# CISC 322

# Assignment 1
# Conceptual Architecture of Apollo
# Feb 18, 2022



## Group 37

| | |
|---|---|
| Karl Dorogy | 18kwd1@queensu.ca |
| Ryan Saweczko | 18rjs5@queensu.ca |
| Tom Lin | 18phl@queensu.ca |
| Briggs Fisher III | 18wbfi@queensu.ca |
| Douglas Chafee | 18dsc1@queensu.ca |
| Itay Ben Shabat | 18ijbs@queensu.ca |

# Abstract

This report is an analysis of the Apollo autonomous driving open software platform's architecture. Through reading the documentation of Apollo, as well as some reference papers on autonomous driving software, we have come to the conclusion that high level architecture is implicit invocation. A significant factor for this system is reliability, ensuring information is sent, received, and used by the specific processes that require it, which can be achieved through the event system implicit invocation uses. We found there were several large components (called modules) within the system that handle different functionality for autonomous driving. These modules include perception to identify the world around the vehicle; prediction to anticipate how obstacles may move - such as pedestrians walking onto the road; routing to determine how to arrive at the destination; control to have the car perform specific desired actions, and many others.

Some of these modules follow other architectural styles depending on their use case. This includes modules such as the control module, which uses a process-control style architecture to moderate the cars actual movement - controlling the brakes, throttle, and steering. Due to the extreme importance of a car's physical movement, the self-correcting and real time updating, a process-control architecture is needed for this module. Another architecture style that can be found in the system is layered/tiered. This is used in areas such as audio, where the raw audio feed from microphones is handled by a driver for it, which then gives the feed to the audio module, where it is processed to find any audio cues required by the system, and the final output being available to other modules. There's clear separation in each step, with each one providing a service to the next level up.

We also investigated the use of concurrency within the system. The system is run in a heavily concurrent environment, with many modules handling different inputs simultaneously to encompass everything required to keep an autonomous car safe. The control and data flow is carefully controlled by the implicit invocation architecture to ensure all the modules function correctly despite the concurrent nature.

# Introduction & Overview

With the future of our transportation comes autonomous driving and Apollo is taking the wheel to deliver a flexible open source software architecture for the development, testing, and deployment of Autonomous Vehicles. Over its releases from version 1.0 to 7.0 Apollo has built up its own open software platform to rival competitors including the Google self-driving car with promises of high performance and reliability achieved through the event system implicit invocation uses. The high performance is achieved through efficient concurrency which is kept consistent by the implicit invocation architecture. This report will structure Apollo's conceptual architecture to provide insight on the organization of such an important large system.

The report is split into seven sections. The first section covers the derivation process which describes how we began our research on Apollo, discussed and agreed upon an implicit invocation architecture, split up our work, and began writing. The second section gives the overall structure of our studied system by visualizing Apollo's conceptual architecture in an implicit invocation style through an informative diagram alongside an explanation of each of the subsystems functions and interactions. The third section gives insight on the evolution of the Apollo system from its first version 1.0 to its current version 7.0 providing critical knowledge of past mistakes and accomplishments. The fourth section divides Apollo system's responsibilities into four major groups. In summary, the first group controls the autonomous A.I's perception and processing, the second group handles the distribution of commands given by the "planning" module, the third group provides tools and support for the other groups, and the final fourth group monitors all other modules and provides an interface for the vehicle's status. The fifth section of our report uncovers that the eleven modules use "cyber channels" in an eventing system that allows processes within modules to be run concurrently. In depth examples are provided and other instances of concurrency are reported. The sixth section provides two sequence diagrams created from two use cases alongside a description for each sequence diagram's modules and connections. The final seventh section includes first, the uncovered limitations of the Apollo architecture despite its strengths. Second, a data dictionary defining the technical terms found throughout our report. Third, a list connecting acronyms to their unabbreviated forms to provide context on our naming conventions. Fourth, a conclusion that summarized the justifications we made for the final architecture and a glimpse of our future work. Finally, the seventh section includes the lessons our group has learned from our studies and analysis.

After researching Apollo's Github repository and website, studying research articles, and reviewing course content, we determined that Apollo has an implicit invocation architecture. We determined there were eleven different subsystems including Perception, Prediction, Planning, Control, Canbus, Human Machine Interface, Localization, Monitor, Guardian, Audio, and Routing. We analyzed each of these subsystems and found other architectural styles used inside specific modules with the Control module using a process-control style architecture to moderate

the movement of the car and the Audio module using a layered/tiered style architecture to feed audio from the microphones to the audio module where audio cues are detected. We learned how to analyze large architectures and how to organize projects. Now that the conceptual architecture has been laid out it is clear what is expected from the concrete architecture and how Apollo can be improved.

# 1. Derivation Process

In documenting and formulating Apollo's conceptual architecture, the derivation process first began with every group member developing a solid foundation and understanding of the background information that is required for understanding autonomous vehicles/driving. Each group member was assigned to read through "A functional reference architecture for autonomous driving" (Behere et al.) and individually research through the documentation of Apollo, drafting a preliminary version of the conceptual architecture based on their understanding of the information they had read in addition to the provided lecture material within CISC 322/326. Then once each member had developed a possible conceptual architecture type a group meeting was held. Our entire group as a whole discussed the possible alternative architectures that each group member found and or created (layered, pipe and filter, implicit invocation, and repository), with each member presenting their reasoning for why they think Apollo uses that architecture. We then picked which modules/components should be included and excluded from the conceptual architecture as well as its overall type of being implicit invocation before eventually ending the meeting by dividing up all the major points we wanted our documentation to cover and each group members' respective designated tasks that individuals were responsible for. Later on, after individuals have done some work on the documentation of their tasks and upon new understandings/findings within the documentation for apollo's open-source platform for autonomous driving cars, our group held additional meetings to iteratively improve on our initial conceptual architecture and sequence diagram user cases, adding additional modules such as routing and audio in addition to discussing their involvement within the system and possible new dependencies that have arisen between components. Figure 1, ultimitally, is the diagram that we decided best represented the conceptual architecture of Apollo.

# 2. Conceptual Architecture

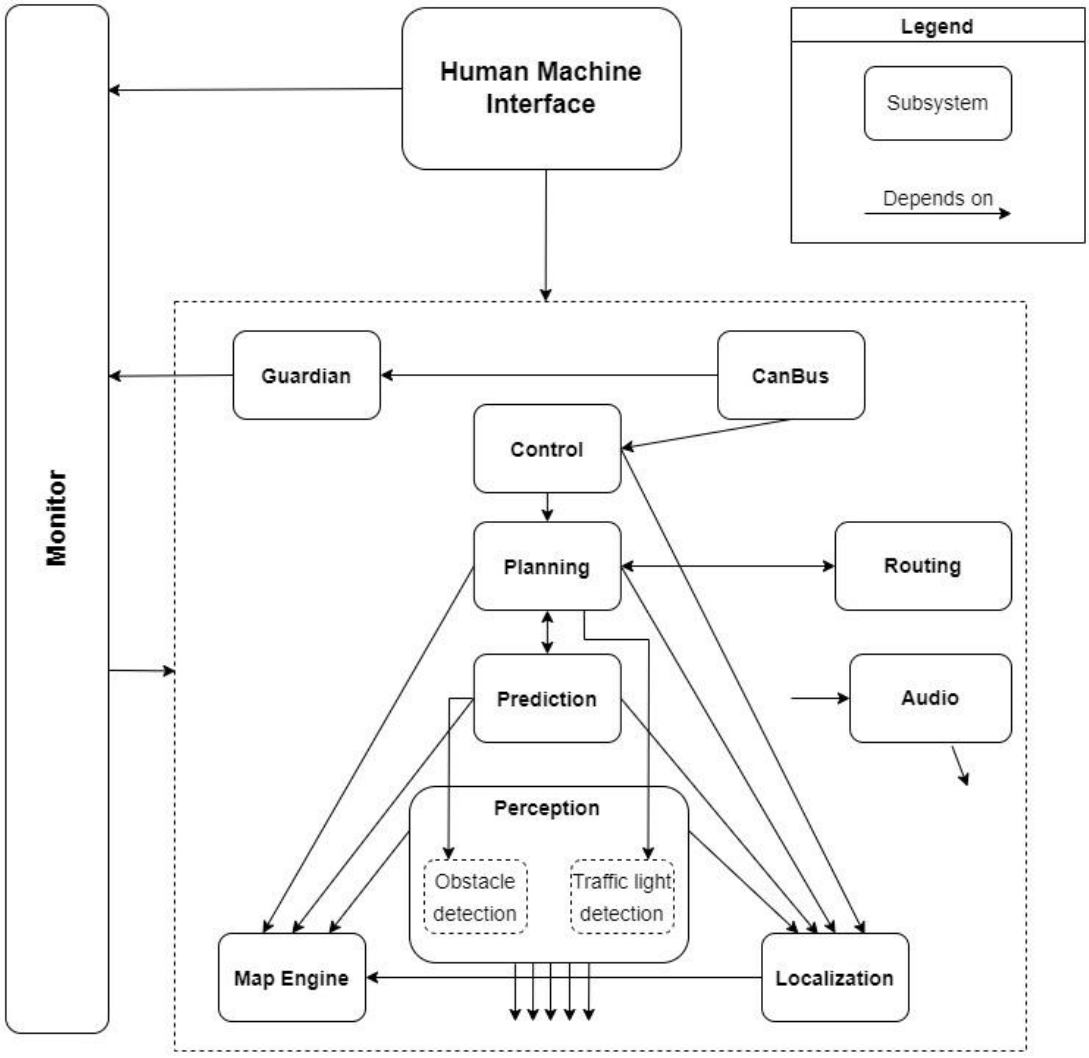


***Figure 1: Conceptual Architecture of Apollo***

The conceptual architecture of Apollo autonomous driving car software is implicit invocation. Using the real-time operating system CyberRT, with cyber channels as an eventing system to pass data between modules as the pipeline. Each of the modules are able to subscribe to specific events (or channels) of this system, such as the Perception module subscribing to /apollo/sensor/velodyne128 for information from a LiDAR camera. Some of the modules have their own architecture style for interactions within sub-modules, which will be explored in more detail. All of the dependency arrows in Figure 1 means the module is subscribed to an event the other module emits.

# Subsystems (Modules)

## *2.1 Perception*

The perception module incorporates the capability of detecting and recognizing obstacles and traffic lights using inputs from multiple hardware components such as cameras, radars, and LiDARs as well as the host vehicle's velocity and angular velocity from the localization module. The perception module outputs the 3D obstacle track with the heading, velocity, and classification information for any nearby obstacles or traffic lights recognised for other modules to use, specifically the prediction module. The perception module contains two submodules:

Obstacle Perception Submodule:
Sub-module that detects, classifies and tracks obstacles in the ROI that is defined by the high-resolution map while also predicting obstacle motion and position information (eg. heading and velocity). It includes LiDAR-based and RADAR-based obstacle perception, as well as the final fusion of both obstacle results. The LiDAR-based obstacle perception predicts obstacle properties such as the foreground probability, the offset displacement, object center and the object class probability using a Fully Convolutional Deep Neural Network. The RADAR-based obstacle perception is designed to process the initial RADAR data, extending the track id, removing noise, building obstacle results, and filtering the results by ROI. The final obstacle result is designed to be the fusion of the LiDAR and RADAR obstacle results.

Traffic Light Perception Submodule:
Sub-module that is able to work both day/night and obtains the coordinates of traffic lights in front of the car by querying HD-Map/MapEngine. The traffic lights are projected from world coordinates to image coordinates using intrinsic and extrinsic parameters of sensors. Then camera hardware selection is performed based on the projection results. A larger ROI in the image is depicted around the projection area of traffic lights where the traffic lights are then detected in the ROI with a surrounding bounding box, and recognized as different color states. The detection and recognition both have high recall and precision but after obtaining the single frame states, a sequential reviser is used to correct the states just in case.

## *2.2 Prediction*

The Prediction module studies and predicts the behavior of all the obstacles detected by the perception module. Prediction receives obstacle data along with basic perception information including positions, headings, velocities, accelerations, and then generates predicted future motion trajectories with probabilities for all perceived obstacles.The prediction message that is then output wraps the perception information.

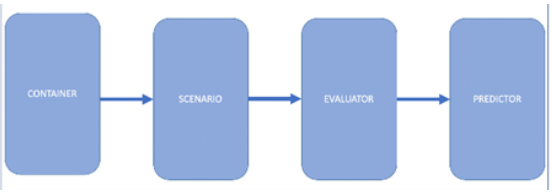


***Figure 2: Prediction Subsystem***

1. *Container*
   Container stores input data from subscribed channels.
2. *Scenario*
   The Scenario sub-module analyzes scenarios such as Cruise, the scenario that includes lane keeping and following, as well as Junction, the scenario that involves either traffic lights and/or STOP signs.
3. Evaluator
   The Evaluator predicts path and speed separately for any given obstacle.
4. *Predictor*
   Predictor generates predicted trajectories for obstacles.

## *2.3 Planning*

The Planning module plans a collision-free and safe trajectory for the control module to execute using several information sources and interacting with almost every other module . Initially, the planning module takes the prediction output and since the prediction output wraps the original perceived obstacle, the planning module subscribes to the traffic light detection output rather than the perception obstacles output. Then, the planning module takes the routing output. Under certain scenarios, the planning module might also trigger a new routing computation by sending a routing request procedure call if the current route cannot be faithfully followed. Finally, the planning module needs to know the location of where the vehicle is currently as well as the current autonomous vehicle information and or status.

## *2.4 Control*

The Control module manages vehicle control commands (steering, throttle, brake) to the chassis based on the planning trajectory and the car's current status. In addition the Control module uses different control algorithms to generate a comfortable driving experience while working in both normal and navigation modes. It has a process-control style architecture to moderate the cars actual movement.

## *2.5 Canbus*

The Canbus module is the interface that accepts and executes control module commands to the vehicle hardware as well as collects and passes the car's chassis detailed status as feedback to the control module of the software system. It has two data interfaces, the first one being the OnControlCommand which is an event-based publisher with a callback function that is triggered when the CanBus module receives control commands and the second one is OnGuardianCommand.

## *2.6 Human Machine Interface*

Apollo’s Human Machine Interface (HMI) module or DreamView is a web application that visualizes the current output of relevant autonomous driving modules such as planning trajectory, car localization, and chassis status in a web-based dynamic 3D rendering of the monitored messages in a simulated world. The HMI module monitors the following messages from Localization, Chassis, Planning, Monitor, Perception Obstacles, Prediction, and Routing. It also provides a general human-machine interface for users to view hardware status, turn on/off of

modules, and start the autonomous driving car as well as debugging tools to efficiently track module issues.

### *2.7 Localization*

The localization module aggregates various data to locate the autonomous vehicle and provide localization services. There are two types of localization modes, RTK (Real Time Kinematic) based method which incorporates GPS and IMU (Inertial Measurement Unit) information and the multi-sensor fusion method where a bunch of event-triggered callback functions are registered and incorporated with GPS, IMU, and LiDAR information. Eventually, the localization module outputs an object instance to be used by other dependent modules.

### *2.8 Monitor*

The monitor module is the surveillance system of all the modules in the vehicle including hardware and software. It receives data from multiple different modules ensuring that all modules are working in the system without any issues while also passing the data on to the HMI for the driver to view. In the event of a software module or hardware failure, the monitor module sends an alert to Guardian which then decides on which action needs to be taken to prevent a crash.

### *2.9 Guardian*

The Guardian module performs the function of an action center and intervenes should the Monitor module detect a failure, essentially making a decision based on the data that is sent by the Monitor. There are 2 main functions of the Guardian module. The first function is allowing the flow of control within the system to work normally (Control signals are sent to CANBus as if Guardian were not present) when all modules are working fine. The second function is when either a hardware or software module crash/failure is detected by Monitor then the Guardian will prevent Control signals from reaching the CANBus module and always bring the car to a stop.

### *2.10 Audio*

The Audio module is responsible for detecting the siren sound coming from active emergency vehicles. It detects an output siren on/off status, moving status and the relative position of the siren based on inputted audio signal data while also displaying an active alert on the Dream view web application (HMI) when an active emergency vehicle is detected. It interacts in a layered/tiered structure, where the raw audio feed from microphones are handled by their respective driver that is then given and or feed to the audio module, where the data is processed to find any specific audio cues that the system is required to detect.

### *2.11 Routing*

The routing module tells the autonomous vehicle how to reach its destination via a series of lanes or roads generating high level navigation information based on requests. The Routing module depends on a routing topology file from the map engine module as well as a routing request (start and end location) in order to output valid routing navigation information.

# 3. System Evolution

This section summarizes the important software changes and implementations of each release from 1.0 to 7.0 inclusive, as well as a summary of how these changes would have affected the system evolution. Important software changes meaning significant improvement to the system's ability to drive autonomously and/or change the module interactions.

## *3.1 Apollo 1.0*

Apollo 1.0 is the first release of the Apollo software and only capable of closely mimicking a human driver's input in controlled, flat environments. This version of the system has modules called localization, perception, prediction, planning, routing, control, and CAN Bus[2].

## *3.2 Apollo 1.5*

Apollo 1.5 introduces LiDAR calibration and a simple function of autonomous driving in the form of following traffic[3].

## *3.3 Apollo 2.0*

Apollo 2.0 is a refined release of Apollo 1.5 with more sensor calibrations and Dreamview, a software for visualizing the outputs of modules like planning trajectory, car localization, chassis status, perception, etc. Dreamview also allows for easier monitoring of modules and hardware as well as a report box so users can share issues as they arise. The planning module now communicates with every module aside from perception who's data is interpreted in prediction[4].

## *3.4 Apollo 2.5*

Apollo 2.5 release was made so Apollo would meet the Level-2 autonomous driving criteria. Now able to safely follow cars in high speed scenarios such as the highway while also using low-cost sensors[5].

## *3.5 Apollo 3.0*

Apollo 3.0 is primarily important hardware changes with no noteworthy software improvements/implementations[6].

## *3.6 Apollo 3.5*

Apollo 3.5's most important new feature is guardian, a safety module that behaves as an action center should the monitor detect a failure. By this iteration, planning has evolved to become more modular and treat every driving use case as different driving scenarios. Meaning an error discovery and correction in one use case will not affect how other scenarios work. The monitor module which constantly checks the hardware and software for errors/failures was also added. This is the only version where the architecture diagram changes to account for the addition of monitor and guardian. The new system has monitor receiving information from every module but HMI, and telling guardian when to intervene and send corrections to the CANBus[7].

### *3.7 Apollo 5.0*
Apollo 5.0 added an easy to use and scalable parallel distributed deep learning platform PaddlePaddle[8].

### *3.8 Apollo 5.5*
Apollo 5.5's perception module received Caution Obstacle. Caution Obstacle tracks all objects that have entered an upcoming junction (only traffic lights and STOP signs). It then evaluates every obstacle's trajectory as independent scenarios and determines the state ahead. Ignore, obstacle(s) have no effect on cars trajectory; caution, obstacle(s) will likely interact with the car; and normal, all obstacles that could not be categorized as ignore or caution. Similarly, control received Model Reference Adaptive Control (MRAC). A new algorithm allowing faster more precise steering control, enabling Apollo to support more driving scenarios like the successive side-pass. Monitor has new important safety checks; status of running modules, monitoring data integrity, monitoring data frequency, monitoring system health (cpu, memory, disk), and generating end-to-end latency stats report. Finally, the Control Profiling Service adds another safety check. Control Profiling Service: runs user input road-tests and or simulation data to systematically and quantitatively analyze the vehicle's control performance[9].

### *3.9 Apollo 6.0*
Apollo 6.0 added emergency vehicle audio detection[10].

### *3.10 Apollo 7.0*
Apollo 7.0 uses a new camera-based detection model similar to SMOKE (replacing CenterNet); a single-stage monocular 3D object detection model trained on waymo open dataset. A new more accurate LiDAR-based obstacle detection model similar to PointPillars, Mask-Pillars is introduced. Planning also added a dead end scenario, where the automated car will reroute itself and perform a three-point turn to turnaround[11].

# 4. Developer Responsibilities & Data Flow

Looking at the software overview of Apollo, we can determine there are four major groups in the division of responsibilities.

The first group handles perception, prediction, and planning. This group would have their work be mainly focused on making the "autonomous" part of "autonomous driving car" work. They will be making sure the A.I responsible for using the camera and sensor will "perceive" the proper obstacles (debris, traffic lights, pedestrians, road lines), "predict" the trajectory of those obstacles (pedestrian walking speed, debris falling speed), and then "plan" to avoid it. We can see from the diagram that in the planning module, it can send data back to the prediction module. This could mean that if a prediction is determined to be less than the ideal rate of accuracy, it is sent back to the prediction module to reevaluate.

The second group handles the control aspect of the autonomous driving car. This group will make sure the car correctly passes the command given in the “planning” module, such as braking, throttling, or steering.

The third group handles HD Map, Localization, CANBus, and Guardian. This group’s primary focus is to make sure the other groups function correctly by providing them with the proper tools and support. The individuals responsible for The HD Map and Localization modules make sure these two modules are feeding the correct information into group 1. They would make sure the HD Map is giving an accurate representation of the road and the Localization module is giving an accurate prediction of where the vehicle is on the map. The individuals responsible for the CANBus and the Guardian are responsible for making sure these two modules are utilizing the information correctly out of group 2 and group 4. They would make sure the Guardian module is operating correctly, as Guardian is a safety module and intervenes if the Monitor module (group 4) detects a failure. They would then make sure the CANBus module is executing the commands out of the Control module (group 2) as well as passing chassis information to the software system.

The last group, group 4, handles the Monitor and HMI modules. This group’s primary focus is to monitor all other modules in the system as well as provide an interface for viewing the status of the vehicle. The individuals responsible for the Monitor module would ensure it is monitoring all other modules correctly and providing adequate information to the Guardian module. The individuals responsible for the HMI module would make sure it is providing the necessary information as well as making the interface simple and easy to navigate.

# 5. Concurrency

The Apollo software is designed to run on the cyberRT framework, which was designed for autonomous driving cars, and is optimized for high concurrency. Most of the modules use an eventing system called “cyber channels”, which is designed to run processes within modules concurrently as messages are received. For example, there are several LiDAR, camera, microphone, and other sensors all have software drivers that work concurrently to continuously feed information into the perception system. Two other systems that are significantly concurrent are the monitor and the HMI.

The monitor works concurrently to all other modules being an introspection module that checks various hardware and software integrity, health, and speed to ensure there are no problems with the system. In order to properly perform this functionality, it needs to run concurrently to everything else in the system, to alert the system. The only interaction this module has with the rest of the system is when it detects a problem, sending an alert to the guardian module to handle the problem.

The HMI (human-machine interface), also known as DreamView is similar to the monitor module in that it introspects the other module, and needs to be concurrent for that reason as well. In displaying information about the other modules, it needs to run concurrently to them, allow

the other modules to process their data, with the HMI displaying it as it happens. For example, when the HMI shows a visual representation of the planning trajectory, the planning module needs to be running to generate the trajectory, as the HMI displays it in real time.

# 6. Sequence Diagrams

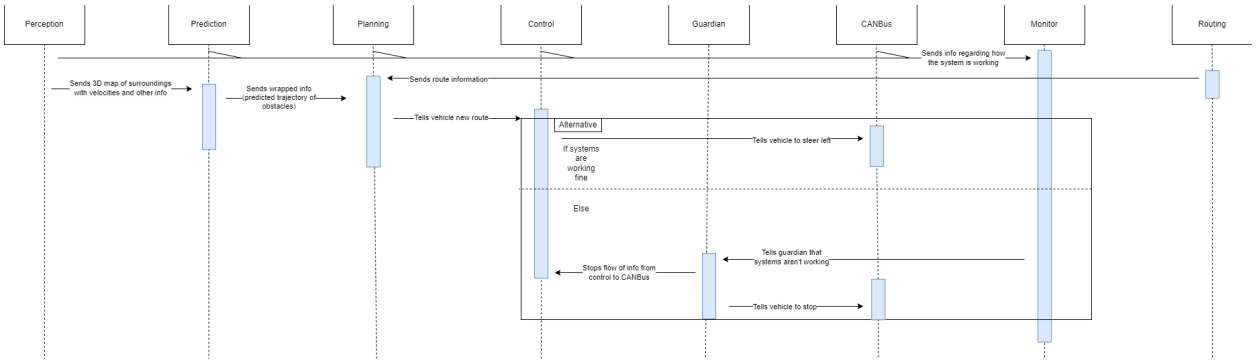


***Figure 3: Use case when the car is driving and the road twists left***

The perception module sends info regarding the car's surroundings to the prediction module which then wraps the information with predicted trajectories of objects in the surroundings and sends it to the planning module. The route for the car is also sent to the planning module where the new route for the vehicle is calculated and instructions to veer left is sent to the control module. The control module sends that information to the CANBus which then causes the car to move in the desired velocity.

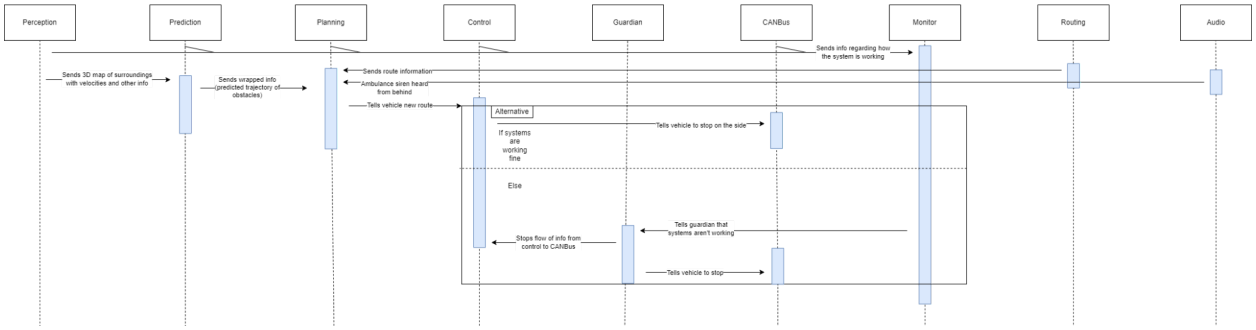


***Figure 4: Use case when an ambulance is heard***

The perception module sends info regarding the car's surroundings to the prediction module which then wraps the information with predicted trajectories of objects in the surroundings and sends it to the planning module. The audio module notices the ambulance and sends that info to the planning module which sends instructions to stop on the side to the control module. The control module sends that information to the CANBus which then causes the car to move in the desired velocity.

# 7.1 Limitations

Through the process of researching the architecture of Apollo we found some limitations of the architecture being used. One potential limitation is in how data is transferred between modules. All data transfer is done through cyber channels, which other systems can subscribe to (hence the implicit invocation architecture). Some of these ways data is transferred might be improved if a pipe-and-filter approach was taken instead. Since a lot of the input data to the system comes from cameras or microphones that provide streams of data, having specific processing steps in a pipe and filter would allow a very clear flow in the software, without reducing the strength of the interchangeability of the modules that currently exists. A specific example would be input from microphones through the microphone driver module could be piped directly to the audio module for detection of sirens without needing to use the cyber channel communication.

A limitation of trying to find the conceptual architecture of the system was some missing documentation about what modules emit or subscribe to events. There are several areas in the documentation where a module says it takes in some input/emits some output, yet nowhere else in the documentation is the other half that communicates with the module.

# 7.2 Data Dictionary

Implicit Invocation: The architectural style where procedures can subscribe to a broadcasting system to receive events from it, or emit events to the broadcast system.

# 7.3 Naming conventions

LiDAR: Light Detection and Ranging

RADAR: Radio Detection And Ranging

HMI: Human-Machine Interface

RTK: Real Time Kinematic

IMU: Inertial Measurement Unit

GPS: Global Positioning System

ROI: Region of Interest - area of an image that is selected and deemed as important; the foreground of the image

## 7.4 Conclusion

In summary, the Apollo autonomous driving open software uses an overarching specialized implicit invocation architecture, whose control component uses a process-control architecture. An implicit invocation architecture is essential for the functionality of an autonomous car as the system needs fast, event based decision making that fits the surrounding environment. Furthermore, the process-control system is instrumental for the deployment of control as its real-time micro adjustments to vehicle inputs must be instant and precise.

Going forward, we will analyze the source code to develop a concrete architecture and confirm the expected component inter-dependencies. We will also look at other open source autonomous driving systems to determine necessary and/or quality of life changes Apollo's system would benefit from.

## 7.5 Lessons Learned

Through the process of discovering and researching the conceptual architecture of Apollo, we learned a lot about searching through large projects to find documentation about how the project functions. One minor discovery that likely saved significant amounts of time was using linux commands like grep and find to quickly find the specific documents we need for different modules, removing the need to search through the hundreds of folders in the repository for all of the readme files.

We also learnt more about software architecture, and how to determine it from documentation throughout the project as well, and work on a team while researching the architecture of software. Each of us had to learn to read through the documentation of the software and create a mental image of what the architecture of the entire software would look like based on that, which took time for each of us to grasp.

# References


1. https://onq.queensu.ca/d2l/le/content/642417/viewContent/3814366/View
2. https://github.com/ApolloAuto/apollo/blob/master/docs/quickstart/apollo_1_0_quick_start.md
3. https://github.com/ApolloAuto/apollo/blob/master/docs/quickstart/apollo_1_5_quick_start.md
4. https://github.com/ApolloAuto/apollo/blob/r2.5.0/docs/specs/Apollo_2.0_Software_Architecture.md
5. https://github.com/ApolloAuto/apollo/blob/r2.5.0/docs/specs/perception_apollo_2.5.md
6. https://github.com/ApolloAuto/apollo/blob/r3.0.0/docs/specs/Apollo_3.0_Software_Architecture.md
7. https://github.com/ApolloAuto/apollo/blob/r5.0.0/docs/specs/Apollo_3.5_Software_Architecture.md
8. https://github.com/ApolloAuto/apollo/blob/r5.0.0/modules/perception/README.md
9. https://github.com/ApolloAuto/apollo/blob/r5.5.0/modules/perception/README.md
10. https://github.com/ApolloAuto/apollo/blob/master/docs/quickstart/apollo_6_0_quick_start.md
11. https://github.com/ApolloAuto/apollo/tree/master/modules/perception