CISC 322/326 - Software Architecture
Assignment #1: Report

Conceptual 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:

With the growing popularity of autonomous driving, Apollo was examined to determine its conceptual architecture. After reading its documentation online and analyzing reference architecture for other autonomous driving platforms, Apollo's conceptual architecture was determined to have a process-control architectural style along with a publish-subscribe style. With these styles, more research was completed about each of the components in the system. Using the proposed conceptual architecture, the situation of an autonomous car using Apollo avoiding an obstacle and the task of a person inputting a destination into a car and using Apollo to take them there was investigated in more detail. To gain a better understanding of Apollo, the concurrency of the system was further examined as well as how the system could be modified in the future. Through the examination of Apollo, our team was able to gain more knowledge about autonomous vehicles as well as overcome any issues that arose during our analysis.

Introduction and Overview:

As the concept of autonomous vehicles grows in popularity, the technology is developing at a rapid rate to the point where basic driving autonomy already exists in motor vehicles. As this technology matures, many car companies in the world such as BMW, Audi, Tesla and Volkswagen already have active development projects in this field. In order to achieve autonomous vehicles, projects like Apollo exist where they focus on the software side of autonomous driving. The Apollo project is developed by Baidu and Kinglong. Baidu is one of China's multinational technology companies that specializes in Internet related services and Artificial Intelligence, and Kinglong is a Chinese bus manufacturer.

Apollo's history began back in 2017 with Apollo 1.0 where it was referred to as the Automatic GPS Waypoint Following. This version was only applicable in an enclosed area such as a test track or parking lot and the project allowed other companies to develop their own self-driving systems based on Apollo's software. Apollo 1.5 focused on the development of fixed lane cruising. With the addition of LiDAR to the autonomous vehicles, Apollo now had improved perception of its surroundings, helping the software have a better idea of the vehicles positioning on the road meaning better generated trajectory paths and safer maneuvering between lanes. In Apollo 2.0, vehicles were enabled to drive autonomously on simple urban roads. Some features included the ability to safely cruise on roads, avoid obstacles, change lanes and stop at traffic lights. Apollo 2.5 introduced their software to geo-fenced highways as well as a camera for obstacle detection, with the addition of maintaining lane control, cruise and avoiding crashing with vehicles ahead. In the Apollo 3.0 update, the main focus was to give developers the ability to build upon low-speed environments. Apollo 3.5 was able to navigate through more complex obstacles, not limited to residential and downtown areas. This update brought in 360 degree vision for the autonomous vehicle, along with improvements to the perception module when dealing with changing urban environments. In Apollo 5.0, they focused on supporting volume production for Geo-fenced Autonomous driving. Apollo 5.5 included further improvements in complex urban scenarios by adding curb to curb driving support. Apollo 6.0 incorporated new deep learning models to further improve some of the existing modules in its architecture. This update also implemented data pipelines services to improve developers' experiences. In the most recent version of Apollo 7.0, 3 brand new deep learning modules were added to improve upon the perception and prediction module.

This report covers Apollo 7.0's all of its architecture. It will begin with the overview of the conceptual architecture of how the concrete architecture for Apollo is implemented. Following will go over the architecture style and the components of Apollo. Then the report will discuss the interactions between the components. To finish off the report, future modifiability support, concurrency and use cases will be touched upon.

Conceptual Architecture:

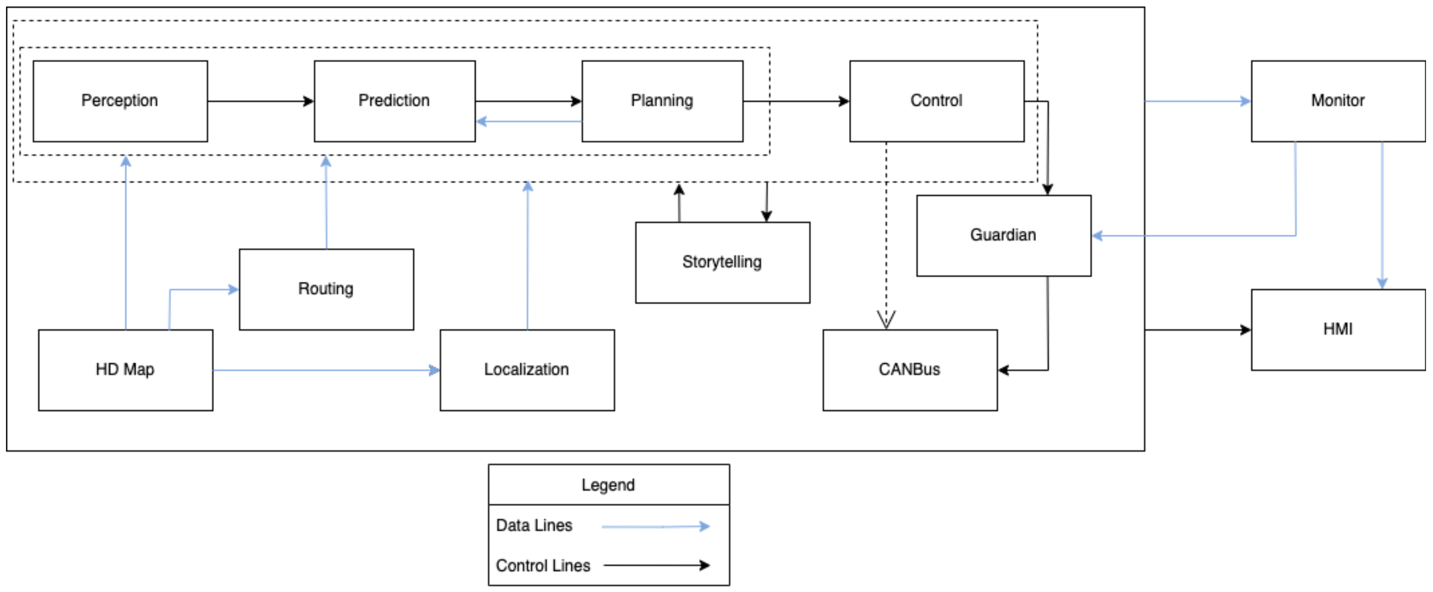


Fig 1: Conceptual Architecture

Architecture Style

The conceptual architecture of Apollo has a process control style. This allows for multiple process variables to be manipulated in the system through a controlled variable, which for autonomous driving is necessary as there are multiple variables that need to be taken into consideration for the system to choose the correct path the car should take. For example, stop signs, traffic lights, and obstacles are only a couple of the variables that need to be considered. With a process control style, the rectangles are considered components, where each component includes the process definition and the control algorithm for that process.

A publish-subscribe style is also used alongside the process control style for our conceptual architecture. This allows for the reuse of components into the system as it only needs to be registered for the specific events in the system. This is beneficial for autonomous driving, as a variable such as the speed limit could change and need to be re-introduced into the system, which this architecture style allows without changing the whole system. Additionally, there can be numerous variables added to the system allowing for better scalability.

Alternative Style

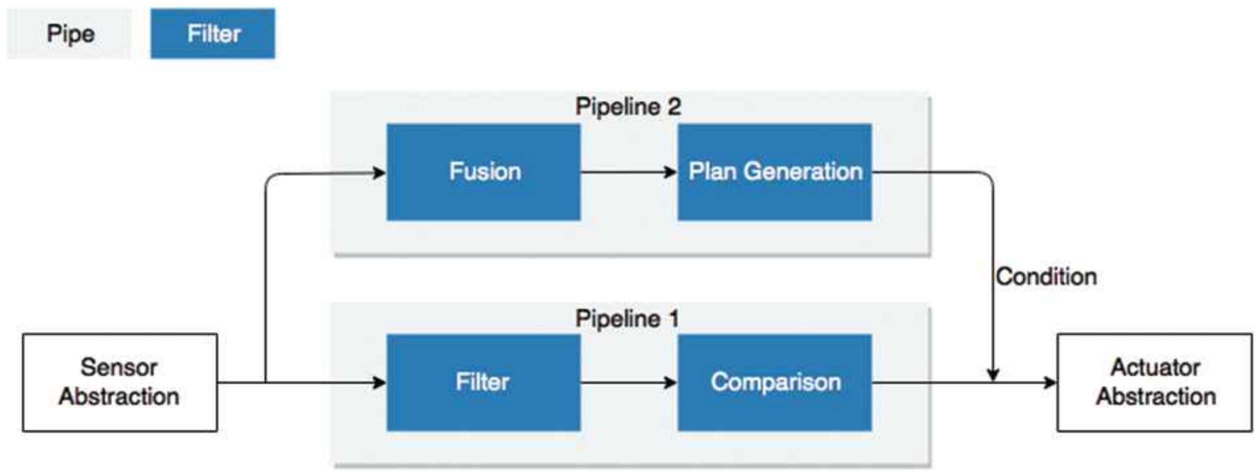


Fig 2: Alternative Architecture

From the reference architecture we know that this style for this system is sequential therefore an alternative architecture that can be used is the pipe and filter style. The different steps of the processes can be represented by the pipes where the output of a step will be the input of the process of the next step in the sequence. These processes are executed by the “filter”. This style does not allow for data to go backward so we cannot take the advantage of reusing the components.

Components

Perception

The perception module in Apollo interprets the data gathered by its sensors into high level concepts of the vehicle's current surroundings. These sensors would include the use of multiple cameras, radars on the front and rear, as well as LiDARs. Combining all three sensors, this module is able to fuse the individual data and determine what’s in front of the sensor. The obstacle sub-module is able to track, detect and classify obstacles. It is also able to predict the position and the motion of the obstacle such as it’s heading and velocity.

In Apollo 7.0 the camera-based obstacle detection model is based on Single-Stage Monocular 3D Object Detection via Keypoint Estimation, otherwise known as SMOKE. Based on a single image, the perception module is able to predict the 3D projected framework and its key points of a vehicle. The perception module borrows codes from CenterNet which in order to modify some parts of SMOKE. Using the 2D bounding box and additional 3D information, the perception module optimizes the predicted position of the obstacles using the geometric constraints. Then using the 3D information that was predicted by the model, it is able to calculate the 3D bounding box of the obstacle.

The perception module also relies on Lidar-based obstacle detection. The Lidar perception module receives messages from a lidar driver, which processes them with detection and tracking algorithms. In order to classify the lidar data into vehicles, pedestrians or obstacles, Apollo uses the PointPillars model as the detection algorithm. The tracking algorithm tracks the detected obstacle based on the detection algorithm's results. Once completed, the tracked obstacles are transferred to the fusion component in order to get fusion with obstacles from other cameras and radars on the vehicle.

Prediction

The prediction component is responsible for predicting each obstacle's behaviors found by the perception modules. With the provided data such as positions, velocities, accelerations and headings, the prediction module will then generate a predicted trajectory for every obstacle. This module has 4 main functions: Container, Scenario, Evaluator and Predictor. The container module stores the perception obstacles data, vehicle localization and vehicle planning data, all provided from the perception module. The scenario module currently has 2 defined scenarios which are Cruise and Junction. Cruise predicts lane keeping and following vehicles while Junction predicts traffic lights and Stop signs. The Evaluator module predicts the speed and the path for any obstacle by calculating a probability for it with the provided model and dataset. Some examples of evaluators would be a Cost evaluator which has its probability calculated with a set of cost functions as well as Junction Map evaluator which has its probability calculated using an semantic map-based CNN model during junction scenarios. The predictor module is the final module of the prediction component when it generates a predicted trajectory for objects. Some objects include and are not limited to Single lane, lane sequence, move sequence, free movement and Junction. The prediction component is responsible for generating and predicting the future trajectory path of any given object or obstacle that is given to it by the perception component.

Routing

The Routing component uses Map data as well as routing requests such as start and end points of a computed path between lanes or on roads and generates high level navigation information. Typically the starting points would be the location of the autonomous vehicle and the ending points would be the given request location. The output of the module would provide information of the routing navigation.

Planning

The planning module uses Localization, Perception, HD Map, and routing data in order to compute a collision-free and comfortable driving trajectory, for execution by the control module. The planning module takes a modular approach to deal with ever changing driving conditions, partitioning and labeling these differing driving conditions as individual use cases, in order to allow better scenario specificity. The planning module initially takes in prediction module’s output, subscribing to the traffic light detection data. After this the component takes in routing’s output, the planning module may or may not request a new alternate routing computation in the event that the current route cannot be feasibly followed. Thirdly, the planning module takes in the localization output in order to spatially map the vehicle’s current location. The output of the planning module is a spatio-temporally mapped, feasible, planned driving trajectory.

Control

The Control component uses generated trajectory input and the current autonomous vehicles status in order to create a comfortable autonomous driving experience using a variety of control algorithms. After the control module has generated a set of instructions, they are passed to the CANBus which in turn controls the throttle, brake and steering. The Model Reference Adaptive Control, also known as MRAC is an algorithm that is designed to effectively reduce the dynamic delay and time latency of the by-wire steering which ensures that the steering control is faster and more accurate for tracking the planning trajectory. The Control Profiling Service looks at road-tests or simulation data in order to make a quantitative analysis of the autonomous vehicles overall control performance. This profiling service provides insight on the current performance and improvements in the module. The control component takes in Planning trajectory, Car status, localization and Dreamview AUTO mode change request. The expected output are the Control commands such as steering, throttle and brakes to the chassis.

CANBus

The CANBus component is responsible for taking in control commands and passing them to the vehicle's hardware as well as chassis information to the software system. The major components in the CANBus modules are the autonomous vehicle itself and the CAN Client.

HD-Map

The HD-Map component acts as a query engine support to provide information about the roads. This module is a bundle of maps. There is the base map which has all the roads, lane geometry and labels, making it the most complete map. The routing map includes the topology of the lanes in the base map. The sim map acts as a lighter version of the base map for dreamview visualization, meaning reduced data and better performance. Apollo allows the functionality to use a different map.

Localization

The localization module deals with computing and answering the query of “where am I” for the autonomous system. The localization system uses 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 location. The module uses satellite location positional information from GPS, LiDAR sensor data to identify objects around the vehicle, and IMU sensor data in order to compute a multi-faceted and multi-dimensional positional understanding of the autonomous vehicle’s current location, in a spatio-temporal context.

HMI

The HMI (Human Machine Interface, or DreamView) module 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, and chassis/vehicle status. By providing this view, the user is able to see what exactly the autonomous system is privy to and what it perceives, which can help with the user's confidence in the system as they are able to gain insight into what the autonomous system can see and what it is planning to do. The HMI web app also provides additional functionality like viewing hardware status, enabling/disabling modules, starting the autonomous driving car, and debug tools to track & log module issues.

Monitor

The monitor module is the surveillance system of all modules operating in the autonomous system. Monitor will watch the status of autonomous system hardware for hardware failure, and report an alert to Guardian in such an event. Monitor also watches for failures in other modules, and similarly will report to Guardian in the event that any such failure is detected. The Guardian module is then able to take decisions based on any such failure events reported by the Monitor component. Additionally, Monitor receives the data from modules and also passes the data along to the Human Machine Interface (HMI/DreamView) module in order to help with the visualization and viewing of module data in the HMI web application.

Guardian

The guardian module focuses on performing decision taking safety actions based on data provided by the Monitor module. In the case that all modules are performing as expected and 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 module crash is detected by Monitor and reported to Guardian, the Guardian module 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. First scenario: If the ultrasonic sensor is running without issue and does not detect an obstacle, Guardian will bring the car to a slow stop. Second scenario: If the sensor is malfunctioning or not responding, Guardian will apply a hard brake and bring the vehicle to an immediate stop. Third scenario: If the Human Machine Interface informs driver of an impending crash and the driver does not intervene for 10 seconds, the Guardian module will apply a hard brake to bring the car to an immediate stop.

Storytelling

The storytelling module deals with coordinating synchronous and cross-module actions. Complex 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 component creates "stories" which are defined as scenarios that need to trigger multiple actions from multiple modules, and it creates these stories based on predefined rules, which can then be subscribed to by other modules like Planning and Control. The storytelling module also allows for specific scenarios to be fine-tuned with regards to their driving experience. This component is a global scenario manager that facilitates safe autonomous operation vehicle operation in varying complex scenarios.

Interaction

As we saw from the reference architecture given in the journal written on autonomous vehicles, the main components of the architecture include Perception, Decision Control and Vehicle Platform abstraction which maps to the Perception, Prediction and Planning in the Apollo's architecture. The Perception component uses its cameras to gather data about obstacles and positions. If any obstacles are detected, it sends the message to the Prediction module that also updates location status of the autonomous vehicle based on information it receives from the Localization module. It then predicts the possible trajectory of the vehicle based on the information received. Localization module receives information about road structure etc. from the HD-map which it uses to give the required information along to Prediction and Planning. The Prediction module also takes in information from the Planning module, which ensures a safe and collision free route, and this module also in turn takes in information from Prediction. The Planning module takes in any traffic light detection from the Prediction module along with information of the location and route in order to calculate the safe and obstacle free route. Storytelling is another module that gives information of the different scenarios to the Planning module in order to execute the routes for the vehicle. Storytelling forms these scenarios by taking in information from the Localization ad HD-map. It might even find an alternative route depending on whether it perceives the current route to be safe. This information is then given to the Control module that uses its algorithms to execute the driving route planned by further passing this information to the CANBus. The CANBus sends the message to the hardware of the vehicle to perform the action.

The Human-Machine Interface (HMI) module and Monitor provide more external support. The Monitor takes in data from all the components mentioned above and passes the information to the HMI to convey the performance of the modules, i.e. if they're functioning properly. In case of any component failures, the Monitor sends a signal to the Guardian which intervenes by using different ways to stop the car 1) It checks the Ultrasonic sensors and decides whether to bring the car to a slow or immediate stop depending on if the sensors are working fine or not, respectively 2) In the special case of the HMI informing the driver of a foreseeable crash and the driver does not respond, the Guardian performs a hard brake to the car to stop it. As mentioned above, the HMI is used to communicate with the driver by displaying information from the Monitor. This data is displayed to the user so they can take the proper action needed in case of an issue.

Future Modifiability Support

The Apollo system structure is laid out and operates in a modular fashion. As described and outlined by the Software Architecture document, Apollo's architecture consists of 12 core software modules that comprise the system - Perception, Prediction, Routing, Planning, Control, CanBus, HD-Map, Localization, HMI, Monitor, Guardian and Storytelling. Due to the modular nature of the system, individual modules partition the various functionality of the Apollo system, allowing for clearer separation between code and their functional intention. This modularity is also helpful for future modifiability support, as code dealing with a particular subsystem within

Apollo can be easily updated and improved, and other modules do not necessarily need to be updated if another module is changed, so long as the output of the previous module is still within expected input parameters and data type.

Additionally, the planning module's modularity is quite scenario specific. This system is even more so subject to updating as new and more focused driving scenarios are created and fine-tuned. Within this module, each driving use case is dealt with as a separate, different driving scenario - this allows for better modifiability, fine-tuning and specific issue fixes for a particular driving scenario in the future, rather than applying future modifications to all driving scenarios which may or may not break existing, working scenarios.

Concurrency

Due to the real-time nature of the Apollo system and its constant need to monitor and react to live input & output data, concurrency is heavily present in the system. Input is constantly read from various input sensors like LiDAR and GPS, whilst Perception, Prediction, Planning, Localization, Control modules and more all interact with each other & work in tandem to perceive, predict, plan, localize, and control the autonomous vehicle. Many of the actions within these modules occur concurrently in order to achieve a particular objective at the same time. For example; the Storytelling module in particular is a global scenario manager that helps coordinate cross-module actions which involve complex scenarios. Complex scenarios may require the involvement and simultaneous concurrent execution by multiple different modules in order to appropriately and safely achieve a proper maneuver. This module was created specifically to avoid a sequential style approach, and allow for a concurrent approach which is more suitable for the Apollo system.

Use Cases

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

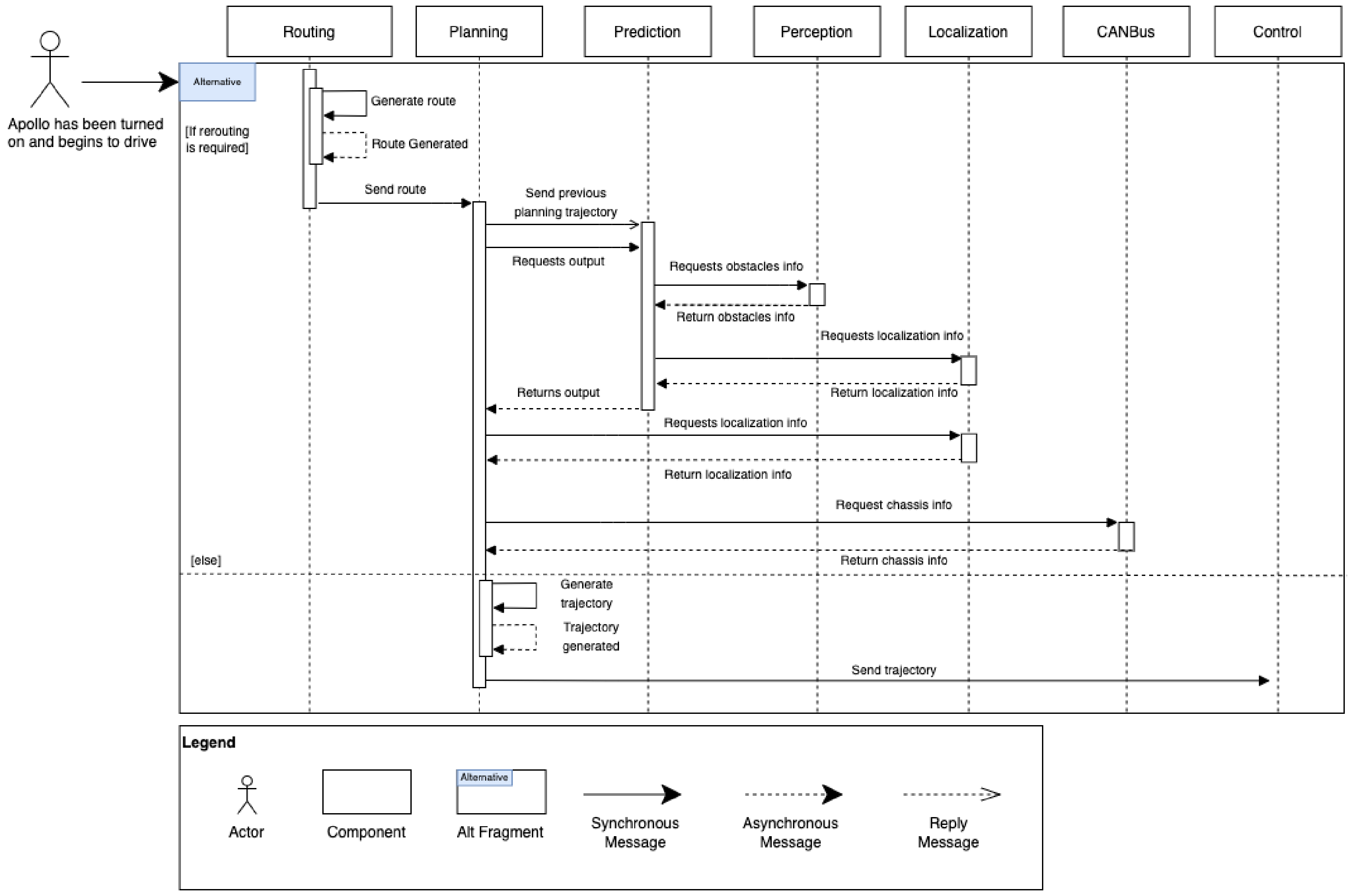
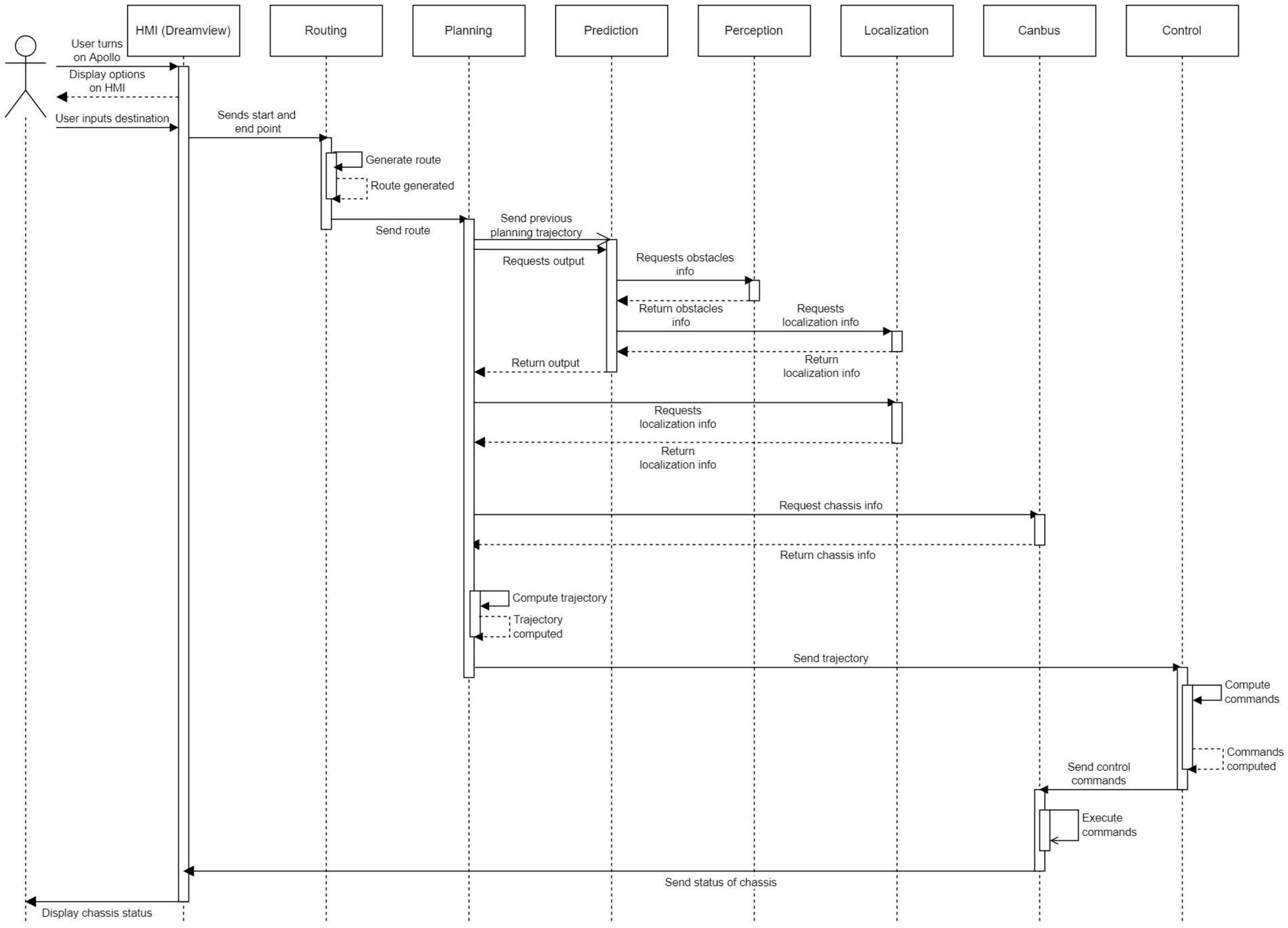


Fig 2: Use case 1: Apollo analyzes component inputs and determines a rerouting is necessary

The first use case assumes that Apollo has started to drive as it focuses on Apollo's reaction to requesting a reroute. First, the initial route is generated in the routing component and sent to the planning component. The planning component requires several input sources to determine a possible trajectory, so it obtains the output of the localization, prediction, CANBus, and routing components. It is at this point where the Alt Fragment in the sequence diagram is invoked. If Apollo computes that it needs to change the current route because of data from other components, it will request that the routing component calculate a new route. Apollo then once again uses this new route in combination with several other inputs to compute a possible trajectory. If the trajectory is valid, then Apollo will send it to the control component which takes care of the physical instructions for the car.

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



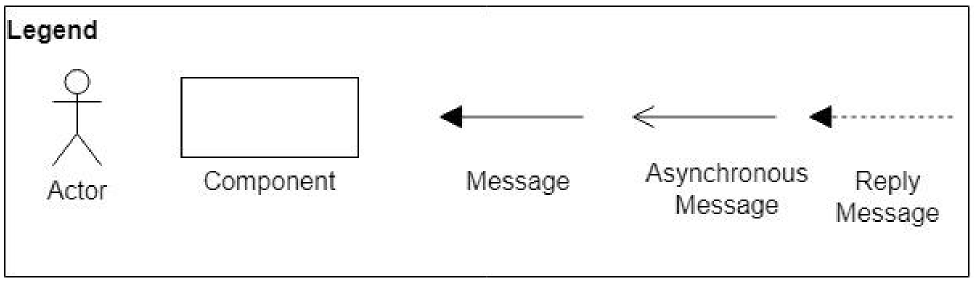


Fig 3: Use case 2: Apollo analyzes component inputs and determines a rerouting is necessary

This second use case displays the process of the user turning on Apollo, inputting a destination, and taking off. It starts off by the user turning on the system via Dreamview. After inputting the desired destination, the Routing module computes the routing navigation information and sends it to the Planning module. The Planning module needs data from multiple different modules to perform its intended task, so it consults the Prediction, Localization, and CANBus modules. The Localization and CANBus modules send back the required information, but the Prediction module requires the planning trajectory from the previous computing cycle. Once obtained, the Prediction module then consults the Perception module for obstacle information, and the Localization module for information. After the Planning module gathers all required input, it can then compute the trajectory and send it to the Control module so it can put the plan into action. The Control module computes the commands (including steering, throttle, brake, etc.) that the CANBus will use to actually execute. After execution, detailed information regarding the chassis' status is made available for the user to view.

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

Conclusions:

In summary, our conceptual architecture for Apollo properly captures the inner workings of how the fundamental components of the software work together to make the product come to life.

As explained in the beginning of the report, Apollo utilizes the process control style for its architecture. This is needed for autonomous driving projects as the idea of having multiple process variables (stop signs, traffic lights, etc) deals directly with the fundamentals of how self driving cars operate. A publish-subscribe style is also used in different components for efficiency.

The system does not largely rely on one single component. All components each play their own important role. Components such as Prediction and Perception deal with the car's surroundings. The Planning, Localization, HD-Map, and Routing components deal with location, whether that be the car's location or the destination of interest. The Control and CANBus components deal with the car itself, generating and sending commands viable for execution. The Monitor component keeps up to date with the state of the system, which includes monitoring the system, and should a failure occur, the Guardian component takes care of it. To make sure of a safe driving experience, the Storytelling component handles scenarios that could result in danger.

Finally, the HMI component (also known as Dreamview) ties it all together by providing an interface that allows the user to interact with the system.

In the future, we will dive into the concrete architecture of Apollo, as well bring forth some enhancements we think can improve the system.

Lessons Learned

As we went through the journal and documentation available for Apollo's architecture, we learnt that the main components of the software architecture are more or less consistent for most variations for this kind of system therefore it was not very difficult to map the reference architecture shown in the journal to the architecture of Apollo. One of the major differences mentioned by the journal and Apollo's architecture is the separation of the Semantic Understanding, World Modeling and Vehicle Platform Extraction. These components might be available in Apollo's architecture but are not explicitly mentioned. Apollo's Github provided very extensive information about its components and that helped us understand their roles in the process of vehicle maneuvering. In order to achieve a more high level understanding of the architecture and its components we learnt to balance our internal and external research, come together as a team and have discussions about our findings. The journal given for the reference architecture was a little out of date (published in 2015) so it was difficult to determine if any of the ideas mentioned were used in any of the current architecture. The amount of information received from resources were very consistent but it was hard to really compare this kind of architecture to a different style as there were not many variations. Lastly, our project is supposed to be specialized for Apollo v7 but not a lot of detailed information was available to understand how it created changes in the architecture, therefore we had to draw conclusions from the previous versions of Apollo.

Fortunately, we were able to overcome these limitations and gain a lot of knowledge about autonomous vehicles from the resources available too. Working together as a team was initially challenging too as we were unsure about how to work virtually due to the COVID-19 pandemic. Moreover, we even found it challenging to figure out how to divide the work but after having a couple of meetings we were able to have a good flow. As a group of 6 members we were able to provide support for each other in more ways than one which resulted in a good work dynamic.

References:

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

[2] Reference Architecture:
https://onq.queensu.ca/d2l/le/content/642417/viewContent/3814366/View

[3] Apollo Software Architecture:
https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_5.5_Software_Architectur
e.md

[4] Autonomous Vehicles Software Architecture: https://encyclopedia.pub/9379

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

[6] External Information About Apollo:
https://www.mo4tech.com/analysis-of-baidu-apollo-autonomous-driving-platform.html

[7] Alternative architecture
https://www.atlantis-press.com/journals/jase/125934832/view#sec-s4