POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

SELF-DRIVING CARS AND OPENPILOT: A COMPLETE OVERVIEW OF THE FRAMEWORK

TESI MAGISTRALE IN INGEGNERIA INFORMATICA

CANDIDATO: FRANCESCO FONTANA

RELATORE: LUCIANO BARESI

ANNO ACCADEMICO: 2020-2021

1. Introduction

The approach of many car manufacturers is that of developing their machine learning algorithm and providing their cars with a precise set of sensors. However, in a trial to democratize self-driving cars and make them available to everyone, Comma.ai with Openpilot offered a single device, able to bring the power of a machine learning algorithm trained on thousands of hours of drive in any compatible car. **Openpilot** is an open-source, semi-automated driving system developed by the company Comma.ai, founded by George Hotz. It is a comprehensive system of driver assistance features supporting a wide range of car models. Over 200 users contributed to its development and each new release of the software includes new functionalities and bug fixes that were made possible also thanks to the users' feedbacks and concrete additions to the code base. Even if a basic documentation is available on the official GitHub repository page, no technical documentation, explaining what each software component of which Openpilot is made does, is available, and also the integration of the different submodules with the main software is not made clear by the documentation, resulting in a lot of confused developers that are willing to contribute but don't know how to work with the code base.
The purpose of the thesis is to explicit what are the relationships of dependency that incur among the submodules and Openpilot, and assess the quality of the software by looking at the way it is tested and analyzing the development process that went through over the years.

2. Submodules

Openpilot needs many components that allow the software to interface with the car and exchange messages with it. These components, after the open-sourcing of the software, were organized in different repositories, allowing to better manage them and have a clear distinction of what role each component plays. In this excursus of the submodules that are available in the main Openpilot repository will be analyzed the functionalities that each one of them provides, how they were tested to ensure the required levels of reliability, and what was the development process they went through.

2.1. Cereal

Cereal is both a messaging specification for robotics systems as well as a generic high-performance inter-process communication protocol enabling the communication among a single Publisher and multiple Subscribers (IPC pub/sub) for all the components that implement it. One of its main purposes is to enable easy and effective logging of all the events that occur during the usage of Openpilot, as well as enabling the different modules of the software to communicate with each other. The main components that cereal provides are *messaging*, which is the actual messaging specification library, and *VisionIPC*, which allows exchanging visual data.
The PubSub mechanism is implemented by the components PubSocket and SubSocket, that allows the processes of Openpilot to subscribe to a certain socket and exchange messages. There can be only one process sending messages on a certain socket, but multiple processes can subscribe to that same socket and receive the messages.

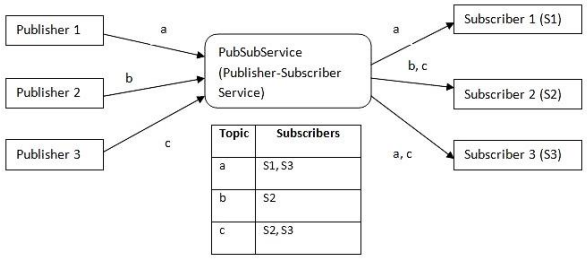

Figure 1 - PubSub design pattern

Similarly, VisionIPC implements the same mechanism to exchange the acquired camera frames and among the different processes. The frames can be encoded in RGB or YUV, according to the usage that will be done with them. The only component managing acting as the Publisher of the camera frame is the process managing the cameras of the device where Openpilot is executed, while the acquired frames are received by the predictive model of the software, that based on what detects in the external environment has to quickly predict where to lead the car.

2.2. Common

The common package contains methods, variables, and processes that are used by all the other packages to perform common operations. The provided functionalities include the API developed by Comma.ai and available at https://api.commadotai.com/, a simple Kalman filter, and a series of methods useful to transform the reference frames that are used by the software. The API provide different functionalities to retrieve the key parameters of the car that are recorded by the device where Openpilot is installed and expose them on a web socket to access them remotely. The key process managing the creation of the web socket and of the response to send to the remote caller is *Athena*.
The Kalman filter is used to smooth the series of values acquired from the different actuators. It uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. The involved variables in the application of the Kalman filtering algorithm are the initial state (x_0), the state transition matrix (A), the state covariance matrix (C), and the Kalman gain (K). The first step of the algorithm is the predictive step, which applies (2.1) to the acquired variables.

$$\begin{bmatrix} AK_0 & AK_1 \\ AK_2 & AK_3 \end{bmatrix} = \begin{bmatrix} A_{00} - K_{00}C_{00} & A_{01} - K_{00}C_{01} \\ A_{10} - K_{10}C_{00} & A_{11} - K_{10}C_{01} \end{bmatrix} \quad (2.1)$$

The second step is the update step, in which the prediction is used to calculate the new state of the system.

$$\begin{bmatrix} x_{00} \\ x_{10} \end{bmatrix} = \begin{bmatrix} AK_0 * x_{00} + AK_1 * x_{10} + k_{00} * meas \\ AK_2 * x_{00} + AK_3 * x_{10} + k_{10} * meas \end{bmatrix} \quad (2.2)$$

The implementation in *common* provides the methods to perform these two steps, optimized for the Openpilot software.
The transformation functionalities provide the methods to transform the different reference frames that are used by Openpilot, which include Geodetic, ECEF (Earth-Centered, Earth-Fixed), NED (North, East, Down), Device, Calibrated, Car, View, Camera, Normalized camera, Model, and Normalized model frames. The transformation functionalities are mainly used on the acquired camera frames, to convert the acquired images in the correct format.

2.3. Laika

Laika is an open-source library for processing GNSS.The Global Navigation Satellite System

(GNSS) refers to the set of constellations of satellites that include Europe's **Galileo**, the USA's **NAVSTAR Global Positioning System (GPS)**, Russia's **Global'naya Navigatsionnaya Sputnikovaya Sistema (GLONASS)**, and China's **BeiDou Navigation Satellite System**. The different constellations of satellites provide signals from space that include positioning and timing data and the GNSS receivers use all these data to determine the exact location.
Laika can process raw GNSS observations with data gathered online from various analysis groups to produce data ready for position/velocity estimation, producing accurate results, readable and easy to use. One of the possible methodologies that can be adopted to determine the position of a GNSS receiver is that of the Time-Of-Arrival. The different satellites have known orbits, so it is possible to determine their positions at any time. Since the signals travel at the speed of sound, by measuring the time that passes from when the signal is sent to when is received is possible to calculate the distance of the receiver from the satellite. The possibility to acquire data from different sources and combine them makes Laika much more precise than the data acquired by U-blox, the GPS device embedded in the Comma device. The data acquired by Laika can be pre-elaborated and cached, allowing the system to quickly access them. Filtering these data using a Kalman filter gives results that are on average 40% more reliable.

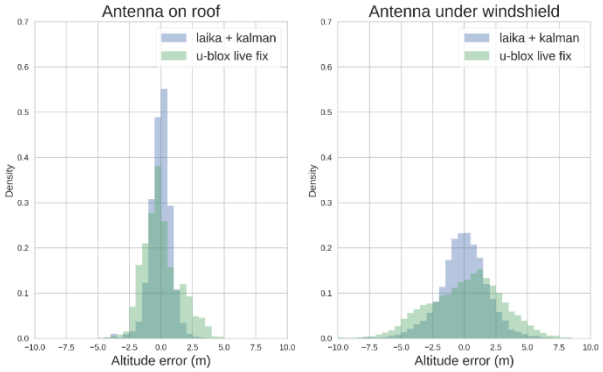

Figure 2 - Comparison of the measurements acquired with Laika and U-Blox.

2.4. OpenDBC

OpenDBC is a repository containing all the reverse-engineered signals corresponding to the supported car. The signals coming from the cars are reversed engineered through Cabana, accessible at https://my.comma.ai/cabana/, which allows visualizing the messages exchanged on the CAN bus and creating a DBC file specific for the car. In the master branch are also provided tools that allow generating a DCB file. A DBC file is a proprietary file format that describes the data over a CAN bus. OpenDBC provides the component to interpret, parse and generate CAN messages that can be correctly interpret by the car as a normal message traveling on the CAN bus, with no distinction from other messages sent by other car components. OpenDBC allows, together with the tool Cabana, to easily add the support to new car models by defining the structure of the message that travel on its CAN bus. Cabana helps to reverse-engineer these messages by interfacing directly with the car and provides an interface to easily generate a new DBC file.

2.5. Panda

Panda is a universal car interface developed by Comma.ai. It connects to the ODB-II port and the camera of the car, supporting the majority of communication busses adopted by many car manufacturers. In combination with OpenDBC, it allows to read and interpret all the signals traveling on the car network. The device, embedded in the newer version of the Comma devices (Comma Two and Comma Three) is based on a STM32 board and is able to support 3 CAN buses, 2 LIN buses, and 1 GMLAN bus to interface with the car. The software that is executed on the Panda device allows Openpilot to send and receive messages, delegating to this component the conversion into signals that can be sent over the buses.

2.6. Rednose

Rednose is a Kalman filter library that can be used for a wide range of optimization problems. In particular, it is used for problems in the field of visual odometry and sensor fusion localization (SLAM). It is designed to provide very accurate results, work online or offline, and be computationally efficient. The library applies the **Rauch-Tung-Striebel** (RTS) Smoother algorithm, which is composed of two passes: the forward pass consists of a standard Extended Kalman Filter (EKF), while the backward pass is introduced to reduce the inherent bias in the EKF estimates. In estimation theory, EKF is the nonlinear version of

the Kalman filter which linearizes about an estimate of the current mean and covariance. The EKF can be considered as the de facto standard in the theory of nonlinear state estimation, navigation systems, and GPS. The library offers a series of helper functions and classes that allow performing the filtering actions.
The Extended Kalman Filter allows to predict values that are much closer to the actual measurements acquired than a normal simulation is able to do.

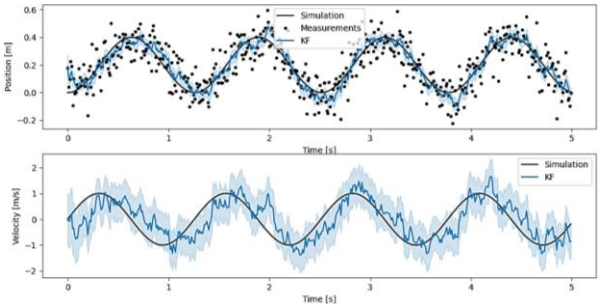

Figure 3 - Kinematik EKF simulation plot

3. Selfdrive

The source code of Openpilot, is contained in the directory *selfdrive* and includes the implementation of all the processes that compose the software.

3.1. AthenaD

This service allows real-time communication with the car. It runs also if the car is parked and not moving and allows to access different functionalities of the car from a dedicated application.

3.2. BoardD/PandaD

This process *boardD* represents the receiving side of the Panda firmware. It parses and sends data through USB by using the library *libusb*. The board daemon is started by the process *pandaD*, which is a Python wrapper of boardD that updates the Panda first. It uses the Python Panda library to configure the device and after that, it launches the main *boardD* process.

3.3. CameraD

The camera daemon captures both the road and driver camera and handles autofocus and autoexposure.
The camera daemon uses VisionIPC, in combination with the Cereal library, to send the frames data to the other component. In particular, the VisionIPC server sends the data frames directly to the model daemon, which uses the frames to compute the predictions.

3.4. ControlsD

This process represents the main 100 Hz loop driving the car. It receives a plan from the planner daemon and constructs the CAN packets required to actuate that plan.

3.5. PlannerD

After processing the camera images through the AI model, Openpilot has to compute a way to bring the car in a position that is coherent with that indicated by the output model. The planner process executes three Model Predictive Control (MPC) loops based on Automatic Control and Dynamic Optimization (ACADO), one for lateral control and two for longitudinal control.

3.6. RadarD

This process parses the data acquired by the radar.

3.7. CalibrationD

This process canonicalizes the acquired frames by converting them into calibrated frames, which are then used by the other Openpilot components. This is important because users can mount their Comma devices in different positions and transforming them allows the model to ignore the error in the predictions that this could introduce.

3.8. LocationD

This process runs a global localizer, which estimates the vehicle position, speed, and acceleration and how they change in the three dimensions. It combines the data coming from multiple sources, including the camera, the GPS and inertial measurement unit (IMU) sensors.

3.9. UbloxD

Comma devices come with a u-blox chip, which is capable of acquiring data from up to three GNSS concurrently, granting a high level of accuracy. U-blox data are acquired by the Panda, published on a dedicated socket, and then parsed.

3.10. ModelD

The main model takes in a picture from the road camera and answers the question "Where should I drive the car?" It also takes in a desire input, which can command the model to act.

3.11. DMonitoringModelD

The Driver Monitoring Model tracks the head pose, eye positions, and eye states. It runs on the Digital Signal Processor to not use CPU or GPU resources needed by the other daemons, giving it room to grow.

3.12. DMonitoringD

The driver monitoring process takes the data elaborated from the driver monitoring model and the other component monitoring the status of Openpilot.

3.13. LoggerD

This daemon subscribes to all the sockets and log all the messages intercepted. It also subscribes to all the device's camera and saves the drive recording.

4. Conclusions

Each submodule is itself a complex process that deals with different aspects of the software, and each component is necessary to make Openpilot work reliably and safely. Due to the risks that malfunctioning of Openpilot could bring, both for the driver and other people on the road, the software is heavily tested and has to respect strict security constraints.
As turns out, the performances of Openpilot are comparable, and often higher, than other car manufacturer's self-driving solutions.

5. References

[1] "comma.ai blog," Comma.ai, [Online]. Available: https://blog.comma.ai/

[2] "openpilot" Comma.ai, [Online]. Available: https://github.com/commaai/openpilot

[3] "DBC Format," 11 October 2017. [Online]. Available: http://socialledge.com/sjsu/index.php/DBC_Format

[4] M. di Preez, "OBD II diagnostic interface pinout," 2 December 2017. [Online]. Available: https://pinoutguide.com/CarElectronics/car_obd2_pinout.shtml

[5] C. Woei-Leong and H. Fei-Bin, "Implementation of the Rauch-Tung-Striebel Smoother for Sensor Compatibility Correction of a Fixed-Wing Unmanned Air Vehicle," November 2011. [Online]. Available: https://doi.org/10.3390/s110403738