Assignment 2: Apollo Concrete Architecture Report

March 21, 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

This report examines the concrete architecture of the Apollo auto platform. After a brief summary of previous work (from Assignment 1), the concrete architecture of Apollo is derived using the Understand tool. This is followed by a reflexion analysis, which identifies important differences between the concrete architecture and our initially proposed conceptual architecture. We then move on to examine a specific subsystem in more detail – namely, the perception module. The concrete architecture of the perception module is derived (using Understand) and reflexion analysis is again performed to highlight concrete/conceptual disparities. Finally, the report ends with the presentation of two use cases, and a brief discussion of the lessons learned.

1. Introduction

In our previous report (A1) we outlined the conceptual architecture of the open-source self-driving platform Apollo. Our conceptual architecture was based primarily on the documentation available on the Apollo GitHub page, as well as the Apollo website. At this point, we had not actually looked at the Apollo source code but were relying solely on the idealized conceptual framework laid out by the developers.

In the present report (A2), we move on the explore the *concrete* architecture of the Apollo platform. This involves recovering the actual (empirically observable) structure of Apollo from an in-depth examination of its source code. In particular, we use the program *Understand* (Version 5.1) to analyze the Apollo codebase and to extract the dependencies between its various subcomponents. Based on the results of our concrete derivation, we then perform a reflexion analysis, examining the discrepancies between our original conceptual architecture and the actual concrete architecture.

The structure of our report is as follows. First, we will provide a summary of our updated conceptual architecture, focusing on any changes made relative to our originally proposed conceptual architecture (in A1). Second, we use Understand – and the provided PubSub graph – to derive the concrete architecture of the Apollo platform. Third, we perform a reflexional analysis, examining the discrepancies between our conceptual/concrete architectures. Fourth, we take an in-depth look at the concrete architecture of the Perception module. Finally, we end by presenting two use cases and discussing lessons learned.

2. Revised conceptual architecture

We have made several changes to our conceptual architecture relative to the original version from Assignment 1. Our original conceptual architecture contained nine modules: (1) Perception, (2) Prediction, (3) Planning, (4) Control, (5) Localization, (6) Routing, (7) CanBus, (8) Monitor and (9) Human Machine Interface (HMI). We initially did not include the Map module, because it was portrayed (in the Apollo documentation) as being part of the cloud platform rather than the open software platform. Upon further inspection, however, it appears

that the Map module is indeed an important part of the system. Several modules – such as Planning and Routing – crucially rely on map information to carry out their tasks successfully. As such, we have decided to update our conceptual architecture to include the Map module, in addition to the modules listed above.[1]

Our new conceptual architecture, with the Map module included, is as follows:

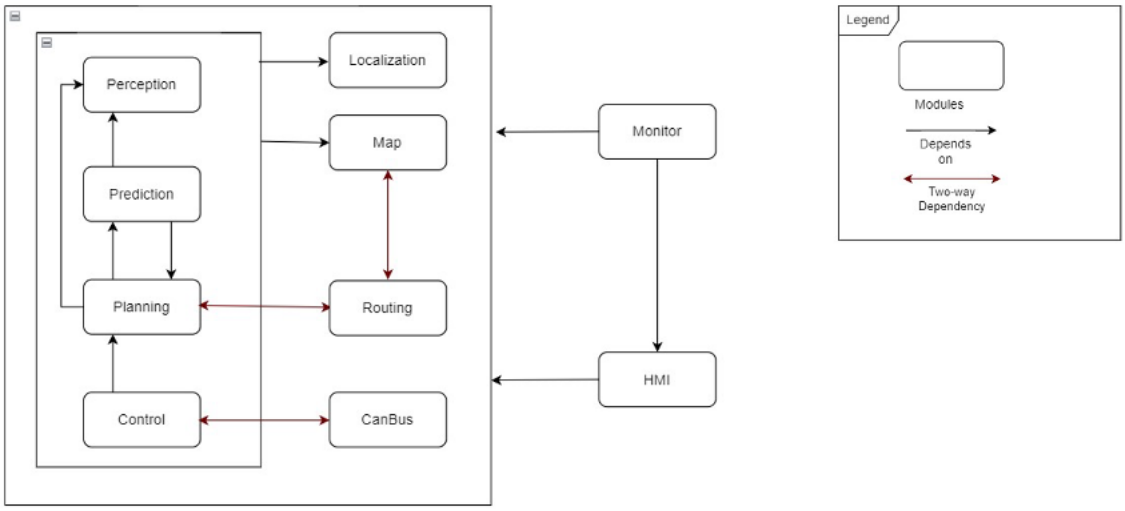

Figure 1. Conceptual dependency graph

Beyond adding the map module, there are a few other small changes. First, we have added a dependency from Planning to Perception. This reflects the fact that the Planning module "subscribes" to the perception module, basing its calculations on inputs from both Prediction *and* Perception (as well as Routing). Second, we have removed an originally-proposed dependency from Routing to Localization. This dependency was meant to capture the fact that Routing relies on map information. However, this is more accurately characterized as a dependency between the Routing module and the newly-added Map module – which is included in our new architecture.

Finally, we have also revised our proposed architectural *style* for the Apollo platform. We initially identified Apollo as having a pipe-and-filter/process-control architecture. However, given the flexible nature of the connections between modules (with any one module being able to "listen" to the output of many others), we have decided that Apollo is better characterized as

[1] We also considered adding the Guardian module as well, but we ultimately decided against it, as the Guardian does not appear to be part of the core system architecture. The Guardian module essentially serves as a breaker system that can override Control in the case of a software or hardware failure. When the vehicle is operating normally, however, "Control signals are sent to CANBus as if Guardian were not present" (https://github.com/ApolloAuto/apollo/blob/f7d1f94f5342737f33b52132bbd5a98243fd820b/docs/specs/Apollo_5.5_Software_Architecture.md#guardian).

having a Publish-Subscribe (PubSub) architecture.

3. Derivation process

To derive the concrete architecture of the Apollo system, we began by analyzing the source code using *Understand* – a code visualization tool that helps to identify interdependencies between a project's subsystems and/or file-structures. We first mapped the Apollo source code into corresponding modules based on our proposed conceptual architecture. Then, using Understand's graphing features, we were able to construct a directed graph of the dependencies between modules, as shown in Figure 2 below. Here the blue lines represent one-way dependencies between modules, while the red lines represent two-way dependencies.

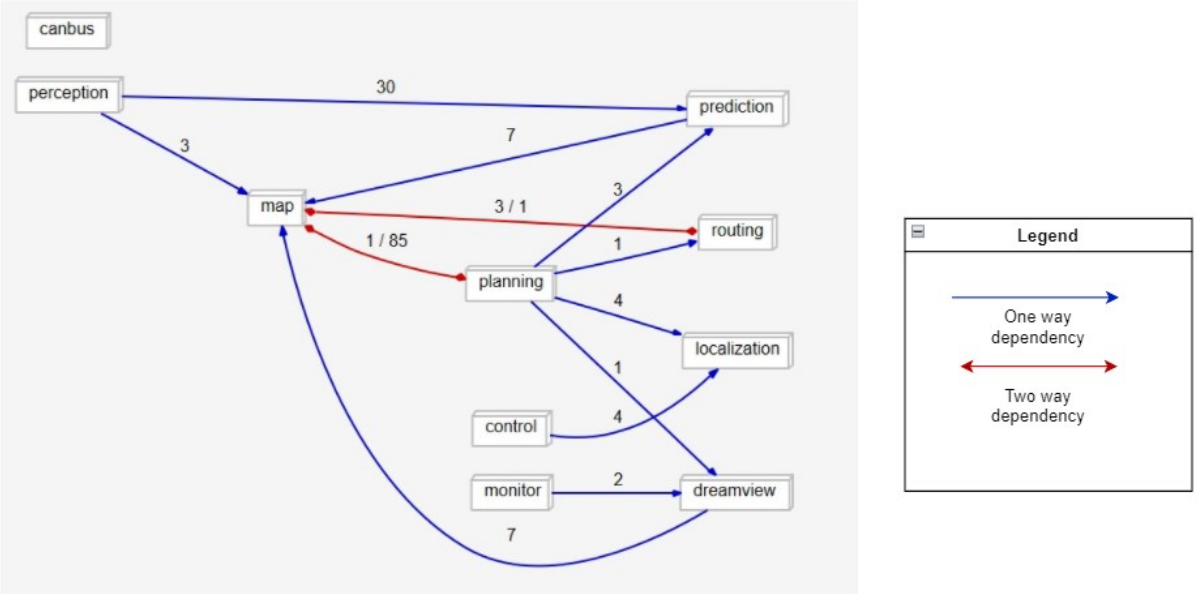

Figure 2. Concrete Dependency Graph from Understand

Simply relying on the Understand tool is not enough to retrieve the full list of concrete dependencies, however, as Understand fails to detect Pub-Sub message traffic. Thus, to supplement our analysis using Understand, we also looked at the various Publisher/Subscriber relations that existed between modules (as outlined in the provided PubSub communication model). Here, we interpreted "subscription" as a kind of dependency relation: if a module A subscribed to the output of another module B, we took this as evidence that module A *depends* on module B (since A requires input from B in order to perform its function).

Having carried out our analysis in Understand – and having looked at the PubSub relations – we then combined the results to arrive at the final concrete architecture of the Apollo system. Our final architecture thus encompasses both the static dependencies that exist in codebase (things like #include statements and functional calls, revealed by Understand) and the PubSub dependencies that result from modules "subscribing" to one another.

4. Final Concrete Architecture

Based on the derivation process described above – which combines static dependencies and PubSub relations – we arrived at the following concrete architecture:

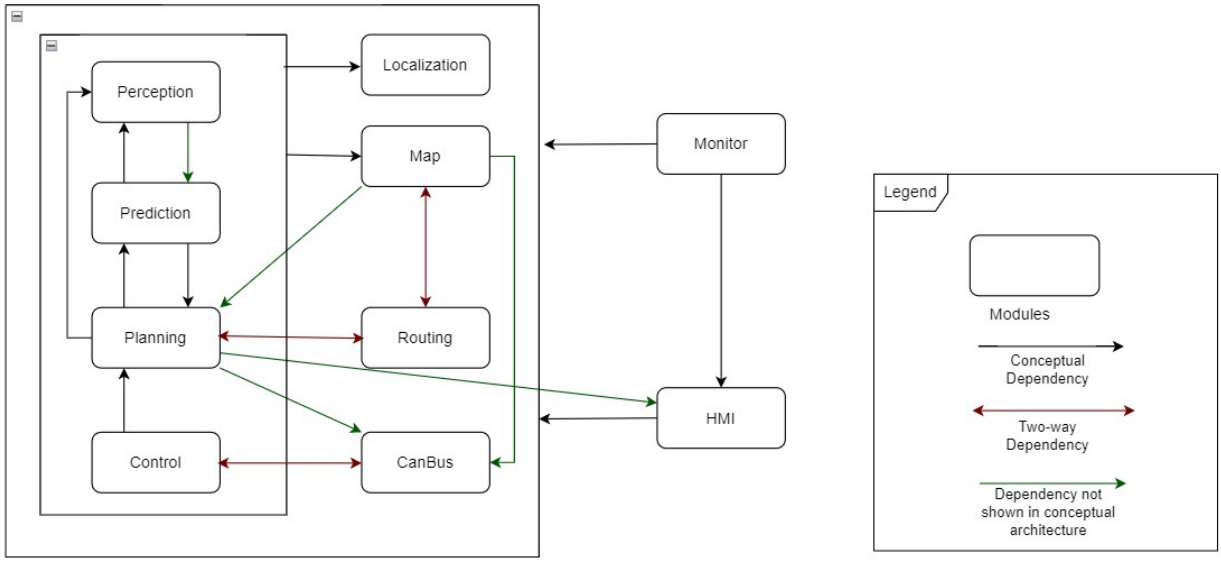

Figure 3. Concrete dependency diagram combining Understand and Pub-Sub traffic

Here the green arrows represent new empirically observed dependencies that were *not* present in our original conceptual architecture. It is worth noting that, although several new dependencies were added (relative to our conceptual architecture), none have been removed. Thus, all of the dependencies that we previously identified in the conceptual architecture do indeed exist in the concrete system. The concrete architecture just has a few *extra* dependencies. These new dependencies will be discussed in detail in our subsequent reflexion analysis.

5. Reflexion analysis

In this section, we outline the new dependencies and provide possible explanations for why they appear in the concrete architecture. Here, arrows from A → B indicate that A *depends on* B.

Perception → Prediction

There are a number of unexpected dependencies between the Perception module and the Prediction module. The sources of these dependencies are several files in the semantic mapping component of the Perception module, including *evaluator_manager.h*, *evaluator_manager.cc* and *mlf_engine.h*. These files import various other files from the Prediction's container submodule (most notably, *obstacles_container.h*), and make use of the obstacle-related classes/functions defined therein.

At a general level, the reason for these dependencies has to do with the Perception module's semantic mapping functionality. In constructing a sematic mapping of its surroundings, the Perception module must label various objects in the environment as being "obstacles". To accomplish this task of labelling obstacles, the Perception module makes use of some previously defined obstacle classed (e.g., Obstacle, ObstacleContainer) declared within the Prediction module. While refactoring *might* be a possibility here, the sheer number of dependencies (30) suggests that the decision to import resources from Prediction (into Perception) is a valid feature of the architecture.

Map → Planning

There is a single dependency from the Map module to the Planning module. Specifically, the Map file *pnc_map.cc* contains an #include statement that references the file *planning_gflags.h*, located in the "common" submodule of Planning.

The imported file *planning_gflags.h* contains a number of command line flags (gflags) that specify default values related to the planning process. These flags appear to be imported into the Map module because their default values are needed for various map-related computations. The dependency seems to be fairly minimal, however, as there is only one function – namely, PncMap::UpdateVehicleState() – that makes use of the imported gflags (and only two of the gflags are ever referenced). Refactoring may be possible in this case.

Planning → HMI

There is also a single dependency from the Planning module to the HMI module. This stems from *open_space_roi_decider.h* file in Planning, which contains an #include statement referencing *map_service.h* in the HMI module (i.e., Dreamview).

The reason for this dependency is somewhat mysterious. Looking at the source code, it's not clear that the imported *map_service.h* file actually gets used by any of the relevant planning functions (in *open_space_roi_decider.h*). This dependency may be a good candidate for refactoring.

Map → CanBus & Planning → CanBus

Aside from the dependencies revealed through Understand, there are also two new PubSub dependencies not included in the conceptual architecture: (i) a dependency from the Map module to the CanBus, and (ii) a dependency from the Planning module to the CanBus. Both of these unexpected dependencies are the result of "subscriptions" to the CanBus – the module that is responsible for interfacing with the vehicle hardware, and publishing information about the vehicle chassis.

The likely reason for these additional subscription-dependencies is the speed advantage of getting vehicle data directly from CanBus (rather than from more peripheral channels). The

CanBus module collects data from the car's chassis, including speed, acceleration, and orientation. By taking advantage of this chassis information, the Map and Planning modules can get near-instantaneous updates regarding the current behavior of the vehicle. For the Planning module – which is responsible for devising a safe path for the vehicle based on its current environment – it would be important to receive this raw data about the car's speed and acceleration in real-time from CanBus. Similarly for the Map module, getting immediate access to information about the car's orientation would be essential. Because of these potential time savings, these pub-sub communications/dependencies were likely an intentional choice and are not candidates for refactoring.

6. Second-Level Subsystem: Perception

The perception subsystem is responsible for perceiving obstacles which the automated driving system must consider when plotting and executing a route. The perception system is the vehicle's sensory system and as such communicates with the vehicle's many different sensors to detect an obstacle's heading, velocity, and classification.

6.1 Architecture

The internal architecture of the perception subsystem is Pub-Sub. In the Pub-Sub relationship the array of sensors can be seen as the publisher and the deep networks which classify the output of the publisher are the subscribers. Within the perception subsystem is a directory labeled "onboard" which acts as the communication step between the hardware and the rest of the perception subsystem. Events dispatched by the sensors are classified into their types. Examples of the possible event classification codes are LIDAR_PREPROCESS, LIDAR_DETECTION, STEREO_CAMERA_DETECTION, etc.

These events are then appended to a message buffer and are consumed by the appropriate process within the prediction subsystem. Image data is subscribed to by the traffic light detection, lane flow, and object detection deep networks, while Lidar data is subscribed to by the point cloud segmentation deep network.

6.2 Fusion/Output

The fusion step is one of the most important steps of the entire perception subsystem. Within this step the data that has been collected, classified, and tracked by the perception subsystem are merged to represent one stream of objects. The signature of an object from camera data is matched with the same objects LiDAR (Light Detection and Ranging) and Radar signature, giving the system a much higher confidence interval than that which operates on one or two sensor types. It is this stream of objects which is streamed to all consumers including the monitor, prediction, and map submodules.

6.3 Perception Reflexion Analysis

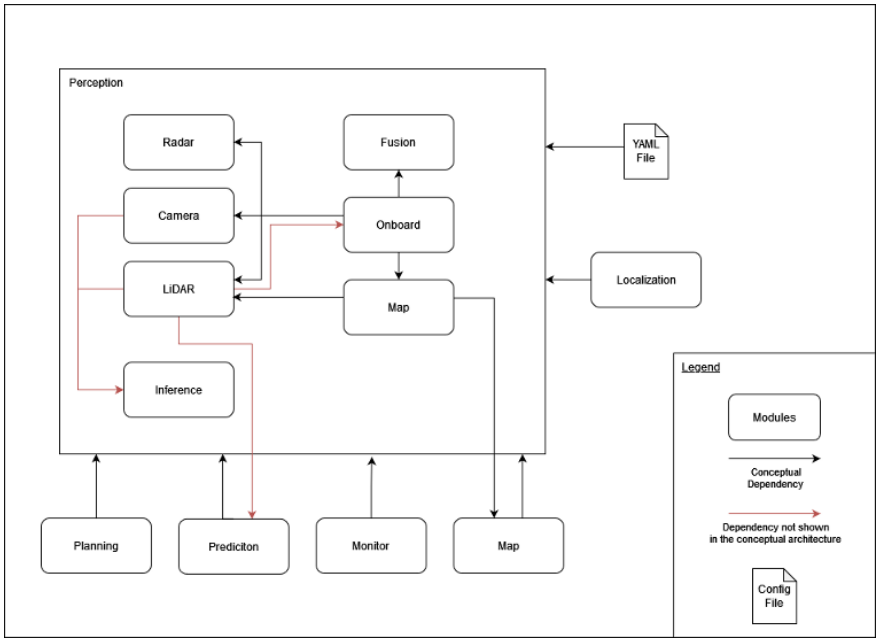

Figure 4. Discrepancies between perception subsystems conceptual and concrete architecture.

Discrepancies

Camera → Inference

The Camera submodule depends on the Inference submodule for region of interest pooling. Images taken by the Cameras are sub-divided into regions of interest or ROI's. These ROIs are much smaller and allow the deep nets which classify the images to work much faster. An example of where Apollo uses this is identifying the color of traffic lights.

LiDAR → Inference

Very similarly to Camera, LiDAR relies on inference for the grouping, reshaping, and processing of "blobs" of regions of interest. Additionally, LiDAR pulls from Inference some configuration for the GPU API used in Apollo called Nvidia Cuda.

LiDAR → Onboard

LiDAR depends on Onboard to classify the types of LiDAR events. Onboard has within in it the types of events which are used to classify all Perception based events.

LiDAR → Prediction

Prediction is used by LiDAR as the predicted obstacles are considered in the multi-LiDAR fusion which occurs within the LiDAR submodule. *Mlf_engine.h* includes the Pose and Obstacle containers from the prediction module.

7. Use cases

7.1 Use case #1: Red light detection

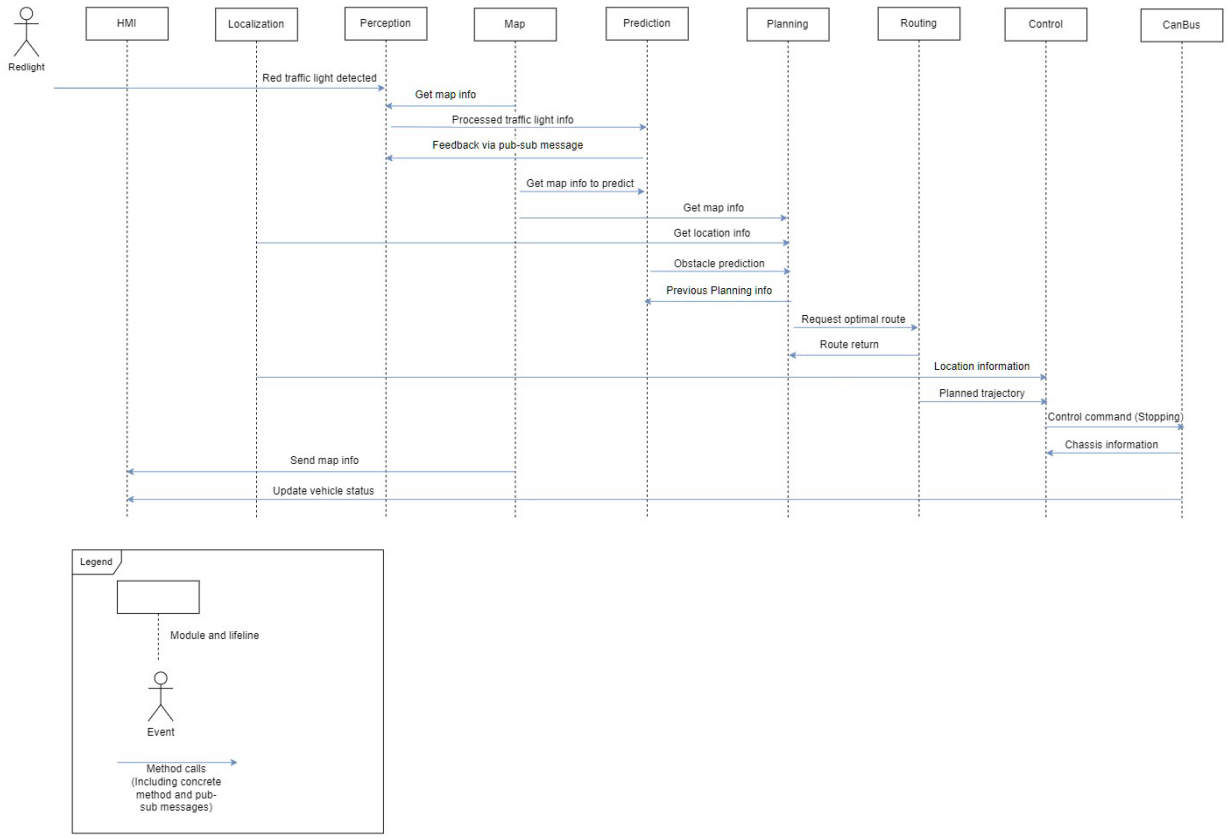

Figure 5. Sequence Diagram for red light detection use case

The above sequence diagram outlines one of the most common use cases of the Apollo self-driving vehicle – automatically detecting and stopping at the red light. In this scenario, 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. Next, the perception module requests map information from the map module, and the final light recognition is sent to the prediction module with the location of the light. After the prediction receives the information, it sends a feedback message back to the perception module via a pub-sub message.

The prediction module uses all the provided information to generate the predicted trajectory and actions for the upcoming redlight. The predicted information – along with the map and localization 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, the planning module sends the planned trajectory *back* to the prediction module for future analysis.

The next step is to execute the plan through the hardware, so that the car will physically stop. This task is initiated by the control module. The control module takes the location information and the previous planned 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 let go of the throttle and apply braking). The decision made by the control module is then sent to the CanBus module for final hardware execution. The CanBus module caries out the hardware commands and sends back the chassis information to control. At this point, the vehicle comes to a stop, and the current status of the vehicle (including the map location) is sent to the HMI/Dreamview module for the user to see in real time.

7.2 Use case #2: Navigation and change of destination

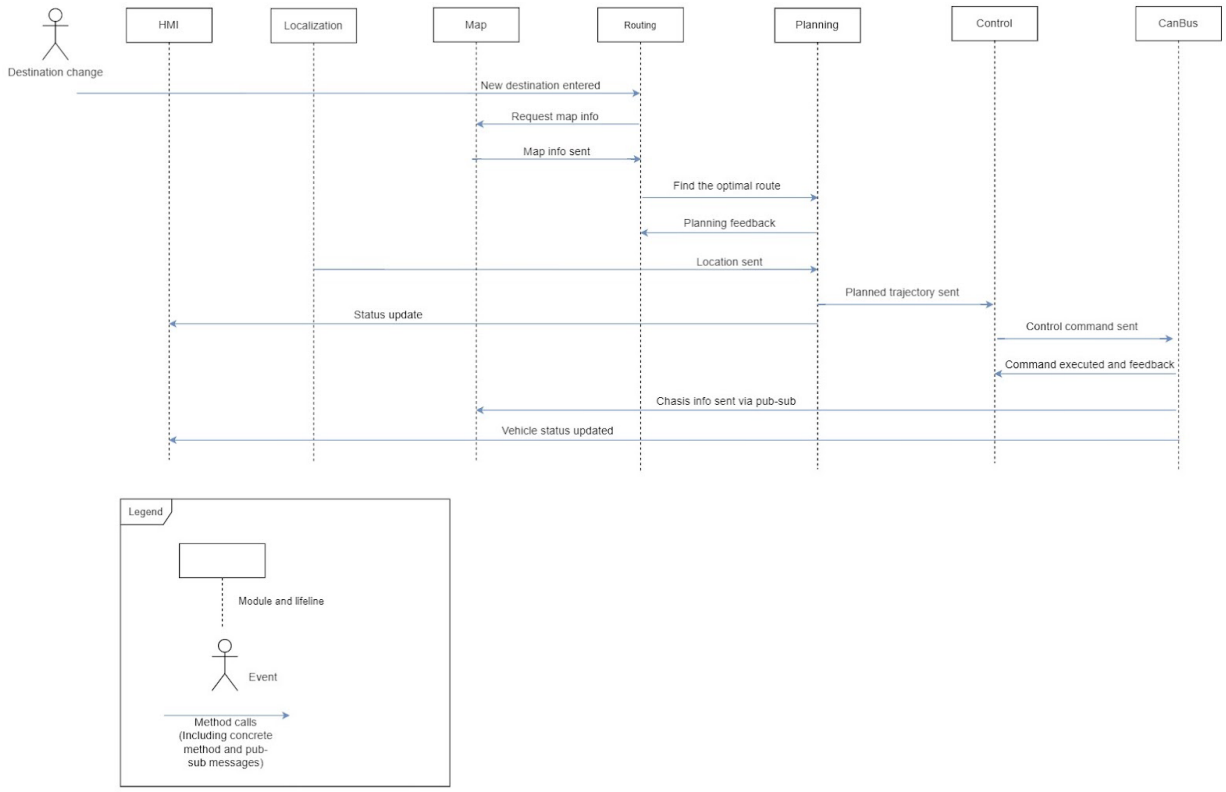

Figure 6. Sequence Diagram for the use case of navigation use case

Another fundamental feature that Apollo supports is *destination-changing*: users can change their destination at any time, and, in response, the autonomous vehicle will generate the most optimal route to accommodate the change request. This process is outlined in the above sequence diagram.

The basic steps involved in updating the vehicle's destination are as follows. First, after the user selects a new destination (by entering it into the HMI), the updated destination information 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 combination of

roads and lanes for the vehicle to follow. The new route – together with location information from the localization module – is sent to the planning module, which uses this information to generate an optimal trajectory for the vehicle.

After the above processing finishes, the planned trajectory is sent to the Control module, which generates commands to be carried out by the vehicle hardware. (Additionally, the planning module also reports back to HMI, via pub-sub message, to update the vehicle's status.) The commands generated by the Control module are then finally relayed to the CANBus module, which directly interfaces with the vehicle's hardware, and sets the vehicle on a trajectory to its new location. Finally, the CanBus module sends chassis information back to the Map module and HMI module (via pub-sub message), thus providing updates about the vehicle's status in real-time.

8. Conclusion and Lessons Learned

In this report, we explored the concrete architecture of the Apollo self-driving platform. The concrete architecture of Apollo was derived using both the Understand tool (which identified static dependencies) and the provided PubSub visualization (which identified PubSub dependencies). This concrete architecture was then compared to our conceptual architecture, and divergencies between the two were analyzed using reflexion analysis. We also explored the concrete architecture of the Perception submodule in particular; here, too, we noted important differences between the conceptual/concrete architecture and offered potential explanations. Finally, we provided two updated use cases that summarize the functioning of the (concrete) system as we now understand it.

Through the process of identifying the concrete architecture of Apollo, our team has gained a better understanding of the differences between a conceptual architecture and a concrete architecture. One important lesson learned from our analysis, is that there are often substantial gaps between the conceptual description of a system and the way in which it is implemented in practice. We were surprised to find that the concrete architecture of Apollo includes dependencies that were completely glossed over in the conceptual documentation. Moreover, it isn't always easy to uncover the reason(s) for these unexpected dependencies. In some cases, we weren't entirely sure why the developers chose to import the files and/or functions that they did. This really highlighted for us the importance of thorough code documentation.

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-QiaochuGuo.pdf