ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
CORSO DI LAUREA MAGISTRALE IN
INGEGNERIA E SCIENZE INFORMATICHE

PROGETTAZIONE E REALIZZAZIONE DI UNO SMART DISTRIBUTED SYSTEM PER IL CONTROLLO DEGLI ACCESSI IN AMBITO AZIENDALE

Tesi in

SMART CITY E TECNOLOGIE MOBILI

Relatore

Chiar.mo Prof. DARIO MAIO

Correlatore

Dott. PIETRO EVANGELISTI

Presentata da

ANDREA CORZANI

Sessione II
Anno Accademico 2013/2014

A Giulia

Indice

Glossario

Stack applicativo Una piattaforma software per lo sviluppo composta da un insieme eterogeneo di applicazioni, ognuna con caratteristiche e funzionalità diverse.

Throughput Si intende per throughput di un canale di comunicazione la sua capacità di trasmissione effettivamente utilizzata.

WebSocket WebSocket è una tecnologia web per la creazione di canali di comunicazione bidirezionali attraverso una singola connessione TCP. WebSocket è progettato per essere implementato tra un browser (client) ed un server, ma più in generale può essere utilizzato anche da qualsiasi genere di applicazione client-server.

Sniffing Attività di intercettazione passiva dei dati che transitano in una rete telematica. Tale attività può essere svolta sia per scopi legittimi (ad esempio l'analisi e l'individuazione di problemi di comunicazione o di tentativi di intrusione) sia per scopi illeciti (intercettazione fraudolenta di password o altre informazioni sensibili).

Handler In programmazione, si tratta di funzioni asincrone (*callback-handler*)

registrate dall'utente per gestire determinati eventi (*event-handler*) generati dal programma, oppure per gestire interrupt di sistema operativo (*interrupt-handler*.

Adapter Con il nome adapter, o adattatore si denota un design pattern utilizzato in informatica nella programmazione orientata agli oggetti. A volte viene chiamato wrapper (ovvero involucro) per il suo schema di funzionamento. Il fine dell'adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti.

DBMS In informatica, un Database Management System, abbreviato in DBMS o Sistema di gestione di basi di dati è un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database (collezioni di dati strutturati). Spesso è definito come "gestore o motore del database" ed è ospitato su un'architettura hardware dedicata oppure su un semplice computer.

HTTP L'HyperText Transfer Protocol (HTTP) è usato come principale sistema per la trasmissione d'informazioni sul web in un'architettura tipica client-server. Un server HTTP generalmente resta in ascolto delle richieste dei client sulla porta 80 usando il protocollo TCP a livello di trasporto. Il client invia un messaggio di richiesta composto da tre parti:

- riga, che include metodo (GET, POST, PUT, DELETE, etc.) e URI (identificativo di risorsa);
- body, il corpo vero e proprio del messaggio che il client potrebbe voler inviare al server;
- header, all'interno della quale sono incluse altre informazioni aggiuntive (cookie, user-agent, etc);

Token Un token per la sicurezza (chiamato anche token hardware, token per l'autenticazione, token crittografico, o semplicemente token) è un "dispositivo" necessario per effettuare un'autenticazione. Può essere di tipo fisico (hardware) oppure di tipo software, ove le informazioni necessarie risiedono direttamente nel computer dell'utente, e non in un oggetto esterno. È questo il caso di un token JSON, una stringa che il client riceve dal server dopo il login, ed utilizza per provare la propria identità.

Deprecato In programmazione, una caratteristica di un programma in passato documentata e considerata ufficiale il cui uso è attualmente sconsigliato a favore di una versione più recente.

Payload Si intende in genere ciò che viene trasportato da un mezzo o servizio di trasporto, in opposizione a ciò che deve essere spostato solo per far funzionare il mezzo o servizio. In informatica è utilizzato per indicare la parte di un flusso di dati che rappresenta esclusivamente il contenuto informativo, e non le informazioni aggiuntive necessarie per il protocollo (*overhead*).

Chunk Una porzione di contenuto informativo codificata come insieme di byte. Solitamente più chunk sono ottenuti dividendo un *buffer* (stringa di byte), operazione talvolta necessaria per ragioni infrastrutturali.

OAuth OAuth è un protocollo di sicurezza aperto, che permette ad uno sviluppatore (previa autorizzazione) di poter sfruttare all'interno della propria applicazione i dati di utenti registrati su servizi esterni. OAuth garantisce ai service provider l'accesso da parte di terzi ai dati degli utenti, proteggendo contemporaneamente le loro credenziali.

Capitolo 1

Introduzione

L'avvento di *Internet of Things* sta progressivamente cambiando il modo attraverso il quale la tecnologia interviene nella vita di tutti i giorni: sempre più spesso infatti si parla di oggetti "intelligenti" capaci di generare contenuto informativo ed interagire con l'uomo o con il resto del mondo attraverso lo sfruttamento della rete internet e altre forme di connettività. Numerose sono le tematiche ICT che trovano applicazione in questo contesto: dai sistemi embedded alle tecnologie di comunicazione wireless, dalle tecniche per la gestione di *Big Data* ai sistemi distribuiti, dall'automazione industriale alla domotica.
Internet of Things rientra come protagonista nell'insieme di tecniche che guidano gli studi e la progettazione di *Smart City*, città del futuro che si pongono come obiettivi l'autosostenibilità ed il miglioramento della qualità di vita dei cittadini; in particolare la distribuzione efficiente dell'energia, i sistemi per il controllo intelligente dei consumi e i dispositivi per garantire la sicurezza degli utenti (rilevamento di intrusioni, controllo accessi) sono le fondamenta per lo sviluppo di *Smart Building*.

I concetti di Internet of Things e Smart Building sono direttamente coinvolti in questo lavoro di tesi, che ha come obiettivo primario lo sviluppo di un siste-

ma intelligente e distribuito per il controllo accessi e gestione presenze in ambito aziendale. Il controllo accessi è un'applicazione integrata nelle procedure interne a numerose organizzazioni, che fanno spesso uso di dispositivi più o meno intelligenti al fine di monitorare le presenze dei propri utenti e, al contempo, automatizzare la procedura di autorizzazione e apertura della porta.

Il lavoro s'inquadra nell'ambito del processo di timbratura e accesso che l'azienda SPOT Software intende adottare; il sistema utilizzato attualmente, oltre ad avere alcuni limiti dal punto di vista della sicurezza, risulta anche di difficile consultazione per l'amministrazione. È proprio in seguito a questo insieme di considerazioni che nasce l'idea di sviluppare una piattaforma distribuita: un applicativo web per la gestione e consultazione delle timbrature, un dispositivo embedded per l'apertura dell'ingresso e un'applicazione per smartphone al fine di consentire agli utenti di interagire con il dispositivo e completare l'accesso.
Grazie allo sviluppo del nodo centrale come piattaforma "cloud", anche altre aziende potranno fare uso del sistema distribuito; sia nella sua versione completa, che automatizza il controllo degli accessi attraverso l'implementazione, configurazione e installazione dello smart device, sia come semplice applicazione per l'inserimento manuale e la consultazione delle timbrature utente.

Obiettivo secondario, ma non meno importante, è dato dalla natura del progetto stesso. Toccare con mano tutti gli aspetti e le tecnologie coinvolte durante lo sviluppo di un sistema come questo, è un'opportunità per aumentare sia le competenze personali, sia, di riflesso, quelle aziendali, in tanti differenti settori chiave della moderna ICT.
Al fine di garantire una maggiore visibilità al progetto, e di renderlo disponibile ad ogni eventuale miglioramento, estensione o approfondimento, tutto il codice sviluppato sarà rilasciato come open source.

Lo sviluppo di questo elaborato è così organizzato: nel primo capitolo sono

introdotte le principali tematiche di IoT facente parti del progetto, dalle tecnologie di comunicazione wireless ai sistemi embedded. Il secondo capitolo segna l'inizio della fase di progettazione, che può essere suddivisa in due distinte parti. Nella prima (cap. 3) si introducono i macro requisiti del progetto e si analizza il problema attraverso una visione globale dei tre "nodi" coinvolti, in particolare per quel che riguarda le diverse interazioni tra essi. La seconda parte è composta dai capitoli 4, 5 e 6, ognuno dei quali è invece esclusivamente dedicato alla progettazione e realizzazione di uno specifico nodo: rispettivamente web application, smart device e applicazione android.

Al termine di questo elaborato sono infine descritti sommariamente i risultati raggiunti, con un accenno ai numerosi sviluppi futuri che potranno vedere coinvolto il progetto.

Capitolo 2

Internet of Things

In questo capitolo sono descritti gli aspetti principali delle tematiche coinvolte durante lo sviluppo del progetto. Inizialmente si introdurrà il paradigma Internet of Things, cercando di enfatizzare il ruolo primario che svolgono le nuove tecnologie nel contesto delle città intelligenti. Successivamente si approfondiranno in modo maggiormente dettagliato le moderne tecnologie di comunicazione NFC e Bluetooth Low Energy, assieme ai sistemi embedded, in considerazione del ruolo primario che giocano all'interno del sistema sviluppato in questo progetto. Parte dei contenuti riportati in questo capitolo derivano direttamente dal materiale di supporto al corso "Smart City e Tecnologie Mobili" [16].

2.1 Overview

Internet of Things (comunemente abbreviato IoT) è un termine coniato per la prima volta nel 1999 da Kevin Ashton, che lo introdusse per riferirsi ad una possibile evoluzione nell'utilizzo della rete Internet. Ashton fu tra i primi ad avere una semplice intuizione: gli uomini hanno tempo, attenzione e accuratezza limitati, di conseguenza se i computer avessero la possiblità di sapere e monitorare con

esattezza tutto ciò che avviene nel mondo circostante, si potrebbero facilmente ridurre gli sprechi, le perdite e i costi, sapendo in qualsiasi momento cosa serve, cosa è necessario fare, cosa è necessario riparare. Il tutto nel modo più efficiente possibile. La rete internet e le tecnologie di comunicazione wireless sono i principali mezzi attraverso i quali gli oggetti, identificabili in modo univoco, possono interagire con i computer inviando informazioni in modo autonomo e continuativo [3].
Circa quindici anni dopo si può affermare con certezza che l'intuizione di Ashton si è trasformata in uno scenario sempre più comune: tanti degli oggetti che ci circondano sono oggi in grado di elaborare informazioni e trasmetterle attraverso la rete internet, basti pensare agli smartphone, ai dispositivi indossabili, ma anche agli elettrodomestici "intelligenti" disponibili sul mercato.

Nell'IoT il concetto di *thing* è in generale applicabile a qualunque entità connessa ad internet ed identificabile in modo univoco, che è in grado di elaborare, memorizzare o trasmettere informazioni riguardanti se stessa o il mondo che la circonda. Grazie all'utilizzo di tecnologia embedded, questi oggetti sono in grado di elaborare dati e trasferirli attraverso Internet, senza la necessità di un'interazione uomo-uomo o uomo-computer. Internet of thing risulta quindi essere l'interconnessione tra l'infrastruttura esistente di Internet e dispositivi di computazione integrati negli oggetti.

La quantità di informazioni generate da questo modello è enorme, così come le potenzialità che ne derivano. Gli esseri umani da sempre utilizzano le informazioni a loro disposizione per prendere decisioni, per determinare il proprio comportamento, di conseguenza l'unione tra la mole di dati, la potenza computazionale elevata degli elaboratori moderni e la velocità di comunicazione delle infrastrutture disponibili, può essere fondamentale per migliorare e facilitare ogni tipologia di processo decisionale.

Anche il fatto che gli oggetti possano interagire tra loro senza necessità di intervento da parte dell'uomo è fondamentale per il miglioramento della vita di tutti i giorni. Gli esempi possono essere fatti su ogni campo, dal comune termostato che regola la temperatura sulla base delle persone presenti in casa, fino alla rete di rilevatori sottomarini che allertano autonomamente un'intera nazione in caso di intenso terremoto, al fine di prevenire i danni di un probabile Tsunami. L'azione autonoma svolta nel secondo scenario proposto avrebbe sicuramente un'efficacia diversa se il compito fosse affidato direttamente ad esseri umani, sia a causa della comprensibile latenza introdotta, sia a causa della più alta probabilità di commettere errori.

IoT può quindi avere un forte impatto sulla maggior parte degli aspetti delle nostre vite, dalla semplificazione delle attività quotidiane al risparmio energetico, dal miglioramento della relazione con l'ambiente alla sicurezza dei singoli individui.

2.1.1 Applicazioni e casi d'uso

Inizialmente il concetto di Internet of Things è stato principalmente associato a comunicazioni macchina-macchina (M2M) nell'industria manifatturiera e nelle industrie di petrolio e gas, anche se spesso il termine è usato anche per descrivere interazioni di tipo uomo-macchina (H2M) mediate da dispositivi come ad esempio uno smartphone e relativa applicazione. Le reti M2M si compongono di device con capacità autonome di *sensing* e di calcolo, capaci di rilevare eventi nell'ambiente circostante, misurare grandezze d'interesse e diffonderle attraverso reti IP. Sebbene M2M sia uno dei pilastri per IoT, non esistono al momento standard per i protocolli di comunicazione, anche se vi sono numerosi tentativi in corso. Questo è uno dei limiti che ne hanno ostacolato maggiormente lo sviluppo, in quanto

spesso i produttori fanno uso di protocolli proprietari che rendono di fatto i propri dispositivi incompatibili con il resto del mondo.

Smart tagging

IoT trova una delle sue principali applicazioni nell'etichettatura (*tagging*) intelligente, operazione attraverso la quale è possibile identificare in modo univoco oggetti o persone. Lo *smart tagging* è un fattore abilitante per tantissimi scenari IoT, basti pensare che se il riconoscimento delle persone è fondamentale per un sistema di controllo accessi, identificare e localizzare con esattezza oggetti ha un ruolo cruciale nelle applicazioni di logistica o gestione automatizzata di magazzino. Inoltre nell'Unione Europea sta acquisendo sempre più rilevanza la tracciabilità degli alimenti, possibile oggi grazie ad etichette intelligenti ricche di informazioni ed aggiornabili lungo tutta la filiera. Le tecnologie utilizzate per lo smart tagging vanno dal classico tag RFID al moderno discendente NFC, dal beacon Bluetooth LE al localizzatore GPS, senza trascurare l'evoluzione dei classici codici a barre tramite l'utilizzo di nuove codifiche.

Smart grid ed internet of energy

Le tematiche del risparmio energetico e dell'impatto ambientale sono tra i principali argomenti che ruotano attorno al mondo IoT. *Smart Grid* è il termine utilizzato per definire una rete elettrica dotata di sensori intelligenti che raccolgono informazioni in tempo reale ottimizzando la distribuzione di energia e minimizzando, al contempo, eventuali sovraccarichi e variazioni della tensione elettrica. Non si ha più così una rete passiva a controllo centralizzato, ma la distribuzione diventa attiva ed i diversi nodi sono coinvolti direttamente nella distribuzione del carico.

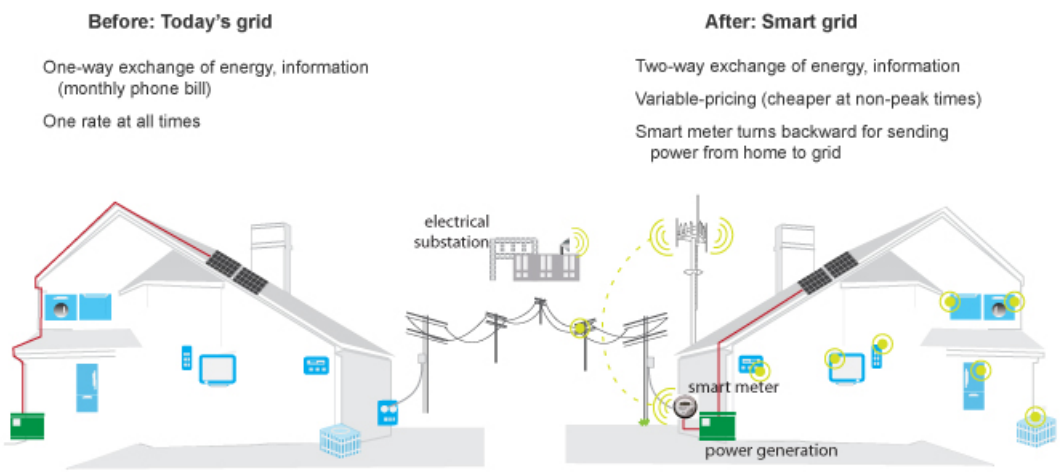

Figura 2.1: La differenza tra la distribuzione di energia passiva e la distribuzione attiva (smart grid)

Internet of Energy (IoE) è una particolare declinazione di Internet of Things e rappresenta la naturale evoluzione di Smart Grid. IoE si pone l'obiettivo di fornire un'unica interfaccia real-time per l'interazione tra reti Smart Grid e un sistema distribuito eterogeneo composto da tanti "oggetti" connessi ad Internet (veicoli elettrici, edifici, dispositivi elettrici), al fine di:

- ottimizzare la produzione di energia bilanciando il carico a seconda delle esigenze;
- ridurre l'impatto ambientale;
- ridurre gli sprechi;
- sfruttare fonti di energia rinnovabile.

Nel paradigma Internet of Energy un ruolo importantissimo lo hanno gli *smart meters*, "contatori" intelligenti che oltre a misurare i consumi di gas, acqua o elettricità, permettono una comunicazione bidirezionale a banda larga con la centrale. È proprio grazie a questa caratteristica che rendono possibile la regolazione del

carico di energia sulla base del consumo istantaneo e consentono la rilevazione remota di guasti e malfunzionamenti senza necessità di intervento sul posto. Per quanto concerne la comunicazione, spesso questi dispositivi fanno uso di *Power Line Communication* (PLC), tecnologia attraverso la quale è possibile trasmettere informazioni attraverso le linee elettriche.

Dallo smart building alla smart city

Smart building rappresenta un esempio significativo dell'applicazione dei concetti di IoT: edfici intelligenti che sfruttano varie tecnologie, dispositivi e sensori, allo scopo di migliorare la propria efficienza energetica, la sicurezza degli individui e il loro comfort. Gli smart building sono tipicamente dei sistemi di controllo distribuiti, in cui la rete di comunicazione tra i diversi dispositivi presenti ed installati, consente ad esempio di regolare la temperatura interna evitando sprechi, regolare l'intensità luminosa o controllare la qualità dell'aria. Oltre al monitoraggio interno dei consumi, una delle caratteristiche principali degli *smart buildings* è la dotazione di generatori di energia elettrica tramite fonti rinnovabili (pannelli solari, pale eoliche, generatori a biomasse) che può essere utilizzata sia per il sostenimento energetico dell'edificio stesso, sia per quello di altri nodi della smart grid qualora la produzione eccedesse il consumo interno.

In ambito domestico solitamente si parla di *Smart Home*, edifici intelligenti che privilegiano maggiormente gli aspetti legati al comfort domestico degli individui. Una smart home è un'abitazione ad alta efficienza energetica, dotata di impianti intelligenti per la regolazione dei consumi, di impianti rinnovabili per la produzione di energia e di smart meters per l'interazione con la smart grid.

Oltre alla sostenibilità energetica, un ulteriore aspetto che accomuna smart home e smart building è sicuramente legato alla sicurezza e al controllo degli accessi, fondamentale sia in ambito aziendale/pubblico, che in ambito domestico. Parte

del progetto svolto è orientato a questo genere di tematiche, focalizzando l'attenzione non tanto sulle questioni di sicurezza (il sistema non ha tra i suoi obiettivi l'*intrusion detection*) quanto sulle tematiche relative al monitoraggio delle risorse umane aziendali.

Smart building e smart home trovano naturale collocazione all'interno delle smart city, città "intelligenti" progettate ponendo l'attenzione su svariati aspetti interdisciplinari con il duplice obiettivo di migliorare la qualità di vita dei cittadini ed essere autosostenibili sia dal punto di vista economico, sia dal punto di vista energetico. Se nei primi anni in cui si è fatto uso del termine "smart city" lo si intendeva principalmente in relazione alle tematiche di architettura e salvaguardia del territorio, oggi lo sviluppo di una smart city include aspetti legati anche al rispetto dell'ambiente, all'educazione digitale dei cittadini e alla digitalizzazione della politica, al controllo del traffico, alla pianificazione dei trasporti pubblici, al monitoraggio dell'illuminazione, alla produzione e distribuzione attiva dell'energia elettrica e via discorrendo. In questo ambito l'importanza dell'ICT cresce giorno dopo giorno grazie alle possibilità e agli scenari abilitati da IoT, che ha un ruolo fondamentale laddove vi sia necessità di acquisire, elaborare o trasmettere informazioni.

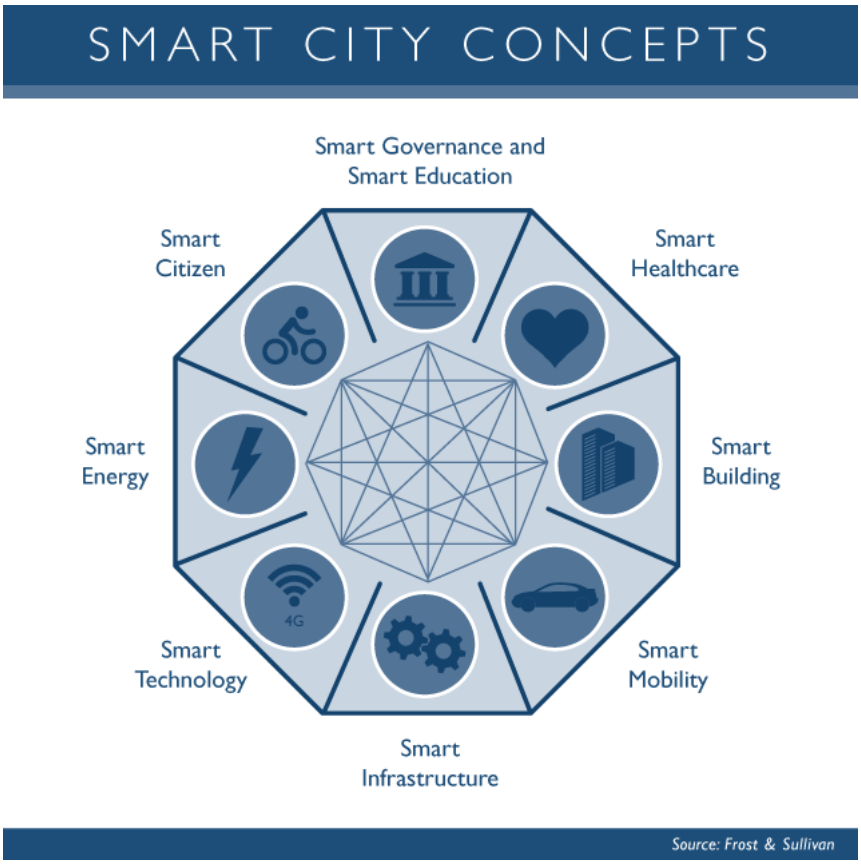

Figura 2.2: I principali settori di applicazione di Smart City

2.1.2 Sistemi distribuiti e Internet of Things

Un sistema distribuito consiste in un insieme di entità computazionali autonome spazialmente separate, collegate tra loro attraverso una rete, che collaborano allo scopo di raggiungere un obiettivo comune, comunicando tipicamente attraverso lo scambio di messaggi. Le caratteristiche chiave di un sistema distribuito sono:

- eterogeneità: i vari processi che concorrono per raggiungere l'obiettivo possono essere completamente differenti sia dal punto di vista dell'hardware, sia dal punto di vista del software;
- scalabilità: cioè la capacità di erogare adeguate prestazioni in termini di *throughput* e latenza verso gli utilizzatori, adeguandosi in modo flessibile alle variazioni di carico di lavoro;
- *fault-tolerance*: la capacità di reagire ai guasti parziali rimpiazzando le parti non più disponibili ed eventualmente variando il livello di servizio offerto.

Presi in considerazione gli aspetti principali e le caratteristiche di IoT, risulta evidente come questo nuovo paradigma si sposi alla perfezione con le caratteristiche di un sistema distribuito: date le limitate capacità individuali e la grande pervasività dei dispositivi intelligenti nel mondo IoT, gran parte di essi può essere naturalmente inserita in un qualunque contesto applicativo distribuito, dove ognuno coopera e collabora con il resto del sistema in maniera proattiva per raggiungere gli obiettivi comuni, dividendo compiti ed aumentando le potenzialità complessive.

Un sistema composto da più dispositivi intelligenti e connessi, comporta presumibilmente una maggiore quantità di informazioni. Emerge quindi ancora una volta l'importanza delle tecniche per la gestione e l'elaborazione di questa grande quantità di informazioni (*Big Data*). Il termine è utilizzato genericamente per

caratterizzare un insieme di dati (strutturati o meno) che normalmente è molto difficile da utilizzare ed elaborare con le tecniche tradizionali. Un sistema distribuito composto da migliaia di sensori o generici dispositivi intelligenti genera un quantitativo di informazioni molto alto e molto velocemente: il fattore di crescita di questi dati solitamente impone di porre dei limiti temporali trascorsi i quali i le informazioni sono eliminate. Riuscire ad analizzare ed aggregare i big data al fine di estrarne informazioni significative (in tempi rapidi), è uno dei temi che suscitano maggior interesse nel mondo dell'ICT moderna, poiché ancora oggi gran parte di questi dati risultano inutilizzabili e di conseguenza scartati.

2.2 Tecnologie di comunicazione

Tra le tecnologie abilitanti per IoT, troviamo sicuramente tutte quelle per la comunicazione *wireless*. Grazie alla possibilità di connessione senza cablaggi, spesso ingombranti e di difficile gestione, tutti i dispositivi sono in grado di interagire e comunicare con il resto della rete attraverso onde radio.
Inerentemente ai dispositivi di telefonia mobile, anche la classica tecnologia GSM si è evoluta notevolmente, in quanto oggi le reti mobili, in particolare 3G e 4G, forniscono una banda più larga persino della comune ADSL domestica. Lo stesso discorso può essere fatto per le reti wi-fi, che con i protocolli più recenti dello standard 802.11[12] arrivano a garantire velocità di upload e download un tempo riservate esclusivamente alle comuni reti ethernet. Se da un lato queste due tecnologie si stanno dimostrando ottime per tutti gli scenari che privilegiano la velocità di comunicazione (ad esempio per la fruizione e distribuzione di contenuti multimediali), dall'altro si può notare come una nuova questione stia emergendo di pari passo con IoT: la possibilità di comunicare a breve distanza mantenendo ridotto il livello dei consumi.

Lo scenario classico delle reti di sensori, prevede tanti dispositivi embedded sparsi nell'ambiente impegnati a rilevare informazioni ed in grado di trasmetterle all'interno della rete. Come spesso accade, questi piccoli dispositivi hanno limitatissime capacità di calcolo e non necessitano di CPU particolarmente potenti, conseguentemente possono essere alimentati con delle batterie oppure auto ricaricarsi (ad esempio con dei piccoli pannelli solari). La necessità di mantenere basso il livello di consumi, allungando quindi la vita del dispositivo stesso, introduce l'esigenza di utilizzare tecnologie di comunicazione poco dispendiose, che limitano il tempo e il raggio di comunicazione al minimo indispensabile.

2.2.1 Near Field Communication

Near Field Communication (NFC) è una tecnologia per la connettività wireless bidirezionale a corto-raggio, intuitiva, semplice e sicura tra dispositivi elettronici. Nasce come combinazione tra la tecnologia RFID (Radio Frequency Identification) e altre tecnologie di connettività. Contrariamente a RFID, NFC include un'importante evoluzione, ovvero la possibilità di interazioni bidirezionali. Di fatto NFC dà la possibilità di instaurare comunicazioni sia tra due dispositivi attivi (dotati cioè di alimentazione), sia tra un dispositivo attivo ed uno passivo. La comunicazione avviene quando due dispositivi NFC-compatibili sono posti ad una distanza di pochi centimetri l'uno dall'altro.

Il "Near Field Communication Forum" è l'ente fondatore dello standard NFC, ed è stato istituito nel 2004 per promuovere l'utilizzo della tecnologia attraverso la definizione di specifiche di sviluppo che consentissero sia l'interoperabilità tra dispositivi e servizi, che l'educazione del mercato nei confronti di NFC.

Caratteristiche principali

NFC non è nato per il trasferimento di grosse moli di dati, ma allo scopo di dotare un dispositivo di un tipo di comunicazione wireless semplice, sicura e veloce da realizzare, che possa servire da ponte a servizi già esistenti o che possa permettere la realizzazione di un nuovo tipo di servizi che fa di NFC il cuore della comunicazione.
Per questo motivo tra le sue caratteristiche non spicca la velocità di trasferimento dati, che è di molto inferiore a quella offerta dal classico Bluetooth o dal Wi-fi.
Si possono così riassumere le caratteristiche maggiormente rilevanti [19]:

- NFC opera a 13.56 MHz con larghezza di banda di circa 2MHz, utilizzando lo standard ISO/IEC 18000-3 per le comunicazioni wireless (RFId) e la velocità di trasmissione dati teorica può variare tra 106 kbit/s e 424 kbit/s;

- i dispositivi NFC sono in grado di ricevere e trasmettere dati simultaneamente, oltretutto con la possibilità di rilevare eventuali collisioni se la frequenza del segnale in risposta non dovesse corrispondere alla frequenza del segnale trasmesso;

- NFC instaura un canale di comunicazione sfruttando l'induzione magnetica tra due antenne poste l'una in vicinanza del campo magnetico generato dall'altra;

- la distanza teorica di funzionamento utilizzando antenne standard è fino a 20 cm, ma nella pratica non si discosta dai 4 centimetri. Proprio a causa di questa caratteristica, le comunicazioni NFC sono considerate tipicamente sicure.

Dispositivi attivi e tag passivi

Prima di discutere le modalità operative che possono essere sfruttate da NFC, occorre definire e distinguere i due principali attori che possono comparire in una comunicazione NFC, ovvero dispositivi attivi e tag passivi.
I chip NFC sono alla base di ogni dispositivo e rappresentano il cuore pulsante della tecnologia. Si tratta dell'insieme composto da un circuito digitale, un'antenna ed eventualmente un elemento sicuro utilizzabile per l'emulazione di una smart card. Applicativamente ogni dispositivo attivo deve implementare le parti obbligatorie dell'NFC Protocol Stack, al fine di supportare obbligatoriamente le modalità di comunicazione peer to peer e lettura/scrittura tag. Un dispositivo NFC attivo può essere uno smartphone, un tablet, un PC o un altro generico dispositivo elettronico dotato di una qualche forma di alimentazione.

Contrariamente ad un dispositivo, un *tag* è un componente elettronico **passivo**. È composto da un chip, che costituisce la parte "intelligente", una memoria non volatile che contiene le informazioni, ed un avvolgimento elicoidale che funge da antenna, necessaria per comunicare con un lettore attivo. I tag possono essere differenziati attraverso caratteristiche come velocità di comunicazione, possibilità di configurazione, capacità di memoria, sicurezza, durata e resistenza. Essendo derivati direttamente da RFID, NFC non ne specifica l'architettura fisica, ma definisce quattro diversi formati supportati Figura 2.3.

I tag NFC hanno la capacità di contenere un grande quantitativo di informazioni che gli utenti possono semplicemente leggere avvicinando il proprio dispositivo al tag. Applicati su semplici oggetti, sono in grado di aumentarne quindi il contenuto informativo. Data la loro discendenza da RFID, i tag intrinsecamente dispongono di meccanismi attraverso i quali è possibile adottarli per etichettare e riconoscere automaticamente cose o persone. Infine, un altro interessante caso d'uso è quello che li vede programmati per comunicare al dispositivo lettore una

	NFC Forum Platform			
	Type 1 Tag	**Type 2 Tag**	**Type 3 Tag**	**Type 4 Tag**
Compatible Products	Innovision Topaz	NXP MIFARE Ultralight / NXP MIFARE Ultralight C	Sony FeliCa	NXP DESFire / NXP SmartMX-JCOP
Memory Size	96 Bytes	48 Bytes / 144 Bytes	1, 4, 9 KB	4 KB / 32 KB
Unit Price	Low	Low	High	Medium / High
Data Access	Read/Write or Read-only	Read/Write or Read-only	Read/Write or Read-only	Read/Write or Read-only
Active Content	×	× / ×	×	× / ✓
Operation Specifications	[TYPE 1 TAG]	[TYPE 2 TAG]	[TYPE 3 TAG]	[TYPE 4 TAG]
NXP Supporting Documents	-	[NXP T2T]	-	[NXP T4T]
NXP Product Datasheets	-	[NXP UL, NXP ULC]	-	[NXP DES]

Figura 2.3: Tabella riassuntiva delle principali caratteristiche dei 4 NFC Forum Type Tag Platform

serie di azioni da svolgere in modo automatico. Si parla di *handover* quando un tag contiene le istruzioni per l'esecuzione di una procedura che altrimenti l'utente dovrebbe completare manualmente sul dispositivo, ad esempio la connessione ad una rete wi-fi protetta da una password molto lunga e difficile da digitare.
Il range molto limitato entro il quale opera NFC garantisce rischi minori dal punto di vista della sicurezza del canale di comunicazione, che quindi può essere utilizzato in modo più agile anche per operazioni che vedono coinvolti dati sensibili, come ad esempio per i pagamenti.

Modalità Operative

NFC può operare, a seconda della tipologia di attori che partecipano alla comunicazione, in tre diverse modalità: lettura/scrittura, card emulation e peer to peer.

Nella modalità lettura-scrittura il dispositivo attivo è detto *initiator* ed è in grado di leggere (o eventualmente scrivere) un tag NFC detto *target*. Il dispositivo

initiator crea un campo magnetico portante e il dispositivo target si alimenta in modo induttivo utilizzando l'energia del campo elettromagnetico generato, diventando a tutti gli effetti un transponder.
Questa modalità operativa è probabilmente la più comune, ed è quella utilizzata quando c'è interazione con tag NFC passivi.

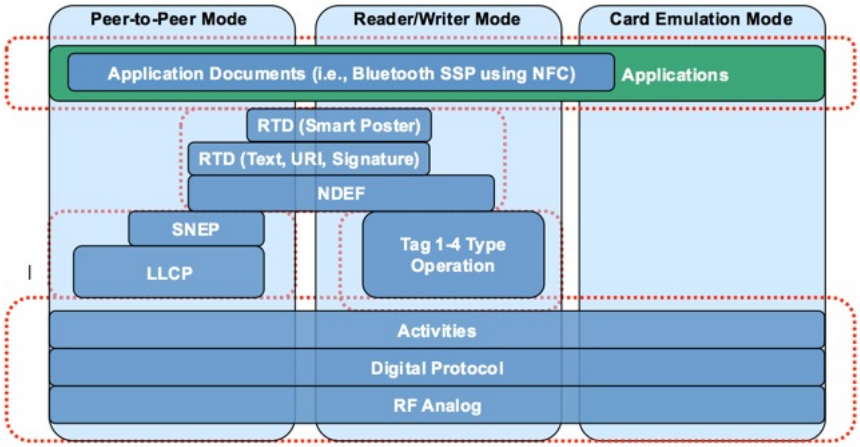

Figura 2.4: Lo stack NFC, diviso nelle tre differenti modalità di comunicazione

La card-emulation permette ad un dispositivo NFC di agire come se fosse una smartcard (contactless) esistente, per comunicare in modo sicuro con lettori esterni. Ai fini della certificazione NFC Forum, la card emulation è una funzionalità facoltativa, poiché non è richiesta come requisito indispensabile. Tipicamente trova impiego negli scenari che vedono normalmente coinvolte le smart-card: mobile ticketing, mobile payment e controllo degli accessi.

La modalità di comunicazione p2p entra in gioco quando due dispositivi NFC attivi necessitano di comunicare in modo bidirezionale. Differentemente dai casi precedenti, initiator e target comunicano a fasi alterne scambiandosi di ruolo: è momentaneamente initiator chi deve trasmettere dati e genera il campo magnetico, mentre è target chi deve ricevere, che disattiva la generazione del proprio campo magnetico. Con la diffusione del chip NFC in gran parte dei moderni smartphone,

è la modalità operativa che si presta ad un maggior numero di scenari, sicuramente la più interessante per applicazioni future.

NDEF

NDEF (NFC Data Exchange Format) è il formato di dati definito da NFC Forum per lo scambio di informazioni in una generica comunicazione NFC. NDEF definisce precise regole di formato e struttura del messaggio, senza però limitare i tipi di informazione che possono essere contenute, in modo poter incapsulare una grande varietà di dati in diversi formati: XML, URL, testo, etc.
Un messaggio NDEF è composto da uno o più record NDEF e l'incapsulamento dei dati avviene proprio all'interno dei questi record, solitamente tipizzati attraverso l'uso dei comuni MIME types, oppure con tipi definiti e standardizzati da NFC Forum. Il dispositivo che riceve un messaggio NDEF è in grado di conoscere la dimensione e il tipo dei dati leggendo esclusivamente l'intestazione del messaggio.

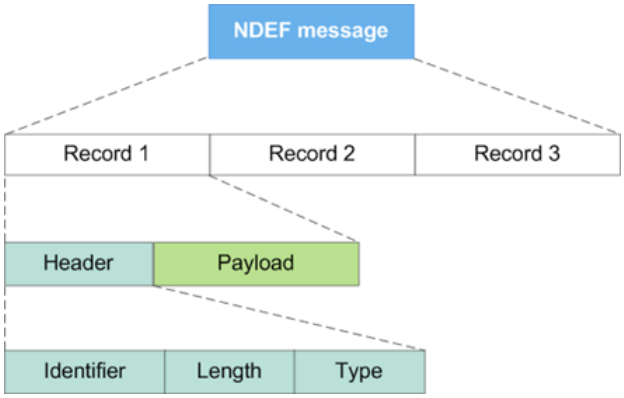

Figura 2.5: Schema esemplificativo di un messaggio NDEF

SNEP ed LLCP

NFC Forum definisce due differenti protocolli per la comunicazione bidirezionale peer to peer, tipicamente implementati in tutti i dispositivi NFC compatibili. Logical Link Control Protocol (LLCP) è utilizzato per la creazione del canale di comunicazione bidirezionale e sicuro (*socket*) tra due dispositivi attivi, canale che può essere utilizzato sia in modo monodirezionale, sia bidirezionale. Si tratta di un protocollo di livello OSI 2 orientato al modello client-server, che prevede sia il livello di trasporto *connectionless*, sia *connection-oriented*. Anche Simple NDEF Exchange Protocol (SNEP) è usato nel contesto delle comunicazioni peer to peer, ma essendo costruito sopra LLCP risulta essere di livello più alto. Si tratta di un protocollo privo di controllo di stato e basato essenzialmente sul modello request/response, dove il client è l'entità che si preoccupa di inviare (PUT) e richiedere (GET) al server determinati messaggi NDEF.

2.2.2 Bluetooth Low Energy

Bluetooth low energy, spesso definito anche come Bluetooth smart, è una tecnologia di comunicazione wireless derivante dal Bluetooth classico (vanilla Bluetooth), con l'obiettivo principale di ridurre consumi e costi di mantenimento, lasciando pressoché inalterato il range di comunicazione. Bluetooth LE è stato introdotto per la prima volta con il nome di "Wibree" dai ricercatori Nokia (2006), ma soltanto nel 2010 il *Bluetooth Special Interest Group* (SIG) ha integrato il protocollo nello standard di specifiche Bluetooth 4.0. Al giorno d'oggi tutti i principali sistemi operativi per i dispositivi mobili e personal computer, supportano nativamente Bluetooth LE, che si stima sarà integrato nel 90% dei dispositivi Bluetooth compatibili entro il 2018 [5].

Caratteristiche principali

Bluetooth low energy è definito dall'ente Bluetooth SIG come: *"the intelligent, power-friendly version of Bluetooth wireless technology"*. Proprio per questa caratteristica, la sua diffusione è stata molto facilitata dall'avvento di IoT, in quanto può essere considerata la tecnologia che offre il miglior compromesso tra range di trasmissione, bassi consumi e velocità di comunicazione. Le caratteristiche principali sono:

- trasferimento dati con pacchetti molto piccoli (da un minimo di 8 ad un massimo di 27 byte) e una velocità teorica di 1Mbps;
- consumi bassissimi sia come picco massimo (circa 30 mA), sia per quel che riguarda le fasi di idle e sleep;
- salto di frequenza, per minimizzare le interferenze nella banda dei 2.4 GHz, incrementando così la qualità e la stabilità del collegamento;
- latenza minima, poiché i dispositivi possono trasferire dati con una latenza di soli 3ms, e tornare velocemente nello stato di sleep;
- irrobustimento del collegamento grazie ad un CRC a 24 bit;
- aumento del livello di sicurezza, BLE utilizza AES-128 encryption per cifrare il canale di comunicazione.

Anche se le specifiche garantiscono velocità di trasferimento dati (teoricamente) buone, questo fattore non è prioritario nella maggior parte dei casi d'uso in cui la tecnologia è impiegata. Bluetooth LE non è adatto per l'invio di grandi quantitativi di informazioni all'interno di una singola connessione (streaming multimediali, streaming di file), mentre lo è molto di più per trasferimenti singoli di piccoli quantitiativi informativi, anche molto frequenti nel tempo. Si pensi ad

BLE vs mainstream technologies

		Classic Bluetooth	**Bluetooth Low Energy**	NFC	Wi-Fi (802.11n)
Range (theoretical)		~100m	~70m	~10cm	~90m
Data rate		1 - 3 Mbps	**1 Mbps**	424 Kbps	300 Mbps
Connection mandatory		Yes	**Yes - No**	Yes - No	Yes
Power consumption	sleep	~80 µA	~1µA	~2µA	~10-30 mA
	idle	~8 mA	~1mA	~45µA	~200 mA
	peak	~30-40 mA	~15-20mA	~30mA	~300-600 mA
Security		56 - 128 bit	128 bit AES	Secure Channels	128 - 256 bit

Figura 2.6: Tabella comparativa tra le caratteristiche di Bluetooth low energy e quelle delle altre tecnologie di comunicazione wireless più diffuse

esempio ad un sensore interrogato una volta al minuto, che invia quindi centinaia di byte per ora: non si tratta di grandi quantità, tuttavia sono dati potenzialmente molto significativi. L'importanza che riveste il basso livello di consumi è ancor più comprensibile se si approfondisce l'esempio fatto. Supponendo che:

- il sensore sia dotato di una batteria 3V - 250mAh (comune CR2032);
- il consumo durante la comunicazione sia di 15mA;
- ogni transazione abbia una durata di 3ms;
- il sensore sia interrogato una volta al minuto.

La durata complessiva della batteria è così deducibile:

- 250mAh / 15mA = 16h = 57'600'000 ms;
- 57'600'000ms / 3ms = 19'200'000 diverse transazioni;
- una transazione al minuto consiste in 1440 transazioni al giorno;

- 19’200’000 / 1440 = 36 anni;

Senza tener conto dei bassissimi consumi nello stato di sleep (nell’ordine dei micro ampere) e dei consumi degli apparati di calcolo e sensoristica, questo semplice esempio mostra che una singola batteria potrebbe alimentare la componente Bluetooth LE di un dispositivo per circa 36 anni, tempo che ovviamente oltrepasserebbe quello di vita della batteria. Data la bassissima quantità di energia giornaliera necessaria, questo caso d’uso potrebbe risultare molto adatto all’installazione di micro sistemi di generazione energia da fonti rinnovabili, altro importantissimo settore di Internet of Energy.

Stack applicativo

L’architettura definita da Bluetooth 4.0 condivide solo parte dei moduli con le versioni precedenti del protocollo. Di fatto Bluetooth LE non è retrocompatbile. In Figura 2.7 è riportata la struttura dello stack:

- il controller è sostanzialmente il modulo radio che utilizza la modulazione in radiofrequenza per comunicare con il mondo esterno; il *link layer* controlla la modalità operativa dell’antenna;

- tra controller ed host è presente la Host Controller Interface (HCI), attraverso la quale i due livelli possono interagire;

- il modulo host è il cuore della tecnologia Bluetooth LE. Il Logical Link Control and Adaptation (L2CAP) è l’unico livello dell’intero stack che è possibile ritrovare anche nelle versioni precedenti del protocollo, ed è il modulo responsabile della gestione del canale di comunicazione con il dispositivo *peer*; il Security Manager gestisce la cifratura del canale di comunicazione e le operazioni per la condivisione delle chiavi, è quindi letteralmente il mo-

dulo responsabile della sicurezza; fanno parte dell'host anche ATT e GATT, che saranno approfonditi in seguito;

- il livello di application è ovviamente costituito da tutte le applicazioni concrete che sfruttano lo stack Bluetooth 4.0 interfacciandosi con ATT e GATT.

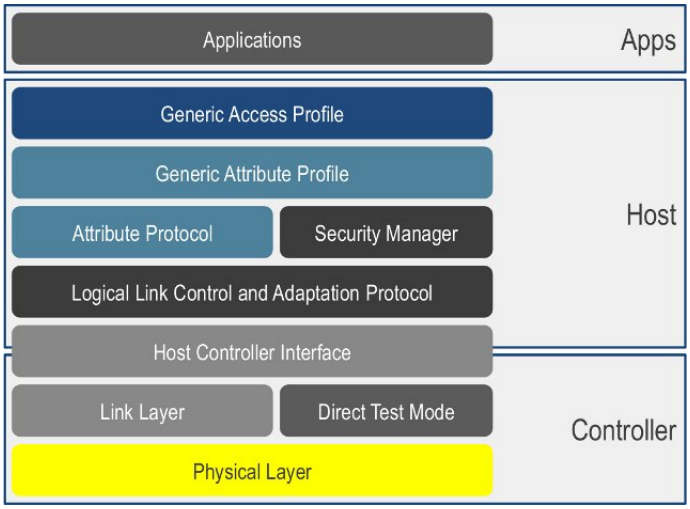

Figura 2.7: Lo stack applicativo definito dal protocollo Bluetooth 4.0

ATT e GATT

Generic Attribute Profile (GATT) prevede un insieme di specifiche per l'invio e la ricezione di piccole quantità informative, definite come "attrbuti", attraverso un canale di comunicazione BLE. Tutte le applicazioni che utilizzano Bluetooth LE sono basate su GATT. Gli attributi sfruttati dalle applicazioni GATT sono invece definiti da ATT (Attribute Protocol) che specifica le seguenti regole:

- i dati sono memorizzati all'interno di attributi;

- gli attributi possono essere letti o scritti da altri dispositivi;

- gli attributi sono etichettati con un identificativo univoco e dovrebbero essere semanticamente raggruppati in classi;
- gli attributi possono avere diverse politiche di accesso (lettura - scrittura) e richiedere diversi livelli di autenticazione;
- gli attributi sono *stateless*.

Handle	**UUID**	**Value**	**Permission**
16 bit	16 - 128 bit	0 - 512 bytes	rw / auth

Il protocollo GATT introduce i ruoli di server, per il dispositivo che implementa il servizio, e di client, per chi invece accede - attraverso gli attributi - ai dati esposti dal servizio. Oltre al concetto di servizio, GATT prevede anche le nozioni di caratteristica e descrittore.
Una caratteristica non è altro che un contenitore al cui interno è presente un singolo valore, arricchito da un numero variabile di descrittori, attributi che aiutano a comprendere il significato semantico del valore della caratteristica aggiungendovi informazioni (ad esempio sul formato, sulla codifica, sull'unità di misura, etc.). Un servizio GATT è quindi un insieme di attributi organizzati gerarchicamente in caratteristiche e descrittori.
L'ente Bluetooth SIG ha definito con precisione un insieme di servizi e profili GATT che i produttori possono implementare per rendere i propri dispositivi standard; i profili sono assolutamente trasversali a numerose discipline ed includono uno o più servizi GATT. Un esempio può essere un misuratore di velocità e cadenza dedicato agli sportivi, un misuratore di pressione sanguigna oppure un termometro. Oltre a quelli standard, è possibile definire anche servizi, descrittori e caratteristiche personalizzate, in particolare quando è necessario realizzare dispositivi che rispondono ad esigenze particolari. Come anticipato in precedenza, GATT regolamenta come gli attributi debbano essere utilizzati per la costruzione

di servizi. GATT utilizza il campo UUID per assegnare un tipo specifico ad un attributo. Sono utilizzate le seguenti classi UUID:

- da 0x1800 a 0x28FF: servizi;
- da 0x2700 a 0x27FF: unità di misura;
- da 0x2800 a 0x29FF: tipi di attributi (0x2800 per un servizio, 0x2803 per una caratteristica);
- da 0x2900 a 0x29FF: tipi di descrittori (*user readable descriptor, valid range descriptor*, etc);
- da 0x2A00 a 0x7FFF: tipi di caratteristiche (*battery level characteristic, datetime characteristic*, etc).

Nella tabella in Figura 2.8 è riportata l'implementazione di un servizio GATT per la misurazione della temperatura e della pressione ambientale. Il servizio include tre differenti caratteristiche in sola lettura:

- *temperature characteristic*, che il dispositivo utilizza per comunicare il valore della temperatura ambientale;
- *humidity characteristic*, che contiene il valore dell'umidità nell'aria;
- *date time characteristic*, usata per riportare la data e l'ora dell'ultima rilevazione ambientale da parte del device.

Le prime due caratteristiche sono decorate da un attributo che ne specifica il formato e l'unità di misura, mentre l'ultima caratteristica, essendo standard, è già ben interpretata dai dispositivi client. Data la difficile lettura della tabella, soprattutto a causa dei numerosi identificativi esadecimali non propriamente intuitivi, in Figura 2.9 è riportata una visualizzazione più semplice ed intuitiva. I colori

dovrebbero aiutare a comprendere meglio la separazione semantica degli attributi.

GATT Example - Attributes View

Handle	UUID	Value	Permission	
0x0100	0x2800	UUID 0x1816	read	Thermometer service definition
0x0101	0x2803	UUID 0x3A00	read	Temperature Characteristic Definition
0x0102	0x3A00	20.7	read	Temperature Characteristic Value
0x0103	0x2904	Float - Celsius	read	Temperature Characteristic Descriptor
0x0200	0x2803	UUID 0x3A01	read	Humidity Characteristic Definition
0x0201	0x3A01	55	read	Humidity Characteristic Value
0x0202	0x2904	Float - Percent	read	Humidity Characteristic Descriptor
0x0300	0x2803	UUID 0x2A08	read	DateTime Characteristic Definition
0x0301	0x2A08	10-26-2014 10:37:22	read	DateTime Characteristic Value

Figura 2.8: Implementazione di un servizio GATT: la struttura degli attributi

GATT Example - Friendly View

Handle	UUID	Value	Permission
0x0100	Primary Service	Thermometer Service	read
0x0101	Characteristic	Temperature Characteristic	read
0x0102	Temperature Characteristic	20.7	read
0x0103	Characteristic Presentation Format	32 bit Float - Celsius	read
0x0200	Characteristic	Humidity Characteristic	read
0x0201	Humidity Characteristic	55	read
0x0202	Characteristic Presentation Format	32 bit float Float - Humidity	read
0x0300	Characteristic	DateTime Characteristic	read
0x0301	DateTime Characteristic	10-26-2014 10:37:22	read

Figura 2.9: Implementazione di un servizio GATT: la struttura degli attributi (visualizzazione intuitiva)

I ruoli e le modalità operative

In seguito alla descrizione di GATT è intuitivamente chiaro che il modello di comunicazione Bluetooth LE si discosti molto da quello proposto ed implementato nel Bluetooth classico (socket punto-punto), poiché è orientato ad un'architettura client-server. I device che interagiscono in una comunicazione Bluetooth LE, assumono due differenti ruoli, solitamente non interscambiabili:

- i dispositivi "centrali" sono quelli che si preoccupano di scansionare l'ambiente circostante alla ricerca di *advertising*, ovvero segnali informativi emessi da altri dispositivi Bluetooth LE;
- i dispositvi "periferici", dal canto loro, emettono advertising nell'ambiente ed aspettano la connessione da parte di dispositivi centrali.

Si delineano quindi anche i due differenti ruoli per il modello GATT client - server. I dispositivi periferici sono i server, implementano uno o più servizi GATT ed emettono nell'ambiente un segnale informativo che può essere interpretato dai dispositivi centrali (ovvero i client), che possono connettersi per sfruttare uno o più servizi GATT.

La sicurezza in Bluetooth LE

Bluetooth LE garantisce la possibilità di cifrare il canale di comunicazione tra due dispositivi utilizzando l'algoritmo AES 128. In questo modo può essere ridotto il rischio di attacchi come *man in the middle* o *eavesdropping*. La sicurezza comincia nel momento in cui uno dei due device effettua una richiesta di *pairing* all'altro. La procedura di pairing prevede inizialmente lo scambio di informazioni legate alle caratteristiche dei due device, dopodiché i dispositivi creano un link temporaneo cifrato utilizzando una chiave corta (tipicamente un numero di sei cifre), attraverso il quale poi avviene lo scambio delle chiavi di cifratura definitive

(*long-term keys*). I device a questo punto sono accoppiati (*bonded*) e tutte le future connessioni tra essi faranno uso delle chiavi memorizzate per ristabilire il canale sicuro.

2.3 Sistemi embedded

Nonostante l'evidenza di quanto Internet of Things sia stato incentivato dai notevoli progressi riguardanti le tecnologie di comunicazione senza fili, non bisogna dimenticare nemmeno il ruolo cruciale avuto dall'elettronica moderna, che ha progressivamente reso i microprocessori sempre più piccoli, potenti ed economici. Proprio grazie a questo sono nati i sistemi embedded, dispositivi elettronici integrati *special purpose* (progettati per una determinata applicazione) e tipicamente non riprogrammabili dall'utente per altri scopi. Questi dispositivi sono basati su un microcontrollore o un microprocessore e dispongono di una piattaforma hardware ad hoc integrata nel sistema che controllano, che li mette in condizione di gestirne tutte o parte delle funzionalità richieste.
I sistemi embedded controllano al giorno d'oggi molti dispositivi di uso comune. Il loro utilizzo spazia da dispositivi portatili, come orologi digitali e lettori MP3, a grandi installazioni stazionarie, come impianti semaforici. Questo genere di dispositivi è largamente impiegato anche all'interno di sistemi complessi (automobili, aerei), di conseguenza il livello di complessità e capacità può quindi variare molto, dai semplici sistemi composti da un unico microcontrollore, a varianti più complesse con multipli chip.

Caratteristiche principali

Gran parte dei sistemi embedded è integrata internamente ad oggetti largamente diffusi su scala mondiale. È importante in questo caso che la loro progettazione

tenga in considerazione diversi aspetti:

- minimizzazione dei costi di produzione, impiegando componenti hardware che soddisfino esclusivamente i requisiti del sistema;
- minimizzazione del fattore di forma, dipendentemente dal contesto nel quale il dispositivo è integrato;
- ottimizzazione del software, la cui esecuzione deve risultare efficiente in relazione alle limitate capacità di calcolo di questi dispositivi;
- affidabilità dell'hardware, che deve spesso resistere ad eventi dannosi, sollecitazioni meccaniche e shock di varia natura;
- affidabilità del software, poiché una volta immesso il dispositivo nel mercato è difficile intervenire con patch per risolvere eventuali problemi;
- longevità, in particolare se l'oggetto in cui il sistema è integrato ha un costo iniziale rilevante e non si tratta di un bene di consumo.

Affidabilità e longevità hanno un'importanza ancora maggiore quando i sistemi embedded sono integrati in dispositivi che devono garantire la sicurezza degli esseri umani, si pensi ad esempio a un'automobile, un ascensore oppure un aereo. Il processo di sviluppo deve in questi casi prevedere accurate procedure di test che possano escludere ogni genere di errore hardware o software.

Tipicamente i sistemi embedded dispongono di interfaccia utente minimale (in alcuni casi del tutto assente), in quanto il dispositivo all'interno del quale sono integrati è spesso molto piccolo e con funzionalità limitate. Si utilizzano talvolta elementi come LED, display LCD o piccoli altoparlanti piezoelettrici, sufficienti per dare un feedback all'utente sullo stato del dispositivo o sull'esito di azioni che esso sta svolgendo.

Alcuni particolari embedded system hanno anche la caratteristica di essere time-critical, ovvero i risultati delle operazioni da essi compiute sono rilevanti solo completate in un determinato intervallo di tempo. Proprio per questo sono stati sviluppati sistemi operativi real time, che migliorano l'efficienza temporale rispetto a quelli tradizionali.

Architetture hardware e software

Non esiste un'architettura di riferimento per i sistemi embedded. Considerata la grandissima varietà di scenari che li vedono protagonisti, le soluzioni sono molteplici. La prima distinzione che può essere fatta riguarda l'unità di elaborazione, che può essere distinta tra general purpose e special purpose.

Con *General Purpose Processor* (GPP) si intendono tutte le soluzioni che prevedono l'utilizzo di una CPU programmabile e in generale in grado di svolgere diverse tipologie di compiti. A differenza di quanto avviene nei microprocessori dei personal computer, nei sistemi embedded la CPU può essere basata sia sulla classica architettura di *Von Neumann*, sia su diverse varianti del modello *Harvard*, così come essere *RISC-based* o *non-RISC-based*. In generale le soluzioni RISC sono preferibili per la relazione vantaggiosa che lega consumo energetico e prestazioni. Le lunghezze di word variano tipicamente dai 4 ai 64 bit, anche se il più tipico formato resta a 8/16 bit. Una tra le famiglie di microprocessori RISC più famose è sicuramente ARM (Advanced RISC Machine), architettura impiegata moltissimo nel contesto dei sistemi embedded e non solo, dai primi lettori mp3 fino ai moderni smartphone.

Application Specific Instruction Set Processor (ASIP) sono invece utilizzati quando i sistemi embedded devono dare soluzione ad uno specifico problema caratterizzato da un numero molto limitato di funzioni, pertanto non è conveniente

fare ricorso ad un'architettura general purpose.
Sono compresi una grande varietà di sistemi, come:

- Field Programmable Gate Array (FPGA) e Digital Signal Processor (DSP), adatte ad elaborazioni molto ripetitive;
- Graphic Processing Unit (GPU) in grado di svolgere calcoli in parallelo per applicazioni che richiedono elevate prestazioni;
- Micro Controller Unit (MCU) per applicazioni a elevata integrazione, di basso costo e potenza elaborativa limitata;
- Network Processors (NP), che includono un set di istruzioni specifico per il networking.

Spesso gli ASIP sono abbinati ad una specifica componentistica hardware dedicata (Application Specific Integrated Circuit), realizzando quindi un unico System on a Chip (SoC), che include assieme all'unità di elaborazione centrale anche gli altri moduli elettronici (ROM, interfacce di I/O, etc.) necessari per renderlo completamente funzionale alle specifiche del contesto in cui è impiegato.

La scelta dell'architettura hardware più appropriata per lo sviluppo della propria applicazione è un compito complesso, in quanto devono essere presi in considerazione numerosi aspetti. In generale non esiste una vera e propria linea guida per la scelta. Se l'utilizzo di piattaforme general purpose consente tipicamente un grado maggiore di flessibilità applicativa, i microcontrollori ad esempio, presentano invece un altro tipo di vantaggi: costi solitamente minori, maggiore affidabilità, consumi ridotti e, per i compiti dedicati, un livello di performance più alto.
Un fattore determinante per la scelta risiede nelle caratteristiche dell'applicativo che si vuole sviluppare, poiché come si è specificato in precedenza, esistono architetture specializzate nello svolgere deteriminate operazioni.

Inoltre, il software che deve essere eseguito nel sistema embedded potrebbe necessitare di macchine virtuali (in caso ad esempio di linguaggio Java), framework, compilatori ed interpreti, che non sempre è possibile utilizzare tutte le differenti architetture.

2.3.1 I single board computer

L'avvento dei *single board computer* (SBC), in particolare di quelli in grado di interagire con periferiche di input-output analogiche o digitali, apre le porte alla prototipazione di veri e propri embedded system in ambito non industriale, scenario fino a qualche anno fa pressoché inaffrontabile per gli sviluppatori sia dal punto di vista dei costi, sia dal punto di vista delle competenze richieste. Le *evaluation board*, prodotte dai fornitori di componenti elettronici e dedicate alla valutazione e prototipazione avevano prezzi inaccessibili e lo sviluppo di applicativi doveva necessariamente essere fatto a livello molto basso.
Arduino e Raspberry pi sono sicuramente i due SBC più famosi e diffusi nel mondo: mentre il primo, nella sua versione iniziale, è costruito attorno ad un processore ATMega a 16MHz, il secondo dispone di una CPU ARM a 700MHz. Questo concetto è sufficiente per introdurre la diversità forse più rilevante: Raspberry pi dispone di caratteristiche hardware più elevate, che di fatto lo accomunano ad un mini computer, mentre Arduino è più simile ad un microcontrollore, nonostante consenta comunque uno sviluppo applicativo di - relativamente - alto livello grazie ad un sistema operativo dedicato. Entrambe le piattaforme possono interagire con General Purpose Input/Output (GPIO) digitali, mentre Raspberry non dispone nativamente di un modulo per la comunicazione con dispositivi analogici.

La diffusione di sistemi di questo tipo consente di realizzare, anche senza avere i mezzi e le tecnologie a disposizione di aziende specializzate nella produzione di sistemi elettronici integrati, soluzioni ad hoc per Internet of Things, facendone

crescere il livello di pervasività e le potenzialità. Nel capitolo 5 si potrà apprezzare come una parte fondamentale del progetto realizzato veda proprio Raspberry pi come piattaforma di base per la realizzazione di un embedded system in grado di comunicare attraverso la rete internet oppure con le nuove tecnologie NFC - Bluetooth LE, capace di semplici operazioni di elaborazione dati e di interagire con circuiti elettronici esterni attraverso interfacce output digitali.

2.3.2 Sistemi embedded e Internet of Things

Il ruolo degli embedded system è fondamentale e abilitante per Internet of Things, poiché è proprio grazie ad essi che è possibile dotare di "intelligenza" oggetti altrimenti privi di capacità elaborativa. Dal telecontrollo alla sensoristica distribuita, dagli *smart meters* agli impianti di illuminazione intelligenti, tutti gli ambiti e le discipline emergenti nel contesto delle smart city vedono impiegati un numero sempre maggiore di sistemi embedded, che grazie alle caratteristiche di compattezza, basso consumo ed economicità, possono adattarsi perfettamente alle nuove esigenze.

Oltre all'utilità nello sviluppo di soluzioni intelligenti e funzionali al miglioramento della vita delle persone, non bisogna dimenticare l'importanza che essi hanno e avranno sempre più nella produzione e memorizzazione di dati per l'universo digitale di informazioni.
Ad oggi circa il 2% di esso è composto da dati provenienti da embedded system che attraverso sensori ed altri sistemi di monitoraggio dell'"universo fisico" sono in grado di creare contenuto informativo e condividerlo, attraverso la rete, con il resto del mondo. Grazie alla crescita del numero di dispositivi embedded connessi, questa percentuale dovrebbe arrivare al 10% nel 2020 (Figura 2.10).

Ancora più importante sarà il ruolo che questi sistemi avranno nel contesto IoT del futuro come gestori di informazioni. I dati devono essere sempre memorizzati

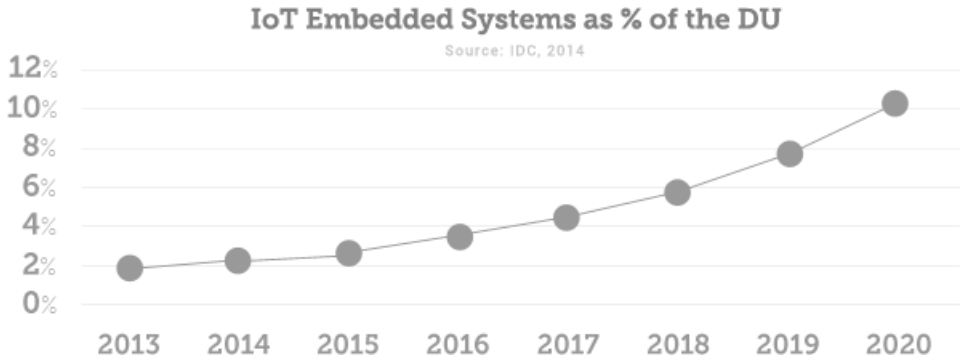

Figura 2.10: I dati prodotti dai sistemi embedded nell'universo digitale

in un "contenitore" (sotto forma di file), sia esso un computer, un tag NFC o una camera digitale. Se ad oggi gran parte dei file sono conservati in modo concentrato su computer, si stima che nel 2020 il 99% dei file presenti nell'universo digitale sarà gestito e memorizzato direttamente dai sistemi embedded [9].

Capitolo 3

Progettazione del sistema distribuito: overview

In questo capitolo sarà discussa la progettazione di massima del sistema, partendo dagli obiettivi che l'azienda vuole raggiungere attraverso lo sviluppo del progetto, e proseguendo con un'analisi dei principali requisiti dell'applicazione, siano essi funzionali o vincoli. Analizzando i requisiti si cercherà di formalizzare l'insieme di tutti i diversi scenari e casi d'uso applicativi, al fine di chiarire le modalità di interazione tra gli utenti e le differenti componenti del sistema. In seguito alla fase di definizione dei requisiti sarà presentato il modello del dominio applicativo, con lo scopo di chiarire la natura delle entità di dominio e l'insieme di relazioni che intercorrono tra di esse. L'evoluzione naturale e conclusione di questa prima fase di progettazione vedrà la presentazione dell'architettura logica del sistema, in cui si cercherà di dare una visione dall'alto della struttura e dell'interazione tra le varie componenti applicative identificate nel progetto.

3.1 Visione

Il controllo degli accessi agli spazi "sensibili" è sempre stato tra i temi più importanti per la sicurezza e l'organizzazione delle persone. Lo sviluppo di un sistema in grado di facilitare lo svolgimento di questo compito in ambito aziendale può diventare un servizio essenziale per molte organizzazioni che necessitano di una piattaforma standard dedicata e facilmente utilizzabile. Inoltre, l'utilizzo di strumenti che interagiscono in modo intelligente con l'utente permettendo di automatizzare i task più semplici, può integrare il sistema all'interno dell'ecosistema che l'internet delle cose sta velocemente popolando.

3.2 Obiettivi

L'obiettivo primario del progetto è quello di migliorare il processo di registrazione delle timbrature per gli utenti dell'azienda, sfruttando le differenti tecnologie di comunicazione wireless in prossimità che i moderni smartphone e i sistemi embedded di ultima generazione mettono a disposizione, ovvero Near Field Communication (NFC) oppure Bluetooth Low Energy (BLE). La comunicazione con l'utente e l'abilitazione all'accesso devono essere compiti svolti da un sistema embedded installato in prossimità della porta, che quindi dovrà identificare in modo univoco gli utenti, validare il tentativo di accesso, registrare la timbratura in una base di dati e aprire fisicamente la porta.

Obiettivo secondario, ma non meno importante, consiste nella realizzazione di un'applicazione web che le organizzazioni aziendali possono sfruttare per tenere traccia delle presenze in ufficio dei propri utenti. L'applicazione dovrà poter funzionare sia senza che l'organizzazione abbia il device installato sulla porta - con l'inserimento delle timbrature esclusivmente in modo manuale, da browser - , sia con il device installato, che renderebbe il sistema automatico.

Infine, la realizzazione dell'intero progetto in forma prototipale è aziendalmente un'opportunità per approfondire e toccare con mano sia le tantissime tecnologie utilizzate per lo sviluppo di ogni singola componente, sia tutti gli aspetti che sono intrinsecamente coinvolti nello sviluppo del sistema come insieme.

3.3 Analisi dei requisiti

Il sistema può essere suddiviso in differenti componenti, di conseguenza anche i requisiti saranno descritti sia come requisiti di sistema nella sua completezza,sia come requisiti delle diverse parti.
Requisiti del sistema:

- il sistema distribuito deve funzionare prediligendo l'utilizzo di un'unica base di dati sul "cloud" (perlomeno dal punto di vista concettuale). Di conseguenza, le interazioni tra le diverse parti applicative devono normalmente avvenire con il supporto della rete internet;
- il sistema deve implementare buoni standard di sicurezza. Anche se i dati di dominio non possono essere considerati confidenziali, si tratta comunque di informazioni sensibili;
- il sistema deve poter essere sfruttato dalle aziende come servizio (software as a service) cloud;
- il sistema deve essere facilmente estendibile, in quanto le specifiche e le funzionalità offerte sono in continua evoluzione;
- il sistema deve essere di facile ed intuitivo utilizzo per gli utenti;
- il sistema deve essere facilmente scalabile.

3.3.1 Requisiti della web application

La web application rappresenta la via attraverso la quale gli utenti delle organizzazioni (compresi gli amministratori) possono accedere ai dati di dominio. Può essere sommariamente descritta attraverso questo insieme di requisiti:

- tutti i servizi primari dell'applicazione devono essere disponibili soltanto agli utenti autenticati, quindi non è prevista una parte "pubblica", escluse le canoniche funzionalità per la registrazione e l'autenticazione;
- l'utente può registrarsi direttamente con email e password, oppure può autenticarsi con il proprio profilo Google;
- il sistema deve prevedere la possibilità per gli utenti di recuperare la password dimenticata;
- all'interno dell'applicazione gli utenti devono essere raggruppati in organizzazioni. Gli utenti potranno avere, per ogni organizzazione a cui appartengono, il ruolo di utente oppure amministratore;
- ogni utente deve appartenere ad una o più organizzazioni;
- quando l'utente effettua il login, deve necessariamente scegliere l'organizzazione per la quale vuole visualizzare i propri dati. Può scegliere di cambiare l'organizzazione selezionata in un qualunque momento durante la propria sessione;
- l'utente ha accesso a due principali sezioni: timbrature e assenze;
- per ognuna di queste, potrà visualizzare, inserire, modificare ed eliminare i dati a lui collegati;

- oltre alle proprie timbrature e assenze, un utente amministratore di una determinata organizzazione deve poter vedere, modificare o eliminare anche tutti i dati relativi agli altri utenti dell'organizzazione;
- in aggiunta alle due viste principali, l'utente deve poter accedere ad una sezione secondaria di amministrazione, che dovrà consentirgli di gestire il proprio profilo e le organizzazioni nelle quali ha il ruolo di amministratore;
- un utente amministratore di una determinata organizzazione può modificare gli utenti che vi appartengono e i relativi ruoli;
- l'amministratore può modificare gli orari lavorativi dell'organizzazione (inizio - fine);
- l'amministratore può modificare le tipologie di assenza che gli utenti dell'organizzazione possono inserire;
- l'applicazione deve esssere cross-browser e cross-device.

3.3.2 Requisiti dello smart device

Il device è l'apparato intelligente (smart) che un'organizzazione può installare internamente al fine di automatizzare il processo di autenticazione dell'utente e apertura della porta, al momento della timbratura. I requisiti sono così sintetizzabili:

- il device ha come scopo principale l'autenticazione dell'utente e l'apertura della porta in caso l'accesso sia consentito;
- il device deve poter essere associato ad una sola ed unica organizzazione;
- il device deve preferibilmente demandare al server il compito di autenticazione utente. Deve sostanzialmente fare da tramite tra utente e server;

- la comunicazione con l'utente deve avvenire utilizzando tecnologie di comunicazione wireless in prossimità, ovvero NFC e BLE;
- la comunicazione con il server deve avvenire attraverso la rete internet;
- l'utente che utilizza il device per effettuare l'accesso, inserisce automaticamente nel sistema la timbratura;
- in condizioni normali, il device non avrebbe necessità di mantenere un database interno degli accessi, tuttavia deve essere prevista una politica di cache che gli consenta di funzionare anche in caso di connettività assente;
- le impostazioni di orario di funzionamento del device devono essere le stesse che gli amministratori impostano nel sistema per l'organizzazione;
- la comunicazione tra device e server deve essere sicura;
- il device deve funzionare anche con un token (o badge), in caso l'utente non abbia a disposizione uno smartphone o sia impossibilitato ad utilizzarlo;
- il device deve essere economico, sia dal punto di vista del costo iniziale, sia dal punto di vista del costo di mantenimento (consumo di energia elettrica).

3.3.3 Requisiti della smartphone app

L'applicazione per smartphone è uno dei due mezzi attraverso i quali l'utente può interagire con il device. I requisiti dell'applicazione sono molto ridotti:

- deve consentire all'utente di comunicare con il device al fine di aprire la porta e inserire automaticamente la timbratura nel sistema;
- la prima schermata che l'applicazione deve mostrare all'utente è quella di login, attraverso la quale può autenticarsi con le proprie credenziali;

- gli utenti che possono accedere nell'app sono gli stessi utenti che possono accedere alla web application. Non è possibile completare il login se non ci si è precedentemente registrati nella web application;
- l'applicazione deve consentire agli utenti registrati l'accesso attraverso le medesime modalità della web application (local authentication, google API);
- sarebbe preferibile evitare lo scambio di credenziali in chiaro direttamente con il device;
- l'applicazione deve consentire all'utente di scegliere quale tra le tecnologie di comunicazione disponibili utilizzare (NFC o Bluetooth LE).

3.4 I casi d'uso

Per una maggiore chiarezza, i casi d'uso dell'utente sono stati suddivisi in due differenti diagrammi. Nel diagramma in Figura 3.1 si mettono in evidenza i casi d'uso relativi agli utenti che utilizzano la web application. Si è scelto di utilizzare il concetto di estensione tra due differenti tipologie di attori per denotare la relazione che c'è tra utente semplice e amministratore: l'amministratore ha accesso a casi d'uso ulteriori, che riguardano la modifica dell'organizzazione su cui ha i diritti. Nel diagramma si cerca di mettere in evidenza il fatto che la prima operazione abilitante per tutti i successivi casi d'uso è la registrazione al servizio, infatti l'unico caso d'uso che vede come protagonista l'utente non autenticato, è il reset della password. Tutti gli altri casi d'uso sono vincolati al login, che a sua volta deve essere preceduto dalla registrazione.

Si può dedurre dal diagramma che i vari casi d'uso sono logicamente raggruppati in tre differenti blocchi, che non a caso riguardano le tre viste a cui gli utenti

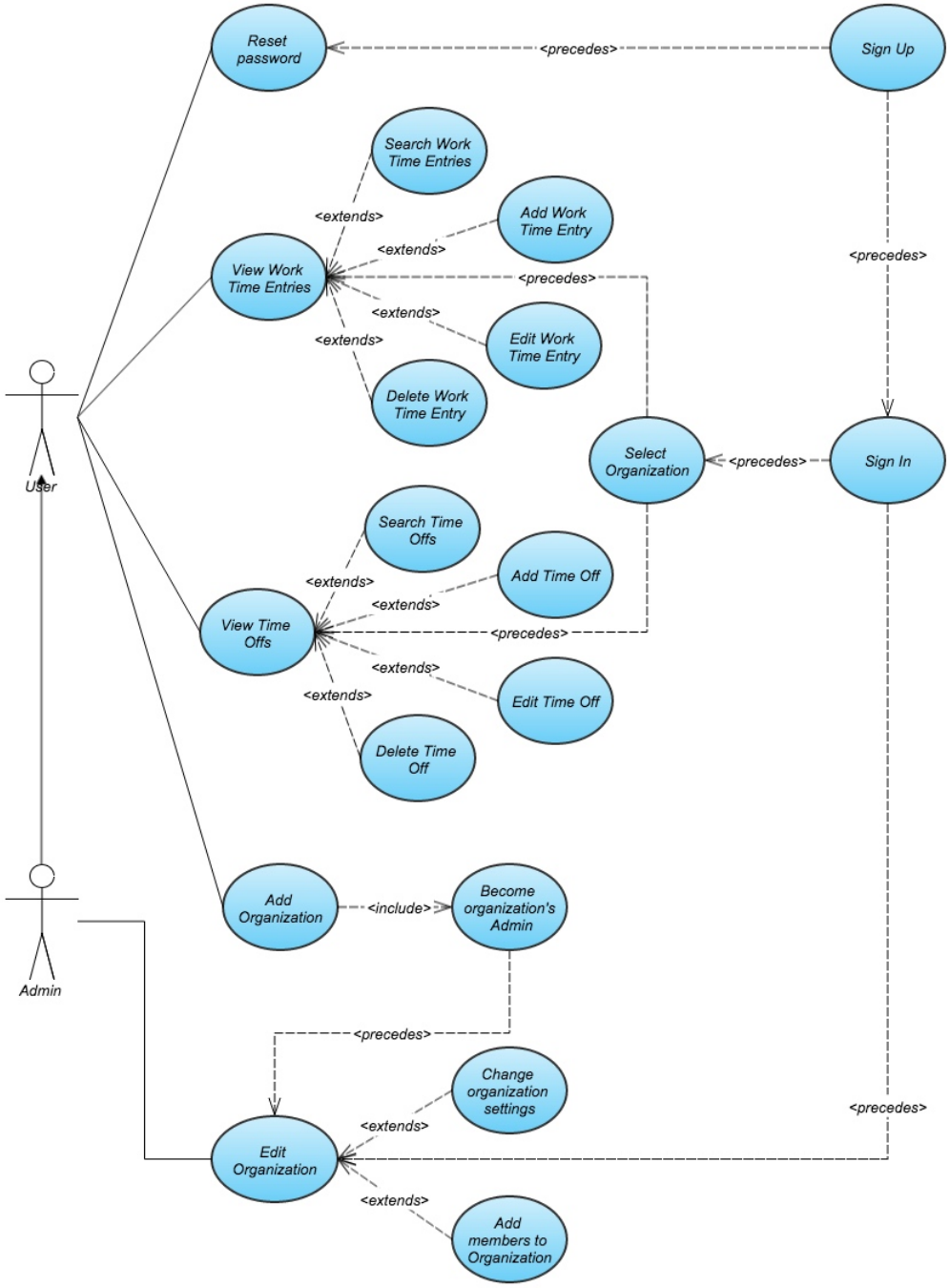

Figura 3.1: Casi d'uso dell'utente che utilizza l'applicazione web.

possono accedere all'interno della web application: timbrature, assenze e organizzazioni. Per quel che riguarda le timbrature e le assenze, agli utenti semplici è concesso pieno controllo soltanto sui record logicamente appartenenti ad essi: a differenza degli amministratori, non possono vedere nè modificare quelli di altri utenti appartenenti all'organizzazione. Sebbene le funzioni di modifica dell'organizzazione (aggiungere utenti, modificare orari di lavoro) siano riservate agli amministratori, nulla vieta ad un utente semplice di creare una nuova organizzazione, della quale sarà - almeno in fase iniziale - l'unico amministratore.

Il secondo blocco di casi d'uso (Figura 3.2) riguarda l'interazione tra utente e sistema per il tracciamento automatico della timbratura al momento dell'accesso in azienda. Come si può facilmente dedurre dall'insieme dei requisiti, le funzionalità necessarie in questo scenario sono piuttosto ridotte. In caso l'utente voglia effettuare l'accesso con l'utilizzo dello smartphone, il prerequisito necessario è il completamento del login verso il server. In questo modo potrà poi comunicare la propria identità e avviare il processo di autenticazione verso il device, utilizzando la tecnologia NFC o BLE. Se - e solo se - il device completerà l'autenticazione utente con successo, sarà automaticamente inserita nel server la timbratura e verrà aperta la porta. L'alternativa all'utilizzo dello smartphone consiste nell'uso di un tag NFC che il sistema associa in modo univoco all'utente.

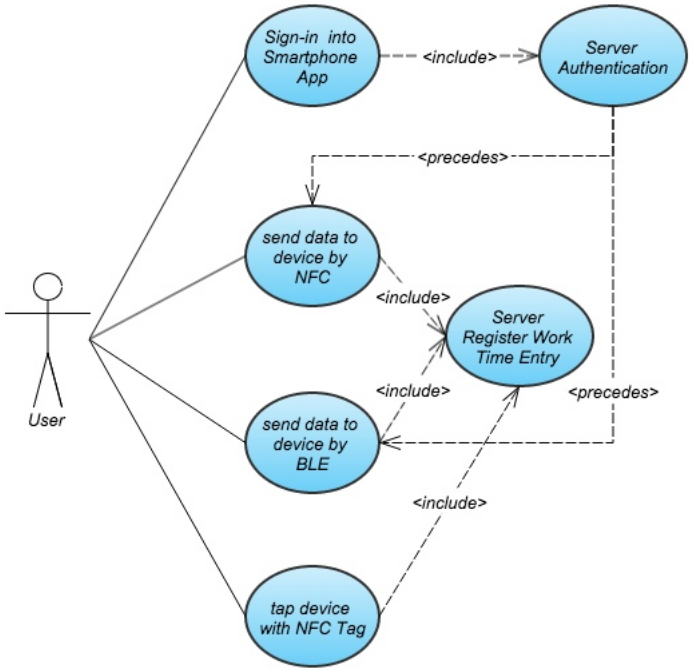

Figura 3.2: I casi d'uso dell'utente che interagisce con il device per l'accesso all'organizzazione.

3.5 Modello del Dominio

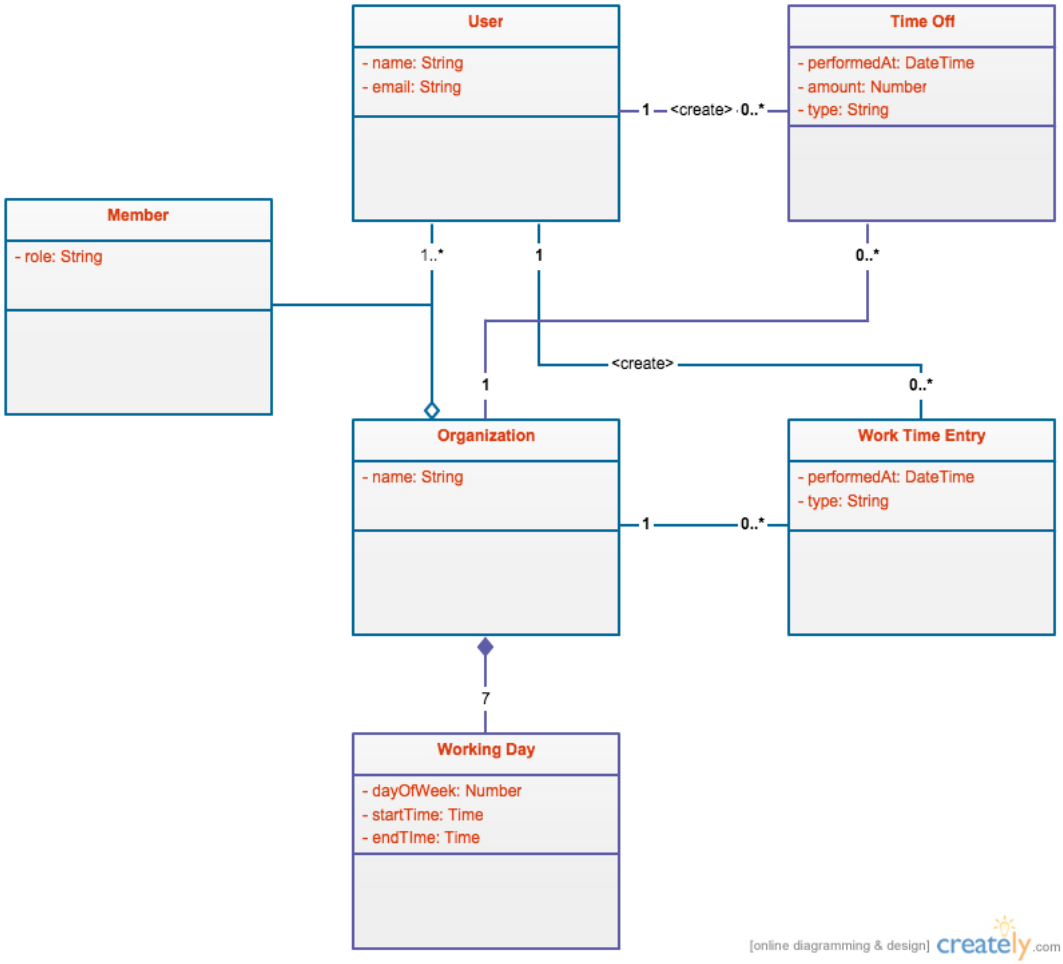

Figura 3.3: Modello del dominio applicativo

Il modello del dominio (Figura 3.3) mette in evidenza le relazioni che intercorrono tra le varie entità del dominio applicativo, e da esso può essere estratto il glossario del sistema. Non viene rappresentato il sistema distribuito dal punto di vista delle diverse componenti e della loro interazione, ma si cerca di focalizzare l'attenzione esclusivamente sulle astrazioni utilizzate per rappresentare la natura delle entità appartenenti al mondo reale coinvolte.

User L'utente che utilizza il sistema. Non è prevista l'interazione con utenti "anonimi", di conseguenza si assume che ogni utente del sistema sia registrato. L'utente rappresenta genericamente il concetto di persona, ed è identificato univocamente da un indirizzo email.

WorkingDay Questa entità è utile per caratterizzare la configurazione di una giornata lavorativa settimanale. Oltre agli orari di inizio e fine, è presente un attributo che rappresenta il giorno settimanale (numero 0 - 7) a cui l'entità si riferisce.

Organization L'organizzazione rappresenta il concetto di azienda, che consiste in un insieme di utenti con ruoli differenti. Oltre all'insieme di utenti, ogni organizzazione è composta da sette differenti caratterizzazioni di giornata lavorativa, una per ogni giorno della settimana.

Member Il member è la relazione che intercorre tra un utente e una determinata organizzazione. La relazione si può leggere come *utente x è membro dell'organizzazione y* e prevede un unico descrittore che identifica il ruolo dell'utente nell'organizzazione.

TimeOff Il *timeOff* non è altro che l'assenza giornaliera di un utente dal luogo di lavoro. Gli attributi che contraddistinguono l'assenza sono: la data, la tipologia di assenza e il numero di ore in cui l'utente è assente.

WorkTimeEntry Una *workTimeEntry* rappresenta il concetto di timbratura. Come l'assenza, anche la timbratura è associata ad un utente e ad un'organizzazione e, oltre al *timestamp* di data-ora nelle quali è avvenuta la timbratura, è classificata ulteriormente attraverso l'attributo tipo.

3.6 Architettura Logica

L'architettura del sistema distribuito può essere rappresentata in questa fase come diagramma di struttura (logica) per dare una visione complessiva di quali saranno le principali componenti dei diversi nodi presenti nel sistema (Figura 3.4). Non saranno ancora presi in considerazione i dettagli implementativi, ma ci si limiterà all'enfatizzazione di alcuni aspetti architetturali del sistema con l'obiettivo di separare le funzionalità e i compiti delle varie parti che lo andranno a comporre. Si utilizzerà la classificazione espressa da Jacobson [14] per suddivedere gli strati software di ogni componente in tre diverse dimensioni: Informazione, Controllo e Presentazione.

Server: Rappresenta il *core* di tutto il sistema, in quanto è il nodo dove risiedono i dati e soprattutto le logiche del modello del dominio. Il server viene interrogato da tutti gli attori presenti nel sistema, in ogni flusso interattivo previsto nei diversi scenari di utilizzo. Può essere ulteriormente suddiviso in una duplice maniera: dal punto di vista della stratificazione verticale, presenta tutti e tre gli strati software precedentemente introdotti. Lo strato di informazione è costituito dall'insieme delle entità rappresentanti i dati del dominio applicativo, lo strato di presentazione è costituito esclusivamente dalle pagine HTML, mentre il controller comprende tutti i moduli che sono utilizzati per l'interazione con i tre differenti client: la web application, il device e l'app per smartphone. A differenza di information e controller, lo strato di presentazione entra in gioco soltanto negli scenari messi in atto attraverso la web application.
Un'altra possibile suddivisione dello schema applicativo è quella che si ottiene dividendo l'architettura server per funzionalità. In questa fase di progettazione è emerso che è principalmente una la funzionalità utilizzata nell'interazione con tutti i differenti client, ovvero l'autenticazione. Per la web application l'autenticazione è il prerequisito principale a tutti i possibili scenari, basti pensare che le

uniche funzionalità accessibili per utenti "anonimi" sono la registrazione e il reset della password, che comunque resteranno parzialmente a carico del modulo dedicato all'autenticazione. Nell'interazione con il device l'autenticazione è usata una sola volta in fase iniziale per quel che riguarda il device stesso, mentre viene utilizzata per riconoscere l'utente ogni volta che effettua la timbratura. Infine, nella smartphone application, l'autenticazione server è utilizzata solo in fase iniziale per abilitare l'utilizzo delle funzionalità centrali dell'applicazione stessa ovvero la comunicazione in prossimità con il device.

Gli altri moduli funzionali che possono essere concettualmente isolati sono quello che comprende la web application e il controller che si occupa dell'interazione con gli smart device delle organizzazioni.

Device application Il device è il nodo che consente all'utente di effettuare l'accesso in azienda e di registrare in modo automatico la timbratura nel sistema, di conseguenza deve essere in grado di comunicare da una parte con l'utente (via NFC e BLE), dall'altra con il server attraverso internet. L'applicazione software prevede un unico strato orizzontale di "controller" in cui vanno a confluire tutti i moduli applicativi utilizzati: quello per il controllo della comunicazione con il server, quello per il controllo della comunicazione con l'utente e infine quello per l'attuazione, che dovrà essere utilizzato, ad esempio, per l'apertura della porta. Poiché si tratta di un modulo esclusivamente dedicato al controllo della comunicazione, non sono contemplati nell'architettura logica gli strati di *presentation* e *information*.

Smartphone application L'applicazione per smartphone è utilizzata dall'utente per effettuare l'accesso in azienda e tracciare automaticamente la timbratura. Orizzontalmente, sono riconoscibili il livello di presentation, ovvero la User Interface dell'applicazione, e il livello di controller, che a sua volta può essere ulteriormente suddiviso in controller di comunicazione con il device (NFC e BLE)

e controller di comunicazione con il server per l'autenticazione. Anche in questo caso, lo strato di information non è significativo.

In ognuno dei tre nodi brevemente descritti in precedenza non sono stati affrontati gli aspetti di maggior dettaglio, poiché saranno discussi in seguito. Per dare una visione generale delle interazioni che intercorrono tra le varie componenti strutturali fin qui considerate, possono essere consultati i diagrammi di sequenza UML in Figura 3.5 e Figura3.6.

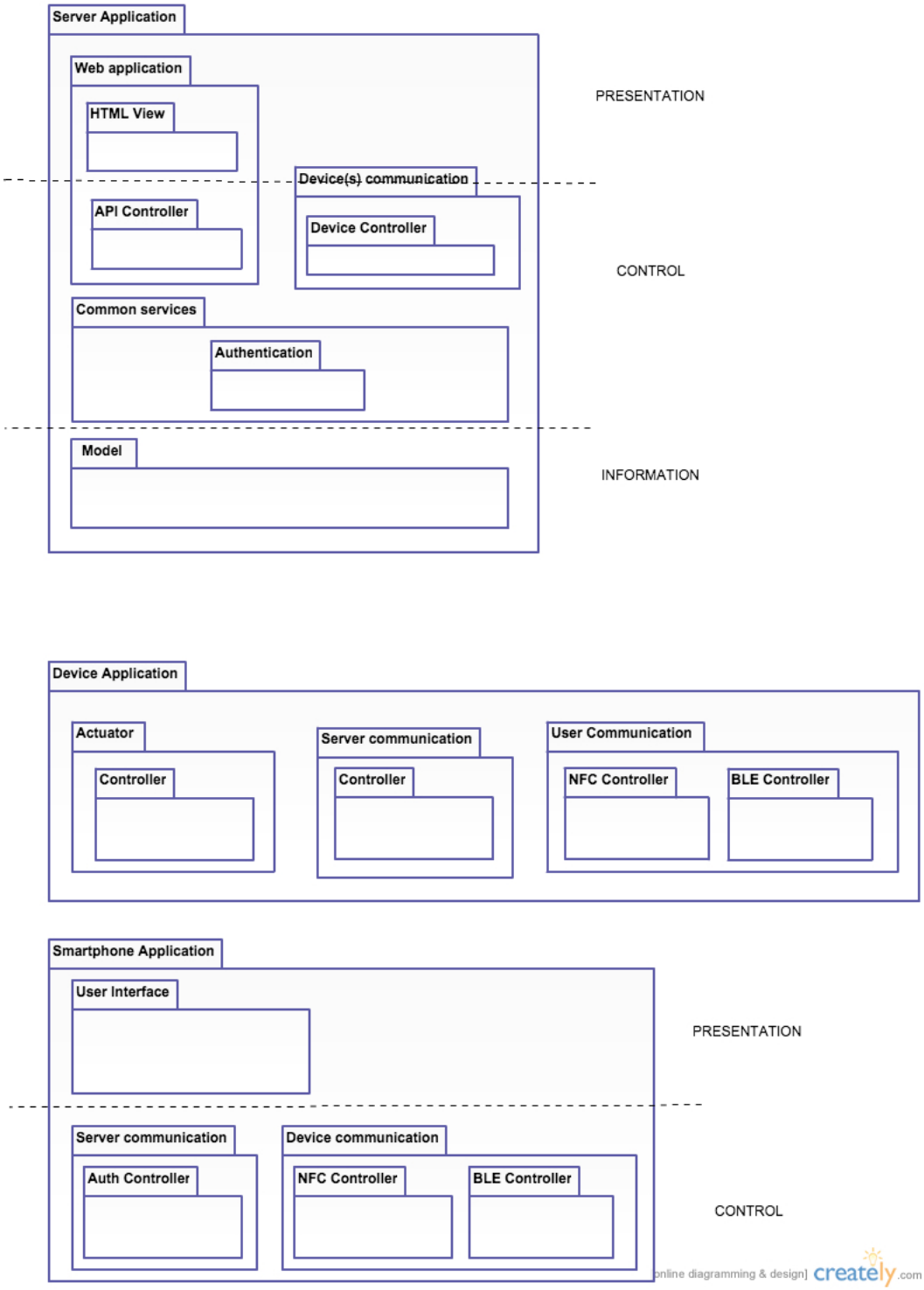

Figura 3.4: Architettura logica del sistema

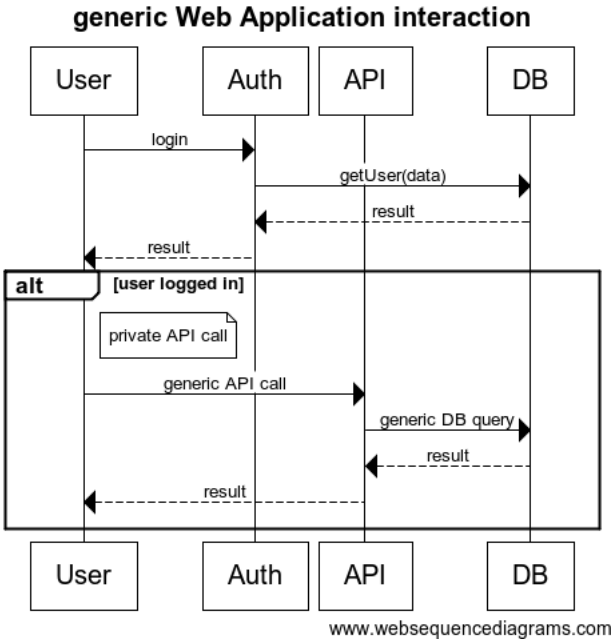

Figura 3.5: Diagramma di sequenza relativo al generico flusso interattivo tra i nodi del sistema coinvolti durante l'utilizzo della web application.

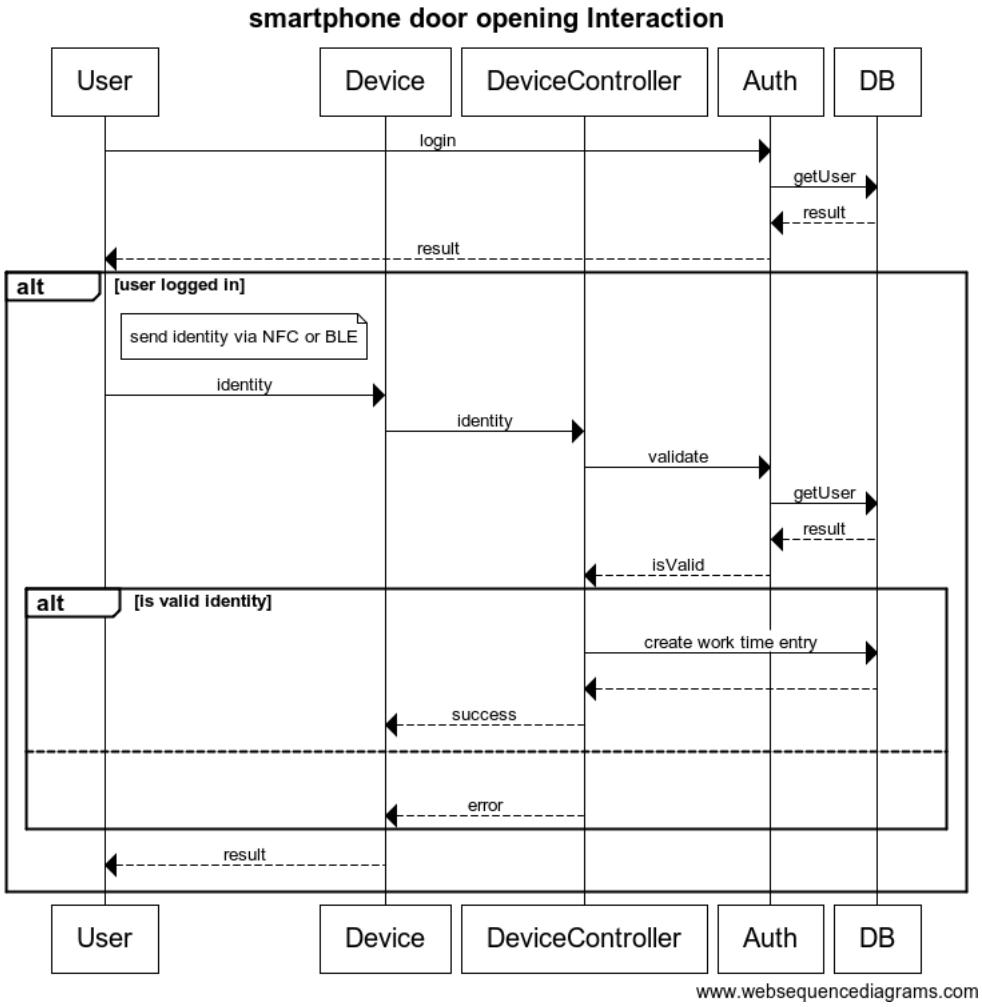

Figura 3.6: Diagramma di sequenza relativo al flusso interattivo tra i nodi del sistema al momento dell'apertura porta con l'utilizzo dello smart device.

Capitolo 4

Sviluppo della web application

In questo capitolo sarà discussa in modo più dettagliato la progettazione del nodo centrale del sistema, uno stack applicativo suddiviso in due differenti blocchi: il back-end per la parte server, ed il front-end per la parte client. Dopo una descrizione dei diversi framework e librerie utilizzati per la realizzazione del sistema, saranno discussi in modo più dettagliato soltanto gli aspetti principali legati alla progettazione e all'implementazione dei due moduli applicativi, tralasciando molti dei dettagli implementativi ritenuti secondari e che possono poi essere approfonditi direttamente analizzando il codice finale del progetto.

4.1 Tecnologie utilizzate

Scopo di questa sezione è l'identificazione e l'introduzione delle tecnologie utilizzate per l'implementazione della web application. Si è scelto di anticipare questa discussione rispetto ai paragrafi successivi di progettazione e sviluppo perchè lo studio delle diverse tecnologie utilizzate (di ultima generazione) ha richiesto una buona parte del tempo necessario alla realizzazione di questa tesi.

4.1.1 I requisiti delle moderne web application

Oggigiorno le web application rivestono un ruolo sempre più importante nel mondo dei servizi software, basti pensare al crescente numero di utenti che frequentano Facebook, Twitter piuttosto che Amazon, YouTube o Google. Ci si sta progressivamente muovendo verso nuovi modelli di business, in cui spesso i servizi sono offerti in modo gratuito agli utenti, in cambio di semplici informazioni che vengono successivamente rielaborate e utilizzate per campagne pubblicitarie e di marketing [7]. Per questo motivo, il valore di una web application è soprattutto dovuto al numero di utenti attivi, che devono essere costantemente invogliati e spinti ad utilizzare l'applicazione prevenendo quindi il rischio di abbandono.
Oltre alla natura del servizio offerto dall'applicazione, diventa quindi fondamentale per il suo successo il livello di esperienza utente, ovvero ciò che una persona prova nel momento in cui utilizza il prodotto o servizio. L'esperienza d'uso concerne gli aspetti esperienziali, affettivi, l'attribuzione di senso e di valore collegati al possesso del prodotto e all'interazione con esso, ma include anche le percezioni personali su aspetti quali l'utilità, la semplicità d'utilizzo e l'efficienza del sistema. Gli utenti che utilizzano e conoscono internet e il mondo delle web application sono sempre più esperti, e sono in grado di giudicare la qualità di un'applicazione sulla base di pochi minuti (o addirittura pochi secondi!) di navigazione e utilizzo della stessa. Ovviamente, se l'esperienza utente non è sufficientemente curata, molti utenti si sposteranno presumibilmente su un'altra applicazione analoga e abbandoneranno definitivamente quella che li ha delusi. Anche se non è oggetto principale di questa tesi, è quindi bene considerare e introdurre alcuni aspetti e caratteristiche di un'applicazione web, che vanno ad incidere direttamente (o indirettamente) sull'esperienza utente.

Performance: un buon livello di performance si ha quando non ci sono lunghi tempi di attesa tra l'azione richiesta dall'utente e il suo completamento. Nove

volte su dieci, il *collo di bottiglia* per quanto riguarda le performance di una web application è rappresentato dall'interazione con il server. Dato il grande numero di utenti che un server potrebbe dover servire contemporaneamente, si possono verificare condizioni in cui un alto numero di richieste implica un grosso carico di lavoro nel server (possibili cause: context switch, accesso al db, etc), che rallenta sensibilmente il tempo di risposta.

Scalabilità: per migliorare le performance in caso di aumento o diminuzione degli utenti, solitamente si cerca di realizzare un sistema scalabile, ovvero in grado di crescere o diminuire di "scala" in funzione delle necessità e delle disponibilità. Un sistema non è scalabile quando nonostante un aumento delle risorse a lui disponibili, non è in grado di migliorare le prestazioni, in particolar modo ciò avviene quando si ha a che fare con un collo di bottiglia. La scalabilità può essere di due diversi tipi: verticale quando si aumenta la potenza di calcolo di una singola macchina, mentre è orizzontale quando si aumentano il numero di macchine dedicate al sistema.

Performance e scalabilità sono tipicamente caratteristiche che dipendono dall'architettura di back-end del sistema, ma non dal front-end. Infatti, se qualche anno fa poteva verificarsi un sensibile calo di performance in seguito ad un carico troppo elevato nelle applicazioni lato client, oggi con l'aumento delle performance dei vari client e l'utilizzo di browser moderni è possibile sviluppare applicazioni client-side native (HTML, CSS, JavaScript) ricche dal punto di vista grafico ma molto leggere.

L'esperienza utente può anche essere fortemente condizionata da fattori e caratteristiche del front-end applicativo, quali ad esempio la responsività e l'usabilità.

Responsività: per responsività si intende la capacità di un'applicazione web di adattarsi al meglio alla risoluzione del browser che l'utente sta utilizzando. La

necessità delle web application responsive va di pari passo con il sempre più alto numero di differenti dispositivi utilizzati per fruirne i contenuti. Gli utenti non navigano soltanto con il PC, ma utilizzano spesso e volentieri device con diverso fattore di forma, come smartphones o tablets. A causa di ciò è bene prevedere un utilizzo dell'applicazione su tante diverse risoluzioni, al fine di rendere ottimale l'esperienza dell'utente sia quando utilizza un normale browser desktop, sia quando utilizza un device con risoluzione ridotta. Spesso si confonde il concetto di responsività con quello di adattività. La differenza principale è che in un design adattivo, a differenza di quello responsivo, vi sono delle precise risoluzioni "target" sulle quali viene esplicitamente sviluppato il layout del sistema.

Usabilità: l'usabilità è la capacità di un sistema di essere compreso, appreso e attraente per gli utenti durante una comune sessione di utilizzo. La normativa ISO/IEC 2001a [13] esprime l'usabilità in termini di:

- comprensibilità, che riguarda lo sforzo richiesto per capire il sistema;
- apprendibilità, riguarda lo sforzo necessario all'utente per imparare ad usare il sistema;
- utilizzabilità, che si riferisce allo sforzo richiesto all'utente per utilizzare le features del sistema attraverso i suoi controlli.

Connesso al concetto di usabilità vi è senza dubbio l'accessibilità, in particolar modo se l'applicazione deve poter essere utilizzata da persone portatrici di handicap, che necessitano quindi di browser speciali che devono sapere come interpretare al meglio il contenuto HTML delle pagine.

4.1.2 Lo stack applicativo

Un solido stack applicativo è alla base di qualsiasi applicazione web, e rappresenta l'insieme di tutte le tecnologie che vengono utilizzate assieme per fare

funzionare la web application. Non si tratta di un concetto nuovo, infatti già da parecchio tempo si ha a che fare con l'utilizzo di diversi stack, ognuno con diverse peculiarità. Tra i più comuni troviamo sicuramente lo stack *LAMP* (Linux, Apache, MySql e PHP/Python/Perl) e lo stack *WISA* (Windows, IIS, SqlServer e ASP.NET). Entrambi questi stack sono basati su web server classici e database relazionali, che ben si sposano con i framework server side utilizzati, tuttavia, probabilmente in conseguenza del periodo in cui gli stack in questione sono nati, non è incluso nell'acronimo dello stack un riferimento alla tecnologia client-side utilizzata. Sebbene siano di grande popolarità e offrano un buon framework a supporto dello sviluppatore, hanno alcuni noti svantaggi e punti deboli: sono platform dipendenti, richiedono una conoscenza di molti differenti framework/linguaggi, la gestione della scalabilità è un compito complesso e costoso.

Per questi motivi, si è scelto di utilizzare uno stack differente per la web application, **MEAN**. MEAN stack è acronimo di MongoDB, Express, Angular e Node.js, l'insieme di piattaforme open-source JavaScript (e JSON) based, utilizzate in simbiosi per lo sviluppo dell'intera web application. L'utilizzo dello stack MEAN ha diversi vantaggi: innanzi tutto si tratta di piattaforme molto utilizzate negli ultimi anni, si può contare su framework giovani, open source e in continuo sviluppo, ben supportati dalla community. In secondo luogo queste tecnologie (se ben sfruttate) spingono lo sviluppatore a mettere in campo tutte le best practices del mondo web, realizzando un prodotto che rispetta tutti i canoni di qualità delle moderne web application. Terzo vantaggio, ma non meno importante, risiede nel fatto che il processo di sviluppo è reso agile grazie alla seguente peculiarità: l'intero stack applicativo può essere istanziato sulla singola macchina utilizzata per lo sviluppo, dando modo allo sviluppatore di procedere in modo iterativo e costruire l'applicazione funzionalità dopo funzionalità.

4.1.3 Node.js

Il back end della web application è senza dubbio il nodo più importante di tutto il sistema, deve poter interagire con gli utenti che utilizzano l'applicazione attraverso il browser, ma deve anche gestire la comunicazione con l'insieme degli smart device aziendali connessi. Proprio in funzione del suo ruolo centrale, è l'unica entità presente nel sistema che può accedere direttamente alla base di dati.

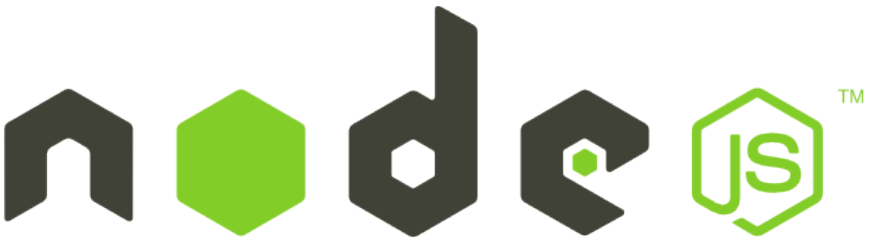

Figura 4.1: Node.js logo

La tecnologia scelta per l'implementazione del back-end è node.js. Node.js è un ambiente di sviluppo open-source e cross-platform per la realizzazione di applicazioni server-side, nato nel 2011. La piattaforma è basata sul JavaScript Engine V8, che è il runtime di Google utilizzato anche da Chrome, le applicazioni sono scritte utilizzando il linguaggio JavaScript e possono essere eseguite su tutti i sistemi operativi più comuni: OS X, Microsoft Windows e Linux [21].

Motore V8

Il motore V8 è una piattaforma open source sviluppata da Google per eseguire codice JavaScript in modo altamente performante. La peculiarità sta nel fatto che

il codice JavaScript non viene interpretato ed eseguito come bytecode, ma viene compilato e trasformato in linguaggio macchina prima dell'esecuzione. V8 può essere utilizzato come motore stand-alone, nel caso di node.js, oppure può essere incluso all'interno di applicazioni più complesse come ad esempio un browser web (Chrome).

Il modello asincrono event-driven

La caratteristica principale di Node.js risiede nella possibilità che offre di accedere alle risorse del sistema operativo in modalità asincrona event-driven, e non sfruttando il classico modello basato su processi o thread concorrenti utilizzato dai comuni web server. Node.js utilizza un event-loop per gestire tutte le operazioni asincrone: nel momento in cui l'applicazione necessita di eseguire un'operazione bloccante, il task e la relativa callback vengono inviati nella coda dell'event-loop e il programma continua la propria esecuzione, servendo le richieste successive. Nel momento in cui il thread delegato dal sistema completa l'operazione richiesta, l'event loop riceve notifica ed esegue la callback precedentemente registrata dall'applicazione. Questa caratteristica consente di gestire in modo performante un grande numero di operazioni (connessioni, operazioni sul DB, operazioni su disco), delegandone l'esecuzione all'event-loop che riesce ad ottimizzarla. Sviluppare applicazioni di networking diventa quindi molto più semplice per gli sviluppatori, che non devono preoccuparsi dei complessi scenari che la concorrenza implica. Oltre ad essere effettivamente di più semplice gestione, questa tipologia di modello è generalmente ritenuta più efficiente nei casi di elevato traffico di rete, perchè consente al sistema di accodare e gestire una per una le numerose richieste degli utenti nel contesto di un singolo processo, a differenza dei modelli multi thread che (almeno nelle versioni rudimentali) gestiscono milioni di richieste con altrettanti differenti processi o threads, incrementando notevolmente l'utiliz-

zo delle risorse dovuto al context switch. Bisogna comunque tenere presente che nonostante la sua natura single-process, il server node.js può scalare facilmente con l'obiettivo di utilizzare un maggior numero di core. Per questa necessità, sono presenti delle API native nel framework che consentono di lanciare ulteriori processi e gestire la comunicazione tra di essi.

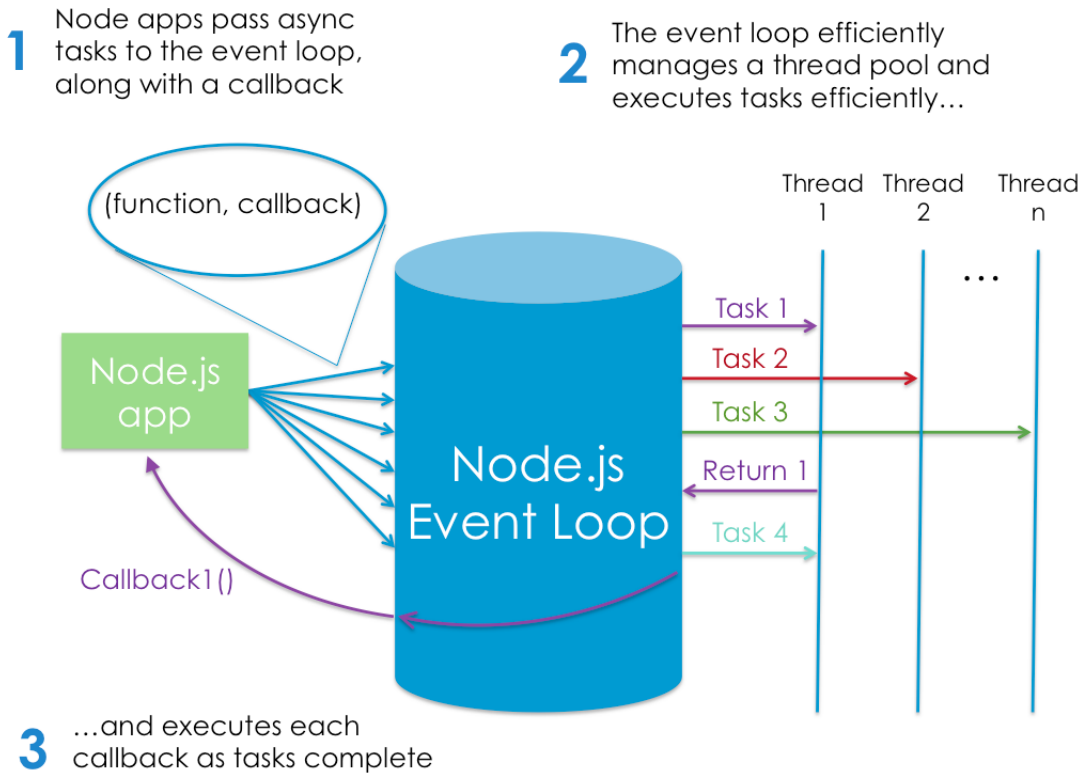

Figura 4.2: Uno schema del funzionamento event driven di node.js

Node package manager

Node package manager (npm) è il sistema di gestione pacchetti e dipendenze di node.js, e dalla versione 0.6.3 è installato in modo automatico con node.js. Npm consente di gestire in modo semplice tutte le dipendenze di un'applicazione node.js e, unito al repository node registry, rappresenta un grande vantaggio

per gli sviluppatori che possono utilizzare migliaia di moduli applicativi più o meno complessi, già sviluppati, testati e recensiti da altri utenti. La filosofia di npm è molto simile a quella che spinge gli sviluppatori all'utilizzo di librerie e framework: non sempre è necessario "reinventare la ruota", soprattutto quando si può contare sulla sicurezza di librerie software open source ben sviluppate e testate.

4.1.4 MongoDB

MongoDB [17] è l'applicazione che corrisponde alla prima lettera dell'acronimo MEAN, ed è un DBMS non relazionale orientato ai documenti. Per la sua ottima compatibilità con il modello asincrono di node.js, la flessibilità che garantisce in fase di sviluppo e le caratteristiche dell'applicazione che si vuole sviluppare, mongoDB può essere considerato una buona scelta come base di dati principale del sistema. Classificato come un DBMS di tipo NoSQL, MongoDB si allontana

Figura 4.3: mongoDB

dalla struttura tradizionale basata su tabelle dei DBMS relazionali, in favore di un'architettura basata su documenti in stile JSON con schema dinamico (MongoDB chiama il formato BSON). Si tratta di un progetto open source nato nel 2007, e ad oggi è senza dubbio il DBMS non relazionale più utilizzato nel mondo [10].

Caratteristiche principali

Le caratteristiche principali di mongoDB sono:

- *Document-oriented*; anzichè spezzare un'entità in tante diverse strutture relazionali, MongoDB dà la possibilità di salvare l'intera entità in un numero minimo di "documents", allo scopo di rendere più rendere più semplice ed intuitivo l'utilizzo dell'entità stessa.

- *Query ad-hoc*; mongoDB supporta la ricerca attraverso campi, query di range, regular expressions. Le query possono ritornare specifici campi di documenti e anch'esse includere funzioni JavaScript custom.

- *Indicizzazione*; tutti i campi possono essere indicizzati.

- *Replicazione*; mongoDB garantisce alta disponibilità dei dati attraverso l'utilizzo di *replica-set*. Un replica-set è composto da due o più copie dei dati. La funzione di replicazione è disponibile built-in, ed è attraverso essa che mongoDB mantiene le copie di replica aggiornate rispetto alla primaria. Quando la copia primaria fallisce, viene eletta una delle repliche.

- *Bilanciamento di carico*, utilizzato da mongoDB per scalare orizzontalmente, facendo uso di sharding. Lo sharding è la separazione dei dati in "fette", che si ottiene definendo a priori i range di una determinata chiave di sharding. Ogni fetta può essere mantenuta in un diverso nodo computazionale, bilanciando quindi il carico.

- *GridFS* è una funzionalità inclusa in mongoDB che consente di poter salvare grosse quantità di dati in *chunks* separati, evitando così di ingigantire la dimensione di un singolo documento.

- *Aggregazione*; attraverso funzioni come quella di map-reduce, è possibile effettuare interrogazioni e aggregazioni simili ai raggruppamenti sql-like (GROUP BY).

- *Capped collections*, ovvero insiemi di documenti indicizzati e di dimensione fissa, molto utili se si vogliono migliorare le performances nel caso si abbia a che fare con dati in sola lettura.

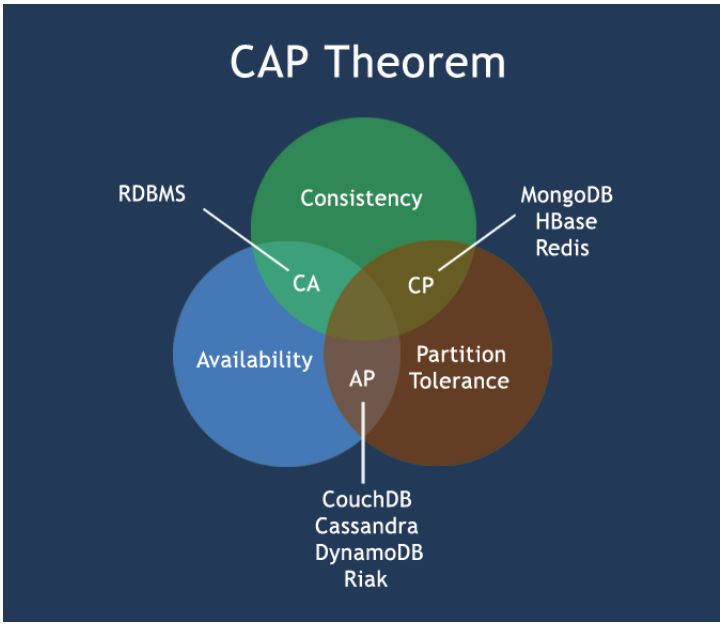

Figura 4.4: Collocazione teorica di mongoDB nel teorema CAP.

Se installato come servizio in produzione, con tutte le diverse istanze distribuite dal provider, mongodb garantisce Consistency e Partition Tolerance. Secondo il teorema CAP [8], non è possibile avere anche la garanzia di Availability, tuttavia mongodb offre ”High Availability” utilizzando le repliche, in particolare il concetto di replica set.

Pro e Contro rispetto ad un modello relazionale

Spesso e volentieri, si tende ad usare mongoDB senza considerare realmente quali siano i pro e i contro rispetto ad un tradizionale modello relazionale. È bene quindi capire quali siano i vantaggi che offre [18]:

Big data. Grazie alla funzionalità di sharding, è semplificata la gestione di grandi moli di dati, bilanciando il carico.

Flessibilità. A differenza dei RDMBS, mongoDB dà la possibilità di avere strutture dati differenti per i diversi dati all'interno di una singola collezione. Questo consente di risparmiare spazio e di gestire facilmente eccezioni.

Velocità. Il tempo di esecuzione delle query può essere tipicamente molto più basso rispetto a quelle dei tradizionali db relazionali, a patto che il modello e la query siano strutturati in modo corretto, e non si faccia lookup su tanti documenti o collezioni diverse.

Gli svantaggi rispetto ad un normale RDBMS, sono i seguenti:

No JOIN. Senza dubbio si tratta dello svantaggio principale nell'utilizzo di mongoDB. In caso ci si renda conto di avere a che fare con un modello relazionale complesso, in cui le query vedono coinvolte diverse tabelle, bisogna ripensare alla struttura dei dati orientandosi (ove possibile) all'uso di documenti innestati.

No transazioni. Un altro importante problema è legato all'impossibilità di gestire le transazioni, poiché le uniche operazioni atomiche per mongoDB, sono quelle su di un singolo documento. Non è quindi indicato ove vi sia necessità di complesse transazioni, come ad esempio nei sistemi di pagamento.

Utilizzo di memoria. A parità di carico rispetto ad un tradizionale db relazionale, MongoDB utilizza tendenzialmente più RAM, in quanto, data la non necessaria consistenza della struttura dei dati all'interno di una collection, deve mantenere in memoria l'insieme delle chiavi.

Problemi dovuti alla concorrenza. MongoDB utilizza un readers-writers lock, che permette letture concorrenti all'interno di un db, ma dà accesso esclusivo al DB in caso di singola scrittura. Inoltre, in caso di processi in attesa, viene sempre data precedenza alle operazioni in scrittura. Ciò comporta qualche problema nel momento in cui vi è un numero molto alto di scritture in contemporanea, che tuttavia può essere parzialmente risolto sia attraverso l'uso di piattaforme server asincrone single-process (node.js), sia grazie alla feature di sharding.

4.1.5 Express.js

Express.js è il framework sfruttato da MEAN per lo sviluppo del web server su node.js. Si tratta di una piattaforma che ha come obiettivo primario quello di fornire allo sviluppatore utili strumenti e funzionalità per realizzare solide API. Express.js si ispira al framework *Sinatra* per *Ruby*, ed è diventato in pochi anni la piattaforma maggiormente utilizzata per creare applicazioni web su node.js.
Queste alcune tra le sue principali caratteristiche:

- meccanismo di routing robusto e centrale, estendibile attraverso funzioni di middleware;
- adatto alla creazione di API RESTful come interfaccia applicativa;
- logica minimalista, express è un middleware leggero che sfrutta al meglio le caratteristiche vincenti di node.js;
- permette l'utilizzo di templating server-side per il rendering delle pagine HTML;
- permette la fruizione di contenuti statici ai client.

Creazione di API RESTful

Si è precedentemente affermato che express.js è adatto alla creazione di API RESTful. Che cosa significa API RESTful? Il termine REST (representational state transfer) venne introdotto nel 2000 da Roy Fielding, uno dei creatori di HTTP, e si riferisce ad un insieme di principi di architetture di rete che definiscono come le risorse sono definite e indirizzate.
REST prevede che la scalabilità del Web e la crescita siano diretti risultati di pochi principi chiave di progettazione:

- lo stato dell'applicazione e le funzionalità sono divisi in risorse web;
- ogni risorsa è unica e indirizzabile con una sintassi universale;
- tutte le risorse sono condivise attraverso un'interfaccia uniforme;
- il protocollo deve essere client-server, stateless, cacheable e a livelli.

Tradotto in modo più schematico e sintetico, l'accesso ad una determinata risorsa web deve essere effettuato attraverso URL semantici e l'uso coerente dei metodi messi a disposizione dal protocollo HTTP. Se considerassimo ad esempio una generica entità thing come risorsa web REST, potremmo avere, per esempio, queste API:

GET	/thing	richiesta di tutte le risorse thing.
POST	/thing	creazione di una nuova risorsa thing.
GET	/thing/:id	richiesta della risorsa thing identificata da id.
PUT	/thing/:id	modifica della risorsa thing identificata da id.
DELETE	/thing/:id	eliminazione della risorsa thing identificata da id.

Express.js, attraverso il suo meccanismo di routing, consente allo sviluppatore di concentrarsi sull'interfaccia delle API che occorre creare, pensando soltanto

in un secondo momento all'implementazione. Apparentemente REST, nella sua versione più semplice, non esplicita in alcun modo meccanismi e protocolli di sicurezza. In realtà, tramite l'utilizzo di funzioni di middleware, express può validare ogni singola richiesta HTTP verso le API, onde evitare ad esempio l'utilizzo di API private da parte di utenti non autorizzati.

In seguito si riporta un piccolo snippet di codice che aiuta a comprendere quanto sia semplice l'utilizzo di express per creare API rest, e relativi middleware. Dei moduli thingsProvider e authService è tralasciata l'implementazione.

```
var express = require('express');
var app = express();
var thingsProvider = require('./thingsProvider');
var authService = require('./authService');

function isAuthenticated(req, res, next) {
  //auth logic
  if(authService.authenticate(req)){
    next();
  }else {
    res.send(403);
  }
}

//No authentication middleware function
app.get('/thing', function (req, res) {
  //Gets all the things
  res.json(thingProvider.list());
})
```

```
//No authentication middleware function
app.get('/thing/:id', function (req, res) {
  //Gets all the things
  res.json(thingProvider.get(req.params.id));
})

//No authentication middleware function
app.post('/thing', function (req, res) {
  thingProvider.create(req.body);
  res.send(200);
})

//Authentication middleware!
app.put('/thing/:id', isAuthenticated, function (req, res) {
  thingProvider.update(req.params.id, req.body);
  res.send(200);
})

//Authentication middleware!
app.delete('/thing:id', isAuthenticated, function (req, res) {
  thingProvider.delete(req.params.id);
  res.send('Got a DELETE request at /user');
})

var server = app.listen(8080);
```

4.1.6 Angular.js

Angular.js è un framework open source sviluppato da Google con lo scopo di assistere lo sviluppatore nella realizzazione di single page applications, ovvero applicazioni client-side costruite su una singola pagina HTML con JavaScript e CSS. L'obiettivo di angular.js è quello di semplificare lo sviluppo e il testing dell'applicazione, proponendo il pattern Model View Controller come architettura di base.

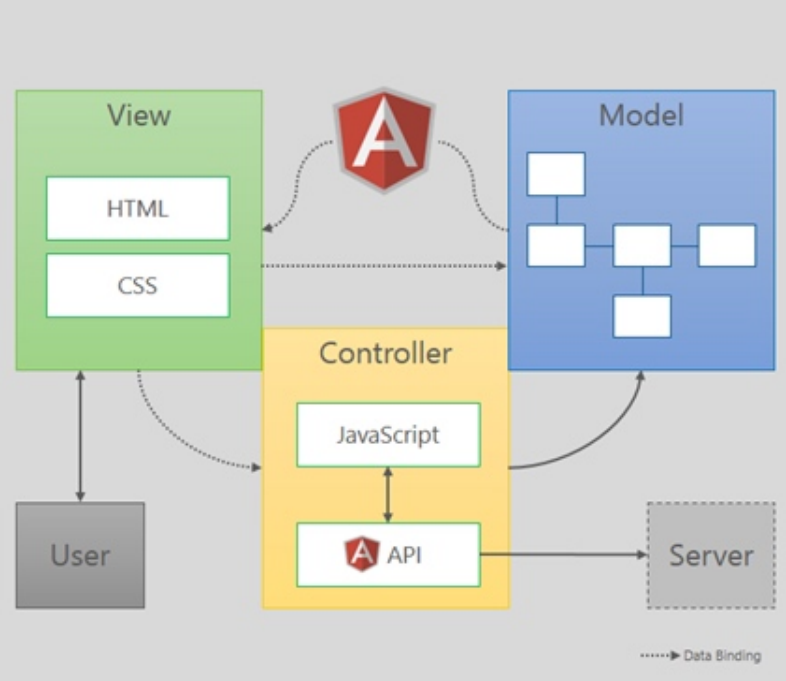

Figura 4.5: Architettura MVC di un modulo angular.js

I concetti di base

Angular è una piattaforma nata per le applicazioni web di matrice *CRUD*, nelle quali quindi si fanno operazioni di *create, read, update, delete* di generiche risorse. In questo caso, il livello di astrazione intrinseco di angular non è limitante per lo

sviluppatore, anzi ne semplifica i compiti. La piattaforma non è però adatta in tutti i casi dove è necessaria una forte manipolazione del DOM, come ad esempio giochi o editors, in quanto lo sviluppatore si troverebbe costantemente costretto a cercare di scavalcare il framework per poter interagire liberamente con il DOM. Lo spirito di Angular.js è riassumibile in pochi punti:

- è bene disaccoppiare la manipolazione del DOM dalla logica applicativa;
- è bene trattare il testing dell'applicazione quanto il suo sviluppo, in quanto la difficoltà di testing dipende molto da come viene scritto il codice;
- è bene disaccoppiare l'applicazione client da quella server: in questo modo lo sviluppo può procedere in parallelo;
- il framework deve guidare lo sviluppatore dal momento in cui viene disegnata l'interfaccia, alla scrittura della business logic, fino al momento del testing;
- è sempre una buona cosa rendere i task comuni triviali e i task complessi possibili.

Alla base della filosofia di Angular.js: directives

Molto spesso, durante lo sviluppo di applicazioni web dinamiche, ci si rende conto di quanto il linguaggio HTML non sia nativamente adatto per gestire la complessità di logiche applicative che compaiono in un qualsiasi sito web che non sia statico. Per risolvere questo problema molto spesso si fa uso di framework o librerie che guidano e aiutano lo sviluppatore nel seguire una certa struttura per l'organizzazione dei file, piuttosto che nell'usare determinate funzioni di utility. Angular.js ha un approccio differente, perchè cerca di minimizzare il gap tra l'HTML classico e quello che l'applicazione deve realmente fare, utilizzando

nuovi costrutti HTML: angular insegna al browser una nuova sintassi attraverso il costrutto “directive”. Alcune directive tra le più comuni sono:

- Data binding, nella forma `{}`.
- Strutture di controllo del DOM per nascondere/replicare codice HTML.
- DOM event handling dichiarativo.
- Supporto di form e validazione.

Overview

Alla base di un modulo angular.js ci sono l’architettura MVC e il data-binding (a due vie) del model nella view. Solitamente, la view è un blocco di HTML che utilizza tante diverse direttive, il model è un insieme generico di dati, mentre il controller è un file JavaScript in cui sono definite le logiche di business del modulo. Angular non prevede obbligatoria la presenza di un controller, ma è assai difficile che capiti di dover sviluppare un modulo privo di una qualunque logica di controllo. Per questo motivo, nell’esempio che segue, sarà rappresentato un modulo completo anche di controller.
Questo esempio, nonostante la sua semplicità, cerca di chiarire i meccanismi alla base della piattaforma angular isolando gli attori in gioco e analizzandone i relativi ruoli.

```
<div ng-app="invoice1" ng-controller="InvoiceController
    as invoice">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="invoice.qty"
      required >
```

```
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="invoice.cost"
      required >
    <select ng-model="invoice.inCurr">
      <option ng-repeat="c in invoice.currencies">{{c}}</option>
    </select>
  </div>
  <div>
    <b>Total:</b>
    <span ng-repeat="c in invoice.currencies">
      {{invoice.total(c) | currency:c}}
    </span>
    <button class="btn" ng-click="invoice.pay()">Pay</button>
  </div>
</div>
```

Analizzando l'esempio riportato sopra, si può notare quanto somigli a normale codice HTML, con qualche elemento di markup diverso dal solito. In Angular, un file come questo è chiamato *template*. Quando Angular lancia l'applicazione, effettua il parsing del codice HTML presente all'interno del template, lo processa utilizzando il proprio *compiler* e infine renderizza il DOM risultante, comunemente chiamato view.

Il primo esempio di nuovo elemento di markup sono le "directives", che applicano un comportamento speciale agli attributi o elementi presenti nell'HTML. All'interno del template sono usate diverse direttive: `ng-app` ed `ng-controller` definiscono rispettivamente il modulo e il controller collegati al template. La parola chiave `as` utilizzata internamente alla direttiva del controller, esplicita il fatto

che l'istanza del controller è mantenuta nella variabile che segue `as` (in questo caso invoice) all'interno dello scope.
In più punti è dichiarato il binding a due vie di un elemento della view con il model, attraverso la direttiva `ng-model`, mentre `ng-click` è utilizzato per agganciare l'evento di click con il relativo handler, che sarà poi definito nel controller. Infine, viene utilizzata la direttiva `ng-repeat` per replicare il pezzetto di HTML di un'option della select e anche per mostrare nella view il totale in tutte le differenti valute.
La seconda tipologia di markup utilizzata nel template è costituita delle doppie parentesi graffe `{{ expression | filter }}`: quando il compilatore incontra questo genere di markup, lo rimpiazza dopo aver completato la valutazione del codice inserito all'interno delle graffe. Una "expression" è uno snippet di codice JavaScript-like che permette di leggere o scrivere le variabili nello scope della view. L'esempio contiene anche un "filter", ovvero un formattatore del valore di una determinata espressione da mostrare all'utente.

```
angular.module('invoice1', [])
  .controller('InvoiceController', function() {
    this.qty = 1;
    this.cost = 2;
    this.inCurr = 'EUR';
    this.currencies = ['USD', 'EUR', 'CNY'];
    this.usdToForeignRates = {
      USD: 1,
      EUR: 0.74,
      CNY: 6.09
    };
```

```
    this.total = function total(outCurr) {
      return this.convertCurrency(this.qty * this.cost,
       this.inCurr, outCurr);
    };

    this.convertCurrency = function(amount, inCurr, outCurr) {
      return amount * this.usdToForeignRates[outCurr] /
       this.usdToForeignRates[inCurr];
    };

    this.pay = function pay() {
      window.alert("Thanks!");
    };
  });
```

In questo file JavaScript è definita la funzione costrutture usata da angular per istanziare il "controller". Lo scopo dei controller è quello di esporre variabili e funzionalità alle espressioni e alle directives. Nel controller sono definiti sia i dati del model che gli handler degli eventi, in entrambi i casi il binding è effettuato in modo dichiarativo sulla view.

Per concludere, la cosa più importante che viene mostrata nell'esempio è il modo in cui sfruttare il meccanismo di *live-binding* offerto dalla piattaforma angular: quando cambia il valore del tag input, i valori delle espressioni sono automaticamente ricalcolati e il DOM è aggiornato contestualmente. Il concetto sotteso a questo comportamento è il binding a due vie.

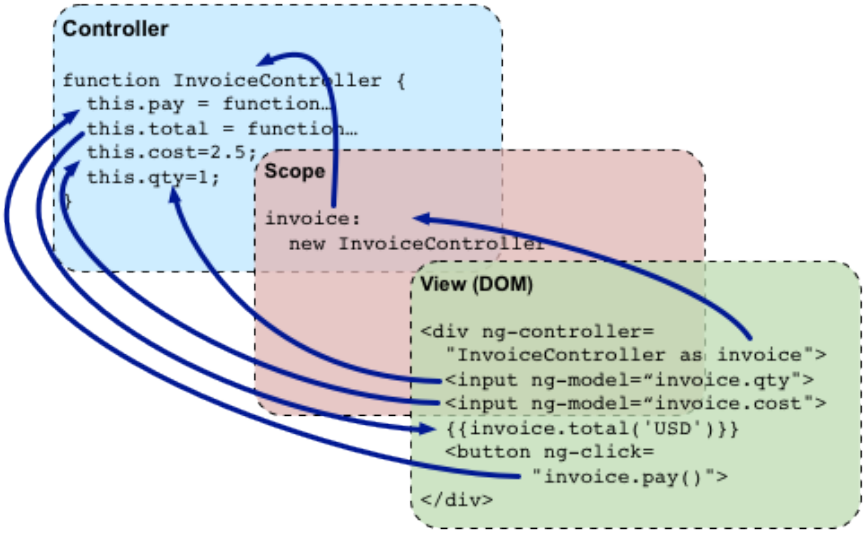

Figura 4.6: Live binding a 2 vie di angular.js

4.1.7 Approccio Agile: bootstrapping del full-stack applicativo

Uno dei requisiti fondamentali nel processo di sviluppo di applicazioni web è la possibilità di sviluppare velocemente applicazioni di qualità. A questo scopo node.js offre alcuni moduli che possono aiutare lo sviluppatore a generare una struttura iniziale per il progetto (*bootstrapping*), che può essere poi facilmente estesa o modificata a seconda delle esigenze. Se il bootstrapping dell'applicazione parte da una buona base, si hanno già disponibili molti dei servizi e delle infrastrutture utilizzate nella maggior parte delle applicazioni, come ad esempio il servizio per l'autenticazione utente, la struttura delle API, la suddivisione architetturale di client e server in MVC. Oltre a tutto questo, spesso il template usato per la generazione sceglie già gran parte dei framework e delle piattaforme lato client e lato server, basandosi sui "best-seller" presenti nel mondo del web, come ad esempio il MEAN stack.

Yeoman

Nel nostro caso il *tool* usato per la generazione dell'architettura di base per il progetto, è **Yeoman**. Yeoman è una utility node.js composta da tre differenti pacchetti, ognuno con differenti funzioni: *yo*, utilizzato per generare lo scheletro applicativo sulla base di un template, *bower* per gestire le dipendenze dell'applicazione client, e *grunt* che tra le altre funzionalità, permette anche di lanciare un'istanza dell'intera applicazione in locale. Yo offre allo sviluppatore la possibilità di scegliere tra tanti differenti generatori già ampiamente utilizzati e recensiti, al posto di crearne direttamente uno nuovo. Dato che l'applicazione web che vogliamo realizzare è piuttosto standard, si è scelto di utilizzare il più comune generatore MEAN, per essere certi di mantene aderenza agli standard e alle *best practices* in voga nel mondo dello sviluppo web: **angular-fullstack-generator**.

Il generatore: angular-fullstack-generator

Come anticipato, il generatore permette di fare *bootstrapping* applicativo di un'applicazione basata sul MEAN stack. L'architettura iniziale proposta dal generatore è piuttosto semplice: per prima cosa c'è una netta suddivisione tra l'applicazione angular e l'applicazione node, che vengono separate in due differenti cartelle, rispettivamente client e server. Per ognuna delle due applicazioni le librerie utilizzate sono definite come dipendenze all'interno di due differenti file JSON: package.json include tutte le dipendenze dell'applicazione node, mentre bower.json quelle per l'applicazione angular. Il generatore utilizza nativamente molte librerie e componenti. Le principali sono, per la parte server:

- **express middlewares**. I middlewares sono utilizzati da express.js per arricchire le proprie capacità. Le librerie `body-parser` `cookie-parser` sono

utilizzate ad esempio per consentire la lettura e la manipolazione dei cookie e del json inviato nel body delle request.

- **jsonwebtoken**. Il modulo è utilizzato per la gestione dell'autenticazione *token-based*, attraverso la quale il server garantisce l'accesso ai client autenticati tramite una stringa cifrata scambiata come stringa JSON.

- **passport.js**. Passport è usato sempre più spesso quando occorre gestire l'autenticazione. Supporta sia l'autenticazione basata su un database locale, che l'autenticazione da un provider di terze parti con protocollo oAuth (Google, Twitter, Facebook, etc). All'interno dell'applicazione possono essere gestite le varie forme di autenticazione sotto forma di differenti "authentication strategies", ognuna delle quali supporta parametri ad-hoc per la configurazione sulla base del provider utilizzato.

- **mongoose**. Mongoose è un Object Data Manager (ODM) per l'interazione con la base di dati, che nel sistema è appunto un'istanza di mongoDB. Mongoose dà la possibilità di modellare semplicemente gli schemi delle differenti entità utilizzate nell'applicazione, definendo i tipi di dato e i vincoli sui differenti campi. Oltre a questo fornisce un insieme completo di API non soltanto per fare le operazioni di CRUD sulle entità, ma anche per definire metodi di istanza o statici che possono essere utilizzati per arricchire le funzionalità del modello dell'entità rappresentata.

- **socket.io**. È una libreria che consente di sfruttare ad alto livello il protocollo di comunicazione Web Socket, basato sullo scambio di messaggi. Il modulo server prevede ovviamente un corrispettivo client, che deve essere implementato nella web app per la ricezione delle notifiche in tempo reale. Il modulo sarà inoltre usato per la comunicazione con il device.

Oltre al modulo client di socket.io, i principali componenti utilizzati dal generatore in aggiunta ad angular sono:

- **Bootstrap**. Bootstrap è uno dei componenti più popolari per la realizzazione di applicazioni responsive: non si tratta di una libreria Javascript, bensì di un framework CSS estendibile, che offre un'ottima base di partenza per lo sviluppo di pagine web HTML. Ha la caratteristica di essere responsivo e dà la garanzia che le applicazioni web sviluppate seguendo i suoi principi base siano fruibili anche da dispositivi mobili.

- **Bootstrap-UI**. Questo modulo, sviluppato dai creatori di angular.js, contiene un'ampia gamma di *directives* angular per la creazione e gestione di componenti grafici complessi implementati seguendo i princìpi di Bootstrap, come ad esempio datepickers, timepickers o dropdown lists.

4.2 Progettazione delle API

Le API che l'applicazione express deve fornire al client angular dovranno avere le seguenti caratteristiche: in primo luogo la firma deve essere REST-oriented e semanticamente significativa, in quanto rappresenta l'interfaccia del metodo che si andrà a sfruttare dal client, in secondo luogo occorrerà proteggere adeguatamente l'utilizzo delle API utilizzando opportuni meccanismi di autenticazione e autorizzazione.

4.2.1 Definizione dell'URL come interfaccia REST

La struttura dell'URL è ciò che definisce la semantica delle API REST. Si sono fatte le seguenti scelte:

- per distinguere le API dagli assets e dalle altre risorse (pagine HTML) che il server mette a disposizione, si è scelto come prima cosa l'utilizzo della preposizione `/api/` come prima parte dell'URL;
- il secondo parametro dell'URL è rappresentato dal tipo di risorsa a cui si vuole dare accesso, che corrisponde quindi ad una particolare entità di dominio;
- il terzo parametro dell'URL è l'identificativo univoco della risorsa (se necessario per il metodo);
- il metodo HTTP utilizzato deve essere contestuale all'operazione che si vuole fare con la risorsa.

Usando questo insieme di regole è possibile rappresentare semplicemente i metodi per l'accesso alle diverse entità, ma non è enfatizzata la dipendenza che ci può essere tra la risorsa che si vuole manipolare, e un'altra risorsa di livello più alto. Per questo motivo si è scelto di inserire un livello gerarchico ulteriore per le risorse che dipendono direttamente da un'altra risorsa, che chiameremo quindi di secondo livello. Le risorse di primo livello sono:

- users;
- organizations.

Quelle di secondo livello:

- work time entries;
- time offs;
- members.

Queste ultime risorse sono tutte contestuali ad una determinata organizzazione, che dovrà quindi essere preposizionata nell'URL. Considerati i requisiti della web application, le API da implementare sono le seguenti:

POST /api/users/

GET /api/users/me

DELETE /api/users/me

PUT /api/users/me

POST /api/users/new

GET /api/organizations/:id

DELETE /api/organizations/:id

PUT /api/organizations/:id

POST /api/organizations/new

POST /api/organizations/:id/workTimeEntries/

DELETE /api/organizations/:id/workTimeEntries/:id

PUT /api/organizations/:id/workTimeEntries/:id

POST /api/organizations/new/workTimeEntries/new

GET /api/organizations/:id/timeOffs/

DELETE /api/organizations/:id/timeOffs/:id

PUT /api/organizations/:id/timeOffs/:id

POST /api/organizations/new/timeOffs/new

GET /api/organizations/:id/members/

DELETE /api/organizations/:id/members/:id

PUT /api/organizations/:id/members/:id

POST /api/organizations/new/members/

Rispetto all'esempio classico di API RESTful, spesso è stato utilizzato il metodo POST (anzichè GET) per il metodo di ricerca di una determinata entità. La modifica è necessaria se si considera la complessità dei parametri per il metodo di ricerca; il passaggio in query string concesso dalla GET sarebbe stato molto più difficoltoso del passaggio nel body della request. Per distinguere il metodo di POST richiamato per creare la risorsa, si utilizza la parola `new` dopo il tipo della risorsa.

Per quanto riguarda la sicurezza, quasi la totalità di queste API deve essere opportunamente protetta in modo che possa essere richiamata soltanto da utenti autenticati (eccetto quella utilizzata per la creazione di un nuovo utente). Oltre ai meccanismi di autenticazione, sono ovviamente necessari anche meccanismi di autorizzazione per discriminare tra i ruoli dei diversi utenti all'interno dell'organizzazione.

4.3 Lo schema delle entità

Lo schema delle entità può essere progettato sulla base del modello del domino definito in precedenza. MongoDB è un DBMS NoSql, quindi offre la possibilità di sviluppare le relazioni tra i documenti in due differenti modalità: inserendo direttamente dei veri e propri documenti all'interno di altri (subdocuments), oppure legandoli attraverso il concetto di riferimento (che molto si avvicina a quello che avviene normalmente con le chiavi nei classici RDBMS). La scelta tra quale delle due tipologie di relazione utilizzare per legare due determinati documenti non è affatto banale, anzi comporta opportune valutazioni sulla base di alcuni principi fondamentali, che si cerca di riassumere:

- Inserire quanto più possibile all'interno di un singolo documento. La forza di mongoDB è l'eliminazione dei JOIN per effettuare query velocissime, di conseguenza è istintivo cercare di inserire in all'interno di un singolo documento quante più informazioni possibili, soprattutto se queste informazioni sono sensate soltanto in riferimento al documento padre.

- Separare i dati che possono essere referenziati da più di un documento in una singola collection. Più che per questioni di spazio di storage, si tratta di consistenza. Se molti documenti fanno riferimento ad un altro determinato documento, replicare quest'ultimo internamente ad ognuno di essi sarebbe molto problematico (e causa di errore) nel caso vi sia la necessità di effettuare una modifica, che andrebbe quindi propagata in ogni replica.

- Difficoltà nel percorrere strutture dati complesse. MongoDB può facilmente contenere delle strutture dati molto complesse e leggerle in blocco molto velocemente, ma non è altrettanto efficiente quando occorre navigare queste strutture dati per ricavarne una piccola parte di informazioni. In questo caso, è consigliato separare i subdocuments obiettivi della/e query, rispetto al parent.

- Per ultimo, considerare che la dimensione dei singoli documenti non può eccedere 16MB.

In generale, come spesso accade, non c'è una precisa ricetta su come ottimizzare il tempo di esecuzione delle query, ma spesso si cerca di partire da uno schema iniziale progettato con scelte sensate e motivate, per poi ottimizzare passo passo in base all'esecuzione delle query. Non è scopo di questo lavoro di tesi effettuare "fine tuning" delle performance di mongoDB, ma si cercherà di dare una motivazione di massima ad ogni scelta fatta per la progettazione dello schema entità.

4.3.1 Le query di ricerca previste

In questo paragrafo saranno brevemente introdotte le query che devono essere effettuate sulle varie entità applicative, separate in tabelle. Nella prima colonna è descritto il caso d'uso in cui la query è necessaria, nella seconda colonna è descritta la tipologia della query, mentre nell'ultima colonna si aggiunge in forma testuale qualche dettaglio. Nei punti in cui non è specificata l'entità, il caso d'uso si riferisce all'entità corrispondente.

User

login, reset password	**find by email**	Estrazione di un singolo user dalla collection facendo match esatto con il campo email.
creazione member	**search by name**	Estrazione di più users dalla collection facendo match in forma "like" con il campo name.
modifica - eliminazione - creazione timbratura	**find by id**	Estrazione di un singolo user con campo id.
registrazione	**-**	Nessuna query di ricerca

Organization

login - ricerca	**find by user**	Estrazione di più organizzazioni dalla collection, ricavando solo quelle a cui l'utente appartiene.
modifica - eliminazione	**find by id**	Estrazione di una singola organizzazione con campo id.
creazione	**-**	Nessuna query di ricerca.

Member

ricerca	**find member into organization**	Query che viene effettuata per proteggere le ogni API che non sia pubblica. Questa è una delle query più comuni nell'applicazione, vengono estratti tutti i member da una organization.
modifica - eliminazione	**find by id**	Estrazione di un singolo member con campo id.
creazione	**-**	Nessuna query di ricerca.

Per risparmiare sul tempo di esecuzione della query di ricerca, si può scegliere di inserire tutte le relazioni member direttamente all'interno dell'entità organization, in modo da ricavare con un unica interrogazione anche tutti gli utenti appartenenti all'organization (con i relativi ruoli).

WorkTimeEntry

ricerca	**search by filters**	Estrazione di più workTimeEntry dalla collection. Può essere fatto filtro per id organizzazione, per id utente, per data, per tipo.
modifica - eliminazione	**find by id**	Estrazione di una singola workTimeEntry con campo id.
creazione	**-**	Nessuna query di ricerca.

TimeOff

ricerca	**search by filters**	Estrazione di più timeOff dalla collection. Può essere fatto filtro per id organizzazione, per id utente, per data, per tipo.
modifica - eliminazione	**find by id**	Estrazione di un singolo timeOff con campo id.
creazione	**-**	Nessuna query di ricerca.

TimeOff e workTimeEntry sono entità piuttosto simili, le classiche operazioni di CRUD non necessitano di query complesse, tuttavia la principale query richiamata per popolare i rispettivi ambienti è vincolata dagli identificativi di organizzazione e utente, oltre ad un filtro per data e per tipo. La presenza di questi identificativi nella query di ricerca rende naturale pensare di inserire il concetto di relazione tra un'entità di questo tipo e la coppia utente-organizzazione.

WorkingDays

ricerca - modifica	**search by organization id**	Query sull'intera collection, facendo filtro sull'organizzazione alla quale sono collegati.

Entrambe le query relative ai working days sono effettuate utilizzando l'organizzazione selezionata all'interno dell'applicazione. Per questo motivo è stato scelto di fare *embedding* di questo schema direttamente all'interno del documento organizzazione.

4.3.2 La definizione degli schema

Sulla base di un'analisi delle differenti query descritte in precedenza, attraverso la libreria mongoose sono stati definiti i seguenti *schema* per tutte le entità di dominio:

```
/**
 * User Schema
 */
var UserSchema = new BaseSchema({
    name: String,
    email: {
        type: String,
        lowercase: true
    },
    hashedPassword: String,
    provider: String,
    salt: String,
    _lastOrganization: {
        type: String,
        ref: 'Organization'
    }
});

/**
 * Organization Schema
 */
var OrganizationSchema = new BaseSchema({
    name: String,
```

```
    members: [MemberSchema],
    settings: {
        defaultTimeOffAmount: {
            type: Number,
            default: 8
        },
        timeOffTypes: [String],
        workingDays: [workingDaySchema]
    },
    hashedPassword: String,
    salt: String
});

/**
 * Organization's Member schema
 * it's not inheriting from base model,
 * cause it's not a db collection
 */
var MemberSchema = new Schema({
    _user: {
        type: Schema.ObjectId,
        ref: 'User'
    },
    role: {
        type: String,
        enum: ['admin', 'user']
    },
```

```
    nfc_uid: String
});

/**
 * Time Off Schema
 */
var TimeOffSchema = new BaseSchema({
    _user: {
        type: Schema.ObjectId,
        ref: 'User'
    },
    _organization: {
        type: Schema.ObjectId,
        ref: 'Organization'
    },
    timeOffType: String,
    performedAt: Date,
    amount: Number,
    description: String
});

/**
 * Work Time Entry Schema
 */
var WorkTimeEntrySchema = new BaseSchema({
    _user: {
        type: Schema.ObjectId,
```

```
        ref: 'User',
        required: true
    },
    _organization: {
        type: Schema.ObjectId,
        ref: 'Organization',
        required: true
    },
    performedAt: {
        type: Date,
        default: Date.now,
        required: true
    },
    workTimeEntryType: {
        type: String,
        enum: ['in', 'out'],
        required: true
    },
    manual: {
        type: Boolean,
        default: false,
        required: true
    }
});
```

Ovviamente ogni entità prevede anche altri campi rispetto a quelli introdotti con il modello del dominio, che non vengono descritti nel dettaglio ma sono comunque necessari alle varie logiche applicative. L'importanza della progettazione dello

schema sta soprattutto nella comprensione della scelta implementativa per la rappresentazione delle relazioni tra le diverse entità, in quanto fondamentale per le performance di mongoDB.

4.4 Struttura del server

Lo scheletro architetturale generato per il server prevede una netta differenziazione tra quelle che sono le funzionalità di autenticazione e le API, che sono separate in due differenti moduli. Queste due "sezioni", insieme alla parte che vedremo in seguito per la comunicazione con il device, rappresentano il core delle funzionalità offerte dal server e possono essere interpretate come lo strato di controller.

4.4.1 API end-point

Il modulo contenente le API è separato dal resto del server, è interno alla folder "/api/" ed è ulteriormente suddiviso in cartelle per entità, in modo da rispecchiare quella che è la semantica REST delle API stesse. Le entità sono le stesse precedentemente definite, ed ogni cartella rappresenta sostanzialmente un *end-point* per l'entità. Internamente contiene quattro file JavaScript:

- index.js, che contiene il mapping tra le routes definite in express e il controller. Nel metodo utilizzato per mappare la route, tra la definizione del pattern e la dichiarazione del corrispettivo metodo di controller, la sintassi di express consente di inserire, separandole con la virgola, tutte le funzioni di middleware necessarie per i meccanismi di autorizzazione;

- entity.model.js, dove sono definiti lo schema dell'entità con i suoi vincoli e tutte le funzionalità o metodi ad esso legati;

- entity.controller.js, modulo in cui sono implementati i metodi delle API per l'accesso all'entità. L'interfaccia del modulo, ovvero i metodi corrispondenti alle varie API, è resa pubblica attraverso la parola chiave di node.js "exports";

- entity.socket.js, l'implementazione di una web socket a cui i client si possono connettere per ricevere aggiornamenti sull'entità.

4.4.2 Autenticazione

Il meccanismo di autenticazione implementato nella web application è la cosiddetta "token based authentication", che può essere estremamente sintetizzato nel seguente flusso: l'utente invia username e password soltanto la prima volta, dopodichè se l'autenticazione va a buon fine, riceve dal server un "token" che potrà utilizzare (solitamente fino ad una determinata scadenza) per effettuare nuove richieste, senza la necessità di inviare nuovamente le proprie credenziali. Il server infatti è l'unica entità in grado di validare l'autenticità del token e al tempo stesso di ricavarne le informazioni dell'utente a cui appartiene. Tutto questo avviene grazie ad un meccanismo di crittografia con chiave simmetrica, chiave che ovviamente è esclusivamente in possesso del server.

I vari moduli relativi all'autenticazione sono contenuti all'interno della folder "/auth". Come già anticipato in precedenza, questi servizi sono utilizzati dai client della web application, dal device (sia per autenticare l'utente che effettua la timbratura che per autenticare sè stesso), e infine anche dall'applicazione smartphone dell'utente. I servizi di autenticazione sono esposti verso l'esterno come metodi HTTP, mentre internamente il server li espone come un modulo "ad-hoc" che può essere utilizzato contestualmente ad altre operazioni, ad esempio per la validazione dei meccanismi di autorizzazione sulle diverse API REST.

Se registrati come semplici utenti locali, al momento del login da form di accesso con email e password, viene sfruttata la "strategia locale" di passport.js, che consiste nella classica ricerca di utente per email e confronto della password sul database con quella inviata. Per ovvie ragioni di sicurezza il salvataggio della password in chiaro non è una prassi consigliata, di conseguenza viene creato e salvato l'hash al momento della registrazione utente, dopodichè l'hash è ricalcolato ad ogni tentativo di login e confrontato con quello memorizzato sul db. Per aumentare ulteriormente la sicurezza, viene utilizzato un "sale" di 16 byte che incrementa esponenzialmente la complessità di un eventuale tentativo di decifrazione. La generazione di hash della password ed il confronto con quella inserita dall'utente durante il login, sono funzioni direttamente eseguite sullo schema grazie a mongoose, che consente di definire dei metodi custom come se fossero utiità agganciate direttamente al document utente presente su mongoDB.

Un altro tipo di autenticazione utente è quella che avviene attraverso l'uso del profilo Google. All'interno della console delle API Google è possibile ottenere una chiave "segreta" (dedicata solo dall'applicazione web registrata) da utilizzare nella configurazione della strategia su passport.js. Una volta configurato passport.js con la chiave, l'applicazione può veicolare gli utenti che desiderano sfruttare il proprio account Google per autenticarsi, ad una pagina in cui il provider richiede esplicitamente all'utente l'autorizzazione. In questo modo gli utenti sono automaticamente registrati come utenti locali dell'applicazione, avendo però come garante della loro autenticità Google. L'autorizzazione è ovviamente richiesta soltanto la prima volta in cui l'utente effettua l'accesso, dopodichè il provider ricorda la scelta dell'utente e fornisce all'applicazione il suo token senza necessità di ulteriore conferma.

4.4.3 Comunicazione con il device

La comunicazione con gli smart device è alla base del meccanismo attraverso il quale l'utente è autenticato al momento della timbratura sulla porta. Si è scelto di utilizzare il protocollo websocket per instaurare il canale tra server e device, in quanto idealmente le due entità dovrebbero sempre essere connessi, poiché l'autenticazione deve preferibilmente essere a carico del server. La comunicazione quindi è basata sullo scambio di messaggi. Relativamente alla timbratura, il primo messaggio parte sempre dallo smart device. L'utente, nella comunicazione con lo smart device, può scegliere se utilizzare direttamente il proprio smartphone, oppure usare il tag nfc a lui associato dall'organizzazione. Onde evitare di scambiare direttamente le credenziali, l'idea è quella di applicare la stessa tecnica messa in atto nell'autenticazione client sulla web application, ovvero lo scambio del solo token di autenticazione. Il flusso è molto semplice:

- l'utente utilizza lo smartphone o il tag nfc per comunicare con il device;
- il device legge l'identificativo univoco del tag (uid) oppure il messaggio ricevuto dallo smartphone e lo invia al server;
- il server valida l'uid del tag nfc o il token e, in caso l'utente sia riconosciuto e gli orari di ingresso siano validi, inserisce nel database la nuova timbratura comunicando la risposta al device. Il device si preoccuperà di notificare l'esito all'utente e di effettuare tutte le azioni previste.

Relativamente al flusso sopra descritto, le tipologie di messaggi inviati da device a server sono (nome, contenuto):

- onTokenSubmitted - token: String , nel caso il device abbia ricevuto un messaggio dallo smartphone dell'utente;
- onNfcTagSubmitted - uid: String , nel caso il device abbia letto un tag nfc;

alle quali il server risponde con un messaggio:

- submitResponse - responseCode: xxx , dove xxx è il codice di risposta del server (http-status like).

Si riporta come esempio la gestione del messaggio tokenSubmitted ricevuto dalla socket:

```
/*
* socket is bound to a single device.
*/
    socket.on("tokenSubmitted", function (data) {

        //use of authorization service to validate the token
        authService.verifyToken(data.token,
          function (err, decoded) {
            if (err) {
                //sends back the error
                return handleError(socket, err, data);
            }

            //If validation succeeds, adds work time entry
            addWorkTimeEntry({
                userId: decoded._id,
                organizationId: organizationId
            }, function (err) {
                if (err) {
                    return handleError(socket, err, data);
```

```
                }

                //after successful adding, emits the response
                //message on the socket
                socket.emit('submitResponse', {
                    responseCode: 200,
                    message: 'successfully authenticated'
                });
            });
        });
    });
```

Il modulo server dedicato alla comunicazione point-to-point con i device installati nelle organizzazioni è all'interno della folder "/device/device.js" e gestisce tutte le socket aperte all'interno di un unico array. Per aumentare il livello di sicurezza, le socket devono essere aperte soltanto in corrispondenza di un token di autenticazione valido. Per le organizzazioni, che utilizzano un determinato device, viene applicato lo stesso metodo di autenticazione che è usato per gli utenti locali. Attraverso la web application, gli utenti amministratori di un'organizzazione hanno la possibilità di impostare una password segreta. Questa dovrà poi essere configurata sul device e sarà necessaria ad ottenere il token dal server, tramite autenticazione HTTP. Al momento della richiesta di connessione iniziale, il token sarà presentato dal device al server che, tramite un protocollo di handshake garantito dal modulo "socketio-jwt", completerà l'autenticazione.

Un altro importante aspetto che riguarda la comunicazione con il device è la gestione dei dati che consentono al dispositivo di garantire una funzionalità minima offline. Sempre attraverso la web socket, il server invia periodicamente al device la lista dei member attivi (con relativi "uid nfc") e gli orari dell'organizza-

zione. In questo modo il device, in caso la connessione non sia momentaneamente attiva, è comunque in grado di riconoscere gli utenti se questi presentano il tag nfc che gli è stato precedentemente collegato, memorizzando la timbratura. Ogni volta che la connettività viene ripristinata, sarà il device ad inviare al server la lista delle timbrature registrate offline. I tipi di messaggi scambiati per garantire l'implementazione di questa funzionalità sono quindi tre:

- workingDays - da server a device;
- members - da server a device;
- data - da device a server.

Si riporta a titolo esemplificativo una parte del codice del modulo, in cui sono esplicitate soltanto le parti di codice significative per la sincronizzazione dei dati offline.

```
...
//module requires

/*
* function called to sync members
*/
function syncMembers(organizationId, members) {
    //retrieve the socket of the organization
    var socket = devices[organizationId];

    if (!socket) {
        return;
    }
```

```
    //emitting member
    socket.emit('members', {
        members: members
    });
};

/*
* function called to sync working days
*/
function syncWorkingDays(organizationId, workingDays) {
    //retrieve the socket of the organization
    var socket = devices[organizationId];
    if (!socket) {
        return;
    }

    //emitting working days
    socket.emit('workingDays', {
        workingDays: workingDays
    });
};

io.use(socketioJwt.authorize({
    secret: config.secrets.session,
    handshake: true
}));
```

```
io.sockets.on('connection', function (socket) {

    var organizationId = socket.decoded_token._id;

    //save reference
    devices[organizationId] = socket;

    //Other methods and function
    ...

    /*
    * Socket receives data message from device
    */
    socket.on("data", function (data) {

        Organization.findById(organizationId,
          function (err, org) {
            if (err) {
                return console.log(err);
            }

            data.data.forEach(function (wte) {
                var userId;
                for (var i = 0; i < org.members.length
                  && !userId; i++) {
                    if (org.members[i].nfc_uid === wte.uid) {
```

```
                    userId = org.members[i]._user;
                }
            }

            if (!userId) {
                return console.log('user not found');
            }

            //user found
            addWorkTimeEntry({
                userId: userId,
                organizationId: organizationId,
                performedAt: wte.performedAt
            }, function (err) {
                console.log(err);
            });
        });

        socket.emit('dataReceived');
    });
});

/*
* As last operation on a new device connection,
* after registering handlers, sends to device data
* about organization (members and working days)
*/
```

```
    Organization.findById(organizationId, function (err, org) {

        if (err) {
            return console.log('cannot sync members');
        }

        socket.emit('members', {
            members: org.members
        });

        socket.emit('workingDays', {
            workingDays: org.settings.workingDays
        });
    });
});

//Public methods of the module
exports.syncMembers = syncMembers;
exports.syncWorkingDays = syncWorkingDays;
```

4.5 Struttura dell'applicazione cient

La client application è stata sviluppata con il framework angular.js, ed è il mezzo attraverso il quale gli utenti dell'organizzazione (compresi gli amministratori) possono interagire con il server per la consultazione e l'inserimento dei dati nel sistema. Si tratta di una single page application, ovvero un'intera applicazione costruita in una singola pagina HTML principale. È angular che si occupa dell'ar-

ricchimento della pagina con le funzioni di navigazione tra i diversi ambienti, che il framework gestisce richiedendo al server il contenuto statico necessario (parti di HTML, css, file javascript).

4.5.1 Architettura delle aree applicative

Come già descritto in fase di presentazione del framework, l'architettura dell'applicazione è model-view-controller. Similarmente a quanto avviene per il server, anche la single page application sviluppata ha una struttura che separa i diversi moduli applicativi per area di competenza. Le aree principali presenti nella struttura rispecchiano all'incirca le entità di dominio coinvolte:

- account, che a sua volta contiene delle sotto aree per le funzionalità legate alla gestione del profilo utente;
- workTimeEntries, per la gestione delle timbrature;
- timeOffs, per la gestione delle assenze;
- organizations, per l'impostazione delle organizzazioni;
- members, per la gestione degli utenti all'interno delle organizzazioni.

Ognuna di queste aree (formalmente un'area è paragonabile ad una view, un modulo) è mappata dal modulo di angular UI-Router con il concetto di "stato". UI-Router, oltre alla possibilità di definire livelli gerarchici tra i diversi stati, consente, per ognuno di essi, la definizione di impostazioni di configurazione. Internamente le aree possono contenere tanti diversi file, ognuno nominato in modo contestuale alla sua utilità, ad esempio:

- entity.js, il file JavaScript principale che contiene le informazioni di configurazione dello stato per ui-router (nome del controller, percorso del template html, dipendenze);

- entity.controller.js, un file JavaScript in cui è definito e implementato il costruttore del controller per il modulo;
- entity.html, Il template (la view) HTML relativa;
- entity.scss, file SCSS per tutti gli stili propri del modulo non condivisi con le altri parti dell'applicazione;
- entity.service.js, che contiene la definizione della factory per l'accesso alle API messe a disposizione dal server per la modifica dell'entità.

Oltre a questi file, solitamente sempre presenti, ne sono talvolta inclusi anche altri, ovviamente relativi al modulo in questione.

4.5.2 UI-Router: la gerarchia degli stati

Un aspetto interessante per l'organizzazione del codice e dell'architettura della single page application, riguarda la suddivisione di tutte le diverse aree applicative in private e pubbliche. Questa suddivisione si è rivelata utile per gestire in modo elegante il passaggio dei dati relativi all'utente loggato alle aree che hanno il login come pre-requisito per l'accesso. A tale scopo, sono stati definiti in UI-Router gli stati astratti *private* e *public*, che vengono poi "estesi" dai vari stati figli che quindi ereditano le caratteristiche dello stato padre. In particolare, per quel che riguarda lo stato *private*, è già implementato al suo interno il reperimento delle informazioni relative all'utente loggato e all'organizzazione selezionata, in modo da non doverlo ripetere all'interno di tutti gli altri stati figli. I due stati astratti sono definiti nella cartella `/app/root/` come private.js e public.js. Si riporta l'implementazione dello stato astratto private e del relativo controller:

```
angular.module('ioAtSpotApp')
    .config(function ($stateProvider, USER_ROLES) {
```

```
        $stateProvider.state('private', {
            abstract: true,
            templateUrl: 'app/app.html',
            controller: 'PrivateCtrl',
            data: {
                authorizedRoles: [USER_ROLES.admin,
                  USER_ROLES.user]
            },
            resolve: {
                authModel: ['Auth',
                function (Auth) {
                        return Auth.getAuthModel().$promise;
                }]
            }
        });
    });

angular.module('ioAtSpotApp')
    .controller('PrivateCtrl',
      function ($scope, Auth, authModel) {

        $scope.parent = {
            currentUser: authModel.currentUser,
            currentOrganization: authModel.currentOrganization,
            isCurrentOrganizationAdmin: function () {
                var org = $scope.parent.currentOrganization;
                var currentUser = $scope.parent.currentUser;
```

```
                for (var i = 0; i < org.members.length; i++) {
                    if (org.members[i]._user._id === currentUser._id
                        && org.members[i].role === 'admin') {
                        return true;
                    }
                }
                return false;
            }
        };
...
    });
```

Come si può notare dall'implementazione del controller, vengono aggiunte allo *scope* le informazioni relative all'utente corrente e all'organizzazione selezionata, scope che sarà ereditato ed esteso dagli stati figli. Anche in questo caso l'autenticazione è gestita tramite un servizio ad-hoc che mette a disposizione funzionalità utilizzate da tutti i moduli dell'applicazione che necessitano di interagire con l'utente loggato.

4.5.3 Un esempio: l'area workTimeEntries

La configurazione del modulo, o meglio dello stato, è definita all'interno di workTimeEntries.js, in cui l'estensione dello stato padre *private* è resa esplicita con l'utilizzo della *dot notation* nella definizione del nome.

```
angular.module('ioAtSpotApp')
  .config(function ($stateProvider) {
    $stateProvider.state('private.workTimeEntries', {
```

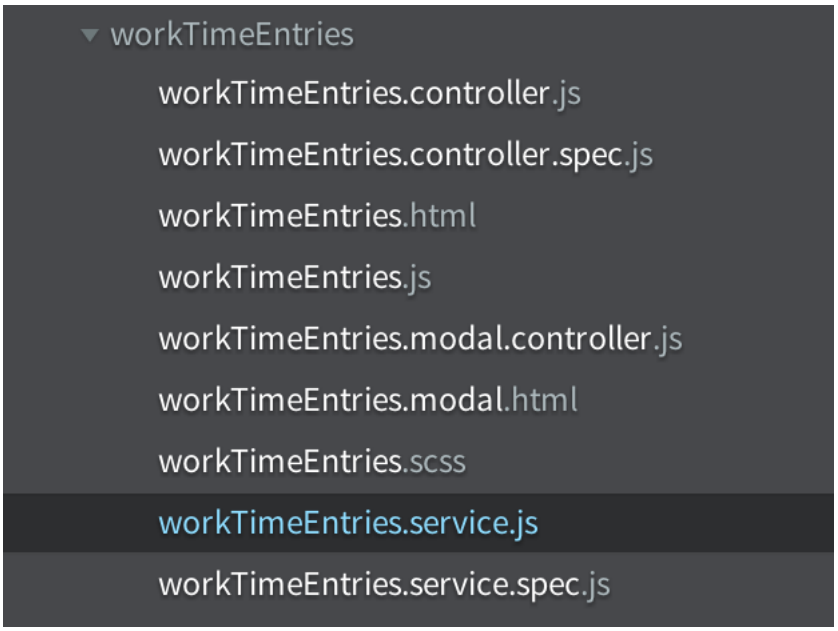

Figura 4.7: Struttura dell'area per le work time entries

```
        url: '/WorkTimeEntries',
        templateUrl: 'app/workTimeEntries/workTimeEntries.html',
        controller: 'WorkTimeEntriesCtrl'
    });
});
```

Nella configurazione di questo stato sono definiti l'url applicativo corrispondente (la route), l'url del template utilizzato per il rendering della view, e il nome del controller utilizzato. Il controller è implementato internamente a workTimeEntries.controller.js, incapsula le logiche di business del modulo, ed è quello che coordina l'interazione con l'utente. Per questioni di leggibilità, non verrà completamente riportato tutto il codice del controller, ma soltanto le aree in cui esso è suddiviso:

```
angular.module('ioAtSpotApp')
    .controller('WorkTimeEntriesCtrl',
```

```
function ($scope, $http, socket, WorkTimeEntries,
  $moment, $modal) {

    $scope.model = new function () {
        var model = this;

        /* ... DOM Bounded model data ... */

    };

    $scope.utils = new function () {
        var utils = this;

        /*... Utility functions ... */
    };


    $scope.actions = {

        /* ... DOM Bounded actions ..*/
    };

    $scope.proxies = new function () {
        var proxies = this;

        /* ... Proxies function to manipulate entity ..*/
    };
```

```
            /* ... init operations ... */
        });
```

La suddivisione del codice interno al controller si è resa necessaria per aumentarne la leggibilità. Come si può notare guardando la firma del costruttore, il controller riceve diversi parametri di ingresso: solitamente si tratta di moduli angular oppure di servizi custom definiti altrove e utilizzati internamente al controller.
Un esempio molto rilevante di servizio è implementato nel file workTimeEntries.service.js, ed è utilizzato per l'accesso alle API per l'entità workTimeEntry, messe a disposizione dal server:

```
angular.module('ioAtSpotApp')
  .factory('WorkTimeEntries',
    ['$resource', function ($resource) {
return $resource(
    '/api/organizations/:organizationId/workTimeEntries/:id', {
    organizationId: '@organizationId',
    id: '@workTimeEntryId'
}, {
 query: {
    url: '/api/organizations/:organizationId/workTimeEntries?page=:page',
    method: 'POST',
    params: {
      organizationId: '@organizationId',
      page: '@page'
    },
    isArray: false //returns an object (that also contains an array)
  },
```

```
    update: {
      method: 'PUT',
      params: {
        organizationId: '@organizationId',
        id: '@id'
      },
      isArray: false
    },
    create: {
      url: '/api/organizations/:organizationId/workTimeEntries/new',
      method: 'POST',
      params: {
        organizationId: '@organizationId'
      },
      isArray: false
    },
    delete: {
      method: 'DELETE',
      params: {
        organizationId: '@organizationId',
        id: '@id'
      },
      isArray: false
    }
  });
  }]);
```

Internamente a questo file è definito l'insieme dei metodi che possono essere richiamati sul servizio per accedere ad una determinata risorsa. Per ogni metodo sono specificati l'url, il metodo HTTP utilizzato e i relativi parametri. Nel momento in cui il controller utilizza uno di questi metodi per l'accesso alla risorsa, è il servizio stesso che si preoccupa di effettuare la richiesta HTTP e tutta la relativa gestione, semplificando notevolmente il meccanismo.
Gli ulteriori file definiti sono la view, il relativo foglio di stile, e tutti quelli dedicati alla gestione della finestra modale per l'inserimento e la modifica di una workTimeEntry.

Capitolo 5

Progettazione e sviluppo dello smart device

In questo capitolo sarà discussa la progettazione e lo sviluppo del componente che integra la web application trasformandola in un sistema "smart" per Internet of Things. Il dispositivo sviluppato sarà il modulo che le aziende potranno installare in prossimità dell'accesso ed attraverso il quale gli utenti completeranno l'apertura della porta e l'inserimento della timbratura. Nonostante si tratti di un nodo periferico, è piuttosto chiara l'importanza del dispositivo all'interno di tutto il sistema, poiché l'obiettivo primario è proprio quello di semplificare i flussi e i processi che normalmente avvengono manualmente.

La discussione prevede un'iniziale panoramica sui componenti hardware e software utilizzati per lo sviluppo del device e la relativa configurazione, al fine di motivare le scelte fatte per le diverse parti del sistema ed esporne le principali caratteristiche. Nella seconda sezione del capitolo sarà invece discussa la progettazione e l'implementazione del software installato sul dispositivo, suddividendo l'intera applicazione nei vari moduli di cui è composta, con particolare enfatizzazione di quelli utilizzati per la comunicazione.

5.1 La base di partenza: Raspberry pi

Il primo step fondamentale per lo sviluppo dello "smart device" è sicuramente la scelta della piattaforma hardware inziale su cui costruirlo. Sebbene i requisiti funzionali del device non siano particolarmente complessi, è bene valutare attentamente anche i requisiti non funzionali (o vincoli) per poter selezionare la base computazionale che meglio li soddisfa, compatibilmente con i rapidi tempi di sviluppo richiesti. Rielaborando l'analisi dei requisiti del dispositivo introdotta nella prima parte di questo lavoro, possono essere estratti i seguenti vincoli:

- il dispositivo deve essere economico;
- il dispositivo deve essere facilmente manutenibile;
- il dispositivo deve avere bassi consumi energetici;
- il dispositivo deve essere piccolo, in quanto deve essere installato in prossimità dell'accesso;
- il dispositivo deve comunicare con l'utente utilizzando le tecnologie NFC e BLE;
- il dispositivo deve essere connesso ad internet per poter comunicare con il server;
- il dispositivo deve essere modulare e facilmente estendibile;
- il dispositivo deve poter interagire con un circuito elettrico esterno per aprire la porta.

Per un sistema hardware/software con dei vincoli così ben definiti, è molto difficile trovare sul mercato un device già pronto all'uso a costi contenuti; in questi casi, la soluzione spesso più sensata risiede nella scelta di un dispositivo di base,

che viene quindi arricchito con differenti moduli hardware o software per poter implementare tutte le caratteristiche richieste. Il primo vincolo preso in considerazione è la necessità di avere un basso consumo energetico. Il dispositivo deve necessariamente rimanere acceso e attivo costantemente, quindi un alto consumo energetico avrebbe un impatto economico non giustificabile, visti i limitati compiti a lui dedicati. La precedente considerazione porta naturalmente ad evitare tutte le soluzioni hardware normalmente utilizzate nei computer di casa e in quelli aziendali, spostando l'interesse verso i *single-board-computer*. Questi dispositivi infatti, oltre ad essere relativamente economici, compatti e a basso impatto energetico, hanno la peculiarità di poter interagire con componenti elettroniche tramite *General Purpose Input/Output* (GPIO), fondamentale nel nostro caso per poter comandare l'interruttore elettrico dedicato all'apertura della porta. I due principali *single-board-computer* presenti sul mercato sono Raspberry pi ed Arduino. La scelta di utilizzare Raspberry pi piuttosto che Arduino ha trovato la principale motivazione nella possibilità di poter sviluppare un progetto software di livello più alto, facendo uso di framework e stack applicativi conosciuti ed astraendo il più possibile dalla natura hardware del sistema sottostante. In aggiunta a ciò, la maggiore potenza di calcolo e la possibilità di multi-tasking nativo offerto dai sistemi operativi, sono sicuramente caratteristiche rilevanti per la collocazione del dispositivo nel contesto di sistema distribuito, in quanto potrebbero sorgere diversi problemi nel caso il *core* del sistema fosse un microcontrollore, anche se di alto livello.

5.2 Configurazione

La definizione dei componenti hardware utilizzati assieme al raspberry per la realizzazione del prototipo è il passo successivo per la realizzazione del sistema.

Sono stati utilizzati una scheda di memoria SD da 16GB e un alimentatore 5V USB per l'iniziale setup del sistema. Con l'obiettivo di mantenere buona compatibilità verso le librerie standard e poter contare su gran parte del materiale di supporto prodotto dalla community di utenti raspberry, si è scelto di installare l'ultima versione del sistema operativo "Raspbian", una particolare distribuzione Linux promossa da *Raspberry Foundation*. Una volta completata l'installazione, è buona prassi aggiornare tutti i pacchetti software installati nel sistema subito dopo il primo login, utilizzando le due istruzioni messe a disposizione :

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

5.2.1 I componenti aggiuntivi per la comunicazione

Il dispositivo deve necessariamente avere tre differenti interfacce con le quali interagire con i restanti attori presenti nel sistema, ovvero gli utenti, ed il server. Fortunatamente, Raspberry pi dispone già di un'interfaccia ethernet, quindi è collegabile direttamente alla rete come un normale computer per poter comunicare con il server attraverso internet.

Diversamente da quanto avviene per la comunicazione con il server, Raspberry pi non dispone delle interfacce Bluetooth Low Energy ed NFC, che devono quindi essere installate come periferiche aggiuntive.

Interfaccia Bluetooth Low Energy

L'utilizzo dell'interfaccia Bluetooth Low Energy necessita di una periferica dedicata, che ad esempio può essere una delle tante differenti antenne bluetooth USB presenti in commercio e compatibili con il protocollo Bluetooth 4.0, prerequisito necessario per lo sfruttamento della tecnologia BLE. Il dispositivo sfrutta-

to, del costo di circa 10 euro, si basa sul chipset Broadcom BCM20702 uno dei più comuni chipset Bluetooth presenti sul mercato di consumo. Il chipset integra un microprocessore con un'antenna ricetrasmittente che opera sui 2.4GHz, e supporta entrambe le modalità di comunicazione Bluetooth (standard e low energy), anche se sarà ovviamente utilizzata soltanto la seconda. Per poter sfruttare l'inter-

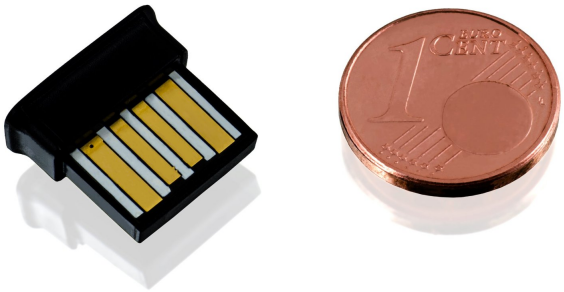

Figura 5.1: Il chipset bluetooth utilizzato: CSL Bluetooth micro adapter

faccia bluetooth low energy, il sistema operativo necessita dei driver e dello stack bluetooth, poiché nativamente non sono installati su Raspbian.

Lo stack blueZ

BlueZ [6] è lo stack Bluetooth ufficiale per i sistemi operativi basati su kernel Linux, si tratta di un progetto open source che dal 2006 supporta tutti i principali protocolli Bluetooth. BlueZ è stato sviluppato inizialmente da Qualcomm, ed è disponibile per kernel Linux dalla versione 2.4.6 in poi. Si tratta di un progetto maturo e relativamente robusto, diffuso su larghissima scala, basti pensare

che è stato integrato nella piattaforma Android fino al 2012. L'obiettivo di blueZ è quello di dare al sistema operativo la possibilità di utilizzare periferiche bluetooth-compatibili, fornendo quindi agli sviluppatori un insieme di API che possono essere sfruttate per comunicare in modo conveniente e *platform-independent* con altri dispositivi, sia nella versione classica (vanilla Bluetooth 2.0) che nella versione low energy. Purtroppo, come ogni tanto succede per progetti open source, la documentazione è pressoché assente, rendendo piuttosto complicato l'utilizzo delle API per gli sviluppatori che non conoscono a fondo la piattaforma. Per questioni di compatibilità con i moduli applicativi utilizzati, la versione di blueZ utilizzata nel sistema è la 4.101. Questi i passi necessari per l'installazione da linea di comando:

```
$ wget http://www.kernel.org/pub/linux/bluetooth/bluez-4.101.tar.xz
$ tar -xvf bluez-4.101.tar.xz
$ cd bluez-4.101
$ ./configure
$ sudo make
$ sudo make install
```

Il completamento dell'installazione avviene dopo diversi minuti, tempo necessario al Raspberry per la compilazione dei sorgenti. La corretta installazione dello stack può essere sottoposta a test collegando l'adattatore bluetooth in una porta USB e provando a lanciare da shell il comando `hciconfig`, una funzione di utilità inclusa in blueZ, che stampa come risultato informazioni sull'adattatore bluetooth utilizzato e la relativa interfaccia:

```
$ hciconfig
hci0:   Type: BR/EDR  Bus: USB
         BD Address: 00:1A:7D:DA:71:0C  ACL MTU: 310:10  SCO MTU: 64:8
```

```
DOWN
RX bytes:467 acl:0 sco:0 events:18 errors:0
TX bytes:317 acl:0 sco:0 commands:18 errors:0
```

Interfaccia near field communication

La tecnologia di comunicazione NFC non è nativamente presente su Raspberry pi, che deve quindi essere dotato dell'interfacciamento hardware (e relativo software) per fornire al dispositivo la possibilità di comunicare con device o tag NFC compatilbili. Differentemente da quanto avviene per i dispositivi Bluetooth, sul mercato non sono disponibili tantissime soluzioni hardware, e spesso il costo di questi dispositivi è decisamente alto se comparato al resto delle componenti, Raspberry compreso. Dopo un'analisi comparativa tra le possibili alternative, si è scelto di utilizzare la board NFC prodotta da Adafruit, una delle aziende leader nello sviluppo di componenti hardware per la prototipazione elettronica non industriale. Questa board si basa sul chip PN532 di NXP semiconductors, che ha

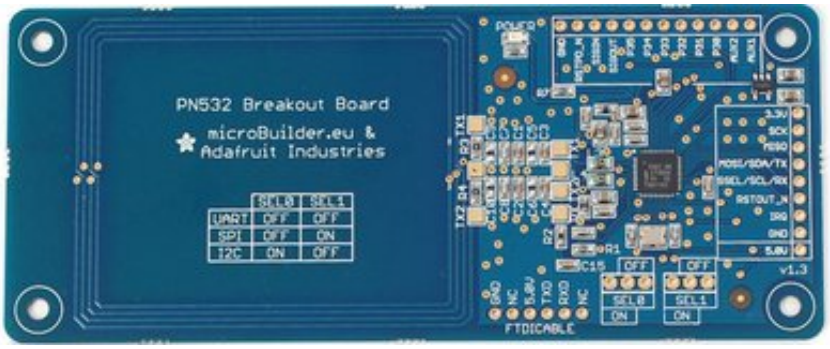

Figura 5.2: Adafruit PN532 breakout board - La scheda utilizzata per la comunicazione NFC

la caratteristica di essere pienamente compatibile con le specifiche NFC-Forum, e di conseguenza è un'ottima soluzione per qualsiasi sistema che debba fare uso di NFC. Come si può dedurre dalla Figura 5.2, sulla parte sinistra è presente un'an-

tenna RFID piuttosto grande, grazie alla quale è possibile interagire con tag anche a 5-6 cm di distanza. Nella parte di destra è invece stampato il circuito integrato. I protocolli di comunicazione implementati dalla scheda sono tre: 3V TTL UART, I2C o SPI.
La scelta di utilizzare una board prototipale, piuttosto che un dispositivo USB è stata dettata dalla maggiore flessibilità che questa offre: essendo NFC una tecnologia "di nicchia", la possibilità di sfruttare differenti interfacce e protocolli di comunicazione per l'interazione con il chipset NFC dovrebbe dare maggiori garanzie dal punto di vista della compatibilità.

Nel nostro caso la comunicazione avverrà attraverso l'interfaccia seriale UART, quindi i due selettori nella parte centrale della scheda dovranno essere entrambi impostati (con l'ausilio dei jumper) ad OFF. Il collegamento tra il lettore NFC ed il Raspberry pi vedrà utilizzati sulla scheda soltanto i pin nella parte inferiore:

- pin 1 - massa;
- pin 2 - non connesso;
- pin 3 - alimentazione;
- pin 4 - tx (GPIO 14 Raspberry pi);
- pin 5 - rx (GPIO 15 Raspberry pi);
- pin 6 - non connesso.

Per poter comunicare con l'interfaccia seriale sul Raspberry pi, è necessario liberare la porta UART, che su Raspbian è nativamente utilizzata per altri scopi [25]. A tale proposito, occorre modificare il file `/boot/cmdline.txt`, eliminando tutti i riferimenti a `ttyAMA0`, che è proprio il nome della UART. Una volta salvato il file, modificare anche `/etc/inittab` ed eliminare la linea in cui si fa ancora una

volta riferimento alla seriale `ttyAMA0`. Dopo un riavvio, il sistema è pronto per la comunicazione.

5.2.2 Lo stack NFC: nfcpy

Si presenta anche in questo caso la problematica relativa alla scelta dello stack applicativo di cui dotare il sistema. Esistono differenti implementazioni di stack NFC, ma non è semplice individuare uno standard *de facto* per la tecnologia. Una delle prime librerie testate è libnfc [15], progetto open source scritto in C che fornisce un SDK e diverse API per lo sfruttamento di Near Field Communication. Il più grande limite riscontrato durante l'esplorazione delle funzionalità offerte da libnfc è quello di non implementare meccanismi di comunicazione p2p con altri dispositivi, di conseguenza per la comunicazione con dispositivi Android sarebbe stato necessario implementare manualmente il protocollo di comunicazione. Per questo motivo, dopo aver utilizzato libnfc nelle fasi iniziali del progetto, soprattutto per la lettura semplice di tag, la libreria è stata accantonata in favore di nfcpy.

Nfcpy [20] è uno stack NFC promosso da Sony e sviluppato interamentein Python, è forse meno diffuso di libnfc ma per certi versi è più completo. Oltre alle funzioni per leggere/scrivere tag, implementa applicativamente anche il protocollo SNEP (Simple NDEF Exchange Protocol) utilizzato dagli smartphone per la comunicazione p2p. Si tratta di un progetto open source molto ben documentato e supportato, il cui obiettivo è quello di garantire un framework per applicazioni NFC facile da utilizzare. Sebbene il chipset PN532 non sia completamente compatibile con nfcpy, dopo aver effettuato numerose prove si è giunti alla conclusione che il connubio tra stack e dispositivo è perfetto per l'utilizzo che lo smart device prevede: comunicazione p2p con smartphone e lettura tag.
I prerequisiti di nfcpy sono:

- python 2.6 / 2.7, che è già nativamente incluso in Raspbian;
- pySerial, per la comunicazione con la scheda NFC via interfaccia seriale (UART). Questo modulo è facilmente installabile utilizzando il package manager di python *pip*.

L'installazione si può completare quindi con i comandi da shell:

```
#install pyserial
$ sudo pip install pyserial

#install launchpad bazaar Version Control System
$ sudo apt-get install bzr

#enters the dir
$ md nfcpy & cd nfcpy

#get the branch of nfcpy
$ bzr branch lp:nfcpy
```

Per poter utilizzare nei propri applicativi nfcpy è sufficiente dichiararne l'importazione come modulo. In ogni caso, è possibile sottoporre a test il funzionamento di nfcpy semplicemente utilizzando uno dei programmi di esempio già inclusi nei sorgenti. Il programma "tagtool.py" può essere eseguito per leggere o scrivere un semplice tag NFC:

```
$  cd nfcpy/examples
$  python tagtool.py --device tty:AMA0:pn53x show
```

Il parametro –device specifica l'indirizzo del dispositivo. Avvicinando un tag al lettore NFC il sarà letto il messaggio NDEF contenuto e stampato a console.

5.2.3 La prototipazione del circuito

Completate le configurazioni hardware e software per l'utilizzo di Bluetooth LE ed NFC, occorre progettare il circuito elettronico di attuazione, che ha come obiettivo primario quello di apertura della porta, dando un feedback all'utente. Il circuito elettronico dovrà permettere al Raspberry di comandare i seguenti componenti:

- un *relay* per controllare un "incontro elettrico";
- un *buzzer* piezoelettrico per emettere feedback audio;
- un led verde per emettere feedback luminoso positivo;
- un led rosso per emettere feedback liminoso negativo.

Ognuno di questi diversi controlli dovrà essere integrato nel circuito elettronico, per poter poi essere collegato e "pilotato" da un singolo pin GPIO dal Raspberry pi. Il circuito elettronico definitivo è quello visualizzabile in Figura 5.3, e si compone di 6 diversi ingressi: quattro ingressi sono i controlli per pilotare i vari componenti, uno è l'alimentazione a 5V e per ultimo c'è la massa.

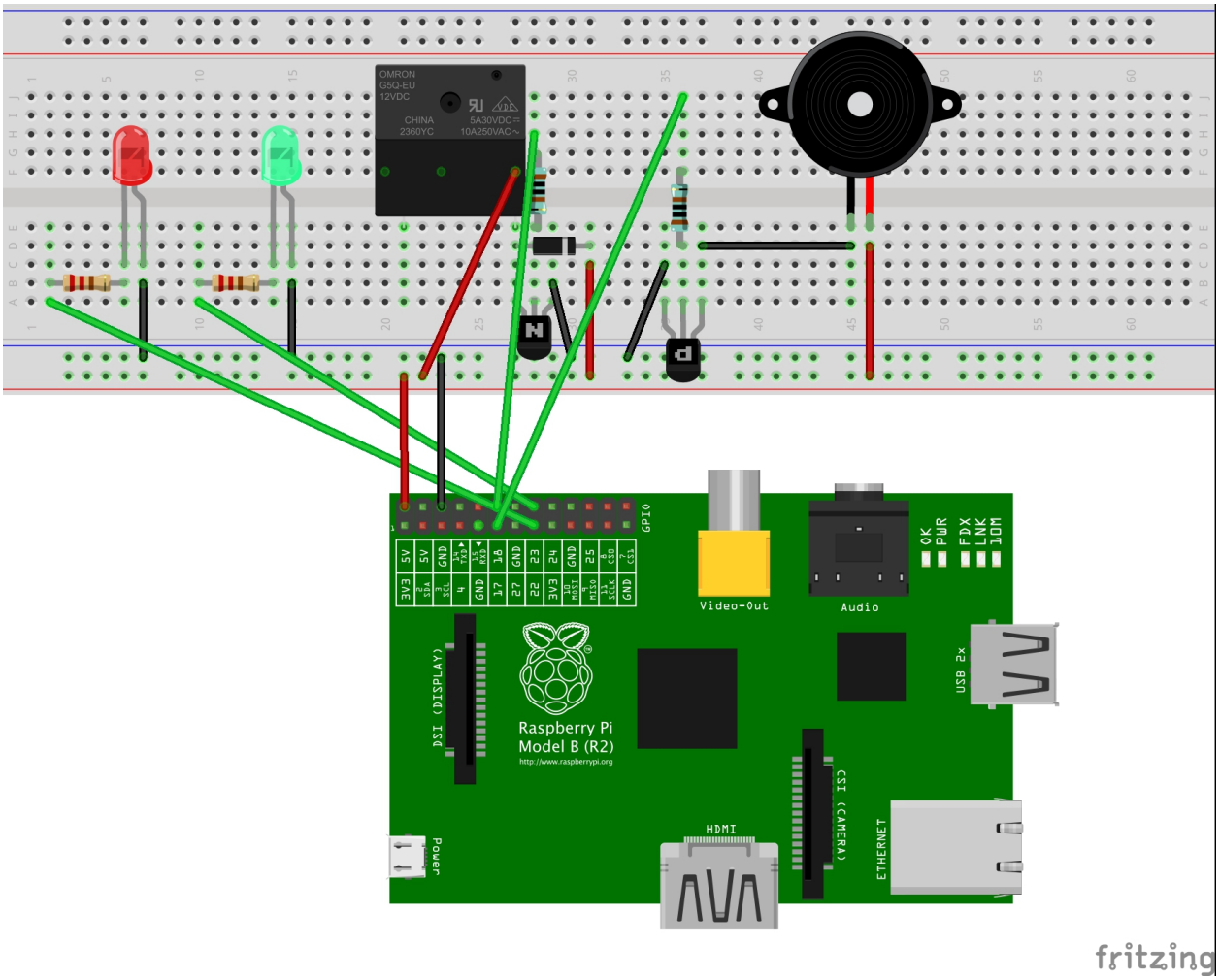

Figura 5.3: Prototipo del circuito realizzato

5.2.4 La piattaforma software utilizzata: node.js

Per quanto riguarda la scelta di linguaggio e piattaforma per lo sviluppo del software sullo smart device, le possibilità sono svariate, e non è assolutamente semplice valutare la scelta migliore. Innanzi tutto occorre considerare quali saranno i vari moduli software di cui l'applicazione necessita: alla base di tutto ci deve essere un modulo per l'interazione web con il server, che deve permettere al dispositivo di comunicare sia con il protocollo web socket, sia con classiche chiamate HTTP. Gli altri due moduli di comunicazione saranno quelli dedicati all'interazione con l'utente: il primo deve presumibilmente essere un programma python in grado di interagire con lo stack NFC, mentre il secondo dovrà preoccuparsi di sfruttare lo stack Bluetooth low energy. Per tutte le attuazioni necessarie, è sufficiente un componente software per pilotare i pin della GPIO, mentre al fine di garantire la possibilita di funzionamento offline è necessario un modulo applicativo dedicato per lo storage di dati in forma semplice.
Ricapitolando, i moduli principali di cui la parte applicativa del progetto dovrà comporsi sono sostanzialmente cinque:

- modulo per la comunicazione con il server via internet;
- modulo di interfacciamento con l'utente via bluetooth low energy;
- modulo di interfacciamento con l'utente via near field communication;
- modulo di attuazione;
- modulo per lo storage locale.

Sebbene Raspberry pi supporti una grande varietà di linguaggi e framework, come C, C++, Python, Pava e probabilmente anche .NET [24], per lo sviluppo si è scelto anche in questo caso di utilizzare linguaggio JavaScript sulla piattaforma node.js.

I motivi sono molteplici: in primo luogo bisogna tenere in considerazione che la piattaforma server dovrà poter comunicare in modo agile con il device tramite il protocollo websocket, di conseguenza la possibilità di utilizzare sull'istanza node del device il modulo client di socket.io è un grande vantaggio. Inoltre, avere a disposizione un package manager come NPM, con migliaia di moduli gratuiti e supportati da una community tra le più attive nel mondo dello sviluppo, rappresenta un altro punto a favore di node.js, soprattutto nell'ottica di sviluppare un progetto che necessita di una grande varietà di librerie, dalla gestione dell'IO con periferiche hardware fino alla comunicazione in rete. Il motore di Node.js (V8) infatti può eseguire nativamente codice C o C++, incapsulandolo ed esponendolo allo sviluppatore come funzioni JavaScript. Ultimo importante motivo che ha guidato la scelta è la scalabilità del framework node.js: Raspberry pi ha risorse computazionali molto limitate, quindi l'utilizzo di framework più pesanti sarebbe stato problematico e probabilmente inutile.

L'installazione di node.js può essere completata utilizzando sorgenti già compilati, in quanto l'installazione tramite compilazione di sorgenti può richiedere diverse ore e dare qualche problema dal punto di vista di gestione della memoria. Sono disponibili online diverse guide, il processo di installazione è molto semplice. Da terminale:

```
$ wget http://node-arm.herokuapp.com/node_latest_armhf.deb
$ sudo dpkg -i node_latest_armhf.deb
```

L'installazione dovrebbe completarsi in pochi minuti, per verificare il buon esito dell'operazione è sufficiente digitare sul terminale:

```
node -v
```

che ovviamente dovrebbe produrre in output la versione di node.js installata.

5.3 Progettazione ed implementazione dell'applicazione

Lo sviluppo dell'intero software che lo smart device dovrà eseguire può essere suddiviso in differenti parti. Inizialmente si cercherà di definire l'insieme dei flussi applicativi che coinvolgono i tre attori del sistema, dopodiché si prenderà in esame ogni singolo modulo per analizzarne nel dettaglio gli aspetti implementativi più rilevanti. Per un'idea più precisa di quelli che saranno i flussi applicativi che vedranno coinvolto lo smart device, sono proposti dei sequence diagram UML in cui compaiono l'utente, lo smart device e il server. Mentre utente e server saranno riportate come singole entità, si dettaglierà maggiormente la struttura dello smart device, che sarà separato nelle sue diverse componenti. Nel diagramma in Figura 5.4 sono riportati i messaggi scambiati tra lo smart device e il server per la sincronizzazione dei dati: il device potrebbe trovarsi momentaneamente offline, ma deve permettere comunque l'accesso agli utenti dell'organizzazione. Di conseguenza deve ricevere dal server i dati attraverso i quali poter effettuare autenticazione offline e, in caso abbia registrato degli accessi offline, sincronizzarli appena la connessione torna attiva. I flussi di comunicazione con l'utente, sono invece rappresentati nei diagrammi in Figura 5.6, Figura 5.7 e Figura 5.5. Il diagramma in cui viene evidenziato l'accesso con l'utilizzo del tag NFC, denota la possibilità di effettuare l'accesso anche in caso il dispositivo sia offline.

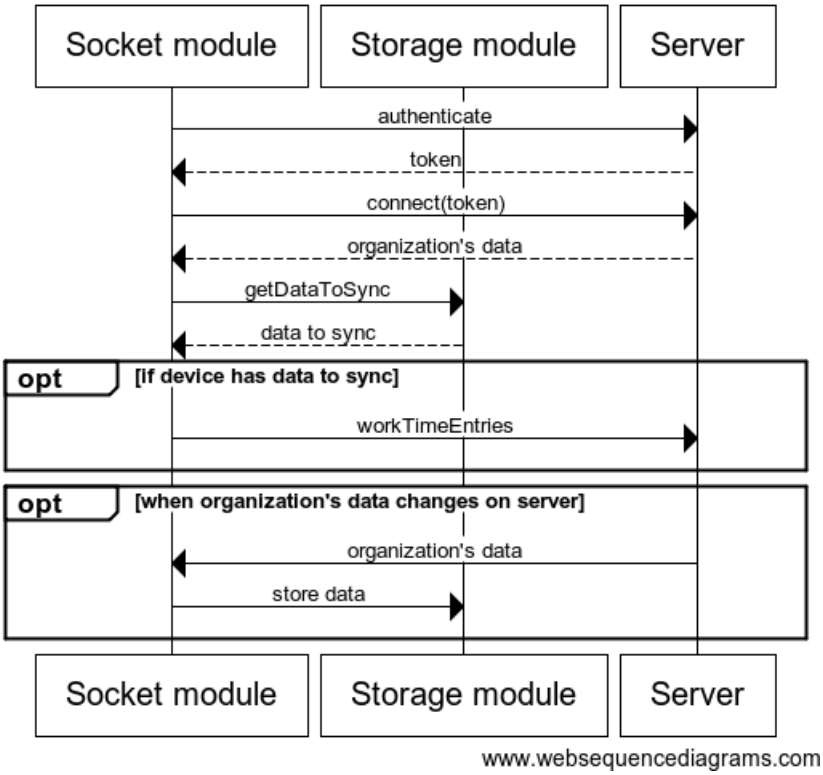

Figura 5.4: Una panoramica dei flussi di comunicazione tra device e server per la sincronizzazione dei dati offline

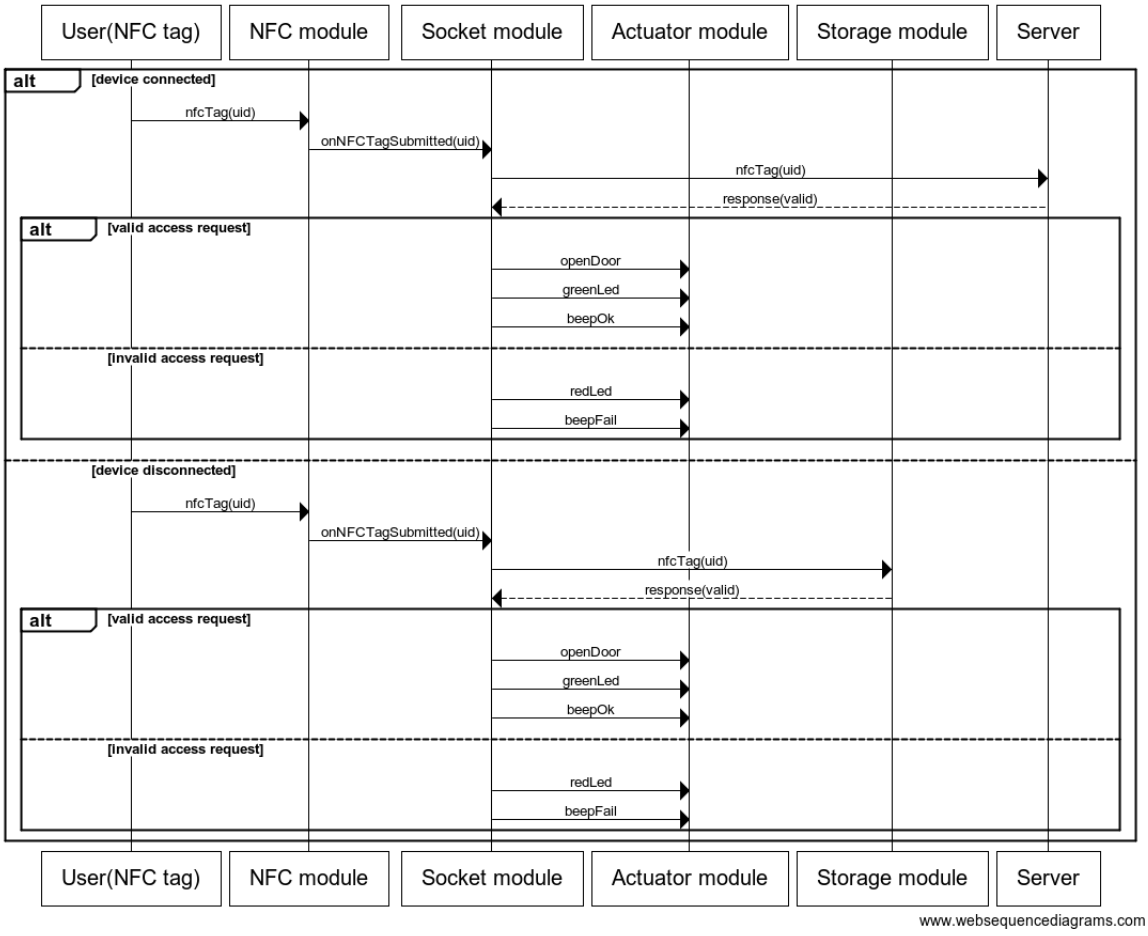

Figura 5.5: Interazione tra utente e smart device per la timbratura con tag NFC

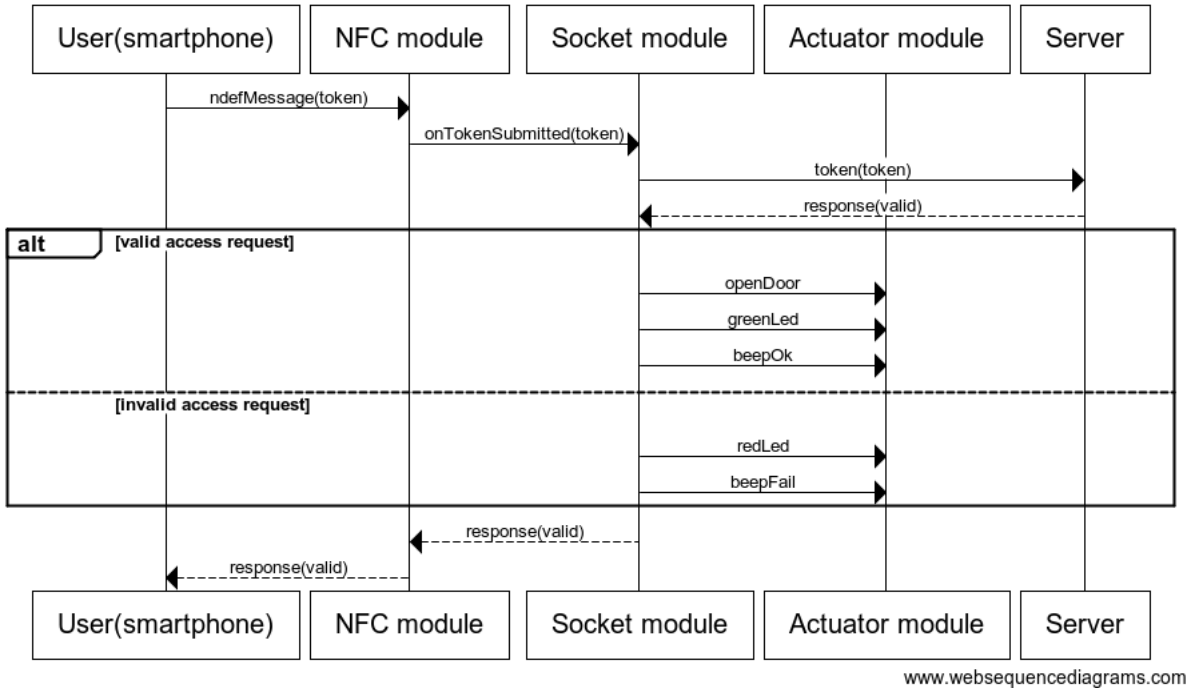

Figura 5.6: Interazione tra utente e smart device per la timbratura con smartphone via NFC

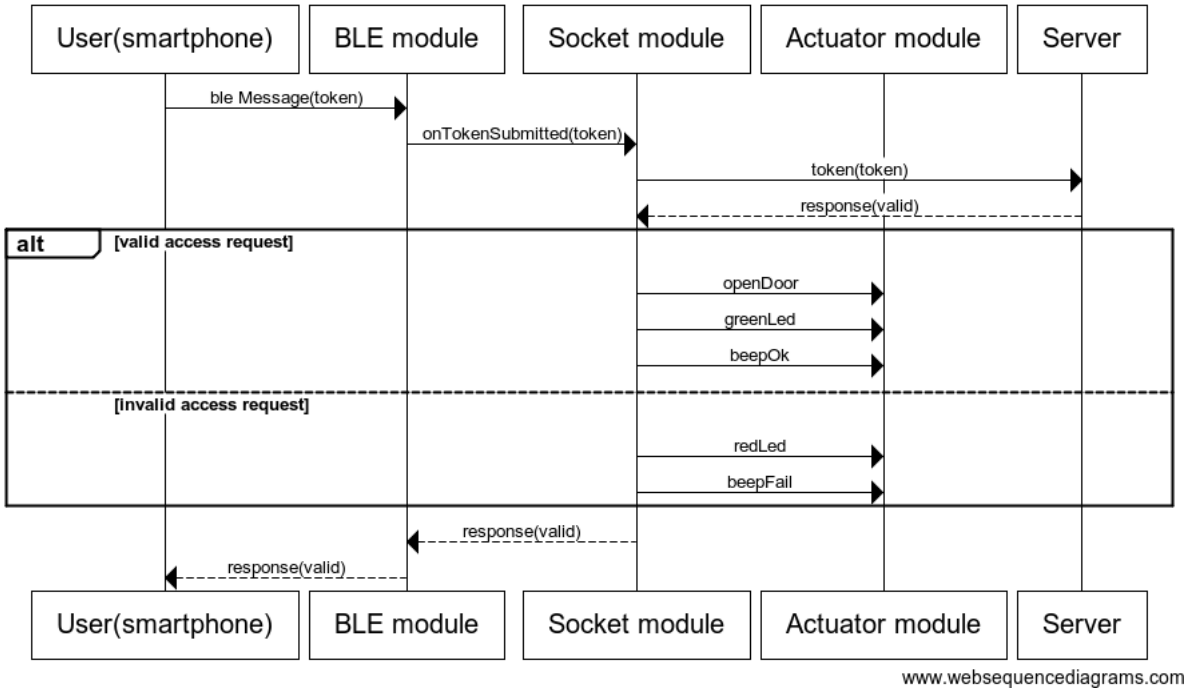

Figura 5.7: Interazione tra utente e smart device per la timbratura con smartphone via BLE

5.3.1 La comunicazione con il server: socket.js

Il primo modulo preso in considerazione è il controller principale dell'applicazione, in quanto deve coordinare i vari componenti e comunicare direttamente con il server. Analizzando i diagrammi di sequenza, si può notare che il modulo socket entra in gioco in tutti i differenti scenari rappresentati:

- durante lo startup del programma deve preoccuparsi di autenticare il device verso il server e connettere la socket utilizzata per la comunicazione;
- può ricevere messaggi dal server contenenti i dati aggiornati dell'organizzazione, per poter funzionare offline;
- è richiamato dai moduli NFC e BLE in seguito ai tentativi di accesso dell'utente, e deve preoccuparsi dell'invio verso il server dei dati ricevuti;
- sulla base dell'esito ricevuto dal server, sfrutta il sistema di attuazione per le azioni di notifica e di (eventuale) apertura porta.

I componenti utilizzati

Il componente npm utilizzato per la comunicazione con il server è socket.io-client, la controparte lato client della libreria socket.io installata sul server. Questa libreria consente di scambiare messaggi con il server utilizzando il protocollo web socket e registrare handler sia in corrispondenza di determinati eventi (ad es. connessione riuscita, connessione chiusa, riconnessione), sia in corrispondenza di determinate tipologie di messaggi ricevuti. L'apertura della socket di comunicazione con il server è subordinata all'utilizzo di un token segreto, che lo smart device ottiene in seguito ad una chiamata di autenticazione HTTP verso il server, utilizzando il package request-json.

Dettagli implementativi

L'interfaccia del modulo socket (che chiameremo controller) è composta dai due metodi attraverso i quali è notificato dai moduli Bluetooth ed NFC, in seguito ad un'interazione con l'utente.

```
...
exports.onNFCTagSubmitted = onNFCTagSubmitted;
exports.onTokenSubmitted = onTokenSubmitted;
```

Il metodo `onTokenSubmitted` è richiamato quando l'utente invia il proprio token di autenticazione allo smart device, attraverso BLE oppure NFC. La sua implementazione è deducibile da quello che è il comportamento atteso: la condizione standard di utilizzo prevede una connessione via socket tra il dispositivo e il server. In tal caso, l'unica operazione che il dispositivo compie è l'inoltro del messaggio ricevuto dall'utente (contenente il token di autenticazione) attraverso il canale sicuro. Sulla base della risposta ricevuta dal server sceglie se aprire la porta e notificare l'esito positivo, piuttosto che notificare semplicemente la condizione di errore.

```
function onTokenSubmitted(stringData, callback){
    if(isOnline()){

        var token = ...

        _socket.emit('tokenSubmitted', {
            token: token,
            workTimeEntryType: type === 'I' ? 'in' : 'out'
        }, function(response){
```

```
            if(response.responseCode == 200){
                actuatorService.openDoor();
            } else {
                actuatorService.error();
            }

            callback(response);
        });

    } else {
        actuatorService.error();

        callback({
            responseCode: 503,
            message: 'device is offline, cannot authenticate'
        });
    }
}
```

Nel caso in cui il dispositivo non sia connesso con il server, non gli è possibile la decodifica del token di autenticazione. Di conseguenza è notificato all'utente l'errore, che però può comunque effettuare l'accesso con il tag NFC a lui associato. A differenza di quel che accade quando l'utente usa l'applicazione per smartphone, è infatti possibile accedere con il tag NFC sia quando il dispositivo è online (demandando quindi le operazioni al server), sia quando il dispositivo è offline, utilizzando il modulo per la gestione dell'autenticazione offline. Il modulo ha la possibilità di immagazzinare informazioni relative all'organizzazione alla quale il device è collegato, per poi poterle sfruttare qualora la connettività internet sia

assente o vi sia un problema sul server.

```
function onNFCTagSubmitted(uid){
    if(_readingUid){
        ...
        _socket.emit('uid', {
            uid: uid
        });

        ...
    } else {
        if(isOnline()){

            ...

        } else {

            if(offlineHelper.authorizeAccess(uid)){
                actuatorService.openDoor();
            } else {
                actuatorService.error();
            }
        }
    }
}
```

Oltre allo stato di funzionamento normale, in cui il lettore NFC è usato dagli utenti per registrare l'accesso e la relativa timbratura, questo può essere sfruttato dagli amministratori dell'organizzazione (attraverso la web application) per leggere l'i-

dentificatore univoco (uid) di un Tag NFC da associare ad un utente. Per questo motivo nello snippet di codice riportato in precedenza è presente il primo `if`.

Come già anticipato in fase introduttiva il modulo deve preoccuparsi anche di gestire la socket verso il server: inizialmente deve autenticarsi e richiedere la connessione sicura, dopodichè, una volta connesso, registrare i vari handler agli eventi sul canale (disconnessione, timeout, etc) in modo da poter reagire nel modo più adatto.

5.3.2 La comunicazione via BLE

Il modulo di comunicazione Bluetooth low energy dovrà essere una delle due interfacce che gli utenti potranno sfruttare, attraverso la smartphone app, per la comunicazione con lo smart device. In buona sostanza si è implementato un servizio GATT con cui l'utente interagisce per autenticarsi. Per questioni di sicurezza, considerato il fatto che le informazioni viaggiano "nell'etere" e potrebbero essere soggette a *sniffing*, si è scelto di richiedere come prerequisito il "bonding" tra lo smartphone utente e il dispositivo.

Il servizio GATT

Lo sviluppo dell'interfaccia del servizio parte dalla definizione dell'interazione possibile tra utente e dispositivo, e dalla tipologia di informazioni scambiate:

- l'utente, tramite smartphone si connette al device;
- invia il proprio token di autenticazione;
- registra la notifica al completamento dell'operazione da parte del device;
- una volta terminata l'operazione, riceve la notifica e si disconnette.

Considerato che la massima quantità di informazioni che può essere trasmessa in una singola operazione da bluetooth LE è limitata a 20 bytes, per la trasmissione del token è necessario prevedere:

- una caratteristica in sola scrittura che l'utente utilizza per scrivere singoli *chunks* di 20 byte ciascuno. Il device, progressivamente, ricostruisce il token pezzo per pezzo;
- una caratteristica in sola scrittura che l'utente utilizza per scrivere l'ultimo *chunk*. In questo modo il device sa che il token è completo, e può cominciare le operazioni per la sua validazione;
- una caratteristica alla quale l'utente si registra per ricevere notifica appena il device, tramite il server, completa le operazioni di autenticazione ed autorizzazione.

Per ragioni di sicurezza, tutte le caratteristiche sono definite come *secure*, in modo che l'interazione sia possibile soltanto in caso il dispositivo e lo smartphone dell'utente abbiano già effettuato un primo *pairing* (tecnicamente, siano tra di loro *bonded*). Sempre per ragioni di sicurezza, si è scelto di aggiungere anche una caratteristica in sola lettura, attraverso la quale il device può presentare una sorta di firma digitale per garantire la propria autenticità. Anche se la caratteristica risulta presente, al momento il meccanismo non è implementato.
È necessario scegliere 5 GUID personalizzati, in quanto nel nostro caso il servizio fornito e le caratteristiche utilizzate non trovano definizione tra i tipi standard definiti nel protocollo GATT. Ricapitolando:
Service f000cc40-0451-4000-b000-000000000000

- DigitalSignatureCharacteristic - **read** - f000cc41-0451-4000-b000-000000000000.
- WriteTokenChunkCharacteristic - **write** - secure - f000cc42-0451-4000-b000-000000000000.

- WriteLastTokenChunkCharacteristic - **write** - secure - f000cc44-0451-4000-b000-000000000000.
- NotifyCharacteristic - **notify** - secure - f000cc43-0451-4000-b000-000000000000.

Dettagli implementativi

Per interagire con lo stack blueZ ed implementare il servizio GATT, node.js ha a disposizione un modulo dedicato e piuttosto completo: bleno [4]. Bleno espone metodi implementati in C sotto forma di API JavaScript, che possono quindi essere sfruttate con molta facilità da un programma node. Ha una sintassi dichiarativa che consente di definire le diverse caratteristiche del servizio GATT da implementare, con la possibilità di impostare per ognuna proprietà e relative funzioni di callback come handler. Questa l'implementazione del servizio:

```
var bleno = require('bleno');
var BlenoPrimaryService = bleno.PrimaryService;
var BlenoCharacteristic = bleno.Characteristic;
...
...

var WriteLastTokenChunkCharacteristic = function() {
  WriteLastTokenChunkCharacteristic.super_.call(this, {
    uuid: 'f000cc44-0451-4000-b000-000000000000',
    properties: ['write'],
    secure: ['write']
  });
};
util.inherits(WriteLastTokenChunkCharacteristic, BlenoCharacteristic);
```

```
WriteLastTokenChunkCharacteristic.prototype.onWriteRequest =

function(data, offset, withoutResponse, callback) {

    var newBuffer = Buffer.concat([dataBuffer, data]);
    dataBuffer = newBuffer;

    _listener.onTokenSubmitted(dataBuffer.toString(),
      function(response){

        var data = new Buffer(4);
        data.writeUInt32LE(response.responseCode, 0);
        notifyCallback(data);

        setTimeout(function(){
            bleno.disconnect();
        }, 500)

    });

    callback(this.RESULT_SUCCESS);
};
...

function SampleService() {
  SampleService.super_.call(this, {
    uuid: 'f000cc40-0451-4000-b000-000000000000',
```

```
      characteristics: [
        new DynamicReadOnlyCharacteristic(),
        new WriteTokenChunkCharacteristic(),
        new WriteLastTokenChunkCharacteristic(),
        new NotifyOnlyCharacteristic()
      ]
    });
}
util.inherits(SampleService, BlenoPrimaryService);

bleno.on('advertisingStart', function(error) {

  if (!error) {
    bleno.setServices([ new SampleService() ]);
  }
});

bleno.on('stateChange', function(state) {

    if (state === 'poweredOn') {
        bleno.startAdvertising('raspy',
          ['f000cc40-0451-4000-b000-000000000000']);
    } else {
        bleno.stopAdvertising();
    }
});
```

Oltre alle API per l'implementazione del servizio e agli handler per la gestione dello stato dell'adapter bluetooth, è stata riportata soltanto la definizione della caratteristica `WriteLastTokenChunkCharacteristic`, in quanto particolarmente significativa per comprendere il comportamento dello smart device. Nel momento in cui l'utente completa la scrittura, il device esegue la callback e, contestualmente, invoca sul listener (il controller dell'applicazione) la funzione `onTokenSubmitted`, passando come parametro il token. A questo punto il device resta in attesa del completamento dell'operazione e, una volta ricevuta risposta, la invia all'utente utilizzando la funzione `notifyCallback`, che tramite apposita caratteristica notifica l'utente. Una volta completato questo flusso di operazioni, il device disconnette automaticamente l'utente, in modo da poter essere utilizzato anche da altri.

Problematiche riscontrate

L'implementazione del servizio bluetooth LE è stata probabilmente la parte più difficile di tutto il lavoro. Sebbene a livello teorico il servizio offerto e il protocollo di comunicazione non risultino particolarmente complessi, l'utilizzo della libreria bleno ha richiesto una fase di test piuttosto elaborata, nella quale si sono effettuati numerosissimi tentativi per cercare di evitare errori e situazioni di stallo che compromettevano il funzionamento del device. Probabilmente la diffusione limitata della libreria e la peculiarità del sistema utilizzato hanno fatto sì che vi fossero diverse situazioni instabili e problematiche complesse non solo da risolvere, ma anche da individuare. Grazie al supporto della piccola community nata attorno alla libreria bleno, si è riusciti ad ottenere una versione stabile del prototipo, ma occorre un'ulteriore analisi approfondita prima della messa in produzione. Questi i principali interventi effettuati:

- l'installazione di blueZ deve essere necessariamente limitata alla versione 4, in quanto la libreria è in conflitto con blueZ 5;
- è stato installato nel raspberry un servizio per lanciare allo startup il demone `bluetoothd`, utilizzato da bleno per l'utilizzo di caratteristiche che richiedono *bonding*;
- in caso di crash dell'applicazione, deve essere manualmente *terminato* il processo `l2cap-ble`, che altrimenti impedisce ulteriori utilizzi dell'interfaccia bluetooth. Per questo scopo è stato utilizzato il file `closing-handlers.js` dentro alla cartella `/config`;
- lo smartphone dell'utente non deve mantenere connessioni aperte con il dispositivo in caso di crash, altrimenti si verificano problemi in caso di connessioni successive. Per questo motivo la disconnessione è forzata dal device, al termine delle operazioni di autenticazione e autorizzazione.

5.3.3 La comunicazione via NFC

La comunicazione NFC entra in gioco in due diverse "varianti" nello scenario di interazione tra utente e dispositivo al momento dell'accesso: l'utente che attraverso la smartphone application trasmette il token di autenticazione al device, e l'utente che utilizza un tag NFC personale per autenticarsi verso il device. In entrambi gli scenari, il modulo node.js utilizzato è sempre lo stesso, ed è definito internamente al file `nfc.js`. Si tratta di un singleton responsabile dell'interazione con il lettore NFC. In realtà, visto che lo stack NFC utilizzato è interamente scritto in python, il modulo sarà composto da due differenti parti: la prima sarà scritta in JavaScript, ed interagirà direttamente con il controller dell'applicazione in corrispondenza di eventi scatenati dalla seconda parte, un programma python scritto proprio per l'utilizzo dello stack NFC.

I tag utilizzati per l'autenticazione utente sono gli NXP nTag21x, che implementano le caratteristiche previste dal tipo 2 definito da NFC forum. Questa particolare tipologia di tag è compatibile con nfcpy, in più possiede alcune funzionalità per la sicurezza che possono essere sfruttate per garantire l'autenticità del tag, caratteristica piuttosto importante se si considera la modalità con cui il tag viene utilizzato nell'applicazione. Oltre all'identificativo univoco, i tag possono restituire (in seguito ad un determinato comando inviato dal lettore), la firma digitale dell'identificativo stesso, firma che può essere verificata utilizzando la chiave pubblica del produttore. L'algoritmo di firma digitale utilizzato da NXP si basa sulla crittografia su curve ellittiche *secp128r1*[23]. Al momento non è ancora stata implementata la verifica della firma digitale, tuttavia è in previsione tra le successive attività progettuali.

Lo sviluppo del programma python

Il flusso delle operazioni che il programma python deve eseguire può essere descritto attraverso il diagramma in Figura 5.8. Il comportamento alla base del programma è un polling sul lettore, grazie al quale resta in attesa di un tag piuttosto che di un dispositivo con cui comunicare in p2p (smartphone). In conseguenza di ciò che l'utente avvicina al lettore, le operazioni sono diverse. Nel caso sia utilizzato un tag NFC, non vi è una comunicazione bidirezionale, in quanto le uniche operazioni effettuate sono di "lettura": una volta che lo smart device ottiene l'identificativo univoco e la firma digitale del tag, il programma python le trasmette a node.js, che si occuperà poi di effettuare le verifiche sul tentativo di accesso ed intraprendere le azioni successive. Diversamente, nel caso in cui l'utente utilizzi il proprio smartphone, la prima cosa che lo smart device deve fare consiste nell'inviargli in un messaggio NDEF contenente sorta di firma digitale, in modo che l'applicazione utilizzata dall'utente possa verificarne l'autenticità. A questo

punto il programma resta in attesa di un messaggio NDEF contenente il token di autenticazione e, una volta ricevuto, lo inoltra al programma node che si occuperà di verificarlo. Terminate le operazioni, il programma dovrà comunicare all'utente l'esito con un ultimo messaggio NDEF.

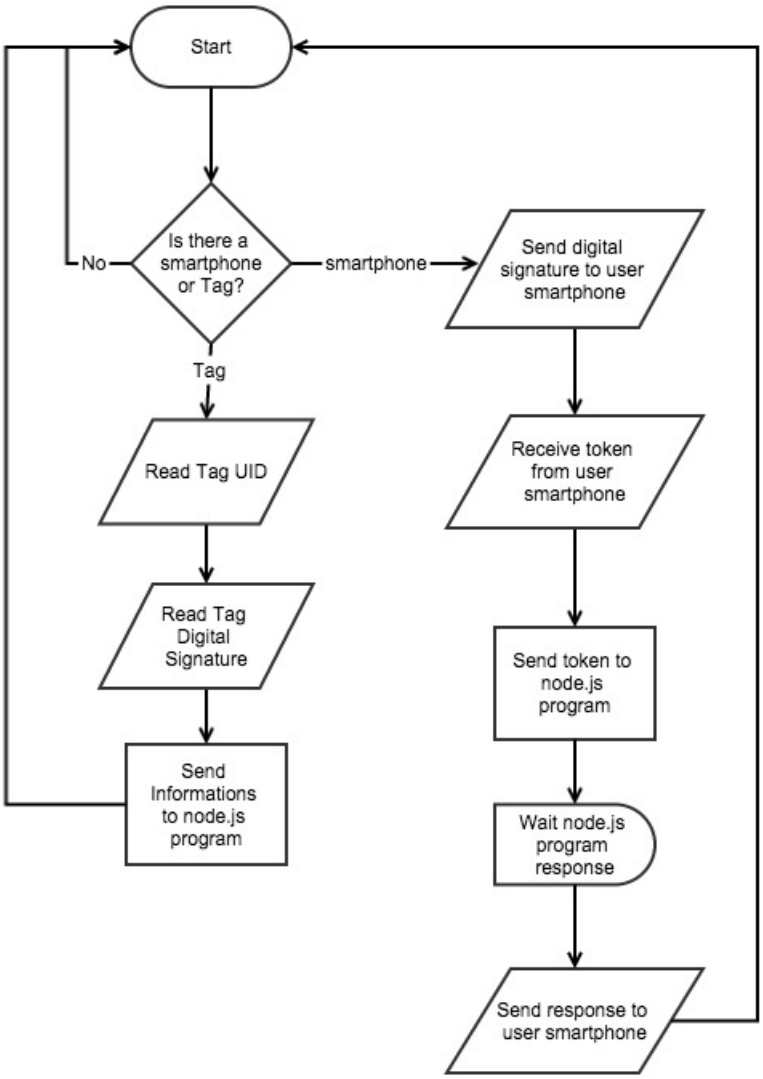

Figura 5.8: Una panoramica dei flussi di comunicazione tra device e server per la sincronizzazione dei dati offline

Il codice python scritto è suddiviso in due differenti file. L'entry point del programma è il file nfc_controller.py, in cui è definita la classe NFCController per la quale si riporta il costruttore che implementa il meccanismo di polling:

```
def __init__(self):

    rdwr_options = {
        'on-connect': self.tag_read
    }

    llcp_options = {
        'on-connect': self.peer_connected,
        ...
    }

    device = 'tty:AMA0:pn53x'

    while True:
        self.p2pInitiated = False
        while(not self.p2pInitiated):
            #status 0
            self.clf = nfc.ContactlessFrontend(device)
            try:
                while not self.clf.connect(llcp=llcp_options,
                        rdwr=rdwr_options):
                    pass
            except:
                pass
```

```
            #status 1
            self.clf = nfc.ContactlessFrontend(device)
            try:
                self.clf.connect(llcp=llcp_options)
            except:
                pass

            #p2p completed, restart from beginning
            #(while true won't never end)
```

Il metodo `self.clf.connect` serve per restare in attesa sul lettore. Si può notare la presenza di due cicli while innestati: il primo è necessario per non far terminare mai il programma, mentre il secondo è utile per gestire lo stato del programma: nello stato iniziale infatti deve restare in attesa di un tag o di un peer, mentre entra nello stato secondario solo dopo aver inviato il primo messaggio al peer, uscendo dal while interno grazie all'impostazione del flag `p2pInitiated`. Le uniche opzioni passate al metodo connect sono le callback che possono essere invocate da nfcpy, di cui si riporta l'implementazione:

```
    def tag_read(self, tag):
        uid = binascii.hexlify(tag.uid)
        signature = None
        try:
            sig = tag.transceive("\x3C\x00", timeout=1)
            assert len(sig) == 32 or len(sig) == 34
            if len(sig) == 34
              and nfc.tag.tt2.crca(sig, 32) == sig[32:34]:
                sig = sig[0:32]
```

```
            signature = binascii.hexlify(sig)
            c = zerorpc.Client()
            c.connect("tcp://127.0.0.1:4242")
            c.tag(uid, signature)
        except:
            pass
        time.sleep(3)

    def peer_connected(self, llc):

        if(not self.p2pInitiated):
            self.snep_client = nfc.snep.SnepClient(llc)
            commThread = threading.Thread(
              target=self.send_signature, args=())
            commThread.start()
        else:
            self.snep_server = CustomSnepServer(llc)
            self.snep_server.start()

        return True
```

Il protocollo di comunicazione p2p utilizzato è SNEP (Simple NDEF Exchange Protocol), che si basa su un canale fisico di comunicazione costruito attraverso *Logical Link Control Protocol (LLCP)*. SNEP prevede un semplice meccanismo di request/response attraverso il quale un client ed un server scambiano informazioni: tipicamente è il client che comanda l'interazione, di conseguenza in caso di comunicazione p2p, i ruoli sono intercambiabili tra i due *peer*. Per questo motivo il programma python assume per la ricezione del messaggio il ruolo di

server (sul thread principale), mentre per inviare i due messaggi sono creati due `nfc.snep.SnepClient` (su due thread a parte). Si riporta a tale scopo l'implementazione della classe `CustomSnepServer`, che viene istanziata nella seconda fase del flusso di comunicazione, quando il canale `LLCP` è già inizializzato.

```
... imports and paths

class CustomSnepServer(nfc.snep.SnepServer):
    def __init__(self, llc):
        print "starting server"
        nfc.snep.SnepServer.__init__(self, llc, "urn:nfc:sn:snep")
        #it also has to be a client to push message
        self.snep_client = nfc.snep.SnepClient(llc)

    #method invoked in a separate thread
    def send_result(self, token):

        #call node.js to validate token
        c = zerorpc.Client()
        c.connect("tcp://127.0.0.1:4242")
        response = c.token(token)

        print "STATUS: sending result to peer"
        #connect to peer (as client) and sends the response
        self.snep_client.connect("urn:nfc:sn:snep")
        ndefRecord = nfc.ndef.Record(
          'application/it.spot.io.doorkeeper',
          'NDEF record',
```

```
            response['message'].encode( "utf-8" ))
        self.snep_client.put(nfc.ndef.Message(ndefRecord))

        self.snep_client.close()

    #method called when peer push an ndef message
    def put(self, ndef_message):
        print "STATUS: received data from peer"

        #everything in the first record
        record = ndef_message.pop()

        thread = threading.Thread(target=self.send_result,
          args=([record.data]))
        thread.start()

        return nfc.snep.Success
```

Il fatto che sia ereditata la classe `nfc.snep.SnepServer` rende possibile l'implementazione del metodo `put`, richiamato da nfcpy corrispondenza di un messaggio inviato dal peer. Il comportamento si può evincere dal codice: una volta estratti i dati dal messaggio NDEF, lo snep server invoca il metodo send_result in un thread separato, con il compito di comunicare a node.js il token ed inviare al peer la risposta.

L'esecuzione e la comunicazione con node.js

Il programma scritto per utilizzare lo stack nfcpy deve poter interagire con l'applicazione principale sullo smart device, quindi è necessario instaurare un ap-

posito canale di comunicazione. ZeroRPC [27] è una libreria molto leggera nata per la comunicazione tra entità attive *server-side*, implementa un meccanismo simile alla tradizionale *remote procedure call*, che consente ad un processo client di invocare su un server l'esecuzione di un metodo ed attendere (in modo asincrono) la risposta. Fortunatamente sono disponibili per zeroRPC le librerie python (installabile tramite pip) e JavaScript (NPM), utilizzate entrambe nella nostra applicazione. Il modulo node.js responsabile per NFC, oltre ad inizializzare il programma python dovrà quindi ricoprire il ruolo di "server", fornendo al client (python) la possibilità di richiamare due differenti funzioni in corrispondenza della lettura di un tag e della lettura di un token.

```
var zerorpc = require("zerorpc"),
    pythonShell = require('python-shell'),
    ecdsa = require('./ecdsa');

var _listener = null;

var server = new zerorpc.Server({
    token: function(token, reply){
        _listener.onTokenSubmitted(token, function(response){
            reply(null, response);
        });
    },
    tag: function(uid, signature, reply){
        ...//Signature validation
        _listener.onNFCTagSubmitted(uid);
        reply(null);
    }
```

```
});

//Bind zeroRPC server to local ip address
server.bind("tcp://0.0.0.0:4242");
//Run python program
pythonShell.run("nfc_controller.py");

module.exports = function(listener){
    _listener = listener;
};
```

5.3.4 L'attuazione via GPIO

Uno dei moduli più importanti è quello di attuazione, nel quale è implementata una delle caratteristiche principali del sistema, ovvero l'apertura automatica della porta. Per poter interagire con il mondo circostante, il raspberry dispone di un insieme di PIN digitali per l'I/O, il cui stato (attivo alto oppure basso) può essere letto o scritto via software. Esistono un gran numero di librerie dedicate a questo scopo, ma per restare coerenti con l'intero "ecosistema" realizzato, si è scelto il modulo NPM `onoff`. La libreria, scritta quasi interamente in JavaScript, sfrutta internamente l'interfaccia Linux `GPIO sysfs` per l'esportazione del controllo di uno o più PIN GPIO dal Kernel allo spazio utente, fornendo API sincrone ed asincrone per la loro lettura e scrittura.

Il modulo può essere sfruttato per aprire il relay collegato alla porta e dare feedback positivo, in caso la fase di autenticazione con l'utente vada a buon fine e gli orari siano validi, per dare un feedback negativo altrimenti. I metodi principali esportati dal modulo sono quindi `error` per notificare l'errore ed `openDoor` per aprire la porta, richiamati dal controller in corrispondenza degli opportuni eventi:

```
var Gpio = require('onoff').Gpio,
    relay = new Gpio(22, 'out'),
    redLed = new Gpio(24, 'out'),
    greenLed = new Gpio(23, 'out'),
    buzzer = new Gpio(18, 'out');

exports.error = function(){
    var cont=0;
    errorInterval = setInterval(function() {
        if(cont<6){
            redLed.writeSync(redLed.readSync() === 0 ? 1 : 0);
            buzzer.writeSync(buzzer.readSync() === 0 ? 1 : 0);
            cont++;
        } else {
            clearInterval(errorInterval);
        }
    }, 100);
};
exports.openDoor = function (){
    greenLed.writeSync(1);
    buzzer.writeSync(1);
    relay.writeSync(1);

    setTimeout(function(){
        greenLed.writeSync(0);
        buzzer.writeSync(0);
        relay.writeSync(0);
```

```
    }, 500);
};
```

I PIN utilizzati sono quattro, ognuno dedicato al controllo di un preciso componente del circuito elettronico realizzato:

- PIN 18 per emettere un suono con il buzzer;
- PIN 23 per controllare il led verde;
- PIN 24 per il led rosso;
- PIN 22 per aprire il relè collegato all'interruttore elettrico sulla porta.

5.3.5 Il meccanismo di storage locale

L'ultimo modulo descritto è quello utilizzato dal sistema al fine di garantire un funzionamento minimo anche offline, `offline-helper.js`. Il modulo deve essere interrogato dal controller quando il server non è raggiungibile, per completare l'autenticazione di un utente che effettua l'accesso utilizzando un tag NFC. In questa prima fase prototipale è stata scelta questa modalità come unica possibilità per l'accesso offline, in quanto l'identificativo univoco presente nel tag è un'informazione non cifrata e comprensibile per lo smart device, a differenza del token di autenticazione inviato dallo smartphone degli utenti. L'identificativo può essere direttamente collegato agli utenti dell'organizzazione nella quale lo smart device è installato, di conseguenza le uniche informazioni di cui il device necessita sono la lista degli UID NFC, e l'insieme degli orari in cui è consentito l'accesso nell'organizzazione. Entrambe le informazioni sono inviate periodicamente dal server attraverso la socket e memorizzate per eventuali interrogazioni. Per il salvataggio dei dati è stato utilizzato il modulo NPM `node-persist`, una semplice libreria

che fornisce un meccanismo di storage locale per scrivere e leggere oggetti JavaScript serializzati in formato JSON. Le funzioni esportate dal modulo sono le 4: le prime due vengono richiamate dal controller per memorizzare nello storage locale i dati inviati dal server, `authorizeAccess` è invocata per delegare al modulo l'autorizzazione di un accesso con tag NFC, mentre syncData è richiamata per inviare al server l'insieme degli accessi avvenuti offline, al fine di inserirli automaticamente nel sistema. Si tralascia l'implementazione.

```
exports.storeMembers = storeMembers;
exports.storeWorkingDays = storeWorkingDays;
exports.authorizeAccess = authorizeAccess;
exports.syncData = syncData;
```

Capitolo 6

Progettazione e sviluppo dell'applicazione Android

In questo capitolo è presentato lo sviluppo dell'ultimo nodo del sistema, ovvero l'applicazione per smartphone che gli utenti potranno utilizzare per interagire con lo smart device. Inizialmente si introduce molto brevemente la piattaforma applicativa utilizzata per lo sviluppo, descrivendo le caratteristiche principali che ne hanno favorito la scelta. Successivamente si descrivono gli strumenti che la piattaforma mette a disposizione, sotto forma di API, per l'utilizzo delle tecnologie di comunicazione NFC e Bluetooth LE, alla base dello sviluppo di questo prototipo. A conclusione del capitolo si fornisce una breve descrizione dell'architettura e del funzionamento dell'applicazione, focalizzando ancora una volta l'attenzione sugli aspetti maggiormente interessanti dal punto di vista dell'architettura del sistema complessivo: l'interazione con il server e l'interazione con lo smart device. Parte del materiale utilizzato in questo capitolo è tratto dal sito Android Developers [1], essenziale sia per sviluppatori principianti, sia per esperti che necessitano di approfondire aspetti tecnici nel dettaglio.

Anche se l'applicazione dovrà in futuro essere rilasciata su tutte e tre le prin-

cipali piattaforme mobile, in questa prima fase lo sviluppo è stato limitato all'ambiente Google Android. Le motivazioni sono molteplici: in primo luogo il tempo necessario per lo sviluppo multipiattaforma sarebbe sicuramente maggiore, poiché sebbene possano essere sfruttati framework per la generazione di codice, la natura funzionale strettamente *platform-dependent* dell'applicazione renderebbe necessari numerosissimi interventi di adattamento. Inoltre, Android è la piattaforma maggiormente diffusa che implementa contemporaneamente sia le API per la comunicazione via Bluetooth Low Energy, sia quelle per la comunicazione via NFC. Lo sviluppo su questa piattaforma rende quindi possibile mettere in pratica tutti i diversi casi d'uso, aspetto fondamentale per la natura di questo progetto. Non è scopo di questo lavoro esaminare l'architettura e il funzionamento della piattaforma in generale, ma è molto importante capire come il sistema consente agli sviluppatori di utilizzare le diverse API per la comunicazione con Near Field Communication e Bluetooth Low Energy.

6.1 Le API NFC per la comunicazione peer to peer

Le API per NFC sono disponibili per gli sviluppatori dalla versione del sistema operativo 2.3 (Gingerbread), di conseguenza Android risulta la prima piattaforma mobile largamente diffusa ad aver integrato il supporto alla tecnolgia. Nelle ultime versioni, oltre alla modalità di funzionamento per la lettura e la scrittura di tag e alla modalità di comunicazione *peer to peer*, Android dà la possibilità di utilizzare *card-emulation*, che consente alle applicazioni di poter essere sfruttate per scenari differenti da quelli abitualmente aperti agli sviluppatori (pagamenti sicuri, etc.). Sebbene sia importante in fase di analisi conoscere a fondo le differenti modalità al fine di scegliere la più adatta ai propri scenari applicativi, nel caso del progetto in esame ci si limiterà a discutere nel dettaglio gli aspetti dedicati all'utilizzo dello

smartphone NFC in modalità *peer to peer*.

6.1.1 Utilizzare NFC

Se lo smartphone dispone del chip NFC, Android mette a disposizione un'*adapter* dedicato alla comunicazione: `NFCAdapter`. L'adapter è alla base di tutte le funzionalità NFC, come ad esempio la scrittura di un tag oppure l'invio di un messaggio NDEF. Se NFC è disponibile nello smartphone, l'istanza dell'adapter può essere ottenuta semplicemente invocando il metodo

```
public static android.nfc.NfcAdapter.getDefaultAdapter(Context c)
```

passando come parametro il contesto dell'`Activity`. Il prerequisito affinché l'operazione vada a buon fine è la specifica dei permessi per l'utilizzo di NFC nel file *manifest* attraverso la stringa

```
<uses-permission android:name="android.permission.NFC" />.
```

Se il metodo statico restituisce il valore `null`, allora significa che NFC non è disponibile per lo smartphone in questione: qualora la tecnologia risulti cruciale ai fini applicativi, è possibile specificare nel file *manifest* che l'applicazione non sia resa visibile ai dispositivi non abilitati, in modo da non creare una cattiva esperienza d'uso.

6.1.2 La creazione di messaggi NDEF

I messaggi NDEF sono alla base della comunicazione NFC e le classi messe a disposizione da Android per la loro manipolazione sono sfruttate sia nel momento in cui si deve preparare il messaggio da inviare (con Android Beam), sia nel momento in cui si devono estrarre le informazioni dal messaggio ricevuto.
Un messaggio NDEF è composto da uno o più record; nel primo di essi sono definite le informazioni relative al contenuto dell'intero messaggio. Il primo record è

infatti utilizzato dal dispositivo che legge il messaggio al fine di comprenderne il tipo di contenuto. L'utilizzo di un numero multipo di record per messaggio è tipicamente limitato al caso in cui la dimensione del *payload* sia maggiore di quella che può effettivamente essere inclusa in una singola istanza. Per la creazione del record, si fa ricorso al costruttore della classe `NdefRecord`, la cui firma è:

```
NdefRecord(short tnf, byte[] type, byte[] id, byte[] payload)
```

Il primo parametro è il *type name format*, il secondo parametro contiene informazioni relative al tipo, il terzo parametro è un identificativo univoco solitamente non utilizzato ed il quarto parametro è il dato contenuto nel messaggio, sotto forma di array di byte. Il type name format consente di dare all'applicazione che interpreta il messaggio NDEF informazioni dettagliate relative al significato dei dati contenuti, come ad esempio il fatto che il tipo del record sia standard (uno tra quelli definiti da NFC forum) oppure personalizzato. Android fornisce due possibilità per creare un messaggio: partendo proprio da un array di record, oppure direttamente da un array di byte. Nel secondo caso è compito del sistema verificare che nell'header vi siano specificate tutte le informazioni necessarie alla tipizzazione del messaggio.
Le firme dei due costruttori sono le seguenti:

```
public android.nfc.NdefMessage (NdefRecord[] records)
```

```
public android.nfc.NdefMessage (byte[] data) Throws FormatException
```

6.1.3 Android Beam

In primo luogo è bene specificare che la piattaforma non fornisce un supporto completo al p2p, in quanto l'unica possibilità per comunicare via *SNEP* con un altro dispositivo è tramite *Android Beam*, un meccanismo attraverso il quale

due smartphone possono scambiare un singolo messaggio NDEF quando vengono avvicinati. L'utilizzo di Android Beam semplifica notevolmente il lavoro dello sviluppatore, svincolandolo completamente da alcuni compiti come ad esempio la gestione del canale di comunicazione *LLC*. Il grosso limite però sta nel fatto che Android Beam è una feature di sistema operativo, un protocollo p2p di alto livello costruito su SNEP che a sua volta utilizza LLCP, ma non è possibile per lo sviluppatore utilizzare direttamente questi due protocolli; l'unica alternativa sarebbe quella di riscriverne una propria implementazione, soluzione piuttosto complessa.

Per utilizzare a Android Beam, allo scopo di inviare un messaggio NDEF ad un altro dispositivo, occorre seguire le linee guida appresso riportate:

1. l'`Activity` che sta inviando i dati deve essere in primo piano; entrambi i dispositivi devono avere sbloccati;

2. i dati inviati devono essere incapsulati all'interno di un `NdefMessage`;

3. il dispositivo che riceve i dati deve supportare il protocollo di push NDEF "com.android.npp" oppure il Simple NDEF Exchange Protocol (SNEP) definito da NFC Forum. Il protocollo "com.android.npp" è richiesto per i dispositivi Android dalla versione 2.3 alla 3.2, mentre sono entrambi richiesti dalla versione 4.0;

4. quando i due dispositivi vengono avvicinati, il sistema operativo mostra l'interfaccia "Touch to Beam" e l'utente "mittente" deve toccare lo schermo del proprio smartphone per completare l'invio del messaggio.

Per l'abilitazione di Android Beam:

1. creare un `NdefMessage` contenente gli `NdefRecords` che devono essere inviati all'altro dispositivo;

2. invocare sull'adapter il metodo `setNdefPushMessage()` passandogli l'`NdefMessage` creato, oppure registrare una callback utilizzando `setNdefPushMessageCallback()`, passando la funzione da richiamare per la creazione del messaggio, non appena il dispositivo rileva un altro smartphone in prossimità.

Tipicamente la funzione `setNdefPushMessage()` non è invocata in corrispondenza di un particolare evento: si sta semplicemente dicendo all'adapter di preoccuparsi di inviare un certo messaggio NDEF non appena si collega ad un dispositivo in grado di comunicare via NFC, ma non è detto che ciò accada. Solitamente si utilizza esclusivamente quando il messaggio NDEF è pronto per essere messo "in attesa" e non varia nel tempo, mentre è bene utilizzare l'altra modalità (con la funzione di callback) se i dati per la creazione del messaggio NDEF sono disponibili solo al momento in cui il dispositivo destinatario è in prossimità.

6.1.4 La lettura di messaggi NDEF

Come anticipato in precedenza, sia per questioni di consumo energetico, che per questioni di sicurezza, Android consente l'utilizzo di NFC soltanto se lo schermo è acceso. Nel momento in cui il sistema riceve un messaggio NDEF piuttosto che un Tag, il comportamento desiderato è quello di delegarne la gestione alle applicazioni direttamente interessate, senza dover chiedere nulla all'utente a proposito di quale utilizzare. Il motivo è chiaro: siccome NFC funziona a cortissimo raggio, la richiesta di conferma forzerebbe l'utente a spostare lo smartphone, interrompendo così la connessione. All'interno dell'applicazione, occorre quindi specificare quali tipologie di informazioni NFC ogni `Activity` è in grado di gestire, facendo in modo che non venga chiesto all'utente quale attivare.
Il sistema Android utilizza il concetto di *Intent* per delegare l'esecuzione di determinate azioni alle applicazioni, la stessa cosa avviene per NFC.

Tag dispatch system

Lo smistamento degli `Intent` è gestito dal *tag dispatch system*, che svolge le seguenti azioni:

1. Cerca di effettuare il parsing del messaggio NDEF ed estrae il MIME type o l'URI che identifica il dato contenuto;
2. Incapsula il MIME type/l'URI e il *payload* all'interno di un `Intent`;
3. Cerca di eseguire l'`Activity` più appropriata, basandosi sull'`Intent`.

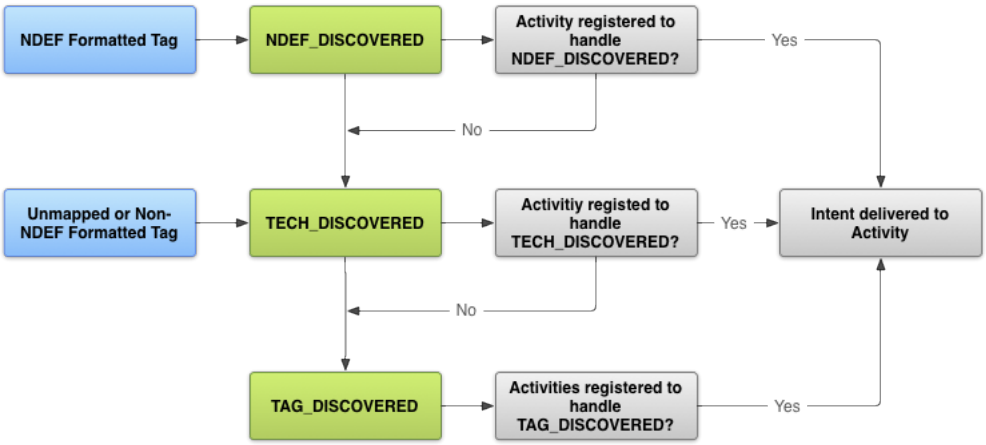

Figura 6.1: Il funzionamento del tag dispatch system

Il diagramma in Figura 6.1 esplicita il comportamento del tag dispatch system quando sono rilevati dei tag. Se al posto del tag, si riceve un messaggio direttamente da un dispositivo NFC, l'`Intent` generato dal sistema è `ACTION_NDEF_DISCOVERED`. Le applicazioni hanno due possibilità per potersi dichiarare interessate a determinati messaggi NDEF: un'opportuna configurazione del file *manifest* consente al Tag Dispatch System di attivare l'applicazione quando si trova in *background*, mentre può essere utilizzato un `PendingIntent`

sull'`Activity` per intercettare il dispatch di un messaggio NDEF se l'applicazione è attiva, prima che il controllo passi al tag dispatch system.

Intercettare messaggi NDEF in Background

Solitamente un'applicazione NFC che può ricevere un messaggio NDEF, filtra per l'`Intent ACTION_NDEF_DISCOVERED`. Se ad esempio si vuole fare in modo che intercetti un messaggio NDEF di tipo `URI` "http://developer.android.com/index.html", devono essere inserite nel *manifest* le seguenti informazioni di filtro:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="http"
              android:host="developer.android.com"
              android:pathPrefix="/index.html" />
</intent-filter>
```

Intercettare messaggi NDEF in Foreground

Per intercettare messaggi NDEF quando l'applicazione è in *foreground*, è necessario per prima cosa creare un `PendingIntent` e il relativo `IntentFilter`:

```
PendingIntent pendingIntent = PendingIntent.getActivity(
    this, 0, new Intent(this, getClass()).addFlags(Intent
        .FLAG_ACTIVITY_SINGLE_TOP), 0);

IntentFilter ndef =
    new IntentFilter(NfcAdapter.ACTION_NDEF_DISCOVERED);
try {
```

```
    ndef.addDataType("*/*");
}
catch (MalformedMimeTypeException e) {
    throw new RuntimeException("fail", e);
}
intentFiltersArray = new IntentFilter[] {ndef, };
```

All'interno dell'handler `onResume` si può poi richiamare sull'adapter il metodo `enableForegroundDispatch()`, passando `PendingIntent` ed `IntentFilter` come parametri. In questo modo l'`Activity` intercetta l'`Intent` prima del tag dispatch system, evitando così di lasciare la gestione del messaggio NDEF ad altre applicazioni che abbiano registrato il filtro nel *manifest*. L'unica accortezza è che occorre sempre richiamare all'interno di `onPause` il metodo per disabilitare la priorità sugli `Intent` in foreground
`mAdapter.disableForegroundDispatch()`, altrimenti Android solleva un'eccezione.

Lettura di un messaggio NDEF da un Intent NFC

Ogni messaggio NDEF che Android riceve è incapsulato all'interno di un `Intent`, di conseguenza è possibile estrarre il messaggio dati in corrispondenza di un `Intent` del tipo `ACTION_NDEF_DISCOVERED`. L'esempio di codice può essere sfruttato sia in corrispondenza di un `Intent` intercettato in background (e quindi all'interno del metodo `onResume`), sia quando l'applicazione lo riceve in *foreground* (evento `onNewIntent`).

```
NdefMessage msgs;
if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(
  getIntent().getAction())) {
```

```
        Parcelable[] rawMsgs =
            intent.getParcelableArrayExtra(
              NfcAdapter.EXTRA_NDEF_MESSAGES);
        if (rawMsgs != null) {
            msgs = new NdefMessage[rawMsgs.length];
            for (int i = 0; i < rawMsgs.length; i++) {
                msgs[i] = (NdefMessage) rawMsgs[i];
            }
        }
    }
```

6.2 Le API BluetoothLE

Dalla recente versione 4.3, Android offre la possibilità agli sviluppatori di interagire con dispositivi Bluetooth Low Energy attraverso il protocollo GATT (acronimo di Generic Attribute Profile). Le API inizialmente prevedevano per lo smartphone soltanto il ruolo di *central device*, mentre recentemente (versione 5.0) è stata aggiunta la possibilità di agire come *peripheral device*, anche se soltanto con la funzionalità limitata di *advertising*. A differenza di quanto avviene per NFC con Android Beam, la piattaforma non fornisce agli sviluppatori protocolli o meccanismi di alto livello per l'utilizzo di Bluetooth LE; se questo può inizialmente rallentare lo sviluppo per il livello di complessità leggermente più elevato, in realtà si trasforma in un vantaggio per la maggiore flessibilità garantita dal fatto che il sistema non "nasconde" nulla. Nel caso dell'applicazione sviluppata, le API sono utilizzate solo per l'interazione con un dispositivo periferico.

6.2.1 Utilizzare BluetoothLE

Come avviene per NFC, Android mette a disposizione un *adapter* dedicato alla comunicazione BLE: `BluetoothAdapter`. L'adapter è lo stesso che il sistema mette a disposizione per la comunicazione Bluetooth classica, ed è accessibile dagli sviluppatori:

```
public static BluetoothAdapter.getDefaultAdapter(Context c)
```

passando come parametro il contesto dell'`Activity`. Il prerequisito affinché l'operazione vada a buon fine è la specifica dell'utilizzo della tecnologia nel file *manifest*, attraverso le stringhe:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

Nel caso in cui l'utilizzo del bluetooth low energy sia essenziale ai fini applicativi, è possibile specificare nel file di *manifest* che l'app non sia resa visibile ai dispositivi non abilitati.

```
<uses-feature android:name="android.hardware.bluetooth_le"
    android:required="true"/>
```

6.2.2 La scansione

Prima di potersi connettere ad un dispositivo periferico è necessaria una scansione nell'ambiente circostante che lo rilevi tra tutti i device attivi e visibili (quindi tra quelli che stanno facendo *advertising*). In questo caso il sistema operativo non interviene con il meccanismo di `Intent`, lasciando agli sviluppatori il compito di gestire un eventuale servizio per la scansione in background. Le ultimissime API di Android Lollipop (5.0) hanno apportato un netto miglioramento per la gestione della scansione di dispositivi Bluetooth Low Energy [26], che è delegata alla classe `BluetoothLeScanner`. Attraverso il metodo `startScan` è possibile lanciare

una scansione specificando i filtri, che consentono di ricercare solo determinati device, i settings, utili per specificare ad esempio se la scansione deve essere fatta dando priorità al basso consumo o alla bassa latenza, e la callback richiamata qualora sia rilevato un dispositivo o si verifichi un errore.

```
BluetoothLeScanner scanner =
    getBluetoothAdapter().getBluetoothLeScanner();
ScanSettings settings =
    new ScanSettings.Builder()
        .setScanMode(ScanSettings
        .SCAN_MODE_LOW_POWER))
        .build();
List<ScanFilter> filters = new ArrayList<ScanFilter>();
scanner.startScan(filters, settings, myCallbackMethod);
```

Per le API di livello precedente, quindi fino ad Android 4.4, la scansione Low Energy era possibile solo attraverso il metodo `startLeScan()`, richiamabile sull'istanza del `BluetoothAdapter`, al quale occorreva passare la funzione di callback.

6.2.3 La connessione con un dispositivo

Per poter interagire con un dispositivo periferico, leggendo e scrivendo le varie caratteristiche del servizio GATT che il dispositivo espone, è necessario connettersi. La connessione ad un GATT server è gestita in modo asincrono da Android. Il metodo `connectGatt()` richiede la specifica della callback che il sistema deve richiamare quando lo stato della connessione cambia, e ritorna un'istanza della classe `BluetoothGatt`. All'interno della callback è possibile ricavare lo stato della connessione tra lo smartphone e il dispositivo, ed eseguire quindi le eventua-

li operazioni al cambio di stato. Tipicamente, quando si interagisce con un server GATT, la prima operazione che si esegue una volta completata la connessione è la *discovery* dei servizi offerti:

```
@Override
public void onConnectionStateChange(BluetoothGatt gatt,
    int status, int newState) {
    if (status == BluetoothGatt.GATT_SUCCESS &&
        newState == BluetoothProfile.STATE_CONNECTED) {
        gatt.discoverServices();
    } else if (status == BluetoothGatt.GATT_SUCCESS &&
        newState == BluetoothProfile.STATE_DISCONNECTED) {
        ...
    } else if (status != BluetoothGatt.GATT_SUCCESS) {
        ...
    }
}
```

La connessione sicura

La connessione tra due dispositivi LE non accoppiati non è sicura, perchè il canale di comunicazione utilizzato non è cifrato. Onde evitare di scambiare informazioni sensibili in chiaro, il protocollo consente a due dispositivi di scambiarsi una chiave di cifratura simmetrica che è poi utilizzata per cifrare il canale di comunicazione. Lo stato nel quale i dispositivi si trovano una volta già scambiate le chiavi di cifratura è detto *bonded*, mentre il processo di accoppiamento e scambio delle chiavi è il `pairing`. Android mette a disposizione un metodo attraverso il quale cominciare il processo di bonding BLE: `createBond()`. Il metodo può essere richiamato direttamente sull'istanza della classe `BluetoothDevice` corri-

spondente al dispositivo che si vuole "accoppiare", ed anch'esso, come il metodo per la connessione è asincrono. Per intercettare i diversi eventi durante il processo di *bonding*, occorre registrare un `BroadcastReceiver` che intercetti i diversi `Intent` sollevati da Android in corrispondenza dei cambiamenti di stato tra lo smartphone device che si sta accoppiando.

```
if (this.mDevice.getBondState() == BluetoothDevice.BOND_NONE) {
    boolean bondingStatus = this.mDevice.createBond();
    registerReceiver(this.mBondingBroadcastReceiver,
        new IntentFilter(BluetoothDevice.ACTION_BOND_STATE_CHANGED));
}

BroadcastReceiver mBondingBroadcastReceiver = new BroadcastReceiver() {

    @Override
    public void onReceive(final Context context, final Intent intent) {
        final BluetoothDevice device =
            intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        final int bondState = i
            ntent.getIntExtra(BluetoothDevice.EXTRA_BOND_STATE, -1);
        final int previousBondState =
            intent.getIntExtra(BluetoothDevice
                .EXTRA_PREVIOUS_BOND_STATE, -1);

        if (bondState == BluetoothDevice.BOND_BONDED) {
            ...
        } else if (bondState == BluetoothDevice.BOND_BONDING) {
            ...
```

```
        } else if (bondState == BluetoothDevice.BOND_NONE) {
            ...
        }
    }
}
```

6.2.4 Interazione con le caratteristiche GATT

L'obiettivo di una connessione con un GATT server è quello di poter interagire con le sue caratteristiche. Le caratteristiche possono essere di diverso tipo, a seconda del quale è possibile effettuare una lettura, una scrittura, oppure ricevere una notifica. L'interazione con le caratteristiche esposte dal dispositivo attraverso uno o più servizi GATT è subordinata a due principali operazioni:

- il dispositivo deve essere connesso;
- deve già essere stata richiamata la callback di successo in seguito alla discovery dei servizi `mBluetoothGatt.discoverServices();`

Quando entrambe le condizioni sono soddisfatte, è possibile interrogare il server GATT utilizzando il metodo `getServices()`, che ritorna una lista di `BluetoothGattService` che all'interno contengono a loro volta la lista di `BluetoothGattCharacteristic` implementate, ottenibile richiamando su ogni istanza di servizio il metodo `getCharacteristics()`. Questo flusso di operazioni è utile ovviamente solo se non si conoscono le caratteristiche esposte dal servizio; in caso si abbia già conoscenza della natura del servizio GATT e di quali caratteristiche utilizzare, le informazioni possono essere recuperate direttamente dall'istanza `BluetoothGatt` tramite il metodo `BluetoothGattService getService(UUID uuid)` e

`BluetoothGattCharacteristic getCharacteristic(UUID uuid)`. Una volta ottenuta l'istanza della caratteristica con cui si vuole interagire, è possibile richiamare una serie di metodi asincroni per la lettura o la scrittura del valore della caratteristica stessa, piuttosto che di uno tra i suoi descrittori.

Lettura del valore di una caratteristica

Questa operazione può essere effettuata direttamente sull'istanza della caratteristica restituita dal sistema Android momento della discovery del servizio GATT, tuttavia è consigliabile, al fine di ricevere valori aggiornati, utilizzare il metodo asincrono `readCharacteristic()` richiamabile sull'oggetto `BluetoothGatt`, la cui risposta si ottiene nella callback `onCharacteristicRead` :

```
@Override
public void onCharacteristicRead(BluetoothGatt GATT,
    BluetoothGattCharacteristic c, int status) {

    byte[] value = c.getValue();
}
```

Scrittura di una caratteristica

Similarmente a quanto avviene per la lettura, anche la scrittura di una caratteristica è un'operazione asincrona, che può essere effettuata modificando il valore dell'istanza (richiamando il metodo `setValue()`) per poi successivamente passarla interamente alla funzione `writeCharacteristic()`. La notifica di completamento dell'operazione si ha nel momento in cui il sistema Android scatena la callback `onCharacteristicWrite`, dove tra i parametri sono presenti la caratteristica scritta e l'esito dell'operazione:

```
@Override
public void onCharacteristicWrite(BluetoothGatt GATT,
    BluetoothGattCharacteristic characteristic, int status) {

    if(status == BluetoothGatt.GATT_SUCCESS) {
        ...
    }
}
```

Ricezione di una notifica

Il protocollo GATT consente al client di sottoscriversi alle notifiche quando determinate caratteristiche cambiano il proprio valore. La sottoscrizione è possibile grazie a due differenti operazioni: l'abilitazione delle notifiche locali, e l'abilitazione delle notifiche remote. L'abilitazione delle notifiche remote comunica al device che al variare della caratteristica deve notificare lo smartphone connesso, mentre l'abilitazione delle notifiche locali serve per dire al sistema operativo Android che deve risvegliare la callback `onCharacteristicChanged` nel momento in cui il device indica/notifica una variazione della caratteristica. Per l'abilitazione delle notifiche remote si modifica il valore di un descrittore "noto" presente in tutte caratteristiche che supportano questa modalità di operazione. Un esempio di codice che può essere utilizzato per l'abilitazione/disabilitazione delle notifiche per una caratteristica:

```
private void enableNotification(boolean enable,
    BluetoothGattCharacteristic notifyCharacteristic) {

    //Enable local notifications
    mConnectedGatt.setCharacteristicNotification(
```

```
        notifyCharacteristic, enable);

    //Enabled remote notifications
    BluetoothGattDescriptor descriptor = notifyCharacteristic
        .getDescriptor(CLIENT_CHARACTERISTIC_CONFIG);
    if(enable) {
        descriptor.setValue(BluetoothGattDescriptor
            .ENABLE_NOTIFICATION_VALUE);
    } else {
        descriptor.setValue(BluetoothGattDescriptor
            .ENABLE_NOTIFICATION_VALUE);
    }
    mConnectedGatt.writeDescriptor(descriptor);
}
```

6.3 L'applicazione sviluppata: doorkeeper

In prima istanza si cercherà di dare una visione generale dell'architettura applicativa, dopodiché saranno discussi i vari moduli utilizzati, entrando nello specifico con esempi implementativi in riferimento all'interazione con lo smart device e con il server.

L'applicazione *doorkeeper* ha l'obiettivo di dare agli utenti dell'organizzazione la possibilità di entrare in comunicazione con lo smart device collocato all'ingresso per poter registrare la timbratura ed aprire automaticamente la porta. Come già anticipato, le tecnologie utilizzate per la comunicazione devono essere quelle disponibili per la comunicazione in prossimità, quindi NFC e Bluetooth LE, le due interfacce di comunicazione sviluppate. Il processo di registrazione timbratura si

compone tuttavia di due fasi principali:

- autenticazione utente tramite chiamata HTTP verso il server, con l'obiettivo di ottenere un *authentication token*;
- invio del token ricevuto allo smart device utilizzando NFC o BLE ed attesa della conferma.

6.3.1 Architettura del codice

Lo sviluppo di applicazioni Android si basa sul concetto centrale di `Activity`. Un'`Activity` è un elemento che aiuta l'utente a compiere una determinata azione, che può essere la visualizzazione, la creazione o la modifica di dati. Quasi tutte le applicazioni Android dispongono di diverse `Activity`, che ovviamente possono interagire tra loro. In un'ottica di pattern model-view-controller, il ruolo più consono per un'`Activity` è quello di controller, poiché è l'oggetto che si occupa di gestire l'interazione tra l'utente e la view, oltre ad essere il responsabile per la renderizzazione dei dati di *model*.

L'applicazione sviluppata è suddivisa in diversi package, ed ognuno di essi contiene soltanto le classi Java ad esso legate. Una prima macrosuddivisione è quella che separa appunto le `Activity` dal resto delle classi. Data la netta differenza tra la prima azione (login server) rispetto all'interazione con lo smart device, si è scelto di sviluppare due differenti `Activity` principali:

- `LoginActivity` che l'utente utilizza per completare appunto il primo login;
- `LoggedInActivity` che è presentata solo al completamento dell'autenticazione e attraverso la quale è possibile interagire con lo smart device per completare l'accesso;

Entrambe le `Activity` in questione ereditano da `BaseActivity`, in modo da poter sfruttare funzionalità comuni (gestione errori, progress bar).
Oltre alle due Activity già menzionate, ne è stata creata anche una terza per mostrare all'utente uno *splash screen* con il logo nel momento in cui avvia l'applicazione, utile per completare alcune operazioni di caricamento in background. Tutte le `Activity` fanno parte del package `it.spot.io.doorkeeper.activities`. Tutti gli altri package sono anch'essi separati sulla base delle varie classi che contengono; differentemente da quanto avviene per il package delle `Activity`, la suddivisione non fa però riferimento alla natura dei componenti Android contenuti, ma piuttosto alle funzionalità che le classi interne forniscono.

- `it.spot.io.doorkeeper.auth` è il package di utilità che contiene le classi per effettuare l'autenticazione utente, sia utilizzando come provider Google, sia utilizzando il provider sul server;

- `it.spot.io.doorkeeper.http` contiene un helper implementato per facilitare la comunicazione HTTP e le relative entità;

- `it.spot.io.doorkeeper.model` ha al suo interno la definizione dell'unica entità di model utilizzata dall'applicazione, ovvero l'utente autenticato (classe `LoggedUser` e interfaccia `ILoggedUser`);

- `it.spot.io.doorkeeper.proximity.ble`, contiene l'helper implementato per la facilitazione della comunicazione bluetooth LE e le relative interfacce;

- `it.spot.io.doorkeeper.proximity.nfc`, come il package "fratello", contiene le entità definite per facilitare la comunicazione in prossimità, utilizzando questa volta la tecnologia di comunicazione NFC.

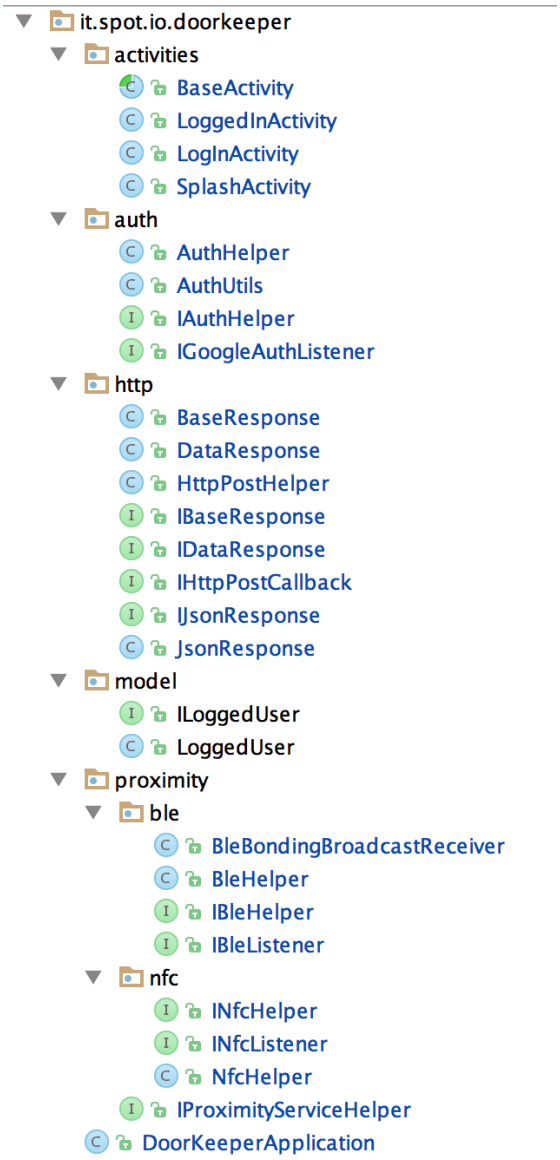

Figura 6.2: L'architettura dell'applicazione Android

6.3.2 La fase di login

La prima azione che l'utente può compiere utilizzando l'app consiste nel login verso il server. Al momento non è gestita anche la fase di registrazione nel sistema, poiché l'applicazione è vista come un accessorio che gli utenti posso-

no utilizzare una volta già inseriti nell'organizzazione. Il login è veicolato dalla `LoginActivity`, che mette a disposizione dell'utente sia il classico bottone con il quale autenticarsi utilizzando il proprio profilo Google, sia due campi di testo attraverso i quali inserire email e password del proprio account locale creato nel sistema.

Tutti i metodi richiamati dalla `LoginActivity` in corrispondenza degli eventi scatenati sull'interfaccia grafica sono gestiti internamente da un'istanza di `AuthHelper`, in modo da poter lasciare la `LoginActivity` più pulita e limitare il codice in essa contenuto a quello necessario per la gestione dell'interazione con l'utente. I metodi disponibili sono definiti nell'interfaccia `IAuthHelper`, e sono suddivisi tra quelli necessari per l'autenticazione Google e quelli per l'autenticazione locale.

- public void refreshLocalLogin(final String localToken,
  final IHttpPostCallback<IDataResponse<ILoggedUser>> callback);
- public void localLogin(final String email, final String password,
  final IHttpPostCallback<IDataResponse<String>> callback);
- public void getLocalUser(final String googleToken, final String email,
  final IHttpPostCallback<IDataResponse<ILoggedUser>> callback);
- public void setupGoogleAuthentication(final Activity activity, final IGoogleAuthListener listener, final int requestCode);
- public boolean googleLogin();
- public void googleLogout().

I primi tre metodi dichiarati in interfaccia incapsulano differenti chiamate HTTP verso il server, alle quali sono passati sia i parametri da serializzare in JSON ed in-

viare in POST, sia la callback da eseguire una volta ottenuta la risposta. Tutta l'infrastruttura asincrona HTTP e la relativa gestione ad `AsyncTask` è incapsulata da `HttpPostHelper`, che implementa anche tutti i meccanismi per la serializzazione e deserializzazione da oggetti JSON a Java e viceversa.

La prima operazione svolta dalla `LoginActivity` consiste nel richiamare il metodo `setupGoogleAuthentication`, al fine di impostare i parametri necessari per il completamento di un'eventuale richiesta di autenticazione Google. Quando l'utente che utilizza l'app si è precedentemente registrato nel sistema con le proprie credenziali, senza utilizzare quindi il provider Google, l'autenticazione avviene attraverso il provider "locale" presente sul server, verso il quale si effettua la chiamata HTTP POST utilizzando il metodo `localLogin`, ricevendo in risposta il token locale. Il metodo `refreshLocalLogin` è utilizzato in coda alla fase di login per "allungare" la vita del token, sostituito dal server con uno più recente e con scadenza più lontana. In questo modo il token può essere memorizzato nelle `SharedPreferences` dell'applicazione, che ad ogni avvio può quindi rinnovarlo ed utilizzarlo per autenticarsi, evitando quindi che sia l'utente a dover lanciare l'autenticazione manuale.

Autenticazione Google

L'autenticazione con il proprio profilo Google è un processo facilitato dalla classe `PlusClient`, utilizzata dall'`AuthHelper` per interagire con le API ed ottenere informazioni relative all'account che l'utente ha registrato sul proprio smartphone. Tutta la gestione del flusso di comunicazione tra sistema Android ed utente, per richiedere l'autorizzazione ed ottenere quindi il token di autenticazione necessario per il login sul server, è incapsulata nelle logiche interne all'oggetto `PlusClient`, il quale notifica il completamento della connessione ad `AuthHelper` attraverso i metodi nell'implementazione dell'interfaccia

`GooglePlayServicesClient.ConnectionCallbacks()`.

Una volta completata la fase di connessione verso i servizi Google è possibile richiedere al provider anche il token *oAuth* di autenticazione, utilizzando (internamente ad un task asincrono) il metodo `GoogleAuthUtil.getToken(...)`. Il prerequisito per la ricezione del token da Google consiste nell'aver registrato all'interno della *API console* i dati dell'applicazione Android, in modo simile a quanto fatto per la web application. Nel caso in non sia completata la registrazione o i dati inseriti non siano validi, la chiamata `getToken` ritornerà un errore. L'autenticazione Google è inizializzata dalla `LoginActivity` internamente all'evento di click del relativo pulsante, in cui si richiama il metodo `googleLogin()` sull'istanza di `AuthHelper`. Nel momento in cui l'applicazione riceve il token da Google, lo converte nell'utente "locale" richiamando la API `/auth/google/getLocalUser`. La notazione "locale" può essere fuorviante rispetto a quello che è in realtà il significato: non si tratta di un utente locale per l'applicazione, ma di un utente locale per il server e memorizzato nella base di dati del sistema. L'autenticazione Google serve quindi da "tramite" per ottenere dal server il token dell'utente "locale" associato, la cui ricezione comporta la fine della fase di autenticazione.

6.3.3 La comunicazione con lo smart device

Completata l'autenticazione l'applicazione presenta all'utente la *view* attraverso la quale è possibile interagire con lo smart device. La comunicazione per la registrazione della timbratura e l'apertura della porta, è gestita interamente dalla `LoggedInActivity`, e si compone sostanzialmente di due differenti azioni:

- l'applicazione deve richiedere allo smart device una sorta di firma digitale, per essere sicura della sua autenticità;

- una volta validato il dispositivo, deve inviargli il proprio token ricevuto dal server e attendere la conferma.

È possibile svolgere l'intero flusso di comunicazione sia con Bluetooth LE, sia con NFC. Il package in cui sono definite le classi *helper* per la gestione delle due tecnologie di comunicazione, è `it.spot.io.doorkeeper.proximity`. Ognuno dei due diversi helper, richiede l'utilizzo prioritario dell'`Adapter` dedicato, su cui solitamente si richiamano differenti operazioni dipendentemente dallo stato in cui l'`Activity` si trova. Per questo motivo, entrambi implementano l'interfaccia `IProximityServiceHelper`:

- public boolean adapterIsOff();
- public boolean isActive();
- public void stop();
- public void pause();
- public void resume();

che oltre al metodo `adapterIsOff()`, utilizzato per verificare se l'utente ha abilitato la tecnologia nel proprio smartphone, definisce anche tutti i metodi richiamabili dall'`Activity` nei principali eventi del proprio ciclo di vita. Il metodo `pause` ad esempio deve implementare internamente tutte le operazioni che è necessario fare sull'adapter prima che l'Activity vada in background.

La comunicazione via Near Field Communication

Le funzionalità per l'utilizzo di NFC sono incapsulate nella classe `NfcHelper`, che `LoggedInActivity` usa per interagire con lo smart device. Per la gestione *intent-based* che Android riserva ai messaggi NFC ricevuti sul dispositivo, parte

del codice deve comunque risedere sull'Activity principale. La classe, oltre alla già citata interfaccia `IProximityServiceHelper`, implementa anche *INfcHelper*, in cui sono dichiarati i seguenti metodi:

- public String readSignature(final Intent ndefIntent);
- public void writeToken(final String token, final boolean isEntrance);
- public String readAuthenticationResult(final Intent ndefIntent);
- public boolean isP2PStarted();
- public boolean isP2PDisabled().

Il metodo `readSignature` è utilizzato per estrarre da un intent contenente un messaggio NDEF (ed intercettato dalla `LoggedInActivity`) la "firma digitale" dello smart device. È il primo metodo richiamato, grazie al quale `NfcHelper` entra nello stato "P2PStarted". Il secondo metodo `writeToken` è invocato dall'`Activity` una volta completata la verifica della firma digitale, al fine di inviare un messaggio NDEF contenente il token di autenticazione al dispositivo. In questo caso l'implementazione fa uso del metodo `setNdefPushMessage` sull'adapter, al quale è passato un `NdefMessage` il cui primo ed unico `NdefRecord` ha come *payload* il solo token, un semplice record testuale *well-known* di tipo `RTD_Text`. L'ultimo metodo di comunicazione è `readAuthenticationResult`, richiamato dalla `LoggedInActivity` per estrarre il risultato dell'operazione dal messaggio NDEF inviato dallo smart device.

Tutto il flusso di operazioni è svolto in una singola interazione tra lo smartphone e lo smart device: l'utente non deve avvicinare ed allontanare più volte il proprio dispositivo al lettore NFC. Il "trucchetto" in questo caso risiede in un'operazione di "chiusura e riapertura" forzata del canale LLC da parte dello smart

device dopo l'invio del primo messaggio, simulando così un nuovo "avvicinamento" tra i due dispositivi. In questo modo lo smartphone può permettersi di inviare il messaggio NFC solo dopo aver ricevuto e validato il primo, operazione altrimenti impossibile senza aver allontanato il dispositivo dal lettore. Questo comportamento viola leggermente l'esperienza d'uso prevista da Android, che propone NFC come una tecnologia orientata allo scambio di un singolo messaggio per ogni interazione.

Tutti i metodi di read dichiarati nell'interfaccia *INfcHelper* hanno una natura sincrona: dato che è il sistema a gestire la ricezione dei messaggi via NFC, in realtà l'Helper è usato soltanto per estrarre le informazioni da quelli già ricevuti (e incapsulati dentro ad `Intent`). L'unico metodo asincrono è quello per l'invio del token, per il quale si effettua una richiesta di push ad `NfcAdapter`. Nel momento in cui l'invio sarà completato dal sistema operativo, `NfcHelper`, che implementa l'interfaccia `OnNdefPushCompleteCallback`, riceverà notifica e la propagherà a sua volta alla `LoggedInActivity` utilizzando il metodo di interfaccia `INfcListener` `onSendTokenCompleted`.

In Figura 6.3 è rappresentato il flusso di operazioni dal momento in cui l'utente avvicina il proprio smartphone al lettore, al momento in cui il device completa la verifica del token. È stato enfatizzato il ruolo dell'app attraverso la suddivisione nelle entità Activity, Helper e sistema operativo, mentre server e smart device compaiono solo come entità singole. Le operazioni non sono numerose, tuttavia il flusso risulta abbastanza intricato per via della gestione della comunicazione peer to peer di Android.

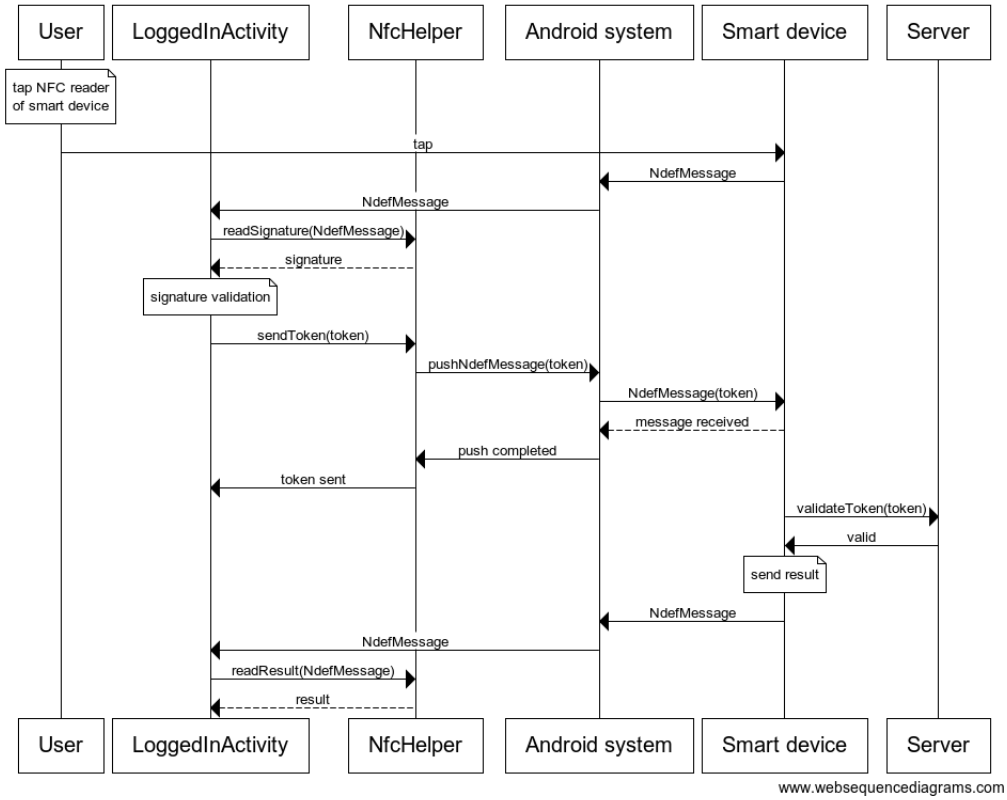

Figura 6.3: Il flusso operativo tra le diverse entità nella comunicazione nfc

La comunicazione via BluetoothLowEnergy

Sebbene la modalità e i paradigmi con cui avviene la comunicazione Bluetoooth LE con il dispositivo si discostino decisamente da quelle NFC, le azioni in carico al `BleHelper` sono le stesse. La prima sostanziale differenza è rappresentata dalla modalità con cui avviene l'interazione: se lo scenario della timbratura può essere considerato ideale per NFC, occorre valutare con attenzione come utilizzare BluetoothLE. Una possibilità prevista dai casi d'uso tipici di bluetooth low energy è la scansione passiva del dispositivo centrale (smartphone) per ricercare dispositivi periferici emettitori di segnale. Con il caso d'uso proposto da questo scenario, sarebbe stato possibile progettare l'applicazione in modo che facesse partire tutto il flusso di comunicazione automaticamente, senza necessità di alcuna interazione con l'utente, ma semplicemente rilevando la vicinanza dello smart-device. Tuttavia, si sono considerati i seguenti aspetti:

- l'applicazione avrebbe dovuto disporre di un servizio in background per la ricerca continua di dispositiviLE, con un impatto non indifferente sul consumo di batteria;
- sarebbe stato complicato "tarare" in modo efficace il funzionamento, poiché l'unico parametro a disposizione per dedurre la distanza (RSSI) varia da dispositivo a dispositivo;
- la porta dell'ufficio potrebbe trovarsi in un luogo di passaggio, di conseguenza si verificherebbero con buona probabilità alcuni casi di timbrature non volute;

Di conseguenza si è scelto di lasciare all'utente il compito di avviare la comunicazione per la timbratura, utilizzando l'unico pulsante presente nella view principale della `LoggedInActivity`. Il pulsante è abilitato solo nel momento in cui il sistema rileva, attraverso una scansione, la vicinanza del dispositivo.

Oltre all'interfaccia `IProximityService`, `BleHelper` implementa anche l'interfaccia `IBleHelper`, dove sono dichiarati i seguenti metodi:

- public void readSignature();
- public void writeToken(String token, boolean isEntrance);

Tutti i metodi in questo caso non hanno valore di ritorno, poiché il sistema Android gestisce l'interazione con un `GattService` in modo asincrono. Per questo motivo, la `LoggedInActivity` implementa l'interfaccia `IBleListener` che è utilizzata per notificare l'`Activity` in corrispondenza di eventi o al momento in cui sono completate determinate azioni da essa richieste all'Helper. I metodi implementati dal listener sono i seguenti:

- public void onBLEDeviceReady();
- public void onReadSignatureCompleted(byte[] result);
- public void onWriteTokenCompleted(int result);

Le caratteristiche implementate nel servizio GATT sono tutte "sicure", di conseguenza sono accessibili soltanto se lo smartphone è *bonded* con il dispositivo. Quando l'helper rileva la presenza dello smart device, si preoccupa di cominciare in modo autonomo e silente il processo di bonding con il dispositivo utilizzando il metodo `BluetoothDevice.createBond`, dopodiché notifica l'`Activity` richiamando `onBLEDeviceReady`, la quale reagirà abilitando il pulsante per permettere all'utente di cominciare la comunicazione. Il processo di accoppiamento avviene soltanto la prima volta, dopodiché il dispositivo ed il device memorizzano le informazioni internamente per poter ricreare il canale sicuro ad ogni connessione successiva. Tutti gli stati e i meccanismi necessari per interagire via BLE con il device sono mascherati alla LoginActivity, poiché è l'helper che nel momento in cui l'utente preme il pulsante si occupa (in sequenza):

- di instaurare la connessione al GATT server presente all'interno del dispositivo attraverso il metodo di istanza del device `BluetoothDevice.connectGatt`;
- di effettuare la *discovery* dei servizi GATT, tramite `GATT.discoverServices`;
- soltanto alla fine, di leggere la caratteristica contenente la firma digitale readCharacteristic.

Tutte queste operazioni sono asincrone, il flusso è gestito internamente da `BleHelper` che registra tutte le opportune callback di notifica. Soltanto nel momento in cui avviene la lettura della *signature*, l'helper notifica la `LoginActivity` tramite chiamata del metodo `onReadSignatureCompleted`, passando come parametro la firma. Ottenuta e validata la firma, l'applicazione procede immediatamente con la fase di scrittura del token.

Prima di effettuare la scrittura del token, l'applicazione abilita le notifiche BLE (sia sul device che in locale) sulla caratteristica utilizzata dallo smart device per notificare appunto il completamento del flusso di autenticazione. Ogni scrittura di caratteristica BLE è limitata a 20 byte, di conseguenza è stato necessario spezzare il token in *chunk* di 20 byte ciascuno, scritti in modo sequenziale su una stessa caratteristica implementata dallo smart device. Un'ulteriore caratteristica è stata sfruttata per scrivere l'ultimo chunk, notificando di fatto lo smart device del completamento delle operazioni di scrittura. È compito del device ricostruire il token appendendo, dopo ogni singola scrittura, i diversi *chunk* ad un unico buffer. Tenuto in considerazione il fatto che il device dovrà comunicare con il server per l'autenticazione del token inviatogli dall'utente, è possibile che si verifichi un piccolo ritardo tra il completamento dell'operazione di scrittura da parte dell'app, e la ricezione dell'esito come notifica. Una volta ricevuta, l'applicazione disabilita le notifiche e lo smartphone è disconnesso dallo smart device. In Figura 6.4 è rappresentato tutto il flusso di operazioni durante l'interazione tra

l'applicazione android e lo smart device, enfatizzando il ruolo dell'applicazione per smartphone attraverso la suddivisione nelle entità Activity, Helper e sistema operativo. Nonostante il flusso sia composto da più operazioni rispetto a quello per la comunicazione NFC, risulta più lineare e meglio comprensibile perchè il sistema operativo entra in gioco soltanto per gestire le comunicazioni di "basso" livello.

Siccome le caratteristiche del servizio GATT sono note all'applicazione, sono memorizzati nell'helper gli identificativi di ogni caratteristica e l'accesso a ciascuna di esse avviene direttamente utilizzando il metodo `BluetoothGatt.getService(ServiceUUID).getCharacteristic(CharUUID)`, senza necessità di ricercarle in modo iterativo interrogando il servizio.

A livello di interfaccia grafica, la `LoggedInActivity` gestisce i tempi di attesa nel flusso di comunicazione BluetoothLE con una `ProgressDialog`, mentre utilizza una `AlertDialog` per mostrare all'utente eventuali errori o messaggi. L'esito finale è comunicato con semplici `Toast` che scompaiono dopo pochi secondi.

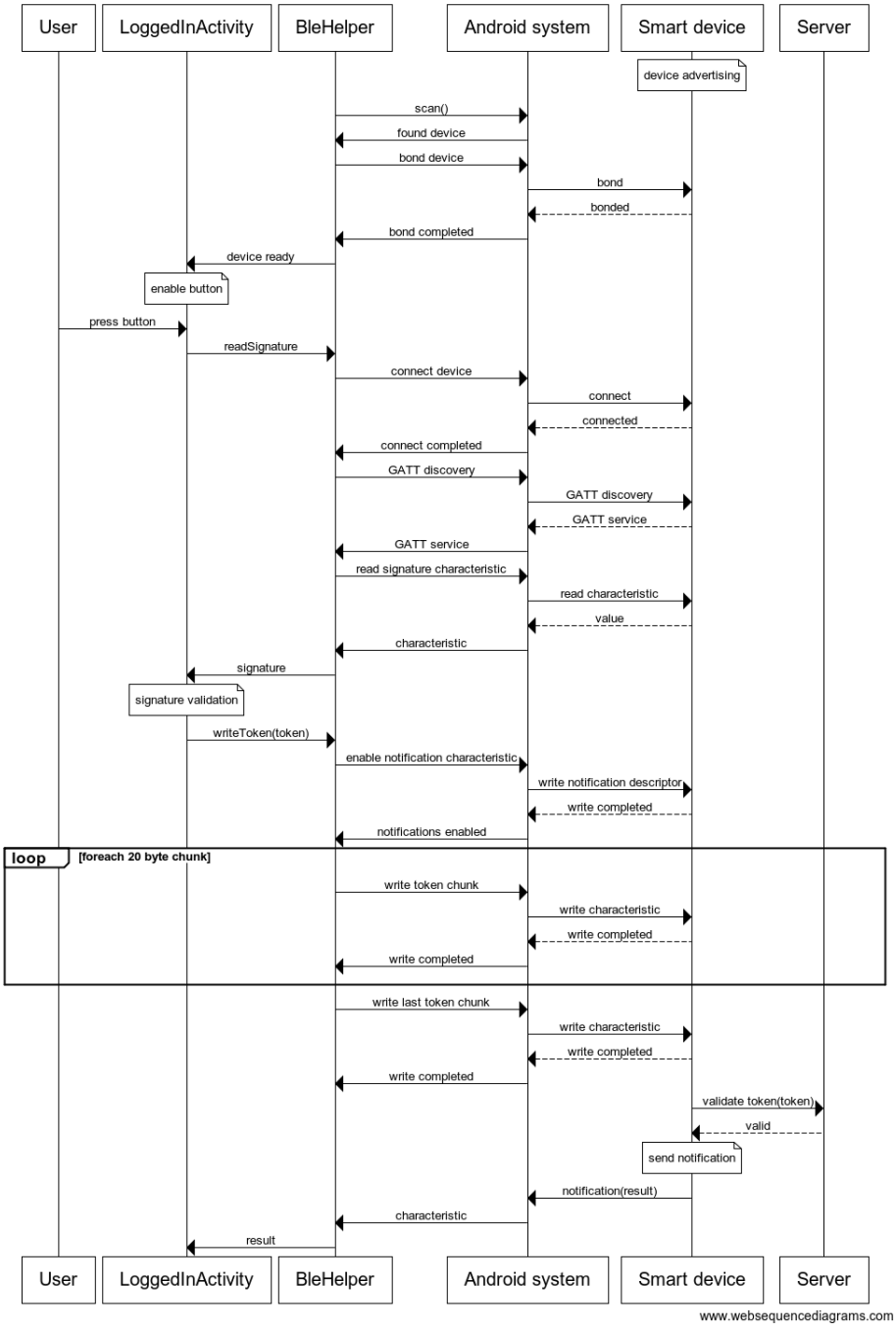

Figura 6.4: Il flusso operativo tra le diverse entità nella comunicazione bluetooth low energy

Capitolo 7

Conclusioni

Anche se la piattaforma sviluppata presenta ad oggi i limiti dovuti alla sua natura prototipale, complessivamente i principali requisiti di progetto sono stati soddisfatti. I tre nodi applicativi comunicano correttamente in tutti gli scenari previsti in fase di analisi, ognuno di essi attraverso l'uso di differenti tecnologie e modelli di comunicazione.
Il sistema realizzato è dedicato alle organizzazioni aziendali che necessitano di uno strumento economico, flessibile e configurabile per il controllo accessi e gestione presenze. Grazie ad esso, gli utenti hanno la possibilità di usare il proprio smartphone come mezzo personale, univoco e sicuro attraverso il quale accedere al luogo di lavoro e, allo stesso tempo, inserire i dati di accesso/uscita in una sola piattaforma web facilmente consultabile per gli amministratori.

Tutto il codice del progetto è rilasciato come open source, ed è pubblicato nel repository GitHub di SPOT Software[1]. La scelta di questa filosofia trova principalmente motivazione nella prospettiva di crescita del progetto stesso, che potrebbe certamente suscitare interesse da parte della community ed ottenerne benefici da un punto di vista della diffusione e della qualità. Inoltre, il contesto di tesi

[1] https://github.com/spotsoftware/io-at-spot

di laurea all'interno del quale il sistema è nato ed implementato (nella sua prima versione di prototipo), lo rende particolarmente adatto come spunto per lavori successivi volti ad estendere il progetto o alcune delle sue parti più significative.

Lo sviluppo ha rappresentato un'opportunità per l'approfondimento di tematiche molto interessanti per le ricadute che possono innescare in ambiti industriali. In ognuno dei tre nodi sono entrati in gioco framework e tecnologie diverse, dando così modo di comprenderne meglio potenzialità e criticità attraverso un loro concreto utilizzo. Per quel che riguarda il nodo "cloud" del sistema, se durante la progettazione del server sono stati presi in considerazione gli aspetti principali legati ad applicativi REST-based con basi di dati non relazionali (in grande fervore nel mondo dei Big Data), per la realizzazione della parte client il principale scoglio è risultato essere l'insieme di problematiche introdotte dai requisiti di responsività ed usabilità, vincoli alla base di ogni moderna web application.
Diverso è stato il processo di sviluppo dello smart device che ha richiesto un insieme eterogeneo di competenze trasversali: dall'elettronica di base all'uso di sistemi operativi linux-based, dall'interazione con input output digitali allo sfruttamento di differenti stack applicativi per la comunicazione wireless con Bluetooth LE ed NFC. Il tutto incapsulato in un'unica applicazione JavaScript su piattaforma node.js, che oltre ad essere fortemente scalabile (quindi particolarmente adatta a sistemi embedded) grazie agli scarsi requisiti di elaborazione richiesti e alla sua natura asincrona, garantisce migliore compatibilità con il server, poiché implementato attraverso la medesima tecnologia.
Sebbene debbano essere completati gli sviluppi dell'applicazione per smartphone anche sugli altri principali sistemi operativi, la scelta iniziale di implementazione su Google Android, motivata soprattutto dalla disponibilità di un framework Java solido e completo dal punto di vista dei servizi di connettività, ha fatto sì che potessero essere testati entrambi gli scenari applicativi previsti per la timbratura

e l'accesso, ovvero comunicazione con lo smart device via NFC e via Bluetooth LE.

A posteriori si può constatare che nonostante il processo produttivo si sia svolto in maniera piuttosto coerente ai principi dello sviluppo agile, il mancato utilizzo sin dall'inizio di un approccio *test-driven* ha sicuramente penalizzato il progetto dal punto di vista della consistenza delle diverse componenti durante lo sviluppo delle nuove funzionalità. Vista la natura molto eterogenea del sistema e le difficoltà riscontrate sia durante l'implementazione funzionale di alcuni dei singoli nodi, sia durante le fasi che hanno visto come protagonista l'interazione tra di essi, è stata data priorità alla verifica dell'effettiva fattibilità dell'intera architettura.
Si farà tesoro di questa esperienza per migliorare il processo di produzione del codice attraverso la metodologia di sviluppo orientata all'utilizzo di test unitari, funzionali e di integrazione che, benché comportino un considerevole investimento iniziale, garantiscono un livello di coerenza e qualità altrimenti difficilmente raggiungibile.

Il progetto nella sua complessità ha messo in evidenza un insieme di fattori difficilmente riscontrabili durante lo sviluppo di una singola applicazione tradizionale non distribuita, fattori che però risultano spesso presenti quando si progettano i moderni sistemi complessi: scalabilità, affidabilità, gestione degli errori di comunicazione e compatibilità tra piattaforme profondamente differenti, sono temi di forte attualità nel mondo di Internet of Things e la loro conoscenza, seppur approfondita nei limiti di tempo e di scenari adatti ad un progetto di questa natura, risulterà un importante vantaggio nell'analisi e nella gestione di problematiche emergenti in questo nuovo paradigma applicativo.

7.1 Sviluppi futuri

Gli sviluppi futuri per questo lavoro sono molteplici: il primo passo che deve essere necessariamente compiuto prima dell'attivazione del sistema consiste nel consolidamento delle sue funzionalità, che andranno racchiuse nei differenti livelli previsti dai test unitari, funzionali e di integrazione.
Secondariamente si completerà l'implementazione di alcuni meccanismi e algoritmi ausiliari previsti nel sistema durante l'accesso utente, in particolare per consentire al dispositivo di verificare la genuinità dei tag NFC utilizzati, e all'app Android di autenticare il dispositivo stesso prima di inviare il token dell'utente. Entrambe le funzionalità sono previste, ma al momento sono presenti solo gli scheletri implementativi.
In terza istanza saranno sviluppate le applicazioni mobile anche per le piattaforme iOs e Windows Phone, al fine di poter dare a tutti gli utenti la possibilità di utilizzare il proprio smartphone senza dover ricorrere all'uso di un tag NFC.

Completate queste attività, il sistema sarà installato e sottoposto a test intensivo esclusivamente in azienda; una prima fase vedrà l'applicazione web pubblicata internamente sulla intranet aziendale e lo smart device per il controllo degli accessi affiancato a quello attualmente funzionante. In questo modo gli utenti potranno gradualmente utilizzare la nuova piattaforma, avendo comunque la possibilità di accedere attraverso il sistema *legacy* in caso di blocchi o comportamenti errati. Terminato il periodo di test e consolidamento del sistema, avverrà la pubblicazione online dell'applicazione web, presumibilmente utilizzando servizi di hosting esterni in grado di garantire maggiore scalabilità e flessibilità rispetto alle soluzioni interne.
In previsione di un'eventuale diffusione del sistema, sarebbe bene verificare la scalabilità del server utilizzando una piattaforma dedicata al test di carico, in grado di "stressare" la base di dati mongoDB attraverso l'esecuzione delle query più

critiche e frequenti. Le scelte progettuali effettuate per gestire le relazioni tra le entità, durante la modellazione e progettazione dei diversi schemi, potrebbero essere riviste in maniera piuttosto radicale qualora dovessero presentarsi situazioni critiche e "colli di bottiglia" non diversamente gestibili.

Anche se la natura open source del progetto consentirebbe alle aziende interessate di acquistare separatamente i componenti ed implementare una propria versione dello smart device, la necessità di dover costruire ed assemblare il sistema embedded complessivo potrebbe risultare fortemente limitante per la diffusione e l'utilizzo del sistema stesso. Per questo motivo saranno valutate diverse ipotesi al fine di realizzare un dispositivo elettronico unico, che integri in una sola scheda di espansione per Raspberry pi, tutti i componenti necessari per completare le sue funzionalità: un modulo per la comunicazione Bluetooth LE, un lettore NFC ed un circuito di attuazione per l'apertura della porta.

Elenco delle figure

Bibliografia

[1] Android developers. http://developer.android.com/.

[2] Angular.js. https://docs.angularjs.org/guide/introduction.

[3] Kevin J. Ashton. The internet of things. http://kevinjashton.com/2009/06/22/the-internet-of-things/, June 2009.

[4] bleno. https://www.npmjs.org/package/bleno.

[5] Skyrocketing demand for bluetooth appcessories for latest phones. http://www.bluetooth.com/Pages/Mobile-Telephony-Market.aspx, January 2014.

[6] Bluez, official linux bluetooth protocol stack. http://www.bluez.org/.

[7] BMIMatters.com. Understanding facebook business model. http://bmimatters.com/2012/04/10/understanding-facebook-business-model/, April 2012.

[8] Brewer. Towards robust distributed systems. https://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, July 2000.

[9] International Data Corporation. The internet of things, data from embedded systems. http://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm, April 2014.

[10] Db-engines ranking. http://db-engines.com/en/ranking.

[11] Robin Heydon. *Bluetooth Low Energy - The Developer?s Handbook*. Prentice Hall, 2012.

[12] Ieee 802.11n-2009 - amendment 5: Enhancements for higher throughput, 2009.

[13] Software engineering - product quality - part 1: Quality model – iso/iec 9126-1:2001, 2001.

[14] Ivar Jacobson Magnus Christerson Patrik Jonsson and Gunnar Övergaard. Object-oriented software engineering - a use case driven approach. 1992.

[15] Libnfc main page. http://nfc-tools.org/.

[16] Dario Maio. Smart city e tecnologie mobili. http://smartcity.csr.unibo.it/smart-city-e-tecnologie-mobili/, June 2014.

[17] Mongodb. http://www.mongodb.org/.

[18] The pros and cons of mongodb. http://halls-of-valhalla.org/beta/articles/the-pros-and-cons-of-mongodb,45/, August 2014.

[19] Nfc forum. http://nfc-forum.org/.

[20] Python module for near field communication. https://nfcpy.readthedocs.org/en/latest/.

[21] About node.js. http://nodejs.org/.

[22] Elvis Pfutzenreuter. Bluetooth: Att and gatt. https://epx.com.br/artigos/bluetooth_gatt.php.

[23] Standard for efficient cryptography - version 1.0. http://www.cryptrec.go.jp/cryptrec_03_spec_cypherlist_files/PDF/01_01sec1.pdf, September 2000.

[24] Bruno Terkaly. Running .net applications on a raspberry pi that communicates with azure. http://blogs.msdn.com/b/brunoterkaly/archive/2014/06/11/mono-how-to-install-on-a-raspberry-pi.aspx, June 2014.

[25] Using the uart. http://www.raspberry-projects.com/pi/programming-in-c/uart-serial-port/using-the-uart/.

[26] David G. Young. Fast background detection with android 5.0. http://developer.radiusnetworks.com/2014/10/28/android-5.0-scanning.html, October 2014.

[27] Zerorpc. http://zerorpc.dotcloud.com/.

Ringraziamenti

Vorrei ringraziare prima di tutti il professor Dario Maio, che con il corso di Smart City è riuscito a trasmettermi passione ed interesse verso molti dei temi alla base del mio lavoro, seguendomi poi con preziosi consigli durante lo sviluppo della tesi.

Un sincero grazie anche ai titolari di SPOT Software Pietro Evangelisti e Maurizio Pensato, che hanno reso possibile lo sviluppo di questo progetto, e a tutti i colleghi che in sette anni di esperienza mi hanno formato da un punto di vista professionale e umano. Grazie ad Andrea ed Elisa... è anche per il vostro aiuto che oggi sono qua.

Grazie a Giulia e ai miei genitori, i due solidi punti di riferimento che ogni giorno mi supportano e "sopportano", spronandomi nel dare sempre il massimo.

Grazie ai miei amici, presenti durante tutti questi anni nei momenti belli e nei momenti brutti.

Grazie ai compagni di corso e di facoltà, con i quali ho condiviso intensamente questo percorso pieno di soddisfazioni e difficoltà.

Grazie a tutte le persone che mi vogliono bene e hanno sempre creduto in me, anche quelle che non ci sono più.