CISC 322

Assignment 2

Concrete Architecture of Apollo

Mar 21, 2022

Group-14: Artemis

Adrian Chu	18anhc@queensu.ca
Alex Baldassare	18cab16@queensu.ca
Randy Bornstein	18rjb10@queensu.ca
Jake Halay	15jmh9@queensu.ca
Kashish Khandelwal	20kmk3@queensu.ca

Table of Contents

Abstract

This report contains a detailed breakdown of the concrete software architecture of the autonomous vehicle software platform known as Apollo[1]. Using Understand, a software analysis program developed by SciTools, and the Apollo GitHub[2] page the concrete architecture was found to have dependencies that differed from that of the conceptual architecture. However, these inconsistencies are explained in several manors and were even to be expected as stated by Murphy, Notkin and Sullivan's[7] software reflection model.

Introduction and Overview

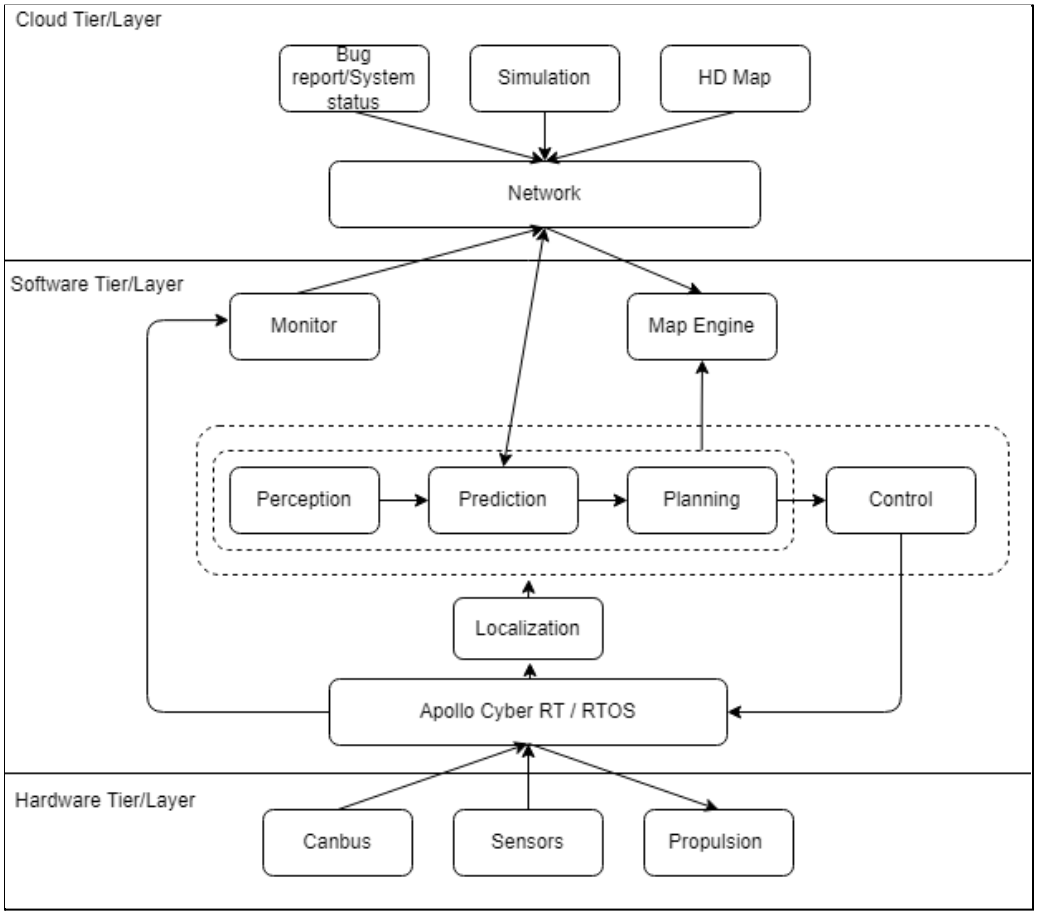
After completing an analysis of the conceptual architecture of the Apollo system in the first report, the concrete architecture of Apollo can begin to be dissected. This report aims to compare the dependencies between the conceptual architecture explored and the concrete implementation. The dependencies of the concrete codebase will be analyzed using Understand, a piece of software by SciTools, which provides a visual representation of the dependencies that exist between components. This visual representation can aid in creating the proper structure of the concrete architecture which can then be used to suggest changes to the conceptual architecture that more accurately describes the concrete implementation. Using this software, an analysis of the high-level architecture can be performed and an analysis of one of the numerous components inside the high-level architecture.

The contents of this report contain an in-depth analysis of the prediction module of the high-level architecture. The prediction module is a key component to aid in the decision-making that Apollo makes, and interacts heavily with other subsystems. This report also contains various diagrams depicting the many aspects of Apollo and how the different components interact with one another. There are several sequence diagrams describing a scenario and the actions that happen between modules within the system to respond to the scenario properly. By combining all the information throughout this report, the differences between conceptual and concrete architecture can be discussed, as well as the reasoning for these discrepancies between the two.

Recap of Conceptual Architecture

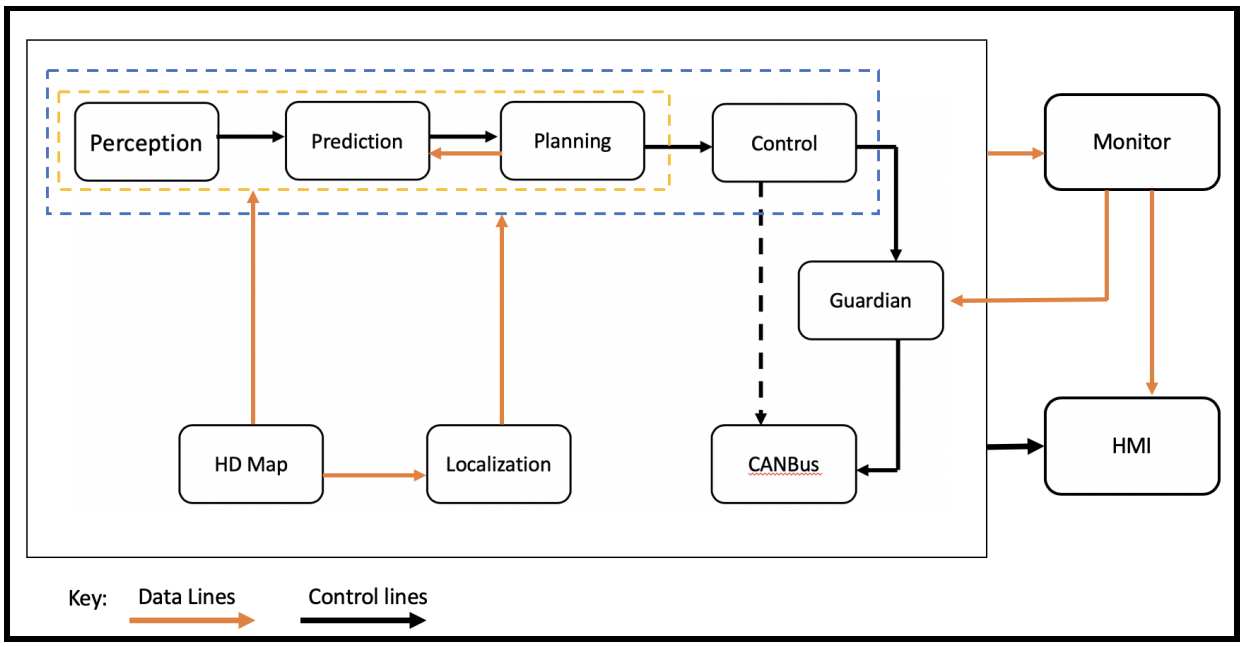
Our group had originally proposed that Apollo's autonomous driving system could be represented by a layered/pipe and filter-style conceptual architecture. [Refer to figure 1]

Figure 1. Initially proposed layered-style architecture



After reconsideration, however, we found that the publish-subscribe architecture (or implicit invocation style) is more suitable due to its loosely-coupled collection of components. [Refer to figure 2]

Figure 2. Proposed publish-subscribe conceptual architecture



In Figure 2, the data and control lines represent relationships between the publishing modules and their subscribers. Modules that receive data lines are subscribed to the publishing subsystem's data output, while modules that receive control lines are subscribed to specific published commands. The perception subsystem performs tasks such as object detection and classification, based on the data published by the HD map module. The prediction module, subscribed to the commands given by perception, estimates how the car will interact with the world in the near future, and vice versa. The planning module interacts with the prediction module by finding practical solutions to these interactions.

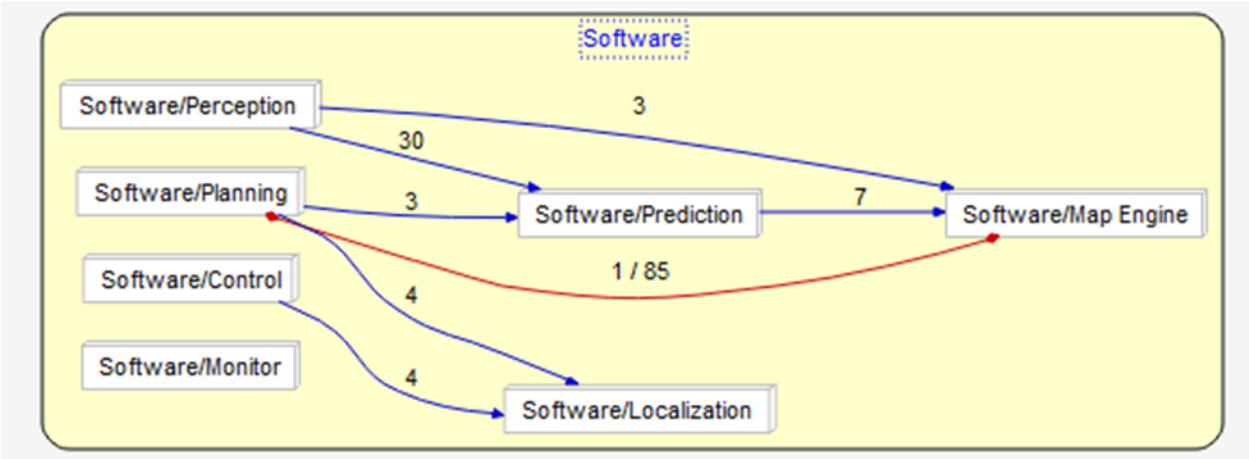
Outside of the group of modules that are subscribed to the HD map data, the control module takes the plans published by the Planning subsystem and relays them to the guardian and CANBus modules as commands to be executed. The guardian makes sure that these commands are safe to execute, and will override them to the CANBus if they are not (eg. in case of an emergency). The CANBus module is the final step in executing the commands given to it by either the control or guardian subsystems.

The monitor subsystem is subscribed to data published by all previously described modules for the sake of tracking the performance and status of the individual components within the system. It then publishes this information to the HMI, or human-machine interface, which relays it to the vehicle operator.

Derivation Process

Since our originally proposed layered/pipe and filter-style conceptual architecture was scrapped, we had to go through the whole process of determining the interaction and dependencies between Apollo's components all over again. This was reviewed by going in-depth of Apollo's source code. Using the Understand 5.1 software tool, we had a brief overview of Apollo's modules and their subsystems. Unfortunately, the source code that belonged to these subsystems was mostly missing. In this case, only the titles of some .cc and header files were given. Hence, our comprehension of the dependencies was limited at this point. This led to Apollo's source code being firstly examined through their entire Github repository with reference to the modules presented in Understand, which after doing so, we were finally able to validate the graph of internal dependencies. This was how we proposed the concrete architecture of Apollo's software system.

Figure 3. Internal dependencies of Apollo's software modules graphed from Understand 5.1



For the uses cases that we have proposed, we followed up on our initially proposed uses cases that were described in our first report on the conceptual architecture. Expanding on the software functionalities of each component, we have narrowed down the use cases on traffic light detection and merging onto a highway to have a more specific view of the publish-subscribe architecture. Again, the Understand tool was used to identify the relevant components and subsystems and trace the methods when the application in the use cases reaches a certain phase.

Concrete Architecture

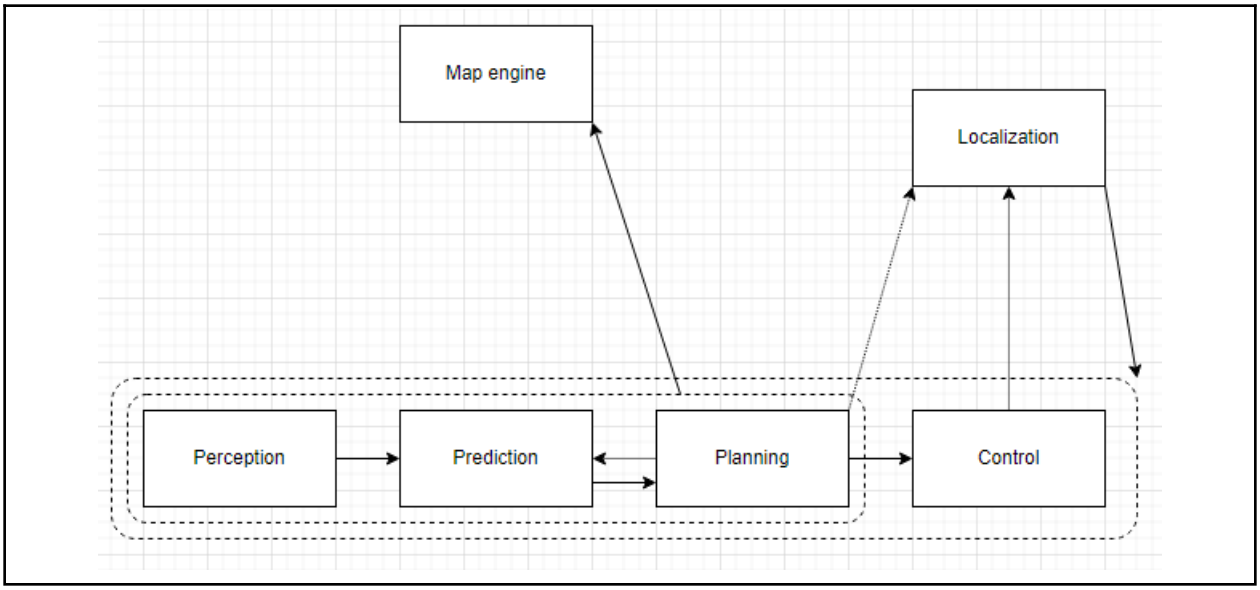
For a brief overview of this section, we will provide a detailed investigation of Apollo's concrete architecture, how it differs from the proposed conceptual architecture, and conduct reflexion analysis on the high-level architecture as well as the prediction module, which is our chosen second-level subsystem.

Finalized Concrete Architecture

As previously mentioned, our initial layered-styled conceptual architecture had to be revised into a new publish-subscribe architecture. Our focus was shifted from a broad overview of the cloud, software, and hardware layers to narrow down to only the software components in Apollo's architecture as it was the basis of the automated system.

Ultimately, this allowed us to settle on the concrete architecture of the Apollo software system, which consisted of the same components such as map engine, perception, prediction, planning, and control. The main discrepancy here is the localization component, which we found to have interacted discreetly with the planning and control components apart from the other software components, as seen from the previous diagrams demonstrating the interactions in the conceptual architecture. This was made evident when looking at the dependencies between modules found in the source code under the control component where code pertaining to the testing while utilizing the localization component was relatively abundant. Hence, we have finalized the concrete architecture of Apollo's software system from this distinction. [Refer to Figure 4]

Figure 4. Finalized concrete architecture of Apollo's software system



Reflexion Analysis

By comparing Figure 4 to Figure 1, we can see that there are divergences in the concrete architecture. One observable change is that the planning module now sends data back to the prediction model, while in the conceptual architecture, their relationship was only one-directional from prediction to planning. This change was likely made because, in practice, the prediction module needs to be used to anticipate the outcomes of the decisions that planning is considering sending to the control module. Without this relationship, the planning module would not be able to accurately weigh its options.

In addition, both the control and planning modules are now dependent on the localization subsystem, when they were not in the proposed conceptual architecture. Understanding that the localization module is responsible for determining the car's location, it is clear why the control and planning modules are subscribed to its data output. Without knowing the car's position in space, these modules would not have enough information to make educated decisions on deciding the vehicle's next moves.

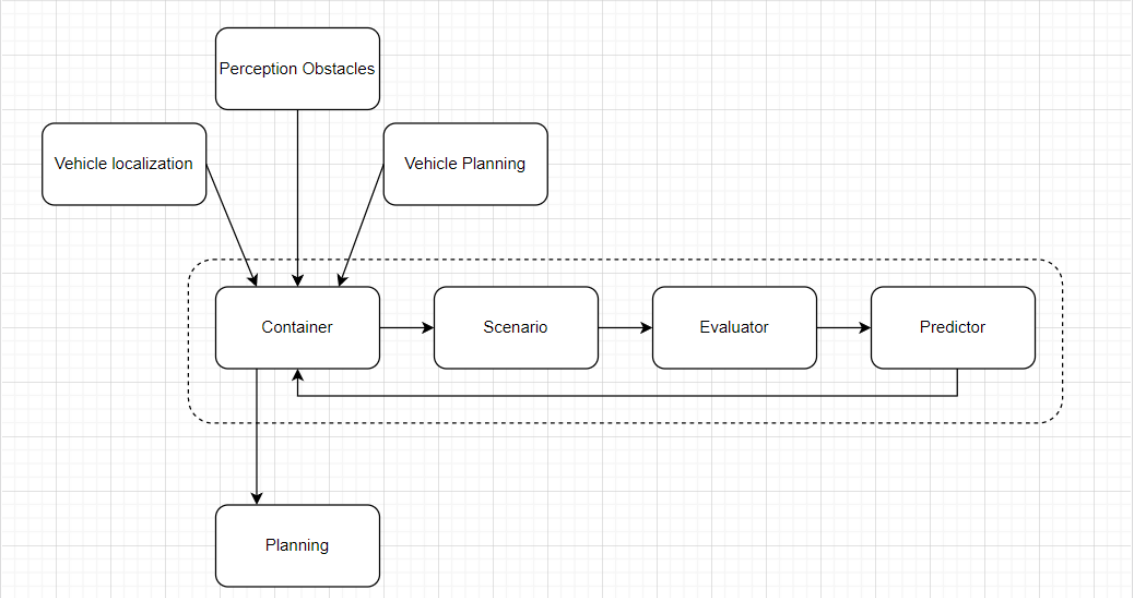
Analysis of Prediction module:

The prediction module is a key module of the Apollo system, providing an integral aspect of autonomous vehicles. This module is responsible for predicting the possible movements and behaviors of obstacles the vehicle may encounter while on the road.

The information output consists of obstacles, their predicted paths, and their priority which can be either ignored, cautious or normal. This information is then relayed back to the planning module to be interpreted, which then decides the best plan for the vehicle.

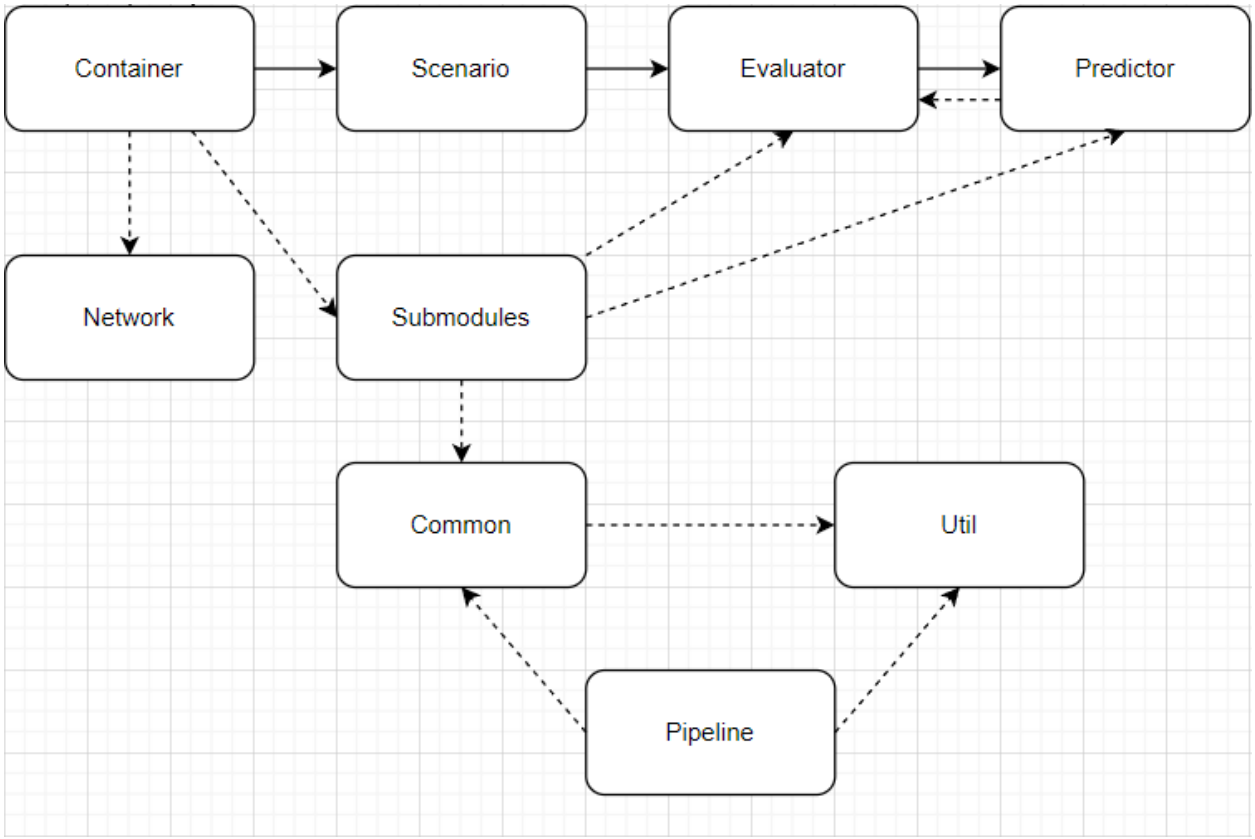
The perception module provides information relating to obstacles in the path of the vehicle, while the planning module provides information on the trajectory of these obstacles. The information from these modules is saved by one of the main components of the prediction module, the container. This information is then passed along the pipeline to the next module, scenario, which analyzes scenarios in which the vehicle could find itself. The evaluator module then predicts paths and speed for each obstacle identified and outputs a probability for each of the paths identified. Finally, the predictor module generates the predicted trajectories for the identified obstacles, as well as their priority level. There are several different outcomes that can be predicted which have been implemented, most of which relate to the movement of an obstacle. Overall, this gives the prediction module a pipe and filter architecture style, as information flows from one module to the next, undergoing transformations in this process.

Figure 5. Conceptual Prediction module architecture



The concrete architecture is relatively similar to the conceptual architecture (as expected). However, there are more modules in the concrete architecture which were not discussed in the conceptual. Most of these discrepancies are a result of the other modules needing ways to communicate with one another, as well as some extra utilities needed. The concrete architecture appears to resemble Figure 6.

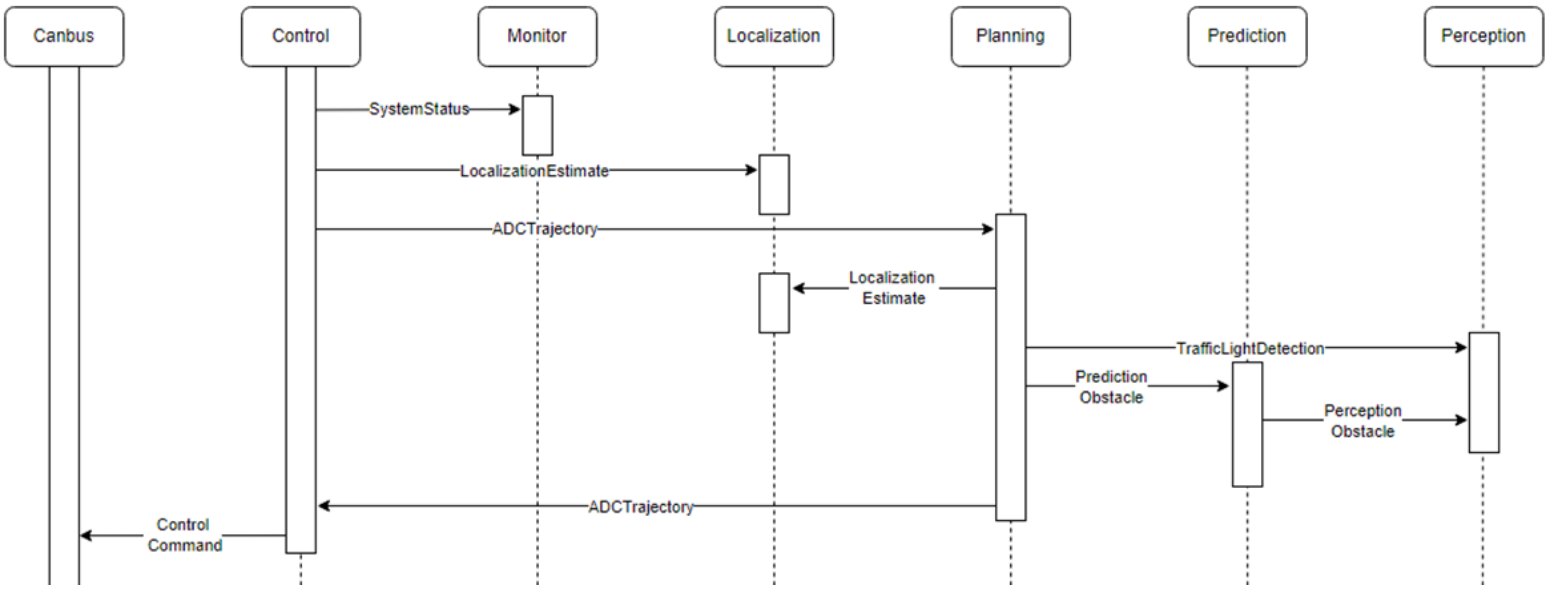
Figure 6. Concrete Prediction Module Architecture



Most of the discrepancies appear to be a result of the evaluator and predictor modules needing additional sub-modules in order to operate correctly. And then these sub-modules have other dependencies that exist even further. The submodules have many dependencies relating to the common module, which contains much of the necessary files for the prediction module to work. Common, as well as Util, contain files that relate to the machine and deep learning aspects of Apollo, which generate many of the probabilities needed for the identified obstacles. These aspects are then used within the evaluator and predictor modules via the submodule component. The pipeline component is mainly an interface between the evaluator component and the map engine component, which is located at a higher level. These two components must interact with one another in order to perform vector net evaluations, which generate short-term trajectory points for obstacles tagged with the “caution” priority level. The final discrepancy is between the container and the network components. The network component contains the necessary files for the RNN evaluator, whose output is then returned to the container component.

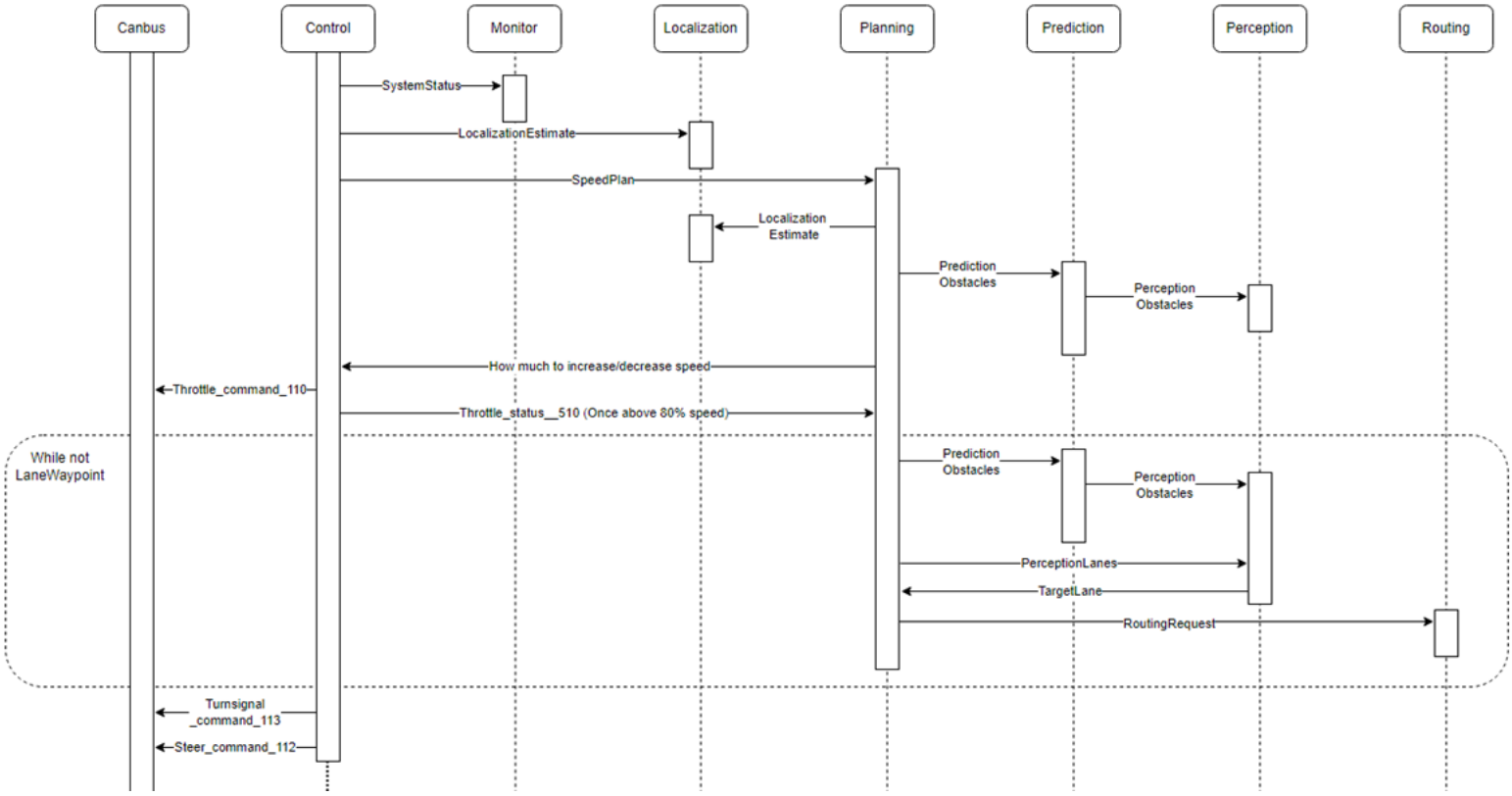
Sequence Diagrams

Sequence Diagram 1: Traffic light detection – Pub-Sub



In this scenario, the system detects a traffic light ahead, and must interpret the lights to decide on its next course of action. The control, which is always running, needs a planning trajectory, the car's status, and the localization estimate as inputs[1] to function. Thus, the control gets this information updated on a constant basis as the environment around the car changes. It first gets the SystemStatus message from the monitor module, then the LocalizationEstimate message by the localization module. Finally, it receives a new ADCTrajectory from the planning module. For the planning module to create the ADCTrajectory, it receives the same LocalizationEstimate message as well as some data from the prediction and perception modules. It receives information about any upcoming traffic lights from the TrafficLightDetection message as well as any information of its surrounding environment from the PredictionObstacle message. It then sends the created ADCTrajectory back to the control module where a ControlCommand is issued to set the trajectory in motion. This is then executed by CANBus as the “CANBus accepts and executes control module commands”[2].

Sequence Diagram 2: Merge onto a highway – Pub-Sub



In the second scenario, the system has to do three things. Increase/decrease its speed to the speed of the highway traffic, activate its turn signal, and merge into an open spot on the highway. First, the control needs to get a SpeedPlan from the planning module as well as the other required inputs. The planning module with the LocalizationEstimate then gets the PredictionObstacle from the prediction module to determine how fast the cars are moving. Planning then sends this information back to the control to issue a Throttle_command_113 according to the CANBus. Once the car has reached a certain speed threshold detected by the Throttle_status_510, it starts looking for an open lane to create a LaneWayPoint. While there is no opening, the planning, prediction, perception, and routing modules are constantly working and communicating to find an open lane for the car to merge onto. Once an open lane is found and it is safe to move, the control issues a Turnsignal_command_113 to the CANBus to activate the blinker and then a Steer_command_112 to the CANBus to execute the merge into the new lane.

Limitations and Lessons Learned

The biggest obstacle for our team during this project was learning to use Understand. This involved identifying and making use of the program's strengths, and overcoming the program's weaknesses. While Understand was instrumental in creating graphs and exploring dependencies within Apollo, the program also had frequent issues with crashing and other bugs. Furthermore, Understand was unable to provide us with all the information we were looking for within Apollo, such as the specific functions and methods that the components use to communicate with one another. From this, we learned that in order to derive the most accurate representation of Apollo's concrete architecture, we needed to both use Understand and manually search through the Github repository. Doing only one of these would not give us all the information we needed.

Another obstacle that we encountered when looking through Apollo's files in its Github repository was that the bulk of the code was written in C++, a language with which some of our group members were not familiar. We overcame this by working together and increasing our communication as a team.

Conclusions

Now after we have studied and deeply analyzed the concrete architecture of the Apollo software, our group found that it has some differences (on comparing our original thought from the first deliverable) when it comes to the elements and their inter-communication involved in the system. This new architecture model presented several unexpected dependencies, which demonstrated that the subsystems are more dependent than what we originally assumed. However, the organization of the subsystems in its publish-subscribe style is set in a way to maximize interactions between each module(s) as much as possible, as well as be reconfigurable with ease. These observations carried over to the in-depth conceptual architectures of the subsystems, demonstrated by our analysis of the subsystems involved in it. (Although each submodule could be having a different architectural style)

References

[1] Open Platform. (2020). Apollo. https://apollo.auto/

[2] Apollo Auto (2021) Apollo [Source code]. https://github.com/ApolloAuto/apollo

[3] Apollo Auto (2021) Control Module [Source code]. https://github.com/ApolloAuto/apollo/blob/master/modules/control/README.md

[4] Apollo Auto (2021) Canbus Module [Source code]. https://github.com/ApolloAuto/apollo/blob/master/modules/canbus/README.md

[5] Baidu Youtube playlist on smart cars: YouTube. (2022). Baidu Smart Transportation. YouTube. Retrieved March 2022, from https://www.youtube.com/playlist?list=PLJPY3CV1nyNzG1NcN1V-Shi36b1Mffp-_.

[6] Understand (5.1). (2020). [Software Analysis platform]. Scitools. https://www.scitools.com/?pk_vid=18e3b2a2ca5ee0f516479110045af56e

[7] Murphy, G., Notkin, D., & Sullivan, K. (1995). Software reflexion models: bridging the gap between source and high-level models (pp. 18–28) [Review of Software reflexion models: bridging the gap between source and high-level models]. ACM SIGSOFT Software Engineering Notes.

--End of Report--