CISC 322
Assignment 2 Report
Apollo: Autonomous Driving Vehicles
Concrete Architecture Analysis
Monday, March 21, 2022

Group 1: Hextech
Benjamin Hui (Ben) (He/him) (18bh13@queensu.ca #20148554)
Fuwei Zhuang (Elina) (She/her) (19fz2@queensu.ca #20189056)
E Ching Kho (Noon) (He/him) (17eck3@queensu.ca #20118077)
Zewen Zheng (Zelvin) (He/him) (18zz114@queensu.ca #20150673)
Yixin Su (Allen) (He/him) (17ys114@queensu.ca #20108833)
Ruiyang Su (Amelia) (She/her) (18rs60@queensu.ca #20151860)

Abstract

The concrete architectural system of Apollo Auto was analyzed to dig out important information that might have changed when putting it beside the original conceptual architecture. The original conceptual architecture was transitioned from a Pipe and Filter focus to a publication-subscription style. The shift from a publication-subscription style was due to its nature of having modules subscribed to one another and would invoke events upon an announcement. When analyzing the concrete architecture of the top-level architecture and a chosen subsystem, we found numerous differences from the conceptual architecture but chose and selected a few to analyze based on its significance to the system. The top-level architecture had some unexpected dependencies as numerous other modules depended upon CAN Bus with its information of the vehicle's chassis. Task manager and Planning were also dependent on DreamView, which bridges the connection between the user interface and the system. We then chose to analyze one of the subsystems, Perception. Perception is responsible for the camera and object detection features of the system. Using its LIDAR, camera, radar, and map system, it can utilize the Pipe and Filter style to its full potential. Using a central body of information, the Base, it can store and redistribute information collected. The analysis of the two previous use cases mentioned in the conceptual architecture report was re-analyzed based on new findings of our concrete architecture and publication-subscription style information messaging.

Introduction

The conceptual architecture of Apollo Auto was discussed in the previous report, and this report discusses the importance of the proper analysis of concrete architecture in a system. As the conceptual architecture looks into the interactions of top-level modules and how they work together to complete the system, the concrete architecture focuses more on the actual implementation and execution of the conceptual architecture. The concrete architecture tends to stray from the original conceptual design by adding in required functions and modules to complete the tasks required of it. In this case, Apollo makes use of 14 essential modules along with the pub-sub architectural style to implement its complex autonomous driving systems. Reflexion analyses were completed to examine the differences and the rationale behind those decisions thoroughly. Important unexpected dependencies were reviewed and demonstrated the extent of changes made to satisfy the constraints of the problems posed. Two use cases were reviewed similar to those mentioned in the previous report to exhibit the scenarios in conjunction with the concrete architecture developed. Using the pub-sub messaging style, we could draw upon an accurate representation of the simulation in a real-life situation. The Apollo system was dissected in detail, and the findings are presented in this report.

Top-Level Analysis

Conceptual Architecture Update

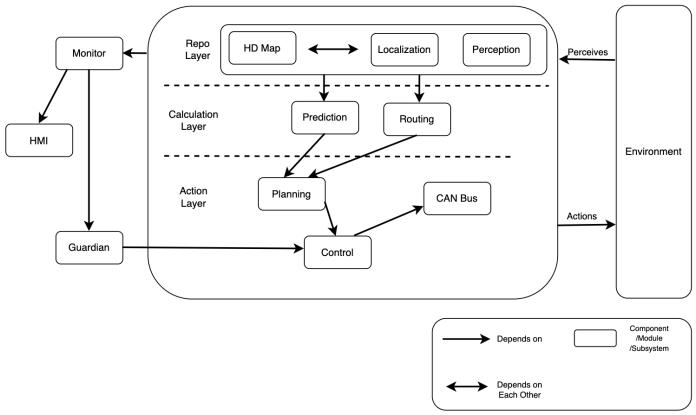

Figure 1. Original Top-Level Conceptual Architecture

We initially studied eleven subsystems of Apollo and came up with the idea that the main architectural style of Apollo is Pipe and Filter, with relevant architecture styles including pub-sub and layered. However, after carefully analyzing the source code on GitHub[2] and the graph visualization of pub-sub communication, we discovered that Drivers, Storytelling and Task Manager are also crucial components to Apollo since it relieves some of the weight on Planning and creates an actual way for software to use hardware. Furthermore, the whole system involves a loosely-coupled collection of components. Each component carries out some operation and may enable other operations instead of simply performing a series of independent computations. Hence, the main architecture style should be Pub-Sub. In addition, some subsystems, such as Dreamview and Perception, may also include Client/Server and Pipe and Filter styles.

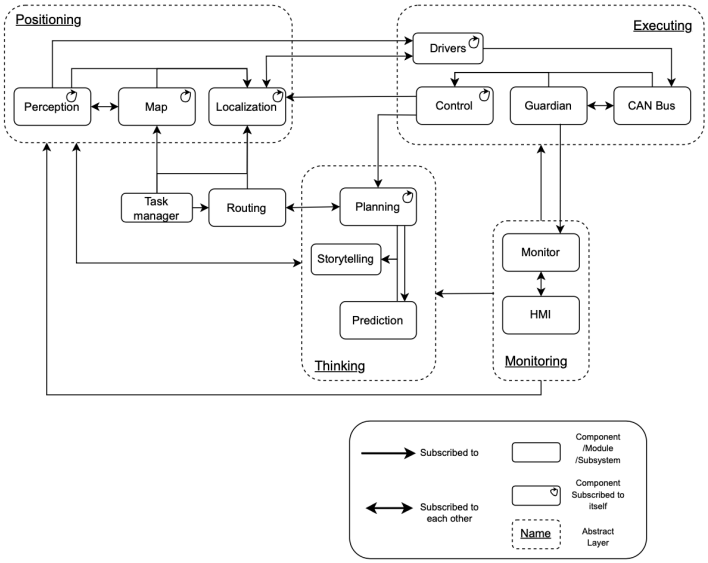

Figure 2. Updated Top-Level Conceptual Architecture

Derivation Process

After studying the source code on GitHub, we have gained a great deal of knowledge of Apollo's Architecture and confirmed many of our conjectures in the initial conceptual architecture. Furthermore, with the help of the source code analysis program Understand and the Graph Visualization of Pub-Sub Communication provided on onQ, we have updated our conceptual architecture and derived the concrete architecture with new dependencies using logical reasoning. In addition, we studied the inner architecture of the Perception subsystem. Finally, we came up with the appropriate ambiguous use cases and sequence diagrams for the concrete architecture.

To better analyze how each module operates, communicates, and completes the process of auto-driving, we first analyzed each module on what information they would send and which modules they would depend on[3], then drew an initial concrete architecture. However, the whole picture looks complicated due to the numerous dependencies, so we classify and integrate modules into groups. After combining the Data & Control Flows, Dependency Graphs, and the derived architecture styles, we divided Apollo's architecture into Thinking, Positioning, Executing, and Monitoring, with Routing and Task Manager stand-alone (**Figure 4**)[1].

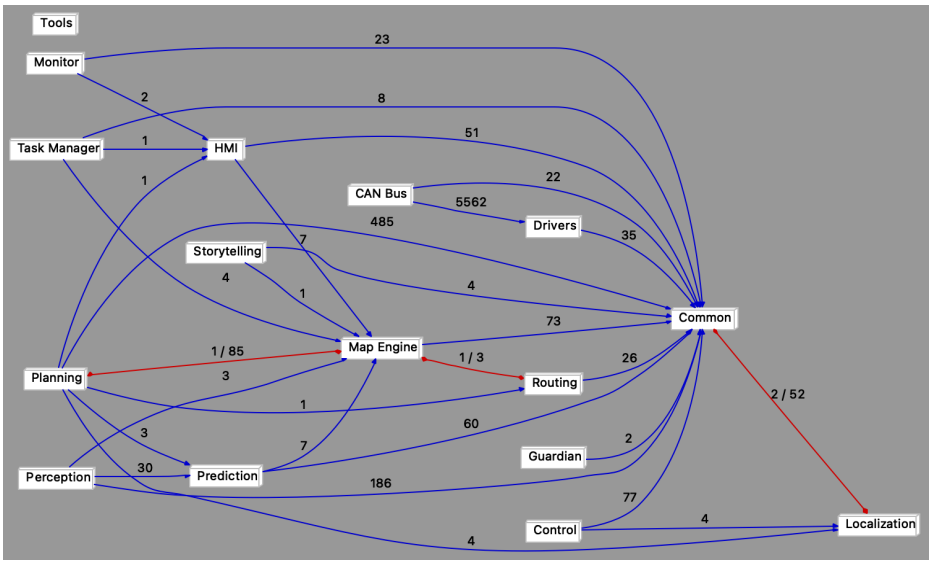

Figure 3. Understand Dependency diagram (Architecture name: ConcreteApollo)

Concrete Architecture

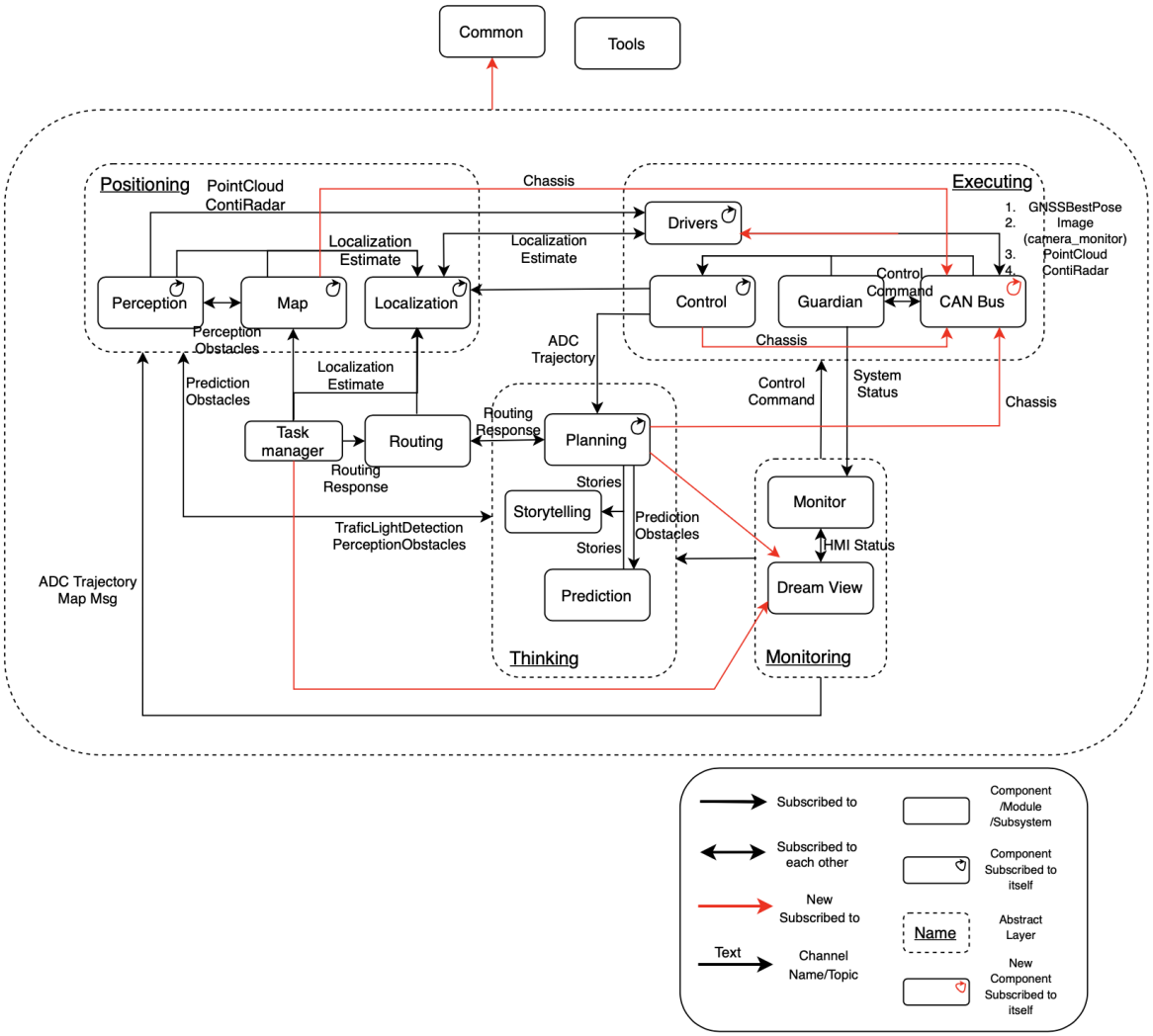

Figure 4. Top-Level Concrete Architecture

Subsystems and their Interactions

Our concrete architecture consists of 14 subsystems, and we divided them into four groups: Positioning, Thinking, Executing, and Monitoring, with Task Manager and Routing modules as stand-alone. Positioning is to collect location and road data within a certain radius of the vehicle. Thinking is where it processes all the thinking, like identifying objects and planning out a collision-free trajectory. Executing is where all the communication with hardware occurs, like sending control commands to the vehicle to execute or letting other modules know how to use the hardware (e.g., camera). Finally, monitoring monitors the health status of all the modules and provides information to the users to view the system.

Positioning: Perception, Map, and Localization

This group contains three modules that help the vehicle find its position on the road and earth while detecting nearby objects. The perception module uses multiple cameras, radars (front and rear), and LIDARs to recognize obstacles and fuse their tracks to obtain a final tracklist. The obstacle sub-module detects, classifies, and tracks obstacles. This sub-module also predicts obstacle motion and position information (e.g., heading and velocity). Finally, we construct lane instances for lane lines by postprocessing lane parsing pixels and calculating the relative lane location to the ego-vehicle. The map module is to get navigation information. It sends localization estimate information to the Localization module. The essence of localization is the perception of the surrounding environment, feature extraction, feature matching, and finally, getting an accurate positioning result. There are two ways to help the vehicle locate, one is RTK, and the other is the multi-sensor fusion.

Thinking: Planning, Storytelling, and Prediction

This group contains three modules to provide a "collision-free" trajectory for the vehicle according to the information collected from other modules. The Planning module receives data from the Perception, Prediction, Map, Routing, and Localization modules. Its goal is to provide a "collision-free" trajectory for the vehicle to navigate through when it encounters various situations. Storytelling is a global and high-level Scenario Manager to help coordinate cross-module actions. In order to safely operate the autonomous vehicle on urban roads, complex planning scenarios are needed to ensure safe driving. This module creates stories of complex scenarios that would trigger multiple modules' actions. The main advantage of this module is to fine-tune the driving experience and isolate complex scenarios by packaging them into stories that can be subscribed to by other modules like Planning, Control, etc. The Prediction module studies and predicts the behaviour of all the obstacles detected by the perception module. Prediction receives obstacle data, and basic perception information, including positions, headings, velocities, accelerations, and then generates predicted trajectories with probabilities for those obstacles.

Executing: Drivers, Control, Guardian, and CAN Bus

This group contains four modules for the subsystems to “run” the vehicle. The Control module uses different control algorithms to generate a comfortable driving experience based on the planning trajectory and the car's current status. The Control module can work both in normal and navigation modes. The Control module receives data from the Planning trajectory, Car status, Localization, Dreamview AUTO mode change request, and outputs Control commands (steering, throttle, brake) to the chassis. Guardian is the safety center of the whole system. Guardian receives data flows from Monitor to ensure all other subsystems are working well. If the data from the Monitor is detected failure, Guardian will prevent the CAN Bus from executing further actions and cut down the connection between Control and CAN Bus. Then Guardian will take actions based on the response of the Ultrasonic sensor and the HMI (Dream View) to choose one of the three preset executing modes. So we know that the CAN bus accepts and executes control

commands and collects the car's chassis status as feedback to control. Moreover, Drivers can get PointCloud RawData. Driver module receives data from Monitor, perception, and localization.

Monitoring: Monitor and Dream View
This group contains two modules to monitor, ensuring all the modules are working well and helping developers visualize the output of other relevant autonomous driving modules. The monitor receives data from Perception, Prediction, Planning Control, HD map, Localization, and other various modules, then transfer the data flows to Dream View for the driver to view the current status. The monitor's function is to supervise the hardware and software components to ensure that all the modules are working without any issue. If it detects a problem, the monitor will alert Guardian for further steps to prevent a crash. In addition, Dreamview or Apollo's HMI module provides a web application that helps developers visualize the output of other relevant autonomous driving modules, e.g., the vehicle's planning trajectory, car localization, chassis status, etc. For example, the Dream View module receives data from Localization, Chassis, Planning, Monitor, Perception, Prediction, Routing and outputs a web-based dynamic 3D rendering of the monitored messages in a simulated world.

Task Manager and Routing
Task Manager receives localization estimate information from the localization module and routing response information from the Routing module. Routing is a module that generates high-level navigation information based on requests. It receives data from the Map module and outputs routing navigation information and paths to Planning to generate the collision-free trajectory.

Top-level Reflexion Analysis

When analyzing the dependencies of the Apollo, we found several discrepancies between our conceptual architecture and the dependency graph generated by Understand and the pub-sub communication model. For example, we initially thought the CAN Bus and Dream View components were only for executing commands and displaying data. Nevertheless, we discovered that many subsystems depend on them.

Control, Map, Planning → CAN Bus
Surprisingly, we found several modules depend on CAN Bus as the dependencies do not appear in the dependency graph on Understand but in the pub-sub communication model. We initially thought that CAN Bus was just a module to receive and execute control commands, but after reading the source code, we found that Map, Planning, and Control modules all depend on its output: Chassis Status. This data is needed when an auto-driving vehicle is trying to integrate reverse gear.

CAN Bus → Drivers
CAN Bus module is a bridge between the car and automatic driving software. Through Drivers, the car body information is sent to Apollo upper software, and control commands are received and sent to the car wire chassis to achieve the control of the car. The three functions of CAN Bus, sending), receiving (CanReceiver), and message management, are implemented in Drivers. Therefore, CAN Bus depends on Drivers to initialize the CAN client.

Task Manager, Planning → Dream View
Task Manager and Planning depend on Dream View's HMI Status. We found one dependency in both Planning and Task Manager modules from Understand, which related to the map service in Dream View, e.g., detecting where the vehicle was in a circular route.

Subsystem Analysis (Perception)

Conceptual View

Main Style: Pipe and Filter
The subsystem Perception utilizes the pipe and filter style due to its nature to retrieve data that passes through filters and ultimately compile them into accurate information that can be transferred to other subsystems. Data is the initial data is then passed onto the three modules Lidar, Camera, and Radar. The three modules collectively gather information on obstacle recognition, traffic lights, lane line recognition, and tracking. The information is then compiled together into fusion, where the data is processed into understandable information. As perception is responsible for understanding the environment around it, it uses the pipe and filter style to put data through a series of filters to create an understanding of its surroundings.

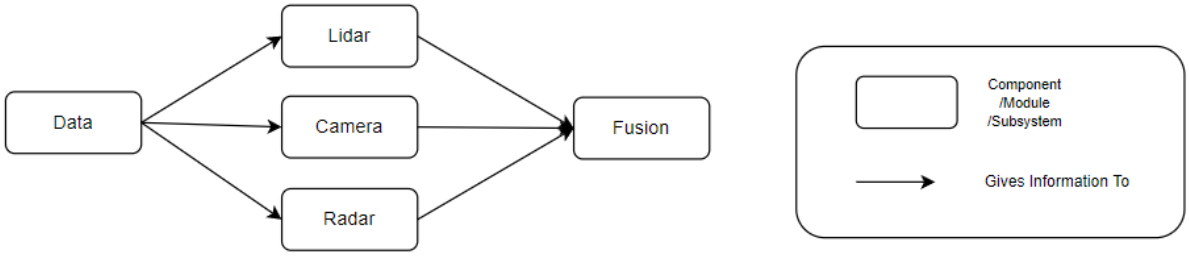

Figure 5. Perception Conceptual Architecture

Concrete View

The concrete architecture of the perception subsystem is modelled in Figure 6 following the pipe and filter style. The perception module initially starts with the production module which contains the starting data and is passed on to inference and onboard. Inference utilizes deep learning models to assist in identifying an object. Onboard receives data from production and map while it processes sensory information as it passes it onto the Lidar, Camera, and Radar. These three modules are responsible for the filter and implementation of the preprocessing of object

recognition, lane recognition, traffic light detection, and tracking of objects. The data is then compiled together into fusion, where it merges and combines all the data. The lib, common and base are responsible for providing the basic calculations and algorithms that are needed by other modules inside the perception subsystem.

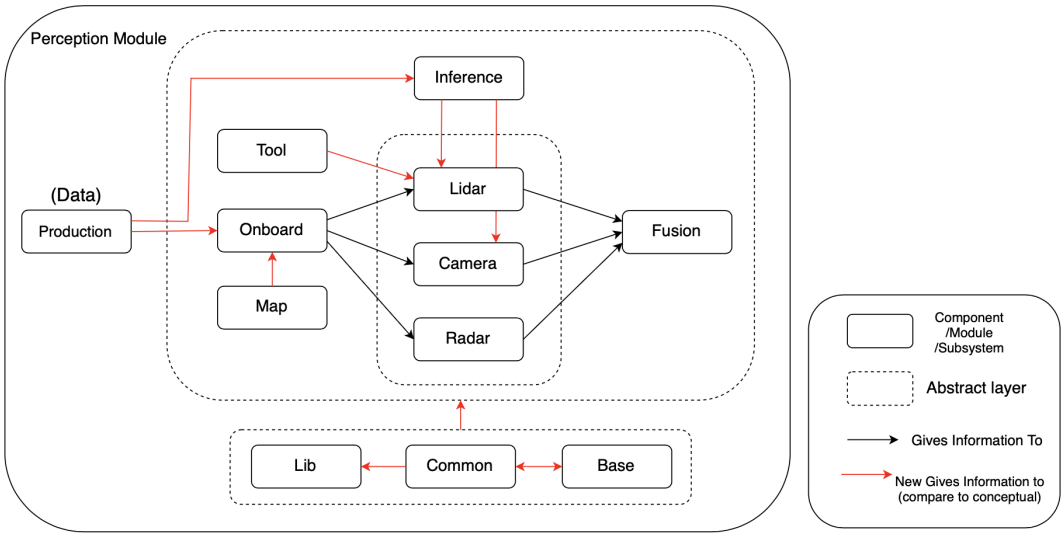

Figure 6. Perception Concrete Architecture

Inner Architecture and its Interactions

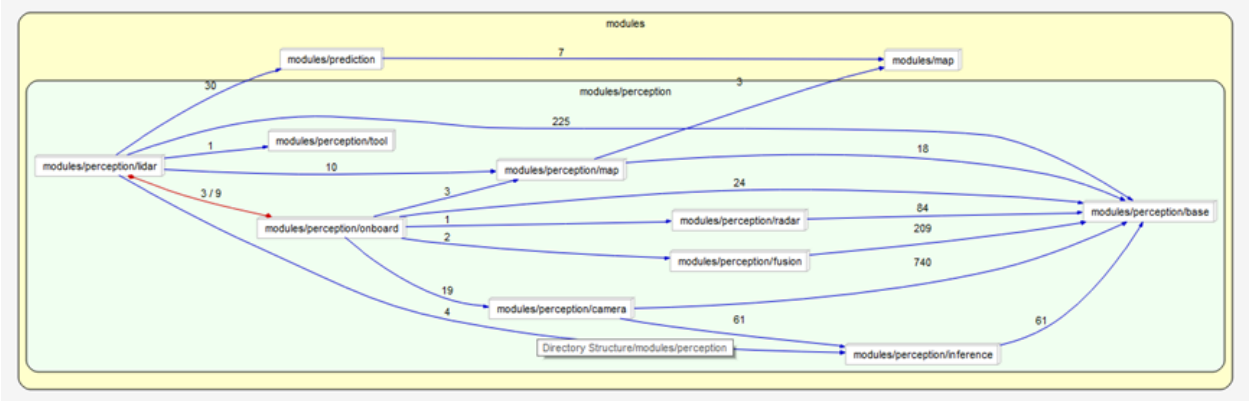

Figure 7. Understand Dependency diagram (Perception)

LIDAR is connected with Tool, Map, Onboard, inference, and base. Moreover, LIDAR is based on the Mask-Pillars. The role of LIDAR is to accurately locate the position of vehicles on the map to make a judgment on the following driving. Therefore, in perception, except for the base, all parts requiring LIDAR are used to predict future actions in unmanned driving. The most frequently used part of LIDAR is prediction, with 30 functions. It is the most core algorithm part of the whole prediction system. The Tool checks the validity of the data returned by the hardware. LIDAR is also used in Map many times, ten functions. To ensure the accurate location of cars, LIDAR is also needed to help Map obtain a more accurate location. Onboard and LIDAR

transmit data to each other, and Onboard gets data for later use. The Inference is not used frequently, but it also carries out important prediction calculations. The difference between this part and prediction is that the prediction data made by prediction is transmitted to external programs, while the Inference is transmitted to internal submodule programs.

Onboard is connected with Map, Radar, Fusion, Camera, Inference, and base. Map analyzes the data provided by LIDAR, Onboard and HDMAP. Onboard is a part that filters information needed by other programs. This is to obtain the data through algorithms to make the map display higher quality to reduce the rate of positioning errors and the incidence of car accidents in unmanned driving. Since Data between Onboard and LIDAR is transmitted to each other, Data obtained from Tool can also be retrieved by Onboard. Radar scans the surrounding objects to obtain the shapes and positions of various objects near the vehicle. However, such data alone is not enough. Therefore, radar needs to obtain more data from Onboard for analysis to have more accurate positioning and better driving judgment in unmanned driving. The Camera is based on SMOKE. The camera as another external device is similar to radar except that the function of the Camera is to obtain nearby photos and transmit them to inference for algorithm processing to obtain a judgment of the nearby environment. All of these parts except the tool are eventually connected to Base for final data combination and analysis. Fusion works in conjunction with the primary sensor frame to output information required from onboard. It also contains sensory data management systems to filter camera inputs according to the appropriate sensory data.

Among these perception parts, two can transmit data, namely LIDAR and Map. LIDAR transmits the acquired data to Prediction. This part exists side by side with the whole perception, and its function is to calculate various operations of the vehicle in driving and make the best prediction. LIDAR can obtain many data from external sensors, and these valuable data are most needed for prediction. The map in Perception transmits the positioning data obtained by itself to the general map program, and only after all data is summarized can more accurate positioning be obtained.

Common: common is where commonly used files are accessed and contain gflag files. gflag files are typically used for debugging and do not hold much significance to our research purpose. In addition, Common contains files like geometry calculations, graphical matrices, algorithms, image processing, sensor management, and even cloud processing files. Common depends on lib and base for its information which is then redistributed to other modules that depend on it such as LIDAR, onboard, map, radar, camera, fusion, inference, and also lib.

Lib: lib is the module that holds configuration files, registrants, mutex locks, and threads. The library is responsible for providing the proper threads and concurrent mutexes for other modules, including LIDAR, Camera, Map, Fusion, and Common. As these modules require the proper

system functionality, the library compensates by containing the implementations of these requirements.

Base: Base is the module containing basic information necessary for other modules to function correctly. The object detection, distortion, and image technology are supplemented in this module and object pools and omnidirectional models.

Subsystem Reflexion Analysis

When analyzing the conceptual and concrete architecture of the perception subsystem, there were some unexpected dependencies while performing the reflexion analysis. We will discuss some of the more important unexpected dependencies.

Production → Onboard & Map → Onboard

Onboard is a module that is required to initialize the data that is given to Lidar, camera and radar. This unexpected module is critical to the function of the perception subsystem. As it acts as a filter for the preprocessing modules Lidar, camera, and radar, its dependency was unexpected as the conceptual architecture did not predict an additional filter. Originally, map was thought of to directly communicate to fusion as it is the core module that is responsible for the compilation of data. However, it works in conjunction with onboard to provide data to the preprocessing layers as it provides onboard with additional information.

Inference → Lidar & Inference → Camera

Initially, Lidar and camera were thought to function with only information given by its previous module (onboard). However, inference is essential to the operation of Lidar and camera as these two modules are responsible for object recognition. Lidar is mainly focused on obstacle recognition and tracking while camera is mainly focused on traffic lane recognition and traffic light detection. Because of this, the deep learning models of inference aid with and is able to speed up the recognition process exponentially.

Use Cases

Use Case 1: *Automatic Rerouting (e.g., encounter a road closure)*

The first use case is when the system encounters an obstacle/wall and needs to turn the car around and reroute its navigation to continue on its journey. The system always has a designated driver module that supplies drivers to modules that require the information. Perception then is responsible for extracting information and provides prediction and map with the information that it has detected an object in its view. The Prediction then accurately pinpoints and calculates the prediction of the object's movement to the planning module. The planning module then receives data of its environment by obtaining the routing response, map information, localization

estimates from its respective modules[4]. At this time, routing is responsible for producing its results and response from the situations given to planning and the task manager. The Map will distribute its share of information of the current map it has generated to planning and localization for a better localization estimate. CAN Bus distributes information regarding the vehicle's chassis and receives commands from control and guardian regarding what actions the vehicle should perform or execute. As the monitor receives status updates from modules, it sends data to the control to inform it of the system status and module information.

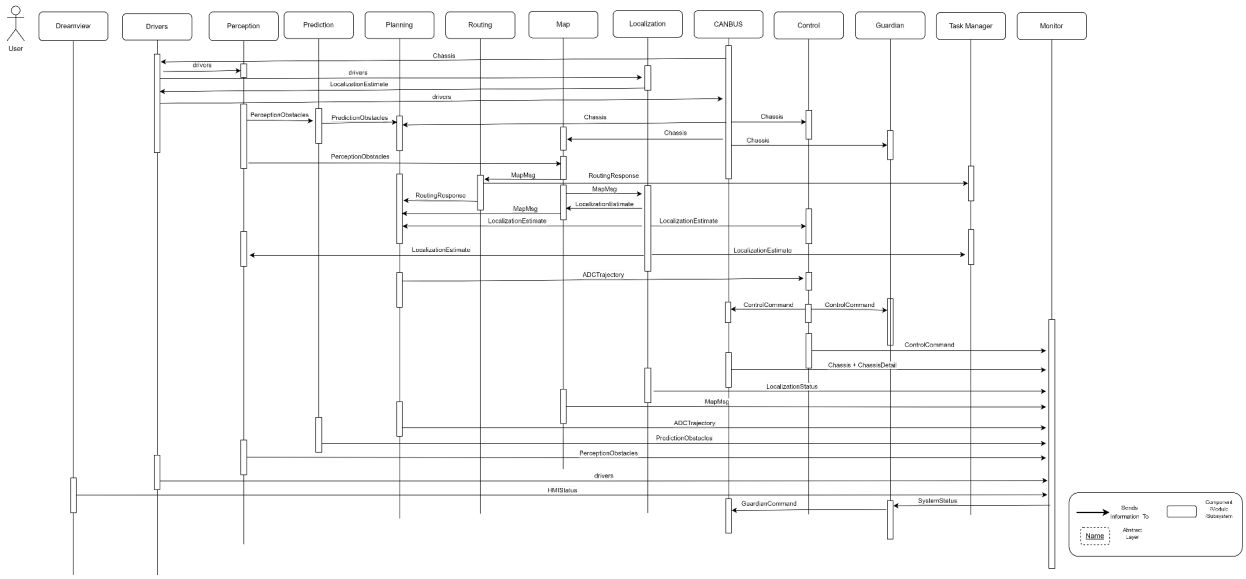

Figure 8. Sequence Diagram of Use Case 1: Automatic Rerouting

Use Case 2: *Automated Valet Parking*

The second use case is Valet Parking, Apollo provides Automated Valet Parking which allows the vehicle to park automatically. Localization gives the current position of the vehicle to find a parking space. The Perception module detects obstacles by looking around (UPA Ultrasonic Radar, APA Ultrasonic Radar, fisheye camera), forward perception (Monocular camera), backward perception (Monocular camera). Prediction receives obstacle data and basic perception information and generates predicted trajectories with probabilities for those obstacles. The Map module also receives information from the Perception module then sends the current information it has generated to the Planning and Localization module. The Planning module then uses the information collected from perception, prediction, and localization to provide a “collision-free” trajectory for the vehicle to navigate through when it encounters various situations. Control modules get the input information from planning, then give tasks to routing to output the desired map. Before the control signals reach the CAN Bus, the monitor will keep track of all the modules and ensure they are working without any issue. In a module or hardware failure, the monitor alerts Guardian, preventing Control signals from reaching CAN Bus and bringing the vehicle to a stop. Dreamview or Apollo's HMI module provides a web application that helps developers visualize the output of other relevant autonomous driving modules such as the

vehicle's planning trajectory, car localization, chassis status, etc. The Dream View module can receive HMI status from the monitor module. CAN Bus will execute the commands if no failure occurs and the vehicle finishes the rerouting process.

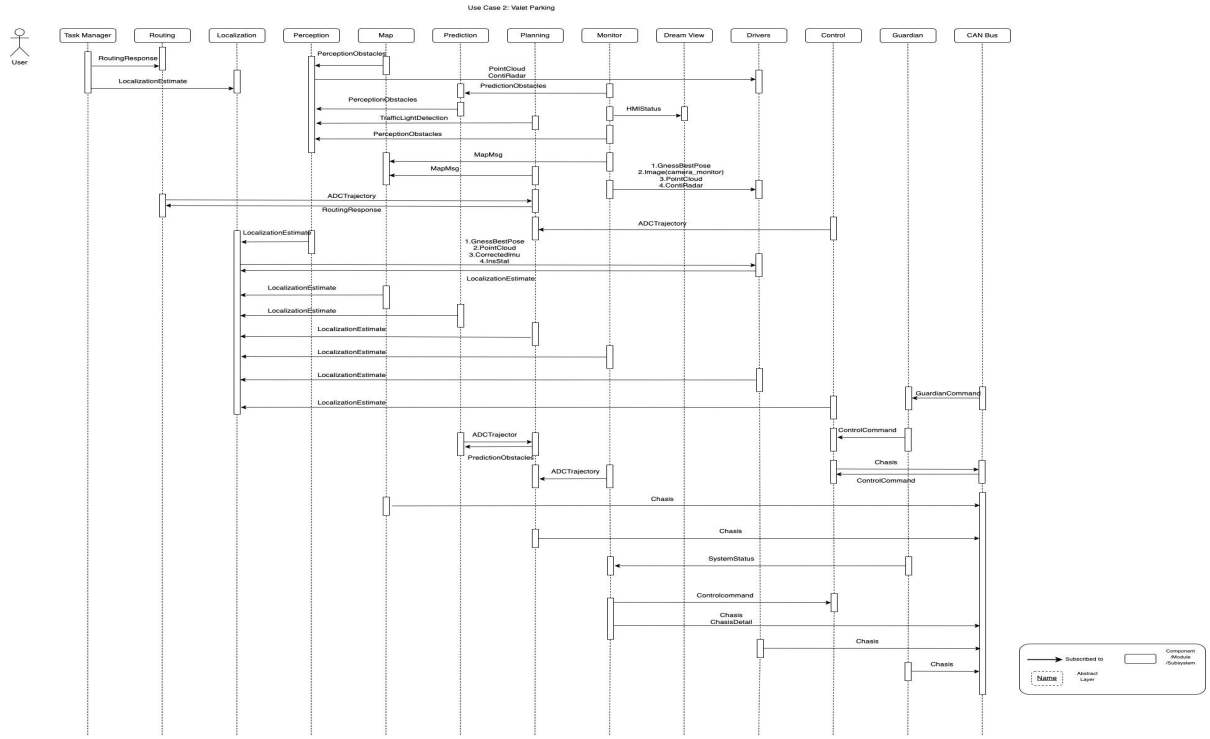

Figure 9. Sequence Diagram of Use Case 2: Valet Parking

Lessons Learned

Some of the lessons we learned were the development and derivations of our updated conceptual and concrete architectures. Through numerous trials and errors, we were able to narrow down our ideas to one centralized diagram. The reflexion analysis was difficult due to the lack of comments given in the source code to accurately pinpoint and understand the reason files were dependent on each other. Due to a large number of dependencies, selecting the most important dependencies proved to be difficult as each and one of them were complex. Another lesson we learned throughout this report was the ability to draw from past mistakes. The previous report showed us that although our thinking was on the right track, we missed a major component and were able to correct it in this report. Lastly, our cooperation and teamwork skills have improved due to the proper communication to delegate and split up the workload. Following the marking scheme, we have been more diligent in fulfilling all the requirements and being as detailed as possible.

Conclusion

In conclusion, by using Understand, the complex relations between Apollo subsystems and modules have been explained visually and textually. The exchange of data flow from modules to databases and within all the modules has given us a more detailed comprehension of how the real Apollo system works. The effort we made to figure out the conceptual architecture and concrete architecture has shown us how a mature corporation builds up such a project starting from scratch. Deriving from conceptual architecture, the concrete architecture is extracted from the system's implementation and consists of many unexpected dependencies between the components. The reflexion analyses performed on both the top-level and perception subsystem revealed a select few of the important unexpected dependencies within the system and the use cases also presented the ambiguous scenarios that could occur within the system.

Data Dictionary

CAN Bus (Controller Area Network bus): Cassis, a vehicle communication protocol.

Pipe and Filter: an architecture style that outlines a series of separate calculations to be performed on data, with each component reading and producing streams of data as input and output.

HMI: human-machine interface.

IMU: Inertial Measurement Unit, a device that can measure and report specific gravity and angular rate of an object to which it is attached.

ADC: Actuated Disc Cutting.

LIDAR: Light Detection and Ranging is a remote sensing method that uses light in a pulsed laser to measure ranges (variable distances) to the Earth.

RTK: Real-Time Kinematic is a technique that uses carrier-based ranging and provides ranges (and therefore positions) that are orders of magnitude more precise than those available through code-based positioning.

References

[1] *Apollo 5.0 Technical Deep Dive. This is part two of the two-part… | by Apollo Auto | Apollo Auto*. (2019, July 3). Medium. Retrieved March 18, 2022, from https://medium.com/apollo-auto/apollo-5-0-technical-deep-dive-d41ac74a23f9

[2] *ApolloAuto/apollo: An open autonomous driving platform*. (n.d.). GitHub. Retrieved March 18, 2022, from https://github.com/ApolloAuto/apollo

[3] daohu527. (2021, 09 10). *dig-into-apollo*. tree/main/modules. https://github.com/daohu527/dig-into-apollo/tree/main/modules

[4] Zhu, F., Ma, L., Xu, X., Guo, D., Cui, X., & Kong, Q. (2018, 08 30). *Baidu Apollo Auto-Calibration System - An Industry-Level Data-Driven and Learning based Vehicle Longitude Dynamic Calibrating Algorithm*. Arvix. https://arxiv.org/pdf/1808.10134.pdf