Concrete Software Architecture of Apollo

CISC 322/326 - Assignment 02
20/03/2022

Authors
Joshua Chai-Tang
Gabrielle Gravel
Matthew Krpac
Jonah Zimmer
Mack Peters

Web Version
https://chai-tang.github.io/CISC-322-Speedy-AI/CISC322-Group28-A2-Report.pdf

1. Abstract

This document takes an in-depth look into the pub-sub architecture used in the Apollo self driving car system. This includes a deeper look at the routing subsystem and visual depictions of the dependencies between the different modules. This document also aims to show the different data structures used within the software to better highlight the flow of data within the software architecture. Thus, this document serves as an analysis of the softwares inner workings and how data flows through each module by looking at the makeup of the code shown within the open source platform.

2. Introduction

2.1. Purpose

The purpose of this document is to bridge the gap between the conceptual architecture derived from a higher level analysis of the Apollo autonomous vehicle system and the more specific and less abstracted real life implementation of the systems architecture. This document will achieve this through the comprehensive analysis of the concrete software architecture present within Apollo. More specifically, an in-depth look at the routing subsystem applied in the Apollo will offer a relevant and effective way to convey necessary information to stakeholders that leans more into the technical details, functionality and implementation of significant components/connectors.

2.2. Organization

This document will be split into three distinct parts each focusing on assembling and delivering a holistic understanding of the concrete software architecture present within the Apollo autonomous vehicle system. This begins by displaying, and subsequently analyzing, the specifics of the routing subsystem found within Apollo; followed by applying this analysis and relating it to the relevant use cases encountered by the system. Finally we will draw conclusions and confer lessons learned to round out the complete study of this system's concrete architecture.

3. Concrete Architecture Style

The Apollo software architecture style closely resembles the publish and subscribe architecture style. The Apollo architecture currently contains 14 modules that communicate with each other through a series of publications and subscriptions, two more than were originally identified in our conceptual architecture analysis. This is largely due to the fact that our original conceptual architecture analysis was based on the apollo 5.5 documentation and the dependency graph provided for us was based on the most recent release. The addition and removal of certain modules speak to the scalability and malleability of the system, which can be adapted to provide better and more well-rounded service. The concrete architecture, as a result of following the pub-sub style, is not organized past these individual modules, meaning that each module is a unique component within the software system. Each module has an important role in ensuring the functionality of the system and is dependent on the other modules in the working system.

The 14 modules are as follows: Perception, Prediction, Planning, Routing, Control, Guardian, Storytelling, CanBus, Monitor, Localization, Task_manager, Map, Drivers, and Dreamview. The Planning module publishes messages to Control, Monitor, and Prediction, while waiting for messages (i.e subscribed to) CanBus, Routing, Localization, Map, Perception, Storytelling and Prediction. The planning module is responsible for calculating the vehicle's route using the data gathered from these modules, the resulting path then needs to be distributed among the modules listed above so that they each can perform their required task with the generated data. The Prediction module subscribes to messages from Storytelling, Planning, Perception, and localization while publishing messages to Planning and Monitor. Prediction is responsible for making calculations and estimates regarding obstacles and the environment surrounding the physical hardware system, the car, mainly to assist the Planning module in its efforts to choose the best course of action.

The remaining modules and their dependencies could be described similarly, but are rather redundant. Further analysis of the routing subsystem and its dependencies can be found in section four of this document. A new module was discovered to be an active participant in the Apollo framework as a result of completing the concrete architecture analysis, Task_manager. The Task_Manager module doesn't publish information to other modules, but does subscribe to information sent by Routing and Localization. Further analysis suggests that Task_Manager is responsible for delegating tasks, and has special routing plans for specific scenarios like parking and dead end management.

Subsequently, the concrete architecture style of the Apollo software system follows the publish/subscribe pattern.

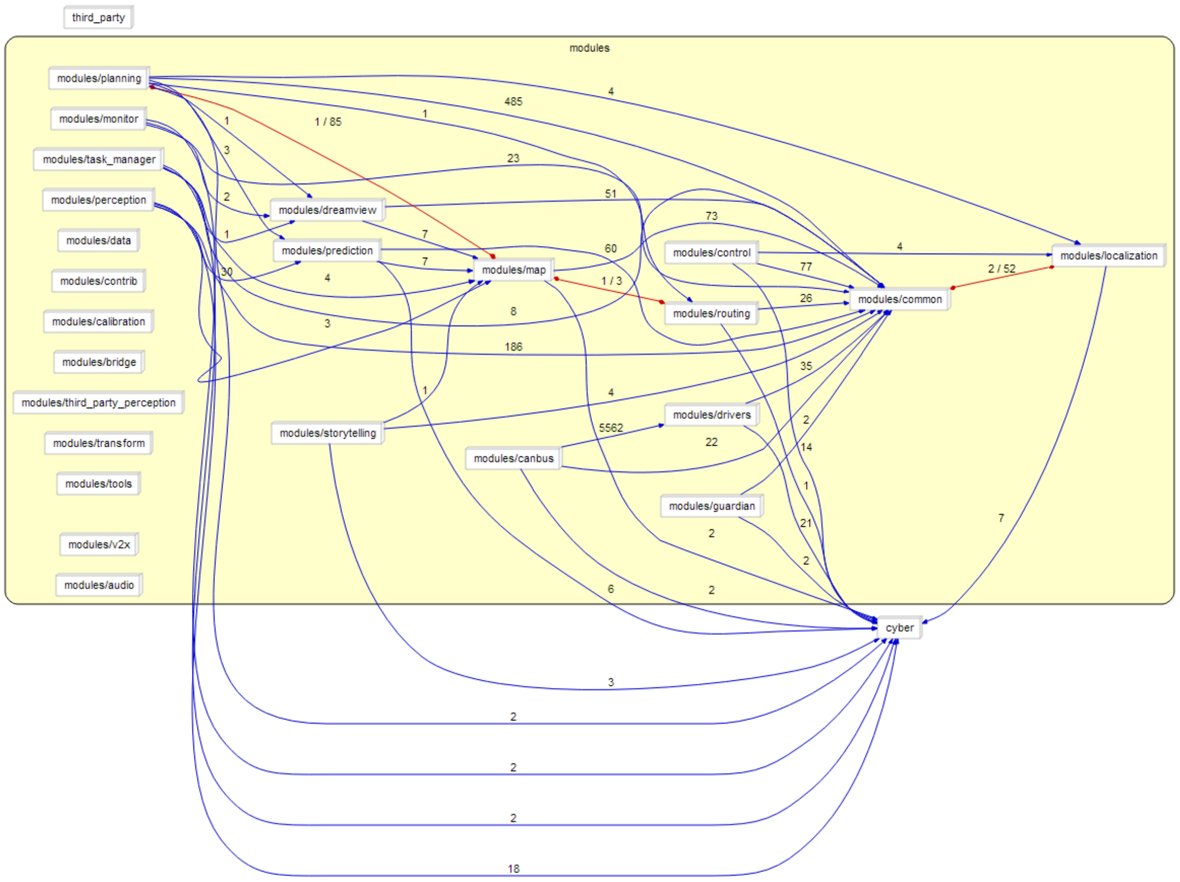


Figure 1 - Module dependency graph, derived using Understand.

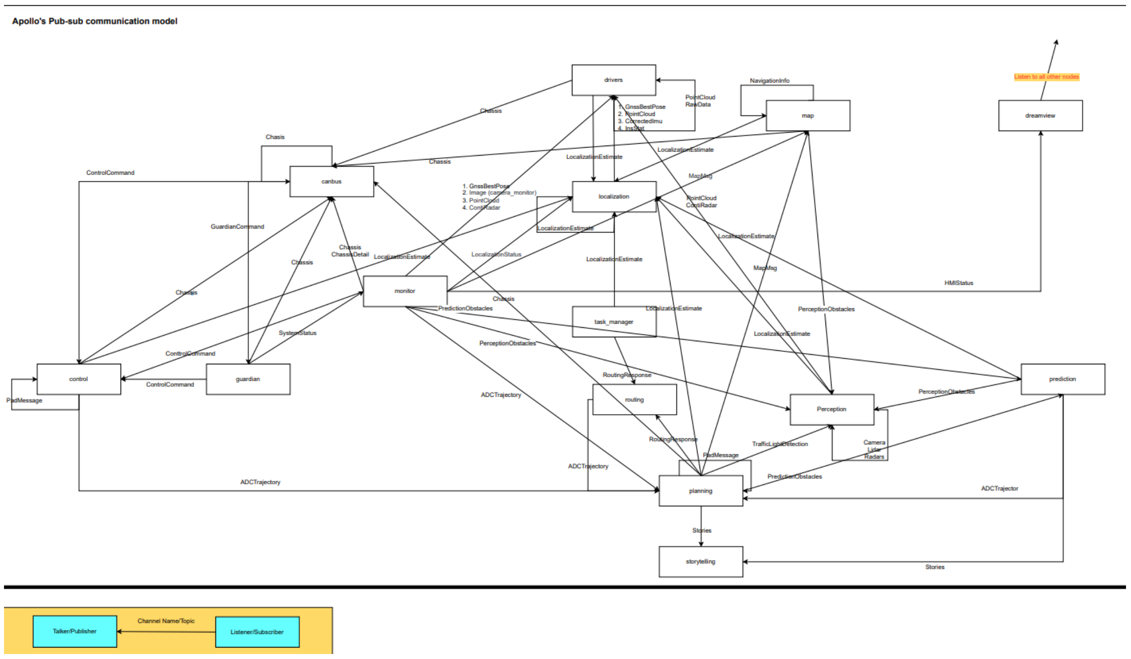


Figure 2 - Apollo's Pub-Sub communications graph. [2]

4. Analysis of the Routing Subsystem

4.1. Conceptual

The routing module is a high level navigation aspect of the Apollo system that uses local map data to determine the best possible route to travel from the vehicle's current location to the desired destination. Thanks to its similarities to third party navigation software, it is able to keep interdependencies to a minimum. This means that it can be scaled independently without the need to worry about subsequent issues arising from unintended knock-on effects.The only direct reliance within the system comes from the planning module which "subscribes" to the routing module's output.

Through our analysis of the relevant architectural data, we came to the conclusion that the Apollo's routing subsystem's method of communication and information dissemination adheres quite closely to the overarching framework laid out by the pub-sub architectural style. This means that the system applies an asynchronous service-to-service form of communication where publishers broadcast information with no knowledge of the recipients while subscribers within the system listen and "subscribe" to relevant and pertinent information to them. This means that the scalability of the system's modules is very effective as no system restructuring is required to implement more expansions.

4.2. Concrete

4.2.1. Description

The Routing subsystem is essentially Apollo's navigator. Its purpose is to generate paths through local roads and lanes for the vehicle to follow based on requests it receives. Requests generally consist of a pair of starting and ending locations. In order to generate efficient routes, this module considers a wide variety of factors such as distance, reachability, lane changes, and parking spots. Like most modules, Routing reports frequent status updates and error messages to the Monitor Log whenever applicable.

4.2.2. Data Structures

-TopoGraph-

Before it can begin calculating routes, this module must first generate a routing topology graph. This graph consists of a series of nodes and edges, representing intersections and roads respectively. Nodes and their edges also contain relevant information about their real locations such as lane width, driving direction, and allowed turns. Thus, this module generates routes by determining a sequence of nodes and edges that connect the desired target nodes as efficiently as possible.

-RoutingRequest-

When the system needs the Routing module to generate a path, it sends a RoutingRequest. A RoutingRequest is defined in the routing.proto file as a ‘Message’ data structure with numerous attributes. These attributes define relevant details about where the vehicle needs to go, and any further constraints there may be on the kind of route needed. Some key attributes are:

- waypoint : A set of at least 2 locations that the route must pass through. The first is the starting point, and the last is the destination.
- blacklisted_lane and blacklisted_road : Sets of street lanes and roads that the vehicle will not path through.

It further contains optional attributes for details about non-essential requirements like targeted parking spaces and dead ends.

-RoutingResponse-

Once a route has been calculated, this subsystem generates a RoutingResponse to publish to all subscribing modules. RoutingResponses are also Messages, like the RoutingRequests. Their only key attribute is RoadSegment: the ordered set of lane waypoints that were calculated as the optimal route. It also contains optional attributes for other details about the route. Some of these are the route length, the request that prompted this response, and the map version that was used.

Most of the attributes within RoutingRequest and RoutingResponse are also Messages, which themselves often contain other Messages and data structures. Resultantly it’s infeasible to give more detailed attribute descriptions within this report.

4.2.3. Components

There are two main components in the Routing module: Topo_Creator and Strategy.

Topo_Creator is responsible for generating the TopoGraph data structures this module needs. It does this by translating data from the Map module into a set of nodes and edges that correspond to the actual locations. Since real roads don’t change often, the routing topology graph doesn’t need frequent updates either. As a result, this subcomponent doesn’t run very often outside of system setup.

The Strategy component is responsible for calculating routes. Given the topology graph generated by the Topo_Creator component, and a set of waypoints from the RoutingRequest, the Strategy component determines the most efficient path to reach all required locations while minimizing distance traveled. It relies on the popular A* search heuristic algorithm to accomplish this. Details on that algorithm can be found in reference [3]. Apollo’s implementation doesn’t make any significant changes to the algorithm, although it does have numerous additional constraints added due to the requirement that the route generated must follow real road laws. This component is meant to be called frequently during regular use, especially since re-routing requests can happen often in the middle of a trip.

4.2.4. Dependencies

Analysis of the source code using Understand found that the Routing module has relatively few dependencies. Like the majority of Apollo's modules, it relies heavily on the common library and the Cyber RT framework for numerous common functions. In particular, the common library's monitor log is needed to publish regular status updates and error messages. It further depends on the Map module for the road data it needs to generate topology graphs. Specifically, the topo_creator.cc and graph_creator.cc subcomponents both import base map and routing map files from the hdmap utilities.

According to Figure 2 this module also listens for ADC Trajectory data published by the Planning subsystem. However there were no references to such a data structure in the Routing module's source code. It is possible that the Figure 2 graph is therefore out of date or incorrect in this regard.

Few subsystems are dependent on the Routing module. As per Figure 2, the Planning module listens for the RoutingResponses it publishes in order to guide its driving plan. The Task Manager also listens for these RoutingResponses in order to monitor its status and provide RoutingRequest prompts. Both the Planning and Map modules contain #include statements for the routing_gflags.h file for configuration and compatibility purposes.

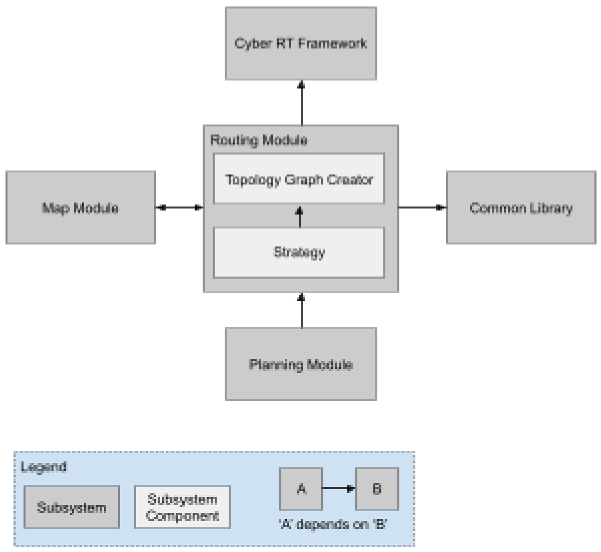


Figure 3 - Routing subsystem component & dependency graph

5. Reflexion Analysis

5.1. High Level

In the conceptual architecture analysis we were quite limited in terms of up to date resources documenting the current version of Apollo (7.0) and the documentation supporting the 5.5 release was used to supplement the core missing information. In the concrete analysis, however, a graph detailing the pub-sub dependencies in the Apollo framework was given which offered more accurate, and up-to-date information pertaining to the software system. This lead to the discovery of two additional modules used in the Apollo communication network that had not been previously identified. The first discrepancy noticed was the lack of HMI present as a module in the newer version, this was a result of the module being renamed to “Dreamview”, but ultimately accomplishes the same goal. Dreamviews goal remains to be a web-based dynamic 3D rendering of the monitored messages,acting as the main interaction point with the users.

The most notable difference found was the existence of another module, not at all mentioned in the conceptual architecture. As we were unable to find documentation supporting it during our initial analysis we can assume that it is either a new module added to support the demands of a growing system such as Apollo, or that it was improperly documented to begin with. This module being the Task_Manager. In the concrete architecture it is found to subscribe to routing and localization which adds a new dependency to each module. We found the addition of the Task_manager module to be justified, as it serves a useful purpose (a place where all tasks can be scheduled instead of spread out) as well as respects the guidelines surrounding a pub-sub architecture style due to increased ease in upscaling.

There was also an unexpected dependency between Planning and CanBus that was not anticipated during the original analysis. Given that the goal of the CanBus module is to accept and execute control module commands it is justified in sending information from the Planning module to the CanBus module since the controls would need to be based on the system’s generated plan.

Another major divergence in the concrete architecture of Apollo lies within the storytelling module. The initial conceptual architecture had anticipated it would receive information from the localization module and Map, however, it is actually subscribed to messages sent by the Planning and Prediction module. The current documentation for the Storytelling module insists that it receives input from the Map and localization module so this is likely an error that should be further investigated.

5.2. Routing Subsystem

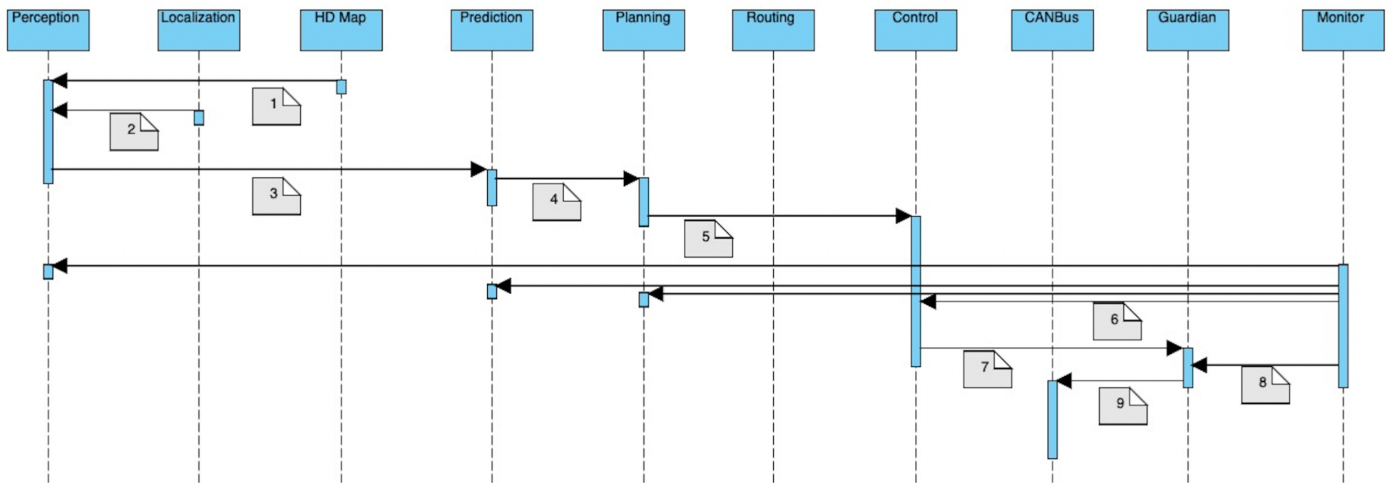
In our initial conceptual analysis, we determined that the Routing subsystem would not be dependent on any other modules. We also thought that the Planning module would be the only one which listened to Routing outputs.

The concrete architecture proved significantly different to those conclusions. Routing is in fact dependent on numerous other modules, including Map, Cyber RT and the common library. Our initial analysis entirely failed to account for the Cyber RT framework and common library components. These are crucial due to the many useful functions and features they contain, which are needed by most modules in the Apollo architecture. The Map module dependency is obvious in retrospect, as it would be redundant to have the Routing module pull map data from an external source when the Map module already accomplishes this.

More surprising are the number of modules which depend on Routing. The Planning module was correctly accounted for in the conceptual analysis, but the Task Manager was not. This is because the original report did not include any information on the Task Manager due to its lack of documentation in the Apollo github. The conceptual analysis also failed to consider that both Map and Planning, as the two modules which interact with Routing the most, would need to include header files from Routing for compatibility.

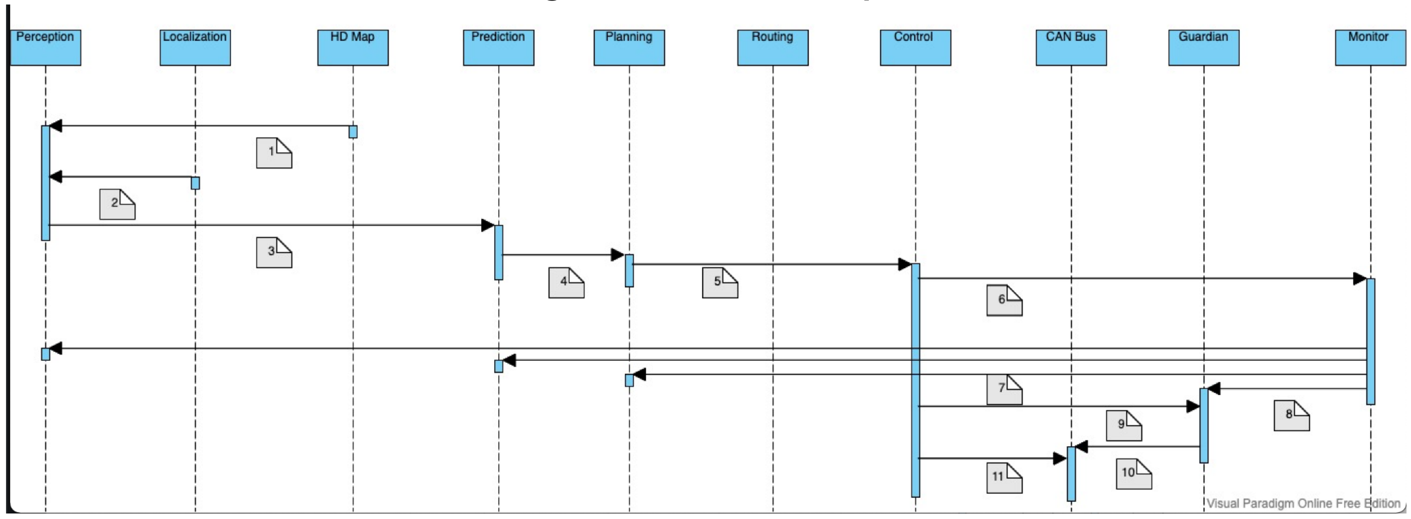
6. Use Cases

6.1. Use Case 1 - Monitor detects critical problem within a module



1.Send detailed information on surrounding area
2. Send current location
3..Send information of surrounding objects in relation to the vehicle in surrounding area
4.Send predicted trajectories of the objects surrounding the vehicle
5.Sends planned route through the nearby area
6.Check functionality of modules to determine if it is safe
7.Send check for alert
8.Sends Guardian Alert of system malfunction
9.Tell car to stop abruptly

6.2. Use Case 2 - Car driving on a street with no problems detected



1.Send detailed information on surrounding area
2. Send current location
3..Send information of surrounding objects in relation to the vehicle in surrounding area
4.Send predicted trajectories of the objects surrounding the vehicle
5.Sends planned route through the nearby area
6.Send functionality request
7.check modules to see if they are running properly
8.Sends Guardian Alert saying all systems working
9.Send check for detected malfunction
10. Send CAN Bus the confirmation the system is working
11. Send the planned route for the car to follow.

7. Derivation Process

The derivation process for determining the concrete architecture of Apollo and its routing sub system primarily relied on using Understand to determine its dependencies and connections with other subsystems. By using Understand we were able to visualize the routing sub-systems dependencies to other sub-systems such as the common library and the Cyber RT Framework. It was only through seeing how the various functions of the Apollo system worked for us to be able to see these dependencies. Furthermore, we could also derive from Understanding what the internal entities are such as the Topo Graph Calculator and how the Strategy module relies on that. While this was all accessible through Understand, Understand poor documentation and lack of other resources lead us to having to look at the source code to the Apollo modules to truly see why all these systems were connected.

8. Lessons Learned

There were a few learning curves we faced when working on the concrete architecture. The first obstacle we faced was the lack of documentation for each of the modules and their functionalities. This led to us having to dive into the source code to get a better understanding of the modules. Unfortunately the functions in the source code also had very little commenting and documentation. We ended up having to read through the source code and track how it functions to understand how each of the modules interact. This would have been less difficult if the source code had better documentation which is something we wish we knew in advance. Another obstacle we faced was using understanding on a different operating system.

One of our members has a linux machine which required a completely different setup method to get the UDB file loaded as well as the code from the repository. While this was solved it would have been good to know that the setup process was much different from that of the tutorial.

9. Conclusions

Using Understand we were able to determine the concrete architecture of the Apollo Model. More specifically we dug into the functionality of the routing subsystem. We learned how the internal dependencies such as the Topo_Creator which is responsible for generating the TopoGraph data structures and the Strategy component are responsible for calculating routes. We also learned more accurately the routing modules dependencies with other systems such as how it relies heavily on the common library and the Cyber RT framework for numerous common functions. Unlike the conceptual analysis where we were referencing older versions of Apollo which lead us to misinterpretations of the systems functionality. Through Understand we were able to get a much more accurate picture of how the system and its subsystems function. For example we found out there was another module which was the task manager or how there were unexpected dependencies between the Planning and CanBus modules. Finally, with this extra information we developed some use cases such as “driving on the street with no problems detected”, which we were able to visualize and explain in greater detail and with more accuracy in regards to how Apollo would actually function.

10. References

[1] Apollo’s Github documentation, written by the Apollo team.
https://github.com/ApolloAuto/apollo

[2] Apollo pub-sub communication model graph, provided by Dr. Bram Adams.
https://onq.queensu.ca/d2l/le/content/642417/viewContent/3865686/View

[3] A* search algorithm, lesson created by Brilliant
https://brilliant.org/wiki/a-star-search/