CISC 322

Assignment 2

Concrete Architecture of Apollo

March 14, 2022

Runfeng Qian (Group Leader)	18rq10@queensu.ca
Wonton Zhou	19bz6@queensu.ca
Ziheng Yang	18zy59@queensu.ca
Runze Lin	18rl30@queensu.ca
Junyu Yan	19jy28@queensu.ca
Haoyun Yang	18hy58@queensu.ca

Abstract

This report goes through a detailed explanation of the concrete architecture of the Apollo open-source autonomous driving program, according to our team's close-examination of its source codes and architecture in the sci-tool understanding software; it also includes outlines and derivation process to the concrete architecture of Apollo. The most crucial part of our report discusses and analyzes the concrete architecture, and the discrepancies between our initial conceptual architecture and concrete architecture. Our mission is to develop a concrete architecture and learn valuable lessons throughout the discovery process. This report also contains two sequence diagrams of use cases, lesson learned and final conclusion.

Introduction

The goal of this report is to determine the concrete architecture of the open-source Apollo autonomous driving platform by analyzing the source codes within it. We approached this goal based on our optimized conceptual architecture and interpretation of the source codes of the Apollo autonomous driving platform.

In the process of writing our conceptual architecture, we analyzed the ReadMe files in each module and subsystems stored in Apollo's Github and figured out the basic relationships between modules since it helps us to build a better understanding of its major parts and to predict their brief connections. While we tackled the concrete architecture, we analyzed the source code in depth to examine the exact relationships between each function and each module, and their function calls. In addition to the conceptual architecture, we added three more subsystems in the concrete architecture, which played vital roles in the Apollo system.

During our analysis of the derivation process, concrete architecture proposal and discrepancy analysis, we acquired more detail and information about the concrete architecture. We described our newfound dependencies and new subsystems in the reflection and analysis section. In addition, we also analyzed the control subsystem in detail both conceptual and concrete. Lastly, Two sequence diagrams were presented to demonstrate our two use cases using the concrete architecture.

Conceptual Architecture Recap

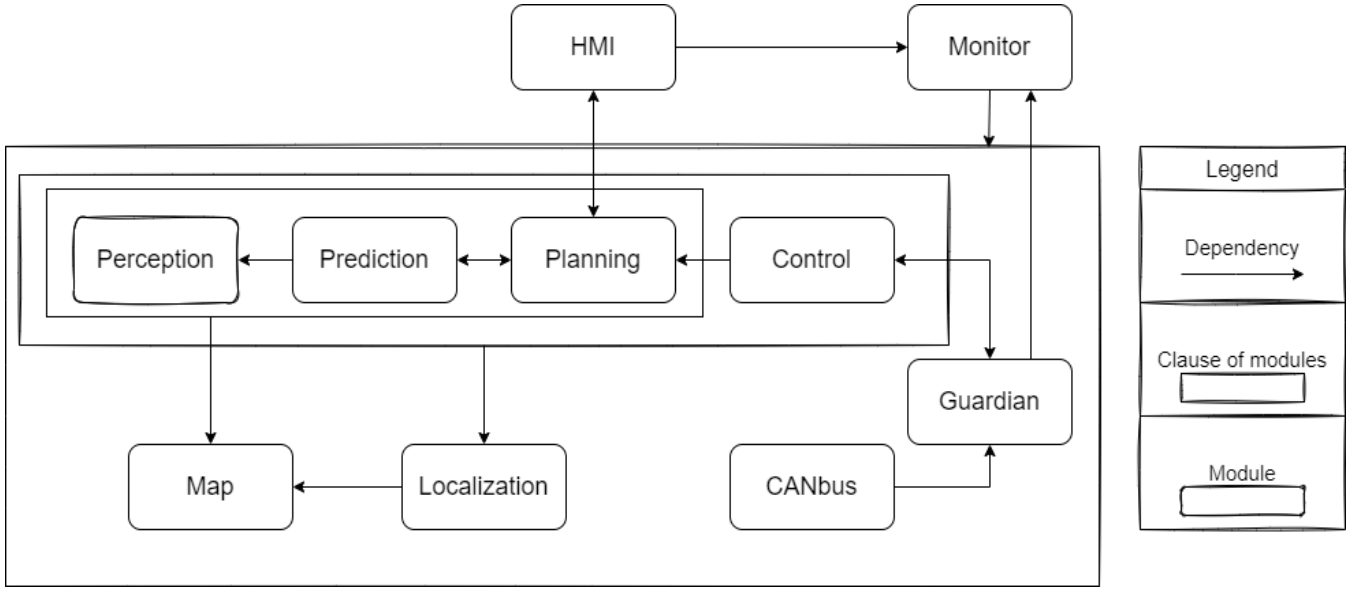


figure 1: top-level conceptual architecture

Map Engine sends map information to the localization component. The localization component obtains the precise location on the map and sends them to other modules. The perception module collects environment information from sensors and receives the information from the localization component. The Perception module sends the 3D obstacle tracks to the prediction component. After combining the outputs from Map, Localization and Perception module, the prediction module will calculate and output information of obstacles annotated with predicted trajectories and their

priorities to the planning module. After obtaining information from previous modules, the planning module generates a feasible trajectory and sends it to the control component. Depending on the planned trajectory, the control module sends the control command to canbus[1]. HMI provides a human-machine interaction interface, visualizes information from all other modules for easy manipulation. Control and guardian modules interact with each other. Monitor module receives data from all other modules.[2]

Subsystems

Planning: The planning module is responsible for planning a feasible spatiotemporal trajectory for the autonomous vehicle based on the perception module and prediction module results.
Perception: The main function of the perception component is detection and classification, since the autonomous driving car needs to know the location of the obstacles and then classify them.
Prediction: Apollo's prediction module receives obstacle information from the perception module, and the autonomous driving system must be able to predict what may happen in the scene for a short period in the future to make the correct movement as we do.
Map: The Map component contains all the map data and provides the corresponding map data function interfaces.
HMI: HMI is the human-machine interface that allows the users to interact with the Apollo autonomous driving system. It is also called Dreamview.
Control: The function of the Apollo control module is to calculate the accelerator, brake, and steering wheel signals of the car according to the trajectory generated by the planning (planning module), and to control the car to drive according to the specified trajectory.
Localization: The localization system is a comprehensive positioning solution with centimeter-level accuracy based on GPS, IMU, HD map, and various sensor inputs.
CANbus: CANbus accepts and executes control module commands and collects the car's chassis status as feedback to control.
Guardian: Guardian monitors the system status of autonomous driving, if it receives an emergency signal and the signal shows the sign of failure, the guardian will take over the control module and emergency braking the car.
Monitor: The monitor module contains system-level software such as code to check hardware status and monitor system health. (This was merged into the new utilities subsystem)

New Subsystems

Utilities(new): While looking deep into the source codes of Apollo, we discovered a folder named common, which contains many dependencies from other subsystems. Hence, we added a new subsystem that includes this "common" folder as a new subsystem — utilities. It has many utility functions like adapters, filters, latency recorders. We also put drivers, monitors, storytelling and task managers into utilities. The folder drivers provide all required drivers for CANbus, radar, camera, GNSS etc. The folder monitor mainly monitors the hardware and software status, displays the cause of the failure when it occurs, and outputs the status to the guardian module for emergency treatment. Storytelling is a global and high-level Scenario Manager to help coordinate cross-module actions. Task Manager monitors and provides information about the performance of other modules. We put drivers, monitor, storytelling and task manager into utilities because all these modules provide support and utility functions for other top-level subsystems. They all essentially serve and help with those core modules (Perception, prediction, planning etc.).

Planning(modified): We merged the routing module into the planning module. The role of the original planning module is to plan a trajectory that the vehicle can travel based on the results of the sensory prediction, current vehicle information and road conditions. The trajectory of the planning module is the short-term trajectory, that is, the trajectory of the vehicle driving in the short term, and the long-term trajectory is the navigation trajectory planned by the routing module, that is, the trajectory from the starting point to the destination. The planning module will first receive the navigation trajectory, and then according to the navigation trajectory and the condition of the road, it will drive along the short-term trajectory until the destination. Since both modules (planning and routing) essentially perform the function of trajectory planning (in long-term and short-term), we have good reasons to combine them into one module.

Derivation Process

We began by mapping the source code folders to subsystems based on our previous conceptual architecture. Then we found that our previous conceptual architecture wasn't complete, and we spent some time fixing them. We saw many unexpected modules and dependencies. Some subsystems have too many dependencies from others, while others almost have no dependence on other subsystems. We found this not normal, and we investigated further into the source codes then merged modules with similar functions.

Apollo is a large open-source platform that contains a significant number of source codes and most of the codes were written in c++ languages and hard to interpret. This makes examining the entire Apollo source codes impracticable since it would be too time-consuming and require lots of preparation. Therefore, we learned to solve the construction of Apollo's concrete architecture by a top-down approach. We figured out what the top-level modules and subsystems are by reading each module's readMe file and briefly examined the source code of specific modules if the readMe did not provide enough information for us to tell their connections between other modules. After that, we use the sci tool to build our own ideal concrete architecture and also add the second level subsystems and modules to make the architecture more reasonable and complete. Some of the second-level modules are known to be branches of top-level modules since they share the same or similar functionalities.

After a complete view of the Apollo program in Understand and the source codes of the systems, we derived a logical, concrete architecture and had a better understanding of the source codes and the overall structure of the Apollo open-source autonomous driving platform. We summarized 10 subsystems and established the concrete architecture based on them. Since the Understand tool is unable to help recover the pub-sub message traffic, we also referred to the pub-sub relation graph provided by the professor. With the help from the understand tool and the graph visualization of pub-sub message traffic, we completed our diagram for the concrete architecture. Then studied the discrepancies using Reflexion analysis. We tried our best to find out details of different dependencies by examining the source code.

By looking at source codes and documentations, we were also able to derive that some subsystem modules use the pipe and filter architecture style.

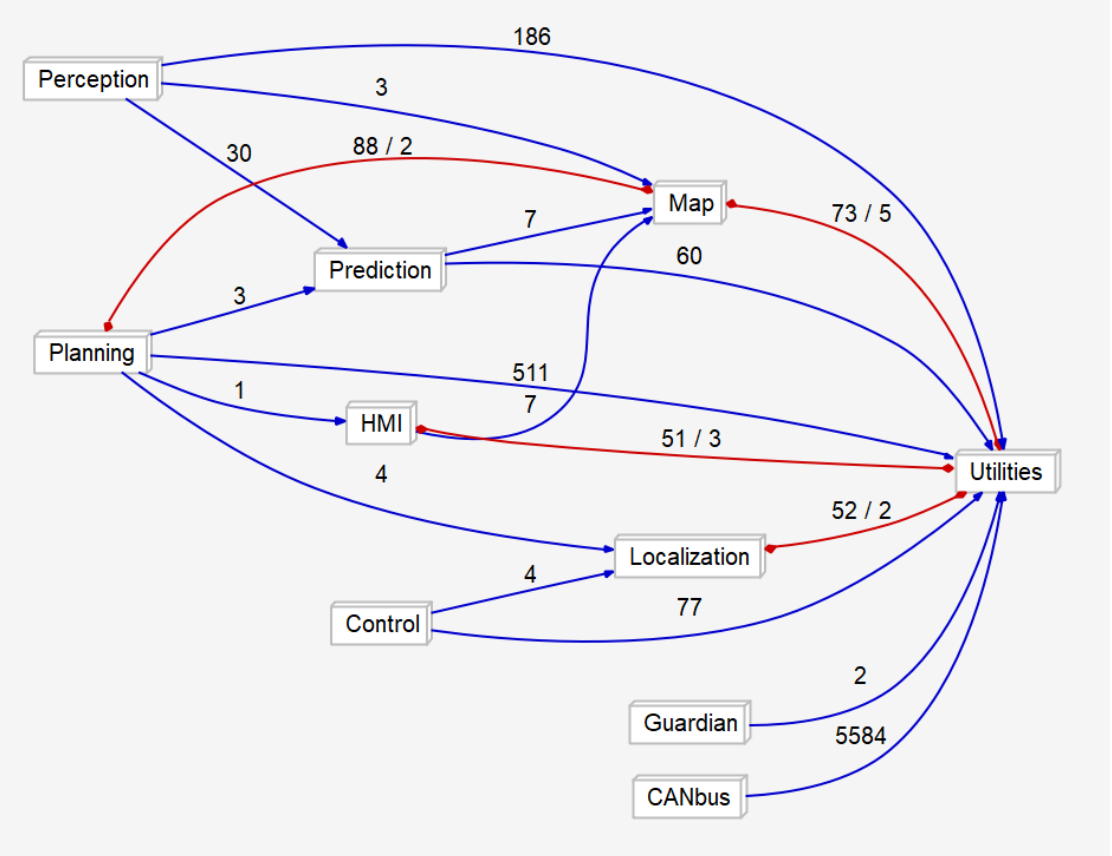


Figure 2: Understand Graph

Concrete Architecture

We obtain the architecture shown below (figure 3) through the derivation process. The concrete architecture maintained the pub-sub architecture style and the pipe and filter architecture style. Cyber module (not shown in the architecture) is a messaging middleware of the pub-sub architecture style. We removed the cyber module from our architecture and kept the dependency relations based on it. A lot of unexpected dependencies appear compared to our previous conceptual architecture. The most significant change of the concrete architecture is the addition of the utilities module. We found that almost all modules can not work alone without help from the utilities module. The utilities module provides many utility functions for other modules to use. It has a ‘common’ submodule with adapters, latency recorder, math support etc. It also has a storytelling submodule to help coordinate cross-module actions, a monitor submodule to monitor the hardware and software status, a drivers submodule to provide corresponding drivers for each hardware, and a task manager submodule to supervise the operation of the entire program.[3]

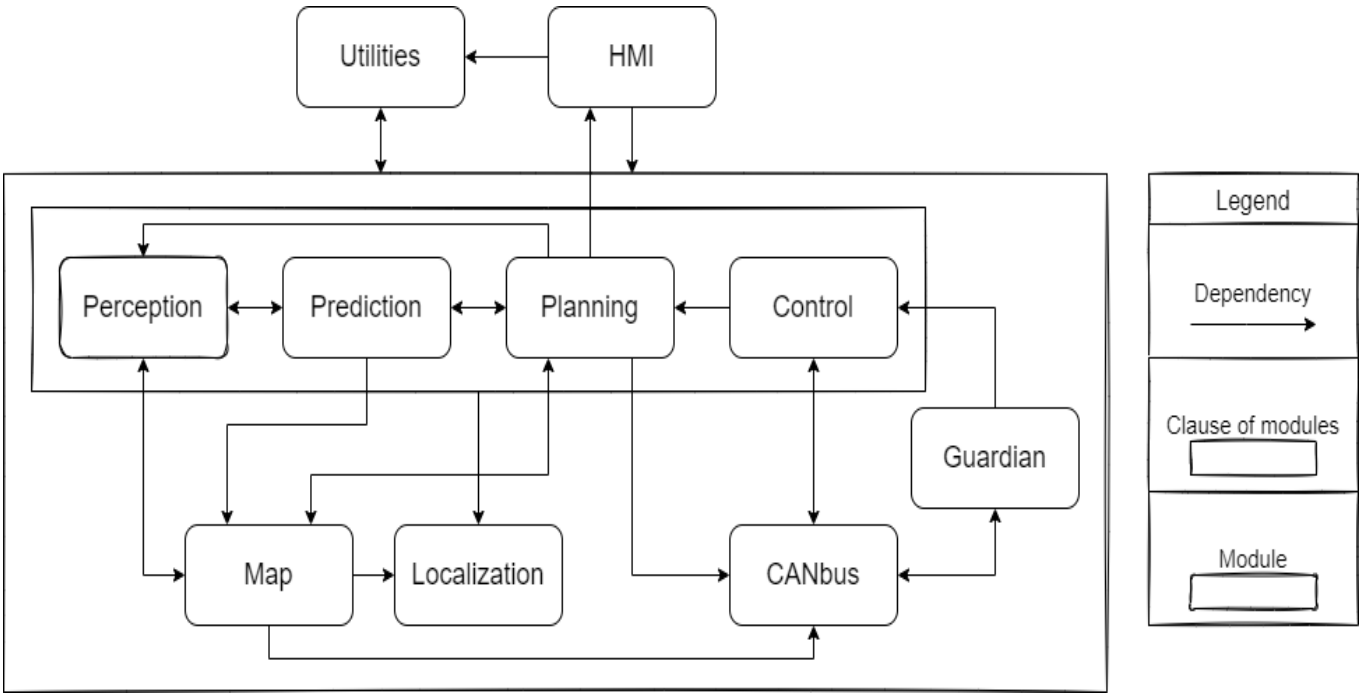


Figure 3: top-level concrete architecture

Second-Level Example

Conceptual Architecture of Control module

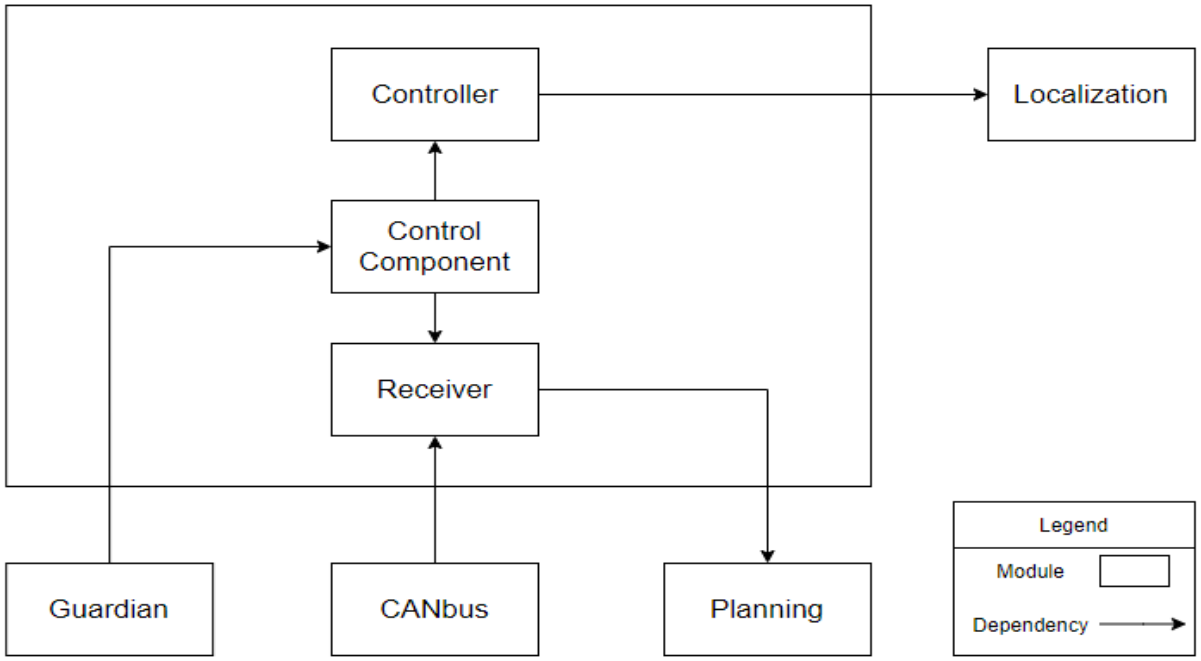


Figure 4: the conceptual architecture of the control module

The conceptual architecture of the control module follows the same style of the Apollo architecture, it sends the message to a pub-sub manager, and one module in control will subscribe the data and process them.

Receiver: the receiver submodule is an information transfer component. It receives the planned trajectory information from the planning module. It can also send the control commands to the Canbus module.

Controller: the controller submodule has all kinds of controller algorithms. It also depends on the localization module and receives a localization estimate from the localization module.

Control Component: the control component submodule is the core of the control module. It first gets the received trajectory information from the receiver module. Then it uses the localization estimate and control algorithm from the controller submodule. After combining all the information, it generates some control commands and gives them to the receiver/sender module, then the receiver module will send the control commands to the Canbus module. The control component submodule will also send vehicle status to the guardian module for it to monitor the status of the autopilot system so that the guardian module will actively cut off the control command output and apply the brakes when there is a module failure status.

Alternative Conceptual Architecture of Control module

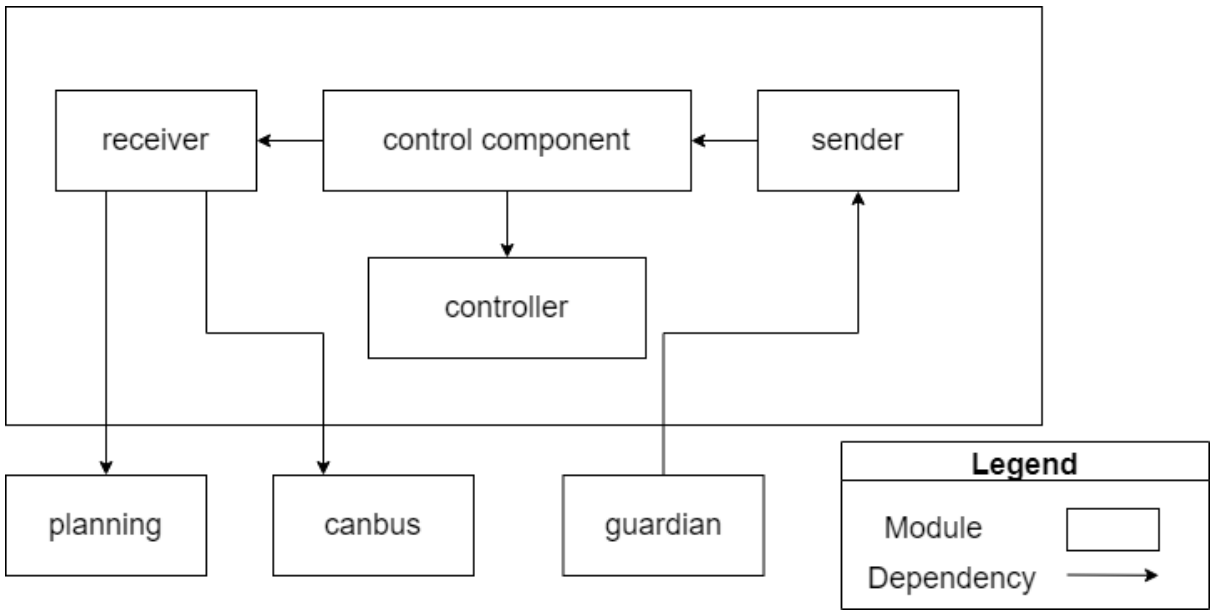


Figure 5: the alternative conceptual architecture of the control module

The alternative conceptual architecture of the control module has four submodules: receiver, control component, controller and sender. It has a pipe and filter architecture style. The receiver module receives and pre-processes the planned trajectory and chassis information from the planning and canbus module. The control component module gets the data from the receiver module. Then it will call different controller algorithms in the controller module. After combining data and the controller algorithm, the control component module will generate a sequence of control commands. Finally, the sender module will get the generated control commands from the control component and send them to the guardian module.

Concrete Architecture of Control module

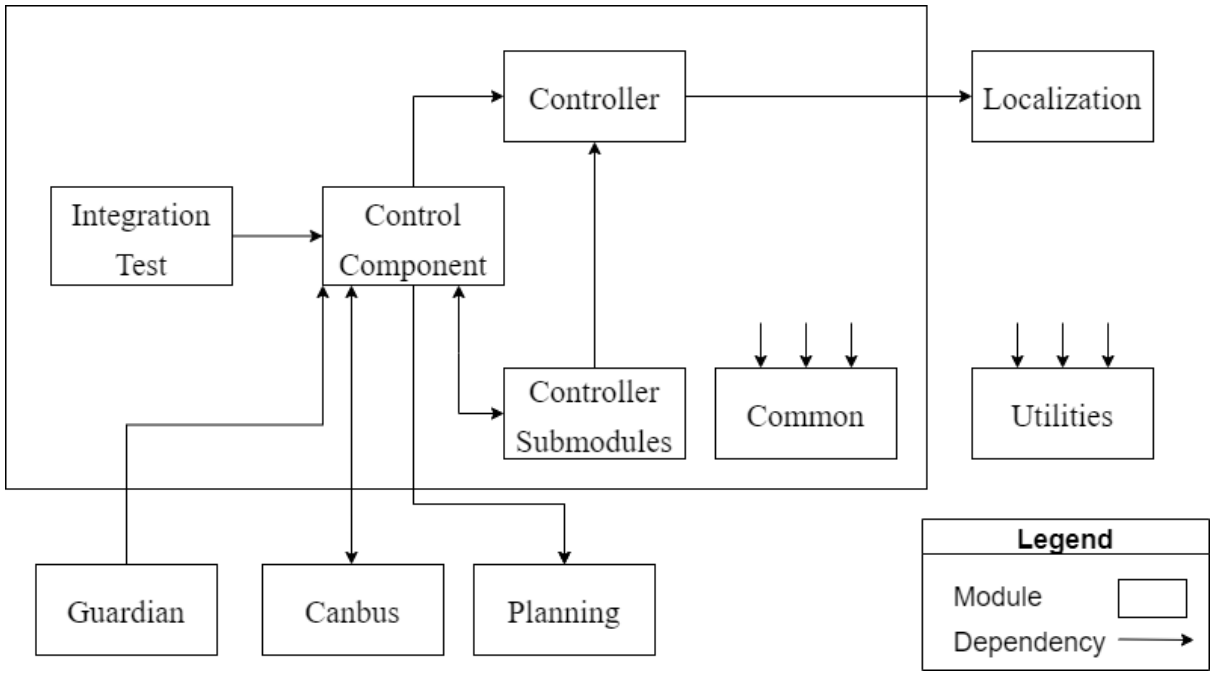


Figure 6: the concrete architecture of the control module

Integration Test: the integration test submodule depends on the control component module; It tests the validity and integrity of other submodules before operation.

Controller: the controller submodule has all kinds of control algorithms. It contains algorithms for MPCcontroller, LATcontroller and Loncontroller. It also has a 'controller agent', which is an agent that other controllers register to, thus enabling access to multiple controllers. It also depends on the localization module and receives a localization estimate from the localization module.

Controller submodules: It is a composite module that contains many smaller 3rd-level modules. It receives planned trajectory and chassis information from the control component. It then pre-processes the received data. It can also use the controller algorithms from the controller submodule and generate a sequence of control commands. After post-processing the control commands, it will send them back to the control component.

Control Component: the control component submodule is the core of the control module. It is the entry point for the control module, and it also manages the operation of the entire control module. It receives planned trajectory and chassis information from the planning and CANbus modules. Then it sends the data to 'Controller submodules' for pre-processing. It can also send the control command from 'Controller submodules' to the Canbus module. Additionally, the guardian module may also get the vehicle status information from it to monitor the status of the autopilot system so that the guardian module will actively cut off the control command output and apply the brakes in the event of a module failure.

Common: the 'common' module provides many utility functions that all other submodules can call in the control module.

Reflexion analysis - High level

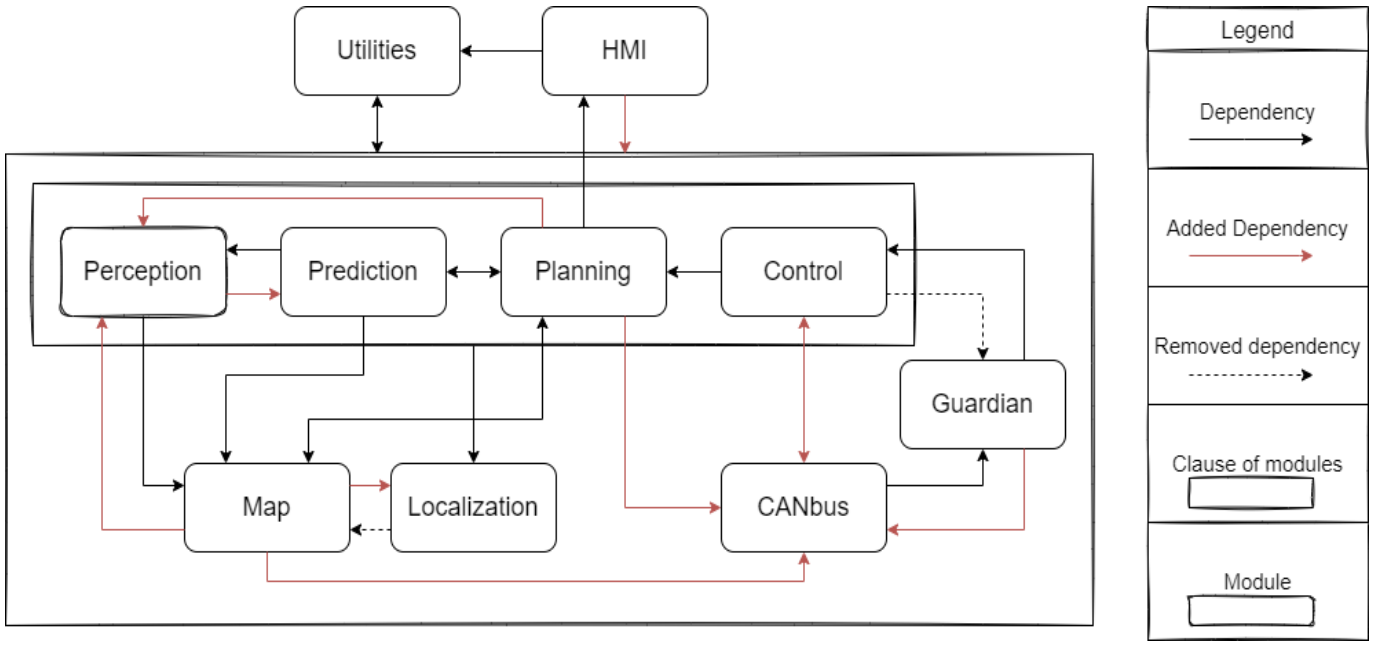


Figure 7: changes

Unexpected dependencies

Dependencies from the Utility subsystem are not included. since many of the modules are not present in the conceptual architecture

Perception → Prediction: This dependency is for adding obstacle priority to the obstacle history map. This process needs an evaluator manager to achieve while the evaluator manager is a part of the prediction module. We found this connection in the file apollo/modules/prediction/evaluator/evaluator_manager.cc.

Planning → Perception: the planning module has an additional dependency on the perception module. It receives traffic light detection information from the perception module.

Planning → CANbus: In concrete architecture we notice that the planning depends on the CANbus module. It is because the planning module needs to get the chassis information, like the car's speed, acceleration to help the planning model to get a more accurate estimate plan while the can is moving.

Control → CANbus: Similar to the Planning new dependency, the control module needs to get the chassis information from CANbus to help the controller know should it apply throttle, brake or steering wheel.

Map → Perception: In the concrete architecture, we added that Map has a dependency on the perception module. After examining the source code, we found that in order for the map to be generated, it will process the information received from lidar, inference, camera, fusion and other submodules in the perception module. The functionalities of these dependencies are to make sure that the map will always be of the most recently updated version.

Map → Localization: In concrete architecture, we notice that map has another dependency, localization. The map module requires the output of the function localization_Estimate, which is stored in the localization module, to generate a map. This newly discovered connection between these two modules is used to provide the map module with the orientation of the vehicle.

Map → CANbus: In concrete architecture, we found that the map module depends on CANbus for the reason that the map module requires the output of the "chassis" function in CANbus in order to generate output with complete functionality.

Map → Planning: There is a pnc_map submodule inside the map module, pnc_map is a module that gets the routing module output and processes the data so that the planning module can reference it to produce the plan. But, because we merged the routing module and planning module into one module. So we have this extra dependency.

CANbus → Control: CANbus module receives a message named ControlCommand from Control module. This message is sent by the control component (apollo/modules/control/control_component.cc). After it receives the message, the command is converted into ASCII then passed on to functions in the Control module.

Guardian → Control: Guardian receives the commands from the control module, and checks the malfunctions of throttle, steering target, steering rate, and safe mode status.
We found this connection in the file"/modules/guardian/guardian_component.cc"

Removed (redundant) dependencies

Control → Guardian: The control module is supposed to depend on the Guardian module based on the functions they have, the Guardian should check the signals the Control module has. We investigated further into the source codes and found out that there is no such dependency which is out of our expectation.

Localization → Map: Localization provides the RTK based method which incorporates GPS LIDAR and IMU information, there's supposed to be a dependency from Localization to Map since it needs the GPS information. However, looking deep into the source codes we didn't find the dependency or relation we expected to be.

Reflexion analysis - Second-level (Control Module)

New (unexpected) modules

Integration Test: The integration test is composed of 4 different tests:hybrid_transceiver test, intra transceiver test, rtps transceiver test and shm transceiver test. It is responsible for testing the validity and integrity of other submodules through an integrated test before operation.

Controller Submodules: The second unexpected composite module is the controller submodules which contain smaller 3rd-level modules. It has a pre-process submodule that receives LocalView written by the control component module and processes it, then two controller submodules will process the information sent by pre-process to generate the controlcommand. Then post-process

will check the controlcommand to handle the error or perform an estop(emergency stop). It will send a stop control command if the error can not be handled.

Unexpected dependencies

Integration Test → Control Component: Integration test tests the validity and integrity of other submodules before operation starts, so integration tests will need the control component(main function) to process the test.

Control Component → Controller Submodules: Control Component has the flag to determine whether it uses submodules or not. If the control component uses submodules in that case, it depends on the preprocessor module to start the submodules.

Controller Submodules → Controller: Most algorithms in the control module are stored in Controller, when Controller Submodules need to generate new commands, they need the algorithms stored in Controller.

Controller Submodules → CANbus: When generating control commands, chassis information is needed. Every function in controller submodules need this information to work(lat_lon_controller,mpc_controller,postprocessor,preprocessor)

Controller Submodules → Control Component: The controller submodules need the local view created by the control component to analyze chassis, localization, and ADCTrajectory in order to process the control command.

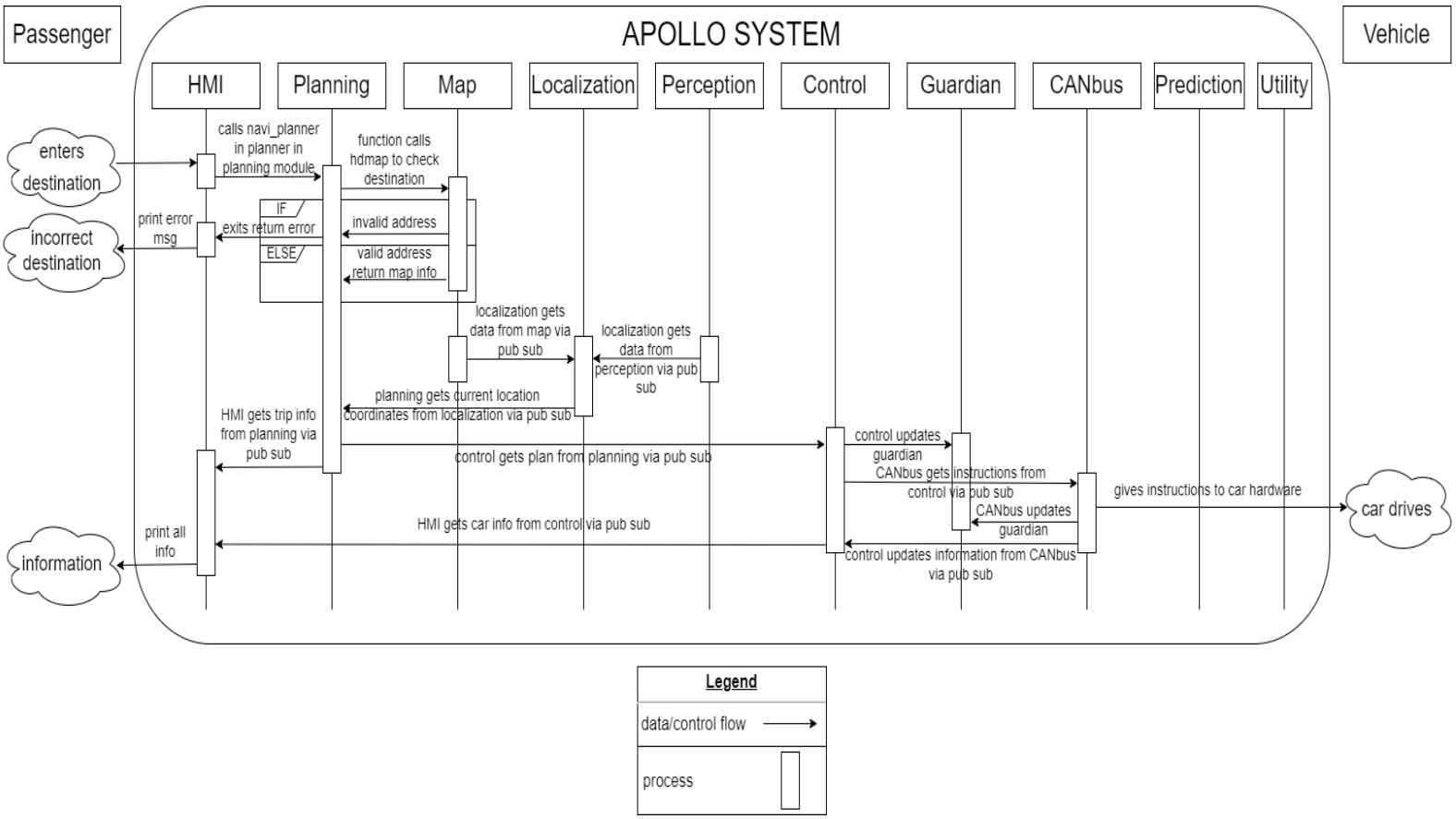
Controller Component → Planning: Because we merge the receiver module to the controller component, so the controller component will directly get the ADJTrajectroy from the planning module.

CANbus → Control Component: At the end of the control component process, it will send the ControlCommand to the cyberRT, and the CANbus will get this control command in order to control the car.

Absences - modules

Receiver: The receiver module is missing from our concrete architecture since its functions are integrated into the ‘controller submodules’ module.

Use Cases

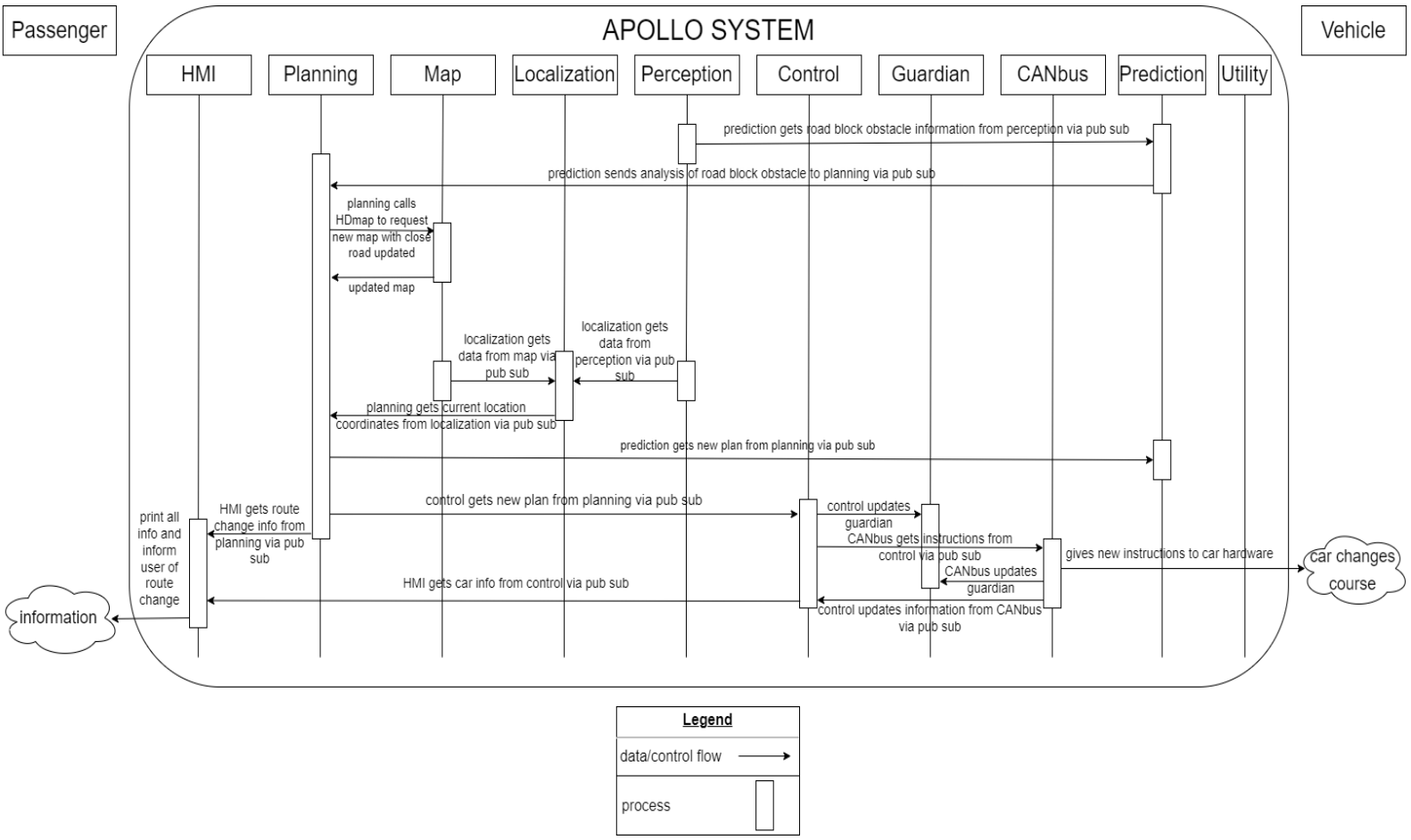


use_case1.drawio (draw.io file link for clearer view)

Use case 1: Apollo plans route and starts trip according to address provided by passenger.

In this use case the passenger first enters the address of their desired destination, the HMI then calls the navi_planner function in the planner folder in the planning module to plan a route. First, planning calls function in HD map to get map info, planning exits if map module finds address entered to be invalid and informs passenger via HMI, if not, map returns the map data to planning module. The routing module within planning then gets current coordinates from the localization module, localization calculates current coordinates using map and perception data. Routing module then uses both map data and localization data to plan a desired route. Once a route is planned and published, control module gets the route and outputs direction instructions. These instructions are then further processed by the CANbus which translates it to outputs that the mechanical components can understand, and send to the car. Lastly, CANbus gathers car information and sends it back to the control module for analysis. HMI meanwhile, gets updated information from many modules like trip plan from planning and car status from control, and outputs it to the passenger.

Since the architecture utilizes pub-sub style, most of the data outputs of modules are published to pub-sub module (CyberRT), and most inputs of modules are acquired by subscribing to pub-sub module (CyberRT).



use_case2.drawio (draw.io file link for clearer view)

Use case 2: Apollo encounters a closed road, and proceeds to recalculate the route.

In this use case, the perception module finds that there are obstacles (road closed signs) in the road ahead. Then it will send this message to the prediction module. The prediction module performs predictive analysis of obstacles ahead based on the received data. Then the prediction module will send the analysis result to the planning module. The planning module will conclude that the road ahead is impassable based on the results of the perception prediction. Then its submodule, routing, will call and access the localization and map module for re-routing. The planning module receives a constant stream of real-time information about the external environment from the prediction module during this process. Then, the planning module will calculate a feasible trajectory and send it to the control module. After processing the planned trajectory by a series of controller algorithms, the control module will generate a sequence of control commands and send them to the Canbus module. Finally, the Canbus module will control the vehicle based on the control commands and avoid the closed road sections.

CANbus gathers car information and sends it back to the control module for analysis. HMI meanwhile, gets updated information from many modules like trip plan from planning and car status from control, and outputs it to the passenger.

Since the architecture utilizes pub-sub style, most of the data outputs of modules are published to pub-sub module (CyberRT), and most inputs of modules are acquired by subscribing to pub-sub module (CyberRT).

Limitations and Lesson Learned

There were numerous limitations our team encountered during the working stages of this report. The first limitation we encountered was interpreting C++ language codes. Our team members have limited knowledge and experience with C++, so the second level architecture of the Control subsystem was quite hard for us to produce. The second limitation is that analyzing github source codes is very time consuming, it is impossible to understand the architecture by just looking at source codes given our time constraint, so we chose to approach the problem of figuring out the connections between subsystems and their functionalities by using mainly the understand tool and interpreting the top level subsystem using ReadMe files and documentations. Another limitation was the discussion of whether some subsystems should stay in the top level concrete architecture or should we put them into other subsystems as they shared lots of functionalities. Initially, we started with 14+ subsystems, which were eventually merged into the 10 module concrete architecture we presented.

From the course materials we've studied, and the limitations we encountered, we learned that pub-sub and pipe and filter architectures are optimal architectures for super complex problems that require concurrency. Secondly, the main difference between conceptual architecture and concrete architecture is that, in concrete architecture, we at times have to split and merge subsystems, depending on their shared functionalities and their shared dependencies with other modules. Also, we gained experience on how to form a decent concrete architecture by using understand and analyzing source code. Lastly, we learned that to approach dependencies, we should accomplish this in three steps: 1. To find out the reason why the unexpected dependencies occur. 2. Put effort into analyzing and searching on dependencies that are missing in our conceptual architecture. 3. Combining the working result of the previous two steps and building a better understanding of the relationship of dependencies. Overall speaking, the course material helped us finish the concrete architecture with better quality.

Our frustration with the source codes led us to learn more about the C++ programming language (syntax mainly). Moreover, by communicating within the group, and with TAs, each one of us learned to communicate as a team, as well as better ways to get the message across more effectively.

Conclusion

To sum up, Apollo open source autonomous driving program is enormous, but with the help from the pub-sub communication visualization graph and the Understand graph we created, we managed to implement the concrete architecture of the Apollo program. The concrete architecture was quite different from its conceptual architecture. The biggest difference is the addition of the Utilities subsystem, most subsystems depend on the Utilities subsystem to work properly since the Utilities module provides many utility functions for other modules to use.

Apollo's concrete architecture met our expectations nonetheless. The architecture itself is still based on pub-sub and pipe and filter architecture design style. As discussed earlier, we added one new subsystem which is the utility subsystem and modified the planning subsystem to enhance and consummate our concrete architecture. The source codes of the Apollo program were a nightmare to analyze, reviewing them gave us a reality of what software development would be, there were plenty of unexpected dependencies and everything was not as clear as what we expected due to the

sheer complexity and size of the architecture, even if there are numerous readme files and comments.

With the implementation of our new understanding of the concrete architecture and the two use cases, we have a more comprehensive understanding of the Apollo architecture. The analysis and knowledge base in this report will be our proposal to the next object, which is adding improvements and new features to Apollo.

Glossary

Conceptual Architecture: the developers' view of the software architecture

Concrete Architecture: Implementation of the specific architecture with the decomposition into specific components and identification of actual relationships

C++: A widely used computer programming language. It is a general-purpose programming language that supports multiple programming paradigms.

Apollo Program: The open-source autonomous driving platform developed by Baidu.

Modules: Important subsystems within the software, each subsystem has one or more specific functions. Modules play a key part in software architecture.

ADCTrajectory: Trajectory planned by the planning module based on the user input.

GPS: Global position system that is used to track location and generate navigation.

LiDAR: Light Detection And Ranging Sensor, determines the distance by pointing a laser at an object and measuring the time it takes for the reflected light to return to the receiver.

IMU: Inertial Measurement Unit, which can report the device(car)’s specific force, angular rate.

Pipe and filter: an architectural style, made up of a number of pipes (connections) and filters (module).

Pub-Sub: an architectural style, which relies on a broadcaster which can receive the messages from the publishers, and send the messages to the subscriber which is waiting for a specific message. subscribers don’t know who sends the messages but only receive them.

RTK: Real-time Kinematic, which is stored in the perception module.

HMI: Human-Machine Interface is a user interface that allows users to interact with the machine.

Subsystem: It is the sub-component of its higher-level system.

Reference

[1] Wang, Y., Jiang, S., Lin, W., Cao, Y., Lin, L., Hu, J., ... & Luo, Q. (2020). A learning-based tune-free control framework for large scale autonomous driving system deployment. arXiv preprint arXiv:2011.04250.

[2] Behere, S., & Törngren, M. (2016). A functional reference architecture for autonomous driving. Information and Software Technology, 73, 136–150. doi:10.1016/j.infsof.2015.12.008

[3] GitHub. 2022. GitHub - ApolloAuto/apollo: An open autonomous driving platform. [online] Available at: <https://github.com/ApolloAuto/apollo> [Accessed 20 February 2022].

[4] Leigh, B. and Duwe, R., 2019. Software Architecture of Autonomous Vehicles. ATZelectronics worldwide, 14(9), pp.48-51.

[5] GitHub. 2022. apollo/README.md at master · ApolloAuto/apollo. [online] Available at: <https://github.com/ApolloAuto/apollo/blob/master/modules/perception/README.md> [Accessed 20 February 2022].