Concrete Architecture of Apollo

CISC 322 Project Deliverable A2

Date: Mar, 21, 2022

Daniel Baik
Oscar Chen
Yanzhang Ma
Tyler Pitcher
Josh Rutledge

Abstract

In this assignment, we map from the conceptual architecture to the concrete architecture using the reflexion models. In our analysis, we discover more modules which are ignored in the conceptual architecture ,and the new dependence between each subsystem. In many different subsystems, we focus on the routing subsystem and find a new dependence pattern inside this system. This new pattern provides us with new logic which can be used on the whole program. In the dependence analysis which provides supporting evidence in the reflection models, we analyze two cases-optimization of the rerouting route and how to handle the traffic accident.

Introduction

Apollo, a software for driverless vehicles, is a complicated open source multi-participant online project which depends on many unspecified programmers. Because the project is still updating and the introduction is facing the public, demonstrating a distinct whole picture of the system architecture is challenging work. Finding the dependence between each subsystem and its module can build a clear concrete architecture graph for understanding the system architecture. To reach the concrete architecture,we start at reference architecture, then we use the reflection model to map from the conceptual architecture and analyze the module dependence by combining the directory structure and material case. In addition, each subsystem is made by many different modules, and to deeply understand the whole system, we further analyze routing subsystems because we think it plays a core role in whole systems and can provide a logic pattern to understand other subsystems.

Architecture Derivation Process

Discovering Apollo’s concrete architecture was a long and tedious task that took several days to accomplish. First, our group’s previous conceptual architecture had to be reviewed before any progress could be attempted on Apollo’s concrete architecture. Once a strong understanding of the conceptual architecture was achieved, we set our eyes on the even more challenging task of understanding Apollo’s directory structure in the Understand software. Utilizing the Understand software meant reorganizing and sorting the entire directory of Apollo. Once Understand’s dependency graph could be generated from the reorganized Apollo directories, we could then start work on analyzing discrepancies between the communication model graph provided for this assignment and our group’s dependency graph. This meant we had to plunge deeper into the Understand software to alter the dependency graph.

In order to understand an entire software's concrete architecture, we must first consult the conceptual architecture. The purpose of reviewing the conceptual architecture is to understand how developers view the system, and it allows us to review gaps between the conceptual architecture and the concrete architecture after building the concrete architecture. Understanding the conceptual architecture makes discovering the concrete architecture less complicated as we start to expect module dependencies between two different modules. Once we understood what to expect from the concrete architecture, we could begin working. By reviewing Apollo's conceptual architecture, we could detect unexpected dependencies within the Understand software.

Now with the knowledge that comes with reviewing conceptual architecture, we could begin working with the Understand software to discover dependencies between modules. The first thing that needed to be done was to separate the files into two categories those related to modules and those not linked to modules. An example of non-related files is those that are a part of Apollo's high-performance runtime framework called Cyber RT. In contrast, files directly related to Apollo's perception or prediction modules were placed into a separate folder. The tricky part of the assignment was not separating and sorting files that had an obvious purpose but instead was understanding what files with little to no documentation did. Files in the drivers or common directories were the most challenging to figure out what their purpose was. Once everything was reorganized, we could indeed start to leverage the power of Understand by building a new dependency graph based off of the new architecture structure. This new graph displays Apollo's concrete architecture by showing the true dependency between existing modules.

Despite having a dependency graph based directly on Apollo's architecture, the work was still not done. We had to analyze the given communication model graph for Apollo with our own dependency graph in order to find unexpected couplings between different modules. The question presented with these unexpected links was whether or not they were valid links at all. It was entirely possible that connections between modules in the graph were created through user error; furthermore, it was likely that links that should exist between modules would not be present in our dependency graph. This could occur by improperly restructuring Apollo's directories. Analyzing the given communication graph and our group's dependency graph ensured that no missing links or links that existed in the graph but not within the software went undiscovered. After all links between modules had been verified, our group now possessed a dependency graph for Apollo's concrete architecture.

The process of understanding Apollo's concrete architecture was a painstaking and tedious task to ensure no mistakes went unnoticed. We began by reviewing the conceptual architecture. Then, we quickly moved on to breaking down Apollo's

directories and files to understand how they work together; furthermore, we wrapped up our work by reviewing the communication graph alongside our own graph to detect potential errors in our work. Comprehending an entire software system is a slow yet rewarding task to accomplish.

Subsystems of the Concrete Architecture

Apollo is a distributed system. Therefore, it has no so-called hierarchical structure but a network structure. A module is called a node, and each node can subscribe and publish messages. And the concrete architecture contains several modules which considered subsystems. Each subsystem has unique functions that form a pub-sub architecture. In the list below there, several critical subsystems are being explained.

Apollo's Subsystems:

Map module:

The map module reads the HD map and transfers it to the Map class for Apollo. The module includes **data:** generated map, **hdmap:** HD map (has an **adapter** that reads the map form XML file), **pnc_map:** a map for planning module, **proto:** elements of the map (pedestrian crossing, lane markings, etc.), **relative_map:** the middle layer between map module and planning module, testdata and tools.

Localization module:

The localization module has two primary functions: output the vehicle location information to the planning module and outputs the vehicle information (by **IMU** and **LiDAR**) to the control module. There are three localization methods. The first is **GNSS+ IMU** localization, the second is **NDT** localization and the third is **MSF** which is a combination of the first two methods. There are **msf, ndt, rtk** in the module which are all localization methods.

Perception module:

The perception module has its module entry **Production** launch corresponding dag. There are multiple subsystems defined by **onboard**. Used to process different sensor information (LiDAR, Radar, Camera). The process of each sensor is basically preprocessing, object identify, regional filtering and tracking. **Inference** is a deep learning model and mainly implemented the deployment if three models of **caffe, TensorRT and paddlepaddle**. The trained model is in "modules\perception\production\data" and process deployment and online calculations. There is also **camera:** lane line recognition, traffic light detection, and obstacle recognition and tracking; **radar:** obstacle identify and track; **lidar:** similar to

radar; **fusion:** fuse the different sensors. The main process is from **production** to **onboard** and through sensors and end with **fusion**.

Prediction module:

The module code process is inside the "**message_process.cc**", through the container, scenario, evaluator, and predictor outcomes the predicted trajectory. The data structure of the module has three containers including **pose:** current vehicle location information, **adc_trajector:** planned trajectory info, and **obstacles:** precepted obstacles info. Two defined **scenarios:** cruse and junction. Several evaluators most are deep learning models including **cyclist_keep_lane:** rule-based model cyclist keeps or changes lane predict, **pedestrian_interaction:** LSTM model pedestrian predict, **cost:** cost functions, **mlp:** NN model vehicle turning or go straight predict, **cruise_mlp:** like MLP evaluator, **junction_map:** junction vehicle predict based on **SemanticMap**. Several predictors include **free_move:** generate free move trajectory points, **lane_sequence:** filter lane sequences, **move_sequence:** like lane sequence, **junction:** obstacles move towards predict, **interaction:** create posterior prediction results.

Planning module:

The function of the planning module is to plan a trajectory that the vehicle can travel based on the result of perception prediction, current vehicle information, and road conditions, then pass the trajectory to the control module. The input of the planning module is in "**planning_component.h**", input parameters are **prediction_obstacles:** obstacles info, **chassis:** speed, acceleration, heading angle, etc. info **localization_estimate:** location info. The output is in "**Planning Component::Proc()**". Map information is directly read in the function but not passed in parameters. The module entry is **PlanningComponent**, it registers, and pub-sub in **Cyber** also registers the corresponding planning class. The execution of the module will be triggered if the message is collected. The planning class mainly implemented two functions, including **ReferenceLineProvider:** generate reference line and the second function is to execute the central planning process. The primary process of the planning module is to choose the corresponding planner, like **PublicRoadPlanner**, which defines scenarios, then separates scenarios into stages, further separates stages into tasks. Once the tasks are executed, the scenarios are completed. **SenarioDispatch** is used to switch scenes based on the reference line generated by the **ReferenceLineProvider**.

Control module:

The control module has **Chassis, LocalizationEstimate, ADCTrajectory** as input and pass-through control module **ControlCommand**. The main entry is in "**control_component.cc**", there are three stages of the main process, read inputs, generate control command and send control command. The module generate

command by "**controller_agent**" include **LonController, LatController, MPCController and StanleyLatControll** which control the main function like moving or stop.

HMI module:

The HMI also know as Dreamview module helps developers visualize the output of other relevant autonomous driving modules. The module will monitor the message from all other modules.

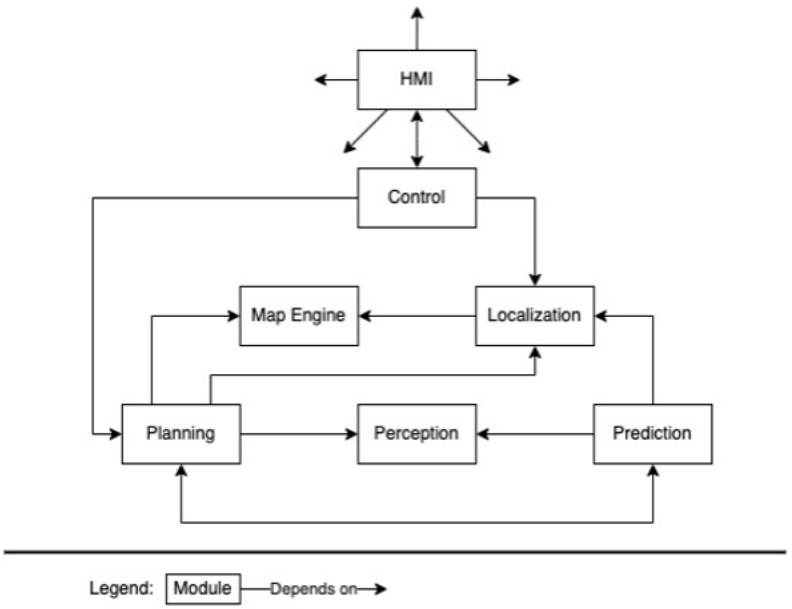
Monitor module:

The monitor module aims to check the status of hardware and monitor the health of system. The hardware includes **camera, gps, etc.** The software includes data integrity, data frequency. And generating end-to-end latency stats report.

High-level Architecture Reflection

Discrepancies do exist between the conceptual and concrete architecture our group is presenting. A storytelling module is not present in the conceptual architecture and was added to the concrete architecture. New dependencies for the map engine and perception modules were discovered. The significant difference between the conceptual and concrete architecture is the splitting of the human-machine interface module into a new monitor module. The figures below display the difference between Apollo's conceptual and concrete architecture.

Figure 1: Apollo's Conceptual Architecture



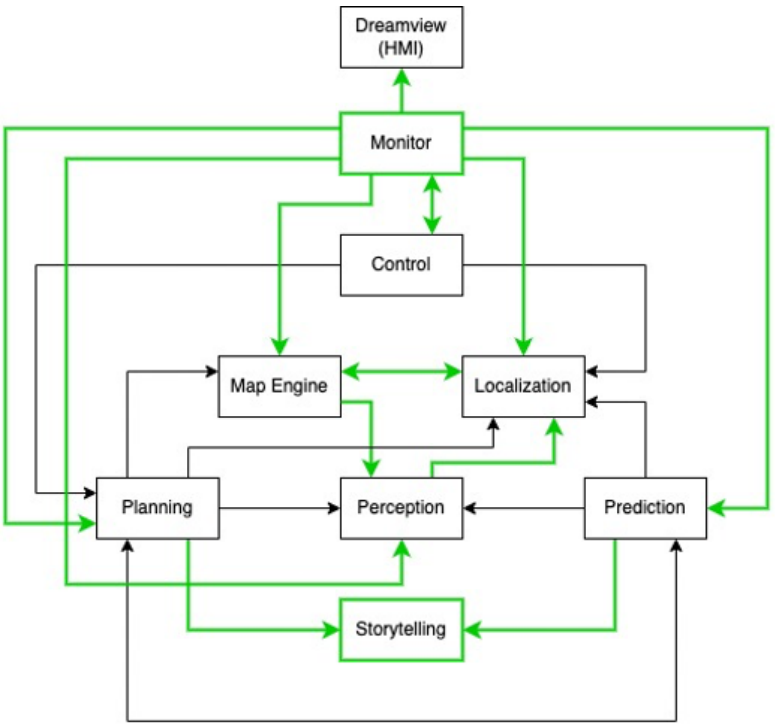


Figure 2: Apollo's Concrete Architecture

Any lines or modules highlighted in figures one or two represent new revelations made while reviewing Apollo's directories and files. There are five reasons why the concrete and conceptual architecture do not align with each other:

1. The storytelling module was not present in our group's proposed conceptual architecture. This was due to the fact that neither the planning nor prediction modules mentioned the existence of the storytelling module. Only after analyzing Apollo's directory structure did the presence of the module become evident.

2. The map engine requires the perception module in order to get the perception obstacles. It is unclear how the map engine utilizes the obstacle data given by the perception module; however, according to the communication model graph given for the assignment the map engine depends upon the perception module.

3. Grouped in with Apollo's map engine are the drivers necessary for the global navigation satellite system and inertial measurement unit necessary for the localization module. Localization utilizes these drivers to localize the car within the high-definition map. This was noted in our group's conceptual architecture; however, what was not clear at the time was that the map received a localization estimate from the localization module. Apollo's own main software overview documentation leaves out the map's dependency on the localization module.

4. Similarly, to the map engine, Apollo's perception module receives a localization estimate from the localization module. Velocity and angular velocity are

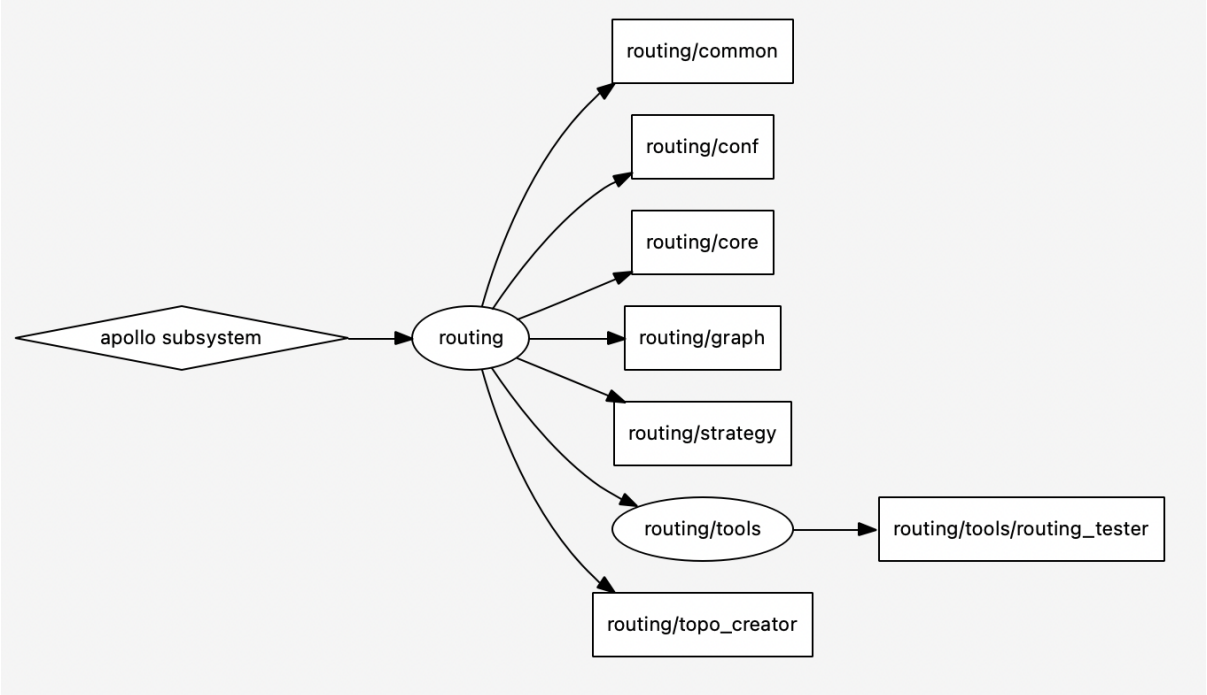
included in this module and are required by the perception module. As the localization unit has evolved, its tasks have changed alongside it; therefore, the perception module has grown to rely on the localization module.

5. At first glance, it appeared that the monitor and human-machine interface appeared to share similar responsibilities and were combined to form one module. After analyzing the dependency graph generated by Understand and looking at the communication graph, it was evident that the two were really separate modules. The monitor's main responsibility is to monitor the entire system and detect errors. While the human-machine interface is mainly used to interact with the users.

Our group's proposed concrete architecture is slightly different from our conceptual architecture; however, the fundamentals are still the same. Two new modules were added to the concrete architecture, and minor connections were added to the dependency where needed. Each new alteration has a valid rationale, as explained above. Changes between conceptual and concrete architecture are normal to find in software projects. They are often introduced by developers because of an increase in efficiency, change in priorities, or other valid reasons.

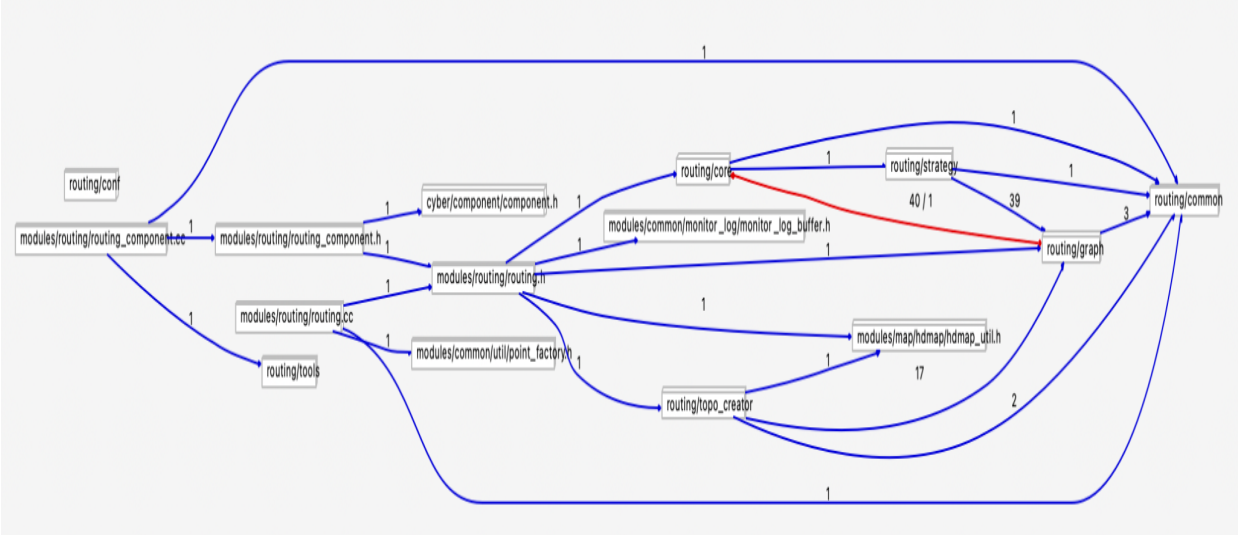
Delve into High-level Subsystems – Routing Module

The first graph shows the general architecture of the routing module.



In this subsystem, it is visible that there are 7 folders directly under routing with another folder located inside the "tools" folder. Now, looking at the more detailed

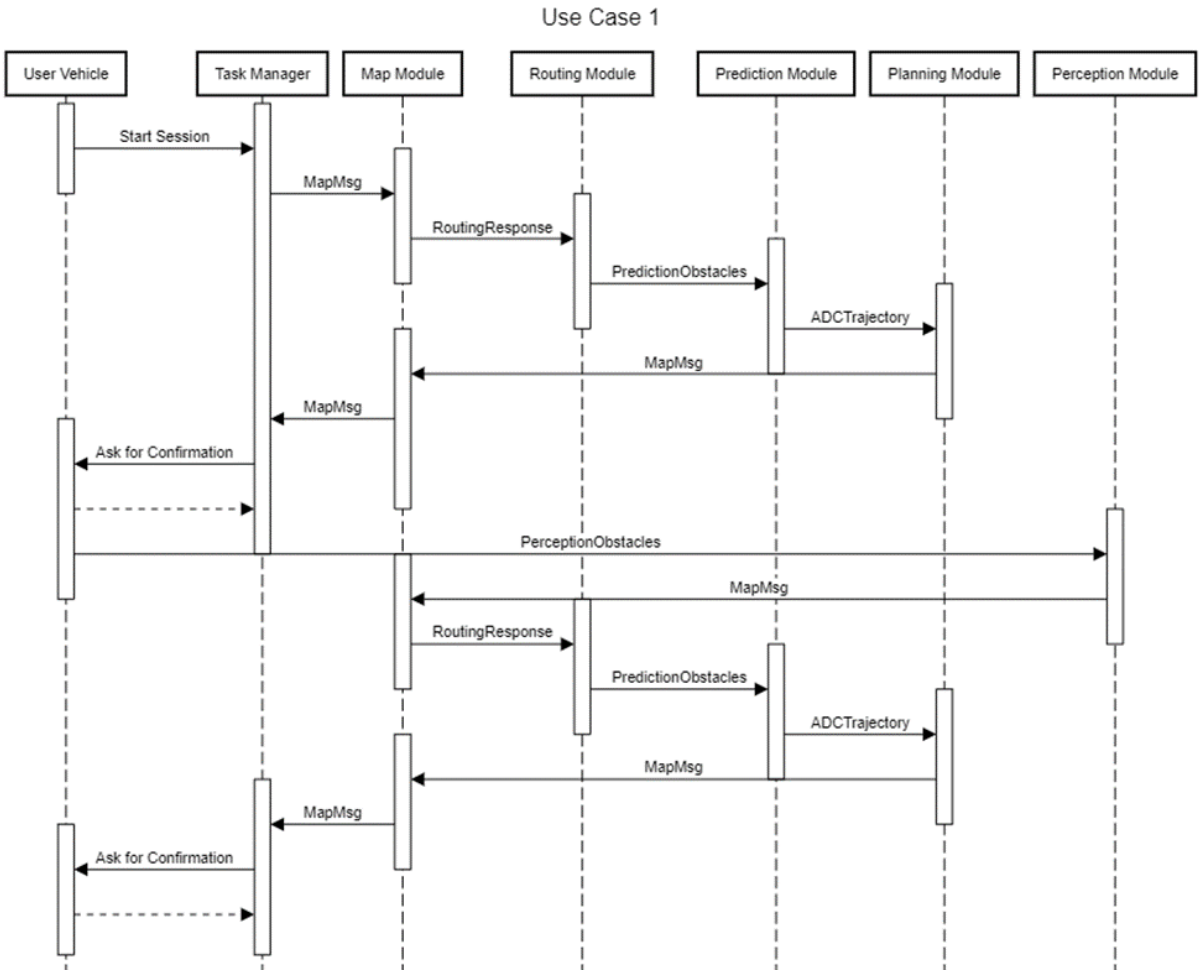
graph that shows internal dependencies inside the routing module, it is visible that there are many dependencies both between inner components inside the routing module and between a routing component and components from other modules in Apollo.



The subsystem architecture has been derived using the butterfly-dependency graph of the routing module. From there on, the next task was to locate all the dependencies and create a new architecture on Understand using the located dependencies whether they are between routing and some other modules or between inner components of the routing module. The second graph shows the internal dependencies of the architecture that was built by sorting the files that are depended on by the routing module's inner files. As shown, a module does not simply consist of dependencies between files inside a single module but the connections between different modules is necessary in order to have all the pieces of a program to work correctly.

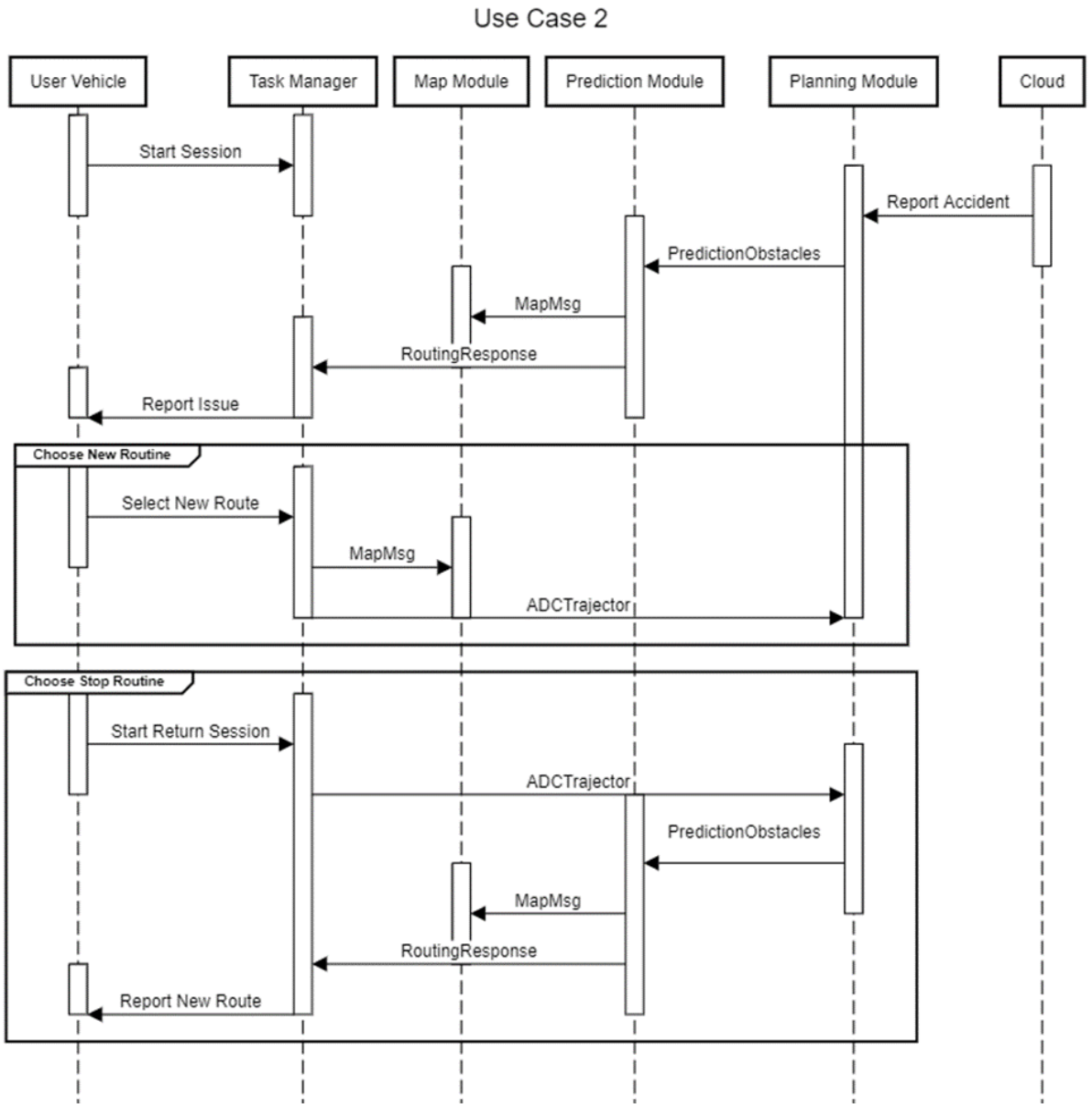
Use Cases

Use Case 1 – Finding and Altering a Route for Optimization



Our first use case involves rerouting a route for optimization considering current traffic conditions. This case begins with the task manager which hosts a session created by the user's vehicle. The task manager delivers a user's route destination to the map module as a MapMsg. The map module sends a RoutingResponse to the routing module. The routing module determines a possible route and transfers this to the prediction module which then predicts the arrival time for the proposed route. The prediction module receives PredictionObstacles. The planning module then receives ADCTrajector aand can set up a plan for the trip. This route is then forwarded to the map module as a MapMsg and applied to the map. Finally, it is sent back to the task manager and to the user vehicle so the trip may start. During the trip, the perception module perceives and notifies the map module of any obstacles in the planned route as PerceptionObstacles. The perception module sends a MapMsg to the map module. The routing module receives a RoutingRespsonse to recalculate the route. Here, the prediction module receives PredictionObstacles and calculates the new route. Next, the planning module is called, and send the new plan to the map module in the form of a MapMsg. Finally, the map updates the task manager which in turn updates the user's vehicle.

Use Case 2 – Accident Detection



Our second use case takes place after an accident occurs. To begin, the cloud platform will locate the accident and make a detailed report to the planning module. After receiving this data, the planning module generates information for the prediction module in the form of PredictionObstacles. The prediction module calculates route information for the map module and task manager. The map module receives a MapMsg. Then, the task manager reports the issue to the user's vehicle. From there the user has two choices, they may choose a new routine, or they may choose to stop the planned routine and return to where they started. If the user selects to start a new routine, the user's vehicle will send the new destination to the task manager and the task manager will forward this information to the map module and the planning module in the form of a MapMsg and ADCTrajector accordingly. The planning module will process the data to other modules the same as previously discussed in Use

Case 1. If the user selects to stop routine, the task manager will give the original start point location to the planning module. And carry out a similar routine.

Conclusion

We map from conceptual architecture to concrete architecture using the reflection model and find there is a monitor system between Dreamview(HMI) and rest subsystems. The dependence between each subsystem is re-organize in the concrete architecture by deeply analyzing the directory structure and filling the gap between conceptual architecture and the concrete architecture which we modified during the analysis. What's more, we deep dig into the routing systems and analysis the material use case which in rerouting a route for optimization considering current traffic conditions and subsystems communication after an accident occurs.The material use case not only demonstrate the core position of the routing systems, but also provide new evidence to build dependence between different subsystems and to investigate the gaps of the architecture. The working experience and result of the concrete architecture will benefit future study on the systems and provide us ideas that what systems can be added or removed..