February 20th, 2022

Apollo Conceptual Architecture Report

Authors

Kennan Bays (18khb5@queensu.ca - #20167866)

Mason Choi (18gc23@queensu.ca - #20165470)

Joshua Kouri (18jak16@queensu.ca - #20158940)

Alexander Nechyporenko (15ann2@queensu.ca - #20021759)

Joshua Tremblay (20jkt1@queensu.ca - # 20247327)

1. Abstract

This report serves to present our concrete architecture of the Apollo system and analysis of the differences between our proposed conceptual architecture and concrete architecture. We will discuss our updated conceptual architecture, provide some sequence diagrams relating to the system and breakdown the subsystems in Apollo. Furthermore, the conceptual and concrete architecture of the second-level subsystem, the control module, is provided and analyzed for any divergences. Similarly, to the previous report, we encountered additional limitations and have lessons learned as a result.

In our previous report, we stated that the architecture style is process-control. However, our new findings are that Apollo follows a publish-subscribe architecture style. This report will reflect this update in conceptual architecture. The process-control architecture style can be more specifically applied to the control module. This module manages the control commands that the vehicle would apply and processes inputs that have been sent from other modules. The use of the Understand tool allowed us to create and analyze both concrete architectures. This tool marked the dependencies between modules so we could understand how they all interact with each other. Unfortunately, Understand cannot display the message traffic that comes from the publish-subscribe architecture. So, the graph visualization of that traffic that was provided to us filled in those missing gaps.

2. Introduction

The intent of this report is to explore the concrete architecture of Apollo Auto, an autonomous vehicle driving software, and detail the process of how our team derived our final concrete architecture and revised our conceptual architecture. This report consists of five sections. Firstly, this report details the derivation process. In the second section, the updated conceptual architecture and its comparison to the concrete architecture will be discussed. In the next section, the concrete architecture will be discussed. The fourth section covers subsystem architecture and their interactions. In the final section, the lesson learned from this report will be discussed. Following this final section are the conclusions, references, and terminology.

In the previous report, our group believed that the structure of Apollo followed a process control architecture. Upon examining the concrete architecture of Apollo, however, our group realized that the style was actually a publish-subscribe architectural style. Our initial conceptual diagram was generated through extensive reading of documentation provided on the Apollo GitHub repository.

Our derivation was driven by the insight provided by the Understand tool and the documentation on the Apollo GitHub repository. Understand allowed us to organize components into a dependency graph, which was further divided into two types of dependencies. These two types of dependencies were legal and illegal dependencies. Legal dependencies are logical connections between components that exist for comfortable reasons, such as resource sharing. Illegal dependencies exist due to an error in the Understand tool, incorrectly categorized file(s), or the design flaw by a developer. Illegal dependencies are examined and removed. Any unexpected

legal dependencies have been analyzed and explained in Section 5.2. Some insignificant or obvious dependencies have been omitted from the discussion.

The sequence diagrams were created by extensive research of the documentation. We did a combination of research regarding specific use cases and choosing functions which would be most responsible for certain functionalities.

3. Derivation Process

Analyzing not just the Apollo documentation but the code snippets interlacing the documents, as well as examining specific pure-code files, our ideas of the conceptual architecture and construction of the sequence diagram were vindicated. However, some aspects of our conceptual model were modified upon a deeper inspection of the studied software. Primarily, our group discovered that our assumed architectural style was incorrect. The software, rather than following a process-control architecture which was assumed due to the existence of the Control module, follows a publish-subscribe architectural style.

To resolve the differences between the conceptual model and the concrete model, dependencies which existed but were not accounted for were mapped onto the previous conceptual model. No modules were unaccounted for and thus, no modules needed to be added. No dependencies between modules had to be removed as they were all correct or unexpected two-way.

Understand provided significant insight into unexpected dependencies. Components that weren't precisely subsystems, such as the Common module, were integrated into our understanding of the system. The slew of dependencies which rely on Common are numerous and will show the amount of files potentially affected by modifications to the Common module. Additional components, such as Control and Misc Drivers, had their relation clarified where documentation did not exist. Overall, Understand was very significant to furthering our understanding of the Apollo software.

To generate the sequence diagram, our previous model, generated alongside the conceptual report, was traced. At each operation, the equivalent function(s) were located in the documentation. It was challenging to produce the equivalent method call due to the sheer amount of calls and determining where function calls, which spanned across dozens of folders, were located in the documentation. It also was clear which functions were receiving the output of functions.

4. Conceptual Architecture

4.1 Updated Conceptual Architecture

Before we began comparing the conceptual and concrete architecture of this software, we updated our proposed conceptual architecture. We now know that the architectural style of Apollo is publish-subscribe, rather than our original proposal of the process-control style. Originally, we had chosen process-control because Apollo has a Control module we had assumed to be the controller of the system, which took variables from several other modules. However,

after reviewing the publish-subscribe architectural style as well as the concrete architecture, we see that it is more fitting for the overall software, while process-control only better fits the Control subsystem.

The software of Apollo consists of multiple modules, which each have their own operations to carry out and end up enabling other operations in the process. When an event is triggered by a module, it implicitly invokes procedures within the other modules in the system through a broadcasting system, which in this case is the Cyber RT component. For example, when the Sensors module detects a traffic light, it triggers an event that invokes a procedure within the Traffic Light Perception module, which then invokes the Prediction module, and so on. Meanwhile, the sensors can continue to analyze the environment around them without waiting for the other modules to finish their procedures, which is highly necessary in order to keep the car functioning safely.

Therefore, our conceptual architecture has been updated to have the architectural style changed from process-control to publish-subscribe.

4.2. Comparing Conceptual to Concrete

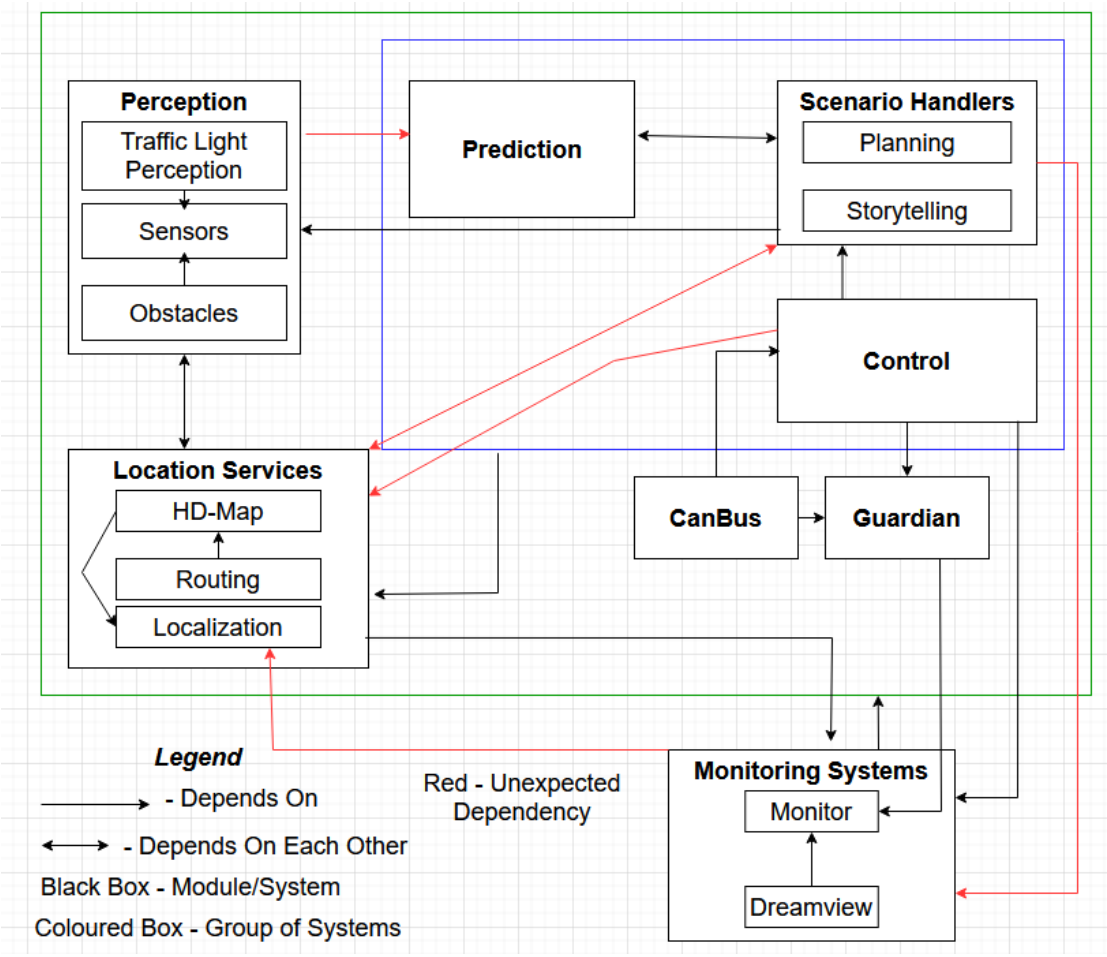

Figure 1: Conceptual Architecture following changes taken from the Concrete Architecture

5. Concrete Architecture

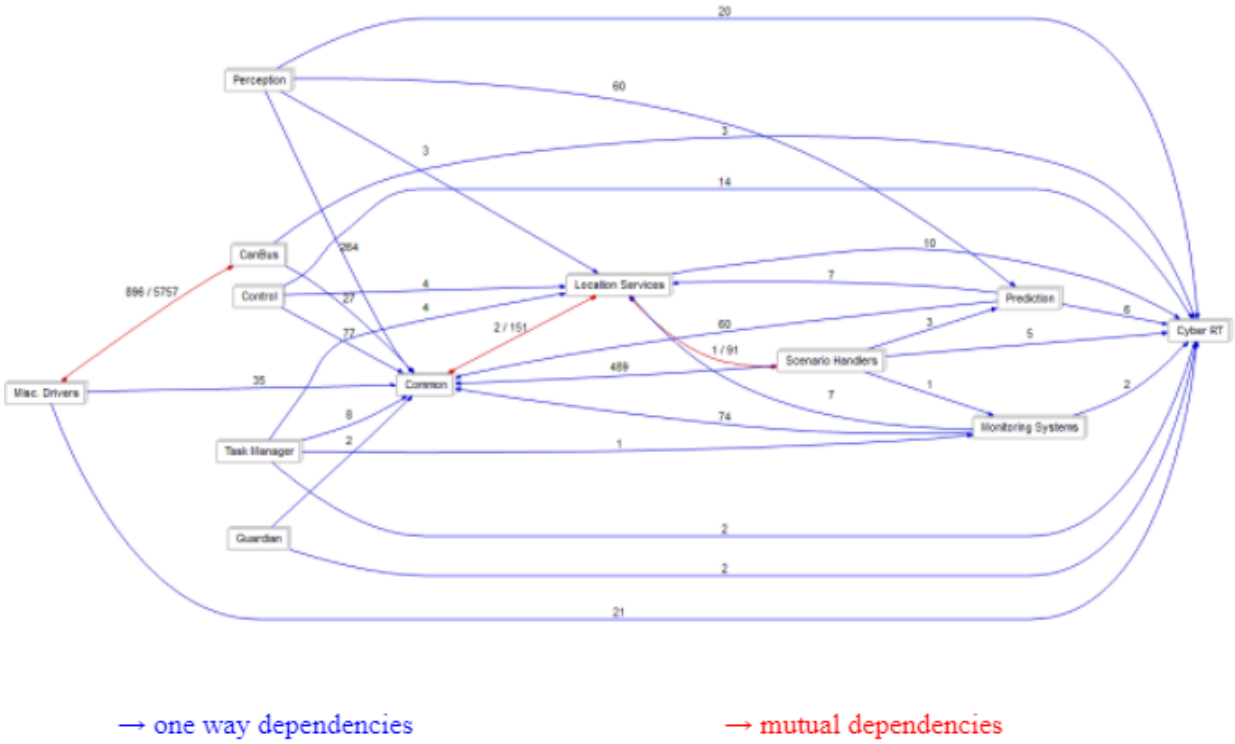

Figure 2: Apollo's Concrete Architecture

From the extracted dependency file of Apollo's software system, we were able to create this concrete architecture. This was done by creating the subsystems based on our conceptual architecture, then sorting the modules as best as we could into them. Some general modules, such as "common", became subsystems of their own, as well as anything that could not be sorted into a specific subsystem using the information we currently have.

5.1. Reflection Analysis

Referring below, we will explore the unexpected dependencies found within the concrete architecture that differs from the conceptual one we created in the original report. Given that the conceptual architecture did not include the central component of Cyber RT and that this component connects to all other components, I will not go into detail of it as it will be redundant. Suffice it to say, Cyber RT is the central framework through which the concurrency network is generated between the subsystems of Apollo. Given the Publish-Subscribe nature of the architecture, Cyber RT acts as a connector of modules that can be added or removed based on the need of the system.

Scenario Handlers → Location Services (Mutual dependency)

File Dependency Description: There is a mutual dependency between files localization_gflags.h and files within the common folder of the planning sub-module within the Scenario Handlers module.

Analysis of Dependency: Exploring this dependency in depth, the localization module attempts to indicate the location of the vehicle. It is quite logical then that the second level subsystem for the planning of the vehicle would need to use the location variables in order to identify possible planning routes for the vehicle. If we examine the dependent file from the localization module, named localization_gflags.h, we will find that it primarily consists of constant variables that indicate all of the relevant information for the localization of the vehicle. The planning sub-module then uses this data to make the relevant calculations based on the provided information.

Control → Location Services

File Dependency Description: From the Control module, we have one of many files within the controller folder that uses the include mechanism to call on Location Services. Looking at a specific file, we have the controller.h file calling the localization.pb.h file.

Analysis of Dependency: Looking at the specifics of this dependency, we see that the controller.h file is retrieving the current status of the vehicle from an orientation standpoint. Meaning, trying to estimate its current positioning and the status of its surroundings. The controller module impacts the output of the system based on the inputs that is fed to it, one of those inputs is this status update that indicates on how the vehicle should adjust its output or calculations.

Perception → Prediction

File Dependency Description: In the conceptual, we made the dependency being from Prediction to Perception, however, it is actually the other way around. There are many dependencies from the Perception module calling on Prediction, but if we focus on the Lidar object detection files, and more specifically, the evaluator_manager.h file, we find one perception call. It calls semantic_lstm_evaluator.h.

Analysis of Dependency: The perception module for object detection using Lidar technology, it needs to analyze its surrounding based on the data feedback from the sensor. This specific call makes use of LSTM neural networks in order to make a prediction the vehicle detected based on the information found from the Lidar sensor.

Monitor Systems → Localization

File Dependency Description: Same as before, dependency is the other way around from the conceptual to the concrete. Monitor depends on Localization, where map_service.h calls map_service.h.

Analysis of Dependency: From a monitoring perspective, this file utilizes the HDmap in order get the status of the vehicle on the road by analyzing different elements, such as the nearest lane positioning, checking routing based on lane type, create a new path for routing, etc. Meaning, that by retrieving relevant information of the cars location and status relative to its environment from HDmap, it then utilizes this information to display the status of the car and to provide information to the driver via the user interface the module Dreamview provides within Monitor Systems.

Scenario Handlers → Monitor Systems

File Dependency Description: If we look at open_space_roi_decider.hfrom the Planning module within Scenario Handlers, it calls file map_service.h within the Dreamview module which is found in Monitor Systems.

Analysis of Dependency: From the open space decider file, the logic is trying to determine the space surrounding its vehicle in order to determine different planning scenarios that the vehicle may find itself in. The map_service in particular gives the information of the cars positioning within the lane and the calculated routes that the vehicle may take. Hence, working in conjunction, the planning module determines its spacing and what the course of action is based on the routing, and other information provided by the monitoring system.

5.2. Sequence Diagrams

(See *Figure 3*) During normal operation, the TLPM repeatedly queries the HD-Map for any potential upcoming traffic lights. If any are detected, the TLPM requests image data from the camera sensor and proceeds to process it for any visible traffic lights. If a traffic light is detected, the TLPM then attempts to identify its state, then sends both its position and state to the Prediction module. The Prediction module then feeds its prediction data into the Scenario Handler modules, and the resulting instructions are sent to the Control module. The Control module then enacts the required changes (such as slowing down for an amber or red light) through the CanBus module.

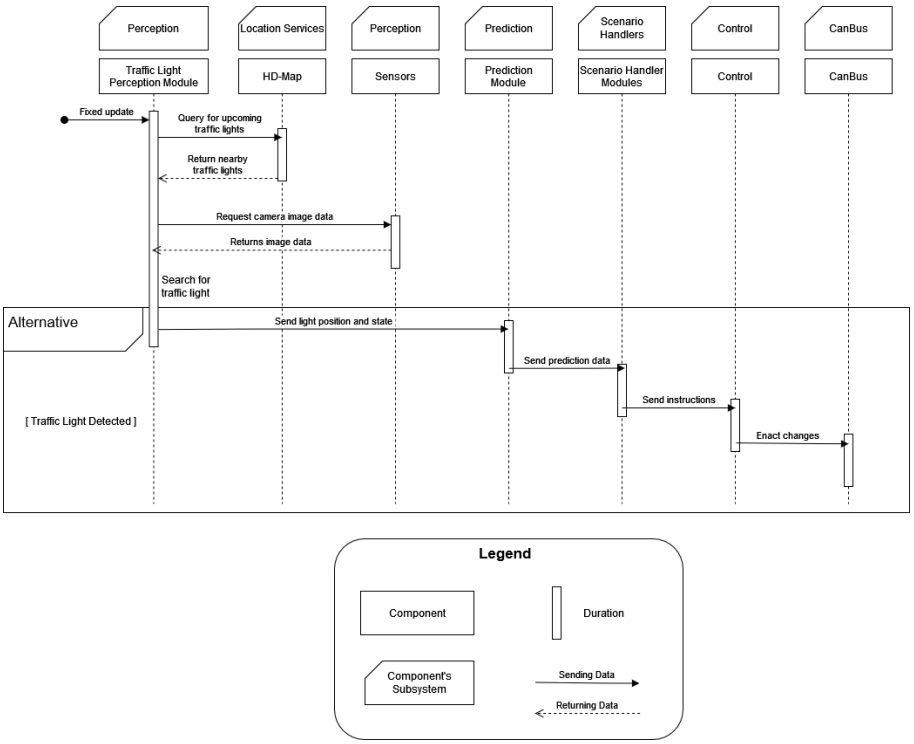

Figure 3: Update Sequence Diagram showing the use-case when the system scans for and reacts to any approaching traffic lights

(See *Figure 4*) At all times, the Monitor module is running and watching over all other modules. If it detects that a module has crashed, it triggers the Guardian module, which then disables the main Control module and takes it place. The Guardian module then checks sensor statuses to determine how quickly the vehicle should be stopped; if no immediate obstacle is detected, the vehicle will gradually be brought to a stop, but if an immediate obstacle is detected, the vehicle is rapidly slowed down for safety reasons.

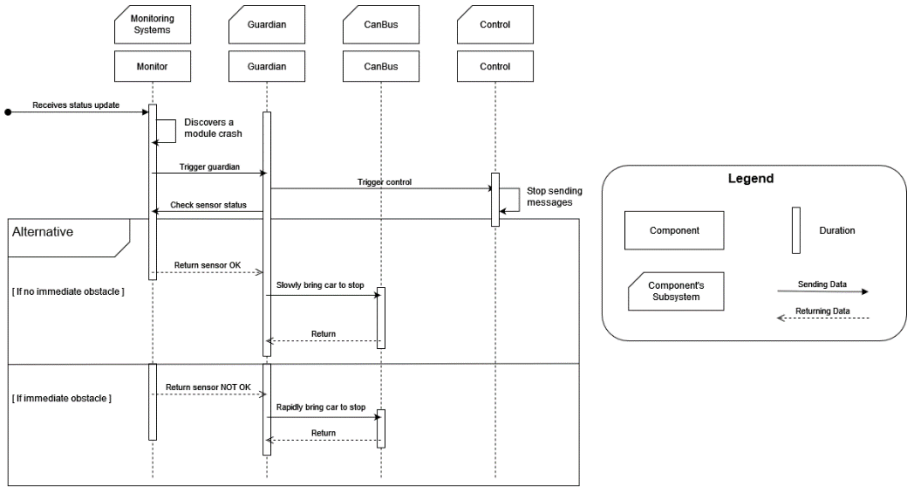

Figure 4: Update Sequence Diagram showing the use-case where the Guardian kicks in after a module crash to stop the vehicle

5.3. Subsystem Breakdown

5.3.0. Common, Misc Drivers and Task Manager

These components were added following an examination of the source code using Understand 5.1. Three modules prudent to mention are the Common, Misc Drivers, and Task Manager modules. These modules do not function as subsystems. However, they are significant pieces of the architecture and thus deserve to be mentioned. Common is an assortment of code not specific to any module yet critical to the functioning of Apollo. Misc Drivers are an assortment of drivers used to communicate with the hardware components of the vehicle. Task Manager, as the name suggests, helps manage events occurring throughout the software.

5.3.1. Perception

The Perception subsystem provides the context for all subsequent decisions made by Apollo by gathering information about the physical surroundings around the vehicle. The Perception subsystem contains three main submodules; Traffic Light Recognition, Obstacle and Sensors [5].

Traffic Light Perception Module (TLPM): The TLPM relies on cameras, a telephoto and wide-angle camera, to gather information in the forward direction of the car. As the car drives, the TLPM repeatedly queries the HD-Map module for upcoming traffic lights. If the module detects a traffic light, the traffic light will be projected onto the images captured by the cameras, allowing the vehicle to create a 3D environment from a 2D picture. Once a traffic light is detected, the TLPM will identify the state of the traffic light. The state of a traffic light could be in any of the following states: red, yellow, green, black, or unknown (unable to detect). The TLPM has a dependency on the Sensors module to send information to project the traffic light onto. However, it is important to note that the TLPM is only dependent on the camera sensors, since the other modules do not detect the light emanating from the traffic light [5].

Obstacle: The Obstacle module is used to detect, classify, and track objects. This module will attempt to predict the position and velocity of detected obstacles. This module is dependent on the Sensors module for information which is required to detect and track objects [5].

Sensors: The Sensors module is a collection of multiple unique perceptive hardware which collect information about the vehicle surroundings. Thus, the Sensors module is, to an extent, a bundle of smaller modules. The perceptive hardware consists of five cameras, two radars, and four LiDAR’s (Light Detection and Ranging). The output of these devices is used to recognize obstacles and combine their individual tracked objects into a final tracking list (list of objects to track) [5].

5.3.2. Prediction

The Prediction module using information gathered by the Perception and Scenario Handler modules to predict the behaviour of detected obstacles. The Prediction module has four

functionalities; *Container*, *Scenario*, *Evaluator*, and *Predictor*. *Container* functionality stores data about perceived obstacles, vehicle localization, and vehicle planning. The *Scenario* functionality takes the data in *Container* to check whether the vehicle is to continue cruising or encountering a junction, which impacts the predicted behaviours of objects around the vehicle. The *Evaluator* functionality predicts the path and speed for objects around the car. Finally, the *Predictor* functionality predicts the trajectory for objects around the vehicle. The summation of these functionalities will be projections of the trajectories of all objects around the vehicle [5].

5.3.3. Scenario Handlers

The Scenario Handlers are a pair of modules which govern how the vehicle reacts to the situation it is in. Scenario Handlers receive information from the Prediction module to process and create a course of action for the vehicle to react to in an attempt to get to the location provided by the Location Services. The two Scenario Handlers are the Planning and Storytelling modules.

Planning: The Planning module aims to create a collision-free trajectory while progressing towards an end goal. The Planning module requires the Routing output to decide where to go, Traffic Light Perception module from perception, and the Prediction output. Thus, while the module is dependent on many other modules, the Planning Module is only dependent on a fraction of the output of the modules it is dependent on. The Planning module addresses how to act by creating “Use Cases” for all driving environments. This allows the developers to tweak reactions to specific environments without affecting other driving behaviour [5].

Storytelling: Certain environments are too complicated for the Planning module to address. Due to the Planning module heading in a modular development approach, a sequential based approach to these scenarios are no longer viable. The Storytelling module is used to handle the complex scenarios that the Planning module is subpar at addressing. The Storytelling module creates “stories” (complex scenarios) which can trigger reactions from other modules [5].

5.3.4. Control

The Control subsystem receives inputs from the Scenario Handler modules which it processes and sends to the CanBus. The Scenario Handler modules generates trajectories for the vehicle to travel. Using the car’s current status and the trajectory information, the Control system generates the commands which will be forwarded to the CanBus.

5.3.5. Location Services

The Location Services modules are used to understand where the vehicle is in relation to the world. Location Services contains three submodules: HD-Map, Localization, and Routing.

HD-Map: The HD-Map (High-Definition Map) module is repeatedly queried by the system for information about the surrounding road systems. An example use of this information is the projection of a traffic light from a 2D picture to a 3D environment. HD-Map has a dependency on the Localization module for information about the exact location of the vehicle.

Localization: The Localization module pinpoints the exact location of the vehicle. The Localization module will calculate the vehicle location by using either the RTK (Real Time Kinematic) based method or the multi-sensor fusion method. The RTK based method requires information from the GPS (Global Position System) and IMU (Inertial Measurement Unit). The multi-sensor fusion method also requires GPS and IMU to locate the vehicle but incorporates LiDAR into the calculation.

Routing: The Routing module generates navigational routes to reach an end location from a start location upon request. This module is dependent on HD-Map to provide a significant amount of information about roads, traffic lights, and other road data. The Routing module uses this information to calculate the optimal route to reach the end location.

5.3.6. CanBus

The CanBus subsystem provides an interface for the Apollo software to communicate with the hardware. Additionally, CanBus passes chassis information to the software systems. The CanBus is a critical piece of infrastructure in the Apollo software system. In the event a system failure is detected by the Monitor, the messages sent by Control to CanBus are stopped. Guardian will send a message to CanBus forcing the car to stop.

5.3.7. Monitoring Systems

The Monitoring Modules gather information from all other modules to ensure that all systems are functioning properly. Two submodules ensure that this information reaches the user; the Monitor module and Dreamview module.

Monitor: The Monitor module is the supervisor (hence, “monitor”) of the other modules. The Monitor module receives data from different modules and forwards this information to the HMI (Human Machine Interface) for the driver to check and make sure they systems are working properly.

Dreamview: The Dreamview module is a web application which satisfies the need for an HMI (Human Machine Interface) in the Apollo architecture. The Dreamview module provides three primary functionalities. Firstly, Dreamview outputs relevant autonomous driving module information for the vehicle user, such as the planned trajectory of the vehicle. Secondly,

Dreamview provides an interface for users to view hardware status, turn off/on modules, and start the autonomous vehicle. Thirdly, Dreamview provides debugging tools. The Dreamview module has a dependency on Monitor to provide information about the system's status.

5.3.8. Cyber RT

This subsystem was added after the Apollo architecture was analyzed using Understand v5.1. The Cyber RT is an open source runtime framework which helps Apollo navigate complex driving scenarios. Apollo is optimized for high performance, low latency, and a high throughput. The demands of driving are variable and prone to change, thus development must be quick. This framework accelerates development while simplifying development and allowing a customizable driving experience. Cyber RT is the pride of Apollo as this feature provides the open source selling feature which Apollo is so reliant on. The multitude of development tools and modular approach of this framework grant public developers with tools to effectively utilize the open source nature of the Cyber RT framework.

6. Subsystem Architecture

6.1. Conceptual Architecture

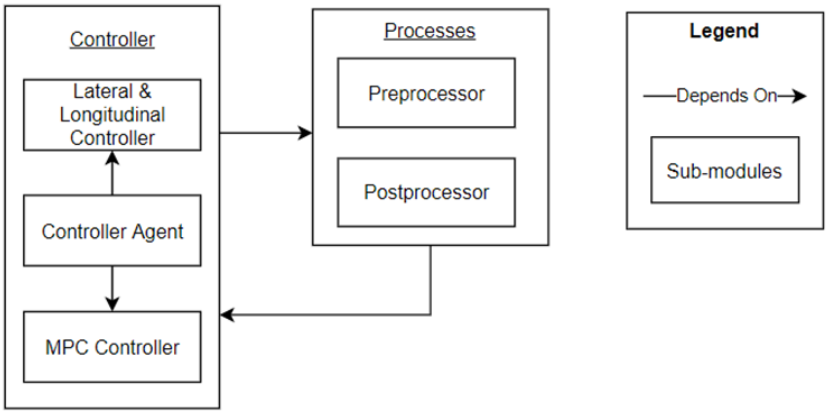

Figure 5: Conceptual Architecture of Control Module

The process-control architectural style can be applied to the control module (see figure #). In this style, there is a process component that manipulates variables in which a controller component determines how those manipulations occur.

In Apollos' case, the process component is divided into preprocessor and postprocessor modules. The preprocessor takes in the input data from other modules to process them before the controller needs to manipulate them. Then, the postprocessor takes in the data after being passed through the controller to ensure they are correct outputs that are used by other modules.

As for the controller component, it has a few subsections that help distinguish between the variables that are manipulated. First, the lateral and longitudinal controller manages values relating to vehicle commands of steering, and of brake and throttle. Second is the model predictive control (MPC) controller. It helps make predictions on which control commands are optimal to execute [1]. Last, the controller agent manages the other two controllers in this component.

6.2. Concrete Architecture

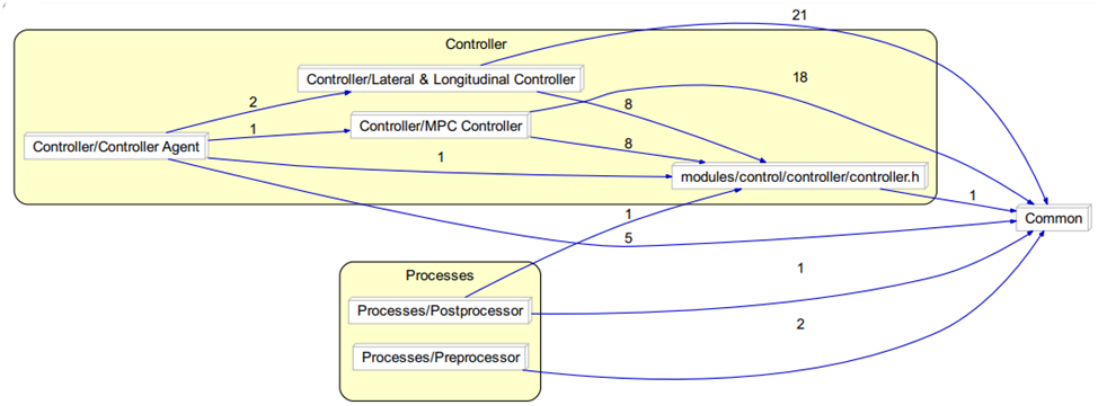

Figure 6: Concrete Architecture of Control Module

Using the Understand tool and the extracted dependency file from Apollo, the concrete architecture of the control module can be created (see *Figure 6*)The common module is a new addition to the architecture. From Apollos' overview on the common module, they use it for code that's not tied to a specific module [2]. In the architecture, it acts as a connector between the controller and processes as both components have dependencies to it.

6.3. Reflection Analysis

This section provides an analysis on the most notable divergences in the control module architecture. They come from the inclusion of the controller.h file and the common module.

controller.h

The modules that have dependencies with this file are the postprocessor, controller agent, MPC controller, and lateral and longitudinal controller. This file acts as base class for all controller modules to build upon. Additionally, the postprocessor uses this file to get data about variables that have been passed through the controller.

common/control_gflags.h

The modules that have dependencies with this file are the preprocessor, postprocessor, controller agent, MPC controller, and lateral and longitudinal controller. This file defines flags for submodules, files, and data found and used inside the control module.

common/dependency_injector.h

The modules that have dependencies with this file are the preprocessor, controller agent, MPC controller, and lateral and longitudinal controller. The controller.h file also has a dependency. This file directly allows the creation of dependencies between files.

common/mrac_controller.h

The module that has dependencies with this file is the lateral and longitudinal controller. This file is an algorithm to create more accurate and faster control commands of steering, brake and throttle that was introduced in version 5.5 of Apollo [3].

common/trajectory_analyzer.h

The modules that have dependencies with this file are the MPC controller, and lateral and longitudinal controller. The file obtains the current location point and manipulates it to help decide on a trajectory.

7. Limitations and Lessons Learned

During the writing of this report, our group encountered some limitations which became difficult challenges to overcome.

- When working on reflection analysis, the Understand tool was a bit unintuitive and can get quite slow when displaying a lot of information. When tracing the sub-components and files within the subsystems, one would find dozens of different boxes and lines going in different directions. When trying to figure out which file interacts with which subsystem, it can become quite difficult to trace it due to the convolution. Given that Apollo is a large system, it is expected that there will be many different interactions at different levels; however, the limitation of the Understand tool underpins the difficulty of being able to cohesively map out those interactions in a digestible fashion.
- There was limited documentation of certain modules, such as dreamview backend and how it operates, along with how to build it locally for a coding environment. This raised the difficulty on comprehension of the system and its interactivity. This also extends to the concern of maintaining standard coding practices for maintaining the submodules properly for future developers working on this project.
- The Understand tool's lack of ability to recover pub-sub message traffic made comparing conceptual architecture to concrete architecture more difficult. This was somewhat mitigated by the provided communication model document.

Despite using previous study groups as a reference to avoid difficulties they encountered, we still learned some important lessons;

- We learned that trying to create a conceptual architecture on a 2nd-level subsystem was more difficult than the top-level system as there was not as much documentation detailing the interactions within the subsystem. We overcame this by looking at the organization of the subsystem and applying background knowledge and outside sources to derive the architecture.
- We learned that generating sequence diagrams which align with the nuances of a concrete architecture can be deceivingly difficult, especially due to the constantly changing state of an autonomous vehicle. We ensured accuracy of our diagrams by referencing outside sources along with the derived concrete architecture.

- Our group learned the value of maintaining the quality of documentation and logs regarding a project. It was difficult to understand the project despite the hundreds of documentation files. The Understand tool aided us in small parts, but it in itself is relatively overwhelming. Our group increased our ability to use new tools and make connections to existing knowledge.

8. Conclusion

In conclusion, while the concrete architecture for Apollo was not too unexpected compared to our conceptual, there were several dependencies that we had not accounted for, as well as a couple of new modules. The most notable one was Cyber RT, which acts as a connector for all the modules. We also analyzed the subsystem architecture for the Control module and found some divergences due to the controller.h file and the common submodule. Overall, we saw the ways in which a system's concrete architecture is not a complete match to its conceptual architecture and learned how to analyze these differences.

9. Terminology

MPC – Model Predictive Control: model of a system to predict behaviour leading to optimizing control actions for correct output [1].

GFlags – Global Flags Editor: allows for debugging, diagnostic and troubleshooting features of a system [4]

Cyber RT - an open source runtime framework which helps Apollo navigate complex driving scenarios and acts as a connector for all the modules

10. References

[1] *Understanding model predictive control*. MATLAB & Simulink. (n.d.). Retrieved March 19, 2022, from https://www.mathworks.com/videos/series/understanding-model-predictive-control.html

[2] *Apollo/modules/common at master · Apolloauto/apollo*. GitHub. (n.d.). Retrieved March 19, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/common

[3] *Apollo/modules/control at master · Apolloauto/apollo*. GitHub. (n.d.). Retrieved March 19, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/control

[4] Marshall, D., & Graff, E. (2021, December 14). *GFlags - Windows Drivers*. Windows drivers | Microsoft Docs. Retrieved March 19, 2022, from https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags

[5] *Apollo/ at master · Apolloauto/apollo*. GitHub. (n.d.). Retrieved March 19, 2022, from https://github.com/ApolloAuto/apollo/tree/master