CISC 322
Assignment 1: Report

Conceptual Architecture of Apollo

February 20, 2022

Team Artemis
Josh Otten (18jbo@queensu.ca)
Aleks Jugovic (16anj@queensu.ca)
Muyun Yang (18my24@queensu.ca)
Chong Guan (18cg14@queensu.ca)
Daniel Jang (17ddhj@queensu.ca)
Wooseok Lee (20wl9@queensu.ca)

Abstract

As a mature and open-source application for autonomous driving, Apollo has experienced six upgrades since the announcement of the first version in 2017, making it an ideal case study to gain a better understanding of software architecture. This report recovers the conceptual architecture of Apollo with emphasis on its subsystems and uses cases. The report remains within the domain of conceptual architecture and does not discuss the details of implementation. Due to being open source, documentation in the project's GitHub repository aided in our understanding of the operation of Apollo, allowing us to separate the system into many different subsystems. Based on the functions and interactions of each subsystem, it was clear that Apollo followed a Publish and Subscribe architectural style. With the cooperation of loosely-coupled components, Apollo accomplishes the goal of being a user-friendly platform to accelerate the development, test, and deployment of autonomous driving technology. As well, this architectural style makes it easy for developers to take part in using, testing and expanding the functionality of Apollo. The analysis of Apollo's conceptual architecture covers how the Publish-Subscribe style of architecture informs the interaction between the modules of Apollo as well as the categorization of subsystems.

Table of Contents

Introduction

The purpose of the report is to analyze the conceptual architecture of Apollo through online documentation and to demonstrate the process of deriving a conceptual architecture. Apollo is an open-source platform that allows for the development, testing, and deployment of self-driving vehicles. The report will focus on Apollo version 7.0.0, but features and modules from previous versions are still discussed if they have not changed in the new version.

We propose that Apollo builds on the Publish-Subscribe architectural style, which provides convenience in upgrading individual modules without needing to modify them all and supplying a window for any developer to take part in the development and make contributions. Both of these are crucial to the function of an open-source platform with a large number of individual components that interact. Due to the property of being low coupling, every subsystem has its own complete working pipeline and is not necessarily dependent on the inner workings of the modules it receives and sends data to. Overall, this makes the Publish-Subscribe style an ideal choice for Apollo.

The first section of the report provides the sequence of reasoning about how our group came up with the chosen style of architecture. The second section describes the derived conceptual architecture of Apollo. The third section is the description of all subsystems. For each module, we discuss their functionality and how interactions take place among these subsystems. The fourth section outlines two distinct use cases accompanied by two sequence diagrams to illustrate data flow and control flow among the subsystems in the architecture. Finally, we summarise what we have learned from the whole process of deriving the conceptual architecture.

Derivation Process

To derive the conceptual architecture of Apollo, our team used a three-step approach of individual brainstorming, consolidation of ideas, and finalization of the architecture.

To start, we all individually looked into online resources for generic autonomous vehicle software architecture to get a general idea of what it will look like. This provided us with a strong foundation on the terminology used in this domain and what the critical components are for a self-driving car. After doing this we had enough information to start looking for Apollo-specific information to build our own conceptual architecture. We found that there were not many resources online specifically for Apollo's software architecture, leading us to depend mostly on the information from the project Github. The software specifications README (https://github.com/ApolloAuto/apollo/blob/master/docs/specs/README.md) provided a lot of information for how the system works and which subsystems are a part of the system. The details of each of the subsystems were able to be derived from each of their individual READMEs, also on the project Github (https://github.com/ApolloAuto/apollo/tree/master/docs/specs).

Using all of this information, each team member had their own ideas for a general conceptual architecture, and we combined everyone's ideas into an initial draft. To do this we needed to identify which subsystems were the most important and what the interactions were between them. The team initially had an alternative conceptual architecture without the routing subsystem as we thought that it was not as significant to be mentioned in this report but as we

were creating the sequence diagram for the normal driving scenario, we discovered that the routing subsystem was important to show its dataflow and its function of being able to generate a high-level route for the car to take.

While doing more research into the documentation, we reached the conclusion that the overall architectural style being used for Apollo is Publish-Subscribe. Using our knowledge of this style, we were able to more easily identify how the subsystems would interact with each other, leading to the finalized conceptual architecture as seen in Figure 1.

Overview of Conceptual Architecture

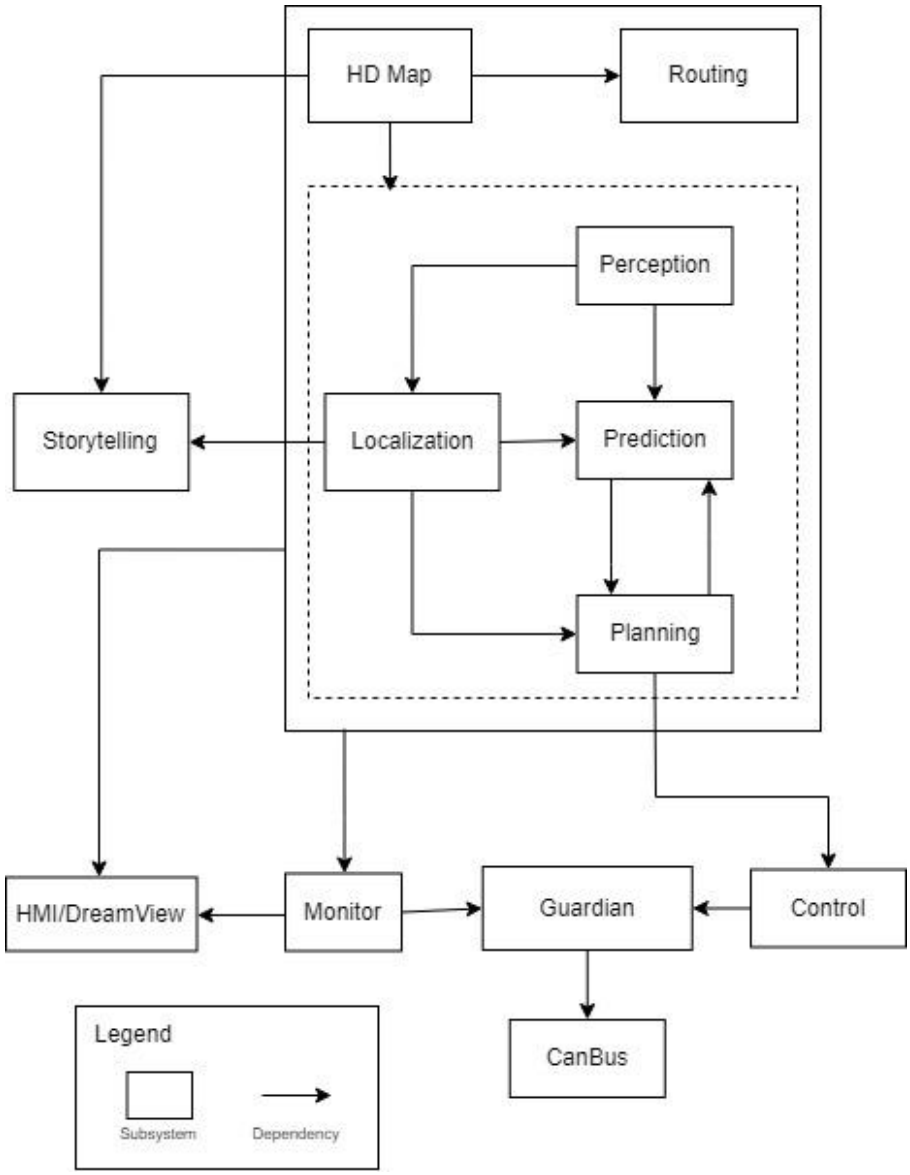

Figure 1. The proposed conceptual architecture of Apollo

The software architecture of Apollo is in the style of Publish and Subscribe, hereon referred to as Pub-Sub. Components, referred to as modules, publish messages to communicate with other modules that have subscribed to these messages, and react to incoming subscribed messages accordingly. Subscribed messages are treated as data input, while published messages are output. The Pub-Sub approach is suitable for this style of application due to its flexibility. Throughout the many versions of the Apollo system, it is afforded the ability to add or change modules as necessary, without majorly impacting other components. Due to the nature of Pub-Sub and modules waiting for subscribed messages before executing, there is no relevant concurrency in the high-level software architecture.

Apollo's software architecture can be split into two main categories of modules, which broadly handle input and output respectively. The input handling modules take the information gathered from the hardware such as various sensors and camera information gathered from the Perception module, and the vehicle information. Through the Prediction, Planning, and Localization modules, the Apollo system creates a trajectory for the vehicle to follow and information about the current state of the vehicle. The output handling modules execute the generated trajectory through the control and CanBus modules, while the DreamView and Monitor modules offer a level of interactivity to the user while providing relevant information about the car.

Subsystems

Perception

The Perception module identifies objects in the world surrounding the vehicle. It detects, classifies, and tracks obstacles, as well as predicts their motion. It also does lane detections. It combines inputs from many sensors, including 3 or 4 LIDAR scanners, 5 cameras, and 2 RADAR scanners. It then uses deep neural networks for object identification. It has a separate process for traffic lights, lane detection, and other objects. It outputs 3D obstacle tracks with information on their position, type, and velocity, as well as traffic light data.

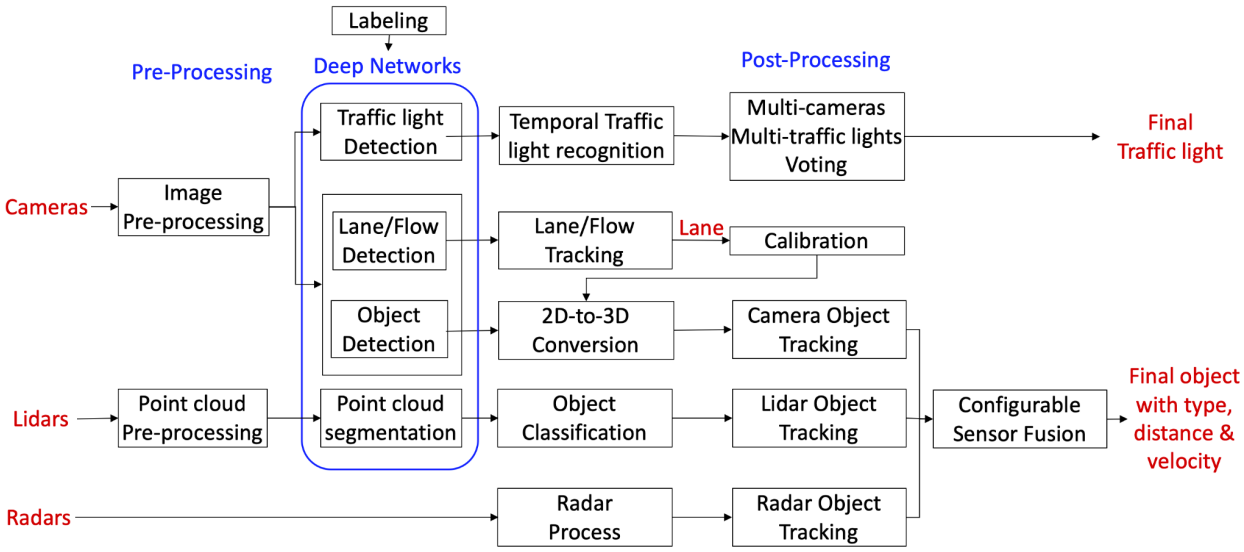

Figure 2. The detailed perception modules

Prediction

Based on the input of obstacles information from the Perception module, the input of localization information from the Localization module, and the input of planning trajectory of the previous computing cycle from the Planning module, the Prediction module predicts the behavior of all the obstacles detected by the Perception module. The Prediction module contains four main functionalities which feed into each other in order to predict its output, these being the container, scenario, evaluator, and predictor. Container stores the inputs data from the three subscribed modules, scenario analyzes the scenario the vehicle is currently in, evaluator predicts the path and speed for any obstacle based on its probability of occurring, and predictor generates the predicted trajectories for the objects.

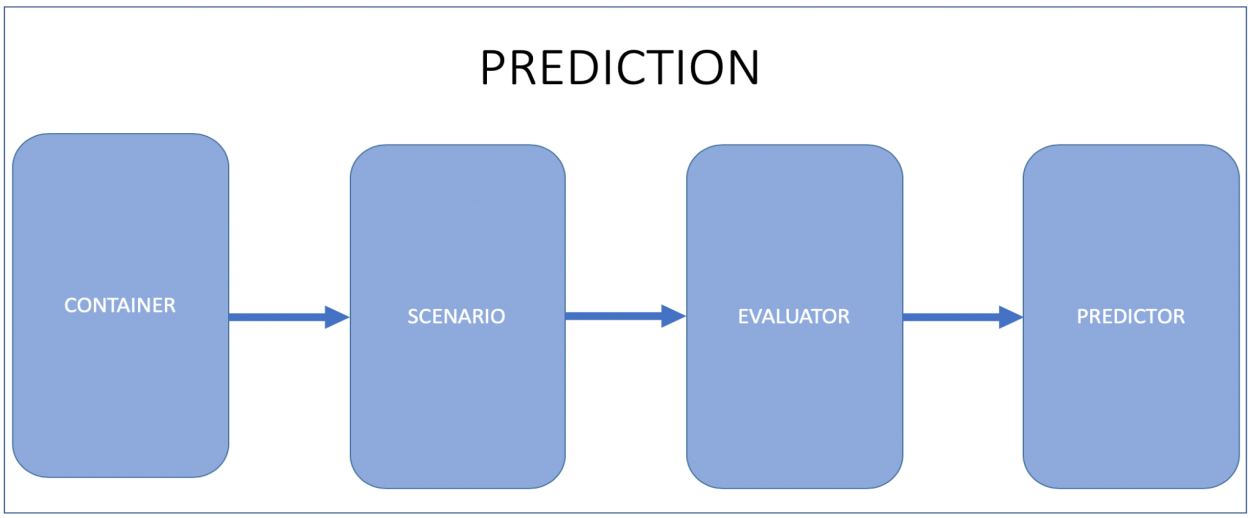

Figure 3. Four main functionalities of the prediction subsystem

Routing

When requested, the routing module generates the high-level navigation based on a routing topology. It takes in map data, and the start and end location of the routing request, and produces routing navigation information.

Localization

The localization module estimates the current location of the vehicle using a variety of data sources. It uses one of two methods to output the location information of the vehicle: RTK (real-time kinematic) and multi-sensor fusion. RTK based uses GPS and IMU (internal measurement unit) information as input to output a localization estimate. The multi-sensor fusion approach uses the same data as RTK along with LiDAR intensity and altitude cues to generate the estimate. The system is able to achieve centimeter localization accuracy in challenging scenes such as highways, tunnels, and city downtowns. This localization data is sent to the

Prediction and Planning modules to produce a more accurate model of the scene the vehicle is currently in.

Planning

The purpose of the planning module is to find a trajectory for the vehicle to take. It does this by taking as input the object detection from the perception module, as well as localization information and routing information. The planning module has multiple scenarios depending on the situation the car is in. These scenarios include when the car is at a stop sign, a traffic light, or a bare intersection, when the car is valet parking or pulling over, as well as a default (lane follow) scenario.

Lane Follow is the default driving scenario that includes but is not limited to driving in a single lane or changing lanes, following basic traffic conventions, or basic turning.

The stop sign scenario, the traffic light scenario, and the bare intersection scenario form the ultimate intersection scenario. This scenario's main actions include stop/approach, creep (move forward slightly), and driving through the intersection.

The stop sign scenario has two different scenarios which are the Unprotected scenario which deals with a crossroad having a two-way Stop and the Protected scenario which deals with a crossroad having a four-way Stop. The difference between the two is when the Unprotected scenario occurs, the system has to move forward slightly and gauge the crossroad's traffic density while the Protected scenario has to gauge the cars that come to a stop in the intersection to fully understand the queue of the vehicle's turn. The behavior of the solution to these scenarios is very human-like.

The traffic light has three driving scenarios which include Protected, Unprotected Left, and Unprotected Right. Just like the stop sign scenario, these sub scenarios are algorithms that are basically step-by-step processes of what a human driver would do to safely and smoothly pass through a traffic light.

The bare intersection scenario is to follow through an intersection without either a stop sign or a traffic light using the three main actions to solve the intersection problem as mentioned.

The parking scenario includes the Valet scenario where the vehicle safely parks in a targeted parking spot and the Pull Over scenario where the vehicle is parking to the side of the road. Apollo 5.5 added a new Park-and-Go scenario which follows up with the Pull Over scenario which allows the vehicle to safely exit from being parked on the side of the road.

Apollo 7.0 added a new dead-end scenario for the vehicle which allows the vehicle to perform a three-point-turn. Originally, Apollo handled all driving use cases as a single scenario but has since evolved to map each driving use case to its own scenario. By splitting up the scenarios into their own subsections, the system is more modular and changes in one scenario's behavior will not affect the way the other scenarios work.

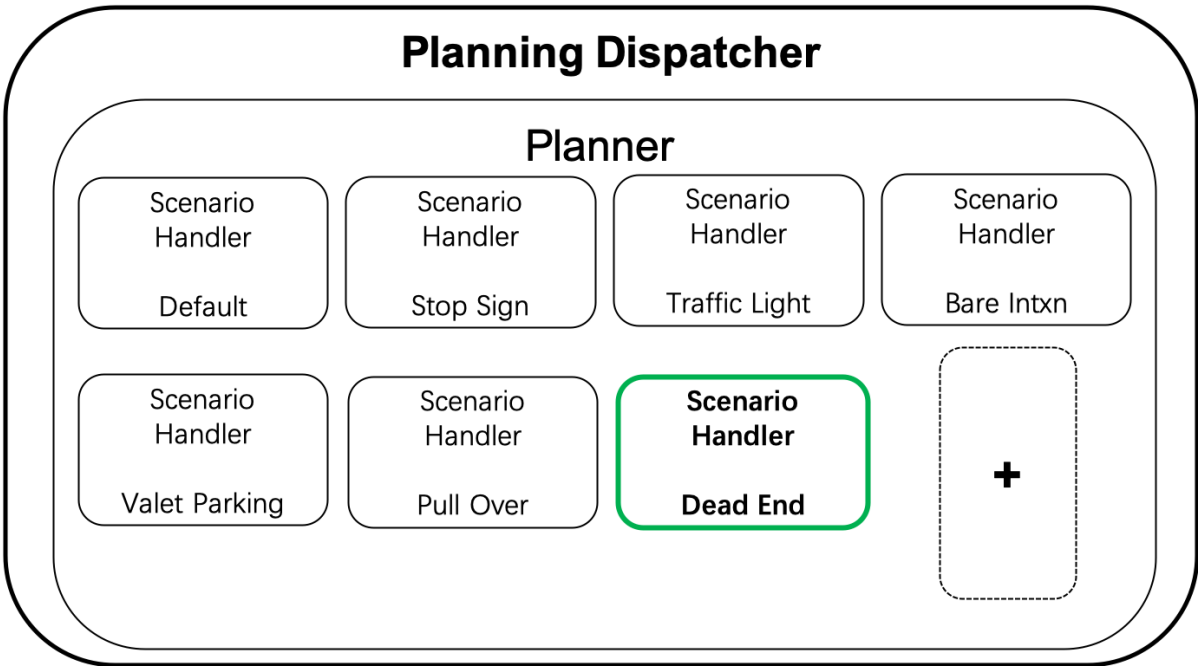

Figure 4. Planning Scenario Architecture

Control

The control module actually controls the car using the CanBus module and executes the planned trajectory. It controls the throttle, brake, and steering. The control module is able to account for the latency of sending the signals down the wire, as well as the control performance. As input, it takes the planned trajectory from the planning module as well as the car status and localization data, and it outputs control commands for the chassis via the CanBus.

CanBus

CanBus (Controlled Area Network Bus) is a vehicle bus standard that allows for communication with the hardware components of the vehicle. For Apollo, the CanBus module sends commands received from the Control and Guardian modules to the vehicle hardware (engine, brakes, throttle) to perform the desired action. When receiving a control command, it will send back detailed vehicle chassis information as a form of feedback. These control signals are blocked from being processed when the CanBus module receives messages from the Guardian module to perform an emergency action on the vehicle hardware.

HD Map

HD Map functions as a library, not through publishing and subscribing to messages. Modules query this module for ad hoc structured information about the roads.

HMI (DreamView)

HMI (human-machine interface), or DreamView module is a web application that allows the user to visualize data and interact with the vehicle. Using the interface, the user can view the

current status of the vehicle hardware and software modules, turn on and off individual modules and toggle the autonomous driving of the car. It receives input from all of the modules in the system to display this data and can send commands to the modules when toggling them.

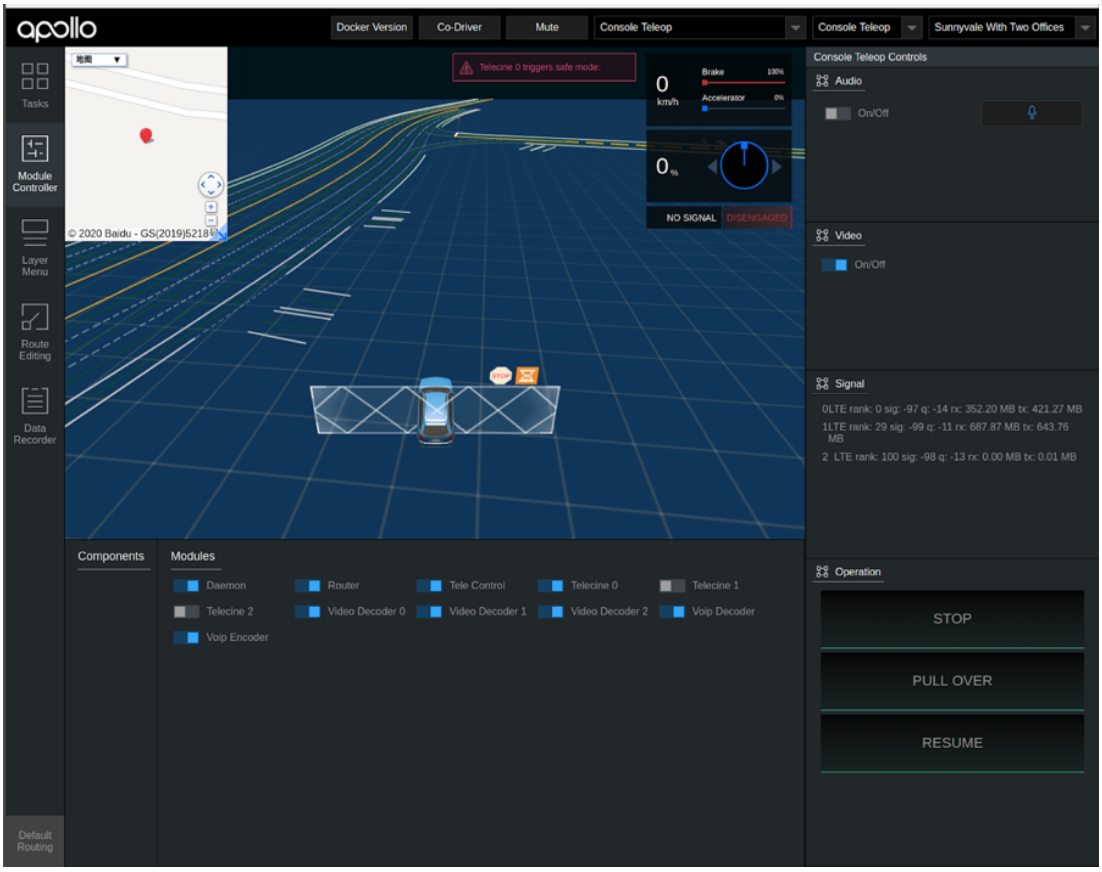

Figure 5. Example of DreamView UI

Monitor

The Monitor module is the surveillance system for all of the software modules of Apollo and the hardware of the vehicle. It will receive status data from all of the modules and send them to the HMI for the driver to view the status of the system. When the Monitor detects a failure in any of the modules or with a part in the vehicle, it will send a message to the Guardian module to perform an emergency stop of the car.

Guardian

Guardian is the safety module for the vehicle that acts as an activity center to make decisions from information being sent from the Control and Monitor modules. If all modules are working correctly the Guardian will continue to allow control signals to be sent to the CanBus to control the vehicle as normal. In the event of a failure of one of the modules detected by the Monitor module, the Guardian will send messages to the CanBus module to block control signals from being processed and bring the vehicle to a stop. It will use information from Ultrasonic sensors to decide how to stop the vehicle. If the sensors are working correctly then Guardian will inform

CanBus to bring the vehicle to a slow stop. If they are not working, a hard-brake instruction will be sent to the CanBus.

Storytelling

The Storytelling module is a high-level, global scenario manager. The module takes data from the Localization module and the HD Map module as input, and produces a ‘story.’ Story, in this case meaning a possible scenario the car can be in. This approach was taken to isolate complex scenarios, to avoid a sequential-based approach to handling these scenarios, and to avoid having certain scenarios impact other scenarios. The module creates complex scenarios that trigger multiple modules' actions as necessary.

Use Cases

Use Case 1: Normal Driving

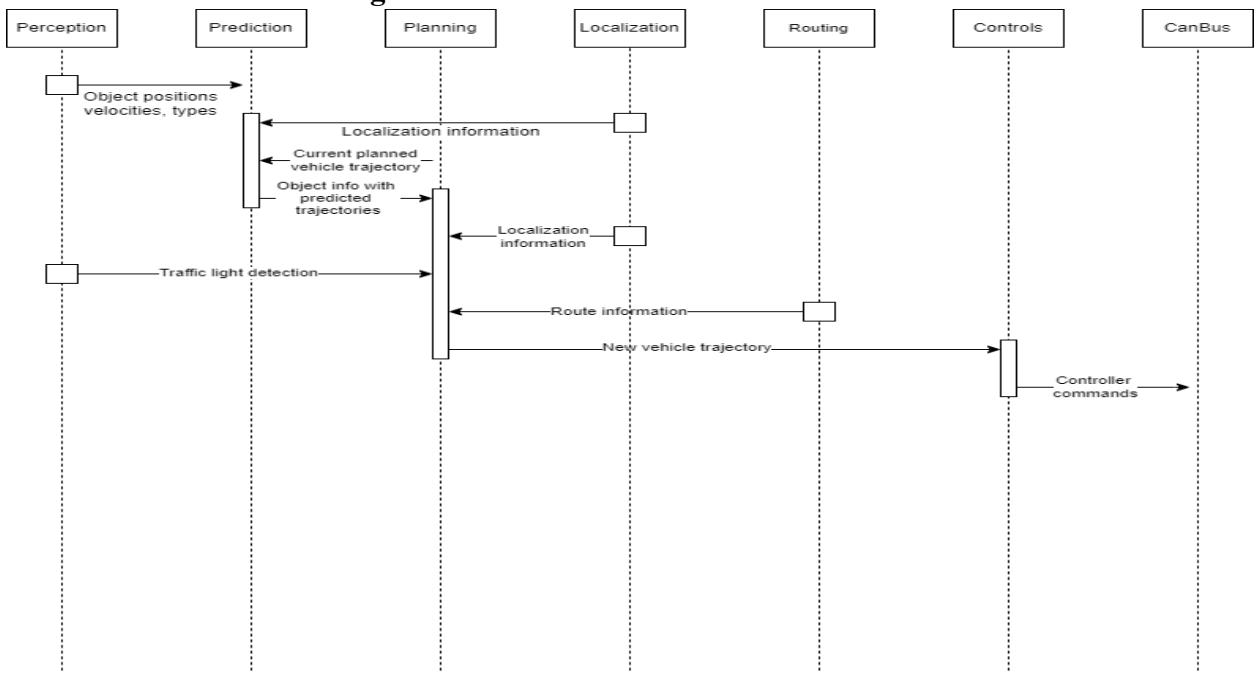

Figure 6. Sequence diagram for the normal driving scenario

This use case is for driving along in normal conditions. This use case was chosen because it would be the most common use case.

The perception module takes the sensor input and uses it to detect objects, including their types, positions, and velocities. This gets sent to the prediction module. The vehicle position information and the current trajectory are sent from the localization module to the prediction module. All of this information is used to predict the trajectories of the previously detected objects. This gets sent to the planning module. The planning module uses localization information as well as traffic light information from the perception module and route information from the routing module to update the planned trajectory of the car. This gets sent to the control module, which converts that trajectory to control commands that can be sent to the CanBus

system that actually enacts the controls when there is no failure found while passing through the Guardian module.

Use Case 2: Emergency/Failure

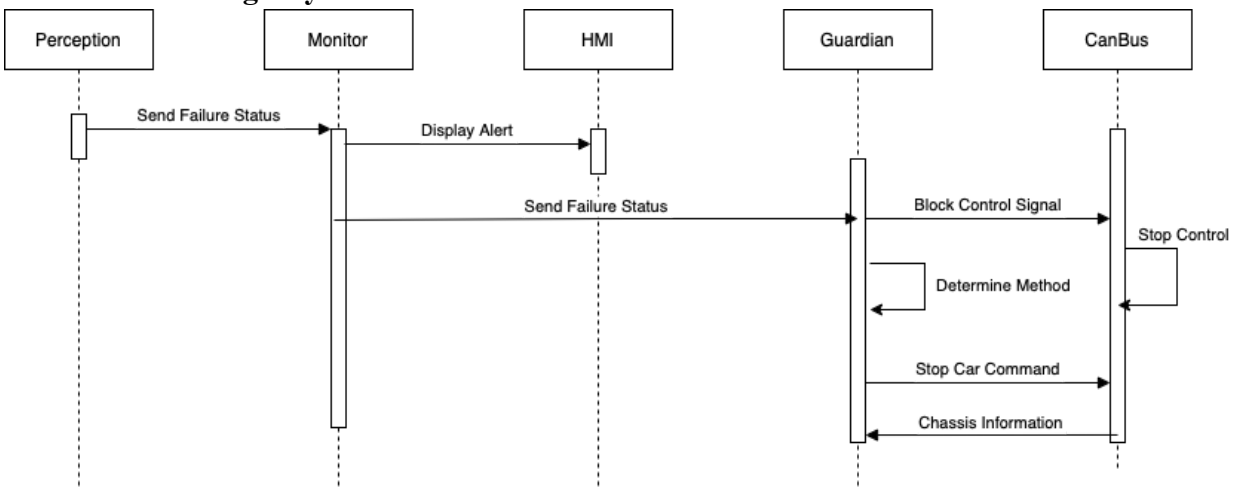

Figure 7. Sequence diagram for Emergency/Failure scenario

This use case is for when a failure is detected in a module or car component forcing the vehicle to make an emergency stop. Since this can occur in any module or hardware component, the case of the Perception module failing was chosen to illustrate one example of this.

The Monitor receives data input from all modules about their current status. This use case starts with the Perception module sending a failure message to the Monitor module indicating that it is no longer working. The first thing the Monitor will do is alert the driver by sending the error message to the HMI component, which will then convert this message to a readable format and display it to the driver. Then, the Monitor module will send a message to the Guardian module to indicate that there has been a failure. The Guardian module will immediately publish an alert about this failure, which the CanBus module is subscribed to. The CanBus module will handle this by calling itself to block actioning any incoming Control module commands. At the same time, the Guardian module will use internal data from sensors to determine which type of stop is required based on the error situation. After doing so, it will send a control command to the CanBus module to stop the vehicle in the correct way. The CanBus module will send back vehicle chassis information as feedback to Guardian. Finally, once the vehicle has stopped, the monitor will publish a message to the HMI module to indicate that the car has performed an emergency stop.

Limitation and Lesson Learned

Through analyzing the architecture of Apollo, our team gained an increased understanding of how large codebases can be structured. In the case of Apollo, it allowed for the exploration into the Publish and Subscribe software architectural style through the application of autonomous vehicles. Through this, the group also obtained knowledge on how autonomous vehicles can function. While exploring the release r7.0.0 of Apollo, previous versions were also explored in order to grasp how the repository evolved over time, and which software modules were modified or added between versions.

Early on in the process of completing the conceptual architecture deliverable, the task of analyzing Apollo's documentation seemed daunting. Perhaps more time than necessary was spent on parsing the repository and the provided references, leading to a general confusion that hindered progress. One of the main limitations that we have faced includes inconsistent documentation updates. For example, the Master branch (or the latest branch) (https://github.com/ApolloAuto/apollo/blob/r7.0.0/modules/planning/README.md) of the planning module inside the repository had very little information about the module with a few images of DreamView screen capture and a diagram without any explanation that caused much confusion between the team member. Later as we have progressed through the report, we've found out that the previous version branch called r6.0.0 (https://github.com/ApolloAuto/apollo/blob/r6.0.0/modules/planning/README.md) had an in-depth explanation of the planning module in detail that was missing from the current version branch. If we did not find this out anytime sooner, we could have well missed out on the crucial information from the repository due to Apollo's poor documentation updates. Although our group had issues with the repository, we learned to always have organized and up-to-date documentation to prevent people from experiencing any hardship that we have faced during the access of the documentation. We also learned that it is important to thoroughly review all project documentation before starting to do work. The team could have saved a great amount of time if we have checked previous version documentation in the beginning.

Conclusion

In conclusion, Apollo is built as a powerful and user-friendly platform to assist in the development of autonomous driving technology. Through our research and analysis, the structure of the software system follows that of the Pub-Sub architectural style, which provides it high flexibility in adding and modifying modules and providing users interfaces to design their own modifications of the Apollo system.

In the Apollo system, the software modules can be roughly classified into two groups, modules handling Input and modules handling Output. Input modules handle data from the environment of the vehicle, including the location and trajectory of the vehicle, as well as the obstacles around the car. Input from various sensors built into the vehicle gathers information on the environment of the vehicle. This data is then processed by the Localization, Prediction, and Planning modules to create a trajectory for the car to follow. This information is then sent to output modules. Relevant information about the vehicle is displayed to the passengers through the DreamView module. Instructions for controlling the vehicle are handled by the Control, Guardian, and CanBus modules to actually move the vehicle, as well as stopping the vehicle in the case of module failure.

Above all, as a mature architecture, Apollo may not have major changes in modules or add any modules in near future. This assumption is due to the core functionality of many of the modules, as well as their presence in many previous versions of the architecture prior to version 7.0.0. The development team of Apollo may focus on the optimization of algorithms and reinforcement learning models, and prepare more available scenarios in planning modules to enhance the user experience of autonomous driving.

Data Dictionary

Architecture: High-level structure and organization of subsystems in software.

Autonomous Vehicle: A self-driving car.

Subsystem/Module/Component: A self-contained system within the overall architecture. These three words are used interchangeably.

DreamView: A web application that helps developers visualize the output of other relevant autonomous driving modules.

HD Map: A high-precision map loader interface.

LIDAR scanner, RADAR scanner: Sensors in perception module to recognize obstacles and fuse their individual tracks to obtain a final tracklist.

README: A README is a text file that introduces and explains a project.

Naming Conventions

Publish and Subscribe Architecture (Pub-Sub): The Publish-Subscribe Architecture is an architecture style that allows messages to be broadcast to different parts of a system asynchronously.

Controlled Area Network Bus (CanBus): A subsystem of Apollo's overall architecture that accepts and executes control module commands and collects the car's chassis status as feedback to control.

Real-Time Kinematic (RTK): Application of surveying to correct for common errors in current satellite navigation systems.

Inertial Measurement Unit (IMU): IMU measures the specific gravity and angular rate of an object to which it is attached.

Global Positioning System (GPS): A global navigation satellite system that provides location.

Human-Machine Interface (HMI): A user interface that connects a person to a machine.

User Interface (UI): The point at which human users interact with a device.

References

[1] Apollo Auto website
https://apollo.auto/

[2] GitHub repository of Apollo Auto
https://github.com/ApolloAuto/apollo

[3] Apollo Auto. (2020, Oct 09). Inside Apollo 6.0: A Road Towards Fully Driverless Technology. Retrieved from
https://medium.com/apollo-auto/inside-apollo-6-0-a-road-towards-fully-driverless-technology-522b2b4295cc

[4] autopi. (2021, March 18). CAN Bus Protocol: The Ultimate Guide (2022). Retrieved from
https://www.autopi.io/blog/can-bus-explained/

[5] Guo Qiaochu. (2020, April 29). Software System of Autonomous Vehicles: Architecture, Network, and OS. The University of Pennsylvania, School of Engineering and Applied Science. Retrieved from
https://fisher.wharton.upenn.edu/wp-content/uploads/2020/09/Thesis_Nova-Qiaochu-Guo.pdf

[6] G. Wan et al., "Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes," 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018, pp. 4670-4677, doi: 10.1109/ICRA.2018.8461224.
https://ieeexplore.ieee.org/document/8461224

[7] Sagar Behere, Martin Törngren, “A functional reference architecture for autonomous driving”, Information and Software Technology, Volume 73, 2016, Pages 136-150, ISSN 0950-5849, doi: 10.1016/j.infsof.2015.12.008.
https://www.sciencedirect.com/science/article/pii/S0950584915002177