CISC 322

Assignment 1: Report

Apollo - Smart Transportation Solution: Conceptual Architecture

Sunday, February 20, 2022

Group: Noah's Arkitects

Ricardo Lopez: 20192523 *19ral1@queensu.ca*

Norman Anderson: 20073995 *17na4@queensu.ca*

Hamiz Jamil: 20147727 *18hj16@queensu.ca*

Micahel Kalpouzos: 20116078 *m.kalpouzos@queensu.ca*

Abstract

The Apollo autonomous driving solution and its accompanying documentation along with reference autonomous driving architectures were analyzed to determine Apollo's conceptual software architecture. Through the collaboration of individual ideas amongst our group, Apollo's high-level conceptual architecture was derived to be a combination of the Process-Control and Pipe-and-Filter styles. Furthermore, we discovered within some subsystems an object-oriented approach was used to encapsulate data relevant to the operation of certain modules. After building a dependency diagram (Figure 1) to describe the structure for the mentioned architecture, each subsystem was detailed and two use-case diagrams were used to depict the flow of data and control whilst a car navigates to its destination and a car stops in front of a traffic light. Along with this, Apollo's concurrency model was laid out to observe parallel operations occurring in the system. Lastly, team issues within Apollo and lessons learned within our group were discussed to evaluate our thought processes amidst deriving Apollo's conceptual architecture.

Table of Contents

Abstract 2
Table of Contents 3
Introduction 4
Derivation 5
Conceptual Architecture 6
Subsystems
- Perception 7
- Prediction 7
- Planning 8
- Control 9
- Localization 9
- Map Engine 9
- HMI 10

Use Cases
- Use Case 1 10
- Use Case 2 11

Concurrency Model 11
Team Issues 12
Current Limitations and Lessons Learned 12
Conclusions 13
Data Dictionary 14
Naming Conventions 14
References 15

Introduction and Overview

With artificial intelligence taking the world by storm, autonomous driving is one of the many aspects which people can see being both implemented and improved in this very generation. With leaders like Tesla at the forefront of autonomous driving, the idea to make an open-source project relevant to autonomous vehicles was one no company had previously explored.

Apollo is an open platform project with the primary purpose of becoming an autonomous driving ecosystem by providing a comprehensive, safe, secure, and reliable solution that supports all major features and functions of an autonomous vehicle. Although Apollo has a variety of aspects that go into making real-world autonomous vehicles, this report will focus on the open-source software platform Apollo both uses and provides to the public. The development team at Apollo has outlined each of their software/hardware components and interactions, documentation, and even tutorials for people to implement Apollo's autonomous driving solution as they wish.

The purpose of this report is to identify and investigate the conceptual software architecture of Apollo. Through Apollo's deeply documented GitHub and website, we were able to determine the subsystems which comprised Apollo's conceptual architecture as being: Perception, Prediction, Planning, Control, Map Engine, Localization, and HMI. We then observed the structural pattern, essential invariants and computational models of the subsystems to conclude the architectural styles implemented by Apollo.

There are a variety of architectural styles in existence, depicted by the architectural principles used when designing the software in question; through careful analysis of these such principles used by Apollo, a combination of three architectural styles was found to define Apollo's software system. At a high level, Process-Control is a style crucial to the operation of Apollo's open-source software system. There are numerous process variables that must be controlled to meet the functional requirements of an autonomous vehicle, as the software itself is fully replacing a human driver. Furthermore, the Pipe-and-Filter architectural style is applicable for the communication between the subsystems which comprise Apollo's software architecture. Certain subsystems require a series of independent computations to be performed on data taken as input, then generate outputs to be used by other subsystems. Lastly, at lower levels within subsystems, an object-oriented style is used to maintain important bodies of information throughout the process execution cycle.

Along with analyzing Apollo's conceptual architecture and respective subsystems and modules, this report will also provide two use cases to describe each subsystem's role and performance, represented through sequence diagrams. Furthermore, we will define a concurrency model representing the parallel process that occurs in the software system. Lastly, we will discuss the team issues within Apollo regarding the division of responsibilities among participating developers, as well as the lessons we learned as a group whilst writing this report.

Derivation Process

Deriving the conceptual architecture of Apollo required extensive research and an in-depth analysis of the operations and dependencies within the subsystems of Apollo's open-source software platform. First, we better developed our understanding of the domain of autonomous driving through the research paper written by Sagar Behere and Martin Törngren: "A functional reference architecture for autonomous driving". This provided us with the knowledge capable of understanding the intrinsic points outlined on the Apollo website and GitHub. Apollo's manifesto and governance provided a concise overview of Apollo's mission, whereas their GitHub documentation showed each subsystem and module and their applications to Apollo's system. From reading through both of these, our team formulated a list of subsystems that we believed comprised Apollo's software platform.

We decided the most vital subsystems/components to Apollo's operations to be: Perception, Prediction, Planning, Control, Map Engine, Localization and HMI. The next step was for each member of our group to use the lecture slides provided for determining a conceptual architecture as well as the Apollo resources mentioned above to curate their own individual opinions of Apollo's architectural style and patterns.

Each group member proposed their idea of Apollo's conceptual architecture and provided reasoning through describing individual components and connectors of their proposed style. Although an object-oriented approach made sense for the encapsulation of data across subsystems, it did not fit the high-level criteria Apollo's software platform needed to operate. The Process-Control style made sense for the high-level, but it was missing the individual computation aspect Apollo required and Pipe-and-Filter offered.

After discussing the fundamentals and benefits of each proposed style, we concluded that Apollo's conceptual architecture uses a blend of Process-Control and Pipe-and-Filter for the high-level architecture, and an Object-Oriented approach for the lower-level within certain subsystems. The derivation of the conceptual architecture is depicted in Figure 1.

Conceptual Architecture

As stated in the derivation process, the Apollo Smart Transportation system is composed of 7 core subsystems and they can be seen in Figure 1. In order to fully replace a human driver, a lot of calculations and planning needs to be done by the system, so it can later be physically executed by the automobile until it reaches the expected destination. These processes can be heavy to execute since multiple of them are running at the same time and they all communicate with each other to plan the direction, speed and other variables that will affect the automobile. This is why a minimum of an 8-core processor and 16GB memory is required to run the system.

After finishing the research of every subsystem and comparing the different points of view in our team, we concluded that a Process-Control architecture style is the best representation for a conceptual architecture of the Apollo system. There are 2 main components in this architecture style: the process definition and the control algorithm. The process definition is handled in the Planning, Prediction and Perception subsystems, where all the process variables are manipulated and updated according to the input given to those components. The control algorithms are implemented on each of those subsystems to manipulate the process variables. Process variables are also controlled by sensors on the car, which can identify surrounding objects and automobile speed. Process variables are expected to store the speed, position, possible obstacles and direction of the automobile. Each of these variables is expected to be near a given reference value, otherwise, the automobile could potentially run into problems.

Within the main conceptual architecture, we also identified that certain components relied on a Pipe-and-Filter architecture style to communicate with each other. The Panning, Prediction and Perception subsystems depend on the input provided by the Map Engine, which is then processed to generate an output, and it is then given as input to the next subsystem. At a lower level, those 3 components implement an Object-Oriented architecture style. This style allows them to identify and protect their data by using encapsulation and abstraction. The objects are responsible for preserving the integrity of the data, which in this case is very important and relevant because the data in those components are eventually stored or given to the process variables.

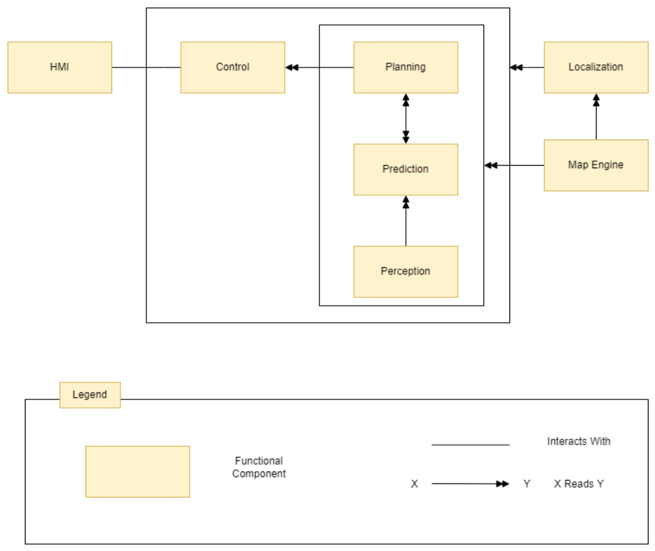


Figure 1. A proposed conceptual architecture for the Apollo Smart Transportation system.

Subsystems

Perception

The Apollo Perception module is a subsystem in charge of detecting and recognizing obstacles and traffic lights. Through input provided by perception submodules, Cameras, Light Detection and Ranging (LiDAR) and RADAR data, the perception module is able to detect, segment, classify and track obstacles in the current range of interest defined by the Map Engine subsystem, specifically the HD Map module. The perception module is at the forefront of Apollo's autonomous driving capabilities and is broken into two key perception components: Obstacle Perception & Traffic Light Perception.

Obstacle Perception

Based on the Fully Convolutional Deep Neural Network, the LiDAR-based obstacle perception predicts the probability of obstacles and their displacement. RADAR-based obstacle perception processes the initial RADAR data by extending the track id, removing noise, building obstacle results and filtering results by a range of interest. Through the fusion of results produced by both LiDAR-based and RADAR-based obstacle perception, the perception module is able to recognize obstacles, as well as predict their motion and position information.

Traffic Light Perception

The Traffic light submodule obtains coordinates of lights in front of the car through coordination with the aforementioned HD Map. Through the parameters of another perception submodule (sensors), traffic lights are projected from real-world coordinates to image coordinates, then detected in the range of interest with a surrounding bounding box to provide recognition of each of their different colour states.

Prediction

The Prediction module predicts the behaviour of all obstacles detected by the Perception module. This module takes in obstacle data as input from the Perception subsystem, including obstacle positions, headings, velocities and accelerations. Through this input data set, the Prediction subsystem generated predicted trajectories with associated probabilities for each trajectory for each obstacle in the input data set. It's important to note the purpose of the Prediction module is only relevant to obstacle behaviour within the range of interest defined by Perception, and this module does not predict nor plan the trajectory of the car itself. The prediction subsystem has four main functionalities depicted through submodules: Container, Scenario, Evaluator, and Predictor.

Container: Stores input data from Perception, Localization and Planning

Scenario: Analyzes scenarios involving the vehicle, including lane-keeping and junctions (traffic lights/stop signs)

Evaluator: Predicts path and speed separately for any given obstacle
Predictor: Generates predicted trajectories for obstacles

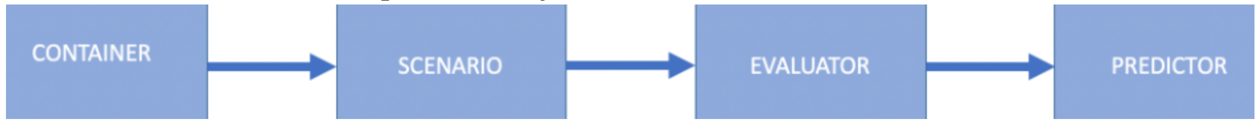


Figure 2. A proposed dependency diagram for the Apollo Prediction module - arrowhead represents data flow.

Planning

The Planning module in the Apollo system uses several outputs from different sources within the program to create a collisionless route. This module's need for information causes it to interact with basically every module in the system to plan the best possible route. This module plans its safe trajectory by receiving inputs from Localization, Perception, Prediction, HD Map (in modules/map/data), routing, and task_manager. The program has a plan for multiple situations like dead ends, stop signs, traffic lights, etc… and using the information it is given it chooses the best plan for the given scenario.

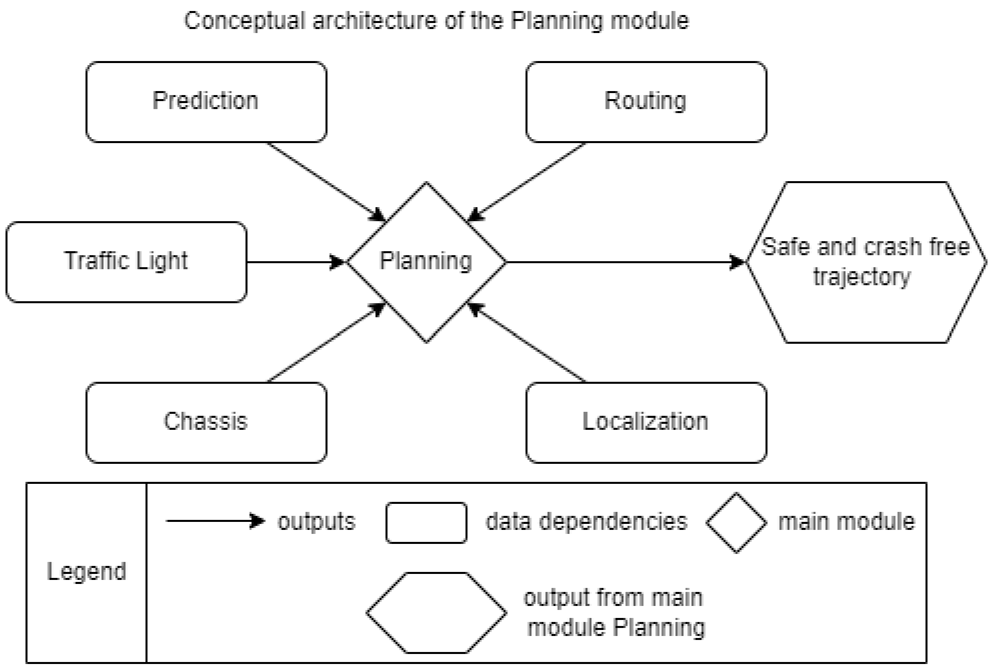

Conceptual architecture of the Planning module
Prediction
Routing
Traffic Light
Planning
Safe and crash free trajectory
Chassis
Localization
Legend
outputs
data dependencies
main module
output from main module Planning

Figure 3. A proposed conceptual architecture for the Apollo Planning module.

Control

Based on the trajectory provided by the planning subsystem, control utilizes different algorithms to navigate the car and generate a comfortable driving experience. Control has introduced the following features most recently:

- Model Reference Adaptive Control (MRAC): This algorithm is designed to decrease steering latency and delays. It also ensures faster and more accurate steering control actions for tracking the planning trajectory.
- Control Profiling Service: This service analyzes the current and past road performance of the car and informs the user of the improvements seen within the module.

Localization

The Localization module provides all the localization services used by the Apollo system. These services allow the system to produce a robust and precise vehicle localization based on multi-sensor fusion in diverse city scenes. The module accomplishes this by using two different localization methods that take different inputs but provide the same output. The RTK localization method takes two inputs: GPS and IMU. The multi-sensor localization methods take three different inputs: GPS, IMU and LiDAR. Both localization methods output an object instance of a defined message. Users can implement new localization methods by creating their own methods and registering them in the system. This module depends on the “Map engine” module to provide the input data that will be used to generate the output.

Map Engine

The Map Engine module is in charge of providing the map data to other modules, such as “Localization” and the group of “Perception”, “Prediction” and “Planning”. The other modules then use the provided data to transform or process the information to generate their output. The Map Engine module has two important submodules “HD Map” and “Relative Map”. The “Hd Map” is the main module that provides the mapping data through the system. The “Relative Map” submodule is a middle layer that connects the “HD Map”, the “Perception” and the “Planning” modules

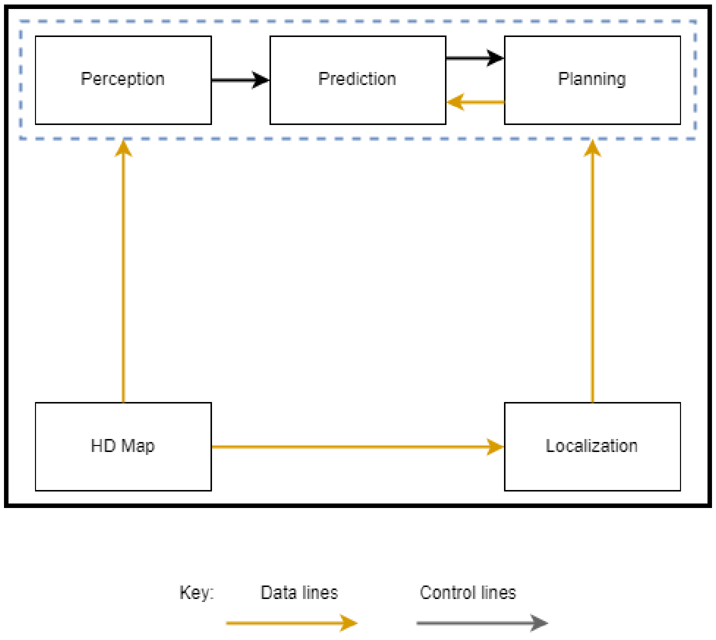


Figure 4. A proposed conceptual architecture for the Apollo Map Engine.

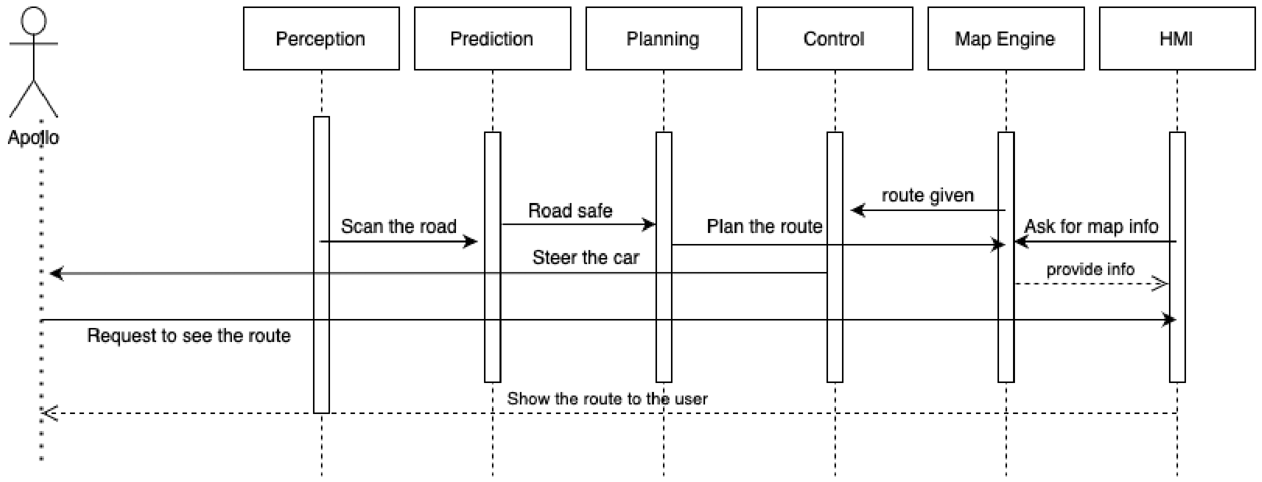
HMI

The Human-Machine Interface, as the name suggests, is a user interface that connects a person to a machine. Within the context of Apollo auto, HMI is a python application based on Flask that uses an HTTP web socket to query ROS modules. The HMI device that's placed on the right-hand side of the dashboard can trigger actions like:

- Change mode, map, vehicle and driving mode.
- Register event handler for changing mode, map and vehicle.
- Start, stop or execute other registered commands for modules.
- Execute registered tools.
- Submit DriveEvent which will be recorded as a ROS message.

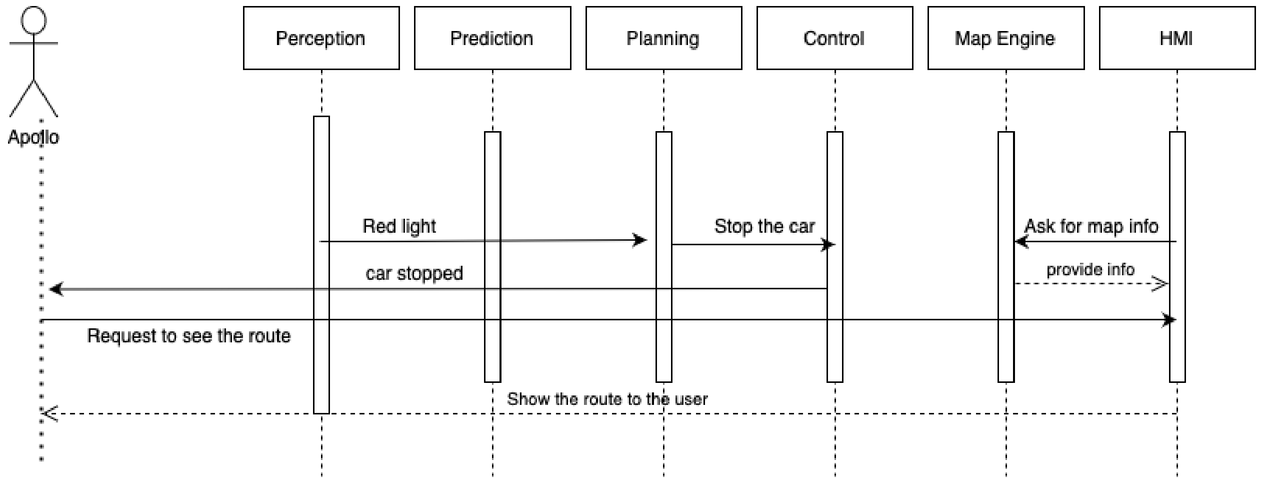
Use Cases

Use Case 1: Car navigating to its destination



In this scenario, the Perception and Planning subsystems work together to navigate the vehicle to its destination, obeying the traffic laws and avoiding any obstacles. Planning will come into play after it's gotten the signal from Prediction that there are no obstacles and uses the Map Engine to come up with a route. The route then is transferred to Control to steer the vehicle in the right direction. HMI, in conjunction with Map Engine will show the user the state of the vehicle and its route.

Use Case 2: Car stopping in front of a traffic light



This time around, the Perception subsystem has detected a traffic red light. As a result, Planning will use this and command the Control subsystem to stop the vehicle.

Concurrency Model

For an autonomous vehicle to function properly there needs to be a concurrency computing model in the architecture of its system to enable different processes of the car, such as analyzing the road and acceleration to communicate and collaborate with each other. This feat is accomplished by Apollo Cyber RT, a centralized runtime framework to allow different parts and algorithms of the vehicle to be executed out of order or in partial order without affecting the final outcome, which is a smooth driving experience.

At the basis of the architecture lies a set of components with predefined inputs and outputs. Each component contains a specific algorithm module built to process a set of data inputs and generate a set of outputs. The Apollo Cyber RT takes these components and combines them with fused data from the sensors of the vehicle to create user-level tasks. Each task is then given an execution priority based on what's needed of the vehicle at the moment and its resource availability. Then, these tasks are executed as optimized threads.

The Apollo Cyber RT is designed as a centralized and parallel computing model, enabling high concurrency of task execution, low latency and high throughput. It enables the Planning, Control, Localization, HMI and Perception to integrate seamlessly on top of its architecture.

Team Issues Within Apollo

The initial Apollo system began at the Baidu research institute in 2013 and was created by small teams that could easily communicate with each other. The project grew exponentially and partnered with 50 partners around the world causing the number of workers to grow exponentially as well. The teams dealt with issues of communication and tracking of changes due to the great increase of developers. The project's solution to the lack of communication was to shift to an open platform. This change allows the thousands of developers and many teams to work on the code concurrently. The Apollo system is continuously being updated and the subsystems are being enhanced for better customer experiences. This shift to an open platform allows developers to work on their specific areas of expertise and not need to re-implement fundamental components. Apollo developers had trouble maintaining data amongst developers due to the fact that there were more different components that needed to be separately accessed. In the 7th version, the Apollo team partnered with Data Pipeline and developed an online development platform that was very user-friendly. Apollo's 7th new development platform connected the HD Map, Production Component, and Apollo Data Pipeline to make development smoother and simpler.

Current Limitations and Lessons Learned

The limitations of the report were the limited amount of conceptual architecture that was presented, as a majority of the findings were about the program's conceptual architecture. The interpretation of conceptual architecture from code is difficult and was challenging at some points. We wish we knew that most of Apollo's documentation of conceptual architecture was derived from earlier versions of the software. This limitation caused a lot of confusion and disagreement among the group and if we knew this ahead of time we could have prevented wasting meeting time.

The team had some issues with deciding the conceptual architecture of the Apollo system. Hamiz and Michael thought that the architecture would mostly be a Pipe-and-Filter style of architecture, while Norman and Ricardo believed the architectural style to be more of a Process-Control style. The team had to work through these differences in opinions and we had two people describe their thought processes of why they believed they had the correct architectural style. We had a vote after and came to the conclusion that both architectures are relevant within the system and that there are some parts that actually include a third style of architecture, the object-oriented style. An issue we overcame throughout the creation of the report was that Leslie was not able to communicate with the group for a while due to personal reasons. Our group had already decided the tasks of each group member for this project and when Leslie contacted us we needed to fairly divide our work again. We had a group meeting and agreed upon how to evenly distribute the workload among ourselves. Leslie was tasked to complete one component from each of our previously decided tasks and every one had four components of the report to complete. The Apollo system has many different versions, an issue we encountered while we were writing this report was our confusion with the differences between the versions and what components have changed throughout the evolution of the program. We dealt with this issue by having weekly meetings to clarify and compare our findings. Overall this team had great communication and problem-solving skills to deal with the issues that arose when creating the report.

Our group together learned many lessons throughout writing this report. The team learned how to cooperate and come to agreements on controversial opinions about the architecture style, recounted in team issues. We learned how to compromise and communicate in a respectful manner when planning out the distribution of the workload. We learned from this distribution of the workload that some of the students had much more work than others and we will redistribute tasks in A2 to make everything fair. The team learned how to set up times and places for meetings and organize our schedules to get the work done on time. The one thing our group would do differently would be to have more meetings more consistently. This change would prevent members from misinterpreting things and have everyone on the same page. This group learned a lot of great lessons from this report and will take our learnings into our next reports and into the real world when we graduate.

Conclusion

Although our group began this report with conflicting ideas as to how Apollo's conceptual architecture is represented, we managed to see the story from each other's points of view, allowing us to merge various ideas and finalize our deduction of the system's architecture. We determined the conceptual architecture's style to be Process-Control and Pipe-and-Filter at the high-level, and Object-Oriented at the low-level within certain subsystems. All three of these styles play a vital role in the operation of an autonomous driving platform, as there is constant data being processed, modified and communicated across a variety of subsystems.

Apollo's goal is to create a virtuous cycle where software and services are deployed into vehicles to obtain data, and that data is used to further improve the autonomous software system. With this iterative cycle of improvement at Apollo, the rate of innovation is at a peak with open-source capabilities allowing source code portions to be modified by a range of talented developers. Apollo's mission has already evolved from simple virtual autonomous driving simulations to the mass-production of autonomous vehicles and is nowhere near stopping anytime soon.

Data Dictionary

Autonomous Vehicle: a self-driving car that is capable of sensing its environment and moving with little to no human input.

Object-Oriented: An architectural style of software systems that is suitable for applications in which a central issue is identifying and protecting related bodies of information.

Process-Control: An architectural style of software systems that is suitable for applications whose purpose is to maintain specified properties of the outputs of the process sufficiently near given reference values.

Pipe-and-Filter: An architectural style of software systems suitable for applications that require a defined series of independent computations to be performed on data.

Subsystem: Component of a large system

Naming Conventions

Global Positioning System (GPS): A global navigation satellite system that provides location, velocity and time synchronization.

Inertial Measurement Unit (IMU): An electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body.

Light Detection And Ranging Sensor (LiDAR): A remote sensing method that uses light in the form of a pulsed laser to measure ranges (variable distances).

HyperText Transfer Protocol(HTTP): The protocol used by the World Wide Web defining how messages are formatted and transmitted between the browser and a web server.

References

- Guowei Wan, Xiaolong Yang, Renlan Cai, Hao Li, Yao Zhou, Hao Wang, Shiyu Song. "Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes," 2018 IEEE International Conference on Robotics and Automation (ICRA), Brisbane, QLD, 2018, pp. 4670-4677. doi: 10.1109/ICRA.2018.8461224. Link
- Apollo Governance. Apollo. (n.d.). Retrieved February 15, 2022, from https://apollo.auto/docs/manifesto.html
- “Apollo Cyber RT Framework.” Apollo, https://apollo.auto/cyber.html.
- Official Apollo GitHub documentation
    - https://github.com/apolloauto
- Apollo’s user tutorial for both developers and individuals looking to use the autonomous driving software Apollo builds
    - https://apollo.auto/devcenter/devcenter.html
- Lecture material regarding the recognition of architectural styles and patterns
    - https://onq.queensu.ca/d2l/le/content/642417/viewContent/3751086/View
- “A functional reference architecture for autonomous driving” by Sagar Behere and Martin Törngren
    - https://onq.queensu.ca/d2l/le/content/642417/viewContent/3814366/View