CISC 322

Assignment 2
Concrete Architecture of Apollo
March 21, 2022

Group 37

Karl Dorogy	18kwd1@queensu.ca
Ryan Saweczko	18rjs5@queensu.ca
Tom Lin	18phl@queensu.ca
Briggs Fisher III	18wbfi@queensu.ca
Douglas Chafee	18dsc1@queensu.ca
Itay Ben Shabat	18ijbs@queensu.ca

Abstract

Using the architectural style of implicit invocation (aka Pub-Sub) and the conceptual architecture previously found for the top-level system of the Apollo autonomous driving project, we now work to build a concrete architecture of the system. We used a graph of all pub-sub communication between each module (provided by the professor), as well as a map of all function calls between modules (created with Understand) to create the concrete architecture. With the concrete architecture, we found that there were several divergences and a couple absences from the conceptual architecture.

All of the absences between the conceptual and concrete architecture can be explained by poor wording within the documentation making us believe there were many dependencies, while the concrete architecture was a subset of the expected dependencies, such as with the monitor module. Major divergences between the concrete and conceptual architectures were the introduction of other new systems we didn't see in the conceptual subsystem, such as a common/utilities system. These were found to be distinct subsystems when extracting source dependencies despite not being well documented. There were also several new dependencies found, such as the modules that depend on drivers.

We also investigated the conceptual and concrete architectures of the perception module. We found the systems within this module, and created a conceptual architecture based on the information we could find in documentation about the module. Then, by investigating in the code using Understand to find dependencies between the modules, we created a concrete architecture. The concrete architecture we found had several divergences from the conceptual architecture, so we then did a reflexion analysis on these differences.

Introduction & Overview

This report focuses on the recovery of Apollo's concrete architecture and a documentation of the divergences and absences found from our initial conceptual architecture. To create a proper concrete architecture, we started with Apollo's structure that we found through our conceptual architecture. We used these subsystems and dependencies as a base for a new architecture that we constructed in Understand, using the provided understand file. We procedurally mapped out a concrete architecture by analyzing each of Apollo's directories, folders, and files to sort them into each of the different subsystems. We found one area, common tools, that didn't fit into any of the original subsystems found, so created a new one for it. When we finished finding the concrete architecture, we moved on to find all the unexpected divergences and absences that were missing from our conceptual architecture. The final concrete architecture is visually presented in figure 4 and its discrepancies from the conceptual architecture are discussed in the reflexion analysis in section 2.

This report consists of six sections. The first section is the derivation process which details the meeting and process our group went through to construct our concrete architecture with the use of Understand and the pub-sub graph provided. The second section describes the

concrete architecture at the top-level with a reflexion analysis that describes the differences found, either absences or divergences, between the two architectures. Some of the dependencies for the monitor were discovered to be the only absences found in the concrete architecture. Common/Tools was the one independent subsystem found for the concrete that was missed in the conceptual. Twelve divergences were found in the dependencies between the concrete architecture's submodules and described in the reflexion. The third section explains the inner architecture of the perception module and its dependencies in both concrete and conceptual views with its own reflexion analysis, explaining divergences within the Lidar, Camera, Onboard, and Radar submodules. The fourth section includes the sequence diagrams visually presenting two different use cases from our concrete architecture. The fifth section includes the data dictionary with a glossary of key terms used in the report, additionally naming conventions are provided with explanations for any abbreviations used. The sixth final section recalls what creating this concrete architecture has helped us learn since the completion of our conceptual architecture, as well as what we have learned as a whole.

After reviewing the directories, folders, and files of Apollo's source code we were able to create the concrete architecture which we compared with our original conceptual views to see how implementation resulted differently than we had anticipated. We discovered that the differences were a result from the introduction of the Common/Tools module that we did not discern in the conceptual architecture, the removal of the Monitor module we presumed was required, as well as many overlooked dependencies. We learned how to find and read git logs to discern the discrepancies by analyzing the code, the commit message, as well as the time committed. We also learned how to use Understand to map these discovered discrepancies onto a concrete architecture that we finalized in figure 4. We used this knowledge not only to conclude our concrete architecture but also to investigate the inner perception module and its submodule's dependencies. We now have a clear understanding of how a large software system's architecture can be drawn directly from the knowledge obtained within the system's source code.

1. Derivation Process

In deriving Apollo's concrete architecture, the derivation process first began with the use of Understand, a software/code analysis tool that allows users to visualize, analyze, and understand the dependencies within a system's source code. An initial group meeting was held, where we imported the provided apollo.udp file into Understand. We then started to create a new architecture together. We first created a corresponding component for each subsystem that was derived from our original conceptual architecture in Assignment 1. Once this was established, as a group we then proceeded to analyze the Apollo system's source code (directories, folders, files), mapping each folder or file of source code to its corresponding subsystem that it belonged to within the new architecture. For some of the subsystems, this mapping was done simply by looking at the source directories naming scheme. For example the directory/folder labeled "planning" was mapped to the Planning subsystem and so on. However, for the specific cases where source code wasn't easily mappable to a corresponding subsystem, we would then together try to understand its functionality within the system by further looking at documentation, opening individual classes, reading developer comments, and inspecting the

names/invocation of methods. This analysis then gave our group an overall better understanding of the specific source code's functionality and thus after a group wide discussion and agreement allowed us to map the source code to its correct subsystem within the new architecture.

This process was then repeated a couple more iterations within a couple more group meetings until finally all the source code was mapped to a desired subsystem within the new architecture. Then using Understand and its ability to analyze the source code in regards to our newly created architecture subsystems, we then created a static dependency graph shown in Figure 1 where the dependencies between subsystems are shown visually.

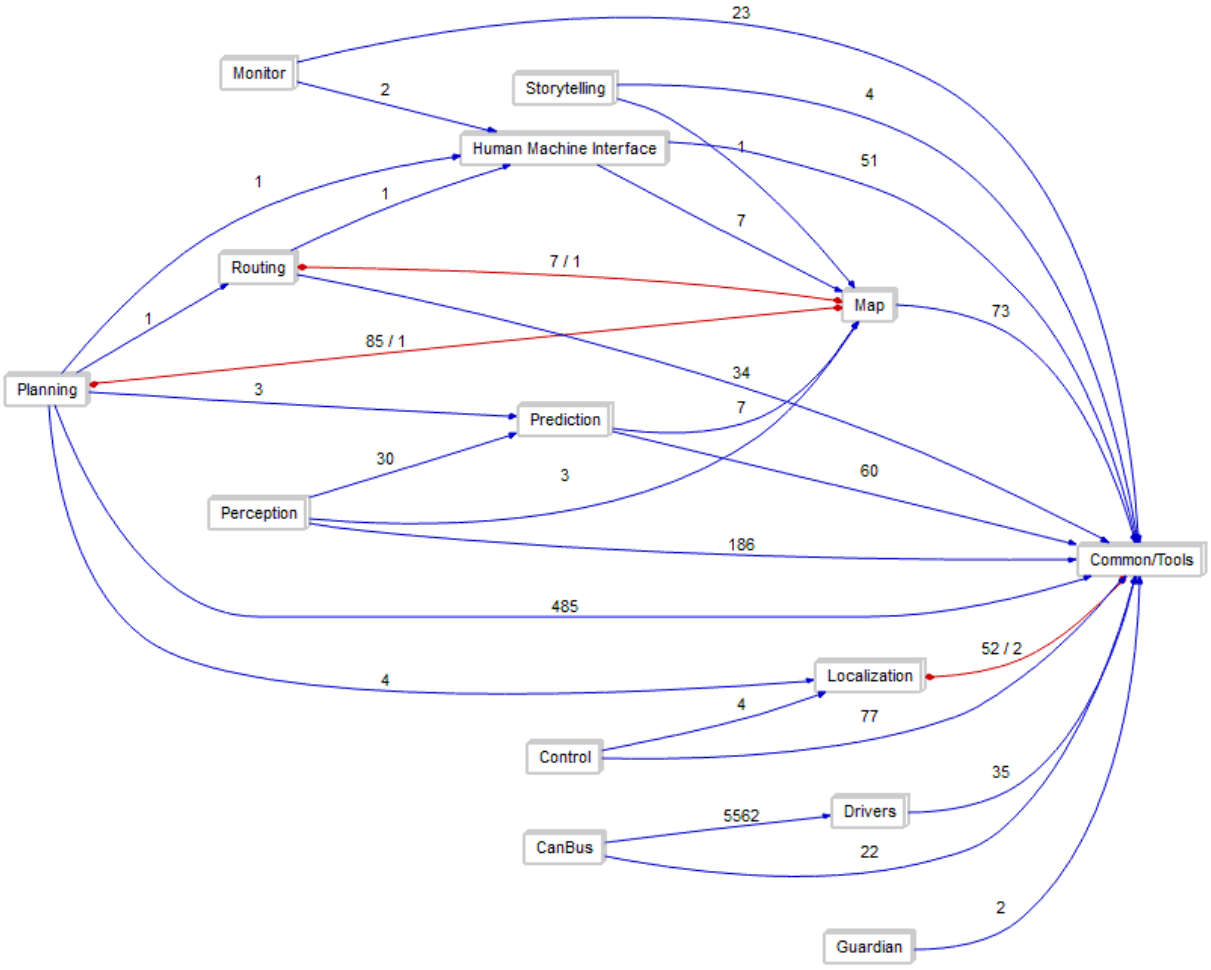

Figure 1: Graph visualization of Apollo source code dependencies.

Then combining both the graph visualization of Apollo's source dependencies shown in Figure 1 and the provided publish subscribe (pub-sub) communication graph[1] shown in Figure 2, we then ultimately concluded Figure 4 as the best representation of Apollo's concrete architecture.

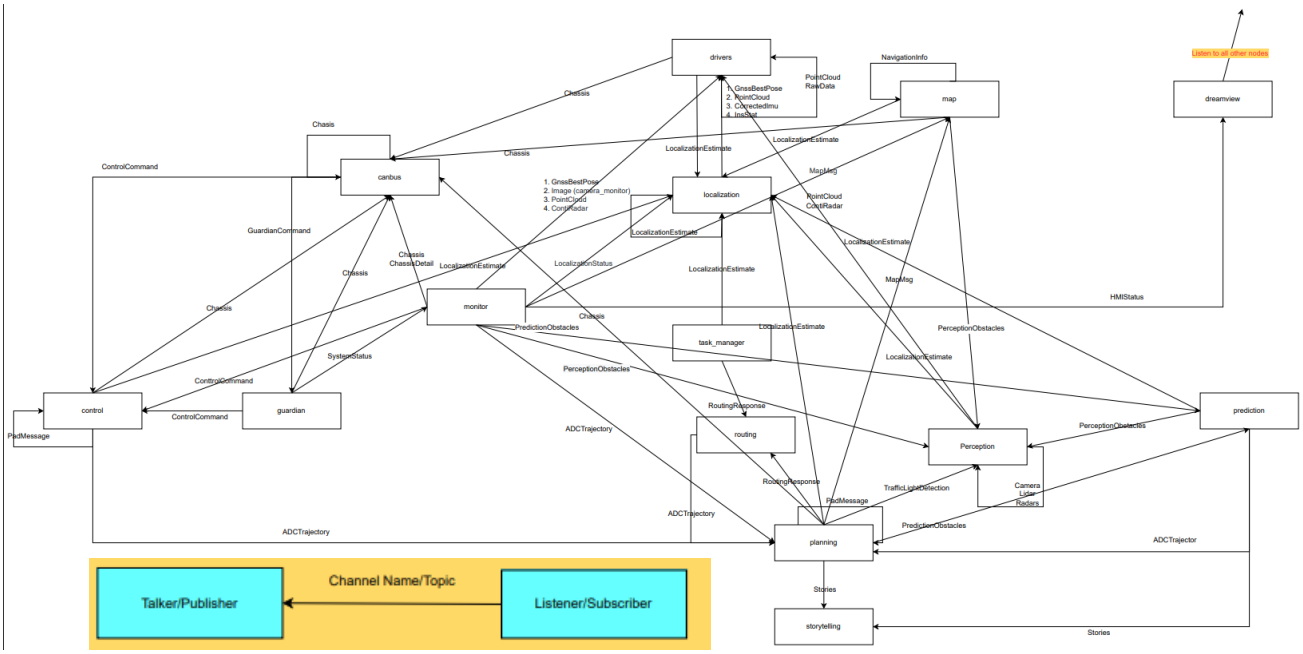

Figure 2: Graph visualization of Apollo system pub-sub message traffic.[1]

2. Conceptual & Concrete Architecture Of Top Level

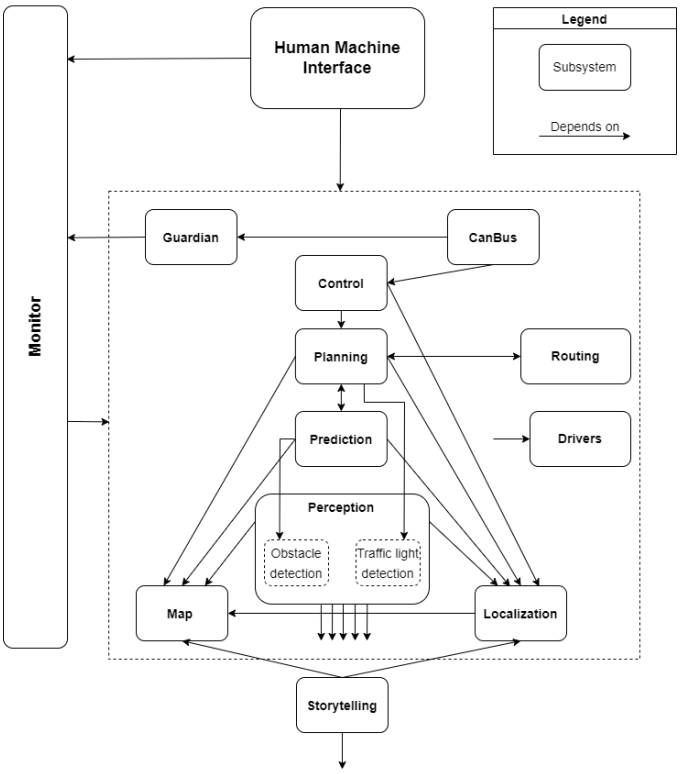

Figure 3: Conceptual Architecture of Apollo

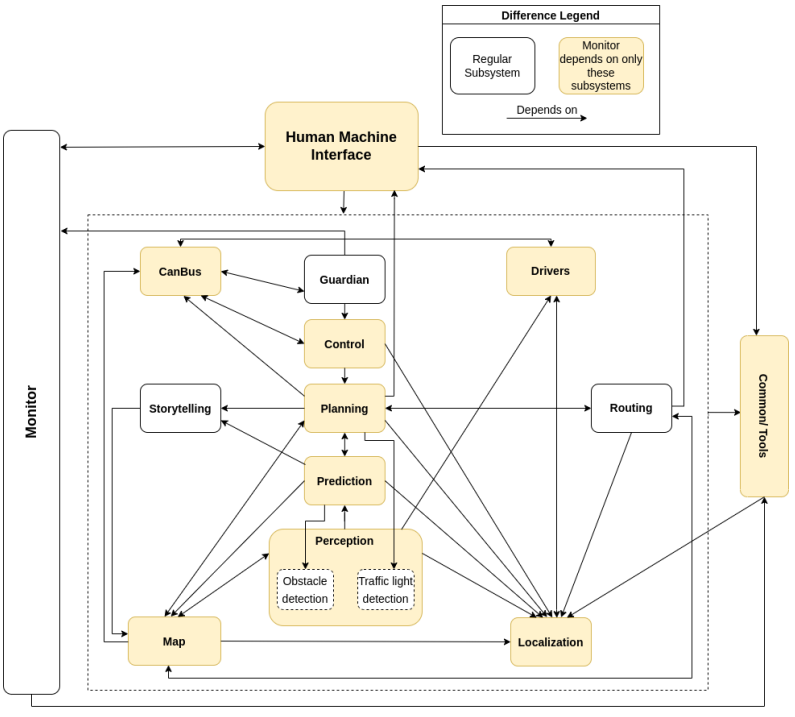

Figure 4: Concrete Architecture of Apollo

The top-level concrete architecture of Apollo has several key changes and interactions when compared to the conceptual architecture. Most of the individual components were found in the conceptual architecture, and described there. The one new module to the concrete architecture is the common/tools module. The purpose of this module is to be a utilities module, where commonly used functions could reside to be used by other modules.

The concrete architecture has a few modules that are depended on by nearly every other module, namely localization, common/tools, and the monitor. The monitor depends on many other modules in order to fulfil its function of everything working properly. The common/tools module is depended on by nearly everything because the purpose of this module is to provide general functions that can be used across modules. Finally, localization is used by nearly all other modules related to driving the car where it needs to be because most of these modules can't plan what to do without knowing where the car is, which the localization module can provide.

Another notable interaction in the concrete architecture is the dependency between the driver module and the perception, localization, and canbus modules. These dependencies occur largely through the use of the implicit invocation architecture. For example, the canbus directly (by calling a function) tells the driver module to create a driver for the hardware canbus, then the driver module can send messages through the data channels back to the canbus module with what happens. Similarly, the perception module subscribes to the events from the lidar, radar, cameras, etc to receive data from the drivers module.

The dependency chain through the central area of the architecture (control depending on planning, which depends on prediction, which depends on perception) is all mainly done through data channels with implicit invocation. The prediction module sends object-detection through the "/apollo/perception/obstacles" to the perception module, which in turn sends PredictionObstacles to the planning module via "/apollo/prediction". Traffic light detection from the perception module is sent directly to the planning module also through a data channel, "/apollo/perception/traffic_light".

2.1. Reflexion Analysis

Our concrete architecture has several differences, mostly divergences, from the conceptual architecture of Apollo. We found one absence, with the monitor depending on different modules that were initially found in the conceptual architecture. We found most of the subsystems had at least one divergence, and discovered a few new subsystems that were not considered in the conceptual architecture yet were their own subsystems in Apollo, such as a common/tools system.

2.2. Absences

Absences refer to dependencies found in the conceptual architecture, but were not found in the concrete architecture. In the Apollo system we found one absence: the monitor subsystem's dependency being reduced by a couple modules.

2.2.1 Monitor

From the documentation investigated for the monitor, we found evidence that the monitor depended on many different systems than what the conceptual architecture said. Based on its

functionality we had concluded all systems related to the autonomous car were monitored by the monitor in the conceptual architecture. Through inspecting dependencies within the code, we found only specific modules needed it. The reasoning we found for this is that the monitor is mainly to ensure the system is healthy and data is moving properly. Due to this, modules like drivers definitely need the monitor, but storytelling does not, since storyteller doesn't deal with the hardware or data as much. Hence, this absence should be resolved by changing the conceptual architecture to reflect this understanding from the code.

2.3. Divergences

Divergences refer to dependencies found within the concrete architecture, but not in the conceptual architecture. They may represent problems within the code base where there are poor separation of concerns, or could be necessary and the dependency should be added to the concrete architecture after the divergence is investigated. We have found several divergences within the Apollo System.

2.3.1. Common / Tools module

New module: common/tools module is an important general utility module used by other modules to obtain system information about the vehicle, and publish buffered information to monitor.

Investigation of Common /Tools lead to finding dependency on one module, Localization. This dependency was added in November of 2017; Common's vehicle_state_provider and vehicle_state_provider_test use localization to find the vehicle's current position. This is a required dependency as vehicle_state_provider is used by other modules to access general information about the vehicle such as dimensions, center of mass, and position.

Every module depends on Common / Tools, there are five frequently used files; monitor_log_buffer.h, map_util.h, status.h, vehicle_config_helper.h, adapter_gflags.h. The importance of monitor_log_buffer is it allows MonitorBuffer objects to publish their logs to monitor topic. Map.util is frequently used as it helps generate maps. Status.h is used for evaluations/comparisons and generating readable error messages for the user. Vehicle_config_helper is used to access vehicle information such as length, width, height, center mass, and position on map / route. The last frequently used file, adapter_gflags is used to monitor sensor calibration. All five files above provide crucial information for the system to run, so it makes sense they are grouped into one utility module.

2.3.2. Routing depends on HMI

After investigation it was found Task Manager depends on Human Machine Interface (HMI) for its MapService class, responsible for generating a map for the current route. This was added December 2020 and is required anytime a route begins, ends, or reaches a deadend.

2.3.3. Routing depends on Localization

Routing's dependency on Localization is in task_manager_component.cc and added in November of 2020. The dependency is there because everytime a new route and map is calculated, the car's current location is required, so routing needs to retrieve this information from the localization module.

2.3.4. Routing depends on Common / Tools

Routing's dependence on Common / Tools makes sense as there are scenarios such as parking_routing_manager where routing needs the vehicle dimensions provided by vehicle_config_helper found in Common to verify parking locations large enough for the vehicle. This dependence was added during July of 2021.

2.3.5. Perception to Drivers

Perception is subscribed to Driver's GnssBestPose, PointCloud, CorrectedImu, and InsStat as seen on Figure 2. Perception reacts to the information published by Drivers.

2.3.6. Perception depends on Prediction

April 2020 a "#include" was used to add a dependency on prediction obstacles_container.h from perception mlf_engine.h. At this moment, the "#include" is irrelevant as all code making use of this include is and has been commented out for over a year.

Another dependency was added in April 2020. Perception requires Prediction's semantic map, one instance is Perception's evaluator manager requiring Prediction's semantic_map.h so it can more accurately track the vehicle on route. This dependency is necessary as accurate spatial awareness is a must for autonomous cars.

2.3.7. Planning depends on CanBus

Planning is subscribed to CanBus chassis publications as chassis publishes chassis specific errors such as brake failure while driving. This dependency is needed so the planned route, vehicle input, and/or human passenger can react safely.

2.3.8. Planning depends on Storytelling

Planning is subscribed to Storytelling's library of simulated scenarios. Planning uses these stories to help choose the route pathing and react to real-time environment changes.

2.3.9. Planning depends on HMI

During July 2021 a dependency from Planning on HMI was made. Planning's open_space_roi_decider file has a "#include" for HMI's map_service.h, however, this dependency should not be there as Planning never makes use of the included file.

2.3.10. Prediction depends on storytelling

Prediction uses Storytelling's library of stories to help evaluate potential obstacles like cyclists and their projected path, which is used in conjunction with planning to avoid collisions. This dependency is necessary for collision-free driving.

2.3.11. Drivers and Localization depend on each other

Localization depends on the drivers for data the driver module retrieves from the various hardware components, such as lidar and radar systems. The drivers module ensures compatibility with these hardware components, then outputs it to channels that the localization module is subscribed to. This is one of the areas we knew a module would depend on drivers within the conceptual architecture, but unsure which. Localization depending on drivers was shown in the graph provided by the professor as a pub-sub based dependency, however we cannot find this channel in the code. This may be a false-positive and should not be in the architecture.

2.3.12. Drivers and CanBus depend on each other

CanBus depends on drivers because the canbus is one of the hardware components, so a driver is required for that piece of hardware, requiring this dependency. The canbus module then sends messages to the driver, specifically the canbus instance, to direct the driver how to handle the canbus' functionality, so the driver depends on canbus module. Both of these dependencies are required, since the driver subsystem is distinct from the canbus subsystem, yet they both need each other for the canbus driver to function.

2.3.13. Map and Routing depend on each other

The routing subsystem depends on the map in a few areas, one main one being the graph_creator entity. It was introduced in 2017 as a way for the routing module to use functions from the map module to properly load xml maps into routing. The inverse dependency, with the map depending on the routing was added in 2017, however the need for this functionality has been removed leaving only the "#include" statement. The map does not really depend on routing, and this include statement should be removed from the code base and the concrete architecture.

3. The Perception Module

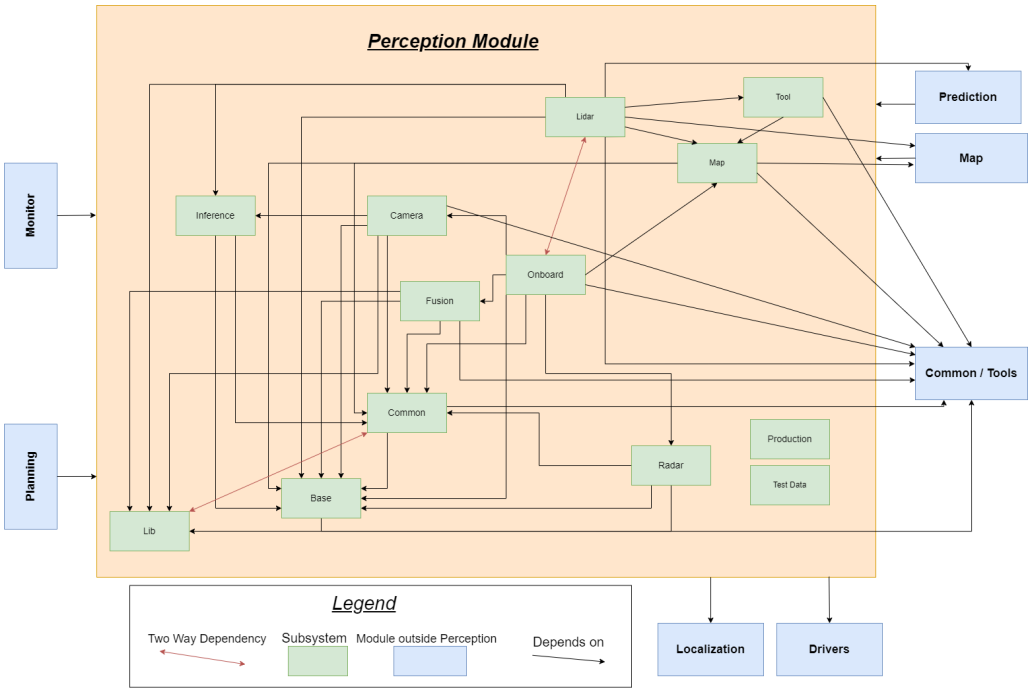

Figure 5. Concrete Architecture of Perception Subsystem

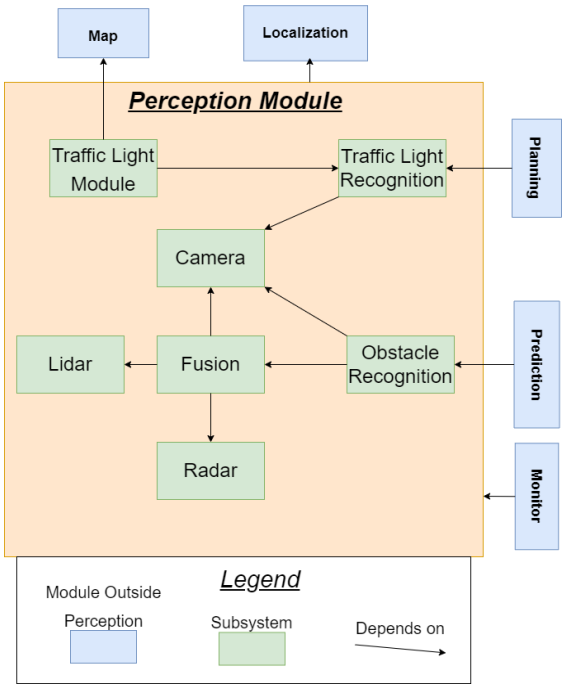

Figure 6. Conceptual Architecture of Perception Subsystem

The inner subsystem of the perception module is a complex web of dependencies. Almost all of its modules depend on either perception/perception/base or perception/perception/common, both of which contain images, models, or image processors. From this, we can conclude a majority of the Perception module is utilizing artificial intelligence in order to determine its outputs.

The most notable submodules within Perception would be Lidar, Camera, Onboard, and Radar. This is because these modules have little incoming dependencies and a lot of outgoing dependencies, meaning they are the core submodules that allow the Perception module to function.

The Lidar submodule has nine dependencies, requiring data from Lib, Inference, Base, Onboard, Map (Submodule), Map (Module), Common/Tools, Tool and Prediction. The Lidar submodule is responsible for utilizing the lidar technology within the autonomous car. The submodule would use the data from the Map modules and compare it to the models and images in Base, to which then it would compute the best path the autonomous car should take in order to avoid possible/imminent obstacles in the way.[3] The Lidar submodule is also the only submodule requiring a data output from the Prediction module, which would be a list of obstacles annotated with predicted trajectories and their priorities.[5] From this, we can deduce that Lidar is a core submodule responsible for the evasion of incoming obstacles.

The Camera submodule has five dependencies, requiring data from Inference, Base, Lib, Common, and Common/Tools. The camera submodule is responsible for utilizing the camera technology within the autonomous car. Similarly to the Lidar submodule, the Camera submodule would compute possible obstacles using the camera as well as data collected from its dependencies.[2]

The Onboard submodule has eight dependencies, requiring data from Lidar, Map, Camera, Fusion, Common, Radar, Common/Tools, and Base. Within the perception/onboard/components folder, we are able to see multiple files with names attributed to each module responsible for avoiding obstacles. With this, we can deduce the Onboard submodule is responsible for utilizing the data it gathers from the core avoidance modules (Lidar, Camera, etc).

The Radar submodule has four dependencies, requiring data from Common, Base, Lib, and Common/Tools. Similarly to Lidar and Camera, Radar also assists in obstacle avoidance by gathering information with radio technology.

Comparing the conceptual and concrete architectures, we are able to notice a few notable observations. First off, the three core obstacle detection components were present in both figures. In both figures, Lidar, Camera, and Radar, can be observed to be important components. In the conceptual architecture, they are observed to be important due to its presence and its role in obstacle detection. In the concrete architecture, they are observed to be important due to their amount of outgoing dependencies and their minimal incoming dependencies. This means that these components are using a lot of data outputs in order to get their module working. A main difference between the conceptual and concrete architectures is the Traffic Light component. If we are to observe the concrete architecture, the traffic light component is spread out sparsely amongst different submodules, while the conceptual architecture has the traffic component as one single submodule. Finally, there is little outside modular dependence in the conceptual architecture, having only the HD-Map Module used in the traffic light component, while in concrete architecture, we can observe two more outside modular dependencies that are present, Common/Tools, and Prediction.

3.1. Reflexion Analysis of Perception Module

3.1. Divergences

3.1.1 Perception/Map depends on Map

Perception/Map depends on Map as there is a “#include” for the Map modules hdmap_util.h file. However, this include is not used and there is a comment block from 2018 describing how its use as a xml map pointer is one of two options for retrieving a map. It appears option two was chosen making this dependency irrelevant.

3.1.3 Lidar depends on Prediction

Perception/Lidar depends on Prediction for feature_output.h for offline obstacle detection in surrounding lanes as well as the vehicle’s lane. This dependency was added January of 2019 and needed because perception also tracks the vehicle’s location using semantic_map from predictions. These imports allow perception to map the position of the vehicle and surrounding objects, information requested by other modules.

3.1.4 Camera depends on Base

Perception/Camera's cipv_camera.h depends on Base's lane_struct.h to create new lane objects as required by the perceived surrounds through the 360 degree camera. This dependency was added during April of 2019 and required as perception needs to identify all the surroundings of the vehicle, and lanes are an essential part of urban driving.

4. Sequence Diagrams

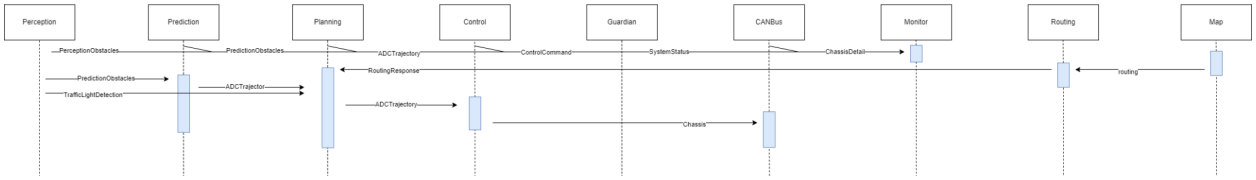

Figure 7. Sequence Diagram when the car is on the road and the traffic light turns red

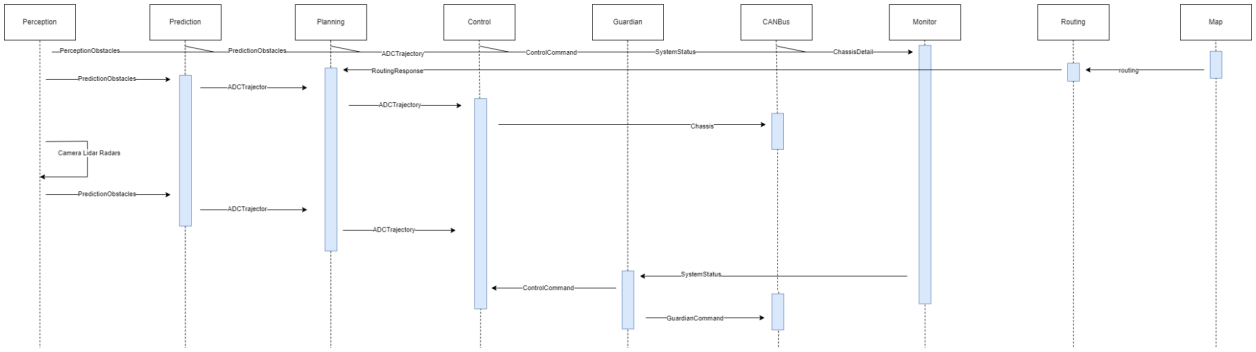

Figure 8. The car is on the road and the car in front slows down but something breaks in the vehicle as it is slowing down

5. Data Dictionary

Implicit Invocation: The architectural style where procedures can subscribe to a broadcasting system to receive events from it, or emit events to the broadcast system, otherwise known as publish subscribe.

5.1 Naming conventions

LiDAR: Light Detection and Ranging

RADAR: Radio Detection And Ranging

Pub-sub: Publish Subscribe

HMI: Human-Machine Interface

6. Lessons Learned

Through this project we have learnt a lot about concrete architectures and how to determine them, how to use Understand to learn more about a code base, and about the Apollo autonomous driving system. Understand was a tool nobody in the group had used before, and learning how to use it gave us interesting insights into how to visualize code dependencies, then be able to look through the code to see its implementation. From the understand graph, as well as the graph from the professor with pub-sub communication, we learnt how to create a concrete architecture by grouping code from different locations into one subsystem to see the dependencies between them.

We also learnt how to find and read git logs to see when different dependencies were added, and using the commit message as well as investigate the code to see why such dependencies were required. Learning to understand how the different cyber channels used to relay information connected to different objects within the code, and how/where these different entities are created was an interesting challenge. Finding the reason for dependencies through cyber channels was a significant difficulty for us, needing to find the .dag files that specified which class read from which channel, and finding the commit message with the reasoning for the dependency from that.

References

1. https://onq.queensu.ca/d2l/le/content/642417/viewContent/3865686/View
2. https://github.com/ApolloAuto/apollo/tree/master/modules/perception/camera
3. https://github.com/ApolloAuto/apollo/tree/master/modules/perception/lidar
4. https://github.com/ApolloAuto/apollo/tree/master/modules/perception
5. https://github.com/ApolloAuto/apollo/tree/master/modules/prediction