February 20th, 2022

Apollo Conceptual Architecture Report

Authors

Alexander Nechyporenko (15ann2@queensu.ca - #20021759)

Kennan Bays (18khb5@queensu.ca - #20167866)

Joshua Kouri (18jak16@queensu.ca - #20158940)

Joshua Tremblay (20jkt1@queensu.ca - # 20247327)

Mason Choi (18gc23@queensu.ca - #20165470)

1.0. Abstract

This report contains our derivation of Apollo v7.0’s requirements (including alternative derivations), our conceptual architecture of Apollo (including a subsystem breakdown, an analysis of the subsystem dependencies, and two sequence diagrams), and a discussion of the limitations we faced while analysing this architecture along with the lessons we learned along the way. The software of Apollo has many different modules. Each module is essential to meeting the functional and non-functional requirements of the system, such as an accurate perception of the environment, the ability to simulate the current environment, or security. Its overall purpose is to help advance the development of autonomous vehicle technology.

In order to figure out the derivation and conceptual architecture of this software, we mainly looked at documentation from the Apollo GitHub repository as well as the company website. To figure out the dependency, our group entered the individual module files and tracked the inputs and outputs of each component. This in turn allowed our group to determine a diagram for the overall software architecture. The most significant subsystems were Perception, Prediction, the Scenario Handlers, Location Services, Control, and Monitoring Systems, as well as two modules called CanBus and Guardian. Additionally, studying the requirements was used to generate a use-case diagram while the system architecture was used to create two sequence diagrams.

The architectural style that we decided on for Apollo is process-control. The controller component in this system is the Control module, and it obtains variables from the perception, prediction, and planning modules.

2.0. Introduction

The report will examine the conceptual architecture of Apollo, an open-source autonomous vehicle driving software, including its subsystems, dependencies between subsystems, and functional and non-functional requirements. The report consists of several parts. It opens with the derivation process, which discusses the software’s requirements alongside some notable use cases. Next, a breakdown of the conceptual architecture is provided. Each subsystem along with its components and dependencies is explained in detail. A description of limitations faced in the construction of this conceptual architecture will be provided, as well as the lessons that were learned from it. Finally, a glossary of important terms and naming conventions as well as a list of references are attached to the end of the report.

Apollo’s derivation was heavily influenced by its ability to meet its Functional and Non-Functional Requirements. The list of Apollo’s Functional Requirements is as follows: accurate perception of the environment, ability to simulate the current environment, accurate map and localization, and intelligent controlling. These Functional Requirements are essential to delivering a smooth driving experience while prioritizing a user’s safety and health. The Non-Functional Requirements are equally critical to the driver’s safety, with those requirements being Concurrency and Security. If the systems of the vehicle are incapable of operating at the rate necessary on the road, where time is a significant factor, then the user will be in extreme danger while operating the vehicle. Unfortunately, the subsystems most likely to be affected by a lack of concurrency are those which determine the driving trajectory of the vehicle, which may leave the user exposed when suddenly encountering many obstacles. Additionally, a lack of security would pose a safety risk to users as their vehicles could be significantly hampered by a cybersecurity threat. How this threat impacts the user is variable, but the harm that could be caused by this threat is immense.

Notably, the Apollo software is incomplete. The success of the program has been limited to testing roads, and it is not fully ready for the real deal. Thus, the conceptual model proposed will likely be outdated soon. In the most recent release alone, new subsystems and modules were added. Currently, the general conceptual architecture can be understood by comparing Apollo to a human driver, who has the slight advantage of knowing every road ahead of time. When it needs to see its surroundings, the Perception subsystem acts as its eyes and gathers information about the environment. Once Apollo has seen, it will need to understand. The Planning and Scenario Handlers subsystems act as Apollo’s brain, allowing it to understand how the world around it is moving and how to proceed in response to that information. The Control and CanBus subsystems act as Apollo’s nervous system, transferring signals to the wheels and engine to turn or change speed. The Location Services are its memory and the Monitor is its self-awareness. Finally, Guardian is the ability for Apollo to detect if it is having a heart attack, and if so, to then stop moving as quickly as possible. These organs and tissues transform Apollo from an average program to an exceptional driver.

3.0. Derivations

Apollo is an open-source architecture with the intention of facilitating advancements within the autonomous vehicle industry that is set to dominate our future roads. To better understand the derivation of such a complex architecture that is endeavouring to create an infrastructure for future autonomous driving, we will explore the functional and non-functional requirements of such a system. In extension to that, we will see what alternatives to these requirements are applicable that may have not been included within the existing architecture of this system.

3.1. Functional Requirements and Use Case

Apollo categorizes their main functional components into accurate perception, simulation, HD map and localization, planning, and intelligent control[1]. Perception involves sensing and obtaining data from the vehicle's environment. Then, that data is processed to predict the behaviour of nearby obstacles (such as movement and positions), and to determine the status of the traffic lights to ensure proper reactions[2]. For simulation, Apollo describes this feature as the ability to virtually drive an autonomous vehicle in order to further test, validate, and optimize the system[1]. This feature benefits both developers and users. For developers, this is an opportunity to gather more data to further improve their system. As for users, they are able to test the system and become familiar with it before deciding whether to implement it on their own vehicles. Next, Apollo has created an HD map to determine the road environment and related objects. Then, through localization, the position of the vehicle's location is accurately identified along with the help of the data from the HD map[1]. These functionalities are useful to the vehicle as it allows it to perform any decisions with precise information. Planning creates a trajectory for the vehicle to use with influence from its predicted data, and the behaviour found in the perception functionality[1]. Planning then allows the control component of the vehicle to use the trajectory produced. Control adapts to many important situations such as road conditions and speed limits[1]. This creates an experience that mimics actual human behaviour in response to those conditions.

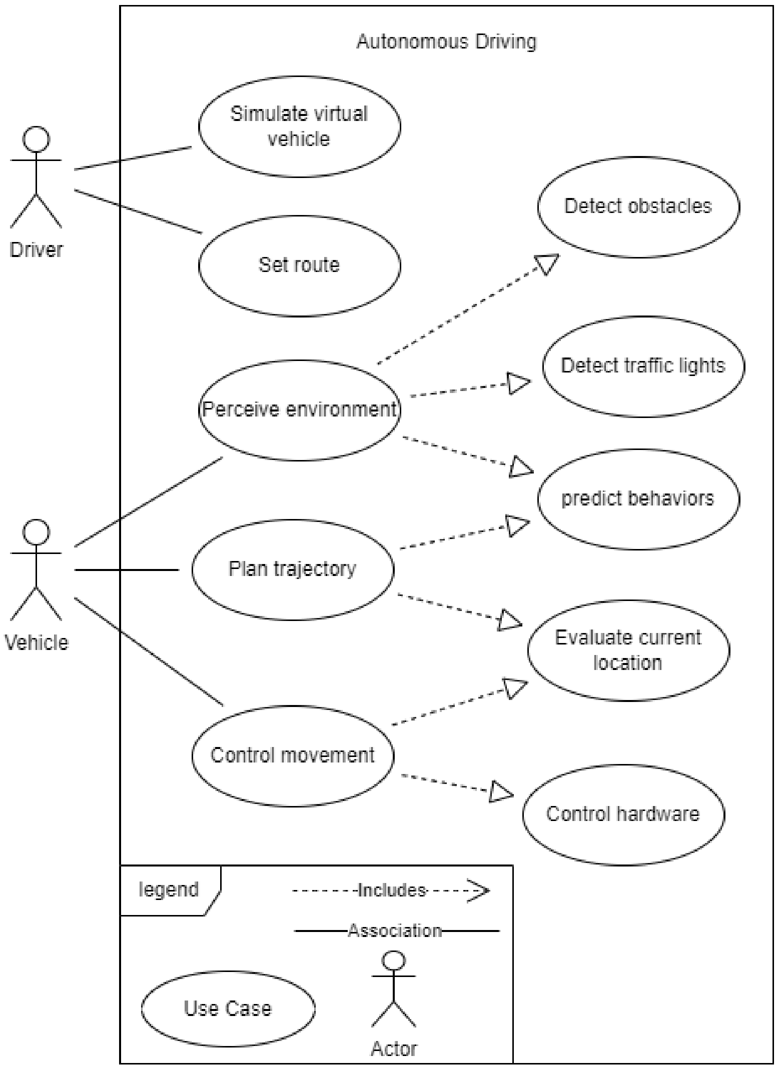

Figure 1: Use-Case Diagram

The use-case diagram to the left (see *Figure 1*) portrays the interactions that a vehicle and its driver can have with the autonomous driving system.

An alternative derivation of the functional requirements of Apollo can be applied through a generic view of autonomous driving. The system is categorized into perception, decision and control, and vehicle platform manipulation. The perception of an autonomous vehicle relies on the detection of data surrounding the vehicle and the resulting interpretation of that data. Decision and control define the way the vehicle can move in its environment according to its route and location, environmental events, and state of its system. Vehicle platform manipulation allows the vehicle to move with the information gathered from decision and control[3]. From these descriptions, Apollos' components can be applied. Perception becomes simpler. There is no implementation of a prediction element through this derivation. Without it, detecting behaviour becomes less accurate. Next, decision and control would encompass HD map and localization, and planning. It would be beneficial to keep these sections separate as they accomplish different tasks for the system. Finally, control applies to vehicle platform manipulation and accomplishes the same task. Simulation is the only feature that is missing in this alternative. However, its presence is important to users as it provides a glimpse into the technology before implementing it in their vehicles.

3.2. Concurrency, Security, and the Alternative of Portability

For such an impactful project that aims to accomplish something as momentous as the facilitation of autonomous driving, there are a couple of crucial non-functional requirements that need to be taken into consideration. At the helm of it, there are two foundational ones, concurrency and security. Given that an autonomous driving system must support many functionalities that need to work in conjunction, where the failure of even one such system could lead to irreparable consequences, concurrency is of absolute necessity. For an instance, the system must support computer vision technology via various sensors such as LiDAR to perceive the roads and then have another component to make assessments based on that information, while another system actually controls the car as a consequence of that information[1]. What this implies is that all of these various components that allow autonomous driving to work must communicate with each other at all times, both asynchronously and synchronously. From a security perspective, one must understand that there is a lot of data processing for autonomous vehicles[4]. Given that the data is sensitive and that it impacts not only the occupants of the car, but also others, such as pedestrians and other car drivers, it is absolutely imperative to keep the data secure. Be it to protect the lives of those out on the road from hacks, or to keep the data secure from being accessed in public due to it containing sensitive information.

Diving deeper into concurrency, Apollo has developed a framework called Cyber RT which enables developers to have a centralized system that manages parallel computing, utilization of high concurrency tasks execution, as well as efficient data output[5]. Given the nature of the problem and the difficult scenarios that arise when driving, the system needs to communicate data efficiently, with a robust mechanism for parallel data processing. Since Apollo aims to provide other autonomous driving developers with a system that facilitates the implementation of driverless cars, Cyber RT needs to provide a baseline efficiency of data transferring, while also giving the flexibility to developers to implement new components that will work in conjunction with the system. Hence, the concurrency of the system is forward thinking in the sense that it plans parallelism and data transferring with the expectation of new modules/components being added to it[5].

From a security perspective, given the delicate nature of maintaining safe driving and the utilizing of huge data sets, security is an essential aspect of Apollo. The security of the system is not exclusively important for maintaining the integrity of the software, but to also maintain the safety as others could be impacted by it on the road. Apollo, for this specific element, tackles safety by maintaining the legitimacy of applications installed to the software and monitoring any suspicious connections that might occur [6]. In essence, for this element, it makes sure that no malicious software is ever downloaded and that no hackers can gain access to the in-vehicle network. The architecture also has a system in place that makes sure no abnormal data or behaviour are recognized within the vehicle, such as having malicious control instructions being used within the system[6].

From an alternative non-functional perspective, one NFR that was not found in the documentation for Apollo which may be of importance is portability. While this software currently supports use for regular consumer vehicles, it could also become possible to expand the applicational use to focus on commercial uses as well. For instance, it would allow for utilization of autonomous vehicles within the Uber/taxi space, where the infrastructure supports commercial uses within those or other domains. This could serve a purpose like allowing for integrated services with app functionalities where the vehicle can get GPS orders to move to a specific location to pick someone up, and then take orders to move to a new location from within the third-party application. Hence, the system would support third-party application integration, since with the rise of autonomous vehicles, the demand would rise for new businesses to use these cars in different ways as well. This software should not be limited alone to the car, but also allow phones and laptops to gain access to it through external applications.

4.0. Architecture

4.1 Overall Architecture

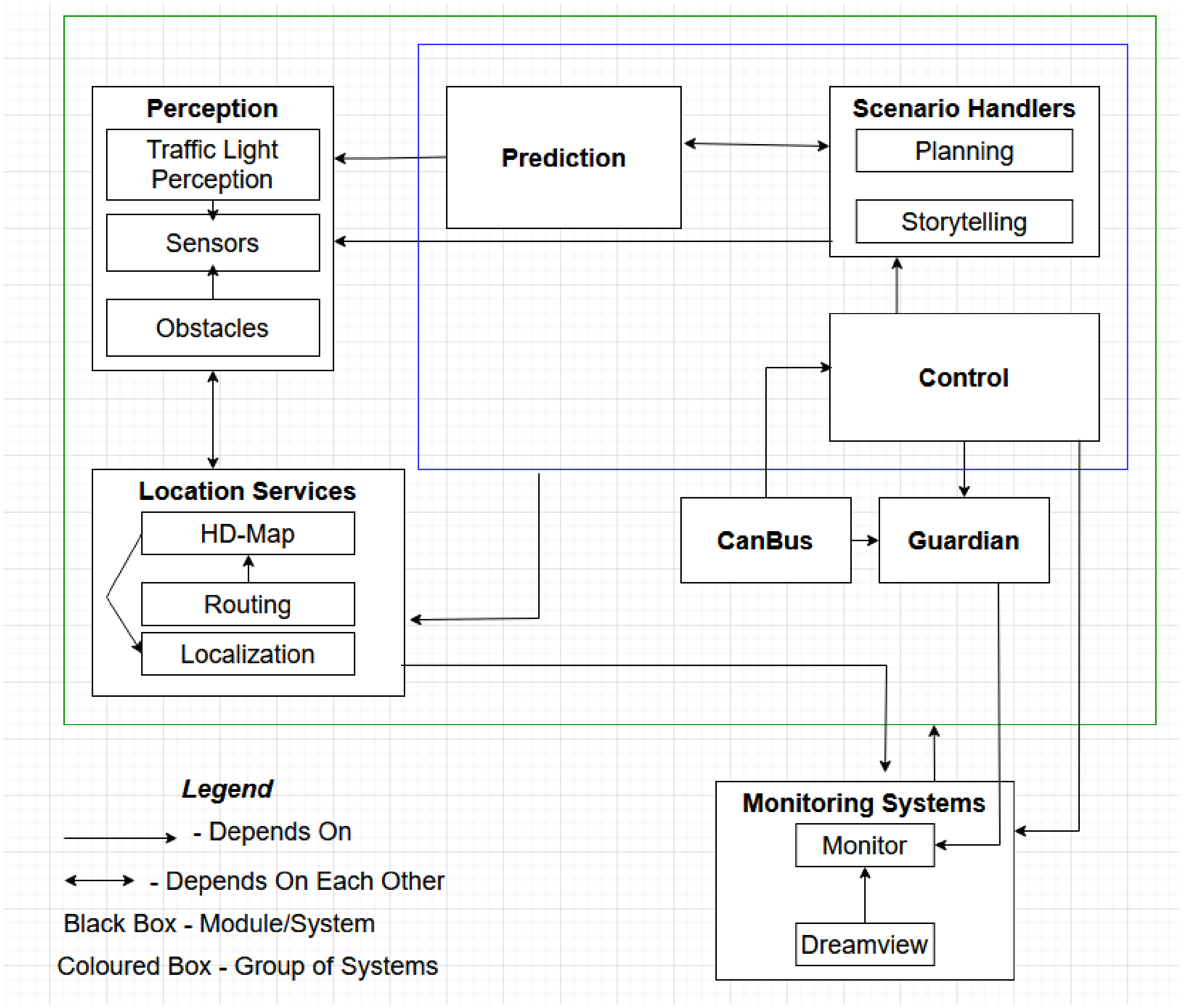

Figure 2: Overall System Architecture

Figure 2 depicts the final architecture. Location Services and Monitoring were grouped into their own subsystems based on similarities and dependencies of the components within them. In addition, we found that the primary module that the vehicle depends on is CanBus, which takes input from the control module and executes it.

Overall, the architectural style used for Apollo appears to be a close match to process-control. Process-control architecture is often used for real-time system software such as that belonging to automobile cruise control, so it makes sense that Apollo is using it as well. To be more specific, a feedback control system is being used, as information about various process variables are used in order to adjust the output of the system. The controller in this system is the control module, and it obtains variables from the perception, prediction, and planning modules.

4.2. Subsystem Breakdown

4.2.1. Perception

The Perception subsystem provides the context for all subsequent decisions made by Apollo by gathering information about the physical surroundings around the vehicle. The Perception subsystem contains three main submodules; Traffic Light Recognition, Obstacle and Sensors [8].

Traffic Light Perception Module (TLPM): The TLPM relies on cameras, a telephoto and wide-angle camera, to gather information in the forward direction of the car. As the car drives, the TLPM repeatedly queries the HD-Map module for upcoming traffic lights. If the module detects a traffic light, the traffic light will be projected onto the images captured by the cameras, allowing the vehicle to create a 3D environment from a 2D picture. Once a traffic light is detected, the TLPM will identify the state of the traffic light. The state of a traffic light could be in any of the following states: red, yellow, green, black, or unknown (unable to detect). The TLPM has a dependency on the Sensors module to send information to project the traffic light onto. However, it is important to note that the TLPM is only dependent on the camera sensors, since the other modules do not detect the light emanating from the traffic light [9].

Obstacle: The Obstacle module is used to detect, classify, and track objects. This module will attempt to predict the position and velocity of detected obstacles. This module is dependent on the Sensors module for information which is required to detect and track objects [10].

Sensors: The Sensors module is a collection of multiple unique perceptive hardware which collect information about the vehicle surroundings. Thus, the Sensors module is, to an extent, a bundle of smaller modules. The perceptive hardware consists of five cameras, two radars, and four LiDAR’s (Light Detection and Ranging). The output of these devices is used to recognize obstacles and combine their individual tracked objects into a final tracking list (list of objects to track) [10].

4.2.2. Prediction

The Prediction module using information gathered by the Perception and Scenario Handler modules to predict the behaviour of detected obstacles. The Prediction module has four functionalities; *Container*, *Scenario*, *Evaluator*, and *Predictor*. *Container* functionality stores data about perceived obstacles, vehicle localization, and vehicle planning. The *Scenario* functionality takes the data in

Container to check whether the vehicle is to continue cruising or encountering a junction, which impacts the predicted behaviour of objects around the vehicle. The *Evaluator* functionality predicts the path and speed for objects around the car. Finally, the *Predictor* functionality predicts the trajectory for objects around the vehicle. The summation of these functionalities will be projections of the trajectories of all objects around the vehicle [11].

4.2.3. Scenario Handlers

The Scenario Handlers are a pair of modules which govern how the vehicle reacts the situation it is in. Scenario Handlers receive information from the Prediction module to process and create a course of action for the vehicle to react to in an attempt to get to the location provided by the Location Services. The two Scenario Handlers are the Planning and Storytelling modules.

Planning: The Planning module aims to create a collision-free trajectory while progressing towards an end goal. The Planning module requires the Routing output to decide where to go, Traffic Light Perception module from perception, and the Prediction output. Thus, while the module is dependent on many other modules, the Planning Module is only dependent on a fraction of the output of the modules it is dependent on. The Planning module addresses how to act by creating “Use Cases” for all driving environments. This allows the developers to tweak reactions to specific environments without affecting other driving behaviour [12].

Storytelling: Certain environments are too complicated for the Planning module to address. Due to the Planning module heading in a modular development approach, a sequential based approach to these scenarios are no longer viable. The Storytelling module is used to handle the complex scenarios that the Planning module is subpar at addressing. The Storytelling module creates “stories” (complex scenarios) which can trigger reactions from other modules [13].

4.2.4. Control

The Control subsystem receives inputs from the Scenario Handler modules which it processes and sends to the CanBus. The Scenario Handler modules generates trajectories for the vehicle to travel. Using the car’s current status and the trajectory information, the Control system generates the commands which will be forwarded to the CanBus [14].

4.2.5. Location Services

The Location Services modules are used to understand where the vehicle is in relation to the world. Location Services contains three submodules: HD-Map, Localization, and Routing.

HD-Map: The HD-Map (High-Definition Map) module is repeatedly queried by the system for information about the surrounding road systems. An example use of this information is the projection of a traffic light from a 2D picture to a 3D environment. HD-Map has a dependency on the Localization module for information about the exact location of the vehicle [8].

Localization: The Localization module pinpoints the exact location of the vehicle. The Localization module will calculate the vehicle location by using either the RTK (Real Time Kinematic) based method or the multi-sensor fusion method. The RTK based method requires information from the GPS (Global Position System) and IMU (Inertial Measurement Unit). The multi-sensor fusion method also requires GPS and IMU to locate the vehicle but incorporates LiDAR into the calculation [15].

Routing: The Routing module generates navigational routes to reach an end location from a start location upon request. This module is dependent on HD-Map to provide a significant amount of information about roads, traffic lights, and other road data. The Routing module uses this information to calculate the optimal route to reach the end location [16].

4.2.6. CanBus

The CanBus subsystem provides an interface for the Apollo software to communicate with the hardware. Additionally, CanBus passes chassis information to the software systems. The CanBus is a critical piece of infrastructure in the Apollo software system. In the event a system failure is detected by the Monitor, the messages sent by Control to CanBus are stopped. Guardian will send a message to CanBus forcing the car to stop [7].

4.2.7. Monitoring Systems

The Monitoring Modules gather information from all other modules to ensure that all systems are functioning properly. Two submodules ensure that this information reaches the user; the Monitor module and Dreamview module.

Monitor: The Monitor module is the supervisor (hence, “monitor”) of the other modules. The Monitor module receives data from different modules and forwards this information to the HMI (Human Machine Interface) for the driver to check and make sure they systems are working properly [17].

Dreamview: The Dreamview module is a web application which satisfies the need for an HMI (Human Machine Interface) in the Apollo architecture. The Dreamview module provides three primary functionalities. Firstly, Dreamview outputs relevant autonomous driving module information for the vehicle user, such as the planned trajectory of the vehicle. Secondly, Dreamview provides an interface for users to view hardware status, turn off/on modules, and start the autonomous vehicle. Thirdly, Dreamview provides debugging tools. The Dreamview module has a dependency on Monitor to provide information about the system’s status [18].

4.3. Subsystem Dependencies

Most dependencies which exist between subsystems follow a rather linear pattern. The number of dependencies a subsystem has increases in relation to the completeness of instructions to be given to the autonomous vehicle at that point in the system. For example, early-on Perception has one dependency while Control and Scenario Handler, two later subsystems, have multiple dependencies. Monitoring, arguably the "last" subsystem in the series of processing an instruction, has the most dependencies.

CanBus → Control Dependency: The Control module creates instructions to be executed to the autonomous vehicle. The data is then outputted by the Control module and received as an input by the CanBus. Thus, CanBus has a dependency on the Control module to generate an input value [14].

CanBus → Guardian Dependency: CanBus has a dependency on the Guardian module which is identical to its Control dependency. Guardian seizes control of the system when a significant issue occurs. However, while seizing control of the system, it functionally acts the same as the Control module. Thus, this dependency is the same as the CanBus → Control Dependency [8].

Control → Monitoring Dependency: If the user wishes to take control of the vehicle and turn off auto-drive, the monitor will send a message to Control which tells it to stop sending messages to the CanBus [14].

Control → Scenario Handlers Dependency: This dependency is part of a semi-linear stream of the generation of an instruction for the autonomous vehicle. The Scenario Handlers generate a trajectory for the vehicle to follow. The Control module translates this trajectory into instructions to be forwarded to the CanBus. This dependency details the last step before an instruction is sent to the vehicle to be executed [12][13].

Control → Guardian Dependency: If the Guardian module sends an interrupt signal, the Control module will stop sending instructions to the CanBus. This dependency is the least used as it is only used as an emergency interrupt [8].

Guardian → Monitor Dependency: The Guardian module will only seize control of the system if the Monitoring subsystem detects an issue with a subsystem component. Thus, the Guardian module is dependent on the Monitoring subsystem to detect an issue with the system's functioning [8].

Location Services → Monitoring Dependency: The user can affect the driving path by interacting with the Dreamview software, which affects the Relative Map, a small component of Location Services [17].

Monitoring → All Subsystems Dependency: The Monitoring subsystem monitors the status of all other subsystems. Thus, this dependency is a series of status checks performed by the Monitoring subsystem [17].

Perception ↔ Location Services Dependency: This dependency is a two-way relation. Firstly, the Perception module is required to help localize the vehicle so that the Localization module can pinpoint the vehicle. The Perception module is dependent on the HD-Map to predict any upcoming junctions, which significantly adds identifying objects and, in particular, traffic lights [10].

Prediction → Perception Dependency: The Prediction module is dependent on the Obstacle submodule of the Perception module to locate objects for the Prediction module. The Obstacle submodule will provide a list of tracked objects for the Prediction module to calculate the location and velocity of [11].

Prediction ↔ Scenario Handlers Dependency: The Scenario Handlers are dependent on the predictions of the Prediction module since understanding the exact location and velocity of objects, such as an oncoming car, are important to the decision making of the Scenario Handlers [11].

Prediction, Scenario Handlers, & Control → Location Services Dependency: All of these subsystems are dependent on the localization feature provided by Location Services to perform their tasks accurately and effectively [8].

Scenario Handlers → Perception Dependency: The Scenario Handler requires a significant amount of information to properly respond to complex situations. The Scenario Handler submodules both require the TLPM submodule of the Perception module to help identify the location of traffic lights [10].

4.4. Sequence Diagrams

During the driving process, the TLPM repeatedly queries the HD-Map for any potential upcoming traffic lights. If there are any, the TLPM then requests image data from the camera sensor and proceeds to search it for any visible traffic lights. If a traffic light is detected, the TLPM attempts to identify its state, and then sends both its position and state to the Prediction module. The Prediction data is then fed into the Scenario Handler modules, and the resulting instructions are sent to the Control module, which enacts the required changes (such as slowing down for an amber/red light) through the CanBus. This can be seen in *Figure 3*.

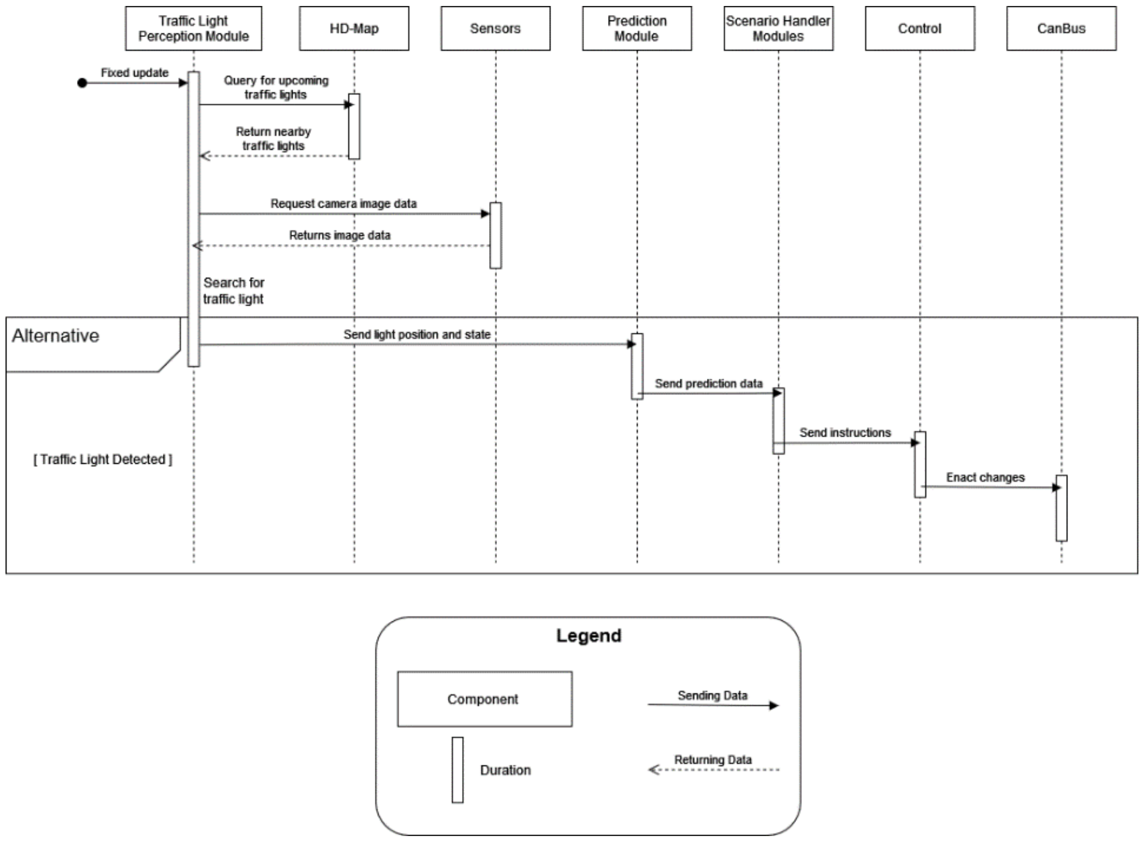

Figure 3: Sequence diagram details the software encountering and reacting to a traffic light

At all times, the Monitor is running and watching over all other modules. If it detects that a module has crashed, it triggers the Guardian module, which then disconnects the main Control module and takes its place. The Guardian module then checks sensor status to determine how quickly the vehicle should be brought to rest; if no immediate obstacle is detected, the vehicle will be slowly brought to a stop, but if an immediate obstacle is detected, the vehicle is slowed down rapidly, for safety reasons. This can be seen in *Figure 4*.

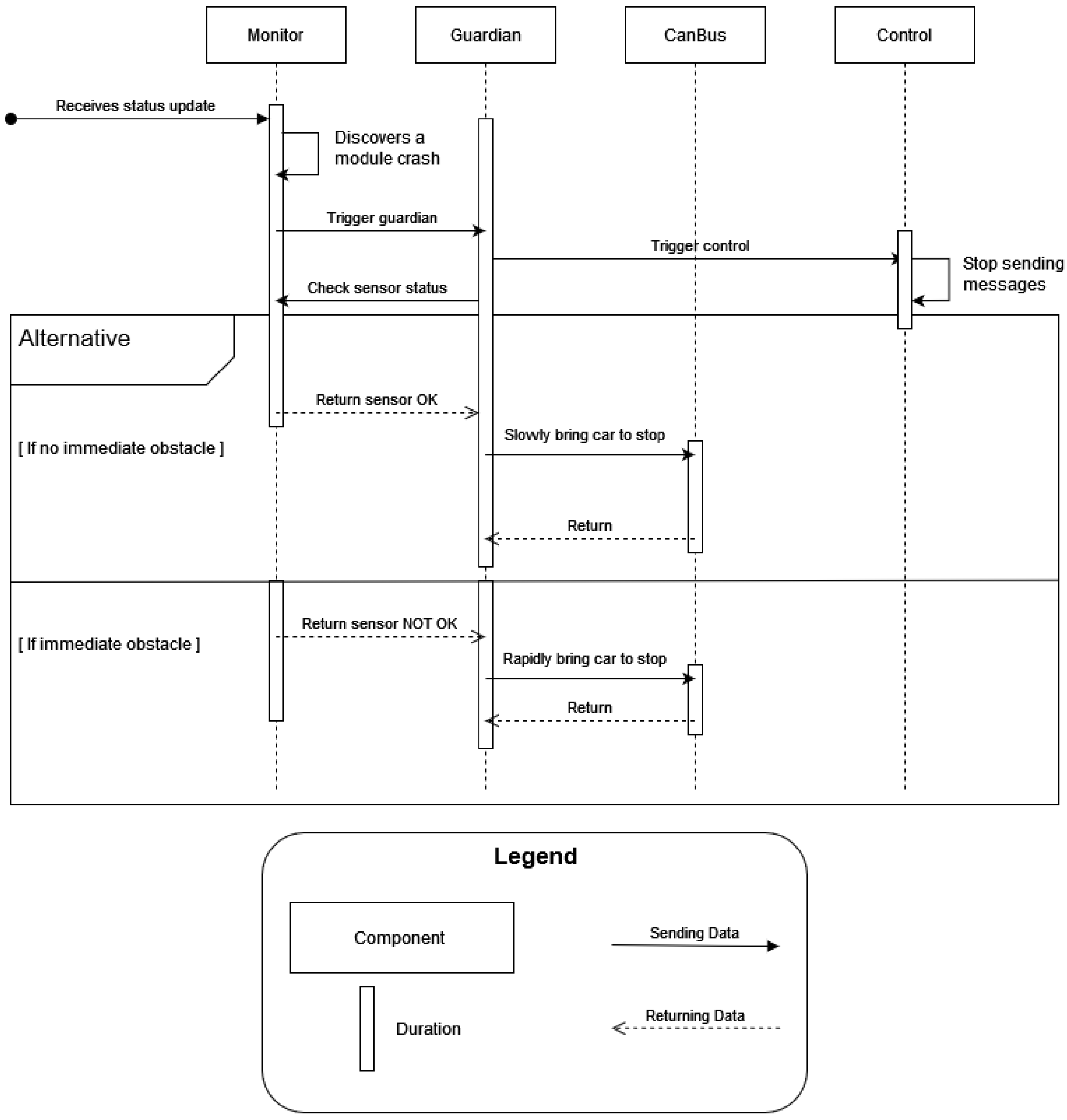

Figure 4: Sequence diagram for the Guardian module seizing control of the system to bring the vehicle to a stop

5.0. Limitations and Lessons Learned

During our writing of the report, our group encountered some limitations which became difficult challenges to overcome.

- The Apollo software is still very young, having been founded less than a decade ago. Thus, documents surrounding newer components, such as Guardian and Storytelling, are sparse.
- Apollo is being developed by Baidu, a Chinese company. Thus, a significant number of documents are written in Chinese, which our team could not understand. While most of these files appear to be translated copies of files which are provided in English, it is impossible to tell if any details are lost due to the language barrier.
- Apollo is currently on release Apollo v7.0. It is difficult to tell if the information within files created during earlier iterations of Apollo, particularly Apollo 2.0, are still applicable several generations later.

Our group endeavored to use previous study groups as a reference to ensure that our group avoided common issues. However, our group still learned several important lessons.

- Sequence Diagrams are event-driven diagrams which start at one point and follow a linear process. Autonomous vehicles are in a state of constant acting, which makes defining a start point difficult. Additionally, the vehicle is constantly reusing, saving, and discarding information, making it difficult to pinpoint a linear process to use. These challenges were overcome by intensely studying individual components to identify a single linear stream of actions.
- Our group learned the value of maintaining the quality of logs and information regarding a project. It was difficult to understand the project despite the hundreds of documentation files provided. Our group increased our ability to learn from documentation and its importance to incoming workers.
- Our group learned about how large development teams think about future developments. Developers left notes about how the construction and organization of certain modules were changing over time and it really educated us on how developers have learned from their experience (e.g., Planning module becoming increasingly modular by Apollo 7.0). For examples, developers struggled with the performance of the Planning module and thusly created the Storytelling module. This shows this innovation of the development team.

6.0. Conclusion

To summarize the conceptual architecture of Apollo, the system is designed to facilitate the advancements within the autonomous driving sector, where it provides the infrastructure needed to accomplish this momentous endeavour. In order to do this, Apollo is built around a robust system of components that support complex data-intensive processes. For instance, Apollo v7.0 enables accurate perceptions utilizing LiDAR and other radar technologies, along with simulating road conditions, or predicting different scenarios on the road. Underneath all of these components, there is a concurrency framework built to form a centralized system that supports dynamic and efficient concurrent communication between these components, while simultaneously allowing for the framework to be tuned for desired intentions by the developers (e.g., if new components need to be added to the framework, or to create new connections between components). Given the data intensity of all of these components and the need for efficient communication between them, the framework supports low latency and high throughput data transferring, along with effective parallel computing models.

Given that the infrastructure supports high power component concurrency, it is crucial that the system is also secure, not only from the standpoint of maintaining the integrity of the data or its exposure, but also for preventing malicious attacks from occurring with these autonomous vehicles that can put other lives in danger. The security protocols are multi-faceted and tackle different areas within Apollo, such as verifying the legitimacy of the applications installed, prevention of external malicious attacks, and checking if any abnormal data or instructions are found within the system.

The architecture system is built on the Process-Control style, given the nature of the feedback control system that autonomous vehicles are reliant on. Taking the different components, such as the perception, prediction and planning modules, the controller system takes the information of the modules in order to adjust the output of the vehicle system. Given the complex nature of autonomous driving and the utilization of a vast number of computing intensive components, it will be interesting to see how this is outlined within the concrete architecture.

7.0. Terminology

HD map: High-definition map which is a highly accurate map used for autonomous driving. Plays an important role for modules, such as perception, simulation, etc.

Localization: Robust method for determining the location of the vehicle, including within challenging locations such as urban downtown, highways and tunnels.

LiDAR: Stands for Light Detection and Ranging, which is a sensor which utilizes a pulsed laser to detect surrounding objects and their distance from the sensor.

CanBus: This executes instructions from the control module, while simultaneously also collecting data on the vehicle’s status to give feedback to the control module

TLPM: Stands for Traffic Light Perception Module, which is a module which obtains the coordinates of the traffic lights by finding the relevant information from HD map.

8.0. References

[1] *Apollo Open Platform*. Apollo. (n.d.). Retrieved February 14, 2022, from https://apollo.auto/developer.html

[2] *Apollo Perception.* Apollo. (n.d.). Retrieved February 14, 2022, from https://apollo.auto/platform/perception.html

[3] Behere, S., &amp; Törngren, M. (2015, December 29). *A functional reference architecture for autonomous driving. Information and Software Technology.* Retrieved February 14, 2022, from https://www.sciencedirect.com/science/article/abs/pii/S0950584915002177?via%3Dihub

[4] Kumar, N. (2020, December 27). *7 data challenges in autonomous driving.* Medium. Retrieved February 14, 2022, from https://medium.datadriveninvestor.com/7-data-challenges-in-autonomous-driving-e21d05dacc3a

[5] *Apollo Cyber RT framework.* Apollo. (n.d.). Retrieved February 14, 2022, from https://apollo.auto/cyber.html

[6] *Apollo Cyber Security.* Apollo. (n.d.). Retrieved February 14, 2022, from https://apollo.auto/platform/security.html

[7] *Apollo/modules/canbus at master · Apolloauto/apollo.* GitHub. (n.d.). Retrieved February 13, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/canbus
[8] *Apollo 5.5 Software Architecture.* Apollo. (n.d.). Retrieved February 14, 2022 from https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_5.5_Software_Architecture.md
[9] *Apollo Traffic Light.* Apollo. (n.d.). Retrieved February 14, 2022 from https://github.com/ApolloAuto/apollo/blob/master/docs/specs/traffic_light.md
[10] *Apollo Perception (2).* Apollo. (n.d.). Retrieved February 14, 2022 from https://github.com/ApolloAuto/apollo/blob/master/docs/specs/perception_apollo_5.0.md
[11] *Apollo/modules/prediction at master · Apolloauto/apollo.* GitHub. (n.d.). Retrieved February 13, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/prediction
[12] *Apollo/modules/planning at master · Apolloauto/apollo.* GitHub. (n.d.). Retrieved February 13, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/planning
[13] *Apollo/modules/storytelling at master · Apolloauto/apollo.* GitHub. n.d.). Retrieved February 14, 2022 from https://github.com/ApolloAuto/apollo/tree/master/modules/storytelling
[14] *Apollo/modules/control at master · Apolloauto/apollo.* GitHub. (n.d.). Retrieved February 13, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/control

[15] *Apollo/modules/localization at master · ApolloAuto/apollo.* GitHub. (n.d.). Retrieved February 15, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/localization

[16] *Apollo/modules/routing at master · ApolloAuto/apollo.* GitHub. (n.d.). Retrieved February 15, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/routing

[17] *Apollo/modules/monitor at master · Apolloauto/apollo.* GitHub. (n.d.). Retrieved February 13, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/monitor

[18] *Apollo/modules/dreamview at master · Apolloauto/apollo.* GitHub. (n.d.). Retrieved February 13, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules/dreamview