CISC 322

Assignment 2
Concrete Architecture of Apollo
March 5, 2022

Zephyrus

Yucan Li 18yl259@queensu.ca

Yuzhe He 18yh46@queensu.ca

Xuchuan Mu 18xm24@queensu.ca

Yiming Zheng 19yz38@queensu.ca

Wenran Hou 18wh10@queensu.ca

Mukun Liu 19ml13@queensu.ca

Abstract

This report mainly focuses on analyzing the concrete architecture of Baidu's Apollo system. We have done the conceptual architecture for Apollo in the previous project and a more integral conceptual architecture will be revised at the beginning of this report. Based on the Apollo document and project from other groups, the architecture style of Apollo is pub-sub. This style constrains the recovery of message traffic between subsystems by the Understand tool. Therefore our group combined the professor's graph of pub-sub message traffic visualization and dependency graph from Understand to derive a complete concrete architecture.

In order to make the dependency between subsystems sound and valid, our group has put most of our research time and effort into looking at system calls and function implementation at the coding levels. Through the help of the Understand tool, the efficiency of the investigation is greatly improved. Focused subsystems can be highlighted and irrelevant dependencies other than focused subsystems can be hidden, these features help us easily recognize important dependencies and relife us from tangled relationships. Some components inside subsystems that we never noticed before are revealed with inextricable dependencies with other subsystems. As a result, the final concrete architecture would show an intricate relationship mapping.

Introduction

The purpose of this report is to deep dive into the Apollo system and construct a concrete architecture. This report will be based on the conceptual architecture in the last research and focus on the code level of the system. Discovering code and finding the implementation of the interaction between different modules. The first part of our report will explain the process of how we derive our final concrete architecture. The next part will discuss one of the subsystems of Apollo. Investigation of the subsystem will be advanced from both conceptual and concrete view and details about its inner structure and interaction with other subsystems will also be explained. The third part will be a reflexion analysis on the overall architecture and a chosen 2nd level subsystem. This part will present the discrepancies between the conceptual and concrete architecture in both subsystem level and overall architecture, including the implementation of the interaction between each module on code level we've founded in conceptual architecture and compare them with new findings for the concrete architecture. A chosen subsystem will be explained in detail in this section. In the fourth part, we will present two sequential diagrams to show a clear workflow of Apollo. Finally, we will combine everything and come up with a conclusion and show our limitations and lessons learned.

In this study of concrete architecture, we are using the Understanding tool provided by SciTools to come up with a graph with all dependencies between each module in the Apollo system. This gives us an overall view of the architecture and helps us to have a better understanding of Apollo. A graph visualization provided by our professor helps us how the Apollo system works on a pub-sub message traffic style.

Derivation Process

In our previous study of Apollo architecture, we identified a set of conceptual architecture diagrams of the system as well as sequence diagrams for two different patterns. This time, we

took a deeper look at Apollo's code in Github and gained a more comprehensive understanding of Apollo's architecture by using a combination of "Understand" and the communication graph given by our professor.

In this study, we compared the "graph visualization of pub-sub communication" given by the professor with our initial version of conceptual architecture to establish some basic understanding of the concrete architecture of Apollo. We were surprised to find that our conceptual architecture was missing some key submodules. We then immediately started to investigate the dependencies between these submodules and our original submodules and brought the source code of the system into "Understand" to see the relationships between the submodules more clearly and intuitively.

However, we encountered a new problem. The professor gave us a graph with some dependencies between submodules, and we read the source code based on these dependencies to understand which data are called in these dependencies. However, we were puzzled by the fact that some of the submodules in "Understand" did not have dependencies like in the relationship graph given by the professor. For example, in the graph visualization of pub-sub communication, the Task_manager module depends on the Routing module, but in "Understand" these two submodules are not connected by lines. At this point, we found that two submodules are not covered, which are the Common and Cyber modules, and then we introduced these two modules in "Understand". We found that all the submodules were connected to Common and Cyber. Combined with the pub-sub structure mentioned by the professor, we deduced that the submodules might communicate with each other indirectly through common and cyber. So we read the code again, and not surprisingly we found modules, some of which do not communicate with each other, but they all have code with reading and writing functions. With reference to the previous Task_manager and Routing, we speculated that the routing response probably is written to the public area by the writer in the Routing and read by the reader in the Task_manager. With this idea, we discovered more direct or indirect dependencies and updated our concept architecture.

As for the sequential diagram, because the number of modules is too large, we decided not to show how all the modules are linked and how the functions between them are called without the sequence diagram. So, after a series of discussions, we decided to show how apollo modules work together in specific situations. At this time, we chose the emergency when the sensor fails. In this case, we think the first thing to do is to find where the connection starts. We then found that the data was first collected by hardware sensors, by knowing the type of data. We directly start reading the source code and find the functions that need to be called for the whole process.

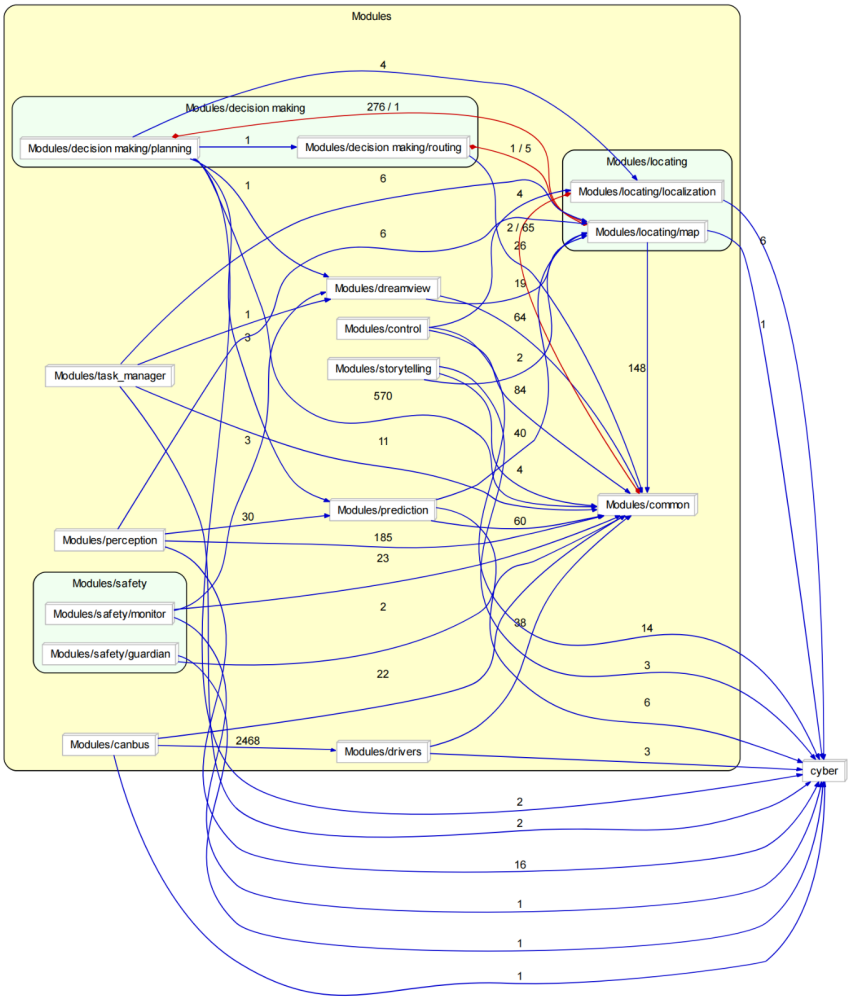

Figure 1: SciTool Understand of Apollo Open Software Platform

Modified Conceptual Architecture

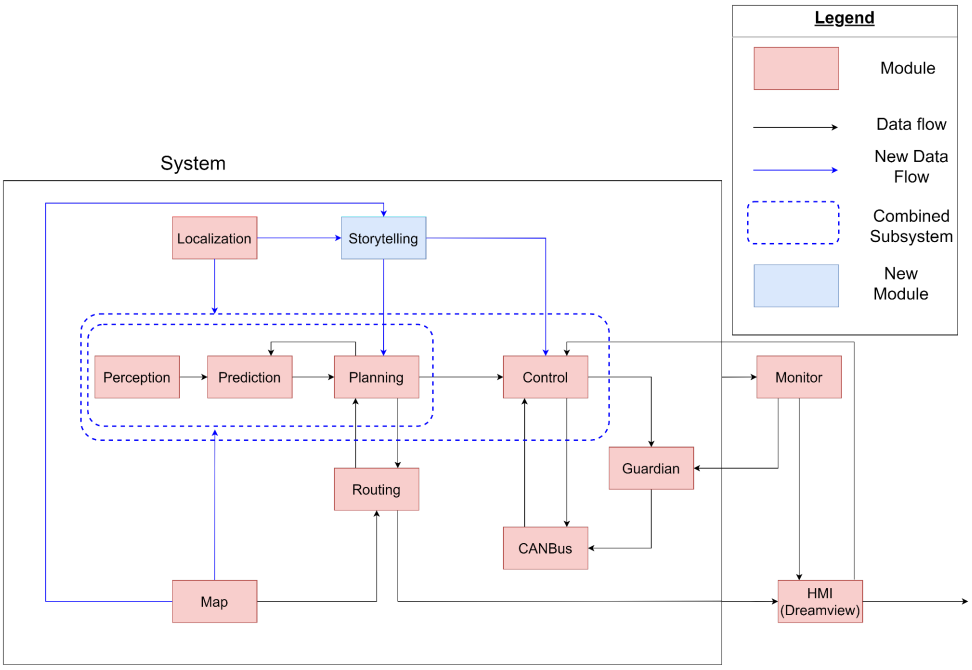

Figure 2: Modified Conceptual Architecture of Apollo Open Software Platform

We have modified our architecture style from pipe-and-filter to publish-subscribe style. We wrongly considered that the process of information transfer took place in the Apollo as a pipeline sequence. However, when we go deeper into the concrete architecture and get a notice from professors and other groups, it turns out that the whole system is more reliant on connections built between publishers and subscribers. As the operating system in Apollo, CyberRT plays an important role. CyberRT creates “channels” used for subsystems to interact, “writers” acted as publishers who published messages throughout the channel, and “readers” acted as subscribers who received dependency function calls. By taking the advantage of using the pub-sub style, autonomous driving could maintain safety since the broadcasting process of delivering messages keeps modules getting the latest updated message without waiting for response time caused by other relevant modules.

We add data transmission from the Map module to the Perception, Prediction, and Planning modules. After going deeper inside the document, we find out that the map provides its HD map for Perception, Prediction, and Planning simultaneously. Besides, the Localization module also provides accurate localization services to Perception, Prediction, Planning, and Control modules.

We added one more module: Storytelling. Storytelling is a missing component which helps coordination of other modules during driving. It will derive complex road scenarios from localization, map modules and publish them as “stories” through CyberRT to its own channel. Other subscribed modules can learn from the stories and improve the driving experience.

Concrete Architecture

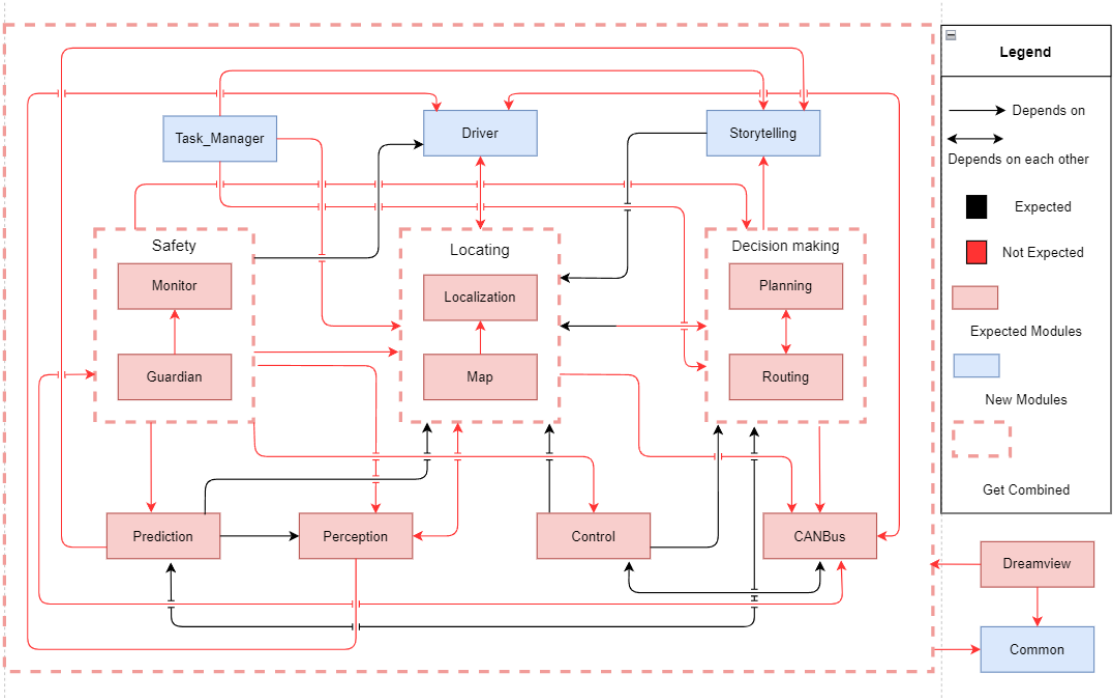

Figure 3: Concrete Architecture of Apollo Open Software Platform

Our group has conducted an analysis based on the Understand file which was provided by the professor. We stick to the pub-sub architecture style as the main concrete architecture style. We have collected some unexpected dependencies between conceptual architecture and concrete architecture. We integrate several modules together, a module called safety consists of monitor and guardian, a module called locating consists of localization and map, and a

module called decision making consists of planning and routing. As can be seen from the graph, the number of unexpected dependencies overwhelms that of expected dependencies. Below are some important reflexion analyses we have done.

Map

Map → Map:

Similar to Perception, the Map module depends on itself for some inner information, NavigationInfo. In map/relative_map/tools, there is a file called navigation_lane.cc providing navigation information, and in relative_map_component.h, the map plays the role of reader to NavigationInfo which was created on its own and sent to cyber.

Map → Canbus:

At first, we consider that map only acts as a publisher that provides APIs to modules like routing, planning, perception, and prediction. However, the message traffic graph indicates the Map needs the Chassis information from the Canbus module. We finally find out that the Map module collects information like the speed and acceleration of the vehicle in order to produce a high-precision map. Canbus stores Chassis information in Common and cyber for the map to read. The Canbus module acts as the controller of Chassis which uses ChassisDetail to share its topics in canbus_component.cc. For Map, as the reader, reads Chassis in relative_map_component.cc.

Map → localization:

Localization stores LocalizationEstimate in Common and cyber for the map to read. The localization stores its topic, LocalizationEstimate in msf_localization_component.h and relative_map_component.cc in Map, reads this topic for map.

Map → perception:

Map depends on Perception for both PerceptionObstacles and lane_boundary information.

In Hdmap_input.cc, it includes lane boundary information such as left_boundary and right_boundary directly from perception as input. For relative_map_component.cc, it uses PerceptionObstacles from common. These objects are used to build the map's structure inside the map module.

Localization

Localization → Drivers

The localization module, as a listener, relies primarily on the four parameters of the driver (GnssBestPose, PointCloud, CorrentedImu, and InsStat). In rtk_localization_component.cc, RTK stores CorrectedImu and InsStat information by reading imu_topic_ and gps_status_topic_. In ndt_localization_component.cc, lidar and odometry call PointCloud and InsStat, respectively. At the same time, the lidar in msf_Localization_component.cc also calls PointCloud. In addition, to get the best GNSS position, GnssBestPose is also called.

Localization → Localization

When RTK finishes processing the GPS message callback, the localization module takes the localization estimate and status and publishes it.

Perception

Perception → Perception

Perception submodules depend on perception/common and perception/lib;

Perception's submodules showed great dependencies on its common and lib file; most of the lidar, radar, camera data computation are implemented via lib. Also, Perception as a writer, stores information in common and cyber files. For example, in cyber/component/component.h, it was included by perception files such as radar_detection_component.h, but they are all the writers for it according to perception objects and readers for other perception objects.

Perception → localization:

Perception depends on LocalizationEstimate via common and cyber

For perception, there is a missing dependency that it has been using LocalizationEstimate for many subsystems. For example, in files such as motion_service.h and radar_detection_component.h, they both use LocalizationEstimate from Localization. The perception uses LocalizationEstimate to estimate if the vehicle has enough speed.

Perception → Drivers: Perception depends on Drivers as it contains basic hardware input for perception: lidar, radar, and camera. In conti_radar_canbus_component and ompensator_component(under lidar) are both included in adapter_gflag.h as a topic and subscribed by perception.

Prediction

Prediction → storytelling:

File prediction_component.cc creates reader for <storytelling::Stories>

Storytelling as a new module we did not include in Assignment 1, it was a role to build up a story for all those modules that require a scenario. In prediction, as we mentioned in the previous report, it contains a scenario submodule. In understanding dependency diagrams, storytelling stores its topic: <storytelling::Stories> inside common and prediction access that topic as a reader via common.

Planning

Planning → Canbus

File *planning_component.h* depends on *modules/canbus/proto/chassis.pb.h*

"chassis.pb.h" file export proto message in order to inform other modules about chassis information. "planning_component.h" is the head file for the Planning module. Inside the head file, chassis information is imported as one of the parameters with information from

other modules like prediction and localization. The chassis information is also called in *planning_component.cc*. The function “bool PlanningComponent::Proc()” uses chassis information and makes analysis on the speed, acceleration, and heading angle to make decisions on planning future routing and turn it into an ADCTrajectory format file.

```
bool Proc(const std::shared_ptr<prediction::PredictionObstacles>&
              prediction_obstacles,
          const std::shared_ptr<canbus::Chassis>& chassis,
          const std::shared_ptr<localization::LocalizationEstimate>&
              localization_estimate) override;
```

Control

The controller module is very similar to our previous result in conceptual architecture. It takes the input of ADCtrajectory, car status(common), localization in the beginning phase of the module through nodes in Cyber created by the control module. Then the data will be sent to be preprocessed and processed by the controllers like what we discussed in the previous project report. However, After reading the source code we find out the control highly depends on the Common submodule; this is due to the fact that multiple controllers are used in this module to maximize the performance of the commands. All of them are highly related to math functions, the control module also modified some of the functions in the common sub-system and saved them in a common file in the Control module.

Monitor

Monitor → Localization:

File localization_monitor.cc receives data about localization status stored in a proto file in the localization module. Also file channel_monitor.cc receives localization estimation. Both parts will be aggregated in the summary_monitor and provide system status as output.

Monitor → {Map:MapMsg, Prediction: prediction obstacles, Perception: perception obstacles, Planning: ADCTrajectory, Control: control command, Canbus:chassis}:

The monitor module’s dependency on the above five modules are subscribed in channel_monitor. The five modules assign their output in common as topics and the channel_monitor creates a reader to subscribe on the topics to accomplish dependency on these modules. An example of the reader receive ADCTrajectory from the planning module: {FLAGS_planning_trajectory_topic, &CreateReaderAndLatestsMessage<planning::ADCTrajectory>}

Monitor → Driver

Files in the monitor module that depend on the driver module can be separated into two parts. Channel monitor and camera monitor from software side read conti_radar, point cloud and sensor image from the driver module. Gps_monitor from hardware reads GNSSBestPose from the driver module stored in gnss’s proto file.

Canbus

Canbus → Driver

There's part of the module implemented in the driver module. Can_client and Can_common. Can_client is an initialized driver for canbus. Canbus needs to initialize a can_client to operate. Can_common is the main function to send and receive data from other modules.

Canbus → Guardian

File component.cc receives data about commands from the Guardian module.

Inside the file, it creates a reader that can read the shared data from the Guardian. The guardian command is a topic in common. The same structure will be used when Canbus is receiving commands from the Control module.

Canbus → Canbus

Canbus needs to initialize Can_client and vehicle factory in the file component.cc. This part will depend on the common and vehicle part of Canbus. For the vehicle factory it means when the Apollo system is in different cars, it needs a specialized controller for the specific car, and each time Canbus will need to initialize a controller for different car models.

Driver

Driver → Canbus

Driver module depends on the chassis information in Canbus. Hardware data stored in Canbus/Vehicle/gem are sent to the protocol_data.h file in the driver. These data contain brakes, steering etc.

Driver → Localization

In the driver module, conti_radar in radar depends on the localization module. In the file conti_radar_canbus_component.cc, a call to LocalizationEstimate is made by running the function PoseCallback in the conti_radar file.

Driver → Driver

In the driver module, PointCloud is called by the parser in Lidar to process the raw PointCloud. In parser.cc, the file resets and publishes the raw PointCloud.

Task manager

Task_manager → Routing

Task_manager depends on routing, the routing response is defined by common as "routing_request_topic"and "routing_response_topic". Then, they are created by the routing module using the CreateWriter function. Once it was created, the routing response can be received by Task_manager.component.cc by the CreateReader function. It can conduct routing modes after getting routing requests from routing modules.

Task_manager → Map

Task_manager submodule depends on map because it carries out some mode of routing which requires the information from map. For example, the task manager includes modules/map/hdmap/hdmap_common.h in order to use the parking space information. When

conducting parking routing mode, it has to check the parking space information with map module.

Task_manager → Localization

Task_manager depends on localization through some indirect communications. As we mentioned in above, localization stores LocalizationEstimate in a public space via Common and cyber. The task_manager_component.cc receives localizationEstimate by localization_reader_ = node_->CreateReader<LocalizationEstimate>. Since task_manager conducted a series of routing modes. It needs localization information when calling a mode.

Task_manager → Dreamview

Task_manager depends on Dreamview by asking for the map service from Dreamview in cycle_routing_manager.h

Summary

Overall, we found some interesting findings after the reflexion analysis. Most of the submodules publish their output as a topic into the common module, whenever a module depends on another module they will create a reader to subscribe on the corresponding topic. The common module is a module that takes charge of the topics communicated through different modules. A very wide use of common is adapters. It is widely used in almost every module. The function of the adapter is to allow every module to assign a topic to common/adapters. Once a topic is assigned, it can be used in different modules, which is similar to global variables in Python. For example, when the canbus module is receiving commands from the control module, the reader reads topic control_cmd which is assigned in adapters. This is one big reason why there are many dependencies added or removed. The dependencies shown on the graph in Understand only show the relations between the module and common, so we need to dive deep into the code level to find out the actual architecture. Besides, the actual implementation from conceptual to concrete is more complex than the conceptual architecture. Some of the implementations of the module may not be able to complete on their own, therefore Apollo has some new modules that could help the implementation process. Our newly discovered module Task Manager is a good example. Last but not least, during our investigation on the conceptual architecture, in order to make the progress more efficient we tried to study the description text inside the Apollo file. However, the Apollo system has gone through seven versions of updates but most of the description text still remains in the old version. Therefore the information we got is a lot behind than the current version.

Second level Subsystem Analysis

Perception is considered one of the most important subsystems in the Apollo system. It is used for recognition, which is like the eyes of the autonomous driving system. The overall structure can be divided into 4 subsystems: public modules, sensor modules, deep learning modules, and fusion modules.

Public modules include common directory and helper functions for perception. It has some files to store and transmit data. The files in this category are: common, lib, production, proto, tool, data, testdata, base, onboard, and map. Sensor’s modules: the three sensors’ files are lidar, radar, and camera. These files are used to handle the corresponding hardware’s input

and process them into binary data and computed results. Deep learning modules: how do sensor files compute those inputs into results? Deep learning is one of the most useful tools to compute. The files include inference and lib. Fusion module: the fusion file as the last of four sections, it fuses all results from previous sensor modules and sends those outside the perception.

Some detailed explanations of each module are listed as follows: **Lidar** is used to implement three main functionalities: lidar_obstacle_detection, lidar_obstacle_segmentation, lidar_obstacle_tracking.[3] **Camera** is used to detect traffic lights, obstacles and is the place where transformation and calibration happened.[4] **Radar** is used to implement the tracker.[5] **Onboard** defines submodules to handle information from sensors including pre-processing, object recognition, region of interest filtering, and tracking. There is a file under Onboard/component called libperception_component_lidar which was compiled by fusion_component.cc, lidar_output_component.cc, radar_detection_component.cc, recognition_component.cc, segmetation_component.cc and detection_component.cc, run it means running everything above. **Tool** represents offline testing tools. **Fusion** fuses the results of sensors. **Inference** consists of deep learning modules and implemented by tensorflow, caffle, torch **Map** receives the input from hdMap and is used for boundary and obstacle recognition. **Common** is the public directory, including some algorithms and functions for other submodules. **Lib** stores some basic libraries including time, threads. **Base** is the base of perception, including some hardware inputs and computations, most of the other submodules depend on this module. **Production** acts as the entrance containing a deep learning module and uses cyber to start. **Proto** means protobuf which is used for storing and transmitting. **Data** stores parameters. **Testdata** contains testing data of above modules.[6]

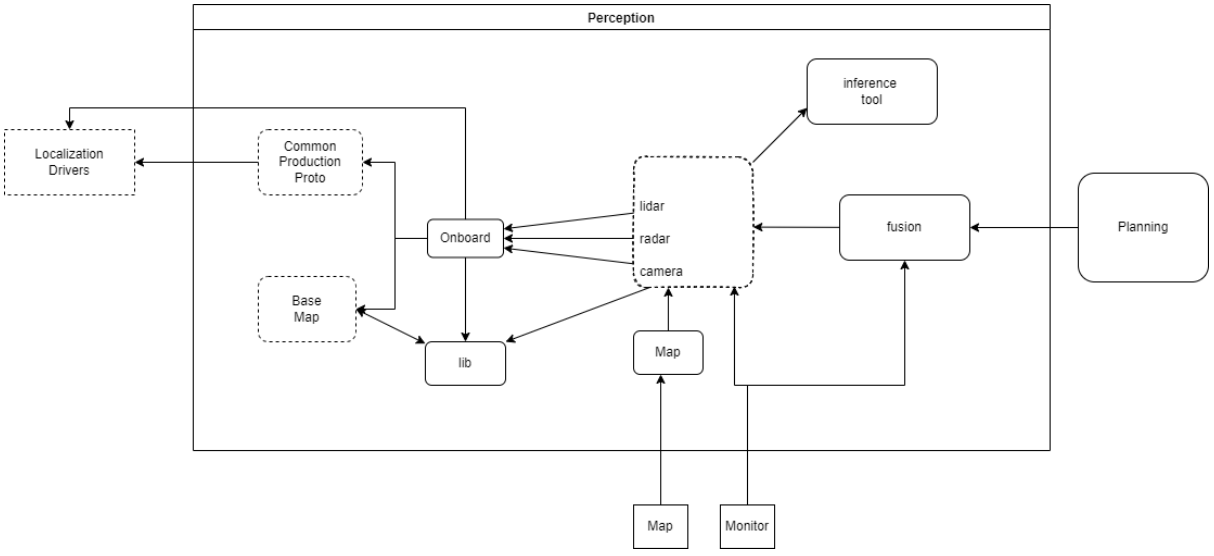

Figure 4: Concrete architecture of 2nd level of Perception module

Reflexion Analysis on 2nd level

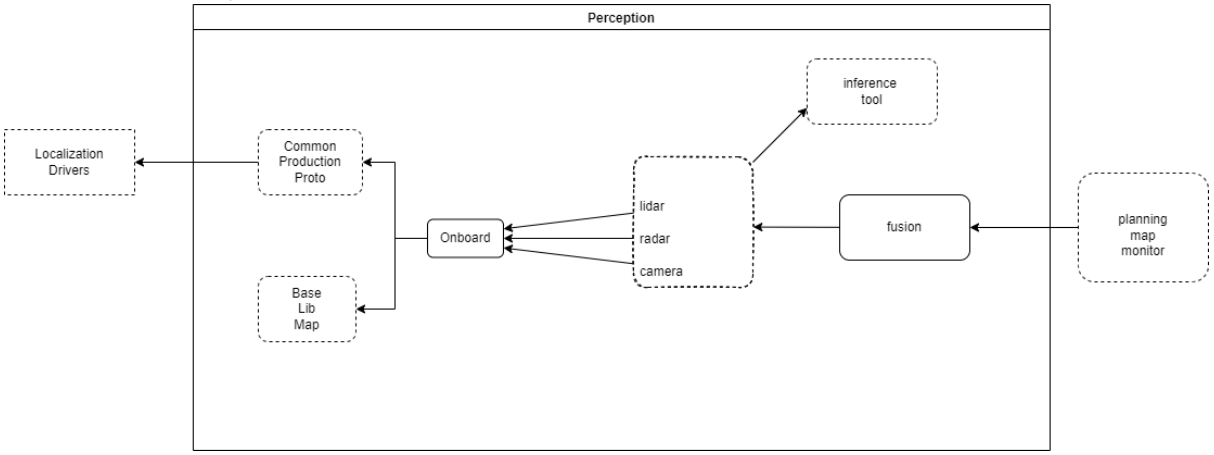

Figure 5: Conceptual architecture of 2nd level of Perception module

In our conceptual architecture, the “common” acts as the public directory holding the information from outside modules and sending them onboard to start the lidar, radar, and camera modules. The three sensors’ modules depend on “lib” and “inference” and the results go to fusion and a fused result is sent to outside modules.

The concrete architecture is described as follows: onboard is the entrance of submodules including fusion, lidar, radar, and camera. It plays a role of a door of perception, the information from outside modules goes in and out with this module. It defines and compiles components and submodules into libperception_component_lidar, which runs sensor submodules when called. The onboard module depends on public modules including common, lib, map, base, production, and proto. Except for proto, which is for storage and transmission, the remaining are the input for onboard or helper algorithms which would be used later on. The sensors’ modules, lidar, radar, and camera are all dependent on onboard, they are started by onboard and access information inside onboard and use onboard as transmission as well. The three modules depend on tools and inference which have neural network models for them to compute the results. The results from them go to fusion which fuses the results into a perception obstacle topic for other outside modules to use. Meanwhile, some of the camera's data such as the lane boundary go directly to other outside modules.

The discrepancies of conceptual and concrete architecture are that we supposed the data goes linearly inside the conceptual architecture which means that the data goes into the common and common send data to each submodule for computation. Finally, the data was sent out via fusion. However, in concrete architecture, data enters the perception in both common module and onboard module, the reason why this happens is that the apollo has a pub-sub style, the inner module can easily access the topic inside linearly and goes into perception from production and through all submodules. Also, the output data is not only exiting perception via fusion only but also from the camera, lidar directly into the prediction and monitor. This is because outputs have different types instead of all being the topic of perception obstacles.

Sequence Diagrams

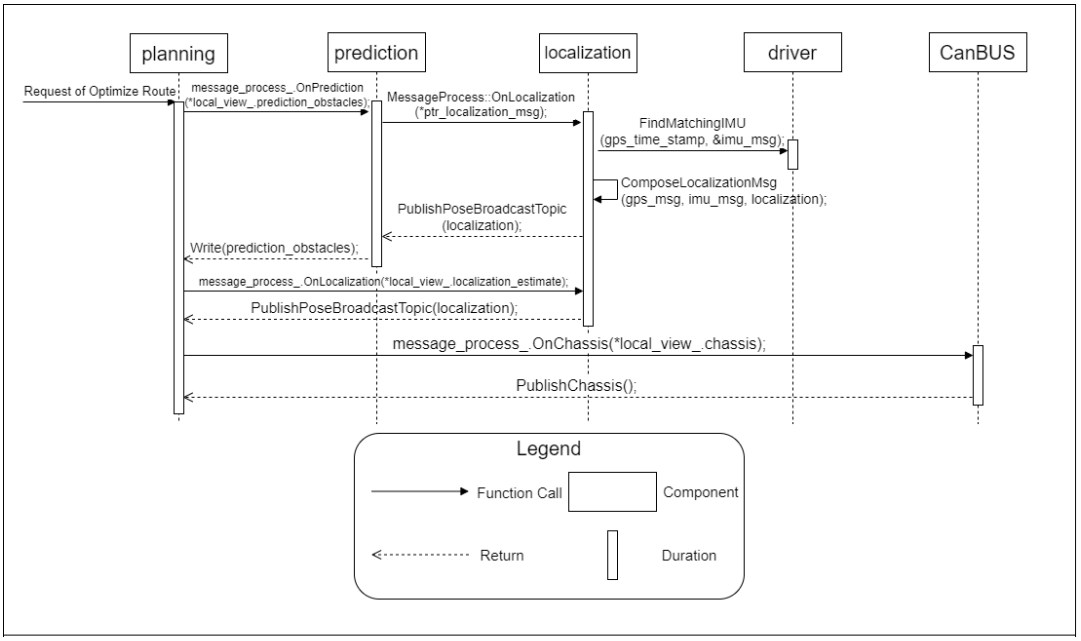

Figure 6: Sequence diagram illustrating the process of optimizing route locally

After the planning module obtains the navigation route from the Routing module, the planning module needs to optimize the navigation route locally. To do this, the planning module calls the data of the other three modules (Prediction, Localization, and CanBUS). Planning retrieves prediction_obstacle information from OnPredication which is in prediction. Prediction receives the request from planning and requests localization to provide the LocalizationEstimate. In localization, it uses the timestamp of GPS to find IMU information. Then LocalizationEstimate is generated by the localization module combining GPS and IMU information. Localization sends LocalizationEstimate via PublishPoseBroadcastTopic to prediction. After analyzing and calculating the new LocalizationEstimate, the prediction will provide the prediction_obstacle information to planning through the Write function. At the same time, localization also sends LocalizationEstimate for planning to help optimize the route. In addition, chassis information in the CanBUS module is also essential. Planning makes a request to CanBUS through the function OnChassis. CanBUS integrates the information of vehicle speed, acceleration, and heading angle and returns it to planning through PublishChassis.

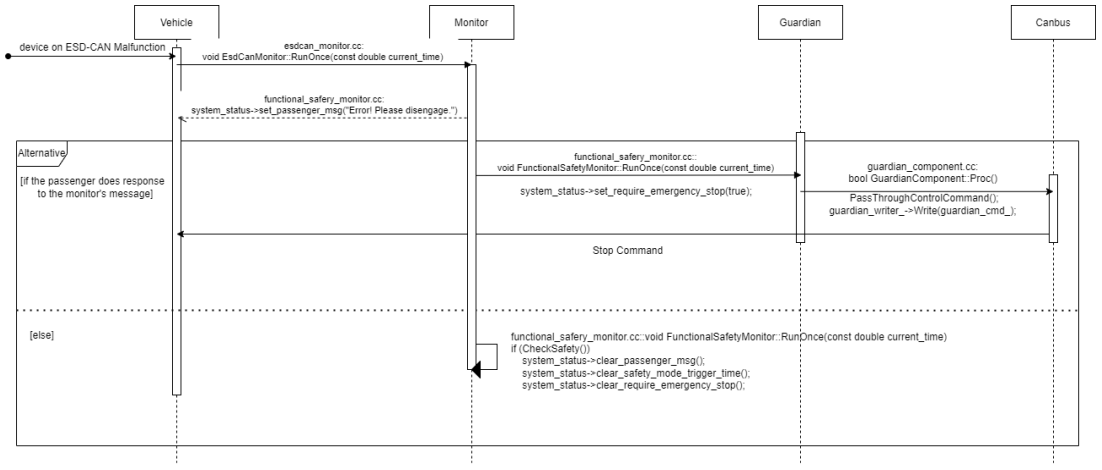

Figure 7: Sequence diagram illustrating the emergency when malfunction happens

This diagram indicates what happens in the emergency during the auto-driving mode. The situation begins with a malfunction in the device on ESD-CAN. Once the malfunction happens, the error is detected by the RunOnce from esd_monitor.cc then the monitor will send the message of the error to the passenger in functional_safety_monitor.cc. Two circumstances might happen afterward. If the passenger does not respond to the monitor's message the sequence will go like the one above. The monitor will call set_require_emergency in functional_safety_monitor which alerts the guardian module by modifying the system status. Once the guardian module receives the alert by the proc() in guardian_conponent.cc, it will bypass the control's commands and write the stop command to the canbus to halt the vehicle for safety. If the passenger starts to control the vehicle, the monitor will clear the emergent situation.

Limitation and Lesson Learned

During the entire process, we find that the following limitation existed in our research:

- Understand 5.1 is a professional and efficient software, but it comes with high learnability and it crashes often. The time we spent on Understand is much longer than we expected.
- The Apollo system includes lots of proto files. They are important because they store a large amount of data. However, we cannot see any dependency relationship about proto files. we can only get information about it from the source code or the file itself.

In the process of concrete architecture, we learned to use Understand to analyze the dependency relationship between each module, the specific way of data transmission and draw dependency graphs. At the same time, we learned to find connections between modules by analyzing the details in the files. In addition, we learned a lot of skills from the code about how to write C++ code concisely.

Conclusion

In conclusion, Apollo's concrete architecture follows the pub-sub style that we derived from the previous project report. Through analysis of the source code, we develop a deeper understanding of the data publish and subscription system taken place in Apollo. A more comprehensive structure inside the Perception module is also discovered. We find that the common module provides universal functions to all of the modules. Modules could communicate in the adapter inside the common module. By doing reflexion analysis on the 2nd level perception module, the data transmitted into the module could be well distributed into different components linearly. In addition, two use cases have been depicted, including malfunction emergencies and the optimization of local routes.

Glossary

Understand: a software that developed by SciTools which helps developers understand the relationship and function calls between source code.

Reflexion Analysis: a process that helps to understand a big system architecture. We are focusing on investigating why the unexpected dependencies are added or removed and the rationale behind them.

Dependency: the state of reliance on others, in this project, is modules/subsystems.

ESD-CAN: Active CAN interface Board for PCLexpress with esd CAN ip-code, CAN-FD capable according to ISO 11898-1:2:2015

Reference

[2]xiaoxq. (2019, November 16). Apollo Cyber RT Terminologies. Retrieved March 18, 2022, from https://github.com/ApolloAuto/apollo/blob/master/docs/cyber/CyberRT_Terms.md

[3]daohu527. (2020, September 18). Lidar. Retrieved March 18, 2022, from https://github.com/daohu527/dig-into-apollo/tree/r7.0.0/modules/perception/lidar#lidar_module

[4]daohu527. (2020, September 18). camera. Retrieved March 18, 2022, from https://github.com/daohu527/dig-into-apollo/tree/r7.0.0/modules/perception/camera#camera_module

[5]daohu327. (2020, September 18). radar. Retrieved March 18, 2022, from https://github.com/daohu527/dig-into-apollo/tree/r7.0.0/modules/perception/radar#radar_module

[6]daohu527. (2020, September 18). Dig into Apollo - Perception. Retrieved March 18, 2022, from https://github.com/daohu527/dig-into-apollo/tree/r7.0.0/modules/perception