CISC 322

Group23: Assignment2

Analysis of Concrete Architecture of Apollo Platform

March 18, 2022

Authors from Wanna-Run

Yuxuan Yang	18yy82@queensu.ca
Jialing Gong	18jg39@queensu.ca
Muyun Sai	18ms78@queensu.ca
Yuxin Huang	18yh98@queensu.ca
Hairuo Li	19hl12@queensu.ca
Songyu Yang	18sy36@queensu.ca

1.Abstract

This technical report will go through a detailed overview of the concrete architecture of Baidu Apollo. As the whole team worked on mapping the source code from Apollo Github Website to our architecture using diagram drawing tool Scitools Understand, It was finalized that the conceptual architecture originally proposed was much too high-level (closer to concrete architecture) which the team needs to summarize over again. However, there is still some useful information that can be used in this assignment. As for further discovery and modification, the report concludes five major parts: the process from mapping code to diagram; concrete architecture explanation; unexpected dependencies discovery; subsystem analysis and sequence diagrams presenting in a more accurate and specific way.

2.Introduction and Overview

The whole report investigates the dependencies that exist in a more specific code base by using the conceptual architecture understanding of Apollo presented in the report in Assignment 1. The approach taken in this report is using the software Understand as the diagram drawing tool, mapping the high-level catalogs of source code to the components of the conceptual architecture and attempting to find a more concise and clear structure of induction that minimizes unexpected dependencies while maintaining the underlying rationale for each mapping. The report also discusses the unexpected dependencies based on the results to better represent the concrete implementation of the Apollo project. To be more specific, on one hand**,** there are five parts present in a relatively low dependencies relation between each other: Prediction and Planning, Planning and Control, Control and Guardian, Guardian and CANBus, Localization and HD-Map. On the other hand, there are two unexpected relations found after further discovery: Localization and Monitor; Planning and HD Map.

Taking advantage of this new perspective from the Apollo project, this report will take a further study on the Routing subsystem: processes and internal structure, with the subdirectories and internal directories used during its operations. The optimized architecture will be used for deeper analysis of the concrete sequence flow and method hierarchy for the two use cases similarly analyzed in assignment 1. Through the analysis, we can discuss the limitations and shortcomings of the perception of architecture in the former report and look more specifically at the detailed software architecture and the connections between the various components. This also facilitated our understanding of the development challenges of code-based maintenance and iteration in the future, as well as a sense of the difficulties that developers face in mastering code.

Finally, concurrency and the limitations of the study are also briefly discussed. The former is discussed mainly in terms of the operational principles of the constructive projection in the form of pub-sub, where the causes and consequences of

concurrency are discussed. The latter is a summary of the limitations in terms of correctness and accuracy based on various objective reasons.

3.Architecture

3.1Recap of Conceptual Architecture

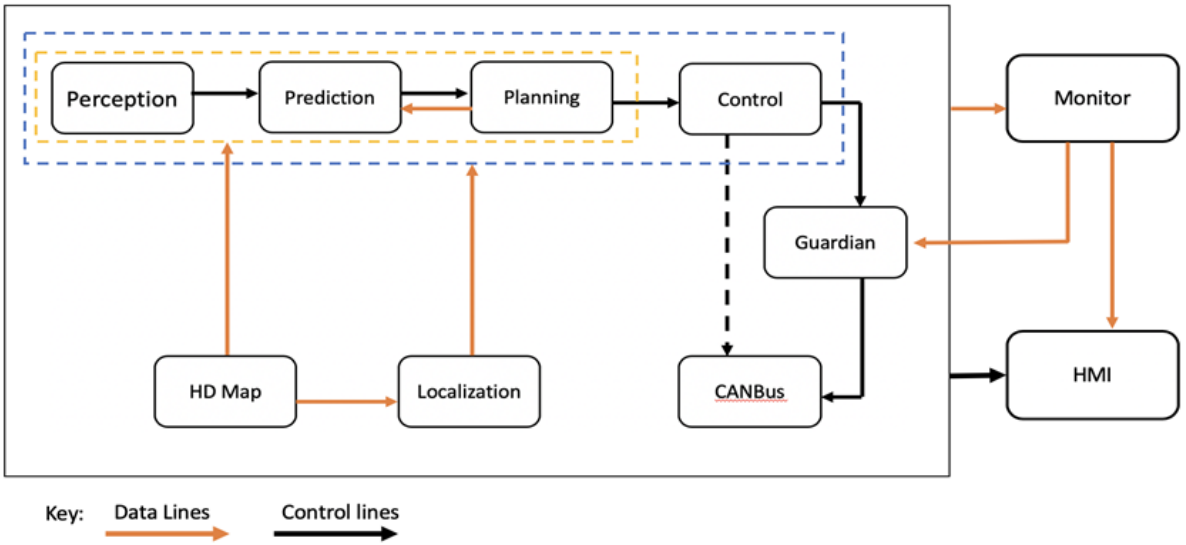

Figure 1: The pub-sub style diagram of conceptual architecture from github repository

3.2Derivation Process

The Understand tools help to map the source code to the conceptual architecture. What the folders contained are reclassified on the modules of the original conceptual structure based on our understanding of folders. After this process has been completed, a dependency map is generated (Figure 1). After remapping the source code under existing modules, many of the dependencies that were not shown before are more visible. The diagram gives the fact that these components have no obvious dependencies on another component that should be expected in the conceptual architecture. This change will be considered for correctness and a decision will be made to update or modify our conceptual view of the system.

It is worth mentioning that some modules in this diagram have significantly small dependencies on each other. In order to minimize unexpected dependencies, such dependencies with too little weight are removed. Then SciTools Understand is used to repeat the filtering of dependencies, the small dependencies would be ignored when concrete architecture is analyzed. It is clear to see the divergence between conceptual view and concrete view during the investigation.

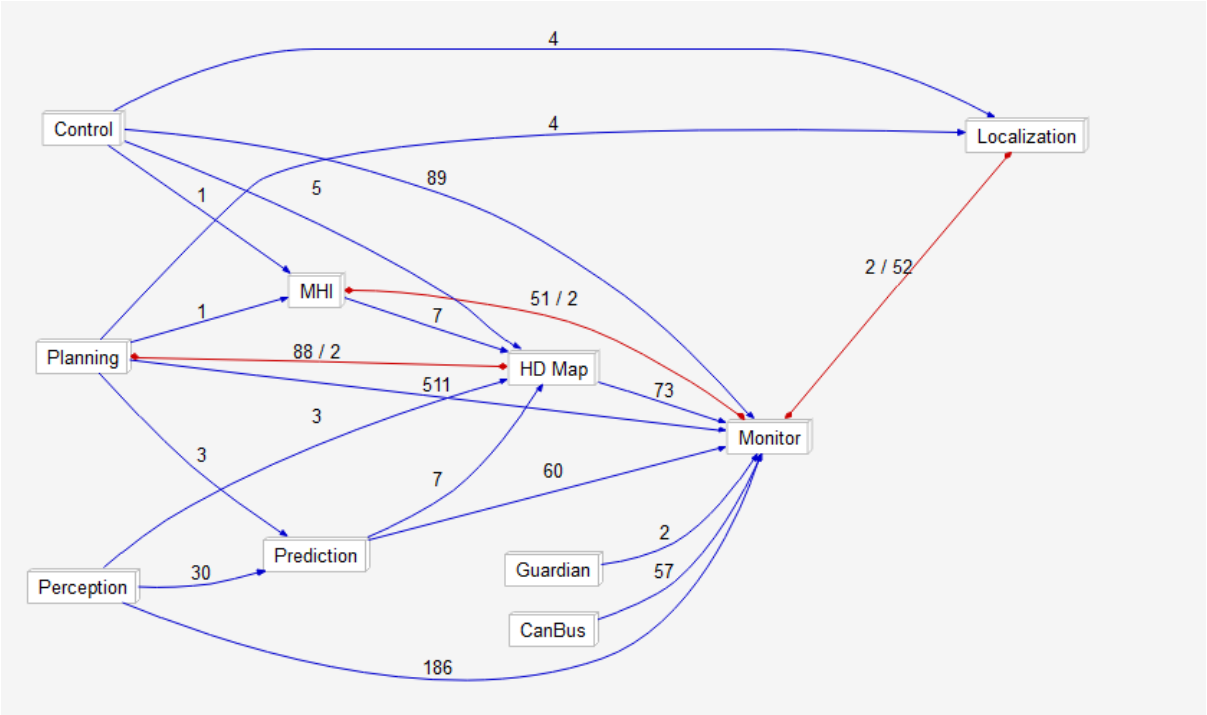

Figure 2: Dependency graph of top-level subsystems produced by SciTools Understand.

3.3Concrete Architecture

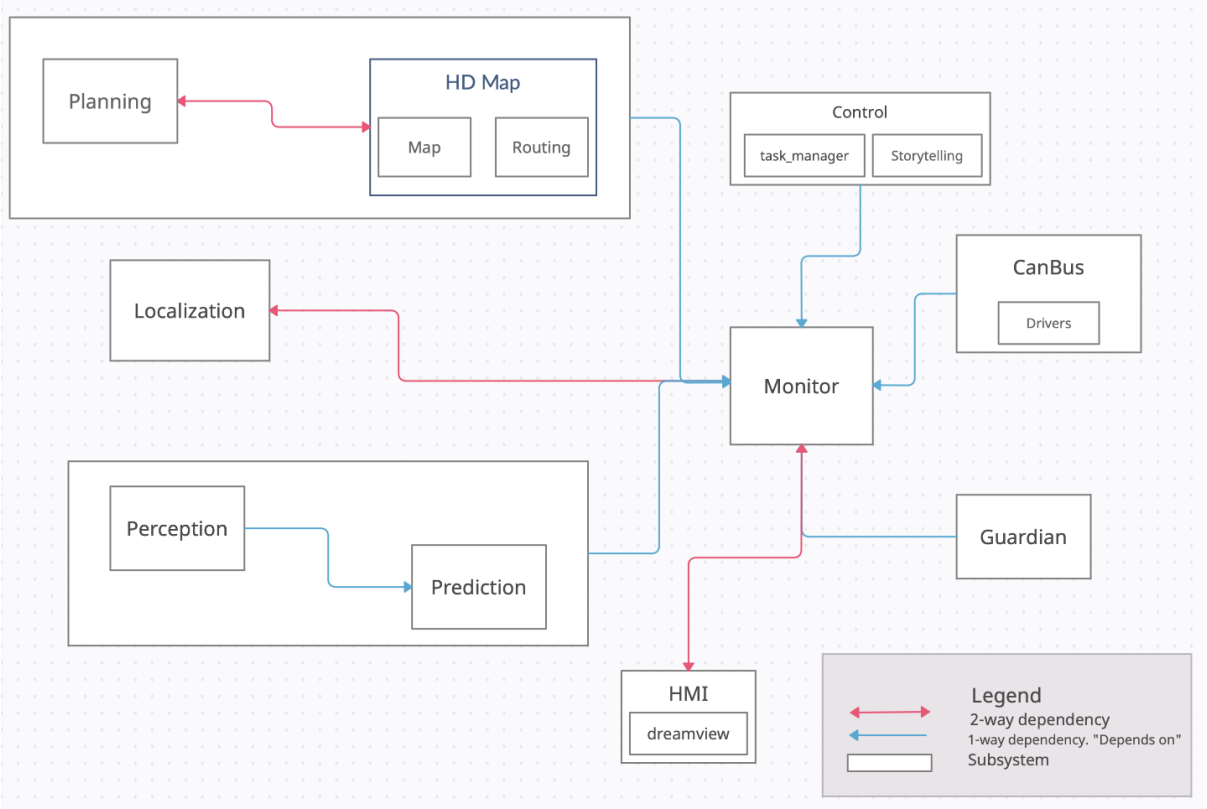

Figure 3: Concrete diagram conducted from dependency graph

For the Concrete Architecture, there is a main subsystem which is the Monitor and all other subsystems are connected with it.

The HMI and the Monitor are two-way dependency. There is a subsystem in the HMI which is dreamview. Next, for the Guardian, it and Monitor are one-way dependency and the Guardian depends on the Monitor. For the CanBus, it has the same connection type with the Guardian, with the one-way dependency and it depends on the information from the Monitor. Also, the CanBus has a subsystem named Drivers. There are two systems called task_manager and Storytelling which are the subsystems for the Control. The system of Control and the Monitor are one-way dependency and the reaction of Control depends on the information of the Monitor. For the system of Localization,it and Monitor are two-way dependency and they depend on each other.

The structures of some of the subsystems I mentioned earlier are relatively simple, but the structures of the next two subsystems will be more complicated.

There is a system and it contains two subsystems which are the Perception and the Prediction. However, there is a connection between them and the Perception depends on the Prediction, also they are one-way dependency. For this whole subsystem which contains Perception and Prediction, it depends on the Monitor and they are one-way dependency. The last subsystem contains 3 levels. For the first level, the whole subsystem depends on the Monitor, also the connection between them is one-way dependency. For the second level, there are two subsystems which are Planning and HD Map. The connection between them is two-way dependency and they depend on each other. Lastly, in the third level for this whole subsystem, there are two subsystems in the HD Map.

4.Unexpected Dependencies Analysis

In the original conceptual framework, our understanding of the interactions among subsystems was limited. By mapping source code to our conceptual framework, we discovered some unexpected dependencies between subsystems. Therefore, in order to improve the conceptual framework and get a more complete and accurate concrete framework, we further analyzed the source code file from Baidu Apollo github website.

In the process of drawing the conceptual diagram, we found that there were a number of dependencies in the conceptual view that were not particularly precise. The most unexpected of these are the dependencies between planning and prediction and the dependency between control and canbus. In Assignment 1 our study of the Apollo system, and in the conceptual diagrams we have drawn, these two groups are very closely related. This part of the difference was first brought to our attention.

Furthermore, the reason for this phenomenon (the disappearance of a large number of dependencies) was initially thought to be caused by the combination of the monitor and guardian, which detects in real time the operation of all the software in the car, so that this causes all the subsystems to depend on the monitor for their operation However, it is only the monitor that actually detects them.

The five dependencies that are missing from the dependency graph are examined in the first instance.

4.1 Missing Dependencies

Further research and comparison by merging and grouping modules shows that several subsystems appear to be missing in their relevance. They are: Prediction and Planning, Planning and Control, Control and Guardian, Guardian and CANBus, and Localization and HD-Map.

As mentioned above, we believe that this is due to the supervisory role of the monitor. We then looked at the code inside the monitor to confirm that they should be linked, e.g. that the data flow and control flow are implemented through the monitor. It should be noted that most of the modules that have a dependency on the monitor have a dependency on the common in the monitor, and not on the source code in the monitor directory. This means that most of them are linked to the common/util module in the common.

We then analyzed the code and found that the function of common/util is mainly to process files (splitting strings, adding pointers) and to read and write to files. In other words, when the other modules get the data, they write it to common/util for processing, and then the other parts read the data needed for the next step. These missing dependencies are therefore all linked through the common/until module. This explains the loss of these dependencies and the increase in dependencies in common.

In addition to the missing dependencies there are some unexpected new dependencies. They are much stronger than expected.

4.2 Localization and Monitor

The dependency between these two modules was originally thought to be a single dependency of localization on monitor, but in concrete's research it was found that there were many dependencies between localization and monitor. In the study it was easy to see that the common/vehicle-state module in monitor is very dependent on localization. The function of the common/vehicle-state module is to represent vehicle information, which includes vehicle coordinates. The acquisition of the vehicle coordinates is dependent on the localization module, which leads to the dependency of the monitor on localization.

4.3 Planning and HD Map

This is also supposed to be a one-way dependency of planning on HD Map, but it is also found to be a two-way dependency in concrete. The part of the HD Map that depends on planning is the pnc-map.

The function of this module is to parse the results of the routing and convert each section into the form of reference_line data. The parsing of the routing data leads to a dependency on planning.

5.Subsystem Analysis: Routing

The main role of the Routing module, as its name suggests, is to generate routing information based on requests. Routing's main functions are under the module directory: routing. Under routing there are several sub folders containing the subsystems that create Routing. The most important subsystems include Common, Graph, Topo_creator, Core, Strategy, Routing result, and Routing component. The Routing component looks externally for the corresponding files based on the requirements within the subsystem.

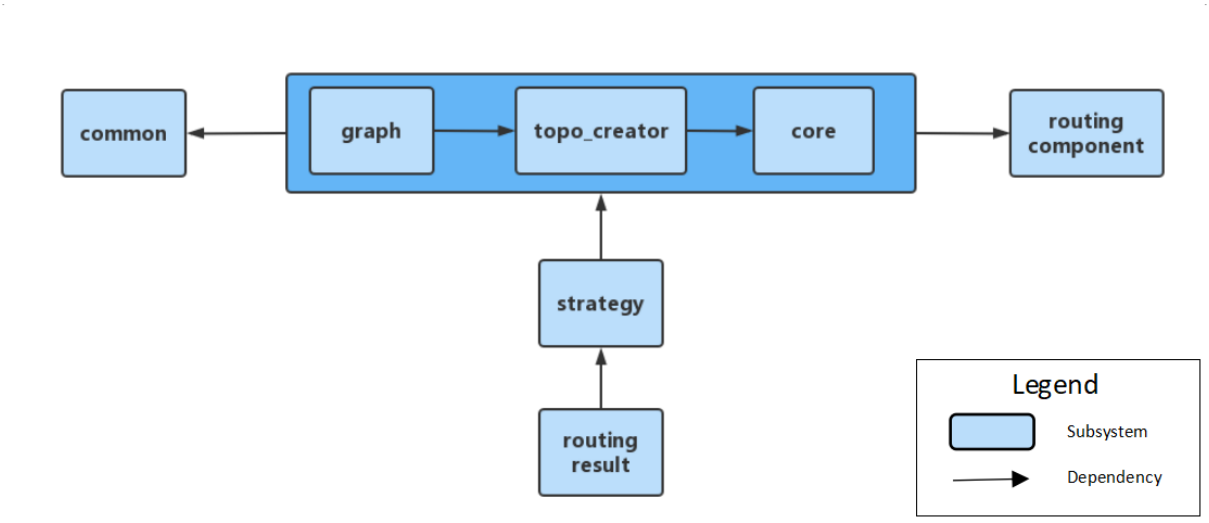

Figure 4: Conceptual diagram of subsystem Routing

There are seven subsystems showing on the graph above. The subsystem of the graph is used to create the high-precision map. Then, the topo_creator uses the map from the graph to transform the information to the data form of topo. After that, the subsystem of the core generates a new map using the transformed topo data. The common contains the initialize variable, the system of graph, tot-creator and core need to get the initialize data from the common. For the routing component, it calls external variables to help generate the map in the form of topo data. The subsystem

of strategy is used to search to find the optimal routing. In order to fulfill this, it needs to get the data from graph, topo-creator and core. The final subsystem is the routing result. It collects and organizes data from all previous subsystems to get the best results.

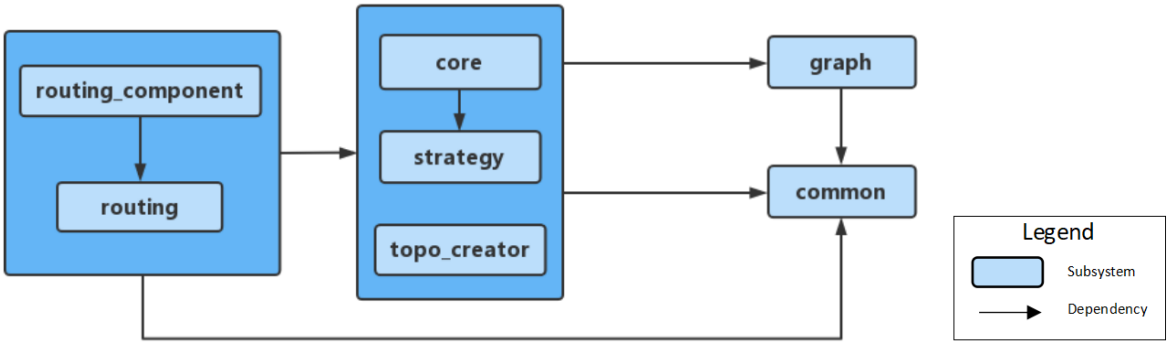

Figure 5: Concrete diagram of subsystem Routing

For the routing, it must get the data from the routing-component to get the variables from external systems. If we want to get the routing, we must get the information of core, strategy and topo_creator. For the core and the strategy, we must get the core first, then based on the new map from the converted topo data, we can search and find the optimal map. In order to get the high-precision map in the subsystem of a graph, it needs to get the data from the core, strategy and topo_creator. For the common, the initialization variable is used for all these subsystems.

In the course of our subsequent research on routing, we discovered that we had misunderstood the function of the graph and strategy parts of routing, which led to some errors in comparing their dependency relationships. As a result, there are some dependency relationships that we did not expect.

5.1 Graph and topo_graph

Based on our previous guesses, because the map type used in routing differs somewhat from that used by humans, routing needs to search for paths by topo's map type, and it is topo_creator that does this. However, to investigate this issue, we found that the problem was probably caused by the fact that the map in poto format was not generated by the topo_creator, but that the graph was already a poto file when it was obtained externally. The function of topo_creator is only to make real-time changes to the variables in the graph, such as traffic lights, turn restrictions, etc.

5.2 Strategy and Core

Initially we thought that the content of the core would process the topo map for subsequent searches, and that the processed map would be used by the Strategy to select the shortest route using the A* algorithm. However, we subsequently discovered that core/navigator.cc in core had a dependency on strategy. After

interpreting the code, we believe that strategy itself only has the function of searching for the shortest path and does not export the organization and output the results. So in fact all this work is done in core/result_generator in core, based on the reference to strategy made by core/navigator.cc to do the search. Therefore, this dependency should be a dependency of core on strategy.

6. Sequence Diagrams

Use case 1: Detecting Routing Error

This sequence diagram which matches the concrete architecture has a few differences with the first sequence diagram in the first assignment about conceptual architecture. We now consider the Monitor module has more dependent relationships with all the other modules and it is in charge of tracking and updating status for the other modules. In this case, the Monitor module keeps tracking on other modules and updates status so that the whole process can execute successfully. When Monitor detects something wrong in one of the other modules, it will send a signal to the Guardian module to check again whether the current routing is feasible or not. After Guardian convinces the unsafety factors, it will stop CANBus from controlling the vehicle and halt the routing process. Then the Monitor also shows the message to the user by HMI.

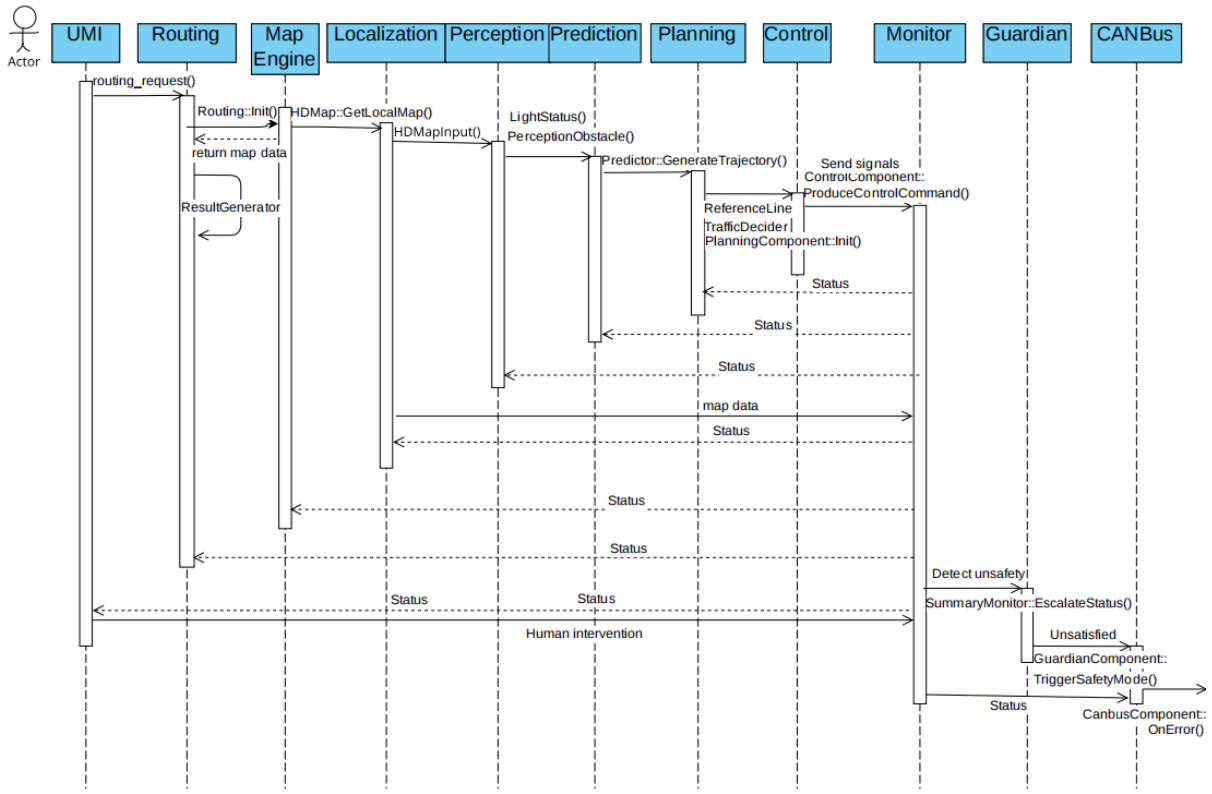

Figure 6: Sequence diagram of detecting routing error

Use case 2: Temporary Parking

The second use case happens when the user has to stop the vehicle accidentally so that the system needs to find a safe parking position for the vehicle. Localization, Perception, and Prediction modules are mainly responsible for detecting the surrounding environment and then the Planning and Control modules will formulate the routes. Meanwhile, the Monitor will keep tracking on all these modules to ensure there is no mistake in the process. Then the Guardian module works to examine if the current position is possible for parking and sends signals to CANBus to continue parking if the condition is safe. Monitor also sends messages to let the user know the situation through HMI.

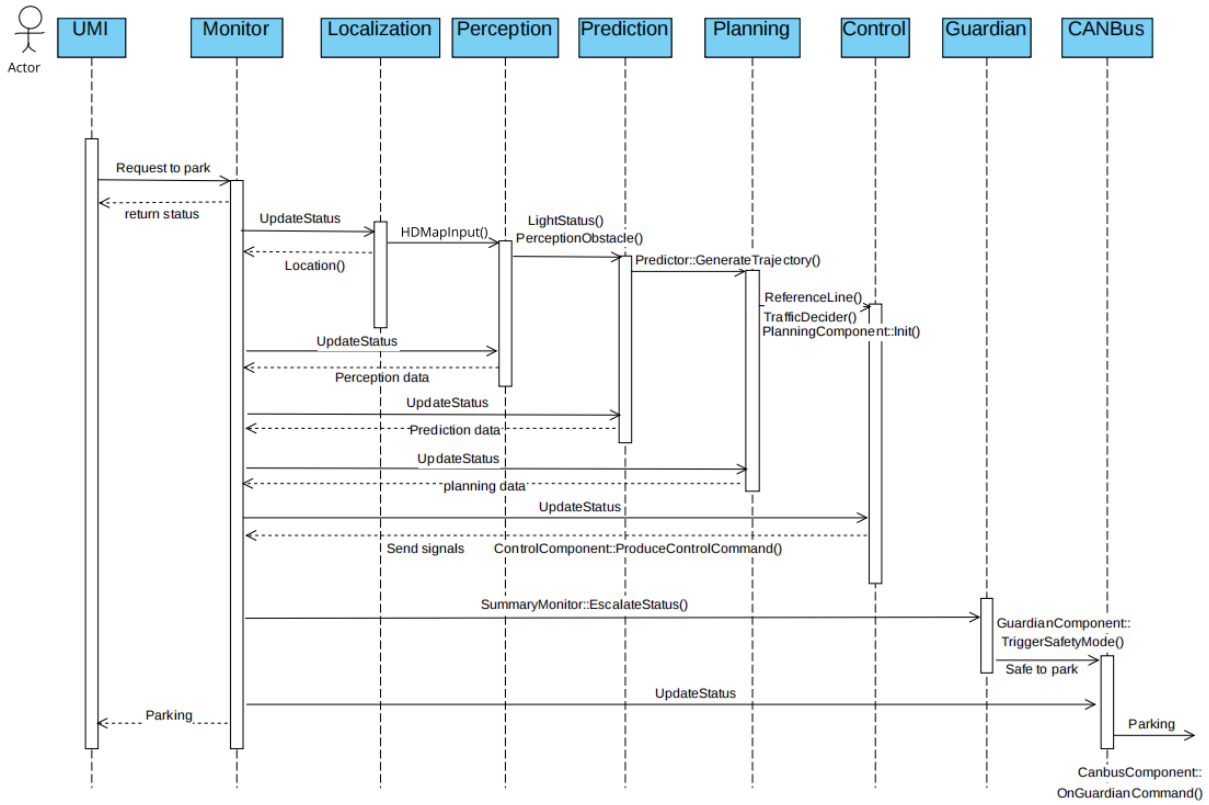

Figure 7: Sequence diagram of temporary parking

7.Team Issues

As the team tried to illustrate the dependencies relation of source mapping and implement it through Understand, it was discovered that some dependencies were more visible than before. and there are some modules that seem relatively essential become less important through the diagram. This makes the later analysis become complex since it is hugely different from the analysis analyzed from conceptual architecture in Assignment 1. Meanwhile, as the conceptual architecture we wrote in Assignment 1 was much too specific, we have to make the conceptual analysis again and make the comparison with concrete architecture explicitly.

8.Concurrency

Apollo is built around a pub-sub architectural style. Pub-sub style is a pattern that senders of messages (publishers) do not program the messages to be sent directly to specific receivers (subscribers), but instead categorize published messages into classes without knowledge of which subscribers. This allows for the management of concurrency to be straightforward to developers, since the components of the system often need to achieve only a corresponding subset of the total messages transmission when events occur. When publishers announce events, the subscriber would register interests on this associating procedure. Then the created tasks will be assigned into a linear sequence within a subsystem, running on threads and are not bound to any individual thread. This process can be shown as Figure 3, in which perception and planning, and localization are two parallel threads. Both of them transits data to monitor and have a dependency with monitor. For the whole Apollo system, Tasks can be run on multiple threads at different stages. This model supports reuse if any components register interests to the publisher and allows for deferred or scheduled processing, providing opportunities for better scalability.

9.Limitations

We do not have a particularly good understanding of the code programming language. The names of some of the files in the various source code directories do not fully demonstrate what these files do. This makes it much more difficult to understand these files. The fact that we do not fully understand the function of each module leads to uncertainty about the exact distribution when we draw the conceptual diagram. In addition, the inability to fully parse the module source code for its specific role makes concept mapping biased.

Besides, because this paper specifically analyzes differences between the conceptual and the Apollo software architecture, It is almost impossible to expand to other areas when discussing the significance of reality. Theref, we give more attention to the Apollo structure than to extended thinking.

10.Lessons Learned

The most important lesson we learned during working on the concrete architecture of the Apollo system is to analyze the divergence between conceptual view and concrete view. The work is all based on our understanding of each subsystem, and learning to use SciTools Understand plays a huge part in this. We are able to know interactions and dependencies on the Apollo software system from an analytical point of view. This is also what we have learned in previous courses.

11.Conclusions

By mapping the source code according to the conceptual structure, reflection analysis was able to discover more specific dependencies between some modules. The unexpected dependencies are particularly important in this REPORT. This is because it provides a more nuanced divergence between conceptual view and

concrete view. Besides, One of the subsystems has been chosen for a more detailed discussion. After the analysis has been carried out and a new concept has been gained, two user cases are discussed. The sequence diagrams are used for a more visual presentation.

In general, this paper focuses on the discrepancies between the conceptual and concrete architecture in the assigned subsystems. Based on all the discussion above, we were able to have a new understanding of the Apollo software architecture.

12.Data Dictionary

SciTools Understand: An application that developers map source directory onto dependency graph.

Dependency: One module depends on another module as the primary source to operate.

The pub-sub architecture style: Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.

Concrete architecture: The structure of each component specific implementation within a system.

Topo graph: Topo graph is exported by proto buffer. Protocol Buffers is an open source project by Google. It is language-independent, platform-independent and very extensible. protocol buffers is commonly used to serialize structured data.

Reference

1. Microsoft Document. “Publisher-Subscriber pattern”. Retrieved from https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

2. salmcode (August 23, 2017).“The First Part of the [Apollo Source Code Analysis] Series [common].” Retrieved from https://blog.csdn.net/learnmoreonce/article/details/77511338

3. Steve (October 20, 2021.)“Apollo 6.0 pnc_map Analysis” Retrieved from https://zhuanlan.zhihu.com/p/419350318

4. Yuchen C. Et. all. (2022).Assignment 1 : Conceptual Architecture of Apollo. Retrieved from https://docs.google.com/document/d/1f3Y1uISdpA9U6SUEgdXLMi7XItG2nFlHU3zRkkc4uMw/edit

5. Baidu. (n.d.). Apollo.(February 20, 2022) Retrieved from https://apollo.auto/

6. Paul (February 5, 2019). “Baidu Apollo's Routing Module”. Retrieved from https://paul.pub/apollo-routing/#id-topocreator