Apollo Concrete Architecture

Date: March 20, 2022

Authors: Apollogizers

Lavi Ionas: 17li3@queensu.ca
Xinyu Chen: 18xc26@queensu.ca
John Scott: 18jbs2@queensu.ca
Baorong Wei: 18bw16@queensu.ca
Zhihan Hu: 18zh22@queensu.ca
Anthony Galassi: anthony.galassi@queensu.ca

Abstract

Baidu's Apollo Auto is a mass-scale open software project that is tackling vehicle autonomy. Their efforts have drastically changed the current state of Apollo's software. In the previous A1 report, the conceptual aspect of the architecture was examined. In this report, we delve deep into the roots of the subsystems and cross analyze their conceptual architecture with their concrete architecture. Using Understand [Reference No.3], an explorative tool that allows us to dig through methods and function calls, our team has discovered new connections and interactions that were not prevalent in the analysis of the conceptual architecture.

In this report, we first analyze and detail the top-level subsystems that make up the concrete architecture. These include the various modules seen in the A1 report, as well as new modules like Dreamview, Storytelling, Tools, Common, Drivers, and Cyber. Our goal was to find where these modules fit in the conceptual architecture of the software. Then, we proceeded to gain a deeper understanding of the works of individual subsystems in order to lay out the patterns and decisions made by developers. Through the analysis of source code and the use of diagrams to layout the system, we achieved a profound understanding that would allow us to change the subsystem given that we were developers on this project ourselves. The ultimate goal was to integrate ourselves into the project under the assumption that we are the developers and understanding the systems would allow us to further their development. Finally, we took what we got from developing the concrete architecture and cross-analyzed it to our previous findings for the conceptual architecture. We came across several discrepancies and differences from the conceptual work and rationalized why they are different. The process of analyzing these critical stages aided us in becoming more aware of software systems and their concrete implementations.

Introduction and Overview

In this report, we analyze and discuss the concrete architecture of Baidu Apollo Auto. Using the software visualization and analysis tool Understand [Reference No.3], as well as the GitHub pages and code comments found in the current codebase of the project [Reference No.1], we accumulate an understanding of the real implementation of this project. When we found minor conflicts between these two resources, we chose to reference the source code since it is commented and documented. This concrete architecture report will be a component of our final report on Apollo Auto architecture.

In this report, we aim to cultivate a clear understanding of Apollo Auto's concrete architecture, as well as its relation to our previously informed analysis of the conceptual architecture. The structure of this report is therefore similar to our conceptual architecture report. First, we break the project down into higher-level modules in the concrete architecture. In doing this we carefully analyze the precise function that this module has, as well as the relationship it has with other modules, referencing both the Understand Software [Reference No.3], as well as the Github readmes and code comments [Reference No.1 & 2]. Next, we construct diagrams in order to represent what we have found, in order to both inform our digression of what is or is not an appropriate analogue to the conceptual architecture, and also to illustrate to the reader what the true structure of the project currently is. Finally, we use this information to then compare what we had found with the conceptual architecture that we outlined earlier, and comment on what dependencies we notice between the theoretical and the real implementation of this project.

Concrete Architecture

Top Level Sub-Modules

The concrete architecture of the Apollo project is significantly more complex than we had originally envisioned in the conceptual architecture. The two most top-level systems are Modules and Cyber.

Hardware Modules

Before the more perceptive and predictive modules can do their job, information about the car's surroundings must be taken by the sensors. Support for this functionality is largely found in the drivers module, which contains drivers for all of the hardware components in the system. Hardware components are also handled by the Canbus

module, which controls communication between computers and micro-controllers in the car and the computers within the car itself using the CAN bus protocol, taking in and executing control module commands, as well as outputting car chassis information. What also interacts with the car directly is the Control module, which takes commands from the system and translates them into smooth and actionable steering, as well as throttle and brake information. Finally, all this hardware is monitored by the Monitor module, which checks the status of the hardware and software, and the system health as a whole.

Perception Modules

With the hardware working correctly, their information must be made more useful for generating driving instructions. The most major module with this function is the Perception module, which takes data from the sensor modules, namely LiDAR RADAR and Image data, as well as YAML files containing information about the camera and RADAR calibration, and information about the car's position and velocity, producing 3D representations of the environment with the heading, velocity, classification information and traffic light detection and state. The perception of the environment is also aided by the Localization module. The localization module handles calculating the specific position of the car, by using one of two methods. RTK (Real Time Kinematic) Localization Takes in GPS and IMU (Inertial Measurement Unit) information, and Multi-sensor fusion localization takes in GPS, IMU, and LiDAR information. Whichever method is used, the module then returns protocol buffer format information about the car's current estimated position. Another perceptive module, the Map module, creates maps of the surrounding area in order to determine factors such as lane conditions etc, and can also determine if moving from one point to another is feasible.

Planning and Prediction Modules

After this informational data is generated, other modules will process this into plans and predictions for the car to use in order to drive. The planning module plans routes for the car to take given information from perceptive modules, specifically Localization, Perception, Prediction, HD Map, routing, and task_manager, and returns a calculated best route to take. A module that utilizes the generated maps, the Storytelling module, takes the map information as well as localization information to create "stories," which contain information that can be subscribed to by other modules in order to navigate more complex scenarios in a well coordinated manner. Another crucial module that uses generated map information is the Routing module, which processes map data and will generate high-level navigational information, given a request in the form of a starting point and ending point. Finally, the most substantial predictive module, aptly named the Prediction module, predicts how obstacles surrounding the car, monitored by

the Perception module, are likely to move. It receives positions, headings, velocities, and accelerations from the Perception module, and calculates predicted trajectories along with their respective probabilities. In the event of a dangerous situation, the Guardian module will bring the car to a stop, also with the help of the TaskManager module.

Peripheral/ Development Modules

Along with these modules, there are various modules that are used specifically for development or are otherwise peripheral to the creation and execution of driving plans. One such module, the DreamView module, provides developers with the ability to debug and monitor the Apollo platform, with an intuitive human-machine interface. Similarly, the Tools module provides the developer with various development tools written in python that are also useful for diagnostic purposes. Another module that does not directly fit into the structure of the program is the Common module, which is “code that is not specific to any module but is useful for the functioning of Apollo.” This includes code like Math, for useful math functions and Status, for reports of the success of different functions.

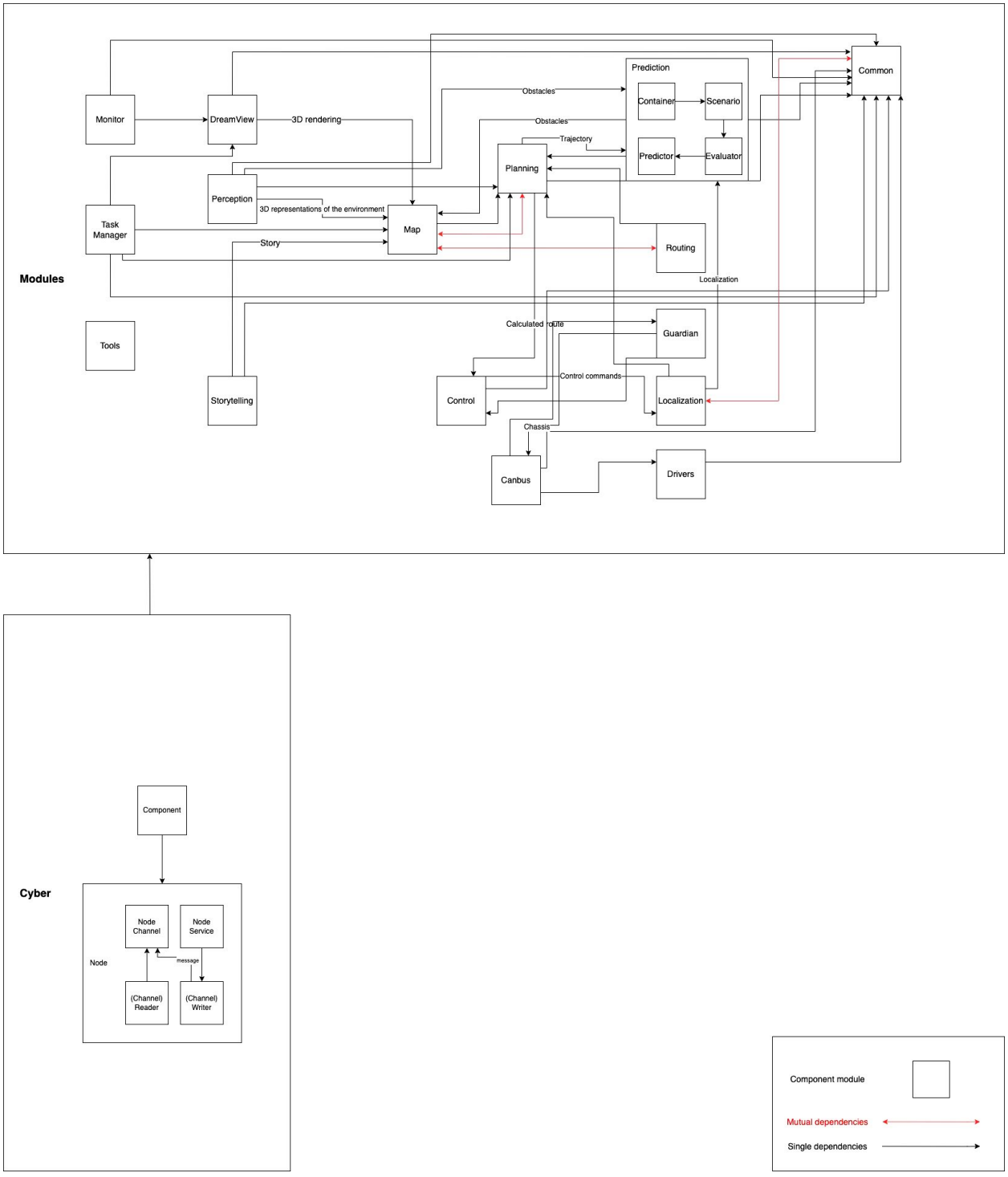
Cyber

Finally, aside from the Modules module, there is also the Cyber module. This is the Runtime Framework for Apollo, and where the Pub-Sub nature of the project is primarily handled, managing the interaction between all of the modules during runtime. It consists of two major modules, the Component module, and the Node module. In their description of Cyber RT the Apollo developers describe it as being “built upon the concept of components.” These components consist of running implementations of the modules that have already been mentioned, which are connected together with Nodes.

Box and Arrow Diagram

We referenced Understand and GitHub [Reference No.2]to create the diagram to visualize the interactions between each component. The black arrow represents single dependencies whereas the red arrow represents mutual dependencies.

* A better quality image of this diagram will be on our website.



Subsystem Analysis

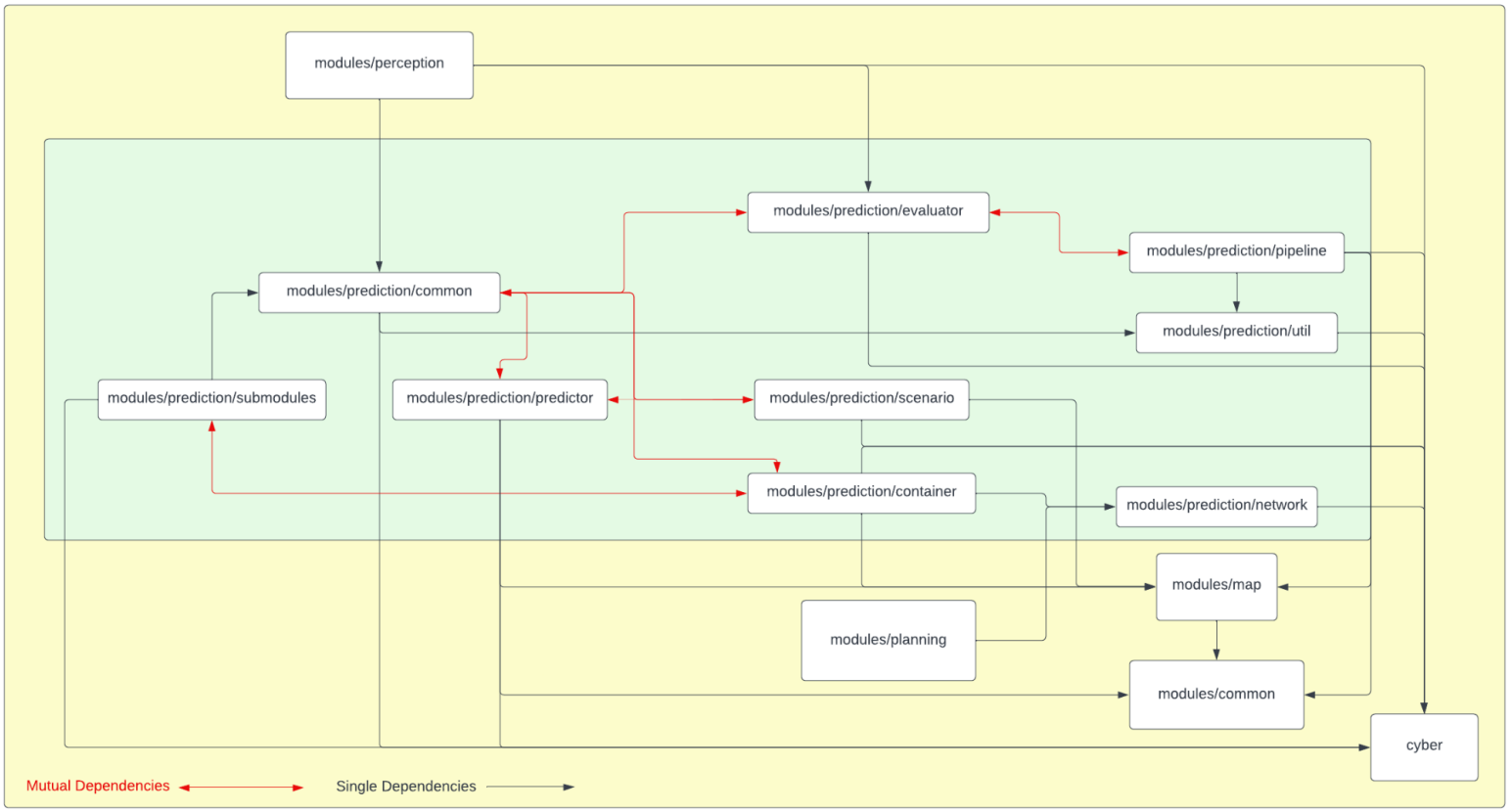
Upon analysis of the prediction module subsystem in the Understand program [Reference No.3], it has many dependencies. There were no mutual dependencies found between a submodule in the prediction module and a module outside of it. Every

module and submodule in the subsystem had a dependency on the cyber module; most submodules also had a dependency on the common module. The submodules common and evaluator in prediction have a dependency on the other top-level module perception.

Within the module prediction, it has many submodules each with a specific role inside the concrete architecture of Apollo. The container submodule stores input data from other modules for use in this overall module and others. The scenario submodule analyzes scenarios which affect the autonomous vehicle. This is broken into two sections, scenarios involving junctions (traffic lights and/or stop signs) and scenarios involving following the lane flow.

The evaluator submodule predicts the path and speed of every obstacle in view of the vehicle. This is then taken and outputs a probability for that specific outcome for the obstacle. This submodule helps to guide the vehicle to avoid obstacles it will collide into without intervention. The predictor submodule generates the predicted trajectories for obstacles. It's currently programmed to support a multitude of the various lane and movement conditions.

The common submodule has mutual dependencies with scenario, predictor, evaluator and container. The only other mutual dependencies in the prediction module are between the evaluator submodule and the pipeline submodule as well as the container and the submodules module with further submodules.



Reflection Analysis

A majority of submodules publish their output as a topic into the common module, and whenever a module depends on another, they create a reader to subscribe to the related topic. A common module is a module that is responsible for topics that are communicated through different modules. Some example analysis listed below:

- CANbus → Guardian: The component.cc file receives data about the command from the Guardian module. In the file, it creates a reader that can read shared data from Guardian. Guardian command is a topic in common. The same structure will be used when Canbus receives commands from the control module.
- Task_manager → Map: The Task_manager submodule depends on Map because it performs some routing patterns that require map information.
- Task_manager → DreamView: Task_manager depends on Dreamview, and it requires Dreamview's map service in cycle_routing_manager.h

Discrepancies with Conceptual Architecture

Within our group, we did not identify the chosen style in our A1, and yet the whole group thought it would be pipe-and-filter in the first place. We took it for granted that the process of information transfer took place in the Apollo as a pipeline sequence. However, when we drill ourselves into the concrete architecture and get some hints from the instructor as well as other groups, we realize that the whole system has proven to be more dependent on the connection established between publishers and subscribers. The Apollo system relies on the asynchronous cooperation of several different modules, each communicating with each other to ensure that the Apollo system is perfectly calibrated for each vehicle that is properly equipped for the system. The system's publish-subscribe architecture style also makes it easier for the system to evolve as new modules are implemented. Cyber RT plays an important role as the operating system in Apollo. Cyber RT creates "channels" for subsystem interaction, with "writers" acting as publishers to distribute information throughout the channel and "readers" as subscribers to receive dependent function calls. By using the advantages of the pub-sub style, the autopilot can remain secure because the broadcast messaging process allows modules to get the latest updates without having to wait for response times from other related modules.

We added a missing module: Storytelling, which is a new module that manages complex scenarios through predefined rules to trigger different actions. Each module has an option to follow this module's signals under specific conditions. It behaves like a

global manager with multiple planning scenarios, involving other modules to ensure safe driving. It can create "stories" from rules and isolate complex scenarios, packaging them into "stories" and publishing them through Cyber RT to its channel. It can communicate with other subscribed modules so that others can learn from the stories to fine-tune the driving experience.

In the previous report, we did not specify the conceptual derivation. As our group worked on the conceptual architecture, we first looked up the Apollo and its domain listed on GitHub [Reference No.1] and OnQ. Through reading these references, we formulated use cases and drew sequence diagrams. In our last report, we are supposed to combine the components used in the sequence diagram into a box-and-arrow diagram to show the static dependencies between components. However, we only referenced the one from Apollo's GitHub page [Reference No.1]. In this report, we have created a new version of the box and arrow diagram, as the previous section shows, to reflect the concrete architecture more clearly.

In the "lesson learned" section of our previous report, we summarized vaguely and inadequately. In this report, detailed lessons learned are provided, and they are adapted to Apollo's setting.

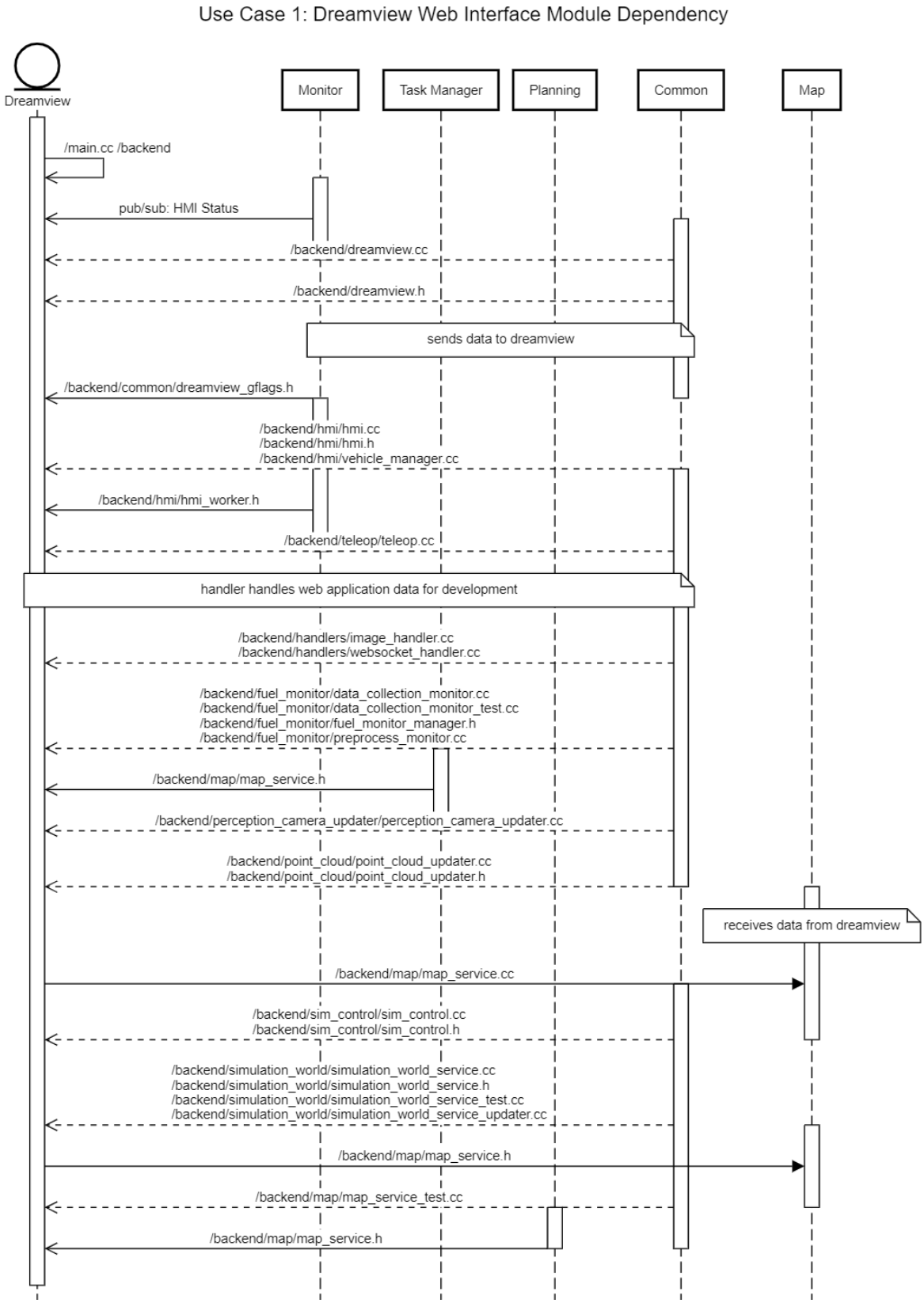
Use Cases

The following sequence diagrams lay out the dependencies and method calls that modules use in the concrete architecture. What follows are two notable use cases that aid in understanding the functional interactions between components. Unlike the analysis done in A1, this use case study takes a deeper dive into the inner workings of the source code. The first use case takes a look at the Dreamview module and how its dependencies affect the web application from where developers can visualize and debug data from all Apollo implementations. The second use case delves into the Control module and how its dependencies and method calls interact with the process of controlling a self-driving car.

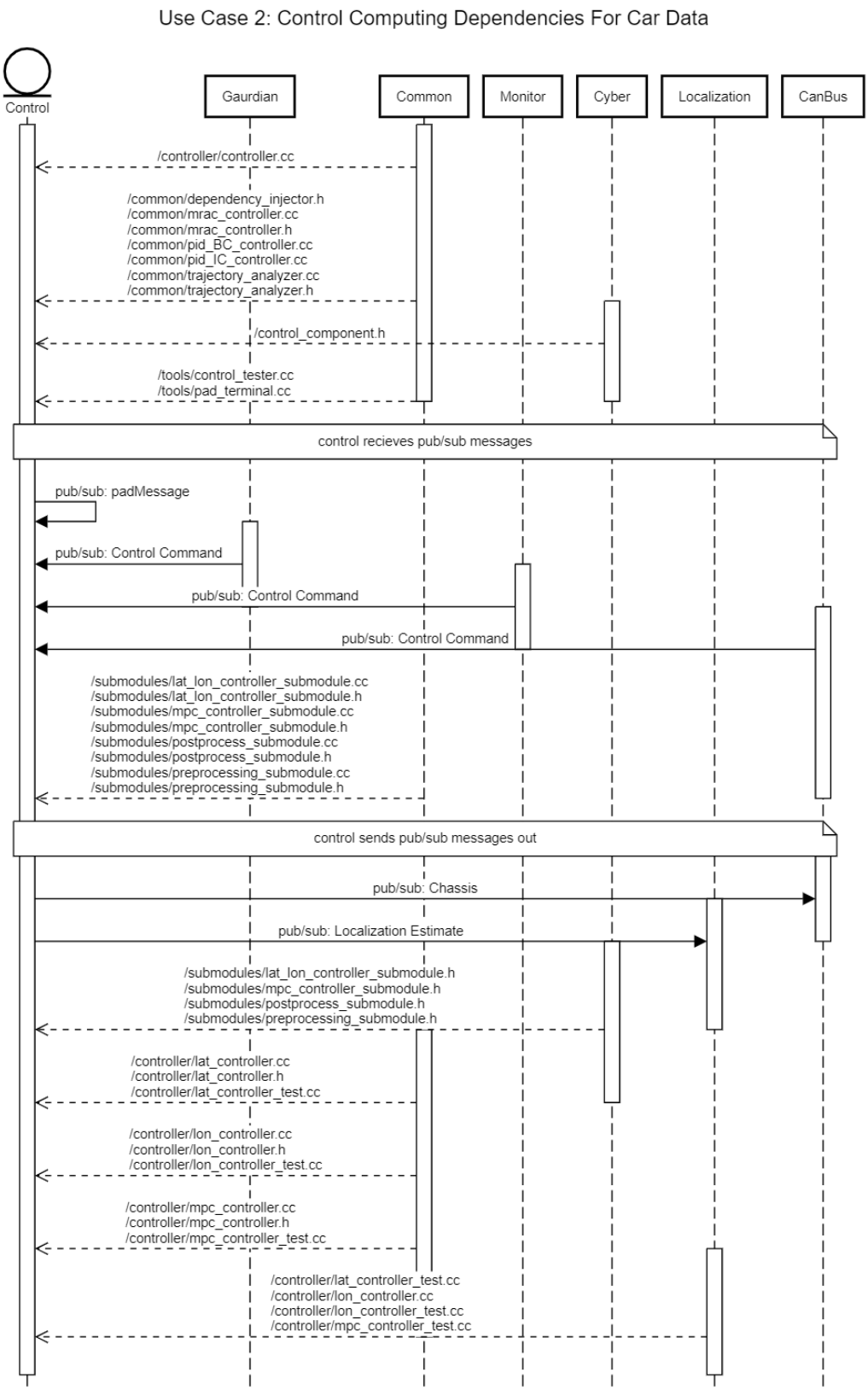
Please note that due to the complexity of said sequence diagrams, they tend to be very large. Hence, when they are inserted into a document they shrink and become practically unreadable. To remedy this inconvenience, high-resolution images will be posted on the group website for viewing / downloading purposes.

Sequence Diagram I: This sequence diagram depicts the interactions between modules when centralized on the Dreamview subsystem. It depicts both the methods called and the pub/sub messages relayed between other subsystems. An important aspect to note is the significance of relaying map_service.cc to the map module. This signifies that the three-dimensional visualization that dream view produces is significant to the software's ability to predict and plan the vehicle's controls.

Sequence Diagram II: This sequence diagram depicts the interactions between modules when centralized on the Control subsystem. The methods and pub/sub messages relayed between modules allow for a deeper understanding of the intricate workings of this software. Notice how there are 4 pub/sub messages going into Control that are defined as Control Command. This indicates that before authenticating an action on the physical car, the software Control module must be authorized by all 4 subsystems to ensure it is a safe and appropriate move.



Sequence Diagram I



Sequence Diagram II

Conclusions

In conclusion, Apollo's concrete architecture fits into the definition of pub/sub architectural style. As we dive deeper into the source code and the diagrams, we gained a more comprehensive understanding of the publisher and subscriber system used in Apollo. Detailed description of top-level subsystems for the entire Concrete architecture as well as their interactions were made. Later, We focused on subsystems and their interactions. We also update our conceptual architecture based on discrepancies between the conceptual and concrete architecture. Additionally, two sequence diagrams using concrete architecture were depicted, focusing on interactions within two subsystems.

Lessons Learned

This report was an eye-opening experience for the real-life applications of discovering and analyzing the methods and functions of a software as large as Apollo. Although it was daunting at first, we quickly got the hang of it and dug deep into the roots of the subsystems. Although in the end, we were successful, there were a few things we learned from some of the struggles we encountered in completing this concrete architecture report. We learned that breaking up the work and working alone in such an open-ended explorative task was beyond our abilities that overwhelmed us. Grasping that much information at once was daunting for the individual, but we came together as a group and shared each other's findings to connect the dots and come to the significant findings we laid out in the report. We learned that working on large software projects requires teams that work together and communicate effectively and thus we ourselves must also employ the same methodology to understand the subsystems.

References

1. ApolloAuto. (2021, December 28). *Apollo/README.md at master · Apolloauto/apollo*. GitHub. Retrieved March 18, 2022, from https://github.com/ApolloAuto/apollo/blob/master/README.md
2. Apollo. (2022, February 21). *apollo/cyber at master · ApolloAuto/apollo*. GitHub.Retrieved March 18, 2022, from https://github.com/ApolloAuto/apollo/tree/master/cyber

3. Adams, B. A. (2022, February 22). *Understand File for Apollo*. Bram Adams. Retrieved March 18, 2022, from https://onq.queensu.ca/d2l/le/content/642417/viewContent/3859027/View
4. Apollo. (n.d.). *Apollo*. David Silver. Retrieved March 18, 2022, from https://apollo.auto/devcenter/devcenter_cn.html
5. Sliver D. S. (2020, March 15). *Tutorial for Apollo Driverless*. David Silver. Retrieved March 18, 2022, from https://www.bilibili.com/video/BV1KE411V79h/?spm_id_from=333.788.recommend_more_video.-1