CISC 332

A1: Conceptual Architecture of Apollo

2/20/2022

Marc Brasoveanu	18mtb5@queensu.ca
Andrew Wilker	18ajw15@queensu.ca
Nathan Perriman	17nmp2@queensu.ca
Theo Raptis	17tjr5@queensu.ca

Table of Contents

Abstract

This report focuses on the breakdown of the conceptual architecture of the Apollo autonomous driving open source library. Through the analysis of several resources consisting mainly of the official Apollo documentation and source code we were able to construct a model of the conceptual architecture that makes up Apollo. We determined that the main architectural styles in use were the layered style, publish-subscribe style, and process control style. We predicted that multiple styles would be in use from the beginning because of the complex nature of the autonomous driving problem. The components in use such as the Map Engine, Localization, Perception, Prediction, Planning, Control, and HMI are all implemented within a combination of the different architectural styles.

Concurrency is a fundamental pillar in the Apollo architecture because of the nature of autonomous vehicles in which several processes are running in parallel, asynchronously. Using the publish-subscribe architectural style is one way that concurrency was implemented because components can wait until a certain condition is reached to engage in a specified behaviour. In the autonomous car problem the environmental context is always changing and needs to be analyzed at the same time as making adjustments to the vehicle movement parameters, this is a prime example of concurrency at work, specifically in the process control style.

We have identified several modules which make up the subsystems of Apollo including: Perception, Prediction, Routing, Planning, Control, CanBus, Localization, DreamView, Monitor, Guardian, and Storytelling. Two major user use cases we identified are when the automobile drives a user to their destination and when the automobile parks itself.

Our analysis of the conceptual architecture will allow us to create a subsequent concrete architecture and furthermore an enhanced architecture.

Introduction

The purpose of this report is to model the key elements of the conceptual architecture that makes up v7 the Apollo autonomous driving open source platform. This report focuses on the Apollo Cyber RT which is a runtime framework for autonomous driving scenarios.

The report starts with a brief description of the derivation process of our conceptual architecture.

The report includes the architectural styles which make up Apollo which we found to be the layered style along with the publish-subscribe style and the process control style. The layers which up the layered style were found to be the RTOS at the lowest level followed by the Apollo Cyber Runtime Framework and then on the top are several components which include the map engine, localization, perception, prediction, planning, control and Human-Machine-Interface (HMI). The publish-subscribe style is in use through “Talkers” and “Listeners” which pass messages between components. The process control style is used to guide the vehicle as a result of environmental and movement parameters fed to the controller.

The next section of the report covers concurrency in Apollo and how the use of the publish-subscribe and process-control architectural styles illustrate the level of concurrency in use. Concurrency is in use with the publish-subscribe style because modules are not required to wait for each other and can communicate asynchronously, waiting for certain events to take place before taking action. The process control style also uses concurrency because the controller adjusts parameters in real time while the vehicle is already in motion to help guide the vehicle. The Storytelling and Monitor and Guardian modules also use concurrency to achieve autonomous driving functionality.

The next section of the report deals with detailing the different subsystems of Apollo which includes the following modules: Perception, Prediction, Routing, Planning, Control, CanBus, Localization, DreamView, Monitor, Guardian, and Storytelling.

The next section is the external interfaces description which details how components perform roles which require the transmission or reception of information. Modules such as the perception, routing, localization, and CanBus modules are detailed here.

The use case section follows and includes two main use cases, the first being an automobile driving a user to a destination and the second being an automobile parking itself.

Two sequence diagrams follow the use case sections which show how each case is modeled through the interaction of its modules.

In the subsequent limitations and lessons learned where we detail what we would do differently next time. Some of the limitations include limited experience with the domain of autonomous driving, only four group members, and missing v7 documentation. Some of the lessons learned were how to model a conceptual architecture and a first look at the autonomous driving domain.

The conclusion section summarizes our findings and leaves a proposal for the next report.

The final sections include a data dictionary in which key terms are explained along with a naming conventions section followed by references.

Derivation Process

The derivation process of the conceptual architecture was a multi-step approach. The first step was the collection of resources which we did in A0. Resources detailing the architecture of Apollo were aggregated on our group website and used as a foundation for understanding the software. The resources we found consisted of official Apollo documentation and source code. The resources provided by the professor on the assignment page were also used to get a background of the autonomous driving field and how to outline a project architecture. From reading the resources and applying the knowledge learned in class we were able to model a conceptual architecture of the library. We resolved dependencies about the way subsystems are connected through reading about the input and outputs of each module. Sequence diagrams were also created through Understand modeled after use cases.

Architecture

High Level Description

The conceptual architecture of a software project covers the abstract structure of the project itself, and as such we focus primarily on important structural details without regard to their actual implementation. We concern ourselves with the following main areas of the conceptual architecture of Apollo; the architectural styles in use, concurrency, subsystems, use cases, and finally sequence diagrams. This section focuses specifically on Apollo’s Open Software Platform.

Architectural Style

The architecture of Apollo’s Open Software Platform follows a combination of the layered, publish-subscribe and process-control architectural styles. The real-time operating system is the lowest layer of this architecture. The second layer of the architecture is the Apollo Cyber Runtime framework (RT). Above the Apollo Cyber RT at the third layer are the map engine, localization, perception, prediction, planning, control and Human-Machine-Interface (HMI) components. These three layers form the three distinct classes of the Open Software Platform architecture.

The first layer of the Open Software Platform architecture consists solely of a real-time operating system, whose purpose is to provide a platform upon which all other layers can operate. The second layer of the Open Software Platform architecture is where the publish-subscribe design patterns can be found. The Apollo Cyber RT framework provides a broadcasting system (bus) through which layer 3 components can pass messages by using “Talkers” and “Listeners” (these are equivalent to the traditional publisher and subscriber roles).

Certain layer three components, namely the planning and control components, utilize a process-control architectural style by using parameters such as speed and acceleration as process variables, and modifying set points for these variables in order to respond to different scenarios.

Concurrency

Concurrency is important for a system such as Apollo, because the execution of concurrent parts allows for the system to quickly perform multiple independent calculations, improving response time. A system designed to navigate an automated car requires a very fast response time from the software, as well as a large number of intensive calculations.

As discussed above, Apollo’s Platform uses a hybrid system, including elements of publish-subscribe and process-control styles. This allows each of the independent modules to run when needed, and they aren’t required to wait for one another. The Control module has almost exclusive authority over the car’s hardware and is constantly adjusting parameters to ensure the car follows its safe trajectory. However, many of the other modules run in concurrence with this as the environment changes, in order to recalculate the route whenever a vehicle’s location, or the status of its surroundings, changes.

When these changes occur, the modules associated with those changes can perform new calculations to determine the vehicle’s best course of action while the Controller module continues to operate the vehicle. Then, the Controller is sent the updated route and can adjust the vehicle accordingly, reflecting the Process-Control style incorporated with the system.

In addition to this, the Storytelling module is a special module that can prepare the vehicle for a complex scenario before it happens. It receives information about the vehicle and its location, then can create a series of Stories that aid the associated modules if they were to occur. The modules subscribe to this Storyteller module, and if the conditions for that Story are met, the published Story can be quickly downloaded by the modules enabling the vehicle to perform the required maneuvers. This is an example of the Publish-Subscribe style incorporated within the system.

Finally, the Monitor and Guardian modules are used to determine the status of the system in motion. As data is sent between the various modules, the Monitor module determines the connections and strength of the various components in order to report their status. In the event of a failure, the Monitor can call forward the Guardian module, which takes priority of the vehicle's hardware over the Control module and brings the car to a stop. This is an example of the Layered style present in the system, as the Monitor can access data at all layers of the system (hardware and software systems) concurrently with other modules.

Subsystems

The Apollo architecture is divided into a series of modules, in which key calculations and calls to the other modules are made to ensure the vehicle's accurate and safe navigation. The key modules are as follows:

Perception —

The Perception module makes use of the 8 required hardware components installed around the car to build a complete picture of the objects surrounding the vehicle. The 2 front cameras are each deployed to check for the presence and the status of any traffic lights in front of the vehicle, so it knows the current state of the intersection. In addition, these cameras act alongside the 2 radars and 4 LiDARs in order to scan the entire area surrounding the car. Anything received by any of these components is output as a set of 3D coordinates, telling the other subsystems where in the space obstacles may exist.

Prediction —

The Prediction module receives information from other modules about the vehicle's motion, localization data, any obstacles that exist in the current space and any perceived movement of the given obstacles. From there it estimates the future trajectories of the obstacles and of itself. It returns the obstacles, now annotated with their predicted movement, and a priority level for that object. Objects that are more likely to cross into the vehicle's path are placed on higher priorities than those moving parallel to the vehicle.

Routing —

The Routing module will plan the route that the vehicle will take to its destination using the map data that it’s been given, as well as its current location and the destination. It uses these to create a route, a series of roads and turns it needs to take in order to reach the destination.

Planning —

The Planning module is a key piece of Apollo’s architecture, as it builds the vehicle’s trajectory to follow. The module receives information about the vehicle, it’s localization, it’s desired route, it’s surroundings and any obstacles, as well as their trajectories and priorities. It uses all of this information to create a full picture of the vehicle’s surroundings, which it uses to decide which of the built-in scenarios it needs to follow (drive, stop sign, traffic light, park, etc). Once it’s decided the scenario that most appropriately fits its current surroundings, it uses that scenario to create a collision-free, comfortable trajectory for the vehicle to follow, and outputs that.

Control —

The Control module is in charge of translating the Planning module’s trajectory into a series of commands for the car to follow. It receives the desired trajectory, the status of the car and vehicle localization data to create a series of commands the vehicle must undergo to follow the given route. This control module has different modes, which can be changed upon request from the Dreamview module.

CanBus —

The CanBus module is an interface used to pass commands in order to control the vehicle hardware. In response to the command inputs, it returns information about the car chassis back into the vehicle’s software.

Localization —

The Localization module is used to find where the vehicle is located, in order to provide an accurate route. The localization module collects data from the GPS, the IMU, and the LiDAR sensors installed in the car to find its location, movement, and surroundings. This lets the system accurately track where the car is located at a given time and returns that estimated location to various other modules that require the car’s location.

Dreamview —

The Dreamview module is an HMI that allows the user to see the status of the vehicle and control its functionality in real-time. This module also allows the user to experiment with other modules as desired.

Monitor —

The Monitor module is used to keep track of all other modules being run in the system. The Monitor receives information about the status of all current modules, the frequency and integrity of transferred data, the status of the computer running Apollo and the hardware located in the vehicle. Each of these statuses is logged and reported to Dreamview system, as well as the Guardian module.

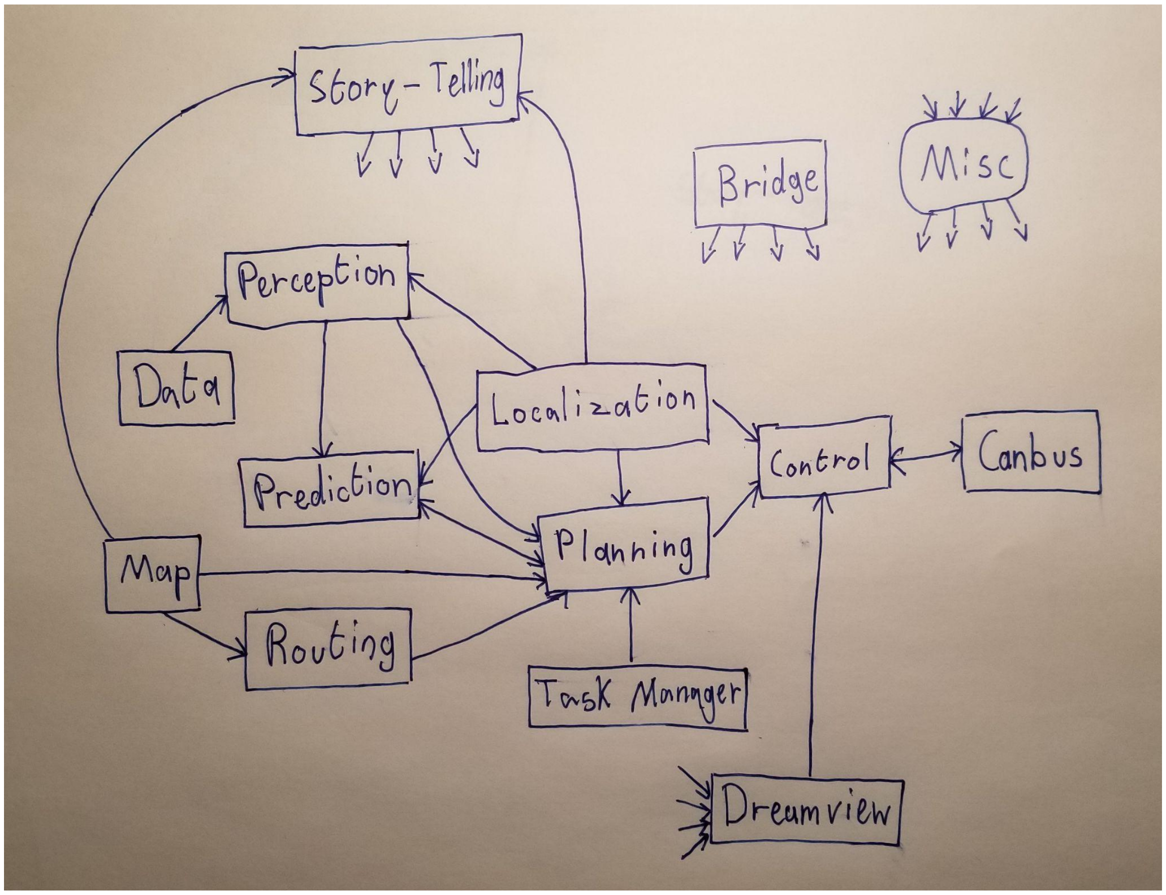
Guardian —

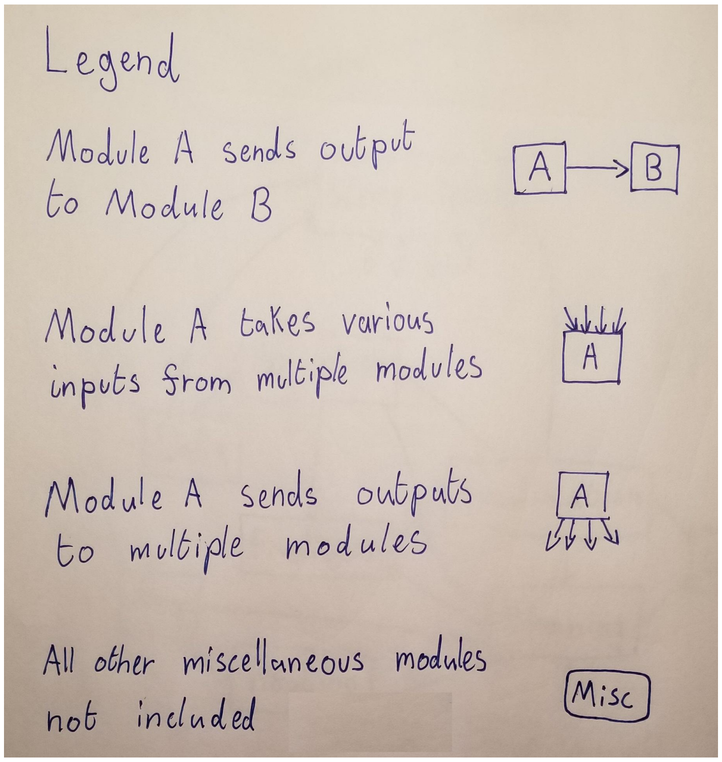
The Guardian module is a special case that only takes effect during a module's failure. If the Monitor detects a module failure in the system, then it will call the Guardian to stop the vehicle. The Guardian module checks if the Ultrasonic sensor is active and scanning for obstacles. If the sensor reports no obstacles, the vehicle is slowly brought to a stop by the guardian. Otherwise, if an obstacle is found or the sensors experienced a failure, the vehicle is immediately brought to a stop.

Storytelling —

The Storytelling module is a high-level scenario manager, which helps coordinate actions between the different modules. It receives data from the map and the vehicle's localization to decide where the vehicle is, which it then uses to create a series of complex scenarios. These scenarios are complicated instances that would require the rapid computation of multiple modules. These computations are published as a Story, which can be subscribed to by many modules at once. This helps fine-tune driving by pre-planning these scenarios and allowing them to be read if needed by many modules simultaneously.

Below is a basic box-and-lines diagram of the modules in the Apollo system:





Legend

Module A sends output to Module B

Module A takes various inputs from multiple modules

Module A sends outputs to multiple modules

All other miscellaneous modules not included

External Interfaces

There are various components within the Open Software Platform that perform roles that require the transmission or reception of information.

To achieve autonomous operation, information regarding the location, environment, and the status of the car itself must be ingested by Apollo. The perception module must receive information regarding obstacles and lane lines from a system of cameras, radar and LiDAR hardware. This data is processed in order to gain information about the surroundings of the vehicle. The routing module must receive route information indicating a start point and endpoint in order to compute a path to the desired destination. In practice, such information would most likely be inputted through the use of a user interface. Route calculation requires that information be received from the HD-Map component which resides at the Cloud Service Platform layer of the broader architecture. The localization module receives information from GPS, LiDAR and inertial measurement unit (IMU) hardware for the purposes of estimating the location of the vehicle. The CanBus module is the interface between Apollo and the vehicle itself, and as such will receive information from vehicle subsystems regarding their operation.

Information regarding the status of the vehicle and the Apollo system is transmitted by the Human Machine Interface (HMI) or DreamView. The module can be used to present the status of the vehicle but can also be used for testing modules. This module can also receive information, as it can be used to control the vehicle in real-time.

Use Cases

In order to illustrate the use of the modules in the system, here are 2 major, essential, use cases within the Apollo architecture

1. The automobile drives the user to a destination
2. The automobile parks itself

Use case 1:

A request is sent (from an external source) to the routing module with information about the automobile location and the automobile's destination. The routing module then takes the stored map from the map module and determines a route to the destination. The routing module passes this information along to the planning module. The planning module takes information from a multitude of modules in order to make a plan in real time that the automobile should follow in order to progress along the route. The important modules that it takes information from are the map, routing, perception, prediction, and localization modules. This, respectively, allows the planning module to factor the following items into its plan: the map of its location, its route, a visualization of its surroundings with identified objects (identified using Baidu's object data from the data module), a prediction of how those objects will act, and a localization of the automobile. The planning module will select from a list of possible scenarios and translate this into a plan. The planning module will then send its plan to the control module and the prediction module. The prediction module factors the planned trajectory into its next prediction cycle. The control module takes the plan and translates it into vehicle control commands. The vehicle controls are sent to the canbus (either directly, or through the miscellaneous guardian module). Finally, the canbus module acts as a medium between the software and the hardware which allows the automobile to move. The system will continue executing these modules until it reaches the destination. Please note that the localization module does in fact take inputs from the map module (and other sources external to the core software modules such as gps) although it may seem otherwise; Apollo has obfuscated this through the storytelling module, which does indeed pass the map over to the localization module.

Use case 2:

Whereas the previous use case was a very general scenario, this next use case is much more specific. For reference, this is the use case: The automobile needs to be parked.

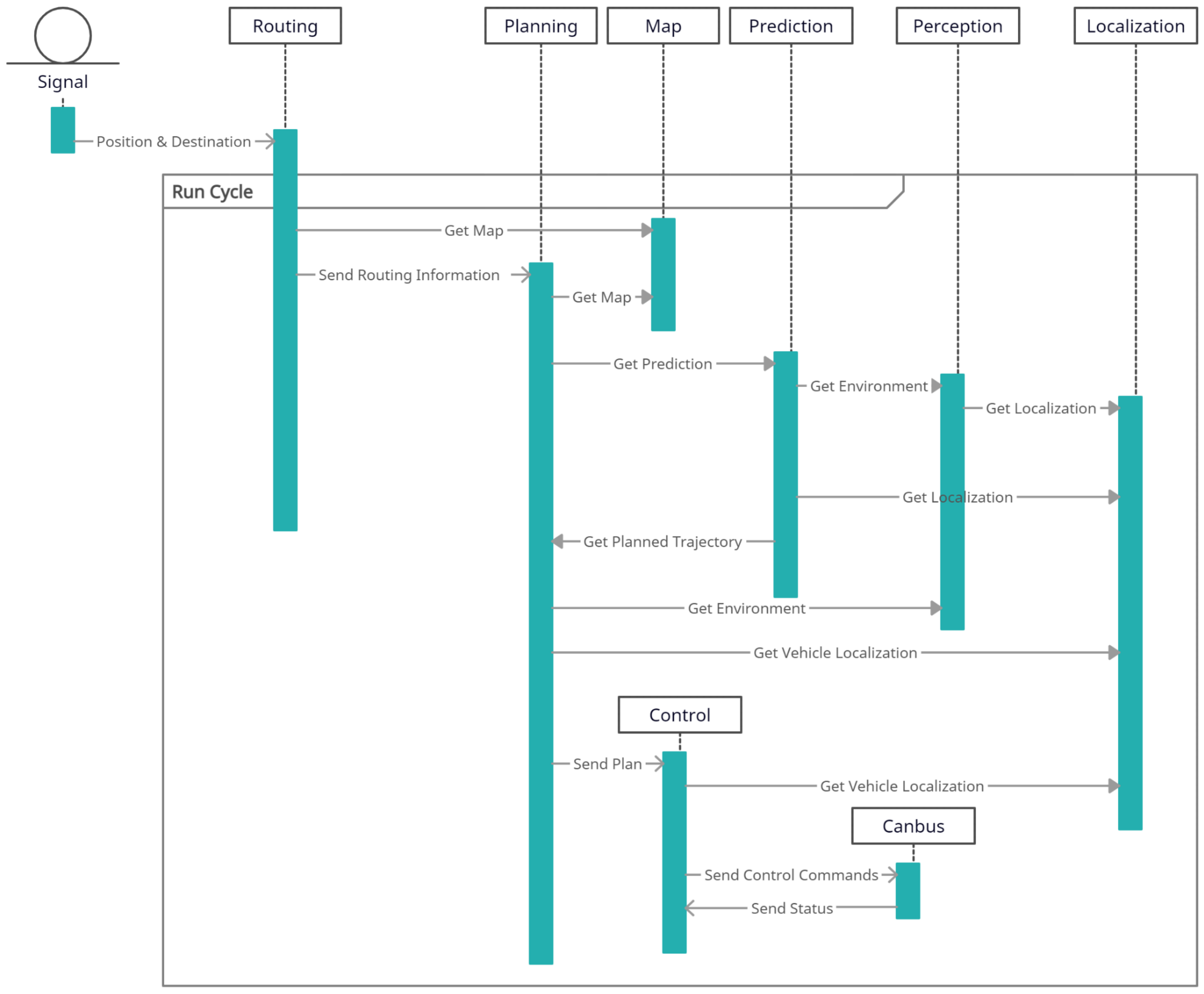
Since the vehicle would already be moving in this use case, we will not see a specific request being sent to the routing module (although the routing module will still be needed). Instead, during one of its cycles, the planning module learns that it is in a parking scenario. Again, it gains information about its current scenario by gathering information from the routing, map, prediction, perception, and localization modules. Once the planning module has a plan on how to park, it sends this plan to the prediction module so it can factor the fact that the vehicle is attempting to park into its prediction. The planning module also sends its plan to the control module, which will translate the plan into the control actions needed to park the vehicle. This is sent to the canbus, like before, which results in the vehicle parking in real time. After the system has completed enough cycles, the vehicle will be parked, ending this use case.

Modules not included in these use cases are: The task manager, dreamview, bridge, and (almost) all miscellaneous modules. Here's a quick overview of why they were either excluded, or not shown in the use cases.

The task manager holds the status of its given instance of the Apollo system; it is not relevant for most use cases as it is a background operation. The dreamview module is a human-machine web interface for developers; it takes input from various modules and gives a visual output, developers also have the option to control the vehicle using dreamview. Dreamview is therefore not present in our use cases, because it concerns the developers, not the users. The bridge module has a UDP socket, seemingly for internet access. In use case 1, the bridge module could be sending the initial signal to the routing module. Other miscellaneous modules are niche or contain basic items such as a library for math.

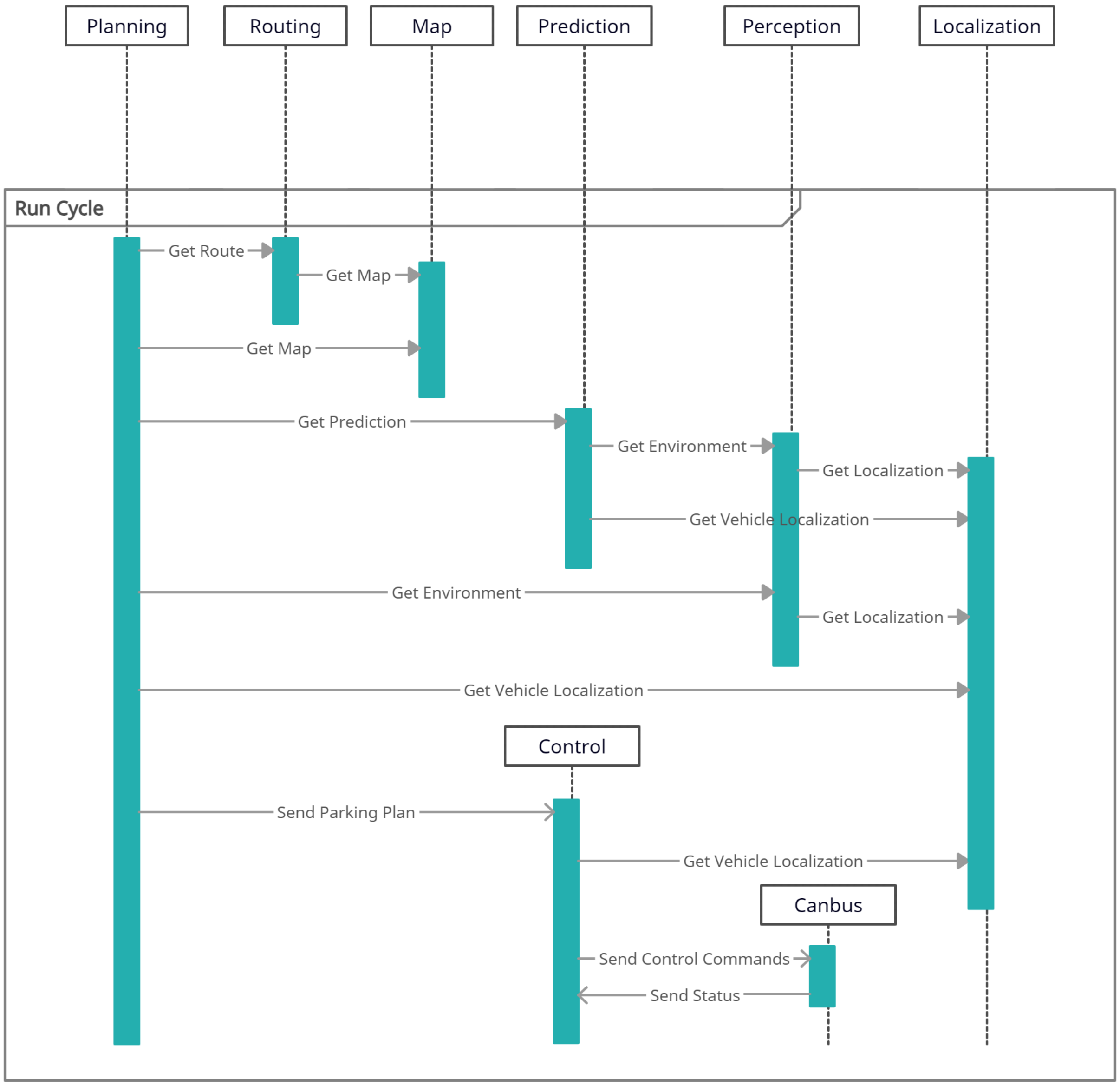
Sequence Diagrams

Here is a sequence diagram for use case 1, where the vehicle must travel to a destination:



The signal sends the position of the vehicle and its destination, which sends routing info to the planning module. The planning module gets information from multiple other modules, and comes up with a solution on how the vehicle will get to its destination, it sends this plan over to the control module, which translates this plan into control commands which is then sent to the canbus. This is all within a run cycle; it is being run constantly.

Here is a sequence diagram for use case 2, where the vehicle must park:



The vehicle is already running, and is thus in a run cycle already. The planning module gathers information from multiple other modules and learns that the automobile should be parked. It comes up with a solution to park the automobile and send this to the control module. The control module translates this plan into control commands which is then sent to the canbus. This is all within a run cycle; it is being run constantly until the vehicle is parked.

Limitations & Lessons Learned

There exist some limitations to this report. One of the primary limitations was the loss of one of our group members, which reduced membership to only 4 people. This meant that each member of the group had to take on a greater workload than initially expected. The result of this is that there may be some sections of our report which do not contain the desired breadth of information due to limitations in how much we were able to complete with less group members than necessary. Some information regarding the conceptual architecture of Apollo was certainly missed in this report due to these constraints. In addition, this also meant that there was less flexibility in how we were able to schedule time to work on the report. As a group, we had to carefully consider how we planned our time for this project since it was more limited than expected.

An additional limitation of the report concerns the Apollo documentation itself. Listed in the assignment deliverables is the instruction that this report should focus specifically on Apollo v7, however various elements of the Apollo documentation do not cover this version of the software. The software architecture documents on the Apollo GitHub for example are only available up to version 5.5. In addition, a group member noticed that sections from the version 5.5 documentation were simply copied from version 3. While some components and the roles of these components are certainly shared among these versions, there were components present in version 7 which were not covered in this older documentation. As well, there will certainly be new functionality in version 7 which would not be present in older documentation. The lack of completeness of the version 7 documentation meant that some sections had to be pieced together using pieces of documentation specific to older versions of the software. While this still likely provides an accurate picture, there is still the possibility that a section may contain incomplete information as a result of this challenge.

While autonomous driving is certainly an interesting topic to cover, it is also a fairly new field. Since none of our group members were familiar with autonomous driving software, there was an additional time investment required in order to gain an understanding of the autonomous driving problem domain and the general structure of it’s solutions. This unfamiliarity limited our understanding of the software since this is the first time any of us have seen such an architecture, as well as being our first foray into the world of autonomous driving.

The assignment was an excellent learning experience for every member of the group. Gaining familiarity with one another and our work schedules will certainly allow us to be more efficient and productive on future assignments. Our understanding of what conceptual architecture is was also certainly enhanced by our work on this assignment.

Understanding the definitions and examples given in class is certainly helpful, however analyzing a conceptual architecture in practice gives us the opportunity to put our knowledge to use. It also provided us an opportunity to see how the high level theoretical descriptions of software architecture compared with that of an actual . The assignment also provided us our first look at autonomous driving software which aside from being an interesting and novel challenge, may turn out to be of immense value if autonomous vehicles become mainstream.

Conclusion

In conclusion some of key findings were that Apollo’s conceptual architecture is structured using a combination of the layered, publish-subscribe and process control architectural styles. A high degree of concurrency is in use implemented through several modules such as the Storytelling and Monitor and Guardian modules. Several important modules are used to ensure the safe and accurate autonomous driving capabilities which include the map engine, localization, perception, prediction, planning, control and Human-Machine-Interface (HMI). Various external interfaces are at play in the exchange of information between modules. The perception module uses sensor data to pass to the routing module which computes a desired path which in turn communicates with the localization module to determine the position of the vehicle. The CanBus module is used to interface between the Apollo software and the vehicle. In the next assignment we will build on our conceptual architecture to attempt to model the concrete architecture, this is why it is important that our current conceptual architecture is well designed and accurate.

Glossary

Naming Conventions

GPS - Global Positioning System. A system that uses a receiver’s connection to a series of satellites in order to determine its location on the globe.
HMI - Human-Machine Interface. A dashboard that is used to connect a user to the machine, by displaying information and controls for the user.
IMU - Inertial Measurement Unit. A device that uses a series of gyroscopes and accelerometers to determine its orientation, angular rate and acceleration.
LiDAR - Light Detection And Ranging. A type of radar used to detect the distance of objects using a pulsed laser.
ROTS - Real Time Operating System. An operating system for real-time applications that process data and events that have critically defined time constraints.

Data Dictionary

Pub-Sub/Publish-Subscribe – Architecture style where modules subscribe to a publisher module
Process Control – Realtime architecture for controlling processes
Concurrency – Multiple items are run at the same time
Signal – A message sent to a module
Localization – Module that lets the automobile know where it is
Canbus – Medium between the automobile's hardware-software components
Control – Module that sends control commands
Dreamview – Developer HMI
Guardian – Safety module
Map – HD map module
Monitor – System monitor
Perception – Module that detects objects in images
Planning – Module that plans the auto's motion
Prediction – Module that predicts what is happening in its environment
Routing – Module that makes routes
Storytelling – Manages other modules
Sequence Diagram – Diagram that shows the sequence of events over its modules

References

1. https://cyber-rt.readthedocs.io/en/latest/CyberRT_API_for_Developers.html#talker-listener
2. https://github.com/ApolloAuto/apollo/blob/master/docs/specs/dynamic_model.md
3. https://github.com/ApolloAuto/apollo/blob/master/modules/perception/README.md
4. https://github.com/ApolloAuto/apollo/blob/master/docs/technical_tutorial/README.md
5. https://github.com/ApolloAuto/apollo/blob/master/docs/FAQs/README.md
6. https://apollo.auto/cyber.html#:~:text=The%20Apollo%20Cyber%20RT%20is%20a%20self%2Dcontained%2C%20open%2D,the%20Apollo%20Cyber%20RT%20framework
7. https://cyber-rt.readthedocs.io/en/latest/
8. https://www.sciencedirect.com/science/article/abs/pii/S0950584915002177?via%3Dihub