Apollo Conceptual Architecture

Date: February 14, 2022

Authors

Lavi Ionas: 17li3@queensu.ca
Xinyu Chen: 18xc26@queensu.ca
John Scott: 18jbs2@queensu.ca
Baorong Wei: 18bw16@queensu.ca
Zhihan Hu: 18zh22@queensu.ca
Anthony Galassi: anthony.galassi@queensu.ca

Abstract

Baidu's Apollo Auto has undergone extensive changes throughout the development of its open-source autonomous driving project. First, we dived into the different modules and their interconnection to find the architecture is efficient and logical in nature. We explored the control and data flow in the project and found that tasks in the control flow are executed linearly, suggesting it is process-oriented. Then, we investigated the concurrency within the system's architecture. Only three modules, perception, prediction, and planning, do not have concurrency within the system.

We have found there are four main sections as of version Apollo 7.0: Cloud Service Platform, Open Software Platform, Hardware Dev Platform, and Open Vehicle Certification Platform. There are three notable use cases for Apollo's software: RoboTax, Minibus, and valet parking. As development continues, their autonomous vehicle software will perform better as they have more data and technology to support the innovation.

Introduction and Overview

In this report, we analyze and discuss the conceptual architecture of Baidu Apollo Auto, an open-source autonomous driving project developed by Baidu, Kinglong, and more than 40 other companies. We present these ideas and concepts using textual analysis as well as various diagrams, to create a clear vision of the key architectural components in the system. Resources and references have come primarily from the Apollo GitHub repository, and additional references will be provided in the appendix. This conceptual architecture report will be part of our final report on Apollo Auto architecture overall with concrete architecture and enhancements.

In this paper we investigate the various aspects of Apollo's architecture in order to gain a greater understanding of both this software in particular, as well as the structure and architectural methods of architecture in general. We begin by examining the component parts of this software, as well as their implementation, establishing a greater understanding of the nature of this software before diving into more abstract views. We then discuss the evolution that the system has undergone, which gives a look at the software from an operational view, allowing us to understand the way that the software is modified throughout time, how it began, and how it got to where it is now. Next, we revisit the interaction of the different modules in the program, this time specifically identifying the control and data flow between these elements. This then leads to our discussion of the concurrency of the software, and which parts do or do not exhibit concurrency with one another. Shifting focus from the software itself, we then canvass the partitioning of responsibilities between the developers themselves, which gives us a more complete understanding of the context in which the software is made, from the development view. Finally, we visually represent some common use cases of the software using sequence diagrams, opening our eyes to the more specific functionality that we may see of this software in real-life applications.

Architecture

Component Parts and Interaction

The system guides and controls autonomous driving. It's broken down into interacting parts based on the functionality of those parts.

Perception Module

The perception module has two key components. The obstacle perception component uses data from LiDAR and RADAR to detect obstacles near the vehicle and fuses the results together. The software then predicts obstacle properties based on a fully convolutional deep neural network. The second key component is its traffic light perception which uses the HD-map module to get the coordinates of traffic lights in the path of the car.

Prediction Module

The prediction module takes the images from the perception module and detects various real-life things from the deep network labeling such as traffic lights, lane/flow of traffic, and 3D objects. It then has sub-modules to track the lane/flow of traffic, convert 2D objects to 3D from the images, as well as recognize traffic light symbols. Further, it calibrates all of this data into the final traffic lights as well as the final objects categorized by type, distance and velocity.

The prediction module learns to predict through various means to improve its accuracy. The first is simulations with the ability for users to input road types, obstacles, driving plans, and traffic light states. The second is an execution mode, which provides users with the ability to run multiple scenarios at the same time and verify modules in the code environment. Next are some of the key features that make the learning so much faster than typical training data. The platform has a built-in virtual training partner to act as a sort of “virtual driving school” to enhance the safety of the vehicles.

Data Scientist

Further, with all of the customization of scenarios and localization, realistic scenarios can be crafted that verify the working state of multiple modules and provide proper testing for the technology. The platform also has an automatic grading system for different routes based on its past observations. By visualizing the real-time road conditions in 3D, it can visualize how the modules will output accordingly.

Planning Module

The planning module handles various scenarios that may occur while driving autonomously. In these specific scenarios it will direct the vehicle through the situation. It is currently configured to handle stop signs, traffic lights, bare intersections, valet parking, dead ends, and pulling over, if necessary.

Guardian Module

The guardian module is a safety module that acts as an action center for the vehicle. If the module were to receive input from the Monitor module or the Control module that indicates a detected or predicted failure in some part of the system, the guardian module would intervene and try to fix the failure or stop the car to prevent potential accidents.

Localization Module

The localization module provides localization services through one of two ways, the RTK (Real Time Kinematic) based method or the multi-sensor fusion method. The RTK method incorporates GPS and IMU (Inertial Measurement Unit) information. Meanwhile, the multi-sensor fusion method incorporates both of those lines of information as well as LiDAR information.

Control Module

The control module is intelligent in the fact that it can handle different road conditions, speeds, vehicle types and CanBus protocols with a high control accuracy. It can do this based on the planning trajectory and the car’s current status. It uses different control algorithms to create a smooth driving experience despite rough conditions.

How The Modules Interact

The perception module and HD-Map module are fed into the prediction module. The HD-Map module enables the localization module to determine the local features of the geographic area, which feeds into the planning module. The prediction and planning module interchange data and control to detect and react to various obstacles, objects or scenarios. Once all of the real-time data is interpreted it's passed to the control module to act on those outputs from the earlier modules.

The control module outputs to the guardian module which is a safety module in case of system failures. In any case, the guardian module feeds data into the canbus module, which is able to act on the vehicle, as it's a microcontroller. From the output of this entire system, it would show the data on the monitor of the system testing it.

System Evolution

Over the course of its development, the Apollo software project has undergone some very major milestones, impressive both when compared with its humble beginnings, and even when compared to other major self-driving software.

But how could it have reached this point? These results are the result of meticulous evolution of the project, as although a solid basis is important, software cannot improve to reach such heights without this careful consideration, both of its past and future.

Apollo 1.5:

In its first update, the Apollo project pivoted from designing their software to work in a controlled and expected environment, to an open environment, which had many more factors that are out of the control of the software designers. As such, this is the largest update, as they had to add significantly more support for sensors, in order to properly account for the unexpected nature of their environment. This support primarily came in the form of LiDAR, camera, and RaDAR capability. In order to take advantage of this new information, the software architects had to add in a Map Engine, Perception, Planning, and End to End module. These work with each other in conjunction to analyze the sensor data, by using it's perceptions to create a virtual map, and plan it's course of action accordingly. This process is also aided by the cloud portion of this software, which provides the HD map and Simulation modules.

Apollo 2.0:

The aim of Apollo 2.0 was to equip the software to deal with simple urban roads, and all of the obstacles that come with them. This necessitates more nuanced functionality, such as the ability to avoid collisions with objects, change lanes, and stop at traffic lights. To this end, upgrades were developed for the camera and RaDAR, which were brought to task for identifying these issues. Besides updates that are explicitly in service of meeting these new challenges, some housekeeping updates were made, particularly considering cloud security, end to end, and Over The Air (OTA) firmware update capability.

Apollo 2.5:

Apollo 2.5 introduced improvements to the camera, the perception, and the planning modules, in order to allow for autonomous operation on geo-fenced highways. The primary objectives of this operation were the ability to maintain cruise and lane control, along with of course avoiding collisions with other cars. Though it can drive on highways, it is only under certain lighting conditions, and with clear markings on the highways.

Apollo 3.0:

Apollo 3.0 shifted some of the focus to addressing low-speed applications, such as minibusses, valet parking, and micro-cars. They developed the Duer car system along with DuerOS to provide a turnkey solution to these applications, along with adding some standardization in the form of the Open Vehicle Interface Standard. With this version comes L4 autonomous driving in a closed venue at low speed settings.

Apollo 3.5:

Apollo 3.5 added the capability to handle much more complex driving scenarios, with a focus on downtown and residential environments. This was accomplished with much more advanced perception algorithms, aided by the addition of full 360 degree camera coverage. This also introduced scenario based planning, which provides algorithmic support for navigation through more complex situations that may be found in more residential or downtown environments, such as unprotected turns and narrow streets. 3.5 also added V2X (vehicle to everything) support, which is a system that can allow the vehicle to more completely communicate with its environment.

Apollo 5.0:

With Apollo 5.0 came the addition of a cloud server Data Pipeline, as well as an upgraded perception model powered by deep learning. Existing features also saw upgrades nearly across the board, encompassing most open software platform modules, as well as hardware components, such as computational unit, GPS, and LiDAR.

Apollo 5.5:

Apollo 5.5 further improves urban driving, introducing curb-to-curb driving support. This came with upgrades to perception, prediction, simulation, planning, and control, as well as additional solutions for intelligent signal control systems, robotaxis, and a CarOS.

Apollo 6.0:

6.0 incorporates much further integration of deep learning models, and improves data pipeline efficacy. This deep learning integration span across perception, prediction, planning, control, and more. There were also further updates in the V2X capability.

Apollo 7.0:

Apollo 7.0 further develops deep learning integration by implementing 3 brand new learning models, as well as a PnC reinforcement learning model, which is used to provide training and simulation evaluation, in order to improve the simulation service. It also introduces Apollo Studio, which, in conjunction with the data pipeline, provides an online development platform for Apollo developers.

Control and Data Flow

Please note, tasks in control flow* are executed linearly, before one task finishes execution, we cannot move to other tasks. Hence, control flow is more process-oriented. However, the data flow* may indicate synchronous transformations. Data is moving around based on the connections among processes, so data flow is more information-oriented. As shown in the graphics, some data can be grouped together, and they are coordinated to process together effectively.

Overall, The control flow defines the order of computation, whereas the data flow defines the data movement.

The following analysis is based on the Hardware Connection and Software Architecture Overview diagrams under the Apollo GitHub README.md file. They

are provided in the Diagram section of this report. In the diagrams, processes are represented by circles, and external entities that interact with the system are presented by rectangles. Component level information is omitted as this document focuses on the conceptual architecture.

Control Flow

Software

Perception, Prediction, Planning, Control, Guardian, and CANBus* are serialized. If a special event occurs, the Control system may branch to CANBus instead of going to Guardian then to CANBus. After executing these tasks, the flow goes to HMI.

Hardware

IMU and GPS Antenna connect through DB9 and TNC respectively to NovAtel GPS Receiver. Through USB, the execution flows between NovAtel GPS Receiver and Neousys 6108GC with the graphics card (GTX1080). The data (GPRMC and PPS) collected by the GPS Receiver signals through customized cables to the Velodyne VLS-128 LiDAR. Then, the LiDAR (Light Detection and Ranging) executes and communicates with the AI GPU computer through 100M Ethernet. The ARGUS FPD-Link Camera and Ultrasonic Sensor connect individually to the GPU computer through USB for processing. The computer also takes command from the accessories, such as the Monitor, Keyboard, and Mouse. The card awaits for the execution from Vehicle Chassis CAN and Continental Radar separately using Vendor CAN cables. Then, the computer communicates with the card (ESD CAN Card 4CH) with the PCI port.

Data Flow

Software

HD Map communicates with Localization and the set of Perception, Prediction, and Planning systems. Localization also communicates with the three as well as Control. Within this flow, Prediction feeds the data to Planning. The Monitor provides data to HMI and Guardian, whereas the rest of the systems feed data to the Monitor. This means that the human input is collected by the HMI.

Hardware

The Car Power System relays data to the Dataspeed 12VDC Power Panel which transmits its data to Velodyne VLS-128 LiDAR (providing range, resolution, and vehicle detection), Continental Rader, the IMU process, NovAtel GPS Receiver,

WIF 4G LTE Router, and Neousys 6108GC with GTX1080 system, the AI GPU computer.

Concurrency of System Architecture

Concurrency means executing multiple tasks at the same time but not necessarily simultaneously. In a single-core environment, concurrency is achieved via a process called context-switching. If it's a multi-core environment, concurrency can be achieved through parallelism.

Concurrency is a property of a system in which several behaviors can overlap in time – the ability to perform two or more tasks at once. In the sequential paradigm, the next step in a process can be performed only after the previous has been completed; in a concurrent system, some steps are executed in parallel.

In activity diagrams, concurrent execution can be shown implicitly or explicitly. If there are two or more outgoing edges from an action it is considered an implicit split. Two or more incoming edges signify an implicit join.

From the software diagram on page 11, we can tell there is a data line from Monitor to Guardian and another data line to HMI. So there exists concurrency in the Monitor. The same reasoning also applies to the HD Map. There are two control lines coming out of Control, one to Guardian and another to CANBus, so there is an implicit split, a.k.a. a concurrency in Control. Guardian itself is receiving a control line from Control and a data line from Monitor, thus a concurrency exists in Guardian. And this also applies to HMI as well, which is receiving a data line from Monitor and a control line from the software. CANBus has concurrency as it receives control from both Control and Guardian. The data flow in Monitor and HMI can also happen synchronously with other data flows.

Implications of Responsibility Division of Developers

Based on the Apollo 7.0 module, there are four main sections: Cloud Service Platform, Open Software Platform, Hardware Dev Platform, and Open Vehicle Certification Platform. Differentiating from previous models, the 7.0 model introduced Apollo Studio that contains a Data Pipeline. Automated Driving Development encompasses many different development components. According to the learning module on the Apollo Auto website, the components include perception, high-precision maps and positioning, planning, control, system, cloud service, software platform, and hardware. The responsibility among developers could be divided based on the general categories listed above. Hence, in general, we would need developers and engineers who specialize in Cloud Service, Automated Driving Development, Front-end, Data Science, Quality Assurance,

Cyber Security, and Hardware. Although these sections are discussed separately below, responsibilities from different positions may overlap. Developers may collaborate in such a scenario.

In order to create clear descriptions of the responsibility division of developers at Apollo, we not only referenced Apollo Auto's official website but referenced job postings from other technology-related companies like Huawei and BlackBerry.

Cloud Computing

Generally, cloud computing includes analyzing the basic infrastructure of a business model, and researching how to transfer different functionalities, like Apollo Data Pipeline, to a cloud-based system. Common roles in cloud computing listed by AWS include Cloud Solution Architect, Cloud Developer, Cloud DevOps Engineer, Cloud Data Engineer, and Cloud Operations Engineer.

Automated Driving Development

Developers specializing in different areas can contribute differently to an automated driving development. Some are responsible for developing and testing algorithms for sensor perception. They could also be involved in the development of platforms, systems, and internal tools for autonomous vehicle Machine Learning (ML) models. Reinforcement Learning Developers would help with behavioral planning, motion planning, and motion control. To develop Apollo, we also need developers who have knowledge in Real-time Operating System (RTOS) and Cyber RT in order to process data and events that have critical time constraints in real-time.

Front-end

Please note, the term “front-end” is not limited to web development but includes general Human-Machine Interfaces (HMI) such as touch screens and mobile devices. We list Front-end development individually here because HMI development requires distinct programming skills, as the programming languages are different. Front-end Developers should provide Apollo users with an intuitive and efficient machine interface design. Depending on the specific functionality, developers could be responsible for different aspects of HMI development, most notably including hardware, software, and infrastructure.

Data Scientist

Baidu Apollo has an open-source platform. Open-source code portions can be modified and open capability components accessible through an API can be replaced with proprietary implementations. All this can then be contributed back to Apollo, redistributed, and commercialized. Although Apollo may require fewer developers due to its open-source nature, it would need Data Scientists to ensure that every commit can follow the license, testing, and coding style guidelines. In

addition, an autonomous car requires a significant amount of data to support its performance. Data come from sensors including but not limited to cameras, RADAR, LiDAR, GPS, location, and AI software. Data Scientists who are able to label, analyze, and train this data will be helpful for the development throughout different levels of autonomous driving.

Quality Assurance

To ensure the electronic-based system is “correctly” controlled by the software-based systems, the role of Quality Assurance is essential for preventing the devices from harming people. They ensure the safety of the systems and verify, validate, and test for automated driving. For example, they need to make sure the safety requirements are met under all potential situations including edge cases.

Cyber Security

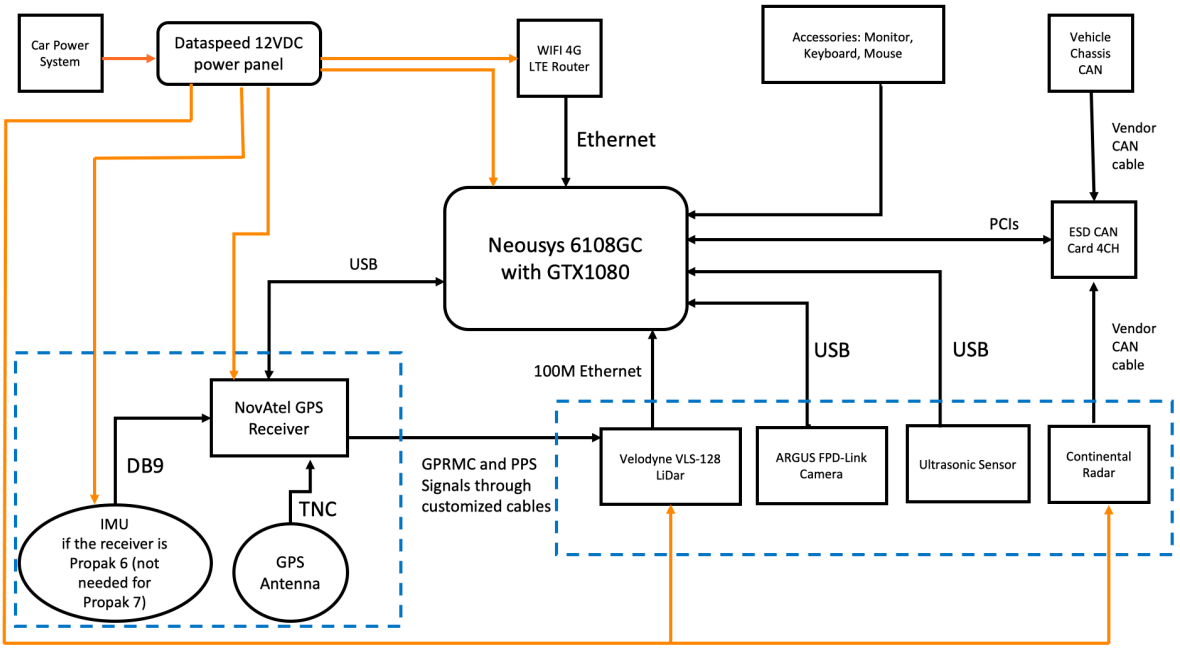
Developers who have expertise in this area measure the liability in order to reduce the connectivity and automation risks. Especially when it comes to an open-source platform like Apollo, the system is exposed to the public, which makes it vulnerable to malware. The sensors tend to be hackable causing significant automation risks. Cyber Security plays an important role in preventing incidents from happening.

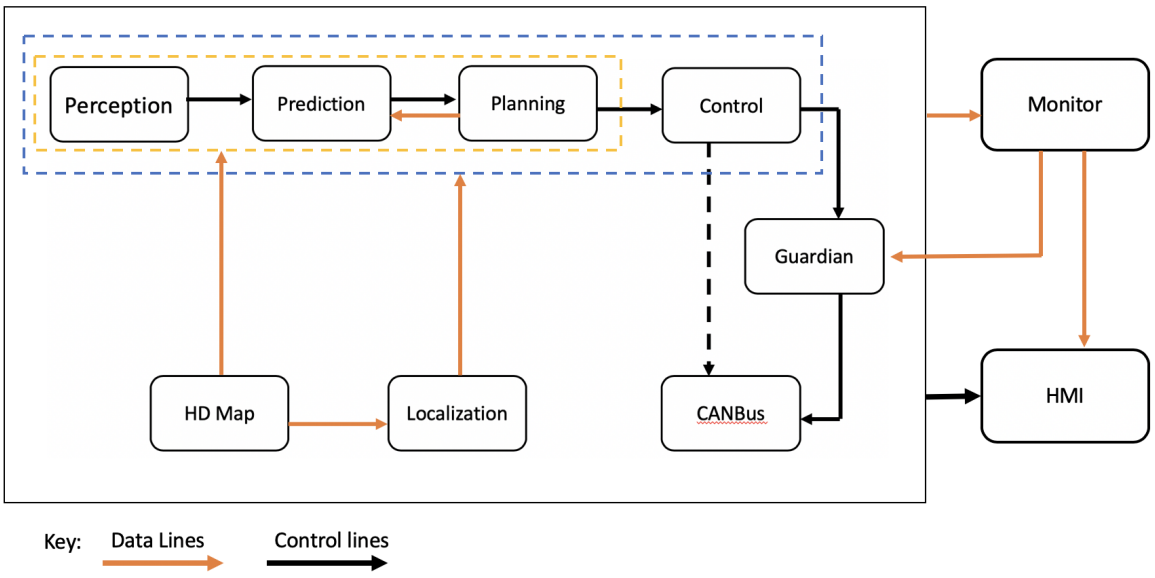
Hardware

Hardware components listed on Apollo’s official website include Control Area Network bus (CAN), LiDAR, and Apollo Official Kit. The term hardware may also include mechanical and electrical/electronic components. The Hardware Developers not only need to work on the integration with the Software Developers, but they also need to incorporate the data-driven design in the hardware. For example, they need to ensure the safety of environment sensor configuration, the computing platform, sensors and actuators.

Diagrams

Simplified evolution of software, provided on Apollo’s github

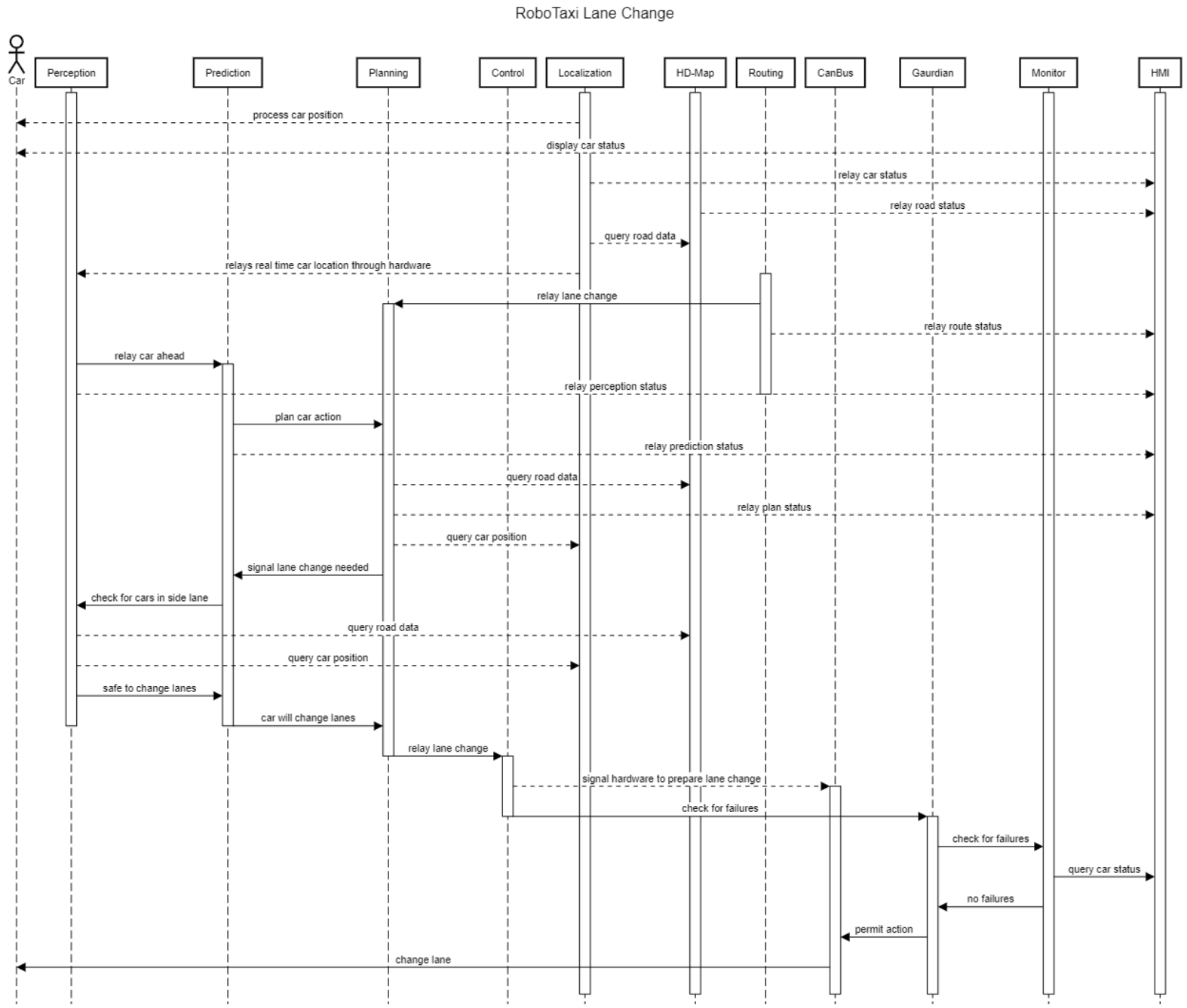




Use Cases

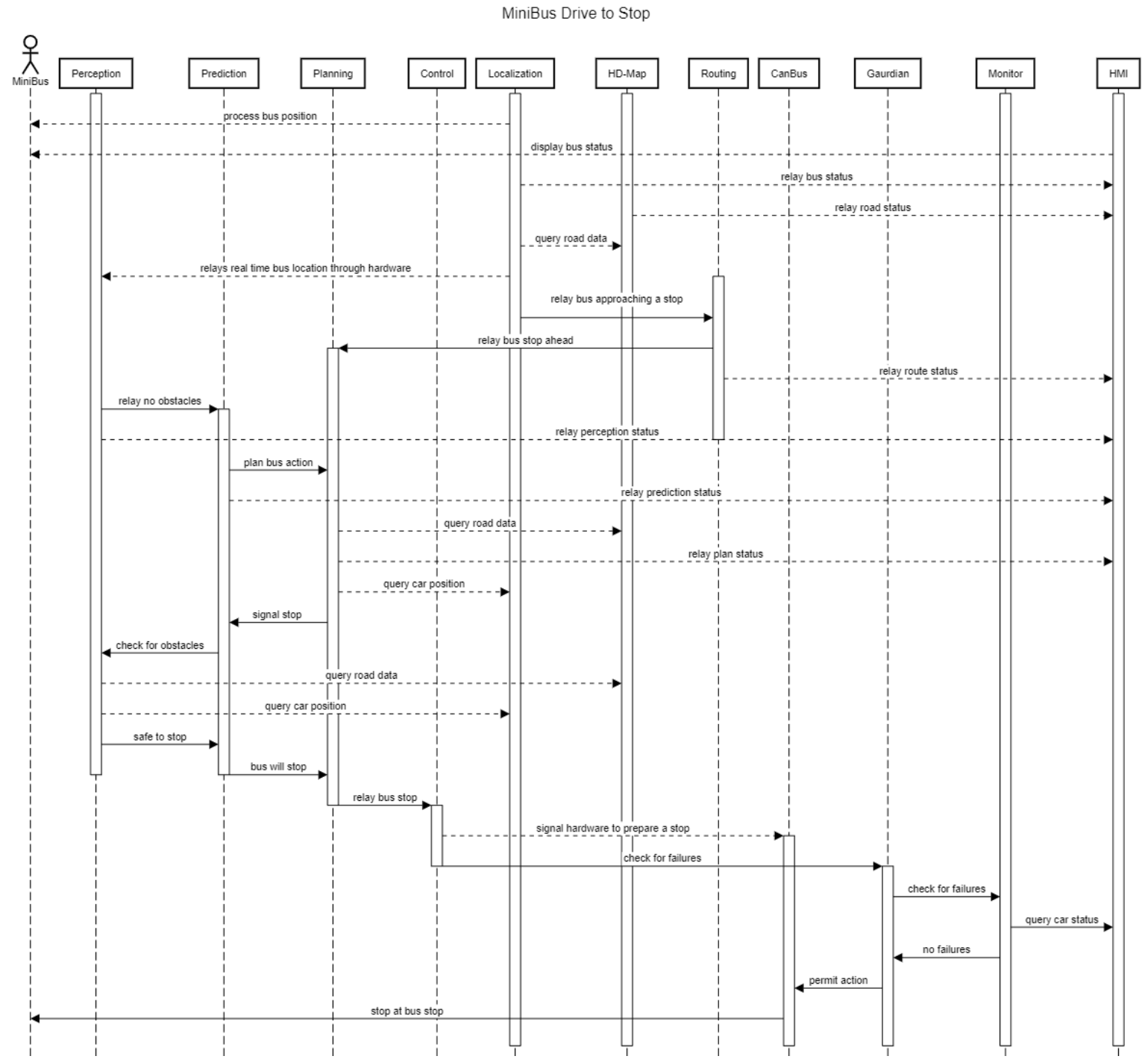
The following sequence diagrams depict the interactions between components of Apollo's software architecture and its flow and control of data. What follows are 3 notable use cases for Apollo's software: RoboTax, Minibu, and valet parking. The interactions between models are depicted in the Control and Data Flow section of this report. Please reference that section for a detailed interpretation of the relations between the individual modules depicted in the following use sequence diagrams.

Additionally, please note that higher resolution images will be available at the group website for viewing / downloading purposes.



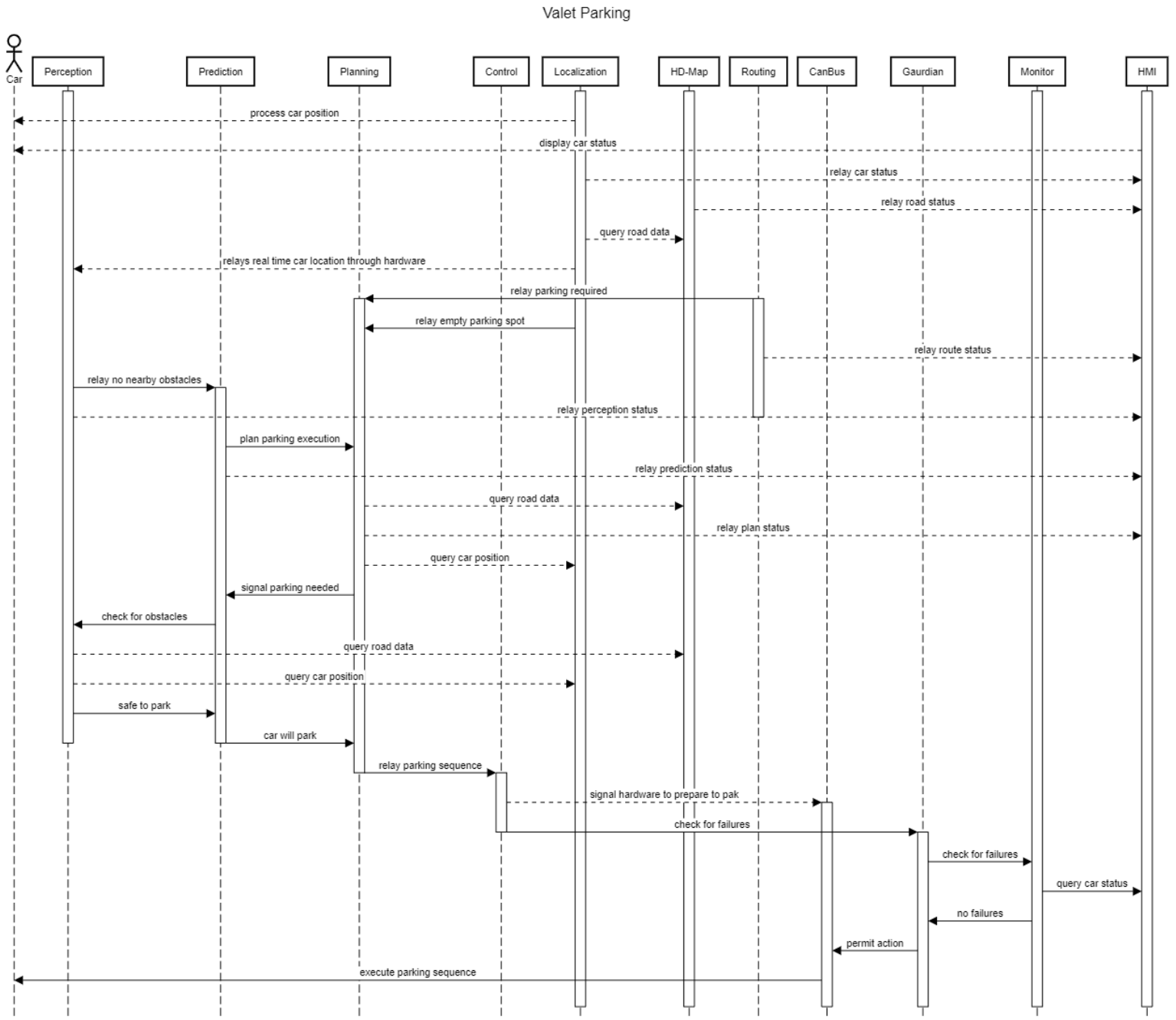
Sequence Diagram I: RoboTaxi Lane Change

This sequence diagram depicts the simplified sequence the software undergoes to safely command a lane change with the RoboTaxi. The act of changing lanes is a complicated algorithm and requires meticulous communication between modules.



Sequence Diagram II: MiniBus Bus Stop

The following sequence diagram depicts the process carried out in the MiniBus' software. The goal of the diagram is to present the steps the architecture needs to take to successfully stop the bus at the right location.



Sequence Diagram III: Valet Parking

The following sequence diagram describes the process of an autonomous valet parking system. This diagram assumes the actual act of parking is a predetermined sequence / program. The car needs to position itself in the correct state, only then will it execute the parking sequence.

Data Dictionary

Control flow: the order of an imperative program execution
Data flow: a flow of data transferring from one part of the system to another
CANBus: stands for Controller Area Network. It allows fast communication between the microcontrollers and devices.

V2X: vehicle to everything communication, enables communication between the vehicle and nearby devices that are relevant for self driving, such as traffic lights

Conclusions

Through our analysis in previous sections, we can get a rough idea of the organization of the Apollo system. This organization includes all components, how they interact with each other, the environment in which they operate, and the principles used to design the software. We can also see the evolution of the software. We can tell the enormous effort the project team has put into this system, as well as the different roles within this team.

Lessons Learned

In this report we collectively learned the conceptual architecture of Apollo and how its system interacts internally. With hindsight, we wish we could have had a greater understanding of architectural systems before proceeding with this project. That would have aided us in creating a more in depth analysis of Apollo's system architecture.

References

Apollo Website: Apollo
Apollo Github: Releases · ApolloAuto/apollo · GitHub