CISC 322/326
Assignment 2: Concrete Architecture of Apollo

March 21, 2022

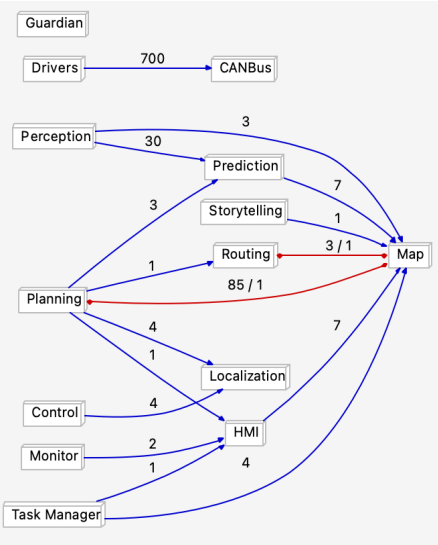
Sebastian Wakefield	19sslw@queensu.ca
Dominic Guzzo	18dpg3@queensu.ca
Oscar Wojtal	18ow2@queensu.ca
Alice Cehic	17ac121@queensu.ca
Cooper Lloyd	17cel4@queensu.ca
Daniil Pavlov	17dp15@queensu.ca

Abstract

In the previous report, it was concluded that ApolloAuto's conceptual architecture uses a publish-subscribe architectural style. The conceptual architecture resulted from the documentation mainly found on the ApolloAuto Github repository, website, and Behere's paper; *a functional reference architecture for autonomous driving*. In this report, the concrete architecture of Apollo will be derived using a combination of Apollo's source code, documentation, our previous conceptual architecture, and our created architecture using the Understand tool.

This report investigates the concrete architecture within the source code for Apollo Auto which is conducted by analyzing the unexpected dependencies and modules present in the concrete architecture but not the conceptual architecture. The process of how the concrete architecture is derived is detailed in the derivation process - where we touch on numerous research papers, Github documentation, and source code that we analyzed to map our concrete architecture with Understand.. The report also focuses on descriptions of subsystems within the architecture - specifically the Perception, Prediction, Localization, Map, Storytelling, CANBus, Control, Monitor, Routing, Drivers, and Human Machine Interface modules. The localization subsystem will also be investigated in its use of the Real-Time Kinematic and multi-sensor fusion method for vehicle localization in our discovered Localization module. Our in-depth analysis of the Localization module helps give a better understanding of which components the module interacts with and why, as well as the many advantages and disadvantages to the system. We move then dive into various unexpected dependencies of the concrete architecture that we gathered using Understand as well as a source code analysis. For our 2nd level subsystem reflexion analysis, although our chosen module differs from the Localization module in which we did in our in-depth subsystem analysis, I have spoken with Professor Adams about doing this reflexion analysis for a differing subsystem (Planning) and have been told that this is fine. Finally, we show 3 examples of possible use cases with this newly improved architecture and then discuss the limitations of the derivation process and lessons learned throughout this project.

Derivation Process



In analyzing Apollo 7.0 source code, we gained valuable knowledge on the concrete architecture of Apollo software, in which we compared our initial conceptual architecture analysis. Reviewing the Apollo Github allowed us to better understand the subsystems and dependencies present in the software's structure. Using this new knowledge, we could add and expand on our previous understandings of Apollo's subsystems and dependencies.

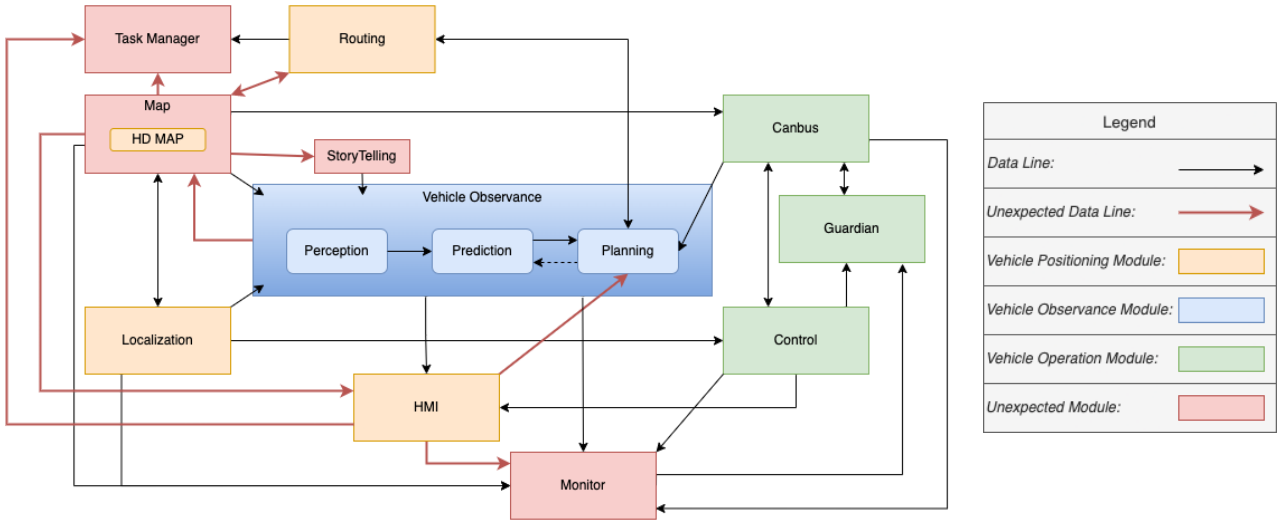
Using the Understand tool, we mapped the subsystem's source code to the conceptual architecture to visualize the dependencies in the software. In addition, the README files for each submodule provided an overview and explanation of the interaction between each module. Utilizing the README files and the Understand module, we discovered the elements of

Apollo's concrete architecture and made connections between the source code and concurrent modules. Furthermore, we found research papers that offered us a greater understanding of the interactive parts of autonomous vehicle hardware, localization aspects, and how the software and users control it.

We used the provided Apollo Pub-Sub communication model to graph the concrete architecture and then found all the unexpected dependencies from our SciTools Understand graph. This allowed us to create a detailed architectural graph representing the Apollo software architecture. In addition, we were able to derive updated sequence diagrams from the information gathered from the README files and provided graphs to illustrate the software dependencies better.

Concrete Architecture

The reflexion analysis is generated by investigating and identifying the difference between the concrete architecture developed by the Understand tool and the conceptual module previously



constructed. From our observations, there are 4 additional modules included in the concrete architecture; Task Manager, Map, StoryTelling and Monitor.

The discrepancies between the conceptual and concrete architecture is pronounced in the Localization and Planning subsystems, each having their own unexpected dependencies that have emerged in the concrete architecture. The localization subsystem's main difference from the conceptual architecture is the inclusion of Real Time Kinesmatic and multi-sensor fusion methods of localizing accuracy. The methods provide all-weather availability and flexibility that enhance the system's ability to localize. The Planning concrete subsystem differs from the conceptual architecture by obtaining information from the Preception, HMI and Map modules to optimize the planning process. The conceptual architecture did not foresee the practicality the unexpected dependencies have on each of the subsystems.

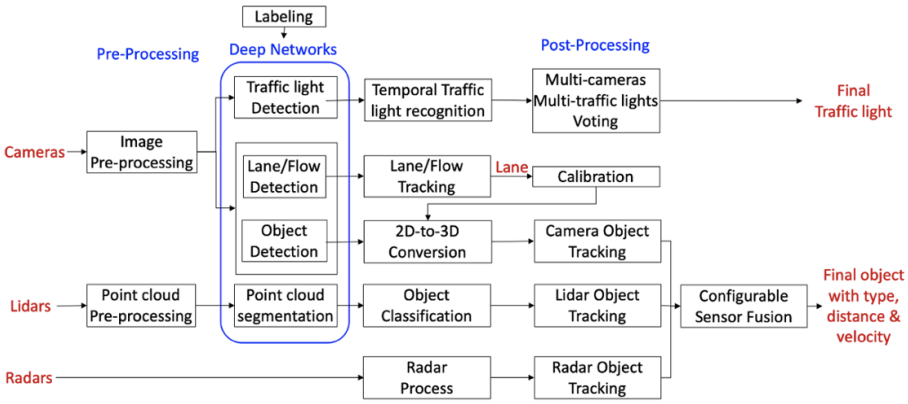
Subsystems

Perception Module

The perception module incorporates the capability of using 5 cameras (2 front, 2 on either side and 1 rear) and 2 radars (front and rear) along with 3 16-line LiDARs (2 rear and 1 front) and 1

128-line LiDAR to recognize obstacles and fuse their tracks to obtain a final tracklist. The obstacle module detects, classifies, and tracks obstacles and predicts the obstacles' motion and position information (e.g., heading and velocity).

This module is of the layered architectural style since it implements data transformation within several layers of abstractions. At the same time, each layer interacts only with the neighboring one, sequentially receiving, transforming, and transmitting data to the next layer of abstractions. The following layers are present: Data Pre-Processing, Labeling, and Data Analysis (Post-Processing). Data Pre-Processing receives data from sensors (such as cameras, lidar, radars) in digital form. Then, preliminarily processes them (with procedures such as position alignment, color calibration, contrast enhancement) for their standardization (calibration) - bringing them to a form suitable for further processing. The next layer is Labeling, which is implemented using Deep Neural Networks to segment, classify, and label the data received from the previous layer. Thus, we get the positions and types of objects on the image or the lidar distance map. The Deep Neural Network technology (created by Baidu) provides segmentation, classification, and labeling of data in a fixed time, unlike algorithmic methods. However, it requires significant computing resources and large amounts of data. Computing resources are provided by using hardware accelerators such as NVIDIA CUDA. The third layer, Data Pre-Processing, performs further analysis of segmented and classified data by algorithmic methods to obtain information about traffic lights and moving and stationary objects in the field of view of vehicle sensors.



The perception module subscribes to various data outputs from other modules, such as the perception information from the Drivers module and vehicles positioning from the Localization module. It uses resources from the Drivers module to attain perception information from the cameras and radars, specifically the PointCloud and ContiRadar classes within the Drivers module. The camera components in the Perception module directly subscribe to the Drivers module to get this raw camera and radar information and parses the data using neural networks and computer vision classes. Furthermore, this module subscribes to the Localization module (precisely the output of the LocalizationEstimate class) to check the vehicle's velocity and angular velocity – allowing the Perception module to better understand the context of what is being perceived.

Prediction

The Prediction module calculates the possible trajectories of objects. It subscribes to the objects perceived by the Perception module and publishes them back through a shared pointer in the *Proc* function. Prediction also subscribes to Localization's messages through the *onLocalization* function to understand and estimate the ego cars position in the environment. It similarly subscribes to the planning module to determine each obstacle's collision risk priority (Ignore, Caution and Normal) for a given plan. The output is then wrapped and published to the planning module which

will from there determine if the route is safe enough and send back updated plans for processing if needed.

Planning

The Planning module creates multiple possible plans that the car can execute. In order formulate these scenarios planning gathers information from: (1) Localization, the ego cars position; (2) Perception, to receive locations of objects; (3) Prediction, to understand where each object could be going; (4) HD-Map in the Map module; (5) Routing, to know where along the map they are in respect to the destination; (6) Task Manager, which helps the car route in specific scenarios ie a dead end. The chosen plan is then dispatched to control for the car to execute.

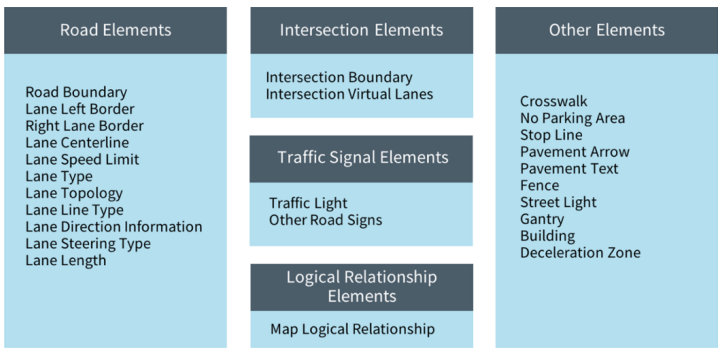
Localization

The localization module aggregates various data to locate the autonomous vehicle using callback functions and timers to attain the car's location in real-time. The benefit of creating this module is that the car's position relative to external objects can be computed so that the vehicle is always at a safe distance from these obstacles. The localization of the car must have single-digit centimeter accuracy and is vital for many of the car's principal utilities, such as object avoidance and parking. This module leverages various information sources such as GPS, IMU, and LiDAR to estimate where the autonomous vehicle is located. Subsystems in the Localization module subscribe to classes such as GnssBestPose, Pointcloud, CorrectedImu, and InsStat in the Drivers module to attain this information and parse it for its two main localization components: Real-time Kinematic and Multi-sensor fusion. The GNSS, a kind of GPS, refers to a constellation of satellites providing signals from space that transmit positioning and timing data to GNSS receivers. The receivers then use this data to determine location. An inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the body's orientation, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. We can then combine GPS and IMU data to localize the car. With LiDAR, we can localize a car utilizing point cloud matching. This method continuously matches the detected data from LiDAR sensors with the pre-existing HD map. This requires constantly updating the HD map (generated in the multi-sensor fusion localization method or *apollo/localization/msf* folder), which can be extremely difficult.

Map

This module is similar to a library. Instead of publishing and subscribing to messages, it frequently functions as query engine support to provide ad-hoc structured information regarding the roads. Apollo HD maps employ the OpenDRIVE format, an industry-wide mapping standard. It serves as an API that makes it easy for everyone to read the same map data (WEI 1).

OpenDRIVE API readable data:



While the overall structure is similar, Apollo's modified OpenDRIVE is very different from the usual OpenDRIVE standard, as they have changed several fundamental details, for instance, how roads are represented. Apollo uses this classification API and GPS data and their perception module to configure the HD map to navigate the road. This Map module contains a submodule called Relative map

that acts as a middle layer connecting the HDMap/Perception module and Planning module. This submodule generates a real-time relative map in the body coordinate system (FLU) and a reference line for planning (WEI 1). The inputs for the relative map module have two parts: offline and online. The offline part is the navigation line (human driving path) and the HDMap information on and near the navigation line. And the online part is the traffic sign related information from the Perception module, e.g., lane marker, crosswalk, traffic light etc. This Relative map module also uses inputs from the Dreamview module, specifically the NavigationInfo class and the Localization module.

Storytelling

Storytelling is a global and high-level Scenario Manager to help coordinate cross-module actions (*apollo/modules/storytelling/README.md*). To safely operate the autonomous vehicle on urban roads, complex planning scenarios are needed to ensure safe driving. These complex scenarios may involve different modules to ensure proper maneuvering. A new isolated scenario manager, the "Storytelling" module, was created to avoid a sequential-based approach to such scenarios. This module creates stories which are high-level scenarios that would trigger multiple modules' actions. The main advantage of this module is to fine-tune the driving experience and also isolate complex scenarios packaging them into stories that can be subscribed to by other modules like Planning, Control etc. Furthermore, the Storytelling module uses the Map module to gain junction, yield sign, stop sign, and crosswalk data to form stories and create the context for other modules such as the Planning and Prediction modules.

CANBus

The CANBus accepts and executes control module commands and collects the car's chassis status as feedback to control (*apollo/modules/canbus/README.md*). This module has two major components; *Vehicle,* which is the vehicle itself and its controller and message manager, and the *CAN Client*, which has been moved to /modules/drivers/canbus since it is shared by different sensors utilizing the CANBus protocol. The CANBus publishes and subscribes to the Control module (specifically the ControlCommand class) to manage control functionality (steering, reversing, braking, etc.) for specific vehicles like Lincoln and Lexus.

Control

The Control module takes the results of the planning trajectory analysis and outputs the control commands to the CANBus where it can be used to control the action of the autonomous vehicle (*modules/control/README.md*). This module uses five interfaces within its concrete functionality: OnPad, OnMonitior, OnChassis, OnPlanning, and OnLocalization (README.md). The OnChassis, OnPlanning, and OnLocalization functions work to control and use the vehicle data using mutex lock guards while keeping logs of the feedback given by corresponding functions. The OnPad and OnMonitor interact with the HMI to give the user the ability to control the vehicle. These control commands are outputted to the CANBus and Planning module, and commanded by PadMessage, Monitor, and Guardian.

Monitor

The Monitor module is used to supervise all modules and hardware features of the vehicle (*apollo/modules/monitor/README.md*). It accepts data from software modules and sends it to the Human Machine Interface, allowing the driver to monitor the status of each component. The Monitor also has a connection with the Guardian in the event of an issue. If a hardware failure is detected, it sends an alert to the Guardian to prevent a crash. It primarily listens to the Control module to allow

the driver to instruct the autonomous vehicle, after which the Monitor can contact the appropriate module to carry out the command.

Routing

The Routing module needs information about the start and end point, to be able to create a usable route (*apollo/modules/routing/README.md*). The Routing module forms a triangular connection with the Map and Routing modules. It uses the task manager and map modules to create a route and sends it to the planning module, where it is processed and planned how the route will be executed. If the route is unable to be executed the planning module sends a request back to Routing for another route. Furthermore, the Task Manager can request the Routing module using the RoutingResponse executable, just like the Planning module, if it detects an issue or needs to call a new route.

Human Machine Interface

In Apollo, DreamView is a web application that allows the user to control and visualize the status of each hardware and software component of the vehicle (*apollo/modules/hmi/README.md*). It receives data from the Task Manager, Monitor, and Planning modules and outputs to the Map module. To allow the user to operate all aspects of the vehicle, the HMI is dependent on and has some type of link to all modules. This ensures that the autonomous vehicle is safe and under control.

Guardian

The Guardian module has dependencies to the Cyber subsystem for interacting with other Apollo components and receiving messages from them. The interaction of the Guardian module with the Perception module allows the Guardian module to receive information/performance data from the Perception module, which is important for ensuring the safety of the autonomous vehicle. The Guardian module fully complies with the Controller architecture by monitoring the status of external sensors and the Perception module can bring the vehicle to a stop by sending the appropriate commands to CANBus.

Drivers

The Drivers module is a set of control program code for the interaction of Apollo with hardware components. The hardware components, interaction with which is provided by the program code of the Drivers module are: Camera, CANBus, GNSS, LiDAR, Microphone, Radar, Smarter Eye, and Video Input/Output. There is also the Tools library that provides common code for image manipulation for elements of the Drivers module. Using the Understand software, we can see which Apollo modules directly interact with the hardware through the Drivers module. In the Apollo Conceptual Architecture stage, Perception was believed to be the main module interacting with the hardware. Research within Concrete Architecture will help us understand if that is the case. Driver.Camera interacts with other components through the Cyber subsystem. The relationship that was not revealed at the Conceptual Architecture stage of the project is the direct interaction of the Driver.Camera with the Perception module to receive processed LiDAR data through this module. Driver.CANBus supports interaction with the CANBus hardware. It can directly be called by the Cyber subsystem and the Perception module. Driver.GNSS provides the operation of satellite positioning devices. It is used to determine the position of the vehicle by the Localization module. Driver.LiDAR ensures the operation of the LiDAR device of the autonomous vehicle and is used by the Perception and Prediction modules to determine the position of the vehicle relative to the environment. It builds a map of distances to obstacles relative to the autonomous vehicle for the Apollo system to have a correct perception of the environment and the ability to predict its changes.

Driver.Radar provides the operation of the Ultrasonic Radar – a device that determines the distance from the vehicle to obstacles in certain directions. Is used by the Perception module.

Task Manager

Task Manager subscribes to the Routing and Map modules and helps the car recognize and execute paths in specific scenarios such as while parking and when in a dead end. This module has 3 main components to it: cycle routing manager, dead end routing manager, and parking routing manager. These 3 components are organized through the use of a task manager component which makes sure the proper routing functionality is done at the right time in order with the help of imported files and functions from the Planning and Localization modules. The cycle routing manager mainly helps reroute the vehicle using the GetNewRouting function and updates the HMI as well as checks if the vehicle can be routed to that destination. The dead end routing manager helps to judge if a car is in a dead end using the function JudgeCarInDeadEndJunction and the help of the Map module as well as reroutes the vehicle. The parking routing manager has a function for judging the size of the parking spot as well as the width of the road, SizeVerification and RoadWidthVerification functions respectively. These functions help inform the system and the user which approach to parking is best and whether the car can be autonomously parked at all based on the surroundings.

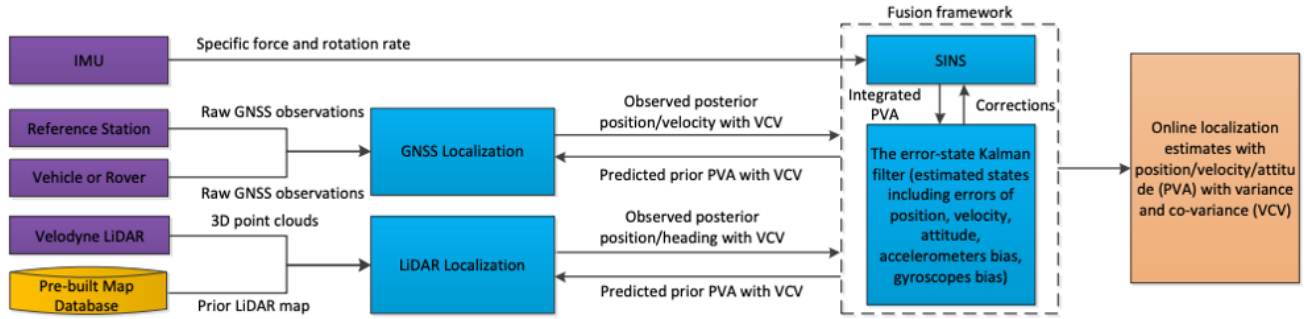
Analysis of Localization Subsystem

Apollo's Localization subsystem provides centimeter-level localization accuracy in disparate city scenes using two main techniques: The **RTK** (Real-Time Kinematic) based method and the **multi-sensor fusion** method. "Multi-sensor function localization works well when the environment is full of 3D or texture features, while RTK performs excellently in open space" (Wan 1). The Localization subsystem uses a publish-subscribe architectural style since it subscribes to the output of numerous perception devices such as the GPS, IMU, and LiDAR. This module has various loosely-coupled components, like the RTK and multi-sensor fusion, that must be reconfigured "on the fly" due to the constant influx of sensor information from the Drivers module and map information from the Map module - which is why publish-subscribe architectural style works well for this system. The architectural style also allows multiple modules to subscribe to its LocalizationEstimate output, such as the Prediction, Perception, and Map modules, and provides strong support for reuse since any component can be introduced into a system simply by registering it for the events of that system.

The RTK (located in the *apollo/modules/localization/rtk* subfolder) uses GPS and IMU information to localize the car's current position. The files in the rtk folder also use GNSS data taken from the Drivers subsystem to update the localization system in the RTKLocalization class, which happens to be the main class that provides RTK localization functionality. Apollo's GNSS RTK module utilizes the help of the multi-sensor fusion framework and achieves a better ambiguity resolution success rate. An error-state Kalman filter is applied to fuse the localization measurements from different sources with novel uncertainty estimation. The most significant advantage of RTK is that it provides almost all-weather availability. "However, its disadvantage is equally apparent that it's highly vulnerable to signal blockages because it relies on the precision carrier-phase position techniques and does not rely on LiDAR like the multi-sensor fusion method" (Wan 2).

The multi-sensor fusion method (located in *apollo/modules/localization/msf*) incorporates GPS, IMU, and LiDAR information to help localize the car's position. This method provides four modes for MSF localization that include; 3-Systems, which uses GNSS localization results at all times and 2-Systems, which only applies GNSS localization results to initialize SINS alignment.

Furthermore, this method also provides three modes of Lidar localization: intensity, altitude, and fusion. You may set the parameter lidar_localization_mode in localization.conf to choose which mode you would like. LiDAR-based localization in the msf folder is done in *apollo/modules/localization/msf/local_map*, making a grid-cell representation of the environment. Each cell stores the statistics of laser reflection intensity and altitude, and the map is organized as a group of map nodes. Moreover, this method also provides NDT-based localization in the modules/localization/ndt/ndt_localization.cc file, a compressed representation of the point cloud map that stores the voxels and their 3-D normal distributions. This provides a straightforward, flexible method that compiles LiDAR pointcloud and other information to achieve high localization accuracy.



Unexpected Dependencies

Below is a collection of unexpected dependencies we identified and analyzed from our concrete architecture that we created using the Understand tool:

Map → Planning
apollo/modules/map/pnc_map/pnc_map..cc depends on
apollo/modules/planning/common/planning_gflags.h
Description: The "planning_gflags.h" file declares numerous variables or flags pertaining to the planning of scenarios, trajectory, navigation, and many others. These variables are indicators of certain events, such as emergency pullover and yield sign scenarios. The "pnc_map.cc" file updates vehicle routing and navigation with functions like ValidateRouting and NextWaypointIndex. This file uses the flags declared in "planning_gflags.h" to perform decisions on routing and mapping.

Perception → Map
apollo/modules/perception/map/hdmap/hdmap_input.cc depends on
apollo/modules/map/hdmap/hdmap_util.h
Description: The "hdmap_util.h" file in the Map module includes a file called "hdmap.h" that has an HDMap class which contains useful "get" functions such as GetYieldSigns and GetNearestLane. The "hdmap_util.h" file also provides a map class with utility functions. The "hdmap_input.cc" file in the Perception module uses input from the HD map to help with perception functionality. This file uses functions provided by "hdmap_util.h" as well as its imported file "hdmap.h" to provide further context to the perception module and initialize a map to attain road boundaries, crosswalks, and other map information.

Storytelling → Map
apollo/modules/storytelling/story_tellers/close_to_junction_tellers.cc depends on
apollo/modules/map/hdmap/hdmap_util.h

Description: The file “close_to_junction_tellers.cc” indicates whether the vehicle is close to a junction or intersection based on information from the Map module. The imported file “hdmap_util.h” provides functions to get road-related information that can be queried and used to notify the storyteller whether the vehicle is close to the junction based on the appearance of a clear area, crosswalk, junction, stop sign, yield sign, or signal.

Routing → Map
apollo/modules/routing/topo_creator/topo_creator.cc depends on *apollo/modules/map/hdmap/hdmap_util.h*
Description: The Routing module file “topo_creator.cc” creates a routing topology from a base map to a routing map. This file includes a map util file called “hdmap_util.h” that includes useful functions for the creation of a base map as well as a routing map. The Routing module uses these functions to make its routing topology by giving these maps as parameters to a graph creation class, called GraphCreator, in the “graph_creator.h” file.

Perception → Prediction
apollo/modules/perception/lidar/lib/tracker/sematic_map/evaluator_manager.cc depends on *apollo/modules/prediction/evaluator/vehicle/semantic_lstm_evaluator.h*
Description: The “evaluator_manager.cc” file in the Perception module evaluates obstacles and creates an obstacle history map. The “semantic_lstm_evaluator.h” file in the Prediction module provides a class called SemanticLSTMEvaluator to evaluate obstacles as well as obstacle history; Evaluate and ExtractObstacleHistory respectively. The Perception module gives perception data to the Prediction model here to evaluate any obstacles and their history and then receives prediction data on how the obstacle may respond.

Prediction → Map
apollo/modules/prediction/container/obstacles/obstacle_clusters.h depends on *apollo/modules/map/hdmap/hdmap_common.h*
Description: The “obstacle_clusters.h” file in the Prediction module is used to analyse obstacles using lane and obstacle information. The “hdmap_common.h” file in the map module is used to get information from many different road aspects, such as lane information, junction information, crosswalk information, and much more. The Prediction module file here uses the LaneInfo class in the Map module to attain lane information and then uses this data along with its obstacle information as input to create a lane graph using the GetLaneGraph function in the ObstacleClusters class.

Task Manager → HMI
apollo/modules/taks_manager/cycle_routing_manager.h depends on *apollo/modules/dreamview/backend/map/map_service.h*
Description: The “cycle_routing_manager.h” file in the Task Manager module keeps track of the vehicle's position in comparison to its start and end route and recalibrates routing when need be. The “map_service.h” file in the HMI module retrieves map elements such as routing paths, nearest lane, waypoint, and even reloads the map. The Task Manager module checks routing information in the CycleRoutingManager class, verifies it, and creates a new routing request in the MapService function in the “map_service.h” file.

Task Manager → Map
Apollo/modules/task_manager/dead_end_routing_manager.cc depends on *apollo/modules/map/hdmap/hdmap_util.h*

Description: The “dead_end_routing_manager.cc” file in the Task Manager module contains a class called DeadEndRoutingManager that has functions for judging if a car is in a dead end and rerouting if this is true. The “hdmap_util.h” has imports and functions that give information on the localization of the car in reference to the road like getting signal information, junction information, and more. The Task Manager module uses information from the Module, specifically the GetJunctions function, to check if the vehicle is in a spot with no junction ahead and therefore is a dead end. The Task Manager then runs a rerouting function, GetNewRouting, to help the car get out of the dead end.

Reflexion Analysis for Tasks (2nd level subsystem of Planning)

The *apollo/modules/planning/tasks* folder contains 3 main components that handle decision making, optimization, and learning of the model – deciders, optimizers, and learning_model folders respectively - so that the Planning module can give accurate data to other planning subsystems such as the scenario handling and open space planning subsystems. The deciders folder contains subfolders that helps the vehicle make driving decisions such as whether to change lanes, what speed to drive at, and when to start the stopping process. These processes leverage the Perception, HMI, and Map modules and their classes to give further direction to these decisions. The optimizers folder finds optimized routes and speeds to get the vehicle to its destination of choice; it uses functions from the Map module to create a road graph and finds the nodes on this graph to optimize efficiency. The learning model is used to help infer what to do based on Localization and trajectory data. Below are a list of unexpected dependencies:

Planning/tasks/deciders/creep_decider → Map
apollo/modules/planning/tasks/deciders/creep_decider/creep_decider.h depends on apollo/modules/map//hdmap/hdmap_util.h
Description: The “creep_decider.cc” file plans how far the car is from an intersection and decides how far the vehicle should move to get to the proper spot in the junction. The “hdmap_util.h” file in the Map module includes map information and has many useful functions in its HDMap class that get data on intersections and other road-related data. The Tasks submodule in the Planning module here uses the Map module to attain distance information on how far the car is to the intersection so it can plan its route accordingly and maneuver the car to the perfect position at the intersection.

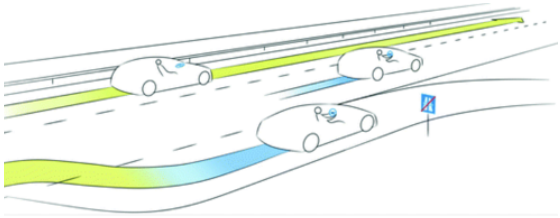
Planning/tasks/deciders/lane_change_decider → Map
apollo/modules/planning/tasks/deciders/lane_change_decide /lane_change_decider.cc depends on apollo/modules/map/pnc_map/route_segments.h
Description: The “lane_change_decider.h” file has a class called LaneChangeDecider to decide whether the vehicle should change lanes or not and has functions for checking if the perception is blocked, if it’s clear to change lanes, and update change lane status – functions IsPerceptionBlocked, IsclearToChangeLane, and UpdateStatus respectively. The “route_segments.h” file provides a class called RouteSegments that contains functions for setting the vehicle's next action, setting the vehicle's previous action, and the current lane segment – functions SetNextAction, SetPreviousAction, and WithinLaneSegment respectively. The tasks submodule (specifically the file “lane_change_decider.h”) in the Planning module uses these functions from the Map module to attain lane information for the vehicles lane changing planning system.

Planning/tasks/optimizers → Planning/reference_line
apollo/modules/planning/tasks/optimizers/road_graph/dp_road_graph.cc depends on apollo/modules/planning/reference_line/reference_line.h

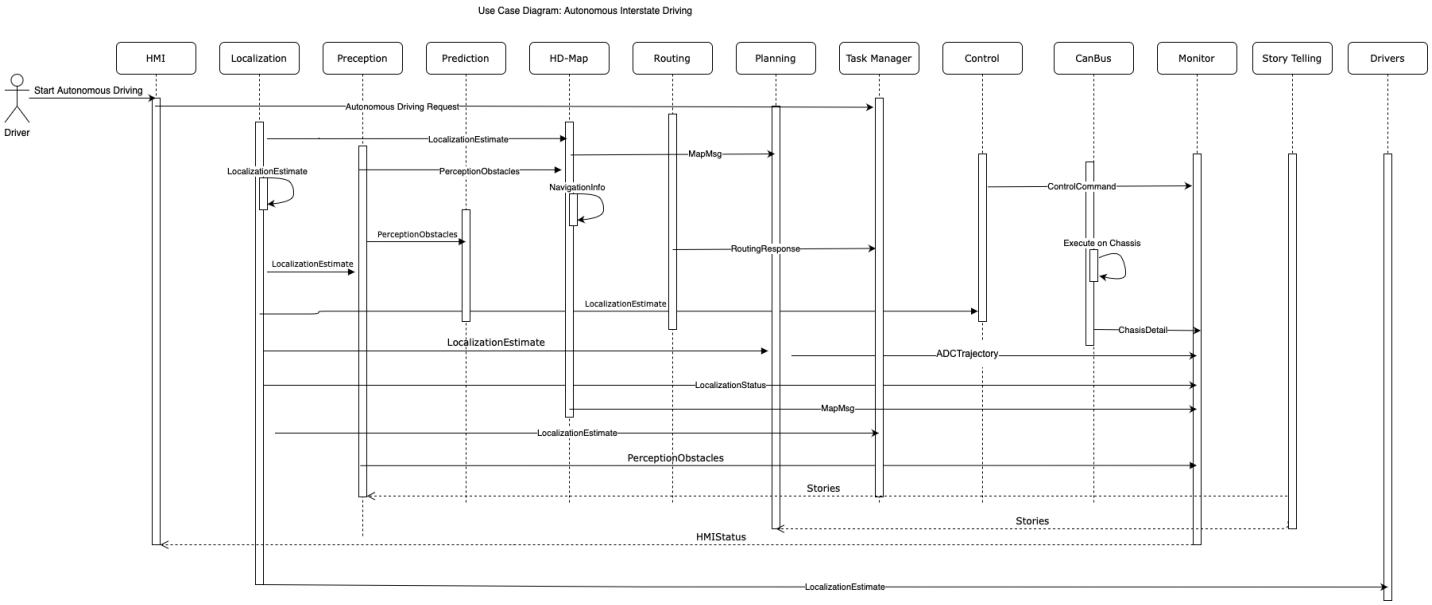
Description: The “dp_road_graph.cc” file in the tasks submodule creates a road graph in the RoadGraph class and generates a minimum cost path to a destination using a dynamic programming algorithmic approach. The “reference_line.h” file uses map geometry and road information to create a road reference line in the ReferenceLine class that also has useful functions getting certain reference points, lane width, speed limits, and more. The “dp_road_graph.cc” file uses this “reference_line.h” files reference nodes for road geometry and puts them together in a road graph so that it can find the min cost path to its destination.

Use Cases

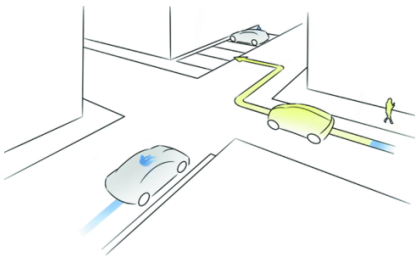
Interstate Pilot Using Driver for Extended Availability



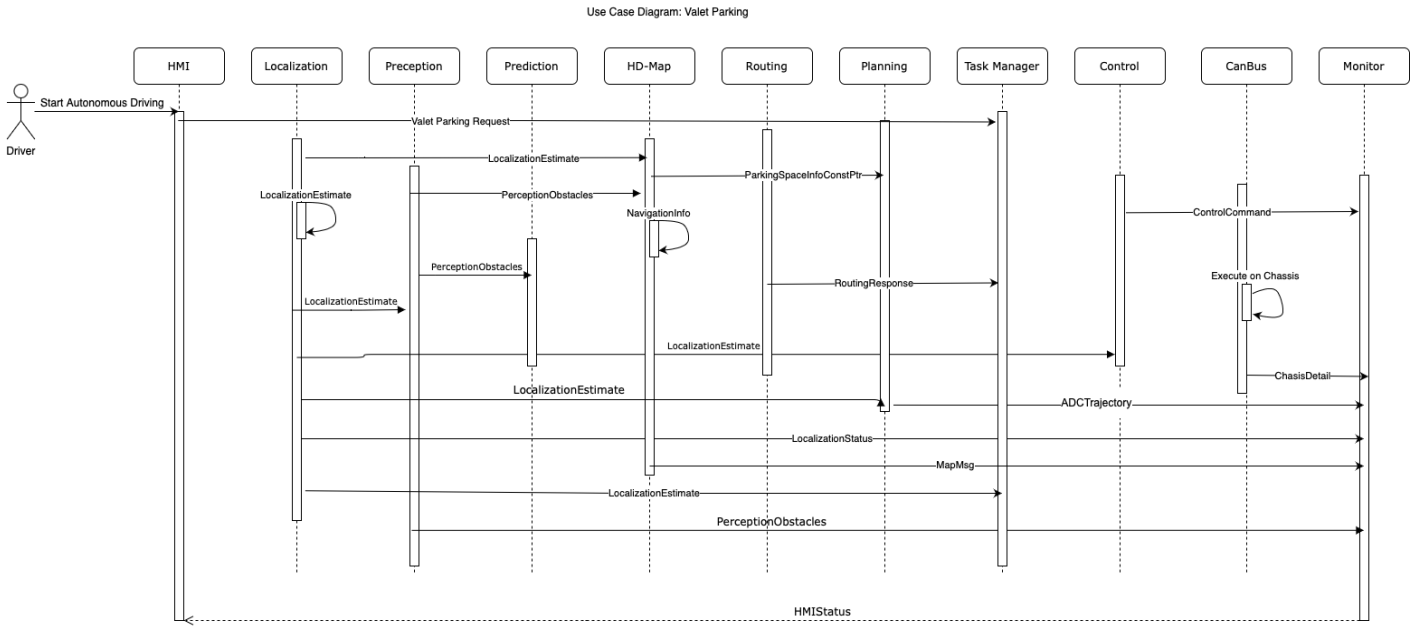
Description: As soon as the driver has entered the interstate, if desired, they can activate the autonomous driving software. The driving robot takes over navigation, guidance, and control until the exit of the interstate is reached. The driving robot safely coordinates the handover to the driver.



Autonomous Valet Parking



Description: Once a driver has reached their destination, they stop the vehicle, exit, and order the driving robot to park the vehicle. It is important to assign a parking lot to the driving robot because the search for the respective parking lot by the driving robot is not taken into consideration for this use case and so, a defined destination for the driving robot is always given.



Data Dictionary

Smarter Eye	“A driving assistant system which provides users with automobile autopilot systems. It uses different technologies such as three-dimensional depth of vision, image processing, video analysis, pattern recognition, and data mining.” (tracxn.com)
Publish-Sub scribe Architecture	An architectural style that allows independently implemented system components to register themselves as a publisher or subscriber for a particular event or message channel. That is, the publisher produces an event or message to the channel, and the subscriber receives that information.
Conceptual Architecture	Describes the system in a general way, without going into implementation details, but describing the requirements for the system and approaches to solving them. For example, a description of a set of system modules and the nature of their interaction without a specific description of the implementation of the modules and details of their interaction, such as data structures.
Ultrasonic Radar	A device for determining distance using reflected ultrasonic waves. The Ultrasonic Radar devices are significantly cheaper than LiDAR devices but give significantly less accurate results.

Naming Conventions

GPS	Global Positioning System
HMI	Human Machine Interface

LiDAR	Light Detection and Ranging
GNSS	Global Navigation Satellite System
RTK	Real-Time Kinematics
CANBus	Controller Area Network Bus

Limitations & Lessons Learned

Lessons Learned:

- We learned how to create a concrete architectural mapping using the Understand tool. This involved lots of documentation reading from individual modules on Github and looking at function calls as well as imports in the source code. Creating this concrete architecture really gave us a better understanding of what belongs where and the actual control flow of the system.
- We learned how to transform our conceptual architecture graph into a newly improved concrete architecture graph with added subsystems and control flow. We initially found it difficult to incorporate both the dependencies from Understand and the Publish-Subscribe graph given by the professor since there were contrasting dependencies. We decided to then solely use Understand along with the mapping we previously had for the conceptual architecture. By doing this, we were able to better understand our progress from A1 and what the main differences were.
- We learned to do further research on google and find research papers on autonomous driving that mentioned the Apollo Auto system. These papers gave us a more in-depth understanding of the concrete architecture and helped us look at the different applications of the many subsystems. Importantly, this also helped us comprehend the different advantages and disadvantages of each functionality – especially when it came to analyzing the Localization module.

Limitations:

- The lack of documentation for certain modules, specifically the Drivers, Guardian and Task manager modules, limited our ability to talk in-depth about the functionality of the system. This forced us to look at the source code and make inferences based on function calls and in-code comments when adding descriptions for certain subsystems. We had group discussions about these systems instead, which in turn helped us brainstorm design concepts.
- There were also limitations of the Understand tool, as it doesn't show all of the dependencies of every subsystem. We noticed this because of the professor's dependency graph that included numerous dependencies not present in our Understand architecture mapping. We found that certain folders and files were not included in the file system in Understand and this clearly accounted for the missing and contrasting dependencies.

Conclusion

After reviewing Apollo's source code through the Understand tool, we have developed its concrete architecture. We were able to uncover the unexpected dependencies and investigate further into the different subsystems that makeup the foundation of the concrete architecture. In the process also uncovered unexpected dependencies and modules that exist in the concrete architecture but not the

original conceptual architecture. The overall process of comparing the conceptual and concrete architecture has demonstrated how interwoven the architecture truly is.

In the future, we will propose enhancements that can be made to Apollo that are not in the current version of the studied software system. The enchantment feature will be presented in two different ways as an SEI SAAM architecture analysis. The group looks forward to using the information gathered from the conceptual and concrete architecture towards proposing features for Apollo.

References

Apollo Documentation:

1. https://github.com/ApolloAuto/apollo

Use Cases:

2. https://link.springer.com/chapter/10.1007/978-3-662-48847-8_2

Map:

3. http://road2ai.info/2018/08/11/Apollo_02/

4. https://www.ntnu.edu/documents/1284037699/1285579906/Gran-ChristofferWilhelm_2019_Master_NAP_HDMaps.pdf/79ef2eec-c9e2-454b-bf14-08d585cf8826

Architecture Styles:

5. https://www.tutorialspoint.com/software_architecture_design/data_flow_architecture.htm#:~:text=Process%20Control%20Architecture,or%20modules%20and%20connects%20them.

Architecture Analysis:

6. https://onq.queensu.ca/d2l/le/content/642417/viewContent/3814366/View

Research papers on Apollo Localization:

7. https://arxiv.org/pdf/1711.05805.pdf
8. https://ieeexplore.ieee.org/document/8461224

Smarter Eye:

9. https://tracxn.com/d/companies/smartereye.com