CISC 322

Assignment 1
Conceptual Architecture of Apollo

Feb 18, 2022

Group-14: Artemis

Adrian Chu	18anhc@queensu.ca
Alex Baldassare	18cab16@queensu.ca
Randy Bornstein	18rjb10@queensu.ca
Jake Halay	15jmh9@queensu.ca
Kashish Khandelwal	20kmk3@queensu.ca

Table of Contents

Abstract

This technical report contains an analysis of the Apollo autonomous vehicle conceptual software architecture. Our conceptual architecture was built using the open-source code in combination with various different reference architectures for autonomous vehicles. After careful consideration, our research found that there were multiple different conceptual architectures that the Apollo system uses. The architecture observed had many features associated with layered styles, client-server styles as well as pipe and filter styles. However, these architectures still do not encompass the various different subsystems required for an autonomous vehicle to function independently of human control.

Introduction and Overview

As technology continues to improve, many tasks previously done by a human can be relegated to autonomy. The next hurdle to overcome in the realm of autonomous driving is ensuring the safety of all users and others on the road. Apollo seeks to provide its users with an open-source platform for self-driving vehicles. As one can imagine, a fully autonomous vehicle is not a simple task and is a highly complex integration of software, hardware, and cloud services. Apollo seeks to make this task more manageable, by providing the fundamental requirements needed for a functional system as well as addressing several non-functional requirements to maintain safety of passengers as well as other drivers on the road.

At its core all autonomous vehicles have a similar architecture, integrating key components such as perception, planning, localization and control. Apollo aims for the most accurate perception possible, integrating a wide variety of different sensors and software to determine the orientation of its surroundings. Planning systems enable autonomous vehicles to predict and adapt to surrounding traffic to keep safe and comfortable conditions for passengers. An important aspect of autonomous vehicles is ensuring the vehicle understands where it is located in space. This is done via localization, which combines sensor measurements as well as GPS for an accurate understanding. And, of course, all of these components are useless without any control of the vehicle. The control module must provide accurate reactions as well as lowest possible latency in order to create a safe vehicle to begin with.

Derivation Process

Throughout the previous weeks, our group has collected and extracted as much as possible about the high level structure of the Apollo software. This was not a very easy task. The derivation step was quite a big barrier in our process to deriving a logical conceptual architecture, especially when we are faced with folders upon folders of files and documentation. On top of this, many pages of provided readings were essentially required under the project's description to learn about architectural knowledge and autonomous driving cars. After the initial wave of feeling overwhelmed, the architecture started to seem clearer and clearer as opened browser tabs began to narrow down and condense.

To ensure an accurate conceptual architecture for Apollo was derived, our group started by reading up on the provided information regarding documenting software architecture[3] and self-driving car architecture[2]. With this information under our belt, it was time to tackle the folder monster, also known as the Apollo GitHub[1]. As time went on, certain folders seemed more relevant and important to the architecture than others. This allowed us to dive deeper into these folders to discover certain components of the system. Reading descriptions of folders in the 'modules' folder as well as making connections to the provided representations of the architecture allowed us to narrow down the main components. With the knowledge of what these components do and how they communicate, we were able to draft a conceptual architecture component diagram and make necessary refinements.

With the information gathered we decided on the main styles for the architecture. Client-server style (Client being the car), Layered style (Cloud, Software, and Hardware layers), and a small pipe-and-filter between some components in the software layer. Now, all that was left was to document the Non-functional requirements (NFRs) about Apollo. Unfortunately, NFRs have yet to be explicitly stated in any Apollo documentation, so we created our own based on requirements we figured were suitable for the system. With the basics of our component diagram done, all that was left was to use the information we had discovered and finish our report and presentation.

Non-Functional Requirements (NFRs)

Given the derivation of Apollo's autonomous driving system, non-functional requirements are discovered to satisfy the different needs of stakeholders involved in the system. These requirements are considered for the synthesis of the conceptual architecture (that will be discussed in the next section).

Reliability

The hardware and software components should not be easily damaged while driving. Bug reports and hardware failures should also be reported to the server promptly.

Availability

Simulation servers should always be online as navigation and cruise control depends on the simulation component heavily. On the contrary, HD map servers should be online only when a map update is issued as the map is supposedly downloaded on the system beforehand and allow the on-board mapping engine to plan routes.

Portability

The operating system should cover a wide range of car types and should not be difficult to implement.

Scalability

The cloud servers should also be able to handle peak periods of simulating and downloading map updates concurrently with the cars' navigation, whilst maintaining an active and online status when the car is at rest.

Performance

Time required for the hardware to communicate with the server and cloud-based, as well as simulations for the driving environment should be kept at a minimum interval to reduce delays.

Modifiability

Variety of sensors can be added/updated and the pipeline can be extended easily. Map and simulation software can also be updated with no change to the cars software or hardware in real time.

Security

The car's GPS location should not be tracked by third party applications and should be kept confidential between the driver and the server manager.

Conceptual Architecture

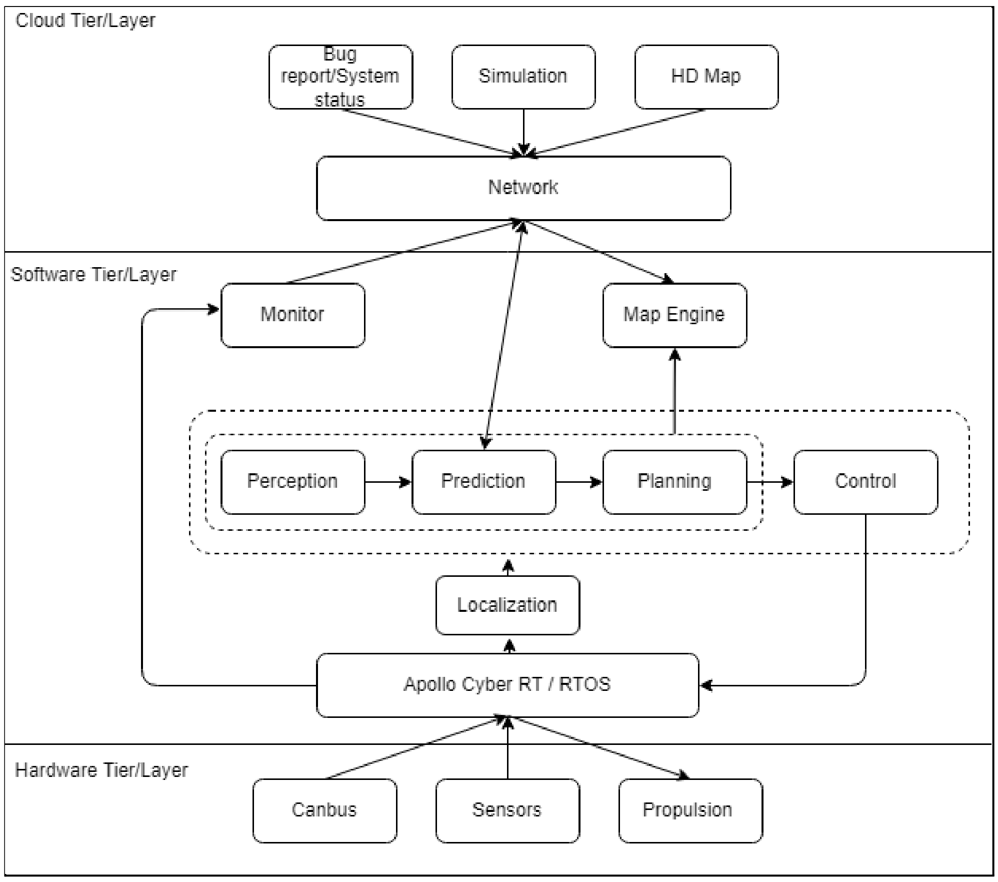

Figure 1. Box-and-arrow diagram of conceptual architecture

Layered Style

The primarily observed architecture style within the Apollo system is the Layered style. This can be easily seen in our diagram of the conceptual architecture (Fig. 1). As depicted in the diagram, there is the Cloud Tier, the Software Tier, and lastly the Hardware Tier. This depiction was determined from the documentation graphic for Apollo 7.0. Each tier has its own form of communication with the adjacent. For the cloud to software communications there is the network aspect, whereas for the software to hardware communications there is the Apollo Cyber RT, Apollo's Real Time Operating System (RTOS). These two systems mark the separation between the layers, essentially acting as the borders themselves.

Starting with the cloud tier, its main functionality reflects the client-server aspect of the architecture described in the next section. Just below that, is the software tier. This showcases the heart of the Apollo system as well as an autonomous car system, or any computer system for that matter. The major components of this tier, in charge of the bulk of the computation, used to perceive the car's environment and take action accordingly, create a pipe-and-filter which will be explored later in further detail. Lastly, at the bottom of the stack, we have the hardware tier. If the software tier was the heart, the hardware tier represents the nervous system and musculoskeletal system. This tier is responsible for supplying the software with all the necessary sensory data as well as controlling the car based on the instructions received. All of this allows for fast and efficient communication between layers.

We found this architectural style fitting to the Apollo system for a few reasons. The first is that it clearly distinguishes the different functionalities of the system. This allows easy distinction between what each component's functionality is inside of its layer. For example, we know that each component in the hardware layer will be directly correlated with certain physical actions of the car or its environment. This style also allows reusability for different implementations of the same layer in different autonomous car designs. Lastly, due to the layered style nature, improvements in the hardware or software layers will instinctively improve the other.

This style overall benefits the functionality of the system, as well as the readability of its architecture.

Client-Server Style

Found under Apollo's layered architectural style, the cloud tier depicts a client-server style architecture. As the diagram for the conceptual architecture suggests, the cloud tier of components governs the interflow of data from software to hardware tiers as the cloud tier serves as a communication medium between the car's software navigation subsystem and hardware monitoring subsystem. Hence, it would be fitting to observe the client-server architectural style as the system distributes and processes data across a range of software and hardware components.

In this use case, components consist of the client, which is the car's software, and the server, which is the Cloud Service Platform offered by the Apollo Open Platform. Both components are connected via the network. To dive deeper into the server components, it can be broken down into the three following subcomponents: a system status reporter, a simulation component, and an HD mapping component, all of which communicate to the car's software monitor and map engine.

The benefits of this architectural style can be observed by the satisfaction of non-functional requirements regarding the functionality of the autonomous vehicle. Starting with reliability, the hardware components are evaluated concurrently as the Cloud Service Platform refreshes its system status. In case of a bug or hardware failure with the sensors or Canbus module, the issue is detected and reported through the network to the server. It is also crucial that the capacity of the Cloud Service Platform can handle immediate responses and feedback from map and simulation software, which the system clearly takes advantage of the scalability of the Cloud Service Platform as it can be updated with little to no cost.

However, the system displays its constraints and liabilities when adapting the client-server style. Mainly, this is due to the network performance being the deciding factor for how well the software and hardware tiers will communicate with the server. Should there be an influx of users simulating or downloading map updates, as well as attempting to establish a connection with the server via limited reception, it will prove to be challenging even for an up-and-ready system. Another concern would be with the robustness of the Cloud Service Platform on whether it is susceptible to interference or failure, which may delay communication between the software and the server.

Nevertheless, considering that if the map engine only requests for simulation and is updated periodically, the concurrent process of interacting with server-side and client-side components can be maintained and benefit from the centralised administration of software updates and hardware monitoring.

Pipe-and-filter Style

Upon exploring the open software platform, the public repository, and the open documentation of the current version and the previous versions (which were available online), our group was able to find that Apollo software has an underlying and evident pipe-and-filter architecture. This architecture encases numerous components, but it can be boiled down to 4 major components: Perception, Prediction, Planning, and Control. These stated components play a vital role in how the automated car's sensors and cameras react with its surroundings as well as provide its real time operating system (RTOS) and other important systems of the car with processed information for the course of actions (like how our eyes take the information through them and then different parts of our brain process this information to formulate meaningful data). Such an architecture allows the enormous flow of data from one segment to another through "pipes"

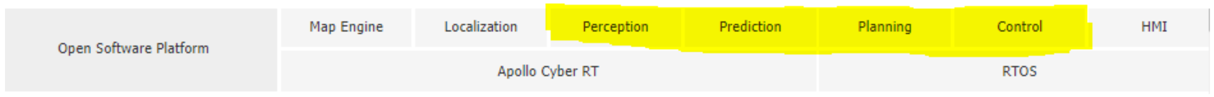

Figure 2. Diagram of Apollo 7.0 Open Software Platform

All these components / "filters" have huge blocks (sub-components, or smaller filter) of code under them, with each providing specific operational benefit :

(1) <u>Perception</u>: It helps in identifying different perception tasks such as classification, detection, segmentation, and learning convolutional neural networks.

(2) Prediction: It helps the system to study different ways to predict how other vehicles or pedestrians might move in front of the car.

(3) Planning: It helps in identifying several different approaches which Apollo uses to develop trajectories for autonomous vehicles.

(4) Control: As the name suggests, it assists the RTOS for steering the car, throttle responses, and brakes to execute the planned trajectory.

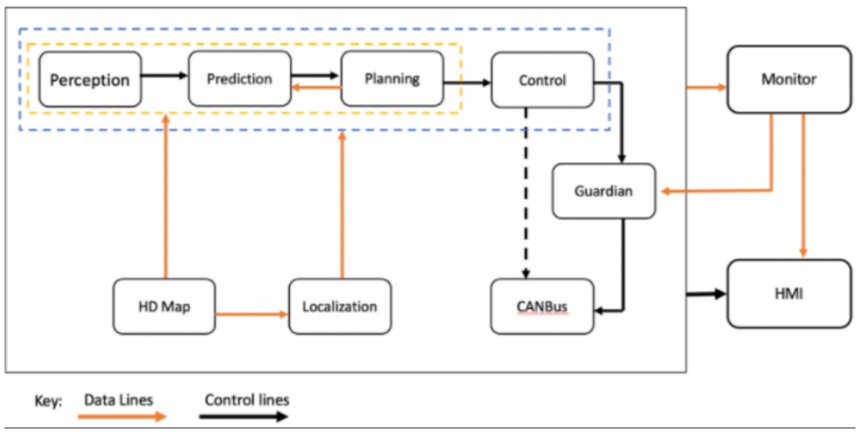

Figure 3. Software Overview

Each filter applies transformations to their input streams, does their computing incrementally so that output is available before all input is consumed, and feeds some other component/s of the architecture further ahead. This provides rapidity computationally. Hence, making the whole computational system quicker and more efficient.

The localization subcomponent is key for many of the other subcomponents to function correctly. The general responsibility of this component is to provide information to the perception, prediction and planning components about the vehicles' physical location. In Apollo's implementation, localization is provided using two methods. The first is using the Real Time Kinematic (RTK) based method, which utilises a combination of GPS and inertial measurement. The second method is a multi-sensor approach incorporating not only GPS and inertial measurements, but also LiDAR measurements. These methods aid the vehicle in perceiving its surroundings and understanding of where it is located relative to its surroundings.

Such an architecture provides many advantages like reusability and ease of maintainability. If we see all the previous releases of Apollo, there have been significant changes in the above-stated modules. As these modules/ components are the most crucial part of the whole car, they require constant upgrades and refinements in their supporting codes and structures so that the system can be made to be more precise, responsive, and updated computationally. Hence, one of the most useful advantages of the pipe-and-filter system is the ability to make enhancements to the existing system.

With the stated facts of the architecture, we can map all these advantages to a few non-functional requirements in one way or another. It provides a huge easy base for modification and makes it easy to implement the same kind of software structure across different cars. Sure, there is a sacrifice on performance as new information which is fed manually as well as ones that are derived by the neural networks and convolution system lead to an increase in the complexity but only to a debatable scale.

External Interfaces

Not much data is transmitted to/from the system as a great portion of the functionality manifests within the software and hardware layers of the car. From our findings, there are only three cases where this happens as a nature of the client-server architecture style: sending system information and bug reports, sending and receiving values to/from the cloud simulation function, and updating the systems map engine.

For sending system information and bug reports from the system to the cloud server, the monitor component is responsible for this functionality. As stated in the source code, the monitor module's responsibility is to check hardware status and monitor system health, as well as sending bug reports. This could be very useful in the event of an accident as the user could either ensure that everything was functional at the time of the accident, or document the possible point of failure that could have caused the accident, opening opportunities for further upgrades and modifications.

Unfortunately, scarce information was found regarding the simulation aspect of the system as the information is contained in the cloud level. We suspect that the prediction module communicates with the simulation module over a network connection to simulate certain situations in close to real-time behaviour.

Lastly, there is the map engine. This map engine is what we assume to be the location of the locally pre-downloaded map that the system accesses for planning ahead for variables and parameters, such as lane numbers, road styles, road bends, hazards, and many more. This map is believed to be updated over a network connection every few months to include support for more roads or cities.

Use Cases

Use case 1: Obstacles ahead and emergency braking

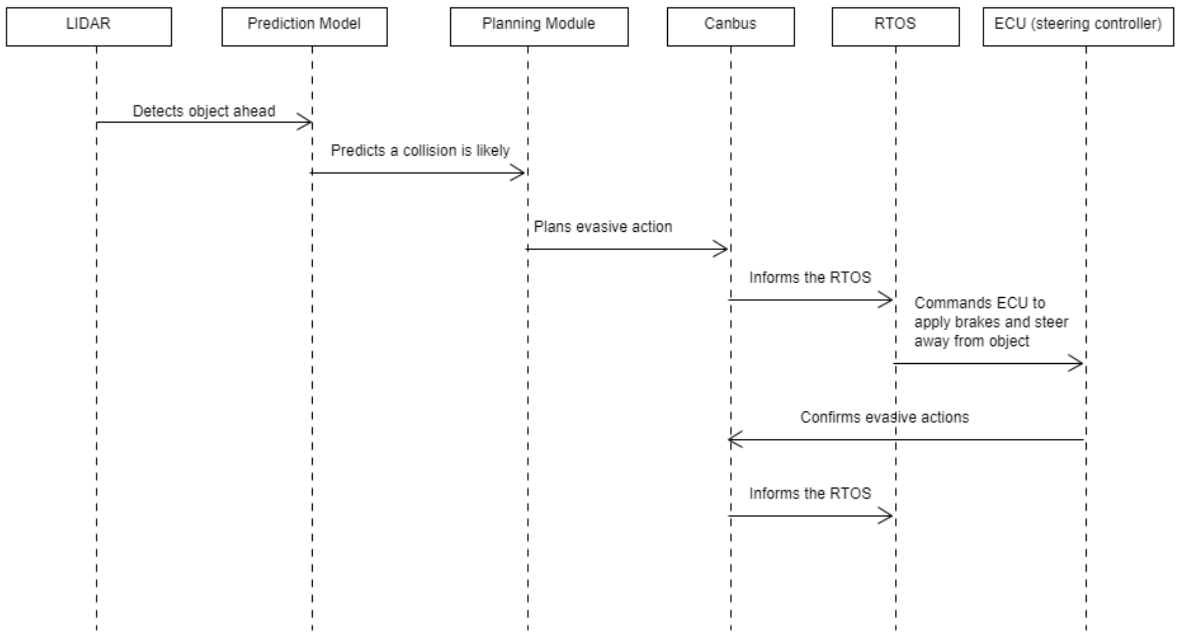

Figure 4. Sequence diagram of emergency manoeuvre use case

In this scenario, the car detects an obstacle ahead, and must make adjustments to its course in order to avoid a collision. The obstacle is first detected by the LIDAR system, which then updates all the known information used in the prediction model. Using the difference in relative velocities between the car and the object, the system predicts an imminent collision. This prompts the planning module to create a list of instructions the car must follow in order to avoid the collision, specifically an immediate decrease in speed and a change in steering. This information is sent to the RTOS through the Canbus, and the RTOS commands the ECU and its appropriate subsystems to follow the planning module's instructions. The car applies the braking and steering changes as needed, and these actions are relayed back to the RTOS through the Canbus.

Use case 2: Speed control on a highway

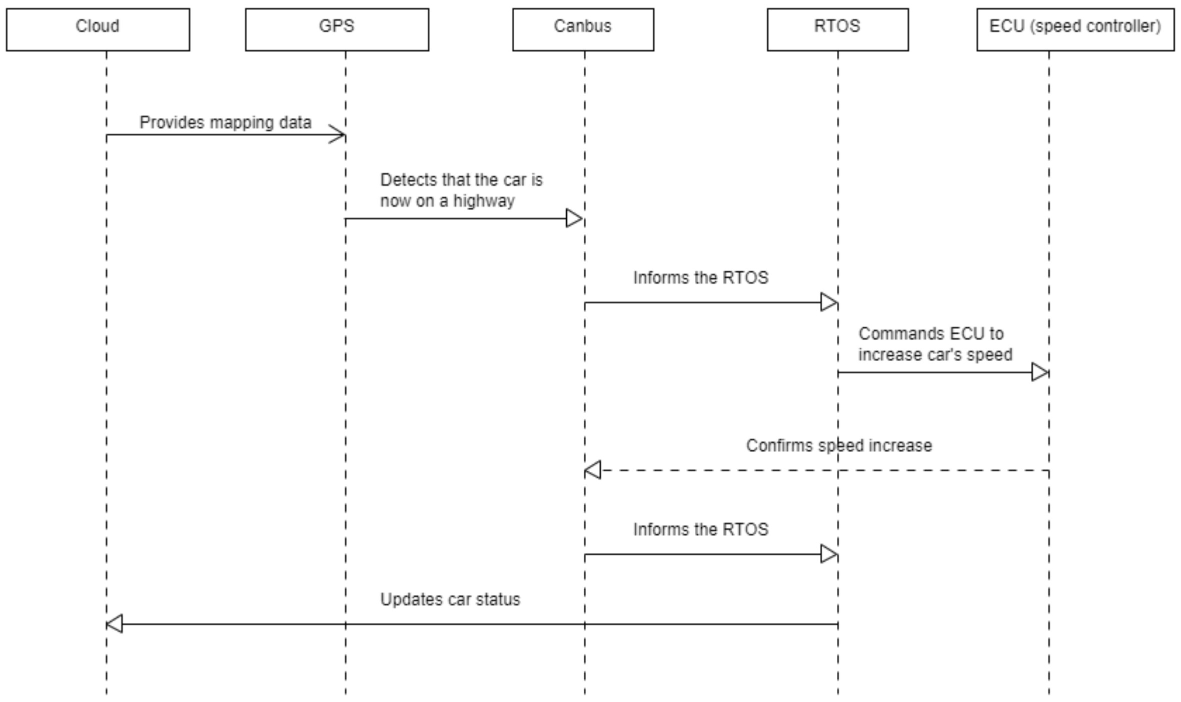

Figure 5. Sequence diagram of speed control use case

In this second use case scenario, the car drives onto an on-ramp leading to a freeway, which has a higher speed of traffic than the previous road. Using road information acquired from the cloud mapping data, the GPS detects that the car is now on a road with a higher speed limit than before. This is relayed to the RTOS through the Canbus system. Based on this information, the RTOS commands the car’s throttle controller to safely increase speed. Once this instruction has been executed, the ECU responds to the RTOS with the new throttle information, through the Canbus. Finally, the RTOS updates the car’s status stored in the cloud with the new speed and location information.

Lessons Learned

Over the course of analysing and documenting the conceptual architecture of Apollo, it was inevitable that challenges regarding information gathering and presenting the architecture to unfamiliar stakeholders.

To start with the challenge imposed by information gathering, it was evident that Apollo had updated its software seven-fold with changes to code and documentation. It was difficult to capture the conceptual architecture as the developers emphasised on the coding aspects and had not comprehensively disclosed details of the comparatively general Apollo Open Platform.

To tackle such an issue, we had to expand our attention to the functionality of other autonomous driving systems and understand Apollo's inner structure by referencing other systems. It is important to note that many autonomous systems share similar modules and components and hence, the scope of our investigation was enlarged.

We also discovered that not all stakeholders would be able to understand the functionality of the system, not to mention the inner subsystems and how the components would aid the user. Since the resources given were more suited for backend developers and system administrators, it was crucial that the conceptual architecture documented was straightforward and understandable by all stakeholders, including users and marketers alike.

As a result, this project has honed our skills in synthesising information by gathering abstract and lengthy documentations. This has allowed us to identify and elaborate on how each of the proposed architectures would benefit the system as well as the stakeholders' requirements.

Conclusions

In conclusion, the Apollo system embeds the three conceptual architecture styles--layered style, client-server style, and pipe-and-filter style. There is a close-knit dependency between components as information flows from one tier to the other in a concurrent manner. This has allowed the Apollo system to function seamlessly in realtime to ensure the safety of drivers and passengers, as well as other road users. Conceptually, Apollo's architecture breeds room for user feedback and further modifications as seen from each patch. The inner structure of software and hardware functionalities would be discussed in detail in the next deliverable.

Glossary

Apollo 7.0: The latest official version of Apollo software architecture which introduces Apollo Studio and enhances Apollo Perception and Prediction modulesApollo Perception and Prediction modules

Apollo Open Platform: Apollo’s coding platform which includes the Cloud Service, Open Software, Hardware Development, and Open Vehicle Certificate platforms

Canbus: A message-based protocol designed to allow the Electronic Control Units (ECUs) found in today’s automobiles, as well as other devices, to communicate with each other in a reliable, priority-driven fashion.[4]

Client-server style: An architecture of a computer network in which many clients (remote processors) request and receive service from a centralised server (host computer).[5]

Non-functional requirements (NFRs): System attributes that serve as constraints or restrictions on the design of the system across the different backlogs.[6]

Pipe-and-filter style: An architectural pattern which has independent entities called filters (components) which perform transformations on data and process the input they receive, and pipes, which serve as connectors for the stream of data being transformed, each connected to the next component in the pipeline.[7]

References

[1] ApolloAuto. (n.d.). ApolloAuto/apollo: An open autonomous driving platform. GitHub. Retrieved from https://github.com/ApolloAuto/apollo

[2] Behere, S., & Törngren, M. (2016). A functional reference architecture for autonomous driving. Information and Software Technology, 73, 136–150.

[3] Gorton, I. (2011). Documenting a software architecture. Essential Software Architecture, 117–128.

[4] Smith, G. M. (2021). What is Can bus (controller area network). Dewesoft. Retrieved from https://dewesoft.com/daq/what-is-can-bus#what-is-can-bus

[5] Encyclopædia Britannica, inc. (n.d.). Client-server architecture. Encyclopædia Britannica. Retrieved from https://www.britannica.com/technology/client-server-architecture

[6] Author Dean Leffingwell -, -, D. L., & Leffingwell, D. (2021, August 21). Nonfunctional requirements. Scaled Agile Framework. Retrieved from https://www.scaledagileframework.com/nonfunctional-requirements/#:~:text=Nonfunctional%20Requirements%20(NFRs)%20define%20system,system%20across%20the%20different%20backlogs.&text=They%20ensure%20the%20usability%20and%20effectiveness%20of%20the%20entire%20system.

[7] Hasan, S. (2019, May 26). Pipe and filter architecture. Medium. Retrieved from https://syedhasan010.medium.com/pipe-and-filter-architecture-bd7babdb908

--End of Report--