CISC 322/326 - Software Architecture
Assignment #2: Report

Concrete Architecture of Apollo v7

Team Bingus
Haadia Mufti - haadia.mufti@queensu.ca
Emily Poon - 16ejp6@queensu.ca
Kevin Shroff - 17kss@queensu.ca
Oliver Cao -17oyc@queensu.ca
Gregory Secord - 19gdjs@queensu.ca
Connor Colwill - 18cgc5@queensu.ca

Abstract
This technical overview report will go over and compare the details of the concrete architecture and a revised conceptual architecture of Apollo. It will outline our derivation process, and explain the rationale for each subsystem and connection in our architecture. Using software by SciTools such as Understand we were able to map the source code to our architecture, coming to the conclusion that our original proposed conceptual architecture lacked certain components and dependencies. Fortunately, that was to be expected from our conceptual architecture. Using Understand we were able create a new concrete architecture of Apollo which included any missing modules or dependencies from the conceptual architecture. The report also focuses on the CANBus subsystem where we compared the findings of this particular module from our conceptual architecture and the new found concrete architecture. Furthermore, we described 2 use case diagrams of how the Apollo system analyzes component inputs, rerouting and navigation. Lastly we go over the conclusion and findings as well as the lessons learned during the process of writing this report.

Introduction and Overview
Using knowledge from Apollo's conceptual architecture presented in our first report, this report will dive deeper into the concrete dependencies discovered based on Apollo's source code and how our team revised our conceptual architecture.

The first section is the derivation process which describes the process of mapping the high-level directories of Apollo's codebase onto the modules of our conceptual architecture by using the program Understand. By doing so we find more optimal structures that minimize unexpected dependencies from our conceptual architecture. The second section goes over the newfound concrete architecture which encompasses the publish-subscribe style and a high-level brief overview with diagrams illustrating the connections between modules. With this new perspective of Apollo's architecture, the third section will dive deeper into the CANBus subsystem which is responsible for sending information to the dependent modules such as planning, HD map and control. Lastly, the fourth section is a high level reflexion analysis that compares the differences found between the conceptual architecture from the first report and the concrete architecture from the Apollo source code by going over the missing Common and Cyber subsystem and talking about the importance and the dependencies these 2 modules have with the rest of the subsystems.

Derivation of the sequence diagrams of Use Case 1 and Use Case 2 was based on Apollo's source code and documentation. Using this information and the dependency graphs by Understand, as well as the pub sub visualization, we determined how the subsystems are dependent on each other and figured out the responsibilities of the individual modules and came up with the Use cases.

Architecture
Derivation process
In order to derive the conceptual architecture of Apollo, we used the software offered by SciTools called Understand that helped us to visualize the source code of Apollo. Since we derived the conceptual architecture in our previous report, we had the base of what the architecture subsystems would be. Then,

using Understand, we distributed all the Apollo modules into these subsystems that generated a dependency graph. The graph helped us see the various dependencies each module had with each other, and this information was used to see what was overlooked while making the conceptual architecture, for example, adding the module Cyber that was previously not known by us. Furthermore, we were also given a graph visualization of the pub-sub communication that provided us information about any dependencies that might have not been shown by Understand. Using this information, we were able to create the concrete architecture of Apollo and then perform a reflexion analysis where we highlighted the reasons for any new dependencies that the graphs showed us.

A similar strategy was also used to derive the architectures for the subsystem, for which we chose CANBus. For the conceptual architecture, we went through the GitHub of Apollo where we got all the information needed to create the conceptual architecture of CANBus. The concrete architecture was derived using Understand once again. Understand helped us take a closer look at each of the subsystems, with this we were able to see all the dependencies to and from CANBus and a reflexion analysis was performed on this too.

Concrete Architecture

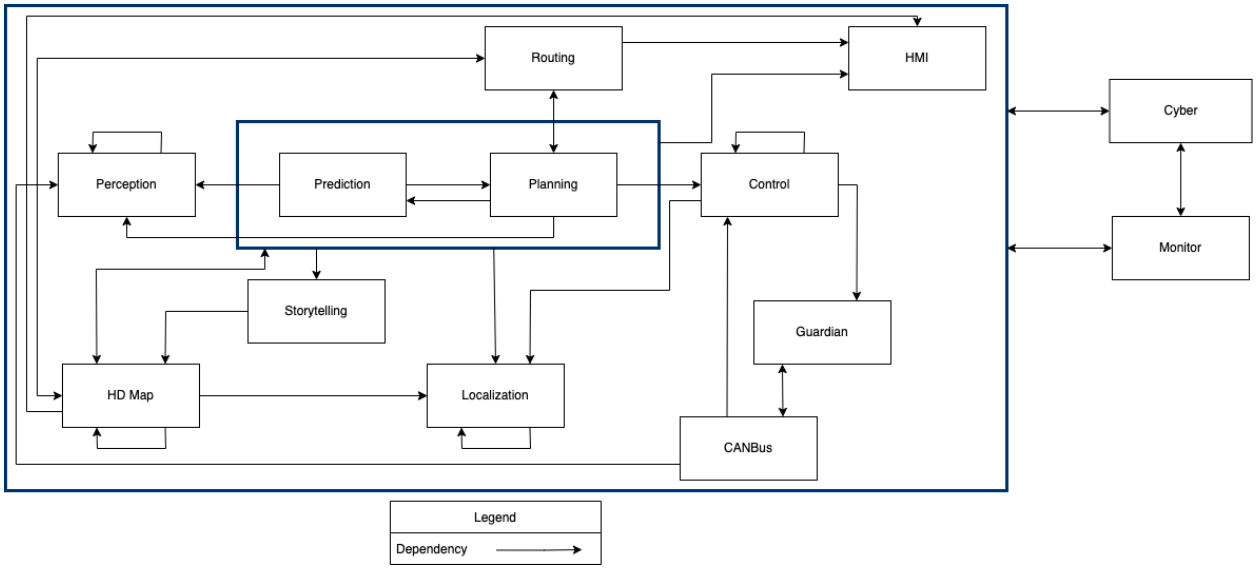


Figure 1: Concrete Architecture

Architectural Style

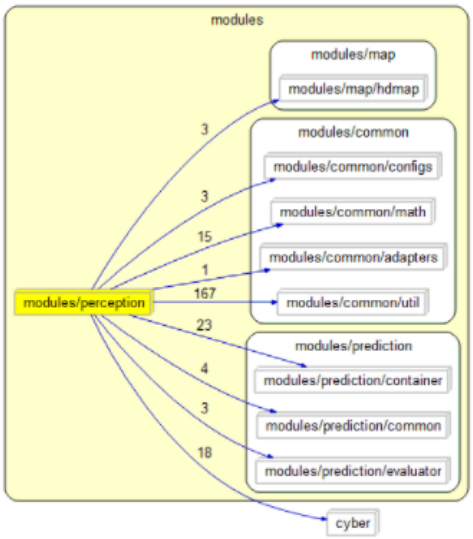
Apollo is built around a publish-subscribe style. In Figure 1 , the data lines represent published data and the control lines represent the published topics. Based on the communication between these components and the intent of Apollo being ideally used for multiple types of vehicles, publish-subscribe is crucial for the system. The architectural style of publish-subscribe most importantly allows the system to react quickly to its environment. With autonomous driving there are multiple unexpected situations that can occur while driving, such as an obstruction on the road or even a car swerving into you and at that moment the system needs to be readily available to manage it, which a publish-subscribe style easily allows.

High-Level Concrete Subsystems

The following offers a high-level, brief overview of the functionality carried out by the subsystems that comprise Apollo. (Note: → Blue line indicates one-way dependency relation for all the diagrams)

Perception

Perception is responsible for interpreting the data gathered by the onboard vehicle sensors, as well as understanding this in relation to map data. Primary sensors interpreted by Perception include cameras, radars, and LiDAR (Light Detection and Ranging) sensors. The perception subsystem then uses its interpretation of the data to form a model of the environment and surroundings in relation to the vehicle. This also involves determining other vehicles on the road, the road itself, obstacle detection, and more.



Prediction

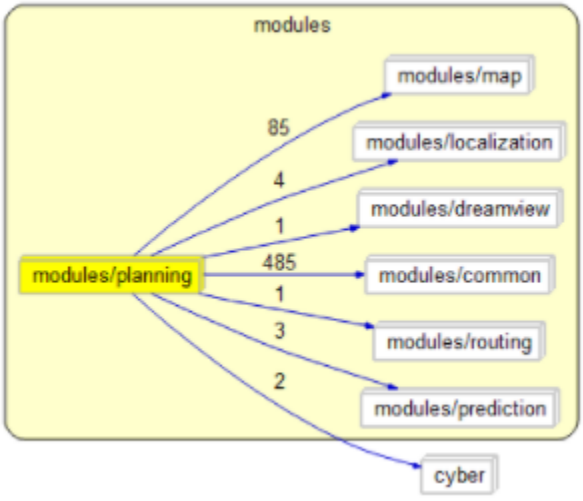
The prediction subsystem is responsible for predicting the behaviour of obstacles detected by the prior Perception subsystem. The prediction subsystem calculates predicted obstacle trajectory, by using and contextually interpreting data such as positions, velocities, and acceleration, all in context of the map that is relevant to the vehicle's current location. This subsystem has 4 main functions - Container, Scenario, Evaluator, and Predictor, which work in tandem to predict and generate a future trajectory path of detected obstacles.

Routing

The routing subsystem is responsible for generating high-level routing navigation information about a particular chosen destination. In order to determine this, the subsystem uses map data as well as requests for routing data such as the start and end points of computed paths between lanes or roads.

CANBus

The CANBus subsystem is responsible for passing control commands to the vehicle's hardware in order to execute a given control command (i.e. brake, accelerate, steer), as well as reading chassis information to the software system.



Planning

The planning subsystem is responsible for computing a collision-free, comfortable driving trajectory for the control subsystem to execute. In order to achieve this, planning primarily uses output generated by the localization, prediction, map, and routing subsystems in order to compute a spatio-temporally mapped, feasible, and planned trajectory.

Control

The control subsystem is responsible for controlling the vehicle and creating a comfortable autonomous driving experience.

This is achieved using a variety of control algorithms as well as interpreting localization data and current vehicle status. Depending on Control's interpretation of the input data, the subsystem passes contextually appropriate instructions regarding steering, brakes, acceleration and more to the CANBus subsystem for execution.

Map
The Map subsystem acts as a query engine support to provide information about the roads to the system. This subsystem consists of a bundle of maps, which include road, lane geometry, labels, routing and lane topology, and more map information.

Localization
The localization subsystem deals with computing the answer to the question of "where am I" for the autonomous system. This is determined by using a combination of sensors and data in order to answer this question and provide the system with improved accuracy and precision with regards to the system's understanding of its own location.

Dreamview
The Dreamview subsystem is a web application intended to be viewed and operated by the user. It provides the user with a user-interface that visualizes, in real-time, the current output of main autonomous driving modules like planning trajectory, localization, vehicle location on map, and chassis/vehicle status. Dreamview also provides additional functionality like viewing hardware status, enabling/disabling modules, starting the autonomous vehicle, and debug/issue logging.

Monitor
The Monitor subsystem is responsible for surveilling the operation of all subsystems in the autonomous system. Monitor watches the status of autonomous system hardware for hardware failure, and reports an alert to Guardian in such an event. Monitor also watches for failures in other subsystems, and similarly will report to Guardian in the event that any such failure is detected.

Guardian
The Guardian subsystem is responsible for performing decision taking safety actions based on data from the common subsystem. In the case that all subsystems are functioning normally, Guardian allows the flow of control to work normally, whereby Control signals are sent directly to CANBus. However, in the event that a subsystem crash is detected by Monitor and reported to Guardian, the Guardian subsystem will block Control signals from being sent to CANBus. After this, it will then decide on one of three ways to stop the vehicle based on certain data.

Storytelling
The Storytelling subsystem is responsible for dealing with coordinating synchronous and cross-module actions. Complex driving scenarios may require the involvement of different modules in order for a particular scenario to be successfully executed - this is where the Storytelling module comes in. The Storytelling subsystem also allows for fine-tuning of specific driving scenarios to specifically target and improve the driving experience.

Cyber

The Cyber subsystem is the open-source, high performance runtime framework that all the other subsystems use and run on. The Cyber subsystem is designed with a centralized and parallel computing model in mind, targeting and affording the Apollo platform high concurrency, low latency and high throughput, in order to meet the high performance requirements of autonomous driving.

Subsystem Conceptual Architecture: CANBus

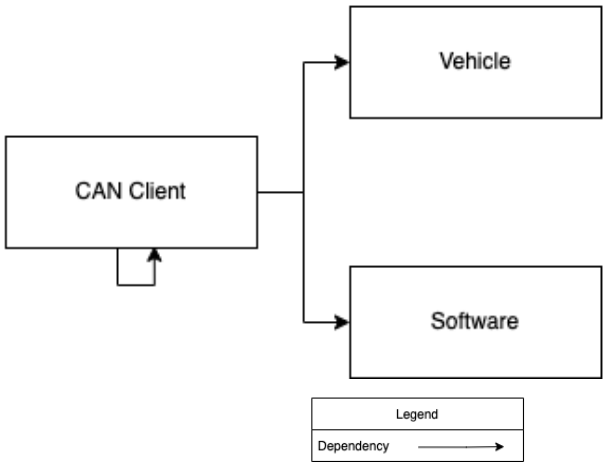


Figure 2: Conceptual architecture of CANBus Subsystem

Here we see a client-server style with the CAN Client providing/retrieving data from the vehicle and software portions of Apollo. Client-server style was chosen as it is used for applications that involve distributed data and processing across a wide range of components. In the specific case of Apollo, the CANBus system is responsible for sending information about the chassis to several different software components, including planning, HD map, and control. The CANBus system is used as the 'bridge' between the software and hardware components, so it is required to store and retrieve data whenever requested by either side. In Figure 2 one can see the concrete architecture of the CANBus system, and its components whose details are listed below.

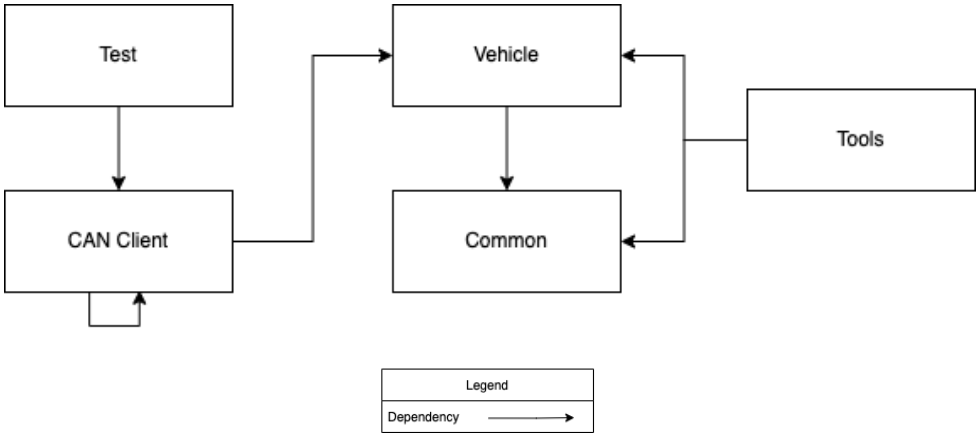


Figure 3: Concrete architecture of CANBus Subsystem

Test
The test component holds tests which check if the CANBus system is working properly

CAN Client
The CAN Client is the central module in this system, as it is the component which sends driving instructions to control, as well as outputting chassis status to various software components.

Vehicle
The vehicle component is the physical vehicle with its controller and message manager. The CAN Client will send control commands to the car for driving.

Common
The common component holds various flags and initialization data for CANBus. It defines functions such as chassis message publishing, how to interpret received commands, location of test files, and more.

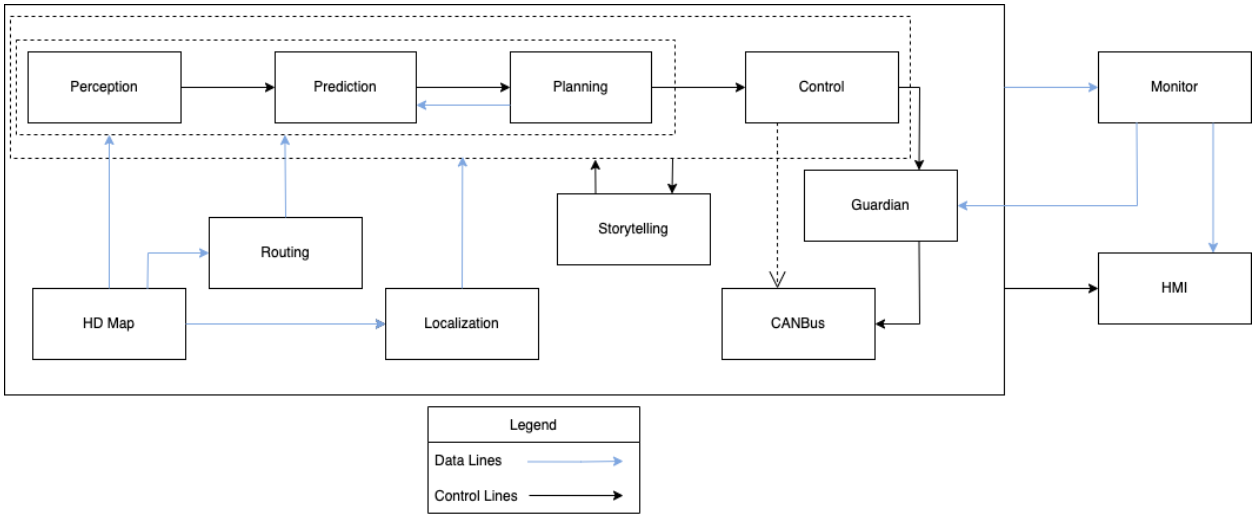
Tools
The tools component is mainly responsible for defining driving motions such as throttle, acceleration, hand braking, and gear shifting.

Reflexion Analysis

High Level Reflexion Analysis

This Reflexion Analysis compares the differences between our conceptual architecture, the concrete architecture, any missing/new modules, and associated rationales for this.

Here is the diagram for our Conceptual Architecture that was produced earlier:



Note: *Common and Cyber are subsystems which every subsystem is dependent on. They are excluded from the below discussion unless relevant. More information regarding Common and Cyber is in the* ***Missing & New Subsystems*** *section.*

Perception → Prediction

In our Conceptual Architecture, Perception is dependent on the data from HD Map, Routing, and Localization. Whilst the dependency of Map is also true in the Concrete Architecture, there are no dependencies on Routing or Localization. There is also another discrepancy - a dependency which is missing. In the Concrete Architecture, Perception is also dependent on Prediction. The rationale for this is that the definition of perception of a vehicle's surroundings in the Apollo system includes the prediction of any obstacle behaviour. Thus, the perception of the vehicle's current surroundings includes not only the environment around it and any detected obstacles, but the predicted behaviour of those detected obstacles too.

Prediction → Map
In our Conceptual Architecture, Prediction is dependent on the Perception subsystem, as well as HD Map, Routing, and Localization, and takes data input from the Planning subsystem. In the Concrete Architecture, Prediction is only dependent on Map. This is a discrepancy that is present within our conceptual architecture, as well as even on Apollo's own documentation for the Prediction subsystem, which cites "Obstacles information from the **perception module**, Localization information from the **localization module**, Planning trajectory of the previous computing cycle from the **planning module**" as inputs for the Prediction subsystem, which lines up with our Conceptual Architecture. This is because Prediction communicates with the other subsystems through the Common subsystem.

Routing → Map
In our Conceptual Architecture, Routing is dependent upon HD Map. This is also the case in the Concrete Architecture.

Planning
In our Conceptual Architecture, Planning is dependent on Localization, Perception, HD Map, and Routing. The dependencies identified with Localization, Map, and Routing are also present in the Concrete Architecture, however there are some discrepancies; Perception is not a dependency that is present in the Concrete Architecture. Additionally, the Planning subsystem also has dependencies on Dreamview (HMI), and Prediction. Apollo's own documentation for the Planning subsystem says that it takes "Localization, Perception, Prediction, HD Map (in modules/map/data), routing, task_manager." as input, matching up with our Conceptual Architecture, however Perception was not a present dependency in the Concrete Architecture. The dependency on Dreamview is also unexpected as it is neither presented in Apollo's own documentation nor our Conceptual Architecture. A rationale for these divergences may be that in order for Dreamview to carry out its graphical user-inclined representations of subsystem data, it needs to cross-communicate with the Planning subsystem, or perhaps even take user inputs through the Dreamview module which may affect operation of the Planning subsystem.

Control
In our Conceptual Architecture, Control is dependent on Localization and Planning. The Planning dependency is unexpectedly not present in the Concrete Architecture, despite being mentioned as input in the documentation, which was the rationale for including Planning as a dependency. Communication with these subsystems goes through the Common subsystem in the Concrete Architecture.

CANBus → Guardian
In our Conceptual Architecture, CANBus is dependant on Guardian, however this dependency does not exist in the Concrete Architecture. This is because Control commands do not go through the Guardian to the Control subsystem, but rather go directly to the Control subsystem, and then the Guardian subsystem may intervene in the scenario that it detects something wrong/feels the need to intervene, depending on the status of other subsystems & input data.

Map
Map isn't dependent on any subsystems in our Conceptual Architecture, however this is not the case in the Concrete Architecture, where Map is dependent on Planning and Routing subsystems. This is because the Map subsystem has a submodule called "Relative Map" which uses these dependencies to behave as a middle layer between modules in order to generate a real-time relative map, used for navigation.

Localization
In our Conceptual Architecture, Localization is not dependent on any subsystems. This is also correctly the case with the Concrete Architecture.

Dreamview/HMI
In our Conceptual Architecture, HMI is dependent on the output of all subsystems, however in the Concrete Architecture it is only dependent on Map and Common. This is in contradiction to the input described on the Dreamview documentation, which lists Localization, Planning, Monitor, Perception, Prediction, and Routing as inputs. A rationale for the missing dependencies is that communication with these subsystems must have gone through Common instead (Common is expanded upon later in the Missing & New Subsystems section).

Monitor
Monitor is dependent on the output of all subsystems minus HMI. In the Concrete Architecture however, it is only dependent on Dreamview. There is cross-communication between Dreamview and Monitor regarding the reporting of hardware and software status, and system health monitoring. The rest of the communication needed by Monitor is accomplished through Common.

Guardian
Guardian is dependent on Control and Monitor in our Conceptual Architecture. In the Concrete Architecture, Guardian is only dependent on Dreamview. This dependency is used to take in user inputs through the user-facing Dreamview app. Guardian communicates with other subsystems through Common.

Storytelling
In the Conceptual Architecture, Storytelling is not dependent on any subsystems. In the Concrete Architecture however, Storytelling has a dependency on Map. Stories, which are scenarios that need to trigger multiple actions from multiple modules, need to be mapped to a map in order to be executed.

Missing & New Subsystems

Common is a subsystem that **every** aforementioned subsystem is dependent on in the concrete architecture. Similarly, Cyber is also a subsystem that every subsystem is dependent on in the concrete architecture. Both of these subsystems were missing from our Conceptual Architecture. Common is used for some common shared functionalities between subsystems - code which is not specific to any particular subsystem. The subsystem also allows for subsystem communication through adapters and topics. Common was not explicitly referred to in the Apollo documentation which is why it was missed in the Conceptual Architecture.

Cyber on the other hand represents the open-source "Apollo Cyber RT" runtime framework that all of the subsystems run on. It is required for the integration and execution of all the other subsystems in the Apollo system. The rationale for Cyber's exclusion from the Conceptual Architecture was also similarly not referred to in the Apollo documentation, as well as the fact that a runtime framework could be interpreted as an external system in and of itself.

Subsystem Reflexion

Our conceptual architecture for the CANBus subsystem was based on the implementation details found in the "apollo/modules/canbus/README.md" file. Using the program Understand, we are able to extract its genuine interactions with other modules, thus providing the concrete architecture. There were unexpected changes when we viewed the conceptual and concrete architectures side-by-side. The differences are discussed below.

Test

The Test component is present so that the Canbus system can be checked if it's working properly. It is rational for this component since it is imperative that the subsystem that interacts with the control commands and the vehicle itself is constantly working as expected.

Common

This component is more or less embodied by our Software component in the conceptual architecture. It is rational for it to be its own component however, since a Software component would embody too many things that each deserve to be a sub-subsystem in and of itself.

Tools

Going off of the aforementioned component, this too was something that we categorized in the Software component. As also said before, since the Tools component deals with important driving motions, it too should be dealt with as an individual entity.

CAN Client -> Software

As explained before, the Common and Tools components were embodied by the Software component and the CAN Client also depended on the Software component in the conceptual architecture. But with our concrete architecture, we find that the CAN Client has no dependencies. This makes sense because the CAN Client itself embodies the software that sends driving instructions, chassis status, etc. It does not call on another subsystem to do that work.

Test -> CAN Client

The Test component depends on the CAN Client since that is the subsystem that it monitors.

Vehicle -> Common
This dependency is needed to allow the controller to create and distribute messages for other modules to subscribe to.

Tools -> Vehicle
This dependency is needed to allow cooperation between the vehicle and the code instructions of the throttle, acceleration, gear shifting, etc.

Tools -> Common
This dependency is needed since the Tools component relies on test files, which are defined in "apollo/modules/canbus/common/canbus_gflags.h".

Diagrams

1. Apollo analyzes component inputs and determines a rerouting is necessary

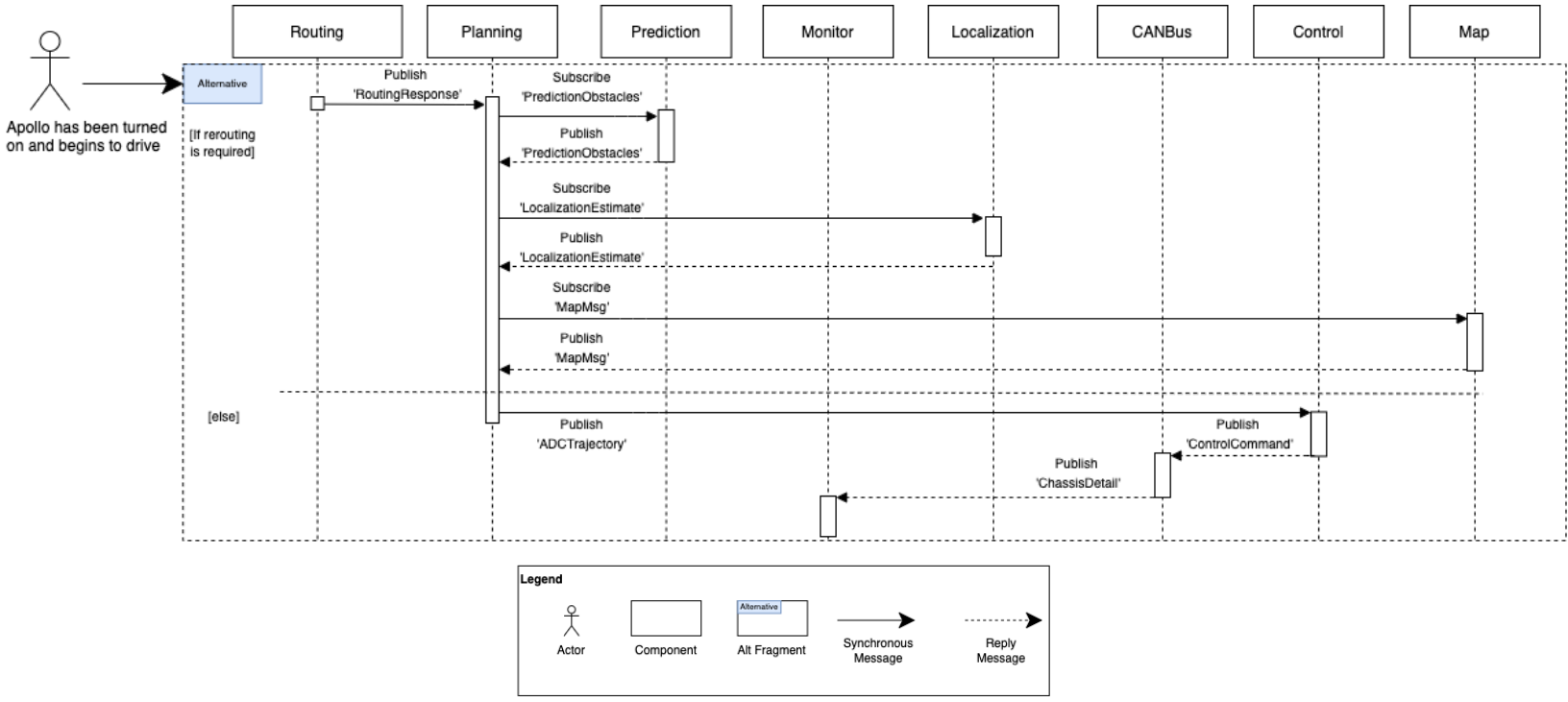


Figure 5: Sequence Diagram of Use Case 1

The first use case assumes that Apollo has been started through Dreamview as it focuses on Apollo's reaction to requesting a reroute. First, the initial route is determined and 'RoutingResponse' is published for Planning to receive. Planning then subscribes to several messages from Prediction, Localization, and Map so that it may create a driving plan. Planning is continuously taking in new data about its surroundings from various sensors, and in this use case, it has now determined the route is no longer valid. Before a trajectory is sent to Control to develop the physical vehicle commands, the process will now restart. A new route is generated and sent to planning through the RoutingResponse message. Planning now takes updated outputs

of messages such as PredictionObstacles, MapMsg, and LocalizationEstimate which it will use to calculate a new trajectory. If it is deemed successful, this trajectory will be sent to Control which will publish the car commands for CANBus to use. Finally, CANBus sends the ChassisDetail message to Monitor, which will then send a message to display the result on dreamview (not pictured).

2. A person inputs a destination and Apollo takes them there

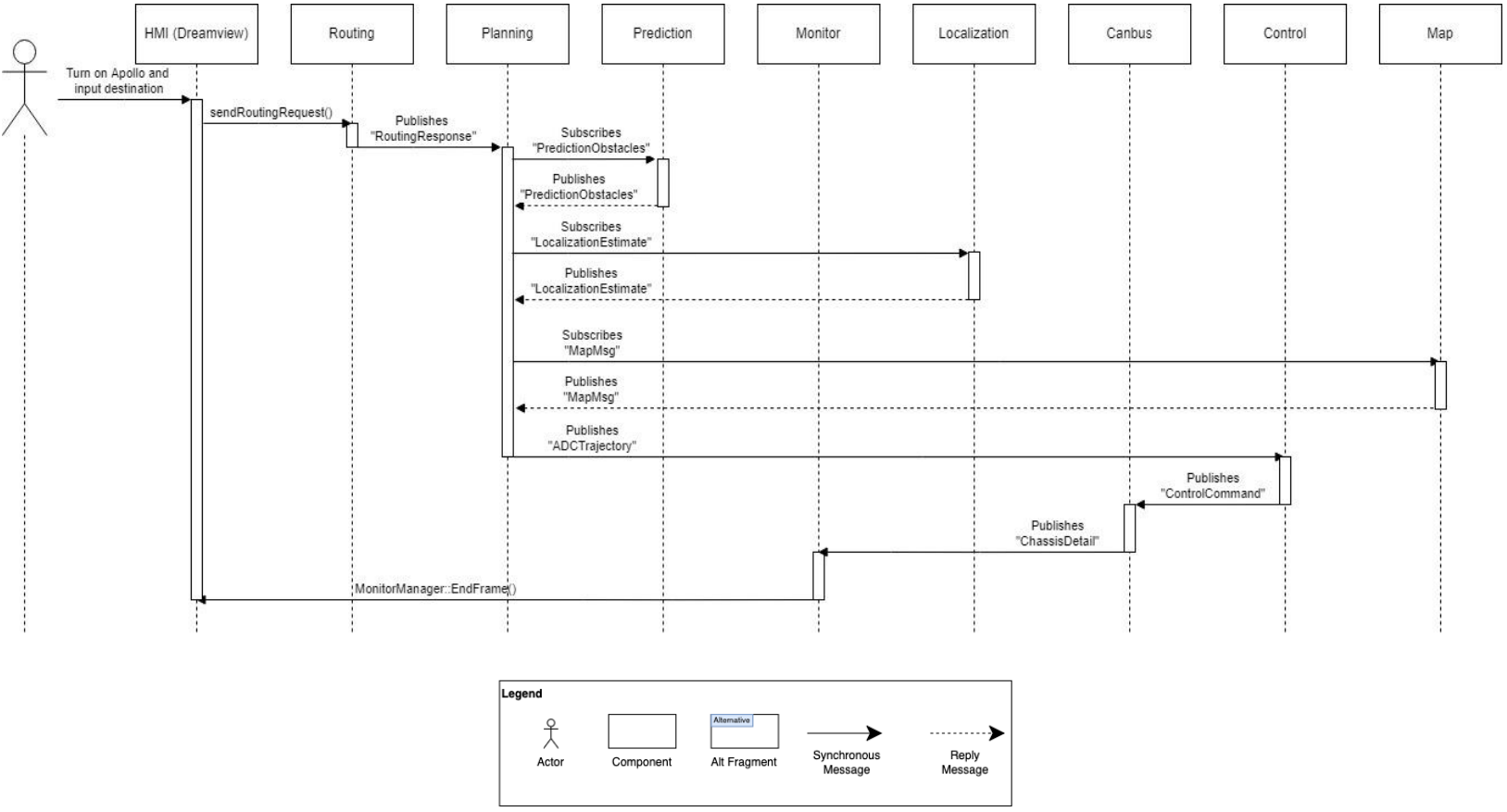


Figure 6: Use Case 2: A person inputs a destination and Apollo takes them there

This second use case describes the process of a user inputting a destination and being driven to it by Apollo. It starts with the user turning on Apollo and using Dreamview to input a route. Once the button is pressed, the method “sendRoutingRequest()” is triggered which prompts the Routing module to compute

the routing request. It is then computed and incorporated in a message published by the "RoutingComponent::Proc()" method. The Planning module subscribes to Routing, gets that routing request, and then subscribes to the Prediction, Localization, and Map modules. The modules then use their own processing methods to publish "PredictionObstacles", "LocalizationEstimate", and "MapMsg" messages, respectively. Now the planning trajectory is computed, and the Planning module publishes the "ADCTrajectory" message that is subscribed to by the Control module. The computation is done and wrapped in the message "ControlCommand", which is published and then listened to by Canbus, where the commands are executed. Details are gathered, and then the "ChassisDetail" message is published for the Monitor module to listen to. After executing the "MonitorManager::EndFrame()" method, which publishes all monitor logs, Dreamview takes that input and displays it to the user via the screen.

Naming Conventions

IMU: An Inertial Measurement Unit - device used to measure force, angular rate, velocity, inertia of an autonomous vehicle

LiDAR: Light Detection and Ranging sensor technology, used to determine objects and pedestrians, cyclists, animals, and other vehicles

CANBus: Microcontroller communication standard developed for vehicle electronics communication

HMI: Human Machine Interaction

Conclusion

Using Apollo's source code to visualize the dependencies in Understand, as well as the given visualization, we were able to analyze the concrete architecture of Apollo. The team was able to find unexpected new dependencies as well as discovering and adding a new subsystem called "Cyber" into our concrete architecture. With each new dependency we were able to find reasoning behind the connection through our old conceptual architecture, as well as the source code and through using Understand. Furthermore, we were able to determine that CANBus has a client-server style architecture style with the help of Understand we were also able to perform a reflexion analysis on its architectures too. Although Understand was a great tool to use, we had a few issues of readability as well as trying to find concise information to support the reasoning behind each new dependency in our concrete architecture

Lessons Learned

As we came up with the concrete architecture of Apollo, we faced some issues. Although Understand provided us with a lot of support with regards to seeing the dependency of each of the modules, we found that a lot of these modules were dependent on each other therefore making the architecture readable was fairly challenging. Another challenge was producing concise information. The reflexion analysis showed us a lot of unexpected dependencies in the architecture so we had to choose the dependencies to talk about in order to fit it in this 15 page report. There was an overwhelming amount of information so even keeping track of it was fairly challenging. This also proved to be a challenge when producing the use case diagram as there was a lot of interdependence between the modules; even finding the methods associated with them was confusing. The pub sub dependency graph given by the professor was a major help here as it gave us reference to what was published and subscribed.

On the positive side, Understand really gave an extensive understanding of the source code of Apollo's architecture and its subsystems. Going through it helped us gain a much deeper understanding of its architecture and we saw the usefulness of its process. With the help of the detailed documentation on Apollo's GitHub we were able to perform the reflexion analysis pretty well and also gain a lot of knowledge from it. With everything going in person, the team was also able to meet in person and we were able to have some productive discussions about the project.

References

[1] Apollo Module Breakdown: https://github.com/ApolloAuto/apollo/tree/master/modules

[2] PubSub Dependency graph:
https://onq.queensu.ca/d2l/le/content/642417/viewContent/3865686/View

[3] Previous year projects:
https://research.cs.queensu.ca/home/ahmed/home/teaching/CISC322/F18/index.html

[4] Apollo Understand Diagram:
https://docs.google.com/document/d/1qcHmRh1gAGTZMorCl1ILjomHqvamk6pjCgH2GdOogFs/edit?usp=sharing

[5] Apollo Prediction Subsystem Documentation
https://github.com/ApolloAuto/apollo/tree/master/modules/prediction

[6] Apollo Planning Subsystem Documentation
https://github.com/ApolloAuto/apollo/tree/master/modules/planning

[7] Apollo Control Subsystem Documentation
https://github.com/ApolloAuto/apollo/tree/master/modules/control

[8] Apollo Relative Map Documentation
https://github.com/ApolloAuto/apollo/tree/master/modules/map/relative_map

[9] Apollo Dreamview Subsystem Documentation
https://github.com/ApolloAuto/apollo/tree/master/modules/dreamview