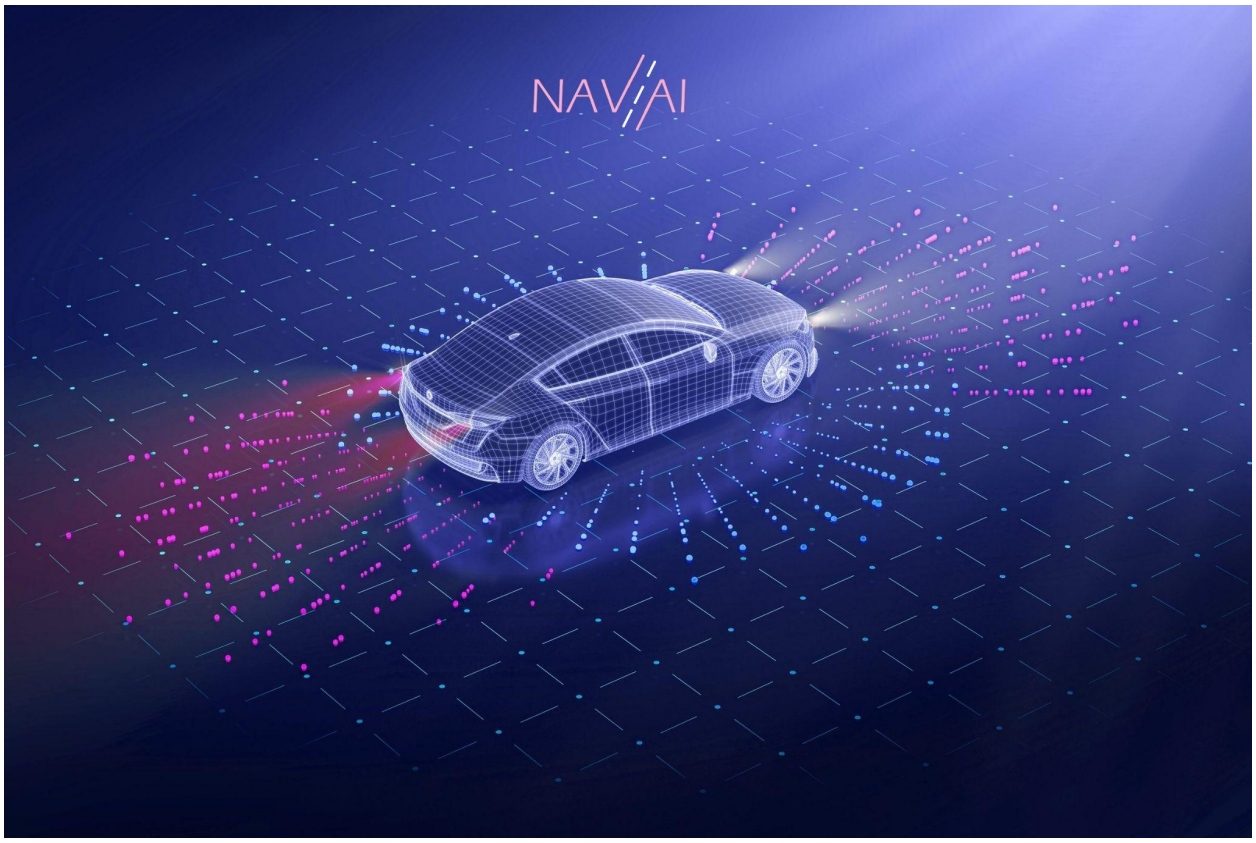
NavAI™



"Sophisticated, Self-Driving Software
Hands-free navigating experience"

Michael Midthassel, Mark Merson, Nyrah Balabanian, and Max Brizzio

CS-347 Spring 2023

Section 1: Introduction

The objective is to implement a software with IoT into a reliable and readily available self-driving vehicle of the future. The development of the software will go into an embedded system with an emphasis on its real-time availability. Special features that this vehicle will highlight are dynamic cruise control, safety & leisure and satellite GPS navigation.

Development Process approach and new releases

For the scope of this project, it is recommended to incrementally release updates and new features over time. This will effectively be accomplished with an iterative process model. An iterative development will effectively lay the foundation for the software. After the first release, it is expected that there will be a shift to an evolutionary approach. Gradually more features mostly in the leisure and safety category will be added in future releases. The shift in approaches will increase efficiency between new releases and ensure they are released in a timely manner.

Understanding the Problem

To understand the problem we are given to develop a self-driving vehicle, the stakeholders are first and foremost identified. The stakeholders for this software is Alset who will utilize this technology to implement in their self-driving vehicles. Next data, functions, and features are discussed in order to break down the problem to smaller parts. Some of these features include motion sensors, GPS and navigation, typical leisure, battery and motor, bandwidth and connection of the vehicle itself. To further categorize these features to efficiently approach the problem, we plan to separate features developed into outside of the vehicle sensors, inside the vehicle functions, Cruise control, and electric motor operation.

Testing

Once we tackle each of these categories it is vital to test that the software works. An analysis model can be made by testing each feature to see under what conditions it will work under. Furthermore, we can also test where the vehicle can drive and its efficiency in different driving conditions. Due to the nature of our objective, it is imperative that the systems implemented work seamlessly together. To achieve this, we need a system that is available in all sorts of conditions and situations. It's also important to note that because of the goal we are reaching for, safety is extremely important. To keep safety in mind, it's necessary to have a system that can be reliable by accommodating said conditions.

Documentation

For an ambitious project like this one, it is necessary for us to keep a clear outline of every task done. Whether it is research, brainstorming, or commits, the entire team needs to be on the same page when it comes to the bigger picture. While delegating tasks is the best way to get through this project in a timely fashion, each task completed must be recognized by each team member.

Our Team

The team leading this project is Mike, Mark, Nyrah, and Max. We are computer science and cybersecurity students at Stevens Institute of technology and consist of different skills. Our team is detail oriented, task-oriented, people-oriented, writing skilled, curious, patient, and coding skilled.

Mike - Languages: C/C++, Java, Python, Swift, Javascript, Scheme - CS courses taken: 115, 135, 284 - Qualifications: project management, detail-oriented, analysis	Mark - Languages: C/C++, Java, Python, Scheme - CS courses taken: 115, 135, 284, 334, 382, 385, 392, 396, 496 - Qualifications: Detail-Oriented, Patience, Coding
Nyrah - Languages: C, C++, C#, Python, Java - CS courses taken: 101, 115 - Qualifications: Communication, TaskOriented, People-Oriented	Max - Languages: Python, Java, C/C++, Scheme, Swift - CS courses taken: 115, 284, 135, 334, 306, 385, 382 - Qualifications: Problem Solving, Writing Skill, Curiosity

Section 2: System Architecture

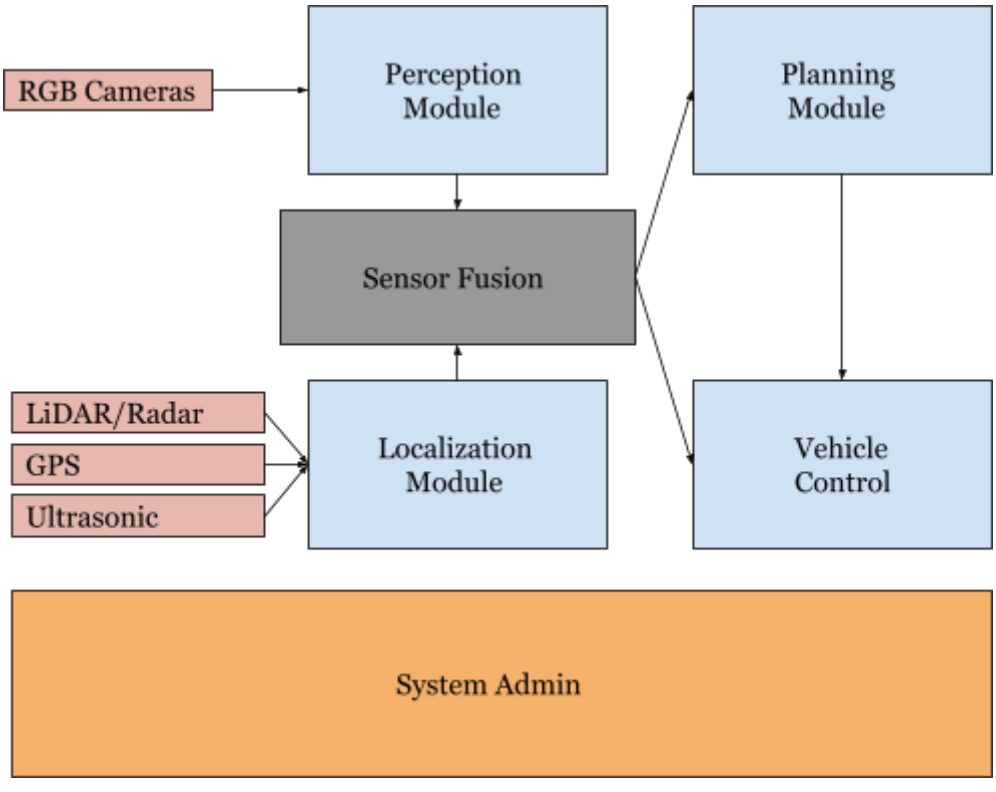
Overview

The design of the software system for an autonomous vehicle is of the highest significance since it lays the foundation for the system as a whole to function properly and securely. A well-designed architecture is required to ensure that the many different components of the software can work together reliably and without friction. Because of the potential threat that autonomous vehicles pose to the safety of passengers as well as other people who use the road, this is of the utmost importance.

Excellent design makes scalability and system maintenance simpler, which is necessary as autonomous driving technology evolves and improves over time. The performance of the software is another essential part of the design of an autonomous vehicle. This is because the software must analyze massive amounts of data in real time and make decisions in split seconds. Performance should be tuned so that a well-designed software system for a self-driving vehicle may work smoothly and effectively even in demanding driving circumstances. This will allow the system to run smoothly and effectively.

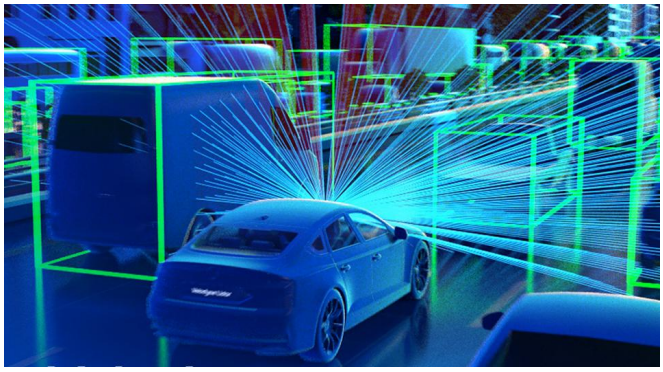
The architecture of a software system for an autonomous vehicle is a crucial part of the design and development of the system. It is also necessary for ensuring the performance, scalability, and performance of the system.

Identification of Functional Interfaces



Sensory Input Systems

LiDAR



LiDAR is fantastic at interpreting 3D information which is why it is practically always used in some form with self-driving vehicles. Some drawbacks could be that colors on the road won't be seen (which is solved with RGB cameras). LiDAR sensors emit pulses of laser beams and gather information about whatever is around the vehicle. Since multiple lasers are emitted per second, an accurate map can be drawn of the surroundings.

Radar

LiDAR and RADAR form the primary set of sensors that provide imaging, detection, ranging, tracking, and sensing of the drive location [5]. Radar sensing relies on radio waves to detect an object's velocity, angle and distance. Radio waves can be emitted in long ranges and are great when utilized for adaptive cruise control.

GPS

In order to have our self-driving vehicle work well in today's streets, it's imperative that we include some kind of navigation system for it to understand where it is, where it's going, and how to get there. An obvious choice is GPS which can be used to pinpoint where the vehicle is at any given moment. The vehicle needs a way to process its environment using the many sensors so after feature maps are generated by fusion systems, the vehicle will be able to create maps of its surrounding environment.

After creation of maps, the vehicle has a sense of spatial awareness but now it needs to go somewhere. This is where our path planning system comes into play. Using software that interprets output from our fusion system, the vehicle will use important pieces of data in its sensory information to plan where it will need to go in a safe, efficient manner that protects other drivers on the road and pedestrians alike.

Ultrasonic

Ultrasonic sensing is an essential part of autonomous vehicles for objects in close proximity and in areas with low visibility. This type of sensor will also allow the vehicle to gather information to see where it is. Ultrasonic sensors send out high-frequency sound waves to gauge the proximity of objects [3]. These types of sensors are already implemented in many vehicles and are especially useful for assisted parking. Although ultrasonic sensors have been implemented in vehicles already, its full potential has not been fully harnessed to apply it to self-driving vehicles. Crowdsourcing the data from millions of drivers can be the key to IoT automotive. IoT automotive can enable new features such as finding parking spots in urban areas or road hazards.

RGB Cameras

The advantages of RGB cameras are that we actually get colors on the road hence RGB which is imperative to drive in today's world with stop signs, traffic lights, and construction warnings. All of these use color to convey either the importance, or the instructions each street sign needs to convey which is why our system needs to accommodate it. A massive drawback to RGB-only cameras is that they have no context to analyze 3D data. We as humans have built context for our world since we could see but computers can have a very tough time with spatial awareness. Another obvious drawback is that of weather patterns. Rain, snow, lightning, anything that changes visibility can cause issues in the system.

Localization Module

The goal of the localization module is to use 3d data to understand the distances between objects and the subject vehicle. In order to get this usable 3d data, we take images from our LiDAR, radar, and sometimes ultrasonic sensors (ultrasonic is mainly used in self-parking situations but may also be utilized when navigating tight streets). After analyzing key fragments of these images that point out extrusions, we can see that some of these extrusions are objects of interest e.g. other vehicles, pedestrians, signs, traffic lights. While the localization module does not actually have the ability to read signs or traffic lights, these objects being highlighted in the output makes giving context to extrusions very easy using the perception module.

Perception Module

The primary function of the perception module is to contextualize the environment around the vehicle to assist the localization module in building maps of the vehicle's surroundings. This is done by collecting data from various RGB cameras on board positioned in different locations. The Perception module will work to combine the data

from each camera into a cohesive map that can draw details in from color to give context to 3d data fetched by our localization module which will draw distances to the objects detected.

Planning Module

After sensing and perception stages are complete, an output is fed into our planning module. This output, being the feature vector generated by our fusion system, needs to be used in order to make decisions for the vehicle. Our planning module will have three different layers in order to make the safest decisions on the road. These layers include mission planning, behavioral planning, and local planning [7].

Mission level planning is responsible for making the highest-level decisions such as planning route to destination (using search algorithms such as BFS or Dijkstra's), computing optimal trajectories and purely makes decisions based off of pre-existing conditions such as traffic law i.e. no real-time analyzation of surrounding vehicles or pedestrians. Behavioral planning directly makes decisions based on real-time conditions such as weather variation. Finally, local planning is meant to scan the vehicle's surroundings in order to avoid obstacles [6]. With these three components, a motion plan is finally fed into vehicle control with the objective of outputting the safest, most-efficient route.

System Administration

As self-driving vehicles become more common, it's important to ensure that they operate reliably and safely. One way to do this is by collecting and analyzing logs from each module in the vehicle's architecture. System administrators play a critical role in this process, as they are responsible for setting up and managing the logging infrastructure.

Logging is the process of recording events and activities that occur within a system. In the context of a self-driving vehicle, logging can help to identify issues with individual components and the overall system. For example, if the vehicle's sensors are not detecting objects accurately, this could be reflected in the logs from the sensor module.

To collect logs from each module in the architecture, the system administrator will need to configure each component to output logs in a standard format. This could be done using a logging framework or library, which can provide a consistent way of generating logs across different programming languages and platforms. The administrator will also need to specify where the logs should be stored and how long they should be retained.

Once the logging infrastructure is set up, the system administrator can begin monitoring the logs to ensure that everything is performing to standards. This involves reviewing the logs on a regular basis and looking for patterns or anomalies that could indicate a problem. For example, if there is a sudden increase in the number of errors being logged from a particular module, this could be a sign that something is wrong.

In addition to monitoring logs for issues, the system administrator can also use them to optimize performance. For example, if a particular module is generating a large number of logs, this could be a sign that it is being overworked and may need to be optimized or scaled up.

To make the most of logging data, it's important to have a robust analytics platform that can process and analyze the logs in real-time. This could involve using machine learning algorithms to identify patterns and anomalies that might be difficult to detect manually. The system administrator may also need to work closely with data analysts and other stakeholders to ensure that the logging data is being used effectively.

In a self-driving vehicle, the system administrator serves as the interface between the driver and the vehicle, acting as the Human Machine Interface (HMI). This is because the system administrator is responsible for configuring and managing the software and hardware systems that control the vehicle's autonomous functions. As such, the system administrator must ensure that the vehicle is able to communicate effectively with the driver and respond appropriately to their input. This involves configuring the HMI to display information in a clear and concise manner, and to provide feedback to the driver in real-time. The system administrator also plays a critical role in ensuring that the vehicle's autonomous functions are safe and reliable, and that they meet the required performance standards.

Fault management is an important aspect of ensuring the reliability and safety of self-driving vehicles, and it is the responsibility of the system administrator to manage and resolve any faults that may arise. This involves monitoring the vehicle's various systems and components, including the sensors, software, and hardware, to detect any faults or malfunctions. The system administrator must then diagnose the root cause of the fault and take appropriate actions to resolve it, which may include repairing or replacing faulty components, updating software, or reconfiguring the vehicle's settings. In addition to resolving faults, the system administrator must also work to prevent future faults from occurring by implementing effective monitoring and maintenance strategies, such as regular software updates, hardware inspections, and system tests. By taking a proactive approach to fault management, the system administrator can help to

ensure that self-driving vehicles operate safely and reliably, even in challenging conditions.

Feature Fusion and Interprocess Communication

Sensor Fusion

Now that we know how to utilize the advantages of both LiDAR and RGB, we need a method to integrate the sensors in a way that preserves the key pieces of information. It turns out that brilliant engineers at Google have thought of such a system that could work very efficiently for this purpose. It is the Multi-Modal Fusion Transformer (shorthand TransFuser). The TransFuser works by processing data from each sensor individually through a convolutional neural network (cnn) to be molded into feature maps. These feature maps are generated by the cnns ability to make out distinct shapes in images which we can dictate to our liking. Such shapes could be that of traffic lights, octagonal stop signs, triangular yield signs, vehicles, pedestrians and much more. After highlighting these shapes in various resolutions, the TransFuser outputs one final feature vector which combines feature maps of both sensors multiple times at different resolutions [2]. This feature vector is then fed to our navigational system which from there can process the maps to make decisions such as vehicle control.

Vehicle Control

Overview

The integration of several technologies, including computer vision, radar, lidar, and GPS, is required for vehicle control in our autonomous vehicle that utilizes sensory information to make judgments in real-time. Our vehicle's ability to observe its surroundings and make judgments about how to safely traverse them depends mostly on its sensors. The sensors provide the vehicle information about its surroundings, including the location of other vehicles, people on foot, and barriers, as well as details on the road's conditions and traffic flow [8]. The vehicle's internal computer uses this information to make judgments in real time on how to steer, brake, and accelerate the vehicle. In order to interpret the sensory data and produce the orders that govern the movement of the vehicle, the computer employs advanced algorithms.

For instance, the computer swiftly examines the data and determine whether to slow down or come to a full stop in order to prevent an accident if the vehicle's sensors identify a person crossing the road in front of it. Similar to this, the vehicle may change its speed and steering angle to safely negotiate a turn if it senses that it is approaching a curve or a crossroads. The vehicle's control system must also take into consideration a

number of variables, including traffic laws, road signs, and other drivers' actions. Moreover, it must be able to alter its control inputs in response to changing road conditions, such as wet or slick roads.

Control Elements

The following two elements make up the core of an autonomous vehicle's whole control system:

- *Longitudinal Control:*
  Taking into account the longitudinal dynamics of the ego vehicle, this component regulates the longitudinal motion of the ego vehicle. The throttle and brake inputs to the ego vehicle, which affect its motion (velocity, acceleration, jerk, and higher derivatives) in the longitudinal direction, are the controllable variables in this scenario.

- *Lateral Control:*
  Taking into account the lateral dynamics of the ego vehicle, this component regulates the lateral motion of the vehicle. The ego vehicle's steering input, which controls its steering angle and heading, is the controllable variable in this scenario. Keep in mind that heading and steering angle are two distinct concepts. [8] Whereas heading is concerned with the orientation of the ego vehicle, steering angle specifies the orientation of the steerable wheels and hence the direction of motion of the ego vehicle.

Control Schemes

Autonomous vehicles use a range of control strategies to effectively and securely traverse the highways. Rule-based control, behavior-based control, and model-based control are the three basic kinds of these control systems that may be generally defined.

Programming the vehicle to adhere to a set of established rules and restrictions is known as rule-based control. For instance, the vehicle may be set up to always respect the speed limit, stop for pedestrians at crosswalks, and keep a safe distance from other vehicles. While rule-based control is very easy to deploy, it could not be flexible enough to accommodate unforeseen occurrences or changing road conditions.

On the other hand, behavior-based control entails instructing the vehicle to act a specific way depending on how it perceives the surroundings. This strategy enables the vehicle

to respond to novel circumstances and modify its behavior as necessary. For instance, if the vehicle notices an obstruction on the road, it can slow down and try to steer around it. Rule-based control is more rigid than behavior-based control, while behavior-based control takes more intricate programming and may be more difficult to test and debug.

Model-based control makes predictions about the behavior of the vehicle in various circumstances using mathematical models. The physics rules and the physical characteristics of the vehicle serve as the basis for the model. Using this strategy, the vehicle may adjust its performance depending on a number of variables, including speed, safety, and fuel economy. Model-based control uses very complex algorithms and can offer fine control over the vehicle's motions, but it also needs precise models.

In order to travel the roadways safely and effectively, our vehicles employ a mix of these control strategies. For instance, our vehicle uses rule-based control to adhere to traffic regulations and behavior-based control to respond to unforeseen circumstances like construction or road closures.

Section 3: Requirements

3.1: Vehicle Control Functional Requirements

3.1.1 - Self Parking

3.1.1.1 - Set self-Park mode

Pre-Condition:
- Vehicle is able to recognize an opening in a valid area to park
- Vehicle is in a 0.10 mile radius of the destination
- Vehicle is able to measure the distance between the two vehicles and see if it can fit

Post-Condition:
- Self park button light is on and vehicle begins aligning itself

Requirements:
- Retrieves data through crowdsourced data to determine if it is a valid place to park the vehicle
- Identifies that the vehicle is in a .1 radius of the destination using its current location and comparing its distance from the destination

- Identifies if the vehicle need to parallel or perpendicular park based off LiDAR data the vehicle collects to determine how the vehicle can be parallel to the vehicles surrounding the open spot

3.1.1.2 - Lining up to park

Pre-Condition:
- The vehicle is in self park mode

Post-Condition:
- Vehicle stores velocity, angle, and distance to turn/back into space

Requirements
- LiDAR scans area of a potential parking spot and stores information about angle, velocity, and distance to get into the space
- Vehicle moves to a position where it is able to turn to get into the space

3.1.1.3 - Parallel parking

Pre-Condition:
- Vehicle identified vehicles are parallel to the curb
- Vehicle measured and stored data from the LiDAR about its angle, velocity, and distance to back into the space

Post-Condition:
- Vehicle is parallel to the curb and 3-5 in away from it.
- Vehicle is put into park

Requirements
- Vehicle is lined up and starts reversing and cutting into the space
- Vehicle recognizes the time which the direction of the vehicle needs to be cut into

3.1.1.4 - Perpendicular parking

Pre-Condition:
- Vehicle identified that vehicles are perpendicular to the curb

Post-Condition:
- Vehicle is parallel to the surrounding vehicles
- Vehicle is in line with the lines of the parking space
- Vehicle is put into park

Requirements:
- Vehicle is lined up and makes a wide enough turn into the space
- Checks to see if it is aligned with vehicles and parking lines
- Reverses out readjusts angle if it is not aligned

3.1.2 - Autonomous Driving

3.1.2.1 - Cruise Control Activation

Pre-Condition:

- Vehicle is in a driving mode that supports cruise control activation
- Vehicle speed is within the acceptable range for activation
- Driver is seated in the driver's seat and is ready to activate

Post-Condition:

- Cruise control system is activated and maintains the vehicle's speed
- Vehicle's speed can be adjusted using the control interface
- Vehicle can disengage the cruise control system at any time

Requirements:

- Receives input from the driver
- Receives input from System Admin
- Localization Module detects the distance between the vehicle and the vehicle in front of it and adjust speed accordingly

3.1.2.2 - Accelerate to speed limit

Precondition:

- Vehicle is on and currently in drive
- Vehicle is going slower than speed limit
- Path is not blocked by obstacles

Postcondition:

- Vehicle is no longer accelerating

Requirements:

- Sensor Fusion sends obstacle data to planning module
- Localization Module calculates MPH/KMH using sensor data and pulls speed limit data from maps API
- System Admin correctly identifies when the vehicle is under the current speed limit
- System Admins ends signals to vehicle control to adjust vehicle speed until speed limit reached

3.1.2.3 - Apply Brakes when obstacle detected within 1000 m

Precondition:

- Vehicle is on and currently in motion
- Vehicle is moving faster than the speed limit
- Obstacle detected in path closer than 1000 m

Postcondition:

- Vehicle is currently at or below speed limit
- Vehicle is no longer braking

Requirements:

- Sensor Fusion detects when obstacle appears directly in front of vehicle's current direction within 1000 m
- System Admin is able to calculate necessary braking distance given vehicle's current speed, distance to obstacle, and road conditions
    - Localization Module able to measure distance to object
    - Localization Module able to calculate current speed
    - Perception module able to detect different road conditions (snow, icy, wet, dry, dirt)
- Vehicle control adjusts vehicle speed according to braking distance

3.1.2.4 - Lane Control and Merging

Pre-Condition:

- Vehicle is in motion
- Vehicle is operating in a multi-lane highway environment

Post-Condition:

- Vehicle successfully navigates between lanes and merges with other vehicles on the road
- Lane control system is disengaged once the lane change is complete

Requirements:

- Localization Module must be able to detect and recognize lane markings and other vehicles in adjacent lanes
- Must merge safely into the desired lane and disengage once the lane change is complete

3.1.3 - Driver Control Override

Pre-Condition:

- Vehicle is on and is in motion
- Driver is about to override

Post-Condition:

- Driver has control of the vehicle
- Driver control system is deactivated, and vehicle stays at a constant pace

Requirements:

- Must be able to detect when the driver takes control of the vehicle

3.2: Planning Module

3.2.1 Update destination arrival/distance

Precondition:

- Current location has changed
- Surrounding traffic on route has changed

Postcondition:

- Estimated time of arrival is updated
- Distance to arrival is updated

Requirements:

- Retrieve information of the current location of the vehicle
- Compare it with the gps route and the fastest way to get to the destination versus the current status of the ETA and distance
- Recognize the difference and store the new values of the ETA and distance from destination

3.2.2 Get Current Location

Precondition:

- Vehicle is turned on and is able to connect with the GPS system

Postcondition:

- Current location of the vehicle is inputted into the GPS system
- Location ticker is on the map system on the dashboard

Requirements:

- Vehicle connects to GPS and pin-points the longitude and latitude of where the vehicle is
- Stores this information and inputs it into a real time map

3.2.3 Get Next Move

Precondition:

- Vehicle has a stored list of instructions for where to go next

Postcondition:

- Current step is executed and moves on to the next stored instruction

Requirements:

- Vehicle is connected to the GPS and has the most updated current location
- Current location is linked with the directions to analyze where in the steps the vehicle is

3.2.4 - Refresh Feature Map

Pre-Condition:

- Vehicle has been using a feature map that needs to be updated
- Vehicle is connected to a network or another external device that can provide a feature map

Post-Condition:

- Feature map is refreshed and ready for use by the vehicle

Requirements:

- Data from the perception module must be able to be read
- Localization module uses data from perception module to maintain an accurate feature map

3.3: System Admin Module

3.3.1 - Run Diagnostics Check

Pre-Condition:

- Vehicle is on
- Vehicle is not performing system update

Post-Condition:

- All faults risen in hardware have been detected
- Notification is sent to HUD

Requirements:

- System Admin has access to check all components of functional architecture
- Faults detected must be able to be resolved by System Admin Module

3.3.2 - HUD (Heads up Display)

3.3.2.1 - Start HUD

Pre-Condition:

- Vehicle is started
- HUD is not already on

Post-Condition:

- HUD is ready for driver interaction
- Vehicle information is displayed
- HUD Navigation Menu is displayed

Requirements:

- System Admin requests vehicle location information from planning module
- HUD will automatically turn on when vehicle is turned on

3.3.2.2 - Map Mode

Pre-Condition:

- HUD is already launched and is not in Map Mode
- Location Information is available from Planning Module

Post-Condition:

- Map is displayed on HUD
- Driver is able to interact with display

Requirements:

- Driver is able to select Map Mode from HUD Navigational Menu
- Vehicle Location information is fetched from Planning module
- Maps API is interacted with to display vehicle location embedded in geographical map

3.3.2.3 - System View Mode

Pre-Condition:

- HUD is already launched and not in System View mode
- Vehicle information is available from System Admin

Post-Condition:

- Vehicle information is displayed on HUD:
    - Speed, Charge %...

Requirements:

- System Admin is able to fetch speed and charge% and displays it to the HUD
- Driver is able to perform diagnostic checks
- Engineer is able to sign in to system admin to launch system updates or get vehicle logs.
- Vehicle Information is refreshed every 10ms

3.3.3 - Get Logs

Pre-Condition:

- Vehicle's system have been generating logs
- Vehicle is on and functioning properly

Post-Condition:

- Logs are retrieved and available for analysis

Requirements:

- All localization sensors are working properly and activated
- Localization sensors

- Vehicle software must be able to have access to these logs and process them accordingly

3.3.4 - Perform System Update

Pre-Condition:
- Vehicle is active and functioning properly
- Vehicle is connected to a network or another external device that can provide the update

Post-Condition:
- System update is installed and functioning properly
- The vehicle's systems may need to be restarted or re-initialized

Requirements:
- System Admin must be able to:
  - Receive and install the update
  - Verify the authenticity and integrity of the update before installing it to ensure it cannot be compromised
- The system update needs must be compatible with the vehicle's modules and hardware
- The update process must be able to handle interruptions or eros during the installation and recovery from any failures

Section 4: Requirements Modeling

4.1: Use Case Scenarios

4.1.1 - Self-Parking Module

Pre-Condition:
- The vehicle is equipped with a self-parking module and the sensors are on.
- The vehicle is near a parking spot.

Post-Condition:
- The self-parking module is activated and the vehicle is parked safely in the designated spot.

Trigger:
- The driver activates the self-parking module.

- The self-parking module receives data from the sensors and detects the available parking spot.
- The system calculates the distance and angle required for the vehicle to park safely in the spot.
- The module sends a signal to the car to start the parking maneuver.
- The vehicle starts moving and the module adjusts the steering and speed to park the car safely in the spot.
- If there is an obstacle or the system detects an unsafe condition, it alerts the driver and cancels the parking process.
- Once the vehicle is parked, the module sends a signal to the driver indicating that the parking process is complete.

4.1.2 - Lane Control and Merging

Pre-Condition:

- The vehicle is equipped with lane Control and merging system and the sensors are on
- The vehicle is in a lane and there is a vehicle in front of the car

Post-Condition:

- Lane control and merging system is activated and the car merges safely into another lane.

Trigger:

- The driver activates the lane control and merging system

1. The lane control and merging system receives data from the sensors and determines the location of the car and other vehicles on the road.
2. The system calculates the distance and speed of the car in front and checks if it is safe to merge into the other lane.
3. If it is safe to merge, the system sends a signal to the car to change lanes.
4. The vehicle merges into the other lane and the system adjusts the speed to maintain a safe distance from other vehicles.
5. If there is an obstacle or the system detects an unsafe condition, it alerts the driver and cancels the merging process.

4.1.3 - Perform System Update

Pre-Condition:

- The vehicle is equipped with a system update feature and there is a new update available.

Post-Condition:

- The system update is successfully installed and the car operates with the updated software

Trigger:

- The driver initiates the system update process

1. The vehicle detects the availability of a new system update and notifies the driver.
2. The driver initiates the system update process and the vehicle connects to the internet to download the update.
3. The vehicle verifies the integrity of the downloaded update and prepares for installation.
4. The vehicle installs the update and reboots the system.
5. The vehicle verifies the successful installation of the update and notifies the driver.

4.1.4 - Update Destination

Pre-Condition:

- Vehicle is on
- If current route is planned, destination has not been reached
- New destination is valid address

Post-Condition:

- Navigation display will update route on HUD
- New destination notification pushed to HUD

Trigger:

- Driver inputs new destination into navigation display

1. System Admin checks if new destination is a valid address (able to be read by maps API)
    a. If it is not valid, push notification to driver hud and exit process
2. If current route is in progress
    a. Send signal to planning module to update current destination
    b. Push notification to driver hud that new destination is current route
3. If no route is in progress
    a. Send signal to planning module to set destination
    b. Push notification to driver hud that destination is set to route

4.1.5- Push Notification

Pre-Condition:

- Driver HUD is on and software is up to date
- System Admin receives request to push notification to driver hud
- HUD is not on DND (do not disturb)

Post-Condition:

- Notification displayed on HUD for 5 seconds or driver manually acknowledged

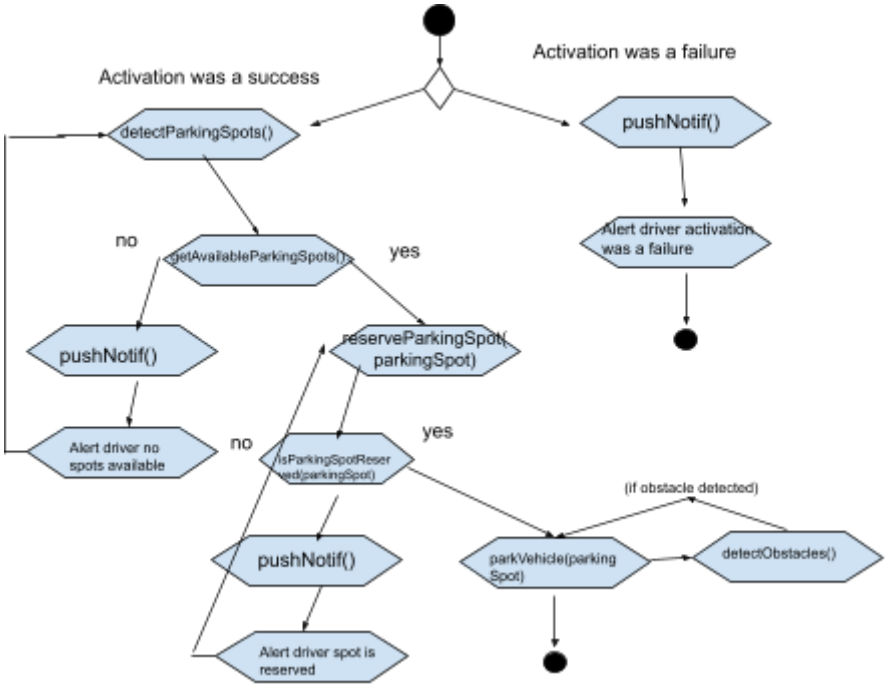
Trigger:

- System Admin receives signal to push notification given title and message

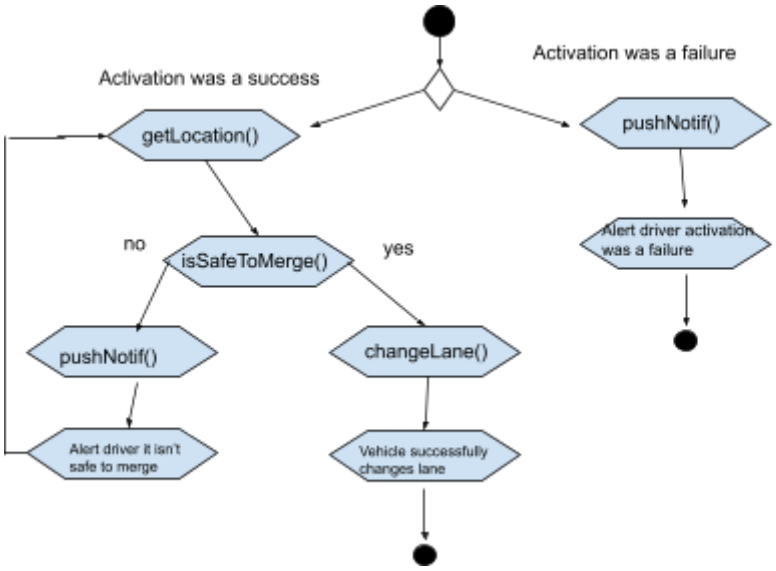
1. System Admin updates HUD with a push notification with given title and given message

4.2: Activity Diagrams

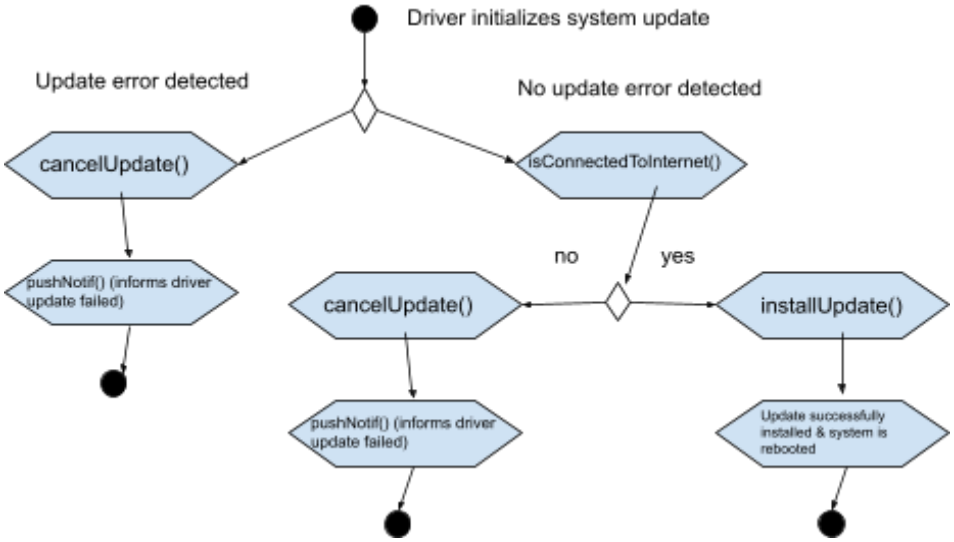
4.2.1 - Self-Parking Module



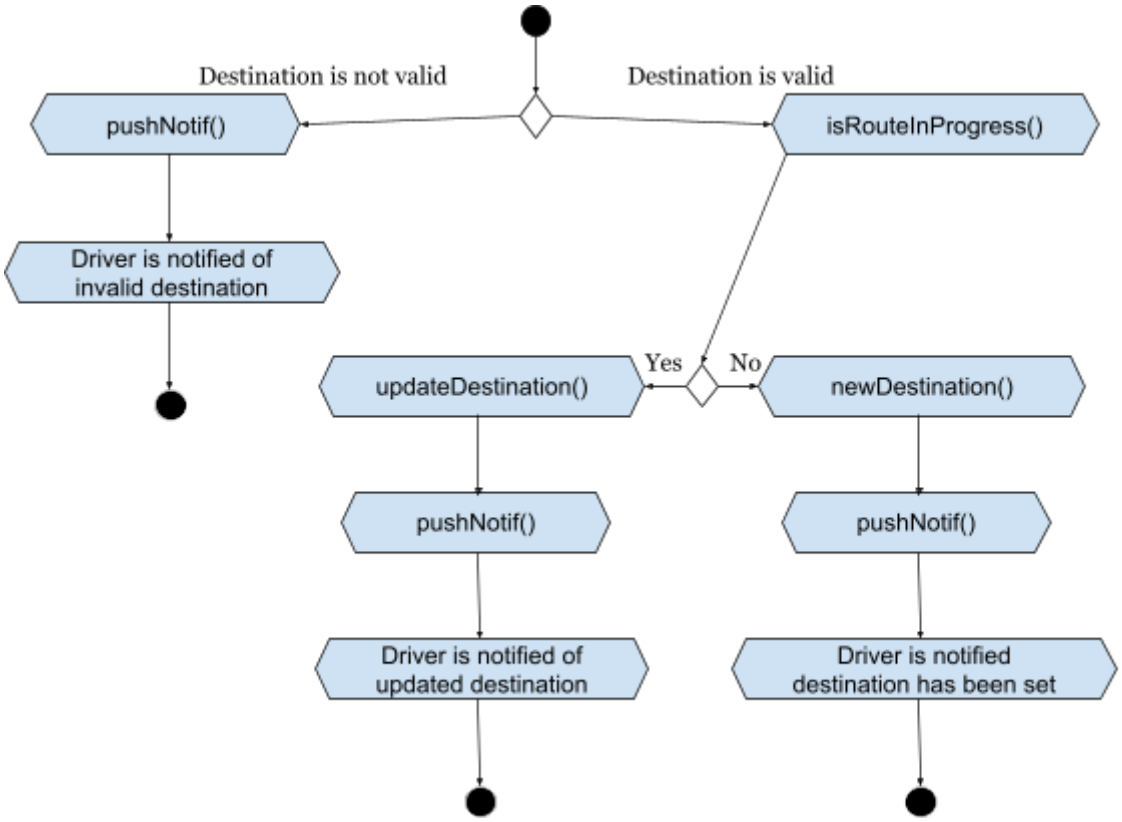
4.2.2 - Lane Control and Merging



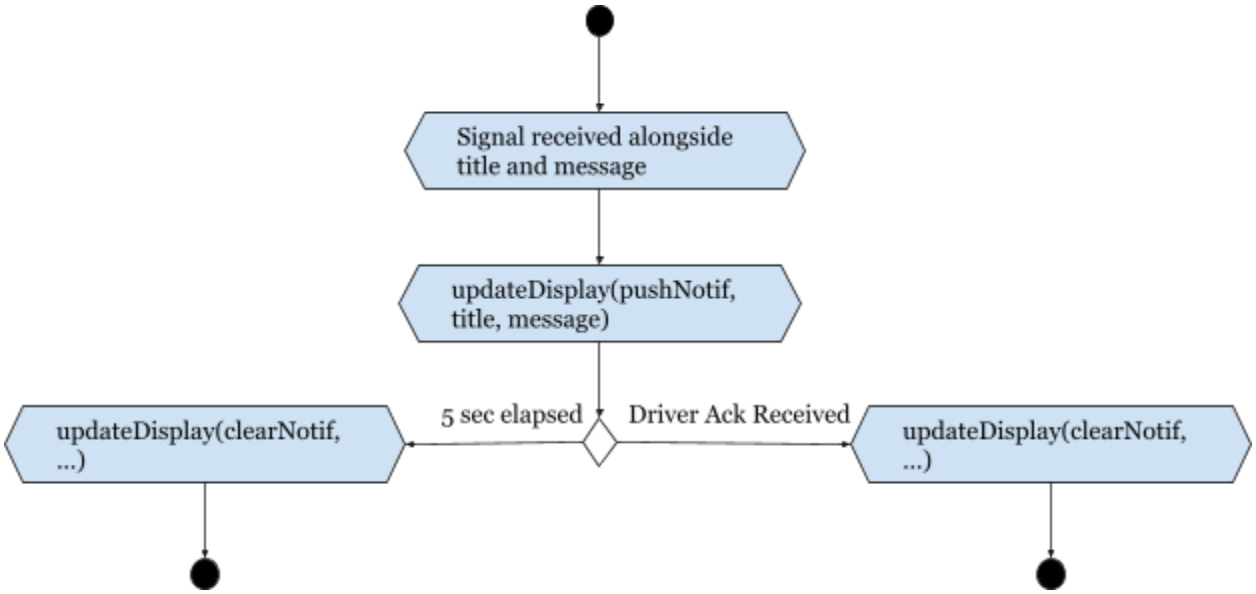
4.2.3 - Perform System Update



4.2.4 - Update Destination

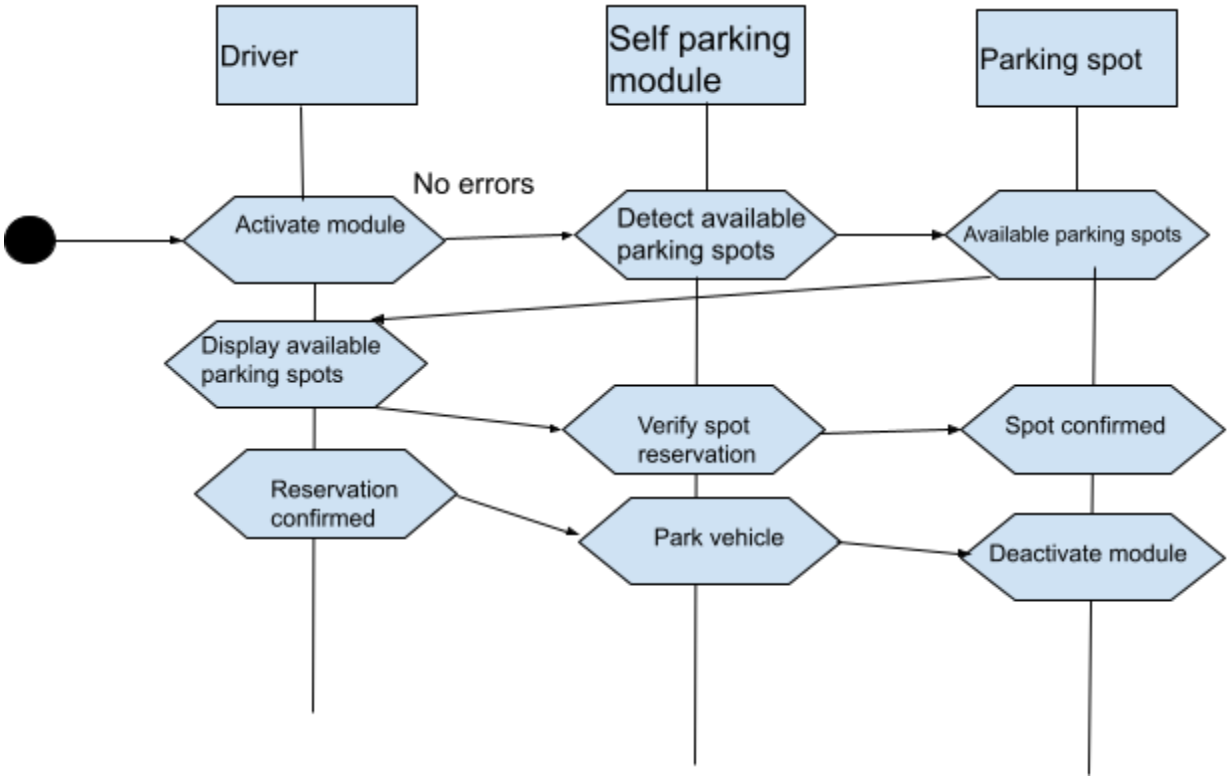


4.2.5 - Push Notification

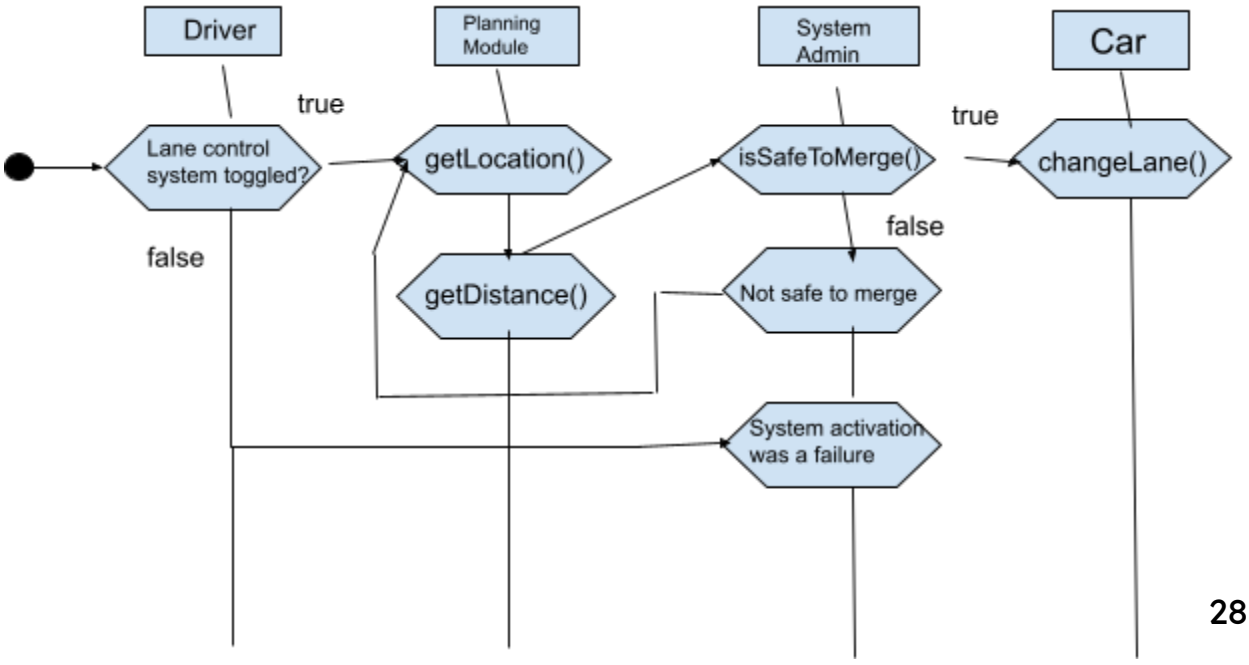


4.3: Sequence Diagrams

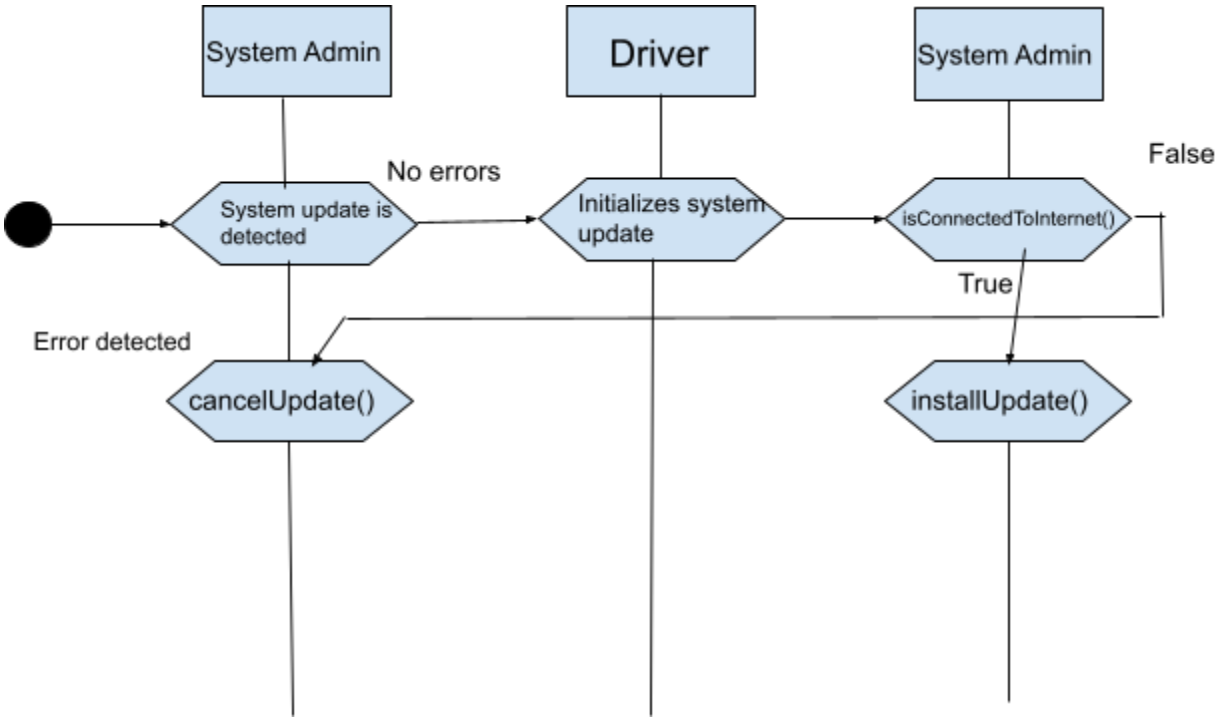
4.3.1 - Self Parking Module



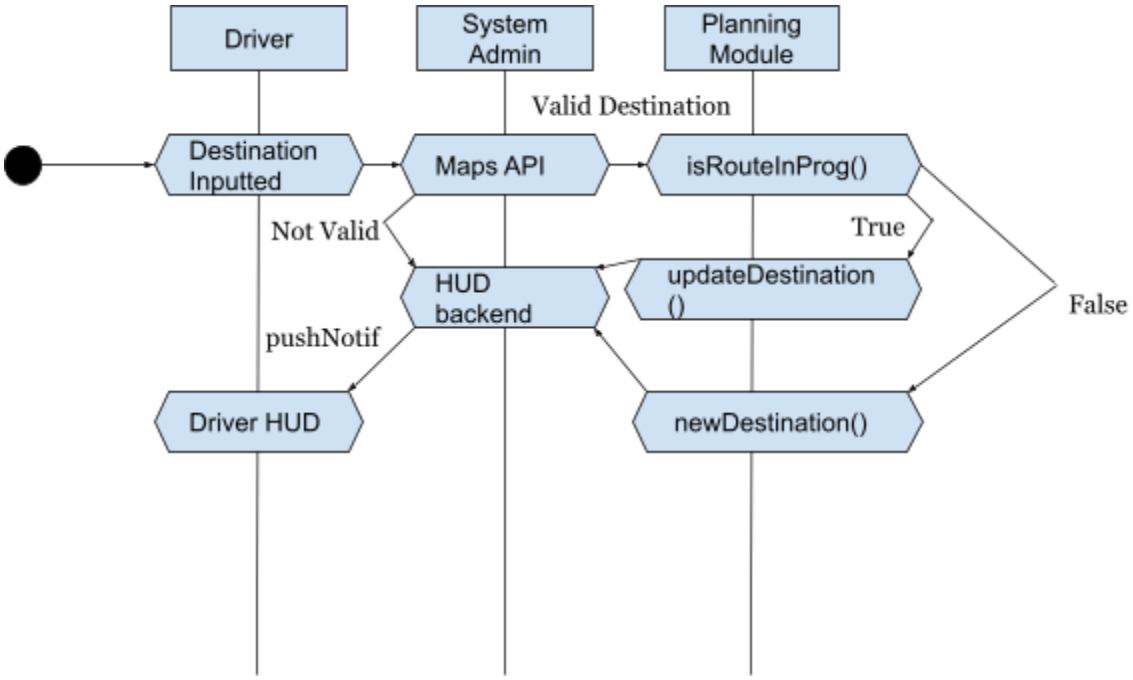
4.3.2 - Lane Control and Merging



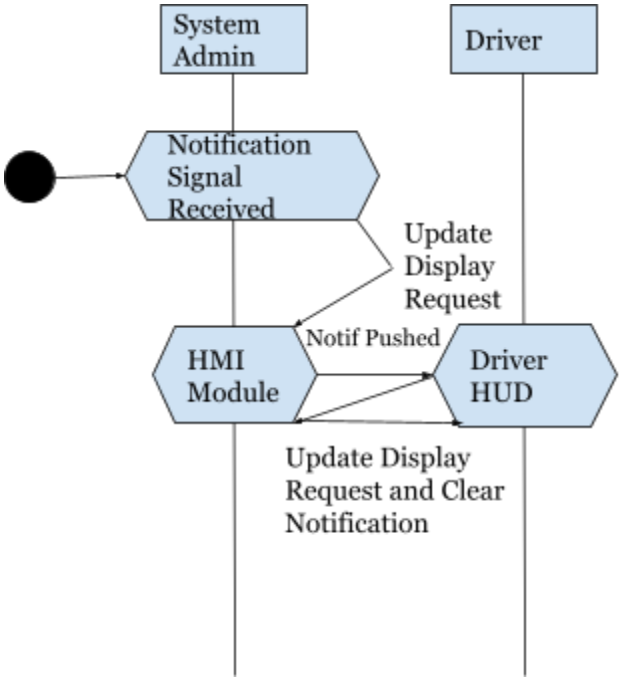
4.3.3 - Perform System Update



4.3.4 - Update Destination



4.3.5 - Push Notification



4.4: Classes

4.4.1 - Self-Parking Module

- SelfParkingModule - class to control self-parking behavior
    - Attributes:
        - parkingSpace: information about the available parking space (e.g. size, location)
        - ultrasonicSensors: array of ultrasonic sensors to detect nearby obstacles
        - parkingCamera: camera to monitor parking behavior
        - steeringMotor: motor to control steering
        - speedMotor: motor to control speed
        - parked: boolean indicating whether the vehicle is currently parked or not
        - currentLocation: current location of the vehicle

    - Methods:
        - getAvailableSpace(): returns information about the available parking space
        - setUltrasonicSensors(): sets up the array of ultrasonic sensors
        - setParkingCamera(): sets up the parking camera

        - steer(): controls the steering motor to guide the vehicle into the parking space
        - accelerate(): controls the speed motor to move the vehicle into the parking space
        - park(): parks the vehicle in the available space
        - unpark(): unparks the vehicle from the parking space
        - updateLocation(): updates the current location of the vehicle
        - isParked(): returns whether the vehicle is currently parked or not

4.4.2 - Lane Control and Merging

- LaneDetection Class:
    - Attributes:
        - lanes (list of Lane objects): list of all detected lanes in the current frame
    - Methods:
        - detectLanes(): detects all lanes in the current frame and updates the lanes attribute
        - getLanePositions(): returns a list of tuples representing the left and right boundaries of each lane in the form (left_x, right_x)
        - changeLanes(): merges lane:s that are close together and updates the lanes attribute
        - getClosestLane(): returns the lane closest to the vehicle's current position
        - isVehicleInLane(lane): returns true if the vehicle is currently in the specified lane, false otherwise

- Lane Class:
    - Attributes:
        - left_boundary (list of tuples): list of coordinates representing the left boundary of the lane
        - right_boundary (list of tuples): list of coordinates representing the right boundary of the lane
    - Methods:
        - getCenterLine(): returns a list of coordinates representing the center line of the lane

4.4.3 - Perform System Update

- SystemUpdate Class:
    - Attributes:

        - version (str): the current version of the system
        - update_available (bool): true if an update is available, false otherwise
    - Methods:
        - checkForUpdate(): checks for updates and updates the update_available attribute
        - downloadUpdate(): downloads and installs the available update
        - getVersion(): returns the current version of the system
        - setVersion(version): sets the current version of the system to the specified value. (only to be used during testing)
        - isConnectedToTheInternet(): checks to see if system admin is connected to the internet

4.4.4 - Update Destination

- HUD: Class meant to represent Driver HUD and its components
    - NavPage - class to hold all navigational display module items
        - Attributes
            - route(Route): current route representing next moves on the road
            - currentLocation(Location): current location of the vehicle
            - destination(Location): current destination of the route
        - Methods
            - getCurrentLocation(): gets current location of the vehicle
            - setCurrentLocation(): updates current location of the vehicle
            - getDestination(): gets destination of the vehicle
            - getRouteInProgress(): bool representing if a route is currently in progress
            - updateRoute(): updates current route
            - updateDestination(): updates current destination
            - newDestination(): updates destination given driver request
- Additional Classes Needed for Implementation
    - Path: Class meant to represent street and direction of travel
        - Attributes
            - street(str): string representing street name
            - direction(str): String representing direction of travel in degrees
        - Methods
            - getStreet(): gets street
            - setStreet(): updates street

            - getDirection(): gets direction
            - setDirection(): updates direction
            - to_string(): gets string representing street and direction in a print-friendly form

- Route: Class representing routes which are the sequence of paths to travel to a destination
    - Attributes
        - paths(list of path): stack of paths needed to be taken to get to destination in order
    - Methods
        - pop(): pops path on top of stack to get next path in route
        - push(): pushes path to top of stack
        - setPaths(): updates stack of paths
        - getPaths(): get stack of paths
- Location: Class representing any location's specific geographical information
    - Attributes
        - address(str): string representing address of location
        - latitude(float): float number representing latitudinal value of the location
        - longitude(float): float number representing longitudinal value the location
    - Methods
        - getPreciseLocation(): gets tuple (latitude, longitude)
        - setPreciseLocation(): updates latitude and longitude
        - getAddress(): gets address string
        - setAddress(): updates address string

4.4.5 - Push Notification

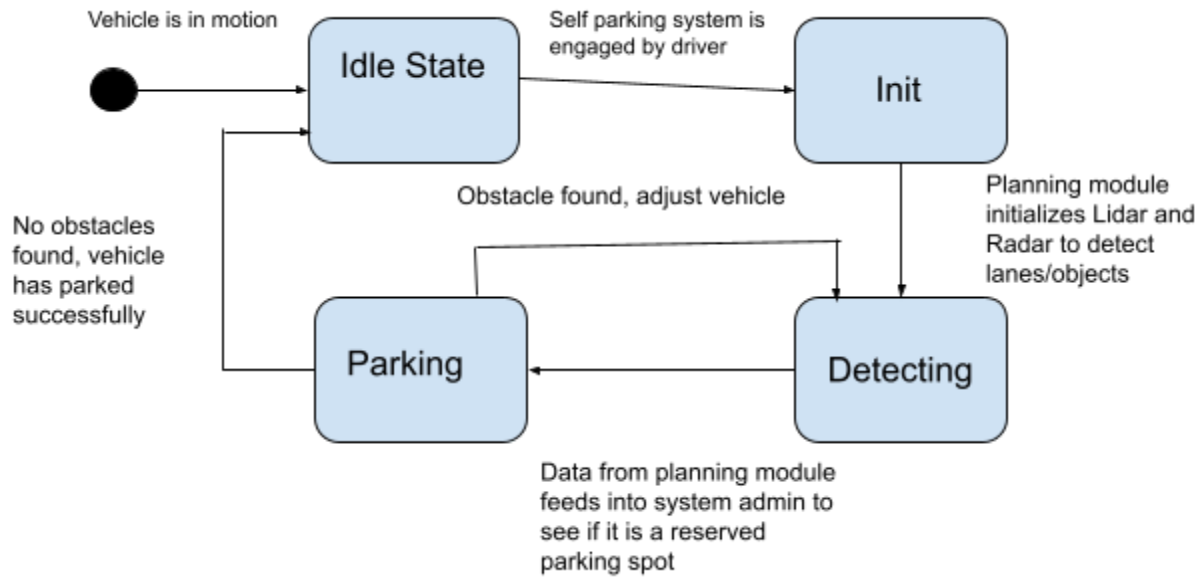
- HUD Class
    - Notification: Class representing any Notification pushed to HUD
        - Attributes
            - length(int): length of dialog box in pixels
            - height(int): height of dialog box in pixels
            - location(tuple of ints): location of bottom right corner of dialog box on HUD in pixels

- title(str): title of Notification
- message(str): body message of Notification

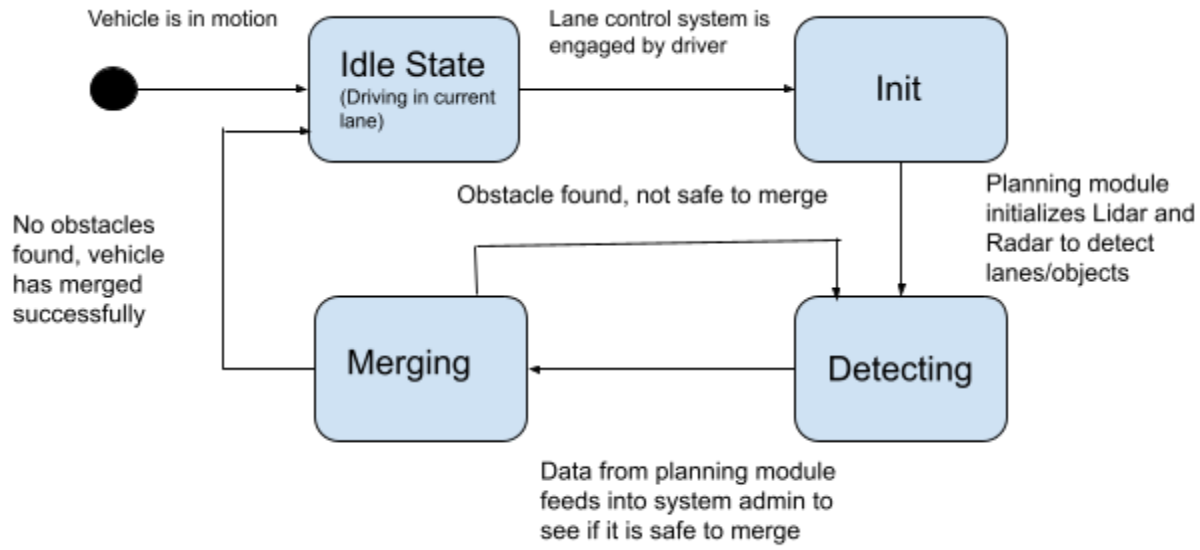
- Methods
  - getDialogBoxDetails(): gets details regarding location of dialog box
  - setDialogBoxDetails(): updates details regarding location of dialog box
  - getTitle(): gets title
  - setTitle(): updates title
  - getMessage(): gets message
  - setMessage(): updates message
  - updateDisplay(): pushes update display request to System Admin
  - clearDisplay(): clears notification dialog box from display

4.5: State Diagrams

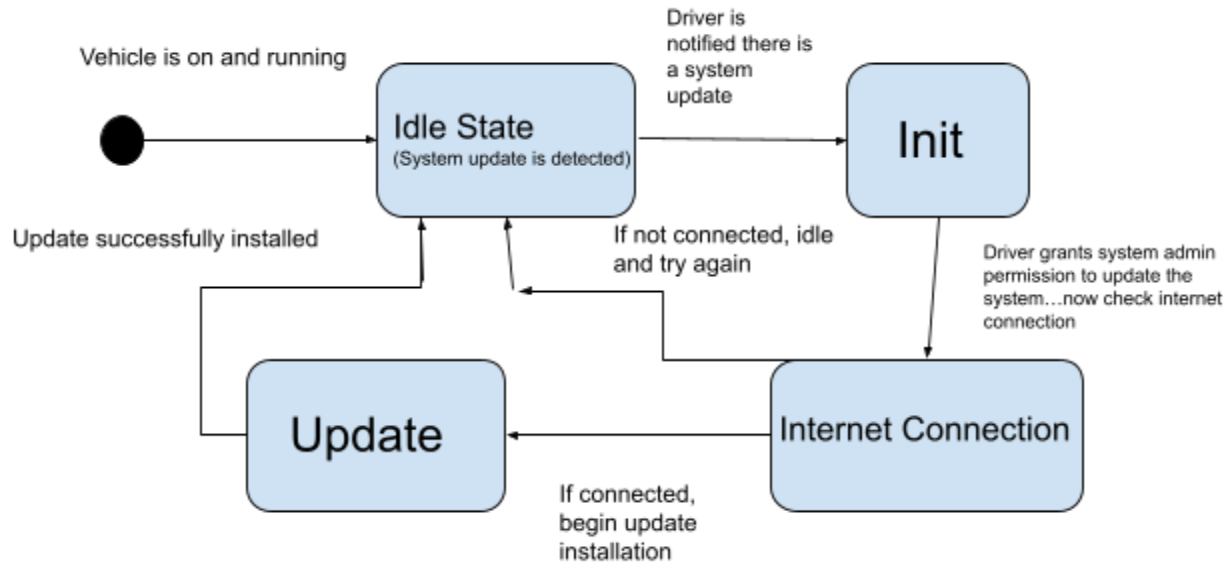
4.5.1 - Self-Parking Module



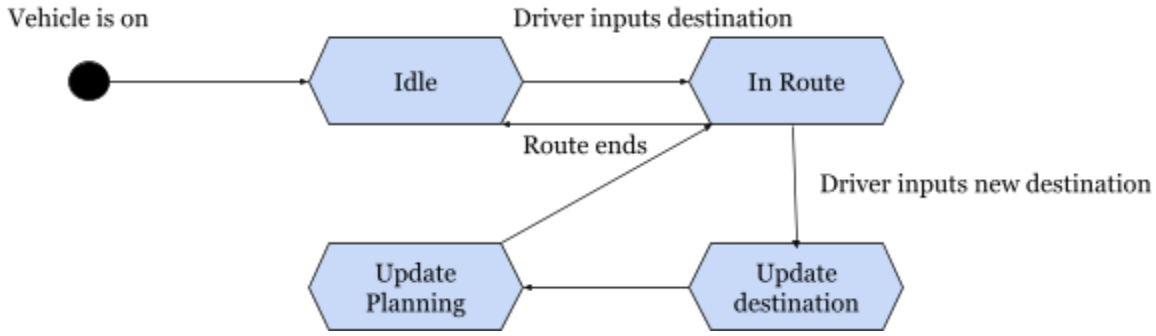
4.5.2 - Lane Control and Merging



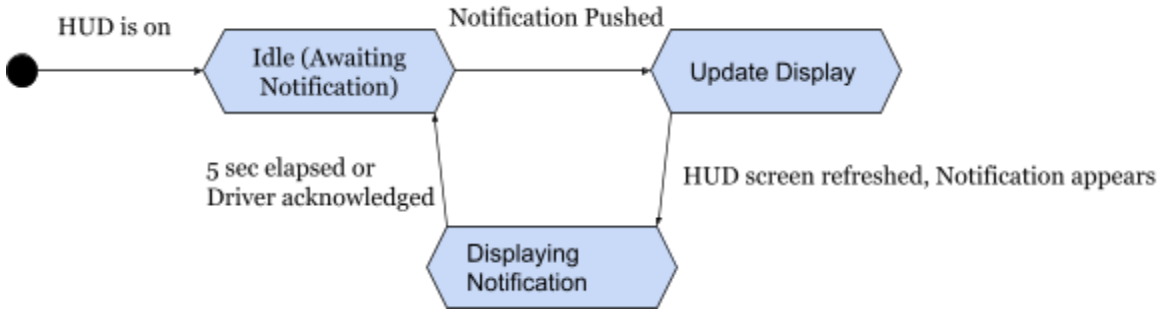
4.5.3 - Perform System Update



4.5.4 - Update Destination



4.5.5 - Push Notification



Section 5: Design

5.1: Software Architecture

Data-centered architecture:

Feasibility for IoT HTL: Feasible.
Fit for IoT HTL: Data is a central part of IoT HTL, and data-centered architecture is appropriate for handling and processing data.
Pros:

- It provides a central data repository, which makes it easier to manage and maintain data. It enables sharing data between different components of the system.

Cons:

- It is not suitable for complex applications that require complex interactions between components.

Data flow architecture:

Feasibility for IoT HTL: Feasible.
Fit for IoT HTL: Data flow architecture is suitable for systems that have a high volume of data flow, which is a common characteristic of IoT HTL.
Pros:

- It is flexible and can handle a large volume of data. It is easy to extend and modify the system as the data requirements change.

Cons:

- It may be difficult to understand the system's behavior as the data flows through different components.

Call return architecture:

Feasibility for IoT HTL: Not feasible.
Fit for IoT HTL: Call return architecture is not suitable for IoT HTL because it does not deal with data processing, which is the main focus of IoT HTL.
Pros:

- It is suitable for small systems with a limited number of components.

Cons:

- It is not appropriate for complex systems that require the exchange of data between different components.

Object-oriented architecture:

Feasibility for IoT HTL: Feasible.
Fit for IoT HTL: Object-oriented architecture is appropriate for IoT HTL because it allows for the creation of objects that represent real-world devices and sensors, which is the foundation of IoT HTL.
Pros:

- It enables the creation of reusable code, which makes it easier to maintain and extend the system. It supports the development of modular and scalable systems.

Cons:

- It may require significant resources to implement and may be difficult to understand for developers who are not familiar with object-oriented programming.

Layered architecture:

Feasibility for IoT HTL: Feasible.
Fit for IoT HTL: Layered architecture is appropriate for IoT HTL because it allows for the separation of concerns, which is critical for managing the complexity of IoT HTL.
Pros:

- It enables the creation of modular and scalable systems. It promotes separation of concerns, making it easier to maintain and extend the system.

Cons:

- It may lead to performance issues if the number of layers is too high, and communication between layers is not optimized.

Model view controller architecture:

Feasibility for IoT HTL: Feasible.
Fit for IoT HTL: Model-view-controller architecture is appropriate for IoT HTL because it separates the data processing logic from the user interface, which is critical for managing the complexity of IoT HTL.
Pros:

- It separates concerns, making it easier to maintain and extend the system. It promotes the creation of modular and scalable systems.

Cons:

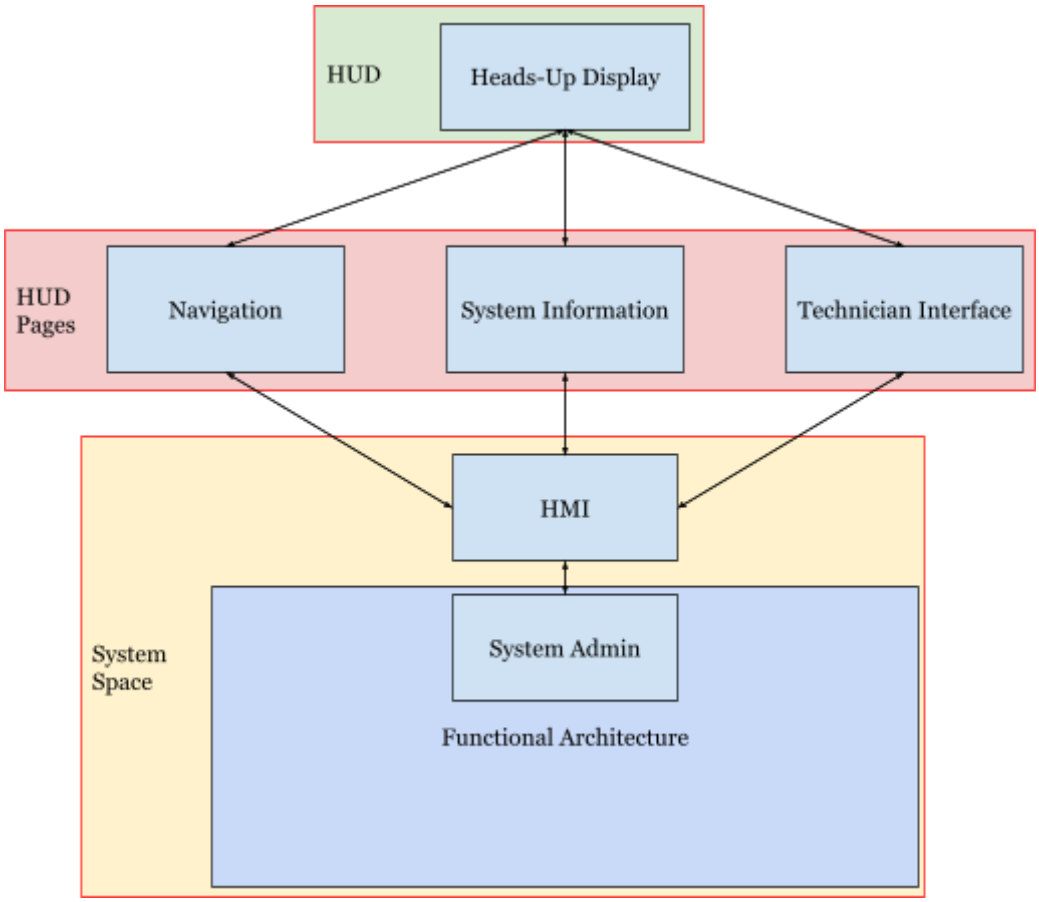
- It may be difficult to understand for developers who are not familiar with the model-view-controller paradigm.

After analyzing the information provided, it has been determined that the optimal software architecture for IoT HTL would be a blend of layered and object-oriented architecture. This approach would facilitate the development of a flexible and expandable system, which could partition responsibilities and encourage the reuse of components. Additionally, it would permit the utilization of object-oriented programming principles to depict actual devices and sensors, a fundamental element of IoT HTL. Nevertheless, it is crucial to consider the appropriate number of layers to guarantee optimal performance and communication between them.

5.2: Interface Design

In order to implement a secure, efficient, and easy-experience HUD for not just drivers to interact with but also technicians and system administrators, we need a streamlined approach to having tech interact with people. In order to achieve this, we have the following design to allow different modules in the vehicle to interact smoothly.

Figure 5.2.1 - Design of Interface



Here there are 3 main components of this design: The HUD, HUD pages, and system space. The purpose of the HUD section is to simply hold information pertaining to the display itself and interactions with it. The HUD is the part of the design that the driver or the technician will directly interact with. These interactions are served as requests sent from the HUD to the HUD Pages which helps to demarcate difference in logic between navigational pages i.e. the map page and all functions pertaining to navigation from system information which stores analytics and statistics of the vehicle from the technician interface which stores all necessary resources for technicians to deal with. Finally, these HUD pages will make requests to system space namely the HMI (Human Machine Interface). Drivers nor technicians are allowed to make changes or interact with the System Space to maintain security of the system.

5.2.1 - Navigational Page

The navigational page of the interface design is as stated, meant to hold everything pertaining to navigation operations in the vehicle. The following operations will need to be implemented for the navigational page to work properly:

- NavPage - class to hold all navigational display module items
    - Attributes
        - route(Route): current route representing next moves on the road
        - currentLocation(Location): current location of the vehicle
        - destination(Location): current destination of the route
    - Methods
        - getCurrentLocation(): gets current location of the vehicle
        - setCurrentLocation(): updates current location of the vehicle
        - getDestination(): gets destination of the vehicle
        - getRouteInProgress(): bool representing if a route is currently in progress
        - updateRoute(): updates current route
        - updateDestination(): updates current destination
        - newDestination(): updates destination given driver request
- Requests
    - Vehicle Location
    - Current Destination
    - Current Route
    - Current Path
    - Update Destination
    - End Route

To reiterate, these requests will be made from the navigational page to the HMI where in System Admin, logic of these requests are carried out and if information is requested,

it is passed through the HMI back to the page of origin, in these cases the navigational page. As long as the navigational page is currently showing on the display, the following attributes should be updated with the specified frequency of requests:

Vehicle Location	1 sec
Current Path	New path taken
Current Route	Destination updated

5.2.2 - System Information Page

The system info page is meant to store all information pertaining to the vehicle itself. While less dynamic requests are made (unique requests by drivers), static requests such as periodic checking are very common. The following operations are needed to implement this page:

- SysPage - Class to hold all sys info page information
    - Attributes
        - speed(float): current speed of vehicle
        - charge(float): current charge level of vehicle
        - tirePressures(list of floats): list containing current tire pressures of each tire
        - isUpdateAvail(bool): Bool representation of if a software update is available
        - runtime(int): representing in seconds the amount of time the engine has been running
    - Methods
        - updateSpeed()
        - updateCharge()
        - updateTirePressures()
        - checkPressures()
        - softwareUpdateAvail()
        - getRuntime()
- Requests
    - Vehicle Speed
    - Vehicle Charge
    - Tire Pressure (Specific tire)
    - Software Update Available?

While the class for the system information page is listed below, the requests are meant to be the operations at the higher level with class methods being used in runtime operation of the vehicle not by technician nor driver. These attributes will be updated according to the following request frequency table

Vehicle Speed	0.25 seconds
Vehicle Charge	0.5 seconds
Tire Pressure (all tires)	0.100 seconds
Software Update Available	10 minutes

5.2.3 - Technician Interface Page

This page is simply the launchboard page for the technicians planning to work on the vehicle. Through this page can technicians log in to their system admin accounts, check/download logs, view system messages and perform system updates.

- TechPage - Class holding information pertaining to technician interface
    - Attributes
        - AuthLogs
        - AccessLogs
    - Methods
        - signIn(uid, pwuid): method to make sign in attempt using uid (user) and pwuid (password)
        - downloadUpdate
        - installUpdate
        - updateFullSuite(): method to call the above two methods to make system update
- Requests
    - Validate Sign in Attempt
    - Start Software Update
    - Download Logs

While features aren't what is important on this page, questions of accessibility and proper authentication is very important here. It is imperative that only authorized users are allowed to access past the log in page and auth logs are kept and maintained whenever someone logs in. In addition to auth logs, access logs should also be created to

hold information about which technician doing what job in the interface with accompanying information such as time accessed, uid, and request accessed.

5.3: Component-level Design

To transform the structural elements of a self-driving car into procedural descriptions of software components, we need to identify the major components of the car and understand their functions. Here is a high-level breakdown of the components and their respective software components:

5.3.1 - Perception System

- **Camera perception module:** This module processes the images captured by the car's cameras to identify objects in the environment, such as other vehicles, pedestrians, and traffic signs. It uses computer vision techniques such as object detection and classification algorithms.
    - Receive camera data
    - Prepare camera data for processing
    - Apply computer vision algorithms to identify objects
    - Update perception data with identified objects

- **Lidar perception module**: This module processes the point clouds generated by the car's Lidar sensors to identify objects in the environment, such as other vehicles, pedestrians, and obstacles. It uses algorithms such as clustering and segmentation to group the points into objects.
    - Receive Lidar data
    - Prepare Lidar data for processing
    - Apply clustering and segmentation algorithms to group points into objects
    - Update perception data with identified objects

- **Ultrasonic perception module:** This module processes the distance measurements generated by the car's ultrasonic sensors to identify obstacles in the environment.
    - Receive ultrasonic data
    - Prepare ultrasonic data for processing
    - Analyze data to identify obstacles in the environment
    - Update perception data with identified obstacles

- **Radar perception module:** This module processes the radar data generated by the car's radar sensors to identify objects in the environment, such as other vehicles,

pedestrians, and obstacles. It uses signal processing techniques such as filtering and beamforming to extract information from the radar signals.

- Receive radar data
- Prepare radar data for processing
- Update perception data with identified objects

- **Perception fusion module:** This module integrates the data from all the perception sensors to create a unified representation of the environment around the car. It uses data association techniques to group the detections from different sensors into objects and track their movements over time.

- Receive perception data from all perception modules
- Prepare perception data for fusion
- Use data association techniques to group detections from different sensors into objects
- Update perception data with fused objects and their attributes

5.3.2 - Navigation System

- **Localization module:** This module uses data from the car's sensors and maps to determine the car's position in 3D space. It uses techniques such as sensor fusion and filtering to improve the accuracy of the position estimate.

- Receive perception data and map data
- Prepare data for localization
- Apply sensor fusion and filtering techniques to estimate the car's position in 3D space
- Update localization data with position estimate

- **Map building module:** This module uses data from the car's sensors to build a map of the environment around the car. It uses techniques such as Simultaneous Localization and Mapping (SLAM) to build a map as the car moves.

- Receive perception data
- Prepare data for map building
- Apply SLAM algorithms to build a map of the environment
- Update map data with new map features

- **Path planning module:** This module takes the car's current position and destination and plans a high-level, collision-free path for the car to follow. It uses techniques such as A* search and graph-based planning. With a consistent heuristic, A*

is guaranteed to find an optimal path without processing any node more than once and A* is equivalent to running Dijkstra's algorithm with the reduced cost **d'(x, y) = d(x, y) + h(y) − h(x)**.

- Receive localization data and destination
- Prepare data for path planning
- Apply A* search or graph-based planning algorithms to plan a path to the destination
- Update path data with planned path

- **Trajectory planning module:** This module takes the high-level path and generates a low-level, smooth trajectory for the car to follow. It takes into account factors such as the car's dynamics, the road geometry, and the speed limits.
  - Receive path data
  - Prepare data for trajectory planning
  - Apply trajectory generation algorithms to plan a smooth trajectory along the path
  - Update trajectory data with planned trajectory

5.3.3 - Control System

- **Drive-by-wire module:** This module sends control commands to the car's actuators, such as the throttle, brakes, and steering. It translates the planned trajectory into actuator commands in real-time.
  - Receive trajectory data
  - Prepare data for actuator commands
  - Translate trajectory into actuator commands
  - Send actuator commands to the car's actuators

- **Vehicle dynamics module:** This module models the car's dynamics, such as its mass, engine power, and tire friction. It uses this model to predict the car's behavior in response to control inputs.
  - Receive data on the car's dynamics
  - Prepare data for modeling
  - Use modeling techniques to create a dynamic model of the car
  - Update model data with predictions of the car's behavior

- **Motion planning module:** This module generates a motion plan, which specifies the car's desired speed and acceleration over time. It uses techniques such as optimal control and model predictive control to generate the plan.
    - Receive updated localization, perception, and trajectory data
    - Prepare data for motion planning
    - Apply motion planning techniques to generate a speed and acceleration plan for the car
    - Update motion plan data with planned speed and acceleration

- **Motion control module:** This module uses feedback control to adjust the car's speed and acceleration to follow the planned trajectory. It uses techniques such as proportional-integral-derivative (PID) control to adjust the car's actuators in real-time.
    - Receive data from the drive-by-wire module and the motion planning module
    - Prepare data for motion control
    - Use feedback control techniques such as PID to adjust the car's actuators in real-time
    - Update control data with adjusted actuator commands

5.3.4 - Human Interface System

- **User interface module:** This module provides a graphical interface for the passengers to interact with the car. It displays information such as the car's status, the planned route, and the estimated time of arrival.
    - Receive data on the car's status, route, and estimated time of arrival
    - Prepare data for display
    - Display information on a graphical user interface
    - Update display data with new information

- **Audio module:** This module provides audio feedback to the passengers, such as turn-by-turn directions or alerts about potential hazards.
    - Receive audio data
    - Prepare audio data for playback
    - Play audio feedback to the passengers
    - Update audio data with new audio feedback

- **Touchscreen module:** This module handles touch inputs from the passengers, such as setting a destination or adjusting the climate controls.
    - Receive touch data

- Prepare touch data for processing
- Process touch inputs and update the car's settings
- Update touch data with processed inputs

- **Input recognition module:** This module recognizes other forms of input from the passengers, such as voice commands or gestures.
    - Receive input data
    - Prepare input data for processing
    - Recognize input commands using natural language processing or computer vision algorithms
    - Update input data with recognized commands

5.3.5 - Safety System

- **Collision avoidance module:** This module uses perception data to detect potential collisions with other objects in the environment. It generates a warning or takes action to avoid the collision if necessary.
    - Receive perception data
    - Prepare data for collision avoidance
    - Detect potential collisions using perception data
    - Generate a warning or take action to avoid the collision if necessary

- **Emergency braking module:** This module detects potential emergencies, such as sudden obstacles in the car's path, and applies the brakes automatically to prevent a collision.
    - Receive perception data
    - Prepare data for emergency braking
    - Detect potential emergencies using perception data
    - Apply brakes automatically to prevent a collision if necessary

- **Lane-keeping module:** This module uses perception data to keep the car centered in the lane and prevent it from drifting off course. It can also handle lane changes and merges.
    - Receive perception data
    - Prepare data for lane-keeping
    - Keep the car centered in the lane using perception data
    - Handle lane changes and merges if necessary

- **Vehicle monitoring module:** This module monitors the car's performance and alerts the driver if any issues are detected, such as low tire pressure or engine problems.
    - Receive vehicle data
    - Prepare data for monitoring
    - Monitor the car's performance using the data
    - Generate an alert if any issues are detected

Component-level design is a critical aspect of building a self-driving car. This approach breaks down the complex system into smaller, more manageable components, allowing for easier development and testing of each module. By identifying the major components of the car and understanding their functions, software engineers will have the ability to develop robust and efficient software components to control each aspect of the car's behavior.

Section 6/7: Coding/Tests

ALL of the following implementations are based off of this State struct that keeps track o the state of the vehicle at all times with various data types

```
struct State {
   // Navigation related fields
   double latitude;          // Current latitude of the car
   double longitude;         // Current longitude of the car
   double altitude;          // Current altitude of the car in meters
   double speed;             // Current speed of the car in km/h
   double heading;           // Current heading of the car in degrees (0 = North)
   double distance_traveled;   // Total distance traveled by the car in km
   int satellites_used;       // Number of GPS satellites currently in use

   // Perception related fields
   bool is_obstacle_detected;      // Is there an obstacle in the car's path?
   bool is_traffic_light_detected; // Is a traffic light in view of the car's cameras?
   bool is_pedestrian_detected;    // Is a pedestrian in view of the car's cameras?
   double distance_to_obstacle;    // Distance to the nearest obstacle in meters
   double obstacle_size;          // Size of the detected obstacle in meters (e.g. width,
height)

   // Control related fields
   double steering_angle;  // Current steering angle of the car's wheels in degrees
```

```
    double throttle;       // Current throttle position (0-100%)
    double brake;          // Current brake position (0-100%)
    double acceleration;   // Current acceleration of the car in m/s^2

    // Battery related fields
    double battery_level;  // Current battery level in percent
    double charging_rate;  // Current charging rate in kW
    double estimated_range; // Estimated range of the car in km based on current battery
level

    // System related fields
    bool is_autonomous_mode_enabled;   // Is autonomous driving mode currently
enabled?
    bool is_vehicle_connected;         // Is the car currently connected to the internet?
    bool is_software_update_available;  // Is a software update available for the car's
system?
};
```

The rest of the code for this section will be in the folder in the GitLab titled “NavAI v. 1.6.” If you want us to paste the code into the document please let us know and we will it was just being annoying with the formatting and was wasting too much time!

References

[1] "How Self-Driving Vehicles Work: Sensor Systems." *Udacity*, 27 Sept. 2022, https://www.udacity.com/blog/2021/03/how-self-driving-vehicles-work-sensor-systems.html#.

[2] Prakash, Aditya. "Sensor Fusion for Self Driving." *Autonomous Vision Blog*, 11 May 2021, https://autonomousvision.github.io/transfuser/. Accessed 11 February 2023.

[3] Asiag, Jodi. "How Ultrasonic Sensor Data is Powering Automotive IoT." Otonomo, 17 April 2021, https://otonomo.io/blog/ultrasonic-data-automotive-iot/.

[4] Lee, B. Wei, Y. Guo, Yuan. "AUTOMATIC PARKING OF SELF-DRIVING CAR BASED ON LIDAR" *ResearchGate*, September 2017. https://www.researchgate.net/publication/319853826_AUTOMATIC_PARKING_OF_SELF-DRIVING_CAR_BASED_ON_LIDAR.

[5] https://resources.system-analysis.cadence.com/blog/msa2022-the-use-of-radar-technology-in-autonomous-vehicles

[6] Bautista-Camino, Pedro, et al. "Local Path Planning for Autonomous Vehicles Based on the Natural Behavior of the Biological Action-Perception Motion." energies, 2022.

[7] "Motion planning module in Autonomous Vehicle — Mission planner." Medium, 25 November 2020, https://medium.com/intro-to-artificial-intelligence/motion-planning-module-in-autonomous-vehicle-mission-planner-671b2155dec1. Accessed 19 February 2023.

[8] https://arxiv.org/pdf/2011.08729.pdf