CISC 322

Assignment 2
Concrete Architecture of Apollo

Monday, March 21, 2022

FortyOne

Adam Cockell - 18aknc@queensu.ca
Ashton Thomas - 18ast8@queensu.ca
Dahyun JIN - 18dj8@queensu.ca
Kevin Subagaran - 19kks1@queensu.ca
Udbhav Balaji - 19ub@queensu.ca
Udit Kapoor - udit.kapoor@queensu.ca

Table of Contents

Abstract

Our previous report explored the Apollo open source project to discuss the various components and interactions within the system. In hindsight, we realize our report failed to discuss the architectural style of the system in depth, and identify the PubSub style in particular. Thus, we will use the proposed architecture from group 6 to support our own findings in the reflexion analysis. Based on the conceptual architecture and analysis of the source code using SciTools Understand, this report will propose a concrete architecture for the Apollo project.

Introduction

In this report we aim to determine the concrete architecture of Apollo and perform a reflexion analysis between it and the conceptual architecture discussed in the report from Group 6. This analysis is supported using Understand to extract code dependencies within the system and identify an optimized concrete architecture. We will discuss any discrepancies between the conceptual and concrete architectures discovered in reflexion analysis and make changes to the conceptual architecture in order to minimize unexpected dependencies and better match the concrete architecture.

Aside from reflexion analysis, we will discuss the addition of two new components – Monitor and Guardian – and how they integrate with the existing components in the system, with further analysis provided on one of the new components. Following that, we will cover two use cases in the context of the extracted concrete architecture, and how the Apollo system works to solve a given problem. Finally, we will discuss issues our team encountered in researching and writing and conclude with lessons learned while working on this report.

Our previous report only briefly mentioned the pipe-and-filter architectural style and how it related to the Apollo project at a high level. However, that style fails to capture the complexity of the system and interactions within it. In this report, we will combine our previous assessment of the Apollo architectural style with that included in the report written by group 6. Our conceptual architecture was expanded to include Publish-Subscribe, Repository, Process Control and Client-Server styles to better represent the flow of data and control within the Apollo open source system.

Architecture

Review of Subsystems

Map Engine: This module provides high-fidelity map data - using the HD-map component - to the next 4 modules.

Localization: This module provides high-accuracy positional data based on GPS, IMU, and other systems to the next 4 modules.

Perception: This module detects obstacles and the general surroundings of the vehicle using cameras, radar, and LiDAR. This information is sent to the Prediction and Planning modules.

Prediction: This module uses current velocity data of detected obstacles to calculate where they will be in the future. This information is then sent to the Planning module.

Planning: This module is responsible for choosing the path of least risk given all the information from the prior modules. Additionally, the chosen path must abide by traffic laws, and be comfortable to navigate for any passengers on board as the chosen path is followed by the Control module.

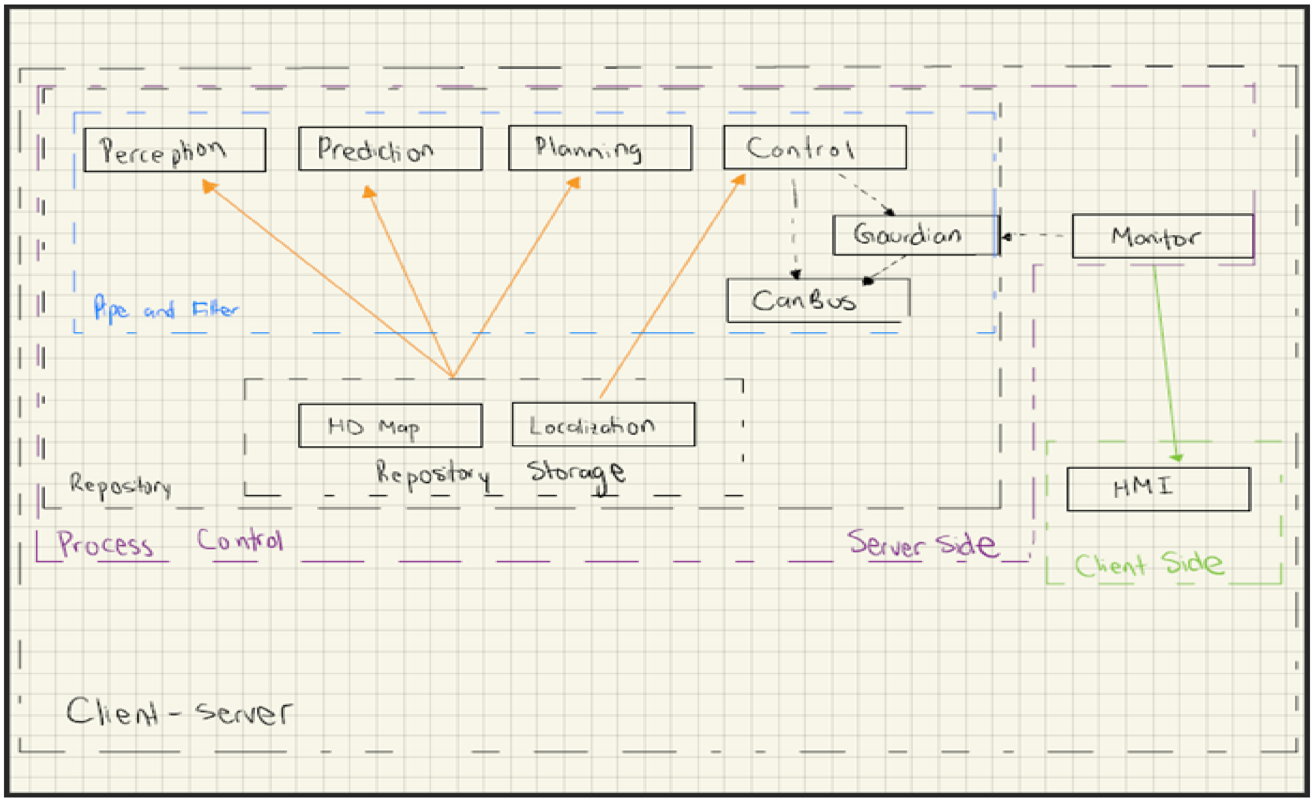
Control: This module takes in input from the Planning, Localization and Dreamview/HMI modules to output control commands, which include steering, throttle and braking, to the chassis.

HMI: This module provides a web-application that helps developers visualize the output of other relevant autonomous driving modules. This can include driving trajectories, car localization, etc.

Monitor: This module monitors the other modules to verify that all the systems are functioning normally. This includes hardware resource usage, data integrity, and latency metrics.

Guardian: This module secures the entire system in case of failure in a component. When a failure is detected, control flow is stopped so that potentially dangerous commands are not issued to the vehicle.

Conceptual Architecture



Using Group 6: Dash's A1 report, we were able to fix the errors made in our previous submission, and modified the conceptual architecture section. The pipe-filter, repository, process control, and client-server architectures are used to construct this complicated design, which are quite dependent on each other.

The pipe-filter style is the primary style that allows Apollo to dynamically produce control commands for the automobile. Perception, prediction, planning, and control modules are the filters, and the pipe is the control or data flow between them. The perception system detects and recognizes barriers and traffic signals before sending the information to the prediction module. The prediction then utilizes the data to estimate obstacle trajectories before passing the information on to planning to plot the course. Finally, the control module receives a copy of the trajectory from planning, which is used to produce the car's control commands. As a result, this pipe and filter design outputs the car's control directives from real-time data.

The repository style sits on top of the pipe and filter styles. The map engine and localization together form a central data repository that delivers data to perception, prediction, planning, and control modules through data lines. It's worth noting that, while the map engine and localization as a whole are the fundamental shared data, localization requires data from the map engine in order to build a positioning solution.

Meanwhile, a process control style is maintained with the inclusion of two new components: monitor and guardian. These two components collaborate to provide a safe driving experience. The Monitor module keeps track of the status of all modules, as well as their data integrity and frequency. If the monitor identifies a problem, it will alert the guardian module and interfere in the drive. The guardian has control algorithms that create a control command and deliver it to CANbus directly.

Next, Apollo has a client-server design. HMI is a client that shows the user all of the relevant data and vehicle status, and is then in charge of displaying all of the input data on the stage. The following modules are server-side and deliver data or control commands to the localhost port on a regular basis. Finally, the PubSub architectural pattern ensures little risk of performance degradation and not a lot of complex programming is required, providing a great deal of scalability and flexibility (Toolworks, 2022).

Derivation Process

To derive the concrete architecture of Apollo, we analyzed the static dependencies in the code base and looked at the pub-sub message traffic between different modules.

We used a tool called SciTools Understand that allows us to visually analyze the dependencies of a code base. We used an extracted dependency file to visualize the file structure and dependencies of the system. A rough mapping was generated by loading up the top level folders and the graph can be seen in figure 1.

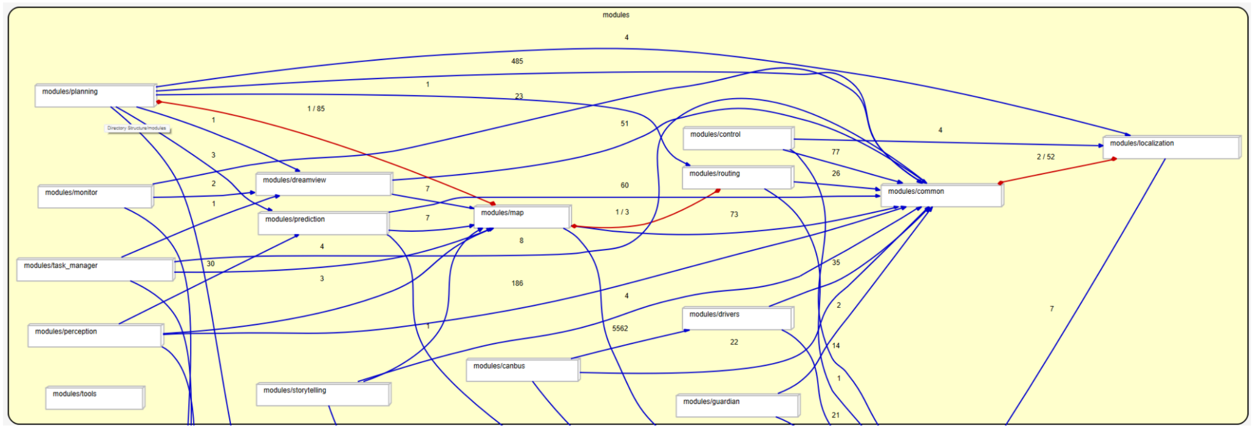


figure 1

This was a good starting point to fine-tune the mapping. We started by finalizing the subsystems. We added a common utilities module since it was large and most modules used files from the common module. After that, we started mapping the source code to the different subsystems so we could better analyze the architecture with the help of Understand.

We kept simplifying the modules by combining ones that fit together to better match the conceptual architecture. This also enabled us to visualize the dependencies better. Figure 2 shows a much simpler graph. After reaching this step, we could remove connections with too few dependencies, since almost everything has a dependency in a given module in such a large code base.

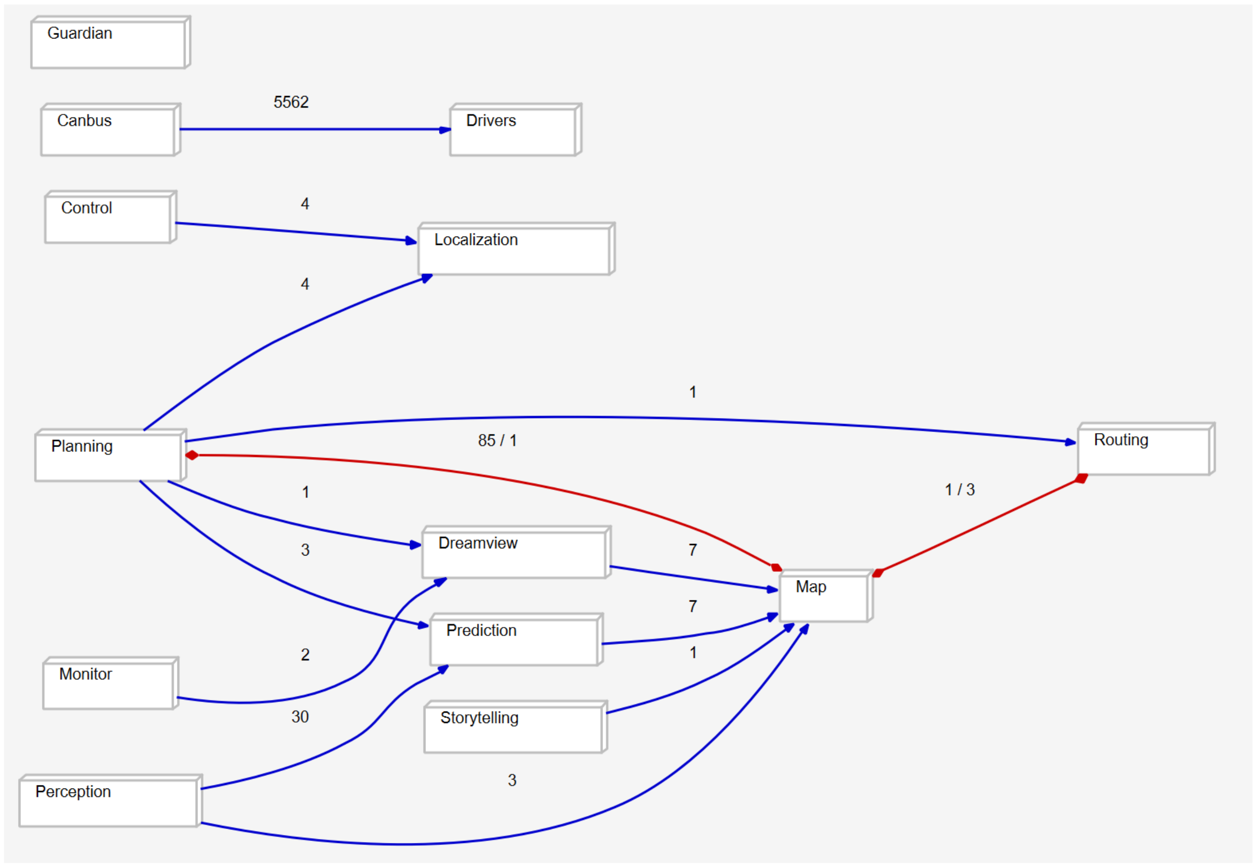


figure 2

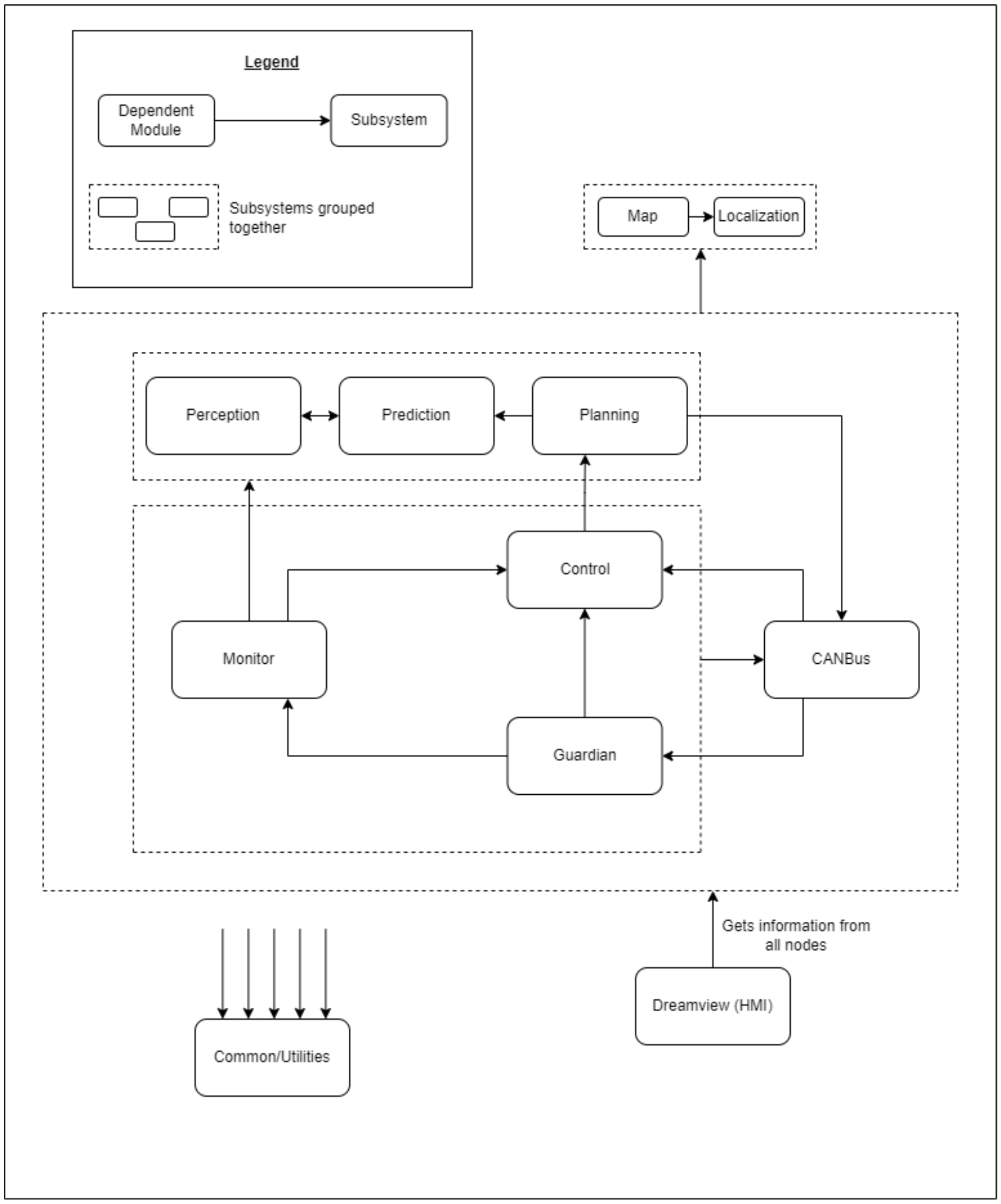
Combining this data with the pub-sub message traffic graph, we were able to extract the final concrete architecture of the system.

Concrete Architecture

The concrete architecture of Apollo looks quite similar to the conceptual architecture, but shows that the modules are more interdependent on each other than what it looks like.

The software follows a Pub-Sub architecture model, where several modules "publish" their results and data which can be "subscribed" by other modules for their use. While many modules don't have static code dependencies on some other modules, they subscribe to their data outputs to function correctly (Spurrett & Jackson, 2001).

We kept most of the subsystems the same as the conceptual architecture, but added a common/utilities module since it was too large to ignore and almost all other subsystems depended on it. We also grouped together modules with common properties to make the architecture easier to understand. For example, Perception, Prediction and Planning are all closely linked together and act as the main core of the system. Monitor module can then have a single dependency on this group rather than multiple individual dependencies. Apart from this, we decided to not make too many modifications to the original subsystems since they were very important to the software.



Control is a subsystem that uses different algorithms for a comfortable driving experience based on the planning trajectory and the car’s current status. Also, Control works both in normal and navigation modes. Based on Apollo 5.5, Control got new features such as Model Reference Adaptive Control(MRAC) and Control Profiling Service ("Scitools Licensing | Download Understand", 2022).

The purpose of MRAC is to effectively offset the by-wire steering dynamic delay and time latency. In addition, it also ensures faster and more accurate steering control actions for tracking the planning trajectory.

Control Profiling Service provides a systematic quantitative analysis of the vehicle's control performance via road-tests or simulation data. Furthermore, it provides users with insights into the current performance and improvements seen within the module and helps support the high-efficiency iterative design cycle of the vehicle's control system.

Control gets inputs such as planning trajectory, car status, localization, and dream view AUTO mode change request while output control commands like steering, throttle and brake to the chassis. In the conceptual view, Control is controlled by a planning module and control guardian and CANbus. Data including Control is shared to monitor and control HMI. Unlike conceptual view, in concrete view, Control has direct connection to monitor, and also Map and Localization are grouped separately from Control.

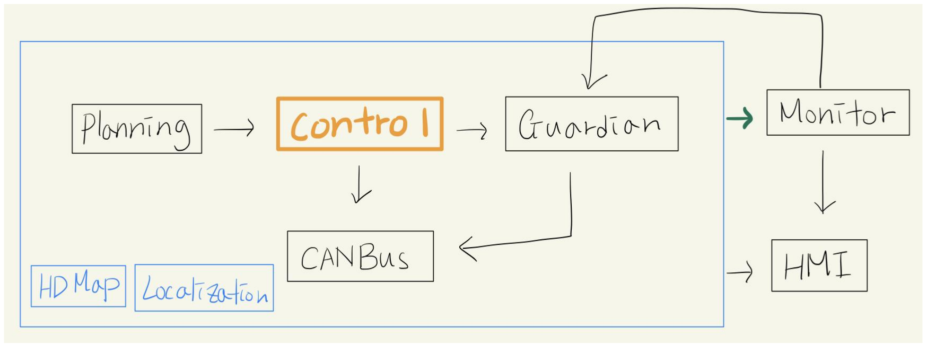


Figure of control in conceptual architecture

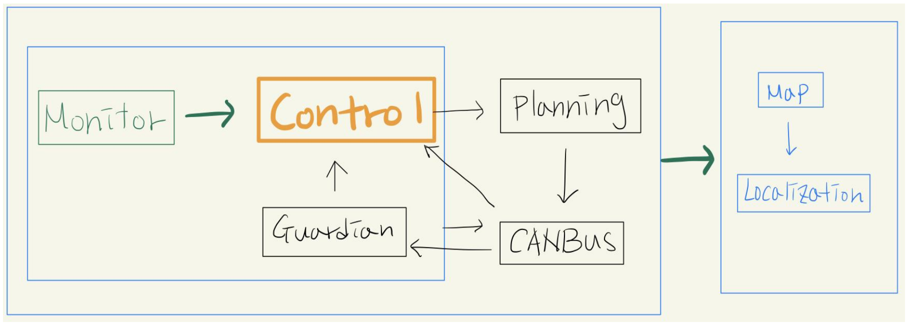


Figure of control in concrete architecture

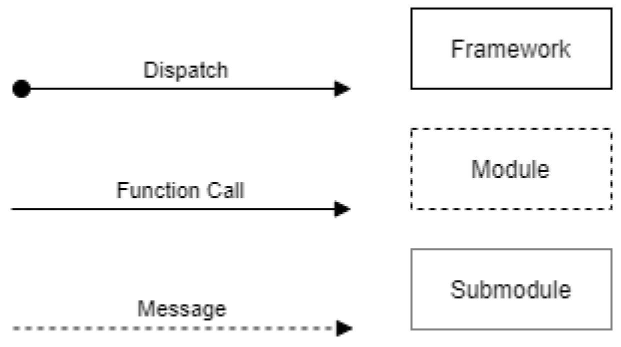
Discrepancies Between Conceptual and Concrete Architecture

As stated above, we have tried to minimize the changes from the Conceptual architecture to the Concrete architecture. A major change that we did to the high-level architecture of Apollo is the addition of the Utilities module, which effectively links all the other modules together. The addition of this module helps facilitate the flow of data and control across the system, thus making Apollo as robust as possible. Along with this, another modification is that modules with similar tasks are grouped together in order to enhance user readability and understanding. Where processes are closely linked together, this works really well as it indicates that a set of modules are closely linked and are needed together in order for the system to work effectively and efficiently. Additionally, the architecture style used in our concrete architecture is Pub-Sub, moving away from the conceptual architecture, where we have multiple styles (Pipe-Filter, Repository, Process Control and Client-Server styles) incorporated into the same architecture.

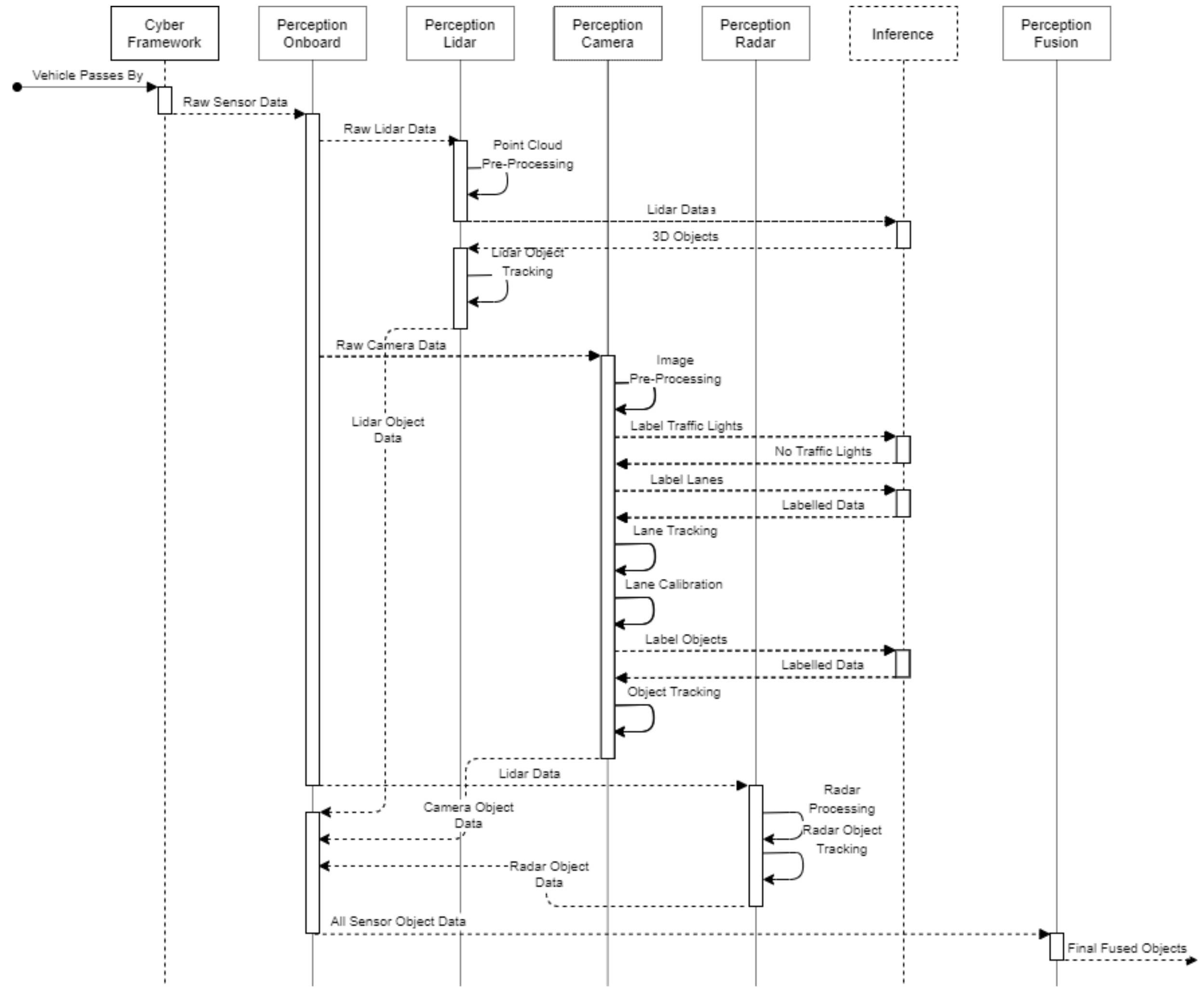
Apart from this, the Control module also has great features that will be incorporated into the architecture such as MRAC and CPS, which help the system react better to unexpected stimuli in the environment and help make Apollo a safe, stable option on the road. This includes getting more detailed data regarding car trajectories, as well as improved localization data, helping the car plan a better trajectory in times of need ("features — SciTools", 2022).

Reflexion Analysis: It is the analysis done of the architecture that goes through the components and subsystems to find gaps in the architecture. In our concrete architecture, a gap that might have to be filled would be the efficiency of data and flow control throughout the system. This would further improve response time as well as decrease the resource demand on the system. A major change in this respect would be regarding the HMI module. In the conceptual architecture, it only has the Monitor module as a dependency, while in the concrete architecture, it has all the modules as dependencies. In order to investigate this further, the information being passed onto the module needs to be further understood. This module builds a web-application for developers to visualize key aspects of the autonomous system, such as planning trajectories, car localization, etc. Since it takes input from all the other modules, a direct link between the other modules and the HMI might improve efficiency but reduce understanding of the architecture itself and how this module works.

Use Cases



Legend



Use Case #1: Perception Module: A Vehicle Passes a Moving Apollo-Equipped Car

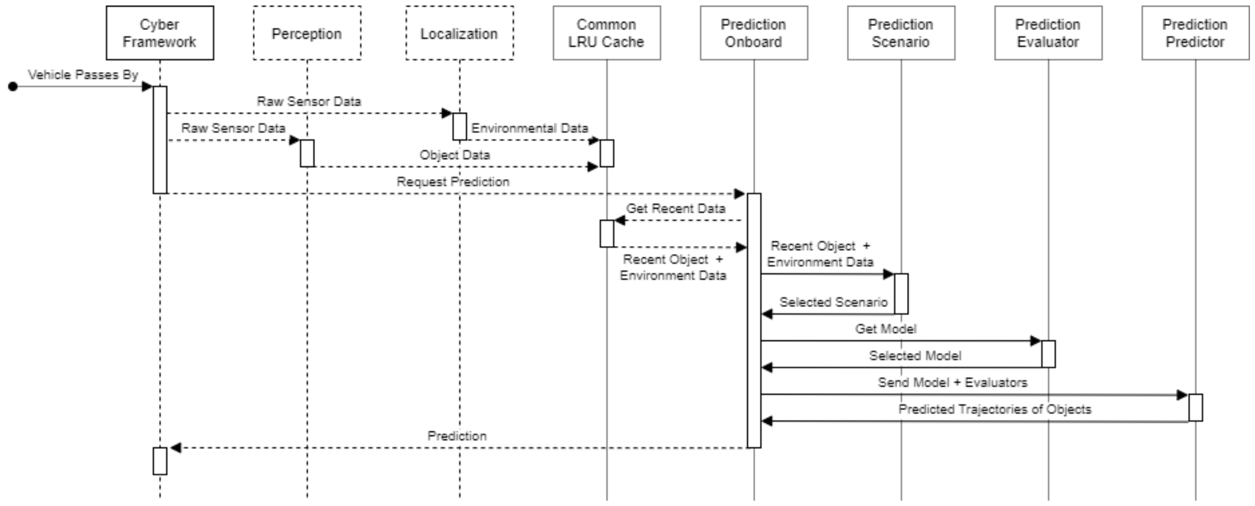
When a vehicle passes by, the raw data from the Apollo-Equipped Car's sensors is sent to the Perception module's Onboard submodule. The Onboard submodule then sends each sensor's raw data to its respective Perception submodule for processing.

When the Perception Lidar submodule receives the raw data it first performs point cloud pre-processing and sends the data to the Inference module. The Inference module then returns the 3D Objects from the point cloud data, the objects are then tracked by the Lidar submodule and finally returns it to the Onboard submodule.

When the Perception Camera submodule receives the raw image it first pre-processes the image and then sends the data to the Inference module. The Inference module then labels the data. The data is returned to the Camera submodule, where the submodule learns that the image has no traffic lights. The Camera submodule then sends the data to the Inference module for lane labeling, after receiving the labeled data it then tracks and calibrates lanes. The Camera submodule once again sends the data to the Inference module to label objects and then tracks the objects. Finally, the Camera submodule sends the data to the Onboard submodule.

When the Perception Radar submodule receives the raw data it first processes the data. It then tracks the objects found in the data and returns it to the Onboard submodule.

After the Perception Onboard submodule receives all the object data, it then sends it to the Perception Fusion submodule to fuse the data to create the obstacle data with positions, velocities, accelerations, and more.



Use Case #2: Prediction Module: A Vehicle Passes a Moving Apollo-Equipped Car

When a vehicle passes by, the raw data from the Apollo-Equipped Car's sensors is sent to the Perception and Localization modules to get object and environmental data. After the modules process the raw data from sensors, the data is sent to the Common LRU Cache submodule. The LRU Cache stores the most recently detected obstacles and environmental data, which will be used by the Prediction module. The Cyber Framework then requests a prediction from the Prediction Onboard submodule.

The Prediction Onboard submodule first get's the most relevant object and environmental data from the LRU Cache submodule. It then sends the data to the Prediction Scenario submodule to select a relevant scenario, in this case would be "regular road" with "normal" object prioritization.

The Onboard submodule then requests a model and evaluator from the Prediction Evaluator submodule, and then sends them to the Prediction Predictor submodule which returns the predictions. Finally, after receiving the predictions the Onboard submodule sends the prediction to the Cyber Framework.

Lessons Learned

We learned a lot of things while working on the concrete architecture. We learned how to use SciTools Understand to analyze the code dependencies of a codebase and learn more about its architecture. Apollo is an incredibly complex software that has been in development for almost a decade and it took a lot of reading code, documentation and PubSub relations to understand the concrete architecture.

Conclusion

From the mappings deduced for assignment 1 and analyzing Apollo's codebase using SciTools Understand, we were able to find the Concrete Architecture of the software. While it was similar to the Conceptual Architecture in many ways, there were also many key differences. There were significantly more dependencies between modules, likely due to the Concrete Architecture evolving over time from its initial start as the Conceptual Architecture we deduced.

Additionally, after analyzing Apollo's codebase and using SciTools Understand we were able to create the concrete sequence diagrams for our selected use cases. By contrasting the concrete and conceptual sequence diagrams it is apparent that the software uses a pub-sub message system. This system was chosen as it is the best fit for a real-time operating system, and allows for easy real-time communication between modules and their submodules. It also suits the system as it streams sensor data directly to the perception and prediction modules. The concrete sequence diagram also showcases the cyber framework and its communication between modules.

References

[1] *features — SciTools*. SciTools. (2022). Retrieved 22 March 2022, from https://www.scitools.com/features.

[2] *Scitools Licensing | Download Understand.* Licensing.scitools.com. (2022). Retrieved 22 March 2022, from https://licensing.scitools.com/download.

[3] Spurrett, D., & Jackson, C. (2001). Virtually incomprehensible: pros and cons of WWW-based communication and education. *SA Journal Of Information Management, 3(1)*. https://doi.org/10.4102/sajim.v3i1.122

[4] Toolworks, S. (2022). *User Guide and Reference Manual [Ebook] (pp. 0-22).* Scientific Toolworks, Inc. Retrieved 22 March 2022, from https://documentation.scitools.com/pdf/understand.pdf.