CISC 322/326 Assignment 2
Concrete Architecture of Apollo

March 21th, 2022

Group #7: ArchiTiger
Poppy Li 20181706 19xl12@queensu.ca
Xuan Xiong 20147035 18xx15@queensu.ca
Yuen Zhou 20186821 19yz57@queensu.ca
Yingjie Gong 20144264 18yg24@queensu.ca
Zhimu Wang 20190758 19zw28@queensu.ca
Baisheng Zhang 20094496 17bz15@queensu.ca

Table of Contents

1.0 Abstract

The Apollo system was investigated in the last report to determine its conceptual architecture. This technical report will go through a detailed overview of the concrete architecture of Baidu Apollo Autonomous Driving. After investigating the source code and using the udb supplied, five new subsystems were added: Routing, Storytelling, Driver, Common. Then, we discovered many new dependencies. At the same time, some predicted dependencies in the conceptual architecture were removed. After three nights mapping source code to our architecture using Scitools Understand, we found out that the initial proposed conceptual architecture was incomplete since it did not provide several components and did not capture many essential dependencies (Apollo Auto, 2018).

2.0 Introduction and Overview

While looking at the dependencies of these subsystems, we decided on a pub-sub style to be most fitting for our concrete architecture. This report aims to explore the concrete architecture of the Apollo Autonomous Driving System. We explain how we derive our final concrete and revised conceptual architecture. The report has four main content sections. The first section is the derivation process, where we discuss how we derived our architecture.

The second section documents the discrepancies between conceptual and concrete architecture. We constructed a modified conceptual architecture diagram (see Figure 1) using only documentation on Apollo without looking into the code. We have derived our finalised conceptual architecture through iteration and challenges of our original conceptual architecture (see Figure 2). The research will discuss the top-level concrete subsystems and their interactions with the architecture style and give a detailed introduction to the new subsystems, including unexpected dependencies that will include files, components and source code, and the meaning of the existence of the dependencies. Besides, the concrete and conceptual architecture will be compared by performing a reflection analysis. Then, we choose Perception to be the 2nd level subsystem. We present a subsystem decomposition, which investigates the inner architecture of this module in detail in both conceptual and concrete views.

The third section presents two Sequence diagrams for non-trivial use cases. This part aims to show how each part interacts with others. The first scenario is switching lanes, and the second shows Automatic Parking.

The fourth section reviews limitations in our analysis and lessons we learned during the process of derivation. Finally, we will finish things with a conclusion, followed by a list of references.

3.0 Derivation Process

The derivation process started by revisiting the conceptual architecture of the Apollo system from the previous assignment. We realized that the original diagram could not represent the actual conceptual architecture by examining it closely. The original conceptual architecture described relationships among different systems using data and control lines, whereas an actual conceptual architecture should be about dependencies. Therefore, all of our group members gathered together to analyze which components rely/depend on another one, with the help of the developer’s documentation and the graph visualization of pub-sub communication.

Once we derived a new conceptual architecture, we started building the concrete architecture by utilizing a software called Understand, which immensely helped us find Apollo's concrete architecture. It is a powerful tool for visually analyzing existing dependencies among 2nd level subsystems. By importing the source code, Understand enabled us to see the relationships in the Apollo system without knowing the code in detail. Subsequently, we created a new architecture and a series of components to represent the subsystems in the conceptual architecture.

When everything was appropriately placed, Understand generated our desired concrete architecture diagram that we can further analyze. Compared to conceptual architecture, more dependencies were added, and some of the original dependencies were removed due to the introduction of new subsystems. In addition, the generated concrete architecture confirms our previous conjecture of the architecture style, that it is a publish-subscribe style. From the experience of manually categorizing Apollo source files, we had a better understanding of the internal relationships of the Apollo system.

4.0 Conceptual Architecture

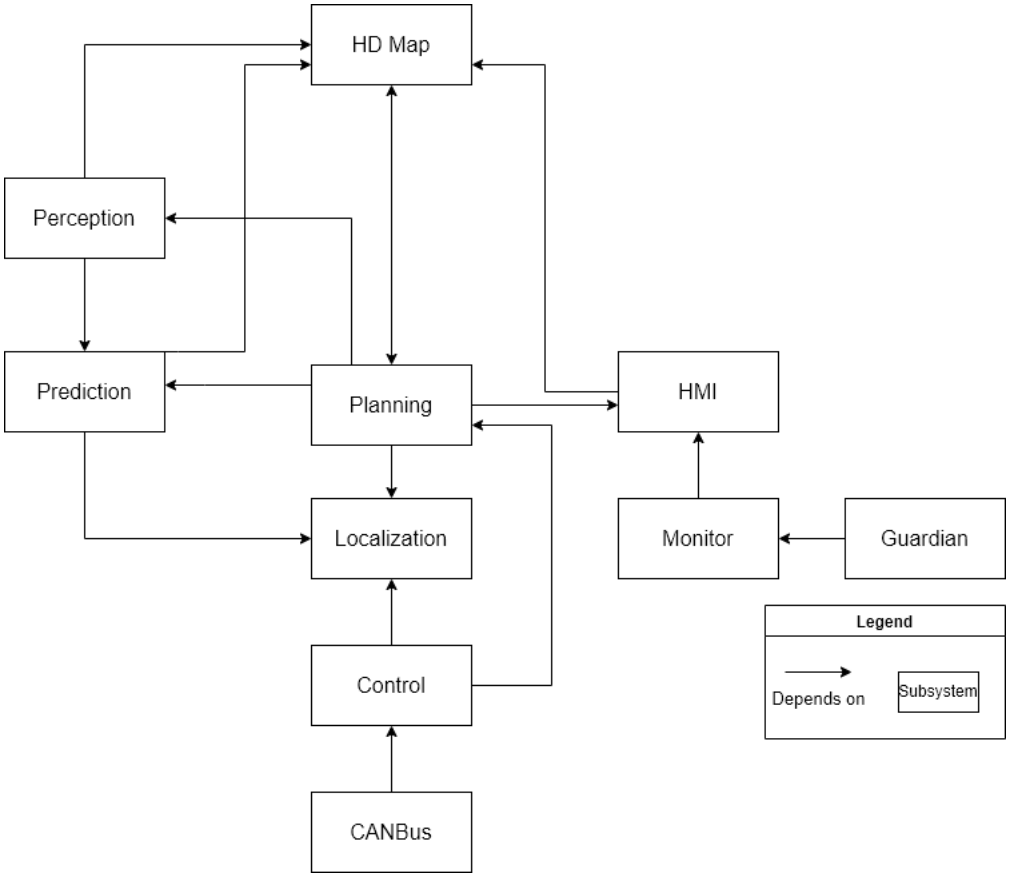

Figure 1. *Updated Conceptual Architecture*

By examining the original conceptual architecture and asking the professor, we concluded that our proposed conceptual architecture in the previous assignment was not technically a conceptual architecture. A conceptual architecture should be represented using dependencies rather than control and data lines. Hence, we deeply discussed the actual dependencies in the system and came up with this new conceptual architecture.

5.0 Concrete Architecture

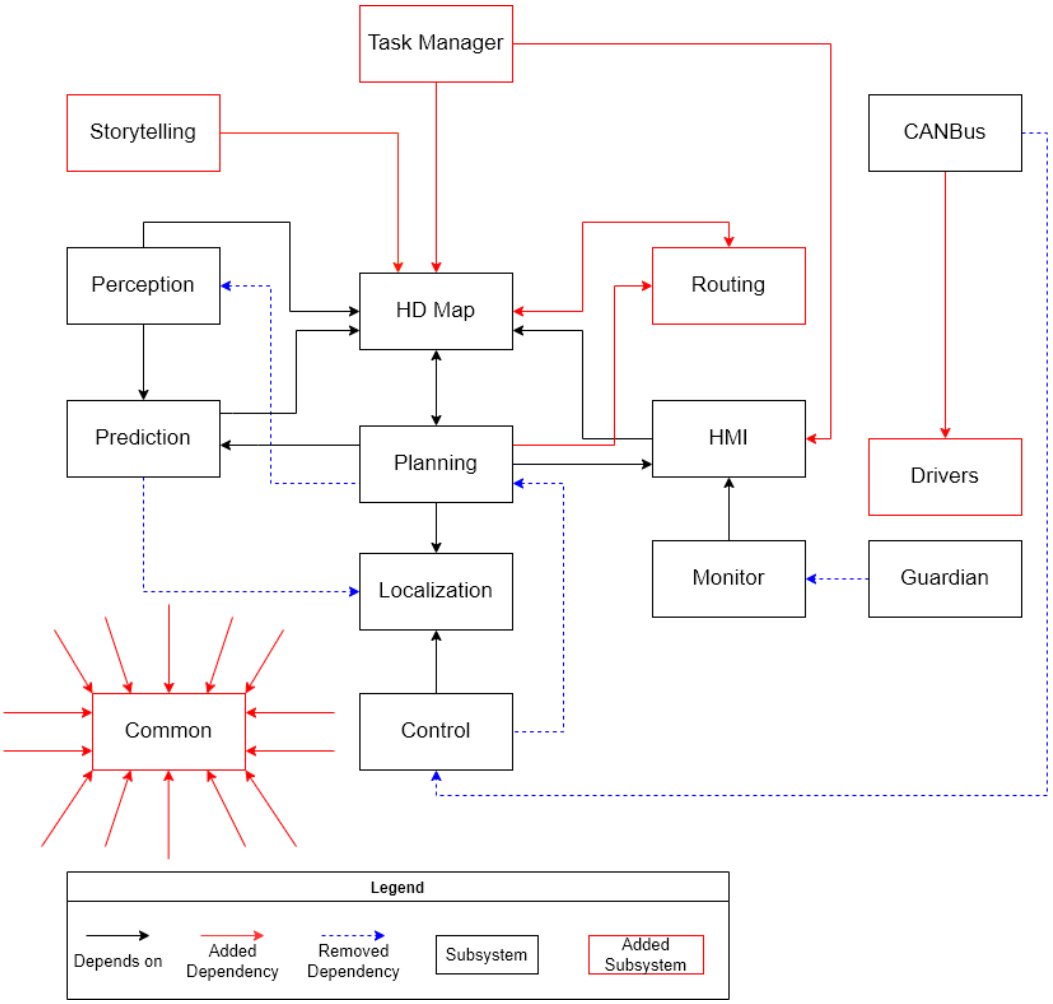

Figure 2. *Concrete Architecture Built by Understanding*

We redraw the graph generated from Understand to distinguish added and removed dependencies. They are identified with different colours. From it, we again conclude that the main architectural style of Apollo is the Publish-Subscribe style(Implicit Invocation Style). Most communications occur in the Common module. Modules publish events and subscribe to events that they have an interest in.

5.0.1 Concrete Subsystems and Interactions

5.0.1.1 New Subsystems

In the derived concrete architecture, five new subsystems are introduced:
Routing**:** When a user enters a destination, the Routing module gives several ways of reaching the destination.

Storytelling**:** This new module is introduced to the system to specifically manage complex scenarios and create stories for other modules to act. It helps coordinate cross-module actions. In the detailed pub-sub communication diagram, planning and prediction subscribe to its stories.

***Drivers*:** This component is adapted for the specific sensor, get sensor data and transfer to the Apollo data struct. It explains why Monitor, Localization and Perception subscribe to drivers for radar data, etc.

***Common*:** This module contains useful files that improve the functioning of Apollo. For example, it contains a math folder involving many useful mathematical libraries. Common is like a library with shared resources for every module in the system.

***Task Manager*:** This module subscribes to localization to get the information of car location and routing for routing response. The generated trajectory will then be presented on the HMI.

5.0.1.2 Subsystem Review

The following modules have already been presented in the conceptual architecture.

***Perception*:** It is used by the system to detect surroundings such as the traffic light, other vehicles and obstacles on the road. The Perception module is one of the most critical parts of the AD system. It is considered to be the "eye" of the system. In other words, no further actions can be activated without this module.

***Prediction*:** As its name implies, the Prediction module predicts perceived obstacles' possible future motion. For instance, we anticipate if a car on the other lane would switch lanes by observing certain features.

***Planning*:** This module is responsible for making plans for the autonomous vehicle to take. For instance, overtaking the car in front of the autonomous car,

***Control*:** The Control module performs the plans already set by generating commands to the car, such as accelerating, turning the right turn signal on, etc.

***Guardian*:** It is a safety module that plays the role of an Action Center, and it should intervene once Monitor detects a failure.

***CANBus*:** This module is the interface in charge of passing commands generated in the Control module to the vehicle hardware system.

***HD-Map*:** The full name is a High-definition map. This module acts as a library in the system that provides ad-hoc structured information regarding the roads. It allows the AD vehicle to analyze its surroundings in real-time.

***Localization*:** This module utilizes various information sources such as GPS to estimate where the vehicle is.

***Monitor*:** This module monitors all the modules in the vehicle, including hardware. In addition, it also receives data from a bunch of modules and sends the data to HMI for the driver to view vehicle status.

***HMI*:** A user interface establishes the connection between the machine and the user. It can view the vehicle status, control other modules, and boost the user's experience.

All the modules indicated above work closely to make the whole system function. The Perception module perceives surrounding environments with the help of cameras and radars. It returns information about traffic lights, other traffic participants and obstacles on the roads.

And then, it sends the data to the prediction module, which subscribes to Localization, Planning and Perception obstacle messages. When the Localization module transmits the result to the Prediction module, it updates the inner stored status and is triggered when it gets obstacle data from the Perception module. The HD Map module is in the system to help provide valuable roads information. The above explains why most components in the system depend on HD Map.

The following important module is the planning module, as it nearly interacts with all other modules. It takes the output from the prediction module about the surrounding obstacles. However, the prediction module does not return traffic light information. Thus, the planning module subscribes to the traffic light detection output to get complete road information. Notice that there is no longer a dependency from planning to perception. It is because of the introduction of the Common module, which handles inter-process communication. Planning components can now retrieve information from Common components. After that, it should take routing module output and localization output to make reasonable plans. Notice that a new module, task manager, is created for subscribing output from localization and routing. Now, this single module contains all the needed information. Therefore, the planning module can retrieve information only from the task manager.

The new storytelling module here is responsible for analyzing complex scenarios. Its existence avoids a sequential approach where multiple modules should work together to analyze complex planning scenarios. That is to say. The storytelling module is created for boosting the computation ability. Both planning and prediction subscribe to the stories to ensure safe driving.

The control module then takes the output from the planning module and generates corresponding commands. These commands will then be passed to the CANBus, which manages the transmission of commands to the hardware system. The CANBus now has a new dependency on the new module drivers since drivers contain sensor data of the CANBus in order to inform the monitor if CANBus can be successfully started. If all modules are working correctly, data flow remains the same as if the guardian were not present. By contrast, if one of the modules fails, the guardian module prevents commands transmission from the control module to the CANbus and brings the car to a stop.

Last but not least, the monitor module acts as a supervisor for all the modules and the hardware. It subscribes to all the module's events and then passes received data to HMI. HMI visualizes modules' output, such as the planned route, for the user. Monitor has a dependency on HMI because it utilizes backend functions from the HMI module. These two modules have a layered relationship that monitor provides services(software and hardware status) to HMI.

5.0.2 Reflexion Analysis

5.0.2.1 New Dependencies

CANBus => Drivers

Rationale: The module "Drivers" in Apollo provides driver support for hardware-related components. From the dependencies shown by Understand, we found the directory "modules/canbus/vehicles" includes the components supporting the vehicle controller and related functions. This finding explains the main reason why CANBus depends on Drivers.

Moreover, Apollo allows secondary developers to create their CanClient inherited from the class in “modules/drivers/canbus/can_client” as an open-source platform.

TaskManager => HMI
Rationale: The TaskManager module provides functions to manage routings; some of these functions depend on the support from Dreamview(HMI). The dependency arrow showing this characteristic is the one from “modules/task_manager/cycle_routing_manager.h” to “modules/dreamview/backend/map”.

TaskManager => HD Map
Rationale: In order to manage Routing, the TaskManager module needs the backend code support from the HMI module and needs to call the functions in the HD Map module to get the original map information. There are two parts in the TaskManager depending on the “modules/map/hdmap”, one is “modules/task_manager/parking_routing_manager.h” and the other is “modules/task_manager/dead_end_routing_manager.h”.

Everything else => Common
Rationale: The Common module is newly created to improve internal communications among different modules. It acts as an extensive library containing shared resources such that modules can retrieve necessary information generated by other modules inside the Common module. Therefore, every module in the system depends on the Common module. More specifically, every module has a dependency on “common/adapters/adapter_gflags.h” to communicate.

Storytelling => HD Map
Rationale: The Storytelling module is added to the system to deal with complex scenarios and generate stories for other modules. In order to create stories, it requires high environmental fidelity map information from HD map as input. Specifically, there is a dependency from “modules/storytelling/story_tellers/close_to_junction_teller.cc” to “modules/map/hdmap/hdmap_util.h”.

HD Map <=> Routing
Rationale: Similar to storytelling, the Routing module requires map data as well in order to generate high-level navigation information. Therefore, there is a dependency from the Routing module to the HD Map module. The HD Map module relies on a projected route based on the source code to analyze road information along the route in real-time. Thus, there is also a dependency from the HD Map module to the Routing module.

Planning => Routing
Rationale: The new Routing module also introduces a new dependency from the Planning module. The Planning module relies on Routing output to calculate appropriate plans. Under certain situations, the Planning module may request another Routing computation if the current route output cannot be faithfully followed.

5.0.2.2 Removed Dependency

Planning=>Perception
Prediction=>Localization

Rationale: Compared to Conceptual Architecture, there exist two removed dependencies. For these two removed dependencies, we explain that they are not responsible for a highly

complex function. For instance, the Localization module will send the LocalizationEstimate to the Planning module. Planning gives the feedback to HD Map, and HD map shows this information to all the modules that participated in planning the route (e.g., Prediction and Perception). The primary function of the Prediction module is to predict the obstacle, and this does not need to look at Localization all the time. HD Map can also provide all its needs, so the Prediction module does not depend on Localization. Similarly, the Planning module does not depend on Perception because it can work out the plans according to the processed information provided by HD Map. There is no need for the precise data to be directly received by the Planning module.

Control=>Planning
Guardian=>Monitor
CANBus=>Control
Rationale: These three dependencies removed are because we did not count in the Common module in our conceptual architecture. The Common module includes many functions and variables shared by all modules, such as adapters, configs, math, monitor log, vehicle status. This module realized the communication across modules without the messy dependency relation.

5.0.3 Chosen 2nd-Level and Reflexion Analysis

We will explain the internal architecture of Perception and its interactions from a conceptual and concrete point of view and point out the reflection analysis performed for the architecture of Perception. Multiple cameras, radars, and LiDARs are used in the perception module to spot obstacles and fuse their distinct tracks into a final track list. The obstacle sub-module handles obstacle detection, classification, and tracking. Perception also forecasts obstacle movement and position.

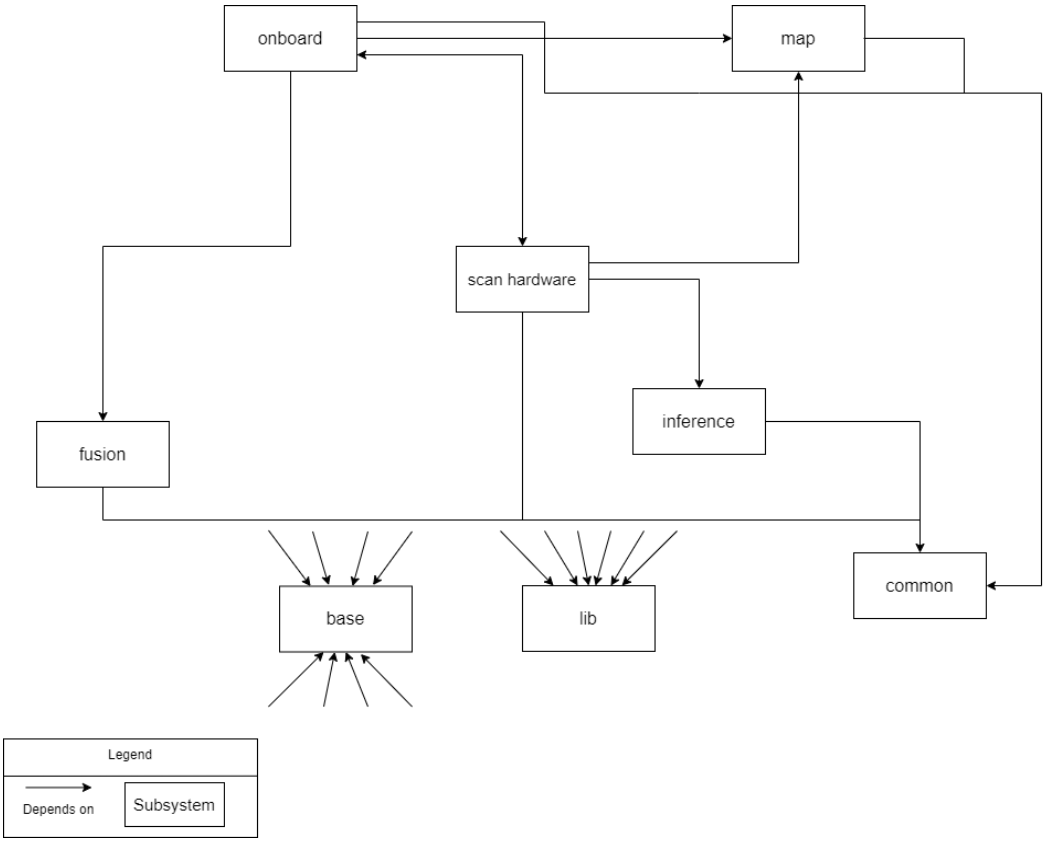

Figure 3. *Conceptual Architecture of Perception*

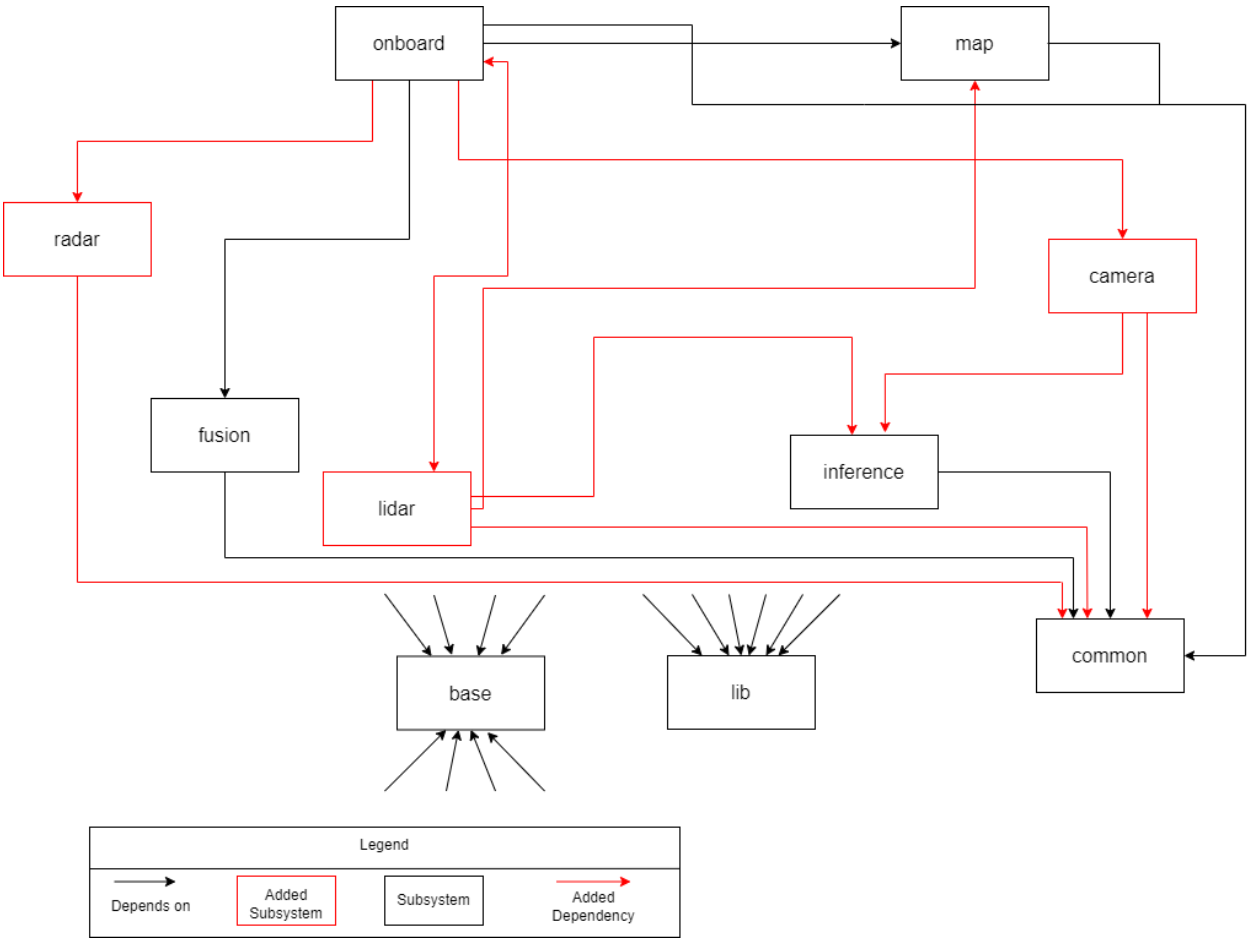

Figure 4. *Concrete Architecture of Perception*

Onboard: defines several submodules for processing different sensor information (Lidar, Radar, Camera). The entry point for each submodule is in the onboard directory, and the process for each sensor is roughly similar and can be divided into preprocessing, object recognition, region of interest filtering and tracking.

Inference: deep learning inference module, deep learning models need to be deployed after training, and inference is the process of deep learning deployment, in fact, the deployment process will do acceleration of the model, the main implementation of caffe, TensorRT and paddlepaddle 3 kinds of model deployment. The trained deep models are placed in the "perception\production\data" directory, and then loaded and deployed and computed online by the inference module.

Camera: mainly implements lane line recognition, traffic light detection, and obstacle recognition and tracking.

Radar: mainly implements obstacle recognition and tracking (here we mainly do obstacle tracking).

Lidar: mainly implements obstacle recognition and tracking, such as segmentation, classification and identification of point clouds.

Fusion: does the intersection and fusion of the sensing result from the above sensors.

Map: It contains the hdmap module inside, which does not subscribe or publish any content. It helps to provide valuable road information in the system.

Lib: manages and maintains information such as threads and queues generated by each module.

Base: holds test content for almost all modules as well as header files, etc., and provides the base for system implementation.

Common: A real-time consolidation of all modules. It contains information like vehicle model and vehicle status. It is also the core hub of Apollo system operation.

5.0.3.1 New Dependencies

Scan Hardware => Camera & Radar & Lidar
Rationale:

- When other modules call scan hardware to get around obstacles and so on, we have always thought that he is simply interdependent with Onboard and relies on the Common module and Inference module. However, the fact is that the whole system calls different hardware inside the Scan Hardware module to operate according to the actual situation, and each hardware module has different dependencies inside the system;
- For the Radar module, it is a basic dependency from Radar to Common;
- Followed by the Lidar module, who is interdependent with the Onboard module, which is interdependent with the Onboard module and is also associated with the Map module, which is then linked to the Common module; finally, there is the Camera module, which is very similar to the Lidar module, the only difference being that the Camera module does not come with the Onboard module.

5.0.4 Concurrency

Apollo has a particular scheduling system to meet real-time performance needs, enabling Apollo to achieve concurrency and multithreading. Apollo's action mode comes from multiple interacting processes, and there is concurrency between processes. They seem to handle different processes at the same time. The data of time is that multiple processes have been switched and run. Concurrency occurs in the Apollo perception module - responsible for checking obstacles around the car, consisting of 5 nodes (LiDAR, Radar, Fusion, Traffic Light preprocessing and Traffic Light process). Each node can be regarded as a thread. These threads work for the perception module, and there will be connections between threads. For example, "Fusion requires output data from lane post-processing, camera processing, and radar processing" (Alcon et al.,2020). Such high concurrency improves the performance of the perception module and allows Apollo to handle detecting obstacles well.

6.0 USE CASES

6.0.1 Use Case 1

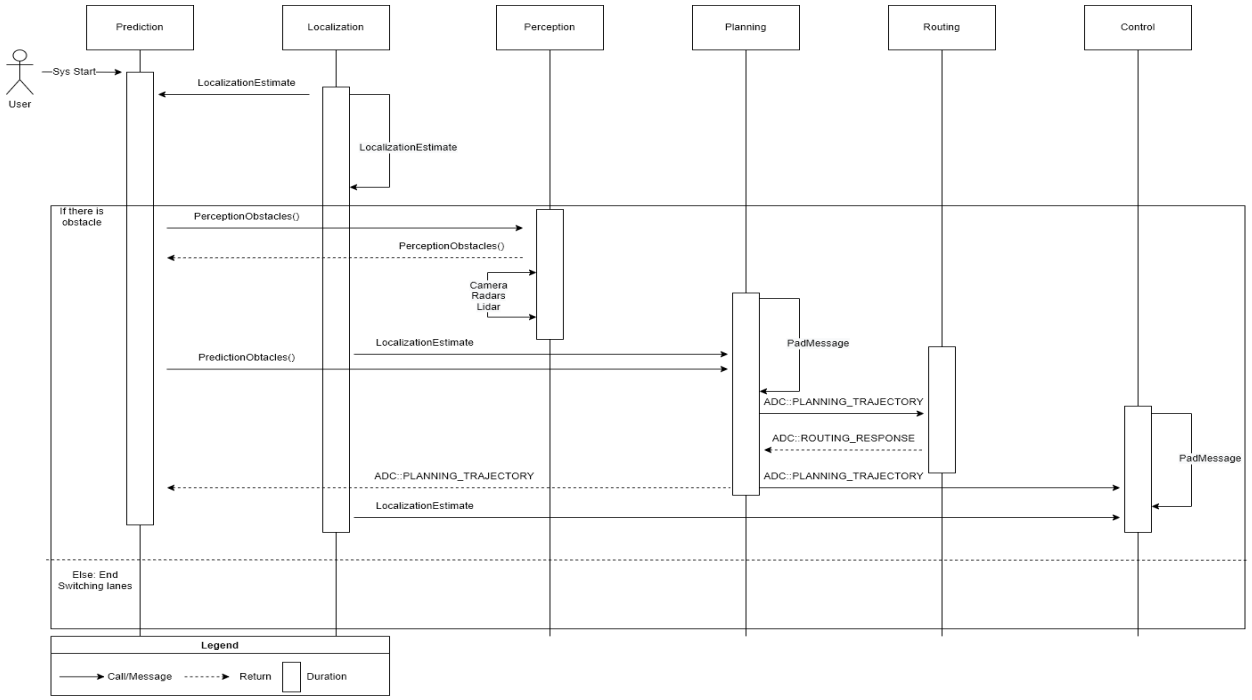

Figure 3. *Use Case 1: Switching Lanes*

This use case is the autonomous solution given by Apollo System to realize the switching lanes action of a running car. Suppose all other subsystems not listed in the sequence diagram above work well. The Prediction module detected some obstacle forehead of the moving car according to the message published by the Localization module through Localization Estimate. In order to avoid crashing into the obstacle, the Prediction module calls the function PerceptionObstacles() to get the precise distance to and the size of the obstacle through sensors contained in the Perception module. After receiving the message returned by the Perception module, the Prediction module calls PredictionObstacle() to notify the Planning module to generate a plan for Switching lanes. Once the Planning module gets the LocalizationEstimate information, it calls ADC:: PLANNING_TRAJECTORY to get a route provided by the Routing module. Then the Routing module returns a route by ADC:: ROUTING_RESPONSE to Planning. Finally, the Planning module finishes the ADC:: PLANNING_TRAJECTORY function by sending the route to the Control module. At the same time, the Planning module informs the Prediction module to continue to keep track of the traffic conditions ahead. Before completing the switching lanes, the Control module keeps correcting direction according to the data provided by Localization Estimated.

6.0.2 Use Case 2

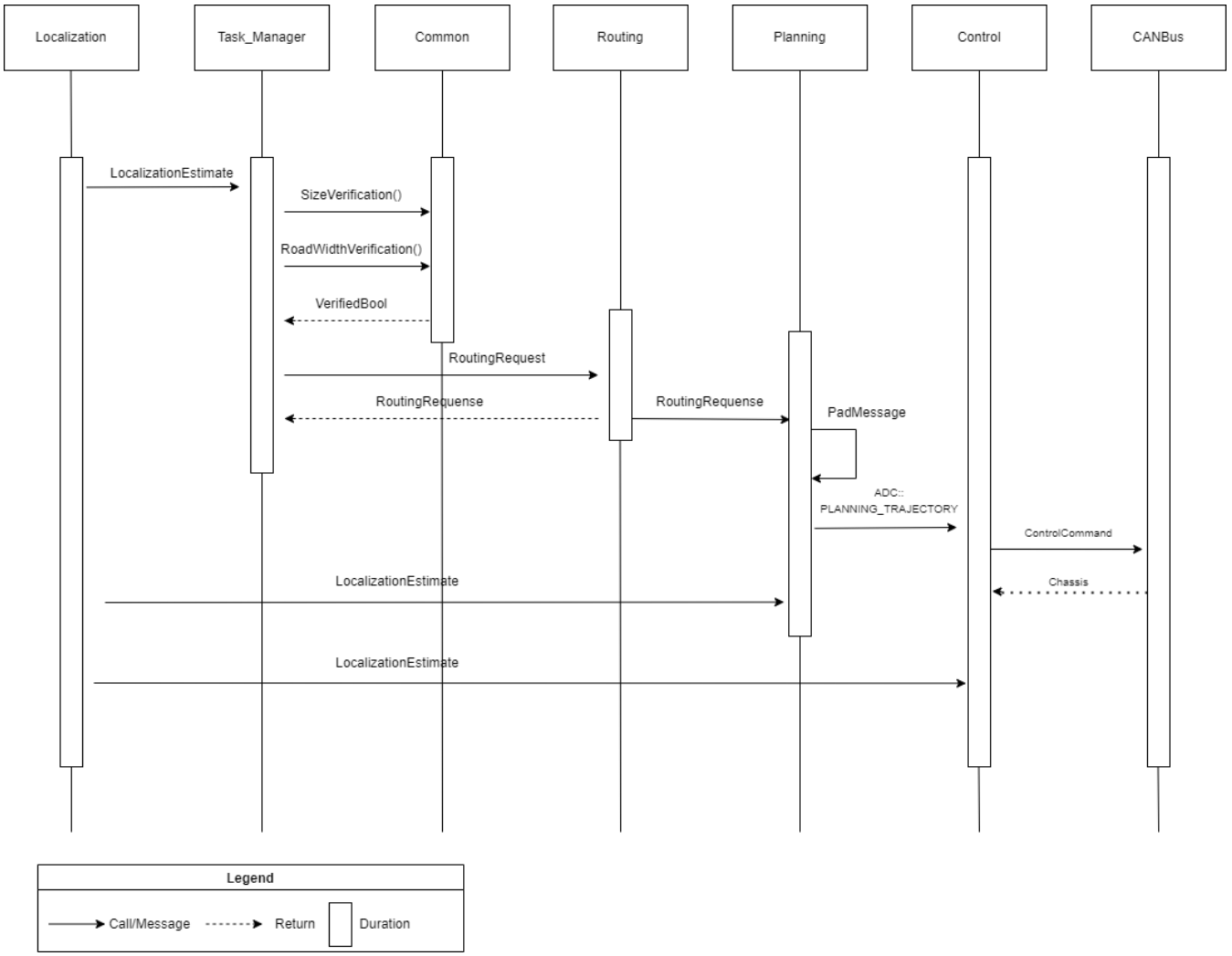

Figure 4. *Use case2: Automatic Parking*

This use case is an autonomous solution given by Apollo System to achieve automatic car parking. Assume that all other subsystems not listed in the sequence diagram above are working well. Furthermore, the task_manager module detects obstacles around the car based on the messages issued by the Localization module via Localization Estimate. It sends it to the Common module, and Common returns a confirmation function. Immediately after that, the Common module gives a RoutingRequest to the Routing module and gets the feedback from Routing. This feedback is also handed over to the Planning module simultaneously. When the Planning module gets the RoutingResponse and LocalizationEstimate information, it calls ADC::PLANNING_TRAJECTORY to get the route provided by the Routing module, and Control sends a ControlCommand to the CANBus module to allow the CANBus to manage the transmission to the hardware system.

7.0 Team issues and Lesson learned

7.0.1 Team issues

Since Apollo's subsystems are very numerous and complex, some of them have similar functions, making it difficult for us to analyze the dependencies between modules. On

the other hand, the content of the ReadMe document of some modules is not straightforward. It does not reflect the function of each module and why there are dependencies between them, and it is difficult for us to understand the specific function of each module. In this case, later maintenance will be challenging.

7.0.2 Lesson learned

We comprehended a lot while exploring Apollo's concrete architecture. We understood why conceptual architecture and concrete architecture do not match. There are missing relations in conceptual architecture. Unlike conceptual architecture, concrete architecture has more functionality and uses different mechanisms. The most important thing is that we learned how to use Understand to analyze the dependencies between modules. The analysis results allow us to clearly understand the role of each module through the conceptual architecture. Understand also lets us know the unbelievable massive magnitude of a modern codebase. If we do not use Understand, it is tough to form a concrete architecture through the conceptual architecture by studying the dependencies from each module by ourselves because concrete architecture is much more complex than conceptual architecture.

We also learned a deep understanding of the importance of teams. When the team members do not understand how to use Understand, other team members will actively help them. For example, they will explain how to analyze the dependencies between modules step-by-step through a meeting. Teamwork has played a very significant role.

8.0 Limitations

This time the assignment has more restrictions than the previous one. Firstly, *SciTools Understand* is complex software. Using it to analyze the dependencies between modules always takes us much time. It often crashes, and then we have to start again from the beginning, which limits our overall analysis of Apollo and our time in meetings. Secondly, when we analyzed the concrete architecture of Apollo, we discovered that the module ' Task Manager' was not completed. Since Task Manager is a new system of Apollo 7.0, it is still in the development stage, so many functions are not finalized. The Task Manager is supposed to be essential in managing different routings and prioritizing threads. Unfortunately, very few documents talk about Task Managers, and the developer supplies no ReadMe document. It shows a dependency between Task Manager and Routing in the graph visualization of pub-sub communication, but it is challenging to find the dependency in the documentation. Overall, all these factors restrict our study of Apollo's specific architecture.

9.0 Data Dictionary

Subsystem: Part of a large system, which can be used as its configuration or device.

Dependency: A basic function, library or code fragment is called dependency in programming, which plays a vital role in the work of different parts of code

SciTools Understand: A static code analysis tool that can realize complete code navigation, control flow chart generation, and check whether the code meets specific coding standards.

10.0 Name Conventions

UDB (user database): the database created for storage and use of user data.

ADC (actuated disc cutting): A circular cutting method with two main characteristics.

HD map (high-definition map): High precision maps are usually used for automatic driving and contain details that usually do not exist on traditional maps.

HMI (Human-Machine Interface): A user interface or dashboard enables a person to connect to a machine, system, or device.

CANbus (controller area network): A message-based protocol that aims to allow electronic control units (ECUs) and other devices in vehicles to communicate with each other in a reliable, priority-driven manner.

LiDARs (Light Detection and Ranging): A remote sensing method that uses light in the form of a pulsed laser to measure distance.

11.0 Conclusion

Conceptual Architecture defines a conceptual form of system in which ideas or concepts are introduced within the system, and the concrete architecture should be more specific. In this report, we mainly studied the concrete architecture of Apollo and the direct dependencies of subsystems. We concluded that the pub-sub style is the most suited one for the concrete architecture of Apollo. When analyzing the conceptual architecture to get the concrete architecture, we realized that the diagrams in the last report could not represent the actual conceptual architecture. By analyzing the developer's documents and graphical visualization of pub-sub communication, we get a new conceptual structure and use it to deduce the concrete architecture. The reflexion analysis shows the new dependencies between various subsystems, and all subsystems require the Common. Through the observation and analysis of the concrete architecture of Apollo, combined with the help of Scitool's understanding, we can have a deeper understanding of Apollo.

12.0 References

Miguel Alcon Universitat Politècnica de Catalunya, Alcon, M., Catalunya, U. P. de, Center, H. T. B. S., Tabani, H., Center, B. S., Center, J. A. B. S., Abella, J., Center, L. K. B. S., Kosmidis, L., Francisco J. Cazorla Barcelona Supercomputing Center, Cazorla, F. J., University, K. S., University, B., Tech, N. M., Pisa, U. of, & Metrics, O. M. V. A. (2020, March 1). En-route: Proceedings of the 35th annual ACM Symposium on applied computing. ACM Conferences. Retrieved March 21, 2022, from https://dl.acm.org/doi/10.1145/3341105.3373938

Apollo Auto. (2018, July 18). Apollo 3.0: Entering the New Era of autonomous driving. Medium. Retrieved March 21, 2022, from https://medium.com/apollo-auto/apollo-3-0-entering-the-new-era-of-autonomous-driving-d781bc769cef