CISC 322
Assignment 1
Conceptual Architecture of Apollo's Open Software Platform
Sunday, February 20, 2022

Apollo-gies in Advance

Cameron Beaulieu	19cgb@queensu.ca
Truman Be	18tb18@queensu.ca
Isabella Enriquez	18ipe@queensu.ca
Josh Graham	18jg48@queensu.ca
Jessica Li	19jal@queensu.ca
Marc Kevin Quijalvo	18mkq@queensu.ca

Table of Contents

Table of Contents **2**

Abstract **3**

Introduction **3**

High-Level Conceptual Architecture **4**

- Perception 5
- Prediction 6
- Planning 6
- Routing 6
- Control 7
- Guardian 7
- CANBus 7
- HD Map 7
- Localization 7
- Storytelling 8
- Monitor 8
- HMI 8

Evolution of the System **8**

Concurrency **9**

Division of Responsibilities for Developers **10**

Diagrams **11**

External Interfaces **12**

Use Cases **13**

Conclusion **14**

Limitations and Lessons Learned **14**

Data Dictionary **15**

Naming Conventions **16**

References **16**

Abstract

In this report, our team discusses the high-level conceptual architecture of Apollo, an open-source platform for the development, testing, and deployment of autonomous vehicles. It specifically focuses on Apollo 7.0, released on December 28, 2021. Included in this discussion is the conceptual architecture derived by our team, the rationale behind our derivation, explanations for each subsystem and connector (control line or data line) referred to in the derived architecture, and two noteworthy use cases.

Our team determined that the conceptual architecture followed the "Publish-Subscribe" architectural style (also known as the "Implicit Invocation" style). It includes twelve modules, or subsystems, namely: Perception, Prediction, Planning, Routing, Control, Guardian, CANBus, HD Map, Localization, Storytelling, Monitor, and HMI. Each module has unique functionality and interactions within the system, which are detailed in this report. To further clarify the control and data flows within the derived architecture, two use cases of the Apollo system—(i) making a left turn on a green light; (ii) changing lanes on a highway—are analyzed and depicted with sequence diagrams. The evolution of Apollo, its concurrency, the division of responsibilities for participating developers in the context of this conceptual architecture, and Apollo's external interfaces are also touched on as supporting discussions. All our key findings are summarized in the report conclusion, along with proposals for future directions.

Finally, outside of the scope of Apollo, our report outlines any noteworthy limitations that the team encountered and lessons we learned in our research and writing of this report.

Introduction

Since the invention of the modern automobile, automotive technology has rapidly evolved. Today, the most prominent topic in this field is the notion of the autonomous vehicle—that is, a vehicle which is aware of its surroundings and is capable of navigating with little to no engagement from the driver. Until recently, autonomous vehicles were one of few rapidly growing technologies not majorly influenced by open-source, with companies like Uber and Google opting for proprietary technology.

In a pursuit to change that approach, Baidu launched the Apollo project, an open-source autonomous driving platform, in 2017. The project's Open Software Platform—the basis of this report—empowers developers and other stakeholders to more easily develop, test, and deploy autonomous vehicle projects. Since its initial release, Apollo has become arguably the largest open-source autonomous driving platform in the world, with a growing portfolio of contributing partners including Ford Motor Company, Microsoft, and Intel Corp, among hundreds of other companies.

As of version 7.0, released in 2021, Apollo's Open Software Platform has twelve top-level modules (excluding the Cyber/RTOS and V2X modules in accordance with Professor Bram Adams's guidelines). As such, software architecture is of utmost

importance to this project, as it helps the developers organize this complex—and given the subject matter, safety-critical—system, including all the modules, their interactions with each other, and the overall principles guiding the system's continuous design and evolution.

For this initial report, our team discusses Apollo's conceptual architecture, derived largely from documentation available in the official GitHub repository for the project. Through analysis of this documentation—which details the different modules and how they may interact with other modules, changes and new features introduced by each version of the project, and how to use the platform and/or contribute to it—our team agreed that the architecture for Apollo falls under the "Publish-Subscribe" style, a software architectural style most suited for applications with loosely-coupled collections of components, each having their own operations which may trigger other components' operations. In the Apollo system, we concluded that these components refer to the twelve modules: Perception, Prediction, Planning, Routing, Control, Guardian, CANBus, HD Map, Localization, Storytelling, Monitor, and HMI.

Following our team's derivation process and justification of the proposed architecture, we break down each module, including their function and how they interact with other modules (including the control and data flows among them), ultimately making sense of how these subsystems fit in the proposed architecture. We then discuss how the Apollo system has been evolving, concurrency in the system, the implications of the conceptual architecture for the division of responsibilities for participating developers, and the system's external interfaces. To further support our derived architecture, we give it concrete context by illustrating two use cases important for the autonomous driving platform: (i) making a left turn on a green light; and (ii) changing lanes on a highway.

We conclude the report with a summary of our key findings, as well as proposals for future directions. In reflecting on our process, we also documented noteworthy limitations and lessons learned. Through constructing and exploring the conceptual architecture of Apollo, our team has attained a thorough understanding of this system's structure, functional requirements, and essential relations between its subsystems. We believe that this research experience will help us immensely in the future, leaving us better equipped to identify the Apollo system's concrete architecture.

High-Level Conceptual Architecture

Our derivation process for the conceptual architecture was split between analyzing the higher level documentation and the lower-level subsystem documentation. In terms of the higher-level documentation, we began by analyzing previous, less complex versions of the architecture, that being versions 5.5 and earlier. With a general understanding of the system, we then analyzed the changes that came with versions 6.0 and 7.0, which introduced new high-level modules such as Storytelling and Routing, as well as modifications to current modules such as Perception to integrate deep learning models. To enhance our understanding of the high-level architecture, we

continued by reading through the documentation of each high-level module to gain insight on the interdependencies, as well as the control flow of the system.

Through this process, we decided that the best representation of the conceptual architecture would be through a “Publish-Subscribe” architecture (Figure 1), with the data lines representing published data and the control lines representing the published topics (these topics are data, combined with some type of command). The modules in the architecture consist of those related to the planning of the car, such as Perception, Prediction, Planning, and Routing, those related to the data the car is receiving, such as HD Map, Localization, and Storytelling, those related to the control of the car’s mechanical control, such as Control, Guardian, and CANBus, and finally those related to the external UI of the car, such as Monitor and HMI.

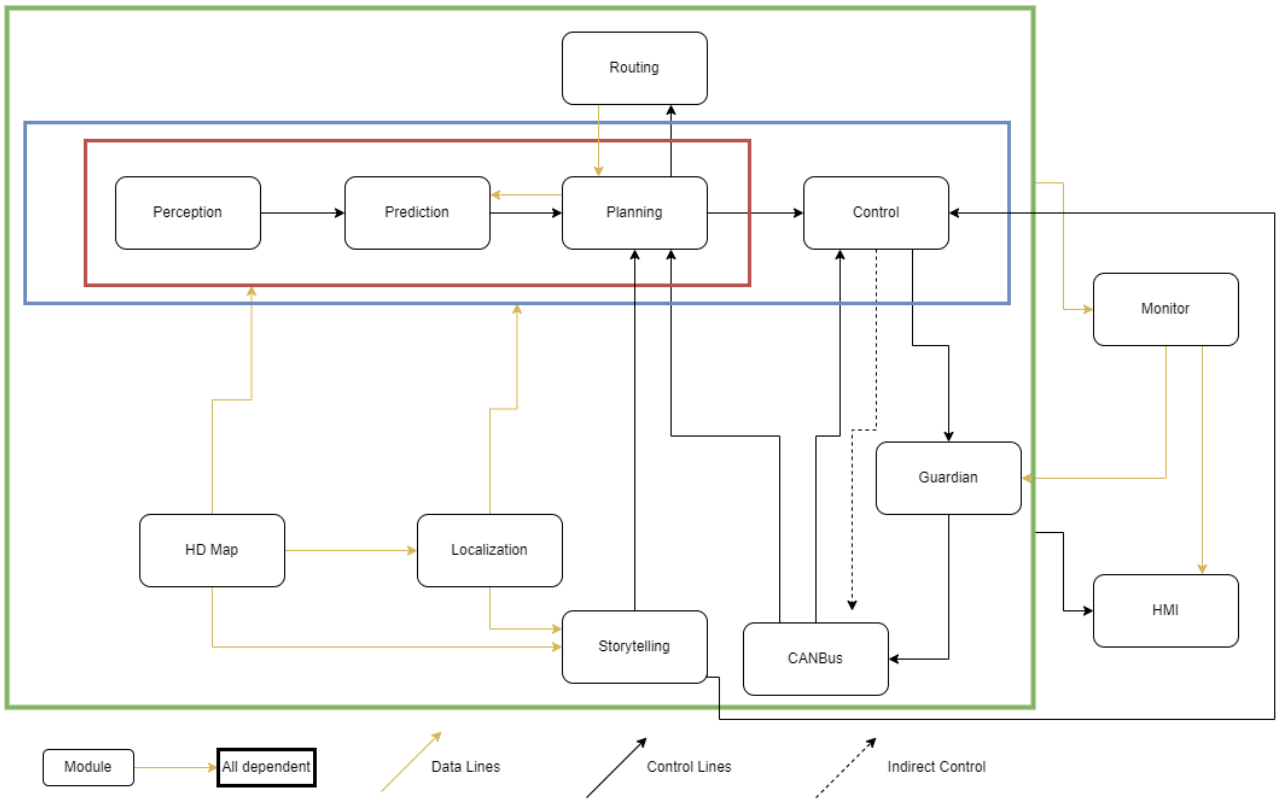

Figure 1. Proposed Conceptual Architecture of Apollo 7.0

Subsystems

Perception

The Perception module is responsible for utilizing the data from various cameras, as well as LiDAR systems, to recognize, classify, and track obstacles. Each obstacle can either be classified as “ignore”, “caution”, or “normal”. With each obstacle, the Perception module performs preprocessing on the images, utilizes deep learning models (improved in Apollo 6.0 and 7.0) to identify obstacles, and finally performs post processing to predict the obstacle motion and positional information (heading and

velocity) relative to the location of the car. The Perception Module receives sensor information from published data from the Localization module, as well as obstacle detail from the HD Map module. Upon completion, the Perception module outputs a tracklist of obstacles with their heading, velocity, and classification information, as well as data related to traffic light detection and recognition. This output triggers the Prediction module, which is explained further below.

Prediction

The Prediction module is responsible for estimating the behaviour of all perceived obstacles within the tracklist created by the Perception module. This is done through creating a “container” for each of the tracked obstacles, identifying the current scenario as either Cruise (lane-keeping and following) or Junction (traffic lights or stop signs), evaluating the speed and path of each obstacle, and finally based on the evaluation, predicting the trajectory of the obstacle. The Prediction module is subscribed to the tracklist and traffic light data from the Perception module, the location information from the Localization module, and the car trajectory from the Planning module. This trajectory information is wrapped around the obstacle tracklist and traffic light data and is published to the Planning module, triggering the module in the process.

Planning

The Planning module is responsible for planning a safe and collision-free trajectory for the car based on the various information from various other modules, such as Perception, Localization, etc. The Planning module is intended to be implemented in a scenario-based fashion, allowing for a high level of modifiability in the event that the behaviour for a certain scenario must be changed (i.e. if the behaviour for making a left turn at an intersection with a traffic light must be changed, it would not affect the behaviour of the Planning module for other scenarios). This highly modifiable scenario-based approach also allows for the addition of new scenarios, such as the dead-end and three-point turn scenarios that were added in Apollo 7.0. The Planning module is subscribed to the obstacle and traffic light data from the Prediction module (which contains the data from the Perception module), the current route data from the Routing module, the current location data from the Localization module, the scenario data (stories) from the Storytelling module, the current chassis status from the CANBus module, and the current map from the HD Map module. Upon completion of planning, the Planning module will either request a new route from the Route module or send the plan to the Control module.

Routing

The Routing module is responsible for coming up with a route, given a start point (generally the current vehicle location) and an endpoint, which includes passage lanes and roads that the car will take to reach its destination. This route is generated upon the Planning module publishing a routing request, containing the start and endpoints, in which the Routing module will calculate a route using the supplied location information, and will then publish a routing response to the Planning module, containing the optimal route.

Control

The Control module is responsible for generating control commands using various algorithms based on the planned trajectory and current status of the car. These commands include steering, speed, throttle, etc. The Control module is subscribed to the Planning module for the current plan, the CANBus module for the current chassis status, and the Storytelling module for overall module coordination. The Control module then publishes control commands to the CANBus module for execution indirectly through the Guardian module.

Guardian

The Guardian module is an emergency module that will override the commands published from the Control module provided there is a detected crash of another module. Provided there is a crash, the Guardian module will adjust the control commands based on three scenarios: if the data it receives from the ultrasonic sensor does not include a detected module, the car will be brought to a slow stop, if the sensor is not responding, the Guardian module will immediately stop the car, and finally if the HMI informs the driver of an impending crash and they do not intervene within a 10-second window, the Guardian module will immediately stop the car. The Guardian module is subscribed to module data from the Monitor module which will contain information on any detected crashes. In the case of a module crash, the Guardian module will publish its own control commands to the CANBus module. Otherwise, it will allow for the normal flow of the control commands from the Control module.

CANBus

The CANBus module is responsible for executing the control commands received from the Guardian module (or indirectly from the Control module), as well as collecting the car's chassis status. The CANBus module will function in either 2 modes: OnControlCommand or OnGuardianCommand, depending on the current control commands being received. Finally, the CANBus module is responsible for collecting and publishing all chassis related information to the modules that are subscribed to it, such as Control and Planning.

HD Map

The HD Map module contains the data of the map in which the car is currently driving (including roads, buildings, etc.). The HD Map Module is subscribed to no other module and publishes the map to the Perception, Prediction, Planning, Localization, and Storytelling modules.

Localization

The Localization module uses two different modes to aggregate data and locate the autonomous vehicle. The two different modes are Real-Time Kinematic Localization mode—which allows the Localization module to use both GPS and inertial data to calculate the car's location—and Multi-Sensor Localization mode—which relies on GPS, inertial sensors, and LiDAR sensors to calculate the car's location. The Localization module receives the published current map from the HD Map module and publishes an estimate of the car's location to the subscribed modules (Perception, Prediction, etc.).

Storytelling

The Storytelling module is a high-level Scenario Manager that is responsible for coordinating cross-module action by running various planning scenarios. These planning scenarios, known as “stories”, are used to avoid sequential module execution, as these modules will all be able to follow the same story. The Storytelling module is subscribed to the Localization module and HD Map module for data on the car’s current position and the map, which are used for scenario planning. The Storytelling module then publishes these modules to subscribed modules, being Control and Planning.

Monitor

The Monitor module is responsible for tracking all information of the modules and hardware of the system. This information includes system health, data integrity, data frequency, latency stats, etc. The Monitor module is subscribed to all other modules, except for HMI, to evaluate their status. The Monitor module then publishes this information to the HMI, allowing the user to view the module and hardware status information and provided any module or hardware crashes, it will publish an alert to the Guardian module so that it can take the required action.

HMI

The HMI module, also known as DreamView, is a visualization of the current planned trajectory, car localization, and chassis status that is displayed to the user. This visualization allows the user to view all hardware statuses, turn on/off modules, and start the autonomous car. As well, the HMI provides debugging tools for the user. The HMI is subscribed to all modules, including the Monitor module, for the information it displays to the user.

Evolution of the System

The evolution of Apollo, primarily through its use of modules, is demonstrated across its versions through a variety of features. For example, with v1.0.0, Apollo was not able to perceive obstacles and lacked the Perception module entirely, meaning vehicles were not able to drive on any sort of public roads wherein it could interfere with traffic. Instead, in this earliest version, Apollo was only able to drive using only its GPS. Over time, however, Apollo introduced new modules which improved the experience and introduced important features. One such example is in v1.5.0, wherein the Perception module was released and allowed the system to register real, outside data to be used for control. This also resulted in the development of the Prediction module to use Perception data for predicting future movements, and consequently the Planning module to use Prediction data to plan the vehicle's movement along a given route.

With the introduction of new functionality through modules, Apollo gains new functionality. However, new functionality yields new considerations about the system and the interactions between the modules themselves. After the introduction of appropriate modules to enable autonomy in the vehicle, safety and security concerns arose, leading to the development of the Monitor and Guardian modules to examine the

system diagnostics and react appropriately. With each incremental addition supporting more complex features, capabilities of the system increase and demand for supporting functionality as a consequence. As an example, the Guardian module was introduced to combat system failure only once Apollo was capable of autonomous driving, which could result in testing and/or usage in real-world environments like in traffic. Prior to Apollo's autonomy, the system itself was unfit to drive in traffic, and so safety concerns arose with the newfound capabilities of the system.

Concurrency

Apollo implements concurrency through the Cyber Runtime (RT) environment, which is a centralized, parallel computing solution that was developed to support the Apollo platform. Apollo Cyber RT not only allows the plug-and-play functionality of the individual modules but also includes a unique runtime environment that provides efficient interprocess communication, which is necessary to achieve concurrency in the system with its many moving parts. Cyber RT demonstrates high concurrency to allow for continuous and uninterrupted communication, which is essential in autonomous driving for safety in order to react proactively to potential hazards.

Concurrency is a necessity in the application of autonomous driving, with the possibility to harm passengers if subsystems fail or do not interact accordingly. In the context of the Apollo system, concurrency is needed to allow simultaneous computations and consequent reactions. In the simplest case, the process of Perception and Control, ignoring middle-man subsystems, must occur concurrently to allow perception and processing of the vehicle's surroundings and appropriate reaction by the vehicle itself. In reality, this concurrent process involves more than just the Perception and Control subsystems, but a majority, if not all the subsystems present in the Apollo system for a range of safety and functional needs.

Touching more specifically on the subsystems, concurrency in the system can be further examined. To begin, the Prediction module relies heavily on the perception module to take in data from surroundings and actively predict scenarios. Afterwards, data from the Prediction module, as well as other modules including Routing are passed to the Planning module, which makes decisions that are both efficient and safe. While these subsystems seem to be synchronous but not necessarily concurrent, the Control module, which is responsible for vehicle control, demonstrates a need for concurrency with these modules, as it must be capable of operating independently; if the dependent modules experience a bottle-neck in performance and are delayed, the Control module must appropriately maintain control rather than waiting for the processing of other modules in a synchronous manner. This remains true especially when examining the Storytelling module, which will plan a scenario by using the Control module while examining incoming data and concurrently determining if the scenario should be maintained. Besides maintaining control over the vehicle hardware under computational strain, the Guardian module is an essential safeguard mechanism that is dependent on the other modules, such that it appropriately controls the vehicle to safety under system failure when a module stops functioning. This is a necessity in the system and must be

concurrent with all other modules to ensure that system failures can be caught as they arise. With respect to the human-computer interactions, the Monitor module reads system diagnostics by concurrently reading data from all other models, including both software and hardware modules. This data is then displayed through the Apollo HMI. The Monitor must be concurrent with the rest of the system to appropriately retrieve accurate data and pass it to the Monitor, displaying real-time diagnostics to the user.

Division of Responsibilities for Developers

Based on the conceptual breakdown of the Apollo system, it can be inferred that the developers responsible for the implementation of the system were broken down into subteams. Since the overall software architecture of the system can be characterized as a Publish-Subscribe style, each of the modules can be independently integrated into the system. Without a strict reliance on the total ordering of module development (that is, requiring modules to be developed in a specific, synchronous order), developers could be broken down into subteams to work on any specific module(s) that can be done concurrently. An example of this is comparing the development of the Perception module against the development of the Localization module. The two modules themselves are not entirely related with one another and are both used in a loosely coupled fashion through the planning module and other intermediary subsystems (e.g., Perception interacts with Prediction, whereas Localization interacts with Routing).

With respect to the division of developers into smaller subteams to work on the development of individual modules, there is additionally a hierarchy to the priority of modules to be implemented, with others being put into a backlog for the future. This is demonstrated clearly through version updates to Apollo, and the features introduced in each. Since v1.0.0, functionality in terms of the Control and HMI modules were seen as essential to the system and were shipped with the platform on release. Approaching later versions, new functionality corresponding to good features that were not seen as essential as the described modules are introduced. Perception and appropriate Planning modules were not introduced until v1.5.0, whereas other features like the Guardian module were implemented later (v3.0.0).

Despite the division of responsibility across developers throughout the development of the Apollo system, all developers share specific responsibilities and must have specific knowledge in common across the developers. Importantly, not only are developers responsible for appropriate documentation and styling adhering to established rules, but developers must also possess a fundamental understanding of the system and conceptual architecture. It is important for all active developers to understand the architecture to acknowledge how all the modules are used, and how they may interact. Cyber RT, the custom-made runtime environment made for the Apollo system must be understood by developers to ensure appropriate interactions and deployment of the system. Cyber RT is standardized and enforced in the system as a high performing solution for the computationally-expensive autonomous driving that must be accommodated.

Diagrams

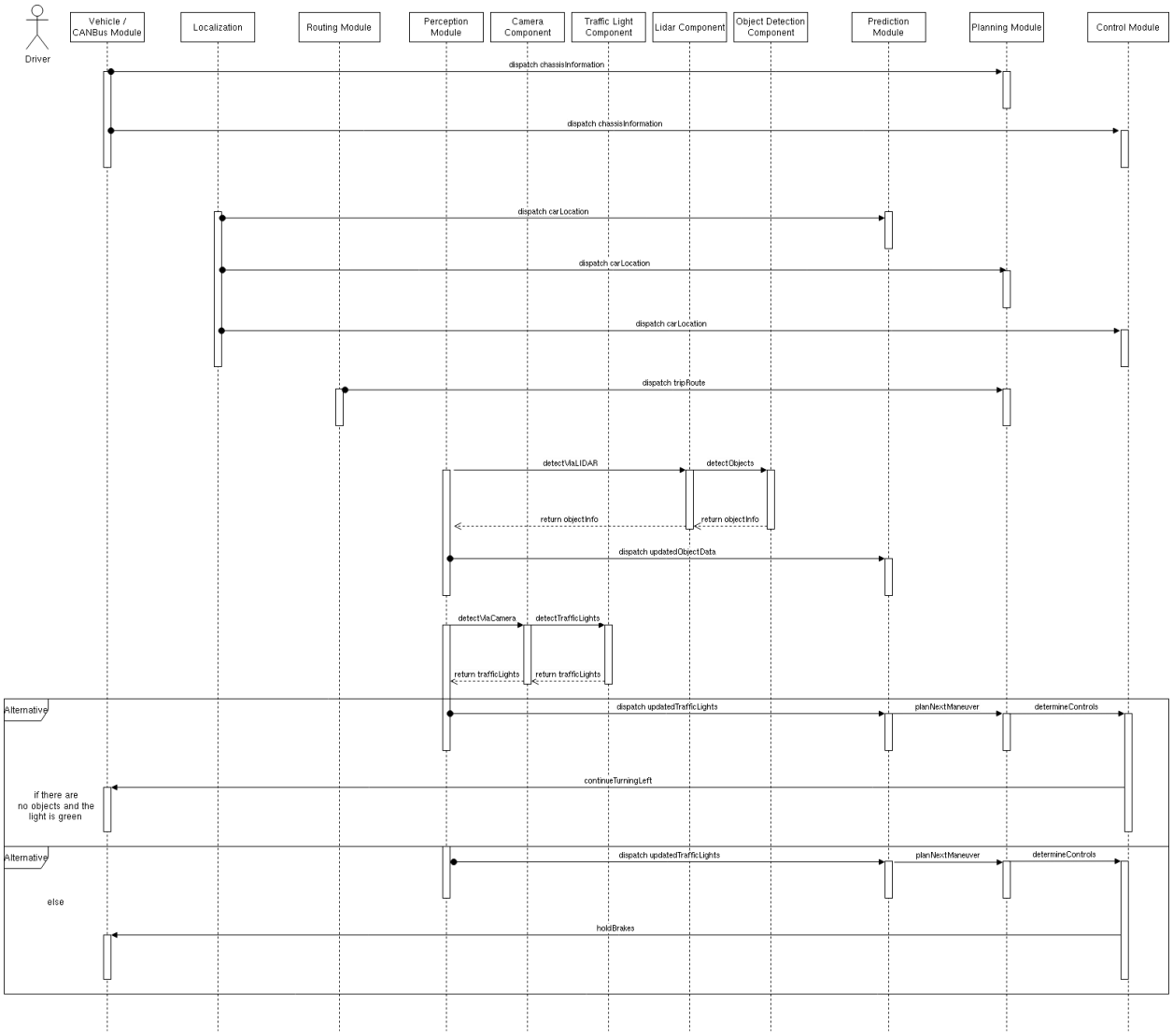

Figure 2. Sequence diagram for a use case involving a left turn at a stoplight.

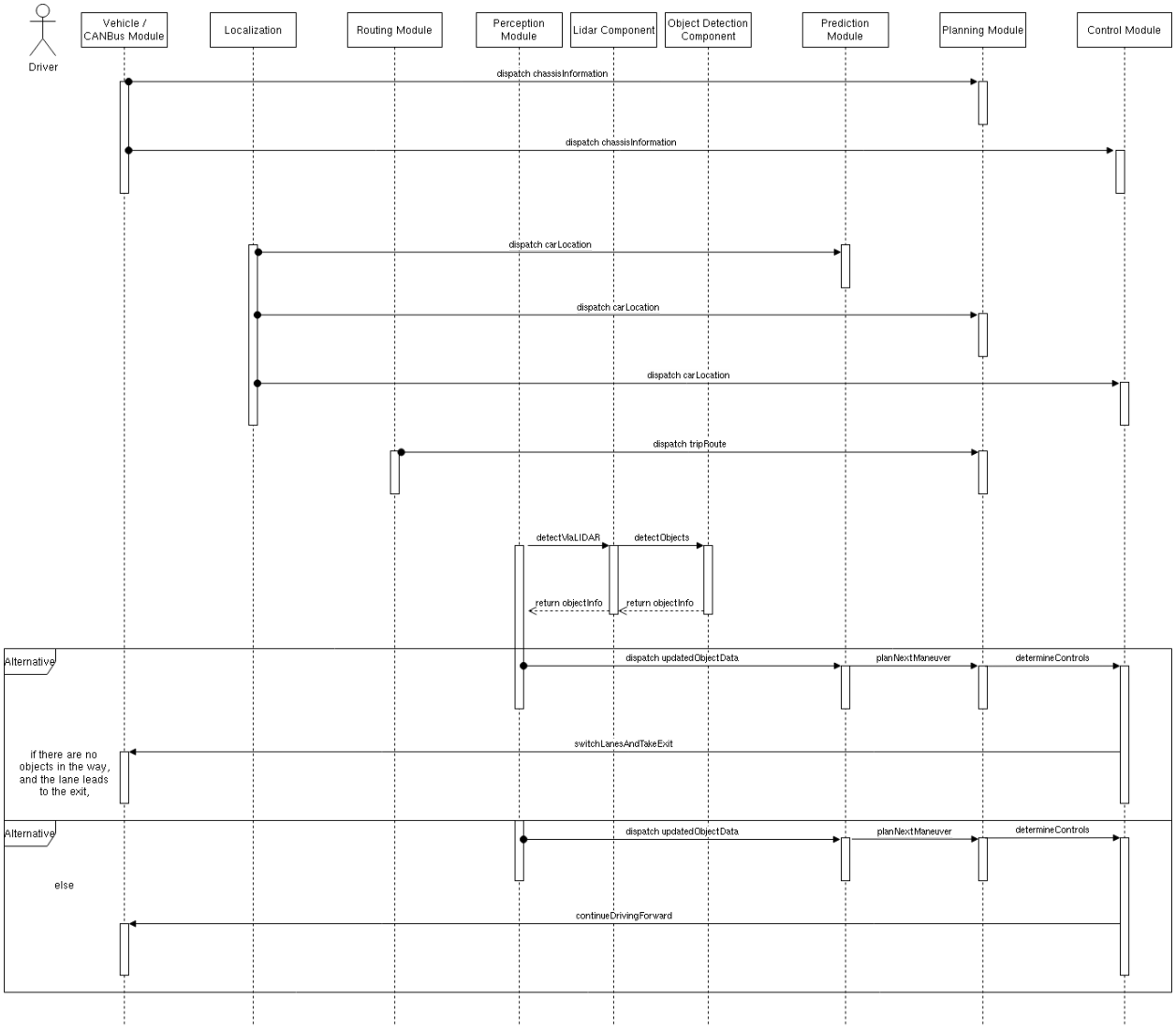

Figure 3. Sequence diagram for a use case involving a lane change at a highway exit.

External Interfaces

The primary external interface in the Apollo system is the HMI, also referred to as DreamView. This interface is a web application that is intended to provide important information to the user of the autonomous vehicle. Data flows from the Monitor module to the HMI visualizing the current output of relevant modules. In the event of hardware or module failure, the user will be informed through the HMI which provides debugging tools. Aside from viewing hardware and module status, the HMI also allows the user to turn on/off modules and start the autonomous vehicle.

Due to the inherent dangers of driving, the Apollo system needs a safe way to monitor and test scenarios before physically being in these situations. This is dealt with using Dreamland, Apollo's web-based simulation platform. Dreamland allows users to simulate a variety of scenarios of varying difficulties with the ability to change variables

such as road types, obstacles, driving plans, and traffic light states. This interface also provides functionality to run multiple scenarios in parallel for more extensive testing. Whenever a scenario is run on Dreamland, there is a 3D visualization to show how the system performs. There is also an automatic grading system that is run for each scenario judging 12 metrics including collision detection, red-light violation detection, and speeding detection. Each day, the current Apollo Github repository is run against all sample Dreamland scenarios to document how safe the current build is.

These external interfaces act as the main source of information for users and developers. Users that are physically running the Apollo system have access to the HMI, however, anyone can experiment with the Dreamland simulations. Dreamland is particularly useful for open-source developers wanting to contribute. While it can be used with the most up-to-date build, it can also be used for testing new versions that are being worked on. This ensures that developers can have confidence in their contributions, knowing that these external interfaces improve the overall safety of the system.

Use Cases

The inner workings of the Apollo architecture can be further illustrated through the following use cases. These use cases will demonstrate the various interactions between critical system components.

One use case involves a user that wants to make a left turn at a stoplight. To complete the task, the car must automatically check for incoming traffic, and proceed when the path is clear and the traffic light is green. Before proceeding with the left turn, certain modules must have access to existing information such as the local positioning of the car, the status of the car’s chassis, and information about the trip route that needs to be taken. As shown in Figure 2, the CANBus, Localization and Routing modules publish bits of information to other modules that make use of it. This includes the status of the car's chassis, the car's location, and the trip route, respectively. After the Prediction, Planning and Control modules receive the information they need, the car can proceed with the left turn. First, the Perception module triggers other sub-components to detect various objects in view. The camera component sends image information to the traffic light detection component to detect the state of traffic lights. The LiDAR component sends information to the object detection component to find moving objects, or any other objects that may be hazardous. Once the Perception module has all of the newly updated information, a message is published to notify other modules about changes to the scene. Information about the objects in the scene are then passed to the Prediction module. From there it will predict the future state of objects in the scene. The new predictions then get sent to the Planning module along with the data from the Perception module. From here there are two possible scenarios. Either the stoplight is green, and there is no incoming traffic, meaning that the car can continue with the turn, or the car can not advance due to a red light or incoming traffic. The Planning module determines what to do depending on the scenario, and then sends data to the Control module. The Control module then translates the plan to commands that are sent to the

CANBus module to execute, and ultimately steer the car. In the case that the path is clear and the light is green, a command to continue with the turn will be sent. In the event that the path is not clear, or the light is red, a command to hold the brakes will be sent.

Another use case involves a lane change on the highway to take an exit. As shown in Figure 3, the sequence of events between the system components remains largely the same. Localization, routing, and chassis information gets sent to the modules that need that information. In the perception stage, we mostly care about the LiDAR component, since we want information about other moving objects (i.e. other vehicles) around the car on the highway. Similar to the previous use case, information about moving objects is sent to the Prediction module to anticipate the future state of the scene, then that information is sent to the Planning module. Then, there are two main scenarios that the Planning module can make. Either the car has clearance to make a lane change, or there is no space or it is too dangerous to make a lane change. The Planning module can also make use of the data previously received from the Routing module to determine if it makes sense to make a lane change. The Planning module can decide to make a lane change if the exit leads to the destination. The Planning module sends its data to the Control module to translate the plan into actions that the CANBus module can execute. Either it sends a command to turn into the appropriate lane, or it sends a command that tells the car to keep driving forward.

Conclusion

In conclusion, we discovered that the conceptual architecture of Apollo's Open Software Platform is comprised of twelve main modules that work together in a "Publish-Subscribe" (or "Implicit Invocation") style to power autonomous vehicles. We documented the purpose and functionality of these modules, how both users and developers may interact with them, as well as their relationships with each other as they work together in the system. Our findings have allowed us to draw conclusions on how the system has evolved over time. We have also discovered the importance of concurrency and how CyberRT implements this, along with how a development team would delegate the different subsystems amongst themselves.

Apollo is a truly revolutionary, one of its kind platform. Its open-source nature enables developers and curious users to experiment with Apollo's extensive features and modules on their own, something that not many other autonomous vehicle softwares are capable of. Our team is excited to see what may come out of Apollo in the following years and believe that the platform has the potential to dominate the autonomous driving market.

Limitations and Lessons Learned

While collaborating on this assignment, our team faced a few limitations regarding the information available about the Apollo Platform. It was difficult to find the complete software architecture for Apollo 7.0 and we frequently had to search through

the GitHub repository to gather information from previous versions. We ended up referencing the most recent software architecture available (Apollo 5.5) to supplement areas where we could not find information related to Apollo 7.0. Additionally, there were inconsistencies in the architecture in terms of modules that were mentioned in previous versions but seemed to be excluded from the 7.0 architecture on Apollo's website. This included the Routing, CANBus, Monitor, Guardian, and Storytelling modules. It is unclear whether these components were integrated or combined with other modules in the newest version of Apollo, but we decided to keep them as part of the conceptual architecture because they perform key functionalities in connecting the various parts of the system.

Despite our difficulties, our team had many positive takeaways from working on this project. We all learned to appreciate the importance of open communication and were able to work backwards from the deadline in order to distribute the workload and plan our time accordingly. We found that scheduling weekly meetings was the most effective in moderating our progress and was a great opportunity to come together to share ideas, research, and determine an actionable plan moving forward. We decided to work on the assignment in stages, where we would first divide up different questions to research in order to familiarize ourselves with the architecture before splitting up various components of the report and presentation based on our findings. Learning about the Apollo Platform also allowed us to discover the world of autonomous vehicles and all the various components that are involved. Our team was able to familiarize ourselves with the structure of the GitHub repository in order to find the information we needed. Going forward, we will continue to research and take detailed notes on the architecture and where to locate things in the repository to save time, as well as ensure that every member on the team has a concrete understanding before proceeding with the report.

Data Dictionary

Autonomous vehicle: A vehicle that can drive and perceive its environment without a human controlling it.

Conceptual architecture: A very high-level organizational view of a system, highlighting modules and the relationships between them.

Control flow: The logical path between modules during execution of software.

Data flow: When data is passed from one module to another.

Module: A bundle of code composed of many related functions contributing to a common feature.

Open-source: A code-base that is publicly accessible to be used or contributed to.

Publish-Subscribe: An architecture style involving loosely-coupled collections of modules. Each module is responsible for an operation, which may enable other modules in the process.

Naming Conventions

GPS: Global Positioning System

HMI: Human-Machine Interface (DreamView)

LiDAR: Light Detection and Ranging

UI: User Interface

References

1. https://github.com/ApolloAuto/apollo
2. https://apollo.auto/
3. https://m.futurecar.com/4866/Chinas-Baidu-Inc--Opens-its-Apollo-Go-Robotaxi-Service-in-Shanghai
4. https://opensource.com/article/18/4/apollo-open-autonomous-vehicle-platform