Overweight and overload implementation in Apollo systems

Hairuo Li

Queens University, Canada

19hl12@queensu.ca

Abstract. Apollo is a high-performance, flexible architecture developed by Baidu, Kinglong, and a consortium of more than 40 companies for the purpose of accelerating the development, testing, and deployment of Autonomous Vehicles. This article first describes Apollo in terms of the functionality of the system and the interaction between its components, and then analyzes the conceptual architecture through system evolution, control and data flow, and concurrency, while illustrating the impact of the division of developer responsibilities on this. Then will go through a detailed overview of the concrete architecture of Baidu Apollo with working on mapping the source code from the Apollo Github Website to our architecture using the diagram drawing tool Scitools Understand. And divided into five parts, the report concludes with five major parts: the process from mapping code to diagram; concrete architecture explanation; unexpected dependencies discovery; subsystem analysis, and sequence diagrams presented in a more accurate and specific way. Finally, this report will propose a specific feature or enhancement to the current concrete architecture. The impact that this feature or enhancement can have on the non-functional requirements and stakeholders of the system will be explored based on the existing architecture and component interactions. In this regard, this report discusses two specific approaches that can be implemented and compares the two approaches through SAAM analysis to determine the best approach.

Keywords: Apollo Systems, SAAM analysis, Implementation.

1. Introduction

This report focuses on the problem of adding a new feature to Apollo cars to check for overweight and overcrowding, which are very dangerous and illegal, but there is no feature in Apollo to check for this behavior, so this report examines it.

Past research has focused on the functionality of each part of the apollo system, the bursts between modules, the data and control flows between them, and the source code of each module and its architecture. 457910 This report will first study the detailed architecture, and by studying the perceptual architecture, the source code, and the dependency graph, this report will come up with a new This report will first further investigate the detailed architecture by examining the perceptual architecture, as well as the source code, and the dependency graphs.

In this report, two specific implementations are investigated, and how they specifically perform this function by adding it to the Monitor module or adding it to the CANBus module, how they interact with other modules, how they control the basin data flow, and how these two different implementations affect the architecture, which is further compared using the SAAM approach [1], and how they affect the code. and how they would affect the stability, maintainability, and performance of the code, to arrive at the relatively better implementation

2. Architecture

2.1. Conceptual Architecture

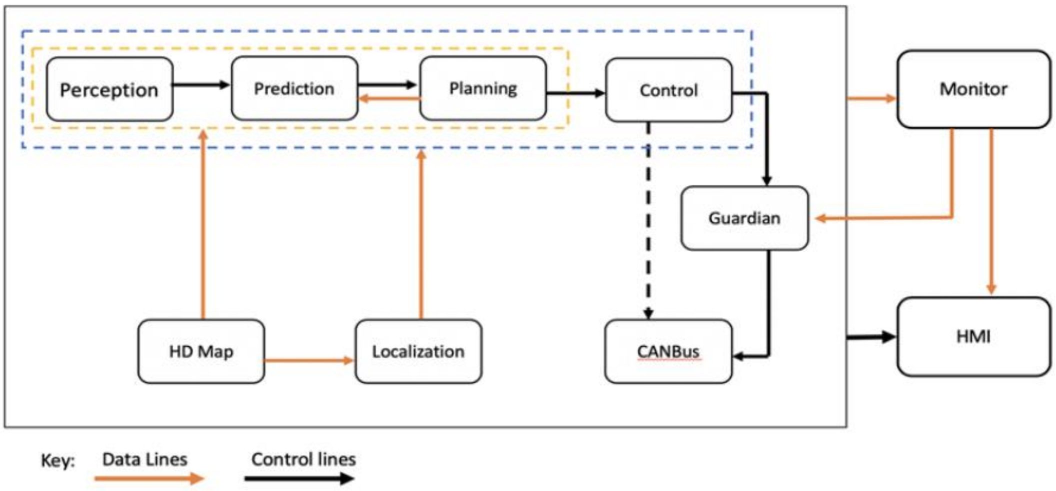

Figure 1. The pub-sub style diagram of conceptual architecture from the GitHub repository.

This report starts with a preliminary study of the conceptual architecture in GitHub (Figure 1), which is roughly divided into three levels according to functional categories: vehicle state information input and output, route planning, and vehicle control. The input and output of the car information contains Monitor and HMI modules, the car control part contains Control, Guardian, and CANBus module, the route planning part contains, Perception, Prediction, and Planning and the route planning to provide a high-definition map of HD Map and car surroundings and car location of Localization module [2].

2.2. Dependency Map

In the next step based on the understanding of the source code, the content contained in the source code is reclassified on the modules of the original conceptual structure [2]. Upon completion of this process, a dependency map is generated (Figure 2). After remapping the source code under the existing modules, many dependencies that were not shown before became more visible. The diagram gives the fact that these components have no obvious dependencies on another component that should be expected in the conceptual architecture.

It is worth noting that some of the modules in this diagram have very minor dependencies on each other. To minimize unexpected dependencies, such dependencies with too little weight were removed. The dependencies were then repeatedly filtered using SciToolsUnderly, and small dependencies were ignored when analyzing the specific architecture. During the investigation, the divergence between the conceptual architecture and the concrete architecture was clearly visible

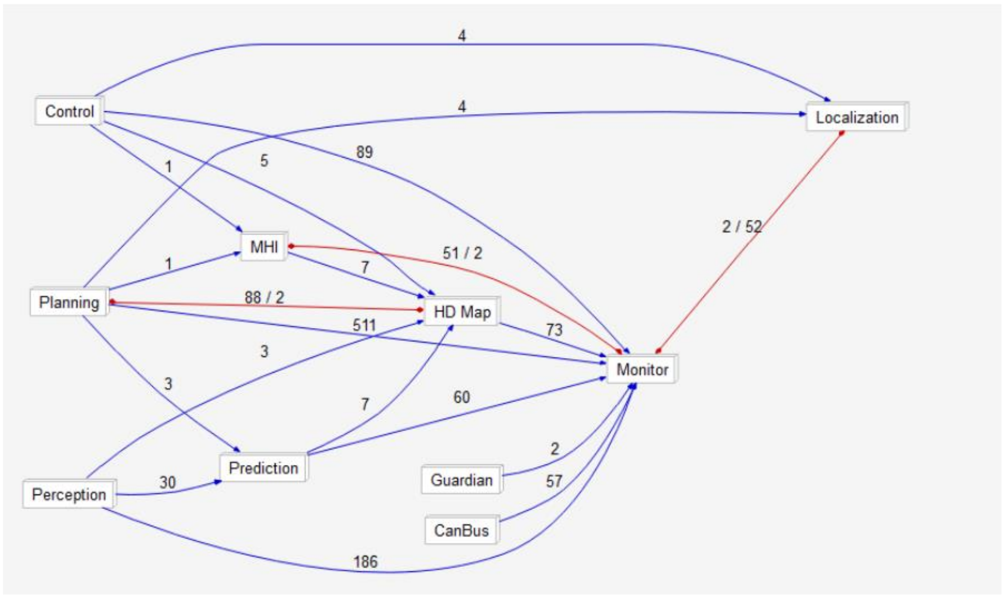

Figure 2. Dependency graph of top-level subsystems produced by SciTools Understand.

2.3. *Concrete Architecture*

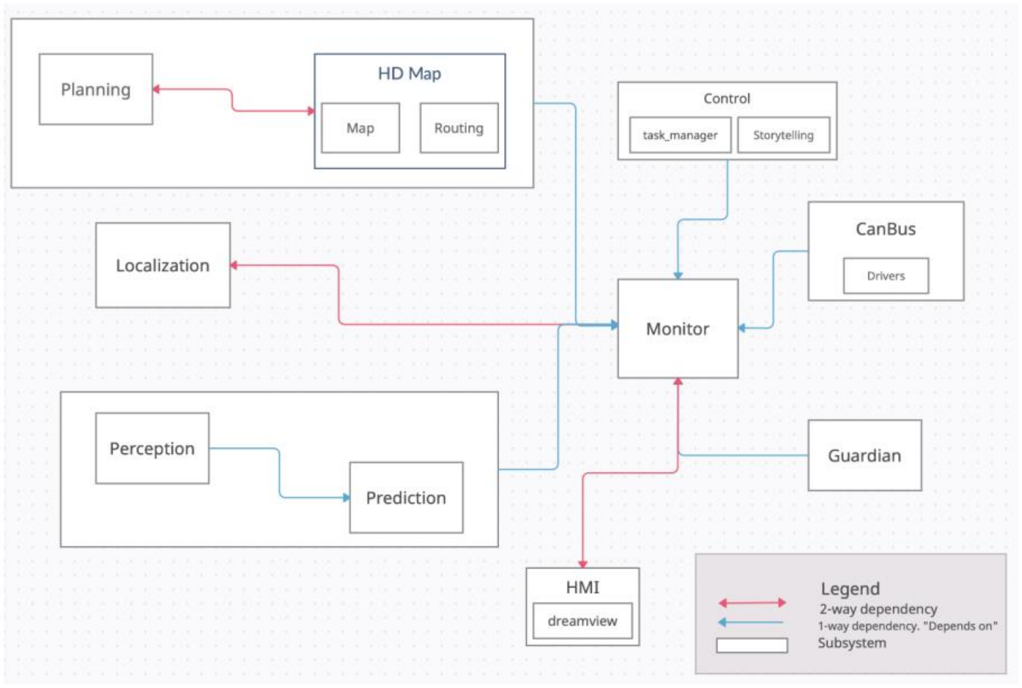

Figure 3. Concrete diagram conducted from the dependency graph.

For the Concrete Architecture, there is the main subsystem which is the Monitor and all other subsystems relate to it.

The HMI and the Monitor are a two-way dependency. There is a subsystem in the HMI which is dream view. Next, for the Guardian, it and Monitor are one-way dependencies, and the Guardian depends on the Monitor. The CANBus, has the same connection type as the Guardian, with a one-way dependency and it depends on the information from the Monitor. Also, the CANBus has a subsystem named Drivers. There are two systems called task_manager and Storytelling which are the subsystems for the Control. The system of Control and the Monitor are one-way dependencies, and the reaction of

Control depends on the information of the Monitor. For the system of Localization, it and Monitor are two-way dependencies, and they depend on each other.

The structures of some of the subsystems I mentioned earlier are relatively simple, but the structures of the next two subsystems will be more complicated.

There is a system, and it contains two subsystems which are Perception and Prediction. However, there is a connection between them, and the Perception depends on the Prediction, also they are a one-way dependency. This whole subsystem which contains Perception and Prediction, depends on the Monitor, and they are a one-way dependency. The last subsystem contains 3 levels. For the first level, the whole subsystem depends on the Monitor, also the connection between them is a one-way dependency. For the second level, there are two subsystems which are Planning and HD Map. The connection between them is two-way dependency and they depend on each other. Lastly, in the third level for this whole subsystem, there are two subsystems in the HD Map.

3. Proposed Enhancement

This report will discuss the addition of a feature to the Apollo autonomous driving system to detect if a car is overweight or overloaded. This new feature would eliminate overweight and overloading, which are both dangerous and illegal behaviors in traffic driving, and would greatly increase the safety of self-driving cars.

3.1. Used Architecture Styles

The Apollo self-driving car uses the pub-sub architecture style. We need to register overcrowding or overload as a new event in the system. When the system starts. It will start to detect and see if the car is overloaded or overcrowded at this time along with other systems, and if so, the car will be disabled from Start. The use of this style of architecture ensures that problems with the vehicle are detected promptly without delaying route planning.

3.2. Current Concrete Architecture

A previous study of the detailed architecture concluded that the Apollo Car architecture consists of 8 interacting components. Monitor, Locate, Sense, Predict, Plan, Control, Guardian, and CANBus. the main subsystems affected by our addition are the Monitor, Guardian, and CANBus components. [6, 7, 8]

3.3. Monitor (Alternative) implementation

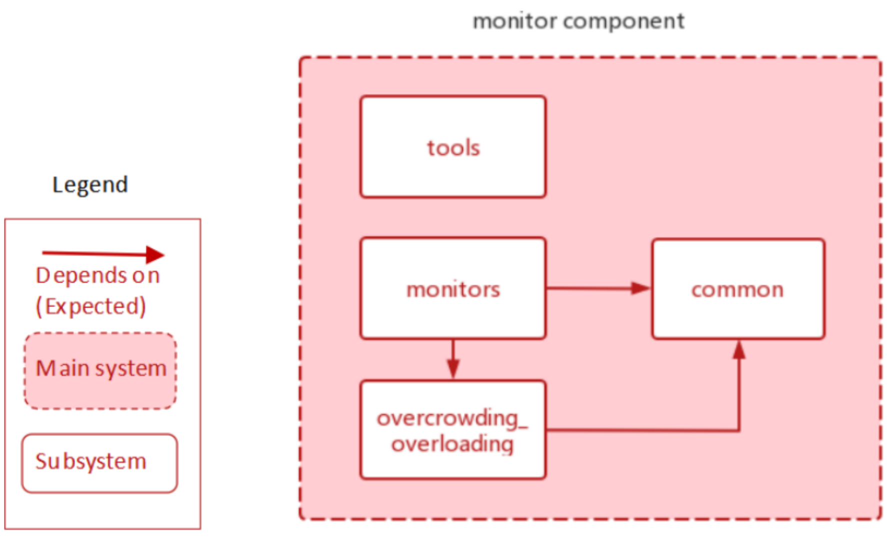

Figure 4. The interaction within the monitor subsystem.

As an alternative to implementing the overcrowding_overloading function, we have decided to add this module to the Monitor. We will add new data to the vehicle state in common/vehicle_state, such as the upper limit criteria for vehicle weight and people, and then the new overcrowding_overloading function will get the current vehicle weight and people data from the sensors (face recognition, and vehicle weight) and use the common_math are then compared to identify whether the vehicle is overweight or overloading, and if so the monitor gets the exception information and passes it to the HMI for a display to the user.

However, using this method to implement Overcrowding/overloading slows down the overall process of the program, as the car has to be maneuvered after the overcrowding_overloading has been identified. This requires Overcrowding/overloading to send the required movement of the car to the control, which in turn sends the specific operation to the CANBus for execution, leading to additional dependencies and more system operations, all of which result in a longer time between the identification of the overcrowding and the response of the car, as well as this can lead to a waste of system resources (the car is planning its route at the same time as it reacts) and thus to possible unsafe problems such as starting the car with an overload.

In order to avoid this waste of system resources and longer reaction times, we have chosen to abandon this approach for our new function. Furthermore, it is difficult to solve the problem of detecting the vehicle again while it is in motion, e.g. after stopping to pick up a new passenger or load a new load, because the function to obtain the real-time status of the vehicle itself is in the CANBus and the need to pull the information away would lead to an additional operation, which would not be possible to determine in time after the vehicle has been started.

3.4. CANBus (Actual) Implementation

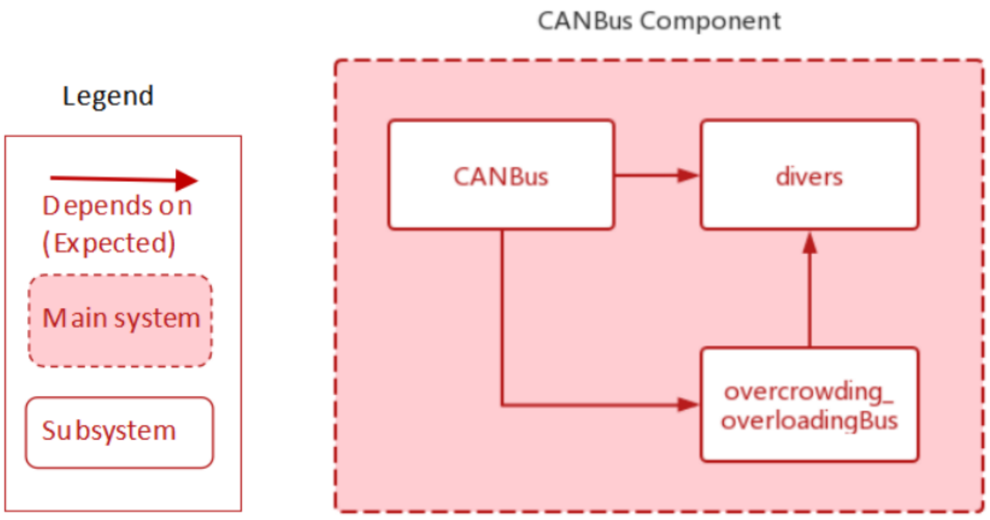

Figure 5. The interaction within the CANBus subsystem.

The actual implementation of overcrowding/overloading is to add this function to CANBus. CANBus_drivers drive the sensing system inside the vehicle and uses this module to provide real-time information about the occupants and load inside the vehicle for overcrowding/overloading, and then from CANBus, it tells the vehicle whether it is overcrowded and overloaded by comparing the standard weight and occupant limit from monitor_common, and then returns the overcrowding information to the control as status information. The CANBus also outputs this exception information to Guardian, then to Monitor, and finally to the user from the HMI.

In this case, there are no additional dependencies, the workflow is already in place and is determined before the CANBus executes the command from the control, so that the detection can be done completely before the car is started, thus avoiding the situation where the car is started and not yet judged. Furthermore, the CANBus itself is a module that periodically returns the vehicle status to control, so that it can also make a determination while the vehicle is in motion and only needs to notify control if it detects a significant weight change or an increase in personnel. This ensures that the vehicle is safer to drive and does not consume so many system resources that it slows down the process.

With this approach, we also need to add a new code in the control to determine the status of the overweight and overload of the vehicle, as well as a proxy to output an overweight and overload alert in the HMI and add the vehicle's original weight limit and occupant limit information to monitor_common_vehicle_state, which is not available in the original code.

3.5. SAAM Analysis

After proposing the enhancement, we should consider how the new enhanced component should interact with other components within the system, what interfaces it should be connected to and what kind of changes it will make to the system interaction at different interfaces.

We explored two implementable options for various variable factors. One is to put it in the monitor to implement the feature, and the other is to put it in the CANBus to monitor the in-vehicle situation. It is undeniable that both approaches can achieve the purpose of our idea, but we prefer to go for a more optimized result in terms of NFRs and beneficiaries.

3.5.1. Approach1(Monitor). Our first approach is to add this functionality directly to the Monitor component. This function enables the vehicle system to directly monitor the interior environment to confirm whether there is overcrowding and overloading.

We will add new vehicle state data in common/vehicle_state, such as vehicle weight and the upper limit of the number of people, then the new overcrowding_overloading function will take from the sensor (face recognition, vehicle weight) and use common_math to compare to identify whether the vehicle is overweight Or overloaded, if so, the monitor gets the exception information and passes it to the HMI for a display to the user[9].

This design can use overcrowding and overloading as part of the Monitor subsystem. This means that the Monitor can directly get the command returned after monitoring overcrowding and overloading. It can then interact with the control system or the main system to determine if there is overcrowding and overloading and what the vehicle should do next[7].

3.5.2. Approach2(CANBus). Our second approach is to add functionality directly to the CANBus component to enable the vehicle system to monitor the environment inside the vehicle directly. This design treats Overcrowding/overloading as part of the CANBus subsystem, which means that the CANBus can directly get the command returned in Overcrowding/overloading and then interact with control to determine what the vehicle should do next. It is worth mentioning that the most important non-functional requirements (NFRs) regarding enhancement are considered to have a low running speed

In fact, Apollo has been so well conceived in terms of software that we can only consider further enhancements in terms of safety and driving. Previously, CANBus served as a bridge between the control and the guardian, communicating abnormal situations and returning the required action.[10] But this decision was made after the route planning. If a vehicle is found to be overloaded/overweight after this planning, the process will stop. The route planning is done again until the next safety clearance is obtained. This can cause process waste to some extent.

By comparing the first implementation, we found that interacting as a subcomponent in the CANBus does not monitor the number of passengers in real-time. In the real case, the load weight only changes every time you get on and off the vehicle. We prefer to make a determination of overcrowding or overload after planning each travel path. This effectively reduces the number of data streams in the system and gets permission for safety when the vehicle is ready to operate.

Table 1. A comparison of the two approaches regarding performance and maintainability.

NFR'SImpacted	Approach1(Monitor)	Approach2(CANBus)
Performance	Plan the journey only afterchecking whether the car isovercrowding and overloading	Determination ofovercrowding or overloadingis made after planning asingle path of travel
Maintainability	This can lead to longer timesbetween identifying the systemovercrowding and the car'sresponse, and this can lead toa waste of system resources	No new dependenciesappear between higher-levelsubsystems

4. Conclusion

This report first investigates the conceptual architecture, then through the study of the source code and the use of SciToolsUnderly to construct a dependency graph, then from the dependency graph we infer the detailed architecture, and then in order to improve the safety of Apollo, we come up with an enhancement, i.e., a module (subsystem) that can additionally detect if the car is overweight and overloaded. Two different implementations of this enhancement were also analyzed for SAAM. We performed a SAAM analysis of two different implementations of this enhancement and found that implementing the system overweight/overload in the CANBus subsystem would achieve the results we wanted with the fastest synchronization. The results we wanted to achieve with the fastest synchronization. By comparing many sensors and formulas for detecting weight and the number of people with the CANBus subsystem, we avoided a huge impact on the overall structure without introducing additional complexity. By deriving the function definitions and analyzing the functions themselves, we concluded that the non-functional requirements (NFR) are highly maintainable and low risk. Ultimately, after careful planning and analysis, we believe that this is the best approach available to address safety issues such as overload and overweight.

References

[1] Kazman, R., Bass, L., Klein, M., Lattanze, T., & Northrop, L. (2005). A Basis for Analyzing Software Architecture Analysis Methods. Software Quality Journal, 13(4), 329 – 355. https://doi.org/10.1007/s11219-005-4250-1

[2] Apollo Auto/apollo. (2022). [C++]. Apollo Auto. https://github.com/ApolloAuto/apollo (Original work published 2017)

[3] Microsoft Document. " Publisher-Subscriber pattern " . Retrieved from https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

[4] salmcode (August 23, 2017). "The First Part of the [Apollo Source Code Analysis] Series [common]." Retrieved from https://blog.csdn.net/learnmoreonce/article/details/77511338

[5] Steve (October 20, 2021.) " Apollo 6.0 pnc_map Analysis " Retrieved from https://zhuanlan.zhihu.com/p/419350318

[6] Baidu. (n.d.). Apollo. (February 20, 2022) Retrieved from https://apollo.auto/

[7] Autonomous Driving Apollo Source Code Analysis Series, System Monitoring Part (II): How Monitor Module Monitors Hardware - Tencent Cloud Developer Community - Tencent Cloud. (n.d.). Retrieved October 9, 2022, from https://cloud.tencent.com/developer/article/1999068

[8] Autonomous Driving Apollo Source Code Analysis Series, Safety Guardian Part (I): Why Emergency Braking? -ZhiHu. (n.d.). Retrieved October 9, 2022, from https://zhuanlan.zhihu.com/p/455148614

[9] Apollo system monitoring module analysis (next part). (n.d.). Zhihu Column. Retrieved October 17, 2022, from https://zhuanlan.zhihu.com/p/450627615

[10] Apollo introduction of the Canbus module (VIII) - know. (n.d.). Retrieved October 9, 2022, from https://zhuanlan.zhihu.com/p/85083829