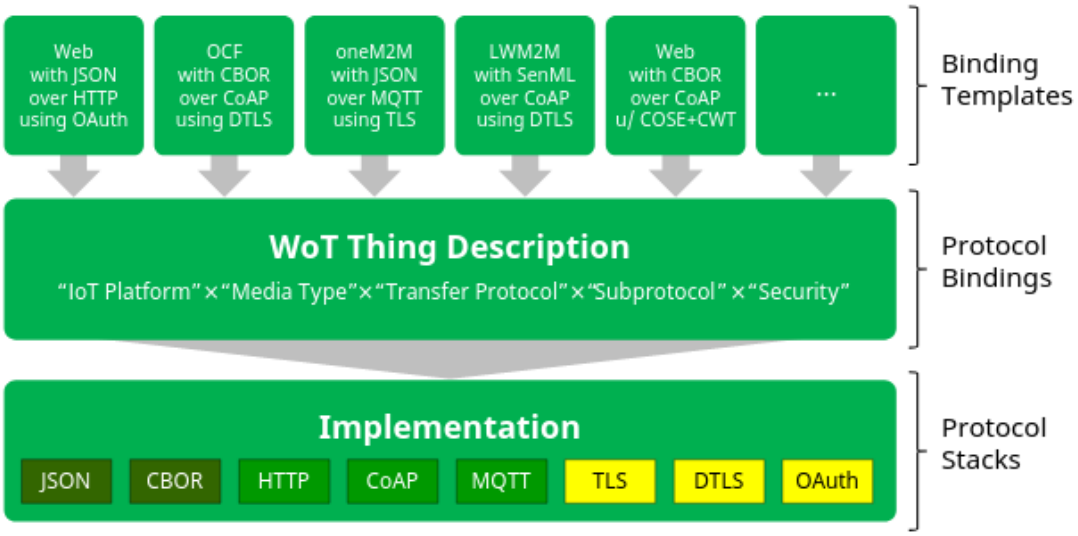


Figura 2.4: Dai Binding Templates ai Protocol Bindings

I meta-dati di comunicazione dei Protocol Bindings si estendono su cinque dimensioni:

- **IoT Platform**
  Le piattaforme IoT spesso introducono delle modifiche proprietarie ai protocolli, quali header HTTP specifici o opzioni CoAP. I Form presenti nella TD devono contenere le informazioni riguardo a queste modifiche.
- **Media Type**
  Le piattaforme IoT differiscono spesso nei formati di rappresentazione (o nella loro serializzazione) utilizzati per lo scambio dei dati. I Media Type servono proprio per identificare questi formati.
- **Transfer Protocol**
  Si tratta dei protocolli standard dello strato Applicazione (HTTP, MQTT, CoAP, ecc.) senza nessun tipo di opzioni specifiche o meccanismi di sotto-protocollo.
- **Subprotocol**
  I Transfer Protocol possono avere dei comportamenti particolari che richiedo-

no un'ulteriore specifica per poterci interagire con successo. Questa ulteriore specifica è il sotto-protocollo. Un esempio di sotto-protocollo è la tecnica *long polling* per HTTP che ovviamente non può essere identificata tramite lo schema standard dell'URI del Transfer Protocol.

- **Security**
  I meccanismi di sicurezza possono essere applicati a diversi livelli dello stack di comunicazione e possono essere anche utilizzati insieme, col fine di completarsi a vicenda.

2.1.5 Scripting API

Questa sezione è informativa.

Le Scripting API[14] sono un building block opzionale che definisce un insieme di API basate sulle specifiche di ECMAScript, simili alle API per Web browser. Le Scripting API vengono implementate in uno o più script eseguiti dalla Thing di riferimento e ne definiscono il suo comportamento, il comportamento dei Consumers e quello degli Intermediari.

Generalmente, la logica dei dispositivi IoT viene implementata a livello di firmware così da essere eseguita più velocemente e perfettamente compatibile con l'hardware del dispositivo. Nel caso delle Scripting API ciò non avviene, esse infatti, come si è detto nel paragrafo precedente, implementano la logica tramite script di programmazione che vengono eseguiti in un sistema runtime, chiamato WoT Runtime. L'utilizzo degli script dà un duplice vantaggio: da un lato, essi sono completamente riusabili, per cui possono essere utilizzati da altre applicazioni IoT senza dover essere riscritti da capo (a differenza del firmware che è specifico di ogni dispositivo), dall'altro, incrementano la produttività e riducono i costi di interoperabilità.

All'interno degli script vengono definite le strutture dati e gli algoritmi che permettono di produrre, esporre, consumare, fetchare e scoprire le Thing Descriptions. Il sistema runtime di una Thing istanzia una serie di oggetti che agiscono come interfacce per le altre Thing e per le loro Interaction Affordances.

Dato che le Scripting API necessitano di molte risorse per poter essere eseguite, solitamente vengono implementate sui nodi più potenti della rete, come per esempio i gateway.

Nelle Scripting API sono presenti tre elementi principali:

- **Oggetto API WoT**
  Questo oggetto rappresenta l'entry point dell'API, viene esposto come singleton e definisce i metodi per consumare, produrre e scoprire una Thing a partire dalla sua TD.
- **Interfaccia ConsumedThing**
  Rappresenta un'API client definita dal Consumer per operare sulla Thing: leggere, scrivere ed osservare proprietà, invocare azioni e sottoscriversi o cancellarsi ad un evento.
- **Interfaccia ExposedThing**
  Questa interfaccia estende l'interfaccia precedente e rappresenta l'API del server che esegue la logica della Thing. Al suo interno sono definiti gli handler per gestire le richieste (proprietà, azioni ed eventi) verso la Thing.

2.1.6 Security and Privacy Guidelines

Questa sezione è informativa.

La sicurezza è un problema trasversale nel WoT e dovrebbe essere considerata in ogni aspetto del design di un sistema, specialmente se nel sistema viaggiano dati sensibili. Le linee guida sulla sicurezza e sulla privacy si estendono a tutti i building blocks discussi precedentemente e per ognuno di essi si specificano quali sono i punti deboli, quali sono i principali attacchi che si possono subire e quali sono le contromisure da adottare.

Il documento sulla sicurezza e sulla privacy[15] ha uno scopo prettamente informativo, infatti non impone nessun modello concreto. Non introduce neanche nuovi meccanismi per la sicurezza, ma fa riferimento a quelli che sono già presenti sul mercato (crittografia a chiave privata, a chiave pubblica, OAuth ecc.). Fornisce anche una serie di test per validare il modello o i modelli di sicurezza scelti.

Le principali pratiche di sicurezza per i sistemi WoT sono le seguenti:

- **Pratiche di sicurezza nella progettazione di una Thing Description**
  - Proteggere i dati contenuti nella TD quando vengono acceduti, trasmessi in rete e salvati in uno spazio di archiviazione
  - Evitare l'esposizione di campi immutabili nella TD, specialmente se contengono informazioni associabili ad una particolare persona fisica
  - Minimizzare l'esposizione di informazioni pubbliche riguardanti una Thing (versione del software o sistema operativo)
  - Limitare gli accessi alla TD
- **Pratiche di sicurezza per la protezione dei dati sensibili nel sistema**
  - Utilizzare protocolli di comunicazione sicuri, come HTTPS (con TLS), CoAPS (con DTLS) e MQTTS (con TLS)
  - Informare gli utenti riguardanti quali informazioni personali possono essere esposte e quali no
- **Pratiche di sicurezza per la progettazione di interfacce di rete WoT**
  - utilizzare schemi di controllo degli accessi (autenticazione e autorizzazione)
  - Evitare l'esecuzione di processi funzionali complessi e pesanti computazionalmente senza autenticazione del Consumer
  - Minimizzare le funzionalità e la complessità delle interfacce di rete, mantenendo solamente quelle strettamente necessarie
  - Effettuare fasi di validazione e testing (inclusi i fuzz test) complesse e complete

Ogni sistema WoT si adatta meglio ad uno specifico meccanismo di sicurezza piuttosto che ad un altro. La sua scelta deve essere ponderata, tenendo presente che i modelli più sicuri sono anche i più complessi e i più dispendiosi al livello di risorse, per cui non possono essere implementati direttamente su quei dispositivi che hanno capacità computazionali ridotte (provocando un rallentamento o addirittura un

crash dell'intero sistema), ma soltanto sui nodi più potenti, come ad esempio i gateway.

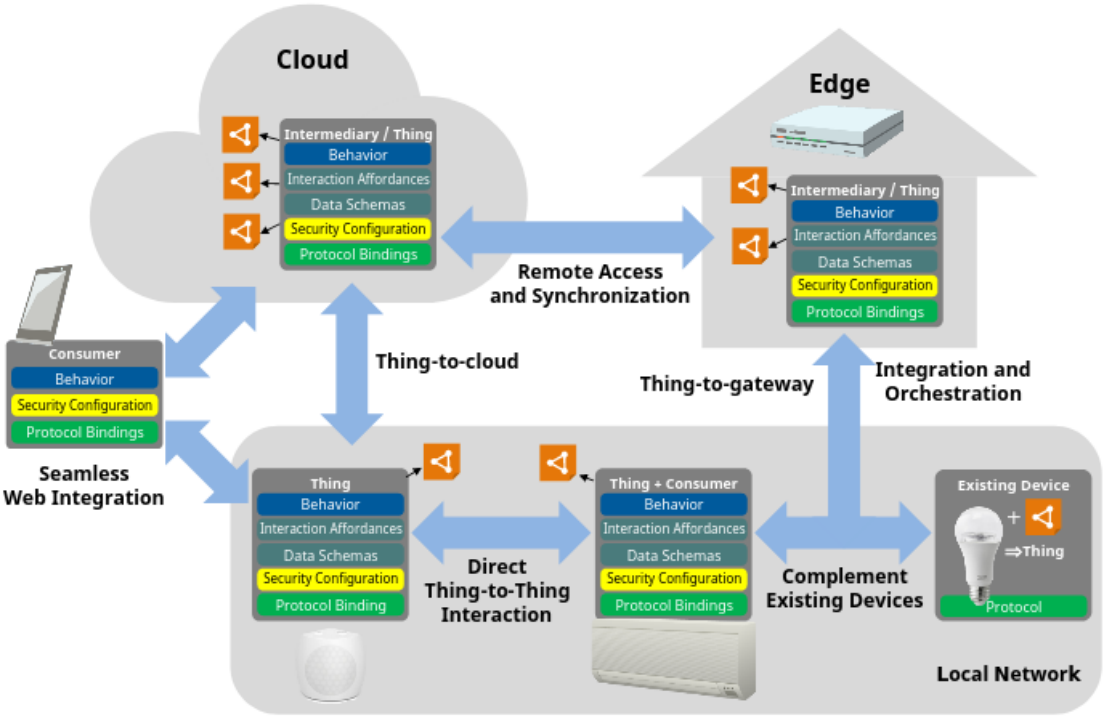


Figura 2.5: Architettura astratta del W3C WoT

Nella Figura 2.5 è mostrata una possibile configurazione di un'architettura del WoT, dove per ogni componente è specificata la struttura dei building blocks.

2.1.7 WoT Servient

Questa sezione è informativa.

Un Servient è uno stack software che implementa i building blocks visti precedentemente. Un Servient è in grado di eseguire, esporre e consumare le Thing e in base al Protocol Binding scelto, può agire sia come client che come server.

La comunicazione tra una ConsumedThing e un'ExposedThing implementate da due Servient può essere diretta o indiretta: è diretta nel caso in cui entrambi i

Servient utilizzano gli stessi protocolli di rete, risultando così mutualmente accessibili, mentre è indiretta nel caso contrario, cioè quando i Servient utilizzano protocolli diversi o sono connessi a reti differenti che necessitano di un'autorizzazione per potervi accedere (es. firewall).

Praticamente, un Servient è costituito dal codice di programmazione scritto dagli sviluppatori del sistema WoT che, prendendo in input la TD, le Scripting API e i Protocol Bindings, definisce la logica della Thing, le istruzioni che faranno parte del suo main loop, il modo in cui devono essere gestite le Interaction Affordances e il comportamento degli eventuali Consumers.

Se nelle sezioni precedenti abbiamo descritto l'architettura astratta del WoT, ora, attraverso il Servient, ne vedremo l'implementazione concreta.

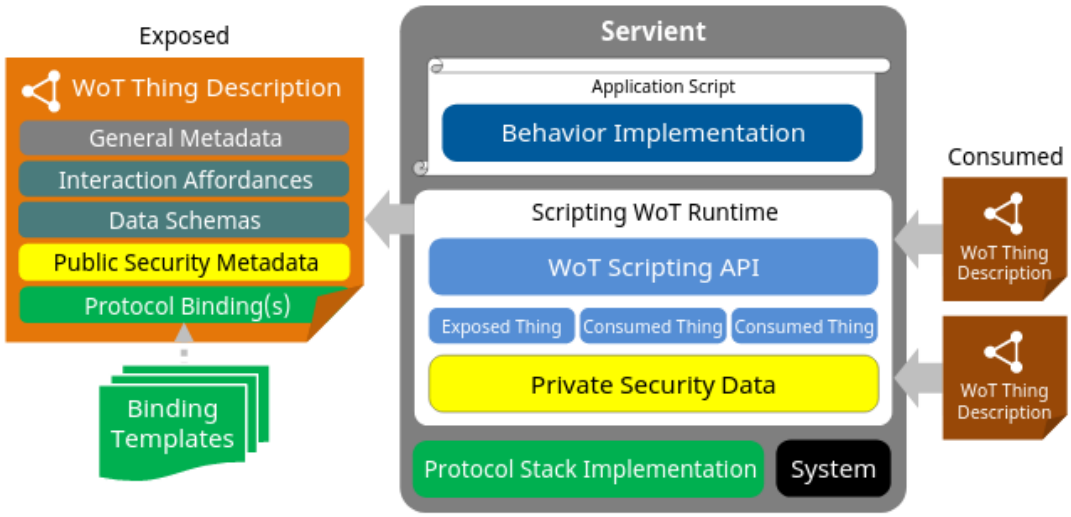


Figura 2.6: Implementazione di un Servient attraverso le Scripting API

Nella Figura 2.6 sono mostrati i moduli che compongono l'architettura di un Servient e anche se alcuni di loro li abbiamo già analizzati nelle sezioni precedenti, mancano ancora alcuni tasselli fondamentali per avere il quadro completo.

Di seguito verranno specificati, per ogni modulo, il suo ruolo e le sue funzionalità:

- **Behaviour Implementation**
  Il "comportamento" definisce la logica dell'applicazione eseguita dalla Thing. E' costituito da diversi aspetti: il comportamento autonomo della Thing (es. misurazioni dei sensori e loop di controllo per gli attuatori), gli handler per gestire le Interaction Affordances (ad es. il codice che viene eseguito concretamente quando un'interazione viene attivata), il comportamento del Consumer (che consuma la Thing) e il comportamento dell'Intermediario (es. fare da proxy per una Thing o comporre più entità virtuali).

- **WoT Runtime**
  La logica della Thing e il suo modello di interazione sono implementati in un sistema a runtime, chiamato WoT Runtime. Esso mantiene l'ambiente di esecuzione per l'implementazione del comportamento della Thing, è in grado sia di esporre e consumare Thing, sia di fetchare, processare, serializzare TD. Solitamente la logica dell'applicazione dovrebbe essere eseguita in un ambiente isolato in modo tale da prevenire accessi non autorizzati a dati sensibili (Private Security Data).
  Il WoT Runtime, poi, fornisce una serie di operazioni per gestire il ciclo di vita delle Thing (LCM) o, più precisamente, le loro astrazioni e descrizioni software, e per farlo utilizza un sistema di interfacce interne.
  Gli ulteriori compiti del WoT Runtime sono: l'implementazione dei Protocol Bindings definiti nei Binding Templates e l'interazione con il sistema sottostante della Thing per accedere all'hardware locale (sensori e attuatori) o agli spazi di archiviazione.
  Il WoT Runtime sfrutta le Scripting API per assolvere i suoi compiti.

- **Protocol Stack Implementation**
  Lo stack dei protocolli di un Servient, da un lato, implementa l'interfaccia delle ExposedThings, mentre dall'altro, viene utilizzato dai Consumer per accedere alle ConsumedThings (Thing remote). È il componente che produce i messaggi di protocollo concreti per interagire all'interno della rete.
  I Servient possono implementare più protocolli in modo tale da rendere la piattaforma IoT di riferimento interoperabile con le altre. In questo caso

però, per ogni protocollo supportato, deve essere presente la sua specifica nei Protocol Bindings.

- **System API**
  L?implementazione di un WoT Runtime può accedere all'hardware locale o ai servizi di sistema tramite API proprietarie o altri mezzi. Un dispositivo può trovarsi fisicamente esterno al Servient (dispositivo legacy), ma connesso tramite dei protocolli proprietari. In questo caso, il WoT Runtime può accedere al disposito legacy con questi protocolli e poi esporlo come Thing attraverso le Scripting API. Un Servient può agire come gateway per il dispositivo legacy, ma è un'opzione da considerare solo in caso non possa essere descritto tramite una TD. Questo meccanismo, però, non fa parte dello standard W3C WoT poiché ne limiterebbe la portabilità.

2.2 Implementazioni concrete del WoT Servient

Nella sezione precedente si sono descritti i componenti dell'architettura del Web of Things e l'implementazione astratta del WoT Servient.

In questa sezione, invece, si andranno ad esaminare due implementazioni concrete open-source del WoT Servient per Arduino, disponibili sui repository GitHub dei rispettivi sviluppatori, così da fare un confronto con l'implementazione oggetto di questo progetto.

Le implementazioni open-source del Web of Things non si fermano a due, ne sono state sviluppate molte altre, principalmente su piattaforme differenti dai sistemi embedded. Esse sono state raccolte nel corso degli anni dall'Interest Group del W3C[16], il quale gestisce un forum con il fine di discutere delle ultime novità tecniche, casi d'uso e requisiti del mercato open-source, riguardanti le tecnologie del Web e del Web of Things.

Per ogni implementazione viene fornito il link al sito web o al repository degli sviluppatori, il nome dell'organizzazione a capo del progetto, le piattaforme e i linguaggi di programmazione con cui è stata realizzata, la licenza e una breve descrizione[17].

2.2.1 WoT Arduino

WoT Arduino[18] è un'implementazione del WoT Servient per le schede Arduino UNO con Ethernet Shield (basato sul controller di rete Wiznet W5100) per la connessione in rete. È scritta in C++ ed è sviluppata dal W3C/ERCIM (European Research Consortium for Informatics and Mathematics).

Attraverso questo tool l'Arduino viene convertito in un Web Server configurabile, il quale può interagire con uno o più sensori o attuatori. Le richieste sono in gestite in un formato JSON non standard, in quanto si ha una diversa rappresentazione dei dati in memoria in modo da ridurre lo spazio da loro occupato. Infatti, gli sviluppatori hanno improntato tutto il progetto in tal senso, definendo dei meccanismi e delle strutture dati per diminuire sia la quantità di memoria occupata durante l'esecuzione del sistema che la complessità computazionale, in modo da velocizzare le interazioni tra client e server e tra server ed i suoi sensori o attuatori.

Gli oggetti in formato JSON sono rappresentati in memoria come nodi di alberi binari bilanciati (AVL). Essi vengono allocati staticamente tramite l'utilizzo di array poiché l'uso delle funzioni `malloc` e `free` potrebbe causare problemi ai microcontrollori.

Come per gli oggetti JSON, anche le Thing Properties sono rappresentate in memoria come nodi di alberi binari bilanciati ma, in questo caso, c'è da fare un passaggio in più. Dato che le proprietà sono implementate come array JSON associativi che mappano il nome della proprietà con il suo valore JSON (coppie nome-valore) mentre i nodi sono indicizzati con valori numerici, viene definita una tabella dei simboli (hash table) che mappa i nomi delle proprietà in simboli numerici.

Oltre alle Thing Proprerties, anche le Thing Actions e i Thing Events sono implementati tramite array JSON associativi con l'aggiunta di un handler che gestisce la loro invocazione.

E' presente, poi, un meccanismo di garbage collection basato sull'algoritmo di *mark and sweep* che viene invocato quando la memoria disponibile sta per terminare e le celle di memoria contenenti oggetti JSON non più raggiungibili

vengono reclamate.

Infine, per ridurre la dimensione dei messaggi scambiati in rete in formato JSON è presente un meccanismo di codifica e decodifica che li converte in binario e per ridurre lo spazio in memoria RAM occupato dalle variabili di tipo stringa, le si salva nella memoria flash tramite la macro `F`.

I protocolli utilizzati sono: HTTP al livello applicazione e TCP al livello trasporto.

Il tool si compone di 12 file di libreria, 11 file di implementazione e 1 sketch da flashare sull'Arduino.

Utilizzo

Innanzitutto occorre stabilire se usare per il Web Server un indirizzo IP fisso o dinamico tramite il protocollo DHCP. Allo stesso modo si può decidere di utilizzare un indirizzo IP e una porta prefissati per il gateway oppure il servizio di DNS multicast per individuarli dinamicamente.

Una volta stabilito questo, si può passare alla definizione della Thing e dei suoi componenti.

La Thing, in questa implementazione, viene gestita tramite un oggetto di tipo `Thing` che va dichiariato all'interno dello sketch principale. Sucessivamente si definiscono le proprietà della Thing e dato che sono gestite tramite una tabella dei simboli, implementata tramite la libreria *Names.h*, sono registrate come simboli. Un simbolo è un oggetto `Symbol` che viene definito a partire dal nome della proprietà in questione. Un esempio di dichiarazine di un simbolo è il seguente: `Symbol foo = names->symbol(F(''foo''));`.

Ad una proprietà si possono assegnare sia valori di tipo base, sia oggetti JSON. Nel caso di un valore di tipo base, lo si può assegnare direttamente all'oggetto `Thing` tramite il metodo `set_property(PROPERTY_NAME, JSON::TYPE(VALUE))`, nel caso invece di un oggetto, prima lo si deve dichiarare, poi si devono definire i suoi campi e solo successivamente lo si può assegnare alla Thing.
Per recuperare il valore di una proprietà si usa il metodo
`get_property(SYMBOL_TABLE_NAME, PROPERTY_NAME)`.

Anche le azioni e gli eventi devono essere registrati come simboli prima di poter essere invocati. Le azioni vengono invocate tramite il metodo
`int invoke(Symbol action, ResponseFunc response, ...)`. Se l'azione non prevede una risposta, allora si deve passare `NULL` come secondo parametro. I parametri successivi sono opzionali e rappresentano i parametri richiesti in input dall'azione. Se l'azione termina con successo, il metodo `invoke` ritornerà un id positivo che verrà inviato all'handler della risposta che è definito nel modo seguente:
`void (*ResponseFunc)(unsigned int id, Thing *thing, ...)`.
Per lanciare gli eventi, invece, si deve chiamare il metodo
`void raise_event(Symbol event, ...)`, il quale reindirizza la chiamata all'handler definito come segue: `void (*EventFunc)(Symbol event, Thing *thing, ...)`.

Per visualizzare la TD si utilizza il comando `print()`, mentre per invocare il garbage collector si usa il comando `WebThings::collect_garbage()` fornito dalla libreria *WebThings*. Questa libreria fornisce anche i metodi di gestione delle Interaction Affordances citati precedentemente.

All'interno del metodo `setup()` di Arduino si deve, obbligatoriamente, connettere il server alla rete e avviare il protocollo di trasporto tramite `transport.start()` ed esporre la TD. Nel metodo `loop()`, invece, si gestiscono la coda degli eventi con `event_queue.dispatch()` e le richieste verso la Thing attraverso il comando `transport.serve()`.

2.2.2 Arduino RDF Server

Arduino RDF Server[19] è, come WoT Arduino, un'implementazione compatibile con le schede Arduino UNO con Ethernet shield, scritta in C++ e sviluppata dal LIRIS Lab di Lione, in Francia.

Anche in questo caso l'Arduino viene convertito in un Web Server configurabile, il quale espone le funzionalità dei vari sensori e attuatori ad esso collegati, risponde alle richieste in formato RDF, gestisce le richieste di riconfigurazione e può essere riavviato senza la perdita dei dati di configurazione. Questo è possibile grazie al fatto che i dati vengono memorizzati nella memoria EEPROM.

Ogni funzionalità viene esposta tramite un *Servizio*, il quale è descritto tramite le RESTful Web API Hydra[12]. Un servizio può essere configurato, abilitato e disabilitato a runtime senza la necessità di flashare un nuovo sketch contenente le nuove istruzioni. Nel momento in cui si fa richiesta di un servizio viene invocata una funzione che interagisce con il sensore o con l'attuatore collegato e ne modifica lo stato. Le funzioni possono prendere in input o restituire in output delle variabili che prendono il nome di *Risorse*.

I pacchetti scambiati tra client e server vengono trasmessi tramite il protocollo CoAP al livello applicazione, mentre al livello trasporto si utilizza il protocollo UDP.

Il tool è composto da sette file: sei dei quali, *Coap.h*, *ResourceManager.h*, *Semantic.h*, *Coap.cpp*, *ResourceManager.cpp* e *Semantic.cpp*, sono librerie e loro implementazioni, mentre il restante, *WebServer_Hackfest.ino*, costituisce lo sketch da flashare sull'Arduino.

Utilizzo

Come prima cosa è necessario configurare quali funzioni devono gestire quali pin della scheda Arduino, in modo tale da avere una corrispondenza logica tra il servizio e il sensore o attuatore a cui fa riferimento.

In secondo luogo occorre definire la Thing Description ed i relativi servizi. Entrambi devono essere definiti all'interno della libreria *Semantic.h* in formato JSON. I servizi (scritti in Hydra) vanno definiti all'interno dell'array
`const char CAPABILITIES[] PROGMEM` e per ognuno bisogna specificare i seguenti campi:

- `@id`: URL del servizio
- `@type`: tipo di operazione in Hydra (tipo di default hydra:Operation)
- `method`: metodo di accesso al servizio (GET, POST o PUT)
- `label`: etichetta riferita al servizio (opzionale)

[12] `http://www.hydra-cg.com/`

- `expects`: identificatore della risorsa da passare in input alla funzione invocata dal servizio (solo per POST e PUT)
- `returns`: identificatore della risorsa restituita in ouptut (solo per GET e POST)

Prima di poter invocare un servizio è necessario abilitarlo tramite la richiesta PUT `enable`, alla quale si deve passare l'identificatore del servizio e il numero di pin dell'Arduino nei quali è collegato il sensore o l'attuatore oggetto del servizio. La richiesta PUT `reset`, invece, resetta tutti i servizi.

Una volta descritti i servizi si passa alla definizione delle relative funzioni. Esse vanno definite nel file *Semantic.cpp*, mentre nel file di libreria *Semantic.h* vanno inserite soltanto le intestazioni.

Una funzione non deve ritornare nulla (`void`) e deve avere obbligatoriamente i seguenti parametri:

- `uint8_t pin_count`: numero di pin utilizzati
- `uint8_t pins[n]`: array all'interno del quale vanno inseriti i numeri dei pin impiegati (`n` deve corrispondere a `pin_count`)
- `JsonObject& root`: documento JSON corrispondente al corpo della richiesta al servizio
- `uint8_t results`: array in cui sono memorizzati quei dati che verranno inviati al client come risposta dell'invocazione del servizio

Terminata la scrittura della funzione, la si deve linkare al corrispondente servizio nel file *WebServer_Hackfest.ino* tramite l'istruzione `coap.addOperationFunction(&FUNCTION_NAME, (char* )‘‘SERVICE_ID‘‘);` da inserire all'inizio della funzione `setup()` di Arduino. Infine si definiscono le risorse all'interno della libreria *Semantic.h*.

Come per i servizi, anche le risorse vanno definite in un array in formato JSON (`const char VARIABLE_NAME[] PROGMEM`), con la differenza che si ha bisogno di un array per ognuna di loro. L'array deve avere lo stesso nome della risorsa e deve contenere, tra gli altri, i seguenti campi obbligatori:

- `@type`: identificatore della risorsa che deve corrispondere a quello inserito nei campi `expects` e `returns` presenti nella definizione del servizio che la gestisce
- `value`: valore della risorsa (inizialmente è vuoto)

Infine, sempre in *Semantic.h*, si devono inserire all'interno dell'array `PGM_P const json_resources[] PROGMEM` i nomi degli array di ciascuna risorsa definiti precedentemente e contestualmente, si aggiorna la costante `RESOURCE_COUNT` per farla corrispondere al loro numero.

La TD con i relativi servizi è esposta nella root (`/`) del Web Server e può essere acceduta tramite una richiesta GET.

Se volessimo fare un parallelismo tra i componenti di Arduino RDF Server e quelli dell'architettura del WoT, potremmo dire che i servizi del primo corrispondono alle Interaction Affordances del secondo. In particolare, le proprietà e le azioni del WoT sono denominate risorse e funzioni in Arduino RDF Server. Per quanto concerne gli eventi in Arduino RDF Server, essi non sono ancora stati implementati. Il protocollo di comunicazione (Protocol Binding) scelto è CoAP, mentre le Scripting API sono le API Hydra.

In conclusione, le due implementazioni risultano essere incomplete, poiché offrono soltanto funzionalità di base e non supportano l'intero set di specifiche del Web of Things. Attualmente, esse non vengono aggiornate già da alcuni anni. La prima dal 2016, mentre la seconda dal 2017.

Parte 2

Embedded WoT Servient

Capitolo 3

Progettazione

Questo capitolo offre una panoramica della fase di progettazione del software Embedded WoT Servient. In particolare, dopo aver definito l'obiettivo ed i requisiti principali, viene descritta l'architettura del sistema. Di ogni componente, quindi, vengono illustrate nel dettaglio le caratteristiche e le funzionalità, oltre alla descrizione del ruolo assunto all'interno dell'architettura.

3.1 Obiettivo

Lo scopo del progetto è lo studio e la realizzazione dello stack software WoT Servient per sistemi embedded (es. Arduino, NodeMCU ecc.), rispettando le specifiche ed i componenti della sua architettura, descritta nel capitolo 2.

La ragione che ha portato allo sviluppo di questo lavoro di tesi è stata la presa di coscienza della mancanza di un'implementazione del WoT Servient per questa tipologia di sistemi che risulti essere completa, funzionante, coerente con le specifiche del Web of Things e aggiornata alle ultime tecnologie.

La difficoltà nella realizzazione di un sistema come il WoT Servient risiede nel fatto di riuscire a semplificare ed ad ottimizzare la complessa struttura di funzionalità che offre, rendendola compatibile e soprattutto eseguibile da quella categoria di sistemi le cui capacità computazionali, risorse e autonomia risultano limitate, che prende il nome di sistemi embedded.

Alla base di questo progetto si è presa in esame l'implementazione del WoT Servient di Eclipse Foundation thingweb.node-wot[13], sviluppata con il linguaggio di programmazione JavaScript. La si è scelta, non perché sia compatibile con i sistemi in questione, infatti non lo è, ma perché rappresenta la prima implementazione del WoT Servient ufficialmente riconosciuta dal WoT, costantemente aggiornata e per questi motivi, viene presa come modello per tutti i lavori che si svolgono in questo ambito. Di questa applicazione, si sono analizzati i componenti, la loro struttura interna e il modo in cui si interfacciano con l'utente, trasportandoli, poi, nel mondo dei microcontrollori.

Un requisito fondamentale del sistema in oggetto è la facilità d'uso nell'interazione con l'utente finale. Infatti, con pochi semplici passi, deve essere possibile navigare al suo interno, comprendere ed eseguire le sue funzionalità.

Per questo motivo, oltre alle funzionalità tipiche del WoT Servient, ne sono state implementate di nuove, come ad esempio la creazione automatica dello script da far eseguire al microcontrollore e l'inserimento di funzioni da file C esterni, tutte con lo stesso fine di rendere le attività più complicate facili e trasparenti per l'utente.

Nelle sezioni seguenti si utilizzeranno i termini sketch o file di scripting per fare riferimento al file, contenente la logica della Thing, in esecuzione sui sistemi embedded, mentre per riferirsi a questi ultimi si useranno alternativamente i termini microcontrollori o board/schede di prototipazione.

3.2 Requisiti

Ora si andranno a descrivere i requisiti funzionali e tecnici che l'applicazione deve soddisfare.

3.2.1 Requisiti funzionali

L'implementazione del WoT Servient deve fornire le seguenti funzionalità:

[13] `https://github.com/eclipse/thingweb.node-wot`

- **Creazione Thing Description**
  L'utente deve poter essere in grado di definire l'insieme dei meta-dati che andranno a costituire la TD esposta dalla Thing. Oltre alla possibilità di inserire i meta-dati standard, cioè quelli stabiliti dalle specifiche del W3C, all'utente è permesso anche inserirne degli altri, a completamento o a supporto dei precedenti.
- **Esposizione Thing Description**
  La Thing deve essere processata e renderizzata leggendo la sua TD. L'utente deve poter accedere e interagire con le sue Interaction Affordances, per cui, in dettaglio, deve poter leggere il contenuto delle sua proprietà, invocare le azioni e ricevere gli eventi in tempo reale.
- **Creazione file di scripting**
  L'utente, una volta definita la TD, deve poter creare il file di scripting (sketch) nel linguaggio Embedded-C compatibile con i sistemi embedded, contenente la logica dell'applicazione, che verrà eseguito dalla Thing.
- **Compilazione file di scripting**
  L'utente, una volta creato il file di scripting, lo deve poter compilare sulla specifica board scelta come Thing, così da verificare la presenza di eventuali errori sintattici e/o semantici da correggere.
- **Flashing file di scripting**
  L'utente, una volta compilato lo sketch, deve poterlo caricare (upload) come firmware e farlo eseguire al microcontrollore,il quale, assumerà a tutti gli effetti il ruolo di Thing e sarà in grado di parsare e gestire tutte le richieste provenienti dai Consumers.

3.2.2 Requisiti tecnici

Ora si andranno ad approfondire i requisiti del sistema da un punto di vista tecnico:

- **Visualizzazione Thing Description**

La TD deve poter essere processata tramite le tecnologie del Web e le API RESTful, per garantire l'interoperabilità dell'applicazione. Per questo motivo è necessario che la Thing sia connessa in rete, che abbia un indirizzo IP e un URL in modo da poter essere raggiungibile attraverso una chiamata GET (tramite ad esempio un browser). Lo stesso ragionamento si estende a sua volta anche alle Interaction Affordances, per cui deve essere possibile accedere alle proprietà e agli eventi tramite richieste GET ed invocare le azioni mediante richieste POST.

- **Protocol Bindings**
  Dato che le interazioni con la Thing devono avvenire tramite le tecnologie del Web, è necessario fornire un binding con i protocolli compatibili con questo tipo di tecnologie e che permettono lo scambio di messaggi tra entità connesse alla rete Internet, come ad esempio l'HTTP e le WebSocket.
- **Semantica**
  I meta-dati delle Thing si devono basare su ontologie note e adeguate alla tipologia di entità che rappresentano.

3.3 Architettura

In questa sezione viene presa in esame l'architettura dell'Embedded WoT Servient, analizzandone i diversi componenti e i flussi delle principali funzionalità offerte.

3.3.1 Componenti

L'unico componente del sistema con cui l'utente interagisce è la **Thing CLI** (command-line interface). La Thing CLI è un'interfaccia a riga di comando che, tramite una procedura guidata completamente interattiva, permette all'utente di definire la logica della Thing, in particolare di:

- inserire i meta-dati della Thing Description e salvarla in un file JSON
- inserire le proprietà, specificando per ciascuna il nome ed il tipo

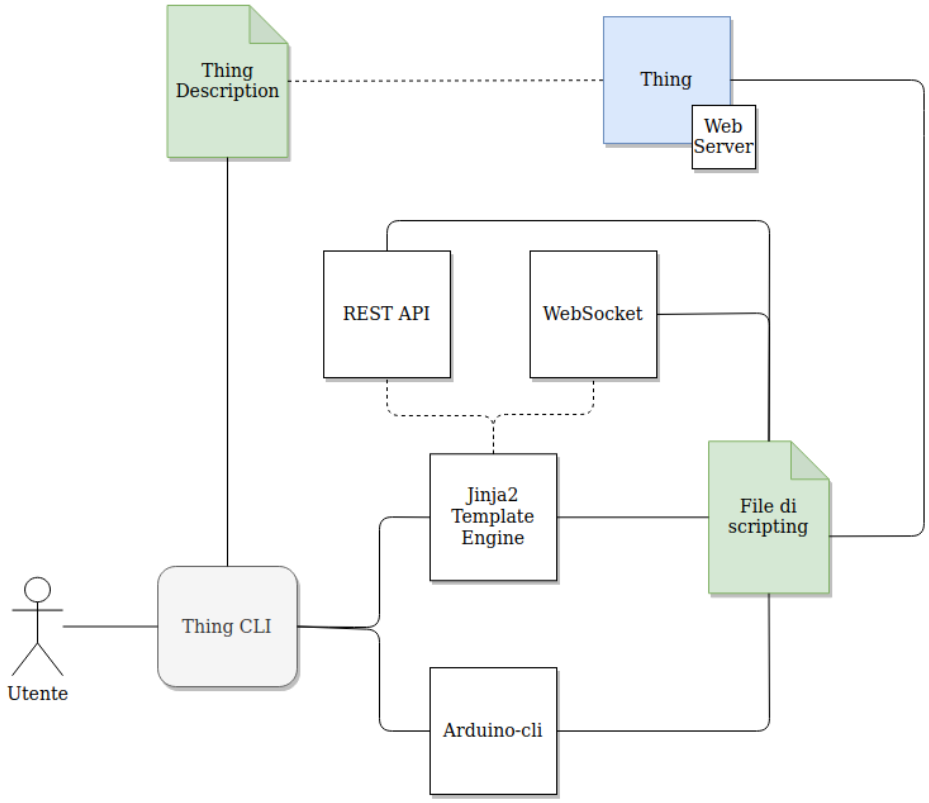


Figura 3.1: Architettura dell'Embedded WoT Servient

- inserire le azioni, specificando per ognuna il nome, l'insieme dei parametri in input (nome e tipo), il tipo di output ed il corpo della funzione invocata
- inserire gli eventi, specificandone il nome ed opzionalmente gli schemi di sottoscrizione, notifica e cancellazione a cui i relativi messaggi devono sottostare
- inserire l'SSID e la password della rete a cui la Thing si dovrà connettere
- inserire le porte per l'accesso ad Internet e al canale WebSocket
- inserire librerie, costanti, variabili e funzioni addizionali e codice generico al file di scripting, sia come integrazione delle funzionalità già offerte dalla Thing, sia per aggiungerne di nuove
- inserire il corpo delle funzioni sia tramite riga di comando che da file
- inserire nuovi tipi di messaggi da poter essere scambiati lungo il canale

WebSocket

- compilare il file di scripting sulla board
- caricare il file di scripting compilato sulla board per la sua esecuzione

Le azioni appena elencate sono a disposizione dell'utente ed è lui che decide quali eseguire e quali no semplicemente dandone il consenso o il dissenso.

Le parti di implementazione del Server Web, delle API REST e delle WebSocket, di creazione del file di scripting, di esecuzione dei comandi per la sua compilazione e il suo caricamento sono completamente trasparenti all'utente e vengono avviate, una per una, mano a mano che egli esegue l'azione corrispondente.

- **Server Web**
  Il sistema embedded espone la Thing attraverso un Server Web raggiungibile tramite un URL. L'URL è composto dall'indirizzo IP del server seguito dal nome che l'utente ha attribuito alla Thing. Nella root (`/`) del server viene visualizzata la TD in formato JSON. Ad ogni Interaction Affordance viene fatto corrispondere un endpoint concatendando l'URL della root del server al tipo (*properties* per le proprietà, *actions* per le azioni e *events* per gli eventi) e al nome dell'interazione. Il server rimane in attessa su ciascuno di questi endpoint e nel momento in cui riceve una richiesta da parte di un client, la processa e invia al mittente la risposta insieme al codice di successo o di errore dell'operazione (200 OK o 405 Method Not Allowed). Le risposte del server sono tutte in formato JSON e per le proprietà si accettano solo chiamate GET, mentre per le azioni solo chiamate POST.
- **WebSocket**
  Il protocollo WebSocket viene utilizzato principalmente per la gestione degli eventi. E' stata scelta questa tecnologia poiché si adatta bene alla comunicazione asincrona tipica degli eventi. La Thing, infatti, espone un Server WebSocket che resta in ascolto sui vari endpoint degli eventi. Nel momento in cui un Consumer si sottoscrive ad un particolare evento, inviando un messaggio di richiesta di sottoscrizione, il server aprirà un canale di comunicazione riservato e lo terrà aperto finché non cadrà la connessione o il client

non si disconnetterà. Ogniqualvolta si verifichi l'evento a cui il Consumer si è sottoscritto, il server gli invierà un messaggio di notifica attraverso il canale aperto precedentemente. Per garantire l'interoperabilità dei protocolli, tramite WebSocket sono state supportate anche le richieste verso le proprietà e le azioni, come avviene con l'HTTP.

- **Template Engine**
  Mediante il Template Engine viene generato il codice contenuto nel file di scripting nel linguaggio di programmazione Embedded-C (un misto tra C e Java). Questo tipo di linguaggio è quello che si utilizza generalmente per la programmazione dei microcontrollori come Arduino ed è il linguaggio di default dell'interfaccia Arduino IDE[14]. Nel corso dell'interazione con la Thing CLI, l'utente sceglie quali sono le funzionalità che la sua Thing dovrà avere, la Thing CLI, poi, le comunica al template engine, il quale genera la parte di codice che le implementa. Il codice corrispondente alle funzionalità che l'utente non sceglie non verrà generato di conseguenza.
- **Arduino-cli**
  Arduino-cli è un'interfaccia a riga di comando, come lo si evince dal nome, utilizzata per compilare e caricare il file di scripting sui sistemi embedded. Oltre ad avere queste due funzionalità, l'interfaccia permette di installare i driver per le varie tipologie di sistemi embedded, di installare le librerie incluse negli sketch, di verificare quali board sono connesse e su quali porte ecc. Questi strumenti sono importanti, soprattutto perché per compilare uno sketch, è necessario avere installato tutte le librerie chiamate al suo interno, mentre per caricarlo su una board, è necessario aver installato i suoi driver.

3.4 Scelte progettuali

Il sistema embedded scelto per sviluppare e testare l'applicazione oggetto di questo lavoro di tesi è il NodeMCU (tecnologia ESP8266). Questa scelta ha vincolato, di conseguenza, la tipologia di librerie e di codice generati dal template engine

[14] `https://www.arduino.cc/en/main/software`

e contenuti nel file di scripting, per essere compatibili con il sistema hardware del NodeMCU.

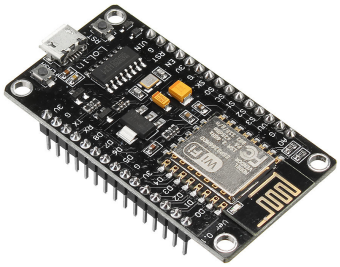


Figura 3.2: Microcontrollore NodeMCU

Il linguaggio per la programmazione di microcontrollori scelto, come è stato accennato nella sezione precedente, è l'Embedded-C di Arduino IDE.

I Protocol Bindings scelti per la gestione delle richieste Web sono l'HTTP e le WebSocket, mentre come Scripting API sono state preferite le API REST.

La visibilità e l'accesso alla Thing sono rimasti circoscritti alla sola rete Internet locale, senza quindi avere la possibilità di essere consumata al di fuori.

Capitolo 4

Implementazione

L'implementazione del WoT Servient ha richiesto l'integrazione di diversi componenti al fine di fornire le funzionalità illustrate nel capitolo precedente. Di seguito, verranno descritte le tecnologie utilizzate ed i moduli sviluppati, specificandone i motivi della scelta e le modalità di integrazione. Allo stesso tempo, si analizzeranno i problemi implementativi più significativi specificando come sono stati risolti.

4.1 Tecnologie

Trattandosi di un progetto "full-stack" sono risultate necessarie numerose tecnologie. La componente principale del sistema Servient è la Thing CLI, l'interfaccia a linea di comando con la quale l'utente interagisce, implementata tramite il linguaggio di programmazione Python v3.6.x e la libreria Click[15] v7.x. La Thing CLI si interfaccia con il template engine Jinja2[16] v2.10.x e con la piattaforma arduino-cli[17] v0.9.x, la quale, a sua volta, interagisce con la scheda di prototipazione NodeMCU v1.0 (ESP-12E Module). Le librerie di Arduino utilizzate sono

[15] `https://palletsprojects.com/p/click/`
[16] `https://palletsprojects.com/p/jinja/`
[17] `https://arduino.github.io/arduino-cli/`

principalmente tre: ESP8266WebServer[18], WebSockets[19] v2.1.x e ArduinoJson[20] v6.14.x.

4.1.1 Python

Tra i motivi che mi hanno portato alla scelta del linguaggio Python ci sono sicuramente la sua facilità d'uso, i suoi costrutti intuitivi, il numeroso insieme di librerie di cui dispone, le quali lo rendono un linguaggio versatile, adatto allo sviluppo di qualsiasi applicazione. E' un linguaggio multi-paradigma, per cui supporta la programmazione orientata agli oggetti, strutturata e funzionale. Ha un forte sistema di tipi che viene eseguito a runtime, per cui non occorre assegnare i tipi alle variabili essendo inferiti dal compilatore. E' un linguaggio interpretato ed il suo interprete supporta la modalità d'uso interattiva (REPL) attraverso la quale è possibile inserire codice direttamente da un terminale, visualizzando immediatamente il risultato. Usa un sistema di garbage collection per la liberazione automatica della memoria.

E' un linguaggio in continua evoluzione, versione dopo versione vengono aggiunti nuovi costrutti e nuove funzionalità, soprattutto anche grazie alla sua community.

Le principali librerie di Python che ho utilizzato insieme a `click` e `jinja2` sono:

- **os**
  Questo modulo offre una serie di funzionalità dipendenti dal sistema operativo. Le funzionalità che ho utilizzato riguardano il parsing dei percorsi di file e cartelle, tra cui la verifica della loro esistenza, la creazione di cartelle, il cambio della cartella di lavoro e il recupero delle variabili d'ambiente.
- **sys**
  Questo modulo fornisce l'accesso sia a variabili, sia a funzioni che interagiscono con l'interprete di Python. E' stato impiegato per la gestione della

[18] `https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WebServer`
[19] `https://github.com/Links2004/arduinoWebSockets`
[20] `https://arduinojson.org/`

terminazione del programma principale a seguito di errori nell'inserimento dei dati. Gli errori riguardano ad esempio il fallimento della validazione della TD o la mancata terminazione con successo dei processi di compilazione e caricamento dello sketch.

- **json e jsonschema**
  Della libreria `json` è stato utilizzato il metodo `load` per convertire un file o una stringa in formato json in un dictionary e il metodo `dump` per l'operazione opposta. La libreria `jsonschema`, invece, è servita per validare la Thing Description specificata dall'utente rispetto allo schema JSON stabilito dal W3C. L'utente, infatti, può proseguire nella definizione della Thing solamente nel caso in cui la TD sia valida.

- **subprocess e shlex**
  Il modulo `subprocess` è stato utilizzato per eseguire comandi bash tramite Python, mentre il modulo `shlex` è stato impiegato per parsare correttamente i comandi bash e renderli eseguibili dalle shell UNIX. Si è usufruito dei due moduli per eseguire i comandi dell'interfaccia arduino-cli in modo trasparente per l'utente.

4.1.2 Click

La libreria Click permette di creare interfacce a linea di comando, altamente configurabili, con l'utilizzo di relativamente poche istruzioni. I principali punti di forza sono: la possibilità di definire comandi innestati, la generazione automatica della pagina di help e il supporto del caricamento lazy dei sotto-comandi a runtime. È una libreria strutturata a classi, in cui si possono definire tipi e comandi personalizzati, ma anche decoratori per i comandi stessi. I metodi che mette a disposizione sono semplici e molto intuitivi, eccone alcuni che ho utilizzato per la realizzazione della Thing CLI:

- `@click.group()`: consente la definizione di un gruppo di comandi
- `@click.command()`: permette di definire uno specifico comando

- `click.invoke(nome_comando)`: consente di invocare il comando passato tra parentesi
- `@click.pass_context`: passa il contesto da un comando ad un altro all'interno di un gruppo di comandi
- `click.prompt()`: permette l'inserimento di dati (input) da tastiera
- `click.confirm()`: rappresenta una scelta per l'utente, il quale la può confermare o rifiutare
- `click.echo(stringa)`: stampa a schermo la stringa in input (funziona come una semplice `print()` di Python)

I metodi preceduti dalla `@` rappresentano delle macro. Ad ogni comando deve essere associata una funzione chiamata callback, la quale viene invocata nel momento in cui si digita il comando corrispondente sul terminale e si clicca INVIO. Click permette anche di chiamare un comando all'interno di un altro inserendo nel codice del chiamante il metodo `invoke()` e passandogli il comando chiamato. È possibile, infine, associare delle opzioni e argomenti ad un determinato comando, rispettivamente, attraverso i metodi `click.option()` e `click.argument()`, ma non li ho utilizzati nell'implementazione del Servient.

Per creare una CLI, innanzitutto occorre definire la callback generale che incorpora i vari comandi e solo successivamente le callback di ogni comando. Grazie a questa gerarchia, all'interno del metodo `main` dell'applicazione è necessario invocare solamente la callback generale, a cui poi spetterà il compito di chiamare gli altri comandi.

4.1.3 Jinja2

Jinja2 è un template engine compatibile con il linguaggio Python, di cui condivide la sintassi di alcuni costrutti. Viene utilizzato principalmente da framework per applicazioni Web come Flask[21]. Supporta l'ereditarietà dei template, in quanto è possibile usare uno stesso layout o uno simile per tutti i template del sistema, possiede una sandbox e supporta completamente la codifica Unicode. Nonostante

[21] `https://palletsprojects.com/p/flask/`

venga impiegato maggiormente per creare file in HTML, XML o in altri formati di markup, Jinja2 può essere utilizzato per creare qualsiasi tipo di file dato che il file in output non necessita di un'estensione specifica.

Un file di template è costituito da una parte statica e da una parte dinamica. La prima rimane sempre fissa, non viene modificata nelle diverse esecuzioni del template, mentre la seconda, come lo si evince dal nome, cambia a seconda del tipo di istanza con cui si fa la resa del template. Solitamente, nella parte dinamica sono presenti variabili e/o espressioni che vengono rimpiazzate con dei valori specifici nel momento della resa del template. La componente dinamica di Jinja2 è rappresentata da una variabile racchiusa tra parentesi graffe doppie, in questo modo: `{{ nome_variabile }}`.

L'interazione con Jinja2 richiede tre fasi principali:

1. Nella prima fase si deve caricare il file di template all'interno dell'ambiente di esecuzione tramite la classe `loader` e poi successivamente lo si richiama tramite il metodo `get_template()`.
2. Nella seconda fase si istanziano le variabili di template con i valori corrispondenti.
3. Nella terza fase, infine, si fa la resa del template tramite il metodo `render()`, a cui si devono passare in input le variabili istanziate nella fase precedente. Si devono passare tutte le variabili di cui il template fa uso, come in un dizionario: le chiavi sono i nomi delle variabili, mentre i valori possono essere di qualsiasi tipo ma solitamente sono oggetti poiché hanno una capacità espressiva più elevata dei tipi base e consentono di ridurre il numero di variabili da dover passare. Il risultato della terza fase è la stringa che si ottiene dalla sostituzione delle variabili con i valori effettivi, la quale può essere salvata in un file. Questo processo prendere il nome di rendering.

In Jinja2, infine, sono presenti alcuni dei costrutti tipici dei linguaggi di programmazione, come ad esempio l'`if`, il ciclo `for` e le funzioni che prendono il nome di macro.

4.1.4 Arduino-cli

Arduino-cli, come ho specificato nel capitolo 3, è un'interfaccia a riga di comando sviluppata da Arduino per i sistemi embedded. E' un container di comandi e per ognuno di loro è dedicata una pagina di documentazione, la quale può essere visionata tramite il comando `help`. Supporta la maggior parte delle tipologie di microcontrollori e delle librerie a loro dedicate. E' la controparte a riga di comando dell'IDE di Arduino,

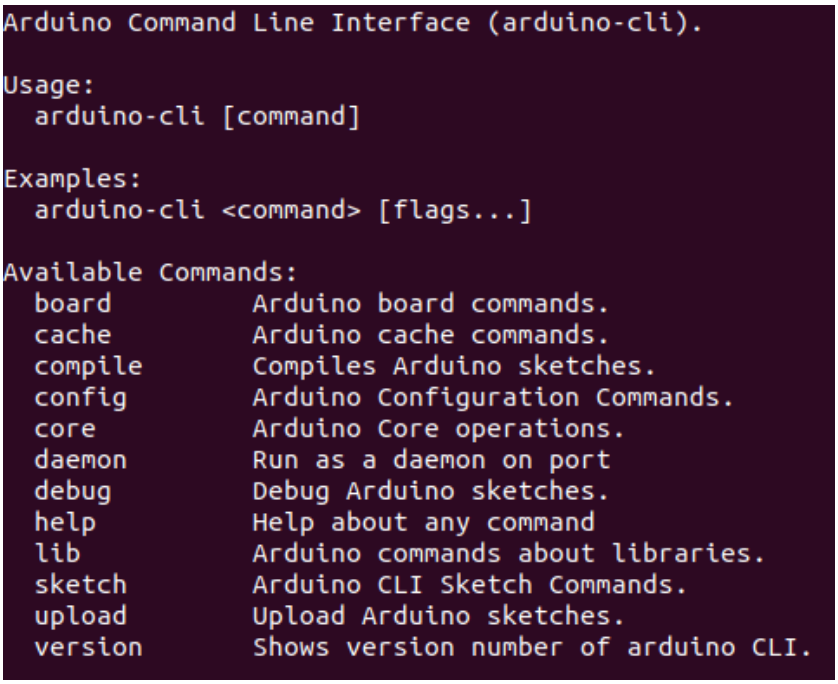


Figura 4.1: Interfaccia di arduino-cli

I comandi principali che mette a disposizione dell'utente sono:

- `init`: inizializza l'ambiente di Arduino e genera il file di configurazione *arduino-cli.yaml*
- `compile`: effettua la compilazione dello sketch nella board specificata.
- `upload`: effettua il flashing dello sketch nella board connessa alla porta specificata.

- `core`: permette di visualizzare i core (driver) per i microcontrollori installati e di installarne di nuovi. Questo comando svolge una funzione molto importante, infatti è possibile compilare e caricare uno sketch in uno specifico sistema solo se si posseggono i relativi driver.
- `lib`: gestisce le librerie di Arduino, consente di installare nuove librerie e di visualizzare la lista di quelle installate
- `board`: visualizza i tipi di board e su quali porte sono connesse e la lista di tutte le board compatibili con l'interfaccia

4.1.5 ESP8266WebServer

ESP8266WebServer è la libreria compatibile con la tipologia di board NodeMCU, impiegata per implementare il Web Server.

Per definire un Web Server, innanzitutto, è necessario stabilire il numero della porta sulla quale esporlo ed i vari endpoint sui quali rimanere in attesa di richieste da parte dei client. Un endpoint viene definito tramite il metodo `on(endpoint, tipo_di_richiesta, handler)`, il quale prende in input tre parametri: l'URL della richiesta su cui rimanere in attesa, il tipo di richiesta, ad esempio `HTTP_GET` o `HTTP_POST`, e l'handler. L'handler è una funzione che viene invocata per gestire la richiesta e al suo interno deve essere presente il messaggio di risposta, che viene inviato al client tramite il metodo `send(codice_http, tipo_risposta, risposta)`. Il primo parametro costituisce il codice HTTP della risposta, ad esempio `200` per indicare il sucesso o `400` per indicare un errore, il secondo rappresenta il tipo della risposta (content-type) che può essere `application/json` o `text/plain` ad esempio e infine, il terzo parametro contiene il messaggio di risposta che viene inviato al client.

Definiti questi parametri, è possibile far avviare il server e connetterlo ad Internet in modo da poter gestire tutte le richieste definite precedentemente. Per avviare un server si utilizza il metodo `begin()`, mentre per gestire le richieste si utilizza il metodo `handleClient()`.

La libreria ESP8266WebServer si base sul protocollo HTTP al livello applicazione e sul protocollo TCP/IP al livello trasporto.

4.1.6 WebSockets

WebSockets è una libreria per microcontrollori che permette, come lo si evince dal nome, di gestire canali di comunicazione sfruttando il protocollo WebSocket, sia lato server che lato client.

Come per il Server Web, anche per il Server WebSocket occorre definire la porta sulla quale esporlo, ma qui non abbiamo la gestione delle richieste GET o POST come avviene per il Web Server, poiché al loro posto si rimane in ascolto di eventi.

Gli eventi sono dei messaggi che vengono intercettati dal server e processati in base al loro tipo. Esistono diversi tipi di messaggi WebSocket, tra i principali abbiamo:

- `WStype_CONNECTED`: questo messaggio viene inviato dal client la prima volta che si connette al server WebSocket. Nel momento in cui verrà ricevuto dal server, esso aprirà il canale WebSocket e lo terrà aperto finché il client si disconnette o cade la connessione.
- `WStype_DISCONNECTED`: questo messaggio viene ricevuto dal server nel caso in cui un client, con cui ha aperto un canale di comunicazione, si dovesse disconnettere.
- `WStype_TEXT`: questo tipo è il più diffusso, infatti identifica tutti quei messaggi che vengono scambiati tra client e server durante le loro interazioni.
- `WStype_PING` e `WStype_PONG` corrispondono rispettivamente ad un messaggio di PING e di PONG. Solitamente un messaggio di PING viene inviato automaticamente dal mittente prima di un qualsiasi scambio di messaggi con il destinatario oppure viene inviato dal client ogni tot intervalli di tempo per mantenere il canale attivo.

Per avviare un server WebSocket si utilizza il metodo `begin()`, come avviene per il server Web e successivamente si definisce la funzione che intercetta e processa

gli eventi attraverso il comando `onEvent(funzione_gestione_eventi)` e lo si fa mettere in ascolto di eventi tramite il metodo `loop()`. La funzione di gestione degli eventi viene chiamata ogniqualvolta si riceve un messaggio e al suo interno è definito un costrutto `switch-case`, il quale intercetta il tipo di messaggio, si sposta nel ramo `case` corrispondente e infine lo processa. Per memorizzare l'elenco dei client connessi, il server mantiene una tabella in cui associa all'IP di ogni client un id numerico. Questo id è progressivo e viene incrementato mano a mano che si aggiungono client.

Il server può rispondere al client inviandogli un messaggio attraverso il metodo `sendTXT(num, messaggio)`, dove `num` è l'id del client. Il messaggio può essere solo di tipo stringa.

4.1.7 ArduinoJson

ArduinoJson è la libreria che consente ai sistemi embedded di parsare dati in formato JSON. Ho dovuto utilizzare questa libreria poiché la Thing Description e tutti i messaggi scambiati tra Thing e Consumers sono di questo formato.

Tramite questa libreria si possono definire oggetti e array di tipo JSON, solo però a partire da un documento JSON, il quale deve essere dichiarato precedentemente e che corrisponde ad un dictionary di Python.

Sono forniti anche metodi per la serializzazione di documenti JSON in formato stringa e deserializzazione di stringhe in documenti JSON.

4.2 Moduli

Per l'implementazione del sistema Embedded WoT Servient sono stati realizzati due moduli: il primo contiene la definizione della Thing CLI, mentre il secondo è costituito dal file di template per generare il codice in esecuzione sulla Thing.

4.2.1 Thing CLI

La Thing CLI mette a disposizione dell'utente quattro comandi, ognuno dei quali implementa una funzione specifica: `start`, `build`, `compile` e `flash`.

```
Usage: buildThing.py [OPTIONS] COMMAND [ARGS]...

  WoT module for build TDs and executable scripts for embedded systems

Options:
  --help  Show this message and exit.

Commands:
  build    Build executable Embedded-C File
  compile  Compile Embedded-C File
  flash    Flash Embedded-C File
  start    Start wizard
```

Frammento 4.1: Interfaccia Thing CLI

Attraverso il comando `start`, l'utente può avviare la procedura guidata di definizione della TD e di tutte le informazioni necessarie al file di template per generare il codice del file di scripting. Come prima informazione l'utente dovrà inserire il nome della Thing, il quale poi verrà visualizzato nell'URL della home del server web. Successivamente dovrà inserire sia i meta-dati globali della Thing, sia quelli specifici di ogni Interaction Affordance. I meta-dati più importanti sono quelli contenuti nell'array `forms`, in cui vanno inseriti: l'endpoint dell'interaction, il tipo di contenuto della richiesta (per tutte le richieste è `application/json`) e il tipo di operazioni supportate per quella richiesta. Il tipo di operazioni supportare variano da risorsa a risorsa, ad esempio per una proprietà è possibile leggere e scrivere il suo contenuto specificando, rispettivamente, il meta-dato `readproperty` e `writeproperty`, per un'azione è possibile invocarla con `invokeaction`, mentre per un evento è possibile sottoscriversi con `subscribeevent` e cancellarsi con `unsubscribeevent`.

Per quanto riguarda le proprietà, è necessario specificarne il tipo. I tipi supportati sono: `boolean`, `integer`, `number`, `string`, `null`, `array` e `object`. Lo stesso vale anche per i parametri in input e l'output delle funzioni invocate dalle azioni

e per i campi degli schemi degli eventi (sottoscrizione, dati e cancellazione). Ogni tipo poi ha associati dei sotto-campi specifici, ad esempio il valore massimo e minimo di un array, le proprietà di un oggetto, il tipo degli elementi di un array ecc. A questo scopo è stata implementata una funzione che prende in input il tipo base e dinamicamente permette all'utente di inserire i relativi sotto-campi.

Sono stati implementati dei tipi di dato personalizzati per far fronte ai tipi specifici del formato JSON, come ad esempio stringhe il cui primo carattere non sia un numero, interi non negativi, interi diversi da zero, oggetti JSON e così via. Per definire un tipo personalizzato utilizzando la libreria Click, è necessario ridefinire il metodo `convert()` della classe `click.ParamType`, il quale viene invocato dalla libreria per verificare se un dato ha o meno quel tipo specificato, se non lo ha si restituisce un errore. In questo caso l'utente dovrà ri-inserire un nuovo dato del tipo corretto.

```
class StartWithoutNumberStringParamType(click.ParamType):
    name = 'string'

    def convert(self, value, param, ctx):
        try:
            if(value[0].isdigit() == True):
                raise ValueError
            else:
                return str(value)
        except TypeError:
            self.fail('Expected string, got {a} of type {b}\n'.format(a=value, b=type(value).__name__), param, ctx)
        except ValueError:
            self.fail('This Element MUST start with a character, not with a number\n', param, ctx)
```

Frammento 4.2: Esempio di definizione di un tipo di dato personalizzato per la libreria Click

Per quanto riguarda le funzioni associate alle azioni, sono stati implementati due metodi di inserimento del loro codice. Il primo è il più immediato e consiste nel far inserire all'utente le istruzioni, una dopo l'altra, in un'unica riga della CLI, mentre il secondo consiste nel caricare un file in Embedded-C dove precedentemente sono state definite una o più funzioni e inserendo il nome di quella da associare all'azione, viene prelevato il suo codice e viene assegnato automaticamente all'azione in oggetto.

Terminato l'inserimento di tutti i meta-dati delle TD si chiede all'utente se vuole generare il file di scripting o terminare l'interazione con l'interfaccia. In entrambi in casi, comunque, viene generato un file JSON contenente la TD appena definita. I meta-dati sono memorizzati in un dictionary di Python incluso nel contesto (`ctx`) dell'interfaccia Click e dato che il contesto non si perde tra un comando e l'altro ma viene passato ad ognuno di loro, i dati memorizzati al suo interno possono essere acceduti in qualunque momento.

Nel caso in cui l'utente continui l'interazione con la CLI, gli verrà chiesto di inserire le informazioni necessarie al template engine per generare il file di scripting. Se l'utente, invece, avesse già a disposizione una TD memorizzata in un file JSON, potrebbe saltare la procedura precedente e attraverso il comando `build` darlo in input alla CLI, la quale estrarrà i dati e li memorizzerà nel dictionary corrispondente.

Le ultime due operazione che l'utente può svolgere sono la compilazione e il flashing del file di scripting sulla board NodeMCU tramite i comandi `compile` e `flash`. Per esporre queste due funzionalità viene sfruttata l'interfaccia arduino-cli, i cui comandi vengono eseguiti in background in modo trasparente per l'utente. La procedura di interazione con arduino-cli si articola nel modo seguente:

1. In primo luogo si verifica se l'interfaccia arduino-cli è disponibile ed in caso contrario si procede all'installazione dell'ultima versione rilasciata.
2. In secondo luogo si controlla se sono presenti i driver per il NodeMCU ed in caso negativo si prosegue come sopra.
3. Come terzo punto si chiede all'utente di installare le librerie aggiuntive, rispetto a quelle di default (ESP8266WebServer, WebSockets e ArduinoJson), che ha utilizzato nel file di scripting.
4. Infine si procede alla compilazione e/o al flashing del codice. Importante: l'operazione di flashing necessita che la board sia connessa tramite una porta seriale.

4.2.2 Template

Il file di template prende in input due variabili, più precisamente due dictionary: il primo contiene le informazioni della TD, mentre il secondo include quelle relative alle librerie presenti nel file di scripting.

Il file di template segue la struttura degli sketch scritti attraverso l'IDE di Arduino, quindi all'inizio si includono le librerie e si definiscono le variabili globali, poi si implementano i due metodi principali di un programma per Arduino, `setup()` e `loop()` e infine si definiscono le funzioni.

Prima di includere le librerie, si effettua una fase di pre-processing sui dati contenuti nella TD che consiste nell'andare a modificare gli endpoint di ogni Interaction Affordance per farli essere dinamici in base all'indirizzo IP che viene assegnato al server web nel momento in cui si connette alla rete.

Terminata questa fase, si procede all'inclusione, prima delle librerie di default, poi di quelle aggiunte dall'utente, le quali, essendo memorizzate in un array, vengono inserite una per una tramite un ciclo `for`. Successivamente si definiscono le variabili relative al ssid e alla password della rete Wi-Fi a cui il server si dorvrà connettere, ai tipi di protocolli utilizzati (http e ws), ai numeri delle porte su cui esporre il server web e WebSockets, al nome attribuito alla Thing e i documenti JSON per gestire le proprietà di tipo complesso (oggetto o array) e gli eventi WebSocket.

Le proprietà, le azioni e gli eventi, come avviene per le librerie, sono memorizzati all'interno di array così da avere la possibilità di iterare tra i loro elementi tramite cicli `for`. Per ogni proprietà si definisce il nome, il tipo e il valore iniziale. Per le azioni, il nome, il numero di input e lo schema per ognuno di loro. Per gli eventi, infine, si dichiara il nome ed i vari schemi di sottoscrizione, notifica e cancellazione.

A questo punto si dichiarano gli endpoint delle varie richieste su cui sia il Server Web che WebSocket si mettono in ascolto e si definiscono anche gli oggetti corrispondenti ai due server.

Con l'inserimento delle richieste e degli oggetti server termina la definizione delle variabili e si passa alla compilazione dei metodi `setup()` e `loop()`.

Il codice del `setup()` viene eseguito solamente una volta dal microcontrollore e come prima istruzione. Nel `setup()` si connette il server ad Internet, si dichiara la variabile di tipo stringa a cui viene assegnata la TD, la quale verrà esposta sulla home del server, si configura il server per la gestione degli endpoint tramite il metodo `on()` e alla fine si avviano i due server.

Il codice del `loop()` invece viene eseguito di continuo dalla board e contiene, di default, solamente due istruzioni: la chiamata al metodo `handleClient()`, in modo tale che il server web possa gestire tutte le richieste in arrivo dai client e al metodo `loop()`, affinché anche il server WebSocket possa fare altrettanto.

All'interno dei metodi `setup()` e `loop()`, l'utente può inserire istruzioni aggiuntive tramite la Thing CLI.

Terminata questa fase, si passa alla dichiarazione degli handler per gestire le richieste e delle funzioni associate alla azioni. Possono essere presenti ulteriori funzioni aggiunte dall'utente tramite la Thing CLI a completamento o a supporto delle funzionalità della Thing. Tra le funzioni, poi, ne è presente una, `handleInputType()`, che ha il compito di verificare la correttezza del contenuto delle richieste POST per invocare le azioni. Controlla se, per ogni parametro da passare in input all'azione, il valore corrispondente presente nel messaggio di richiesta del client ha il tipo corretto. Se la verifica non va a buon fine, l'azione non viene invoacata. Gli input delle azioni vengono inviati dal client all'interno del corpo della richiesta e non come parametri nell'URL.

Come ultimo passaggio, si definisce la funzione di gestione dei messaggi WebSocket.

Ovviamente, se l'utente decide di non inserire nessuna proprietà, nessuna azione o nessun evento, la parte corrispondente non viene generata nel file di scripting.

In conclusione, per le Interaction Affordances è stato supportato sia il protocollo HTTP, sia il protocollo WebSocket, tranne che per gli eventi, nel confronto dei quali non è stato possibile implementare il protocollo HTTP di longpolling poiché le tecnologie a disposizione non lo hanno permesso.

Capitolo 5

Validazione

In questo capitolo verrà presentato il caso d'uso dello smart parking, sul quale è stata validata la piattaforma dell'Embedded WoT Servient descritta nei capitoli precedenti. Inizialmente verrà descritto lo scenario e le tecnologie impiegate e successivamente si porrà l'attenzione sulla sua implementazione concreta sviluppata tramite il WoT Servient.

5.1 Caso d'uso: Smart Parking

Lo smart parking è l'utilizzo della tecnologia e dell'Internet of Things per individuare quali spazi sono occupati e quali disponibili in un parcheggio, col fine di creare una mappa dei parcheggi in tempo reale. Esso si colloca all'interno del concetto della Smart City.

Gli obiettivi dello smart parking sono:

- aiutare i guidatori a trovare parcheggio facilmente e velocemente (es. tramite mobile apps)
- mettere a disposizione informazioni che aiutano i vigili o altri incaricati a identificare le eventuali violazioni
- aiutare le persone a scegliere mezzi di trasporto alternativi nel caso in cui i parcheggi siano esauriti

- ridurre la congestione del traffico
- migliorare le condizioni ambientali: emissione di gas tossici e qualità dell'aria

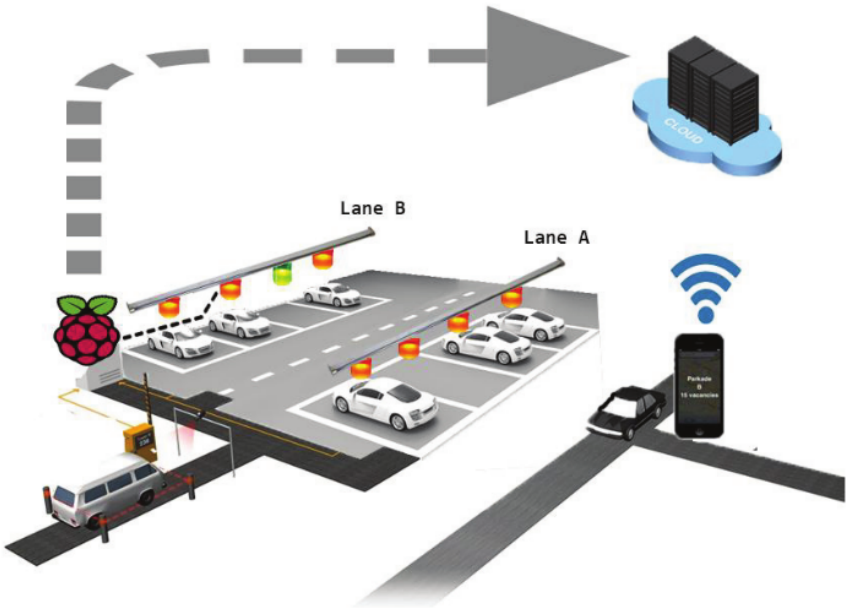


Figura 5.1: Smart Parking

L'architettura dello smart parking è composta da quattro elementi: sensori, protocolli di rete, sistemi software e piattaforme Cloud.

L'obiettivo dei sensori è quello di verificare che un'area di parcheggio sia disponibile oppure no e comunicarlo al sistema di parcheggio tramite un gateway, in modo tale che il guidatore ne sia notificato. Esistono diverse tipologie di sensori, tra i quali, i più importanti sono: le videocamere, gli smartphone, i sensori ultrasonici, ad infrarossi, radar, magnetometri e geomagnetici.

I protocolli di rete rappresentano lo strumento per trasferire le informazioni raccolte dai sensori al sistema di parcheggio. Nello smart parking esistono due tipologie di protocolli di rete, una è rivolta agli utenti e l'altra viene utilizzata dai sensori (in alcuni casi se ne ha soltanto una utilizzata da entrambi).

L'architettura di rete deve essere in grado di supportare i centinaia di migliaia di utenti che potenzialmente possono accedere alla rete e deve rispettare le

seguenti caratteristiche: basso consumo di energia, minimo ritardo e affidabilità nella trasmissione dei dati. I principali protocolli di rete sono: protocolli wireless per IoT (LPWAN e LR-WPAN come LoRa, NB-IoT e ZigBee), il WiFi, la tecnologia Bluetooth, i protocolli cellulari (3G/4G) e cablati (Ethernet, USB e comunicazione seriale).

La componente rappresentata dai sistemi software è di vitale importanza poiché consente di manipolare le informazioni ricevute dai sensori. Le soluzioni software non devono svolgere soltanto il compito di immagazzinamento dei dati ma deve soprattutto fornire dei servizi che possano aiutare gli utenti a risparmiare carburante, tempo e denaro.

Le principali funzioni svolte da un sistema software devono essere:

- verifica della disponibilità di un parcheggio (libero, occupato, prenotato)
- prenotazione di un parcheggio
- pagamento della sosta
- comunicazione eventuali violazioni
- prolungamento della sosta

Non meno importanti sono le fasi di analisi e predizione dei dati, attraverso le quali gli sviluppatori del sistema di parcheggio possono verificare l'uso che ne fanno gli automobilisti, identificare quali sono i loro feedback e se è necessario, modificare o aggiungere funzionalità per migliorare l'interazione con l'utente finale.

Il modo migliore per immagazzinare e analizzare l'enorme quantità di informazioni raccolte dal sistema software è tramite le piattaforme cloud.

Le piattaforme cloud permettono di centralizzare le informazioni provenienti da diversi provider di parcheggi così da veicolare una richiesta da parte di un utente a tutti i provider connessi. Il cloud agisce anche da intermediario tra i sensori e le applicazioni.

I fattori che contribuiscono all'uso delle piattaforme cloud sono: la capacità di storage, la capacità di calcolo, la capacità di comunicazione (di scambio dei dati tra le varie piattaforme), la scalabilità, la disponibilità (i dati memorizzati su cloud sono sempre disponibili) e l'interoperabilità.

5.2 Implementazione del caso d'uso

Per implementare il caso d'uso dello smart parking sono stati impiegati tre tipi di sensori: l'HC-SR04 a ultrasuoni, l'Adafruit VL53L0X e lo Sharp IR a infrarossi. Tutti e tre sono sensori di distanza e tramite l'impostazione di una soglia in centimetri, verificano se un'automobile la oltrepassa oppure no. Se un'automobile la oltrepassa significa che è parcheggiata ed il parcheggio corrispondente risulterà occupato, mentre se nessuna macchina oltrepassa la soglia, il parcheggio risulterà libero.

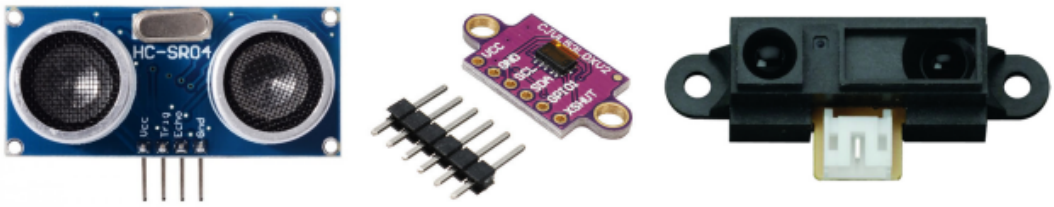


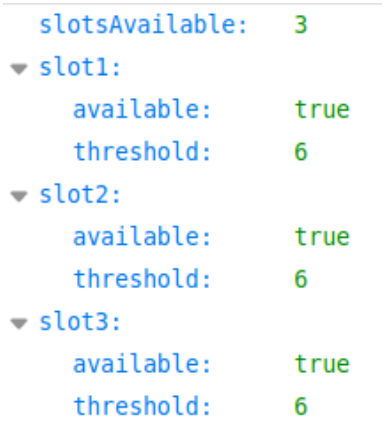
Figura 5.2: I tre sensori, da sinistra verso destra: l'HC-SR04, l'Adafruit VL53L0X e lo Sharp IR

La logica dell'applicazione è stata suddivisa in due parti: la prima parte si occupa della gestione dello smart parking ed interagisce con i sensori, mentre la seconda parte si occupa dell'esposizione della Thing sulla rete Internet. Le due parti vengono eseguite da due microcontrollori differenti, per la prima è stato utilizzato un Arduino UNO, mentre per la seconda un NodeMCU. I motivi di questa scelta sono principalmente due: il primo è dovuto al fatto che i tre sensori supportano una tensione di 5V disponibile solo sul modello Arduino UNO e il secondo è che solamente il NodeMCU ha il modulo Wi-Fi integrato per la connessione ad Internet, indispensabile per esporre una Thing.

Ogni sensore gestisce un parcheggio e verifica se è libero oppure se è occupato da un'automobile. Lo stato di un parcheggio viene visualizzato sia in un display LCD posizionato all'entrata del parcheggio (A -> disponibile, O -> occupato), sia nella

Thing Description esposta dalla Thing. La logica della Thing è stata implementata tramite il sistema Embedded WoT Servient oggetto di questo lavoro di tesi.

Nella TD sono state definite quattro proprietà di tipo oggetto, la prima corrisponde al numero di slot disponibili, mentre le restanti tre identificano i tre parcheggi. Ognuna di queste tre proprietà ha due campi: lo stato del parcheggio, un booleano che è `true` se il parcheggio è disponibile ed è `false` nel caso opposto e la soglia del sensore identificata da un numero intero. Ogni sensore ha un diverso range di misurazione, per cui le soglie hanno dei valori massimi e minimi che vanno rispettati. Oltre a queste proprietà sono state aggiunte tre azioni, le quali permettono all'utente la modifica delle soglie dei vari parcheggi.



Frammento 5.1: Elenco delle proprietà definite nella TD esposta dalla Thing

Le comunicazioni tra l'Arduino UNO e il NodeMCU avvengono tramite seriale e hanno luogo in entrambe le direzioni, cioè dall'Arduino UNO al NodeMCU e viceversa. L'Arduino UNO comunica al NodeMCU il cambio di stato di un parcheggio e le soglie iniziali, mentre il NodeMCU comunica all'Arduino le eventuali modifiche delle soglie da parte dell'utente.

5.2.1 Grafici

Per calcolare l'accuratezza dei tre sensori sono stati effettuati due tipi di test:

1. Nel primo test si è impostato un valore di soglia pari a 6 per tutti e tre i sensori e sono state condotte 10 simulazioni per ogni parcheggio. Per ogni simulazione si è verificato il corretto funzionamento del sensore.

2. Nel secondo test si sono variate le soglie dei sensori e si sono eseguite 10 simulazioni su ogni parcheggio, per ogni diverso valore della soglia. I valori di soglia scelti sono stati: 5, 10, 20.

I risultati dei due test sono stati raccolti nei seguenti grafici. I valori sono in percentuale.

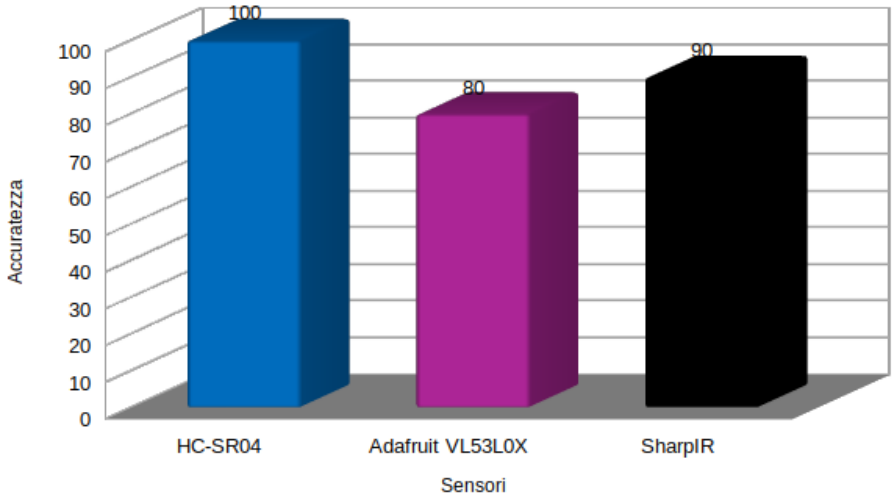


Figura 5.3: Validazione-Test 1

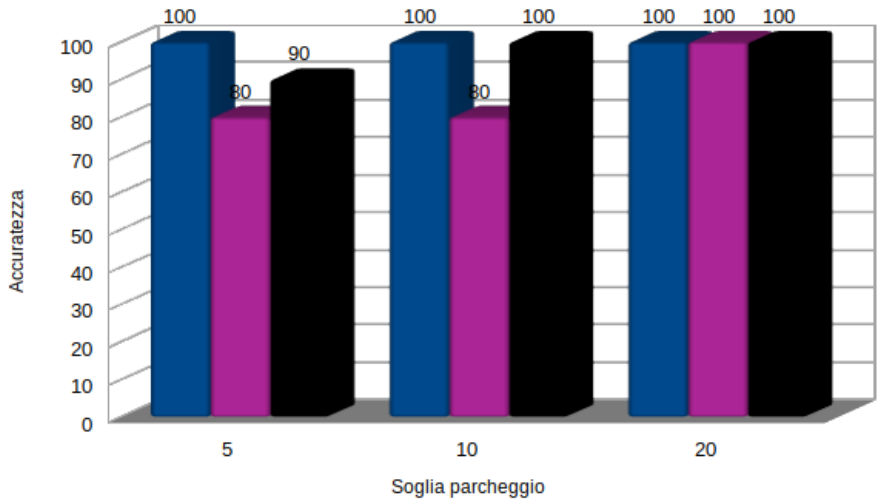


Figura 5.4: Validazione-Test 2

Come si può notare dal primo grafico, il sensore HC-SR04 è il più accurato, infatti ha restituito sempre il valore corretto, al secondo posto si posiziona lo Sharp IR con un valore errato, mentre al terzo l'Adafruit VL53L0X con ben due valori errati. Dal secondo grafico, invece, si evince che al crescere del valore di soglia, l'accuratezza dei sensori aumenta.

Conclusioni

In questo elaborato è stata presentata la realizzazione della piattaforma Embedded WoT Servient per l'esposizione di Thing, basate sullo standard del W3C, su sistemi embedded. In primo luogo è stata necessaria una fase preliminare di studio degli aspetti tecnologici relativi al mondo dell'IoT e del problema dell'interoperabilità, per poi passare ad un'analisi dettagliata dell'architettura del WoT proposta dal W3C. A ciò è seguita la fase di progettazione ed implementazione dei vari componenti che costituiscono il sistema nella sua interezza.

Il lavoro svolto si è rivelato molto lungo e impegnativo, infatti si trattava di realizzare un progetto molto complesso, in cui si voleva far approdare l'articolata architettura del WoT Servient, composta da un elevato numero di funzionalità avanzate, nel mondo dei sistemi embedded, dove il concetto che risuona più spesso è la limitatezza di risorse e di capacità computazionali. Oltre a questo, si voleva proporre un sistema altamente intuitivo e facilmente fruibile, con il quale l'utente potesse interagire con il minor numero di passaggi possibile.

Per quanto riguarda il prossimo futuro, l'intenzione è quella di passare da un'interfaccia a riga di comando ad un'interfaccia grafica, sicuramente più intuitiva e accessibile e di fornire il supporto ad ulteriori tipi di microcontrollori.

Il lavoro non deve essersi considerato definitivamente concluso, infatti sarà necessario mantenerlo costantemente allineato con l'evoluzione dei documenti e degli standard del W3C. Infatti capita spesso che vengono effettuate modifiche sostanziali all'architettura tra una versione e l'altra degli standard.

Bibliografia

[1] Satista. *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025* (2016). [ultima visita 07.03.2020]. URL: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`.

[2] Shancang Li, Li Da Xu, Shanshan Zhao, *5G Internet of Things: A Survey*. In: Journal of Industrial Information Integration, Volume 10. Pagine 1-9. ISSN 2452-414X (2018).

[3] Shancang Li, Li Da Xu, Shanshan Zhao, *The internet of things: a survey*. In: Information Systems Frontiers. Pagine. 243-256 (2015).

[4] Radatz J., Geraci A., Katki F. *IEEE standard glossary of software engineering terminology*. In: IEEE Std 610.12-1990 (1990).

[5] Noura M., Atiquzzaman M., Gaedke M. *Interoperability in Internet of Things: Taxonomies and Open Challenges*. In: Mobile Networks and Applications 24, 796−809 (2019).

[6] Manyika J., Chui M., Bisson P., Woetzel J., Dobbs R., Bughin J., Aharon D. *The internet of things: mapping the value beyond the hype*. In: McKinsey global institute. McKinsey Glob Inst 3 (2015).

[7] W3C. *About W3C* (2020). [ultima vista 24.02.2020]. URL: `https://www.w3.org/Consortium/`.

[8] W3C. *Web of Things Working Group* (2020). [ultima visita 24.02.2020]. URL: https://www.w3.org/WoT/WG/.

[9] W3C. *Web of Things (WoT) Architecture* (2020). [ultima visita 26.02.2020]. URL: https://www.w3.org/TR/wot-architecture/.

[10] W3C. *Web of Things (WoT) Thing Description* (2020). [ultima visita 26.02.2020]. URL: https://www.w3.org/TR/wot-thing-description/.

[11] Tim Berners-Lee. *Linked Data Design Issues* (2006). [ultima visita 26.02.2020]. URL: https://www.w3.org/DesignIssues/LinkedData.html.

[12] W3C. *JSON-LD 1.1* (2019). [ultima visita 26.02.2020]. URL: https://www.w3.org/TR/2019/CR-json-ld11-20191212/.

[13] W3C. *Web of Things (WoT) Binding Templates* (2020). [ultima visita 26.02.2020]. URL: https://www.w3.org/TR/wot-binding-templates/.

[14] W3C. *Web of Things (WoT) Scripting API* (2019). [ultima visita 26.02.2020]. URL: https://www.w3.org/TR/wot-scripting-api/

[15] W3C. *Web of Things (WoT) Security and Privacy Guidelines* (2019). [ultima visita 26.02.2020]. URL: https://www.w3.org/TR/wot-security/.

[16] W3C. *Web of Things Interest Group* (2020). [ultima visita 27.02.2020]. URL: https://www.w3.org/WoT/IG/.

[17] W3C. *Web of Things Implementations* (2020). [ultima visita 27.02.2020]. URL: https://www.w3.org/WoT/IG/wiki/Implementations.

[18] GitHub. *Web of Things Framework for Arduino* (2016). [ultima visita 27.02.2020]. URL: https://github.com/w3c/wot-arduino.

[19] GitHub. *Arduino RDF Server* (2017). [ultima visita 27.02.2020]. URL: https://github.com/ucbl/arduinoRdfServer.

Ringraziamenti