CISC 322

Assignment 2: Report

Concrete Architecture of Apollo

Team Artemis

Josh Otten (18jbo@queensu.ca)

Aleks Jugovic (16anj@queensu.ca)

Muyun Yang (18my24@queensu.ca)

Chong Guan (18cg14@queensu.ca)

Daniel Jang (17ddhj@queensu.ca)

Wooseok Lee (20wl9@queensu.ca)

Abstract

After the construction of the conceptual architecture on Apollo, we intended to go further on exploring the detailed architecture. This report recovers the concrete architecture of Apollo with new subsystems and dependencies. In order to discover how the concrete architecture varies from the conceptual architecture, the report will not remain at the level of reading documents but keep searching in the code. By using the dependency graph from the Understand tool, it is easy to figure out how subsystems are related. However, to go further into dependencies, we still have to look into the code in Github. Under the Pub-Sub style of architecture, we improved conceptual architecture and included a couple of new subsystems and dependencies. Due to the technology of message subscription, some dependencies are implicit to the code but revealed in dependency graphs. We made a comparison between the two architectures to explain how the architecture evolved. Furthermore, we chose localization as the level-two subsystem which would be analyzed. In this section, we presented the conceptual and concrete architecture for the localization module and broke down the subsystem into several functioning components. In the end, we proposed two use cases as usual.

Table of Contents

Abstract **2**

Table of Contents **3**

Introduction **4**

Derivation Process **4**

Updated Conceptual Architecture **5**

Concrete Architecture **7**

New Subsystems **8**

Common 8

Drivers 9

Task Manager 9

New Dependencies **10**

Second-Level Subsystem (Localization) **11**

Use Cases **13**

Use Case 1: Normal Driving 13

Use Case 2: Emergency/Failure 14

Current Limitations and Lessons Learned **15**

Conclusion **15**

Data Dictionary **16**

Naming Conventions **16**

References **17**

Introduction

The purpose of the report is to recover the concrete architecture through the analysis of the software dependencies among all the subsystems and perform a further reflexion analysis. The report will introduce what is new in the concrete architecture and explain it at the code level.

The first section is to present the process of our derivation for extracting the concrete architecture, the second-level subsystem and use cases. The next section is to demonstrate graphs of the updated conceptual architecture and the concrete architecture. The following two sections are to go deep into the concrete architecture and explain the new components that did not appear in the conceptual architecture.

The analysis section starts with the introduction of a second-level subsystem and its conceptual and concrete architectures. The following section is the reflexion analysis, exploring the changes between the two architectures. In the concluding section, two use cases are the first ones to be discussed, and it is followed by lessons we learned in this assignment and the conclusion of the report.

The crucial point of the report is to propose a concrete architecture with a comprehensive explanation of it. We presented a graph about the concrete architecture and two sections describing new subsystems and new dependencies. To have a clear view of the architecture, we also provided an in-depth analysis of one of the subsystems.

Derivation Process

To derive the concrete architecture, we used a similar series of steps to assignment 1: individual research, consolidation of ideas, and finalization of the architecture.

For the individual research, everyone used the Understand tool and the provided analysis file for the derivation of the concrete architecture. The provided publish-subscribe diagram [3] was also used. Each person made their own version of the concrete architecture and chose which subsystem they thought was best to study. After this was done, they were consolidated into one concrete architecture and refined as a group. The concrete architecture was also created using Understand and refined as a group.

After the creation of our final concrete architecture, we have noticed that the arrow direction was almost opposite compared to the conceptual architecture that we have previously built [2]. The original arrow was pointing to subsystems with the meaning of “data flows into” rather than “depends on” which caused the mixup. We have updated the conceptual architecture diagram so that the arrows are now pointing in the right direction.

Reflexion analysis was completed on all deviations between the conceptual and concrete architecture. For each deviation found, we used the Understand tool to point to which files in Github [1] were the reason for a dependency between the subsystems. Then, using the blame functionality on Github [1] we were able to find the commit that introduced the dependency and deduced the reasoning from a combination of the commit message and the code that was inserted.

In selecting a second-level subsystem to analyze further, localization was selected due to its numerous connections to other subsystems, and its inclusion of numerous, well-defined subsystems.

Due to not being previously required to create architectures for any subsystem, a conceptual architecture was created through referencing the localization documentation, and through consulting published literature on similar localization systems. In creating the concrete architecture for the localization subsystem, the Understand tool was used to see how each subcomponent interacted with each other while referencing the source code aided in developing the concrete architecture. In performing the reflexion analysis, individual source code files aided in mapping the conceptual architecture to the concrete architecture. In comparing the two architectures, the rationale for discrepancies was found and the concrete architecture was finalized.

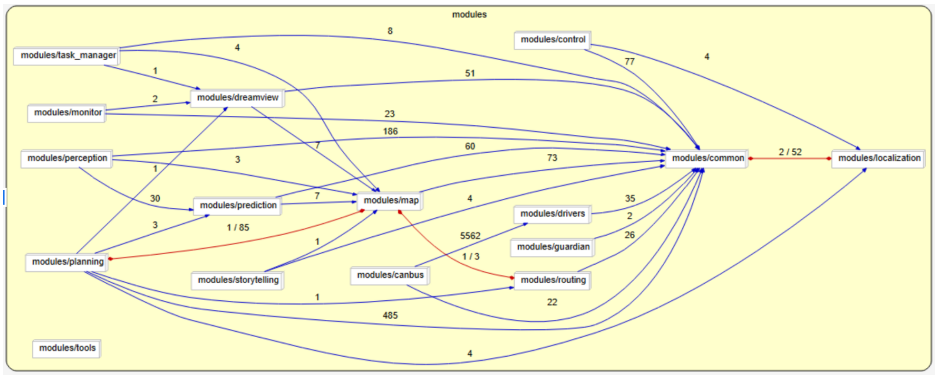

Figure 1: Part of the Dependency graph generated by the Understand tool

Updated Conceptual Architecture

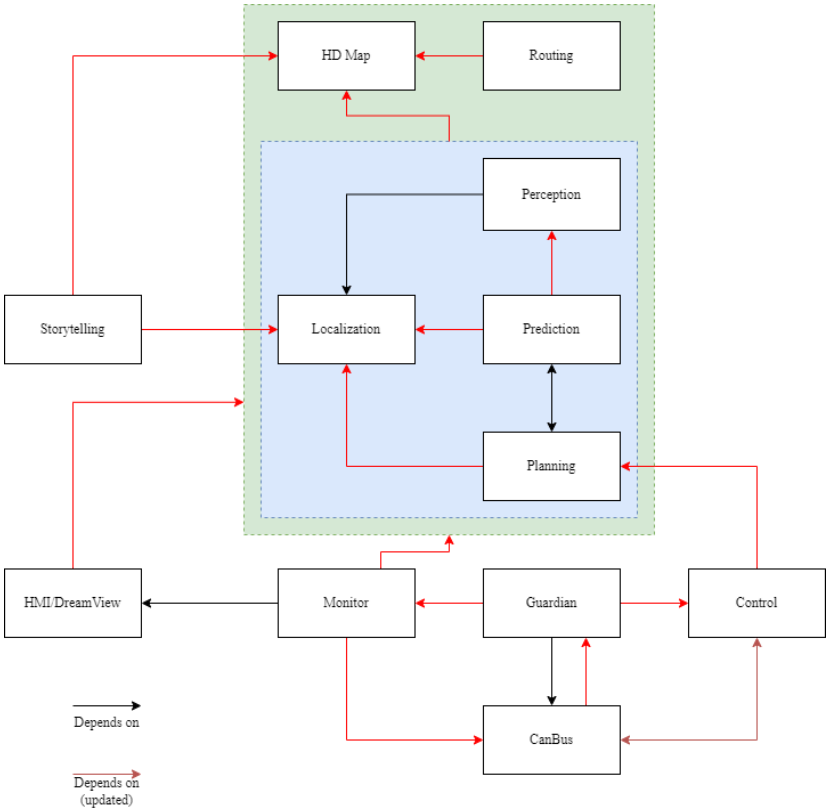

Figure 2. Updated Conceptual Architecture

Our updated conceptual architecture can be seen in Figure 2. We made two major types of changes to the original conceptual architecture [2] based on our increased understanding of dependency arrows and through analyzing more of Apollos README files [1].

Changing the direction of dependency arrows occurred for the following subsystems: Control -> Planning, Prediction -> Perception, Prediction -> Localization, Planning -> Localization, Guardian -> Control, Monitor -> Outer Box, HMI -> Outer Box, Storytelling -> HD Map, Routing -> HD Map, and Inner box -> HD Map. When we created the diagram, we interpreted an arrow pointing towards another subsystem to indicate that data was being sent from one subsystem to the subsystem where the arrowhead was pointing towards. For example, since Control takes the plan from Planning, we had an arrow pointing from Planning to Control. However, this should be flipped as Control has a dependency on Planning since it subscribes to messages from that subsystem. This was the same case in every arrow change outlined previously, and since the concrete architecture uses this method, we think it was necessary to update the conceptual architecture with this change to have a unified approach to the dependency arrows across the two.

The second change involved creating a few new dependencies in the architecture as reanalyzing Apollos README files [1] indicated that some subsystems conceptually should have had an arrow between the two. The first new dependency was from CanBus -> Guardian. This is because whenever a command is sent to the CanBus, it will always return Chassis information to the command sender. Guardian can send commands to the CanBus and thus it will receive this Chassis information back. The second new dependency is Control <-> CanBus. We realized through the documentation that Control can directly send commands to the CanBus without needing to go through the Guardian, indicating that there needed to be a dependency between the two subsystems. As with Guardian, CanBus will also send Chassis information back to Control which is why the dependency is a double arrow. The final new dependency is between Monitor -> CanBus. We previously had no dependency that allowed Monitor to check the status of the hardware of the vehicle which is something it does as outlined in the documentation. Since the CanBus is the link between the software and hardware of the autonomous vehicle, this dependency is required to reflect this specification.

Concrete Architecture

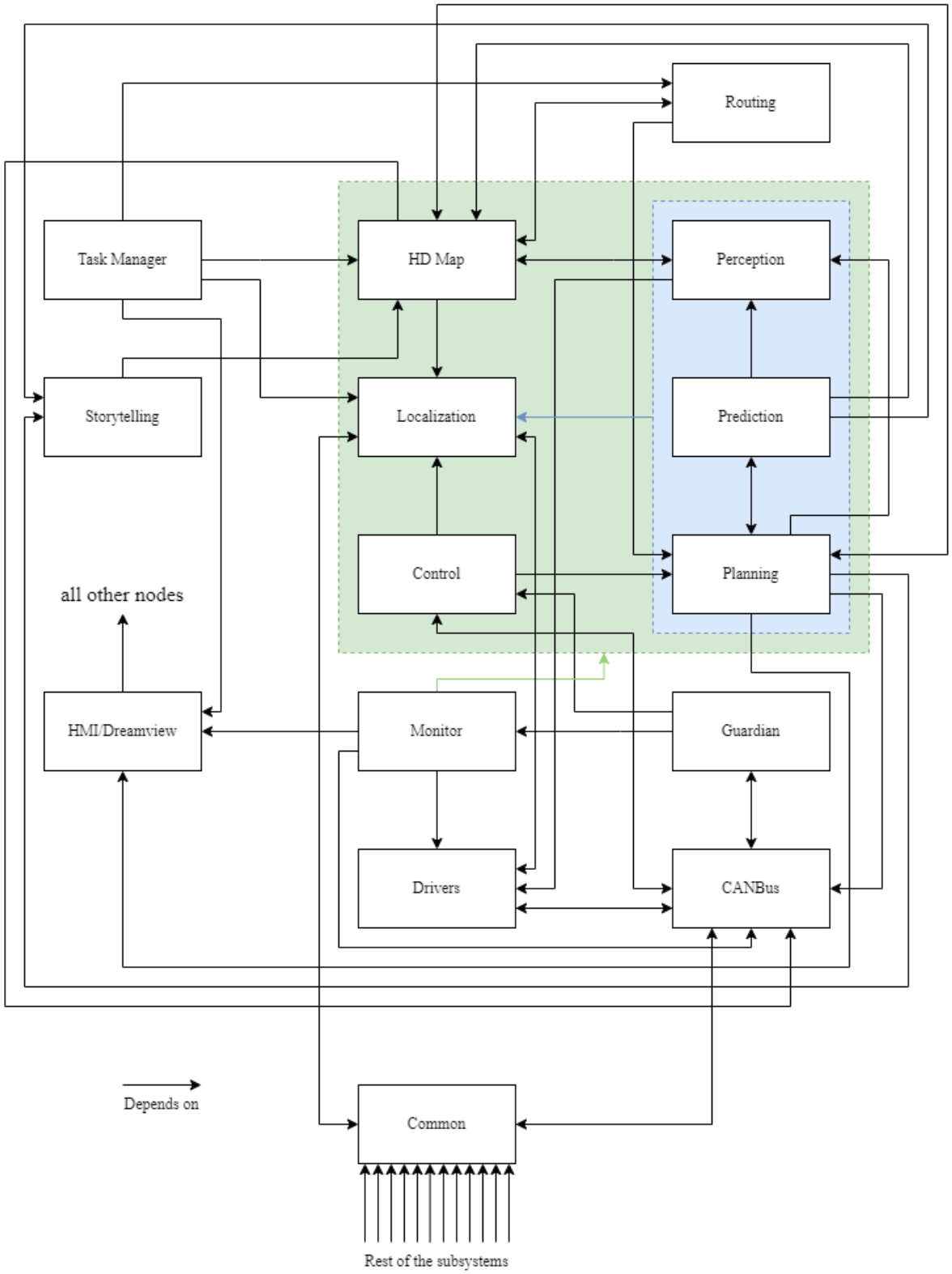

Figure 3. High-Level Concrete Architecture

Based on the new Understand tool and Pub-Sub diagram [3], we built Apollo's concrete architecture from our conceptual architecture diagram, as shown in Figure 3. The main architectural style uses a mix of concrete methods and pub-sub messages.

The perception module uses multiple sensors to identify objects, such as LIDAR and RADAR scanners. Important information such as position, velocity, and the state of traffic lights is output to other modules. The prediction module listens to multiple sources like storytelling and predicts the behavior of objects detected by other modules, while the planning module finds the best trajectory for the vehicle. Then it gives instructions for many other modules. The localization module estimates the current position of the vehicle using various data sources. Routing module constantly generates and updates advanced navigation based on routing topology. The control module uses the CanBus module to actually control the car and execute the planned trajectory. The CanBus module sends the received commands to the vehicle hardware to perform the desired action. The monitoring module monitors status data from most modules, allowing drivers to view system status and send notifications to guardians in the event of a malfunction. HD Map is a library with modules like perception, planning, and routing that query structured information about roads. The HMI/DreamView module can also listen to the HD Map, as it can use the maps generated to allow the user to visualize data and interact with the vehicle. The guardian is the safety module of the vehicle, making decisions based on the information sent by the control and monitoring modules. The Storytelling module is an advanced global scenario manager. This module creates complex scenarios, triggering the actions of multiple modules as needed.

There are three new modules in the concrete architecture. Common modules contain basic information such as car type and functions such as math. The Task Manager module acts as a relay station for some other modules to exchange information. Most of the driver modules are related to the hardware involved in the driving process, such that many modules depend on it.

New Subsystems

Common

The Common subsystem contains code that is useful for the functioning of Apollo but not specific to any subsystems [1]. Some examples of useful code include Adapters where a large number of topic names are defined to be used by different subsystems to communicate with one another. The data folder [1]contains vehicle configuration data including the brand of the car, the body measurement of the car to its center, calculated values of minimum turn radius, maximum steering angle, etc. Filters folder [1]contain code for different filtering functions including MeanFilter class that is used to smoothen a series of noisy numbers, such as sensor data, and DigitalFilter class that is used to filter signals with its level of frequency that outputs filtered signals based on a certain cutoff frequency. KVDB [1] is a lightweight key-value database to store and delete system-wide parameters. There are other useful codes in the Common subsystem such as Latency_recorder, Math, Monitor_log, Proto, Status, Util, Vehicle_state, that are also being used by other subsystems. The Common subsystem depends on two subsystems which are the Localization subsystem and the CANBus subsystem. The Vehicle_state [1] from the Common subsystem takes in the localization information of the vehicle as a parameter and the vehicle's Chassis information from the CANBus subsystem to update the status of the vehicle.

Drivers

Drivers subsystem is a subsystem that has a driver program to operate hardware like camera and radars that is attached to the vehicle.

The Monitor subsystem depends on the Drivers subsystem [3]. For hardware, it checks GPS information based on the input of the GNSS system from the Drivers subsystem. For software, it checks if the radar and point cloud runs or not, which are subsystems of the Drivers module.

The Localization subsystem depends on the Drivers subsystem as it uses motion compensation for point cloud service to optimize the accuracy of Nodes in the map. The Drivers subsystem depends back on the Localization subsystem as it uses the GPS in the Localization subsystem in the data parser of the GNSS system (transformation of the coordinate system) and the Drivers module also publishes [3] correct IMU data given by the input from the Localization subsystem.

The Perception subsystem also depends on the Drivers subsystem [3] for the point cloud service and the use of the continental radar hardware.

The CANBus subsystem depends on the Drivers subsystem as several classes [1] including Can Client, Message Manager, Can Sender, and Can receiver, in the Drivers subsystem are instantiated in the CANBus module. Can Client class [1] communicates using messages via Can senders and Can receivers where CANBus gets the input message (control command) through the Can Client class. The Drivers subsystem depends on the CANBus subsystem in testing the Message Manager class where the chassis detail of the vehicle is used as input data from the CANBus subsystem.

Task Manager

The Task Manager subsystem acts as a subscriber [3] to two subsystems which are the Localization subsystem and the Routing subsystem. The Task Manager component runs the localization callback as it receives the localization data or runs the response callback as it receives the routing response data. The Task Manager subsystem also uses the position data from the Localization subsystem as an input to request new routing requests through the Routing subsystem.

As the DreamView subsystem is a web-based dynamic 3D rendering of the monitored messages in a simulated world [1], the message handler of the simulated world uses the functions from the Task Manager subsystem including dead-end routing task, cycle routing task, and parking routing task.

The cycle routing manager code [1] uses the map service functions from the DreamView subsystem to check whether the current route is allowed.

The dead-end routing manager from the Task Manager subsystem uses one of the classes called JunctionInfoConstPtr [1] to gain junction information from the HD Map subsystem. The parking routing manager from the Task Manager subsystem uses one of the classes called ParkingSpaceInfoConstPtr [1] to gain information about the parking spaces through the HD Map subsystem.

New Dependencies

Routing -> Planning

The dependency of Routing and Planning is from a callback function in the routing_dump.cc file, which records debug information when the Planning module functions properly with the Routing module [1].

Prediction -> Storytelling

The Prediction module takes the story message from the Storytelling module to optimize the accuracy of distinguishing obstacles and generating trajectories [1]. Since the Storytelling module was created to be a high-level Scenario manager to handle complex urban-city driving scenes, prediction of the vehicles' obstacles is optimized by having information about what “story” the vehicle is currently in.

Planning -> Storytelling

The Planning subsystem receives a “story” from the storytelling module. The story message will be processed by the planning module to recreate real-time scenarios which include crosswalks, junctions, traffic lights, stop signs, and yield signs [1]. This dependency was added to tag data received and improve the online automated planning of the system.

Planning -> Perception

The Planning module subscribes to the Perception module, which publishes TrafficLightDetection data for Planning to receive [2]. As Planning creates the plan for the autonomous vehicle motion, it was important for this dependency to be added as seeing a traffic light and the color of the traffic light will alter the plan for the vehicle to take.

Planning -> CanBus

The Planning module takes input from CanBus to monitor the status of the vehicle chassis [2].

HD Map -> Planning

The HD Map module has a dependency with the GFlags generated by the planning module [1]. This was added in one of the map files within HD Map to add nodes representing the current plan of the vehicle on top of the generated map.

HD Map -> Perception

The HD Map module subscribes to the Perception module to receive PerceptionObstacles from it [3]. As the HD Map subsystem functions to generate maps of the world surrounding the vehicle, one of the core feature requirements is to outline obstacles that are detected by the Perception subsystem. This required the obstacles identified by Perception, and thus a dependency was required between the subsystems.

HD Map -> Routing

The HD Map module uses a routing response generated by the Routing subsystem in the pnc_map.cc file. This routing response is used to output the planned route of the vehicle onto a map for the driver of the vehicle to see. This dependency was required as there would be no other way for a map to be generated of the current route without having the Routing module send this information to the HD Map subsystem [1].

HD Map -> CanBus

The HD Map module subscribes to chassis information as input that is published by the CanBus module. This is to generate a relative map from navigation routes [1].

HMI/DreamView -> All other nodes

As the HMI/DreamView module in Apollo represents the interface for the driver of the vehicle to interact with the system and view the status of other subsystems, this dependency was added in order for the interface to be able to see what is going on in the entire software system of Apollo [2]. By listening to all other modules, the user will be able to live track every update that is occurring in the system.

Second-Level Subsystem (Localization)

We are analyzing the Localization subsystem and will present both the conceptual and concrete architecture of the subsystem. The localization module estimates the current location of the vehicle using a combination of different data sources which include LiDAR, RTK, and IMU. Each data source has its own benefits and drawbacks, for example, LiDAR works best in areas with 3D features while RTK performs at its best in open space [4]. Apollo uses a unique system that fuses the data received from each of these sources, leveraging their benefits to achieve centimeter localization accuracy in challenging scenes such as highways, tunnels, and city downtowns [4].

Conceptual Architecture

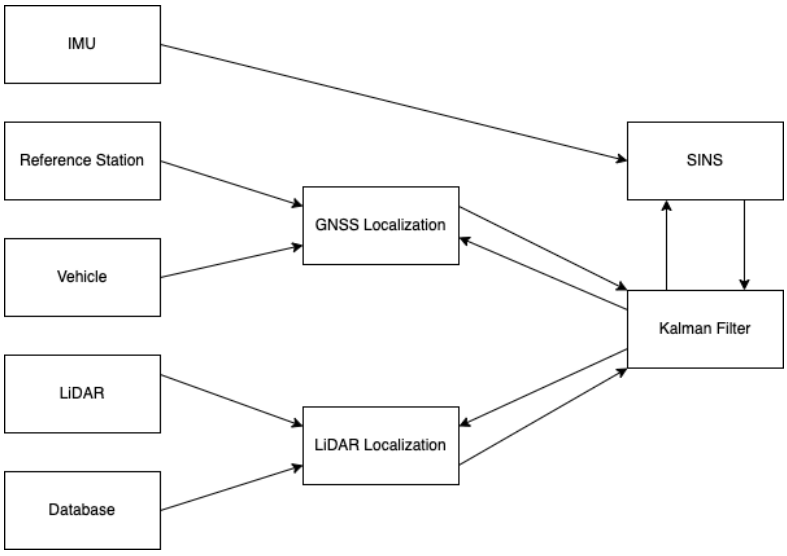

Figure 4. Second-Level Conceptual Architecture

The conceptual architecture of the Localization subsystem can be seen in Figure 4. The subsystem estimates the position, velocity, and altitude of the vehicle by combining sensor input and prebuilt LiDAR map data from a database. This data is sent to the GNSS and LiDAR systems to generate a localization estimate of the cars' position and velocity combined with variance [4]. The fusion framework takes both of these localization estimates, along with data from the IMU to account for the force and rotation rate of the vehicle [4]. An error-state Kalman filter is applied to combine the localization data from the IMU, LiDAR, and GNSS sensors [4]. This produces the localization estimate that can then be used by other subsystems in Apollo. Localization follows a pipe and filter architectural style as the entire process of generating a localization estimate involves individual components receiving data as input, applying their own transformation to it, and then outputting this newly modified data to another inner-subsystem.

Concrete Architecture

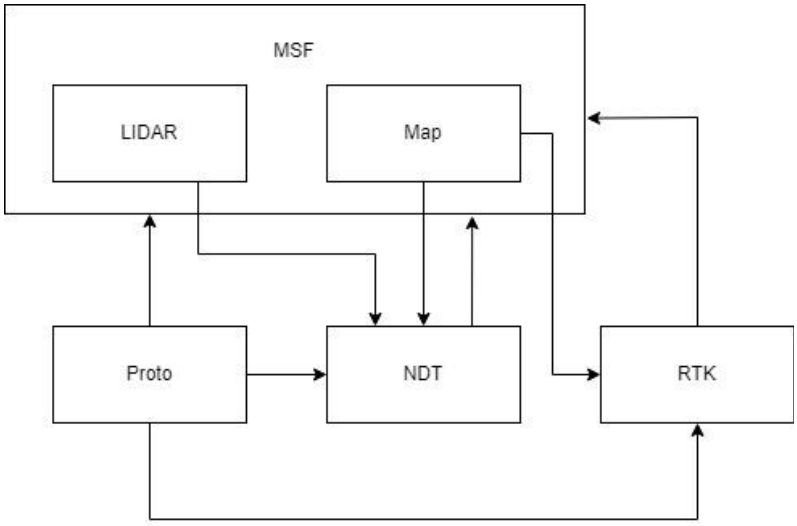

Figure 5. Second-Level Concrete Architecture

The concrete architecture of the Localization subsystem can be seen in Figure 5. The main components include RTK localization (Real-Time Kinematic), NDT localization (Normal Distributions Transforms), and MSF localization (Multisensor Fusion Localization). The Proto subsystem handles most input/output, including GPS, IMU, GNSS, and SINS inputs. Other important information is also collected from Proto as it is where messages are received for the Pub-Sub architecture style. LiDAR handles Velodyne LiDAR [1]. Map information is handled within MSF, from which NDT bases its NDT maps and where RTK gets its map information.

Reflexion Analysis

The fusion framework is the main focus of the localization subsystem in Apollo, utilizing reference information, vehicle information, LiDAR, and map information to generate localization data. MSF represents the SINS and Kalman filter components of the conceptual architecture. NDT represents the LiDAR localization while RTK represents the GNSS localization. A key difference between the two architectures is in the way that input is handled. MSF is the main focus of Apollo's localization subsystem since it combines aspects of the two other systems of localization. As a result, components such as LiDAR and map information are handled within the MSF subcomponent

while RTK and NDT depend upon MSF for the data relevant to them. As a result, those subsystems are included inside the larger MSF subsystem.

Vehicle information and IMU data are handled within Proto due to Proto handling communication with other subsystems. As a result, those two subsystems in the conceptual architecture are combined together with the rest of Proto's functionality. Since reference information is only handled within the RTK subsystem, those two components are combined in the concrete architecture so there is no explicit connection there when compared to the conceptual architecture. The MSF subsystem encompasses both the Kalman filter and SINS components of the conceptual architecture and is combined as a result.

When looking through the structure of the localization subsystem, several design patterns appear. The three methods of localization are kept largely separate, with RTK and NDT communicating with MSF to obtain necessary information. Each of these three subsystems follows a similar nomenclature and file structure with the main localization components being named similarly and their smaller component files following a similar pattern. Furthermore, the MSF subsystem further breaks down its localization into subcomponents each following a similar naming convention but instead using the individual input sensor names as a reference. e.g. LiDAR, GNSS, and other hardware inputs.

Use Cases

Use Case 1: Normal Driving

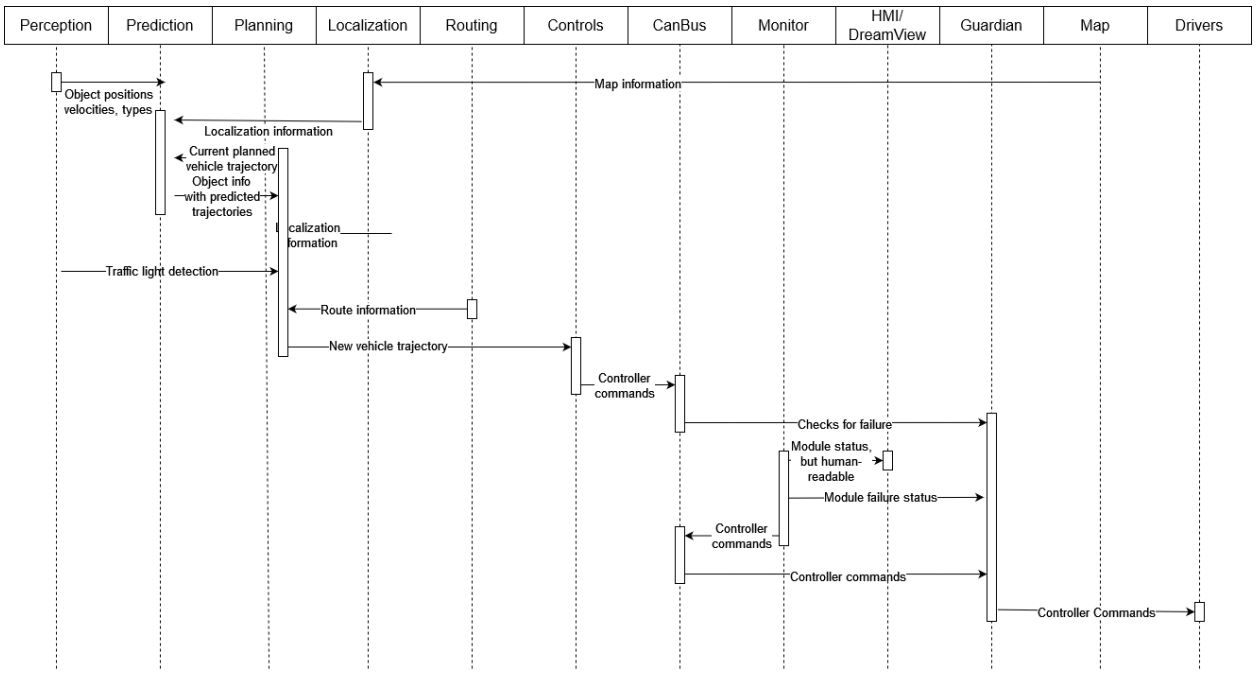

Figure 6. Sequence diagram of normal driving

This use case is for driving along in normal conditions. This use case was chosen because it would be the most common use case.

The perception module takes the sensor input and uses it to detect objects, including their types, positions, and velocities [1]. It uses the Drivers module to interact with the sensor hardware. This gets sent to the prediction module. The vehicle position information and the current trajectory are sent from the localization module to the prediction module [3]. All of this information is used to predict the trajectories of the previously detected objects. This gets sent to the planning module. The planning module uses localization information as well as traffic light information from the perception module and route information from the routing module to update the planned trajectory of the car [1]. The localization module gets map data from the Map module [1]. The resulting localization data gets sent to the control module, which converts that trajectory to control commands that can be sent to the CanBus system that actually enacts the controls [1]. While this is all happening, the Monitor system is monitoring all the other modules and checks for failure. The monitor system also outputs human-readable data to the DreamView module, which displays it to the user [1]. When there is no failure found, the control commands are passed through the Guardian module to the Drivers module to output to the hardware [1].

The main difference between the concrete architecture and the conceptual architecture was that the conceptual architecture was missing the Drivers module and the Map module.

Use Case 2: Emergency/Failure

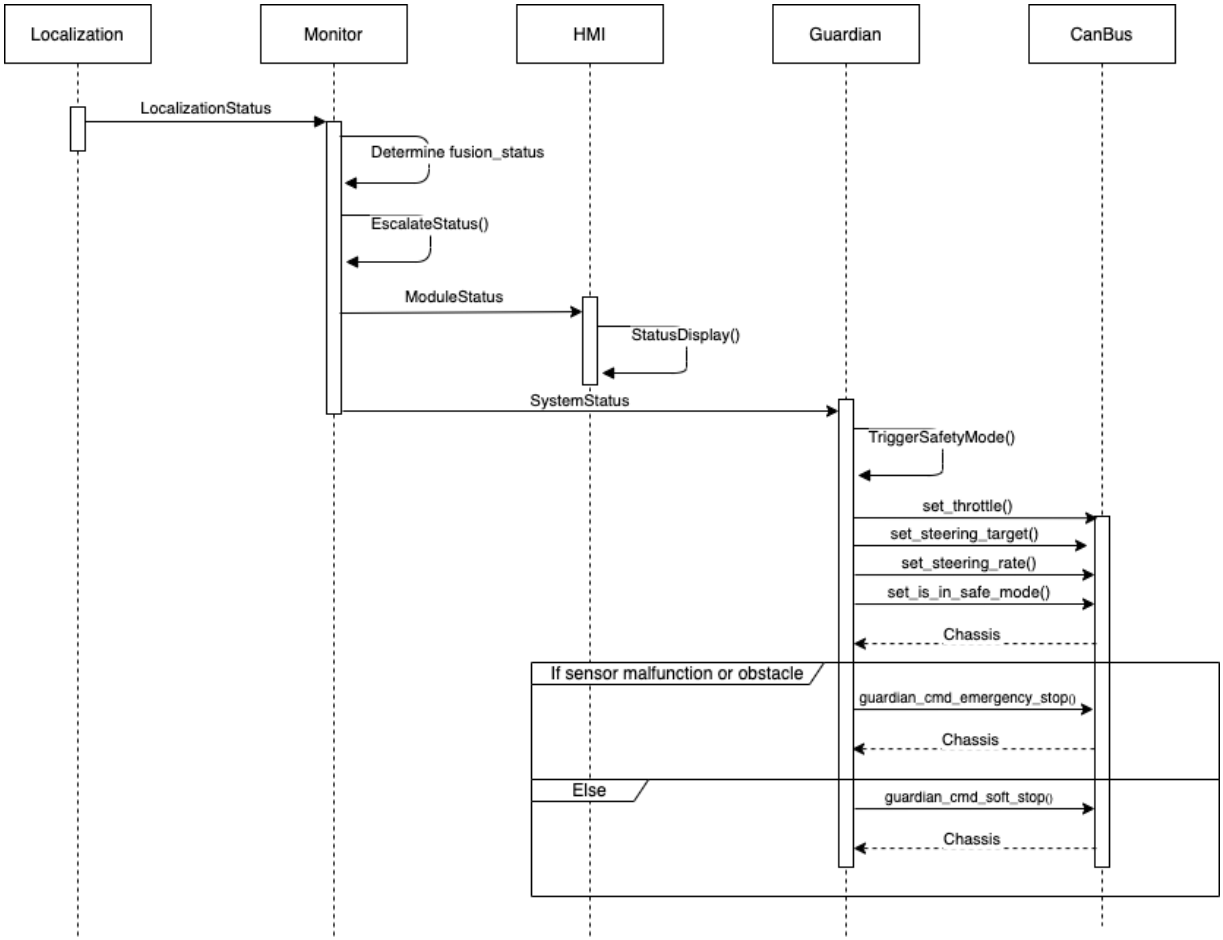

Figure 7. Sequence diagram of emergency/failure

This use case is for when a failure is detected in a module or car component forcing the vehicle to make either an emergency stop or a soft stop. Since this can occur in any module or hardware component, the case of the Localization module failing was chosen to illustrate one example of this.

The Monitor subscribes to status messages published by the Localization module [3]. This use case starts with the Localization module publishing a status that the Monitor module receives, indicating that it is no longer working [3]. After receiving the message Monitor will determine the status of the message using an internal method, and will call its internal method EscalateStatus() to propagate the failure to the required other modules in the system, the first of which is the driver of the vehicle [1]. Monitor publishes this failure status, and HMI will receive it and alert the driver using its internal frontend libraries [1]. Then Monitor will publish the failed system status to the Guardian component as the vehicle now needs to be stopped [3]. Internally in Guardian, it will see the failure in the message and call its internal method TriggerSafetyMode() to begin the stop of the vehicle [1]. No matter what kind of stop is required, Guardian will send control commands to the CanBus to control the initial throttle and steering of the car, along with triggering the safe mode in CanBus [1]. Then Guardian will check the status of the ultrasonic sensors to see if it is not working properly, or if there is an obstacle detected in the way [1]. If this is the case, Guardian will send an emergency stop command to the CanBus to perform. Otherwise, Guardian will send a soft stop command that will bring the vehicle to a slow stop. In all of the commands sent to CanBus, the module will send back chassis information to Guardian as a form of feedback control [3].

Current Limitations and Lessons Learned

Throughout this assignment, we learned how to effectively use the Understand tool for deriving the concrete architecture of a system. One limitation we have is that there are a few dependencies where we don't know what they are for. Another limitation we had was that commit messages from the Apollo developers were short and sometimes vague, which made it hard to deduce reasons for changes during reflection analysis. For some subsystems like the Task Manager, the documentation was poor so it wasn't clear what its function was. There were also too many files so looking at every single file would have been a waste of time.

Conclusion

In conclusion, under the Publish-Subscribe style of architecture, the report proposed a systematic concrete architecture, where each module can subscribe messages from any module. It gives the system high flexibility in modifying dependencies among subsystems.

Compared to the conceptual architecture, the concrete architecture uses a mix of concrete methods and pub-sub messaging. It also adds three new modules, which include some aspects of the hardware usage. The concrete architecture focuses more on the complex relationships between each subsystem and the invocation of actual information in real life.

In constructing a conceptual architecture for the localization subsystem, the architecture resembles a pipe and filter architecture style. When placed within the publish-subscribe style that the main architecture of Apollo follows, localization exemplifies how multiple architectural styles can be combined in the creation of a large software system. In localization, inputs are obtained through the pub-sub architecture which is then processed accordingly with the RTK, NDT, and

MSF localization methods used by Apollo in a pipe and filter style, and finally published as the localization output. The differences between the conceptual and concrete architectures of the localization subsystem show how architecture can be changed from how a system was planned while still remaining recognizable, as exemplified by the MSF subsystem.

In addition, two common use cases are discussed in the report. One case occurs when all systems work properly. Apollo first collects information through perception, localization, and map modules, then sends data to processing modules, like planning and prediction modules. After making judgments on the scenarios that cars may be positioned in, control commands will be sent to control modules and the cars will perform corresponding actions. However, when errors are discovered in the system, the second use case will arise. Because the monitor module keeps receiving status messages from every other module, it will be the first one to be alarmed and will notify HMI and Guardian modules as well. The HMI module will display messages to drivers and the Guardian module will respond accordingly.

Data Dictionary

Architecture: High-level structure and organization of subsystems in software.

Subsystem/Module/Component: A self-contained system within the overall architecture. These three words are used interchangeably.

DreamView: A web application that helps developers visualize the output of other relevant autonomous driving modules.

HD Map: A high-precision map loader interface.

README: A README is a text file that introduces and explains a project.

Chassis: the base frame of a motor vehicle.

Naming Conventions

Pub-Sub - Publish and Subscribe Architecture

RTK - Real-Time Kinematic

NDT - Normal Distributions Transforms

CanBus - Controlled Area Network Bus

IMU - Inertial Measurement Unit

RTK - Real-Time Kinematic

NDT - Normal Distributions Transforms

IMU - Inertial Measurement Unit

GPS - Global Positioning System

GNSS - Global Navigation Satellite System

LiDAR - Light Detection and Ranging

References

[1] GitHub repository of Apollo Auto
https://github.com/ApolloAuto/apollo

[2] Conceptual Architecture (Assignment 1)
https://aleksjug.github.io/artemis/

[3] Apollo's pub-sub communication model graph
https://onq.queensu.ca/d2l/le/content/642417/viewContent/3865686/View

[4] G. Wan et al., "Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes," 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018, pp. 4670-4677, doi: 10.1109/ICRA.2018.8461224.
https://ieeexplore.ieee.org/document/8461224