Assignment 3: Architectural Enhancement

April 10, 2022

Group 39

Name & Student Numbers:

Denise Micu (20061143)

Aidan Richards (20102555)

Max Beninger (05852667)

Leo Toueg (20062092)

Sean Guo (10157679)

Spencer Venable (20059095)

Abstract

The present report examines our proposed enhancement to Apollo's architecture: the addition of a remote summoning feature. After a brief discussion of the proposed summoning feature, we suggest two possible ways that such a feature could be implemented within Apollo's architecture. We then proceed to perform a SAAM architectural analysis, which involves assessing the impact of each implementation on various stakeholders and non-functional requirements. Once we have settled on a particular implementation, we go on to discuss the possible effects that its implementation will have the overall system, and present two relevant use cases. Finally, the report ends with a discussion of the possible risks of implementing this enhancement, and a brief discussion of the lessons learned.

1. Introduction

In our previous work, we examined the conceptual and concrete architecture of the Apollo self-driving platform. First, in A1, we proposed a preliminary conceptual architecture based on the available documentation on Apollo's GitHub page. Then, in A2, we compared our initial conceptual architecture against Apollo's actual codebase using reflexion analysis. Based on our findings across these two previous assignments, we arrived at the following high-level architecture for the Apollo system:

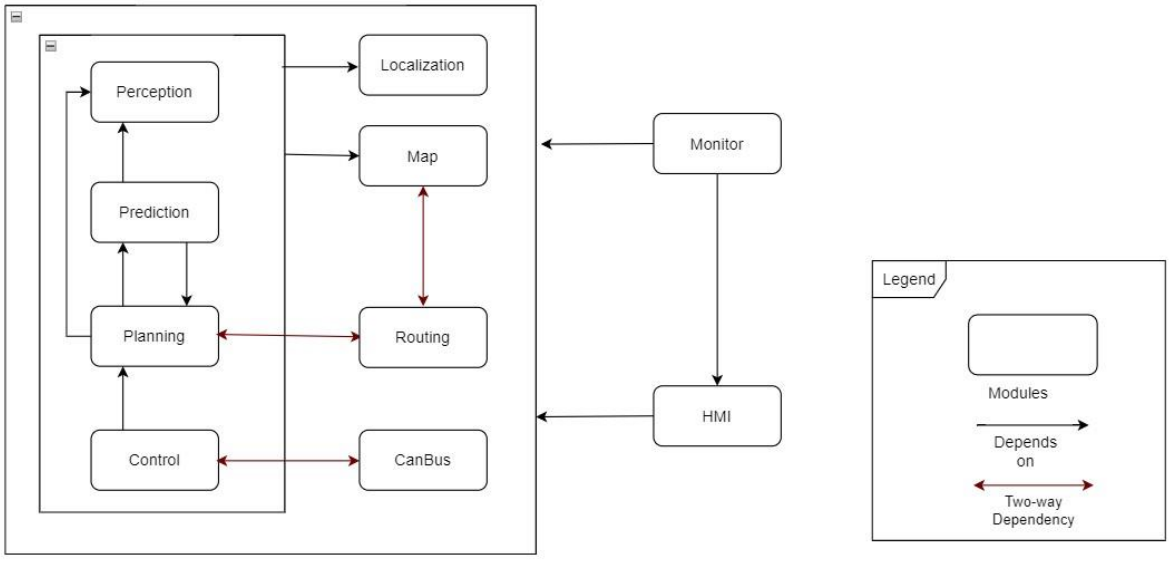

Figure 1. High-level architecture of the Apollo system

The aim of the current report, however, is to go beyond a purely descriptive analysis and to propose a *new* enhancement. Specifically, we propose to add a ***remote summoning*** feature to the current Apollo architecture. Summoning refers to the user's ability to remotely beckon the vehicle from a short distance away (e.g., across a parking lot). When summoned, the vehicle

should start up, and autonomously navigate to the user's current location – avoiding any obstacles along the way. It should also be possible for the user to cancel a previously initiated summoning request ("unsummoning"), thereby instructing the vehicle to return to its original position. Summoning is an existing technology that is already available in Tesla™ autonomous vehicles[1], but it has not yet been implemented in the Apollo platform. While Apollo does have some subcomponents for dealing with parking-lot stations – such as its "valet-parking" scenario handler – there is currently no way for a user to summon their vehicle *from a remote location*.

In terms of potential interactions, summoning component(s) will likely need to communicate with the parts of the system that are responsible for high-level planning and navigation. This includes modules such as Planning, Routing Localization, and HD Map. Such interactions make sense, since the summoning procedure will require the vehicle to locate itself in space – relative to the summoning user – and then plan a trajectory that safely guides the vehicle to the user's location. Otherwise, the system can function as it normally would in other situations. The exact details of how summoning interacts with other parts of the architecture, however, will depend on how summoning is implemented (see section 2).

The overall structure of our report is as follows. First, we suggest two possible ways of implementing the summoning feature within the existing Apollo architecture. Next, we perform a SAAM analysis in order to weigh the pros and cons of each approach. After making our final decision, we then proceed to discuss the effects that our chosen implementation will have on the rest of the system – including its impact on various other modules, as well as its impact on maintainability, testability, and performance. Finally, we present two use cases involving summoning, and further discuss potential risks that may be introduced by adding a summoning feature.

2. Possible Implementations of the Summoning Feature

There are multiple ways that a summoning feature can be implemented within the existing Apollo architecture. This section outlines *two* such possible implementations, which will later be compared using SAAM analysis.

Option 1: Summoning is distributed across existing modules

[1]https://www.tesla.com/ownersmanual/modely/is_is/GUID-6B9A1AEA-579C-400E-A7A6-E4916BCD5DED.html

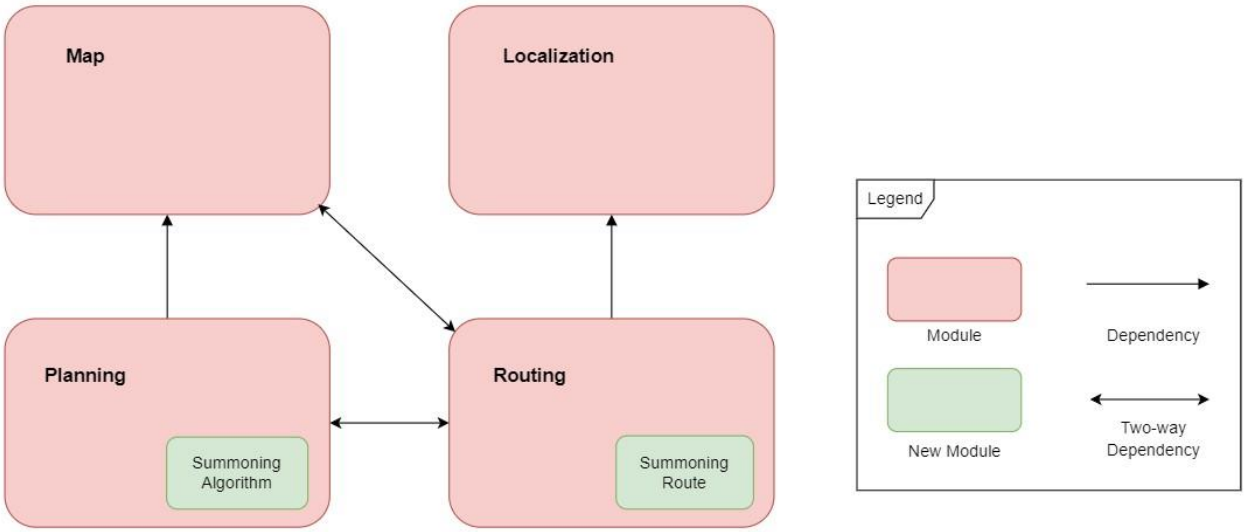

Figure 2. The implementation architecture of the enhancement

The above figure illustrates the first way of implementing the summoning feature: namely, summoning functionality may be embedded within the existing Planning and Routing modules. In this scenario, a special summoning algorithm will be implemented in the Planning module that specifically handles the planning of trajectory during summoning or unsummoning. The start and destination points are also sent to the routing module, which generates an optimal path from the vehicle's current location to the location of the user. Both modified Planning and Routing modules interact directly with the Map and Localization module, thus ensuring that the summoning procedure has access to the most up-to-date location information (which is crucial for successful navigation).

The benefit of this implementation is the new features are implemented directly into the existing modules without the need of introducing a new module (which could be costly and time consuming). Meanwhile, the downside of this design is potentially increased memory cost and run time due to new files and code inserted into the modules.

Option 2: Adding a dedicated Summoner module

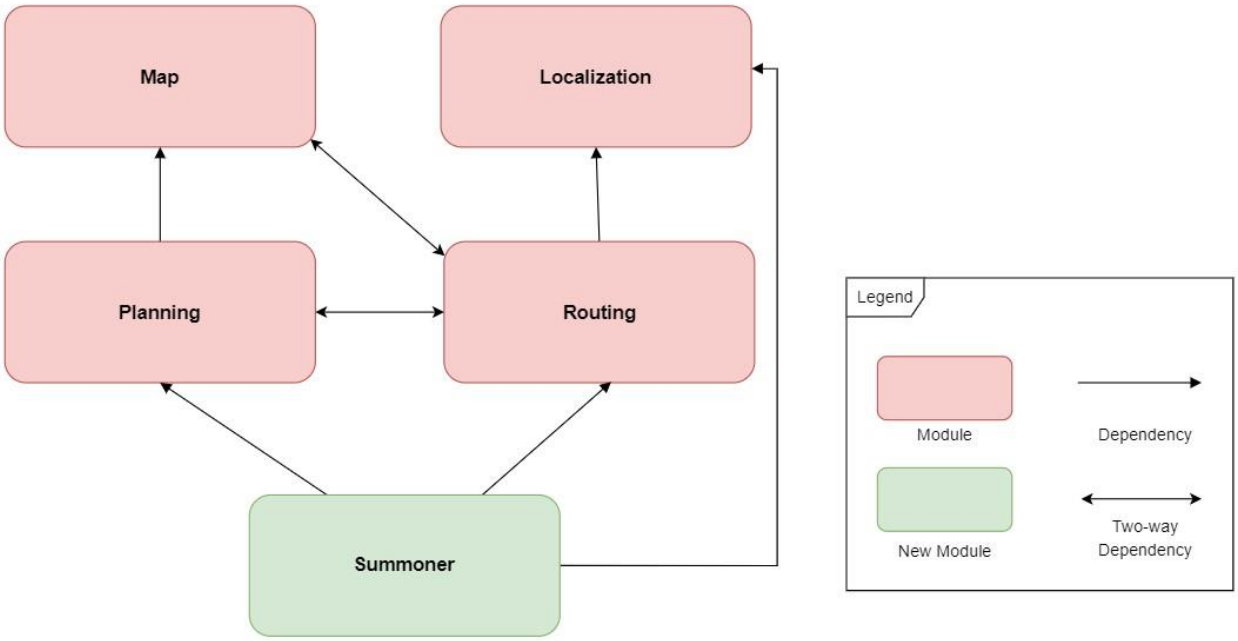

Figure 3. The alternative implementation architecture of the enhancement

As illustrated in Figure 2, an alternative way to implement summoning is to create a new dedicated Summoner module. In this case, the Summoner module would independently handle summoning requests, storing information about the summoning location, and calculating the optimal trajectory on its own. The Summoner module would only be activated upon a summoning/unsummoning request, and it would interact directly with the Planning and Routing modules.

A benefit of this implementation is that, if summoning is carried out by a dedicated Summoner module, there will be no additional processing demands placed on pre-existing modules like Planning and Routing. By contrast, a downside of adding a Summoner module is that it could be difficult to integrate with the current architecture, and it will most likely generate additional dependencies (either through method-calls or Pub-Sub traffic) that are not presently accounted for.

3. SAAM analysis

Having outlined the two possible implementations of the summoning feature, we now turn to consider how these two implementations will impact the relevant stakeholders. In order to do this, however, we first need to (i) identify the stakeholders in question and (ii) identify the Non-Functional Requirements (NFRs) that matter to them. For this summoning feature enhancement, the main stakeholders – and their associated NFRs – can be broken down as follows:

Developers

There is a wide range of developers working on the Apollo open-source software platform. The project started with developers working at the Baidu Research Institute, but – since being made open source in 2017 – it has grown to include developers from all over the world.

Non-Functional Requirements:

Development Speed - The summoning feature must be easy to implement such that it can be fully implemented within a 6-month period.

Implementability - The new feature must be possible to implement without major changes to the architecture of the system.

Project sponsors

Funding of the project is mainly limited to Baidu; however, there are a few other partners that contribute. This funding is required to pay developers, and to develop the physical aspects of the technology.

Non-Functional Requirements:

Development Cost - The new feature must be cheap to implement, costing no more than 10 million in additional development, testing, and project costs.

Software and Product Testers

The software testers are responsible for testing new features and ensuring the different modules work together correctly. Apollo supports the ability to test the car in a simulated environment to better test new features. The Product testers then test these changes in the real world on actual vehicles.

Non-Functional Requirements:

Testability - The new feature must be testable using Baidu's virtual testing simulation without requiring changes to the simulation.

Project Managers

The project managers at Baidu are responsible for different components of Apollo's development and ensuring the different components are developed in a logical and timely manner.

Non-Functional Requirements:

Development Speed - The summoning feature must be possible to implement concurrently with other development so as to not delay the release date of V8.

Customers

Baidu has partnered with several different automakers for the mass-production of vehicles augmented with certain aspects of Apollo's software. We will consider a future version of one of these models that is sold with the summoning feature.

Non-Functional Requirements:

Performance - The app must confirm that the car has received the summoning request within 2 seconds.
Performance - The map displaying the car's movements must update the car's position at least once every 5 seconds.

Now that we have identified the stakeholders and their NFRs, we can proceed to consider how the two proposed implementations of summoning – the "embedded approach" and the "external approach" – will impact these goals. First, for the developer, the main NFR is that the new feature be quick to implement (<6 months) and be implementable without major changes to the current architecture. Both requirements are most easily met by the embedded option. On the embedded approach, Summoning will use several of the modules which are already in place, including Routing, Planning, and Localization. These modules normally pass information between each other to do similar tasks, such as navigating through traffic lights and/or parking lots. Thus, it would be quick and easy to go with the embedded code option and would prevent unnecessary additions to the architecture.

For the project sponsors, the embedded option would also be preferable for financial reasons. Implementing an entirely new module – and performing the necessary integration testing for it – is likely to be a time-consuming and expensive process. Implementing summoning within an existing module (or modules), by contrast, would allow for some code and test reuse, thereby saving both time and money.

Baidu has developed an in-depth simulation to test the Apollo software in a simulated environment. This simulator allows the code to be tested in its entirety using realistic inputs and situations. Since this simulation is designed for Apollo's current architecture, adding a new module may require modifications to the simulation's interface to keep it compatible. For this reason, the embedded option would also be optimal for the software testers as they would only have to add a 'summoning scenario' to test the additional feature.

The last thing to consider is the customer's perspective. Adding a summoning module may slightly speed up summoning related tasks. The reason for this is each module operates independently, so having a dedicated module would mean that the summoning related computational tasks would be prioritized. This speed increase would be relatively small, however, as the summoning algorithm would have to wait for inputs from other modules.

Implementation Choice and Its Effect on NFR's

NFR	Effect (embedded implementation)	Effect (external implementation)
Development Speed	***Positive.*** Would speed up development by taking advantage of the module architecture already in place. Reuse of route planning and trajectory code, which is needed for summoning.	***Negative.*** Would require additional time to add new module and modify architecture.

Development Cost	***Positive.*** Lower cost due to less testing and implementation time.	***Negative.*** Additional code must be used to create the new module as it cannot take advantage of the structure currently in place for implementing additional planning scenarios.
Implementability	***Positive.*** Easier to implement because it aligns with current architecture.	***Negative.*** Requires major changes to the architecture by adding and connecting a new module.
Testability	***Positive.*** Less testing required and testing can make use of vehicle simulation software that is already developed.	***Negative.*** Greater amounts of testing required due to the new module (e.g., interface testing).
Performance	***Neutral or Negative.*** Possible bottlenecks or delays due to summoning code having to share resources with other tasks done by the module.	***Positive.*** Summoning related computations would have top priority, however, there would still be delays as the summoning module waits for input from other modules.

While the NFR's for the customer may be slightly easier to meet by having a dedicated summoning module, the embedded option is by far the best way to implement the summoning feature. Adding the additional module would result in significantly more work for the developers and testers, be extremely costly for the project sponsors, and likely result in project delays (due to the difficulty in designing and testing a new module). For the remainder of this report, then, we are opting to go with the first "embedded" implementation of the summoning feature.

4. Effects of the Enhancement

Having decided on a specific implementation of the summoning feature, we now summarize the possible effects that our implementation will have on the rest of the system. Overall, the effects should not be too drastic. Given the Pub-Sub architecture of Apollo – and the scenario-based architecture of the Planning module – it should be possible to add a summoning scenario (in Planning) without altering many of the other components.

Effects on the current architecture

- ***Planning*:** A summoning algorithm would be implemented in the planning module – as a new "Summoning scenario" – which handles the planning of trajectory during summoning. No new dependencies need to be added.
    - Summoning folder added in directory: apollo/modules/planning/scenarios
- ***Routing*:** The routing module receives the start and destination points, along with the 'summon' tag. It then calculates the optimal path for the car to travel to the location of

the user. Additionally, Routing now has to keep track of possible changes in the user's location. This might result in a new dependency from Routing to HD Map.
    - Potential changes to file: apollo/modules/routing/routing.cc
- ***Map***: The HD Map will now have to represent the User's location (outside the vehicle), as well as the location of the vehicle itself. Presumably this will be linked to an app on the user's phone.
    - Potential changes to file: apollo/modules/map/hdmap/hdmap.cc
- ***Localization*:** From an architectural point of view, this module wouldn't be impacted, but it would receive requests from a new sub-module, without "knowing" it (seeing it as a normal Planning or Routing request).

Effects on maintainability

Maintenance of the system is made slightly more difficult by the addition of summoning, as there will be new submodule – i.e., the summoning scenario in Planning – which may have dependencies on other pre-existing components. However, the summoning feature is not expected to drastically affect the complexity (or, by extension, the maintainability) of the architecture.

Effects on Testability

Testing the system is made more difficult due to the nature of the enhancement. With the new addition, a complete test of the system will now need to include vehicle summoning simulations of some degree. These are difficult to thoroughly test as there is a large degree of variability in the different situations the vehicle will encounter when being summoned. Things like determining if a summon location is viable at all, finding the most optimal route to the summon location, or determining where is best to park for the user pickup will all need in depth testing.

Effects on performance

As mentioned above, it is possible that implementing the summoning features within the existing modules of Summoning and Routing might result in a higher processing load within these modules (which could hypothetically slow them down). This is probably not a large concern, however, as the Planning module has been designed to allow for dynamic switching between traffic scenarios. Moreover, summoning is not expected to be a highly-costly operation – it simply applies many of the system's existing capabilities (e.g., perception, obstacle detection, route planning) to a slightly new context.

5. Use Cases

Use Case Scenario 1: User summons the vehicle to his/her location

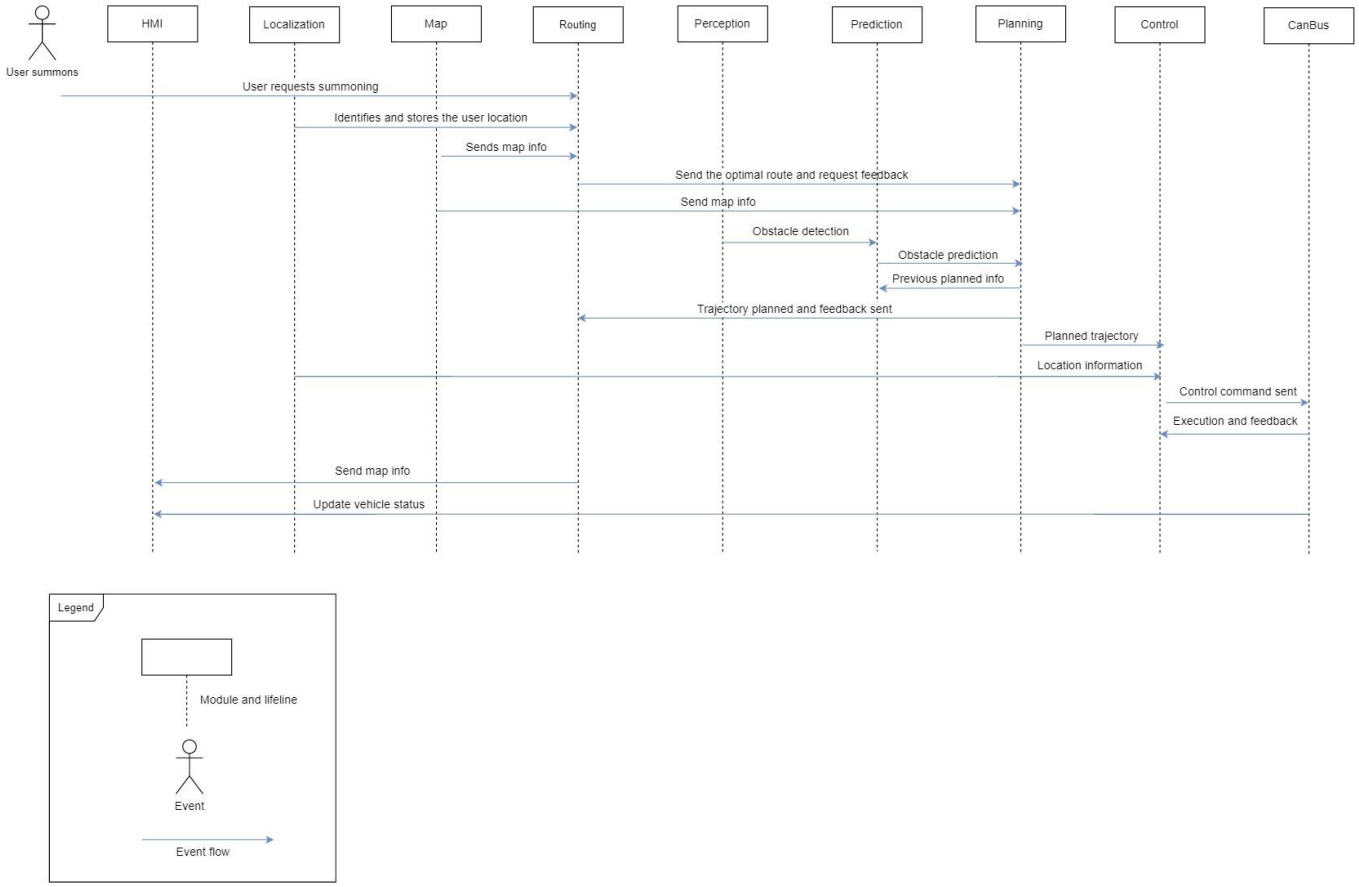

Figure 4. Sequence diagram for summoning use case

The above sequence diagram outlines the event flow in the case where a user summons the vehicle to their current position. The sequence begins with the summoning request: then, upon request, the Routing module generates an optimal path of travel based on the current location (and destination) of the vehicle. The Routing module then sends the optimal route to the Planning module for initiation, and requests feedback.

In order to ensure safe travel, the Perception module is continuously operational throughout the summoning process. Perception monitors the environment for potential obstacles and passes this information to Prediction – which handles the task of predicting the obstacles' expected trajectories. Finally, the predicted obstacle information is sent to the Planning module. Using all the information it receives (about location, obstacles, and route), the Planning module generates an optimal vehicle trajectory, and sends this trajectory to the control module (for translation into vehicle commands).

Next, the Control module passes the control commands to the CanBus module for hardware execution. The CanBus module sends the hardware commands to the car chassis (and also relays the chassis information back to Control). Finally, the current status of the vehicle –

including the map location – is sent to the HMI/Dreamview module for the user to see in real time.

Use Case Scenario 2: User cancels the summoning and instead unsummons

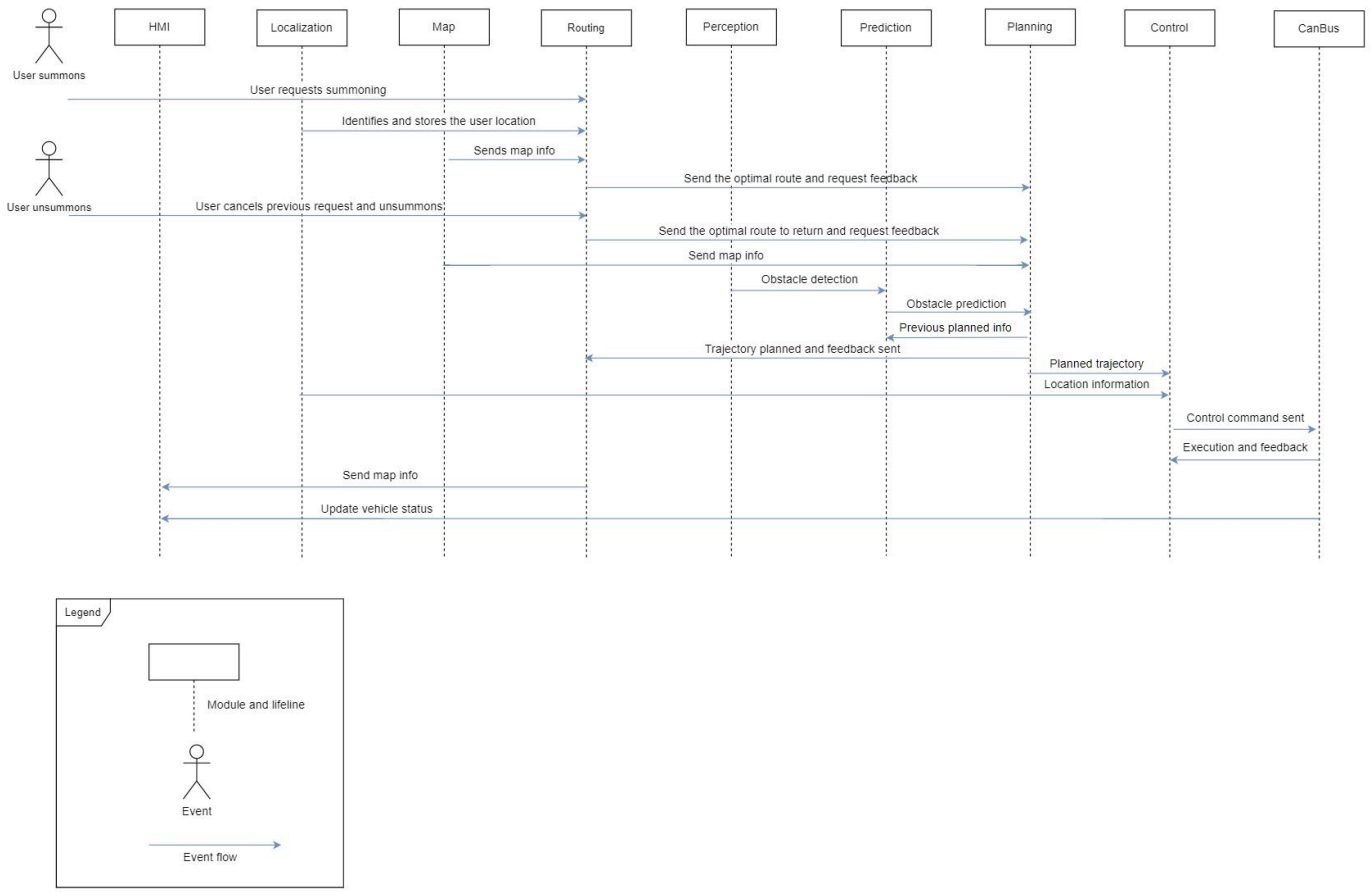

Figure 5. Sequence diagram to unsummon after summoning request sent

The new enhancement also supports requests to *unsummon* the vehicle – i.e., cancelling a previously made summoning request. The execution process here is very similar to that of the regular summoning request. As detailed in above sequence diagram, the user first sends the initial request to summon the vehicle to his/her location, which is received by the Routing module. The Routing module then uses location information to generate the optimal path for the vehicle to travel to the user. During this time, the user has the option to unsummon the vehicle and send it back to its original location. The unsummon request is received by the same Routing module – which calculates the best return path – and passes this information on to Planning. From this point on, the vehicle follows the same execution path as in the previous use case.

6. Plans for Testing

For the embedded implementation, there are two main parts to the embedded code. The first part is in the routing module and it is responsible for generating a summoning route to follow based on the phone GPS location from where the request was sent to the car's current position. The other component of the code is the summoning algorithm itself which is located in the planning module. This algorithm will act similarly to other 'scenarios' in the planning module such as stopping at a red light or changing lanes.

The first step in testing would be to individually test the summoning algorithm and summoning route generation code to ensure both work correctly when supplied with fake inputs. To perform system testing, the virtual testing environment that Baidu designed for Apollo would be used. This includes a virtual car that has an interface similar to a real Apollo car that can interface directly with the software. This testing would put the car in a variety of virtual situations to test the feature's code in a realistic environment. The simulation would go through various unexpected situations that might occur in the real world such as extreme weather, losing signal, and being summoned to an unreachable location. Lastly the feature would be tested in the real world along with the other upgrades that have been worked on for that release.

7. Potential Risks

There are potential risks that must be assessed before the implementation of the summoning feature. Two such risks pertain to: (1) security and (2) safety.

First, the summoning feature adds many potential security risks. While the vehicle is in transit between its location and the summoner's location it will be operating without anyone inside the vehicle. If the vehicle does not have a strong passenger authentication system, it could lead to someone other than the anticipated passenger getting in the vehicle and perhaps taking control of it or tampering with the car's hardware/software. A possible way to mitigate these potential issues is to implement a digital key which is associated with the summoner's mobile device, and the vehicle only unlocks its doors when said mobile device is within a certain proximity.

Second, the addition of a summoning feature also raises concerns about safety (both for pedestrians, and for the vehicle itself). In ordinary driving scenarios, the user is in the vehicle and can potentially override the system in cases of emergency. Since the user is not in the vehicle during the summoning procedure, however, it will be imperative to ensure that the vehicle properly avoids obstacles – especially in unpredictable environments like parking lots. Indeed, Tesla's Smart Summon feature has occasionally resulted in vehicle damage.[2]

[2] https://kmph.com/news/local/mans-tesla-crashes-into-pole-using-smart-summon-valet-feature

8. Conclusion and Lessons Learned

In this report we have explored the possibility of adding a summoning feature to the Apollo platform. We began by presenting two alternative ways to implement the summoning feature: (i) implementing summoning within existing modules (the “embedded approach”), or (ii) implementing summoning as a separate module altogether (the “external” approach). We then performed a SAAM analysis to adjudicate between these alternatives – ultimately, deciding that the embedded approach was preferrable. Next, we examined proposed changes would affect the system, including the impact on maintainability, evolvability, testability and performance. Finally, we provided two relevant use cases, and discussed potential risks that might arise when implementing a summoning feature.

This exercise has taught us a lot about how to approach the task of updating and enhancing a software system. One clear lesson is that proposed enhancements should be sensitive to the current architecture of the system. In the case of Apollo, for instance, the scenario-based architecture of the Planning module has been designed to allow for the easy addition of new traffic scenarios. Thus, the obvious way of implementing a summoning feature is to make use of this template and add summoning as a new “scenario” within Planning. Another important lesson – gleaned from the SAAM analysis – is that enhancements should be guided by the needs of the stakeholders. At the end of the day, it is the stakeholders’ opinions that matter, so every decision about enhancements should consider the potential effects on the Non-Functional Requirements that the stakeholders care about.

References

Apollo. (n.d.). Retrieved February 16, 2022, from https://apollo.auto/index.html

ApolloAuto. (n.d.). Apollo/modules at master · ApolloAuto/apollo. GitHub. Retrieved February 16, 2022, from https://github.com/ApolloAuto/apollo/tree/master/modules

Guo, Q. (2020, April 29). Software system of autonomous vehicles: Architecture ... Software System of Autonomous Vehicles: Architecture, Network and OS. Retrieved February 21, 2022, from https://fisher.wharton.upenn.edu/wp-content/uploads/2020/09/Thesis_NovaQiaochuGuo.pdf

Model Y owner's Manual. Tesla. (n.d.). Retrieved April 9, 2022, from https://www.tesla.com/ownersmanual/modely/is_is/GUID-6B9A1AEA-579C-400E-A7A6-E4916BCD5DED.html

Writer, S. (2021, August 18). *Ai-packed 'robocar' puts Baidu in the self-driving fast lane*. Nikkei Asia. Retrieved April 9, 2022, from https://asia.nikkei.com/Business/China-tech/AI-packed-robocar-puts-Baidu-in-the-self-driving-fast-lane2#:~:text=BEIJING%20%2D%2D%20The%20vehicle%20has,to%20develop%20autonomous%2Ddriving%20technology

Wikimedia Foundation. (2022, January 19). *Apolong*. Wikipedia. Retrieved April 9, 2022, from https://en.wikipedia.org/wiki/Apolong