Udacity Self-Driving Car – Capstone Project

Team: Millenium Vulkan

Name	Email
George Terzakis	terzakig@hotmail.com
Martin Herzog	martin.mh.herzog@gmail.com
Yuda Liu	liu_yuda@163.com
Atul Acharya	Atul.acharya@gmail.com
Yoni Azuelos	yonazuel@hotmail.com

Cohort: Graduating October 2017

Project Video:
Youtube Link

Udacity Self Driving Car – Capstone Project

Introduction

This is the final capstone project in Udacity's Self-Driving Car (SDC) Nanodegree. In this project, we write code that will autonomously drive Udacity's self-driving car "*Carla*", an actual car equipped with necessary sensors and actuators, around a small test track. We test and develop the code on a simulated car using a simulator provided by Udacity.

The project brings together several modules taught in the Nanodegree: Perception, Motion Planning, Control, etc. The code is written using the well-known Robotics Operating System (ROS) framework which works both on the simulator as well as the actual car.

System Architecture

The architecture of the system is depicted below. The system consists of three key modules: **Perception**, **Planning**, and **Control**. Each module consists of one or more ROS nodes, which utilize *publish/subscribe* (pub-sub) and *request/response* patterns to communicate critical information with each other while processing information that helps to drive the car.

At a high level perspective, the **Perception** module includes functionality that perceives the environment using attached sensors (cameras, lidars, radars, etc.), such as lane lines, traffic lights, obstacles on the road, state of traffic lights, etc. This information is passed to the **Planning** module, which includes functionality to ingest the perceived environment, and uses that to publish a series of *waypoints* ahead of the car along with target velocities. The waypoints constitute a projected trajectory for the car. The **Control** module takes the waypoints and executes the target velocities on the car's controller. The car is equipped with a *drive-by-wire* (DBW) functionality that allows the controller to set its throttle (acceleration and deceleration), brake and steering angle commands. When the commands are executed, the car drives itself on the calculated trajectory. The entire cycle repeats itself continually so that the car continues driving.

The modules are implemented as a set of ROS Nodes. Some nodes are implemented by Udacity, while other nodes are implemented by the project team. This document mainly describes the nodes implemented by the team.

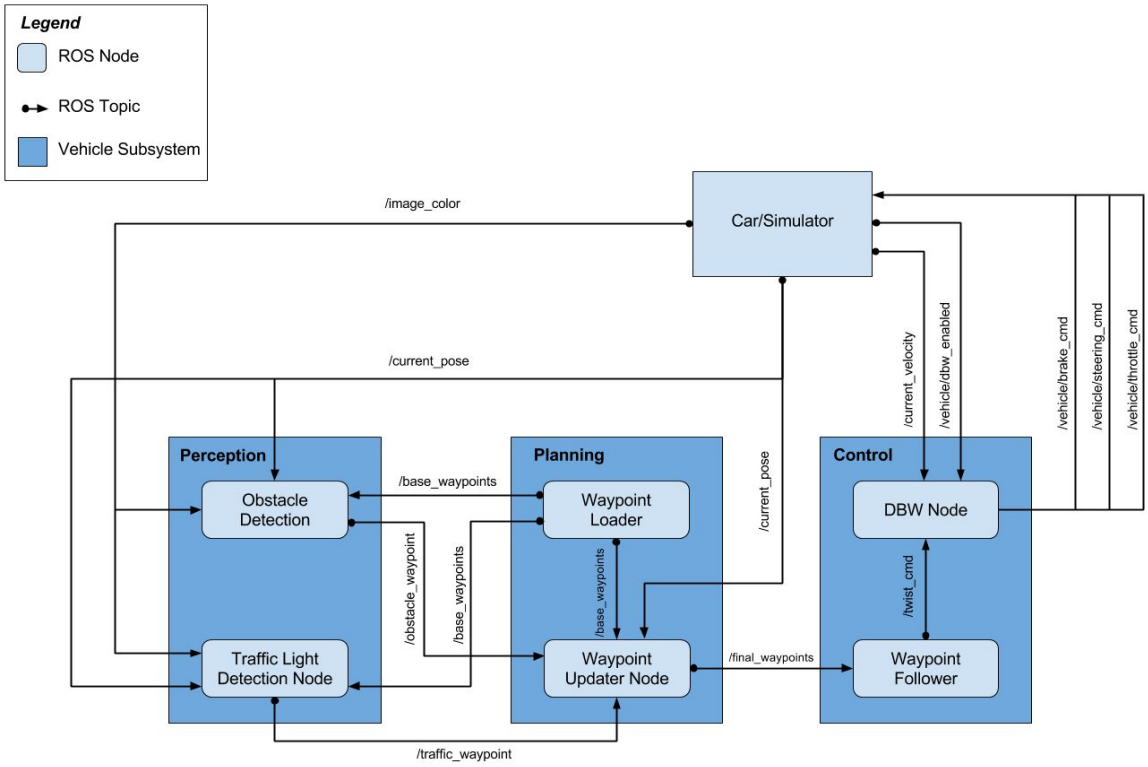


Figure 1 - System Architecture

A Note on ROS Node Implementation

A ROS node is an entity (a Python or C++ program) that performs certain useful actions. ROS utilizes two key mechanisms to communicate between nodes.

- **Publish/Subscribe mechanism**
  In a pub/sub system, nodes can be *Publishers* or *Subscribers* or both. A Publisher publishes its information on designated *Topics*. Topics are simply channels that are open to any Subscriber. *Messages* published on a topic are preformatted so that a subscriber knows how to parse them. Published messages are simply broadcast on a topic, the order of delivery and reception is not guaranteed. It is up to the publisher and subscriber to agree on exactly the content and the format of the message. There is no explicit acknowledgment of the receipt of a message. A publisher, thus, does not wait or block for message delivery, thus ensuring parallel processing of message consumption. A subscriber, on the other hand, also does not wait for message reception; it simply runs a designated callback function every time a message is received. Any number of

publishers may publish to a topic, and any number of consumers may consume messages from a topic. The pub/sub system thus ensures *loose coupling* between message senders and recipients, while still allowing for scalability of publishers/subscribers.

- **Request/Response mechanism**
  The request/response mechanism, on the other hand, is the opposite of publish/subscribe system. In this, a traditional client-server model is assumed, where a client sends messages to a server that is explicitly waiting (i.e. blocking) for a message; the server only performs its functions when it receives a request. When the server completed its requests, it typically informs the client, and returns. The client can then continue further processing. Request/response mechanism is used for critical or time-sensitive communication where message delivery is fully expected and guaranteed, and where a node (a client or server) absolutely needs to process a request before moving further.

  In the project, we use both mechanisms and indicate which ones are used.

Implementation

As the Figure 1 above shows, there are six ROS nodes in three different modules. Of these, we implement three ROS nodes: **Traffic Light Detector**, **Waypoint Updater** and **Drive-by-Wire (DBW).** Of the other three nodes, two are implemented by Udacity: **Waypoint Loader node**, and **Waypoint Follower**. The last one (Obstacle Detector) is implemented as part of Traffic Light Detector

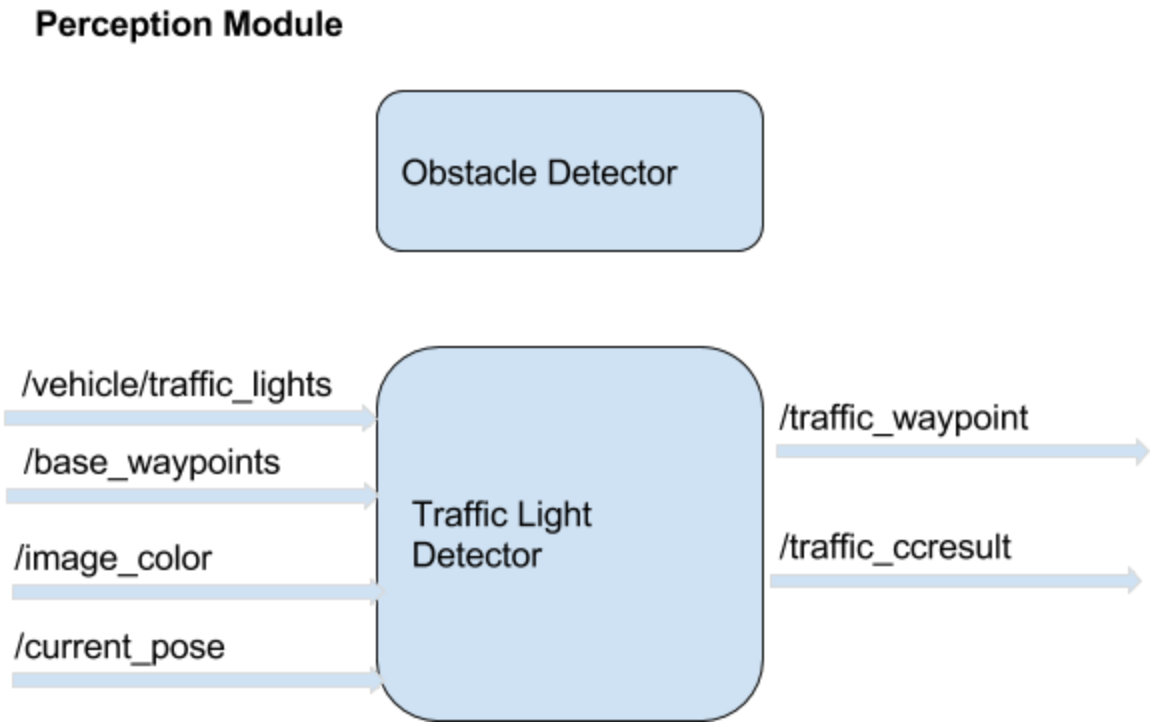
Perception Module

The Perception module takes inputs from various sensors attached to the car and makes sense of the environment. These sensors can be:

- **Lidar** – that detects a 3D point-cloud of the car's environment using lasers
- **Radar** – that detects objects (cars, pedestrians, obstacles) in the car's immediate vicinity using millimeter waves
- **Camera** – that captures images (such as traffic signs, obstacles, pedestrians, etc)

A typical perception module performs ***sensor fusion***, gathering all raw data from various sensors, to make sense of obstacles and free space ahead of, and around, the vehicle. In our case, we use the car's camera to capture and process images of traffic lights and detect the state of these lights.

In this project, the Perception module has two ROS nodes: **Obstacle Detector** and **Traffic Light Detector**. The Obstacle Detector is not required by the project rubric. The discussion below is for the Traffic Light Detector node.



The **Traffic Light (TL) Detector** uses camera images to detect traffic lights and their states, and publishes information about the next traffic light ahead of the vehicle.

The TL Detector subscribes to three topics:

- **/base_waypoints** – provides a complete list of waypoints on the course
- **/current_pose** – provides the car's current position on the course
- **/image_color** – provides an image stream from the car's camera

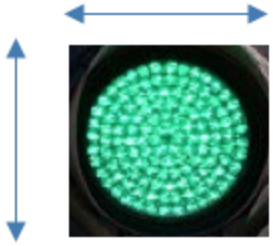
Using Cartoon Templates for Reliably Detecting Red Traffic Lights

Although we know the camera intrinsic parameters f_x and f_y to a certain degree of accuracy, the point of intersection of the camera optical axis with the image plane (i.e. the "center" of the image, typically denoted as (c_x, c_y) is not given. Thus, it is not really possible to obtain a realistic estimate of where the traffic lights appear in the image and therefore the actual scope is the entire image.

However, it is possible to obtain a reasonable estimate of the **size** of an image patch containing the traffic light as follows:

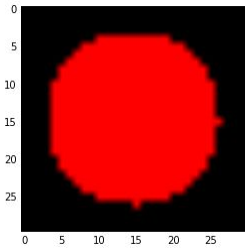
$$w_p = f_x \frac{W_{tl}}{d_{tl}}$$
$$h_p = f_y \frac{H_{tl}}{d_{tl}}$$

where W_{tl} and H_{tl} are the width and height of a single-color traffic light, i.e., the rectangle that inscribes a round red/yellow/green light indicator as show below:



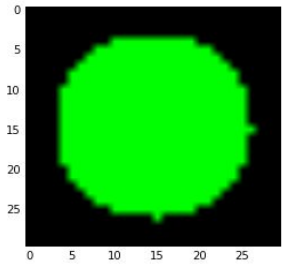
In the particular case of a **red** traffic light, the color and shape would make it highly distinctive in an image.

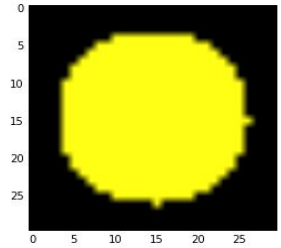
A neural network trained discriminatively to recognize a red traffic light would ideally store a general representation of the red traffic light which can be loosely regarded as a "cartoon" image of a **red disk inscribed in a black square** as shown below:



Thus, to circumvent the time-consuming procedure of training a convolutional classifier to detect traffic lights in images, we instead **tried to guess what would be the pattern of the weights of a fully trained "red light-vs-all" classifier would look like** if we could actually visualize them. We therefore used this simple pattern as a resizable **template** that we can **slide** and **correlate** with the image in order to observe a best match. If indeed the image contains a red traffic light somewhere, then the **normalized correlation coefficient** should be high (we observed that **above 0.7** yields 100% accurate true – positives and true-negatives). Although the rosbag data did not contain frames in which the traffic light is red, we conjecture that our cross-correlation based classifier will be able to discern red light confidently. To search for a matching patch in the incoming image, we use equations $w_p = f_x \frac{W_{tl}}{d_{tl}}$ and $h_p = f_y \frac{H_{tl}}{d_{tl}}$ to **appropriately resize the template**, thereby avoiding exhaustive search across multiple scales.

We used a similar approach to detect **yellow** and **green** traffic lights discriminatively. In particular, we created templates patches of black-filled squares containing a yellow and green disk inscribed as shown in the images below:





Empirically we determined that red templates are matched with high accuracy. Thus, once a red light is recognized, the image is not scanned for yellow and green ones. When however the best red correlation score is **below 0.7** the classifier examines the green scores and if they also are below 0.7, we eventually move on to the yellow correlation scores. If all scores turn out to be below 0.7, the classifier decides that classification is **inconclusive**.

Extending the Classifier using Linear SVMs

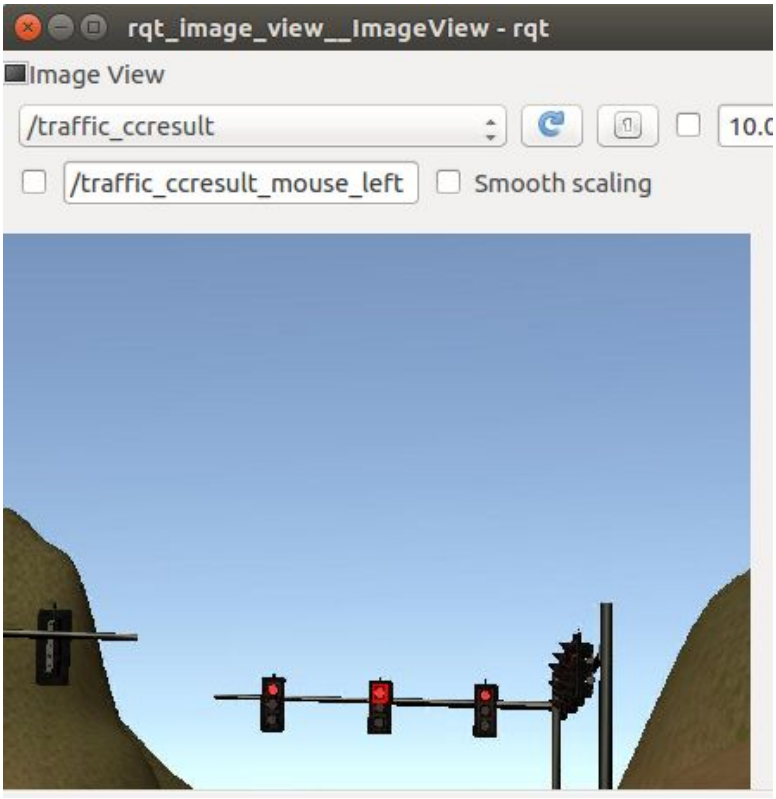
It should be noted that our approach can be easily extended to a classifier using **linear support vector machines**, which can be much more easy to train (as opposed to a convolutional neural network) to recognize simple traffic light patches. For the needs of this assignment, we only needed to enable the car to avoid violating red traffic lights and we therefore treated yellow and green states as “don’t care”. However, with the use of simple linear SVMs, the method can most probably detect all three colors very accurately and with a small training set that should comprise no more than 50 images per traffic light color.

The TL Detector Node and Relative Topics

Upon detection, the TL detector *publishes* the *index* of a waypoint that situated slightly ahead (3 positions ahead) of the stop line of the approaching traffic light. This way, the car is able to stop even if the red light has switched on very late. The aforementioned index along with the state of the traffic light is published in **/traffic_waypoint**. The message type of **/traffic_waypoint** is essentially a 32-bit integer. We used **the 16 lower bits for indexing** the traffic light waypoint and

the upper 16 to store the traffic light state (i.e., **RED**, **GREEN**, **YELLOW**, **UNKNOWN**). In the case that no traffic light exists ahead, then the message contains -1. Dividing the message into a lower and upper word allowed us to use the existing message structure with the need to employ something elaborate such as a MultiArray. The figure below illustrates our message formatting:

Furthermore, the node publishes a topic named **/traffic_ccresult** which marks the detected traffic light with a **rectangle** colored accordingly (i.e., red, green or yellow). The Figure below illustrates red light detection.



Planning Module

The following schematic shows the nodes in the Planning module.

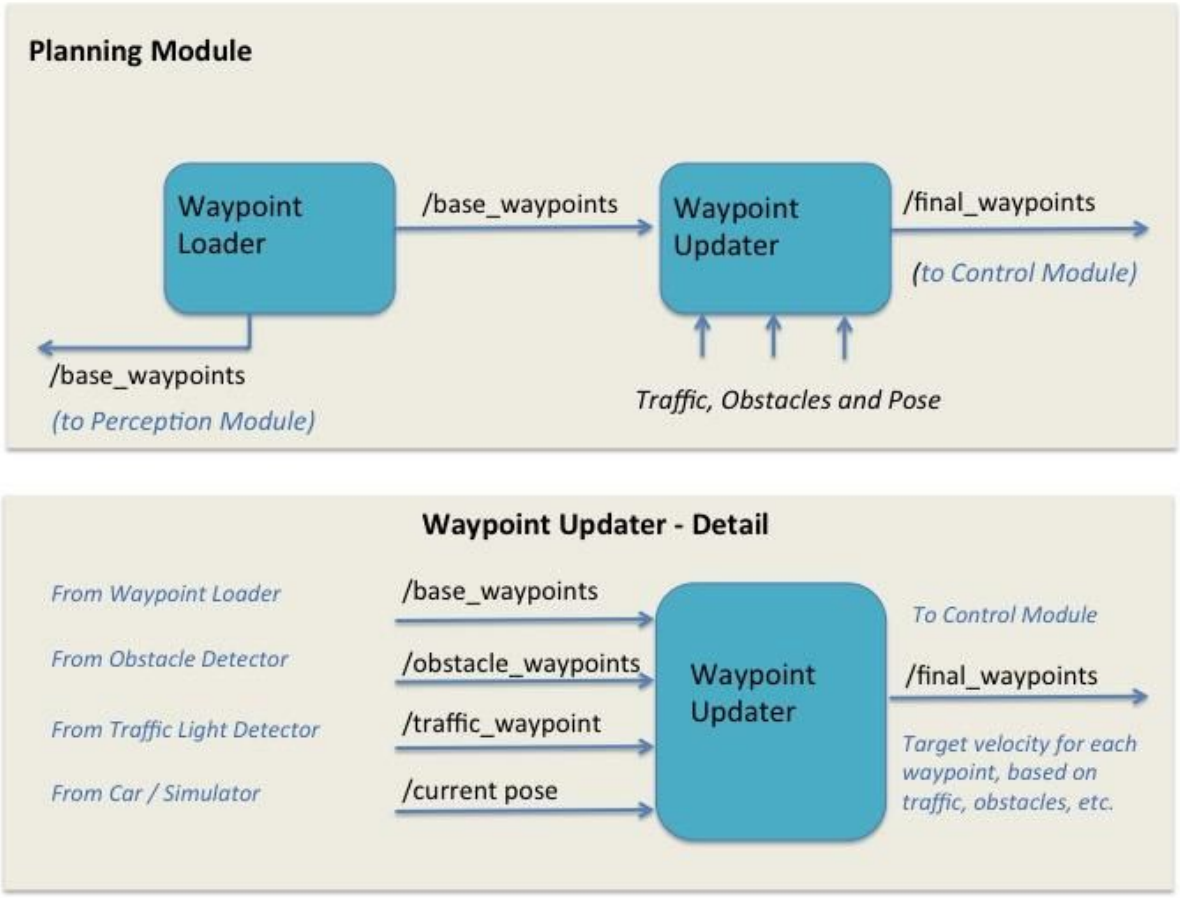


Figure 2 - Planning Module

Our project implements the Waypoint Updater node. The purpose of the Waypoint Updater node is to update the target velocities of final waypoints, based on traffic lights, state of the lights, and other obstacles.

The node subscribes to the following topics:

- **/base_waypoints** – containing all the waypoints in front of, and behind, the vehicle
- **/obstacle_waypoints** – the waypoints that are obstacles (not used). This information comes from the Perception module
- **/traffic_waypoint** – the waypoint of the **red** traffic light ahead of the car. This information comes from the Perception module
- **/current_pose** – the current position of the vehicle, as obtained by the actual car, or the simulator

The Waypoint Updater publishes information to a single topic:

- **/final_waypoints** – this is a list of <N> waypoints ahead of the car (in order from closest to farthest) and the target velocities for those waypoints.

The Updater node first extracts its own position.

Brute force vs KD-Tree based Nearest neighbor search

The waypoint updater searches for the **next** waypoint in front of the own position. This means it is required to first locate the **closest** waypoint to the car's location, and thereafter decide whether this waypoint lies **in-front** of **behind** the car. A single search is a linear operation with number of steps $O(N)$ in the number of waypoints (N). On the other hand, if the waypoints are stored in a KD-tree structure, then the nearest neighbor search is $O(2N^{\frac{1}{2}})$ in the **worst case,** meaning that the search will 98% of the time faster than linear search in the worst case scenario. In practice, filling the KD-tree with roughly 10,000 points took approximately 20 seconds, but we did not observe a significant difference in performance versus the brute – force approach. In the current implementation, the use of KD-tree is disabled to avoid delays upon launching the simulation, but it is an available option.

Reaction on traffic lights

The car reacts in different ways on traffic lights:

1. **Stop**
   The car smoothly stops in configured distance DIST_STOP_TL in front of the stopping line.
   This state is used if the traffic light is red or yellow and was not decided before to drive over the stopping line of the traffic light (see override).
2. **Slowdown**
   The car slows down and reaches the half maximum allowed speed at the configured distance in front of the stopping line.
   This state is used, for example, if the traffic light went from red to yellow and the car is near the stopping line. So it's always used if it's not safe to drive with maximum velocity towards the stopping line.
3. **Drive at maximum velocity**
   This is the standard state if no other state is active.
4. **Override the stopping line**
   This state ensures that the car does not brake too late in front of the stopping line if no reaction time is left. It is especially used if the traffic light goes from red to yellow and the time to the stopping position is too low. This ensures that the car does not come to a stop in the middle of the intersection.

The maximum allowed velocity is read from the waypoints which are published by the waypoint loader. Depending on the different states mentioned above the target velocities of the upcoming waypoints are published as a message to the **/final_waypoints** topic. The Updater node publishes 80 waypoints ahead of the car to allow for the control module to safely project a trajectory in front.

Control Module

The following schematic shows the nodes in the Control module.

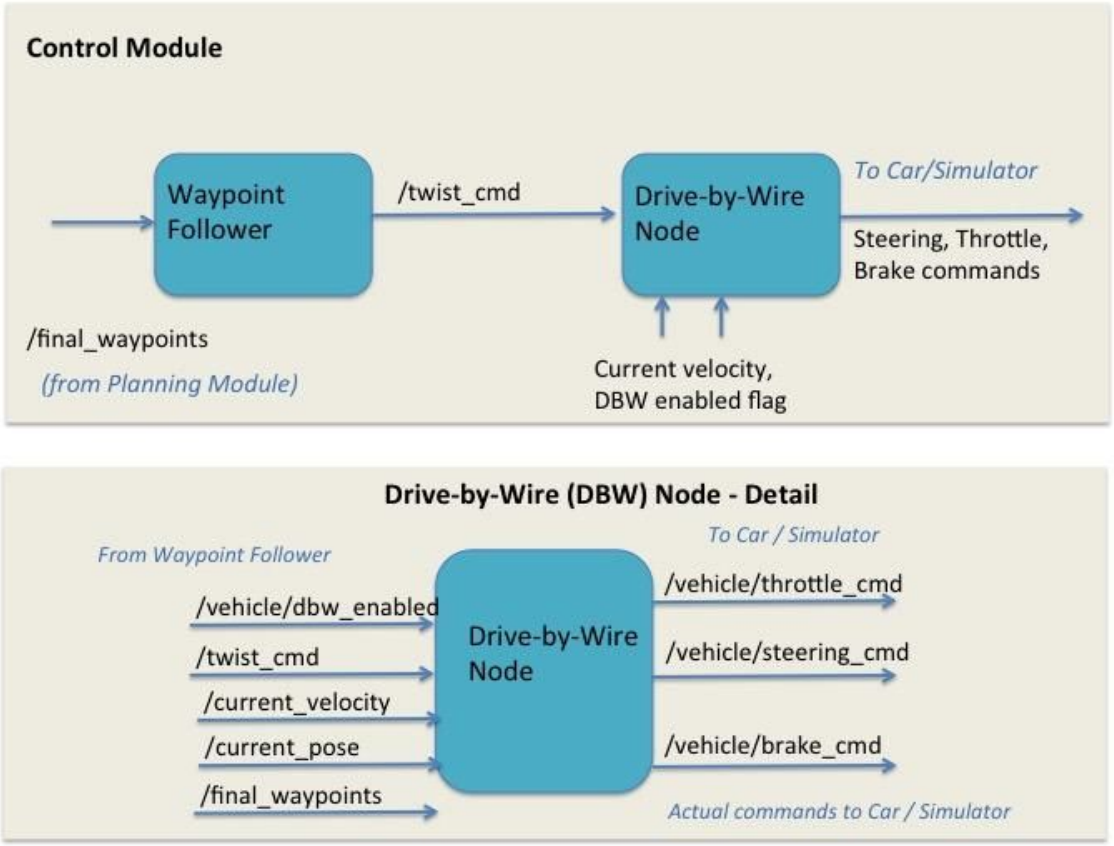


Figure 3 - Control Module

Our project implements the Drive-by-Wire (DBW) node.

The DBW node commands the car or the simulator by providing it with appropriate commands to control the **throttle** (acceleration/deceleration), **brake** and **steering angle**.

The DBW node consumes the messages on the **/twist_cmd** topic, which are emitted by the Waypoint Follower node.

- **/twist_cmd** – contains commands for proposed linear and angular velocities

- **/vehicle/dbw_enabled** – a flag to show if the car is under DBW control or Driver control
- **/current_pose** – contains current position of the car
- **/current_velocity** – contains current velocity of the car
- **/final_waypoints** – contains the list of final waypoints

The DBW node instantiates two controllers - **Longitudinal Controller** and **Lateral Controller** – to calculate the linear and angular velocities for the car / simulator. If the car is under DBW control, the node calculates throttle and brake values using the longitudinal controller. It also calculates the steering angle using the lateral controller. These values {throttle, brake, steer} are then output to the appropriate topics for the simulator or actual car to execute.

The ***longitudinal controller*** takes as input the **target speed** (as mentioned in the **/twist_cmd**), the **current speed** and **time delta**. First, a feedforward control is used in following way. From the *speed error* (target speed minus current speed) a target acceleration is calculated. Here some tuning can be done to get the right feeling for approaching the target speed. The *target accelerations* are limited to *decel_limt* (default -5 m/s²) and *accel limit* (default 1 m/s²). In addition, *jerk* is limited to 5 m/s³. From this target acceleration the necessary force is calculated using the vehicle mass (F = m*a). In addition, the feedforward control contains a *resistance force* depending on the current speed with a polynomial of degree 2. This is used to model e.g. air forces (squared dependant on velocity). It was only tuned in the simulator not using real data. In a real car, it would be an online estimator, but the acceleration torque must be a known quantity. This is available in a real car but not for the simulator.

We use a **PID controller** to achieve a better control performance as closed loop controller. It takes into account an *estimated speed error*, not the actual speed error per se, which also takes into account *acceleration and deceleration limits, and target acceleration*. Otherwise, a too high speed error would be used when the target velocity jumps abruptly.

The PID control parameters **Kp**, **Ki** and **Kd** are just set by empirical observation and test experience, and are not further optimized. They are high in value (multiplied by the vehicle mass) because the output is a force in Newton. At or near standstill, the integral-part of the PID controller is frozen to avoid a wind-up. It can be also reset. This is done **if drive by wire is disabled** (*dbw_enabled* == *false*).

If the resultant force is positive, that value is used for the **throttle command**. The throttle command is between 0 and 1. It represents the acceleration pedal position. A torque would have been a lot better. *It was estimated that throttle position of 1.0 (100%) refers to 3.0 m/s² acceleration.* But measuring from the simulator was not really working. This acceleration is then transformed in a force

and used for normalization. Next problem is that the throttle position is not linear to the acceleration. This is not taken into account (so the PID has to do more).
If the force is negative then the brake torque is calculated using the vehicle mass, the wheel radius and the number of wheel. The brake command is not filled immediately if the force is negative. A deceleration deadband (default 0.1 m/s^2) is taken into account to avoid toggling between throttle and brake.

Similarly, the ***lateral controller*** takes a **target yaw rate**, the **current speed**, **current position**, the **following waypoints,** and calculates the required **steering angle**, while attempting to be within required tolerance limits for minimum and maximum steering angles.

The *lateral controller* also uses a feedforward control to set a good steering angle using the current speed and the target yaw rate. Therefore the wheelbase, the steering ratio, and the maximum allowed lateral acceleration are taken into account by the Ackermann model.

In addition, the following waypoints in a configurable view range (10 m, 15 m also working) are interpolated by a polynomial of 3rd degree in vehicle coordinates. To improve this further, the view range could be made speed dependent. Then the **cross track error (CTE)** is calculated using the distance of the current vehicle position to the generated polynomial trajectory. This CTE is fed into a PID controller to get rid of static control errors and to achieve a closed loop controller. Finally, the output of the PID is added to the steering angle of the feedforward part.

The final commands of the *longitudinal controller* and *lateral controller* are published to the following topics:

- /vehicle/throttle_cmd – the throttle value
- /vehicle/steer_cmd – the steering angle value
- /vehicle/brake_cmd – the brake value

Conclusion and Results

The car performs well under simulated conditions.

At the end of the waypoint list, the maximum allowed velocity from the waypoint loader is ramped down to zero. Only one waypoint has zero velocity, therefore the car sometimes overshoots the end, and other times it stops correctly. If multiple waypoints with zero velocity were set, the car would correctly stop.

A video of the simulator performing correctly is shown here: https://www.youtube.com/watch?v=1KDDv5UTwig&feature=youtu.be