POLITECNICO DI MILANO
FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

I.DRIVE: Progetto e sviluppo di un data logger multisensoriale per studi di interazione veicolo conducente

Relatore: Ing. MATTEO MATTEUCCI

Tesi di laurea di:
ALESSANDRO GABRIELLI
Matr. 801067

Anno Accademico 2014 - 2015

Alla mia famiglia.

Prefazione

In questa tesi è stato progettato e sviluppato un sistema per l'acquisizione e il salvataggio di dati provenienti da diversi sensori installati su una autovettura. I sensori a bordo del veicolo consentono di registrare le prestazioni e reazioni del guidatore nelle varie condizioni di guida e l'acquisizione di dati ambientali. Per realizzare l'architettura software è stato utilizzato il framework ROS (Robot Operating System). Questo framework è molto conosciuto e usato nell'ambito della robotica vista l'elevata modularità e l'utilizzo di un modello publish/subscribe per la comunicazione tra i vari moduli. Inoltre ROS fornisce utili strumenti per lo sviluppo e il debug.

Per la parte di salvataggio dei dati è stato utilizzato MongoDB, un Database NoSQL basato su documenti dinamici (BSON) che fornisce ottime performance e un modello dei dati dinamico ovvero non legato a uno schema fisso. Il sistema presenta anche la possibilità di riprodurre in ROS i dati salvati all'interno del database MongoDB così da poterli visualizzare nuovamente.
In questa progetto è stata posta particolare attenzione alla sincronizzazione dei clock dei vari componenti del sistema e alla affidabilità e facile individuazione di eventuali malfunzionamenti. Per la sincronizzazione di alcuni tra i sensori presenti sono stati utilizzati i protocolli di sincronizzazione PTP e NTP. Per la gestione dell'affidabilità e l'individuazione dei malfunzionamenti è stato sviluppato un modulo software che si occupa della gestione dello stato dei vari nodi software presenti nel sistema. Per rappresentare lo stato di ogni nodo è stata utilizzata una macchina a stati finiti.

Il nostro sistema è stato testato in laboratorio e ha mostrato una buona affidabilità permettendo l'acquisizione di tutti i dati provenienti dai sensori. Nei test effettuati è emerso che la quantità di dati salvata è di circa 10 GB ogni 5 minuti.

Abstract

an acquiring and storing system of data coming from different sensors installed on a car is designed and developed in this thesis. The sensors inside the car allow to record the performance and the reactions of the driver in different driving conditions and to acquire environment data.
The development of the software architecture has been done using the ROS(Robot Operating System) framework. This framework is well-known and it is used in the robotics field due to its high modularity and it has a publish-subscribe model for the communication between modules. ROS also provides useful tools for developing and debugging.

For the data storage part has been used MongoDB, a NoSQL Database based on dynamic documents (BSON) that provides very good performance and a dynamic data model. The system also has the possibility to replay, in ROS, the data saved in the MongoDB database to see them again.
In this project has been done particular attention on clock synchronization for the different system devices and on the reliability and easy identification of malfunctioning. PTP and NTP synchronization protocols are used for the synchronization of some of the sensors. A software module that manages the different states of the software nodes present in the system is developed to manage the affidability and the identification of malfunctioning. To represent the state of each node has been used a finite state machine.

Our system has been tested in the lab and has shown a good reliability allowing the acquisition of all data coming from the sensors. In the tests it is discovered that the amount of data stored is approximately 10 GB every 5 minutes.

Ringraziamenti

Voglio ringraziare tutta la mia famiglia, in particolar modo i miei genitori, Roberto e Patrizia, che mi sostengono quasi sempre in tutto quello che faccio ed è grazie a loro se sono diventato quello che sono e se ho avuto la possibilità di intraprendere e completare questo percorso.
Ringrazio la mia amica Stefania e il mio amico Damiano per l'affetto, il sostegno e la pazienza che hanno dimostrato verso di me in quest'ultimo periodo e per essere stati sempre presenti nei momenti di difficoltà.
Ringrazio inoltre tutti i miei amici e tutte le persone che ho conosciuto durante questi anni di studi e che hanno reso questo percorso indimenticabile, in particolar modo, Giulia, Giorgia e Silvia perché sono riuscite a rendere le giornate di lavoro sulla tesi meno pesanti.
Infine voglio ringraziare il professor Matteo Matteucci che mi ha dato la possibilità di lavorare a questa tesi.

Indice

Elenco delle figure **XI**

Elenco delle tabelle **XIII**

1 Introduzione **1**

2 Stato dell'arte **5**

2.1 Veicoli autonomi 5
2.2 Auto autonome 6
2.3 Progetti esistenti 8
2.3.1 I primi veicoli autonomi 8
2.3.2 DARPA Grand Challenge 12
2.3.3 Auto autonome moderne 15
2.3.4 Progetti riguardanti l'analisi del guidatore 18
2.4 Robot Operating System 20
2.5 MongoDB 23

3 L'hardware del veicolo I.DRIVE **27**

3.1 Il Veicolo I.DRIVE 27
3.2 Sensori ambientali 28
3.2.1 LIDAR 28
3.2.2 Ricevitore GPS 31
3.2.3 IMU 35
3.2.4 Telecamere esterne 36
3.3 Sensori per il guidatore 37
3.3.1 Sensori per l'acquisizione di parametri fisiologici 37
3.3.2 Telecamere interne 40
3.4 Computer 41
3.5 Analisi dei consumi del sistema 42

4 Architettura Software **43**

4.1 Architettura generale 43
4.2 Log dei dati in MongoDB tramite ROS 44

4.3 Sensori ambientali 46
4.3.1 Velodyne 46
4.3.2 GPS Velodyne 48
4.3.3 IMU 52
4.3.4 Prosilica GC1020 54
4.4 Sensori per il guidatore 58
4.4.1 Procomp Infiniti 58
4.4.2 Empatica E4 60
4.4.3 Axis P1343 63
4.5 Visualizzazione dei dati 65
4.5.1 RVIZ 65
4.5.2 ROS e Pylab 68
4.6 Sistema di controllo dello stato dei nodi ROS 69
4.6.1 Macchina a stati 69
4.6.2 Heartbeat 71
4.6.3 Visualizzazione dello stato dei nodi 73
4.7 Sincronizzazione 74
4.7.1 Sincronizzazione del PC 74
4.7.2 Sincronizzazione delle Prosilica GC1020 77
4.7.3 Sincronizzazione degli altri sensori 79

5 Risultati **81**
5.1 Descrizione dell'intero sistema 81
5.2 Dati sperimentali 81

6 Conclusioni e sviluppi futuri **85**
6.1 Conclusioni 85
6.2 Sviluppi futuri 85

A Manuale di installazione e avvio **87**
A.0.1 Requisiti 87
A.0.2 Installazione e configurazione software 87
A.0.3 Avvio del sistema 90

Bibliografia **93**

Elenco delle figure

2.1 Alcuni dei primi prototipi di veicoli autonomi: (a) Shakey, (b) Stanford Cart, (c) DARPA Autonomous Land Vehicle, (d) Ground Surveillance Robot 10
2.2 I primi prototipi di auto autonome: (a) VaMP, (b) ARGO 11
2.3 I partecipanti alla DARPA Grand Challenge del 2005 : (a) Stanley, (b) Sandstorm, (c) H1ghlander, (d) kAT-5 14
2.4 I partecipanti alla DARPA Urban Challenge del 2007 : (a) Boss, (b) Junior, (c) Odin, (d) Talos 16
2.5 Esempi di auto autonome moderne : (a) AnnieWAY, (b) Deeva, (c) MadeInGermany, (d) Prototipo sviluppato all'interno del progetto Google car, (e) Modello Lexus all'interno del progetto Google car 19
2.6 Struttura base di ROS 22
2.7 Esempio di schermata di RVIZ 23
2.8 Esempio di schermata rqt con diversi plugin visualizzati 24
2.9 Struttura dati di un database MongoDB 25
2.10 Esempio di un documento MongoDB 25

3.1 Foto del veicolo I.DRIVE: Tazzari zero 28
3.2 LIDAR: (a) Velodyne HDL-32E, (b) Esempio di nuvola di punti acquisita dalla Velodyne HDL-32E 29
3.3 Sensori aggiuntivi della Velodyne HDL-32E: (a) Sistema di riferimento dell'IMU, (b) Ricevitore GPS Garmin 18LV 30
3.4 Formato del pacchetto UDP dei laser (Porta 2368) 32
3.5 Formato del pacchetto UDP del GPS e dell'IMU (Porta 8308) . . 32
3.6 GPS Yuan10 con ricevitore Skytraq S1315F-RAW 33
3.7 Xsens MTi con sovrapposto il sistema di riferimento fisso 36
3.8 Camera Prosilica: (a) Prosilica GC1020, (b) Ottica Theia MY125M 37
3.9 Esempio di immagini acquisite dalle prosilica: (a) Immagine acquisita tramite l'ottica theia, (b) Immagine acquisita tramite l'ottica standard 38

3.10 Sensore per i parameteri fisiologici: (a) Procomp Infiniti, (b) Esempi di sensori disponibili per la Procomp Infiniti: in alto partendo da sinistra abbiamo il sensore per acquisire i dati riguardanti EMG, EKG, EEG e conduttanza della pelle; in basso partendo da sinistra abbiamo i sensori per acquisire temperatura corporea, respirazione e battito cardiaco 39
3.11 Braccialetto Empatica E4 40
3.12 Camera Axis P1343: sono presenti 2 telecamere sul veicolo I.DRIVE, una che punta verso il guidatore e una che punta verso la strada . 41
3.13 Computer Shuttle Slim 42

4.1 Schema che mostra come la macchina virtuale sia usata per acquisire i sensori fisiologici nel nostro sistema rispetto agli altri sensori 44
4.2 Schema BSON del topic `velodyne_points` utilizzato dal database MongoDB 49
4.3 Schema BSON del topic `velodyne_packet` utilizzato dal database MongoDB 50
4.4 Schema BSON del topic `imu_data` utilizzato dal database MongoDB 51
4.5 Schema dei nodi ROS utilizzati per estrarre i dati dal GPS della Velodyne 52
4.6 Schema BSON del topic `fix` utilizzato dal database MongoDB . . 53
4.7 Schema BSON del topic `vel` utilizzato dal database MongoDB . . 53
4.8 Schema BSON del topic `time_reference` utilizzato dal database MongoDB 54
4.9 Schema BSON del topic `/xsens/imu` utilizzato dal database MongoDB 55
4.10 Schema BSON del topic `/xsens/mag` utilizzato dal database MongoDB 55
4.11 Finestra di calibrazione della camera 57
4.12 Schema BSON dei topic `prosilica1/image_raw` e `prosilica2/image_raw` utilizzato dal database MongoDB 58
4.13 Schema BSON dei topic `prosilica1/camera_info` e `prosilica2/camera_info` utilizzato dal database MongoDB 59
4.14 Schema BSON del topic `procomp_sensor` utilizzato dal database MongoDB 61
4.15 Schema BSON del topic `empatica_sensor` utilizzato dal database MongoDB 64
4.16 Schema BSON dei topic `axis1/image_raw/compressed` e `axis2/image_raw/compressed` utilizzato dal database MongoDB 65
4.17 Finestra RVIZ utilizzata nel nostro sistema: a sinistra le 4 finestre del tipo "Image" e a destra la finestra del tipo "PointCloud2" . . 67

4.18 Configurazione del veicolo I.DRIVE che ha catturato le immagini visualizzate nella precedente finestra di RVIZ 67
4.19 Finestra utilizzata nel nostro sistema per visualizzare i dati del sensore Procomp Infiniti 69
4.20 Finestra utilizzata nel nostro sistema per visualizzare i dati del sensore Empatica E4 70
4.21 Macchina a stati finiti 71
4.22 Finestra generale in cui vengono visualizzati gli stati dei nodi e le statistiche dei topic 73
4.23 Dettaglio della finestra generale 74
4.24 Struttura gerarchica a strati del protocollo NTP 75
4.25 Architettura master-slave gerarchica per la distribuzione del clock del protcollo PTP 78

5.1 Sistema I.DRIVE: (a) Veicolo I.DRIVE con installati i sensori, (b) Sensori ambientali (Velodyne, Prosilica) posizionati sul tettuccio del veicolo, (c) Telecamere Axis montate all'interno dell'abitacolo 83

A.1 Finestra di configurazione della scheda di rete della macchina virtuale 88
A.2 Applicazioni *Procomp_server* e *EmpaticaBLEServer* avviate . . . 90

Elenco delle tabelle

3.1 Messaggio NMEA RMC 31
3.2 Tabella di confronto delle caratteristiche tra i ricevitori GPS Garmin 18LV e Yuan10 33
3.3 Messaggio NMEA GGA 34
3.4 Messaggio NMEA GSA 35
3.5 Elenco dei componenti del sistema con relativa tensione di alimentazione e consumo massimo stimato 42

4.1 Tabella degli indirizzi IP statici assegnati 45

5.1 Dati riassuntivi dei test effettuati sull'intero sistema 82

C A P I T O L O

1. Introduzione

I.DRIVE (*Interaction between Driver, Road Infrastructure, Vehicle, and Environment*) è un progetto interdisciplinare del Politecnico di Milano che si occupa di acquisire, analizzare e modellizzare l'interazione tra guidatore, le infrastrutture stradali, il veicolo e l'ambiente. All'interno del progetto I.DRIVE è stato portato avanti il lavoro di tesi descritto in questo elaborato.

In questa tesi è stato infatti progettato e sviluppato il sistema *I.DRIVE Data Logger* occupandosi dell'installazione di diversi sensori a bordo del veicolo laboratorio di I.DRIVE, di acquisire i dati provenienti da tali sensori e di effettuarne il salvataggio all'interno di un database NoSQL chiamato MongoDB. I dati salvati all'interno del database per mezzo del nostro sistema di acquisizione, presentano dei meta dati aggiuntivi come ad esempio la tipologia di messaggio dal quale provengono e la data in cui sono stati salvati all'interno del database. Questi meta dati vengono utilizzati per permettere al nostro sistema la riproduzione dei dati salvati all'interno del dabatase in momenti successivi per una loro analisi offline. Essendo parte del progetto I.DRIVE che si occupa di studiare l'interazione tra guidatore,veicolo e le infrastrutture stradali i sensori vengono utilizzati per percepire l'ambiente circostante alla macchina e per capire lo stato fisico e i comportamenti del guidatore durante la guida.

La scelta di utilizzare un database MongoDB è dovuta alle performance che questo database NoSQL fornisce ed al fatto che, essendo MongoDB basato su documenti dinamici, permette una facile evoluzione del modello dei dati. Il modello dei dati dinamico è molto utile poiché ci consente di non dover definire, per ogni singolo sensore, un modello a priori ma questo viene automaticamente definito a seconda del sensore acquisito.

Lo sviluppo del sistema è avvenuto utilizzando ROS (*Robot Operating System*), un sistema ampiamente conosciuto e utilizzato per le applicazioni robotiche. Il framework ROS consente di avere una elevata modularità del codice che ne facilita il riutilizzo, e utilizza un sistema publish/subscribe per la comunicazione tra i vari moduli. Inoltre presenta molti moduli software già sviluppati da terzi e integra utili strumenti per lo sviluppo e il debug. Questo ci ha permesso di sviluppare un architettura software flessibile che consente modifiche e aggiunte di

moduli software nel tempo mantenendo intatta la robustezza del sistema. Questa è una parte fondamentale per il nostro progetto poiché permetterà, negli sviluppi futuri, una facile gestione di moduli aggiuntivi per i sensori o l'introduzione di nuove funzionalità.

Oltre alla realizzazione del sistema stesso, uno degli obiettivi che ci siamo posti durante lo sviluppo, è stata quella di acquisire i dati dei sensori in maniera che fossero sincronizzati tra di loro, specialmente quelli che in futuro saranno utilizzati per la ricostruzione dell'ambiente in 3D e per la guida autonoma. Per raggiungere questo scopo sono stati utilizzati nel progetto alcuni protocolli di sincronizzazione quali *Precision Time Protocol* (PTP) e *Network Time Protocol* (NTP) e le informazioni di tempo provenienti dal ricevitori GPS disponibili a bordo.

Un altro degli obiettivi che ci siamo posti è stato quello di sviluppare un sistema, robusto e affidabile, che permettesse, tramite uno schermo a bordo del veicolo, di individuare eventuali malfunzionamenti così che, nel caso se ne verificasse uno, il guidatore possa individuarlo immediatamente consentendo di non perdere dati utili. Per fare questo è stato modificato un modulo software esistente, sviluppato presso il Politecnico di Milano, che permette la gestione dello stato dei vari nodi mediante un sistema di comunicazione client-server e l'utilizzo di macchine a stati finiti che contengono i possibili stati in cui si può trovare ogni nodo e le transazioni consentite. Oltre a permettere di conoscere lo stato di ogni nodo, questo modulo monitora che ogni nodo rimanga attivo per l'intera esecuzione del sistema. Per fare questo ogni nodo deve inviare continuamente per tutta la sua esecuzione un messaggio di *"alive"* al server.

La struttura della tesi è la seguente:

- Nel Capitolo 2 sono illustrati alcuni esempi di veicoli autonomi sviluppati nel tempo partendo dai primi progetti di veicoli autonomi e dai primi modelli di auto autonome. Successivamente, sono presentati alcuni esempi di auto autonome, in particolare quelle che si sono distinte in alcune competizioni o che sono oggi in uno stato avanzato di sviluppo. Vengono poi descritti alcuni progetti riguardanti l'acquisizione dei dati del guidatore per lo studio dei comportamenti di quest'ultimo. Infine verrà descritto il framework ROS che è stato utilizzato nello sviluppo del sistema e il database MongoDB.

- Nel Capitolo 3 vengono descritti i componenti hardware utilizzati nella nostra configurazione partendo dall'auto e descrivendo singolarmente ciascuno dei sensori utilizzati.

- Nel Capitolo 4 viene introdotta l'architettura software realizzata per il nostro sistema. Viene inizialmente descritto il sistema di salvataggio dei dati seguito da una descrizione dei vari moduli software utilizzati per l'acquisizione dei sensori e il relativo modello BSON per la rappresentazione del

dato. Successivamente viene descritto il software sviluppato per la visualizzazione dei dati e per il controllo dello stato del sistema. Infine viene descritto come è stato risolto il problema della sincronizzazione.

- Nel Capitolo 5 viene descritto come sono stati montati i sensori a bordo del veicolo I.DRIVE e vengono riportati i dati osservati durante i test.
- Nel Capitolo 6 vengono illustrate le conclusioni e vengono suggeriti possibili estensioni e miglioramenti del progetto e gli sviluppi futuri.

C A P I T O L O

2. Stato dell'arte

In questo capito verrà fornita una descrizione sullo stato dell'arte dei veicoli autonomi ed in particolar modo delle auto autonome. Inoltre vengono descritti dei progetti riguardanti l'acquisizione dei dati del guidatore.

2.1 Veicoli autonomi

L'automazione dei veicoli è una sfida che esiste da quando esistono i veicoli stessi. Ancora oggi la totale automazione di un veicolo di terra rappresenta una grande sfida, soprattutto in presenza di terreni sconnessi e in ambienti complessi e dinamici come le strade di una qualsiasi città. I veicoli autonomi possono essere utilizzati in diversi ambiti, come per esempio, l'esplorazione di un terreno impervio e pericoloso o il raggiungimento di luoghi difficili da raggiungere per un essere umano.

Esistono diverse tipologie di veicoli autonomi di terra con diverse forme, dimensioni, sistemi di locomozione e applicazioni. Infatti sono stati stati utilizzati negli anni diversi veicoli come base per lo sviluppo di un sistema di guida autonomo, ad esempio si è utilizzato autovetture, mezzi militari, camion e bus. Tutti questi veicoli autonomi anche se molto diversi tra loro hanno tutti in comune delle caratteristiche:

- Sono equipaggiati con diversi sensori che permettono di percepire l'ambiente circostante
- Sono in grado di effettuare almeno parte delle loro mansioni in maniera autonoma

Al giorno d'oggi esistono molti progetti in uno stato di lavoro avanzato che riguardano la guida totalmente autonoma di un veicolo in un ambiente complesso e dinamico. Questo è stato possibile negli ultimi anni grazie ai grandi investimenti fatti in questo settore e ai progressi che sono stati fatti sia a livello hardware sia a livello software nel processare le informazioni e i segnali provenienti dai sensori, nella visione artificiale e nella teoria del controllo che hanno reso possibile

incrementare la capacità di rappresentare/analizzare, percepire e rispondere a condizioni dinamiche.

2.2 Auto autonome

Le auto autonome fanno parte dei veicoli autonomi e mantengono le capacità di trasporto di una normale autovettura. Facendo parte dei veicoli autonomi le auto senza conducente hanno la capacità di percepire l'ambiente circostante tramite diversi sensori. Le informazioni provenienti dai diversi sensori vengono poi analizzate e fuse tra loro consentendo alla macchina di poter navigare senza l'intervento dell'uomo. Le informazioni provenienti dai sensori vengono inoltre utilizzate per aggiornare una mappa così da permettere alla macchina di tenere traccia dello stato del mondo circostante.

Negli Stati Uniti la National Highway Traffic Safety Administration (NHTSA)ha proposto una classificazione formale dei sistemi esistenti secondo i seguenti livelli [2]:

- **Livello 0**: il guidatore controlla completamente l'auto per tutto il tempo
- **Livello 1**: alcuni controlli individuali dell'auto sono automatizzati come per esempio il controllo elettronico della stabilità (ESP) o il sistema automatico di frenata
- **Livello 2**: almeno due controlli sono automatizzati contemporaneamente come per esempio il cruise control adattivo combinato con il sistema di mantenimento della corsia
- **Livello 3**: Il conducente può cedere il controllo totale di tutte le funzioni critiche per la sicurezza in certe condizioni. L'auto rileva quando le condizioni richiedono al conducente di riprendere il controllo dell'auto e fornisce un periodo di transizione sufficientemente confortevole per il guidatore per poterlo fare
- **Livello 4**: l'autovettura esegue tutte le funzioni critiche per la sicurezza per l'intero viaggio con il conducente che non deve controllare il veicolo in nessun momento. Visto che in questo caso l'auto controlla tutte le funzioni dall'inizio alla fine del viaggio, incluso il posteggio, sono incluse in questa categoria anche le macchine senza nessuno a bordo.

Le auto completamente autonome ovvero di livello 4 possono portare molti vantaggi tra i quali:

- Evitare gli incidenti causati da errori del guidatore dovuti, per esempio, ad un prolungato tempo di reazione, ad una guida distratta o aggressiva

- Aumento della capacità delle strade e riduzione della congestione dovuta al traffico visto la capacità che si avrebbe con le vetture autonome di gestire il flusso del traffico
- Limiti di velocità più elevati visto le capacità di reazione di una vettura autonoma
- Lo stato di chi è all'interno dell'auto non avrebbe più importanza visto che in una macchina autonoma non importerebbe più se gli occupanti siano minorenni, patentati, ubriachi, ciechi o in generale in uno stato alterato
- Riduzione della necessità della polizia stradale e della segnaletica stradale fisica poiché le auto autonome potranno ricevere le comunicazioni necessarie per via elettronica
- Aumento dello spazio all'interno dell'abitacolo visto che verrebbe rimosso il volante e i pedali e non ci sarebbe più la necessità per il guidatore di sedersi davanti
- Riduzione dei furti d'auto

Nonostante i vari vantaggi che l'utilizzo di un'auto autonoma comporta ci sono alcune sfide tecnologiche e legali che tuttora persistono:

- Affidabilità del software: dipendendo in maniera completa dal software, un'auto autonoma deve garantire una affidabilità molto elevata così da evitare malfunzionamenti che potrebbero portare ad incidenti
- Sicurezza: essendo controllata da un computer un'auto autonoma potrebbe essere compromessa come potrebbe essere compromesso anche il sistema di comunicazione tra le varie vetture
- Definizione di uno spettro radio per la comunicazione tra le vetture
- Il sistema di navigazione di una macchina autonoma sarebbe sensibile ai differenti tipi di condizioni meteo, sopratutto in caso di neve e pioggia, in maniera maggiore di un uomo poiché alcuni sensori in queste condizioni non sono affidabili
- Le auto autonome necessitano di mappe di alta qualità per funzionare correttamente. Le mappe dovrebbero inoltre essere aggiornate e in caso non lo fossero la vettura dovrebbe essere in grado di avere un comportamento sicuro e affidabile
- Le infrastrutture stradali potrebbero necessitare di modifiche per poter funzionare in maniera ottimale con un'auto autonoma. Un esempio potrebbe essere un aggiornamento dei semafori che permetta loro di comunicare con le auto.

- Perdita della privacy: la posizione della propria vettura sarà sempre tracciata ed inoltre verranno condivise informazioni con le altre vetture e con le infrastrutture stradali

- Attuazione di un quadro giuridico e la definizione di regolamenti governativi specifici per le auto autonome e per la responsabilità dei danni in caso di incidente

- Resistenza da parte delle persone a lasciare il completo controllo alla loro auto

L'ultimo punto sopra elencato è il punto focale del progetto I.DRIVE ovvero, lo studio delle reazioni del guidatore quando è l'auto a prendere l'iniziativa nel caso di sistemi di assistenza alla guida (frenata assistita, ABS, ecc. . .) o nel caso di auto completamente autonoma.

2.3 Progetti esistenti

In questo sezione verranno descritti alcuni esempi di progetti riguardanti veicoli autonomi che sono stati sviluppati durante gli anni al fine di comprendere i sensori solitamente utilizzati in questi ambiti ed in particolare nelle auto autonome. Inoltre verranno descritti alcuni progetti che si focalizzano sull'acquisizione di dati del guidatore così da comprendere meglio i sensori utilizzati in questi ambiti e come vengono acquisiti.

2.3.1 I primi veicoli autonomi

Di seguito verranno descritti i primi progetti di veicoli autonomi che sono stati sviluppati [14]. I primi progetti di seguito descritti riguardano più robot che veri e propri veicoli mentre gli ultimi due progetti riguardano vere e proprie auto.

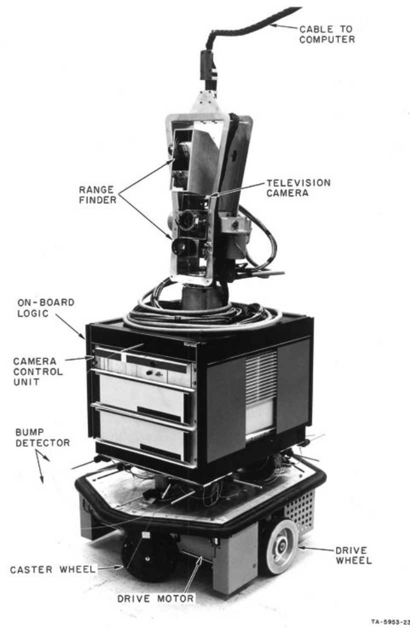
Shakey [24] (Figura 2.1a) fu sviluppato presso la SRI international negli Stati Uniti dal 1966 al 1972. Shakey è considerato il primo robot che presentava comportamenti autonomi. Questo robot era costituito da una piattaforma con ruote ed era dotato di una telecamera orientabile, ultrasuoni per misurare la distanza e sensori di tatto. Shakey era collegato tramite una connessione RF al computer mainframe SDS-940 che si occupava della navigazione e della esplorazione. Shakey poteva ricevere comandi scritti in inglese che venivano inviati tramite terminale e permettevano al robot di conoscere il task da eseguire. Il robot era in grado di identificare e spostare grossi blocchi di legno per il laboratorio. Questo progetto stabilì le basi funzionali e di performance per i robot mobili identificando le carenze tecnologiche.

Stanford Cart [23] (Figura 2.1b) fu sviluppato presso il laboratorio di inteligenza artificiale dell' università di Stanford dal 1973 al 1981. Lo Stanford Cart era un robot mobile controllato da remoto che permetteva l'esplorazione di un ambiente evitando gli ostacoli per mezzo di un sistema di visione stereo. Questo robot era dotato di una singola camera che veniva mossa in ognuna delle 9 differenti posizioni predefinite grazie alla sua semplice base mobile. Le immagini risultanti venivano inviate, per essere processate, a un mainframe KL-10 che non era montato a bordo del robot. Tramite l'estrazione delle caratteristiche e la correlazione tra le immagini era possibile ricostruire il modello 3D della scena. Il modello 3D veniva poi utilizzato per pianificare un percorso fino alla destinazione che non presentasse ostacoli. Questo sistema era molto lento e permetteva di avanzare di 1 metro ogni 15 minuti.

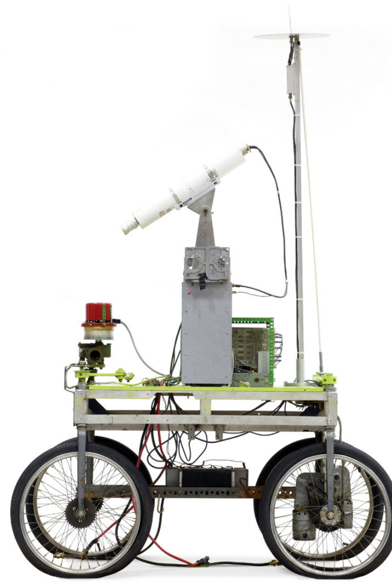
DARPA Autonomous Land Vehicle [12] [28] (Figura 2.1c) fu sviluppato negli Stati uniti dalla DARPA nell'ambito del programma Strategic Computing dal 1985 al 1988. L'ALV è stato costruito partendo da un veicolo standard a 8 ruote capace di raggiungere una velocità di 72 km/h su una strada normale e una velocità di 29 km/h su terreni accidentati. L'ALV poteva trasportare 6 scaffali di apparecchiature elettroniche in un ambiente senza polvere e con aria condizionata prendendo l'energia necessaria dalla sua unità di potenza ausiliaria diesel da 12 kW. Inizialmente i sensori montati a bordo dell'ALV consistevano in una videocamera a colori e in uno scanner laser che restituiva i dati ad intervalli di 1-2 secondi.
I dati dalla videocamera e dallo scanner laser venivano processati da un modulo apposito che generava un modello della scena.
La prima dimostrazione in cui l'ALV seguì la strada che doveva percorrere si svolse nel 1985 ad una velocità di 3 km/h su un percorso diritto di 1 km. Successivamente, nel 1986, si svolse un'altra dimostrazione in cui l'ALV seguì a 10 km/h un percorso di 4.5 km che presentava curve e diverse tipologie di terreno. Infine nel 1987 si svolse una dimostrazione a una velocità media di 15.5 km/h (massimo 21 km/h) su un percorso di 4.5 km con diversi tipologie di terreno e diverse larghezze stradali in cui era necessario anche evitare ostacoli presenti sul tracciato.

Ground Surveillence Robot [15] (Figura 2.1d) fu sviluppato negli Stati Uniti presso la Naval Ocean Systems Center dal 1985 al 1986.Il progetto GSR riguardava lo sviluppo di una architettura distribuita modulare e flessibile per l'integrazione e il controllo di un sistema robotico complesso, utilizzando un veicolo corazzato M-114 da 7 tonnellate come banco di prova. Il veicolo, utilizzando una serie di ultrasuoni fissi e orientabili e una architettura distribuita blackboard implementata su più PC, dimostrò di poter seguire autonomamente sia un altro veicolo sia un essere umano a piedi.

VaMP [20] [13] (Figura 2.2a) fu sviluppato in Germania presso l'università



(a)



(b)

(c)

(d)

Figura 2.1: Alcuni dei primi prototipi di veicoli autonomi: (a) Shakey, (b) Stanford Cart, (c) DARPA Autonomous Land Vehicle, (d) Ground Surveillance Robot

(a)

(b)

Figura 2.2: I primi prototipi di auto autonome: (a) VaMP, (b) ARGO

Bundeswehr di Monaco dal 1993-1995. La VaMP fu una delle prime vere auto autonome e era basata su una Mercedes 500 SEL riprogettata in modo che fosse possibile controllarne i freni, l'acceleratore e lo sterzo tramite i comandi di un computer. Questa auto era in grado di guidare nel traffico intenso per lunghe distanze senza l'intervento umano grazie alla visione artificiale che le permetteva di riconoscere rapidamente gli ostacoli in movimento, come le altre macchine, evitandoli o superandoli autonomamente. La VaMP si basava solo sull'utilizzo di 4 camere, 2 che guardavano di fronte alla vettura e 2 che guardavano dietro la vettura.
Nel 1995 la VaMP fu testata su un lungo tragitto da Monaco a Odense in Danimarca. L'auto percorse 1600 km, 95% dei quali percorsi in maniera totalmente autonoma senza nessun intervento umano.

ARGO [21] (Figura 2.2b) fu sviluppato in Italia presso il dipartimento di ingegneria dell'informazione dell'università di Parma. L'auto ARGO era basata su una Lancia Thema 2000 con installato sulla colonna dello sterzo un motore elettrico, che permetteva al veicolo di sterzare autonomamente. La vettura era dotata solamente di sensori passivi e non invasivi, infatti era equipaggiata con solo un sistema di visione stereoscopica che consisteva in due camere in bianco e nero sincronizzate. Il sistema di visione artificiale consentiva alla vettura di estrarre informazioni sulla strada e l'ambiente circostante permettendo così alla macchina di seguire la corsia, localizzare gli ostacoli sul percorso e di effettuare cambi di corsia. Nel Giugno de 1998 ARGO fu sottoposto ad un test estensivo in cui sono stati percorsi circa 2000 km sulle autostrade italiane in modalità automatica ed in condizioni ambientali reali.

2.3.2 DARPA Grand Challenge

La *DARPA Grand Challenge* è stata una competizione per veicoli autonomi americani, finanziata dalla Defense Advanced Research Projects Agency (DARPA), l'ente di ricerca più importante del dipartimento della difesa degli Stati Uniti. Questa competizione, iniziata nel 2004, aveva lo scopo di stimolare lo sviluppo delle tecnologie necessarie per creare veicoli terrestri completamente autonomi in grado di portare a termine un percorso fuoristrada in un periodo di tempo limitato. La terza edizione di questa competizione, che cambiò nome in DARPA Urban Challenge, si svolse nel 2007 e prevedeva un percorso che simulava un ambiente urbano.
Tra i migliori progetti che parteciparono a questa competizione citiamo:

Stanley [27] (Figura 2.3a) fu sviluppato negli Stati Uniti presso l'università di Stanford nel 2005. Stanley partecipò e vinse la DARPA Grand Challenge del 2005 percorrendo i 212 Km del percorso previsto in 6 ore e 54 minuti. Il progetto era basato su una Volkswagen Touareg R5 Diesel a 4 ruote motrici. La macchina è stata modificata per poter comandare tramite un attuatore elettrico l'acceleratore e il freno. Inoltre è stato montato un motore elettrico DC alla colonna dello sterzo per poterlo comandare elettronicamente. La vettura era equipaggiata superiormente con 5 sensori di distanza laser SICK che puntavano nella direzione di guida con diverse angolazioni, una camera a colori per percepire la strada a grandi distanze anch'essa puntata in avanti, 2 sensori RADAR a 24 GHz per la rilevazione di grandi ostacoli a grandi distanze, un GPS e una IMU. Inoltre i dati provenienti dal veicolo stesso come velocità e angolo di sterzata, venivano comunicati al sistema tramite CAN bus.

Sandstorm [7] (Figura 2.3b) fu sviluppato negli Stati Uniti presso l'università Carnegie Mellon dal 2004 al 2005. Sandstorm partecipò alla DARPA Grand Challenge del 2004 dove nessuno dei veicoli terminò il percorso ma, Sandstorm, fece registrare la distanza percorsa maggiore di 11.78 Km. La macchina partecipò poi alla DARPA Grand Challenge nel 2005 arrivando al secondo posto percorrendo i 212 Km del percorso in 7 ore e 5 minuti. Il progetto si basa sul modello M998 HMMWV del 1986. Questo veicolo è stato modificato ponendo un ingranaggio dietro al volante, all'inizio della colonna dello sterzo per essere comandato tramite un motore DC. Questo sistema permetteva così di comandare lo sterzo. Inoltre per comandare i freni venne posto un motore elettrico che permetteva di premere il pedale del freno e, per comandare l'acceleratore, venne sostituita la valvola della pompa dell'iniezione con una valvola comandata da un motore DC. I sensori montati a bordo del veicolo nel 2004 includevano: 3 LIDAR fissi, un LIDAR orientabile, un RADAR, un paio di camere per la visione stereo, un GPS e una IMU. Per la competizione del 2005 vennero aggiunti al parco dei sensori 3 LIDAR fissi.

H1ghlander [7] (Figura 2.3c) fu sviluppato negli Stati Uniti presso l'università Carnegie Mellon nel 2005. L'H1ghlander partecipò alla DARPA Grand Challenge nel 2005 arrivando al terzo posto percorrendo i 212 Km del percorso in 7 ore e 14 minuti. Questo veicolo è stato sviluppato dallo stesso team del veicolo Sandstorm descritto precedentemente con il quale condivide il software e i sensori montati a bordo. Una differenza importante è il modello di veicolo utilizzato poichè l'H1ghlander è basato su un HUMMER H1 del 1999. Un'altra differenza presente è il sistema di sterzo poichè, nell'H1ghlander, lo sterzo è stato rimosso e sostituito con un sistema di sterzo totalmente idraulico. Infine i controlli dell'acceleratore per l'H1ghlander sono gestiti in maniera totalmente elettronica visto che il motore lo consente e il freno viene sempre attuato da un motore Dc ma che attua direttamente il cilindro principale dei freni.

Kat-5 [25] (Figura 2.3d) fu sviluppato negli Stati Uniti presso la GrayMatter Inc.nel 2005. Il KAt-5 partecipò alla DARPA Grand Challenge nel 2005 arrivando al quarto posto percorrendo i 212 Km del percorso in 7 ore e 30 minuti. Il Kat-5 è bastato su una Ford Escape ibrida del 2005. L'interfacciamento con i comandi primari del veicolo (Sterzo,acceleratore e freno) avviene per mezzo di un sistema drive-by-wire della Electronic Mobility Controls personalizzato. Questo sistema consiste in attuatori e servo motori montati sulla colonna dello sterzo, sul pedale del freno, sul filo dell'acceleratore e sulla trasmissione automatica. Il Kat-5 utilizza 2 LIDAR orientabili e un INS/GPS.

Boss [8] [9] (Figura 2.4a) fu sviluppato negli Stati Uniti presso l'università Carnegie Mellon nel 2007. Boss partecipò alla DARPA Urban Challenge nel 2007 arrivando al primo posto percorrendo i 96 Km di percorso urbano in 4 ore e 10 minuti. Boss si basa su una Chevrolet Tahoe del 2007 modificata con un sistema drive-by-wire commerciale in cui sono presenti motori elettrici che consentono di sterzare, schiacciare il pedale del freno e spostare la leva del cambio. Questo sistema consente anche tramite CAN bus di comunicare con il modulo di controllo del motore. Boss utilizza una combinazione di sensori, per la maggior parte attivi, per poter navigare in maniera sicura in un ambiente urbano. La macchina è dotata infatti di: un LIDAR Velodyne a 64 piani, 5 RADAR di cui 4 puntano in avanti e uno montato nella parte posteriore del veicolo, 3 sensori SICK che puntano anteriormente, 2 sensori SICK rivolti sui due lati, un SICK montato posteriormente, 4 LIDAR fissi che puntano in avanti, 2 camere per la visione stereo, un GPS e una IMU.

Junior [22] (Figura 2.4b) fu sviluppato negli Stati Uniti presso l'università di Stanford nel 2007. Junior partecipò alla DARPA Urban Challenge nel 2007 arrivando al secondo posto percorrendo i 96 Km di percorso urbano in 4 ore e 29 minuti. Junior si basa su una Volkswagen Passat Wagon del 2006 opportu-

(a)

(b)

(c)

(d)

Figura 2.3: I partecipanti alla DARPA Grand Challenge del 2005 : (a) Stanley, (b) Sandstorm, (c) H1ghlander, (d) kAT-5

namente modificata con un sistema drive-by-wire che consente di comandare i principali controlli del veicolo quali sterzo, freni, acceleratore, leva del cambio, freno di stazionamento e indicatori di direzione. Per la navigazione inerziale la vettura si basa un un sistema che include un GPS a doppia frequenza, una IMU e l'odometria delle ruote. Inoltre i sensori montati a bordo della vettura includono: 2 sensori SICK posti sui 2 lati, un sensore laser che punta in avanti, un LIDAR Velodyne a 64 piani per il rilevamento di ostacoli e veicoli in movimento, 2 sensori SICK nella parte posteriore del veicolo, 2 LIDAR fissi montati sul paraurti anteriore e 5 RADAR montati intorno alla griglia anteriore.

Odin [5] (Figura 2.4c) fu sviluppato negli Stati Uniti presso l'università Virgina Tech nel 2007. Odin partecipò alla DARPA Urban Challenge nel 2007 arrivando al terzo posto percorrendo i 96 Km di percorso urbano in 4 ore e 36 minuti. Odin si basa un una Ford Escape Hybrid del 2005. Nella Ford Escape Hybrid i comandi dello sterzo, del cambio e del sistema di valvole a farfalla sono già di fabbrica drive-by-wire per cui possono essere controllati elettronicamente semplicemente emulando i segnali di comando eliminando la problematica di do-

ver aggiungere ulteriori attuatori per questi comandi. Per percepire l'ambiente circostante Odin è dotato di: 2 sensori di distanza laser che puntano in avanti, un sensore di distanza laser montato posteriormente, 2 sensori di distanza laser SICK anteriori, 2 sensori SICK posti ai 2 lati della vettura e 2 camere a colori in configurazione stereo poste anteriormente che però non furono usate per la competizione finale.

Talos [17] (Figura 2.4d) fu sviluppato negli Stati Uniti presso il Massachusetts Institute of Technology (MIT) nel 2007. Talos partecipò alla DARPA Urban Challenge nel 2007 arrivando al quarto posto percorrendo i 96 Km di percorso urbano in circa 6 ore. Talos è basato su una Land Rover LR3 modificata con un sistema drive-by-wire che permette il controllo delle funzioni primarie del veicolo. Talos era nato come progetto di auto autonoma che non facesse uso di sensori costosi infatti era dotato solamente di 12 LIDAR SICK posti tutti intorno al veicolo, 15 radar anch'essi posti tutti intorno al veicolo e 5 camere. Durante il progetto però venne aggiunto ai sensori già presenti un LIDAR Velodyne a 64 piani che modificò l'obiettivo iniziale di non usare sensori costosi. La Velodyne ebbe un ruolo centrale per il rilevamento degli ostacoli intorno al veicolo.

2.3.3 Auto autonome moderne

Di seguito verranno descritti alcuni progetti di auto autonome che sono ancora in fase di sviluppo così da capire le ultime tecnologie sviluppate ed utilizzate in questo ambito.

AnnieWAY [4] [3] (Figura 2.5a) è stato sviluppato in germania presso il Karlsruhe Institute of Technology (KIT) a partire dal 2007. AnnieWAY è basato su una Volkswagen Passat modificata con degli attuatori controllabili elettronicamente tramite CAN-bus per l'acceleratore, i freni, la trasmissione e lo sterzo. A bordo del veicolo troviamo: un sensore RADAR a 24 GHz, un LIDAR Velodyne a 64 piani, 2 camere per una visione stereo a colori ad alta definizione, 2 camere per una visione stereo in bianco e nero, un GPS e una IMU. Inoltre, AnnieWAY è dotato di un sistema di comunicazione tra veicoli che si basa sullo standard wireless 802.11p che permette la comunicazione fino a 800 m di distanza. Questa vettura ha partecipato alla DARPA Urban Challenge del 2007 non terminando il percorso a causa di un problema nel software mentre nel 2011 partecipò alla Grand Cooperative Driving Challenge (GCDC) vincendola [3]. La GCDC è una competizione che mira ad accelerare l'attuazione e l'interoperabilità della comunicazione wireless tra auto autonome. Questa vettura è utilizzata e sviluppata all'interno di un progetto che si chiama KITTI che è portato avanti dalla Karlsruhe Institute of Technology e dalla Toyota Technological Institute at Chicago. Questo progetto si occupa di sviluppare nuovi benchmarks nel campo della visio-

(a)

(b)

(c)

(d)

Figura 2.4: I partecipanti alla DARPA Urban Challenge del 2007 : (a) Boss, (b) Junior, (c) Odin, (d) Talos

ne artificiale in particolar modo gli ambiti di interesse sono l’odometria visuale, rilevamento di oggetti 3D e inseguimento di oggetti 3D.

Deeva [6] (Figura 2.5b) è stata sviluppato in Italia presso l’Artificial Vision and Intelligent Systems Laboratory (VisLab) a partire dal 2015. Deeva si basa su una Audi A4 2.0T FWD del 2013 modificata per ospitare un sistema drive-by-wire che, tramite degli attuatori comandati tramite un CAN-bus, permette di controllare le funzionalità primarie del veicolo (acceleratore, freno, sterzo) e opzionalmente quelle secondarie (cambio, tergicristalli e indicatori di direzione). Questo progetto è l’evoluzione di un progetto nato nel 2013 con il nome di BRAiVE e utilizza una tecnologia proprietaria basata sulla visione artificiale che gli permette di raggiungere due obiettivi principali: impiego di sensori a basso costo ed elevato livello di integrazione dei sensori con il veicolo infatti, esteticamente, la vettura pare essere tradizionale auto.
Questa vettura monta a bordo 13 coppie di camere stereo tutte a colori tranne 4 camere che sono a infrarossi e sono poste sugli specchietti retrovisori. Le camere

si dividono in 2 sistemi diversi:

- Visione di prossimità: un sistema è composto da 4 paia camere stereo che sono posizionate: una nella griglia anteriore, sulla parte superiore del bagagliaio e sui due specchietti retrovisori. Questo sistema è usato durante le manovre

- Visione in lontananza: un sistema è composto da 9 paia di camere stereo che sono posizionate: 4 posizionate dietro il parabrezza per rilevare ostacoli davanti alla vettura,1 posizionata in ciascuno degli specchietti retrovisori, 1 posteriore utilizzata per il rilevamento di ostacoli dietro la vettura, 2 montate sui parafanghi anteriori orientate verticalmente

Inoltre la vettura monta a bordo: 1 sensore laser a 8 piani posizionato in mezzo al paraurti anteriore, 2 sensori laser a 4 piani sui due lati del paraurti anteriore, un sensore laser a 4 piani in mezzo al paraurti posteriore e un GPS/IMU.

MadeInGermany [11] [10] (Figura 2.5c) è stata sviluppata in Germania presso l'Autonomous Lab a partire dal 2010. MadeInGermany si basa su una Volkswagen Passat Variant 3c equipaggiata con un sistema drive-by-wire che permette di accedere ai comandi dei freni, del motore, dello sterzo e del cambio tramite CAN-bus.La vettura è dotata di un LIDAR Velodyne a 64 piani, 3 sensori di distanza laser montati sul paraurti anteriore, 3 sensori di distanza laser posti sul paraurti posteriore, 3 RADAR posti anch'essi sul paraurti anteriore, 2 camere per la visione stereo poste sul parabrezza, 1 sensore per l'odometria posto sulle ruote posteriori ed un GPS. Questa macchina possiede il permesso di circolare in Germania nel traffico reale testando le funzioni di guida autonoma. Inoltre, questa vettura, nel 2015 è stata testata in Messico e ha percorso in maniera totalmente autonoma 2250 Km di autostrada e 150 km attraverso le città.

Google car [1] (Figura 2.5d, Figura 2.5e) è un progetto sviluppato negli Stati Uniti presso Google Inc. a partire dal 2009. Il progetto Google car è probabilmente il progetto che riguarda la guida autonoma più conosciuto visto il grande impatto mediatico che Google riesce a ottenere. Questo progetto comprende 3 diversi tipi di veicoli: una Toyota Prius, uns Lexus RX450h e un prototipo di auto progettato da zero dalla stessa Google. Tutte le vetture montano gli stessi sensori che sono: un LIDAR Velodyne a 64 piani, 3 RADAR posti sul paraurti anteriore, un RADAR posto sul paraurti posteriore, una camera posta vicino allo specchietto retrovisore, un encoder montato sulle ruote posteriori, un GPS e una IMU. All'interno del progetto Google car a oggi sono stati percorsi più di un milione di miglia su strade pubbliche in maniera totalmente autonoma e ancora adesso circolano a Mountain View, California e ad Austin, Texas. Le vetture appartenenti al progetto Google car sono state coinvolte in 15 incidenti minori

durante gli 1,9 milioni di miglia percorsi tra guida autonoma e guida manuale. Nessuno di questi incidenti è stato causato dalla guida autonoma ma la maggior parte degli incidenti è stato fatto in guida manuale o la colpa era degli altri automobilisti.

2.3.4 Progetti riguardanti l'analisi del guidatore

In questa sezione verranno descritti alcuni progetti [18] che si focalizzano sull'acquisizione dei dati del guidatore e sulla loro analisi in modo da capire quali sensori vengono usati in questo ambito e come vengono acquisiti.

UTDrive [26] è un progetto che ha come obiettivo quello di acquisire dati del guidatore per poter analizzarne il comportamento mentre interagisce con i sistemi vocali o mentre effettua attività secondarie comuni. Il sistema è composto dai seguenti sensori: un array di cinque microfoni omnidirezionali posizionati vicino al parasole del guidatore, un microfono ad archetto indossato dal guidatore, due camere firewire di cui una punta verso il guidatore e una punta la strada di fronte alla macchina, sensori di pressione per il freno e l'acceleratore, sensore di distanza anteriore, GPS e un sensore biometrico che misura battito cardiaco e pressione del sangue. Inoltre vengono acquisiti i dati di angolo di sterzo, velocità del veicolo e accelerazione tramite i segnali CAN-Bus provenienti dalla porta OBD-2 del veicolo. Per acquisire i sensori viene utilizzato un DAC (*Data Acquisition Unit*) che permette di acquisire i dati a 100 MHz. I dati che sono stati salvati presentano tutti un timestamp per la sincronizzazione.

NUDrive [19] è un progetto che si occupa di acquisire i dati del guidatore ed è stato portato avanti presso la Università Nagoya in Giappone. In questo progetto sono stati installati diversi sensori all'interno di una Toyota Hybrid Estima tra i quali abbiamo: 5 camere CCD tre che puntano verso il guidatore da diverse angolazioni, una che guarda i piedi del guidatore e una che punta anteriormente, una camera omnidirezionale montata sul tettuccio dell'auto, 11 microfoni omnidirezionali posti in diverse posizioni all'interno dell'abitacolo, un microfono ad archetto indossato dal guidatore, due sensori di distanza sono montati anteriormente, sensori di pressione dell'acceleratore e del freno, un sensore che misura l'angolo di sterzata, un GPS, un accelerometro 3D, un sensore che misura la velocità del veicolo, un sensore che misura il battito cardiaco, un sensore che misura la conduttività della pelle e un sensore che misura la sudorazione sul palmo della mano. Ogni nodo del sistema di salvataggio permette di registrare 240 GB di dati video a 1.4 milioni di pixel e 29.4 fps che corrispondono a 90 minuti di video.

UYANIK [16] è un progetto portato avanti ad Istanbul, Turchia che si occupa di acquisire i dati del guidatore. Per questo progetto è stata utilizzata una Renault Sedan. A bordo della vettura sono installate 3 camere a 30 fps due che

(a)

(b)

(c)

(d)

(e)

Figura 2.5: Esempi di auto autonome moderne : (a) AnnieWAY, (b) Deeva, (c) MadeInGermany, (d) Prototipo sviluppato all'interno del progetto Google car, (e) Modello Lexus all'interno del progetto Google car

puntano verso il guidatore e una che punta in avanti, 3 microfoni montati all'interno dell'abitacolo, un microfono ad archetto, una IMU 3D, un GPS, i sensori di pressione del pedale dell'acceleratore e del freno, un laser anteriore. Viene utilizzato anche il CAN-Bus per acquisire la velocità del veicolo e i giri al minuto del motore. Il sistema di acquisizione è formato da 3 sottosistemi sincronizzati tra di loro: il sottosistema per l'acquisizione del video, il sottosistema per l'acquisizione dell'audio e il sottosistema che acquisisce il resto dei sensori.

2.4 Robot Operating System

In robotica, scrivere del software è difficile poiché, esistendo diversi tipi di robot con componenti hardware completamente diversi, e di conseguenza il riutilizzo del codice non è semplice. Inoltre, il codice sviluppato in robotica deve implementare diversi livelli di astrazione partendo dal livello più basso dei driver e salendo attraverso funzionalità di più alto livello quale percezione, guida autonoma, pianificazione e localizzazione. Vista che le competenze richieste sono molto ampie e vanno oltre quelle di un singolo ricercatore le architetture software nella robotica devono anche supportare l'integrazione del software su larga scala. Per poter risolvere queste problematiche molti ricercatori, negli anni, hanno sviluppato una grande varietà di framework che permettessero di gestire la complessità facilitando la prototipazione rapida. Ognuno di questi framework era sviluppato per un particolare ambito e questo ha portato ad una frammentazione nei sistemi software sviluppati in robotica. ROS (*Robot Operating System*) è un framework di programmazione generale che permette la modularità del codice e il suo riutilizzo ed è diventato con il tempo lo standard di fatto per lo sviluppo di software nella robotica.

Robot Operating System (ROS) è un framework di programmazione flessibile, adatto a molte situazioni nell'ambito dello sviluppo software in robotica ed è stato sviluppato dalla *Stanford Artificial Intelligence Laboratory* e dalla *Willow Garage*. ROS è un sistema operativo open source per la programmazione dei robot che fornisce i servizi tipici di un sistema operativo inclusi l'astrazione dell'hardware, il controllo a basso livello dei dispositivi, la gestione dello scambio di messaggi tra processi e la gestione dei pacchetti. ROS fornisce anche strumenti e librerie per ottenere, costruire, scrivere ed eseguire codice tra più computer.

Un sistema creato utilizzando ROS consiste in un certo numero di processi, potenzialmente eseguiti su diversi computer, chiamati *nodi*, che sono connessi a runtime secondo una topologia peer-to-peer. I nodi sono moduli software indipendenti che eseguono i calcoli e che solitamente sono associati a una singola funzionalità o a un singolo componente. I nodi ROS possono essere raggruppati in pacchetti tramite la creazione di una cartella contenente un file XML che ne descrive il contenuto e ne indica le dipendenze. Per permettere una mag-

giore flessibilità, i nodi ROS possono essere scritti in quattro diversi linguaggi di programmazione: C++, Python, Octave, LISP. Nodi sviluppati con differenti linguaggi di programmazione possono coesistere nello stesso sistema perché in ROS la specifica è contenuta al livello dei messaggi.

Per supportare lo sviluppo multi linguaggio, i messaggi inviati tra i differenti nodi sono definiti per mezzo di un IDL (*Interface Definition Language*) che è indipendente dal linguaggio di programmazione utilizzato. L' IDL utilizza un breve file di testo per descrivere i campi contenuti in ogni messaggio e consente la composizione di più messaggi. I generatori di codice specifici per ogni linguaggio supportato generano le implementazioni dei messaggi nel codice nativo che vengono serializzate e deserializzate da ROS quando i messaggi vengono ricevuti. Di base ROS contiene già diversi tipi di messaggi che permettono di trasportare diversi dati ma permette anche di creare dei nuovi messaggi personalizzati.

Un nodo invia un messaggio pubblicandolo su uno specifico *topic* che viene identificato per mezzo di una stringa che ne rappresenta il nome. Un nodo che è interessato a un certo tipo di messaggio si sottoscrive al topic appropriato. Ci possono essere simultaneamente più nodi che pubblicano sullo stesso topic e più nodi che si sottoscrivono allo stesso topic. Inoltre un singolo nodo può pubblicare e sottoscriversi a più di un topic. Sebbene il modello di pubblicazione-sottoscrizione basato sui topic sia molto flessibile il suo funzionamento è soltanto asincrono. Per permettere la comunicazione sincrona, ROS fornisce un sistema chiamato *servizio*. Ogni servizio è definito da una stringa che ne identifica il nome e un paio di messaggi: uno per la richiesta e uno per la risposta. Al contrario dei topic un solo nodo può pubblicare un servizio con un particolare nome.

La topologia peer-to-peer richiede un meccanismo di ricerca per consentire ai processi di trovarsi tra di loro a runtime; il **master** ha questo ruolo. Il nodo master infatti si occupa di mettere in comunicazione chi pubblica il messaggio (Talker) con chi si sottoscrive a quel messaggio (Listener). Una volta che i nodi si sono trovati essi comunicano tramite una connessione peer-to-peer. La parte di negoziazione della connessione peer-to-peer e la sua configurazione avviene tramite il protocollo XML-RPC (Figura 2.6). Il master fornisce anche un server per i parametri che sono condivisi e accessibili tramite le API di rete.

Una forza di ROS è la disponibilità di vari nodi e pacchetti già esistenti, attualmente ci sono migliaia di pacchetti ROS disponibili in repository pubblici. La maggior parte di questi pacchetti implementa una funzionalità specifica come ad esempio uno specifico driver per uno specifico sensore.

Un'altra importante caratteristica di ROS è l'insieme degli strumenti di sviluppo presenti. Questi strumenti di sviluppo permettono l'introspezione, il debug, il log dei dati e la visualizzazione dello stato del sistema in fase di sviluppo. Il meccanismo di pubblicazione/sottoscrizione consente la facile introspezione del flusso dei dati attraverso il sistema, rendendo più facile individuare la presenza di errori. Gli strumenti di ROS sfruttano questa capacità di introspezione grazie

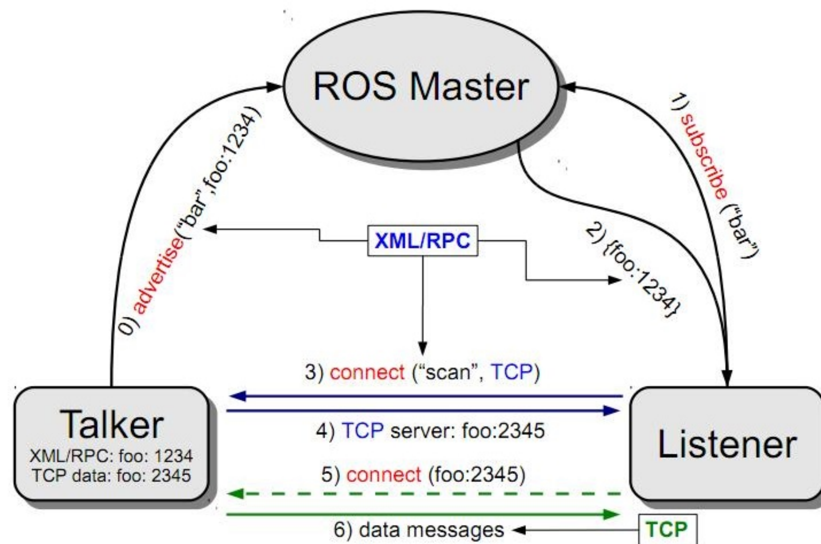


Figura 2.6: Struttura base di ROS

ad un vasto numero di utilità grafiche e da linea di comando che permettono di semplificare lo sviluppo e il debug del software.

Lo strumento di sviluppo più conosciuto è *RVIZ* che permette la visualizzazione tridimensionale di molti dei più comuni tipi di messaggio forniti da ROS come, ad esempio, immagini provenienti da una videocamera, una scansione di un laser, una nuvola di punti tridimensionale. Nella Figura 4.17 è possibile vedere un esempio di schermata RVIZ in cui a sinistra sono visualizzate le immagini di 4 camere e sulla destra è presente la visualizzazione di un LIDAR Velodyne a 32 piani.

Un altro importante e conosciuto strumento di sviluppo è *rqt*, un framework basato sulla libreria Qt per sviluppare interfacce grafiche per il proprio robot. Tramite rqt è possibile per esempio visualizzare e analizzare un sistema ROS mostrandone i nodi e le connessione presenti tra i vari nodi oppure andando a monitorare i topic presenti (banda utilizzata, messaggi inviati, messaggi persi, ecc. . .). Nella Figura 2.8 è possibile vedere un esempio di finestra rqt con visualizzati alcuni plugin. In alto è possibile vedere il plugin per visualizzare una pagina web, quello per pubblicare dei topic con valori fissi o calcolati tramite una espressione, quello per comandare un robot e quello per selezionare il livello del logger. In basso invece troviamo il plugin che visualizza e filtra i messaggi ROS e il plotter che permette di visualizzare i valori numerici in un grafico 2D.

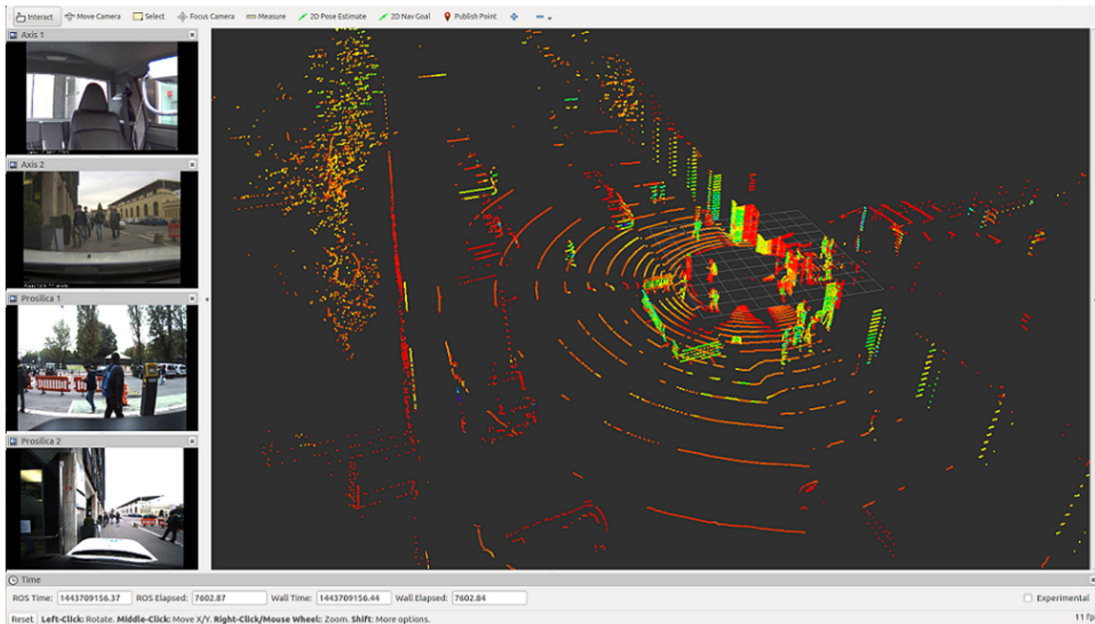


Figura 2.7: Esempio di schermata di RVIZ

2.5 MongoDB

Come goà descritto nell'introduzione per il salvataggio dei dati è stato scelto un *database NoSQL* ovvero un database che non utilizza il modello relazionale solitamente utilizzato nei normali database. Il database NoSQL che è stato scelto è *MongoDB* visto la sua elevata popolarità tra i database NoSQL e le prestazioni a cui questo DB può arrivare se confrontato con gli altri database NoSQL. Inoltre la scelta è stata guidata dalla presenza in ROS di un pacchetto che gestisce il salvataggio su questa tipologia di database.

MongoDB è un database NoSQL open source orientato ai documenti, in quanto si basa su documenti in stile JSON con schema dinamico che vengono chiamati *BSON*. Il formato BSON è una struttura dati composta da coppie campo valore. Il vantaggio di utilizzare dei documenti con una struttura dinamica è che questa permette una facile evoluzione del modello dei dati. La struttura dati in un database MongoDB (Figura 2.9) è così definita:

- Un *database* MongoDB contiene un insieme di *collezioni*
- Una collezione contiene al suo interno un insieme di *documenti*
- Un documento (Figura 2.10) è un insieme di *campi*, ognuno dei quali è una coppia chiave-valore (dove la chiave è una stringa, un valore è un tipo di base o un documento o un array di valori)

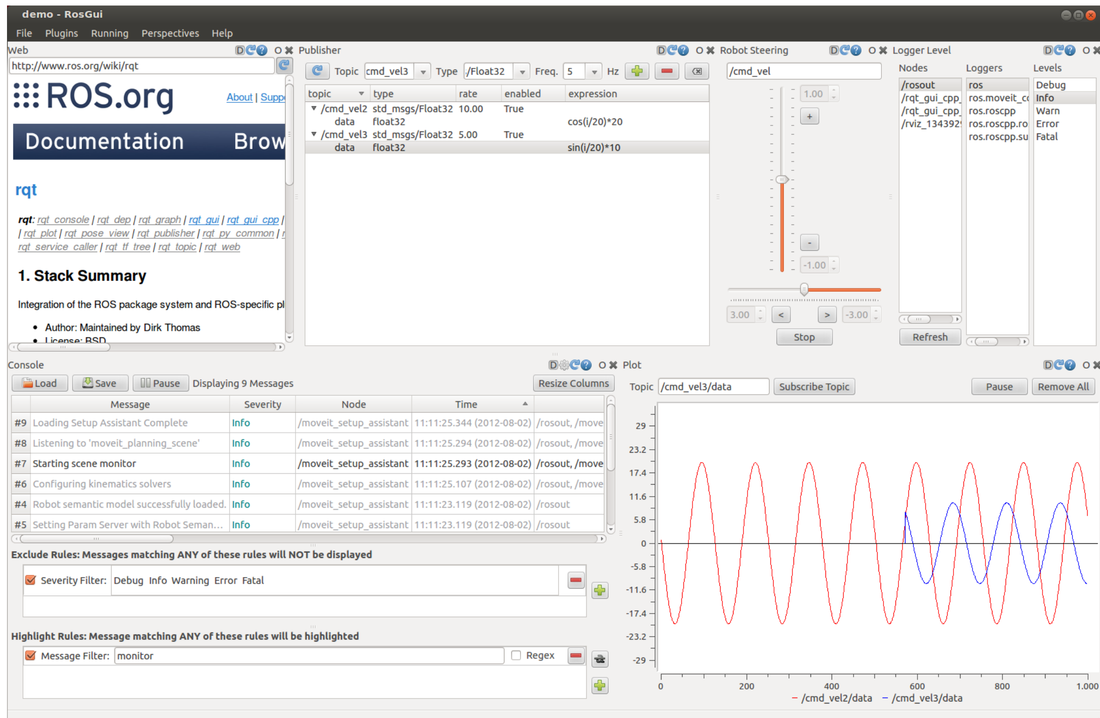


Figura 2.8: Esempio di schermata rqt con diversi plugin visualizzati

Inoltre MongoDB permette di avere una alta affidabilità grazie alla ridondanza dei dati e permette una elevata scalabilità orizzontale grazie all'auto sharding, un metodo che permette di salvare i dati su più macchine. Infine MongoDB supporta i più popolari linguaggi di programmazione tra i quali: Pyhon, C++, Java, C# e JavaScript.

Le caratteristiche del database MongoDB lo rendono perfetto per la nostra applicazione in cui è prevista l'archiviazione di una grossa quantità di dati il cui schema può variare. Inoltre l'elevata scalabilità orizzontale può essere utile in caso si voglia aumentare lo spazio di archiviazione aggiungendo nuovi computer all'architettura che è stata sviluppata.

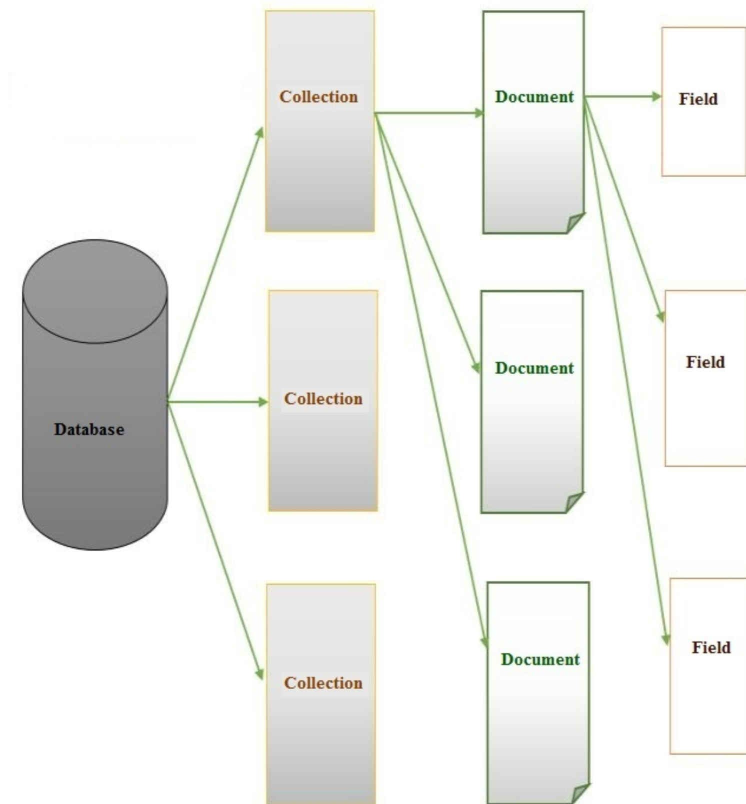


Figura 2.9: Struttura dati di un database MongoDB

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
               street: "123 Fake Street",
               city: "Faketon",
               state: "MA",
               zip: "12345"
             }
}
```

Figura 2.10: Esempio di un documento MongoDB

C A P I T O L O

3. L'hardware del veicolo I.DRIVE

In questo capito verrà fornita una descrizione dell'hardware utilizzato per la realizzazione del sistema I.DRIVE che consiste nel veicolo e in tutti i sensori presenti.

3.1 Il Veicolo I.DRIVE

Il veicolo utilizzato è una Tazzari zero (Figura 3.1), un'auto a due posti con alimentazione e trazione elettrica prodotta dall'azienda italiana Tazzari. La vettura utilizza batterie agli ioni di Litio con una tensione nominale di 80 V e una capacità di 160 Ah che permettono un'autonomia di circa 140 km.

Una ricarica totale, da zero fino al 100% della capacità della batteria, richiede circa 9 ore con batterie in buono stato. Al termine della fase di ricarica vera e propria, il sistema inizia un ciclo di equalizzazione che ha lo scopo di livellare la tensione dei singoli elementi della batteria, in modo da ottimizzare l'autonomia del veicolo e garantire una maggiore durata nel tempo delle batterie. Le batterie vengono utilizzate inoltre per alimentare tutti i sensori addizionali che sono installati a bordo della vettura.

Il motore elettrico è di tipo asincrono trifase a quattro poli e privo di spazzole. Il motore sprigiona una potenza di 15 KW e una coppia massima di 150 Nm. La vettura raggiunge una velocità massima di 100 km/h. La vettura supporta 4 modalità di guida tra cui:

- **Modalità STANDARD**: corrisponde a un'opzione di guida brillante ed adatta all'uso quotidiano in città.

- **Modalità ECONOMY**: permette condizioni di utilizzo meno gravose per le batterie e l'autonomia massima andando a limitare l'accellerazione e la velocità massima.

- **Modalità RAIN**: consente prestazioni simili a quelle della modalità standard, ma con accelerazione e assistenza alla frenata parzializzata, in modo

Figura 3.1: Foto del veicolo I.DRIVE: Tazzari zero

da consentire maggior aderenza e sicurezza in condizioni di pioggia o fondo bagnato.

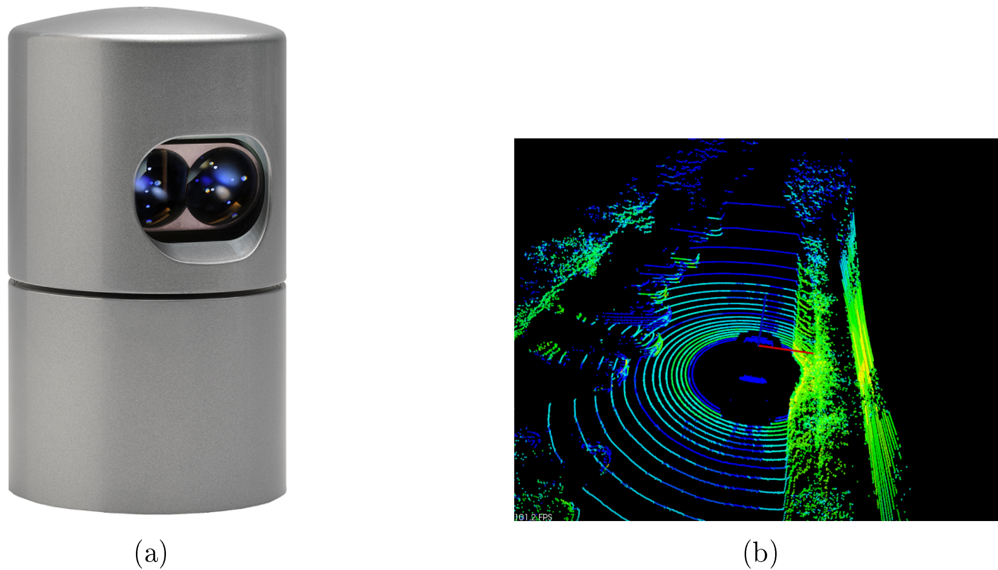
- **Modalità RACE**: consente di ottenere le massime prestazioni dal veicolo, in termini di accelerazione, ripresa e velocità massima. In tale configurazione l'autonomia risulterà ridotta.

3.2 Sensori ambientali

Per l'acquisizione dei dati sull'ambiente circostante e sulla persona alla guida del veicolo sono stati installati diversi sensori di cui verrà fornita una descrizione in questa sezione.

3.2.1 LIDAR

Il LIDAR (*Laser Imaging Detection and Ranging*) è una tecnologia di rilevamento che misura la distanza illuminando un target con un laser e analizzandone la luce riflessa. Nel nostro caso abbiamo utilizzato una *Velodyne HDL-32E* (Figura 3.2a) che permette, tramite l'utilizzo di 32 coppie laser-rilevatore allineati verticalmente da $+10.67°$ a $-30.67°$, di avere un campo di vista verticale di $41.3°$. Grazie al design con testa rotante la Velodyne HDL-32E permette di avere un campo di vista orizzontale di $360°$.



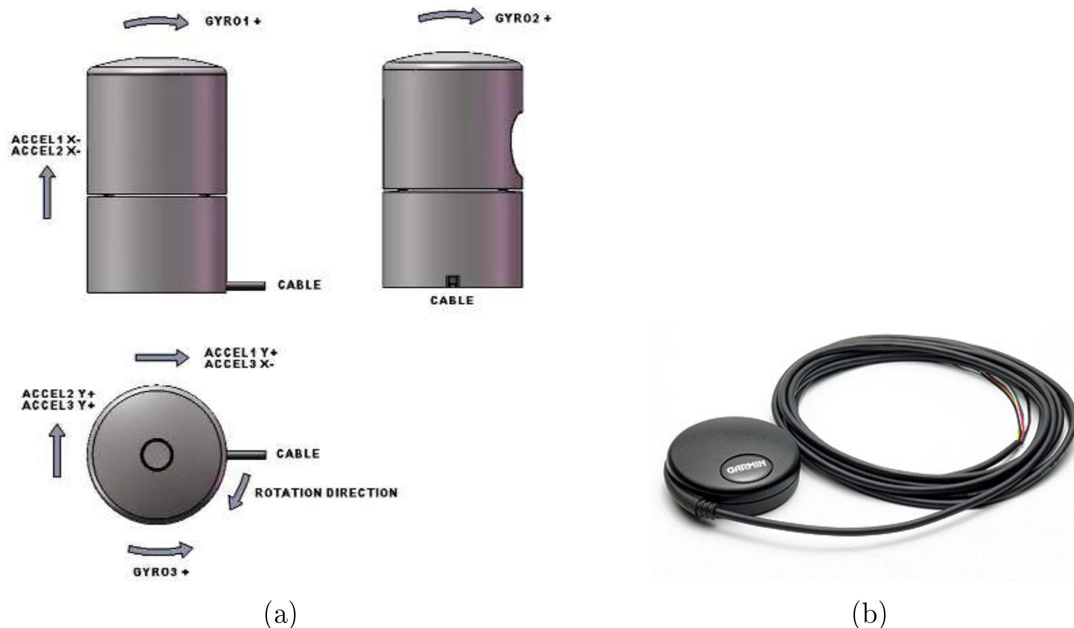
(a) (b)

Figura 3.2: LIDAR: (a) Velodyne HDL-32E, (b) Esempio di nuvola di punti acquisita dalla Velodyne HDL-32E

La Velodyne HDL-32E genera così una nuvola di punti (Figura 3.2b) fino a 700000 punti al secondo in un intervallo di distanze che varia da 1 m a massimo 70 m con un'accuratezza di +/- 2 cm. La Velodyne HDL-32E permette anche la modifica della frequenza di rotazione che può essere selezionata dall'utente in un intervallo che va dai 5 Hz ai 20 Hz (frequenza di rotazione standar 10 Hz). Questo LIDAR può operare anche in ambienti esterni avendo la certificazione IP67 che gli permette di avere una protezione totale contro la polvere e una protezione da immersione temporanea in acqua. Inoltre possiede una temperatura di esercizio che va dai $-10°$ C ai $+60°$ C ed è alimentata tramite un alimentatore esterno che fornisce 12 V.

La Velodyne HDL-32E contiene al suo interno una IMU (*Inertial Measurement Unit*), un sistema elettronico basato su sensori inerziali. L'IMU utilizzata dalla Velodyne HDL-32E è costituita da 3 giroscopi e 3 accelerometri a 2 assi posizionati come in Figura 3.3a. Gli accelerometri presenti sono in grado di misurare le accelerazioni lineari presenti sui 3 assi di riferimento mentre i giroscopi montati misurano la velocità angolare intorno ai 3 assi. È inoltre equipaggiata anche con un ricevitore GPS (*Global Positioning System*) esterno Garmin 18LV (Figura 3.3b) che fornisce la posizione e viene utilizzato per sincronizzare i dati generati dai laser con la precisione degli impulsi di tempo forniti dal GPS. Se il GPS non viene connesso alla Velodyne quest'ultima utilizzerà un proprio clock interno che può essere soggetto ad una deriva di 5 secondi al giorno.

Il ricevitore GPS Garmin genera dati secondo lo standard $GPRMC NMEA



(a) (b)

Figura 3.3: Sensori aggiuntivi della Velodyne HDL-32E: (a) Sistema di riferimento dell'IMU, (b) Ricevitore GPS Garmin 18LV

a 1 Hz e comunica con la Velodyne tramite una porta seriale RS-232.

Nello standard NMEA ogni messaggio è una stringa ASCII con uno specifico formato. Tutti i messaggi standard iniziano con il simbolo del dollaro seguiti da due lettere di prefisso che definiscono il dispositivo che utilizza questo tipo di messaggio (per i ricevitori GPS il prefisso è "GP"). Il prefisso è seguito da tre lettere che definiscono il contenuto del messaggio. Nel nostro caso il messaggio contiene le informazioni minime del GPS ed è indicato utilizzando la sigla "RMC" . Tutti i campi contenuti nel messaggio sono separati da una virgola. Infine il messaggio termina con un carattere di a capo. Un esempio di messaggio proveniente dal GPS utilizzando dalla Velodyne è:

$GPRMC,123519,A,4807.038,N,01131.000,E,022.4,084.4,230394,003.1,W*6A

Nella Tabella 3.1 è possibile vedere il significato di ogni singolo campo del messaggio NMEA RMC.
La comunicazione tra la Velodyne e il PC avviene per mezzo di una connessione Ethernet. La Velodyne invia in uscita 2 pacchetti UDP, uno contenente i dati dei laser e inviato sulla porta 2368 (Figura 3.4) e l'altro contenente i dati del GPS e della IMU e inviato sulla porta 8308 (Figura 3.5). I dati dei giroscopi, degli accelerometri e delle temperature contenuti nel pacchetto UDP devono essere moltiplicati per uno fattore di scala che vale rispettivamente:

- Fattore di scala del giroscopio: 0.09766 deg/sec

Campo	Significato
123519	Ora UTC del fix della posizione (12:35:19 UTC)
A	Stato del GPS: A = fix trovato o V = fix non trovato
4807.038,N	Latitudine e relativa direzione (48 deg 07.038' N)
01131.000,E	Longitudine e relativa direzione (11 deg 31.000' E)
022.4	Velocità al suolo, in nodi
084.4	Direzione di movimento, in gradi reali
230394	Data (23 Marzo 1994)
003.1	Variazione magnetica in gradi
W	Verso della variazione magnetica
*6A	Checksum che inizia sempre con un asterisco

Tabella 3.1: Messaggio NMEA RMC

- Fattore di scala dell'accelerometro: 0.001221 G
- Fattore di scala della temperatura: 0.1453 + 25° C

3.2.2 Ricevitore GPS

Oltre al ricevitore GPS Garmin montato sulla Velodyne è stato utilizzato un ulteriore ricevitore GPS che si chiama Yuan10. L'utilizzo di un ulteriore ricevitore GPS è dovuto ai problemi che abbiamo riscontrato nell'utilizzare, tramite ROS, il ricevitore GPS Garmin per la sincronizzazione del clock del PC. I problemi di sincronizzazione maggiori per il GPS Garmin sono dovuti al fatto che deve essere acquisito tramite pacchetti UDP e che i dati del GPS acquisiti devono passare per una catena di nodi ROS prima di poter essere utilizzati. In questo modo, i dati temporali dal GPS Garmin ricevuti, presentano un ritardo variabile non accettabile per la sincronizzazione fine del PC. Il ricevitore GPS Yuan10, infatti, è utilizzato solamente per la sincronizzazione del PC e i dati provenienti da esso non vengono salvati. Vengono salvati solamente i dati sulla posizione provenienti dal ricevitore Garmin della Velodyne.

Il ricevitore GPS Yuan10 (Figura 3.6) monta al suo interno il ricevitore *Skytraq S1315F-RAW*. Questo ricevitore possiede 65 canali GPS e permette di avere il primo fix della posizione in un tempo molto breve.

La grande sensibilità del ricevitore (-148 dBm) in caso di avvio a freddo gli permette di acquisire, tracciare e ottenere il primo fix della posizione in ambienti in cui il segnale del GPS è debole. Inoltre l'elevata sensibilità durante il tracciamento (-161 dBm) permette una copertura continua della posizione in quasi tutti gli ambienti di applicazione. Nella Tabella 3.2 è possibile vedere un confronto tra le caratteristiche dei due GPS utilizzati nel sistema.

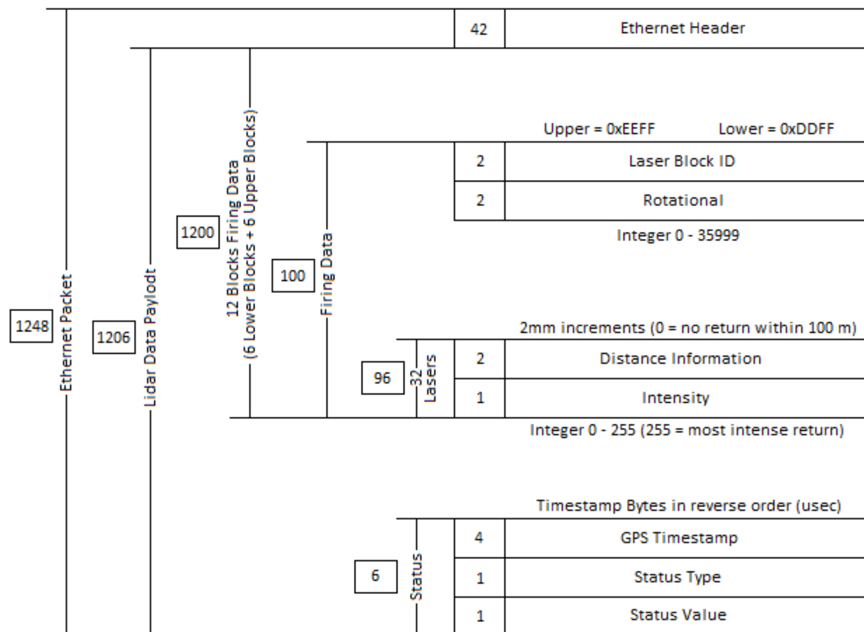


Figura 3.4: Formato del pacchetto UDP dei laser (Porta 2368)

42	Ethernet Header
14	Not used
2	Gyro 1
2	Temp 1
2	Accel1 X
2	Accel1 Y
2	Gyro 2
2	Temp 2
2	Accel2 X
2	Accel2 Y
2	Gyro 3
2	Temp 3
2	Accel3 X
2	Accel3 Y
160	Not used
4	GPS timestamp from top of hour
4	Not used
72	NMEA sentence
234	Not used

Figura 3.5: Formato del pacchetto UDP del GPS e dell'IMU (Porta 8308)

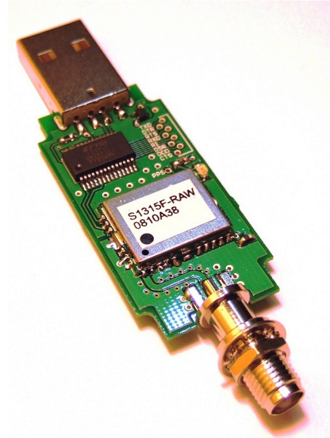


Figura 3.6: GPS Yuan10 con ricevitore Skytraq S1315F-RAW

Caratteristiche	**Garmin 18LV**	**Yuan10**
Tempo avvio a caldo	1 s	1 s
Tempo avvio a caldo	38 s	<29 s
Tempo avvio a freddo	45 s	29 s
Riacquisizione	< 2 s	1 s
Sensitività	-185 dBW minimo	-148dBm, -161dBm
frequenza di aggiornamento	1 Hz	1, 2, 4, 5, 8, 10 Hz

Tabella 3.2: Tabella di confronto delle caratteristiche tra i ricevitori GPS Garmin 18LV e Yuan10

Questo GPS può fornire i dati sulla posizione, ad una frequenza massima di 20 Hz (nel nostro caso viene acquisito a 1 Hz), in formato binario o con lo standard *NMEA*. Il formato NMEA utilizzato da questo GPS è di tre tipi: RMC e GGA e GSA. Ognuno dei diversi messaggi NMEA ha un contenuto informativo diverso. Lo standard NMEA RMC è già stato trattato precedentemente per il GPS Garmin equipaggiato sulla Velodyne (Tabella 3.1).
Nel messaggio NMEA GGA sono invece contenute le informazioni sul fix del GPS. Un esempio di messaggio NMEA GGA è:

$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,,*47

NellaTabella 3.3 è descritto il significato di ogni singolo campo del messaggio NMEA GGA.

Nel messaggio NMEA GSA sono contenute le informazioni sui satelliti. Un esempio di messaggio NMEA GSA è:

$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39

Campo	Significato
123519	Ora UTC del fix della posizione (12:35:19 UTC)
4807.038,N	Latitudine e relativa direzione (48 deg 07.038' N)
01131.000,E	Longitudine e relativa direzione (11 deg 31.000' E)
1	Qualità del fix: 0 = invalido 1 = fix del GPS 2 = fix del DGSP 3 = fix del PPS 4 = cinematica in tempo reale 5 = float RTK 6 = stima 7 = modalità di input manuale 8 = modalità di simulazione
08	Numeri di satelliti tracciati
0.9	HDOP (*Hotizontal Dilution Of Precision*)
545.4,M	Altitudine, in metri, sopra il livello del mare
46.9,M	Separazione geoidale, in metri
Campo vuoto	Tempo trascorso dall'ultimo aggiornamento DGPS
Campo vuoto	ID della stazione DGPS
*47	Checksum che inizia sempre con un asterisco

Tabella 3.3: Messaggio NMEA GGA

NellaTabella 3.4 è descritto il significato di ogni singolo campo del messaggio NMEA GSA.

Questo ricevitore GPS, se usato nel formato binario, permette di utilizzare RTKLIB. RTKLIB è una libreria open source che consente il posizionamento preciso attraverso il GNSS (*Global Navigation Satellite System*). Questa libreria supporta varie modalità di posizionamento tra cui il DGPS (*Differential Global Positioning System*) che permette di migliorare il posizionamento basato sul GPS combinando i dati provenienti da due ricevitori GPS, uno posizionato sul veicolo del quale si vuole conoscere la posizione e uno posizionato a terra in una postazione fissa. Questa modalità di posizionamento permette così di ridurre l'errore sulla posizione da 3-5 m a 10 cm. AL momento non si fa utilizzo della modalità DGPS nel nostro sistema poiché non esiste un nodo ROS che permetta l'utilizzo di RTKLIB.

Il GPS Yuan10 si connette al PC per mezzo di un cavo USB standard.

Campo	Significato
A	selezione della modalità del fix (A = selezione automatica tra le modalità del fix 2D o 3D, M = selezione manuale)
3	Modalità del fix 1 = non valido 2 = Fix 2D 3 = Fix 3D
04,05...	Codici PRN (*Pseudorandom Noise Number*) che identificano i satelliti usati per il fix (lo spazio massimo nel messaggio è di 12 satelliti)
2.5	PDOP (Diluition of Precision)
1.3	HDOP (*Hotizontal Dilution Of Precision*)
2.1	VDOP (*Vertical Dilution Of Precision*)
*39	Checksum che inizia sempre con un asterisco

Tabella 3.4: Messaggio NMEA GSA

3.2.3 IMU

La IMU (*Inertial Measurement Unit*) che è stata utilizzata è una Xsens MTi (Figura 3.7) che integra al suo interno un processore di segnali a bassa potenza che fornisce l'orientamento 3D utilizzando i dati 3D calibrati dell'accelerazione, la velocità di rotazione e del campo magnetico terrestre. Al suo interno infatti la Xsens MTi presenta un magnetometro 3D (bussola 3D) che fornisce i dati sul campo magnetico terrestre,un accelerometro 3D che fornisce i dati sulle accelerazioni lineari e un giroscopio 3D che fornisce la velocità di rotazione. La frequenza di acquisizione massima alla quale la Xsens MTi arriva è di 100 Hz. Nella nostra configurazione la frequenza di acquisizione è impostata su quella massima di 100 Hz. Tutte le letture calibrate dei sensori presenti (accelerazione, velocità di rotazione, campo magnetico terrestre) utilizzano la regola della mano destra per definire il sistema di riferimento come visibile nella Figura 3.7.Questo sistema di riferimento è fisso rispetto al corpo del sensore. Il sistema di riferimento fisso della terra che viene utilizzato come riferimento per calcolare l'orientamento del sensore è anch'esso definito con la regola della mano destra con:

- X positivo quando punta verso il nord magnetico
- Y segue la regola della mano destra (ovest)
- Z positivo rivolto verso l'alto

La IMU Xsens Mti comunica con il PC per mezzo di un adattatore USB standard.

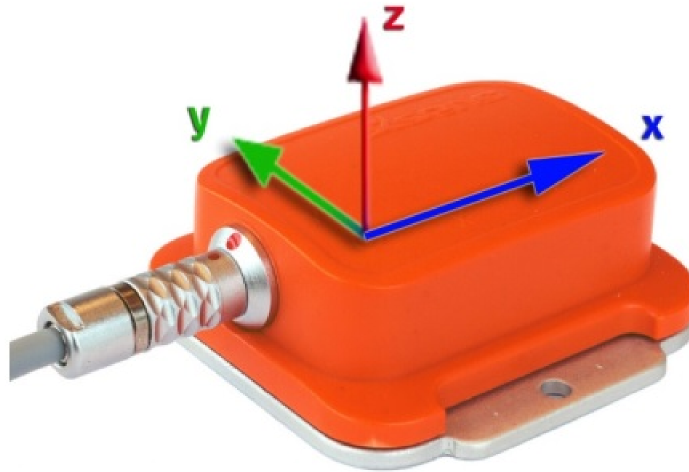


Figura 3.7: Xsens MTi con sovrapposto il sistema di riferimento fisso

3.2.4 Telecamere esterne

All'esterno del veicolo, sopra al tettuccio, sono state montate due telecamere *Prosilica GC1020*,una rivolta anteriormente sulla strada e una rivolta sul lato destro del veicolo. Queste telecamere vengono utilizzate per acquisire le immagini dell'ambiente esterno.

La Prosilica GC1020 (Figura 3.8a) è una camera di rete che utilizza una connessione ethernet per comunicare con il PC. Questa camera supporta la connessione Gigabit ethernet per consentire l'invio di immagini non compresse ad un frame rate elevato. La Prosilica GC1020 monta un sensore Sony ICX204 CCD Progressive da 1/3" che permette di acquisire immagini a colori fino ad un massimo di 33 fps alla risoluzione massima di 1024 x 768. Questa camera supporta diversi formati in uscita che sono:

- Formati mono: Mono8, Mono12, Mono12Packed
- Formati a colori RGB: RGB8Packed, BGR8Packed
- Formati raw: BayerRG8, BayerRG12, BayerGR12Packed

Le Prosilica GC1020 che utilizziamo sono state configurate per funzionare a 30 fps alla risoluzione di 1024 x 768 e utilizziamo il formato BayerRG8 così da limitare la dimensione dei dati che vengono forniti in output. La prosilica necessita di una alimentazione esterna da 5-25 V per mezzo di un connettore proprietario.

La Prosilica GC1020, rivolta anteriormente sulla strada, monta un'ottica particolare chiamata Theia MY125M (Figura 3.8b) che grazie alla lunghezza focale di 1.3 mm permette di avere un campo visivo orizzontale di 125° e un campo

visivo verticale di 109° con una distorsione molto bassa. Inoltre quest'ottica presenta una rapporto focale di F/1.8. La telecamera rivolta verso il lato destro del veicolo, invece, monta un'ottica standard con una lunghezza focale di 3.5 mm e un rapporto focale di F/1.4. In Figura 3.9 è possibile vedere la differenza di ampiezza del campo visivo tra le due ottiche utilizzate.

(a) (b)

Figura 3.8: Camera Prosilica: (a) Prosilica GC1020, (b) Ottica Theia MY125M

3.3 Sensori per il guidatore

Per l'acquisizione dei dati sulla persona alla guida del veicolo sono stati installati diversi sensori di cui verrà fornita una descrizione in questa sezione.

3.3.1 Sensori per l'acquisizione di parametri fisiologici

I sensori per l'acquisizione dei parametri fisiologici sono stati utilizzati per poter acquisire dati RAW sui parametri fisiologici della persona alla guida del veicolo. Sono state utilizzate due diversi tipologie di sensori che verranno di seguito descritte.

Procomp Infiniti

La scheda Procomp Infiniti (Figura 3.10a) è un encoder a 8 canali utilizzato per acquisire in tempo reale, per mezzo di diversi sensori, i parametri fisiologici di una persona. Di questi 8 canali 2 campionano a 2048 campioni/sec e gli altri 6 campionano a 256 campioni/sec. La Procomp Infiniti campiona i segnali in ingresso provenienti dai sensori, li digitalizza, li codifica e trasmette i dati acquisiti all'interfaccia TT-USB. Per la trasmissione dall'encoder all'interfaccia TT-USB viene utilizzato un cavo in fibra ottica che permette massima libertà di movimento, fedeltà del segnale e isolamento elettrico. La lunghezza massima del cavo in fibra ottica che può essere utilizzata, senza avere un degrado del

(a) (b)

Figura 3.9: Esempio di immagini acquisite dalle prosilica: (a) Immagine acquisita tramite l'ottica theia, (b) Immagine acquisita tramite l'ottica standard

segnale, è di 7,62 metri. L'interfaccia TT-USB si occupa quindi di ricevere il segnale dall'encoder in formato ottico e lo trasforma nel formato USB in modo da comunicare con il PC per mezzo di una USB standard.

Tutti i sensori non sono particolarmente invasivi e richiedono, in alcuni casi, una piccola preparazione per essere utilizzati. I sensori forniti con la Procomp Infiniti permettono di acquisire dati riguardanti:

- Elettromiografia (EMG)
- Elettrocardiogramma (EKG)
- Elettroencefalogramma (EEG)
- Conduttanza della pelle
- Temperatura della pelle
- Respirazione (forma d'onda, frequenza e ampiezza)
- Battito cardiaco (forma d'onda, frequenza cardiaca e ampiezza) (BVP)

Ogni canale della Procomp Infiniti può acquisire indistintamente qualsiasi sensore fatta eccezione per il sensore dell'elettromiografia che necessita di uno dei canali a 2048 campioni/sec.
L'encoder è alimentato da quattro batterie di tipo AA che permettono di avere un'autonomia che varia da un minimo di 20 ore a un massimo di 30 ore di utilizzo.

(a) (b)

Figura 3.10: Sensore per i parameteri fisiologici: (a) Procomp Infiniti, (b) Esempi di sensori disponibili per la Procomp Infiniti: in alto partendo da sinistra abbiamo il sensore per acquisire i dati riguardanti EMG, EKG, EEG e conduttanza della pelle; in basso partendo da sinistra abbiamo i sensori per acquisire temperatura corporea, respirazione e battito cardiaco

Empatica E4

Empatica E4 (Figura 3.11) è un braccialetto che si indossa al polso e permette di acquisire in tempo reale alcuni parametri fisiologici grazie ai sensori che sono presenti al suo interno. I sensori presenti sono:

- Termometro a infrarossi: questo sensore permette di acquisire la temperatura della pelle. Il sensore ha una frequenza di campionamento di 4 Hz con una risoluzione di 0.02° C.

- Accelerometro a 3 assi: questo sensore permette di acquisire l'accelerazione lineare presente sui 3 assi in un intervallo che va da -2 g a +2 g. La frequenza di campionamento del sensore è di 32 Hz.

- Sensore fotopletismografico (PPG): questo sensore permette di misurare tramite dei LED e dei fotodiodi la variazione del volume del sangue nell'arteria sotto la pelle dovuto al ciclo cardiaco(BVP). La frequenza di campionamento di questo sensore è di 64 Hz.

- Sensore dell'attività elettrodermica (EDA): questo sensore permette di misurare tramite due elettrodi le caratteristiche elettriche della pelle in un intervallo che va da 0.01 µSiemens a 100 µSiemens. Il sensore ha una frequenza di campionamento di 4 Hz e una risoluzione di 900 pSiemens.

Tutti questi sensori sono presenti nella parte sottostante del braccialetto E4 tranne gli elettrodi del sensore EDA che sono posti sul cinturino. Il braccialetto presenta due modalità di funzionamento:

Figura 3.11: Braccialetto Empatica E4

- Modalità recording: in questa modalità il braccialetto salva i dati acquisiti nella memoria interna in cui è possibile salvare fino a circa 60 ore di attività. I dati possono essere scaricati successivamente su PC per mezzo di un cavo USB e di un software apposito

- Modalità streaming: in questa modalità il braccialetto deve essere connesso ad uno smartphone o ad un PC tramite bluetooth. La connessione bluetooth consente la visualizzazione e il salvataggio in tempo reale dei dati acquisiti dal braccialetto.

La modalità che viene utilizzata per i nostri scopi è la modalità streaming poiché siamo interessati ai dati in tempo reale. Il PC quindi per poter comunicare con il braccialetto deve essere dotato di un adattatore bluetooth prodotto dalla Bluegiga poichè è l'unico compatibile con il software fornito con il braccialetto. Il braccialetto Empatica E4 è alimentato da una batteria integrata che si ricarica in circa 2 ore e permette un'autonomia in modalità streaming di circa 20 ore e in modalità recording di circa 36 ore.

3.3.2 Telecamere interne

All'interno dell'abitacolo del veicolo sono state montate 2 telecamere *Axis P1343*, una utilizzata per riprendere il guidatore ed una utilizzata per riprendere la strada. Queste camere sono state utilizzate per acquisire le immagini del guidatore.

La Axis P1343 (Figura 3.12) è una telecamera di rete che utilizza una connessione ethernet per comunicare con il PC. La telecamera Axis P1343 monta un sensore RGB CMOS Progressive Scan da 1/4" che permette di ottenere un flusso video a colori a 30 fps con una risoluzione massima di 800x600. Questa telecamera fornisce un flusso video in formato H.264 o Motion JPEG. La nostra scelta è ricaduta sul formato video Motion JPEG che fornisce una eccellente qualità di

Figura 3.12: Camera Axis P1343: sono presenti 2 telecamere sul veicolo I.DRIVE, una che punta verso il guidatore e una che punta verso la strada

immagine a discapito di un consumo di banda considerevole.
L'accesso al flusso del video Motion JPEG avviene per mezzo di un URL come di seguito riportato:

http://<camera ip>/mjpg/video.mjpg?fps=0&resolution=800x600

L'ottica montata sulla telecamera che punta il guidatore presenta una lunghezza focale di 2.3 mm e un rapporto focale di F/1.4 mentre l'ottica montata sulla telecamera che punta la strada ha una lunghezza focale variabile da 3 mm a 8 mm e un rapporto focale di F/1.4.

Questa telecamera può essere alimentata tramite alimentazione esterna da 8-20 V per mezzo di un connettore proprietario oppure può essere alimentata direttamente tramite la porta ethernet grazie al supporto del PoE (Power over Ethernet). Nel nostro caso la telecamera è alimentata tramite alimentazione esterna.

3.4 Computer

Il computer utilizzato è uno *Shuttle slim* (Figura 3.13) che permette di avere in dimensioni contenute un processore Intel Core i7-4790S con 4 core e una frequenza base di 3.2 GHz, scheda video integrata Intel HD Graphics 4600, 8 GB di RAM, 1 disco a stato solido da 64 GB su cui è installato il sistema operativo e 1 hard disk rotativo a 7200 RPM da 750 GB dove vengono salvati tutti i dati acquisiti.

Il sistema operativo installato sul computer è Ubuntu-Linux nella versione 14.04. La scelta di Ubuntu è dovuta al fatto che questo sistema operativo è perfettamente compatibile con ROS Indigo e la totalità dei sensori sopra descritti fatta eccezione per i sensori per l'acquisizione dei parametri fisiologici. Questi ultimi sensori infatti funzionano solo su sistemi operativi Windows e quindi è stato necessario creare una macchina virtuale con sistema operativo Windows 8 tramite il software VirtualBox.

Figura 3.13: Computer Shuttle Slim

Il computer riceve tutte le letture effettuate dai sensori, elabora i dati acquisiti, permette la visualizzazione in tempo reale dei dati acquisiti ed infine effettua il salvataggio dei dati ricevuti.

3.5 Analisi dei consumi del sistema

Nella Tabella 3.5 vengono elencati i componenti del nostro sistema che necessitano di un'alimentazione esterna con la relativa tensione di alimentazione e consumo. Il consumo complessivo massimo del nostro sistema è di 97,3 W.

Componente	**Tensione di alimentazione**	**Consumo**
Velodyne HDL-32E	12 V	12 W
Yuan10	5 V	2.5 W
Xsens MTi	5 V	2.5 W
Prosilica GC1020	12 V	2.9 W
Axis P1343	12 V	6.4 W
PC shuttle slim	12 V	65 W
Switch Netgear GS308v2	12 V	6 W

Tabella 3.5: Elenco dei componenti del sistema con relativa tensione di alimentazione e consumo massimo stimato

C A P I T O L O

4. Architettura Software

In questo capitolo verrà fornita una descrizione dell'architettura software sviluppata per il progetto I.DRIVE. Come prima cosa verrà data una descrizione generale dell'architettura sviluppata. Verrà poi descritta la tecnologia utilizzata per il salvataggio dei dati. seguita dalla descrizione approfondita dei moduli usati per comunicare con i sensori. Successivamente verrà descritto il modulo software usato per controllare che il funzionamento dell'intero sistema e infine verranno descritti le tecnologie usate per la sincronizzazione tra i sensori.

4.1 Architettura generale

Per sviluppare il nostro sistema abbiamo utilizzato ROS, un framework open source che è diventato uno standard in molte applicazioni nel mondo della robotica. Una delle caratteristiche fondamentali di ROS è la modularità resa possibile grazie alla struttura basata su pacchetti, nodi, messaggi e servizi. Sfruttando questa caratteristica è stato possibile realizzare una architettura modulare per il nostro progetto.

La ragione principale che ci ha portato a sviluppare una architettura modulare in ROS è che in questo modo, abbiamo la possibilità di aggiungere, sostituire e togliere facilmente componenti (ad esempio i sensori) e i relativi moduli software senza andare a compromettere l'intero sistema. Questo consente all'intero progetto di essere facilmente modificabile man mano che lo sviluppo prosegue. La nostra architettura presenta così nodi indipendenti per ciascuno dei sensori e per ogni funzionalità specifica.

La scelta di utilizzare ROS ha comportato l'utilizzo di una macchina con sistema operativo Ubuntu Linux nella versione 14.04 poiché perfettamente compatibile con ROS Indigo e la quasi totalità dei sensori da noi utilizzati. Entrambi i sensori dei parametri fisiologici (Procomp Infiniti e Empatica E4) però, non presentano la possibilità di essere utilizzati con un sistema operativo Linux. Per questo motivo è stato necessario creare una macchina virtuale con sistema operativo Windows 8 utilizzando il software VirtualBox. Sulla macchina virtuale sono stati semplicemente installati i software e le librerie che consentono l'utilizzo dei

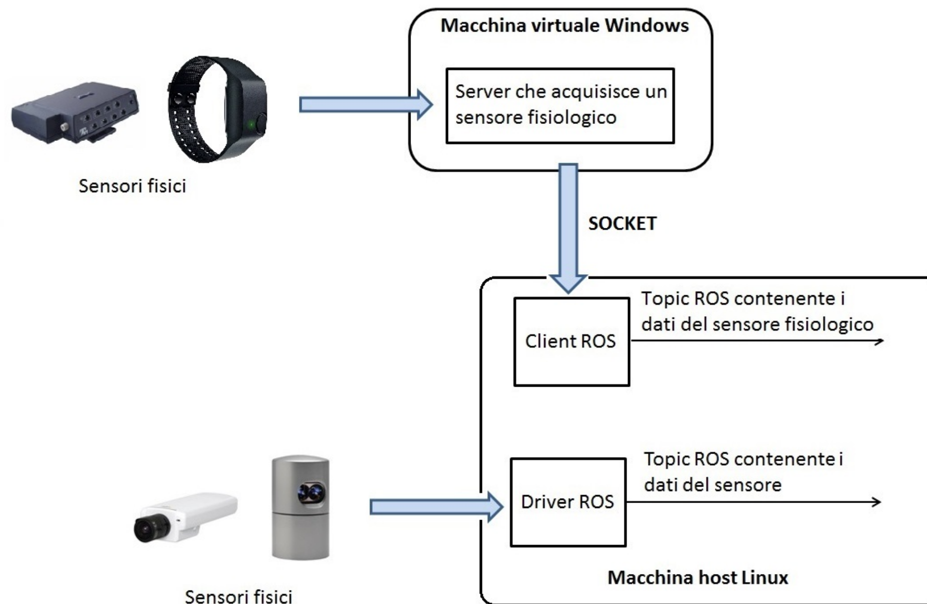


Figura 4.1: Schema che mostra come la macchina virtuale sia usata per acquisire i sensori fisiologici nel nostro sistema rispetto agli altri sensori

sensori dei parametri fisiologici. In Figura 4.1 è possibile vedere uno schema che mostra come la macchina virtuale sia utilizzata nella nostra architettura per acquisire i sensori dei parametri fisiologici rispetto agli altri sensori.

La macchina virtuale e la macchina fisica presentano entrambe un indirizzo IP statico così che possano comunicare facilmente tramite socket TCP. L'indirizzo IP statico è stato utilizzato anche per tutti i sensori che presentano una interfaccia ethernet. Gli indirizzi IP statici assegnati sono visualizzabili nella Tabella 4.1.

4.2 Log dei dati in MongoDB tramite ROS

All'interno di ROS è già presente un pacchetto che permette di interagire con un database MongoDB. Questo pacchetto si chiama `mongodb_store` e contiene nodi e librerie che permettono il salvataggio e il recupero dei dati relativi a ROS in un database MongoDB utilizzando C++ e python. Le funzioni principali di questo pacchetto sono il salvataggio e il recupero di singoli messaggi, il logging dei topic nel database e la riproduzione dei dati salvati. Questo pacchetto è suddiviso due sotto pacchetti che presentano funzioni diverse:

- `mongodb_store` è il pacchetto che si interfaccia con il database e permette il salvataggio e il recupero dei dati e permette inoltre la riproduzione dei

Sensore	Indirizzo IP
Computer	192.168.1.3
Macchina virtuale	192.168.1.10
Velodyne	192.168.1.201
Prosilica GC1020 1	192.168.1.121
Prosilica GC1020 2	192.168.1.122
Axis P1343 1	192.168.1.101
Axis P1343 2	192.168.1.103

Tabella 4.1: Tabella degli indirizzi IP statici assegnati

dati salvati indicando il nome del topic che si vuole andare a riprodurre

- `mongodb_log` è il pacchetto che si occupa di prendere i dati provenienti dai topic e, utilizzando la libreria messa a disposizione dal pacchetto `mongodb_store`, salvarli nel database replicando i campi del messaggio nella struttura del documento MongoDB. Al documento salvato vengono aggiunti dei meta dati come la data di inserimento nel database del messaggio, il nome del topic dal quale proviene il messaggio e il tipo del messaggio utilizzato dal topic. Questi meta dati servono successivamente per poter riprodurre i dati salvati.

`Mongodb_log` è formato da un nodo python che analizza il tipo del messaggio pubblicato sul topic e a seconda del tipo di messaggio sceglie se andare ad utilizzare o meno dei nodi scritti in C++ e quindi più efficienti. I tipi di messaggio per cui `mongodb_log` fornisce dei nodi scritti in C++ sono: `sensor_msgs/CompressedImage`, `sensor_msgs/PointCloud`, `tf/tfMessage`, `rviz_intel/TriangleMesh`. Il nodo python è stato successivamente modificato per poter supportare uno nodo aggiuntivo C++ che permettesse il log del messaggio `sensor_msgs/Image`.

`mongodb_store` contiene al suo interno diversi nodi ognuno dei quali con una diversa funzionalità, ma quello che viene usato nella nostra applicazione è il nodo `mongodb_play` che permette la riproduzione dei dati salvati nel database. Il nodo `mongodb_play` consente la riproduzione di uno o più topic i cui dati sono stati salvati sul database MongoDB. Questo nodo prevedeva, per ogni topic, la riproduzione di tutti i dati presenti nel database e non la riproduzione di solo una parte dei dati; è stato quindi modificato per poter ricevere 2 argomenti aggiuntivi che indicano la data di inizio e la data di fine per cui si vuole riprodurre i dati. La modifica effettuata comporta l'esecuzione di una query al database MongoDB che va a selezionare solo i dati nel periodo di tempo interessato; per riprodurre i messaggi al tempo corretto di acquisizione il nodo crea quindi un propio clock simulato. Il nodo `mongodb_play` utilizza i parametri di ROS `mongodb_host` e

mongodb_port nei quali deve essere indicato l'indirizzo IP e la porta del database MongoDB che si vuole utilizzare.

Nella nostra applicazione viene eseguito un nodo Mongodb_log per ogni topic che si vuole andare ad acquisire. Questo è stato fatto poiché così si ha un nodo indipendente che effettua il log per ciascun topic e permette un miglior controllo sul funzionamento di ciascuno di essi. Inoltre, in questo modo, è possibile escludere facilmente uno o più nodi che si occupano del log.

4.3 Sensori ambientali

In questa sezione vengono descritti i pacchetti ROS utilizzati per poter acquisire e configurare i diversi sensori ambientali indicandone le principali caratteristiche che sono state utilizzate.

4.3.1 Velodyne

Per il sensore LIDAR Velodyne HDL-32E è già presente un pacchetto ROS che presenta al suo interno il driver e altri strumenti utili per configurare il dispositivo e convertire i dati raw dei laser acquisiti dalla Velodyne in un messaggio ROS che permette una facile visualizzazione degli stessi. Questo pacchetto si chiama velodyne. Il pacchetto velodyne è composto da due pacchetti distinti:

- velodyne_driver: questo pacchetto, come indica esplicitamente il nome, contiene il driver per la Velodyne
- velodyne_pointcloud: questo pacchetto contiene il nodo per la conversione dei dati raw acquisiti dalla Velodyne, grazie al driver ROS, in un messaggio ROS sensor_msgs/PointCloud2

Il pacchetto velodyne_driver contiene al suo interno il nodo velodyne_node che cattura i dati provenienti dalla Velodyne e li pubblica in formato raw. Questo nodo pubblica i dati raw ricevuti (tipicamente per una intera rotazione del dispositivo) su un topic chiamato velodyne_packet che è del tipo velodyne_msgs/VelodyneScan. Inoltre il nodo pubblica un topic chiamato diagnostics che contiene informazioni sullo stato del sensore utilizzando il messaggio diagnostic_msgs/DiagnosticStatus.
Il messaggio diagnostic_msgs/DiagnosticStatus è semplicemente formato da un numero che identifica lo stato di operatività (OK = 0, WARN = 1, ERROR = 2, STALE = 3), una descrizione sul componente che riporta lo stato, un messaggio che descrive in maniera dettagliata lo stato e una stringa che rappresenta univocamente l'hardware che ha inviato lo stato.

Il messaggio velodyne_msgs/VelodyneScan invece è un tipo personalizzato ed è definito all'interno del pacchetto velodyne. Questo messaggio è composto da un header contenente il timestamp dell'ultimo pacchetto fornito dalla

velodyne incluso nel messaggio e da un vettore di messaggi `velodyne_msgs/VelodynePacket`. Il tipo `velodyne_msgs/VelodynePacket` è anch'esso definito nel pacchetto `velodyne` ed è formato dal timestamp del singolo pacchetto UDP ricevuto e dai dati raw ricevuti per il singolo pacchetto UDP.

I parametri che possono essere impostati per questo nodo sono:

- *frame_id*: serve per definire il sistema di riferimento da utilizzare (default: "velodyne")
- *model*: il modello della Velodyne utilizzato visto che il nodo supporta diversi modelli (default: "64E")
- *npackets*: numero di pacchetti pubblicati per ogni messaggio `velodyne_msgs/VelodyneScan` (default: un numero sufficiente per un completo giro della velodyne)
- *rpm*: specificare i giri al minuto che la Velodyne deve fare (default: 600 che corrisponde a 10 Hz)

Il pacchetto `velodyne_pointcloud` contiene al suo interno un *nodelet*, chiamato `CloudNodelet`, che legge i dati raw pubblicati sul topic `velodyne_packet` dal driver della Velodyne e li converte in un messaggio `sensor_msgs/PointCloud2` che viene pubblicato sul topic `velodyne_points`. Oltre alle informazioni sui punti XYZ il tipo di messaggio `sensor_msgs/PointCloud2` include informazioni sulla intensità di ciascun punto. In aggiunta, il formato del messaggio pubblicato dal nodelet include un header, la struttura 2D della nuvola di punti (larghezza e altezza), un array di `sensor_msgs/PointField` che descrive i canali presenti per ogni punto e il loro layout, la dimensione in bytes occupata da ciascun punto, la dimensione in bytes di ogni riga e un array che contiene tutti i dati dei punti.
I parametri configurabili per questo nodo sono:

- *max_range*: indica il valore massimo in metri entro cui devono stare i dati da pubblicare (default:130 m)
- *min_range*: indica il valore minimo in metri entro cui devono stare i dati da pubblicare (default:2 m)
- *calibration*: indica il percorso del file YAML che contiene le informazioni specifiche sulla calibrazione del sensore

Il pacchetto `velodyne_pointcloud` include al suo interno anche un nodo chiamato `gen_calibration.py` che permette la generazione del file YAML di calibrazione partendo da un file di calibrazione in formato XML (fornito insieme al dispositivo Velodyne).Il file YAML ha una struttura gerarchica e al suo interno sono contenute le informazioni di calibrazione per ciascuno dei 32 laser. Il file

YAML generato da questo nodo è quello che deve essere passato come parametro al nodelet `CloudNodelet`.

I pacchetti ROS descritti fino ad ora si occupano di estrarre e convertire solamente i dati relativi ai laser della Velodyne. Per fare questo estraggono i dati provenienti dal pacchetto UDP sulla porta 2368 che è stato descritto nel Capitolo 3 (Figura 3.4). Come decritto nello stesso capitolo la Velodyne contiene anche un GPS e una IMU e i dati relativi a questi sensori vengono inviati tramite il pacchetto UDP sulla porta 8308 (Figura 3.5). Per estrarre i dati del GPS e della IMU contenuti nel pacchetto UDP è stato necessario sviluppare un nodo python apposito poiché in ROS non è presente nessun pacchetto che si occupa di estrarre questi dati.

Il nodo sviluppato si chiama `velodyne_extra.py` e si occupa appunto di ricevere i pacchetti UDP sulla porta 8308 pubblicando su due differenti topic i dati della IMU e del GPS. In particolare il nodo si occupa di estrarre il messaggio NMEA del GPS pubblicandolo sul topic chiamato `nmea_sentence`.Questo topic utilizza il messaggio `nmea_msgs/Sentence` , formato da un header contenente il timestamp fornito tramite il pacchetto e da una stringa contenente la frase NMEA.

Inoltre il nodo estrae e converte, dal pacchetto UDP ,i dati degli accelerometri e dei giroscopi provenienti dalla IMU e li converte in un messaggio `sensor_msgs/Imu`. Questi messaggi vengono pubblicati sul topic `imu_data` che è formato da: un header, un quaternione per l'orientamento con relativa matrice di covarianza (non utilizzato in questo nodo), un vettore di 3 elementi per la velocità angolare con relativa matrice di covarianza e un vettore di 3 elementi per l'accelerazione lineare. La frequenza con cui il nodo Velodyne acquisisce i dati è di 100 Hz.

I dati provenienti dai topic `velodyne_points`, `velodyne_packet` e `imu_data` sono salvati, all'interno del database MongoDB, secondo la struttura in Figura 4.2, in Figura 4.3 e in Figura 4.4.

4.3.2 GPS Velodyne

Come descritto nel Capitolo 3 il GPS Garmin della Velodyne fornisce un messaggio nello standard $GPRMC NMEA. Nello standard NMEA ogni messaggio è una stringa ASCII con una struttura ben definita. ROS fornisce già un pacchetto che permette l'analisi della stringa NMEA proveniente da un topic e la converte in un messaggio standard di ROS. Il nodo che permette questo si chiama `nmea_topic_driver` ed è contenuto nel pacchetto `nmea_navsat_driver`. Il nodo `nmea_topic_driver` si sottoscrive quindi ad un topic il cui nome deve essere `nmea_sentence` e il quale formato del messaggio deve essere `nmea_msgs/Sentence`. Il *frame_id* e il *timestamp* impostati nell'header del topic `nmea_sentence` vengono utilizzati anche negli header dei messaggi che il nodo pubblica. Questo nodo pubblica 3 diversi topic per ogni messaggio NMEA che riceve e sono:

Key	Value	Type
(1) ObjectId("566021e27526b054bf4...	{ 11 fields }	Object
_id	ObjectId("566021e27526b054bf4dec75")	ObjectId
_meta	{ 5 fields }	Object
topic	/velodyne_points	String
stored_type	sensor_msgs/PointCloud2	String
latch	false	Boolean
stored_class	sensor_msgs.msg._PointCloud2.PointCloud2	String
inserted_at	2015-12-03 11:05:06.304Z	Date
width	38244	Int32
is_dense	true	Boolean
point_step	32	Int32
height	1	Int32
header	{ 3 fields }	Object
stamp	2015-12-03 11:05:06.270Z	Date
frame_id	velodyne	String
seq	0	Int32
data	<binary>	Binary
fields	Array [5]	Array
0	{ 4 fields }	Object
datatype	7	Int32
count	1	Int32
name	x	String
offset	0	Int32
1	{ 4 fields }	Object
datatype	7	Int32
count	1	Int32
name	y	String
offset	4	Int32
2	{ 4 fields }	Object
datatype	7	Int32
count	1	Int32
name	z	String
offset	8	Int32
3	{ 4 fields }	Object
datatype	7	Int32
count	1	Int32
name	intensity	String
offset	16	Int32
4	{ 4 fields }	Object
datatype	4	Int32
count	1	Int32
name	ring	String
offset	20	Int32
row_step	1223808	Int32
is_bigendian	false	Boolean

Figura 4.2: Schema BSON del topic `velodyne_points` utilizzato dal database MongoDB

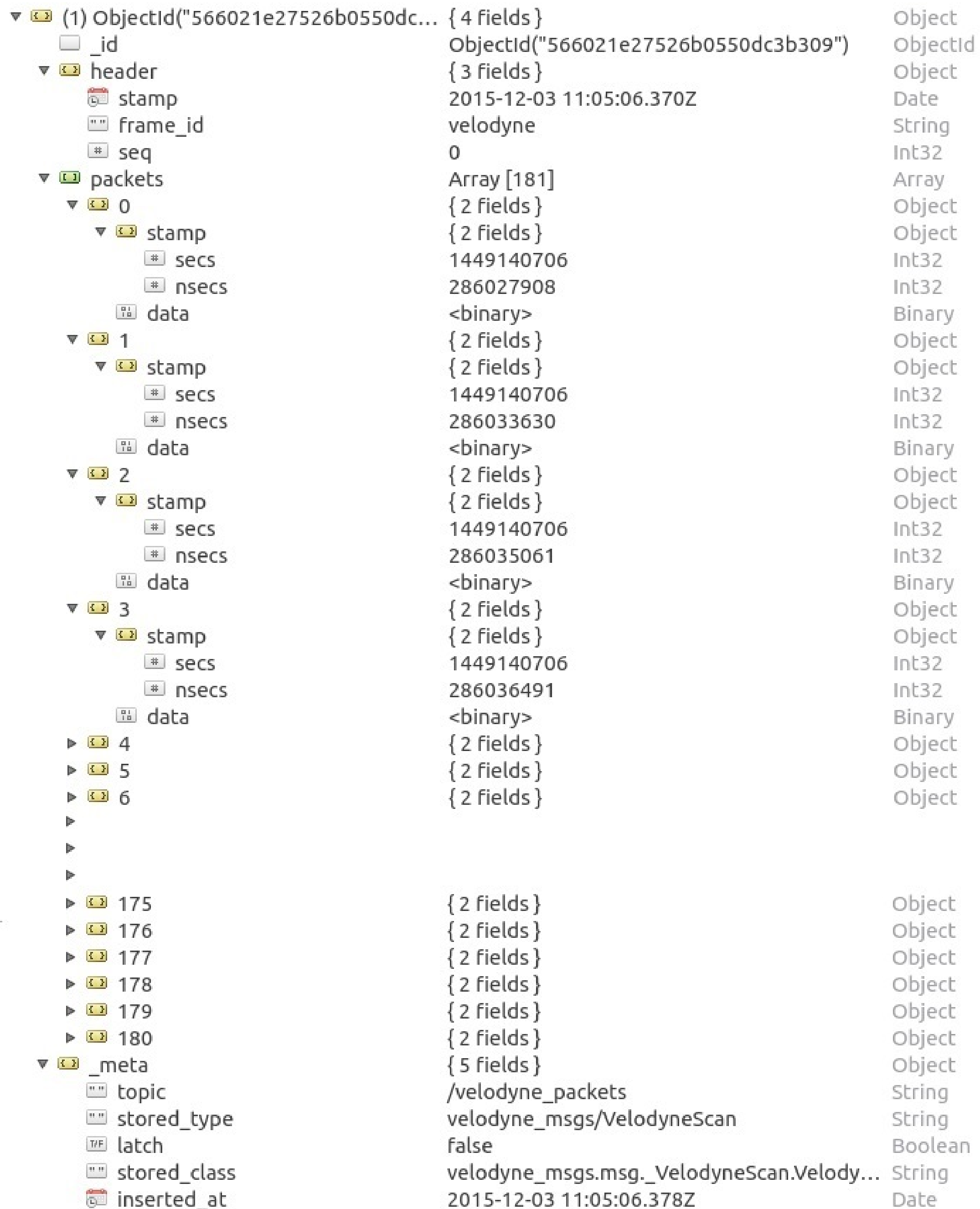


Figura 4.3: Schema BSON del topic `velodyne_packet` utilizzato dal database MongoDB

(1) ObjectId("5641fdb77526b063e5f...	{ 9 fields }	Object
_id	ObjectId("5641fdb77526b063e5f1e2c3")	ObjectId
linear_acceleration_covariance	Array [9]	Array
orientation	{ 4 fields }	Object
angular_velocity_covariance	Array [9]	Array
orientation_covariance	Array [9]	Array
header	{ 3 fields }	Object
stamp	2015-11-10 14:59:56.277Z	Date
frame_id	velodyne	String
seq	1	Int32
linear_acceleration	{ 3 fields }	Object
y	-0.742637	Double
x	9.294936	Double
z	-0.197637	Double
angular_velocity	{ 3 fields }	Object
y	-0.184085	Double
x	-0.073293	Double
z	-0.071589	Double
_meta	{ 5 fields }	Object
topic	/imu_data	String
stored_type	sensor_msgs/Imu	String
latch	false	Boolean
stored_class	sensor_msgs.msg._Imu.Imu	String
inserted_at	2015-11-10 14:22:46.898Z	Date

Figura 4.4: Schema BSON del topic `imu_data` utilizzato dal database MongoDB

- `fix`: questo topic utilizza il messaggio `sensor_msgs/NavSatFix`. Questo messaggio contiene un header ROS standard, le coordinate del gps espresse in latitudine, longitudine (entrambi in gradi) e altitudine (in metri), una matrice di covarianza e un campo che indica lo stato del GPS. Il campo dello stato contiene informazioni sulla qualità del fix del GPS e il tipo di sistema globale di navigazione utilizzato (GPS, GLONASS, COMPASS, GALILEO). Lo stato è utilizzato poiché il nodo `nmea_topic_driver` continua a ricevere messaggi NMEA anche quando il fix del GPS non è avvenuto e tramite lo stato presente nel messaggio pubblicato è possibile capire se i dati presenti nel topic sono validi.

- `vel`: questo topic utilizza il messaggio `geometry_msgs/TwistStamped`. Questo messaggio contiene un header ROS standard e un campo `geometry_msgs/Twist`. Il `geometry_msgs/Twist` è formato da un vettore a 3 elementi contenente la velocità lineare e un vettore a 3 elementi contenente la velocità angolare. La velocità viene pubblicata solo quando il GPS fornisce informazioni valide sulla velocità poiché il nodo non calcola la velocità basandosi solamente sulla posizione.

- `time_reference`: questo topic utilizza il messaggio `sensor_msgs/TimeReference`. Questo messaggio contiene un header ROS standard e un riferi-

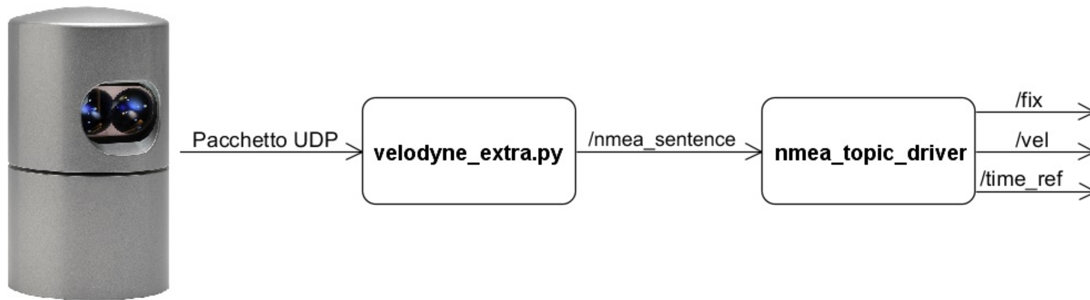


Figura 4.5: Schema dei nodi ROS utilizzati per estrarre i dati dal GPS della Velodyne

mento temporale che nel caso del nodo `nmea_topic_driver` è il riferimento temporale presente nella stringa NMEA.

Il nodo `nmea_topic_driver` ha un parametro booleano che si chiama *useRMC* che indica se lo standard utilizzato per il messaggio NMEA è GGA o RMC. Se settato a vero i dati sono generati da un mesaggio NMEA RMC altrimenti vuol dire che i dati sono stati generati a partire da un messaggio NMEA GGA. Utilizzando un messaggio NMEA RMC vengono fornite le informazioni sulla velocità.

Il topic `nmea_sentence`, al quale il nodo `nmea_topic_driver` si sottoscrive, viene pubblicato dal nodo descritto nella sezione precedente chiamato `velodyne_extra.py` (Figura 4.5).

I dati provenienti dai topic `fix`, `vel` e `time_reference` sono salvati, all'interno del database MongoDB, secondo la struttura in Figura 4.6, Figura 4.7 e in Figura 4.8.

4.3.3 IMU

All'interno di ROS è già presente un pacchetto che contiene il driver per la Xsens MTi, ma presenta un problema: il tipo di messaggio utilizzato per il campo magnetico terrestre viene pubblicato come `geometry_msgs/Vector3Stamped` anziché come `sensor_msgs/MagneticField`. Quindi è stato sviluppato un altro pacchetto chiamato `xsens_driver_airlab`.
Il nodo contenuto nel pacchetto chiamato `xsens` è stato sviluppato in C++ utilizzando la libreria fornita dalla Xsens che permette di gestire la connessione con il dispositivo e l'acquisizione delle misure. Questo nodo pubblica un topic, chiamato `/xsens/imu`, contenente i dati della IMU tramite il messaggio `sensor_msgs/Imu` e un topic il cui nome è `/xsens/mag`, contenente i dati del campo magnetico terrestre utilizzando il messaggio `sensor_msgs/MagneticField`.

Il messaggio `sensor_msgs/Imu` è formato da un header, un quaternione per l'orientamento e la sua matrice di covarianza, un vettore di 3 elementi per l'ac-

Key	Value	Type
(1) ObjectId("5659a9d27526b010e30...	{ 9 fields }	Object
_id	ObjectId("5659a9d27526b010e300599c")	ObjectId
status	{ 2 fields }	Object
status	0	Int32
service	1	Int32
_meta	{ 5 fields }	Object
topic	/fix	String
stored_type	sensor_msgs/NavSatFix	String
latch	false	Boolean
stored_class	sensor_msgs.msg._NavSatFix.NavSatFix	String
inserted_at	2015-11-28 13:19:14.512Z	Date
altitude	nan	Double
longitude	9.229208	Double
position_covariance	Array [9]	Array
0	0.000000	Double
1	0.000000	Double
2	0.000000	Double
3	0.000000	Double
4	0.000000	Double
5	0.000000	Double
6	0.000000	Double
7	0.000000	Double
8	0.000000	Double
header	{ 3 fields }	Object
stamp	2015-11-28 13:19:11.036Z	Date
frame_id	velodyne	String
seq	1	Int32
latitude	45.478192	Double
position_covariance_type	0	Int32

Figura 4.6: Schema BSON del topic `fix` utilizzato dal database MongoDB

Key	Value	Type
(1) ObjectId("566021e27526b055c59...	{ 4 fields }	Object
_id	ObjectId("566021e27526b055c5954d73")	ObjectId
twist	{ 2 fields }	Object
linear	{ 3 fields }	Object
y	-0.000000	Double
x	0.000000	Double
z	0.000000	Double
angular	{ 3 fields }	Object
y	0.000000	Double
x	0.000000	Double
z	0.000000	Double
header	{ 3 fields }	Object
stamp	2015-12-03 11:04:59.940Z	Date
frame_id	velodyne	String
seq	1	Int32
_meta	{ 5 fields }	Object
topic	/vel	String
stored_type	geometry_msgs/TwistStamped	String
latch	false	Boolean
stored_class	geometry_msgs.msg._TwistStamped.Twist...	String
inserted_at	2015-12-03 11:05:06.594Z	Date

Figura 4.7: Schema BSON del topic `vel` utilizzato dal database MongoDB

(1) ObjectId("5641fdb77526b063de9...	{ 5 fields }	Object
_id	ObjectId("5641fdb77526b063de9003be")	ObjectId
_meta	{ 5 fields }	Object
topic	/time_reference	String
stored_type	sensor_msgs/TimeReference	String
latch	false	Boolean
stored_class	sensor_msgs.msg._TimeReference.TimeRe...	String
inserted_at	2015-11-10 14:22:46.907Z	Date
header	{ 3 fields }	Object
stamp	2015-11-10 14:59:56.281Z	Date
frame_id	velodyne	String
seq	159	Int32
source	velodyne	String
time_ref	{ 2 fields }	Object
secs	1447167596	Int32
nsecs	281342029	Int32

Figura 4.8: Schema BSON del topic `time_reference` utilizzato dal database MongoDB

celerazione lineare e la sua matrice di covarianza, un vettore a 3 elementi per la velocità angolare e la sua matrice di covarianza. L'accelerazione viene espressa in m/s^2 e la velocità angolare in rad/s. Il messaggio `sensor_msgs/MagneticField` è formato da un header,un vettore di 3 elementi che contiene i valori del campo magnetico terrestre espressi in mG e la sua matrice di covarianza.

Inoltre il nodo permette di settare la frequenza alla quale acquisire i dati dalla Xsens e nel nostro caso è stata impostata a 100 Hz. Per poter utilizzare la Xsens è stato necessario creare una regola udev in linux così da garantire i permessi di accesso in lettura e scrittura al dispositivo. Il file, chiamato *rules.d*, contenente la regola creata è stato posto nella cartella /etc/udev/.La regola creata è così definita:

```
SUBSYSTEM=="usb", ATTRidVendor=="0403", ATTRidProduct=="d38b",
MODE:="0666"
```

Come è possibile vedere nella regola, per identificare la Xsens sono stati usati due ID univoci che sono il codice del venditore ("0403") e il codice del prodotto ("d38b"). Con questa coppia di ID è possibile identificare univocamente la Xsens e assegnargli i permessi di lettura e scrittura.

I dati provenienti dai topic `/xsens/imu` e `/xsens/mag` sono salvati, all'interno del database MongoDB, secondo la struttura in Figura 4.9 e in Figura 4.10.

4.3.4 Prosilica GC1020

ROS fornisce diversi nodi per per poter lavorare con le telecamere: driver, strumenti per la calibrazione e strumenti per poter visualizzare le immagini in tempo reale acquisite dalle telecamere. Tutti questi nodi sono compatibili con la Prosilica GC1020.

(1) ObjectId("566021e37526b05676c...	{ 9 fields }	Object
_id	ObjectId("566021e37526b05676c71e3e")	ObjectId
linear_acceleration_covariance	Array [9]	Array
orientation	{ 4 fields }	Object
y	0.055454	Double
x	0.123395	Double
z	-0.723456	Double
w	0.676985	Double
angular_velocity_covariance	Array [9]	Array
orientation_covariance	Array [9]	Array
header	{ 3 fields }	Object
stamp	2015-12-03 11:05:06.871Z	Date
frame_id	/imu	String
seq	215	Int32
linear_acceleration	{ 3 fields }	Object
y	0.492080	Double
x	-2.293511	Double
z	9.488605	Double
angular_velocity	{ 3 fields }	Object
y	0.013616	Double
x	0.023484	Double
z	-0.012813	Double
_meta	{ 5 fields }	Object
topic	/xsens/imu	String
stored_type	sensor_msgs/Imu	String
latch	false	Boolean
stored_class	sensor_msgs.msg._Imu.Imu	String
inserted_at	2015-12-03 11:05:06.881Z	Date

Figura 4.9: Schema BSON del topic `/xsens/imu` utilizzato dal database MongoDB

(1) ObjectId("5641fdb77526b064ed2...	{ 5 fields }	Object
_id	ObjectId("5641fdb77526b064ed2e3d17")	ObjectId
magnetic_field_covariance	Array [9]	Array
0	1.000000	Double
1	0.000000	Double
2	0.000000	Double
3	0.000000	Double
4	1.000000	Double
5	0.000000	Double
6	0.000000	Double
7	0.000000	Double
8	1.000000	Double
header	{ 3 fields }	Object
stamp	2015-11-10 14:22:47.086Z	Date
frame_id	/imu	String
seq	226	Int32
_meta	{ 5 fields }	Object
topic	/xsens/mag	String
stored_type	sensor_msgs/MagneticField	String
latch	false	Boolean
stored_class	sensor_msgs.msg._MagneticField.Magneti...	String
inserted_at	2015-11-10 14:22:47.087Z	Date
magnetic_field	{ 3 fields }	Object
y	0.077379	Double
x	0.057733	Double
z	-0.604750	Double

Figura 4.10: Schema BSON del topic `/xsens/mag` utilizzato dal database MongoDB

Il pacchetto che contiene il driver ROS per la Prosilica si chiama prosilica_ camera. Il nodo che fa da driver si chiama prosilica_node e permette tramite la *Prosilica GigE SDK* di modificare le impostazioni della camera tramite un file di configurazione e di riceve le immagini acquisite dalla camera. Questo nodo pubblica l'immagine raw ricevuta tramite un topic che utilizza un messaggio del tipo sensor_msgs/Image, contenente oltre che all'immagine non compressa un header. Inoltre per ogni immagine che il nodo riceve viene pubblicato un altro topic che contiene le informazioni sulla calibrazione della camera tramite un messaggio del tipo sensor_msgs/CameraInfo. Quest'ultimo messaggio contiene: un header, la risoluzione della camera, il modello di distorsione utilizzato, una matrice intrinseca 3x3, e altre due matrici (matrice di rettifica e matrice di proiezione) utilizzate solo dalle camere stereo. Questo nodo è stato modificato per permettere di andare ad abilitare il protocollo PTP nelle impostazioni della camera.

Per identificare la Prosilica alla quale connettersi il nodo prosilica_node riceve in ingresso 2 parametri: l'indirizzo IP della camera e il GUID che rappresenta un ID univoco che identifica la camera. Se viene settato solo il GUID il nodo si connetterà alla camera che presenta quel ID univoco altrimenti se entrambi i parametri sono settati il nodo si connetterà alla camera che verifica entrambi i parametri. Se nessuno dei due parametri è settato il nodo si connetterà alla prima camera disponibile. Nel nostro caso entrambe le Prosilica hanno un indirizzo IP statico che viene utilizzato dal nodo.

Per la calibrazione della camera viene utilizzato un pacchetto che si chiama camera_calibration il quale sfrutta la libreria *OpenCV*. Il nodo utilizzato per la calibrazione si chiama cameracalibration.py. Per effettuare la calibrazione occorre possedere una grande scacchiera di dimensioni note e che sia rettangolare in modo che, durante la calibrazione, sia possibile riconoscerne l'orientamento. Inoltre occorre avere di fronte alla camera un'area di circa 5 m x 5 m ben illuminata e libera da ostacoli così da poter muovere la scacchiera liberamente. Successivamente si può avviare il nodo cameracalibration.py al quale bisogna passare come parametro le caratteristiche della scacchiera (numero di quadrati sulle righe e sulle colonne e dimensione in millimetri del quadrato) e il nome del topic sulla quale la camera pubblica le immagini. Per la calibrazione di una camera con una schacchiera 8x6 con un quadrato che misura 108 mm il comando per eseguire la calibrazione è:

```
rosrun camera_calibration cameracalibrator.py -- size 8x6 --square
        0.108 image:=/camera/image_raw camera:=/camera
```

Una volta avviato il nodo si aprirà una finestra (Figura 4.11) che presenterà le immagini in tempo reale catturate dalla camera evidenziando la scacchiera. Inoltre la finestra presenta 3 barre che stanno ad indicare la qualità della calibrazione sull'asse delle X, delle Y e rispetto a profondità e inclinazione.

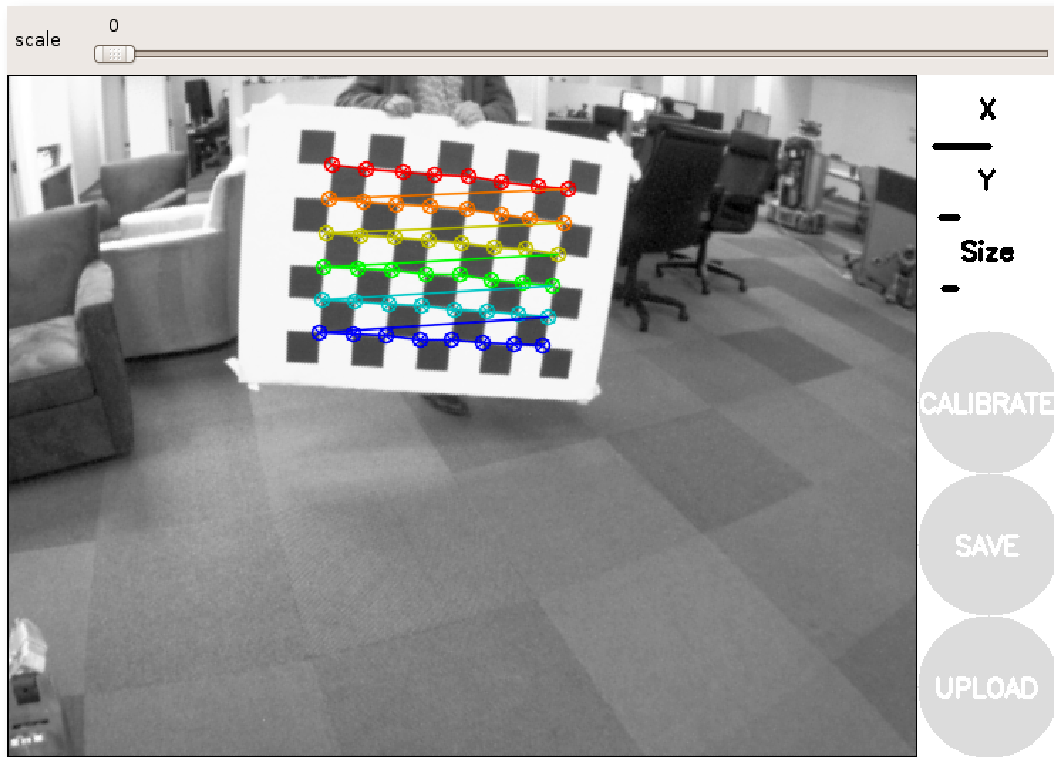


Figura 4.11: Finestra di calibrazione della camera

Per avere una buona calibrazione è necessario muovere la scacchiera per tutto il campo di visione della camera in modo tale che:

- La scacchiera sia posta a sinistra, a destra, in altro e in basso rispetto al campo visivo della camera
- la scacchiera deve essere mossa per l'intera profondità del campo visivo della camera
- La scacchiera deve essere inclinata a destra,a sinistra, in alto e in basso

Una volta che il nodo dispone di abbastanza dati per la calibrazione il bottone "CALIBRATE" potrà essere premuto. Dopo che la calibrazione è terminata sarà possibile inviare i parametri della calibrazione alla camera tramite la pressione del bottone "COMMIT".

Utilizzando per i nostri scopi due camere Prosilica è stato necessario istanziare due nodi `prosilica_node`. Nel nostro caso abbiamo il nodo `prosilica1_driver` che pubblica i topic `prosilica1/image_raw` e `prosilica1/camera_info` e il nodo `prosilica2_driver` che pubblica i topic `prosilica2/image_raw` e `prosilica2/camera_info`.

(1) ObjectId("5641fdbcd814673564c...	{ 9 fields }	Object
_id	ObjectId("5641fdbcd814673564cb3564")	ObjectId
header	{ 3 fields }	Object
seq	0	Int32
stamp	2015-11-10 14:22:52.015Z	Date
frame_id	high_def_optical_frame	String
encoding	bayer_rggb8	String
height	768	Int32
width	1024	Int32
is_bigendian	0	Int32
step	1024	Int32
data	<binary>	Binary
_meta	{ 5 fields }	Object
inserted_at	2015-11-10 14:22:52.236Z	Date
stored_type	sensor_msgs/Image	String
topic	/prosilica1/image_raw	String
latch	false	Boolean
stored_class	sensor_msgs.msg._Image.Image	String

Figura 4.12: Schema BSON dei topic `prosilica1/image_raw` e `prosilica2/image_raw` utilizzato dal database MongoDB

I topic `prosilica1/image_raw` e `prosilica2/image_raw` utilizzano entrambi un messaggio del tipo `sensor_msgs/Image` per cui, all'interno del database MongoDB, utilizzano la stessa struttura rappresentata in Figura 4.12. I topic `prosilica1/camera_info` e `prosilica2/camera_info` utilizzano anch'essi un messaggio dello stesso tipo, per cui presentano nel database MongoDb la stessa struttura rappresentata in Figura 4.13.

4.4 Sensori per il guidatore

In questa sezione vengono descritti i pacchetti ROS utilizzati per poter acquisire e configurare i diversi sensori utilizzati per acquisire dati sul guidatore indicandone le principali caratteristiche che sono state utilizzate.

4.4.1 Procomp Infiniti

Come descritto in precedenza, il sensore Procomp Infiniti non presenta le API per Linux ma soltanto per sistemi operativi Windows e pertanto non è stato possibile creare un driver ROS che permettesse la lettura diretta dei dati dal dispositivo. Per poter acquisire i dati è stato quindi necessario creare un sistema client-server, tramite *socket TCP*, tra la macchina virtuale Windows del sistema I.DRIVE e la macchina fisica Linux che ne è host.

Il server è stato creato sulla macchina virtuale utilizzando il linguaggio C++ e le *TTLAPI*, le API fornite dal venditore della Procomp Infiniti. Tramite le TTLAPI il server creato attende la connessione di una scheda Procomp Infiniti e se ne rileva la presenza crea un socket TCP e attende sulla porta 27015 la

Key	Value	Type
(1) ObjectId("5641fdbc7526b0681e6...	{ 13 fields }	Object
_id	ObjectId("5641fdbc7526b0681e68fd7c")	ObjectId
roi	{ 5 fields }	Object
width	1024	Int32
do_rectify	false	Boolean
y_offset	0	Int32
x_offset	0	Int32
height	768	Int32
D	Array [5]	Array
0	0.003850	Double
1	-0.003960	Double
2	-0.002720	Double
3	0.000150	Double
4	0.000000	Double
P	Array [12]	Array
0	589.695310	Double
1	0.000000	Double
2	1121.928670	Double
3	0.000000	Double
4	0.000000	Double
5	640.319270	Double
6	857.732140	Double
7	0.000000	Double
8	0.000000	Double
9	0.000000	Double
10	1.000000	Double
11	0.000000	Double
width	1024	Int32
_meta	{ 5 fields }	Object
topic	/prosilica1/camera_info	String
stored_type	sensor_msgs/CameraInfo	String
latch	false	Boolean
stored_class	sensor_msgs.msg._CameraInfo.CameraInfo	String
inserted_at	2015-11-10 14:22:51.979Z	Date
K	Array [9]	Array
0	272.490410	Double
1	0.000000	Double
2	498.030920	Double
3	0.000000	Double
4	272.424060	Double
5	366.778280	Double
6	0.000000	Double
7	0.000000	Double
8	1.000000	Double
height	768	Int32
header	{ 3 fields }	Object
stamp	2015-12-03 11:05:13.181Z	Date
frame_id	high_def_optical_frame	String
seq	0	Int32
R	Array [9]	Array
0	1.000000	Double
1	0.000000	Double
2	0.000000	Double
3	0.000000	Double
4	1.000000	Double
5	0.000000	Double
6	0.000000	Double
7	0.000000	Double
8	1.000000	Double
binning_y	1	Int32
binning_x	1	Int32
distortion_model	plumb_bob	String

Figura 4.13: Schema BSON dei topic `prosilica1/camera_info` e `prosilica2/camera_info` utilizzato dal database MongoDB

connessione del client. Una volta che il client richiede la connessione il server inizia ad acquisire i dati dalla Procomp Infiniti e li invia al client.

Le TTLAPI permettono, per ogni canale della Procomp Infiniti utilizzato, di acquisire i dati disponibili fino al momento della richiesta che non sono ancora stati letti. I dati acquisiti non presentano nessun dato temporale per cui viene preso come timestamp il tempo nel quale è stata fatta l'acquisizione. La richiesta di dati è temporizzata tramite un timer e viene fatta ogni 125 ms ovvero ad una frequenza di 8 Hz. Questa frequenza di acquisizione consente di avere circa 32 campioni per ogni singola acquisizione fatta su un canale che funziona a 256 campioni/sec e circa 256 campioni per un canale che funziona a 2048 campioni/sec. Una volta acquisiti, i dati vengono inviati al client connesso tramite socket TCP. Per ogni acquisizione fatta vengono inviati oltre ai dati acquisiti anche informazioni aggiuntive quali: il tempo di acquisizione in secondi e nanosecondi, un id identificativo del sensore dal quale provengono i dati, la frequenza stimata di acquisizione e il numero di campioni presenti.

Una volta che i dati acquisiti sono stati inviati dal server essi vengono letti da un client che è stato implementato in un nodo ROS. Il nodo ROS che è stato sviluppato in C++ si chiama `procomp_client` ed è contenuto all'interno del pacchetto `idrive_data_logger`. Questo nodo, tramite socket TCP, esegue un client che tenta la connessione al server creato ed è identificato dall'indirizzo IP che è stato assegnato staticamente alla macchina virtuale ovvero 192.168.1.10. Una volta che la connessione al server è stata stabilita il nodo va a leggere i dati inviati dal server e li converte in un messaggio creato appositamente e chiamato `idrive_data_logger/Bio_sensor`.

Il messaggio `idrive_data_logger/Bio_sensor` è composto da: un header standard di ROS, un campo che indica il tipo di sensore al quale appartengono i dati presenti nel messaggio, l'unità di misura dei dati, la frequenza a cui sono stati acquisiti i dati, il numero di dati presenti nel messaggio ed un array dinamico che contiene i dati acquisiti. Questo messaggio viene pubblicato su un topic il quale nome è `procomp_sensor`. La frequenza con il quale questo nodo legge i dati dal socket è di 100Hz.

I dati provenienti dal topic `procomp_sensor` sono salvati, all'interno del database MongoDB, secondo la struttura in Figura 4.14.

4.4.2 Empatica E4

Anche il sensore Empatica E4 non presenta le API per Linux ma soltanto per sistemi operativi Android e iOS e pertanto non è stato possibile creare un driver ROS che permettesse la lettura diretta dei dati dal dispositivo. L'azienda produttrice fornisce altresì un server TCP per Windows che permette di ricevere i dati in tempo reale da più dispositivi Empatica E4. Per poter acquisire i dati dal sensore è stato quindi necessario creare un sistema client-server tra la macchina virtuale Windows e la macchina fisica Linux analogo a quanto fatto per il sistema

Key	Value	Type
(1) ObjectId("566021e27526b053f58...	{ 8 fields }	Object
_id	ObjectId("5641fdf57526b063a2ddd737")	ObjectId
_meta	{ 5 fields }	Object
topic	/procomp_sensor	String
stored_type	idrive_data_logger/Bio_sensor	String
latch	false	Boolean
stored_class	idrive_data_logger.msg._Bio_sensor.Bio_se...	String
inserted_at	2015-11-10 14:23:49.637Z	Date
sample_number	25	Int32
MU	degree Celsius	String
header	{ 3 fields }	Object
stamp	2015-11-10 14:23:54.291Z	Date
frame_id	/bio_sensor	String
seq	0	Int32
sensor_type	Temperature	String
freq	256	Int32
data	Array [25]	Array
0	25.234793	Double
1	25.234793	Double
2	25.234793	Double
3	25.234793	Double
4	25.230373	Double
5	25.234793	Double
6	25.230373	Double
7	25.234793	Double
8	25.230373	Double
9	25.234793	Double
10	25.234793	Double
11	25.234793	Double
12	25.234793	Double
13	25.234793	Double
14	25.239214	Double
15	25.230373	Double
16	25.230373	Double
17	25.234793	Double
18	25.234793	Double
19	25.234793	Double
20	25.234793	Double
21	25.234793	Double
22	25.230373	Double
23	25.230373	Double
24	25.230373	Double

Figura 4.14: Schema BSON del topic `procomp_sensor` utilizzato dal database MongoDB

Procomp. Il server TCP fornito permette di connettere il sensore Empatica E4 e creare una conessione TCP alla quale un client si può connettere tramite *socket TCP*. Dopo che un client si è connesso, il server si mette in attesa di un comando da parte del client.

La struttura dei messaggi che possono essere inviati dal client e dei messaggi di risposta del server sono ben definiti. Ogni messaggio infatti è una stringa ASCII che termina con un carattere di a capo ("\n") e le richieste inviate dal client devono avere la seguente struttura:

`<COMANDO> <LISTA ARGOMENTI>.`

Le risposte ai comandi che il server riceve mantengono la stessa struttura della richiesta, ma sono precedute dal carattere "R":

`R <COMANDO> <LISTA ARGOMENTI>.`

A seconda del valore che assume il campo `<COMANDO>` si possono effettuare diverse operazioni. Di seguito l'elenco dei comandi disponibili:

- *server_status*: permette al client di conoscere lo stato del server (OK o ERR) e nel caso di errore di avere la causa dell'errore
- *device_list*: permette al client di sapere il numero di dispositivi Empatica E4 connessi al server con relative informazioni quali ID del dispositivo connesso e nome del dispositivo connesso
- *device_connect*: permette al client di connettersi ad uno dei dispositivi Empatica E4 disponibili andando ad indicarne l'ID univoco
- *device_disconnect*: permette al client di disconnettersi da uno dei dispositivi Empatica E4 al quale si era precedentemente connesso andando ad indicarne l'ID univoco
- *device_subscribe*: permette al client di attivare o disattivare la ricezione del flusso dati in tempo reale del sensore per cui si è fatto richiesta

Infine la struttura dei messaggi contenenti i dati provenienti dai sensori che il server invia al client ha la seguente struttura:

`<SENSORE> <TIMESTAMP> <DATO>`

I valori che può assumere il campo `<SENSORE>` sono ben definiti e sono:

- *E4_Acc*: identifica i dati sull' accelerazione
- *E4_Bvp*: identifica i dati sulla variazione del volume sanguigno (BVP)
- *E4_Gsr*: identifica i dati sulla risposta galvanica della pelle

- *E4_ Temp*: identifica i dati sulla temperatura della pelle
- *E4_ Ibi*: identifica i dati sull'intervallo di tempo interbattito
- *E4_ Hr*: identifica i dati sul numero di battiti al minuto
- *E4_ Battery*: identifica i dati sul livello della batteria

Il campo `<TIMESTAMP>`, fissato dal server, definisce il tempo in cui è stato acquisito il dato presente nel messaggio ed è indicato in secondi passati dalla data 1 Gennaio 1970, GMT.

Per poter interagire con il server di Empatica è stato sviluppato in C++ un nodo ROS chiamato `empatica_client` che è contenuto all'interno del pacchetto `idrive_data_logger`. Questo nodo, tramite socket TCP, fa da client e tenta la connessione al server di Empatica che è identificato dall'indirizzo IP che è stato assegnato staticamente alla macchina virtuale ovvero 192.168.1.10. Una volta che la connessione TCP al server è stata stabilita il nodo non fa altro che inviare diversi comandi andandone a interpretare le risposte.

Il nodo invia inizialmente i comandi per conoscere lo stato del server dopodiché cerca di connettersi al primo dispositivo Empatica disponibile sul server. Una volta connesso ad un dispositivo il nodo invia i comandi per potersi sottoscrivere a tutti i dati dei sensori disponibili. Ogni volta che riceve i dati in tempo reale provenienti dai sensori, il nodo li pubblica sul topic `empatica_sensor`. Questo topic utilizza un messaggio creato appositamente e chiamato `idrive_data_logger/Bio_sensor`. Il messaggio `idrive_data_logger/Bio_sensor` è composto da: un header standard di ROS, un campo che indica il tipo di sensore al quale appartengono i dati presenti nel messaggio, l'unità di misura dei dati, la frequenza a cui sono stati acquisiti i dati, il numero di dati presenti nel messaggio ed un array dinamico che contiene i dati acquisiti. La frequenza con il quale questo nodo legge i dati dal socket è di 100Hz.

I dati provenienti dal topic `empatica_sensor` sono salvati, all'interno del database MongoDB, secondo la struttura in Figura 4.15.

4.4.3 Axis P1343

All'interno di ROS è già presente un pacchetto che contiene il driver per la Axis P1343 il cui nome è `axis_camera`. Questo pacchetto contiene al suo interno un nodo python chiamato `axis.py` che si occupa di acquisire le immagini dalla camera Axis. Questo nodo pubblica un topic che contiene le immagini compresse che acquisisce utilizzando un messaggio `sensor_msgs/CompressedImage`. Esso è formato da un header, una stringa che ne definisce il formato e dai dati dell'immagine compressa. Per ogni immagine che il nodo acquisisce viene pubblicato un secondo topic che contiene le informazioni sulla calibrazione della camera tramite

Key	Value	Type
(1) ObjectId("566021e27526b05429c...	{ 8 fields }	Object
_id	ObjectId("566021e27526b05429c8b370")	ObjectId
_meta	{ 5 fields }	Object
topic	/empatica_sensor	String
stored_type	idrive_data_logger/Bio_sensor	String
latch	false	Boolean
stored_class	idrive_data_logger.msg._Bio_sensor.Bio_se...	String
inserted_at	2015-12-03 11:05:06.131Z	Date
sample_number	1	Int32
MU	relative	String
header	{ 3 fields }	Object
stamp	2015-12-03 11:05:03.095Z	Date
frame_id	/bio_sensor	String
seq	0	Int32
sensor_type	BVP	String
freq	64	Int32
data	Array [1]	Array
0	-463.305115	Double

Figura 4.15: Schema BSON del topic `empatica_sensor` utilizzato dal database MongoDB

un messaggio del tipo `sensor_msgs/CameraInfo`. Quest'ultimo messaggio contiene: un header, la risoluzione della camera, il modello di distorsione utilizzato, una matrice intrinseca 3x3, ed altre due matrici (matrice di rettifica e matrice di proiezione) utilizzate solo dalle camere stereo. Se i dati della calibrazione non sono disponibili o incompatibili con la modalità video utilizzata i dati di calibrazione non vengono forniti. Il nodo `axis.py` presenta alcuni parametri che possono essere configurati:

- *hostname*: serve per indicare l'indirizzo IP della camera e di default è 192.168.0.90
- *username*: serve per indicare l'username utilizzato per accedere alla camera e di default è "root"
- *password*: serve per indicare la password utilizzata per accedere alla camera e di default è vuota
- *width*: serve per indicare la risoluzione orizzontale e di default è 640
- *height*: serve per indicare la risoluzione verticale e di default è 480
- *frame_id*: serve per indicare un frame id da utilizzare nell'header dei topic e di default è "axis_camera"

Per i nostri scopi e per come sono configurate le Axis che utilizziamo viene fornito al driver solamente l'indirizzo IP poiché come username e password vengono utilizzati i valori di default.

(1) ObjectId("566021e2693deed670...	{ 5 fields }	Object
_id	ObjectId("566021e2693deed670b89e49")	ObjectId
header	{ 3 fields }	Object
seq	1	Int32
stamp	2015-12-03 11:05:06.616Z	Date
frame_id	/axis_camera	String
format	jpeg	String
data	<binary>	Binary
_meta	{ 5 fields }	Object
inserted_at	2015-12-03 11:05:06.729Z	Date
stored_type	sensor_msgs/CompressedImage	String
topic	/axis1/image_raw/compressed	String
latch	false	Boolean
stored_class	sensor_msgs.msg._CompressedImage.Com...	String

Figura 4.16: Schema BSON dei topic `axis1/image_raw/compressed` e `axis2/image_raw/compressed` utilizzato dal database MongoDB

Utilizzando per la nostra applicazione due camere Axis è stato necessario istanziare due nodi `axis.py`. Abbiamo così il nodo `axis1` che pubblica i topic `axis1/image_raw/compressed` e `axis1/camera_info` e il nodo `axis2` che pubblica i topic `axis2/image_raw/compressed` e `axis2/camera_info`.

I topic `axis1/image_raw/compressed` e `axis2/image_raw/compressed` utilizzano entrambi un messaggio del tipo `sensor_msgs/CompressedImage` per cui, all'interno del database MongoDB, utilizzano la stessa struttura rappresentata in Figura 4.16.

4.5 Visualizzazione dei dati

Una volta che l'intero sistema è stato avviato, uno dei modi per capire se tutti i sensori stanno funzionando e acquisendo correttamente è quello di visualizzare i dati che vengono acquisiti. Inoltre, la visualizzazione dei dati permette di capire rapidamente se quello che viene acquisito dai sensori è quello che ci si aspetta o sono presenti problemi quali, ad esempio, il disallineamento dei sensori o immagini delle camere sfuocate. Per fare questo, all'interno del nostro sistema basato su ROS, è necessario andare a visualizzare il contenuto dei vari topic. ROS mette già a disposizione degli strumenti più o meno avanzati a tale scopo. Di seguito verranno descritti i sistemi e i nodi sviluppati per la visualizzazione dei dati provenienti dai sensori.

4.5.1 RVIZ

Uno degli strumenti che ROS mette a disposizione e che permette la visualizzazione di alcuni dei tipi di messaggi standard di ROS è *RVIZ*. Questo strumen-

to permette inoltre, utilizzando le informazioni provenienti dalle libreria *tf*, di mostrare i dati dei vari sensori in un sistema di riferimento a scelta.

Nel nostro sistema, RVIZ è utilizzato per visualizzare i dati provenienti dalle due camere Axis P1343, dalle due Prosilica GC1020 e dalla Velodyne. Nella nostra configurazione (Figura 4.17, Figura 4.18) vengono utilizzate quattro finestre del tipo "*Image*" e una finestra più grande del tipo "*PointCloud2*". La finestra del tipo "Image" cosente di visualizzare le immagini che provengono da un messaggio del tipo `sensor_msgs/Image`. Ognuna di queste quattro finestre è collegata al topic corrispondente di ognuna delle camere presenti che nel nostro sistema sono:

- `axis1/image_raw/compressed`
- `axis2/image_raw/compressed`
- `prosilica1/image_raw`
- `prosilica2/image_raw`

La finestra del tipo "PointCloud2" consente di visualizzare una nuvola di punti proveniente da un messaggio del tipo `sensor_msgs/PointCloud2`. Per questa finestra è possibile andare a selezionare anche lo stile da utilizzare per la visualizzazione dei punti della nuvola. Infatti è possibile selezionare tra:

- *punti*: ogni singolo punto ha una dimensione fissa sullo schermo di 3 pixel per 3 pixel. Questo stile consente spesso di avere una definizione migliore da lontano ma appena ci si avvicina la densità decresce
- *cubi* : ogni singolo punto è rappresentato come un cubo nel mondo reale

Per definire il colore di ciascun punto che viene visualizzato viene utilizzato il valore di intensità (i) che è fornito con ciascuno di essi. Per definire il colore vengono utilizzati 4 parametri che sono definibili in RVIZ e sono:

- *Intentità minima* (min_i)
- *Intentità massima* (max_i)
- *Colore minima* (min_c)
- *Colore massimo* (max_c)

Per calcolare il colore da assegnare al punto l'intensità viene normalizzata rispetto ai valori di intensità minima e massima secondo la semplice formula:

$$norm_i = (i - min_i)/(max_i - min_i)$$

Dopodiché viene calcolato il colore da utilizzare utilizzando i valori del colore massimo e minimo:

$$c_finale = (norm_i * max_c) + ((1 - norm_i) * min_c)$$

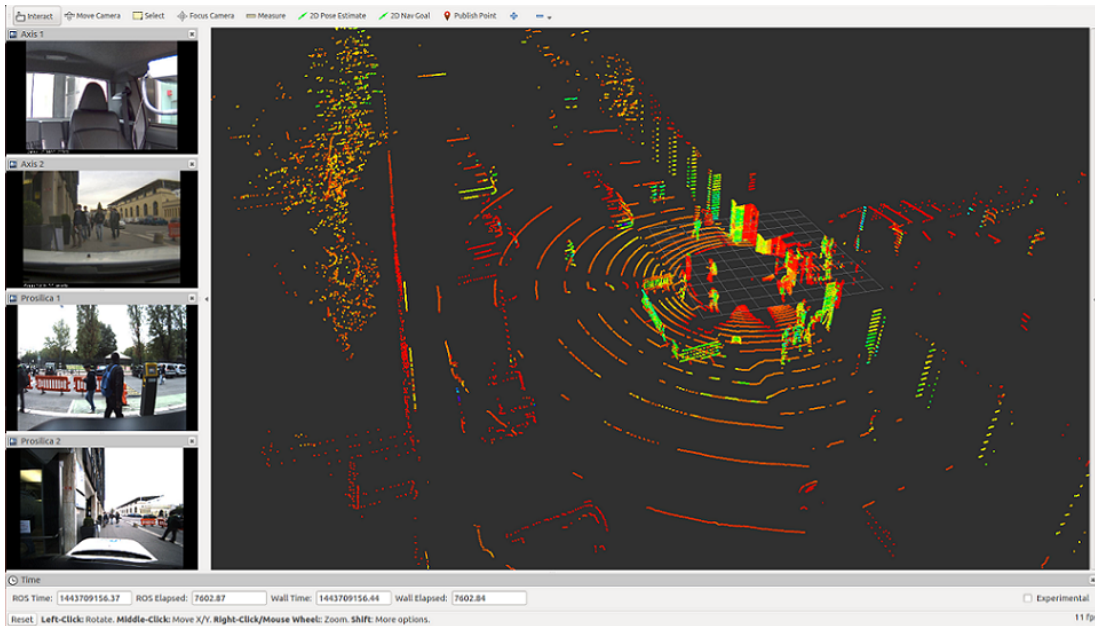


Figura 4.17: Finestra RVIZ utilizzata nel nostro sistema: a sinistra le 4 finestre del tipo "Image" e a destra la finestra del tipo "PointCloud2"

Figura 4.18: Configurazione del veicolo I.DRIVE che ha catturato le immagini visualizzate nella precedente finestra di RVIZ

4.5.2 ROS e Pylab

Per i dati provenienti dalla Procomp Infiniti e da Empatica E4 non è stato possibile utilizzare RVIZ poiché i rispettivi nodi ROS pubblicano un messaggio non standard del tipo `idrive_data_logger/Bio_sensor` ed inoltre era necessario visualizzare questi dati su dei grafici per capirne l'andamento. È stato così necessario sviluppare 2 diversi nodi, uno per ciascun sensore. Entrambi questi nodi ROS sono stati sviluppati in python visto che fornisce delle librerie con cui è facile gestire gli array ed inoltre presenta una libreria che si chiama *Pylab* che permette la creazione di grafici in maniera molto semplice.

Il nodo che si occupa di visualizzare i dati provenienti dalla Procomp Infiniti si chiama `plotter_procomp.py`. Questo nodo non fa altro che sottoscriversi al topic `procomp_sensor` che contiene i messaggi del tipo `idrive_data_logger/Bio_sensor` e andare a visualizzare in una finestra (Figura 4.19) i dati che sono contenuti nel messaggio utilizzando la libreria Pylab. Per poter visualizzare i dati tramite Pylab bisogna indicare per ogni grafico due array: uno che contiene i dati da visualizzare sull'asse delle X e uno da visualizzare sull'asse delle Y. Nel nostro caso sull'asse delle Y vengono posti i valori dei vari campioni e sull'asse delle X vengono posti i timestamp dei vari campioni. Come descritto precedentemente la Procomp Infiniti non fornisce un timestamp per ogni valore fornito, ma fornisce un timestamp per un insieme di valori. È stato quindi necessario calcolare un timestamp il più possibile reale per ognuno dei campioni. Per fare questo è stato utilizzato il timestamp presente nel topic ed è stato associato all'ultimo valore presente in ordine cronologico dopodiché, sapendo che l'intervallo di tempo che passa tra un'acquisizione e un'altra per la Procomp è di circa 125 ms e sapendo il numero di campioni presenti nel topic, si è calcolato l'intervallo di tempo che approssimativamente intercorre tra 2 campioni. Una volta calcolato tale intervallo è stato possibile ricostruire in maniera approssimata il timestamp di tutti i valori presenti.

Il nodo che si occupa di visualizzare i dati provenienti dal sensore Empatica E4 si chiama `plotter_empatica`.Questo nodo si sottoscrive al topic `empatica_sensor` che contiene anch'esso messaggi del tipo `idrive_data_logger/Bio_sensor`. Il principio di funzionamento di questo nodo è del tutto analogo al nodo `plotter_procomp.py` infatti anch'esso legge i dati dal topic e li va a visualizzare in una finestra utilizzando la libreria Pylab (Figura 4.20). La sola differenza presente è che, nel caso di Empatica E4, si ha a disposizione il timestamp di ogni valore acquisito dal sensore e quindi non è necessario dover calcolare un timestamp approssimativo per i campioni.

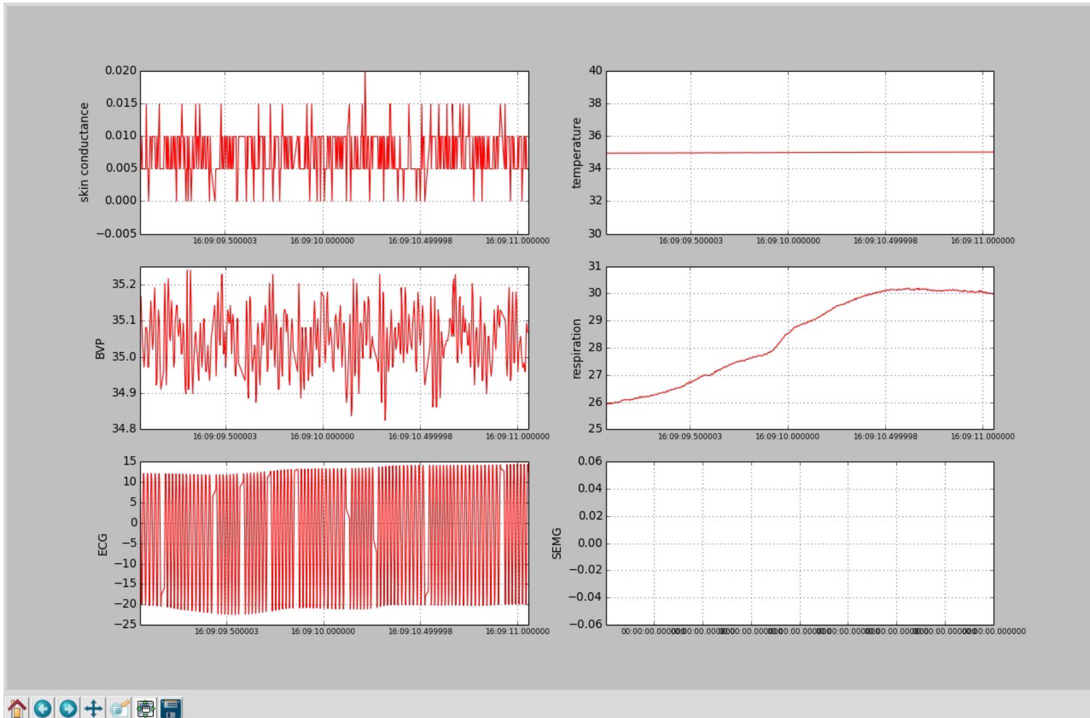


Figura 4.19: Finestra utilizzata nel nostro sistema per visualizzare i dati del sensore Procomp Infiniti

4.6 Sistema di controllo dello stato dei nodi ROS

In questa sezione viene descritto il sistema di controllo dello stato dei nodi ROS che permette di sapere in ogni momento lo stato in cui è il nodo e se è avvenuto una chiusura imprevista di uno di essi. In particolar modo viene descritta la macchina a stati e il nodo che la utilizza e infine viene descritto il pannello dal quale è possibile visualizzare lo stato dell'intero sistema.

4.6.1 Macchina a stati

Per poter gestire lo stato di un nodo è stato necessario definire una macchina a stati finiti (Figura 4.21). Ogni nodo ha quindi la propria macchina a stati che viene gestita da un nodo ROS chiamato `Heartbeat`. I possibili stati della macchina a stati sono:

- *INIT*: questo stato indica che il nodo si sta avviando e si sta inizializzando oppure che è in attesa che il sensore dal quale deve leggere i dati sia collegato o disponibile
- *STARTED*: questo stato indica che il nodo è avviato e sta funzionando correttamente leggendo i dati che provengono dal sensore

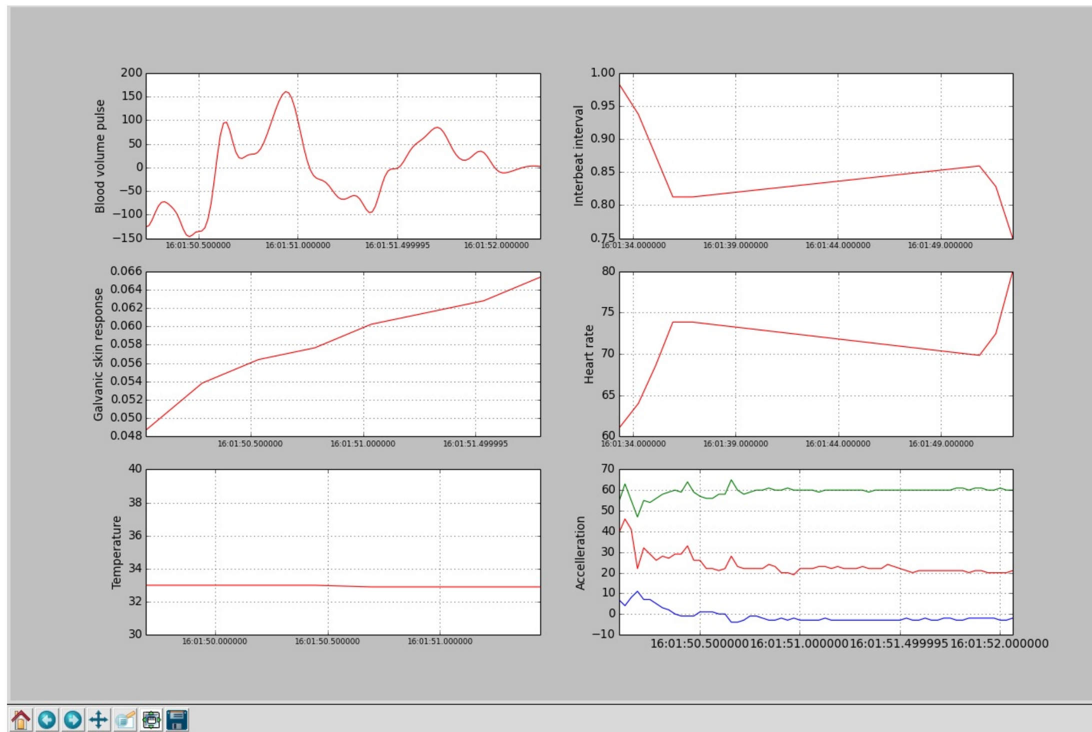


Figura 4.20: Finestra utilizzata nel nostro sistema per visualizzare i dati del sensore Empatica E4

- *STOPPED*: questo stato indica che il nodo non sta funzionando oppure che ha subito una chiusura improvvisa

Ogni nodo prima di essere avviato si trova nello stato iniziale di STOPPE. Le transizioni consentite tra gli stati sono così definite:

- Da STOPPED a INIT : è la classica transizione iniziale appena il nodo viene avviato ovvero il nodo appena avviato passa nella fase di inizializzazione.

- Da INIT a:

  – STARTED: in questa transizione il nodo è stato inizializzato e ora passa al normale funzionamento

  – STOPPED: in questa transizione il nodo durante l'inizializzazione ha incontrato un problema e si è spento

- Da STARTED a:

  – STOPPED: in questa transizione il nodo dal funzionamento normale è stato spento oppure è avvenuta una chiusura imprevista

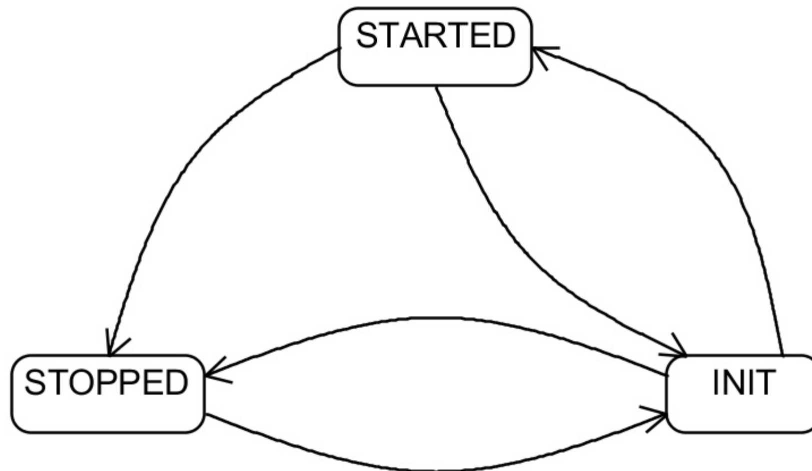


Figura 4.21: Macchina a stati finiti

  - INIT: in questa transizione il nodo passa dal funzionamento normale alla fase di inizializzazione come per esempio nel caso in cui durante il funzionamento normale il sensore dal quale il nodo stava acquisendo venga staccato

4.6.2 Heartbeat

Per gestire le macchine a stati di tutti i nodi presenti nel sistema è stato modificato un pacchetto già esistente chiamato `heartbeat`, sviluppato presso il Politecnico di Milano. Questo pacchetto inizialmente permetteva di gestire una singola macchina a stati che rappresentava lo stato dell'intero sistema ed era differente da quella presentata nella sezione precedente. Per gestire la macchina a stati di ciascun nodo questo pacchetto implementa, tramite i servizi di ROS, un sistema client-server in cui esiste un nodo centrale che fa da server e gestisce le varie macchine a stati e riceve gli aggiornamenti di stato provenienti dai vari nodi del sistema che fungono da client.

All'interno del pacchetto `heartbeat` è presente il nodo che funziona da server e si chiama `server_node`. Questo è il nodo principale che è stato modificato per poter gestire più macchine a stati. Ogni macchina a stati all'interno del nodo è identificabile per mezzo del nome univoco del nodo a cui appartiene.
IL nodo `server_node` gestisce le richieste provenienti dai client per mezzo di 3 differenti servizi ROS che sono definiti all'interno del pacchetto e sono:

- `RegisterNode.srv`: è il servizio ROS che permette al client di inviare la richiesta per registrarsi al server e avere quindi la possibilità di gestire il proprio stato. Questo servizio nella richiesta richiede che sia indicato il

nome del nodo e un *timeout* in secondi. Il server, in risposta alla richiesta, invia un valore booleano che se vero sta ad indicare che la registrazione è avvenuta con successo

- `UnregisterNode.srv`: è il servizio ROS che permette al client di inviare la richiesta per cancellare la propria registrazione al server. Questo servizio nella richiesta richiede che sia indicato solamente il *nome del nodo*. Il server in risposta alla richiesta invia un valore booleano che se vero sta a indicare che la cancellazione è avvenuta con successo

- `SetState.srv`: è il servizio ROS che permette al client di inviare una richiesta per l'aggiornamento del proprio stato. Questo servizio nella richiesta richiede che sia indicato il *nome del nodo*, lo *stato* in cui è il nodo e lo *stato* che vuole raggiungere. Il server in risposta alla richiesta invia lo stato corrente in cui è il nodo dopo la richiesta. Il campo stato utilizzato in questo servizio è un messaggio ROS definito all'interno del pacchetto `heartbeat` che si chiama `heartbeat/State`. Questo messaggio contiene il nome del nodo e un valore numerico che rappresenta lo stato in cui è la macchina a stati (STOPPED = 0, INIT = 1, STARTED = 2).

Inoltre il server pubblica su un topic chiamato `state` un messaggio del tipo `heartbeat/State` in cui viene indicato lo stato corrente di ciascuno dei nodi registrati al server.

Un nodo ROS che vuole poter gestire il proprio stato deve instanziare l'oggetto *HeartbeatClient* nel caso fosse scritto in C++ altrimenti se scritto in Pyhton deve istanziare l'oggetto *HeartbeatClientPy*. Questi due oggetti presentano le stesse funzionalità e permettono al nodo che ne istanzia uno, la gestione delle richieste al server tramite i servizi ROS sopra descritti.
Durante la creazione dell'oggetto un nodo può indicare un *timeout* in secondi che è inviato al server nella richiesta di registrazione del nodo. Questo timeout è utilizzato dal server per capire se il nodo è ancora attivo oppure ha subito una chiusura imprevista. Per fare questo, il server va calcolare se il tempo che intercorre tra due messaggi, del tipo `heartbeat/Heartbeat`, inviati dal client è maggiore del timeout indicato. Nel caso in cui il tempo passato tra i due messaggi sia maggiore del timeout, il server imposta lo stato del nodo a STOPPED. Il nodo client, nel caso indichi un timeout, deve quindi inviare continuamente al server su un topic, chiamato `heartbeat`, un messaggio del tipo `heartbeat/Heartbeat` che permette al server di capire che il nodo è ancora vivo.

Visto che nel nostro sistema è necessario gestire lo stato di ogni nodo ROS presente, si è resa necessaria la modifica del codice di ciascuno dei nodi per poter implementare al loro interno il client heartbeat. Nel codice C++ di ognuno dei nodi del nostro sistema è stato necessario inserire le istruzioni per instanziare l'oggetto HearbeatClient, indicando anche il timeout desiderato. Tramite questo oggetto, all'interno del nodo ROS, è stato quindi possibile utilizzare i

I.DRIVE Data Logger

LOGGERS

Sensor	Empatica	Procomp	Axis 1	Axis 2	Prosilica 1		Prosilica 2		Xsens IMU	
Sensor node state	stopped	stopped	stopped	stopped	stopped		stopped		stopped	
Topic name	/empatica_sensor	/procomp_sensor	/axis1/image_raw/compressed	/axis2/image_raw/compressed	/prosilica1/image_raw	/prosilica1/camera_info	/prosilica2/image_raw	/prosilica2/camera_info	/xsens/imu	/xsens/mag
Topic logger state	stopped	stopped	stopped	stopped	stopped	stopped	stopped	stopped	stopped	stopped
Delivered msg / sec	0	0	0	0	0	0	0	0	0	0
Dropped msg / sec	0	0	0	0	0	0	0	0	0	0
Traffic (Kbytes / sec)	0	0	0	0	0	0	0	0	0	0

LOGGERS

Sensor	Velodyne		Velodyne IMU		Velodyne GPS	
Sensor node state	stopped		stopped		stopped	
Topic name	/velodyne_points	/velodyne_packets	/imu_data	/fix	/vel	/time_reference
Topic logger state	stopped	stopped	stopped	stopped	stopped	stopped
Delivered msg / sec	0	0	0	0	0	0
Dropped msg / sec	0	0	0	0	0	0
Traffic (Kbytes / sec)	0	0	0	0	0	0

Figura 4.22: Finestra generale in cui vengono visualizzati gli stati dei nodi e le statistiche dei topic

diversi servizi per registrarsi al server, aggiornare il proprio stato e cancellare la registrazione dal server. Inoltre all'interno del ciclo principale di ogni nodo l'oggetto HeartbeatClient è utilizzato per inviare, ad ogni iterazione, il messaggio del tipo `heartbeat/Heartbeat` al server. Le stesse modifiche sono state effettuate anche al codice Python dei nodi andando però ad istanziare l'oggetto HeartbeatClientPy.

4.6.3 Visualizzazione dello stato dei nodi

La visualizzazione dello stato di ciascun nodo è affidata al nodo python `GUI.py` che è contenuto nel pacchetto `idrive_data_logger`. Questo nodo si sottoscrive al topic `state` pubblicato dal nodo server di heartbeat e va a visualizzare, tramite la libreria PyQt, una finestra (Figura 4.22, Figura 4.23) con lo stato di ciascuno dei nodi presenti nel sistema.

Il nodo `GUI.py`, inoltre, permette di visualizzare le statistiche di ogni topic che viene salvato nel database. Per poter visualizzare le statistiche dei topic viene utilizzato uno strumento messo a disposizione da ROS e che deve essere attivato andando a impostare a vero il parametro *enable_ statistics*. Una volta attivato, le statistiche dei topic saranno pubblicate sul topic `statistics` utilizzando un messaggio `rosgraph_msgs/TopicStatistics`. Questo messaggio contiene: il nome del topic, il nome del nodo che pubblica il topic, il nome del nodo che si è sottoscritto al topic, il tempo di inizio e fine della finestra in cui vengono calcolate le statistiche, in numero di messaggi inviati, il numero di messaggi persi e il traffico di dati in byte. Il nodo `GUI.py` non fa altro che sottoscriversi anche al topic `statistics` e visualizza per ogni topic: il numero di messaggi inviati su quel topic al secondo, il numero di messaggi persi al secondo e il traffico in Kbyte al secondo.

LOGGERS

Sensor	Prosilica 1		Prosilica 2		Xsens IMU	
Sensor node state	stopped		stopped		stopped	
Topic name	/prosilica1/image_raw	/prosilica1/camera_info	/prosilica2/image_raw	/prosilica2/camera_info	/xsens/imu	/xsens/mag
Topic logger state	stopped	stopped	stopped	stopped	stopped	stopped
Delivered msg / sec	0	0	0	0	0	0
Dropped msg / sec	0	0	0	0	0	0
Traffic (Kbytes / sec)	0	0	0	0	0	0

Figura 4.23: Dettaglio della finestra generale

4.7 Sincronizzazione

Una parte fondamentale dell'intero progetto è stata la sincronizzazione dei dati acquisiti. Questo problema è stato risolto solo in parte poiché, molti dei sensori a nostra disposizione, non presentano nessun protocollo che ne permetta una sincronizzazione accurata. Gli unici sensori tra quelli utilizzati che presentano un protocollo di sincronizzazione infatti, sono le Prosilica GC1020. Un sensore che comunque possiamo considerare perfettamente sincronizzato è la Velodyne visto che utilizza come tempo di acquisizione quello del ricevitore GPS Garmin del quale è equipaggiata.

Per migliorare il più possibile la sincronizzazione e la temporizzazione di tutti i sensori che non presentano nessun protocollo di sincronizzazione, si è scelto di utilizzare un ricevitore GPS per sincronizzare il clock del PC. La scelta di utilizzare il tempo acquisito dal ricevitore GPS è dovuta al fatto che i ricevitori GPS, grazie ai satelliti GPS ai quali sono agganciati, forniscono un valore di tempo molto preciso. In questa sezione verranno descritti in maniera dettagliata i protocolli di sincronizzazione utilizzati e come sono stati utilizzati all'interno della nostra struttura software.

4.7.1 Sincronizzazione del PC

In questa sezione verrà descritto il funzionamento del protocollo NTP e come quest'ultimo sia stato utilizzato nella nostra architettura.

Network Time Protocol

Il protocollo NTP (*Network Time Protocol*) è un protocollo di sincronizzazione utilizzato per sincronizzare i clock dei computer all'interno di una rete a commutazione di pacchetto, quindi con tempi di latenza variabili e inaffidabili. L'NTP è un protocollo client-server appartenente al livello applicativo ed è in ascolto sulla porta UDP 123. Il funzionamento dell'NTP si basa sul rilevamento dei tempi di latenza nel transito dei pacchetti sulla rete. Attualmente è in grado di sincroniz-

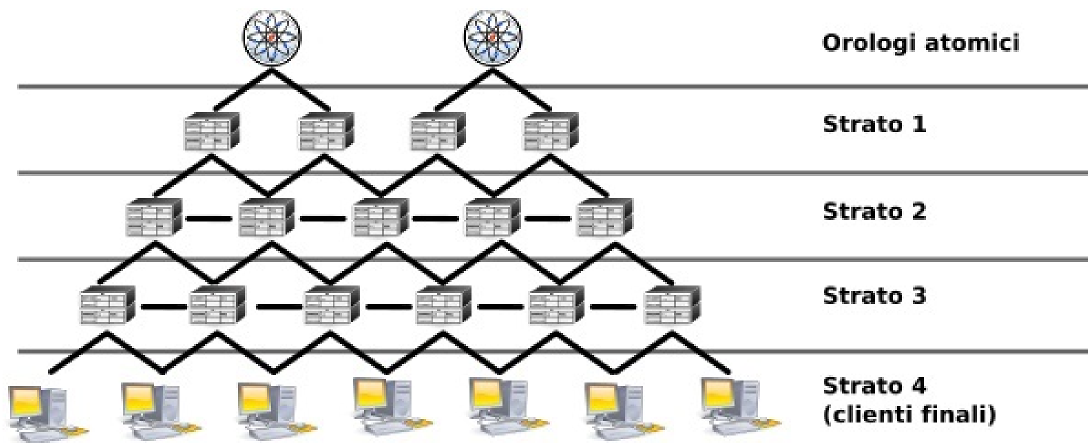


Figura 4.24: Struttura gerarchica a strati del protocollo NTP

zare gli orologi dei computer su internet entro un margine di 10 millisecondi e con un'accuratezza di almeno 200 microsecondi all'interno di una LAN in condizioni ottimali.

I diversi server NTP sono organizzati in una struttura gerarchica a *strati* (Figura 4.24), dove lo strato 1 è sincronizzato con una fonte temporale esterna quale un orologio atomico, GPS o un orologio radiocontrollato, lo strato 2 riceve i dati temporali da server di strato 1, e così via. Il numero massimo di strati è 15 poiché lo strato 16 è utilizzato per indicare che un dispositivo non è sincronizzato. Un server si sincronizza confrontando il suo orologio con quello di diversi altri server di strato superiore o dello stesso strato. Questo permette di aumentare la precisione e di eliminare eventuali server scorretti. Un server NTP è in grado di stimare e compensare gli errori sistematici dell'orologio hardware di sistema, che normalmente è di scarsa qualità.

Per ottenere le migliori prestazioni è utile che la parte relativa alla sincronizzazione dell'orologio sia implementata nel kernel del sistema operativo, piuttosto che affidata a un processo utente (tutte le recenti versioni del kernel Linux hanno questa caratteristica). Il dato a 64 bit scambiato dal protocollo è suddiviso in 32 bit che definiscono i secondi e altri 32 bit per la parte decimale, potendo così rappresentare un intervallo di 2^{32} ed una risoluzione temporale teorica di 2^{32} secondi. Ogni 2^{32}secondi l'intervallo temporale descritto dal protocollo si ripete, per cui è necessario specificare a quale periodo ci si riferisce. Questo non è un problema poiché 2^{32} secondi corrispondono a circa 136 anni.

In Linux per poter utilizzare il protocollo NTP si utilizza solitamente il demone *ntpd*. Il server NTP creato dal demone può essere configurato tramite il file di configurazione /etc/ntpd.conf. Oltre ad aggiornare l'orologio di sistema, ntpd ne stima l'errore sistematico, ed è in grado di correggerlo, evitando un andamen-

to irregolare del tempo, e migliorando la precisione quando il computer non è connesso alla rete. Per avviare il demone ntpd è necessario utilizzare il seguente comando:

```
service ntp start
```

Una volta avviato il demone rimarrà sempre attivo e non sarà più necessario avviarlo ogni volta che lo si vuole utilizzare.

Utilizzo del server NTP

Nella nostra architettura il server NTP è di strato 1 ovvero è sincronizzato con una fonte temporale esterna. La fonte temporale utilizzata nel nostro caso è un ricevitore GPS.

Inizialmente si era pensato di utilizzare con il server NTP il ricevitore GPS Garmin della Velodyne, ma durante lo sviluppo quessta ipotesi è stata scartata poiché presentava diverse problematiche. Il primo problema era che i dati del ricevitore GPS, venendo acquisiti tramite i pacchetti UDP inviati dalla Velodyne e dovendo passare per una catena di nodi ROS prima di poter essere utilizzati (Figura 4.5), presentavano una latenza variabile e non calcolabile. Questa latenza faceva in modo che il tempo del GPS utilizzato per la sincronizzazione fosse in ritardo rispetto al tempo reale non permettendo così una sincronizzazione precisa del clock del PC. La seconda problematica che è emersa nell'utilizzo del GPS Garmin, è stata riscontrata nell'utilizzo del pacchetto ROS `ntpd_driver`. Questo pacchetto fornisce un nodo chiamato `ntp_driver` che avrebbe dovuto prendere il tempo fornito dal GPS su uno specifico topic e condividerlo per mezzo di una memoria condivisa al server NTP. Purtroppo questo nodo non funzionava a dovere e non permetteva di ottenere un offset accettabile tra il tempo della macchina e quello fornito dal GPS. Visto i problemi riscontrati con il GPS Garmin si è utilizzato un secondo GPS con il solo scopo di sincronizzare il PC. Il GPS utilizzato per questo scopo è il GPS Yuan10.

Per poter utilizzare il nostro ricevitore GPS Yuan10 è stato necessario utilizzare il demone *gpsd*. Gpsd è un demone che riceve i dati da un ricevitore GPS e li rende disponibili a diverse applicazioni tra cui ad esempio un software di navigazione o un server NTP. Molti ricevitori GPS sono supportati, sia che essi siano seriali, USB o Bluetooth. Una volta connesso il ricevitore GPS Yuan10 è possibile avviare il demone gpsd tramite il seguente comando:

```
gpsd -D 5 -N -n /dev/ttyUSB0
```

Ad ogni messaggio del GPS contenente un timestamp, il demone gpsd, impacchetta il timestamp ricevuto con il tempo locale corrente e lo invia ad un segmento di memoria condivisa identificato da un ID. Il server NTP, per poter andare a leggere i dati da questo segmento di memoria condivisa, deve conoscere l'ID utilizzato dal demone gpsd. Per andare ad indicare al server NTP l'ID

del segmento di memoria condivisa, bisogna modificare il file di configurazione ntpd.conf andando ad aggiungere le seguenti righe:

```
server 127.127.28.0 minpoll 4 maxpoll 4
fudge 127.127.28.0 time1 0.9999 refid GPS
```

Nelle righe riportate sopra è possibile vedere come viene fornito l'id "GPS" tramite il campo *refid*. A questo punto il nostro server NTP sincronizza il clock del PC con il clock ricevuto dal GPS Yuan10 con una differenza tra i due tempi di al massimo 6 millisecondi negli esperimenti effettuati.

4.7.2 Sincronizzazione delle Prosilica GC1020

In questa sezione viene descritto il funzionamento del protocollo PTP e come quest'ultimo sia stato utilizzato nella nostra architettura.

Precision Time Protocol

Il protocollo PTP (*Precision Time Protocol*) è un protocollo utilizzato per la sincronizzazione dei clock di una rete di computer. Su una rete locale, il protocollo PTP, riesce a raggiungere una precisione che è sotto al millisecondo, rendendolo adatto a sistemi di misura e di controllo.

Il protocollo PTP presenta una architettura master-slave gerarchica per la distribuzione del clock (Figura 4.25). In questa architettura, un sistema PTP consiste in uno o più segmenti di rete e uno o più clock. Un *clock ordinario* è un dispositivo con una singola connessione di rete e può essere o la fonte (*Master*) o il destinatario (*Slave*) della sincronizzazione. Un *clock di confine* presenta più connessioni di rete e può sincronizzare con precisione un segmento di rete con un altro segmento.

Per la sincronizzazione un master viene selezionato per ognuno dei segmenti di rete del sistema. L'origine del tempo di riferimento viene chiamato *grandmaster*. Il granmaster trasmette le informazioni sulla sincronizzazione ai clock presenti nel segmento di rete. Un sistema PTP semplificato consiste in un clock ordinario connesso ad una singola rete senza la presenza di clock di confine. Un grandmaster è eletto e tutti gli altri clock si sincronizzano direttamente con lui.

La sincronizzazione e la gestione di un sistema PTP è ottenuto grazie allo scambio di messaggi attraverso la rete. Il protocollo PTP utilizza i seguenti messaggi:

- *Sync, Delay_Req, Follow_up e Delay_Resp*: sono messaggi utilizzati dai clock ordinari e dai clock di confine e servono per comunicare le informazioni relative al tempo utilizzate per la sincronizzazione dei clock attraverso la rete

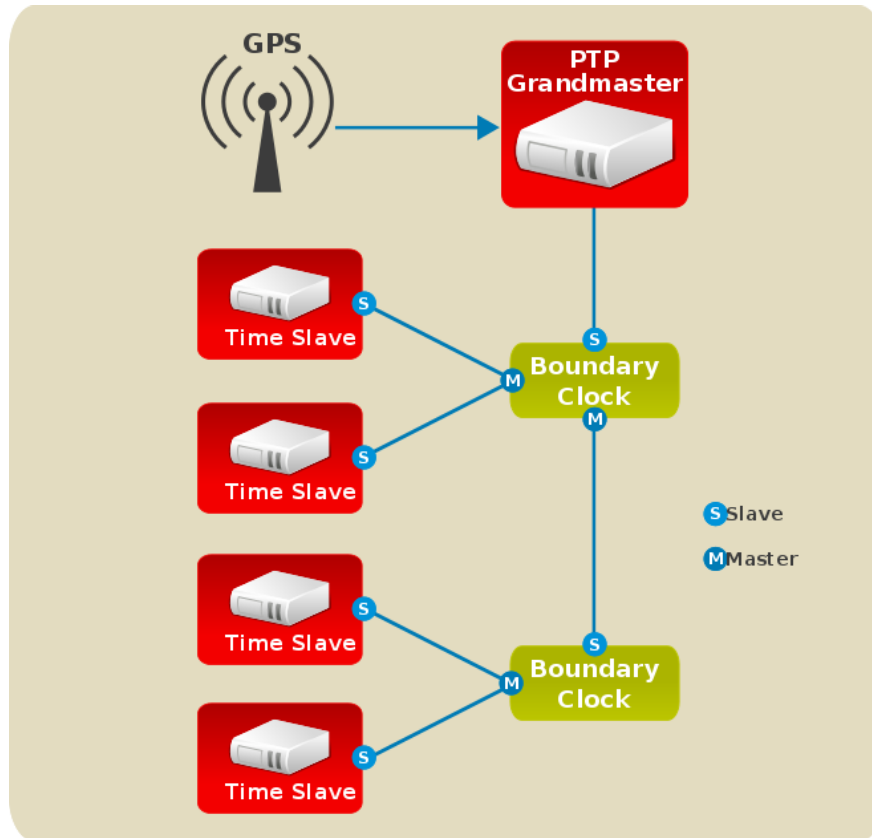


Figura 4.25: Architettura master-slave gerarchica per la distribuzione del clock del protcollo PTP

- *Announce*: sono messaggi utilizzati da un algoritmo particolare per costruire la gerarchia dei clock e selezionare un grandmaster
- *Management*: sono messaggi utilizzati dal gestore della rete per monitorare, configurare e mantenere un sistema PTP
- *Signaling*: sono messaggi utilizzati per le comunicazione non critiche tra i clock

In Linux per poter utilizzare il protocollo PTP si utilizza il demone *ptpd*. Per poter avviare il demone ptpd facendo in modo che utilizzi il clock del computer come master bisogna utilizzare il seguente comando:

```
ptpd2 -M -C -i eth1
```

In questo comando bisogna indicare anche l'interfaccia di rete sulla quale distribuire le informazioni del clock provenienti dal master che nel nostro caso è eth1.

Utilizzo del server PTP

Nella nostra architettura il protocollo PTP è utilizzato per sincronizzare i clock delle camere Prosilica GC1020 con il clock del computer. Per poter sincronizzare le Prosilica con il clock del PC è necessario che quest'ultimo faccia da master. Per avviare il master sul PC basta utilizzare il demone ptpd con il comando descritto in precedenza. L'avvio del demone ptpd deve precedere l'avvio dei driver ROS delle Prosilica GC1020 perché queste ultime sono state configurate in maniera da decidere in maniera automatica se essere master o essere slave. Nel caso infatti in cui il master non sia attivo nel momento in cui i driver della Prosilica vengono attivati accade che una delle due Prosilica si elegga a master e uno a slave. Questo permette, nel caso il master non sia attivato, di avere le immagini delle due camere sincronizzate tra loro anche se non lo saranno con il computer.

Analizzando la precisione tra il tempo del computer e quello presente nei fotogrammi delle camere si nota una differenza di circa 33 millisecondi che sono semplicemente dovuti al fatto che il timestamp al fotogramma è assegnato all'inizio dell'esposizione. Considerando quindi che le Prosilica acquisiscono a 30 fps, è facile intuire che quella differenza di 33 millisecondi è semplicemente il tempo che intercorre tra l'acquisizione di un fotogramma e l'altro e che l'invio del fotogramma avviene alla fine della sua acquisizione e quindi con 33 millisecondi di ritardo.

4.7.3 Sincronizzazione degli altri sensori

Tutti i sensori che non presentano un protocollo di sincronizzazione, esclusa la Velodyne che è sincronizzata con il proprio GPS Garmin, utilizzano come timestamp delle loro acquisizioni il tempo del PC nel momento in cui i dati arrivano al driver ROS. I dati di questi sensori quindi presentano un timestamp che non indica il tempo vero di acquisizione ma il tempo in cui questi dati sono giunti al driver ROS. Escludendo l'errore introdotto dalla latenza della rete ethernet o dalla seriale che non possiamo in nessun modo eliminare o calcolare, abbiamo cercato di limitare l'errore tra il tempo vero di acquisizione e il tempo del PC utilizzato sincronizzando il clock del PC tramite il protocollo NTP con un GPS. Con questi accorgimenti abbiamo migliorato la precisione del timestamp associato ai dati acquisiti e di conseguenza è migliorata la sincronizzazione dei sensori che non presentano nessun protocollo di sincronizzazione.

C A P I T O L O

5. Risultati

5.1 Descrizione dell'intero sistema

I sensori a bordo del veicolo I.DRIVE nono sono stati montati in maniera definitiva, ma sono stati montati in maniera provvisoria poiché il nostro scopo, prima di un'installazione definitiva, era quello di integrare i sensori a livello hardware e software e verificare i requisiti di alimentazione e di risorse.

Il veicolo I.DRIVE con a bordo tutti i sensori è possibile vederlo in Figura 5.1a e, come si può notare, è presente una struttura sopra il tettuccio della macchina che monta la maggior parte dei sensori ambientali e, all'interno, si possono vedere alcuni dei sensori del guidatore, in particolare le telecamere Axis.

La struttura sopra il veicolo (Figura 5.1b) monta, al centro, la Velodyne con posteriormente il relativo GPS Garmin mentre, frontalmente, sono presenti le due telecamere Prosilica GC1020, una rivolta verso la strada (ottica Theia) e una verso il lato destro del veicolo. La IMU Xsens invece non è installata esternamente ma è posizionata all'interno del veicolo.

Nell'abitacolo (Figura 5.1c) è possibile vedere le due telecamere Axis P1343, una rivolta verso il sedile del guidatore e una rivolta verso la strada. Per quanto riguarda i sensori fisiologici, il braccialetto Empatica E4 è indossato dal guidatore mentre i sensori della scheda Procomp Infiniti vengono posizionati sul guidatore.

Internamente viene posizionato anche il PC Shuttle utilizzato dal sistema.

5.2 Dati sperimentali

Vista l'impossibilità di usare su strada il nostro sistema, sono stati svolti dei test in laboratorio per mettere alla prova l'intero sistema e valutarne così l'affidabilità, la quantità di dati che viene raccolta e il carico del sistema.

Il test finale è stato effettuato con l'intero sistema montato e andando a raccogliere i dati sulle prestazioni del PC attraverso il monitor di sistema. I dati riassuntivi raccolti durante il test sono visibili nella Tabella 5.1 ed è possibile vedere come ogni 5 minuti vengano salvati su discro circa 10 GB di dati, dovuti per

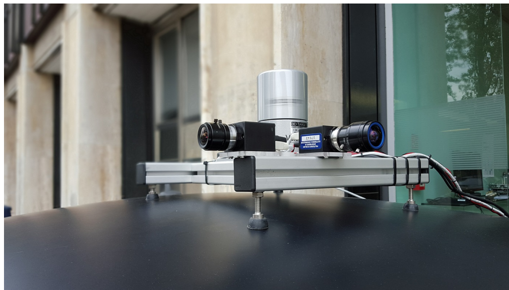
Minuti	RAM utilizzata	Spazio occupato
0 min	5.2 GB	0 GB
5 min	6.2 GB	17.1 GB
10 min	6.2 GB	27.9 GB
15 min	6.3 GB	39.8 GB
20 min	6.2 GB	47.2 GB
25 min	6.2 GB	57.9 GB

Tabella 5.1: Dati riassuntivi dei test effettuati sull'intero sistema

la maggior parte ai dati provenienti dalle telecamere e dalla Velodyne. Durante i test alcune risorse del PC si sono dimostrate appena sufficienti nella gestione dell'intero sistema software. In particolare viene sfruttata la quasi totalità della RAM disponibile (8 GB), soprattutto se si aggiunge il sistema di visualizzazione dei dati, e il disco rigido rotativo da 7200 RPM è al limite delle sua velocità in fase di scrittura. Nel futuro, sopratutto se si volesse espandere ulteriormente il parco sensori, occorrerà fare particolare attenzione a questi due aspetti. La potenza di calcolo fornita dal PC, invece, è risultata essere ampiamente sufficiente con un utilizzo dei processori che oscillava per la maggior parte del tempo tra il 35% e il 55%.

Oltre ai dati sul carico del sistema, durante il test, è stato monitorato l'andamento dell'offset tra il tempo del PC e il tempo fornito dal ricevitore GPS Yuan10 tramite il server NTP. Questo offset oscilla tra i 3 ms e -3 ms circa con un offset massimo rilevato di 6 ms.

(a)



(b)

(c)

Figura 5.1: Sistema I.DRIVE: (a) Veicolo I.DRIVE con installati i sensori, (b) Sensori ambientali (Velodyne, Prosilica) posizionati sul tettuccio del veicolo, (c) Telecamere Axis montate all'interno dell'abitacolo

C A P I T O L O

6. Conclusioni e sviluppi futuri

6.1 Conclusioni

Il risultato ottenuto del lavoro svolto è stato la realizzazione di un sistema affidabile, robusto e, per quanto possibile, sincronizzato in termini di tempo per l'acquisizione e il salvataggio dei dati provenienti dai sensori montati a bordo del veicolo I.DRIVE. Il sistema sviluppato presenta una grande flessibilità e modularità, fondamentale in un progetto simile poiché consente di effettuare facilmente cambiamenti nella configurazione dell'intero sistema senza difficoltà in particolare per quanto riguarda l'aggiunta di nuovi sensori. Il sistema sviluppato ha mostrato nei test svolti di essere affidabile e di poter gestire senza problemi il carico di lavoro che gli viene richiesto.

6.2 Sviluppi futuri

Attualmente sulla vettura sono montati un numero di sensori non molto ampio e diversificato, soprattutto per quanto riguarda i sensori montati esternamente alla vettura che dovrebbero consentire in un futuro la guida autonoma. Il sistema è quindi migliorabile aggiungendo altri tipi di sensori come RADAR e LIDAR fissi e sostituendo i sensori ora presenti che non presentano protocolli di sincronizzazione del clock.

Sempre per quanto riguarda l'hardware un miglioramento che può essere fatto è quello di utilizzare un disco SSD per il salvataggio dei dati visto le prestazioni enormemente migliori rispetto ad un normale disco rotativo da 7200 RPM. Questo potrebbe consentire di acquisire alcuni dati in un formato migliore come ad esempio le immagini delle Prosilica GC1020 che potrebbero essere acquisite in RGB anziché in BayerRG8 perché non si andrebbe a saturare la banda in scrittura sul disco rigido.

Un altro sviluppo che può essere portato avanti sul sistema, è quello che permette, in maniera automatica, alla fine di ogni raccolta di dati effettuata con la vettura, di inviare ed integrare i dati presenti nel database locale dell'auto

con un database centralizzato. Questo permetterebbe di avere un unico database centralizzato contenente tutti i dati raccolti e permetterebbe dopo ogni raccolta dati di liberare spazio nel database locale.

Altri progetti che si baseranno sul lavoro svolto in questa tesi prevederanno in particolare l'analisi e l'utilizzo dei dati acquisiti. I dati dei sensori montati all'interno dell'abitacolo saranno utilizzati per analizzare e modellizzare i comportamenti del guidatore mentre, i dati dei sensori montati esternamente alla vettura, verranno utilizzati per poter ricostruire all'interno di un simulatore l'ambiente attraversato dalla vettura.

Infine, in futuro, si vorrà sfruttare la configurazione attuale dei sensori esterni montati sull'auto (integrandoli con altri sensori) per sviluppare un sistema di guida autonoma.

APPENDICE

A. Manuale di installazione e avvio

Di seguito verranno indicati i requisiti e le istruzioni da seguire per poter far funzionare il sistema sviluppato all'interno della tesi.

A.0.1 Requisiti

I requisiti necessari per poter utilizzare il sistema sono:

- Sistema operativo `Ubuntu Linux` in versione 14.04
- Framework ROS in versione `Indigo Igloo` installata
- `VirtualBox` con `extension pack` e pacchetto `DKMS` installati. La macchina virtuale creata deve aver installato un sistema operativo `Windows`.

A.0.2 Installazione e configurazione software

Macchina virtuale Windows

La macchina virtuale deve essere configurata in modo tale da utilizzare come impostazione di rete una `"scheda di rete bridge"` (Figura A.1). Questo per consentire alla macchina virtuale di poter comunicare con la macchina fisica attraverso la rete. Inoltre la macchina Windows deve utilizzare l'indirizzo IP statico 192.168.1.10.

La macchina virtuale al suo interno deve avere installato i software necessari al funzionamento dei sensori Procomp Infiniti ed Empatica E4 e in particolare:

- Software della Procomp Infiniti fornito insieme al sensore su sopporto CD
- Server per il sensore Empatica E4 fornito tramite eseguibile e scaricabile all'indirizzo `http://get.empatica.com/win/EmpaticaBLEServer.html`

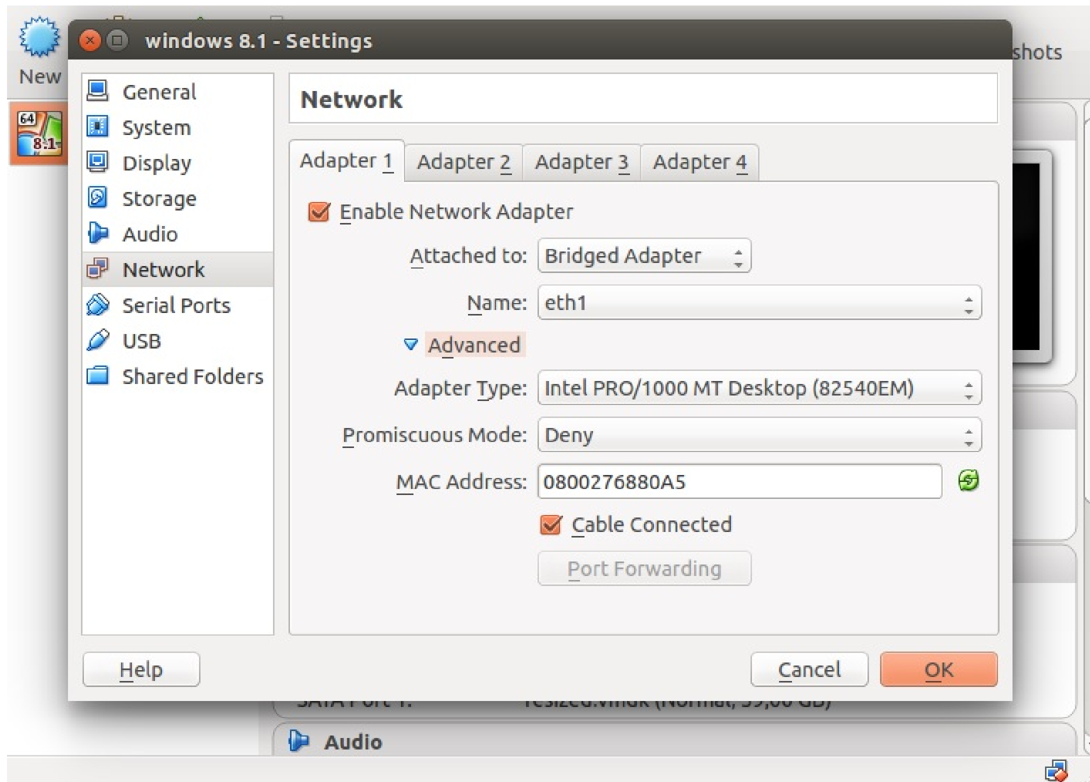


Figura A.1: Finestra di configurazione della scheda di rete della macchina virtuale

Macchina Linux

La macchina Linux deve essere configurata in modo da avere una connessione di rete con `MTU` impostato a 9000 e indirizzo ip statico 192.168.1.3.
Per poter utilizzare la IMU è necessario creare una regola udev così da garantire i permessi di accesso in lettura e scrittura al dispositivo. Il file, chiamato `rules.d`, contenente la regola creata è stato posto nella cartella /`etc/udev/`. La regola creata è così definita:

```
SUBSYSTEM=="usb", ATTRidVendor=="0403", ATTRidProduct=="d38b",
                    MODE:="0666"
```

Qui di seguito verrà descritto come effettuare l'installazione e la configurazione dei pacchetti software aggiuntivi per Linux.

- Installare **mongoDB** tramite il comando:

  ```
  sudo apt-get install python-pymongo mongodb
  ```

  Una volta installato mongoDB è necessario configurarlo. Come prima cosa bisogna creare la cartella che verrà utilizzata da mongoDB per salvare i

dati acquisiti. Per poter creare ed assegnare alla cartella i giusti permessi bisogna eseguire i seguenti comandi:

```
mkdir <percorso cartella>
sudo chown mongodb:mongodb <percorso cartella>
```

Una volta creata la cartella bisogna configurare mongoDB in modo tale che vada ad utilizzarla. Per fare questo bisogna andare a modificare il file che si trova nel percorso `/etc/mongodb.conf` andando ad impostare nel campo `dbpath` il percorso della cartella appena creata. Una volta modificato e salvato il file è necessario riavviare il servizio mongoDB tramite il comando:

```
sudo service mongodb restart
```

- Installare server **NTPd** tramite il comando:

```
sudo apt-get install ntp
```

  Il server NTPd per poter funzionare correttamente necessità dell'aggiunta al file `/etc/ntp.conf` delle seguenti linee:

```
server 127.127.28.0 minpoll 4 maxpoll 4
fudge 127.127.28.0 time1 0.9999 refid GPS
```

- Installare **PTPd2** scaricando l'ultima versione dal sito `http://sourceforge.net/projects/ptpd/files/`. Una volta scaricato eseguire i seguenti comandi nella cartella scaricata:

```
./configure make make install
```

- Installare **GPSd** tramite il comando:

```
sudo apt-get install gpsd
```

- Scaricare i pacchetti ROS presenti nel sito `https://github.com/AIRLab-POLIMI/iDrive` e copiarli all'interno del proprio workspace ROS. Compilare il progetto ROS tramite il comando:

```
catkin_make
```

A.0.3 Avvio del sistema

Una volta installato tutti i pacchetti sopra elencati è possibile avviare l'intero sistema. Per avviare il sistema occorre effettuare i seguenti passaggi:

- Avviare GPSd tramite il comando:

  ```
  sudo gpsd -D 5 -N -n /dev/ttyUSBX
  ```

  Con /dev/ttyUSBX che indica la porta USB utilizzata dal GPS Yuan10 come ad esempio /dev/ttyUSB0.

- Avviare il server PTPd tramite il comando:

  ```
  sudo ptpd2 -M -C -i ethX
  ```

  Con ethX che indica la porta alla quale le camere Prosilica sono connesse come ad esempio eth1.

- Avviare la macchina virtuale e far partire i due eseguibili chiamati **Procomp_server** e **EmpaticaBLEServer** (Figura A.2) . Per consentire la connessione del braccialetto Empatica E4 è necessario premere il tasto Start presente sul `Empatica BLE Server`.

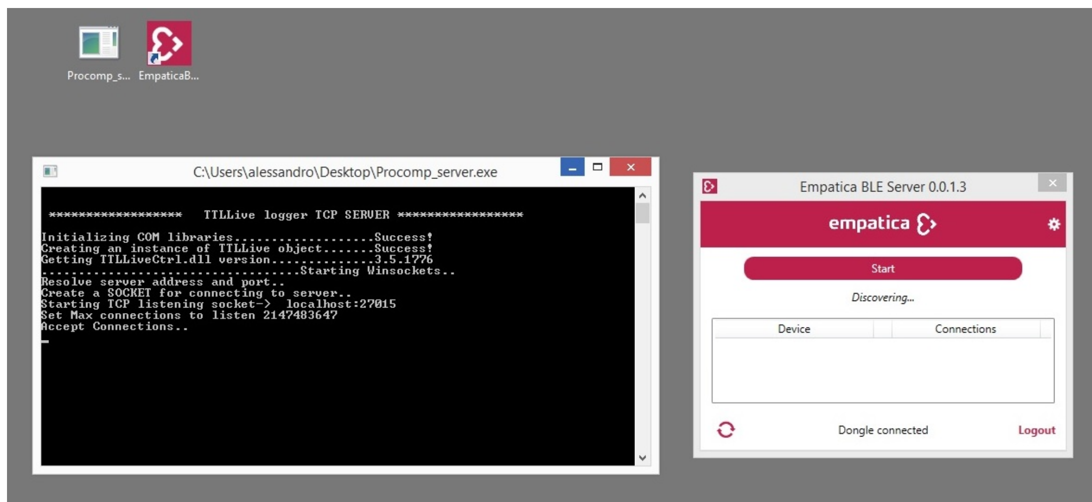


Figura A.2: Applicazioni *Procomp_server* e *EmpaticaBLEServer* avviate

- Avviare il sistema ROS vero è proprio tramite i seguenti comandi:

  ```
  cd <nome cartella workspace ROS>
  source devel/setup.bash
  roslaunch idrive_data_logger idrive_data_logger.launch
  ```

- Per visualizzare i dati provenienti dai sensori bisogna utilizzare il comando:

```
roslaunch idrive_data_logger idrive_data_logger_debug.launch
```

Bibliografia

[1] How google's self-driving car works. `http://www.Spectrum.ieee.org`, 2013).

[2] U.s. department of transportation releases policy on automated vehicle development. *National Highway Traffic Safety Administration*, 2013.

[3] Frank Moosmann Benjamin Ranft Holger Rapp Christoph Stiller Andreas Geiger, Martin Lauer and Julius Ziegler. Team annieway's entry to the grand cooperative driving challenge 2011. *Intelligent Transportation Systems, IEEE Transactions on*, 13:1008–1017, 2012.

[4] Martin Lauer Benjamin Ranft Holger Rapp Julius Ziegler Andreas Geiger, Frank Moosmann. Team annieway's entry to gcdc 2011. 2011.

[5] Ruel Faruque Michael Fleming Chris Terwelp Charles Reinholtz Dennis Hong Al Wicks Andrew Bacha, Cheryl Bauman. Odin: Team victortango's entry in the darpa urban challenge. *Journal of field Robotics*, pages 467–492, 2008.

[6] Alberto Broggi, Stefano Debattisti, Paolo Grisleri, and Matteo Panciroli. The deeva autonomous vehicle platform. *Intelligent Vehicles Symposium (IV),IEEE*, pages 692 – 699, 2015.

[7] David Ray Joshua Anhalt Daniel Bartz Tugrul Galatali Alexander Gutierrez Josh Johnston Sam Harbaugh William Messner Chris Urmson, Charlie Ragusa. A robust approach to high-speed navigation for unrehearsed desert terrain. *Journal of field Robotics*, pages 467–508, 2006.

[8] Drew Bagnell Christopher Baker Robert Bittner M. N. Clark John Dolan Dave Duggins Tugrul Galatali Chris Geyer Michele Gittleman Sam Harbaugh Martial Hebert Thomas M. Howard Sascha Kolski Alonzo Kelly Maxim Likhachev Matt McNaughton Nick Miller Kevin Peterson Brian Pilnick Raj Rajkumar Paul Rybski Bryan Salesky Young-Woo Seo Sanjiv Singh Jarrod Snider Anthony Stentz William "Red" Whittaker Ziv Wolkowicki Chris Urmson, Joshua Anhalt and Jason Ziglar. Autonomous driving in urban envi-

ronments: Boss and the urban challenge. *Journal of field Robotics*, pages 425–466, 2008.

[9] John Dolan Paul Rybski Bryan Salesky William “Red” Dave Ferguson Michael Darms Chris Urmson, Christopher Baker. Autonomous driving in traffic: Boss and the urban challenge. *AI magazine*, 30:17–28, 2009.

[10] Miao Wang Raul Rojas Daniel G ohring, David Latotzky. Controller architecture for the autonomous cars: Madeingermany and e-instein. 2012.

[11] Miao Wang Raul Rojas Daniel G ohring, David Latotzky. Semi-autonomous car control using brain computer interfaces. *Proceedings of the 12th International Conference IAS-12, Advances in Intelligent Systems and Computing*, 194:393–408, 2012.

[12] R. Terry Dunlay. Obstacle avoidance perception processing for the autonomous land vehicle. *Robotics and Automation. Proceedings. 1988 IEEE International Conference on*, 2:912–917, 1988.

[13] Dirk Dickmanns Thomas Hildebrandt Markus Maurer Frank Thomanek Ernst Dieter Dickmanns, Reinhold Behringer and Joachim Schiehlen. The seeing passenger car ’vamors-p’. *Intelligent Vehicles ’94 Symposium, Proceedings of the*, pages 68–73, 1987.

[14] Douglas W. Gage. Ugv history 101:a brief history of unmanned ground vehicle (ugv) development efforts. *Unmanned Systems Magazine*, 1995.

[15] Scott Y. Harmon. The ground surveillance robot (gsr): An autonomous vehicle designed to transit unknown terrain. *Robotics and Automation, IEEE Journal of*, 3:266–279, 1987.

[16] Aytül Erçil Baran Çürüklü Hakkı Can Koman Fatih Taş Ali Özgür Argunşah Serhan Coşar Batu Akan Harun Karabalkan Emrecan Çökelek Rahmi Fıçıcı Volkan Sezer Serhan Danış Mehmet Karaca Mehmet Abbak Mustafa Gökhan Uzunbaş Kayhan Eritmen Mümin Imamoğlu Çağatay Karabat Hüseyin Abut, Hakan Erdoğan. Real-world data collection with “uyanik”. *In-Vehicle Corpus and Signal Processing for Driver Behavior*, pages 23–43, 2009.

[17] Seth Teller Mitch Berger Stefan Campbell Gaston Fiore Luke Fletcher Emilio Frazzoli Albert Huang Sertac Karaman Olivier Koch Yoshiaki Kuwata David Moore Edwin Olson Steve Peters Justin Teo Robert Truax John Leonard, Jonathan How and Matthew Walter. A perception-driven autonomous urban vehicle. *Journal of field Robotics*, pages 727–774, 2008.

[18] Pınar Boyraz Lucas Malta Chiyomi Miyajima Kazuya Takeda, John H.L. Hansen and Hüseyin Abut. International large-scale vehicle corpora for research on driver behavior on the road. *Intelligent Transportation Systems, IEEE Transactions on*, 12:1609–1623, 2011.

[19] Chiyomi Miyajima Norihide Kitaoka Kazuya Takeda Lucas Malta, Akira Ozaki. A multimedia corpus of driving behaviors. *Multimedia Signal Processing, 2009. MMSP '09. IEEE International Workshop on*, pages 1–6, 2009.

[20] S Furst F Thomanek Markus Maurer, Reinhold Behringer and ED Dickmanns. A compact vision system for road vehicle guidance. *Pattern Recognition, 1996, Proceedings of the 13th International Conference on*, 3:25–29, 1996.

[21] Gianni Conte Alessandra Fascioli Massimo Bertozzi, Alberto Broggi. Vision-based automated vehicle guidance: the experience of the argo vehicle. *Tecniche di Intelligenza Artificiale e Pattern Recognition per la Visione Articiale*, pages 35–40, 1998.

[22] Suhrid Bhat Hendrik Dahlkamp Dmitri Dolgov Scott Ettinger Dirk Haehnel Michael Montemerlo, Jan Becker. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, pages 569–597, 2008.

[23] Hans P Moravec. The stanford cart and the cmu rover. *Springer*, 1990.

[24] Nils J. Nilsson. Shakey the robot. *Technical report*, 1984.

[25] Powell M Kinney Cris Koutsougeras Paul G Trepagnier, Jorge Nagel and Matthew Dooner. Kat-5: Robust systems for autonomous vehicle navigation in challenging and unknown terrain. *Journal of field Robotics*, pages 509–526, 2006.

[26] Amardeep Sathyanarayana John H.L. Hansen Pongtep Angkititrakul, Matteo Petracca. Utdrive: Driver behavior and speech interactive systems for in-vehicle environments. *Proceedings of the 2007 IEEE Intelligent Vehicles Symposium*, pages 566–569, 2007.

[27] Hendrik Dahlkamp David Stavens Andrei Aron James Diebel Philip Fong John Gale Morgan Halpenny Gabriel Hoffmann Sebastian Thrun, Mike Montemerlo. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, pages 661–692, 2006.

[28] Dan Shapiro. Three anecdotes from the darpa autonomous land vehicle project. *AI Magazine*, 2008.