CISC 322
Assignment 2

Concrete Architecture of Apollo's Open Software Platform

Monday, March 21, 2022

Apollo-gies in Advance

Cameron Beaulieu	19cgb@queensu.ca
Truman Be	18tb18@queensu.ca
Isabella Enriquez	18ipe@queensu.ca
Josh Graham	18jg48@queensu.ca
Jessica Li	19jal@queensu.ca
Marc Kevin Quijalvo	18mkq@queensu.ca

Table of Contents

Abstract

This report focuses on the concrete architecture of Baidu's Apollo, an open-source autonomous driving platform. It delves into the concrete architecture of the system as a whole, as well as that of the Prediction subsystem.

The concrete architecture our team derived shows that the system follows the Publish-Subscribe architectural style, as our conceptual architecture had, in combination with the Object-Oriented style, which was absent from our conceptual architecture. All 12 modules included in the conceptual architecture still exist in the concrete architecture, along with one addition, the Common module. However, while the subsystems remain largely the same, the interactions between them in the concrete architecture differ significantly from those in the conceptual architecture, in that the interactions are more complex. In our separate exploration of the Prediction subsystem, both the conceptual and concrete architectures we derived follow the Pipe and Filter style. The conceptual included four submodules — Container, Scenario, Evaluator, and Predictor—and followed a linear flow (a maximum of one pipe, or connection, going in, and one pipe going out), while the concrete added the Base and AI Model submodules, as well as many more pipes.

The discrepancies between the conceptual and concrete architectures for both the Apollo system and the Prediction subsystem were investigated using the software reflexion framework, the findings of which—including the rationale for unexpected dependencies and subsystems—are detailed in this report. Additionally, we outline our derivation process for Apollo's concrete architecture and revisit the two use cases we previously explored with our updated understanding of the architecture. Finally, our report concludes with our key findings, as well as noting prominent limitations and lessons our team has learned thus far.

Introduction

Launched in 2017, the Apollo project is an open-source autonomous driving platform that gives developers and other stakeholders the power to develop, test, and deploy autonomous vehicle projects with ease (Watkins, 2018). Since launched, it has become arguably the largest open-source autonomous driving platform in the world, with powerful global partners such as Microsoft and the Ford Motor Company (*Apollo*), as well as more than 200 individual contributors (*Apollo's GitHub Page*).

Following our first report, in which our team dove into the conceptual architecture of the Apollo system, this report looks at the system's concrete architecture. Derived through investigation and analysis of source code, the concrete architecture ended up being significantly different from the conceptual architecture we proposed in our first round of research. While the concrete architecture still held up with our proposal of the Publish-Subscribe style in our first report, our team realized it also exhibited the Object-Oriented style. In addition, the concrete architecture included all of the conceptual architecture's 12 subsystems—Perception, Prediction, Planning, Routing, Control, Guardian, CANBus, HD Map, Localization, Storytelling, Monitor, and HMI—but also introduced the Common subsystem, which serves as a library shared by all the other subsystems. The most distinct difference, however, was seen in the interactions between subsystems, in which our concrete architecture had significantly more. While the presence of discrepancies was to be expected, as this is generally the case in practice, our team

further investigated these changes using the software reflexion framework to uncover the rationale behind them.

To dive even deeper into Apollo, our team investigated the Prediction subsystem by constructing both conceptual and concrete architectures for this inner system. Both were found to follow the "Pipe and Filter" architectural style, however, discrepancies existed again between the architectures. The conceptual architecture included four submodules—Container, Scenario, Evaluator, Predictor—while the concrete architecture introduced the Base submodule in addition to the original four, as well as more interactions between the submodules (called "pipes" in the pipe and filter style). Though the changes between the architectures were lesser in number than that of the system as a whole, our team applied the software reflexion framework once again, detailing the rationale and other findings in this report.

Beyond concrete architecture derivation and reflexion analysis for both the top-level system and a second-level system, our team reviewed the use cases we had detailed in our initial report: (i) making a left turn on a green light, and (ii) changing lanes on a highway. With our renewed understanding of Apollo, we updated these use cases to more accurately reflect the concrete architecture.

Finally, we conclude by summarizing our key findings, as well as detailing any noteworthy limitations and lessons we learnt in the research and writing of this report.

Derivation Process

The derivation process of the concrete architecture consisted of examining the documentation and source code provided on GitHub in combination with an analysis of overall file structure on Scitools Understand. We began by examining folders and nested files to determine their function and relation to the modules discussed in our conceptual architecture. Based on nested folder and file names, we grouped files relevant to any specific module together where appropriate to derive the system's concrete architecture. With respect to files not originally found in any module folders (e.g., common and tools folders), we examined lower-level folders and concrete code to appropriately group it with a specific module. By the end, we were left with a reasonable concrete architecture, while leaving behind a host of files with no accurate placement in the system. We deliberated and decided that the set of files not able to be attributed to a single module alone were to be grouped as the Common module that is accessed by various other modules, as a library of sorts. Lastly, using our derived architecture, we examined the Publish-Subscribe communications occurring in the system, and combined them together to have a detailed final concrete architecture that also depicts the data flow between modules.

High-Level Architecture

Overview

After performing the above derivation process, we ended up with the below high-level architecture (Figure 1), which we concluded to exhibit a Publish-Subscribe style, similar to our conceptual architecture, as well as an Object-Oriented style. The Publish-Subscribe style was still apparent due to the large number of subscriptions between the various submodules, shown using the red arrows below. In terms of the Object-Oriented aspect of the architecture, our derivation process added a number of new

dependencies which did not align with the subscriptions within the system. As well, through our analysis of individual files, we noticed many dependencies were handled through importing classes from other modules, which would further suggest that there is an Object-Oriented aspect to the system.

In terms of subsystems, we maintained all the subsystems from the conceptual architecture, while adding one new system called Common, which acts as a shared library that all the other subsystems are dependent on. However, while we did not add new subsystems other than Common, we did add some new functionality to the modules, described below.

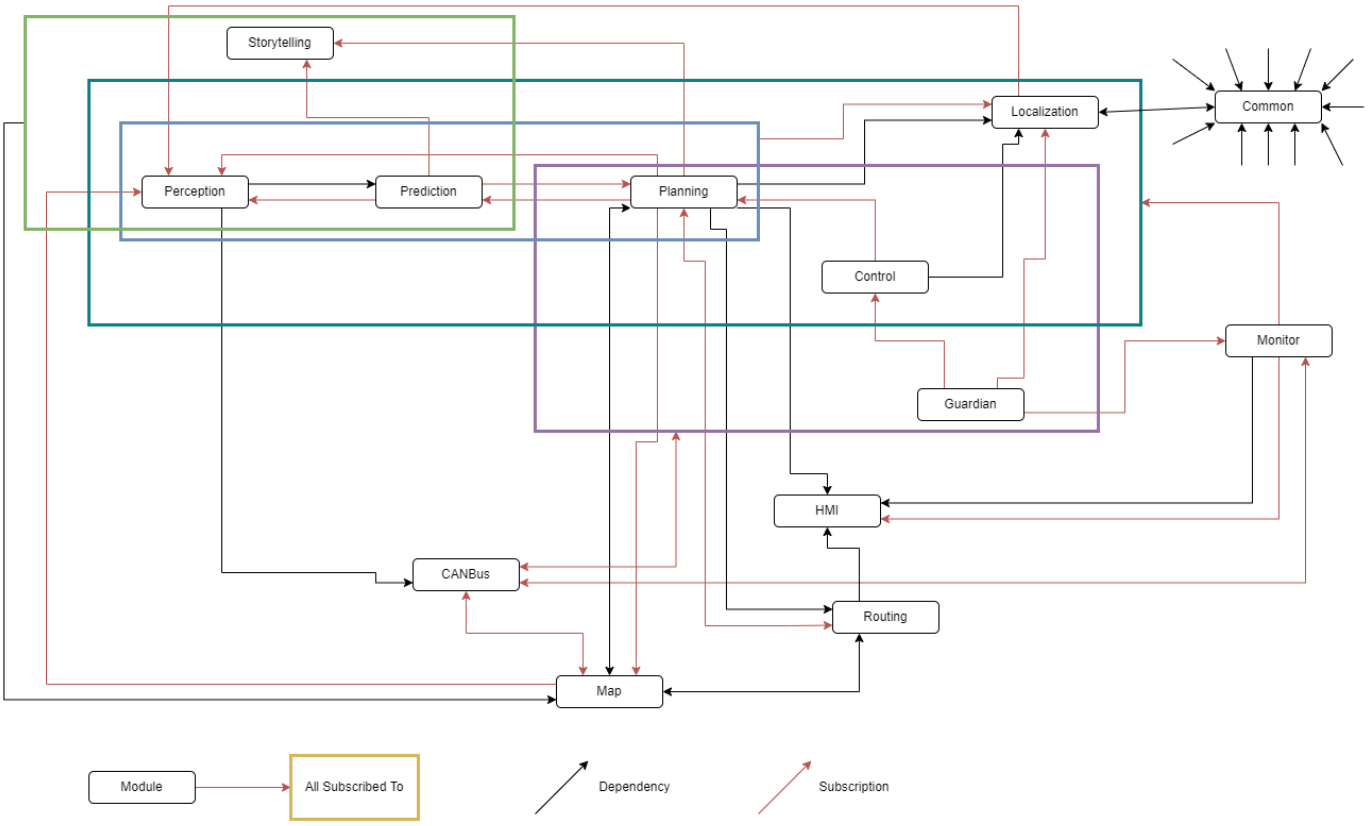

Figure 1: Concrete High-Level Architecture

We will now briefly dive into each subsystem, looking at its functionality, subscriptions, and data dependencies within the overall system. Note, that unless specified otherwise, all subsystems are dependent on the library resources provided by the Common subsystem.

Perception

The Perception subsystem is still responsible for utilizing data from various cameras, as well as the LiDAR systems, to recognize, classify, and track obstacles, exporting this information in the form of a tracking list. The Perception module contains all files related to image detection sensors/devices, including camera, video, etc. (as well as the associated drivers), the preprocessing of images, involving various deep learning algorithms, and the

postprocessing of images, including files relating to the tracking of objects. The Perception module is subscribed to the Map and Localization modules for the current map, as well as the localization estimate. Lastly, the Perception subsystem depends on CANBus and Prediction for the current chassis status and predicted trajectories.

Prediction

The Prediction subsystem continues to hold the functionality of estimating the behaviour of all perceived obstacles within the tracking list created by the Perception subsystem, including both traffic lights and other objects. For more details on this subsystem, see below.

Planning

The Planning subsystem is responsible for planning a safe and collision-free trajectory for the car. To do this, it contains submodules utilized for trajectory calculations, scenario handling, based on the scenarios created from the Storytelling module, and other files related to planning of the car's required actions, such as lane changes, speed adjustments, etc. The Planning module subscribes to the tracking list from Perception, the obstacle predictions from Prediction, the chassis status from CANBus, localization estimate from Localization, map from Map, scenarios from Storytelling, and has data dependencies to Routing and Localization.

Storytelling

The Storytelling subsystem remains as a high-level Scenario Manager, responsible for cross-coordinating submodules through the creation of various scenarios. The goal of providing these scenarios is to align the car on working towards the "story" deemed optimal. This module contains the files related to the "storytelling functionality" and is subscribed to no other modules, while being dependent only on Map and Common.

Routing

The Routing subsystem continues to be responsible for generating the optimal route, given a start and endpoint, including the lanes and roads required to reach the destination. The Routing module contains various files and subsystems related to graph calculations, strategy, and a task manager which is required for managing the computations required to build the optimal route, depending on the identified scenario (dead end, three-point turn, etc.). Finally, the Routing module is subscribed to the Planning subsystem, listening for any routing requests, and is dependent on the HMI and Map modules.

Map

The Map subsystem generates three different maps used by other subsystems (HD Map, PNC Map, and Relative Map), providing them to other modules when required. The module contains files related to the generation of the maps, requiring dependencies to Planning and Routing for the current planned route, utilized to generate the PNC and relative map. Finally, the Map subsystem is subscribed to the CANBus subsystem for the current chassis status.

Localization

The Localization subsystem still utilizes two different modes to aggregate data and locate the autonomous vehicle, through utilizing various sensors (GPS, LiDAR, and inertial). The

Localization module contains files related to the GNSS and its associated drivers, as well as files required in creating the localization estimate of the car. Localization has only one subscription: the Perception module, specifically related to the sensor calibration files located within the drivers contained within the Perception module.

Control

The Control subsystem is responsible for generating the control commands, including steering, acceleration, and braking, to operate the car to execute the planned trajectory of the car. The Control subsystem contains files related to latitudinal and longitudinal calculations, as well as pre- and post-processing files required to create control commands. Finally, the Control module subscribes to the Planning subsystem for the planned trajectory of the car, the CANBus module for the chassis status, and depends on the Localization module for the localization estimate.

Guardian

The Guardian subsystem, similar to the conceptual architecture, acts as a safety system for the car, and overrides the control commands created by the Control module in the event of the Monitor subsystem detecting a module failure. As well, the Guardian module subscribes to the CANBus module for the chassis status, the Monitor module for the module statuses, the Control module for the current control commands, and finally the Localization module for the localization estimate.

CANBus

The CANBus subsystem executes the control commands from either the Control module or the Guardian module, provided the Guardian module has overridden control, as well as collecting and publishing the chassis status of the vehicle. The CANBus module contains various vehicle-specific files for executing control commands, as well as communication and client files for receiving the commands. Lastly, the CANBus module subscribes to the Map module for the map data, the Planning module for the current trajectory, the Control and Guardian modules for the control commands, and the Monitor module for the module statuses.

Monitor

The Monitor subsystem still tracks the status of all the other modules, detecting the event in which any module failure occurs. The Monitor module contains files related to monitoring the software subsystems, as well as files for monitoring the hardware of the car. It is subscribed to Perception, Prediction, Planning, Control, Localization, CANBus, and HMI, all to track their statuses, and has a dependency on the HMI module.

Common

The Common subsystem acts as a library for the other modules, offering various tools such as math libraries, statuses of important functions, the vehicle state, etc. The Common module subscribes to no other modules, but has a dependency on the Localization module for the current localization estimate.

Reflexion Analysis

After completing our concrete architecture (Figure 1) our group began to analyze the discrepancies with our proposed conceptual architecture (Figure 2). We performed a

reflexion analysis by investigating the divergences between the two architectures to determine what caused them to occur. We have outlined below what each divergence was and why it occurred, organized by module.

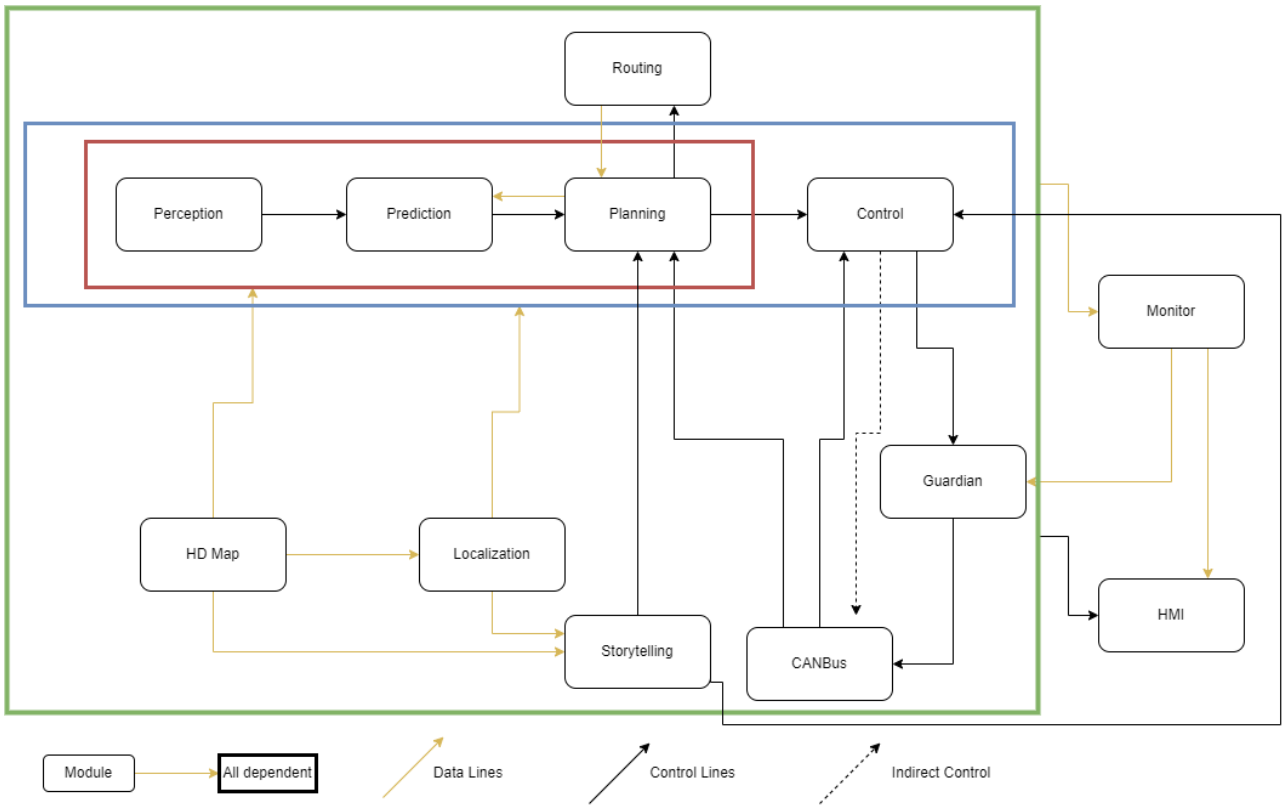

Figure 2: Conceptual High-Level Architecture

Perception

The Perception module was found to have dependencies to the Prediction and CANBus modules, which were not shown in our conceptual architecture. Perception utilizes the Obstacle class from the Prediction module to help identify what obstacle it is perceiving. The CANBus module is used for communication and publishing of messages, as well as to initialize the sensors used for Perception.

Prediction

The Prediction module only had one new dependency uncovered through our concrete architecture which was Storytelling. Prediction subscribes to the Storytelling module, using information from planned scenarios to improve the accuracy of its predictions.

Planning

The Planning module has dependencies to the Perception and HMI modules that were not originally seen in our conceptual architecture. The Perception module publishes the tracking list of perceived obstacles. We assume that originally the Planning module relied on the HMI module to obtain map information, however, this dependency seems to have been replaced by directly depending on the Map module.

Storytelling
The Storytelling module has no new dependencies from the concrete architecture, however, we found that it is not dependent on the Localization module. Storytelling is only dependent on the Map module for data on the car's current position on the map.

Routing
The Routing module has two new dependencies that were discovered with the concrete architecture, HMI and Map. The HMI dependency originally was used to obtain information about the map, however, this dependency seems to be unused now, having been replaced by dependencies to the Planning and Map module.

Map
The concrete implementation of the Map module has four new dependencies: Perception, CANBus, Planning, and Routing modules. The Perception and CANBus modules are dependencies mainly through their relation to the Map module as a result of the Publish-Subscribe communication processes; the Map module subscribes to the Perception module to gather information of its physical environment for geographic data, and subscribed to the CANBus module for necessary chassis information. The Planning module is seemingly an outdated dependency, with not enough critical use of the module to justify a divergence. Lastly, the Routing module is a dependency of the Map module as it provides the optimal route to the Map module.

Localization
The Localization module has a new dependency to the Perception module. This dependency is based on the Publish-Subscribe communication, with the Localization module subscribing to the Perception module to retrieve sensor calibration configurations. One dependency that differs is the Map module, which is nonexistent in the concrete architecture, likely since the Map module is subscribed to other modules that provide information about the vehicle's position and details.

Control
The Control module did not have any new dependencies, though instead had some divergences with respect to missing dependencies, including the dependencies on the CANBus module and Storytelling module. The onboard communication systems allow for communication of essential information (e.g., chassis status) directly without the need for a dependency.

Guardian
The Guardian module has one new dependency through the Localization module. This particular connection is the result of the Publish-Subscribe nature of the two modules, as the Guardian module subscribes to the Localization module for an estimate of the vehicle's immediate location.

CANBus
The CANBus module is subscribed to three new dependencies found in the concrete architecture: Map, Monitor, and Planning. The Map module publishes map data, providing the CANBus with accurate information regarding its position in the environment. The

Monitor module publishes status information of the other modules, and the Planning module publishes the current trajectory that the car must follow.

Monitor

The Monitor module was found to have one new dependency in the concrete architecture, the HMI module. User interaction information is provided from the HMI to the Monitor to modify module configurations. When creating our conceptual architecture, we believed that the Monitor module was dependent on every module except the HMI. We have discovered through creating the concrete architecture that this is not the case. The Monitor is not dependent on the Routing, Map, Storytelling, or Guardian modules. This is because these modules do not make decisions or observations related to the vehicle's current trajectory, obstacles, or speed. This implies that these modules cannot cause an immediate safety concern and any error within these modules will be found and reported by a higher priority module.

Common

The Common module is a module that was completely new when creating our concrete architecture. Every other module in the system depends on Common for various tools such as adapters, configurations, math functions, and various other utilities that are ubiquitous throughout the system. Common is only dependent on the Localization module, which it uses to determine the current localization of the vehicle.

Low-Level Architecture: The Prediction Subsystem

Overview

To dive further into the architecture of Apollo, our team investigated the Prediction subsystem, including determining its own architecture and submodules. This subsystem's main purpose annotates obstacles around the vehicle with forecasted trajectories—including probabilities for an obstacle's path and speed—and assigned priorities. In the context of Apollo, an obstacle priority is the level by which the vehicle needs to be cautious of that obstacle. Priority levels include: "ignore," in which case the obstacle can safely be ignored; "caution," where an obstacle has a high possibility of interfering with the vehicle's trajectory; and "normal," the default priority value in which case that obstacle does not fall under the prior two priorities.

Similarly to what we did with the high-level architecture, our team derived a conceptual architecture for Prediction (Figure 4), primarily based on documentation provided in the Apollo GitHub repository. Our team then constructed a concrete architecture for the subsystem (Figure 3), utilizing the Understand tool and investigating source code to understand the purpose of each of Prediction's submodules, as well as how they interact with the other submodules.

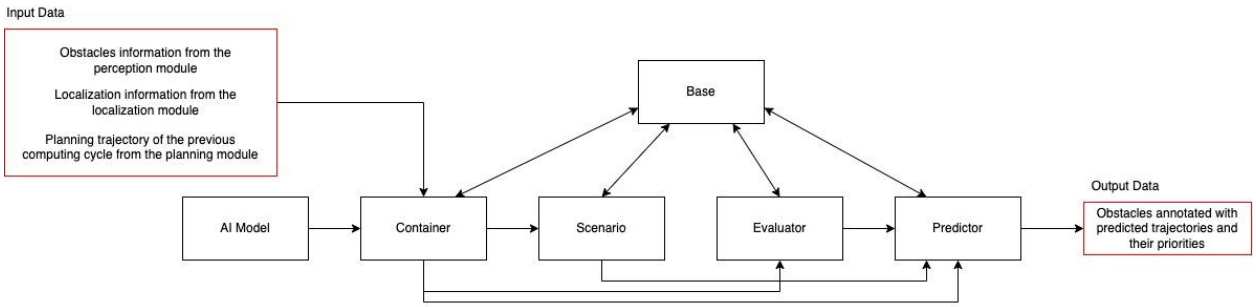

Figure 3: Concrete architecture of the Prediction subsystem

Both our conceptual and concrete architectures concluded that the Prediction subsystem followed the Pipe and Filter style, which is used for applications that conduct a series of independent operations on data, typically with some order imposed. For the concrete architecture specifically, we deduced six main submodules, referred to as "filters" in this style (save for AI Model, which acts as a "source"): Base, AI Model, Container, Scenario, Evaluator, and Predictor. Each filter takes in input data, applies an operation to that data, then outputs the new data, passing it along to the next filter using a pipe (in the case of Predictor, the output produced is not passed to another filter and is the final output for Prediction). In the following subsections, we take a look at each of the submodules individually to get a better understanding of the subsystem as a whole.

Base

The Base submodule consists of common functions, methods, and components used across the Prediction subsystem. As such, pipes exist both ways between this submodule and the four main submodules. These submodules pass in data that needs to be transformed using a function, method, or component within Base, and receive it back with the transformations applied.

AI Model

The AI Model submodule contains the neural network model for the Prediction submodule. This model is used to adapt to the rapidly changing input (the vehicle's dynamic environment and trajectory) and produce accurate predictions. This submodule is unique from its siblings in that it does not have any pipes feeding into it, and instead only one pipe coming out from it. Henceforth, this submodule is referred to as a "source" rather than a "filter."

Container

The Container submodule can be considered the first filter in the architecture. It stores the neural network from the AI Model submodule, as well as the input data: obstacles from the Perception module, vehicle localization from the Localization module, and the vehicle planning trajectory of the previous computing cycle from the Planning module. Each obstacle is assigned its own container, created, registered, and managed by the Container Manager within the submodule.

Scenario

The Container submodule passes the data it has stored to the Scenario submodule. In this filter, the data is analyzed to determine the current type of scenario, which is either the

"Cruise" scenario—including lane-keeping and following—or the "Junction" scenario—which has to do with intersections, with traffic lights and/or STOP signs. Additionally, the data is used to determine the priority for each obstacle, either "ignore," "caution," or "normal" (the default value), as explained previously.

Evaluator

The Evaluator submodule also receives data from the Container submodule. It is responsible for evaluating the trajectory and speed probabilities of each obstacle. There are 10 different evaluators, each of which calculates probability differently—such as with certain functions or machine learning models—or calculates probability for certain scenarios or obstacles—such as the Social Interaction evaluator, which specifically evaluates pedestrians prioritized at the "caution" level.

Predictor

The Predictor submodule, which is the final filter, produces the output data for the Prediction subsystem. It receives data from Container, Scenario, and Evaluator. Using this data, Predictor generates the final forecasted trajectories for each of the obstacles, annotated with their respective priority. This final output is used mainly by the Planning module to determine the safest trajectory for the vehicle, preventing any collisions with obstacles in their current and predicted states. Similar to Evaluator, Predictor has nine different types of predictors, used for different cases such as obstacles moving freely, obstacles moving along a single lane, etc.

Reflexion Analysis

Conceptual Architecture

In order to perform a reflexion analysis, we compared our concrete architecture to a conceptual architecture derived from the Apollo documentation for the Prediction module (Figure 4). By looking at the documentation, we can see that conceptually, the Prediction module has a linear dependency graph. The output from one module simply goes to the next module, until it reaches the final Predictor submodule, where it generates a final predicted trajectory for all obstacles, which is then published for other modules in the system to use.

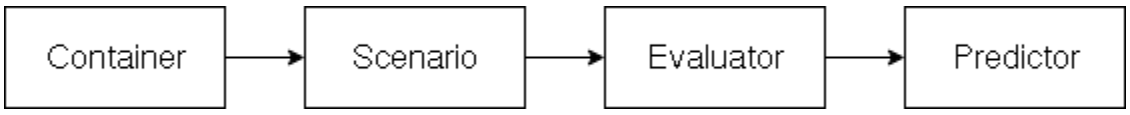

Figure 4: Conceptual architecture of the Prediction subsystem

Connections with Outer Components

Conceptually, the Prediction subsystem collects data from the Localization, Planning, and Perception modules. The conceptual architecture shows that the Container is mainly responsible for collecting this data and providing it to subsequent submodules in the pipeline. Although these modules communicate with the Prediction module via publish-subscribe with the Container, our concrete architecture also shows a new dependency on the Perception module. The Evaluator has a new dependency on the Perception module, because the Semantic LSTM evaluator requires the LiDAR sensor data directly. Apart from this, we also find that the Predictor submodule is not responsible

for data publishing. Prediction data is sent back to the Container, which then publishes the data.

AI Model

For our concrete architecture, our team decided to add a new component that cannot be found in the conceptual architecture. This module is dedicated to the various AI models used across the Prediction component. In the conceptual architecture, the Evaluator submodule is the main user of the different AI models, however, in our concrete architecture, we find that the AI models serve as a data source for the Container module instead, in addition to the Localization, Planning, and Perception modules.

Container

As stated previously, the Container gained a new dependency on the AI models. Previously, the conceptual architecture did not show this dependency due to the lack of a dedicated AI Model component. We also find that the Container now acts as a dependency for the Evaluator and Predictor submodules as well, as opposed to just being a dependency only for the Scenario submodule. These new dependencies will be outlined in their own sections.

Scenario

For the Scenario submodule, there are no new dependencies. We find that this submodule still only relies on the data provided by the Container submodule in order to determine which scenario the vehicle is in. In terms of outgoing connections, the Scenario submodule is no longer a dependency for the Evaluator submodule. This will also be explained below.

Evaluator

The Evaluator previously relied on the Scenario submodule in the conceptual architecture. In our concrete architecture, we find that the Evaluator relies on the Container module, and not the Scenario module. Our concrete architecture shows that this pipeline is no longer linear. This is because the Evaluator just needs the Container's obstacle data and input from the AI Model in order to determine future object behaviour. In reality, it does not need information about the scenario in order to calculate it. As mentioned previously, there is a new dependency on the Perception module, which is a higher-level module. The Evaluator needs the LiDAR data directly for the Semantic LSTM evaluator. For outgoing connections, the Evaluator still acts as a dependency for the Predictor module and does not act as a dependency for any new submodules.

Predictor

The conceptual architecture shows that the Predictor only relies on the Evaluator for the potential object paths. However, our concrete architecture shows that it depends on a lot more than just the Evaluator. First, there is a new dependency for the Container. The various predictors (used to detect whether an object is moving freely, or moving along a lane, for example) require input data directly from the Container. This includes obstacle data provided by the AI Model and the Perception module. Secondly, the Predictor also depends on the Scenario submodule. When the Predictor submodule publishes data about object predictions, it injects the scenario that each object is in. This is done by calling the Scenario Manager.

Diagrams

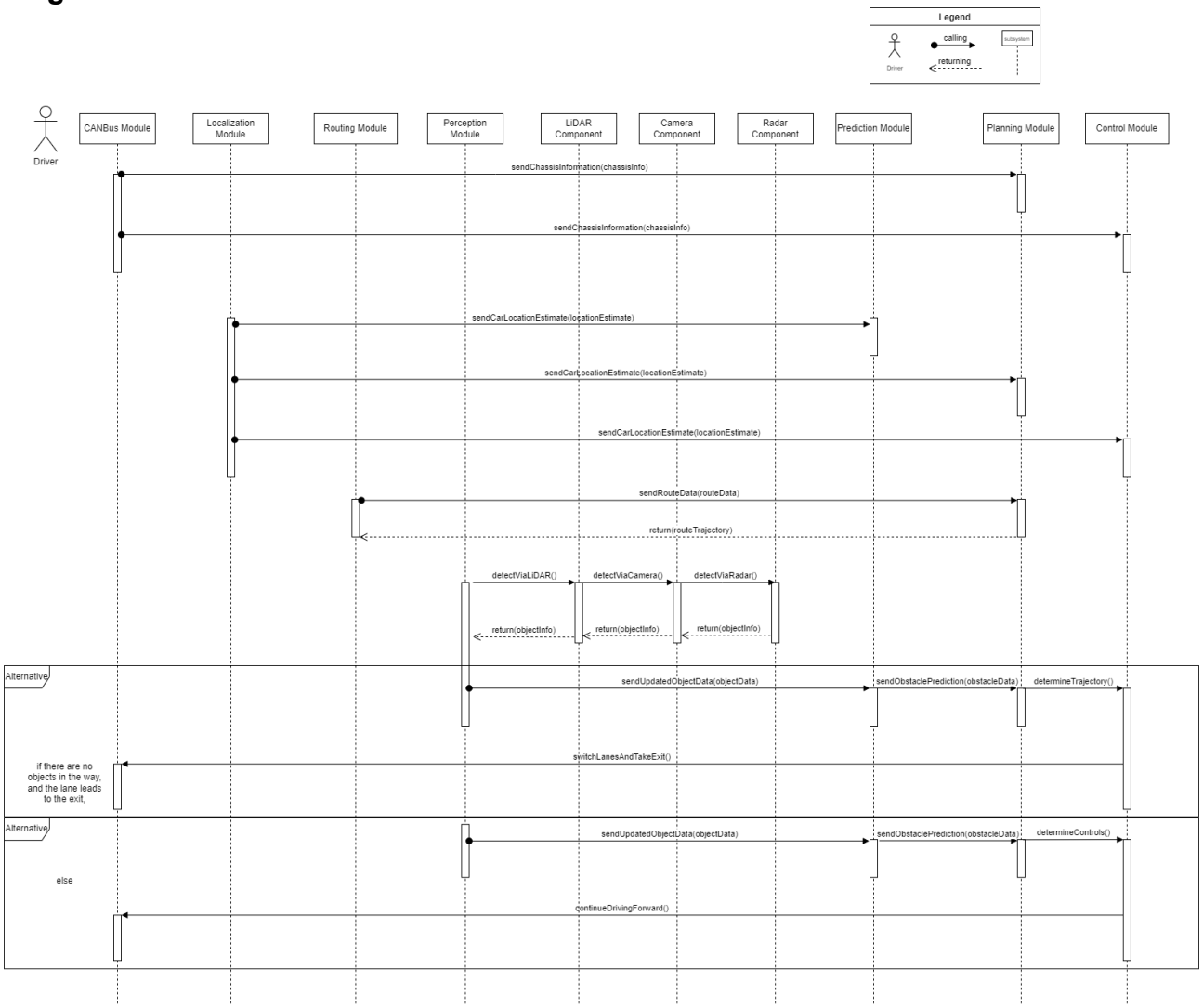

Figure 5: Sequence diagram for a use case involving a lane change at a highway exit

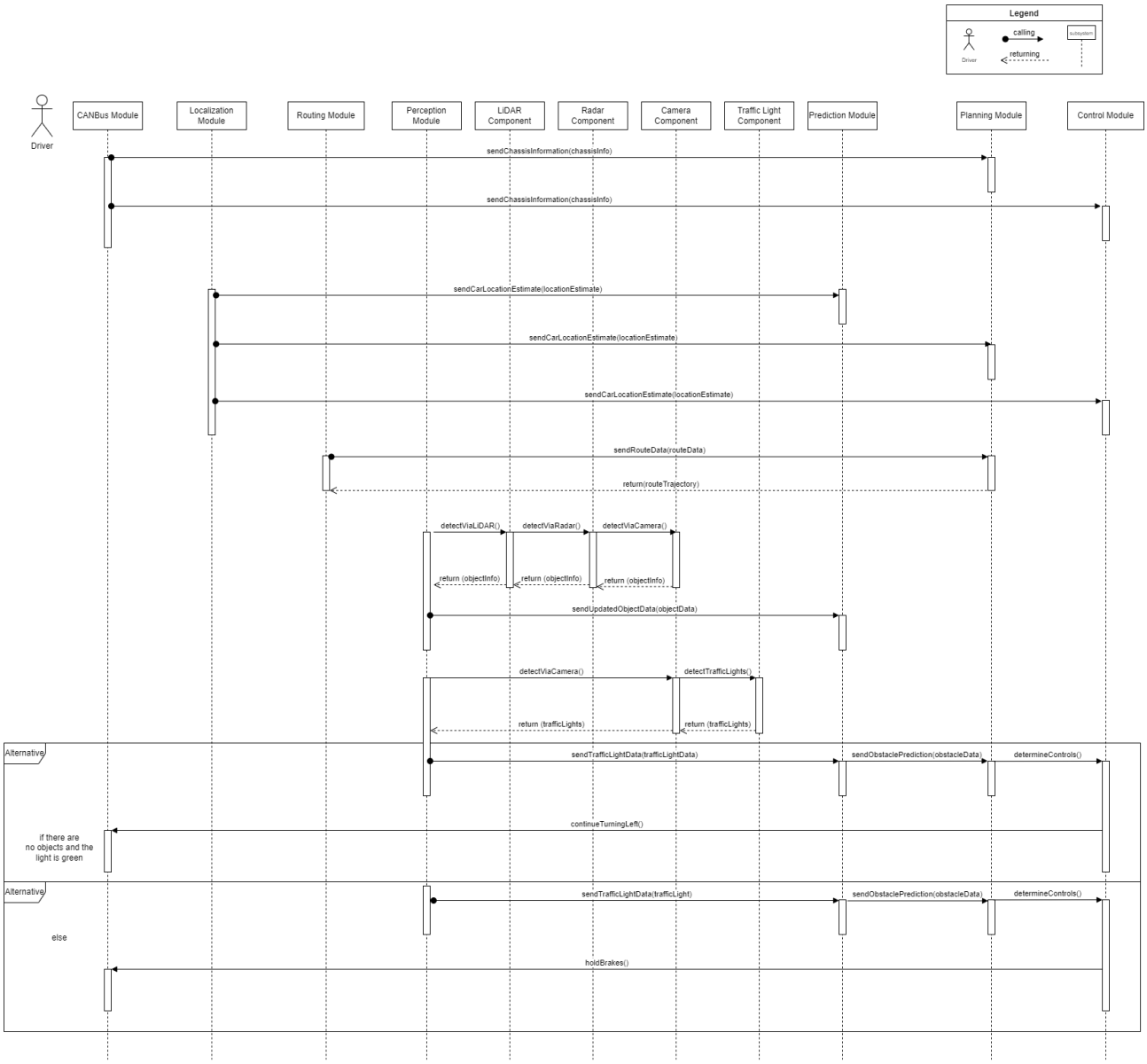

Figure 6: Sequence diagram for a use case involving a left turn at a stoplight

Use Cases

The first use case (Figure 5) details the scenario where a user is exiting off of the highway. To perform this successfully, the car must check its surroundings to make sure it is clear before changing lanes and merging off the highway. The Apollo system starts with the CANBus module publishing chassis information to be used by the Planning and Control modules. The Localization module also publishes location estimates to be used by the Prediction, Planning, and Control modules. The Routing module and Planning module are both subscribed to each other, the Routing module starts by sending over the route data and the Planning module responds by returning the trajectory of the car's path. Following this, the Perception module obtains information from its components (LiDAR, Camera, and Radar) to perceive the car's surroundings. If there are no obstacles, the car is free to proceed and the Perception module publishes the updated object data from the components to the Prediction module. This module predicts the trajectory of nearby

objects and publishes this to the Planning module, which publishes the car's trajectory to the Control module. From here, the Control module will publish instructions to the CANBus to exit from the highway. In the situation that there is an obstacle that prohibits the car from exiting the highway, the Control module will publish instructions to the CANBus to continue driving forward.

The second use case (Figure 6) details the scenario where a user is making a left turn at a stoplight. In order to make a successful left turn, a car must first check if the traffic light is green and then only proceed if there is no incoming traffic or pedestrians. Similar to the first use case, the CANBus, Localization, and Routing module will start by publishing information to their subscribed modules and providing information related to the car's chassis, location, and route. The Perception module then captures information through its LiDAR, Camera, and Radar components regarding the status of the car. This updated data is published to the Prediction module and then the Perception module uses its Camera and Traffic Light components to get the data regarding the status of the traffic light. The Planning module then publishes all of the data to the Prediction module which continues to publish to Planning and then to Control to determine the actions which the car should take. In the scenario that the light is green and there are no obstacles, the Control module will publish instructions to the CANBus to proceed with the turn. Otherwise, the instructions will be to continue holding the breaks.

Conclusion

In conclusion, through our derivation process of both utilizing Scitools Understand and analyzing the GitHub repository, our group discovered the concrete architecture of Apollo's Open Software Platform was built upon 13 subsystems, resulting in an architecture which contains elements of both a Publish and Subscribe architecture, as well as an Object-Oriented architecture. We then dove deeper into the Prediction subsystem, creating both the conceptual and concrete architectures of the subsystem, both architectures containing four submodules, all working in tandem as a Pipe and Filter architecture. With the architectures completed, it allowed us to perform reflexion analyses on both the high-level architecture, as well as the architecture of the Prediction subsystem. Through this analysis, we identified various divergences between the conceptual and concrete architectures, such as new and missing module dependencies. Lastly, with this process having strengthened our understanding of the Apollo architecture, we explored two different use cases to better illustrate the interaction of the subsystems during the system's use.

In conclusion, our team has greatly enjoyed diving into the code base to derive the concrete architecture of the Apollo Open Software Platform and has shown us how common it is for concrete architectures to differ from conceptual architectures.

Limitations and Lessons Learned

Apollo's architecture is far from simple and involves a variety of complex, intertwined components. Given that the Understand software was incapable of mapping Publish-Subscribe interactions between modules, we had to cross-reference our architecture with the graphical representation of interactions between modules in order to piece together our final concrete architecture. Additionally, the lack of documentation within

the repository made it difficult for us to identify the purpose of various modules and files as well as make those connections with other dependencies.

Through investigating the concrete architecture of Apollo, we further discovered the importance of communication within our team. We found success in having every team member derive the concrete architecture themselves before meeting to discuss our rationale and piece together our final architecture. This allowed us to all develop a thorough understanding of Apollo's architecture and work more efficiently in putting together our findings. For the future, we plan on continuing to hold regular meetings in which we will communicate with each other and delegate tasks to collaborate effectively.

Data Dictionary

Autonomous vehicle: A vehicle that can drive and perceive its environment without a human controlling it.

Conceptual architecture: A very high-level organizational view of a system, highlighting modules and the relationships between them.

Concrete architecture: The structural make-up of a piece of software, containing modules and their interconnections

Module/Subsystem: A bundle of code composed of many related functions contributing to a common feature.

Object-Oriented: An architecture style involving data representations and their associated operations encapsulated in abstract data types referred to as objects

Open-source: A code-base that is publicly accessible to be used or contributed to.

Pipe and Filter: An architectural style involving a series of independent computations performed on data. Each computation inputs and outputs streams of data, enabling consequent computations.

Publish-Subscribe: An architecture style involving loosely-coupled collections of modules. Each module is responsible for an operation, which may enable other modules in the process.

Naming Conventions

GNSS: Global Navigation Satellite System

GPS: Global Positioning System

HMI: Human-Machine Interface (DreamView)

LiDAR: Light Detection and Ranging

PNC Map: Planned Course Map

UI: User Interface

LSTM: Long short-term memory

References

Baidu. (n.d.). Apollo. Retrieved February 12, 2022, from https://apollo.auto/

GitHub, Inc. (2017, July 2). *Apollo's GitHub Page*. GitHub. Retrieved February 12, 2022, from https://github.com/ApolloAuto/apollo

Watkins, D. (2018, April 20). *Autonomous Car Platform Apollo Doesn't Want You to Reinvent the Wheel*. Opensource.com. Retrieved February 15, 2022, from https://opensource.com/article/18/4/apollo-open-autonomous-vehicle-platform