POLITECNICO
DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Autoconfigurazione di un veicolo in ambito Car Sharing

Relatore
Fulvio Corno

Candidato
Antonio Risoli

Tutore Concept Reply
Stefania Basciu

Ottobre 2017

A Nonno Totò e Nonno Peppe

Ringraziamenti

Ringrazio i miei relatori, il Prof. Fulvio Corno e la Manager Stefania Basciu, per il loro supporto e la loro disponibilità nei miei confronti durante il percorso di tesi.

Desidero ringraziare la Concept Reply nella persona di Paolo Carletto che mi ha concesso di svolgere il mio percorso di tesi in questa azienda. Altresì si ringraziano i dipendenti della Concept per le infinite partite a biliardino durante la pausa pranzo e il supporto datomi quando ne ho avuto bisogno.

Come potrei dimenticarmi di tutti i fine settimana di questi ultimi 5 anni, un grazie agli amici del gruppo WhatsApp "Gli amici di Pulcinella" per le pizze e le risate (e il dialetto mai perduto).

Il ringraziamento più grande però va alla mia Famiglia, Mamma e Papà, ai sacrifici dei miei genitori che mi hanno permesso di arrivare fin qui, a mia sorella Martina. I sacrifici dei miei genitori però non sarebbero potuti avvenire senza che i miei nonni e nonne a loro volta non avessero fatto sacrifici per i miei genitori. Per questo e per tutto il sostegno che mi hanno dato da sempre un grazie va alle mie nonne Elvira e Elvira. Ed un pensiero a Nonno Peppe e Nonno Totò che invece mi hanno protetto da lì sù e a cui dedico questa tesi.

Un grazie va anche a tutti i miei zii ma in particolare a zio Franco, zia Imma, zia Marzia e zio Peppe che mi hanno sempre sostenuto e aiutato specialmente in questi anni.

Un grazie agli amici di sempre di giù ed, inoltre, in ordine di come li ho conosciuti a Nello, Antonello e Antonio e ai compagni di studi Simone, Federico, Alessio e Stefano.

Indice

Elenco delle figure

Introduzione

Contesto di riferimento

Car Sharing

Il *Car Sharing* è un servizio commerciale erogato da aziende, il più delle volte private, nell'ambito della mobilità urbana e sostenibile che permette agli utenti, previa prenotazione, di noleggiare un veicolo per un periodo di tempo breve. L'idea di tale servizio è nata in seno ad associazioni ambientaliste all'inizio degli anni Novanta in Svizzera. Si utilizza l'autovettura come mezzo di trasporto collettivo ad uso individuale. Esso nasce con l'obiettivo di sopperire ad esigenze di mobilità per spostamenti di tipo sporadico o occasionale. Se da un lato è un servizio di trasporto pubblico, dall'altro offre agli utenti i comfort del trasporto privato tenendo però presente che da un punto di vista economico si distribuiscono i costi di acquisto, manutenzione, tasse, assicurazioni su una pluralità di utenti con ovvi vantaggi per il singolo e per la collettività[4, 21].

Tra questi sicuramente vanno evidenziati sia l'economicità del servizio per l'utilizzo occasionale del veicolo che se paragonato ai costi per la gestione di un veicolo privato per gli stessi spostamenti porta ad un risparmio in media del 40% per utenti che percorrono meno di 5000 Km all'anno [16], sia la versatilità del servizio offerto praticamente 24 ore al giorno con la possibilità di scegliere il veicolo più adatto per la situazione. I veicoli inoltre subiscono una manutenzione costante con conseguenti benefici sulla sicurezza degli stessi. Come precedentemente accennato il *Car Sharing* è anche associato alla sostenibilità ambientale (non a caso è stato ideato da un'associazione ambientalista) infatti si è stimato che una vettura che copre 20000 km in un anno sostituisce almeno 8 auto private che insieme avrebbero percorso un totale di 27000 Km nello stesso arco temporale con un risparmio di circa il 30% delle emissioni inquinanti[16, 21].

IoT & Automotive

L'acronimo IoT, abbreviazione dell'inglese *Internet Of Things* ovvero l'Internet delle cose è stato usato per la prima volta da Kevin Ashton, ricercatore al MIT di Boston, nel 1999 per indicare un insieme di dispositivi elettronici (spesso collegati a sensori e/o attuatori) capaci di comunicare tra di loro e scambiarsi informazioni in modo automatico. Stime recenti

prevedono per il 2020 dai 50 agli 80 miliardi di dispositivi connessi. Alla pervasività di tutti questi dispositivi si affianca e si affiancherà lo sviluppo del software con gli obiettivi di aiutare le persone, prevedere delle situazioni e in base alle circostanze effettuare delle operazioni. Si vanno dunque a delineare vari scenari ciascuno specializzato su un particolare campo di applicazione (casa, città, energia, ecc..) che portano alla creazione di sub-filoni dell'IoT chiamati Smart Home, Smart City, Smart Grid e così via. Tali scenari sono possibili grazie ad una tendenza verso lo sviluppo e l'utilizzo di software open source e alla facilità di accesso ai driver dei dispositivi hardware (sensori, attuatori).

L'industria automobilistica per motivi di sicurezza e di competizione tra le aziende ha avuto inizialmente delle difficoltà ad integrare l'IoT all'interno dei veicoli per via della segretezza del software, dei driver e relativo hardware utilizzati. Ma la portata innovativa dell'IoT ha fatto si che gli OEM si iniziassero ad approcciare alla possibilità di installare nuovo software, anche aggiornabile via OTA al fine di aggiungere nuove funzionalità ai veicoli.

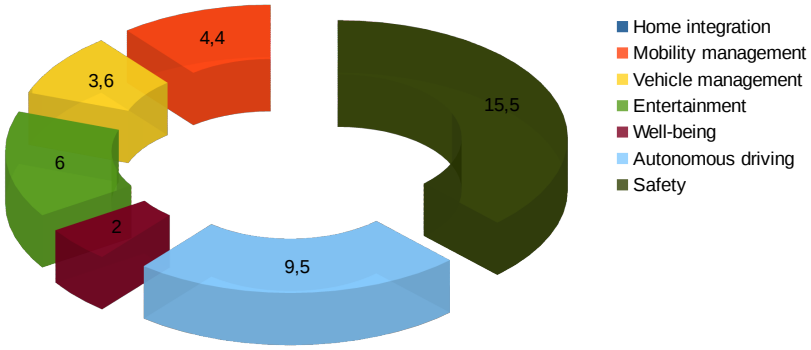
Infatti secondo *Telefonica* il valore industriale della tecnologia in ambito automotive dovrebbe crescere in modo impetuoso, dall'attuale 10% fino al 90% nel 2020. Così come Mary Barra, CEO in General Motors ha affermato che l'industria automobilistica cambierà nei prossimi 5-10 anni di più di quanto sia cambiata negli ultimi 50 anni.

Nella figura 1 sono mostrate le previsioni effettuate dal Professore Stefan Bratzel per la Strategy& nel 2015 [27]. Si prevede che grazie all'introduzione dell'IoT nel settore automotive si avrà un mercato potenziale che passerà dagli attuali 40.3 miliardi di euro ai 122.6 miliardi nel 2021. Come si evince dal grafico i settori trainanti saranno la guida autonoma, il benessere all'interno del veicolo, l'entertainment e la safety. Il veicolo si evolve e viene indicato con il termine *veicolo connesso*.

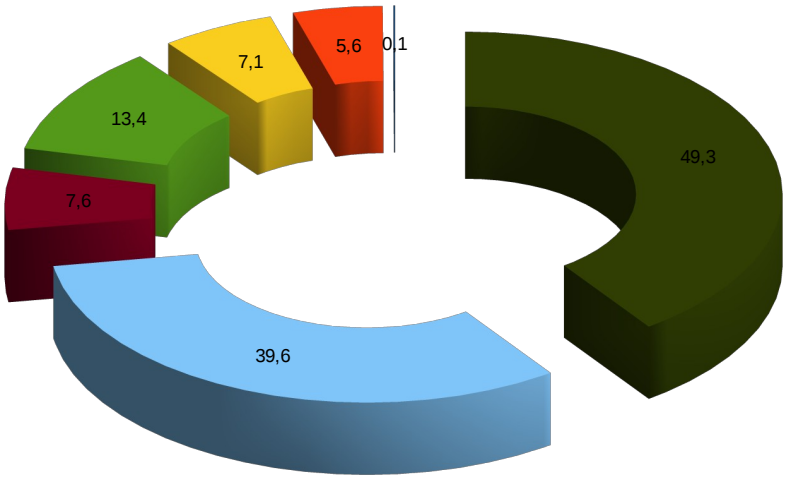
Il *veicolo connesso* è in grado di comunicare con gli altri veicoli (V2V), con l'infrastruttura stradale (V2I) e con i pedoni (V2P).

Il V2V permette ai veicoli di comunicare tra di loro, si scambiano informazioni sulla posizione reciproca evitando incidenti e sulle condizioni del traffico tramite protocollo IEEE 802.11p. Il veicolo è inoltre in grado di individuare i pedoni, V2P, sia intercettando le onde radio emesse dai loro smartphone sia tramite tecnologie laser diminuendo quindi il numero degli investimenti accidentali. Ma esso è anche in grado di comunicare con l'infrastruttura, V2I, ovvero strade, parcheggi, forze dell'ordine.

Tali tecnologie permettono lo sviluppo di innumervoli scenari. Il veicolo sarà in grado di prevenire gli incidenti ma nel malaugurato caso che se ne verifichi uno esso può comunicarlo tempestivamente alle forze dell'ordine che mobiliteranno i soccorsi. Potrà comunicare la sua posizione alla SmartHome del conducente in modo che possa sapere quando accendere il riscaldamento, il forno, le luci ecc... . Il veicolo potrà inoltre interfacciarsi con lo smartphone in modo da arricchire l'entarteinment con i contenuti multimediali presenti all'interno del device. Attualmente ciò è già possibile insieme all'intercettamento delle chiamate e alla risposta automatica.



2016 Totale € 40.3 miliardi



2021 Totale € 122.6 miliardi

Figura 1. Prospettive di marketing settore automotive [27]

Car Sharing & IoT

Il *Car Sharing* come lo intendiamo oggi, rappresentato ad esempio dalle società Enjoy [8] e Car2Go [5], è basato sul concetto di *veicolo connesso*. Tramite il collegamento del veicolo alla rete internet è possbile sbloccarlo da remoto (Keyless system) tracciarne la posizione, prenotarlo per un dato utente e sapere le condizioni interne del veicolo (es. livello del carburante). Le tecnologie attuali permettono di ipotizzare delle possibili evoluzioni.

Sicuramente lo sviluppo della guida autonoma influenzerà anche il mondo del *Car Sharing*, non è difficile immaginare un veicolo che permetterà all'utente di rilassarsi durante il viaggio e autonomamente giungerà a destinazione, una fusione e intersezione tra il *Car Sharing* e il servizio taxi a guida autonoma.

Esistono già dei progetti che permetteranno all'utente del servizio *Car Sharing* di trovare e/o prenotare il parcheggio più vicino.

Il navigatore satellitare presente all'interno dei veicolo potrà fornire informazioni in tempo reale sulle condizioni del traffico, suggerire percorsi più brevi, panoramici o veloci studiando le abitudini dell'utente.

Il V2P e il V2V saranno applicati nei prossimi anni anche ai veicolo del *Car Sharing* come su tutti gli altri veicoli.

Per il benessere dell'utente grazie a sensori di temperatura, umidità, pressione, pioggia si potrà gestire i finestrini in maniera automatica, chiudendoli ad esempio in caso di pioggia. Attraverso algoritmi di Machine Learning si studierà il comportamento dell'utente in base ai dati dei sensori e in relazione ad essi si attiverà il climatizzatore o si apriranno i finestrini. Studiando la voce dell'utente si potrà intuirne le emozioni ed avviare la riproduzione di musiche compatibili con il suo umore.

Ogni utente ha una sua fisionomia, delle preferenze di guida, uno stile di guida personale. Analizzando tali caratteristiche si potrà regolare la posizione dei sedili in funzione della sua statura, la posizione degli specchietti retrovisori, ottimizzare i paramteri del motore in base al suo stile di guida.

In questo ultimo caso è l'utente che trae beneficio dall'integrazione tra IoT, Machine Learning e *Car Sharing*. Infatti tali informazioni relative all'utente possono essere salvate ed usate da ogni veicolo della flotta del *Car Sharing* proprio perchè questi sono *veicoli connessi*.

In questo scenario le industrie coinvolte nella progettazione e realizzazione delle centraline elettroniche atte ad aggiungere al veicolo le funzionalità richieste dal *Car Sharing* (tracciamento GPS del veicolo, consuntivazione dell'utilizzo del servizio per uno specifico utente, sbloccaggio del veicolo da remoto) hanno la necessità di ampliare le funzionalità di base per offrire sul mercato prodotti innovativi e vincere sulla concorrenza. Una tra le possibili funzionalità aggiuntive è l'autoconfigurazione del veicolo in base all'utente.

Obiettivo della tesi

L'obiettivo di questa tesi è la realizzazione di un ***Proof of Concept***, abbreviato *POC*, per l'autoconfigurazione di un veicolo in ambito *Car Sharing*. Si vuole realizzare un sistema capace di reimplementare lo sblocco di un veicolo da remoto senza l'utilizzo di una chiave fisica, ormai standard, ma attraverso un'autenticazione biometrica dell'utente.

Nel contempo, all'atto dell'apertura del veicolo, il *POC* dovrà sperimentare la nuova funzionalità del configurare le varie componenti interne (per esempio infotainment, sedili, sterzo, specchietti ecc...), dotate di centralina elettronica, in base alle preferenze dell'utente corrente al fine di farlo sentire a suo agio.

Altresì alla chiusura del veicolo il *POC* dovrà salvare le configurazioni in modo che siano applicabili su qualsiasi altro veicolo della flotta del servizio di *Car Sharing*.

Stato dell'arte sistemi di Car Sharing

Enjoy, CAR2GO

Il punto di partenza è stato capire le interazioni tra utente e veicolo che le attuali compagnie di *Car Sharing* hanno realizzato. Ho dunque analizzato le applicazioni Android delle due compagnie di *Car Sharing* Enjoy e CAR2GO notando che non richiedono permessi bluetooth o NFC, quindi ho ipotizzato che non c'è comunicazione diretta tra veicolo e utente; altresì dovrebbe esistere una comunicazione diretta tra utente e server ed è quest'ultimo che comunica con la centralina di bordo per darle i comandi di blocco / sblocco.

Secure Free-Floating Car Sharing for Offline Cars

La ricerca svolta da Alexandra Dmitrienko e Christian Plappert si è concentrata sullo studio di fattibilità di un sistema capace di sbloccare un veicolo senza che questi sia connesso alla rete internet. La tecnologia abilitante è l'RFID. In questo modo è possibile estendere il servizio di *Car Sharing* anche in zone con scarsa copertura di rete (punti ciechi). L'apertura del veicolo avviene tramite un'autenticazione a due fattori per aumentare la sicurezza del sistema. Il primo fattore di autenticazione è creato durante la registrazione dell'utente al servizio, il secondo è scaricato dal server all'atto della prenotazione di un veicolo. Lo sblocco è pilotato tramite un locker, *TBox, Telematic Box*, un modulo hardware (in questo caso hanno usato un Arduino) che consente al veicolo di comunicare con l'esterno tramite Wifi, Bluetooth, NFC e altresì di interfacciarsi con il le ECU presenti a bordo tramite protocollo CAN e/o LIN.

Il soggetto che produce le centraline, il Car Sharing Provider, per ciascun veicolo memorizza del suo database e nel locker tre stringhe ID_L, K^L_{Auth}, K^L_{Ciph}, vale a dire

l'identificativo del locker del veicolo, una chiave per autenticazione e una per la cifratura dei messaggi inviati locker. Il server inizializza inoltre una smartcard con due file, uno che rappresenta l'utente e uno che rappresenta la posizione del veicolo. Alla registrazione dell'utente, che presenterà un documento di riconoscimento, la smartcard sarà configurata con l'identificativo dell'utente e una chiave relativa all'utente.

Quando l'utente deve utilizzare un veicolo provvede ad autenticarsi presso il server tramite l'identificativo assegnatogli e riceve un token relativo al veicolo. Tale token sarà inviato al locker che ne verifica l'autenticità e se è corretto il veicolo si aprirà. La posizione dell'utente è estratta dal suo telefono ed è inviata al server. Al checkout la posizione sarà presa del veicolo per affettuare la consuntivazione. [20]

Keyless Car Sharing System

Iraklis Symeonidis, Mustafa A. Mustafa, and Bart Preneel hanno proposto un sistema che esula dalla mediazione delle compagnie di *Car Sharing* e si propone di mettere in comunicazione diretta l'utente e il proprietario del veicolo tramite un server. Altresì è un sistema che non fa uso di chiavi fisiche per aprire il veicolo ma sfrutta chiavi elettroniche. Infatti inizialmente l'utente si registra sul server fornendo un documento d'identità (la patente) e le informazioni per contattarlo (es. email). Il proprietario del veicolo si registra sullo stesso server indicando il modello dell'auto, targa, certificati ecc... . Quando l'utente vuole utilizzare un veicolo, il proprietario riceve la richiesta sul suo smartphone e se l'accetta deriva dalla sua chiave principale una chiave temporanea per sbloccare l'auto. Il proprietario può revocare in qualsiasi istante le chiavi generate. Gli utenti inoltre ricevono delle valutazioni dopo ogni viaggio che consentono ai proprietari di accettare o meno le richieste.

I componenti che costituiscono il sistema sono : il KSMS (il server centrale), la KSApp l'applicazione per smartphone che consente ad un utente di prenotare un veicolo, la PD-KSApp l'applicazione per smartphone che consente al proprietario del veicolo di gestire le richieste e la KS-OBU la *TBox* presente a bordo del veicolo che consente di sbloccarlo con la chiave digitale.

Lo studio condotto da Symeonidis, Mustafa e Preneel, più che analizzare nei dettagli il modello hardware/software del sistema, esamina gli aspetti di sicurezza e la privacy del sistema ad alto livello. Tra le minacce di sicurezza vi sono lo spoofing (l'attaccante cerca di assumere l'identità di un utente già presente nel sistema), DoS verso il server, Information Disclousure (modifica, eliminazione dei messaggi scambiati tra le parti , dati in motion), Elevation of privilege (l'attaccante cerca di accedere con un account a privilegio più alto), Tampering with data (modifica dei dati at rest). Per la privacy invece hanno individuato problemi relativi al rilevamento delle abitudini dell'utente. Da ciò hanno stilato i *Security Requirements*:

- autenticazione delle controparti
- integrità dei dati at rest e di quelli in motion

- riservatezza dei dati (solo chi è autorizzato può leggere i dati)
- autorizzazione (solo chi è autorizzato può modificare i dati)
- non ripudio delle prenotazioni o dei viaggi
- disponibilità delle risorse (assets)

Hanno altresì ipotizzato di utilizzare un sistema basato su certificati digitali. [23]

Capitolo 1

Architettura di rete di un veicolo

All'interno di un veicolo sono presenti sia attuatori (motorino finestrino, motorino posizione luci, ecc...) che sensori (stato della porta, stato della chiave, ecc..). Inizialmente, prima degli anni '80, vi era una sola centralina elettronica che era connessa tramite collegamenti point-to-point a tutti i sensori e gli attuatori. Al crescere delle funzionalità offerte dal veicolo sono cresciuti il numero di elementi da connettere. Ecco dunque che un collegamento point-to-point a raggiera non era più sostenibile. Il numero di fili era troppo elevato.

Per ovviare a questo problema varie case automobilistiche iniziarono in modo indipendente a sviluppare il concetto di **rete** interna al veicolo. Invece di utilizzare dei semplici attuatori meccanici si sono utilizzati attuatori meccatronici ovvero dotati di centralina elettronica e si sono interconnessi tramite bus. Così facendo si è ottenuto un *sistema multiplex*. La Bosch ha sviluppato il bus *CAN* (Controller Area Network), la Renault e la Peugeot il bus *VAN* (Vehicle Area Network) e la Ford il *J1850*.

Il veicolo è diventato così un insieme di microprocessori che collaborano e si scambiano informazioni andando a formare una vera e propria rete di calcolatori distribuiti in cui si necessita di allineare costantemente i dati.

Oggigiorno il bus più utilizzato tra quelli citati prima, che è sfruttato per le sue caratteristiche anche in settori non inerenti al mondo automotive, è il **CAN**

1.1 Protocollo CAN

Il CAN (Controller Area Network) è un protocollo per bus seriale di tipo broadcast, multi-master, sviluppato dalla Bosh e largamente utilizzato nel campo dell'automazione. Fisicamente è costituito da un doppino intrecciato e i nodi non hanno un indirizzo, ma rispondono a messaggi specifici di cui loro sono competenti e riconosciuti tramite un identificativo. Esso va a standardizzare i primi due livelli del sistema ISO-OSI. Per ottenere un'elevata affidabilità, la velocità massima è di 1 Mbit/s se la lunghezza del cavo

è minore di 40 m. A livello fisico è utilizzata una codifica NRZ (not return to zero) e i due livelli utilizzati sono detti **dominante**, che rappresenta il livello logico 0 (D bit) e **recessivo**, che rappresenta il livello logico 1 (R bit). Nel caso in cui più nodi trasmettono sul bus nello stesso istante, se qualcuno trasmette il bit dominante (0) questi prevale e il bus assume tale valore. La risoluzione della contesa del bus avviene tramite i bit dell'identificativo del messaggio. L'identificativo è la prima sequenza che viene trasmessa, tutti i nodi che trasmettono sono anche in modalità ricezione. I nodi che ricevono un bit diverso da quello inviato si bloccano e attendono che il bus sia libero. Il messaggio con l'identificativo con più bit dominanti sarà trasmesso e il nodo che trasmette non si accorge neanche che è avvenuta una contesa del bus.

I messaggi scambiati hanno cinque formati:

- **Data Frame** messaggio con dati, fino a 64 bit
- **Remote Frame** messaggio privo di dati, per sollecitare un nodo a trasmettere
- **Error Frame** inviato da chi rileva un errore per sollecitare chi ha trasmesso a ri-trasmettere
- **Overload Frame** per ritardare la trasmissione di un pacchetto
- **Interframe Space** per separare Data/Remote/Error frame

Il tipo di messaggio principale è il *Data Frame*. Le componenti base di tale messaggio sono un identificativo, i dati e il CRC. La versione CAN 2.0A ha 11 bit come identificativo, la versione 2.0B invece 29 bit. Sono presenti 4 bit per indicare la lunghezza del messaggio, da 0 a 64 bit di dati e altri bit di controllo, CRC e sincronizzazione.

Un nodo può essere in tre stati : *Error active* se il nodo funziona normalmente, *Error Passive* se il nodo ha contato un numero di errori superiore a 128 in ricezione o trasmissione, *Bus off* se il nodo ha contato un numero di errori superiore a 256 in trasmissione. Se il nodo è nello stato di *Bus off* non è abilitato a trasmettere sul bus. Dopo un Data o Remote frame è trasmesso un Interframe space composto da 3 R bit (Intermission) e da 8 R bit

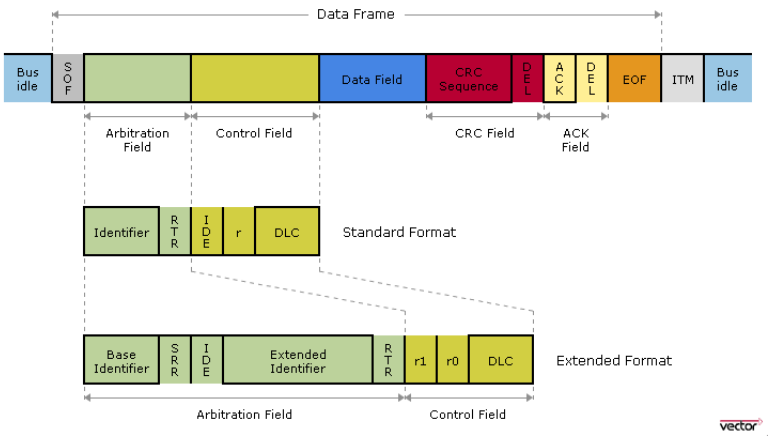


Figura 1.1. Formato di un Data Frame

(Suspend transmission). Quando un nodo in stato di *Bus off* riceve 128 sequenze di 11R bit (128 Interframe space) i contatori sono resettati, il nodo torna nello stato Error Active e può quindi ricominciare a trasmettere.

Si verifica un errore quando:

- Il bit trasmesso è diverso da quello ricevuto (non durante la contesa)
- La regola del bitstuffing non è rispettata
- Il nodo riceve un CRC errato
- Se chi trasmette riceve un R bit invece del D bit nell'ACKSlot (ACKNOWLEDGEMENT Error)

1.1.1 Livello Applicativo

Quello descritto fin'ora era relativo ai primi due livelli (Fisico e Data Link) che sono stati già standardizzati. Il livello Data Link in ricezione fornisce al livello superiore due blocchi di dati : identificativo e messaggio. Il livello superiore è direttamente quello applicativo e per tale livello nel corso del tempo sono stati creati vari protocolli tra i quali ad esempio CANOpen e SAE J1939.

Per far si che le varie ECU possano comunicare tra loro in modo semplice e scalabile, spesso si adotta la procedura di creare un database con il contenuto dei messaggi per ogni identificativo (messaggi applicativi, non quelli del livello datalink). In particolare l'azienda Vector è proprietaria del formato per database detto DBC. È un file che contiene il nome delle ECU presenti sulla rete, i messaggi che possono essere scambiati, chi li invia e chi li riceve e la descrizione dei segnali per ciascun messaggio [17].

1.1.2 Organizzazione del dizionario DBC

Messaggi

Il Data Frame è il messaggio a livello datalink ideale per trasportare informazioni tra le ECU. Basandosi sulla struttura di tale messaggio si è creato un messaggio a livello applicativo contraddistinto essenzialmente da due informazioni : l'identificativo del messaggio (un id univoco su 11 o 29 bit) e il dato raw da trasportare (da 0 a 64 bit).

Segnali

Il dato raw contenuto in un messaggio applicativo di per se può rappresentare qualsiasi informazione. Per automatizzare la comprensione da parte delle ECU del significato dei messaggi si è pensato di dividere questi da 0 a 64 bit in varie sezioni. Ogni sezione rappresenta un *segnale*. All'interno del database DBC esso è identificato da un nome ed è

associato al numero del bit di inizio nel messaggio, alla sua lunghezza in bit, all'eventuale unità di misura, al fattore di scala e all'ordinamento dei byte. L'ordinamento puà essere Intel (little-endian) o Motorola (big-endian). Questo fa si che quando si utilizza il bit di inizio del segnale presente nel database questi corrisponde al bit 0 se l'ordinamento è Intel, non lo è se invece l'ordinamento è Motorola.

All'interno del DBC inoltre è presente un'indicazione del tipo di messaggio. Esso può essere ciclico (è riportata la distanza temporale tra due messaggi consecutivi con lo stesso id in ms) o ad evento (in risposta ad un'altro messaggio CAN o al verificarsi di una serie di condizioni).

Figura 1.2. DB9 connettore per il CAN bus

1.1.3 CAN classi di funzionamento

C-CAN

Il C-CAN è la classe di funzionamento di multiplexing veloce. Ha una velocità compresa tra i 125 kb/s e 1 Mb/s. È utilizzato per la trasmissione di informazioni ad elevata criticità, dove il tempo di risposta deve essere inferiore ai 5 ms. In particolare mette in comunicazione il motore, l'ABS e il cambio.

B-CAN

Il B-CAN è la classe di funzionamento di multiplexing lento. Ha una velocità compresa tra i 32,5 kb/s e 125 kb/s. È utilizzato per la per la trasmissione di informazioni che non necessitano di risposte real time, infatti il tempo di risposta è inferiore ai 100 ms. In particolare mette in comunicazione BCM e ESL.

I-CAN

L' I-CAN è la classe di funzionamento di multiplexing veloce ma adibita al trasporto di informazioni multimediali. Ha una velocità compresa tra i 125 kb/s e 1 Mb/s. È utilizzato per la per la trasmissione di segnali audio/video.

1.2 Protocollo LIN

Una prima opzione per collegare dispositivi a velocità diversa è stata l'utilizzo delle varie classi di funzionamento del CAN. Però, nell'ottica di ridurre le connessioni tra le ECU e ridurre i costi di produzione, sul finire degli anni '90 grazie ad un consorzio tra aziende automotive, un'azienda di semiconduttori e una di informatica, è stato sviluppato il **LIN** (Local Interconnection Network).

È un bus del tipo *single master - multi slave* con singolo filo e velocità intorno ai 20 kb/s. Il LIN è usato come rete locale per il CAN. In particolare mette in comunicazione sedili, luci, tergicristalli, finestrini, ecc..., tutte componenti che non richiedono di essere reattive e che rispetto agli altri nodi della rete scambiano poche informazioni. Per essere integrato con il CAN necessita di un nodo di interconnessione che viene definito Gateway.

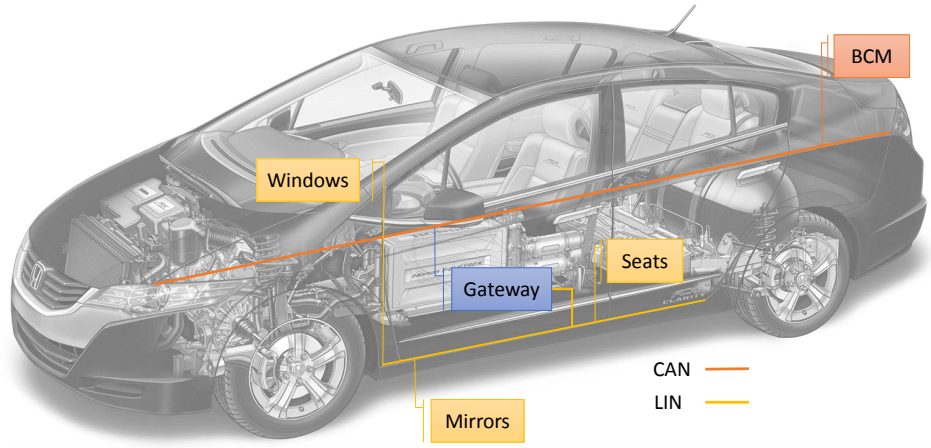


Figura 1.3. CAN-LIN

Attualmente la configurazione più utilizzata è quella di avere un bus CAN a 500 kb/s e alcuni dei nodi CAN con una sottorete locale LIN come mostrato in Figura 1.3.

Il LIN può arrivare a supportare fino a 15 slave. Non necessita di controllo del canale perchè pur essendo broadcast è sempre il master a iniziare la conversazione. Solo uno slave alla volta risponde perchè il master indica un id che è associato ad un solo slave e solo lui può rispondere. I dati trasmessi possono avere lunghezza variabile 2, 4 o 8 byte. Sono inoltre corredati da un checksum e il LIN è in grado anche di rilevare il malfunzionamento di un nodo.

1.3 Centraline principali

Le centraline elettroniche presenti a bordo di un veicolo (ECU), in un contesto di sistema multiplex, sono classificabili in base alla loro criticità che si traduce in termini di priorità e velocità richiesta delle loro comunicazioni. Ad esempio il motore, il cambio e i freni hanno una priorità maggiore rispetto a climatizzatore, blocca sterzo, sedili e specchietti. I primi devono reagire in *real time* quindi devono comunicare con velocità > 200 kb/s, i secondi invece possono comunicare con velocità tra i 30 kb/s e i 125 kb/s.

Engine control module

L'engine control module (ECM) è la ECU deputata all'accensione e gestione del motore. Attraverso sensori di temperatura del motore e composizione dei gas di scarico ha il compito di decidere la quantità di miscela da iniettare nel motore per avere una combustione ottimale e ridurre i gas di scarico. Attraverso il bus comunica le informazioni relative al motore, in particolare la velocità di rotazione e la sua temperatura.

Body control module

Il body control module (BCM) è la ECU che controlla tramite bus i finestrini, gli specchietti, il climatizzatore, il blocca sterzo e l'immobilizer (apertura con telecomando). Essa è in comunicazione tramite collegamenti point-to-point con i sensori delle porte (chiuse/aperte) con il sensore della posizione della chiave (ON/OFF) e il sensore di velocità del veicolo.

Gateway

Il Gateway (GW) è una centralina deputata alla conversione dei protocolli di comunicazione per permettere l'interconnessione di reti differenti, ad esempio in presenza di bus a diverse velocità.

Electronic steering column lock

L'electronic steering column lock (ESL o ESCL) è un attuatore meccatronico in grado di bloccare o sbloccare lo sterzo permettendone o inibendone la rotazione in modo automatico sotto determinate condizioni. È utilizzato come sistema di sicurezza in modo da impedire il furto del veicolo. Può effettuare un lock nel caso in cui il veicolo sia spento da 48 secondi, oppure se la porta viene chiusa o se glielo chiede il BCM. Effettua un unlock invece appena si accende il motore.

1.4 Network Management

Le ECU elencate hanno, come già detto, la capacità di comunicare tra di loro. Al verificarsi di determinate condizioni, rilevate tramite la traduzione mediante il dizionario dei segnali contenuti nei messaggi CAN, le ECU possono attivare le funzioni interne ed eseguire i task che gli competono. L'ESL alla chiusura della porta e sotto la condizione di key OFF e veicolo fermo effetua un lock, l'Airbag in funzione del rilevamento di una pressione eccessiva su un lato della scocca provoca l'esplosione del gas che espande i cuscini salvavita, l'ABS in presenza di pioggia diminuisce il rapporto tra posizione del pedale del freno e forza della frenata ecc... . Ma tutte le ECU hanno anche la necessità di coordinarsi non rispetto agli eventi esterni ma rispetto a loro stesse.

Appena collegati i morsetti della batteria tutte le ECU effettuano il boot e iniziano a trasmettere su CAN. Ma come si accorge il sistema che tutte le ECU sono attive e funzionanti e non ci sono ECU in avaria? Come fanno a non esaurire la batteria dovendo rimanere sempre in funzione (almeno alcune di esse) ?

Solitamente i veicoli utilizzano un protocollo di network management che consente di rispondere proprio a tali domande. Appena una ECU riceve il power on, ossia è connessa ad alimentazione o si riavvia, emette sulla rete CAN un messaggio ciclico che utilizza un range di indirizzi predefiniti in cui dice di essere **alive**. Questi messaggi sfruttano la rete CAN in modo diverso, invece di essere percepiti dalle ECU come messaggi broadcast vedono la rete come se avesse una topologia ad anello. Ogni ECU ha un successore predefinito e il successore dell'ultimo nodo è il primo nodo. In questo modo il sistema si accorge facilmente che qualcuno non si è attivo.

1.4.1 Sleep e Wake Up

Sempre sfruttando questo range specifico degli id dei messaggi, che varia da veicolo a veicolo, è possibile che una ECU dica alle altre che lei ha rilevato le condizioni per andare in basso consumo (sleep) e quindi è in grado di risparmiare batteria. Emette un messaggio in cui un particolare segnale indica che lei può andare in sleep. I processori quando vanno in sleep spengono delle componenti che consumano più energia o che non servono lasciando attiva solo la ricezione degli interrupt da parte del processore stesso.

Ogni nodo conosce il numero di nodi totali sulla rete (N). Quando riceve N-1 indicazioni di sleep, allora invia al nodo successore logico un messaggio CAN sempre di network management in chi si segnala che avendo ricevuto la disponiilità di tutti per andare in sleep la ECU sta passando in modalità sleep.

Le ECU quando sono in sleep sono attive solo in ricezione sul bus CAN. Alla ricezione di un qualsiasi messaggio CAN o al passaggio della chiave da OFF a ON ecco che si risvegliano (fase di wake up) e mandano nuovamente il messaggio di **alive**.

Se le ECU non utilizzassero questo meccanismo la carica della batteria si esaurirebbe in poche ore anche a veicolo spento.

1.5 Diagnosi del veicolo

L'architettura di rete di un veicolo è complessa perchè oltre ai messaggi applicativi e ai messaggi per la gestione della rete stessa vi sono anche i messaggi di diagnosi. Il Diagnostic Trouble Code, comunemente abbreviato in DTC, è un messaggio CAN che è inviato dalla ECU che rileva un errore e solitamente è intercettato dalla BCM e ivi conservato per essere facilmente letto da parte del meccanico attraverso un'opportuna strumentazione (ODB-II connector).

Le informazioni principali riportate all'interno di un DTC sono il codice dell'errore, un codice che identifica un possibile sintomo, un contatore e lo stato dell'errore, se è in real time o se è stato registrato.

Capitolo 2

Il Progetto

2.1 Lo scenario

Viene ora presentata una serie di scenari per introdurre una visione globale del problema da risolvere, ovvero la realizzazione del *POC*, per poi proseguire con l'analisi del progetto dello stesso.

Una persona che sia leggittimata a guidare un tipo di veicolo (automobile) si iscrive al servizio di *Car Sharing* che è gestito dal *POC* e diventa un utente del sistema. Tramite un'interfaccia utente è in grado di vedere le vetture presenti in zona e tramite autenticazione biometrica può sbloccare un determinato veicolo. Il veicolo, al primo accesso dell'utente dopo la registrazione, avrà lo stato di partenza di ciascuna ECU con i parametri inizializzati con dei valori di default. Le ECU sono i controllori delle parti configurabili (sedili, specchietti, infotainment) che saranno appunto nello stato di default.

Un utente del sistema dopo lo sblocco del veicolo procede a modificare gli elementi presenti nel veicolo, ad esempio cambia la frequenza FM dell'infotainment e arrivato a destinazione spegne il veicolo. Lo stesso utente successivemante sblocca un'altro veicolo. La vettura imposterà la frequenza FM dell'infotainment uguale all'ultima frequenza impostata sull'altro veicolo dallo stesso utente.

Due utenti che hanno precedentemente ascoltato frequenze FM diverse durante l'ultimo utilizzo del servizio accedono in sequenza allo stesso veicolo. Il veicolo allo sblocco imposterà per ciascun utente la rispettiva frequenza FM sull'infotainment.

Due utenti stanno visionando la stessa zona tramite l'interfaccia utente. Nel momento in cui il primo utente sblocca un veicolo, tale veicolo non deve essere più visibile. Quando il veicolo è di nuovo libero deve essere nuovamente visibile.

Un utente vuole sbloccare un veicolo ma essendo lontano dallo stesso il processo fallisce. L'utente si avvicina, ripete la procedura e questa volta il veicolo si sblocca.

2.2 Analisi

A partire dagli scenari proposti si sono individuate le caratteristiche che il sistema deve avere e che sono sintetizzate di seguito.

Una società di *Car Sharing* deve poter installare all'interno dei suoi veicoli un componente elettronico, da qui in avanti indicato con il termine *TBox* , capace di comunicare con le ECU (presenti a bordo veicolo) e in remoto sulla rete internet. Le ECU, inoltre, devono poter essere comandabili al fine di poter configurare gli attuatori che sono sotto il loro controllo (es. la ECU dell'infotainment deve poter rispondere a dei comandi esterni). Condizione necessaria ai fini dell'autoconfigurazione è che le ECU siano in grado di ricevere uno stato (che è la configurazione) e di poter comunicare il loro attuale stato. Ciò è dovuto al fatto che il sistema POC si affiancherà al sistema preesistente che già consentiva delle operazioni che l'utente potrà comunque continuare a svolgere (es. cambiare la posizione degli specchietti retrovisori). Nel momento in cui l'utente svolge tali azioni il sistema deve potersene accorgere e ricordarle.

Per la geolocalizzazione del veicolo la *TBox* deve poter estrarre e comunicare in remoto la sua posizione GPS. La *TBox* deve poter conoscere la posizione della chiave di accensione (deve interfacciarsi con il segnale di Ignition Key), deve poter sbloccare il veicolo agendo sull'apertura delle porte ed effettuare un unlock dell'ESL (electronic steering lock, il bloccasterzo elettronico). Deve poter accorgersi che il veicolo è stato chiuso effettuando di conseguenza un lock dell'ESL.

La stessa società di *Car Sharing* deve avere un'infrastruttura informatica che le consenta da un lato di geolocalizzare i veicoli e comunicare con essi, dall'altro deve poter gestire gli utenti. Deve quindi offrire loro la possibilità di registrarsi, di sbloccare ed utilizzare un veicolo. Lo sblocco deve essere possibile solo se l'utente è nelle vicinanze del veicolo. Deve altresì tener traccia di quale utente ha utilizzato quale veicolo e per quale lasso di tempo. Tali dati potranno poi essere utilizzati per la consuntivazione del servizio.

L'utente deve potersi registrare sull'infrastruttura informatica del *Car Sharing* e deve poter visualizzare tramite essa i veicoli disponibili, la loro posizione e deve poterli sbloccare. Nel momento dello sblocco del veicolo deve anche avvenire la configurazione dello stesso rispetto all'utente corrente.

2.2.1 Requisiti del sistema

Si riportano di seguito i requisiti che il sistema finale dovrà avere al fine di poter soddisfare l'obiettivo della tesi.

L'infrastruttura:

R 1.1 Deve consentire la creazione di un account utente

R 1.2 Deve consentire la modifica di un account utente

R 1.3 L'utente deve avere un identificativo univoco all'interno del sistema

R 1.4 Deve riconoscere l'utente mediante autenticazione biometrica

R 1.5 L'autenticazione deve essere valida per non più di 1 minuto

R 1.6 Deve poter conoscere la posizione GPS dell'utente

R 1.7 Deve poter conoscere la posizione GPS dei veicoli

R 1.8 Deve poter conoscere lo stato di un veicolo (libero o occupato)

R 1.9 Deve riconoscere quale veicolo vuole utilizzare l'utente

R 1.10 Deve sbloccare un veicolo solo se l'utente è a meno di 3 m da esso

R 1.11 Deve sbloccare un veicolo solo se è libero all'atto della richiesta

R 1.12 Deve poter configurare un veicolo

R 1.13 Deve poter conoscere la configurazione di un veicolo rispetto all'utente corrente

R 1.14 Deve consentire all'utente di conoscere quali veicoli della flotta sono liberi

R 1.15 Deve poter notificare all'utente il cambio di stato di un veicolo

La TBox:

R 2.1 Deve poter comunicare la sua posizione GPS all'infrastruttura

R 2.2 Deve poter sbloccare il veicolo quando richiesto dall'infrastruttura

- Deve aprire lo sportello lato conducente
- Deve sbloccare il bloccasterzo elettronico

R 2.3 Deve poter configurare il veicolo rispetto a quanto richiesto dall'infrastruttura

R 2.4 Deve poter rilevare la configurazione del veicolo

R 2.5 Deve poter comunicare all'infrastruttura la configurazione del veicolo

R 2.6 Deve accorgersi della chiusura del veicolo, bloccare il bloccasterzo e comunicarlo all'infrastruttura

2.3 Il sistema

2.3.1 L'architettura

Partendo dall'analisi del problema e dai requisiti di sistema, discussi nella sezione precedente, ho progettato l'architettura mostrata in Figura 2.1. È composta da tre blocchi fondamentali:

- Applicazione per telefono cellulare (da qui in avanti chiamata *Mobile App*)
- Server centrale
- *TBox*

Analizzando l'infrastruttura informatica si può notare che essa rappresenta due macro aree. Nello stesso tempo è sia *Model* che *View* come si evince dalla sottosezione 2.2.1. Si nota subito che queste due aree sono ovviamente interdipendenti ma separabili. Per tale motivo l'infrastruttura informatica è stata suddivisa in due parti, da un lato il server che assolve alla funzione di gestione dell'interazione tra utenti e veicoli e dall'altro la Mobile App che è la *View* a disposizione dell'utente per poter interagire con il server stesso. La *TBox* invece è il modulo già definito a livello di analisi che andrà inserito all'interno del veicolo.

Definiti i blocchi fondamentali è bene capire come essi interagiscono e quindi qual è il flusso delle informazioni.

Innanzitutto è ovvio che l'utente inizierà la comunicazione con il server al fine di effettuare una prima registrazione. A questo punto si prospettano due possibili soluzioni all'atto della scelta del veicolo. Sicuramente la Mobile App deve comunicare al server il veicolo scelto e il server deve dare l'approvazione o il diniego. Nel caso di approvazione della richiesta sia il server che la Mobile App potrebbero iniziare la comunicazione con la *TBox* .

Se è la Mobile App a comunicare direttamente con la *TBox* si dovrebbe richiedere a livello implementativo l'utilizzo di protocolli quali NFC o Bluetooth. Questa scelta sicuramente fornirebbe una copertura del servizio maggiore, anche in aree con deficit di copertura di rete mobile come evidenziato nella ricerca di Alexandra Dmitrienko e Christian Plappert ma, viceversa, la funzione del server verrebbe ridimensionata. In questo caso le configurazioni dei veicoli potrebbero essere salvate direttamente sul dispositivo dell'utente e comunicate al veicolo con lo stesso protocollo utilizzato per la richiesta di sblocco. Però in caso di reinstallazione dell'applicazione o nel caso di cambio dello smartphone o di un semplice upgrade del sistema operativo le configurazioni verrebbero irrimediabilmente perse. Si necessiterebbe dunque di un sistema di backup, eventualmente periodico, che coinvolgerebbe nuovamente il server. Si avrebbe una duplicazione delle informazioni.

L'altra opzione, invece, vede per lo sblocco la comunicazione diretta tra *TBox* e server. Come descritto nei requisiti la *TBox* deve comunicare al server la sua posizione GPS e il suo stato (veicolo libero o in uso). Dunque un canale di comunicazione tra le due parti deve comunque essere instaurato, tanto vale che anche il comando di sblocco possa passare su tale canale. In questo modo si evita anche la duplicazione delle informazioni e il flusso di comunicazione diventa molto più lineare. Oggigiorno a meno di aree rurali o montane (dove il servizio di *Car Sharing* non è fruibile) la connessione a internet tramite rete cellulare non è assolutamente un problema. Dunque non vi è più la necessità e il beneficio di utilizzare l'NFC. Il server ha così anche il pieno controllo sulle condizioni per le quali è possibile sbloccare un veicolo.

Rissumendo, la *Mobile App* consente all'utente di visionare i veicoli vicini e di chiedere al server di sbloccare un veicolo a valle della sua autenticazione biometrica. Il server verificando che l'utente è nelle immediate vicinanze del veicolo prescelto provvede a sbloccarlo e in questa fase invia anche le configurazioni per quell'utente, ovvero lo stato per ciascuna ECU. Allo spegnimento del veicolo questi invia al server lo stato delle sue ECU.

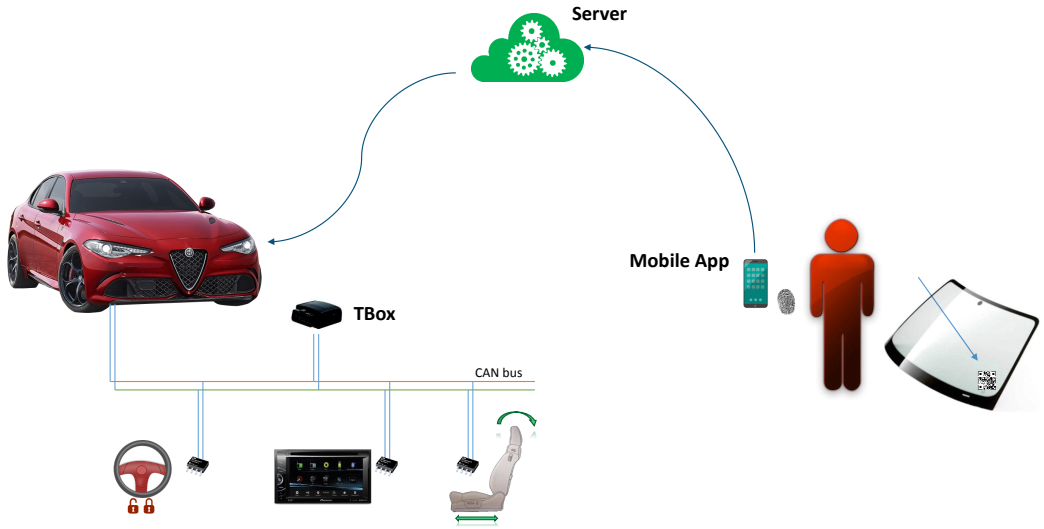


Figura 2.1. Componenti principali del sistema

2.3.2 Scelte implementative

Mobile App

L'utente ha bisogno di uno strumento per poter interagire con il sistema. Le due possibili scelte sono state un sito web e/o un'applicazione per smartphone. Dato il contesto in movimento per l'usufruizione del servizio di *Car Sharing* e data la necessità di geolocalizzare l'utente ho preferito orientarmi sull'applicazione per smartphone, la Mobile App. Essa inizialmente consente all'utente di registrarsi presso il server. Nel momento in cui l'utente vuole utilizzare un veicolo della flotta con essa può autenticarsi presso il server e scegliere il veicolo inquadrando il QR Code presente sul parabrezza del veicolo inviandolo automaticamente al server. Ho scelto di facilitare la scelta del veicolo all'utente consentendogli di inquadrare un QR code che contenga codificata la targa del veicolo stesso piuttosto che costringerlo ad inserire manualmente la targa sull'applicazione. L'applicazione inoltre dà un feedback all'utente sull'esito dello sblocco e configurazione del veicolo. Per lo sviluppo della Mobile App ai fini della dimostrazione del *POC* ho deciso di realizzarla per smartphone aventi sistema operativo Android perché è opensource e lo sviluppo è possibile attraverso qualsiasi sistema operativo (Linux, Windows, Mac OS X). Per quanto riguarda l'autenticazione biometrica dell'utente ho deciso di utilizzare il *fingerprint* perché ad oggi è l'unico metodo presente su molti device di fascia medio/alta e di cui esistono API per Android. L'utilizzo del fingerprint ha automaticamente fissato un constraint sulla versione minima di Android supportata (min. API level 23). Per versioni precedenti non esistono tali API. Per la lettura dei QR code si è utilizzata la libreria Google *com.google.android.gms.CameraSource*. Per la visualizzazione della posizione dei veicoli si è utilizzata la libreria Google Map's API 2.0.

Figura 2.2. Android Logo

La struttura e l'implementazione della Mobile app sono approfondite nel Capitolo 3.

Server

Il server centrale è il cuore del sistema informatico della società di *Car Sharing*. Permette la comunicazione tra l'utente, attraverso la Mobile App, e il veicolo, rappresentato dalla TBox. Consente la registrazione, l'autenticazione dell'utente e il cambio della password per l'account creato dall'utente. Altresì si occupa di comunicare con il veicolo inviando e ricevendo (e memorizzando) le configurazioni.

Il linguaggio scelto per la progettazione del server è stato Python per via della sua grande versatilità, del grande supporto che si può trovare su internet e perché ha una buona curva di apprendimento. Per la memorizzazione dei dati (utente, veicoli, utilizzo

dei veicoli) sul server ho scelto MariaDB perché opensource. Ho deciso di utilizzare il protocollo HTTPS invece dell'HTTP in modo da avere una comunicazione sicura, rispecchiante le proprietà del protocollo TLS, tra la Mobile App e il Server e altresì tra il veicolo e il Server. I dati sensibili sono cifrati all'interno del database.

La struttura e l'implementazione del server sono approfondite nel Capitolo 4.

TBox

La *TBox* è il modulo presente sul veicolo che si occupa da un lato di comunicare con il server (la sua posizione, il suo indirizzo di rete) e dall'altro di comunicare con le ECU presenti a bordo del veicolo. Le ECU tra di loro sono connesse tramite bus CAN e utilizzano messaggi CAN per comunicare. Tali messaggi possono essere sfruttati per configurare lo stato delle componenti del veicolo e leggerne lo stato attuale. È il server a chiedere alla *TBox* di sbloccare il veicolo. In questo caso deve comunicare con la ECU deputata (ESL) e effettuare un unlock del volante e deve aprire la porta. Quando l'utente chiude il veicolo con la chiave fisica, tale modulo, avente alimentazione autonoma, comunica al server l'ultima configurazione e chiude ordinatamente la connessione con il server. Mentre il server è un componente principalmente software che non ha connessioni con il mondo fisico, la *TBox* è prima di tutto una ECU, una centralina hardware che deve interfacciarsi con un veicolo. Internamente al veicolo le ECU comunicano tramite bus CAN (Controller Area Network). La *TBox* deve anche essere in grado di recuperare la sua posizione GPS.

Nella figura Figura 2.4 sono mostrate le ECU principali presenti a bordo di un veicolo e la *TBox* che viene connessa al veicolo tramite bus CAN.

Per soddisfare i vincoli imposti dal sistema è stato necessario individuare un dispositivo che avesse capacità computazionali discrete e che oltre al bus CAN e al GPS potesse ricevere ed inviare segnali fisici. La scelta è ricaduta sul single board computer il **Raspberry Pi v.3**, l'ultima versione disponibile ad oggi della scheda.

Il Raspberry Pi di per sè offre solo la connettività con i segnali fisici tramite i pin GPIO. Per sopperire alla mancanza del bus CAN e del GPS ho analizzato alcune schede di espansione disponibili sul mercato e la migliore rispetto al prezzo, funzionalità e documentazione è stata la 'PiCAN GPS and Accelerometer' della SK PANG Electronics [15] .

Figura 2.3. Python Logo

Definito l'hardware per la *TBox* la scelta è passata al software. Il sistema operativo che ho utilizzato per il Raspberry Pi è stato Raspbian (un fork del sistema operativo Debian). Il PiCAN è controllabile tramite due linguaggi di programmazione nativi il C e il Python. Considerato che la *TBox* deve gestire sia la comunicazione con il Server utilizzando il TLS sia andare a modificare i singoli bit dei messaggi CAN la scelta naturale è ricaduta sul Python.

Per la tesi, non avendo a disposizione un veicolo reale su cui installare una centralina (la *TBox*), si è provveduto ad emulare il sistema a bordo del veicolo. In particolare le funzionalità di base del Gateway e della BCM sono state integrate all'interno della *TBox* al fine di poter utilizzare correttamente l'ESL. Nella figura Figura 2.5 si ha una visione delle centraline utilizzate nel POC. La struttura e l'implementazione della *TBox* sono approfondite nel Capitolo 5.

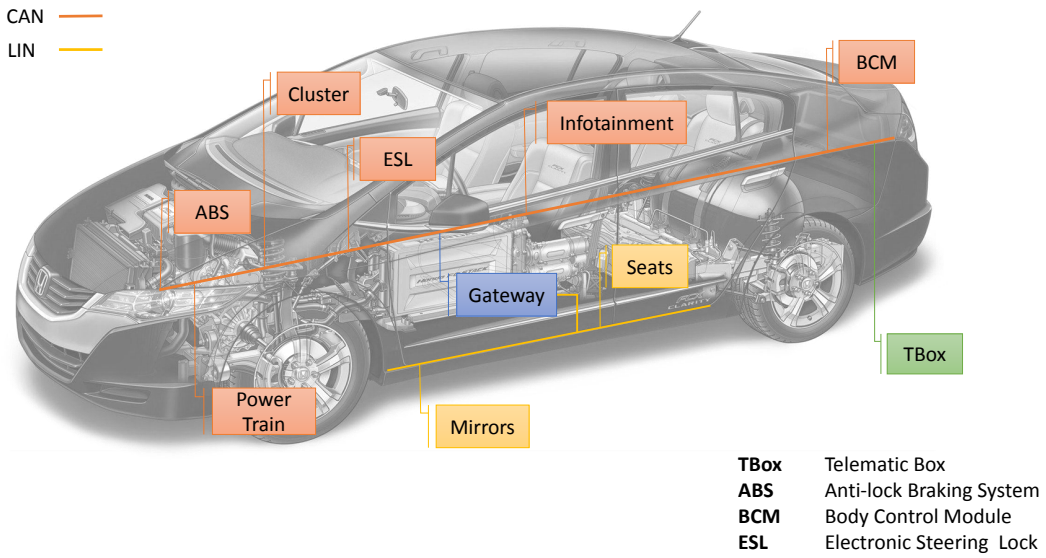


Figura 2.4. Sistema reale dopo l'inserimento della TBox, ECU presenti nel veicolo

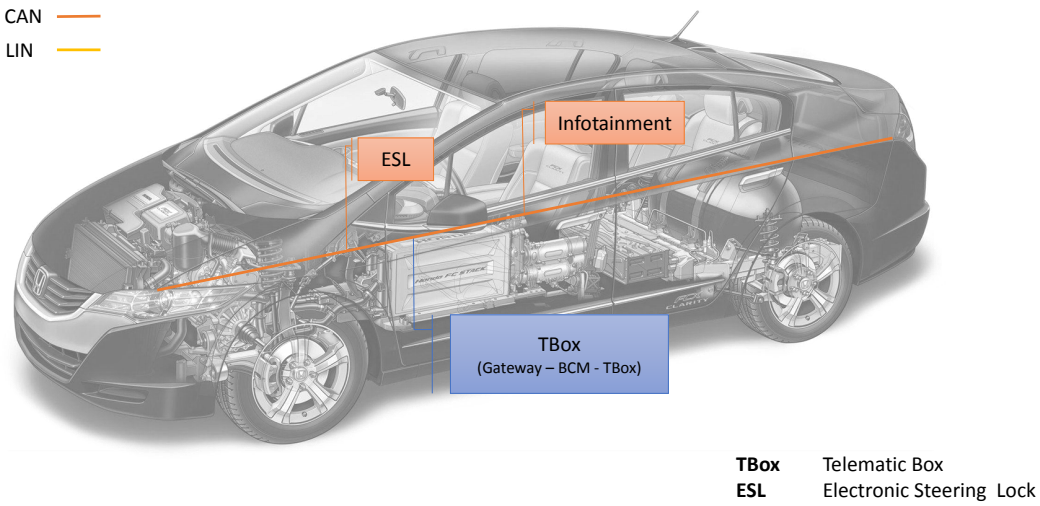


Figura 2.5. Sistema emulato per la tesi

Capitolo 3

Mobile Application

3.1 Introduzione

Lo scopo dell'applicazione mobile è quello di consentire ad un utente di interfacciarsi con i servizi offerti della compagnia di Car Sharing. Tra i vari servizi, quelli accessibili dall'app sono : la registrazione sul Server, nel momento precedente l' utilizzo del veicolo l'autenticazione dell'utente e la scelta del veicolo che si sta per noleggiare in modo tale che possa avvenire lo sblocco e l'autoconfigurazione dellla vettura in base alle preferenze dello stesso.

3.2 Contesto d'uso

Come per ogni sistema relativo al mondo ingegneristico-informatico, il punto di partenza per sviluppare questa componente del sistema è stato quello di identificare la popolazione target a cui l'applicazione mobile si rivolge e il contesto d'uso della stessa. Si noti che l'applicazione è l'unica parte del sistema con cui l'utente deve interagire e per questo motivo la sua progettazione deve rispettare i vincoli imposti dalla UX Design. La fascia d'età degli utenti è individuabile nel range 18 - 50 anni e l'attenzione che l'utente può dedicare all'applicazione sarà pressoché scarsa essendo in procinto di iniziare a guidare. Sulla base di questa premessa il flusso logico dell'applicazione è stato studiato per essere semplice, intuitivo e il più corto possibile.

Altresì si è ritenuto di individuare come dispositivi target i soli smartphone aventi display con diagonale inferiore ai 6.0 in poiché è altamente improbabile che un dispositivo con display maggiore sia usato in un contesto in movimento. Per tale motivo si è ritenuto non utile sviluppare dei layout che si adattassero a dispositivi con schermi xlarge [1].

3.3 Target system

Tra i vari sistemi operativi per dispositivi mobile ho scelto Android. L'applicazione è adattabile (previa riscrittura) per gli altri sistemi ma ciò esula dello scopo di questa tesi. L'applicazione è stata sviluppata per device aventi come sistema operativo almeno Android Marshmallow (minSdk API 23). Tale restrizione è dovuta al fatto che le API per il Fingerprint lato sistema operativo (e conseguente mancanza dell'hardware lato device fisico), sono state introdotte solo a partire da Android Marshmallow. Venendo meno la necessità di utilizzare API retrocompatibili con versioni del sistema operativo inferiori a quella target è stato possibile sfruttare in modo efficace le nuove features introdotte quali l'utilizzo del *KeyStore* con chiavi crittografiche simmetriche (precedentemente utilizzabile solo con chiavi crittografiche asimmetriche (da API 18)) e la funzione *getColor(...)*.

3.4 Schema generale

Di seguito è riportato uno schema riassuntivo con le principali classi relative all'applicazione.

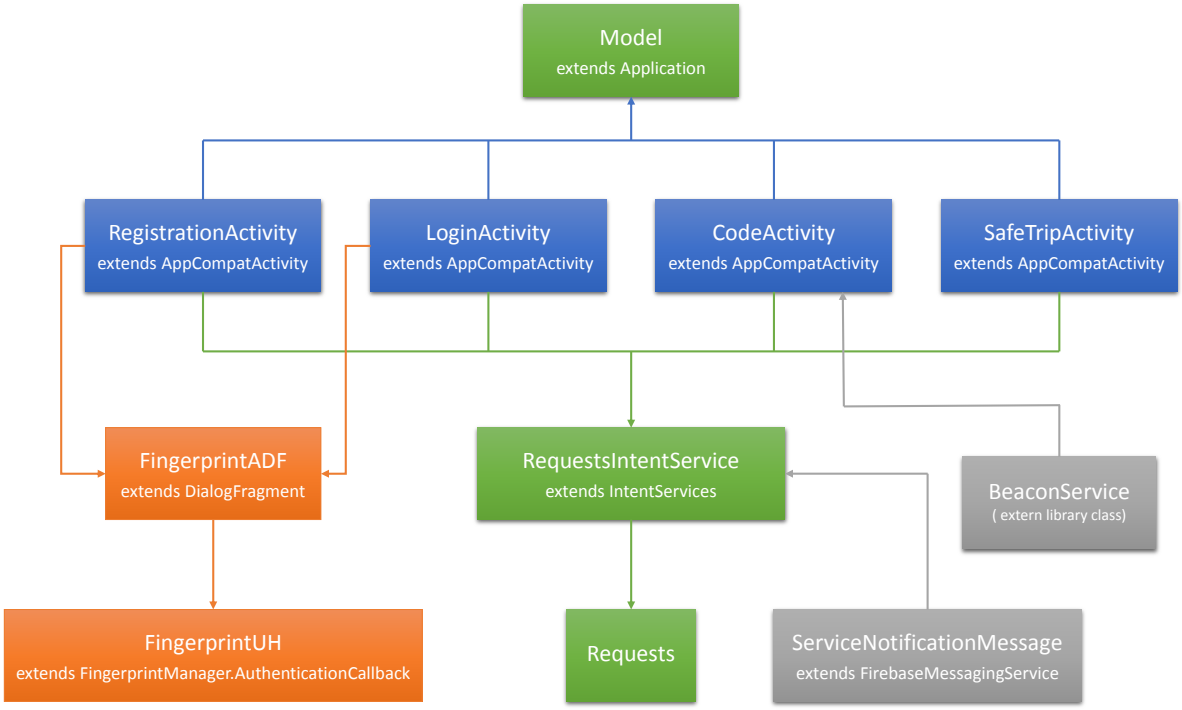


Figura 3.1. Classi principali

3.4.1 Model

La classe *Model* estende la classe Android *Applicaton* che rappresenta il processo lanciato dal sistema operativo.È la prima classe ad essere instanziata dal s.o. e a cui sono legate tutte le componenti come ad esempio i *Service* o le *Activity* ed è raggiungibile da ognuna di esse. È un singleton. Per la sua centralità è atta a svolgere funzioni comuni e a fare da *modello* come suggerisce il nome ovvero come contenitore dei dati. Infatti essa è incaricata di creare e successivamente reperire il file contenente le *SharedPreferences* condiviso a livello di applicazione e di fornirne il riferimento alle Activity che lo chiedono. In tale file sono salvati la password cifrata per accedere al server, l'IV cifrato, lo username dell'utente corrente e un flag che discrimina se l'utente vuole utilizzare il Fingerprint oppure no.

Altresì ha il compito di creare e gestire gli oggetti Cipher e le SecretKey. Per far ciò utilizza l'implementazione Android del *KeyStore*. Tale oggetto è deputato alla conservazioni di chiavi crittografiche rendendole non esportabili all'esterno del device con la possibilità di limitarne l'uso solo per specifici contesti. Salvando nelle *SharedPreferences* sia la password per il server che l'IV cifrati sono necessarie due chiavi crittografiche che sono reperibili dal KeyStore mediante due alias : *mySecretAliasForKey* e *mySecretAliasForKeyPadding* . Tali chiavi sono create quando un utente si registra o effettua il login per la prima volta e distrutte nel momento del logout. Quindi ad ogni ciclo di login/logout le password sono rigenerate rendendole differenti per ogni utente. Tramite l'oggetto KeyGenerator (implicitamente collegato durante la sua creazione al KeyStore) si generano le due sopracitate chiavi crittografiche, quella per la password è creata per essere usata con l'algoritmo AES-256/CBC/PKCS7 mentre quella per l'IV, essendo esso esattamente di 128 bit è creata per l'algoritmo AES-128/ECB/PaddingNone (il solo caso in cui può essere usato perché deve cifrare esattamente un blocco di 128 bit).

Provvede inoltre a registrare l'applicazione per il topic 'maps' all'interno del servizio FirebaseMessaging per ricevere le notifiche push quando uno o più veicoli diventano liberi per mostrarli sulla mappa. Infine è la classe che carica il certificato x509 self-signed del server e fornisce alla classe Requests una implementazione dell'*SSLSocketFactory* e dell'*HostnameVerifier*. La prima è una configurazione del livello di comunicazione SSL che consente di accettare il certificato self-signed del server e la seconda invece consente di aggirare il problema derivante dal fatto che l'hostname dichiarato nel cerificato x509 è differente da quello estratto dalla connessione in quanto consiste nell'indirizzo IP del server invece che nell'URL.

3.4.2 Requests

La classe *Requests* è un singleton che si occupa di inviare le richieste HTTPS al server gestendo la parte di comunicazione, invia e riceve dati in formato JSON ad un Url specificato da chi chiama i suoi metodi. È trasparente alle interfacce REST del server, ignora la

loro esistenza. Gestisce inoltre il timeout delle richieste (impostato a 30 s) ed errori quali network unreachable e malformed JSON. Tali errori sono comunicati a chi ha invocato la funzione tramite un JSON contente il campo *error*.

3.4.3 RequestIntentService

La classe *RequestIntentService* estende la classe *IntentService* a sua volta estensione della classe Service le cui operazioni sono eseguite in un thread secondario e quando non ci sono più richieste pendenti termina automaticamente. Fa parte della famiglia degli Started Service, gli intent che riceve sono inviati alla funzione *onHandleIntent(...)* che esegue le azioni richieste. Il risultato dell'operazione è inviato tramite un intent in broadcast a tutti i componenti che hanno registrato un BroadcastReceiver per la risposta a quella specifica azione. La risposta è inviata tramite l'istanza del *LocalBroadcastManager*. Le richieste che possono essere fatte a tale classe sono :

ACTION_REGISTER Per effettuare la registrazione al server. Invia username e password. La risposta è un JSON con campo *registered* che può avere due valori : true o false e il codice segreto per la modifica della password.

ACTION_LOGIN Per autenticarsi presso il server. Invia username e password. La risposta è un JSON con campo *loggedin* che può avere due valori : true o false

ACTION_CODE Per comunicare al server la scelta del veicolo, la posizione GPS dell'utente e la coppia id-distanza relativa al beacon bluetooth più vicino allo smartphone.

ACTION_RECOVER Per modificare la password sul server, invia un codice segreto univoco per ogni utente e cifrato lato server, lo username e la nuova password.

ACTION_CARS Per ricevere dal server l'elenco dei veicoli liberi e la loro posizione GPS.

ACTION_CARS_ASYNC Emula la richiesta al server per l'elenco dei veicoli ridirigendo in broadcast i dati ricevuti come notifica push da Firebase.

3.4.4 ServiceNotificationMessage

La classe *ServiceNotificationMessage* estende la classe FirebaseMessagingService. Essa è in collegamento con il servizio cloud di FirebaseMessaging che Google mette a disposizione per inviare notifiche push ai dispositivi scelti. In particolare è stato necessario creare un progetto Firebase per consentire all'applicazione di ricevere gli aggiornamenti in tempo reale sulle posizioni delle auto, ad esempio quando una di esse diventa libera. Tramite il *Model* l'applicazione in fase di avvio si era precedentemente registrata per ricevere le notifiche relative al topic 'maps' e tramite la funzione *onMessageReceived(...)* provvede ad inviare una richiesta al RequestIntentService per mandare in broadcast i dati ricevuti.

3.5 Fasi principali

3.5.1 Registrazione

La procedura di registrazione dell'utente è affidata alla **RegistrationActivity**. Essa è definita nel *Manifest* dell'applicazione come *activity launcher*, è la prima ad essere creata e lanciata quando si clicca sull'icona. Se l'utente è già registrato (il suo username è presente nelle *SharedPreferences*) si termina la RegistrationActivity e si lancia l'activity di login rappresentata della classe **LoginActivity**. L'utente inserisce un username a scelta (se non è disponibile gli sarà notificato) e cliccando sul FAB (FloatingActionButton) apparirà un dialog per scegliere la password. A questo punto viene effettuata una richiesta al server tramite il *RequestIntentService* . In fondo a sinistra è presente uno switch che consente all'utente di scegliere se vuole registrarsi oppure effettuare un login tramite un account già creato. La procedura è la stessa di quella già descritta.

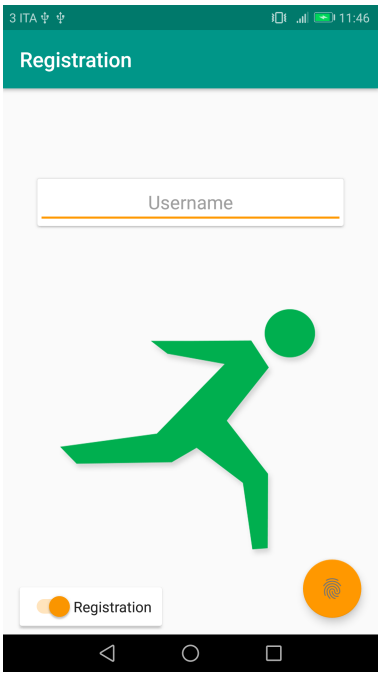


Figura 3.2. Layout dell'activity di registrazione

Il dialog sopra citato è un'istanza della classe **FingerprintADF** a cui viene passato un oggetto di tipo FingerprintManager.CryptoObject creato a partire dall'oggetto Cipher relativo all'utente corrente che è preso direttamente dal Model. Si inserisce la password e si sceglie tramite una *CheckBox* se si vuole salvare la password sul telefono e sbloccarla tramite Fingerprint oppure inserirla manualmante ogni volta. Nel primo caso il dialog modifica il suo layout e attiva un listner associato all'hardware del Fingerprint. Se viene rilevata un'impronta valida registrata a livello di sistema la password è cifrata con AES-256-CBC. Il Cipher usato per tale operazione è stato generato dal Model, sottoposto ad autorizzazione tramite Fingerprint per l'utilizzo ed è stato salvato nel KeyStore del device. La password cifrata è salvata invece nelle SharedPreferences. Allo stesso modo è trattato l'IV utilizzato per la cifratura in modalità CBC della password. È salvato cifrato, con un altro oggetto Cipher non sottoposto però all'autorizzazione dell'utente, con AES-128-ECB (perché il blocco cifrato, l'IV è per sua definizione uguale alle dimensioni del blocco base dell'algoritmo, non ne pregiudica quindi la sicurezza) nelle SharedPreferences. La password è inviata al server insieme all'username codificato come JSON tramite il RequestIntentService (ACTION_REGISTER). Nella RegistrationActivity sono altresì presenti tre *BroadcastReceiver* in ascolto sulle risposte del RequestIntentService, che possono essere la notifica di un errore, il risultato del login o il risutato della registrazione. In

caso di errore questi è notificato all'utente tramite Toast se è generico oppure impostando il tipo dell'errore sulla EditText dell'username (es. invalid username).

Se invece la risposta dal server è del tipo "loggedin" *true* o "registered" *true* allora si termina l'activity corrente e si lancia LoginActivity.

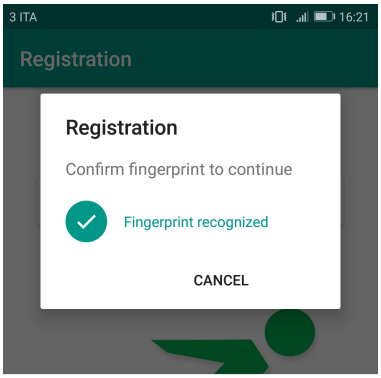


Figura 3.3. Dialog per riconoscimento del Fingerprint

3.5.2 Login

Una volta che l'utente è registrato, ad ogni avvio dell'applicazione la RegistrationActivity provvede a lanciare subito dopo la sua creazione la **LoginActivity**. Tale classe ha due compiti : innanzitutto deve mostrare una mappa con i veicoli liberi vicini all'utente sfruttando le sue coordinate geografiche, inoltre deve autenticare l'utente presso il server. Il layout è simile alla RegistrationActivity. In alto è presente una card che mostra l'username dell'utente corrente, una mappa a tutto schermo mostra la zona limitrofa all'utente e in fondo a destra è presente un FAB per aprire lo stesso dialog della RegistrationActivity. Questa volta però appare direttamente con l'indicazione di procedere all'autenticazione con il fingerprint oppure se l'utente aveva scelto di non utilizzarlo mostra una EditText per inserire la password. Nel primo caso dopo l'autenticazione con fingerprint in locale si ha la decifratura della password e questa è inviata al server tramite richiesta alla classe RequestIntentService (ACTION_LOGIN) altrimenti è inviata direttamente quella inserita manualmente. La risposta a tale richiesta è intercettata da uno dei due BroadcastReceiver instanziati da questa classe. Si può avere il JSON con il campo errore oppure il risultato dell'autenticazione che può assumere il valore *true* o *false* nel campo "loggedin" del JSON ritornato. Se l'autenticazione è andata a buon fine allora viene lanciata la *CodeActivity*.

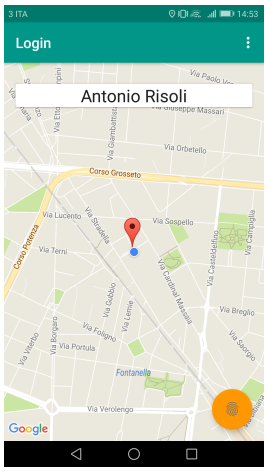


Figura 3.4. Layout dell'activity di login

La mappa presente nel layout è un'istanza del fragment *SupportMapFragment* che sfrutta le Google Map's API 2.0. Per utilizzare tali API è stato necessario creare un apposito progetto Google che tramite un token gestito automaticamente dalla classe del fragment è in grado di effettuare le richieste al server di Google e scaricare le informazioni necessarie.

Lato codice invece la parte grafica della mappa è gestita tramite l'oggetto *GoogleMap* che referenzia la mappa contenuta nel fragment nell'istante in

cui la fase di caricamento della stessa è finita (è referenziata in modo asincrono rispetto all'esecuzione dei metodi dell'activity che la ospita). Tramite questo oggetto è possibile inserire dei marker. La posizione e il contenuto dei marker è reperito tramite la classe RequestIntentService con la richiesta di (ACTION_CARS) che viene esplicitamente invocata dalla Activity nel suo metodo *onResume(...)*. Per ricevere la risposta sempre nell'*onResume(...)* è instanziato un'altro BroadcastReceiver che, una volta che il dato è pronto, parsifica il JSON ricevuto e per ogni veicolo estrae : latitudine, longitudine e targa. Con tali informazioni instanzia un oggetto di tipo *MarkerOptions* e lo inietta nella mappa. Tale BroadcastReceiver inoltre può essere attivato anche in modo asincrono (cioè non per diretta conseguenza dell'invocazione della ACTION_CARS) in quanto il service ServiceNotificationMessage può ricevere una notifica push da Firebase perché almeno una posizione dei veicoli della flotta è cambiata (es. si è liberato un veicolo) e chiedere al RequestIntentService di inoltrare i dati ricevuti come se fossero stati scaricati direttamente dal server. In questo modo si è evitato di fare un polling continuo al server per sapere se vi erano nuovi veicoli disponibili.

Nell'AppBar invece è presente un menù che consente all'utente, cliccando sui tre puntini verticali, di effettuare il Logout. Questa operazione consiste nel chiedere al *Model* di eliminare i dati relativi all'utente corrente ovvero quelli salvati nelle SharedPreferences quindi username, password cifrata, IV e preferenze e nello stesso tempo di eliminare la KeyStore di Android le due cipher-suite salvate, quella per la cifratura della password e quella per la cifratura dell'IV.

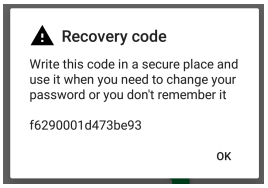


Figura 3.5. Alert con indicazione del codice per cambiare la password.

Se questa activity è lanciata per il risultato positivo di una registrazione allora in questo caso la *LoginActivity* mostra un alert con il quale indica all'utente il codice che potrà essere utilizzato per modificare la password sul server nel caso l'utente la dimentichi. Questo codice è univoco per ciascun utente.

3.5.3 Selezione veicolo

L'ultima fase è la selezione del veicolo da parte dell'utente con conseguente sbloccaggio e configurazione da parte del server. La scelta del veicolo è possibile tramite la **CodeActivity** e il codice QR univoco presente sul parabrezza dello stesso. Il layout dell'activity cambia in base a quali azioni sono state compiute nel processo di selezione. Inizialmente al centro del layout l'activity mostra la preview della fotocamera principale. Tale preview è proiettata all'interno di un FrameLayout i cui figli sono una *SurfaceView* (che contiene effittivamente i bit della preview) e una custom View (*MyView*) trasparente al centro e con i bordi colorati e arrotondati. Grazie al FrameLayout sono combinati in base alle trasparenze e il risultato è mostrato nella figura a lato [3.6].

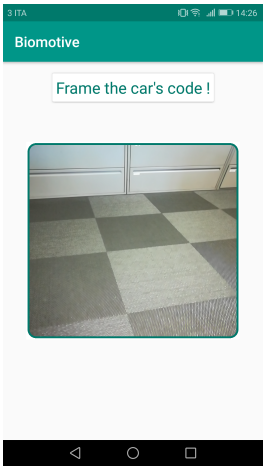


Figura 3.6. Layout dell'activity per la selezione del veicolo

L'hardware della fotocamera è sfruttato tramite la classe CameraSource del package *com.google.android.gms* che offre una serie di soluzioni semplici per problemi comuni. In particolare CameraSource si occupa di scattare delle preview tramite la fotocamera e di inviarle ad un *Detector* (specializzato in questo caso nel riconoscere codici QR e EAN_13) cercando di minimizzare i lag nel mostrare su schermo le preview e fornendo al Detector il numero massimo di preview che riesce a processare nell'unità di tempo. Il detector nel momento in cui rileva uno o più codici QR lo notifica all'activity tramite la callback *receiveDetections(Detector.Detections<Barcode> detections)* [25]. A questo punto l'hardware della fotocamera viene rilasciato così come il Detector e le risorse annesse alla SurfaceView. Viene dunque reso non visibile il FrameLayout e al suo posto viene mostrata una TextView con il codice rilevato.

Contestualmente per avere la certezza che l'utente stia cercando di aprire un veicolo che è nelle sue vicinanze viene inviata al server insieme al codice del veicolo anche la sua posizione GPS. La posizione è rilevata tramite la classe Android *LocationManager* sfruttando sia il sensore GPS che la posizione rilevata tramite la rete internet. Al LocationManager è chiesto di ricevere un aggiornamento sulla posizione una volta al secondo. Al ricevimento della prima posizione il listner del LocationManager è disabilitato per risparmiare energia. L'activity mostra a uno SnackBar con la posizione rilevata (città, strada, civico).

Ad ogni veicolo è associato un Beacon. È un dispositivo a basso consumo energetico capace di emettere un segnale radio compatibile con lo standard Bluetooth 4.0 Low Energy, ognuno di essi ha uno specifico e univoco numero seriale. A partire dalla potenza del

segnale ricevuto, conoscendo tale potenza alla distanza di $0m$ è possibile calcolarne la distanza dal punto di ricezione dello stesso. A riprova che l'utente è nelle vicinanze del veicolo tramite il bluetooth e la libreria esterna Android Beacon Library[24] vengono rilevati i beacon in zona e da quello che risulta a minore distanza ne vengono estratti i tre Id (major, minor e sequential) che lo identificano e la distanza dall'utente. Quando tutti e tre i blocchi di informazioni sono stati individuati (posizione GPS, beacon più vicino e targa) si invia al server il codice QR , la posizione (latitudine, longitudine e incertezza in m) [28, 2] e le informazioni sul beacon (l'id del beacon construito come stringa unica a partire dalle 3 componenti Id1, Id2 e Id3 unite mediante il carattere ASCII ':' e la distanza dall'utente).

Il server riconosce che l'utente si era appena autenticato grazie al token inviato tramite un cookie all'applicazione attraverso la richiesta precedente. Il token ha la validità di un minuto. A questo punto il server procede a verificare che il veicolo non sia attualmente in uso, che la posizione GPS del veicolo e quella dell'utente siano compatibili, che l'Id ricevuto sia effettivamente associato al veicolo e che la distanza rilevata sia inferiore ai $3m$. Se tutte le condizioni sono soddisfatte il server inizia la procedura di sblocco e configurazione del veicolo. Tale procedura può rihiedere alcuni secondi e per dare un feedback all'utente che la richiesta è stata inoltrata viene aggiunta una ProgressBar circolare [3.7]. Il server appena ha sbloccato il veicolo conferma all'applicazione l'operazione e se tutto si è svolto ordinatamente la CodeActivity notifica all'utente il successo della procedura avviando la SafeTripActivity. In caso contrario tramite Toast si notifica all'utente che il veicolo non è disponibile non specificando all'utente quale delle precedenti condizioni non è stata soddisfatta.

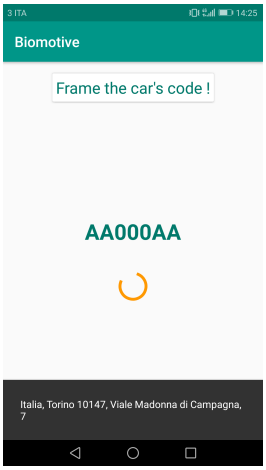


Figura 3.7. Layout dell'activity in attesa dello sblocco dell'auto

3.5.4 Recupero Password

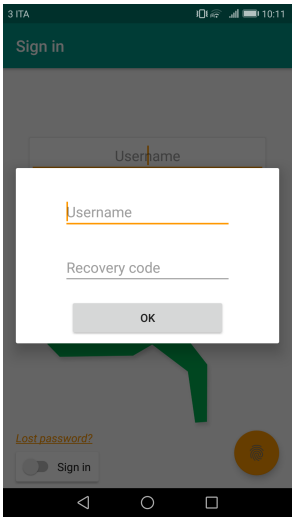


Figura 3.8. Alert per cambiare la password.

Durante la registrazione l'utente ha ricevuto un *Recovery Code*, una stringa alfanumerica di 16 caratteri random generati dal server. Se l'utente dimentica la password, dalla schermata di registrazione, tramite lo switch può passare alla modalità Login. In questa modlità in basso a sinistra appare un link che se cliccato apre il dialog mostrato in figura [3.8]. L'utente inserisce il suo username e il *Recovery Code* . A questo punto potrà scegliere la nuova password che vorrà utilizzare e la richiesta sarà inoltrata al server. Se i dati inseriti sono validi il server risponderà alla richiesta con un 200 ok allora sarà lanciata la *LoginActivity*. In caso contrario invece si rimane sulla *RegistrationActivity* e si avvisa l'utente usando la funzione setError sulla *EditText* relativa all'username.

Requisiti di sicurezza

Di seguito sono riportate le proprietà di sicurezza relative alla Mobile App che sono state prese come linee guida per lo sviluppo della stessa:

- Dati at rest
  - Confidenzialità della password
- Dati in motion
  - Confidenzialità dei dati trasmessi
  - Integrità dei dati trasmessi
- Autenticazione delle parti
- Autenticazione biometrica in locale

Le proprità appena elencate sono state ottenute tramite l'utilizzo del protocollo TLS e della cifratura in locale.

Capitolo 4

Server

4.1 Introduzione

Il Server ha due scopi, da un lato è colui che mette in comunicazione l'utente (rappresentato dall'applicazione per smartphone) e il veicolo (rappresentato dalla *TBox*) e dall'altro è il componente in cui è concentrata la parte principale della logica dell'intero sistema. Infatti è il Server che abilita la *TBox* a bloccare/sbloccare il veicolo, che provvede a far si che solo un utente alla volta possa sbloccare un veicolo, che tiene traccia della posizione dei veicoli, che salva e comunica alla *TBox* le configurazioni. Le comunicazioni con la *TBox* avvengono tramite canali socket stream racchiusi in sessioni TLS/SSL. Dal lato dell'utente invece offre , attraverso delle API REST su protocollo HTTPS, la possibilità di effettuare la registrazione, il login, di visualizzare i veicoli disponibili, di scegliere un veicolo.

Inizialmente, nel primo mese della tesi, si era pensato di utilizzare per lo sviluppo di questo componente il linguaggio C++11 ma dopo accurate ricerche su librerie che consentissero di construire un server basato su HTTPS si è convenuto che i benefici in termini di tempi di esecuzione del prodotto finale erano irrisori rispetto al tempo per lo sviluppo se confrontato con l'utilizzo di un linguaggio più immediato e conciso come Python. Infatti pur restando vero che il tempo di esecuzione di uno script in Python è superiore a quello di un programma in C++ in differenti contesti, con l'attuale potenza di calcolo i due tempi sono trascurabili rispetto alle necessità di reattività del sistema e altresì il beneficio in termini di semplicità e velocità nella programmazione è innegabile.

La versione di Python utilizzata per lo sviluppo è la 2.7.12. Ho utilizzato un PC con 8 GB di Ram e processore Intel Core i7-3610QM. Come IDE invece ho usato PyCharm 2017.1.x Community Edition.

4.1.1 Schema generale

Di seguito è riportato uno schema riassuntivo con le principali classi relative all'applicazione.

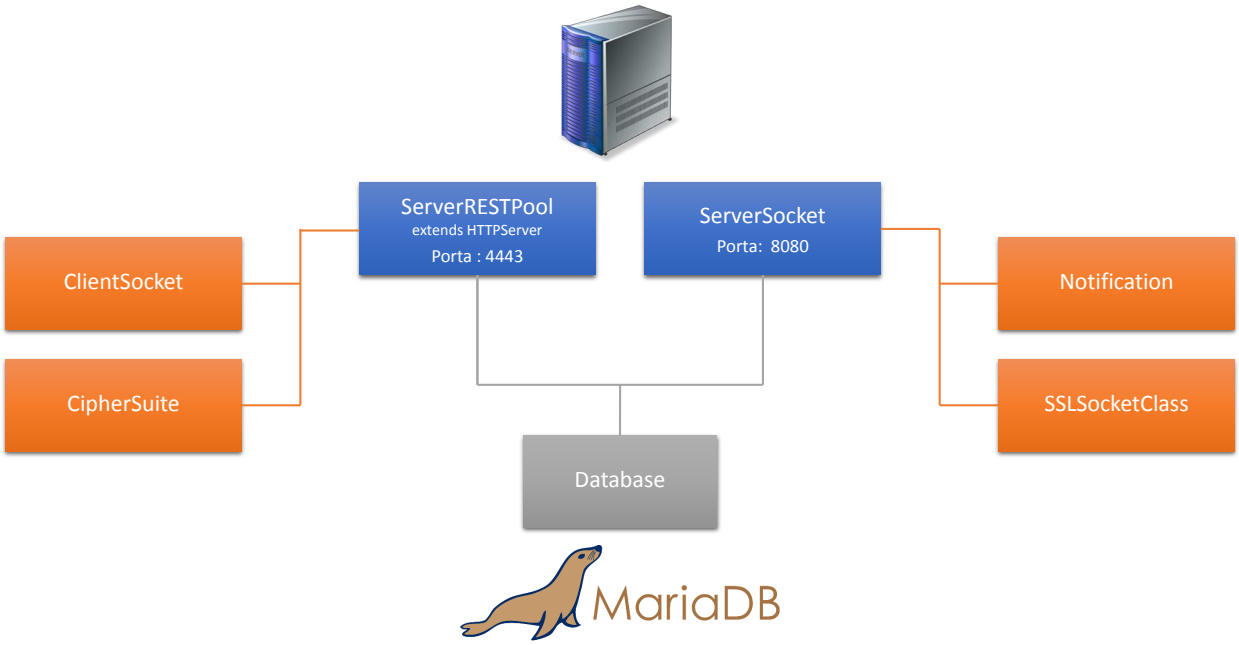


Figura 4.1. Classi principali

Il server è la concatenazione e l'interazione delle classi elencate nella figura. Per avviare ordinatamente il flusso logico di esecuzione, nesessitando di due processi paralleli uno per il ServerRESTPool e l'altro per il ServerSocket ho creato uno script Python addetto a questa funzione. Tale script chiamato *startAll.py* innanzitutto imposta come codifica predefinita delle stringhe l'UTF-8 tramite il seguente codice:

```
reload(sys)
sys.setdefaultencoding("utf-8")
```

A questo punto prende il *pid* del processo corrente ed esegue una *fork()*. Il processo figlio avvierà il ServerRESTPool mentre il padre avvierà il SeverSocket.

4.2 MyDatabase

La classe MyDatabase è l'astrazione dell'interazione con il database ovvero offre dei metodi semplici al livello applicativo (ServerRESTPool, ServerSocket, ClientSocket) occupandosi di dialogare con il database e di intercettare eventuali eccezioni. Il database utilizzato è MariaDB versione 10.0.29-MariaDB-0ubuntu0.16.04.1.

Tutti i parametri ricevuti dalle funzioni di questa classe sono prima convertiti nel tipo di cui si suppone debbano essere e poi ne viene fatto l'escape con la funzione mysql.connector.escape al fine di evitare attacchi di Sql Injection.

4.2.1 Il costruttore della classe

Il costruttore della classe tramite l'oggetto mysql.connector crea l'oggetto *mariadb_connection*, una connessione con il database. Da quest'ultimo vengono derivati un cursor e un prepared cursor. Tramite il cursor si esegue immediatamente

```
self.cursor.execute("SET SESSION TRANSACTION ISOLATION LEVEL
    SERIALIZABLE")
```

Con questo livello si ha il massimo isolamento. I dati condivisi utilizzati dai vari thread (uno per richiesta HTTP) sono salvati nel database ed essendo accessibili solo tramite questa classe che serializza le richieste, l'accesso ai dati è sincronizzato in modo automatico. È il database che si occupa della sincronizzazione.

Nella seconda parte del costruttore si definiscono tutte le query parametrizzate per essere usate con il prepared cursor (ogni thread ha una istanza privata di questa classe e quindi un prepared cursor privato).

4.2.2 Gestione utente

La creazione dell'account utente ovvero il suo inserimento nel database avviene tramite la funzione *insertUser(username, password, recoverycode)* che provvede a salvare i parametri ricevuti nel database andando a cifrare il recoverycode con la chiave pubblica del

server stesso e la password è invece salvata hashata (hmac_sha256) con 28 byte di padding random (salt). In questo modo si evitano gli attacchi hacker Dictionary Attack e Rainbow Table. La tupla salvata è costituita da : username, password_hash, salt, recoverycode_enc. La funzione execute() del prepared cursor nel caso in cui l'username (usato come chiave primaria della tabella sql User) già esista lancia un'eccezione e la *insertUser* fa rollback e ritorna *False*.

La *validateUser(username, password)* invece controlla che la password ricevuta per l'username passato come parametro sia valida : estrae la password_hash per l'utente e il salt, effettua l'hmac con la password ricevuta e il salt estratto e la confronta con la password_hash estratta. Se sono uguali ritorna *True*, altrimenti ritorna *False*.

La *changePasswordself (username, newPsw, recoverycode)* consente di cambiare la password per l'username passato come parametro. Esegue una query per recuperare il recoverycode_enc relativo all'username, lo decifra con la chiave privata e lo contronta con quello passato. Se sono diversi ritorna *False*, altrimenti esegue un update sulla tabella User.

4.2.3 Gestione veicolo

Un veicolo della flotta è rappresentato all'interno del database come un oggetto avente la targa come identificativo e una serie di attributi : l'IP della *TBox*, un *int* che rappresenta se in questo istante è usato da un utente oppure no, l'ultima posizione del veicolo (latitudine, longitudine ed errore in metri) e codice 'k' del beacon associato al veicolo.

La *getCarsPosition()* serve per creare un dizionario con la posizione dei soli veicoli liberi libero nel momento in cui la query è eseguita. Per far ciò, per ogni riga valida del cursore che contiene i risultati della query si provvede a creare un dizionario con le seguenti chiavi :

```
x["id"]  = targa
x["lat"] = lat
x["lon"] = lon
```

Questo dizionario è inserito nel dizionario più grande che mappa tale oggetto su una chiave progressiva. Al dizionario generale infine, si aggiunge la chiave 'len' che indica il numero di oggetti contenuti. Si sarebbe potuto utilizzare una lista di dizionari ma tale implementazione ha semplificato la scrittura del codice in quanto la funzione json.dumps, utilizzata per creare il json da ritornare alla Mobile App, riceve una mappa e non una lista.

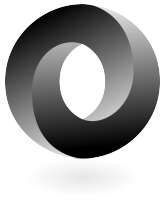


Figura 4.2. Json logo

La funzione *updateCarInfo()* serve per aggiornare le informazioni relative al veicolo (IP della TBox, latitudine, longitudine, errore in metri sulla posizione). In particolare tale funzione è invocata quando la TBox del veicolo comunica con il server informandolo che il suo IP o la sua posizione è cambiata.

La funzione *setCarUsed(isUsed, car, user)* permette al server di modificare lo stato del veicolo (da usato a libero e viceversa) e di monitorare il tempo di utilizzo del veicolo da parte dell'utente per una eventuale consuntivazione futura. Altresì raccogliendo tali dati permette di sapere chi è stato l'ultimo utente ad utilizzare il veicolo.

Le funzioni *getIpForCar(car)* e *getGeoForCar(car)* come dice il nome stesso ritornano l'IP della *TBox* del veicolo e la posizione corrente del veicolo. Entrambe sono valide se e solo se il veicolo non è attualmente utilizzato altrimenti è ritornato il valore *None*.

4.2.4 Gestione configurazioni del veicolo

Quando l'utente si autentica sul server e chiede di sbloccare il veicolo, il server estrae le configurazioni dell'utente corrente dal database e le invia alla *TBox* quando inizia la comunicazione con essa. Quando invece l'utente spegne il veicolo tramite la chiave è la *TBox* a comunicare al server le configurazioni attuali che vengono salvate nel database.

In potenza tali configurazioni possono essere la posizione verticale del volante, la posizione dei sedili, la loro inclinazione, la temperatura dell'aria condizionata, la stazione radio preferita, il volume della radio ecc...

Ovviamente suddette configurazioni richiedono un veicolo di fascia alta. Ai fini della tesi e quindi per la realizzazione del POC, ho limitato le configurazioni alla personalizzazione dell'infotainment. In particolare configuro la frequenza FM e l'on/off dei canali destro e sinistro dell'audio. Gli infotainment presenti in azienda, non essendo di fascia alta, non consentono il pieno controllo tramite CAN bus. Ho dovuto quindi simulare un infotainment che potesse essere pienamente controllabile tramite CAN bus, ovvero che trasmetta le sue impostazioni attuali e che sia in grado di modificarle alla ricezione di messaggi CAN specifici.

Le funzioni *getConfiguration(username)* e *updateConfiguration(username)* servono a recuperare le configurazioni per l'utente corrente e a salvare quelle attive nel momento dello spegnimento del veicolo da parte dell'utente.

L'infotainment realizzato è descritto nel Capitolo 6

4.3 ServerRESTPool

La classe *ServerRESTPool*, come suggerisce il nome, offre delle API REST all'applicazione per smartphone. Questo file include la classe *HttpsPoolThreadServer* che estende

sia la classe HTTPServer che la classe PooledProcessMixIn[18]. La prima semplifica la creazione di un server HTTP perché permette di registrare un Handler della classe BaseHTTPRequestHandler che in modo automatico intercetta le richieste HTTP e le divide per verbi HTTP (get, post, put, delete). Facendo l'override dei suoi metodi es. *do_GET* per il metodo get possiamo eleborare le richieste (si riceve il path completo, va quindi parsificato per sapere a cosa la richiesta sta facendo riferimento). La seconda invece permette di creare un server multi-process e multi-thread in modo da poter servire più client nello stesso momento. Tale classe è stata realizzata da Muayyad Saleh Alsadi sotto licenza PSLF [19].

4.3.1 Il costruttore della classe

Nel costruttore della classe HttpsPoolThreadServer prima di tutto si inizializzano i parametri per il numero di processi e thread. Per le funzionalità che il server in generale deve espletare ho ritenuto opportuno di creare $n - 1$ processi dove n è il numero di core della CPU e $2 * (n - 1)$ thread per ciascun processo invece dei 64 thread per ogni processo che sono creati di default. Sulla piattaforma di sviluppo sono dunque sganciati $7 * (2 * 7)$ thread ovvero 98 thread.

Poiché è necessario proteggere la comunicazione da eventuali attacchi informatici volti ad intercettare le credenziali di un generico utente sia in fase di registrazione che di login, nonchè per garantire la confidenzialità del veicolo che l'utente sta per usare e dello storico dei suoi viaggi si è effettuato un avvolgimento del socket (wrapping) in un contesto SSL. Per far ciò ho utilizzato la classe Python *ssl* dalla quale è possibile creare un *context* e configurarlo con i parametri di sicurezza necessari:

```
ctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
ctx.load_default_certs()
ctx.set_ciphers('TLSv1.2:!aNULL:!eNULL')
ctx.set_ecdh_curve('secp384r1')
ctx.load_cert_chain(certfile=pathCert, keyfile=pathKey)
ctx.options |= ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3 | ssl.OP_NO_TLSv1
                        | ssl.OP_NO_TLSv1_1 | ssl.OP_SINGLE_ECDH_USE
                        |ssl.OP_CIPHER_SERVER_PREFERENCE | ssl.
                            OP_NO_COMPRESSION

self.socket = ctx.wrap_socket(self.socket, server_side=True)
self._init_pool()
```

Per prima cosa dopo aver creato il *context* si caricano i certificati x509 di default del sistema, poi si scelgono come ciphersuite quelle che sono compatibili con la versione 1.2 di TLS dove gli algoritmi di autenticazione e cifratura devono essere entrambi diversi da *null*. Come curva per le eventuali operazioni di crittografia su curve ellittiche (in particolare per EC Diffie-Helmann) si sceglie la *secp384r1* consigliata dall'RFC 6460 [22] che riprende le direttive della NSA Suite B Cryptograph [26]. Si disabilitano tutti i

protocolli precedenti alla versione TLS 1.2 e la compressione dei dati in modo da evitare attacchi quali POODLE, BEAST, Heartbeat Extension, CRIME ecc... Si noti inoltre che si usa l'ordine delle preferenze del Server per le ciphersuite invece che l'ordine del client, in questo modo da prove effettuate tramite il debugger di Android Studio per la cifratura si è passati dalla ciphersuite TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 alla ciphersuite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384. Ovviamente ciò va a discapito della velocità della comunicazione ma si guadagna sensibilmente in termini di sicurezza passando dal livello *secret* a quello *topsecret* della classificazione degli algoritmi di sicurezza da parte dell'NSA [26]. A questo punto si fa il wrapping del socket corrente nel contesto SSL e il risultato è un socket che ha un livello intermedio tra il TCP e l'HTTP ovvero il livello SSL. Infine si avviano i processi e i rispettivi thread con la *_init_pool()* .

4.3.2 UserHandler

La classe *UserHandler* ha essenzialmente due funzioni principali *do_GET* e *do_POST*. Entrambe splittano la variabile di classe *path* rispetto al carattere '/', eliminano la prima posizione che consta di una stringa vuota e cercano la risorsa corrispondente all'indice zero del nuovo vettore (*path[0]*). Nel caso in cui la risorsa non esista ritornano nella risposta HTTP uno status code generico, 400 (client error) per la GET e 401 (non autorizzato) per la POST. La *do_POST* inoltre provvede a controllare l'esistenza del cookie con nome 'token' all'interno dell'header della richiesta. Se il valore associato alla chiave 'token' è un codice valido all'interno del database (esiste e non è scaduto al momento del controllo) allora la richiesta può essere inoltrata verso le funzioni che gestiscono i path che richiedono l'autentcazione preventiva dell'utente. In questo caso vi è solo un path */code* che richiede l'autenticazione preventiva ma tale meccanismo è stato sviluppato per consentire di avere una sessione a livello del protocollo HTTP, che la libreria HttpServer attualmente non implementa, per un eventuale sviluppo futuro di altre risorse REST. Si noti che in questo modo non c'è bisogno di decifrare e validare la password dell'utente ad ogni richiesta ma si fa solo una query.

Registrazione

La fase di registrazione è mappata sul path */register*, metodo POST. La Mobile App inserisce come body della richiesta HTTPS un Json con il seguente formato :

```
{ "username":string, "password":string }
```

che viene analizzato dalla funzione *doRegister()*. Tale funzione valida il formato del Json e prova ad eseguire una *insert* nel database MariaDb del server attraverso l'istanza globale della classe *MyDatabase*, con la funzione *insertUser(username,password,recoverycode)*. Se l'inserimento va a buon fine allora tale funzione ritorna *True* e la risposta alla richiesta HTTPS avrà codice di risposta 201 e il body conterrà il seguente Json :

```
{'registered': "true","recovery":r}
```

dove la variabile *r* è una stringa di 8 byte random che può essere utilizzata dall'utente nel caso dimenticasse la password o la volesse cambiare, un codice che l'utente deve conservare con cura e non farlo conoscere a nessuno. Tale codice sarà inoltre salvato all'interno del DB crittografato con la chiave pubblica del server stesso. Se invece l'username scelto esiste già l'inserimento non avviene e la funzione del database chiamata per tale compito ritorna *False*.

Login

La fase di login è mappata sul path */login*, metodo POST. La Mobile App inserisce come body della richiesta HTTPS un Json con il seguente formato :

```
{ "username":string, "password":string }
```

che viene analizzato dalla funzione *doLogin()*. Tale funzione valida il formato della richiesta e tenta di autenticare l'utente rispetto ai dati contenuti nel database attraverso l'istanza globale della classe *MyDatabase*, con la funzione *validateUser(username,password)*. Se l'inserimento va a buon fine allora tale funzione ritorna *True* e la risposta alla richiesta HTTPS avrà status code 200 e il body conterrà il seguente Json :

```
{'loggedin': "true"}
```

Altresì nell'header della risposta viene settato un cookie con nome 'token', attributo 'secure', validità di 60 secondi e che ha per valore una stringa alfanumerica ottenuta con la funzione *getToken(username)* che ritorna l' HMAC-SHA256(username, rand(28)||ctime()) in formato esadecimale. Tale token è salvato inoltre all'interno del database e associato all'utente corrente. Se l'autenticazione fallisce allora la risposta avrà status code 401. Se invece il formato del Json ricevuto è errato allora la risposta avrà status code 400.

Unlock

La richiesta di unlock per un determinato veicolo è mappata sul path */code*, metodo POST. Il Json ricevuto dal server è validato e usato dalla *doUnlock()* ed ha il seguente formato :

```
{ 'code':string,
  'location':{'lat':double, 'lon':double, 'acc':double},
  'idblue':string,
  'bluedist':double
 }
```

Da tale Json per prima cosa è estratta la targa del veicolo ('code') e dal database sono recuperati l'IP della *TBox* a bordo dello stesso e la sua posizione. Se la posizione è *None* ovvero il veicolo è attualmente in uso si ignora la richiesta di unlock e si ritorna lo status code 412 alla Mobile App. Se invece il veicolo non è usato si provvede a calcolare la distanza in metri tra la posizione del veicolo e la posizione dell'utente (estratta dal Json) e si sommano gli errori delle due misure, se la somma degli errori è minore della distanza tra i due o il codice del beacon non corrisponde

a quello del veicolo o la distanza tra l'utente e il beacon è maggiore di $3m$ allora la richiesta è ignorata e si invia alla Mobile App lo status code 412 nella risposta HTTPS.

Se tutte le precedenti condizioni sono false allora si procede ad instanziare e l'oggetto *ClientSocket* 4.4 e si avvia la sua funzione *start()*. Tale funzione inizia la comunicazione con la *TBox* nel thread corrente e se riesce a sbloccare il veicolo immediatamente ritorna *True* non prima di aver avviato un'altro Thread che si occupa del resto della comunicazione con la *TBox* . In questo caso si ritorna alla Mobile App lo status code 200. Nel caso in cui non si riuscisse ad instaurare la comunicazione con il veicolo la funzione ritorna *False* e la *doUnlock* chiude la comunicazione con la Mobile App con lo status code 412.

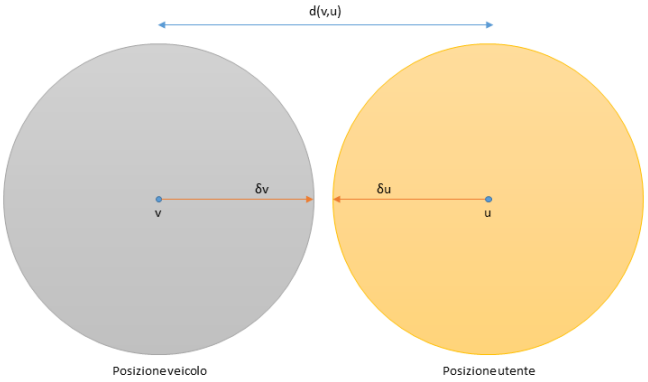


Figura 4.3. Calcolo compatibilità posizioni GPS (in questo caso non compatibili)

Posizione Veicoli

La Mobile App necessita di sapere quali sono i veicoli attualmente liberi e la loro posizione. Tale richiesta è mappata sul path */cars*, metodo GET che non necessita di autenticazione. La funzione invocata è la *doCars()* che estrae la mappa dei veicoli dal database, la trasforma in Json e la invia come risposta alla Mobile App con status code 200.

Cambio, recupero della password dell'utente

Quando l'utente necessita di recuperare o cambiare la password la Mobile App effettua una richiesta htts al path */recoverpsw*. La Mobile App invia il seguente Json :

```
{ 'username':string,
  'password':string,
  'recovery':string
 }
```

La funzione *doChangePassword()* valida il Json ricevuto e ne usa i campi per chiamare la funzione *changePasswordself* del database. Se la richiesta è eseguita correttamente a livello del database questi ritorna *True* e si ritorna lo status code 200 alla Mobile App. Altrimenti 401.

Validazione dei Json ricevuti

I Json ricevuti tramite POST HTTPS sono validati a seconda della risorsa REST da differenti funzioni. In particolare I tipi di Json possibili sono tre :

- Login/Registrazione
- Recupero/Cambio password
- Unlock

per far ciò ho utilizzato la classe python Draft4Validator che valida il Json che gli viene passato rispetto ad uno schema precedentemente definito.

4.4 ClientSocket

Lo scopo principale di questa classe è quello di creare un socket stream per la comunicazione con la *TBox* presente all'interno del veicolo che l'utente vuole utilizzare. Tramite questo socket la *TBox* riceve la configurazione iniziale relativa all'utente corrente e allo spegnimento del veicolo sarà la *TBox* ad inviare sempre tramite questo socket l'ultima configurazione al Server.

Il costruttore della classe *ClientSocket* riceve due parametri : utente e identificativo del veicolo (la targa).

4.4.1 Inizio della comunicazione

Tramite la funzione *start()* della suddetta classe si interroga il database al fine di recuperare l'indirizzo IP del veicolo prescelto. Se questi è 'None' o *False* allora il veicolo è offline oppure è attualmente in uso da un altro utente. In questo caso si ritorna *False* e la classe ServerRESTPool ritornerà uno status code di 401 alla Mobile App come descritto nel capitolo precedente. In caso contrario si cerca di stabilire una connessione TCP alla porta 4333 con il server presente sulla *TBox* . Tale socket è poi passato alla classe SSLSocketClass che a partire dal certificato x509 della Certification Authority responsabile dei certificati per le *TBox* a cui ci vogliamo connettere, al certificato x509 del server e alle chiavi asimmetriche relative a tale certificato esegue un wrapping all'interno di un contesto SSL del socket iniziale. Il risultato di questo processo è un oggetto che estende la classe socket e oltre ad applicare un wrapping SSL sul canale fornisce altre due funzioni, *writeData(data)* e *readData()*, che antepongono al contenuto la lunghezza dello stesso in modo da essere sicuri che in invio tutti i dati siano stati inviati e in ricezione tutti i dati siano stati ricevuti. Tutto ciò è completamente nascosto alla classe ClientSocket che usa unicamente le due funzioni *writeData(data)* e *readData()*.

A questo punto la classe estrae del database la configuratione corrente, tramite la *json.dumps()* converte la mappa ricevuta in una stringa Json e con la *writeData(...)* la

invia alla *TBox* . Se la creazione del socket e l'invio della configurazione è avvenuto con successo allora si setta sul database che l'utente da questo momento sta utilizzando la vettura (la vettura è stata sbloccata e il bloccasterzo è in posizione 'unlock') . Da questo thread si sgancia un altro thread che proseguirà la comunicazione con la *TBox* mentre quello corrente terminerà la connessione HTTPS con la Mobile App nell'oggetto UserHandler che ha creato il ClientSocket corrente. Se invece viene lanciata un'eccezione durante la creazione della connessione il nuovo thread non è creato e si ritorna *False* concludendo la connessione con la Mobile App, status code 412.

4.4.2 Terminazione della comunicazione

Nel caso in cui la connessione alla *TBox* sia avvenuta correttamente e il nuovo thread sia stato quindi sganciato, tale thread esegue la funzione *continueTalk()* la quale resta in ascolto sul socket finchè non riceve la comunicazione dell'ultima configurazione dalla *TBox* . I dati ricevuti sono trasformati in un dizionario di python con la funzione *json.loads(data* e con i dati scompattati si chiama la funzione del database *updateConfiguration(...)* . Si attende altresì che la *TBox* invii la stringa 'BYE' per chiudere la connessione e chiamare una istanza della classe Notification che provvederà a mandare una notifica push alla Mobile App per segnalare che una nuova vettura è disponibile. Il contenuto inviato è la mappa con le posizioni dei veicoli e le loro targhe.

4.5 Notification

La classe Notification sfrutta la classe *request* di Python per inviare una HTTPS post request al server Google Firebase Cloud Messaging (FCM). Il contenuto del Json inviato (chiave 'data') sarà inviato a tutti i dispositivi con la Mobile App e nel caso in cui questi stiano visualizzando la mappa nell'activity di Login la vedranno aggiornarsi. In qualsiasi altra activity o se l'applicazione non è in esecuzione non riceveranno nessun feedback.

Inoltre tale notifica è legata al topic *maps* . Ciò è dovuto alle API di FCM che consentono di inviare le notifiche o ad una lista di ID di dispositivi o su dei canali ognuno avente un topic specifico. Per semplificare la struttura del codice ho fatto in modo che la Mobile App si registri sul topic 'maps' e così sono in grado di inviare in automatico un messaggio a tutti i dispositivi. In ottica futura, nel caso in cui si necessiti di inviare diversi tipi di notifiche push basterà creare nuovi topic (canali) e far registrare di conseguenza la Mobile App a tali nuovi canali.

4.6 ServerSocket

Il ServerSocket di per se non costituisce una classe ma un insieme di funzioni le cui più importanti sono due, una che crea il server TCP e una che processa la connessione. Tale

server elabora le richieste delle *TBox* che pervengono nel momento in cui cambia l'IP del veicolo che ospita la *TBox* oppure la posizione GPS del veicolo stesso.

Il server è in ascolto sulla porta 8080 e appena c'è una connessione TCP il relativo socket viene wrappato dalla classe SSLSocketServer e passato come parametro all'handler della richiesta insieme all'IP della richiesta e l'handler è eseguito nel constesto di un thread one-shot. Il processo si mette in ascolto per la prossima richiesta mentre il thread processa la richiesta.

L'handler tramite la funzione *getSubjectName()* della classe SSLSocketClass estrae l'identificativo del veicolo, ovvero la targa, dal certificato x509 inviato dalla *TBox* per creare la connessione SSL. Estrae dal socket il Json contenente la posizione del veicolo e sull'istanza del database invoca a funzione *updateCarInfo(car, ip, geo['lat'], geo['lon'], geo['acc'])*. Fatto ciò il thread chiude la ordinatamente la connessione, invia una push notification ai dispositivi con la Mobile App installata tramite la classe Notification per avvisare del cambio di posizione di un veicolo (solo se questi è libero, cioè quando viene rilasciato dall'utente) e termina.

4.7 SSLSocketClass

Lo scopo di questa classe è quello di semplificare e omogeneizzare il processo di conversione di un socket TCP in un socket TCP wrappato in un contesto SSL. Tale classe inoltre mette a disposizione tre funzioni che semplificano l'invio e la ricezione dei dati, nonchè l'estrazione del SubjectName della controparte SSL. Per l'utilizzo di tale classe si presuppone che client e server utilizzino una Client Authentication del protocollo TLS/SSL. È pienamente compatibile sia con le versioni di Python maggiori della 2.6.0 sia con quelle maggiori della 3.0.

4.7.1 Il costruttore

Il costruttore della classe riceve una serie di parametri elencati di seguito :

- conn_sock, il socket da wrappare
- serverside, se è un socket lato server
- pathKey, il path della coppia di chiavi asimmetriche relative al certificato da usare
- pathCert, il path del certificato x509 da usare per l'autenticazione
- pathServerCert, il path del certificato della CA della controparte
- host, quando la classe è usata per il client è l'IP a cui connettersi
- port, quando la classe è usata per il client è la porta a cui connettersi

Se si necessita di fare il wrapping di un socket usato lato server prima di tutto si instanzia un constesto SSL

```
ctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
```

usando il protocolo TLS 1.2 . Su tale oggetto si setta il path dove risiede il certificato x509 della CA della controparte (in formato PEM), si specifica path del (nostro) certificato da usare (sempre in formato PEM) e la relativa coppia di chiavi asimmetriche RSA, si chiede che vi sia una Client Authentication

```
ctx.verify_mode = ssl.CERT_REQUIRED
```

e infine dopo aver settato parametri di sicurezza si esegue l'effettivo wrapping del socket nel contesto SSL appena creato.

```
ctx.set_ciphers('TLSv1.2:!aNULL:!eNULL')
ctx.set_ecdh_curve('secp384r1')
ctx.options |= ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3 | ssl.OP_NO_TLSv1 |
    ssl.OP_NO_TLSv1_1 | ssl.OP_SINGLE_ECDH_USE | ssl.
   OP_CIPHER_SERVER_PREFERENCE |ssl.OP_NO_COMPRESSION

self.connstream = ctx.wrap_socket(conn_sock,server_side=True)
```

Nel contesto inoltre si filtrano gli algoritmi per la Cipher Suite con autenticazione e cifratura entrambi diversi da *null*. Come curva per le eventuali operazioni di crittografia su curve ellittiche si sceglie la *secp384r1* consigliata dall'RFC 6460 [22] che riprende le direttive della NSA Suite B Cryptography [26]. Si disabilitano tutti i protocolli precedenti alla versione TLS 1.2, si disabilita la compressione dei dati in modo da evitare attacchi basati sulla compressione e si impone l'ordine di preferenze del server nella scelta della Cipher Suite.

Nel caso in sui il parametro serverside sia settato a *False* invece di creare il contesto ssl si utilizza quello di default contenuto nel pacchetto python *ssl* e si esegue la connessione al server.

```
 self.connstream = ssl.wrap_socket(conn_sock,
                                   ca_certs=pathServerCert,
                                   cert_reqs=ssl.CERT_REQUIRED,
                                   keyfile=pathKey,
                                   certfile=pathCert,
                                   server_side=False,
                                   ssl_version=ssl.
                                      PROTOCOL_TLSv1_2)

self.connstream.connect((host,port))
```

A questo punto nella variabile interna alla classe *self.connstream* è presente il socket wrappato in un constesto SSL.

4.7.2 Estrazione del Subject Name

Nel caso in cui vi sia la necessità di identificare in modo sicuro la controparte, per esempio quando il ServerSocket riceve una richiesta di connessione da un veicolo e vogliamo essere sicuri che stiamo comunicando con quel veicolo e con nessun altro, si può risalire al Subject Name registrato nel certificato del peer. Dall'oggetto che fa riferimento alla connessione si può estrarre il certificato della controparte con la *self.connstream.getpeercert()* e nella mappa ritornata, alla chiave 'subject' è presente una lista di matrici. Se in posizione (0,0) di una delle matrici è presente la stringa 'commonName' allora all'indice (0,1) vi è il Subject Name.

4.7.3 Operazioni di scrittura e lettura dal socket

Lavorare a basso livello (direttamente con i socket) può essere problematico in quanto non è detto che tutti i dati inviati siano ricevuti in una sola lettura poiché vi sono dei buffer sia in invio che ricezione. Per tale motivo ho creato due funzioni base che leggono o inviano esattamente un certo quantitativo di dati :

```
def _myreceive(self,socket, MSGLEN):
    chunks = []
    bytes_recd = 0
    while bytes_recd < MSGLEN:
        chunk = socket.recv(min(MSGLEN - bytes_recd, 2048))
        if chunk == '':
            raise RuntimeError("socket connection broken")
        chunks.append(chunk.decode('utf8'))
        bytes_recd = bytes_recd + len(chunk)
    return ''.join(chunks)

def _mysend(self,socket, msg):
    totalsent = 0
    MSGLEN = len(msg)
    while totalsent < MSGLEN:
        sent = socket.send(msg[totalsent:])
        if sent == 0:
            raise RuntimeError("socket connection broken")
        totalsent = totalsent + sent
```

A partire da queste due funzioni ho creato un 'protocollo di comunicazione' molto elementare che è costituito dall'invio della lunghezza del messaggio su esattamente 8 caratteri (10 in totale in quanto invio o ricevo una stringa esadecimale che inizia con '0x'), seguito dal messaggio stesso. Posso dunque inviare teoricamente fino a 4.294.967.295 di byte in un solo messaggio (0xffffffff)

Tale protocollo è implementato dalle due funzioni *readData()* e *writeData()* . La prima funzione, fa uso della *_myreceive* che richiama due volte, la prima volta chiede di leggere 10 byte (la lunghezza in esadecimale del messaggio) che, trasformati in un *long*, rappresentano il valore da passare alla *_myreceive* per leggere il messaggio effettivo.

```
def readData(self):
    lx = self._myreceive(self.connstream, 10)
    leng = long(lx, 0)

    return self._myreceive(self.connstream, leng)
```

La seconda funzione invece, ricevuto il messaggio da inviare, ne calcola la lunghezza e poi invia tramite la *_mysend* prima la lunghezza su 10 byte come stringa esadecimale e poi invia il messaggio effettivo.

```
def _myreceive(self,socket, MSGLEN):
    chunks = []
    bytes_recd = 0
    while bytes_recd < MSGLEN:
        chunk = socket.recv(min(MSGLEN - bytes_recd, 2048))
        if chunk == '':
            raise RuntimeError("socket connection broken")
        chunks.append(chunk.decode('utf8'))
        bytes_recd = bytes_recd + len(chunk)
    return ''.join(chunks)
```

4.8 Cipher Suite

Considerato la necessità di effettuare operazioni di cifratura e decifratura ho ritenuto utile racchiudere in un unico file le funzioni per gli algoritmi AES_128_CBC ed RSA encryption.

4.8.1 AES

La classe AESCipher riceve come parametro del costruttore la 'key' da utilizzare (16 byte, 128 bit).

In cifratura vengono generati 8 byte random che trasformati in esadecimale sono 16 caratteri ovvero 16 byte che sono utilizzati come IV. Si instanzia un oggetto di tipo AES con l'IV appena creato e modo di cifratura CBC, si fa il padding PKCS7 del messaggio da cifrare e come output si ha la concatenazione dell'IV con il messaggio (con padding) cifrato con l'oggetto AES. Poiché la cifratura restituisce in output una serie di byte multipli del blocco base (32 byte) per una trasportabilità dei dati si esegue un operazione di trasformazione in stringa esadecimale con la funzione *binascii.hexilify(...)*.

In decifratura invece, dal messaggio passato di estraggono i primi 16 byte che sono l'IV e sempre tramite un'istanza dell'oggetto AES se ne fa la decifratura, non prima di aver usato la funzione *binascii.unhexilify(...)* sulla parte restante dei dati.

4.8.2 RSA

La classe RSAEncDec utilizza un path statico per reperire la coppia di chiavi pubblica/-privata e le importa in variabili interne. Ovviamente, data la limitatezza sulla dimensione massima delle operazioni di cifratura asimmetriche (lunghezza della chiave in byte - 11 byte) si è provveduto ad usare tale classe solo per dati molto piccoli come ad esempio il codice di recupero della password per l'utente.

In cifratura tramite la funzione *encrypt()* dell'oggetto che rappresenta la chiave pubblica si procede a criptare il testo passato come parametro. In decifratura invece si utilizza la funzione *decrypt()* della chiave privata. Anche qui si è utilizzata la coppia di funzioni hexilify / unhexilify della classe binascii.

Requisiti di sicurezza per il Server

Segue qui l'analisi di sicurezza per il Server che è stato il filo conduttore per lo sviluppo dello stesso.

- Dati at rest (Database)
  - Confidenzialità della password dell'utente e del codice per il recupero della stessa
- Dati in motion
  - Confidenzialità dei dati trasmessi
  - Integrità dei dati trasmessi
- Autenticazione delle parti
- Disponibilità del servizio

La confidenzialità dei dati at rest è stata ottenuta attraverso la cifratura con chiave pubblica del Server della password e del codice per il recupero della password (la password è hashata, protezione da attacchi dizionario e derivati).

Per i dati in motion le proprietà di sicurezza sono state ottenute attraverso l'utilizzo del protocollo TLS/SSL associato alla comunicazione socket, l'autenticazione delle parti avviene attraverso la client authentication e la server authentication del TLS.

Per la disponibilità dei dati, essendo il server sul sistema Linux, ho attivato l'implementazione dei SYN cookies per l'handshake TCP in grado di evitare l'attacco di Denial of Service SYN Flooding.

```
sudo nano /etc/sysctl.conf
net.ipv4.tcp_syncookies = 1
sudo sysctl -p
```

Tutto ciò è stato realizzato attraverso la libreria Python *ssl* e *Crypto*.

Capitolo 5

TBox

5.1 Introduzione

Nel linguaggio automotive con il termine ECU (Engine Control Unit) si indica un dispositivo a bordo del veicolo dotato di sensori e attuatori che consentono l'iniezione della giusta quantità di combustibile all'interno dei cilindri del motore in base alle condizioni del sistema quali velocità del veicolo, pressione nei condotti, temperatura del motore ecc... Con il passare degli anni tale termine è stato usato per indicare i dispositivi hardware all'interno del veicolo capaci di comunicare tra di loro tramite bus CAN o LIN e dotati di attuatori e/o sensori. Esempi di ECU sono l'unità che controlla l'ABS, quella dell'air bags, quella per il controllo delle luci, quella per il controllo dei finestrini e così via.

Il numero di ECU all'interno di un veicolo può superare tranquillamente le 100 unità per un veicolo di fascia bassa. Per controllare il funzionamento e per la diagnostica di tutte queste centraline vi è una ECU in particolare detta BCM (Body Control Module), il cuore del veicolo.

Con l'evoluzione della tecnologia e con la necessità di tracciare i veicoli si è sviluppato un nuovo dispositivo elettronico dotato come minimo di un microprocessore, un sensore GPS per la posizione e di un modulo per la connettività di rete (GSM/GPRS) chiamato *TBox* (Telematic Box). La *TBox* se dotata di interfaccia CAN può comunicare con tutto o parte del sistema del veicolo interconnesso tramite bus CAN. Questo tipo di centralina è utilizzato sia a bordo dei veicoli appartenenti alle flotte delle compagnie di Car Sharing sia da parte delle compagnie assicurative per tracciare i veicoli e analizzare la dinamica degli incidenti stradali.

Il compito della mia *TBox* è quello di comunicare da un lato con il veicolo tramite protocollo CAN e dall'altro quello di interfacciarsi con il server tramite socket TCP.

5.1.1 L'hardware

La realizzazione della *TBox* per il POC ha richiesto un modulo hardware estremamente versatile per la programmazione e che supportasse un'interfaccia CAN. La scelta dopo una serie di ricerche è ricaduta sulla board Raspberry Pi 3 a cui è stata aggiunta una board con interfaccia CAN, sensore GPS e accellerometro/giroscopio a 6 assi (non usato) chiamata 'PiCAN GPS and Accelerometer' della SK PANG Electronics [15].

Raspberry Pi 3

Il Raspberry Pi è una board sviluppata nel Regno Unito dalla Raspberry Pi Foundation e lanciata sul mercato nel febbraio 2012. Lo scopo era quello di realizzare un dispositivo economico per avvicinare i ragazzi delle scuole all'informatica. Il progetto iniziale ha subito varie revisioni passando per i modelli B, B rev2, 2 e attualmente è in commercio la versione 3 che è quella utilizzata per la *TBox* . Di seguito sono riporte le caratteristiche tecniche :

- Broadcom BCM2837 Quad Core 1.2 GHz
- 1GB LPDDR2 a 900 MHz
- Broadcom VideoCore IV 400 MHz
- Ethernet, WiFi N 2.4 GHz, Bluetooth 4.1 LE
- I^2C, SPI, UART

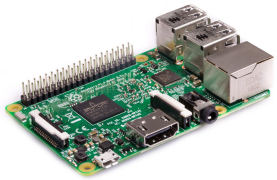


Figura 5.1. Raspberry Pi 3 [14]

La board è fornita senza memoria di massa e senza quindi sistema operativo. Su una scheda SD di 16 GB ho installato la versione ufficiale di linux per Raspberry Pi, Raspbian Jessie, che come suggerisce il nome è una derivata del sistema operativo Debian che utilizza il kernel Linux.

Per quanto concerne la comunicazione della *TBox* (del Raspberry Pi) su bus CAN è stata enormemente facilita dalla scelta del sistema operativo Raspbian che avendo a bordo il kernel Linux 4.9.24 implementa nativamente l'interfaccia SocketCAN introdotta a partire dal kernel Linux 2.6.25. Infatti tale interfaccia estende le Berkeley sockets API e quindi la famiglia di protocolli di rete come il PF_INET (per internet TCP/IP) aggiungendo il protocollo PF_CAN e i driver per comunicare con i controller CAN. In questo modo basta instanziare un socket di tipo PF_CAN e comunicare con le primitive read() e write() . Durante il binding del socket questi è associato ad un controller che può essere reale oppure emulato e tale controller apparirà tra le interfacce del sistema operativo (*ifconfig*).

PiCAN GPS e Accelerometer

La board PiCAN GPS è un controller per bus CAN per il Raspberry Pi 2 e 3. Come controller per il bus CAN utilizza il Microchip MCP2515. Esso comunica con il Raspberry Pi tramite il protocollo SPI sul bus SPI.0 utilizzando entrambi gli slave a disposizione e interrupt sul GPIO25. La programmazione può essere fatta sia in C che in Python (implementa nativamente l'interfaccia SocketCAN a partire dalla versione 3.3).

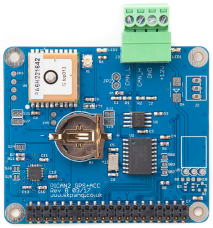


Figura 5.2. PiCAN GPS Accelerometer [12]

I dati del GPS invece sono elaborati dal chipset MTK3339 che ha una antenna integrata con -165 dBm sensitivity, 10 Hz updates e 66 canali. L'MTK3339 comunica con il Raspberry Pi tramite la porta seriale tts0, quindi con il protocollo UART. Esso invia una serie di dati testuali che possono essere letti dal Raspberry Pi ed elaborati. Questo processo è semplificato dal software *gpsd* che riceve i dati sulla seriale e li espone in formato Json su un socket locale TCP/IP sulla porta 2947. Sul modulo vi è inoltre una batteria tampone che velocizza il fix della posizione GPS. Al modulo ho collegato inoltre una antenna attiva tramite il connettore μFL. Sulla board è inoltre presente il controller MPU-6050 che controlla un giroscopio/accelerometro a sei assi.

Interruttore a chiave

Per simulare lo spegnimento del veicolo ho utilizzato un interruttore a chiave connesso al GPIO18 (pin 12) e a ground. Normalmente sul GPIO è presente il valore logico 1 che rimane tale finchè la chiave lascia il circuito aperto. Girando la chiave (e quindi spegnendo il motore) si chiude il circuito e il pin 12 è connesso così a ground. Quando rilevo il valore logico 0 su quel pin allora posso affermare che il veicolo è stato spento ed è quindi nuovamente disponibile per altri utenti.

5.1.2 Il software

Come già accennato il Raspberry Pi 3 utilizza Raspbian come sistema operativo e il linguaggio di programmazione utilizzato è Python versione 3.4.* . Le librerie Python di sistema più utilizzate sono state python-can [13], bitarray, threading, Process, time, netifaces e os.

5.1.3 Schema Generale

Di seguito è riportato uno schema riassuntivo con le principali classi relative alla *TBox* , le linee rappresentano le dipendenze. Segue una foto della *TBox* realizzata.

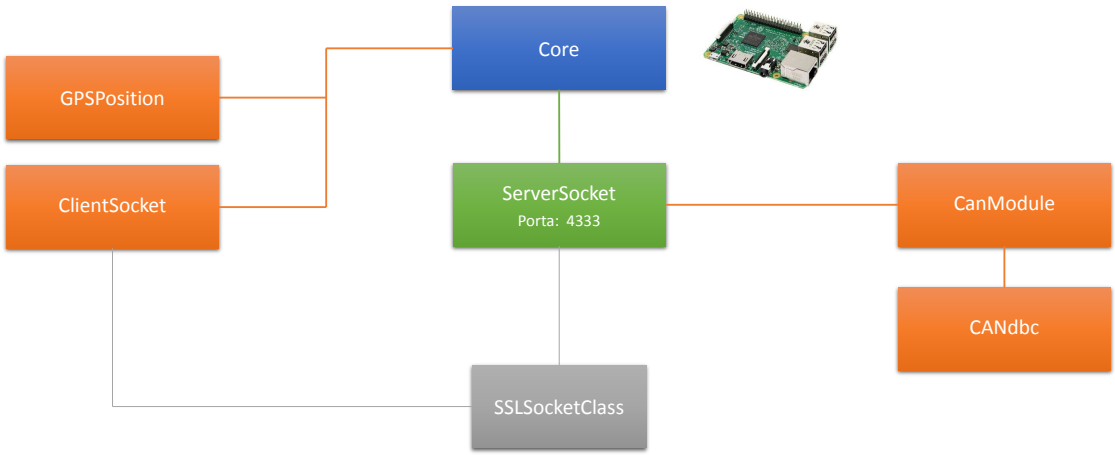


Figura 5.3. Classi principali

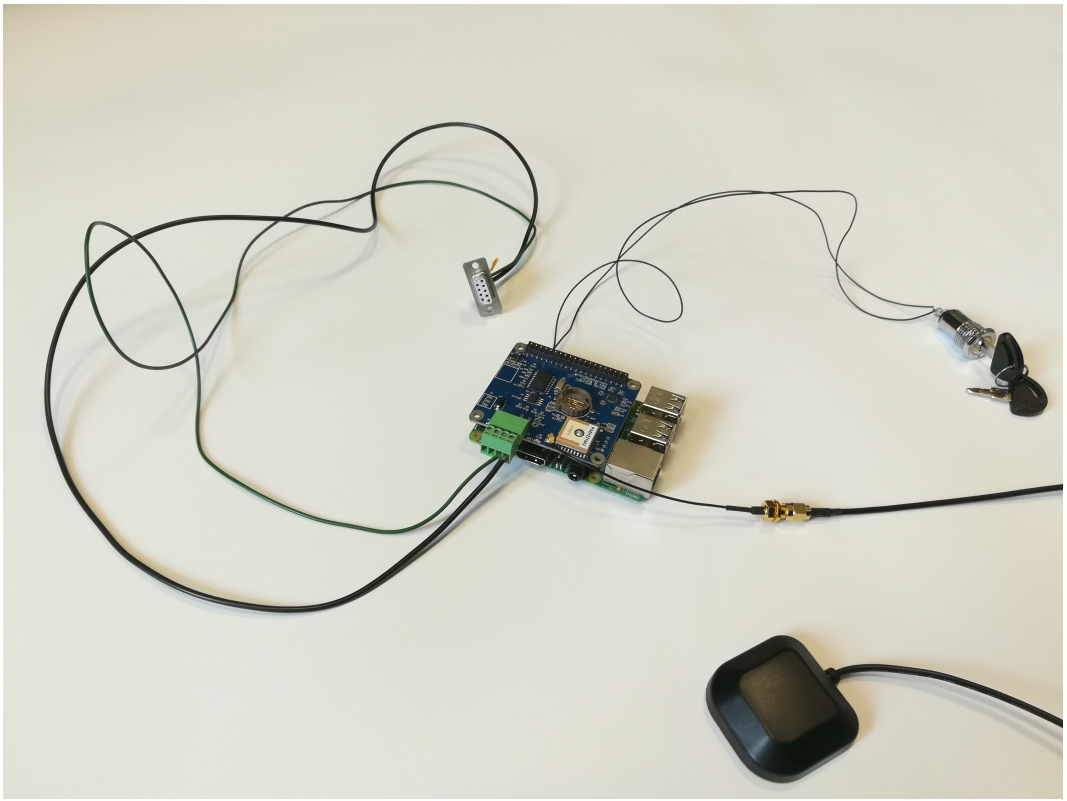


Figura 5.4. *TBox*

5.2 IP e Posizione GPS

Il punto di partenza dell'intero sistema della *TBox* è lo script *Core.py*. Innanzitutto per motivi di degub tramite la classe ConfigParser legge dal file *config* se è eseguito su un Raspberry Pi oppure su un PC e in base allo stato della variabile *is_raspberry* configura la variabile del nome dell'interfaccia per internet. Inoltre se è eseguito sul Raspberry Pi avvia il server per il GPS e setta attiva l'interfaccia can0 relativa al modulo PiCAN con bitrate pari a 500 Kbps.

Lo script prende il pid del processo corrente ed esegue una *fork()*, il padre avvia un thread per il monitoraggio della posizione GPS e della connessione mentre il figlio avvia un'istanza della classe ServerSocket.

Il thread appena citato ogni 15 secondi da un lato interroga il sistema operativo per sapere l'IP dell'interfaccia internet mentre dall'altro, affidandosi alla classe GPSPosition estrae dal modulo GPS la posizione GPS corrente. Se l'IP corrente è diverso rispetto a quello precedente (il sistema è stato riavviato) oppure la posizione corrente è diversa da quella precedente si avvia un'istanza del ClientSocket per notificare i cambiamenti al Server.

GPSPosition

Questa classe rappresenta un wrapper per ottenere l'effettiva posizione GPS che è estratta dal server gpsd e inserita come tre attributi distinti (Latitudine, Longitudine e errore in m) nella classe *GPSCoordinate* . Quest'ultima è in grado di confrontarsi con un'altra istanza della stessa classe e stabilire se le due posizioni sono compatibili rispetto agli errori oppure se sono proprio due posizioni diverse in modo da limitare il numero di aggiornamenti inviati al server. Questo confronto è possibile grazie all'overloading delle funzioni __eq__ e __ne__ che calcolano la distanza tra i due punti con la funzione di sistema *vincenty* e la confrontano con la somma degli errori. Se la somma degli errori è maggiore della distanza tra i due centri allora le due posizioni si intendono compatibili, rappresentano cioè lo stesso punto geografico.

Per ottenere le tre informazioni sulla posizione si utilizza la classe di sistema gps3 da cui si estrae un GPSDSocket() dal quale si fa accesso al DataStream() e poi si chiede all'oggetto GPSDSocket() di connettersi al server GPSD e restare in ascolto. Si cicla sul gps_socket in attesa di nuovi pacchetti, quando c'è un nuovo dato questo è passato al data_stream che ne fa l'unpack e solo a questo punto possiamo accedere ai dati contenuti nel pacchetto. In particolare oltre a latitudine e longitudine si estraggono l'errore lungo x e l'errore lungo y in m e si considera come errore relativo alla posizione il massimo tra i due valori.

ClientSocket

Nel costruttore della classe si definiscono i path dei seguenti file : certificato x509 del veicolo, relative chiavi asimmetriche, certificato della CA del server.

La funzione *runClient()* che riceve come parametri latitudine, longitudine ed errore corrente si occupa di stabilire una connessione TCP/IP protetta dal protocollo SSL con il server. Sfrutta la stessa classe vista in precedenza ovvero la SSLSocketClass 4.7, crea il socket wrappato e tramite questi invia al server un Json con il seguente formato :

```
{'type':"coordinates",'lat':lat,'lon':lon,'acc':acc}
```

5.3 Infrastruttura a supporto del protocollo CAN

Riassumendo brevemente quello descritto fino ad ora da una parte vi è il sistema operativo Raspbian che tramite il kernel Linux mette a disposizione le API SocketCan, dall'altra vi è il database DBC che consente di comunicare con il resto del veicolo (ESL e Infotainment in questo caso) e a chiudere il cerchio vi è il Microchip MCP2515 che è visto da Rasbian come un'interfaccia di rete tipo l'Ethernet su cui leggere e scrivere.

Per sfruttare le API SocketCan su Linux vi sono due linguaggi di programmazione principali : C o Python. Io ho scelto Python perché si integra perfettamente con il resto della *TBox* .

5.3.1 Python-Can

La libreria python-can fornisce un supporto per i controller CAN astraendone le funzionalità di basso livello in modo da offrire una serie di API sia per l'invio che per la ricezione di messaggi CAN.

Inizializzazione del bus

Per la creazione del collegamento con il bus si devono fornire il tipo del bus e il canale. Il Microchip MCP2515 è mappato sul canale 'can0' e come implementazione dello stack di comunicazione si sceglie quella relativa alle API SocketCan di Linux

```
bustype = 'socketcan'
channel = 'can0'
bus = can.interface.Bus(channel=channel, bustype=bustype)
```

Invio di un messaggio

Per la creazione di un messaggio si utilizza la classe *can.Message*

```
msg = can.Message(arbitration_id=0x0f,
                  data=[1, 2, 3, 4, 15, 11, 110, 8],
                  extended_id=False)
try:
    bus.send(msg)
    print("Messaggio inviato")
except can.CanError:
    print("Errore in invio")
```

Ricezione di un messaggio

Ci sono due modi, il primo è quello di ciclare sul bus (metodo bloccante) in attesa di nuovi messaggi

```
for msg in bus:
        print(msg)
```

il secondo consiste nell'istanziare un can.Listener, una classe callback, da passare ad un can.Notifier che la esegue in un thread separato (metodo non bloccante)

```
class CanListener(can.Listener):
    def on_message_received(self, msg):
         print(msg)
listener = CanListener()
notifier = can.Notifier(bus, [listener])
```

5.3.2 CANdbc

Per rendere il codice riutilizzabile, invece che scrivere come costanti statiche gli oggetti di tipo can.Message che mi sarebbero serviti, ho preferito creare uno script (CANdbc.py) che all'interno avesse delle classi per rendere facile il recupero dei messaggi da un qualsiasi file DBC e che dato il nome di un segnale e di un messaggio fosse capace di settare in automatico il valore che si vuole o di estrarne il valore da un messaggio pervenuto che abbia quel nome.

CANdatabase

CANdatabase è la classe che ricevuta una lista di path di file DBC li analizza e ne estrae le informazioni. Il funzionamento della classe prevede di creare dei messaggi statici di tipo *InternalMessage* per ogni istanza della classe stessa che sono manipolabili dall'esterno. Da un lato li crea e dall'altro li conserva con le modifiche apportate pronti per essere inviati sul bus quando serve.

Il costruttore della classe crea due mappe una da identificativo a messaggio e una da nome a messaggio. Poi per ogni path passato nella lista lo analizza e per ogni riga vede se fa match con due espressioni regolari, una per i messaggi e una per i segnali.

I messaggi all'interno del file DBC iniziano su una riga con la stringa *BO_* a cui seguono l'id, il nome, la lunghezza e chi lo invia. L'espressione regolare utilizzata è la seguente:

```
r='BO_\s+(?P<can_id>\d+)\s+(?P<name>\S+)\s*:\s*(?P<length>\d+)\s+(?P
    <sender>\S+)'
reg=re.search(r,line)
```

A partire dal risultato, se è diverso da *None*, che è una mappa, si accede ai campi *can_id*, *length*, *name*. Se $can\_id > 0x7ff$ vuol dire che si ha un identificativo CAN2.0B, cioè esteso. Con questi dati si crea un *InternalMessage* e quest'ultimo oggetto è inserito all'interno delle due mappe di cui sopra.

I segnali invece, all'interno del file DBC, iniziano con la stringa *SG_* seguite da una serie di attributi i cui più importanti ai fini della tesi e per il funzionamento del sistema sono : *name*, *start_bit*, *length*, *endianness*.

Anche qui si costruisce una espressione regolare :

```
r = 'SG_\s+(?P<name>\S+)\s*(?P<is_multipexer>M)?(?P<multiplexer_id>m
    \d+)?\s*:\s*'
r+= '(?P<start_bit>\d+)\|(?P<length>\d+)\at(?P<endianness>[0|1])(?P<
    sign>[\+|\-])\s*'
r+= '\(\s*(?P<factor>\S+)\s*,\s*(?P<offset>\S+)\s*\)\s*'
r+= '\[\s*(?P<min_value>\S+)\s*\|\s*(?P<max_value>\S+)\s*\]\s*"(?P<
    unit>\S*)"\s+(?P<receivers>.+)'
```

Se il risultato dell'espressione regolare è diverso da *None* con i parametri ricavati si instanzia un oggetto della classe *Signal* e il segnale è aggiunto all'ultimo messaggio che è stato trovato. Si fa ciò perché la struttura del DBC è un messaggio seguito da tutti i suoi segnali.

La funzione *getMessageNameByID(id)* estrae se esiste *id* come chiave nella mappa (id,messaggio) il nome del messaggio relativo. Se l'id non è presente ritorna *None*.

La funzione *getIntrnalMessage(name)* estrae se esiste *name* come chiave nella mappa (name,messaggio) il messaggio relativo. Se *name* non è presente ritorna *None*.Il messaggio ritornato è del tipo *InternalMessage*.

La funzione *getMessage(name,data, useRaw=False)* estrae se esiste *name* come chiave nella mappa (name,messaggio) il messaggio relativo. Se useRaw è settato a *True* dal messaggio appena reperito estrae i dati da inviare (un oggetto di tipo binascii) e lo converte in byte. A questo punto viene creato un oggetto di tipo can.Message che a seconda del valore del parametro *useRaw* avrà il campo *data* riempito con i dati del messaggio statico o i byte passati come parametri.

Signal

La classe *Signal* rappresenta un segnale del database in formato DBC. Il costruttore riceve i nome, la lunghezza, lo start_bit e se endianess o meno. Questi dati sono salvati come

variabili di classe e nel caso si abbia un ordinamento Motorola del byte allora si modifica la start_bit nel seguente modo :

```
f = floor(start_bit / 8) * 8       #offset riga
c = ceil(start_bit / 8) * 8 - 1    #padding riga
self.start_bit = f + (c - start_bit)
```

In questo modo ora lo start_bit coincide con il bit in posizione 0 del segnale stesso.

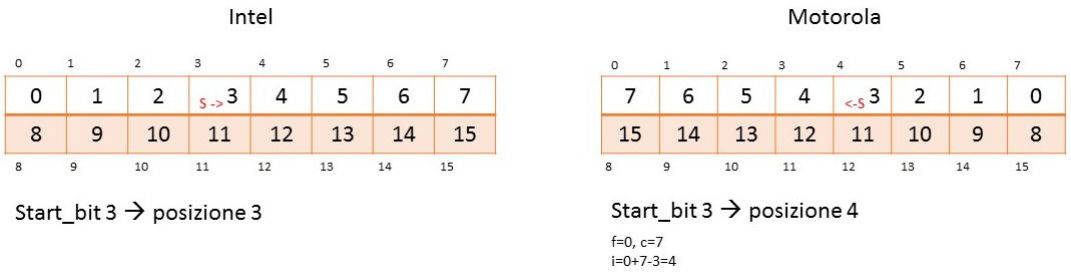


Figura 5.5. Byte Ordering

InternalMessage

Obiettivo primario di questa classe è quello di semplificare il più possibile la gestione dei segnali all'interno del messaggio. Dato un valore deve poter essere inserito correttamente nel messaggio a partire dal bit di inizio trattando i bit di dato come un'unica entità.

Il costruttore della classe riceve la lunghezza del messaggio (dlc) , il nome, se l'identificativo è esteso e l'identificativo stesso. Questi dati sono settati come variabii interne. Oltre a tali variabili è creata anche una variabile interna chamata *raw* di tipo bitarray (array di bit lungo 64 posizioni) con tutti i bit settati a *False* . Infine si crea una mappa che associerà (nome_segnale, segnale).

La funzione *addSignal(signal)* aggiunge *signal* alla mappa dei segnali relativi all'istanza corrente del messaggio.

La funzione *getSignals()* ritorna una lista con tutti i segnali relativi all'istanza corrente del messaggio.

La funzione *setValueSignal(signalName, value)* riceve il nome del segnale e un array come stringa di 0/1 o un bitarray. Dalla mappa si reperisce il segnale che si vuole modificare e grazie allo start_bit e alla lunghezza del messaggio si esegue un ciclo sulle posizioni della variabile interna *raw* impostandola uguale al relativo bit del vettore passato come parametro.

La funzione *getValueSignal(signalName)* riceve il nome del segnale di cui si vuole sapere il valore. Si inizializza una stringa vuota. Dalla mappa si reperisce il segnale che

si vuole leggere e grazie allo start_bit e alla lunghezza del messaggio si esegue un ciclo sulle posizioni della variabile interna *raw* e in base al valore si aggiunge il carattere '0' o '1' alla stringa. All'uscita del ciclo la stringa viene usata come ingresso per il costruttore di un oggetto bitarray che è il valore di ritorno della funzione stessa.

Per rendere più agevole la scrittura e lettura di uno specifico byte all'interno del messaggio invece che di un segnale ho fatto l'override delle funzioni __setitem__ e __getitem__ che corrispondono all'operatore '[]' in lettura o scrittura.

L'override della *__setitem__* riceve due parametri key e value. Si cicla nel range (key*8, (key+1)*8) e per ciascun bit del *raw* se ne setta il valore uguale alla relativa posizione del bitarray value.

L'override della *__getitem__* riceve solo il parametro key.Si inizializza una stringa vuota e si cicla nel range (key*8, (key+1)*8). In base al valore del bit in *raw* si aggiunge il carattere '0' o '1' alla stringa.All'uscita del ciclo la stringa viene usata come ingresso per il costruttore di un oggetto bitarray che è il valore di ritorno della funzione stessa.

La funzione *clearSignal(signalName)* invece provvere a settare tutti i bit corrispondenti al segnale scelto nella variabile interna *raw* a *False* .

Con l'override della funzione *__str__* creo una stringa come un insieme di righe ognuna composta dal nome del segnale e dal valore del segnale per ogni segnale contenuto nel messaggio.

5.4 Unlock e configurazione del veicolo

Descritti gli strumenti a disposizione ora si analizza la logica funzionale alla base della *TBox* .

5.4.1 ServerSocket

La *TBox* ha la necessità sia di comportarsi da client quando deve informare il server principale del cambiamento della sua posizione o del suo indirizzo di rete, sia da server in quanto deve rimanere in attesa di richieste di unlock da parte del server principale stesso che in questo caso si comporta da client. Suddividendo la comunicazione tra *TBox* e server principale nel modo appena descritto ho limitato enormemente il tempo di comunicazione diretta tra server principale e *TBox* . Essi mantengono le risorse occupate solo durante il viaggio dell'utente scambiandosi solo poche decine di byte.

Lo script SocketServer.py quando viene caricato legge il file di configurazione 'config.ini' tramite la libreria di sistema ConfigParser() ed estrae un booleano che indica se lo script è eseguito su Raspberry Pi (in ambiente di testing/produzione) oppure su PC (ambiente di sviluppo). In base a tale variabile instanzia tre variabili che rappresentano i path per il certificato x509 del server principale, il suo certificato x509 e la relativa coppia di chiavi asimmetriche. Tali path sono diversi tra PC e Raspberry Pi.

Il Server

La funzione *simple_tcp_server()* instanzia un socket della famiglia AF_INET (internet) di tipo SOCK_STREAM cioè TCP. Si esegue dunque il binding di tale socket sulla tupla ("0.0.0.0", 4333) ossia in ascolto sulle richieste provenienti da ogni indirizzo IP sulla porta 4333. Si chiama la funzione *listen()* sul socket e si entra in un ciclo while infinito in attesa di connessioni.

Dentro questo ciclo si è bloccati sulla *socket.accept()*. Appena qualcuno si connette al socket la *socket.accept()* ritorna e il socket la creazione di questa connessione è wrappato all'interno di un contesto SSL tramite la classe SSLSocketClass già precedentemente descritta e questa volta il parametro *serverside* è *True* .

```
request_socket, address_port_tuple = sock.accept()
connstream=SSLSocketClass(conn_sock=request_socket,
                          serverside=True, pathKey_=pathKey,
                          pathCert_=pathCert,pathServerCert=pathCA)
request_handler(connstream, address_port_tuple[0])
```

Nel caso in cui chi si connette al server non fosse il server principale allora la connessione è interrotta automaticamente da parte della classe SSLSocketClass in quanto il certificato presentato dalla controparte non è coincidente con quello che ci si aspetta. Se la controparte è accettata (è il server principale) allora si chiama la funzione *request_handler(socket,ip)* che si occupa della comunicazione con la controparte.

L'handler

La funzione *request_handler(socket,ip)* riceve come paramtri l'IP della controparte e il socket di tipo SSLSocketClass con cui poterci comunicare. Per prima cosa legge tramite la funzione *readData()* chiamata sul socket una serie di dati che costituiscono una stringa Json. Da tale stringa tramite la funzione *json.loads(data)* si cerca di convertire la stringa in una rappresentazione in Python di un Json ovvero in una mappa. Nel caso in cui si verifichi un'eccezione perché il Json non è conforme o ha errori di sintassi si chiude la connessione con la controparte. Se lo scompattamento invece va a buon fine si instanzia una *condition variable* a partire dalla classe threading e si crea altresì un'istanza della classe VehicleThread a cui si passano le configurazioni iniziali ricevute dal server principale (estratte dal database in funzione dell'utente che ha chiesto di fare l'unlock del veicolo) e la condition variable appena creata. Si chiama la funzione *start()* sul VehicleThread e si chiama la *wait()* sulla condition variable. Ora il server è bloccato finchè l'utente non chiuderà il veicolo con la chiave, sarà questa la condizione che permetterà di invocare la *notify()* sulla condition variable e quindi di svegliare il server stesso.

Una volta che il server è stato svegliato, quindi il veicolo è stato chiuso, si può chiedere all'istanza della VehicleThread quali sono gli ultimi valori per le configurazioni del veicolo rispetto all'utente corrente. Tali configurazioni sono la frequenza FM, lo stato del canale audio destro (on/off) e lo stato del canale audio sinistro (on/off). Con questi dati si costruisce un Json tramite la *json.dumps(data, separator)*. Si termina il VehicleThread invocando il suo metodo *stop()* e poi se ne fa il *join()*. Si invia il Json al server principale scrivendolo sul socket e si chiude la connessione.

5.4.2 Comunicazione con il veicolo

Tutta la comunicazione con il veicolo è gestita dello script CANModule.py che all'interno contiene una serie di classi che consentono di comunicare con le altre ECU, in questo caso sono l'ESL (Electric Steering Lock) e l'infotainment. Lo script utilizza una serie di classi quali RPi.GPIO per monitorare il segnale fisico di chiave, bitarray, time,threading e ovviamente python-can insieme al CANdatabase. In questa sezione non menzionerò e descriverò in modo dettagliato il significato dei messaggi riguardanti l'ESL perché protetti da segreto aziendale.

Prima di tutto si instanzia il database contente i messaggi e i segnali estratti da due file DBC : 'database.dbc' per il controllo dell'ESL e 'my_db_radio.dbc' per il controllo dell'infotainment. Viene creato un lock che andrà a proteggere le variabili globali contenenti i valori per le configurazioni del veicolo. Si instanziano una serie di messaggi estraendoli dal database con i valori di default, si creano una serie di variabili di supporto per lo stato del messaggio VIN (Vehicle Identifier Number).

VehicleThread

La classe VehicleThread estende la classe threading.Thread che rappresenta un runnable in Python e ne implementa il metodo *run()*.

Il costruttore della classe riceve un oggetto di tipo condition variable (quello creato nell'handler del server della *TBox* e le configurazioni (frequenza FM, canale destro, canale sinistro). Crea un collegamento con il bus CAN

```
bustype = 'socketcan'
channel = 'can0'
bus = can.interface.Bus(channel=channel, bustype=bustype)
```

e instanzia una serie di oggetti appartenenti a classi che estendono la classe threading.Thread che provvedono a gestire i messaggi CAN di tipo ciclico. Si instanzia infine un oggetto di tipo CanListener che estende la classe *can.Listener* e si imposta la variabile interna *stopMySelf* a *False* .

La funzione *run()* fa partire i thread instanziati precedentemente (che invieranno i messaggi ciclici) e invia sul bus CAN una serie di messaggi. Il primo messaggio è di tipo UNLOCK che serve a sbloccare l'ESL. Il secondo messaggio è il RadioCmd su cui si imposta il segnale RadioOnOff (unsigned su 2 bit) pari al valore 1, cioè si chiede all'infotainment di accendersi. Il terzo messaggio infine è sempre un RadioCmd che però ha RadioOnOff settato a 2 (configurazione) e i segnali Frequency, RightVol, LeftVol impostati con i valori passati come parametri al costruttore che sono i valori relativi all'utente corrente. È in questo punto che avviene la configurazione del veicolo. Fatto ciò il metodo run() cicla finché la variabile interna *stopMySelf* è *False* .

La funzione *stop()* viene invocata dal server quando è svegliato tramite la condition variable, cioè quando il veicolo è stato chiuso tramite la chiave. A questo punto si possono stoppare tutti i thread sganciati tramite il loro metodo *stop()* e si invia sul bus CAN l'ultimo messaggio RadioCmd questa volta con il segnale RadioOnOff pari a zero che implica lo spegnimento dell'infotainment. Infine si imposta la variabile interna *stopMySelf* a *True* così da rilasciare le risorse e terminare il thread corrente.

Il server prima di chiamare la *stop()* sul VehicleThread utilizza i metodi getFrequency(), getVr(), getVl() per recuperare i valori dell'infotainment alla chiusura del veicolo.

CanListener

Come accennato la classe CanListener estende la classe *can.Listener* e dunque sovrascrive il metodo *on_message_received(self, msg)*. Il costruttore riceve un oggetto di tipo condition variable (quello proveniente dalla classe VehicleThread proveniente dal server) e lo salva come variabile interna alla classe.

La funzione *on_message_received(self, msg)* analizza l'identificativo del messaggio CAN ricevuto come parametro, se corrisponde al valore decimale specifico (omissis) e il

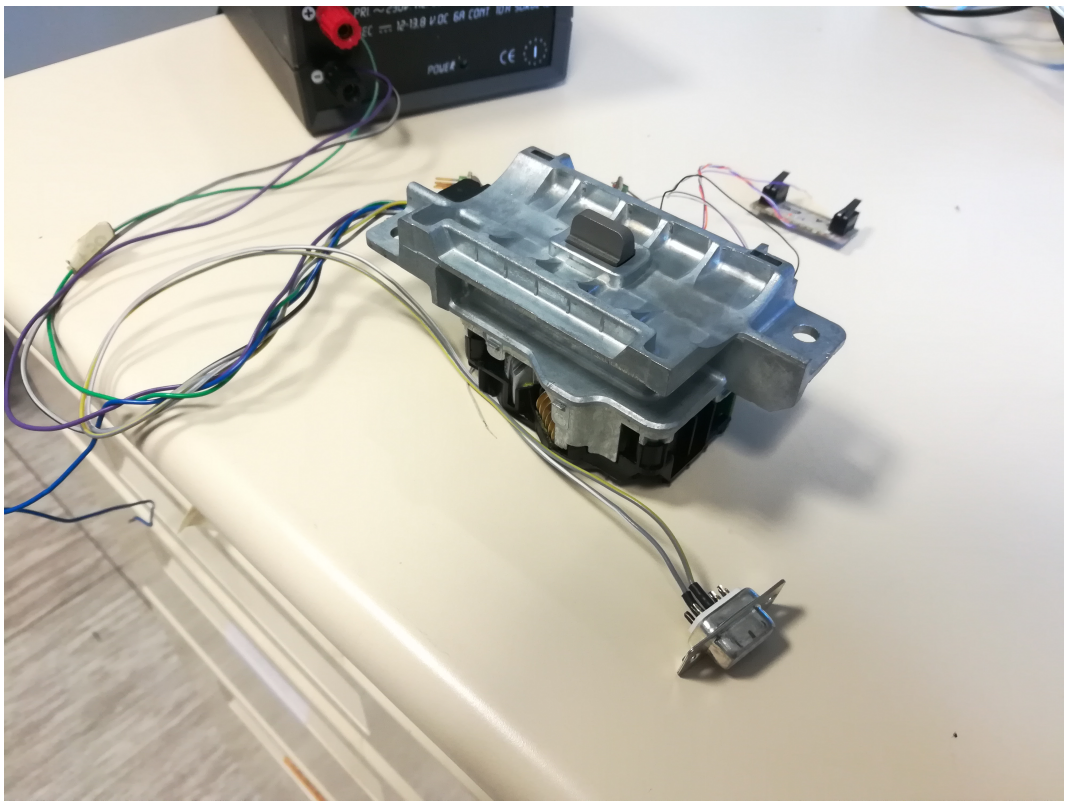


Figura 5.6. ESL

campo 'data' è uguale a quello del messaggio di ESL_REQUEST_LOCK allora è avvenuto il lock dell'ESL,quindi è stato chiuso il veicolo e quindi si può invocare la *notify()* sulla condition variable per svegliare il server.

Se invece il nome del messaggio (recuperato tramite l'identificativo e la funzione getMessageNameById(...)) è 'RadioSts' allora si recupera dal database il messaggio 'RadioSts', si copiano i byte ricevuti nel messaggio interno e dal messaggio interno tramite la sua funzione getValueSignal(nameSignal) si estrae il valore del segnale stesso. I segnali estratti sono Frequency, RightVol e LeftVol che corrispondono alla frequenza FM , allo stato del canale destro (on/off) e allo stato del canale sinistro (on/off).

Il messaggio RadioSts è di tipo ciclico ed è inviato dall'infotainment una volta al secondo.

KeyThread

La classe KeyThread è molto compatta. Come suggerisce il nome estende la classe threading.Thread e controlla il segnale di chiave. Nel costruttore riceve un oggetto di tipo bus CAN e inizializza il driver che controlla GPIO del Raspberry Pi impostando la nomenclatura BCM e tramite il driver setta il pin GPIO_18 in lettura e voltaggio pari al livello alto. Tale pin è collegato tramite l'interruttore a chiave a GND (ground) , il circuito però è aperto, quindi lo stato del pin è a *True* .

La funzione *run()* ogni $200ms$ controlla lo stato del pin 18. Se si chiude il veicolo la chiave viene girata e quindi il cirtuito si chiude portando il il pin al livello del GND quindi a *False* . Quando è a *False* si invia sul bus un messaggio di ESL_REQUEST_LOCK. Ad ogni ciclo dopo la sleep si controlla se si può terminare il controllo e quindi il thread stesso.

Altri thread

Gli altri thread non li posso descrivere nel dettaglio in quanto coperti da segreto aziendale. Servono ad inviare messaggi ciclici il cui contenuto può anche variare nel tempo, ne ho creato uno per ciascun tipo di messaggio perché ogni messaggio ha una logica differente di evoluzione del contenuto. Essi hanno una struttura simile al KeyThread ovvero hanno una funzione stop(). Altresì nel costruttore ricevono un decimale che rappresenta il tempo che deve intercorrere tra l'invio di un messaggio e l'altro. Tale tempo varia da $10ms$ fino a $250ms$. Per essere sicuro che siano inviati sempre con la stessa frequenza o quanto meno con il minor ritardo possibile all'inizio del ciclo prendo il tempo , modifico il messaggio , lo invio , riprendo il tempo, faccio la differenza tra il tempo per il prossimo invio e quanto ne ho utilizzato per le operazioni e metto il sleep il thread per il tempo rimanente.

```
s=time.time()
#modifica e invio messaggio
toSleep=(self.time-(time.time()-s))

if toSleep > 0:
        time.sleep(toSleep)
```

Requisiti di sicurezza per la *TBox*

Anche qui si è fatta preventivamente un'analisi di sicurezza e le proprietà di sicurezza emerse sono state :

- Dati in motion
  - Confidenzialità dei dati trasmessi
  - Integrità dei dati trasmessi
- Autenticazione delle parti
- Disponibilità del servizio

Anche qui sono state usate le stesse tecniche di protezione viste per il server: TLS, SYN Cookies, Client/Server Authentication.

Capitolo 6

Infotainment

6.1 Introduzione

L'idea originale era quella di avere come dispositivi configurabili un sedile comandabile elettronicamente tramite bus CAN e un infotainment entrambi relativi al modello di autovettura Alfa Giulia (Alfa Romeo). Putroppo per limitazioni temporali e di autorizzazioni ciò non è stato possibile. Per tale motivo ho successivamente ridotto il numero di parti configurabili da due a uno, cioè il solo infotainment. Poiché non è stato possibile reperire l'infotainment dell'Alfa Giulia ho provato a configurare altri due infotainment.

Il primo infotainment che ho provato è stato uno prodotto dell'Iveco (modello omissis) che per limitazioni del firmware interno era in grado di emettere il suo stato tramite messaggi CAN (frequenza FM/AM e volume) ma non era in grado di ricevere messaggi CAN capaci di cambiare le due variabili ossia frequenza e volume. È risultato essere non idoneo per la tesi che prevedeva l'autoconfigurazione del veicolo in base all'utente.

Il secondo infotainment analizzato è stato un modello della Parasonic (modello omissis) che pur avendo comandi basati su messaggi CAN questi prevedevano l'utilizzo forzato dell'interfaccia grafica e quindi dell'interazione dell'utente per la configurazione di un qualsiasi parametro. I comandi accettati erano del tipo up, down, left, right, confirm e return. Altresì non era possibile interrogare lo stato di una qualsiasi variabile perché pur comunicando tramite CAN con il resto del veicolo tale infotainment non aveva fisicamente all'interno del suo database i messaggi che lo permettessero.

A questo punto l'unica soluzione per avere almeno un blocco configurabile (l'ESL rientra solo nella protezione del veicolo e non nella configurazione) è stata quella di costruire un blocco che emulasse un'infotainment avanzato con interfaccia grafica e configurabile tramite bus CAN. Per la strutturazione dei messaggi in lettura dall'infotainment mi sono ispirato a quelli utilizzati dall'infotainment Iveco.

6.1.1 L'hardware

Anche qui come per la *TBox* si è necessitato di un Raspberry Pi 3. Inoltre sono serviti due moduli, uno per la comunicazione lato CAN e uno che trasformasse il Raspberry Pi in una radio.

PiCAN2

La board *PiCAN 2* è un controller per bus CAN per il Raspberry Pi 2 e 3. Utilizza il Microchip MCP2515 unito al MCP2551 CAN transceiver. Esso comunica con il Raspberry Pi tramite il protocollo SPI sul bus SPI.0 utilizzando entrambi gli slave a disposizione e interrupt sul GPIO25. L'interfaccia verso il bus può avvenire o tramite il connettore DB9 oppure tramite le tre viti presenti sulla board. Quet'ultime rappresentano il CAN_H, CAN_L e GND. Per comodità invece di usare il connettore DB9 maschio presente sulla board ho preferito saldare i sue segnali CAN_H e CAN_L ad un connettore DB9 femmina.

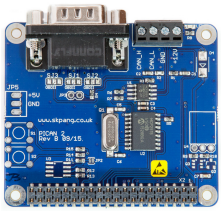


Figura 6.1. PiCAN2 [15]

TEA5767

Per quanto concerne il modulo per la ricezione di segnali FM la scelta è ricaduta sulla board *TEA5767*. Tale board è programmabile tramite comunicazione sul bus I^2C, supporta soft mute, SNC e HCC , canale audio destro/sinistro disattivabile. Ha un FM mixer per le bande FM US/Europe (87.5 MHz to 108 MHz) e Giapponese (76 MHz to 91 MHz). Ha un oscillatore interno operante a 32.768 kHz. È in grado di cercare automaticamente la miglior frequenza nell'intorno di quella selezionata scegliendo quella con segnale più forte tra le modalità mono e stereo channel.

Figura 6.2. TEA5767

6.1.2 Il software

Come per la *TBox* anche qui per il Raspberry Pi si è utilizzato il sistema operativo Raspbian e la libreria python-can [13].

htmlPy

Per l'interfaccia grafica si è utilizzata la libreria htmlPy [9] che si basa sulla libreria grafica PySide, un'implementazione del framework QT per Python. Tale libreria consente di creare in modo semplice una WebView di PySide e popolarla con un file html (supporta HTML5) a cui possono essere associati anche moduli CSS e Javascript. Infatti io ho aggiunto la libreria jQuery per eseguire delle semplici operazioni.

Driver TEA5767

Per il controllo del modulo TEA5757 che comunica con il Raspberry Pi tramite bus I^2C ho utilizzato e ampliato uno script Python realizzato da Dipto Pratyaksa (2015)[7] ripreso da uno script di Nathan Baker (2013)[11]. Allo script di Dipto Pratyaksa ho aggiunto il supporto per il mute on/off dei canali destro e sinistro e la possibilità di selezionare direttamente una frequenza FM. Altresì ho modificato la selezione della migliore modalità mono/stereo per la frequenza scelta introducendo un numero limite di tentativi. Infatti se il segnale era troppo debole, il driver continuava a ciclare alla ricerca di un segnale più forte. Superato il numero di tentativi massimi lo script ora passa alla frequenza successiva verso l'alto o verso il basso in base alla scelta iniziale dell'utente.

6.2 Il Sistema Infotainment

Rispetto agli altri sistemi (Mobile App, Server principale, TBox) questo sistema ha richiesto un numero inferiore di script python ma altrettante ore di programmazione e debug a causa della mancanza di dettagli nella documentazione per la libreria htmlPy.

6.2.1 Deamon per l'infotainment

Dovendo attivare l'infotainmente tramite messaggi CAN vi è stata la necessità di creare uno script che restasse in ascolto di tali messaggi. Lo script *startAll.py* instanzia un database del tipo *CANdatabase* popolandolo con i messaggi contenuti nel file DBC 'my_db_radio.dbc' e crea una connessione con il bus CAN, canale can0. A questo punto invece che utilizzare la classe can.Listener utilizza il costrutto *for* per leggere i messaggi all'interno del bus. Se riceve un messaggio il cui identificativo è associato al nome 'RadioCmd' e il segnale 'RadioOnOff' è uguale a uno allora il processo esegue la funzione *fork()*, il padre continua a ciclare mentre il figlio avvia un'istanza della classe *Infotainment* e quando questa istanza termina (è un'applicazione bloccante) esegue una exit() terminando se stesso.

6.2.2 I messaggi CAN di controllo

Per quanto riguarda i messaggi CAN per il controllo dell'Infotainment tramite bus ho preso spunto dalla logica degli altri costruttori di Infotainment. Vi sono due messaggi, uno per i comandi (ad evento) e uno di stato dell'infotainment (ciclico, 1 secondo), entrambi con ordinamento Intel del byte. Essi hanno gli stessi segnali ma identificativo del messaggio diverso. I segnali sono :

- Frequency, 32 bit, unsigned
- LeftVol, 1 bit, unsigned
- RightVol, 1 bit, unsigned
- RadioOnOff, 2 bit, unsigned
- Volume, 8 bit, unsigned (per sviluppi futuri)

Il segnale RadioOnOff può assumere tre valori : Off (0), On (1), Active (2). Il messaggio RadioCmd ha identificativo 0x7F0, RadioSts ha identificativo 0x7F1. quindi in caso di contesa del bus RadioCmd è trasmesso prima.

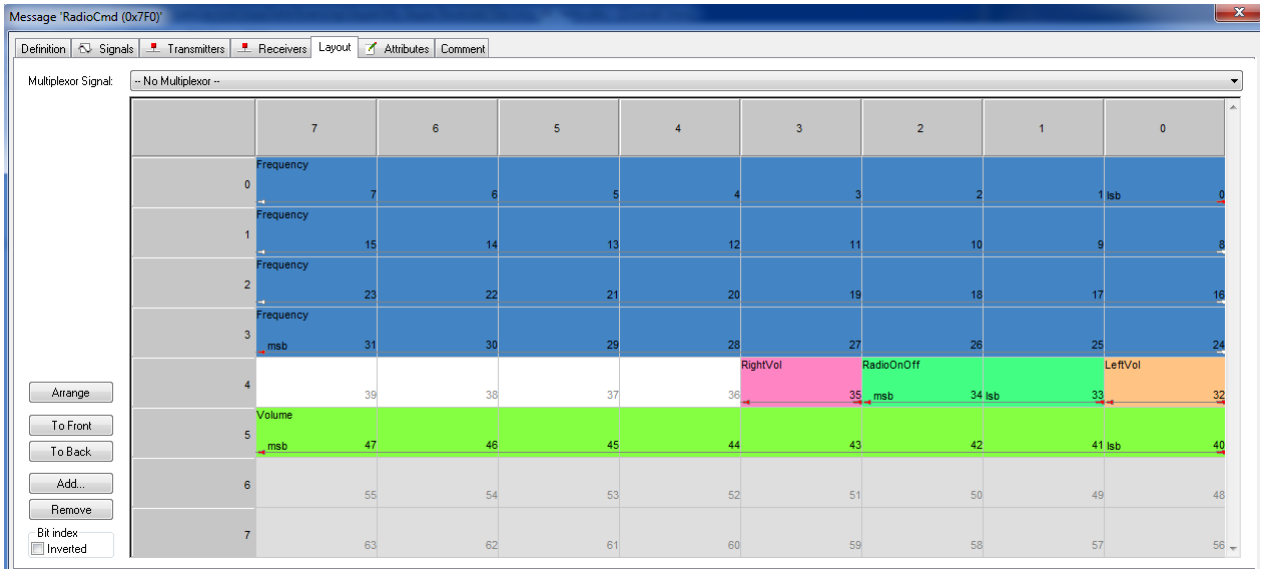


Figura 6.3. RadioCmd, CAN Message

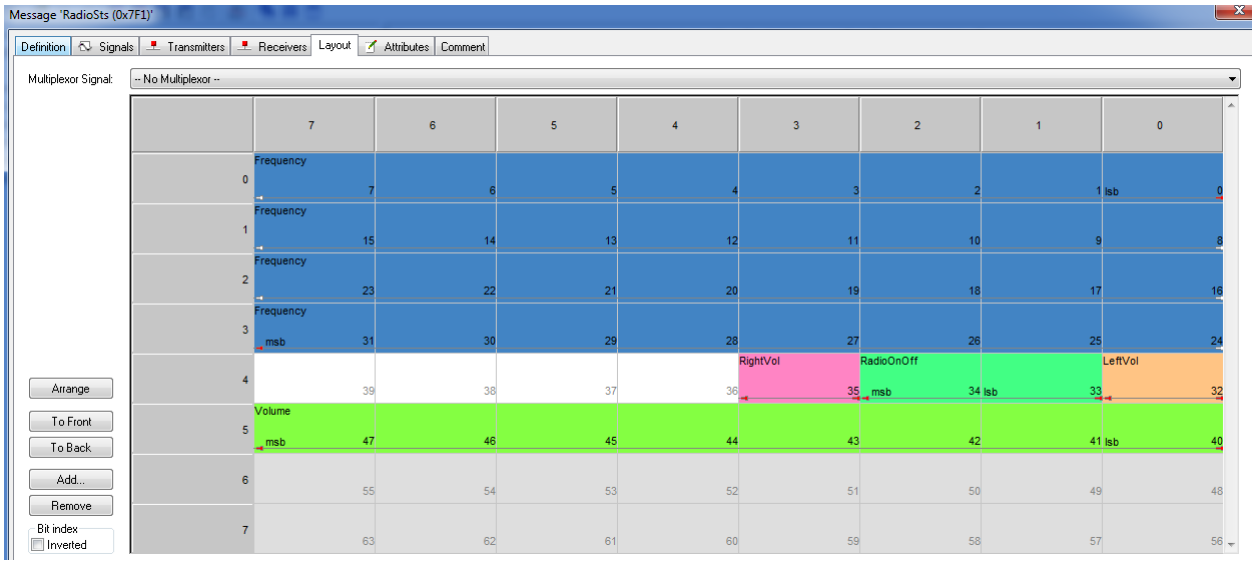


Figura 6.4. RadioSts, CAN Message

6.2.3 Il backend

Lo script back_end.py che globalmente instanzia un'oggetto di tipo *tea5767*, chiamato radio, ha tre classi principali di cui due implementano un thread e l'altra è una classe di supporto all'applicazione in htmlPy (BackEnd).

ThreadSignal

La classe ThreadSignal estende threading.Thread e ne implementa il metodo run(). Il costruttore riceve un'oggetto di tipo back_end e connette il PySide Signal

```
self.backEnd.signalStrenghtSignal
```

alla funzione di classe setSigS la quale riceve un valore e lo setta nel BackEnd tramite la self.backEnd.setSignalStrenght(v).

Qui si è applicato il paradigma Signal-Slot presente in Qt. Poiché l'interfaccia è modificabile unicamente dal thread che ha creato la GUI l'unico modo che si ha per modificare l'interfaccia da un'altro thread è quello di dichiarare dei Signal all'interno della classe BackEnd che quando collegati ad una funzione (Slot) esterna alla classe BackEnd eseguono questa funzione all'interno del thread che ha creato la GUI. Quando un'altro thread deve eseguire un'azione sulla GUI dapprima collega un Signal ad una funzione e poi quando la deve usare chiama sul Signal la funzione emit(). Il Signal è una sorta di prototipo di una funzione generica.

La funzione *run()* della classe estrae il livello del segnale tramite la variabile radio e utilizza la funzione emit() sul Signal collegato precedentemente alla sua funzione interna in modo che quella funzione sarà eseguita nel thread che gestisce la GUI

```
self.backEnd.signalStrenghtSignal.emit(x)
```

CyclicSender

La classe CyclicSender estende anch'essa la classe threading.Thread. Nel costruttore crea un'istanza del CANdatabase e instanzia il collegamento al bus CAN.

La funzione *run()* della classe invece invia ciclicamente una volta al secondo tramite il bus CAN il messaggio 'RadioSts' aggiornandone ogni volta i valori dei segnali in base a cosa l'utente modifica sulla GUI. In modo atomico (tramite un lock globale) estrae dalla classe BackEnd i valori attuali di Frequency, RightVol, LeftVol e li scrive nei segnali. Invia il messaggio e dorme per il tempo rimanente.

BackEnd

La classe BackEnd come ipotizzabile da quanto detto prima innanzitutto fa da 'model' e 'controller' per il pattern MVC della GUI, estende infatti htmlPy.Object. Globalmente

definisce una serie di Signal : frequencySignal, volumeRightSignal, volumeLeftSignal e stop.

Nel costruttore riceve l'istanza di una htmlPy.App a cui è legato (la sua View) e imposta dei valori prefefiniti per la frequenza iniziale, canale audio destro e sinistro. Imposta altresì dei limiti di potenza (max/min) e frequenza (max/min) per non eccedere nella visualizzazione i limiti reali.

Alcune funzioni di classe sono decorate con @htmlPy.Slot(), sono visibili all'interno di htmlPy.App e richiamabili tramite Javascript. Infatti questa classe è come un porting di funzionalità Python all'interno della GUI. Tra queste funzioni ritroviamo le seguenti :

- *frequencyUp(self)* aggiorna la frequenza FM corrente aumentandola di 0.1 e se supera il massimo previsto la riallinea al massimo. Fatto ciò chiama la *setFrequency()* che aggiorna la GUI e *radio.setFrequency(self.frequency)* che imposta sulla radio la frequenza voluta.
- *frequencyDown(self)* aggiorna la frequenza FM corrente diminuendola di 0.1 e se è minore del minimo previsto la riallinea al minimo. Fatto ciò chiama la *setFrequency()* che aggiorna la GUI e *radio.setFrequency(self.frequency)* che imposta sulla radio la frequenza voluta.
- *changeVolumeRight(self)* inverte lo stato del canale destro dell'audio impostandolo sulla radio e modificando di conseguenza il widget che lo rappresenta sulla GUI.
- *changeVolumeLeft(self)* inverte lo stato del canale sinistro dell'audio impostandolo sulla radio e modificando di conseguenza il widget che lo rappresenta sulla GUI.

Altre funzioni sono utilizzate per modificare l'interfaccia rispetto ai messaggi CAN pervenuti :

- *setFrequency_with_value(self,freq)* aggiorna la frequenza FM corrente impostandola al valore ricevuto. Fatto ciò chiama la *setFrequency()* che aggiorna la GUI e *radio.setFrequency(self.frequency)* che imposta sulla radio la frequenza voluta.
- *setVolumeRight(self)* inverte lo stato del canale destro dell'audio impostandolo sulla radio e modificando di conseguenza il widget che lo rappresenta sulla GUI.
- *setVolumeLeft(self)* inverte lo stato del canale sinistro dell'audio impostandolo sulla radio e modificando di conseguenza il widget che lo rappresenta sulla GUI.

Altre funzioni sono di supporto a quelle elencate prima :

- *setFrequency(self)* che aggiorna la GUI, in particolare l'elemento html con id 'freq'.
- *setSignalStrenght(self, v)* che aggiorna la ProgressBar all'interno dell'html modificando via la percentuale di riempimento della barra che la scritta sotto di essa.

6.2.4 Graphical User Interface

La GUI, Graphical User Interface, è implementata dallo script Infotainment.py. Questo script come già detto è eseguito nel contesto di un processo separato rispetto a colui che l'ha lanciato. La motivazione è dovuta al fatto che la classe htmlPy.AppGUI ha un due possibili metodi per avviare la WebView ad essa sottesa che sono start() ed execute(). Entrambi i metodi sono bloccanti e rispondono solo alle callback in formato Qt (creano una sorta di Looper) quindi non potevo eseguire la GUI e gli altri thread (CAN Listener, CyclicSender ad esempio) nello stesso processo. La differenza tra i due metodi è che il primo quando finisce chiude il processo a cui è connesso il secondo invece prosegue nell'esecuzione dello script.

ListnerThread

La classe ListnerThread implementa threading.Thread e come le altre classi a lei simili ha il metodo stop() che ne permette la terminazione. Il suo metodo run() instanzia un oggetto di tipo CanListener e lo associa ad un oggetto can.Notifier. All'invocazione del metodo stop() del thread su tale oggetto sarà chiamato il metodo self.notifier.stop().

La classe CanListener estende la classe can.Listener e come gli altri listener ne implementa il metodo on_message_received. Alla ricezione di un messaggio tramite l'istanza della classe CANdatabase ricava il nome del messaggio dall'identificatico e se questi è 'RadioCmd' in funzione del segnale 'RadioOnOff' esegue due possibii azioni. Se 'RadioOnOff' ha valore uguale a zero vuol dire che l'utente ha chiuso il veicolo e quindi anche l'Infotainment va chiuso, si chiede al BackEnd di chiudere la WebView e stoppare i vari thread e si esegue una exit(), si uccide il processo stesso.

Se 'RadioOnOff' ha valore uguale a due allora è un messaggio con i dati di configurazione relativi all'utente che ha appena sbloccato il veicolo. Da tale messaggio si ricavano la frequenza FM e gli stati dei due canali destro e sinistro. Con tali dati si eseguono delle emit() sui Signal opporturni in modo da far eseguire gli Slot al processo che gestisce l'interfaccia.

```
Frequency = int(m.getValueSignal('Frequency').to01(),2)
Rv=int(m.getValueSignal('RightVol').to01(),2)
Lv = int(m.getValueSignal('LeftVol').to01(), 2)

backEnd.frequencySignal.emit(Frequency)
backEnd.volumeLeftSignal.emit(Lv)
backEnd.volumeRightSignal.emit(Rv)
```

Gli Slot

Per aggiornare l'interfaccia e comunicare i cambiamenti al modulo TEA5767 mentre il Looper della WebView è attivo ho creato una serie di Slot, ovvero funzioni decorate con

@QtCore.Slot(). Esse richiamano delle funzioni della classe BackEnd quali stopApp(), setVolumeRight(v), setVolumeLeft(v) e setFrequency_with_value(f/10). Chi chiede di eseguirle è il thread relativo al CanListener ma sono eseguiti nel thread legato alla GUI.

La GUI

La classe Infotainment può essere considerata un mero script in quanto ha soltanto il costruttore come funzione interna. In tale unica funzione si crea e si avvia un'istanza del ListenerThread. Poi si crea la variabile *app* che è del tipo htmlPy.AppGUI. Su tale oggetto si settano height e width in funzione della dimensione dello schermo del Raspberry Pi (800x400), si dichiara globale l'istanza della classe BackEnd, la si instanzia e se ne fa il bind con la variabile *app*.A questo punto, dopo aver detto all'*app* con quale file html popolare la WebView (*app.template = ("index.html",)*) si collegano tutti i signal (Qt) con i relativi slot, si creano e si lanciano i thread per il rilevamento della potenza del segnale sulla frequenza FM attuale e quello per l'invio dell'attuale stato dell'infotainment su bus CAN. Infine si avvia la GUI con la funzione *app.start()* .

```
app = htmlPy.AppGUI(title=u"Infotainment")
app.width = 800
app.height = 480
app.template_path = "/home/pi/GUI/"
app.static_path = "/home/pi/GUI/"
backEnd = BackEnd(app)
app.bind(backEnd)
app.template = ("index.html", {})
app.start()
```

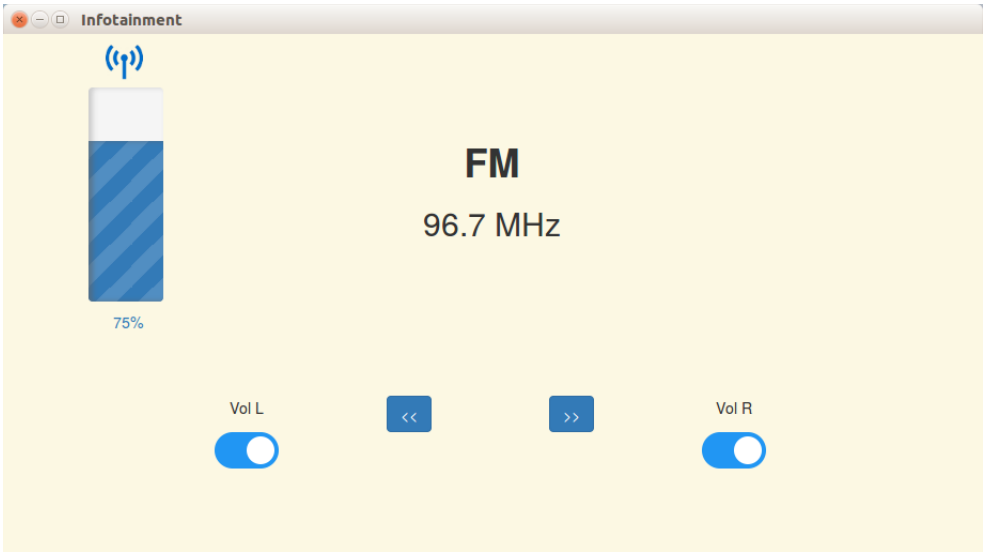


Figura 6.5. La GUI

La View

La View, mostrata in figura 6.5, è destritta in html5 all'interno del file index.html. Poiché la GUI è renderizzata all'interno di una WebView, per evitare che il Raspberry Pi debba per forza essere connesso alla rete per poter scaricare le librerie necessarie ho importato le due librerie jQuery [10] e Bootstrap [3] all'interno dell'html stesso il primo come script, il secondo come style.

Grazie alle caratteristiche del framework Bootstrap ho diviso la pagina html in righe e colonne per poter allineare le varie componenti. Sulla sinistra, in alto, ho posizionato una progress bar verticale che indica in percentuale la potenza del segnale ricevuto. Al centro della scena vi è la frequenza attuale con al di sotto due Button per aumentare o diminuire la frequenza FM corrente. Ai due estremi destro e sinistro ho posizionato due toggle button che permettono di attivare o disattivare i canali audio destro e sinistro.

Sia ai due Button che ai due toggle sono collegati dei listener sulla funzione onClick. L'implementazione dell'onClick è una semplice riga Javascript che richiama un metodo della classe BackEnd che è vista come se fosse una classe Javascript ma in realtà è un ponte di collegamento che consente di chiamare delle le specifiche funzioni Python decorate con @htmlPy.Slot() direttamente dall'ambiente Javascript stesso.

6.3 Interfacciamento con la board TEA5767

Come specificato nella descrizione dell'hardware lo script Python per il controllo della scheda TEA5767 è stato solo modificato da me e creato da Dipto Pratyaksa [7] e Nathan Baker [11].

Inizialmente lo script non funzionava, dopo vari tentativi ho trovato l'errore. Nelle versioni precedenti del Raspberry Pi (2, 1) il bus di default era quello contraddistinto dal numero zero. Nella versione 3 del Raspberry Pi, con l'individuazione automatica dei dispositivi che supportano l'I^2C tramite il tool *i2c* il bus di default è quello con il numero uno. Per fortuna dopo varie ricerche dove la documentazione era datata e imprecisa ho deciso di controllare l'implementazione della libreria di sistema *smbus*. Qui ho trovato, come era ipotizzabile, che il costruttore può ricevere come parametro anche il numero del bus. Modificando tutti i punti in cui veniva instanziato tale oggetto sono riuscito a far partire lo script e a far funzionare in modo standard il modulo TEA5767.

6.3.1 Il Protocollo I^2C e la comunicazione con il modulo

Il protocollo I^2C , nella versione base single-master multi-slave, prevede l'utilizzo di due cavi SDA e SCL il primo cavo trasposta l'indirizzo dello slave da attivare e i dati e il secondo è un segnale di clock controllato dal master a cui gli slave si allineano.

La comunicazione inizia con il segnale di START (SDA basso, SCL alto) e trasmette sette bit che rappresentano l'indirizzo dello slave seguito dall'ottavo che indica se la trasmissione è una read (1) o write (0). Le operazioni sono intese dal punto di vista del master. A seguire vi è un bit di ACK da parte dello slave e seguono frame di ottetti di bit a cui segue un bit di ACK da parte di chi riceve. Quando i frame di dati sono terminati si invia il segnale di STOP (SCL low to high e rimane high, segue SDA low to high).

L'indirizzo del modulo è lo 0x60.

TEA5767 Write Mode

La configurazione della TEA5767 richiede l'invio di 5 byte :

- Primo: Mute, Search Mode, parte alta della frequenza
- Secondo: parte bassa della frequenza
- Terzo : Search Up/Down, Mono/Stereo, Mute Right, Mute Left
- Quarto: StandBy, Band Limits, Soft Mute
- Quinto : PLLREF

TEA5767 Read Mode

La lettura dello stato della TEA5767 richiede la ricezione di 5 byte :

- Primo: Ready, Band Limit, parte alta frequenza attuale
- Secondo: parte bassa della frequenza attuale
- Terzo: Stereo, PLL
- Quarto: Level (potenza segnale)
- Quinto : Non Usato

Utilizzando il datasheet [6] ho dedotto il significato delle varie parti del codice di Dipto Pratyaksa e ho individuato il punto in cui vengono settati i byte per il cambio di un qualsiasi parametro. Lì ho aggiunto il settaggio dei bit relativi al Mute Right e Mute Left che prima erano sempre impostati al valore zero ovvero sempre attivi.

```
if ml==1:
    data[1]=data[1]+0b00000100
if mr==1:
    data[1] = data[1] + 0b00000010
```

Ho creato due funzioni che permettono di modificare lo stato dei due canali destro e sinistro lasciando invariato il resto della configurazione

```
def muteR(self):
  if(self.muteRFlag):
    self.writeFrequency(self.calculateFrequency(), 0,0,0,self.
        muteLFlag)
    self.muteRFlag=0
    print("unmuteR")
  else:
    self.writeFrequency(self.calculateFrequency(), 0,0,1,self.
        muteLFlag)
    self.muteRFlag=1
    print("muteR")
  return ("radio mutedR")

def muteL(self):
  if(self.muteLFlag):
    self.writeFrequency(self.calculateFrequency(), 0,0,self.
        muteRFlag,0)
    self.muteLFlag=0
    print("unmuteL")
  else:
    self.writeFrequency(self.calculateFrequency(), 0,0,self.
        muteRFlag,1)
    self.muteLFlag=1
    print("muteL")
  return ("radio mutedL")
```

Infine ho copiato e modificato la funzione writeFrequency() creando la setFrequency(f) che sfrutta la precedente ma riusa i parametri correnti per canale destro e sinistro dell'audio per impostare la frequenza voluta.

Capitolo 7

Conclusioni

7.1 Valutazioni

7.1.1 Test delle funzionalità

In data 12 settembre 2017, presso l'RF Lab sito al terzo piano dell'azienda, alla presenza della Manager Basciu e di altri due dipendenti della Concept Reply, si sono svolti i test atti a verificare le funzionalità specificate nei requisiti di sistema.

Precondizioni

Strumentazione utilizzata:

Mobile App	:	Smartphone Honor 6X, Android 7.0
Server	:	Asus K55
TBox	:	Raspberry Pi 3 (A) with PiCAN GPS
Infotainment	:	Raspberry Pi 3 (B) with PiCAN2, TEA5767
ESL	:	TRW ESL
QRCode	:	immagine sul PC con valore AA000AA
Beacon	:	model 3M VH, id x:800:1

Si accende il WiFi, si avvia il Server. Si connettono tramite i connettori CAN l'ESL, la *TBox* , l'Infotainment.

TBox Test

Test 1 *R 1.6, R 2.1*
Si connette la *TBox* all'alimentazione (caricatore micro USB 5.0 V, 2 A). Si attende che il led del PiCAN finisca di lampeggiare. A questo punto il segnale GPS è stato rilevato.

Si avvia il programma della *TBox* . Il server riceve la posizione GPS, viene stampata su terminale. Tramite terminale si analizza il contenuto del DB e la posizione risulta aggiornata. **Test Passato**

Test 2 *R 2.6*
A *TBox* accesa e posizione GPS inviata si esegue il login dalla Mobile App e si sblocca il veicolo.

Si inserisce la chiave, la si porta in posizione ON, la si porta in posizione OFF, l'ESL esegue un lock, il server notifica tramite terminale che il veicolo è libero. Transizione rilevata, LOCK eseguito. **Test Passato**

Test 3 *R 2.2*
A *TBox* accesa e posizione GPS inviata si esegue il login dalla Mobile App e si chiede lo sblocco del veicolo.

All'atto della connessione tra *TBox* e server l'ESL effettua un UNLOCK. La *TBox* ha inviato correttamente il messaggio CAN di UNLOCK **Test Passato**

Test 4 *R 2.3*
A *TBox* accesa e posizione GPS inviata si esegue la registrazione dalla Mobile App e si chiede lo sblocco del veicolo.

All'atto della connessione tra *TBox* e server l'Infotainment si accende e si autoconfigura con i parametri di default, cioè canali destro e sinistro abilitato, frequenza FM settata su 102.5. **Test Passato**

Test 5 *R 2.4, R 2.5*
A *TBox* accesa e posizione GPS inviata si esegue la registrazione (Utente Antonio) dalla Mobile App e si chiede lo sblocco del veicolo.

All'atto della connessione tra *TBox* e server l'Infotainment si accende e si autoconfigura con i parametri di default, si imposta la frequenza FM a 101.0, si disabilita il canale audio destro. Si spegne il veicolo tramite chiave. Il server riceve la nuova configurazione. Tramite tarminale si controllano le tabelle del DB, i nuovi valori sono correttamente salvati. **Test Passato**

Test 6 *R 2.3*
A *TBox* accesa e posizione GPS inviata si esegue la registrazione dalla Mobile App e si chiede lo sblocco del veicolo.

All'atto della connessione tra *TBox* e server l'Infotainment si accende e si autoconfigura con i parametri di default. Si spegne il veicolo e si riesegue lo sblocco tramite login dell'utente Antonio. L'infotainment imposta la frequenza FM a 101.0, il canale audio destro è disabilitato. **Test Passato**

Server Test

Test 7 *R 1.1*
Avvia la Mobile App, si effettua il logout. Si procede alla registrazione

Username : Test-01
Password : abcdefg

Il server riceve la richiesta e risponde con codice 201. La Mobile App prosegue con la schermata successiva, la registrazione è avvenuta correttamente. Si riceve il codice alfanumerico per il recupero della password. Tramite terminale si analizza il contenuto del DB, l'utente risulta inserito. **Test Passato**

Test 8 *R 1.2*
Avvia la Mobile App, si effettua il logout. Si effettua lo switch sulla pagina per il login, si clicca sul pulsante per il recupero della password. Nella schermata che appare si inserisce l'username (Test-01), il codice di recupero e la nuova password. Il server processa la richiesta e ritorna il codice 200. Tramite terminale si analizza il contenuto del DB, la password risulta cambiata. **Test Passato**

Test 9 *R 1.3*
Avvia la Mobile App, si effettua il logout. Si procede alla registrazione

Username : Test-01
Password : hjklpoi

Il server riceve la richiesta e risponde con codice 412. La Mobile App notifica all'utente che la registrazione è fallita, username duplicato. **Test Passato**

Test 10 *R 1.5*
Avvia la Mobile App, si effettua il login. Si attende 1 minuto prima di inquadrare il QR code. Il server riceve la richiesta e risponde con codice 412. La Mobile App notifica all'utente che lo sblocco è fallito. Il motivo non viene mostrato all'utente. **Test Passato**

Test 11 *R 1.10*
Si posiziona il beacon alle distanze indicate in tabella rispetto allo smartphone. Si avvia la Mobile App, si effettua il login. Si inqudra il QR code, in caso di sblocco si spegne il veicolo. Il server deve rifiutare la richiesta di sblocco se la distanza dal veicolo dell'utente è superiore ai 3 m .

Segue una tabella che riassume i test effattuati e riporta l'esito della richiesta (posizione GPS e id bluetooth fissi)

Dist. reale	**Dist. misurata**	**Esito**
2.28 m	2.20 m	Accettata
3.98 m	3.75 m	Rifiutata
1.21 m	0.97 m	Accettata
4.85 m	4.53 m	Rifiutata

Il test per ciascuna casistica rispetta l'esito previsto. **Test Passato**

Test 12 *R 1.10*
Si vuole verificare che nel caso in cui le posizioni GPS del veicolo e dell'utente non sono compatibili la richiesta di sblocco deve essere respinta dal server .

Test non possibile all'interno dell'edificio, si è avuto conferma del requisito sfruttando un breve lasso di tempo in cui al riavvio della *TBox* (la fase di fixing della posizione) il GPS del veicolo invia posizioni approssimative. **Test non eseguibile**

Test 13 *R 1.10*
Si vuole dimostrare che l'utente deve essere più vicino al veicolo della flotta che vuole sbloccare rispetto agli altri eventualmente presenti. Al veicolo con targa AA000AA è associato l'identificativo bluetooth x:800:1. Si dispone di un altro beacon con id x:800:2. Non si riporta il valore di x in quanto è l'identificativo primario utilizzato dall'azienda. Si avvia la Mobile App e si esegue il login, si procede con la richiesta di sblocco.

Segue una tabella che riassume i test effattuati e riporta l'esito della richiesta (posizione GPS fissa)

A minor distanza	**Esito**
x:800:1	Accettata
x:800:2	Rifiutata

Il test per ciascuna casistica rispetta l'esito previsto. **Test Passato**

Mobile App Test

Test 14 *R 1.4, R 1.1*
Si Avvia la Mobile App, si effettua il logout. Nella schermata che appare (registration) si inseriscono i seguenti dati

Username : Test-03
Password : abcdefg

Appare la schermata del Fingerprint, provo ad autenticarmi con un dito non registrato sul device, l'app non prosegue con la richiesta, provo ad autentcarmi con un dito registrato.

Il server riceve la richiesta e risponde con il codice 201 (visibile sul terminale). **Test Passato**

Test 15 *R 1.4, 1.14*
Si Avvia la Mobile App, si è già registrati. Nella schermata che appare (mappa veicoli) clicco sul pulsante per autenticami, uso un dito registrato. Il login viene correttamente eseguito, il server ritorna 200. L'app prosegue sulla schermata successiva. **Test Passato**

Test 16 *R 1.11*
Si Avvia la Mobile App, si è già registrati. Nella schermata che appare (mappa veicoli) clicco sul pulsante per autenticami, uso un dito registrato. Il login viene correttamente eseguito. Scelgo il veicolo disponibile inquadrandone il QR code. le condizioni sono soddisfatte. Il veicolo è sbloccato. A veicolo ancora in uso chiudo la Mobile App. La riapro. Il veicolo non è disponibile. Mi autentico, scannerizzo il QR code, la richiesta è rigettata. **Test Passato**

Test 17 *R 1.15*
Si Avvia la Mobile App, si è già registrati. Nella schermata che appare (mappa veicoli) clicco sul pulsante per autenticami, uso un dito registrato. Il login viene correttamente eseguito. Scelgo il veicolo disponibile inquadrandone il QR code. le condizioni sono soddisfatte. Il veicolo è sbloccato. A veicolo ancora in uso chiudo a Mobile App. La riapro. Il veicolo non è disponibile. Spengo il veicolo (chiave da ON a OFF). Sulla mappa appare nuovamente il veicolo. **Test Passato**

7.1.2 Maturità del progetto

Il POC da me realizzato rispetto ai test effettuati sulle funzionalità soddisfa l'obiettivo della tesi.

Il sistema è una buona base di partenza per la costruzione di un sistema reale che sia attivo a regime. Il software riguardante il server e la Mobile App può essere integrato all'interno dei progetti che attualmente gestiscono i sistemi di *Car Sharing* con i necessari adattamenti. Si è dimostrata la possibilità di interagire direttamente con una ECU bypassando di fatto il sistema a bordo del veicolo e affiancandosi ad esso. La *TBox* può essere inserita sia come modulo fisico a bordo del veicolo sia (molto più probabilmente) come estensione software di una ECU già presente.

Se però si analizza la *TBox* all'interno del mondo automotive essa al momento non è direttamente integrabile a bordo di un veicolo. In primo luogo necessiterebbe di una lunga fase di testing e bisognerebbe implementare un sistema di sleep/wake up per soddisfare i requisiti di basso consumo richiesti da un veicolo.

In secondo luogo i messaggi CAN utilizzati sono basati su uno specifico DB per uno specifico veicolo. La segretezza dei DB CAN si riflette anche nello sviluppo della *TBox* infatti non essendoci messaggi e segnali generici interpretabili da qualsiasi veicolo essi vanno scritti brutalmente nel codice, senza poter avere un livello di astrazione e quindi di riutilizzabilità del codice stesso. Ciò vuol dire che per ogni tipo di veicolo diverso va implementata una classe *CanModule.py* diversa.

7.2 Sviluppi futuri

Innanzitutto il primo possibile sviluppo futuro del POC deve essere orientato all'integrazione delle parti meccaniche previste all'inizio ovvero sedili e specchietti. In questo modo l'utente avrebbe un reale coinvolgimento nel processo di autoconfigurazione, migliorandone l'esperienza di guida.

Un altro componente che potrebbe essere integrato è il climatizzatore. Si potrebbero interpolare i dati della posizione dei finestrini, della temperatura interna ed esterna, del sensore di pioggia, del sensore di umidità e della temperatura target del climatizzatore.

In uno scenario non molto lontano nel tempo, avendo a disposizione i sensori appena citati si potrebbero acquisire i dati prodotti e su di essi applicare un algoritmo di Machine Learning al fine di trovare le condizioni di pioggia/temperatura/umidità che portano quello specifico utente ad aprire/chiudere i finestrini o ad attivare il climatizzatore per poterlo successivamante anticipare in queste azioni.

Attraverso un ampliamento delle funzionalità della Mobile App si potrebbe tracciare il percorso dell'utente e in base alla conoscenza della destinazione si potrebbe ridurre

la differenza tra la temperatura esterna ed interna al fine di diminuire lo sbalzo termico all'uscita del veicolo. Ciò richiederebbe un tracciamento delle abitudini dell'utente.

Le possibili evoluzioni del POC appena descritte sono tutte sfumature di una prima applicazione dell' IoT all'interno del veicolo volto a migliorare l'esperienza di guida.

Si noti però che nella descrizione della *TBox* ho lasciato abbastanza vaga la descrizione dell'hardware relativo all'interfaccia di rete concentrandomi di più a spiegare il lato software della stessa. Per il POC tale interfaccia è quella Wifi. Nel contesto reale però non si può ipotizzare che la *TBox* sia connessa alla rete tramite il Wifi ma bisognerà integrare sul Raspberry Pi un modulo GPRS / GSM che attraverso una scheda SIM consenta di collegarlo ad Internet.

Per quanto riguarda invece le interferenze elettromagnetiche, una volta completato il modulo con tutte le funzionalità appena descritte, bisognerebbe certificarlo in modo da poterlo mettere in commercio. Ovviamente per una versione commerciale del prodotto sarebbe auspicabile la creazione di un SoC progettato appositamente che integri internamente i moduli CAN, GPS, GPRS, sensore di temperatura e umidità.

Risulta inoltre necessario assicurarsi che chi si registra sia autorizzato a guidare ovvero che disponga di un documento di guida valido che comporterebbe anche l'autenticazione dell'utente stesso. Per far ciò bisognerà estendere la fase di registrazione nella Mobile App aggiungendo la possibilità di caricare sul Server Centrale una copia del documento di guida, si dovrà introdurre un tempo intermedio in cui la registrazione non sarà pienamente attiva (utente registrato ma impossibilitato a sbloccare un veicolo) che consenta la verifica del documento.

Sempre a riguardo della fase di registrazione sarebbe opportuno far ripetere all'utente la password in modo da essere sicuri che non abbia commesso errori nell'inserirla. Altresì si dovrebbe portare la lunghezza minima della password da 4 ad almeno 8 caratteri. Per motivi di debugging veloce, trattandosi di un POC, si è ritenuto di non modificare tale constraint che in un contesto reale è fondamentale.

7.3 Il POC nel contesto automotive attuale

Alla luce di quanto già precedentemente discusso sul fatto che il mercato automotive è in procinto di subire una evoluzione radicale, il sistema realizzato è direttamente implementabile sulla direzione intrapresa dai produttori di veicoli.

La tendenza è quella di realizzare un solo modello per un determinato veicolo che abbia a bordo tutto l'hardware necessario del modello top di gamma. Il software sarà installato per tutte le componenti ma le funzionalità saranno differenziate in base al prezzo, abilitandole via software con la possibilità di attivarle anche dopo l'acquisto del

veicolo stesso. Ciò permetterà una semplificazione della linea produttiva e una maggiore facilità di rilascio degli aggiornamenti software (via OTA).

Un esempio è l'aggiornmento rilasciato dalla Tesla per aumentare la capacità delle batterie per automobili elettriche da 60 kWh a 70 kWh per facilitare gli spostamenti agli abitanti della Florida prima dell'arrivo dell'uragano Irma (settembre 2017).

Ecco che quindi il software da statico diventa dinamico, facilmente aggiornabile e riutilizzabile. L'integrazione di una componente che comunichi in remoto a bordo di tali veicoli diventa estremamente semplice essendoci già una base software molto stabilizzata.

Non va tralasciato il fatto che mettendo le ECU del veicolo sulla rete internet esso sarà esposto ad attachi informatici e per tale motivo bisognerà focalizzare l'attenzione sulla sicurezza delle comunicazioni.

Un progetto simile al POC da me realizzato è stato sviluppato da poco dalla BMW, in Europa e in America la strada è stata tracciata, in Italia al momento sembra che vi siano prototipi a livello di POC su modifiche ed evoluzioni dell'attuale sistema di *Car Sharing*, ma ancora non è stato annunciato alcun progetto a livello aziendale.

Bibliografia

[1] Android developer. https://developer.android.com. Accessed: 2017-04-30.

[2] Android get address with street name. http://javapapers.com/android.

[3] Bootstrap. http://getbootstrap.com/.

[4] Car sharing, wikipedia. https://it.wikipedia.org/wiki/Car_sharing.

[5] Car2go. https://www.car2go.com.

[6] Datasheet tea5767. https://www.sparkfun.com/datasheets/Wireless/General/TEA5767.pdf.

[7] Dipto pratyaksa for linuxcircle.com. https://github.com/LinuxCircle/tea5767/blob/master/tea5767stationscanner.py.

[8] Enjoy eni. https://enjoy.eni.com/it.

[9] htmlpy. http://htmlpy.readthedocs.io/.

[10] jquery. https://jquery.com/.

[11] Nathan baker. https://github.com/pcnate/fm-radio-python/blob/master/fm-radio-python/tea5767.py.

[12] Pican gps and accelerometer. https://skpang.co.uk/catalog/images/raspberrypi/pican/IMG_0002.jpg.

[13] Python-can. https://python-can.readthedocs.io/en/latest/.

[14] Raspberry pi 3. https://www.raspberrypi.org/app/uploads/2017/05/Raspberry-Pi-3-1-1619x1080.jpg.

[15] Sk pang electronics. https://skpang.co.uk.

[16] Ufficio studi ics, 2005.

[17] Vector signal multiplexing. https://vector.com/portal/medien/cmc/application_notes/AN-ION-1-0521_Extended_Multiplexing_in_DBC_Databases.pdf.

[18] Muayyad Alsadi. Fast concurrent pool of preforked-processes and threads mixin for python's socket server. https://github.com/muayyad-alsadi/python-PooledProcessMixIn.

[19] Muayyad Saleh Alsadi. Fast concurrent pool of preforked-processes and threads mixin for python's socket server. https://github.com/muayyad-alsadi/python-PooledProcessMixIn.

[20] Alexandra Dmitrienko Christian Plappert. Secure free-floating car sharing for offline cars. 2017. doi:http://dx.doi.org/10.1145/3029806.3029807.

[21] Marco Mastretta Claudia Burlando. Car sharing, analisi economica e organizzativa del settore. http://www.icscarsharing.it/wp-content/uploads/2017/01/Libro_Car_Sharing_analisi_economica_e_organizzativa_del_settore.pdf.

[22] IETF. Suite b profile for transport layer security (tls). https://tools.ietf.org/html/rfc6460.

[23] Bart Preneel Iraklis Symeonidis, Mustafa A. Mustafa. Keyless car sharing system. At: KU Leuven, ESAT-COSIC and iMinds, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium.

[24] Joseph Kulundai. Android beacon library. https://altbeacon.github.io/android-beacon-library/.

[25] Giuseppe Maggi. Leggere codici a barre e qr code. http://www.html.it/pag/63571/leggere-codici-a-barre-e-qr-code/. Created: 2017-03-08.

[26] NSA. Suite b cryptography. http://csrc.nist.gov/groups/SMA/ispab/documents/minutes/2006-03/E_Barker-March2006-ISPAB.pdf.

[27] Professor Stefan Bratzel Strategy&. Study at center of automotive management (cam).

[28] Lars Vogel. Android location api with the fused location provider. http://www.vogella.com/tutorials/AndroidLocationAPI/article.html.