CISC322/326 Conceptual Architecture of Apollo

Assignment 1, February 18, 2022

Group 39

Name & Student Numbers:

Denise Micu (20061143)

Aidan Richards (20102555)

Max Beninger (05852667)

Leo Toueg (20062092)

Sean Guo (10157679)

Spencer Venable (20059095)

Abstract

The present report examines the conceptual architecture of the Apollo open-source self-driving car platform, developed by Baidu. Based on an examination of the documentation available on GitHub and the Apollo website, it is concluded that Apollo uses a process-control architecture and consists of the following main components: (1) Perception, (2) Prediction, (3) Planning, (4) Control, (5) Localization, (6) Routing, (7) CANBus, (8) Monitor and (9) HMI (Human-Machine Interface). Our report outlines the relationships between these components (including dependencies and data flow), as well as the evolution of the system over time. We also provide three use cases that demonstrate how the system functions in several common use contexts.

1. Introduction

In recent years, the race to create well performing autonomous vehicles has picked up, seeing not only names such as Tesla competing but as many as 40+ automobile and software companies exploring the creation of autonomous vehicles. As such, the Apollo Platform was created as an open platform solution to autonomous driving. Apollo helps accelerate the development and testing of autonomous vehicles, allowing the software to be distributed, tested and developed by interested individuals.

As a result of Apollo's open platform format, there is a great deal of documentation, code, and information available to better form an understanding and analyze the architecture of the platform. After extensive research, our team has distinguished the system as having 9 pertinent subsystems: Perception, Prediction, Planning, Localization, Routing, Control, CANBus, Monitor, and HMI (Human-Machine Interface). These subsystems work together to create the collective Apollo architecture. From this information, we have identified Apollo as having a Process-Control style architecture, which can be summarized as a software architecture in which the properties of the outputs are maintained near given reference variables, and which controls the execution of the system.

In this report, we will break down the high-level Architecture and sub systems of the Apollo Platform, as well as discuss in detail the interaction and specific function of each subsystem, using sequence diagrams, and visualizing different use cases and how the subsystems interact in those scenarios. Additionally, we will analyze and compare the evolution of the platform by analyzing the improvements and changes made in each evolution. Lastly, the implications for division of responsibilities among participating developers, and conclude with a summary of the discussed findings.

2. Derivation Process

To familiarize ourselves with the Apollo project and self-driving cars in general, we started by searching for relevant YouTube videos that would give us a rough understanding of the domain. We found that the Apollo project is an open-source project developed by the Chinese internet company Baidu, and that it intended to serve as a template for how to create a self-driving car

system. We also learned from the videos that there have been many other solutions built off/incorporated with the Apollo platform including Robotaxis and Minibuses.

The next place we looked for information was a thesis we found called "Software System of Autonomous Vehicles: Architecture, Network and OS" (also listed as a resource on onQ) and a journal article that gave a functional reference architecture for autonomous vehicles. These two sources gave us an understanding of the reference architectures that have been used in the development of autonomous vehicles as well as their benefits/limitations. They also gave us an idea of how autonomous vehicle projects typically divide up functionalities into the categories of Perception, Decision and Control, and Vehicle manipulation.

The final place we looked for information was the documentation on GitHub which contained very detailed information about the purpose of each module as well as their interdependencies. The Apollo GitHub repository gave us some excellent diagrams to work with some of which we included here. It also clearly showed data inputs and outputs for each module and a detailed version history of the development of Apollo.

3. Architectural Overview

Apollo uses a Process-Control software architecture style – a version of data flow/pipeline architectures that work by sending data through various modules. Process-Control architecture is characterized by control loops whereby a controller attempts to achieve a certain outcome as measured by a set of variables. The classic example for this is a car's cruise control where the car's software attempts to maintain a constant speed by increasing or decreasing the throttle based on the car's speed via feedback sent from the speed sensor back to the controller. Apollo's Process-Control architecture is like the above example, albeit with greater complexity. Instead of having just one variable being measured like speed, the car's chassis tracks many different values including speed, acceleration, orientation etc. Also, instead of needing to maintain constant values for these attributes, there is a vector for each attribute which represents the upcoming desired values for speed, acceleration etc. in successive snapshots.

In terms of its high-level structure, we have identified 9 main modules that make up the core of Apollo's open-source architecture: Perception, Prediction, Planning, Localization, Routing, Control, CANBus, Monitor, and HMI (Human-Machine Interface). The dependencies between these components are shown in the following diagram (Figure 1).

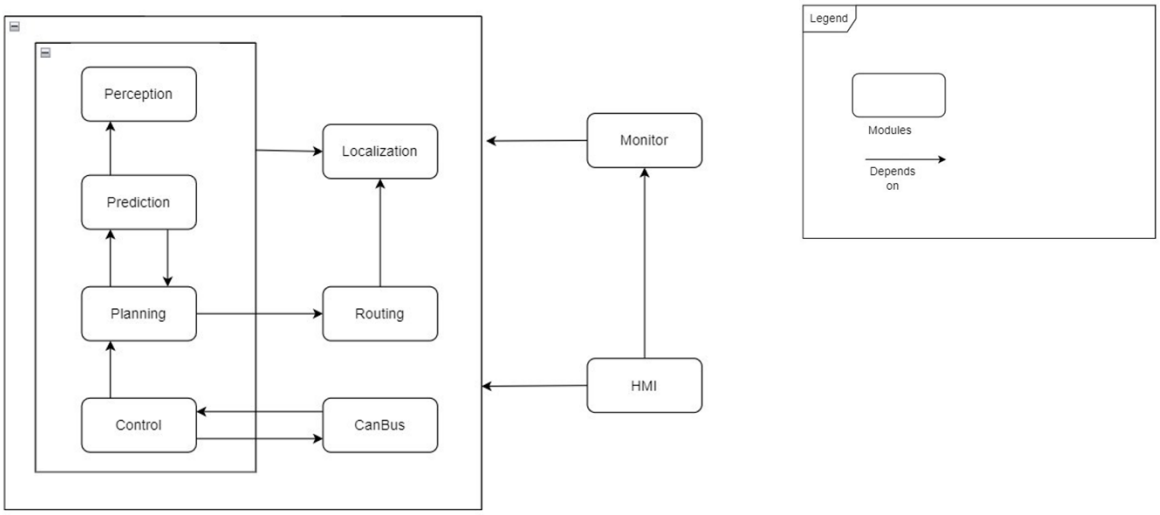


Figure 1: Apollo High-Level Architecture

The "core" of the system is the innermost box containing Perception, Prediction, Planning and Control. These modules collectively perform the function of taking in sensory input, and – through a series of processing steps – generating a set of executable instructions that are relayed to the vehicle's hardware (via the CANBus). Here, each stage of processing is dependent on the one that comes before it: Control depends on Planning, Planning on Prediction, and Prediction on Perception.[1] Crucially, the innermost modules are also dependent on the Localization module, which serves to track the vehicle's current location and speed. The location/speed information provided by the Localization module is important, because it is used to inform predictions regarding nearby obstacles, as well as planning decisions. Planning is further dependent on the Routing module, in particular, since the vehicle's planned trajectory needs to be sensitive to the route inputted by the passenger.

At the periphery of the system, we have the Monitor and the Human Machine Interface (HMI). The Monitor module is responsible for monitoring the functioning of system in order to detect possible failures and/or abnormal behavior. Finally, the HMI module provides the user interface for the system. The HMI takes data from the rest of the modules (including the Monitor) and translates it into a human-readable format, to be presented to the user (here, the vehicle's passenger).

[1] There is a two-way dependency between Prediction and Planning, because the trajectory outputted by planning will have an effect on future predictions.

4. Subcomponents

4.1 Perception

Overview: The perception subsystem is a key component of the Apollo architecture. The main functionality of this subsystem is to recognize and detect obstacles as well as traffic signals. The perception subsystem interacts with the other subsystems, as well as the hardware components, namely using LiDAR[2] and RADAR data to classify and segment obstacles, as well as predict possible motion and information about the relation of the object in respect to the high-resolution map. The traffic light perception component relies on camera input, as well as sensor parameters namely in respect to focal length and position. This allows for a bounding box to identify the light and color states using a convolutional neural network for recall.

Inputs: The perception module takes multiple data inputs in order to create an accurate perception of the region of interest. The inputs of the module are the data from the LiDAR Sensor (Light Detection and Ranging), inputs from the RADAR Sensor, which is a sensor that converts microwave signals into electrical signals and are used to detect motion, and image data. The perception module also obtains data from the extrinsic parameters of radar sensor calibration, extrinsic and intrinsic parameters of front camera calibration, and the velocity and angular velocity of the vehicle.

Outputs: The intended outputs of the perception module are a 3D obstacle track which gives the classification information, and velocity of any obstacles detected in the region of interest as well as the output of the traffic light detection.

Obstacle Detection

Obstacles are identified using both LiDAR and RADAR based perception and using the detection results from both to create a result. The LiDAR obstacle perception implements object segmentation based on the following attributes: foreground probability, object class probability, and offset displacement w.r.t object center using a Convolutional Neural Network.

The RADAR obstacle perception processes the initial RADAR data input, then alters the data by removing noise, extending the track ID, building obstacle results, and filtering for the results based on the region of interest.

The results of both the LiDAR perception and RADAR perception are merged in the obstacle results fusion for more accurate and layered object detection.

Traffic Light Detection

The Traffic Light detection module is used to obtain the exact coordinates of where traffic lights are located. The traffic lights are extrapolated from the real-world coordinates into image coordinates from sensor results. The region of interest is then isolated, and a bounding box is

[2] LiDAR (Light Detection and Ranging) is similar to RADAR except that it uses laser light instead of sound to map its surroundings.

created to isolate and identify the different color states of the lights, a sequential reviser is then used to correct the state of the light.

4.2 Prediction

Overview: The prediction module outputs the predicted behavior of all obstacles perceived by the perception module. The prediction module receives data pertaining to all obstacles as well as their positioning, headings, velocities, accelerations, and then generates predicted trajectories along with a probability representing the likelihood for said prediction to become reality.

Inputs: Data flowing into the prediction module is sourced from the: perception, localization and planning modules. The perception data includes information on the currently perceived obstacles. Data from the localization module provides the predictor context about the vehicle's current position in space. And lastly, the data from the planning module informs the predictor the calculated route of the vehicle from the previous computation cycle.

Outputs: The final output of the prediction module is all obstacles annotated with their predicted trajectories and priority level. The possible priority levels include: ignore, caution and normal (default). Output is sent to the planning module for further processing.

4.3 Planning

Overview: The planning module is responsible for devising a safe path for the vehicle based on its current environment. The module takes in information about the car's surroundings – including data regarding perceived obstacles and traffic signals – and then uses this information to compute an optimal trajectory that avoids obstacles and obeys traffic laws. In terms of its internal architecture, the planning module consists of a series of "scenario handlers" that are designed to deal with different traffic scenarios (e.g., lane follow, stop sign, traffic light, lane transition, etc.).[3] When confronted with perceptual input that matches one these scenarios, the planning module will call the relevant scenario-handler, and this scenario-handler will then work to construct an optimal trajectory.

Inputs: The planning module receives input from Prediction, Localization and Routing modules. The input from the prediction module contains information about traffic signals and the projected behavior of potential obstacles. Inputs from the localization module and routing module are also necessary to ensure that the vehicle stays on the destined route.

Outputs: The planning module provides output to both the control module and the prediction module. The trajectory outputted to the control module is translated into low-level vehicle commands and used to guide the actual behavior of the vehicle. The trajectory information passed

[3] These reflect the current scenarios but more could be added.

back to the prediction model, by contrast, is used to help update predictions about nearby obstacles (based on the vehicle's anticipated movements).

4.4 Localization

The localization module uses different combinations of sensor data depending on its mode to continuously locate the vehicle. The module has two different modes that can be switched between and used to cross verify. The first is called OnTimer and it uses a timer-based callback function to continuously retrieve and combine information from the sensors. This method is RTK-based which stands for Real Time Kinematic and incorporates GPS information with IMU (inertial measurement unit) data to determine current position and orientation. The second mode is called multiple sensor fusion (MSF) and integrates information from a variety of modules but by using event-triggered callback functions as opposed to timer-based. This method incorporates GPS, IMU, as well is LIDAR information (the vision of the vehicle).

The RTK mode is capable of providing high accuracy and can locate the car to within 5-10 cm of precision. While this method is relatively immune to weather obstructing the signal, signal blockages due to atmospheric conditions and physical obstructions (tunnels, underground parking lots etc) sometimes require the inclusion of LIDAR data in MSF mode.

4.5 Routing

The routing module is responsible for generating high level navigation information to ensure the autonomous vehicle can reach its desired destination. It takes the map data, location of start and destination as inputs for analysis to testing different route options by a series of changes to different lanes and roads and eventually comes up with the optimal sequence of roads.

4.6 Control

The control module is responsible for producing a comfortable driving experience and can work in both normal and navigation modes, it consists of five main data interfaces: OnPad, OnMonitor, OnChassis, OnPlanning and OnLocalization, the OnPad and OnMonitor interfaces process data between PAD-based human interface and simulations. The planning module outputs trajectory data which is being passed to the control module which uses it as part of the input to generate control commands. Besides the trajectory data the control module also takes other inputs including the status of the car (i.e. engine state, speed, throttle, steering), the localization information (grid, orientation and velocity) and the applicable Dreamview AUTO mode change request. The control module receives all the inputs and then generates control commands including throttle, brake and steering. The command output is then sent to the CanBus module which is responsible for passing control commands to vehicle hardware for execution.

4.7 CanBus (Control Area Network Bus)

The Canbus module is an interface that connects directly with the control module, it takes the output of the control module, the output commands as its own input and passes them to the vehicle hardware for execution. Besides the hardware execution it is also responsible to pass up the detailed chassis information back to the control module for feedback.

4.8 Monitor

The monitor module contains system level software such as code to check hardware status and monitor system health. It receives data from different system cycle modules, and forwards it to the HMI, ensuring every module is functioning properly. In the event of a failure, it sends data to the guardian module, which makes decisions and communicates directly with CANBus. Things the monitor module oversee are:

- Status of running modules
- Monitoring data integrity
- Monitoring data frequency
- Monitoring system health (e.g., CPU, memory, disk usage, etc)
- Generating end-to-end latency stats report

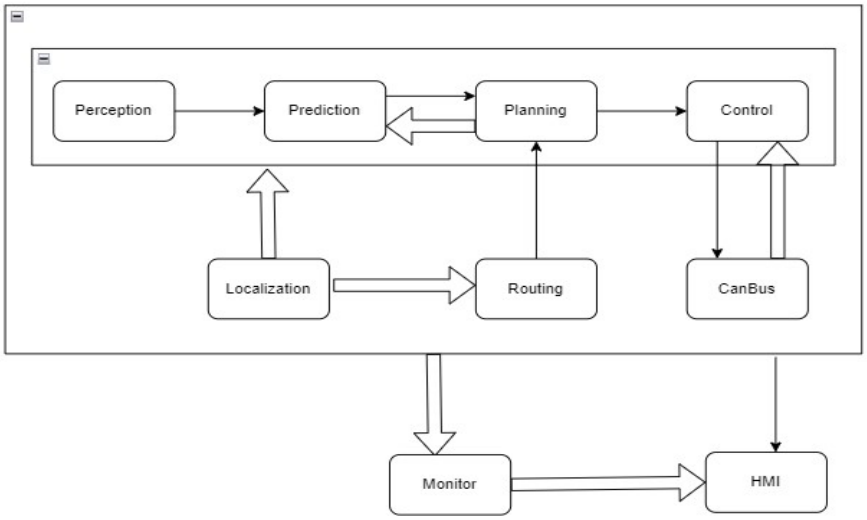
Since we are more interested in the software monitoring, the Monitor module has three different submodules: (1) a Process Monitor that checks if a process is running or not; (2) a Topic Monitor that checks if a given topic is updated normally; and (3) a Summary Monitor that summarizes all other specific monitor's results.

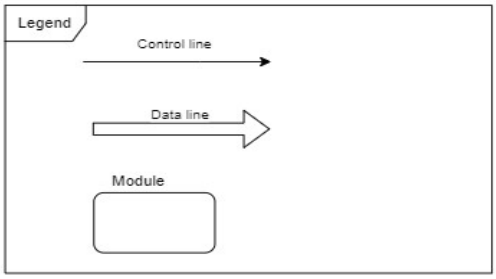
4.9 HMI (Human Machine Interface)

HMI stands for human machine interface, and it is the module responsible for providing an interface with the computer. Also called 'dream view' the module provides a way to visually display the current output of relevant modules using a web interface. For example, it gives the user a visualization of the output of the car localization module (the car's position), planning trajectory, hardware status etc. The HMI module is built off the monitor which is responsible for collecting this information about the current state of the modules and sending the data to the HMI module to be displayed to the user. The HMI also has direct control of the different modules allowing individual modules to be turned on/off. It also provides debugging tools to track module issues.

5. Data Flow

Dataflow refers to the routing of information through the modules that make up the system. With regards to Apollo's self-driving software, there are both control lines and data lines. While the data lines convey bits from one device to another, control lines determine the direction of data flow, and when each device can access the system bus. The data flow can be summarized as follows:





.

Figure 2: Control and data flow between modules

Prediction receives obstacle data along with basic perception information including positions, headings, velocities, accelerations, and then generates predicted trajectories with probabilities for those obstacles. The planning module, which consists of a collection of main driving scenarios, receives the prediction output and the car's status. Then, based on these, the Control module uses different control algorithms to generate a comfortable driving experience (control commands: steering, throttle, brake) to the chassis. The CANBus module takes the control module outputs and collects the car's chassis status as feedback to control (chassis status and chassis detailed status). The Perception, Prediction, Planning and Control modules receive real time localization data from the Localization module through a data line.

At a high level, the system cycle sends data to the monitor module, which updates the Human Machine Interface. The monitor module contains system level software such as code to check hardware status and monitor system health. It also contains software monitors, to ensure each module is operating. As such, the monitor module sits on top of the system cycle from which it receives data and sends data to the HMI.

For more concrete examples on the data flow of the software system, refer to Use case Section.

6. Concurrency

As the vehicle moves through space it is constantly perceiving new obstacles and modifying the current route accordingly. That being the case, there is concurrency when looking at the different modules present in the system. As mentioned, a primary example of this is, as new obstacles are perceived, the prediction module takes that data as input and begins its computations, however, the perception module does not stop perceiving. Each module can be looked at as a separate process, running alongside one another. Data is streamed continuously between all processes, and they are executed concurrently.

The entire system is built utilizing the Nvidia CUDA toolkit, which provides a development environment for C and C++ developers to build GPU-accelerated applications. The advantages of leveraging a GPU for the vast amount of processing the Apollo system undertakes while plotting a route is that thousands of individual threads can be processed concurrently. This is key for Apollo as different threads can be spun to handle each of the different modules required operations.

7. Evolution

The Apollo platform has gone through a number of release cycles, starting with the release of Apollo v.1.0 in 2017 and ending with the release of the current version (Apollo v.7.0) in 2021. The earliest version of the Apollo autonomous driving platform (v.1.0) was highly limited in scope. It was capable of piloting a vehicle in confined "testing areas" using GPS, however it did not have the perceptual capacities necessary to navigate in an open-world environment, or to avoid potential obstacles. Architecturally, the open software platform for v.1.0 was relatively simple: it consisted primarily of a localization module that tracked the vehicle's location, and a control module that outputted vehicle commands (based on the processing of location information).

Apollo v.1.5 saw the introduction of a number of new components, including the perception module, planning module and map engine (the introduction of perception was also accompanied by corresponding hardware upgrades, such as the use of LiDAR sensors). At this point, Apollo could perform the basic task of driving in a fixed lane, but more complicated operations – like stopping at a light – were not supported.

After version 1.5, the high-level conceptual architecture of the Apollo open platform remained relatively consistent; subsequent releases focused primarily on developing and improving the existing components, rather than introducing new ones. Versions 2.0, 2.5, 3.0 and 3.5 improved the collision detection/avoidance capabilities of the perception and planning modules. Additionally, these versions also added a number of new traffic "scenarios" to the planning module

Versions 5.0, v.5.5, v.6.0 and v.7.0 continued to augment the Apollo's perception and planning components, adding deep learning models that aid in object recognition and planning evaluation. Moreover, in v.5.5 a new prediction module was also added, which interfaces with both perception and planning. This prediction module leverages deep learning technology to accurately predict the behavior of nearby objects, such as other cars or pedestrians. Since version

5.5, Apollo offers full “curb-to-curb” driving support – meaning that it can autonomously navigate standard traffic situations, and safely deliver a passenger to their destination.

8. Use Cases/UML Diagram

8.1 Use case #1: Red light detection

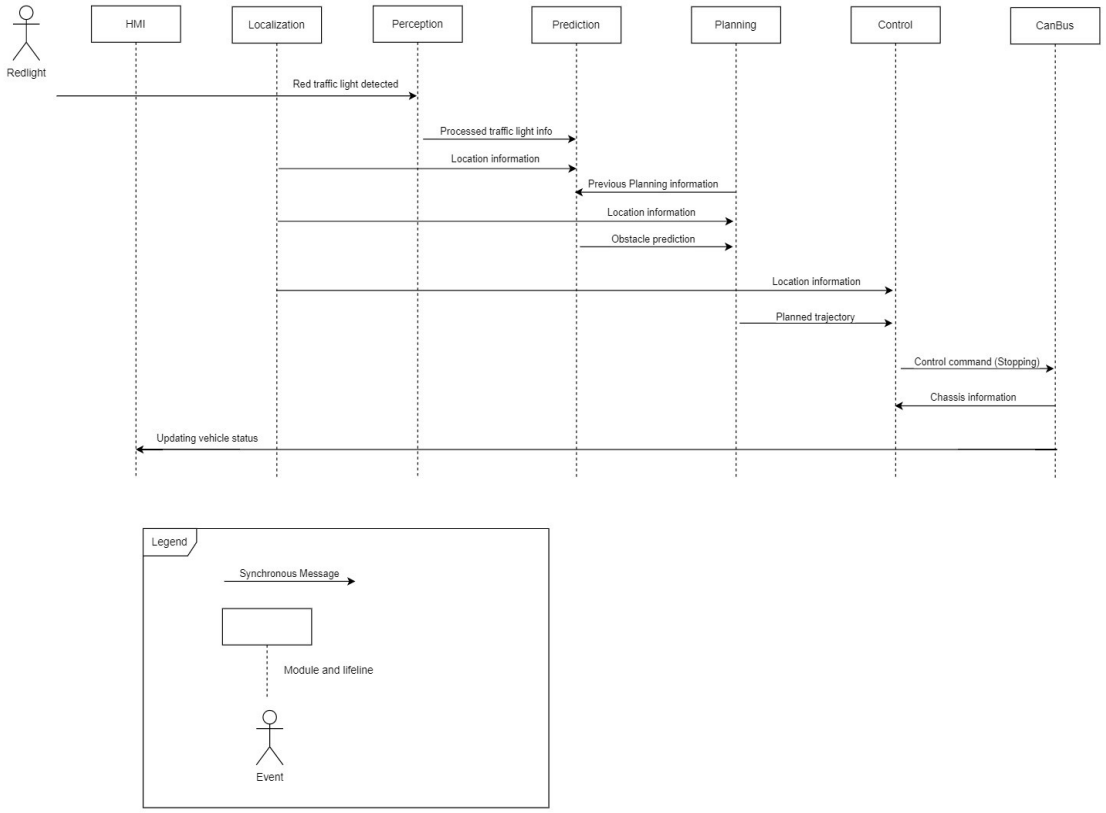


Figure 2: Use Case #1 Diagram

One of the most common use cases of the Apollo self-driving vehicle is to automatically stop at the red light and the process is described by the above sequence diagram. The red traffic light is first discovered by the camera system located inside the perception module then after some internal processing, the red color of the light is recognized, and the final light recognition is sent to the prediction module along with the location of the light to generate predicted trajectory and actions for the upcoming redlight. The predicted information is then sent to the planning module with the aim to generate a collision-free and comfortable trajectory which in this case is to get the vehicle to slow down and eventually come to a stop at the redlight location. After the vehicle successfully finishes planning and making the appropriate decision, it’s time to execute them through the hardware so it can physically stop, and this is done by the control module. The control module takes the location information and the previous planning trajectory as inputs and generates the desired output to achieve the planning result, in this case the control module will decide to have the hardware to let go of the throttle and apply braking. The decision made by the control

module is then sent to the Can Bus module for final hardware execution. And lastly, after the vehicle stops the current status of the vehicle is sent to the HMI module to be updated.

8.2 Use case #2: Maintaining safe following distance

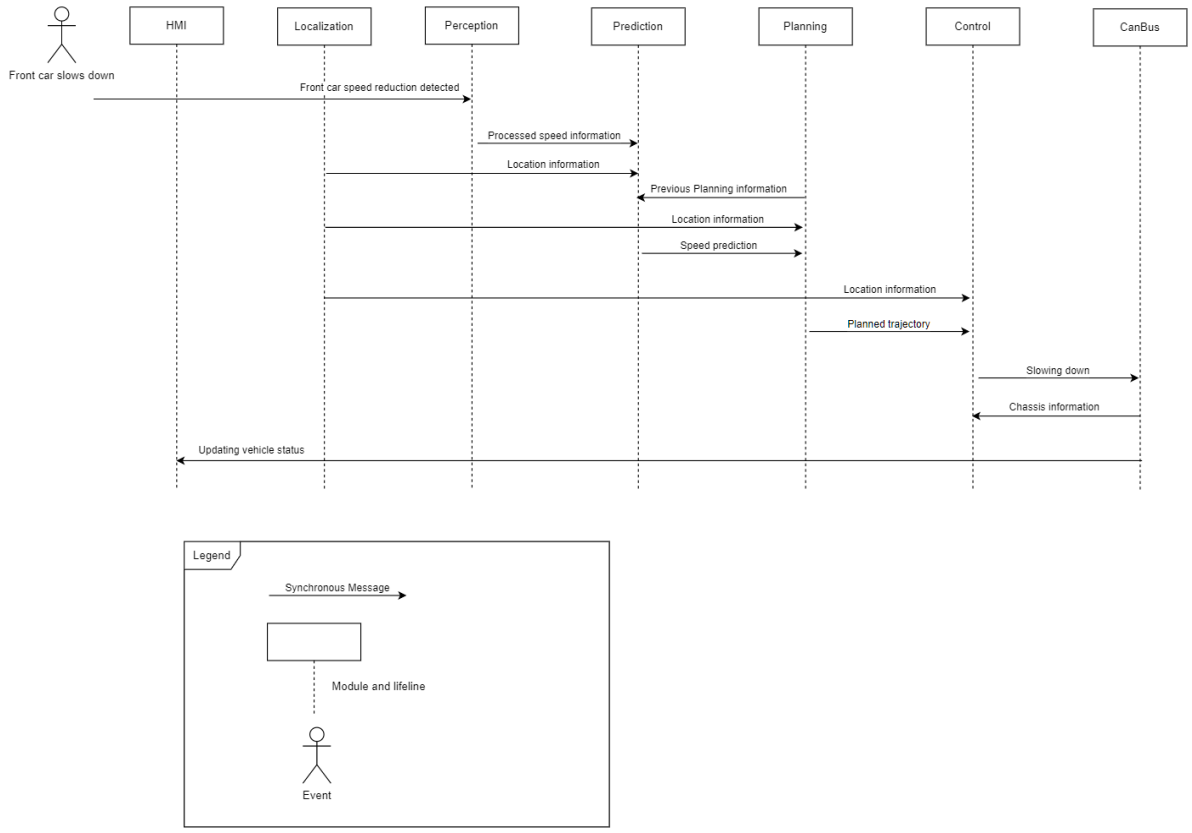


Figure 3: Use Case #2 Diagram

The Apollo self-driving vehicle monitors traffic in real-time and another very common use case is for the vehicle to keep a safe following distance to the front car and the data flow is exactly the same as the previous use case. Everything starts with the cameras in the perception module to capture and detect the front car and its motion. The captured information combined with the location information generated by the localization module is sent to the prediction module to predict the trajectory of the object, after some internal analysis the front car appears to be slowing down so the vehicle's own trajectory is predicted and sent to the planning module to plan out the trajectory. Like the previous case, the planned trajectory is sent to the control module to signal the CANBus module to get the hardware to execute the planned trajectory, which in this case is to slow down and maintain a safe following distance to the front car.

8.3 Use case #3: Navigation and change of destination

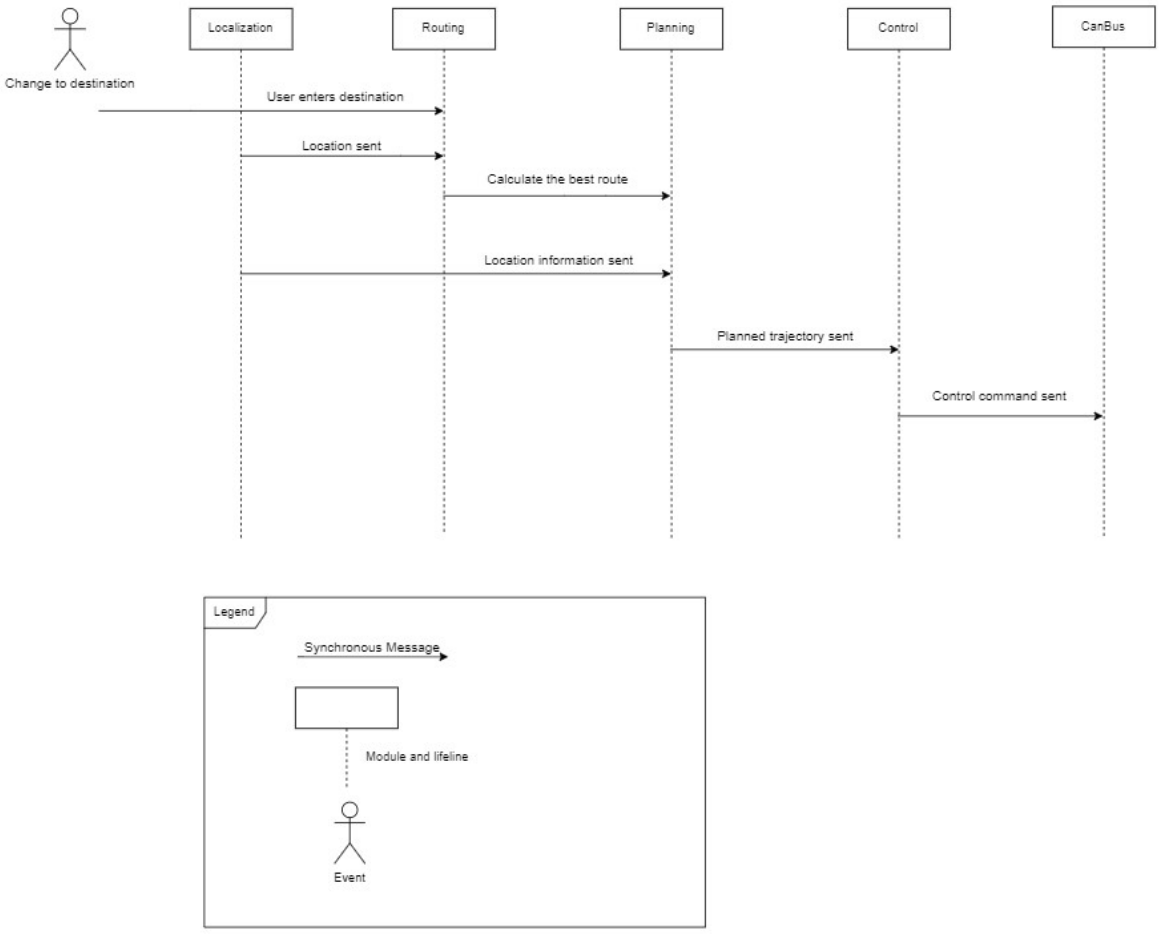


Figure 4: Use Case #3 Diagram

Users could change their destination at any time and the autonomous vehicle is able to generate the most optimal routes to accommodate the change request. This process is described by the above sequence diagram. After the users decide to change the destination and enter the new information, it is sent to the routing module for processing. The routing module analyzes the new destination location by comparing it with the map data and then it comes up with the best roads and lanes combination for the vehicle to follow. The combination is sent to the planning module to plan out the vehicle's optimal trajectory and at this point the flow will follow the previous ones, the planned trajectory is sent to the control module to signal hardware change with control commands. The control commands are then received by the CANBus module to execute the commands and the vehicle will now travel to the new destination.

9. Implications for Division of Responsibilities

Apollo is made available as an open-source platform available via the version control software engineering platform GitHub. As a result of Apollo's mission to provide a reliable ecosystem and solution to autonomous driving available to developers all over the world, they have had to put in place some regulations to ensure they are able to share resources and add improvements effectively.

As outlined in their governance structure, Apollo was created and built upon using a large dataset provided by their host Baidu, the largest internet search engine in China. Despite data from Baidu, Apollo relies on the contributions of individuals, especially data acquired. As such, Apollo pools together data provided to create a data collective that aggregates data collected and can be used to help build the platform. Apollo is modular, allowing participants to use as little or much as needed to propel their own autonomous driving efforts. Apollo has built in measures to guarantee security, handle failing points, and ensure reliability, using a platform such as GitHub also allows for version control and maintenance.

As a result of Apollos open-source format, Baidu has leadership over the project, allowing them to drive important decisions and apply safeguards of the architectural quality. Having the Baidu team lead important overarching decisions allows for the Apollo platform and community to continue to grow and remain active.

10. Conclusion/Lessons Learned

Throughout our report we have broken down and explored the conceptual architecture of the Apollo platform. Through our findings we have identified that Apollo uses an object-oriented programming style with process control elements. The main subsystems have been identified as well as their interactions and dependencies. The individual functionality, and interconnections of these subsystems is what allows Apollo to successfully operate and make developments in the world of autonomous driving. Throughout the process of analyzing and deriving the conceptual architecture our team has had to learn a lot about collaborating in a team. In addition, because of the lack of information available we have better understood how to derive and gain information from the documentation available.

References

Apollo. (n.d.). Retrieved February 16, 2022, from https://apollo.auto/index.html

ApolloAuto. (n.d.). *Apollo/modules at master · ApolloAuto/apollo*. GitHub. Retrieved February 16, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules

Guo, Q. (2020, April 29). Software system of autonomous vehicles: Architecture ... Software System of Autonomous Vehicles: Architecture, Network and OS. Retrieved February 21, 2022, from https://fisher.wharton.upenn.edu/wp-content/uploads/2020/09/Thesis_Nova-Qiaochu-Guo.pdf