HALO: Fault-Tolerant Safety Architecture For High-Speed Autonomous Racing

***Abstract*—High-speed autonomous racing has become an active area of research in recent years. Full-scale autonomous racing competitions such as RoboRace and the Indy Autonomous Challenge have given researchers an opportunity to develop the software stack for high-performance race vehicles which aim to operate autonomously on the edge of the vehicle's limits. These autonomous race cars can race at speeds greater than 150mph, while overtaking another autonomous race car. In order for the autonomous race car to achieve an acceptable level of safety, the software must actively monitor for erroneous operating conditions, as well as faults and errors during its high-speed operation. Achieving safety means ensuring the absence of unreasonable risks due to hazards caused by a malfunction in any of the software's sub-systems and components. While there exist open-source self-driving stacks such as Autoware and Apollo, there is a lack of a comprehensive examination of the safety aspects of autonomous driving at a programming and functional level. This paper presents the first holistic overview of a safety architecture called HALO which has been implemented on a full-scale fully autonomous racing vehicle as part of the Indy Autonomous Challenge. We present a failure mode and criticality analysis for the software stack that could occur in the perception, planning, control, and communication modules. We then provide safety archetypes and runtime monitoring methods to mitigate these faults. We demonstrate the effectiveness of HALO on data gathered from real-world runs of our autonomous race car.**

I. Introduction

In October 2021, a historic autonomous race was held at the Indianapolis Motor Speedway (home to the Indy 500 race) as part of the Indy Autonomous Challenge (IAC) [1]. Following in the footsteps of the DARPA Grand [2] and Urban Challenges [3] in 2004, and 2007, the IAC invited university teams to participate in a competitive autonomous race and showcase their research and ideas to the general public. This was full-scale, fully-autonomous open-wheel racing. Teams were asked to develop the complete autonomous racing software stack for the standardized racing vehicle Dallara AV-21, a modified Indy Lights vehicle, and race each other using a ruleset comparable to human drivers. Nine university teams from all over the world participated in this race. A subsequent race was held at the Las Vegas Motor Speedway track in January 2022. This time it was a head-to-head autonomous race and teams had to overtake an opponent at racing speeds. All nine race cars were built the same, with no hardware advantage from one car to another, and no hardware modifications were allowed. This was purely a *"battle of algorithms"* and a team's perception, planning, and controlstack determined their competitiveness.

Even though a racing circuit provides a less cluttered, and an ideal environment for the autonomous vehicle (e.g. we only race in good weather conditions), software errors can rapidly lead to situations that cause crashes and severe physical damage to the race cars, resulting in cost-and-time intensive repairs. More complex software always goes hand-in-hand with increasing difficulties in safeguarding and verification. Minor errors in assumptions about the vehicle's behavior or its physical capabilities can easily lead to a spin-out and a crash with the track barriers, or in the worst-case with another vehicle. For building a fast fully-autonomous software stack, teams need the ability to detect software issues both at the design time, but more importantly at runtime as well.

In Formula One racing, a Halo is a protective barrier titanium ring around the driver that helps to prevent large objects and debris from entering the cockpit of a single-seat racing car, thereby keeping the driver safe at all time. In this paper we present HALO - a fault-tolerant safety architecture that was implemented on a fully-autonomous full-scale racing vehicle as part of the Indy Autonomous Challenge. The overall safety goal of HALO is to not be responsible for any collisions. To this end, various measures are taken to keep the autonomous vehicle in safe operation at full performance capabilities. We first examine and identify, in detail, a variety of safety-critical faults that could occur in the perception, planning, control, and communication modules of the autonomous racing software stack. We then provide several examples of safety archetypes to mitigate these faults. These runtime monitoring archetypes are not specific to our code implementation but could be used to handle failures in other autonomous vehicles.

This paper has three main contributions:

1) With this paper, we provide the first holistic overview of the software safety architecture for high-speed autonomous racing. Several of the safety features could be translated into regular autonomous driving without loss of generality. This is the first such paper to present such a deep-dive into fault-tolerance and failure modes for an autonomous racing (driving) stack.
2) We conduct a failure mode, effects, and criticality analysis (FMECA) for the autonomous racing software stack by reviewing software modules, their components, assemblies, and subsystems to identify potential failure modes, their causes and effects, and their criticality.
3) Our HALO framework, identifies common failure modes and proposes safety node archetypes for runtime monitoring that can mitigate data health, node health, and behavioral-safety faults that could occur at a sensor level, algorithmic level, or at a system level on the vehicle.

We demonstrate the effectiveness of the HALO framework using ROS data collected from real-world race car runs during the Indy Autonomous Challenge. We describe the failure modes in a comprehensive manner and provide runtime monitoring architectures for handling these failure modes.

II. Related Work

The current landscape of safety for fully autonomous vehicles is vast and spans across a wide spectrum. Here we focus on work that closely aligns with fault-tolerance and safety architectures at code or functional level. There exist several open-source self-driving stacks, such as Baidu Apollo Open Platform [4], openpilot [5], and Autoware Auto [6]. While these provide the functionality for vehicles to drive autonomously, they do not focus on fault-tolerance in great detail. Safety features provided by Autoware only consider monitor computing resources during runtime but does not check on the health of the code or data [7].

Guardauto [8] is a system which proposes a runtime safety method for self-driving stacks. The Guardauto system implements safety features to isolate components for protection, detect anomalies, and reject anomalous data. While the Guardauto system considers defenses to prevent code failures and data failures, it does not discuss what the vehicle should do in the event of a failure during operation. Research into fault tolerance for automobiles has examined detecting faulty sensors [9], [10]. In these approaches, real-time measurements are compared against test statistics to detect anomalies.

Several techniques used for fault detection and isolation are based on methods used more generally for cyber-physical systems. Noteworthy contributions include [11], which uses a model-free approach for fault detection in large-scale cyber-physical systems such as smart buildings. [12] studies fault-tolerant control systems, distinguishing between passive or active systems. Passive systems are those for which it is known in advance what faults are possible, while active systems must actively monitor their subsystems for faults and mitigate them during runtime. [13] proposes a deep learning approach to fault detection in sensor data, using an LSTM model.

In addition to the faults discussed above, the Robot Operating System (ROS) used by open-source driving stacks has also been studied from a fault detection and isolation perspective. ROS1 uses an architecture, with a single ROS master that acts as a registration and lookup service for nodes, parameters, and services. [14] provides fault tolerance of the ROS by providing a backup system for the central point of failure - i.e. the ROS master. ROS2 provides a more robust architecture that does not rely on a single point of failure, but still has known issues with timing and messaging [15]. [16] attempts to detect anomalies by analyzing the timing of ROS2 messages in a system - a sudden change in the publishing rate of a message type is indicative of a fault. [17] studies the adherence of an autonomous driving framework to ISO 26262 software standard. In [18] authors propose a graceful degradation design process to improve the automated driving continuation rate by analyzing tasks in the order of system-level, ECU-level, and microcontroller-level degradation.

There is a noticeable lack of a comprehensive fault modes, errors, and criticality analysis for a real fully-autonomous autonomous vehicle software implementation. We believe our paper addresses this gap, by presenting sensor level, algorithmic level, and system level faults and mitigating them.

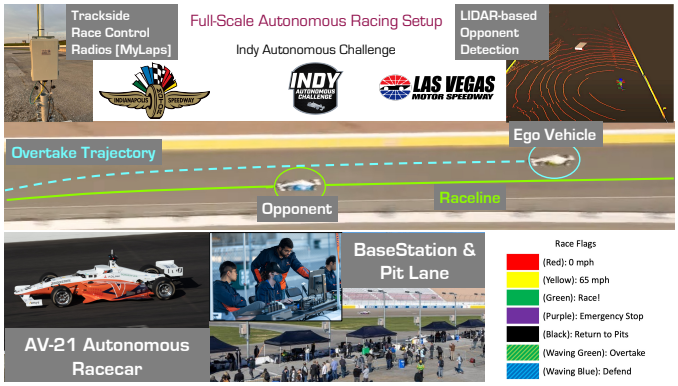

Fig. 1. An overview of the autonomous racing setup in the Indy Autonomous Challenge held at the Indianapolis Motor Speedway and the Las Vegas Motor Speedway. We implemented the HALO safety architecture on a full-scale fully autonomous AV-21 race car.

III. Autonomous Racing Setup

The Indy Autonomous Challenge features nine university teams, each using an AV-21 race car, which is an Indy lights car modified to drive autonomously. The AV-21 race car is retrofitted with a set of sensors and actuators: four RADAR sensors, six cameras and three LiDAR sensors. Each of the sensor modalities covers a 360-degree field-of-view around the vehicle. Furthermore, the vehicle is equipped with dual high-precision Global Navigation Satellite System (GNSS) receivers. This allows the vehicle to reach speeds of up to 175 mph around the oval race circuits - Indianapolis Motor Speedway (IMS) and the Las Vegas Motor Speedway (LVMS).

Figure 1 shows an overview of the autonomous racing setup and systems required for the race. It also outlines a few definitions that we will use throughout this paper. There are two competition modes - single-vehicle time-trials, and head-to-head racing. For the time-trial format, the goal is to go as fast as possible and put up the fastest lap time. For the head-to-head competition, the rules dictate to demonstrate the ability to overtake an opponent at a given speed, which incrementally increases in each lap. For overtaking, the Ego race car must maintain a longitudinal separation from the Opponent race car. This required safe distance is 30 meters for the competition.

At the track, there are radio systems installed in place to help conduct the race and aid with the overall safety of the competition. The racetrack is equipped with a network of mesh radios that enable teams to connect and communicate with the race car from their Pit Box, while the car is on the track. Teams have a Base Station computer in their pits which is able to obtain real-time telemetry data from the car, as well as issue safety-critical commands, in case of an emergency. Teams are forbidden to send any active control commands during the race. The race car is running autonomously *on-its-own*.

A very important part of the race is adherence to Race Flags - which are relayed to the race car as radio signals over the mesh network. The race flags dictate constraints on the actions the race car can perform and also help communicate track status to the vehicle. This race control system is called MyLaps and it is also responsible for measuring the lap time and vehicle speeds during the autonomous race. Depicted in

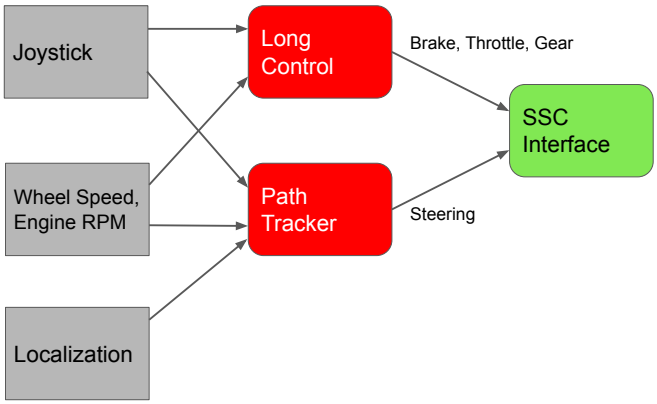

Fig. 2. The control module, shown in red, uses the wheel speed, engine RPM, and vehicle localization to generate actuator commands to drive the vehicle.

Figure 1, are the definitions of all the race flags. The Red flag indicates that the vehicle must come to an immediate stop. The Yellow flag implies caution, and directs the vehicle to remain below 65 mph on the track, or below 35 mph in the pit lane. The Black flag means that the vehicle must return back to the pits from the track. A Purple flag is issued to shut off the engine of the race car. This is used during an emergency or a crash/spin to prevent damage to the vehicle. Teams' software must respond to any flags issued by race control at all times, and must obey the flag constraints. The waving green and waving blue flags are used during overtaking maneuvers and indicate who is the attacker and the defender.

Our autonomous race car's software has five safety-critical modules. These modules contain ROS2 nodes, which communicate over a standard Data Distribution Service (DDS) middleware. When several of these interconnected nodes are responsible for critical components of a vehicle's operation, it is important to monitor not only the expected behavior of each node individually, but also their holistic interaction.

In Sections IV, V, VI, VII, we present a deep-dive of each of the modules, identifying the failure modes, and the criticality of failures. Section VIII presents the safety archetypes and overall fault analysis, followed by a description of the HALO safety architecture in Section IX.

IV. Control Module

Role: The control module is responsible for controlling the steering, acceleration, and gearbox of the AV-21 race car as well as to follow the race (flag) conditions at all times. Shown in Figure 2, the control module consists of two nodes: Longitudinal (Long) Control and Lateral Control (Path Tracker).

Inputs and Outputs: The control module takes the following signals as input (with ROS topic names included):

1) Wheel Speed: This measurement is made available by the Raptor Drive-By-Wire ROS Signal: `/raptor_dbw_interface/wheel_speed_report`
2) Engine RPM: `/raptor_dbw_interface/pt_report`
3) Vehicle Localization: `/vehicle/uva_odometry`
4) Joystick Commands: `/joystick/command`

The control module uses these inputs to determine what speed the car should go at, what the steering angle should be, and what gear the vehicle needs to be in. As output, the control module produces actuator commands for the accelerator, brake, steering column, and gear box:

1) Accelerator actuator command: `/joystick/accelerator_cmd`
2) Brake actuator command: `/joystick/brake_cmd`
3) Steering column actuator command:`/joystick/steering_cmd`
4) Gear shifting command: `/joystick/gear_cmd`

Safety Concerns: The primary safety concern of the control module is to prevent the vehicle from crashing. Examples of such unsafe behavior is attempting to turn too sharply or braking too aggressively at high speeds. We next describe the operation of the all the nodes within the control module.

A. Longitudinal Control

The Long Control Node is responsible for controlling the vehicle's accelerator, brake, and gearbox to maintain a desired velocity. This is referred to as longitudinal control for an autonomous vehicle.

Using the vehicle's current velocity derived from the wheel speed, the Long Control Node uses two PID controllers to bring the vehicle to the desired velocity. One PID controller controls the accelerator and the other controls the brake. The outputs of these controllers represent the amount that the accelerator and brake should be applied, and are sent to the Steering and Speed Control (SSC) Interface, which will pass the data to the Raptor Drive-By-Wire system. In addition to accelerator and brake commands, the Long Control Node also controls the gearbox. Gear shifting is calculated based on the engine RPM and the velocity of the race car, and sent as an integer value to the SSC Interface.

Safety Concerns: The Long Control Node has several in-built safety features. The first safety feature is a check that prevents the node from publishing an accelerator and a brake command simultaneously. This prevents drivetrain damage by trying to apply both the brake and the accelerator at the same time. The second safety feature is a check on the data limits of the accelerator and brake. If either of these commands exceed their acceptable range the node reduces the value to the maximum allowed value. Similarly, if either of these commands fall below zero the node will set the value to zero. The third safety feature is an emergency override from the base station. During testing on the track, the joystick operator in the pit lane can override the accelerator and braking commands. Additionally, the base station operator can make an emergency stop by pressing two bumpers on the joystick simultaneously. This causes an immediate engine shutdown on the vehicle.

B. Lateral Control - Path Tracker

The Path Tracker Node is responsible for controlling the vehicle's steering column to follow a reference path. A path is defined using a sequence of waypoints in a local north-east-down (NED) coordinate frame.

Path Tracker node takes as input the wheel speed, the vehicle's position, and the joystick commands from the base station. The path for the node to follow is an internal state of this node. The Path Tracker node uses three paths - "raceline", which takes the inner line throughout the race track; "overtake," which takes the outer line of the race track for the purposes of overtaking an opponent; and "pits," which enters the pit lane from the inner line of the race track, goes through the pits, and then merges back onto the inner line. Each path consists of equally spaced waypoints defined in a local NED coordinate frame relative to a shared origin point.

The Path Tracker Node uses an Ackermann-adjusted pure-pursuit algorithm [19] to follow the chosen path. A Model Predictive Control [20] approach could also be used in lieu of the pure-pursuit algorithm. The node finds the waypoint on the path that is closest laterally to the current position of the vehicle. From this waypoint, the node then uses a dynamic lookahead distance to find a waypoint ahead of the vehicle. The lookahead distance is computed as a function of the vehicle's velocity. Up to 35 mph, the lookahead distance is set at 15 meters. From 35 mph to 100 mph, the lookahead distance increases linearly from 15 meters to 50 meters. Over 100 mph the lookahead distance is capped at 50 meters.

Safety Concerns: The Path Tracker Node has several safety features built into it. The first safety feature is a check on the data limits of the steering angle. If the command exceeds the maximum angle in either direction, the node reduces the value to be within the acceptable range. The second safety feature is a dynamic adjustment of the maximum steering angle, calculated as a function of the vehicle's velocity. Below 35 mph the vehicle's steering column has a range from 24 degrees in either direction. From 35 mph to 100 mph, the steering range decreases linearly from 24 degrees to 10 degrees. Over 100 mph the steering column has a set range of 10 degrees in either direction. This dynamic adjustment of the maximum steering angle prevents the vehicle from attempting to turn too sharply at high speeds, which could result in losing control of the vehicle. The third safety feature is to check if a valid path is being published at all time. Especially while switching to a new path, the node verifies that the requested path exists.

V. Localization Module

Role: The goal of the localization module is to provide an accurate estimate of the vehicle's position on the race track. Shown in Figure 3, the localization module consists of six nodes: Top GNSS, Bottom GNSS, Top Map-Baselink, Bottom Map-Baselink, EKF, and Topic Multiplexer.

Inputs and Outputs: The localization module takes the following signals as input:

1) Global Navigation Satellite System (GNSS) data from the two Novatel units in the form of latitude, longitude, latitude standard deviation, and longitude standard deviation, published at a rate of 20 Hz: `/novatel_top/bestpos` and `/novatel_bottom/bestpos`
2) Inertial Measurement Unit (IMU) data from the two Novatel units, published at a rate of 125 Hz: `/novatel_top/rawimu` and `/novatel_bottom/rawimu`
3) Wheel Speed: `/raptor_dbw_interface/wheel_speed_report`

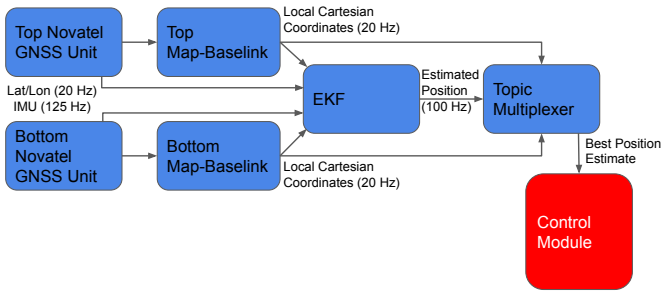

Fig. 3. The localization module, shown in blue, fuses several localization sources to generate a single best position estimate for the vehicle.

It uses these inputs to determine the location of the vehicle in a local Cartesian (x,y) coordinate frame, published on `/vehicle/uva_odometry`.

Safety Concerns: The safety concerns of the localization module are the data rate and data accuracy. Data rate is a concern because when driving at 150 mph, the vehicle will travel over 10 feet in between successive GNSS readings, essentially driving blindly. Low accuracy data is also a concern, because inaccurate localization could cause the vehicle to crash. We next discuss the localization methodology for the vehicle.

A. Localization Methodology - Sensor Fusion

The Map-Baselink Node takes the latitude and longitude values reported by the GNSS units and converts them into an x and y position in a local north-east-down (NED) coordinate frame. In addition to the local (x,y) position, the Map-Baselink Node also computes θ, the heading of the vehicle using hysteresis. The node computes the change in position from one GNSS measurement to the next, and reports that angle as the heading of the vehicle. There is a separate Top Map-Baselink Node and Bottom Map-Baselink Node for the top and bottom GNSS units, and they publish on the topics `/novatel_top/dyntf_odom` and `/novatel_bottom/dyntf_odom` respectively.

The GNSS localization process occurs at the slower rate of 20 Hz. The GNSS positional data is fused with linear acceleration and angular velocity measurements using an Extended Kalman Filter (EKF) [21]. Fusing the 20 Hz GNSS location data with 125 Hz Inertial Measurement Unit (IMU) data and 100 Hz velocity data, we can obtain a steady 100 Hz localization estimate for the race car.

B. Localization Redundancy - Topic Multiplexer

The localization process can produce position estimates that do not accurately reflect the position of the vehicle. Additionally, one GNSS unit can be more accurate than the other. It is the job of the Topic Multiplexer Node to determine which source of localization is the most accurate, and report those values as the position of the vehicle on the

topic $/vehicle/uva\_odom$. When determining which source of localization to use, the Topic Multiplexer Node always prefers the 100 Hz EKF localization over the 20 Hz GNSS localization. The node monitors the covariance of the EKF localization estimation and, if the covariance is greater than an acceptable threshold, the 100 HZ data is considered imprecise and the topic multiplexer defaults to reporting the slower 20 Hz GNSS localization. When reporting GNSS localization, the topic multiplexer prioritizes the higher accuracy output between the two GNSS systems. Slowing down to a slower (but accurate) GNSS is better than inaccurate (but faster) EKF estimates. When this switch in the source occurs, the race car also slows down appropriately since the localization data frequency has reduced by a factor of 5.

VI. Communication Module

Role: The goal of the communication module is to provide sensor readings from the low level hardware (Raptor Drive-By-Wire (DBW) system) to the vehicle's software stack and send actuator commands from the software down to the low level hardware. In addition, the race car needs to maintain a radio link with the base station in the Pits and with with the MyLaps race control flags system. Shown in Figure 4, the communication module consists of five nodes: Raptor DBW, Steering and Speed Control (SSC) Interface, Joystick Vehicle Interface, Race Flag Input, and Telemetry.

Inputs and Outputs: The communication module takes the following signals as input:

1) Vehicle ECU data from the CANBus
2) Race flags from the CANBus
3) Joystick commands from the base station
4) Actuator commands for the Raptor DBW system: `/joystick/accelerator_cmd`, `/joystick/brake_cmd`, `/joystick/steering_cmd`, and `/joystick/gear_cmd`

The communication module provides two outputs. First, it converts the actuator commands into bytes to be put on the CANBus for the Raptor DBW system. Second, it takes the inputs from the CANBus and the base station, and converts them into ROS2 topics for the other modules to use:

1) Wheel Speed: `/raptor_dbw_interface /wheel_speed_report`
2) Engine RPM: `/raptor_dbw_interface /pt_report`
3) Race flags: `/raptor_dbw_interface/rc_to_ct`
4) Joystick commands: `/joystick/command`

Safety Concerns: The primary safety concern of the communication module is a loss of communication. A loss of communication with the CANBus means no sensor data for the software stack, but more critically it also implies that the control module cannot send commands to the Raptor DBW system to control/actuate the vehicle. A loss of communication with the base station presents a danger to the vehicle because it removes the base station operator's ability to intervene in the case of an emergency. Similarly a loss of communication with the MyLaps race control prevents the vehicle from being directed autonomously through flag signaling.

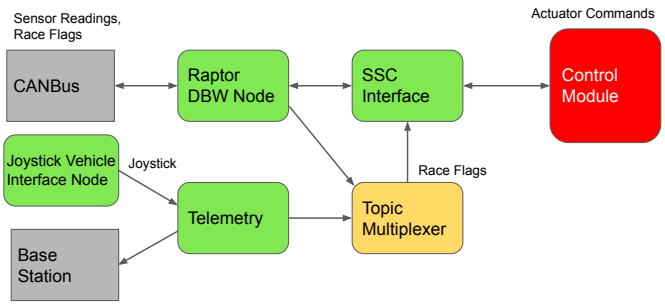

Fig. 4. The communication module, shown in green, is responsible for communicating with the CANBus, race control, and base station

A. Communication With Sensors

The Raptor DBW Node in the communication module is responsible for sending control commands from the control module to the actuators. The Raptor DBW Node subscribes to the control commands coming from the SSC Interface, and converts them into binary vectors to be published on the CANBus. The Raptor DBW Node also provides sensor data off of the CANBus, and converts it into the a ROS2 message. This data includes information about the vehicle's engine, published on the topic `/raptor_dbw_interface/pt_report`, information about the wheel speed, published on the topic `/raptor_dbw_interface/wheel_speed_report`, and information regarding diagnostics of the low-level control system, published on the topic `/raptor_dbw_interface/diagnostic_report`. The Raptor DBW Node also listens to data coming from the MyLaps race control. The data from race control is sent to a radio that publishes the data onto the CANBus. The Raptor DBW Node converts this data into a ROS2 message.

B. Communication With Base Station

The Telemetry node is responsible for sending data from the vehicle to the base station. The data sent to the base station includes the engine temperature, the localization accuracy, the vehicle's current and desired velocity, the lateral error from the vehicle to the set path, the race flag, the vehicle's position, and the reason for the most recent emergency stop (if any). This data is published at 1 Hz to reduce the network bandwidth between the vehicle and the base station.

The Joystick Vehicle Interface Node is responsible for sending joystick commands from the base station to the vehicle. This node listens for commands coming from the joystick attached to the base station, and relays them to the vehicle on the topic `/joystick/command`.

Safety Concerns: To address a loss of communication with the CANBus, the Raptor DBW Node produces a heartbeat to test communications with the sensors and actuators. If the low-level hardware does not receive this heartbeat, the engine will shut off to prevent a loss of control of the vehicle. To address a loss of communication with the base station, the joystick commands include a heartbeat. If the vehicle does not receive this heartbeat for some time, the vehicle will be brought to a stop. This is discussed in detail in Section IX-A.

VII. Perception Module

Role: The goal of the perception module is to detect and track obstacles. These could be static (air-filled pylons) or dynamic (opponent race car). Our software uses LiDAR data for perception. The race car contains cameras and RADAR as well and these could also be integrated into the pipeline. The LiDAR-based perception consists of five nodes: Point Cloud Filtering, Ground Detection, Wall Detection, Euclidean Clustering, and Object Detection.

Inputs and Outputs: The perception module takes LiDAR point clouds and vehicle localization as inputs. The perception module provides the following outputs:

1) The locations of obstacles on the track: `/lidar_bounding_boxes`
2) The distances of the obstacles: `/telemetry /bb_distance`
3) The headings of the obstacles: `/telemetry /bounding_box_array`

Safety Concerns: The primary safety concern of the perception module is the presence of False Positives, and False Negative detection. A false positive is the perception stack seeing an obstacle that does not exist, and a false negative is not seeing an obstacle that is present. False negatives present a highly critical failure mode as the vehicle might not act to avoid a collision with an obstacle, not detected ahead of time. False positive detection can lead to erratic behavior as the vehicle attempts to avoid a non-existent obstacle. We next discuss the perception pipeline in detail.

A. Filtering

The perception module begins with raw point-cloud data from each of the three LiDARs. These point clouds contain dense collections of points, so the first step in the pipeline is to filter the point clouds down to fewer points. The Filtering Node starts with range-based filtering to ignore points that are too far away to be relevant, or are too close and likely belong to the ego vehicle itself. The filtering node also uses angle-based filtering to remove points that are too high above the vehicle and unnecessary for driving on a flat race track. In addition to these filters, the perception module also uses ground plane detection [22] to separate the points into ground points and non-ground points. The final filter used by the perception module detects the track boundaries, so that the vehicle can filter out all points that are outside of the race track. This allows the vehicle to ignore noise caused by grass on the interior of the track, and obstacles detected in the stands.

B. Object Detection and Tracking

At this point, the only remaining points are those that belong to obstacles on the track. Euclidean clustering [23] is performed on these points, to cluster them together into objects. Any cluster containing a critical mass of points are detected as an object, and are fitted to a bounding box to get the size and orientation of the object. The distance to the obstacle is calculated as the distance from the baselink of the ego vehicle to the centroid of the bounding box.

Safety Concerns: As mentioned earlier, the primary safety concern for the perception module is false negative and false positive detection from the LiDARs. After the Euclidean Clustering algorithm, only clusters containing a enough points to exceed a preset threshold is considered to be an object. If this threshold is set too low, the perception pipeline will detect non-existent obstacles as being present, usually from ground points that get misclassified as non-ground. If a non-existent obstacle is detected, the vehicle may swerve to avoid an obstacle that only appears for a fraction of a second. Alternatively, if the object detection threshold is set too high, the perception pipeline may not see a vehicle on track if not enough points pass through the filter to the clustering algorithm. In this case, the ego vehicle could make an early attempt to complete an overtake, thinking it has passed the opponent vehicle. When a middle-ground threshold is chosen for obstacle detection, these problems are rarer but either can still occur. In Section IX we will discuss a behavioral safety node that can handle this situation.

VIII. Safety Analysis and Failure Modes

The overall safety goal of the vehicle is to prevent any crashes. The safety requirements are partially codified in the form of the race flag rules defined in Section III that must be obeyed at all times. In addition to the flag rules, there are other failure modes that pose a risk to the safe operation of the ego vehicle and, therefore, must be handled appropriately by our HALO safety framework. A piece of code crashing means losing some functionality on the vehicle, and presents danger based on the severity of the code failure. For instance, losing the ability to precisely localize, whether because of an inaccurate localization estimate, or because of a lack of localization data, could lead to a catastrophic failure, wherein the vehicle can no longer maintain the desired path. A loss of communications from race control or the base station hinders the ability to bring the vehicle to a stop or shut down the engine in case of an incident on track.

A. Failure Modes

Based on the safety considerations and flow of information discussed for the control, localization, communication, and perception modules in our software stack, we conducted a failure mode analysis to categorize the failures into the following three types:

1) Node Health Faults: A node health fault is a system-level fault that occurs when a safety-critical process (ROS node) crashes, stalls, or otherwise becomes unable to perform its job. In our safety stack, we consider the control module as being the most critical for safe operation of the vehicle. The Long Control Node (Section IV-A) controls acceleration and braking while the Path Tracker Node (Section IV-B) controls steering, and a failure in either is catastrophic, as it directly affects the ability of the vehicle to navigate safely. The SSC Interface Node is responsible for communication with the lower level Raptor DBW hardware, and a failure in this node makes it impossible to send control commands to the

vehicle. Consequentially, monitoring the health and *"liveness"* for safety-critical nodes is a necessity.

2) Data Health Faults: Data health can be defined as a combination of data quality, and data rate. A data health fault can be thought of as a sensor-level fault that may occur when data reported from a vehicle sensor is inaccurate or updated data is not being reported in a timely manner. As an example, consider a critical sensor - such as the GNSS. An inaccurate report of the position of the vehicle can induce errors in our localization estimate, and cause the control module to drift away from the desired path. Likewise, slower GNSS data may cause the vehicle to overcorrect the steering after the vehicle drifts off of its defined path. Data health faults also include limit faults - sending or receiving data that is outside of the expected range, such as attempting to send a steering angle to the actuator that falls outside of the actuator's physical range. These also can include Typeset faults - i.e. publishing an incorrect datatype message on a ROS topic could in fact cause the ROS node to crash. Another example of a data fault for our stack is failure to receive race flag updates from race control, or failing to receive updates from the base station. These periodic *pings* from MyLaps race control and the Pits base station are needed to maintain the ability to bring the vehicle to an emergency stop if necessary.

3) Behavioral-Safety Faults A behavioral-safety fault is an algorithmic-level fault that occurs when the vehicle behaves in an unsafe manner due to poor programming, incorrect logic, or bad implementation of an algorithm or a routine. In the context of our autonomous racing setup, we consider two critical behavioral-safety faults. The first is failure to obey race flags. The second behavioral-safety fault is failure to provide adequate separation when overtaking another vehicle. Recall that after completing an overtake, the leading vehicle must leave a longitudinal separation of at least 30m before "*closing-the-door*" on the opponent and merging back to the raceline. Both requirements are described in detail in Section III.

B. Safety Archetypes and Monitoring Nodes

Each type of safety fault needs to be handled by some node to have an assurance of vehicle safety on the track. A team's ability and comfort level in sending their race cars to the track full-autonomously and unsupervised is directly a function of their confidence in their ability to handle these faults. Node health faults can be handled by monitoring that the safety-critical nodes are still operating and producing data at the expected rate. Similar to a watchdog monitor, this can be achieved by having a node produce a heartbeat signal for the monitor, and the monitor continuously checking the heartbeat as evidence of the node still being operational. One version of the heartbeat is in the form of a boolean flag, published at a set rate. Algorithm 1 gives an example of a Node-health monitor that operates by detecting boolean flag heartbeats. In this algorithm, the node uses a timer to compare the current time against the last time a heartbeat was received. If the difference between these two times is over a set threshold (defined as *safety_threshold*), then the node stops the vehicle. Another variant of heartbeat detection is to have the heartbeat be an integer that increments with each beat. This allows the monitor to detect which heartbeats and how many were not received by recording the value of each heartbeat as it arrives.

Algorithm 1: Node Health Monitoring

```
Function heartbeatCallback(hearbeat_msg):
    heartbeat_value = heartbeat_msg.data
    heartbeat_time = now()
Function timerCallback():
    time_since_heartbeat = now() - heartbeat_time
    if time_since_heartbeat > safety_threshold then
        stop_vehicle = true
        publish(stop_vehicle)
    end
```

Algorithm 2: Data Health Monitoring

```
Function receiveDataCallback(data_msg):
    data_value = data_msg.data
    data_accuracy = data_msg.stddev
    if data_value < lower_bound or data_value >
     upper_bound then
        return
    end
    else if data_accuracy > accuracy_threshold then
        return
    end
    else
        main_routine(data_value)
    end
```

Data health faults are handled by monitoring the accuracy and frequency of the data. The frequency of the data can be checked in a manner similar to detecting node heartbeats, by keeping track of the time since the data message was received. The accuracy of the data can be monitored in several ways. If the data provides an accuracy estimate of its own (for instance the GNSS data), such as standard deviation, that value can be monitored directly. If accuracy is not reported in the data, a routine can be implemented to track the previous values of the reported data, and detect when an unexpected change occurs. Another alternative is to compare the data from multiple sensor sources. The data health monitor can also implement a check on the limits of the data, and reject any data that falls outside of the expected range. Algorithm 2 shows an example of a data health monitor based on accuracy and data limit checking. The algorithm checks if the data (*data_value*) is outside of the defined bounds (*lower_bound* and *upper_bound*). It also checks if the accuracy (*data_accuracy*) is above the defined threshold (*accuracy_threshold*). Only if the data is healthy does the node use the data in its main routine.

Behavioral-safety faults are handled by having a monitor for the MyLaps race flag compliance, and obstacle detection and tracking. This monitor needs to set the desired velocity according to the race flag. While doing so, it is important not to continuously send desired velocity commands, as that would prevent any other safety-node from changing the desired velocity of the vehicle. The monitor should only set the speed

when the flag changes colors, to ensure that the speed only gets set once for each flag. A example algorithm for this flag logic is shown in Algorithm 3. The algorithm checks the new flag against the previous flag, and only if the flag has changed to a new color does it publish a new desired velocity. In addition to speed changes, the behavioral node should monitor obstacles to avoid a crash. The obstacle monitoring must ensure that false positive and false negative detection does not cause the vehicle to behave erratically, as described in Section VII.

Algorithm 3: Behavioral-Safety Monitor: Race Flags

```
Function flagCallback(flag_msg):
    prev_flag = current_flag
    current_flag = flag_msg
    if current_flag == red and prev_flag != red then
        desired_velocity = red_flag_speed
        publish(desired_velocity)
    end
    else if current_flag == yellow and prev_flag !=
     yellow then
        desired_velocity = yellow_flag_speed
        publish(desired_velocity)
    end
    else if current_flag == green and prev_flag !=
     green then
        desired_velocity = green_flag_speed
        publish(desired_velocity)
    end
```

IX. HALO Safety Architecture

We now present the HALO safety architecture for our software stack. HALO comprises of a combination of nodes working alongside the main software stack, which ensure that the race car operates safely at all times. The composition of HALO is derived based on the failure mode and criticality analysis, as well as the safety archetypes presented in Section VIII. While we present HALO within the context of our implementation for a high-speed AV-21 race car, several of the HALO nodes can be generalized to other autonomous driving stacks. The HALO safety architecture on our vehicle consists of four nodes, each of which are described next.

A. HALO Node I: Graceful Stop

The Graceful Stop Node's role is to bring the vehicle to a stop in a safe and controlled manner. This is different from an emergency stop which immediately locks the brakes and shuts off the vehicle's engine. An emergency stop is very risky and a considered as a last resort to stop the vehicle. Locking the brakes at high-speeds can cause the vehicle to immediately lose traction and spin out. Instead, the Graceful Stop Node brings the vehicle to a stop in a more gradual manner. When the Graceful Stop Node detects that the vehicle needs to be brought to a stop, it sends a boolean data flag to the Long Control Node (Section IV-A - responsible for maintaining a desired velocity). The Long Control Node, upon receiving this boolean data flag, sets the desired speed of the vehicle to zero,

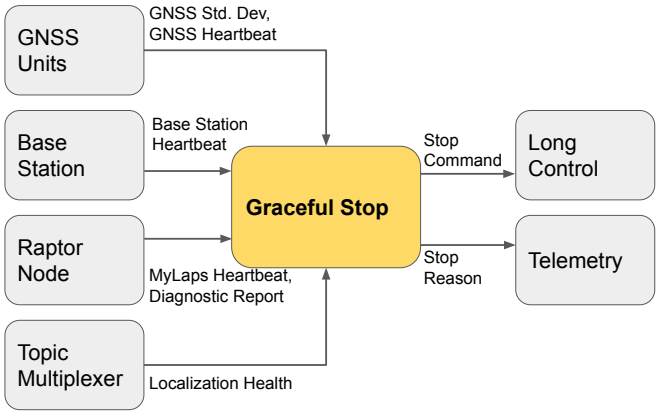

Fig. 5. The Graceful Stop Node provides data health monitoring, and brings the vehicle to a gradual stop upon detecting a fault.

and thus brings the vehicle to a gradual safe stop. In addition to setting the desired velocity, the Long Control Node also sets an internal flag - as long as this internal flag is in effect, the Long Control Node will not change the desired speed from zero. To remove this flag, the long control node must not receive a stop flag from graceful stop for a set amount time (in our architecture, 5 seconds). This behavior prevents the race car from unexpectedly exiting the graceful stop mode.

In addition to implementing the graceful stop routine, the Graceful Stop Node also applies data health monitoring on critical data, which can be seen in Figure 5. If any of these checks is failed, the Graceful Stop Node will bring the vehicle to a stop. One such check is whether the MyLaps race flag data has not been received in some time (25 seconds). This may occur if the MyLaps system has an outage in some sector of the track. Similarly, the Graceful Stop Node implements a check on whether the base station heartbeat data has not been received in some time (10 seconds). The Graceful Stop Node also checks if the GNSS data has not arrived for some time (500 milliseconds). Additionally, the graceful stop node checks the accuracy of the GNSS data using the latitude's and longitude's standard deviation reported as part of the GNSS data. Both of these values must be below a set threshold (35 centimeters) to continue driving. The final data check is on a diagnostic report from the low-level hardware. This diagnostic report indicates an issue with the low-level hardware, in which case, the Graceful Stop Node brings the vehicle to a stop.

The data faults checked by the Graceful Stop Node do not all have the same level of severity, as shown in Figure 6. The vehicle may continue operation after a base station heartbeat loss or a MyLaps timeout by restoring communication with the base station or race control respectively. Similarly, the vehicle may recover from a GNSS dropout by restoring communication with the satellites. A high GNSS standard deviation can be solved by restoring communication with the RTK correction service. The most severe fault checked by the Graceful Stop Node is the diagnostic report errors. In the event of a diagnostic report error it is advised to abandon the run and bring the vehicle back to the pits for more in-depth diagnosis.

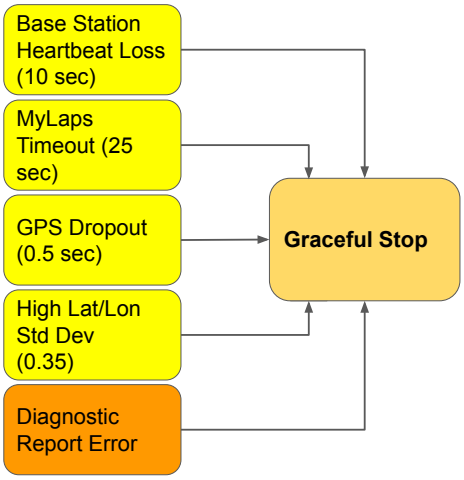

Fig. 6. The severity of faults occurring in the graceful stop node. Yellow represents recoverable faults, while orange represents non-recoverable faults.

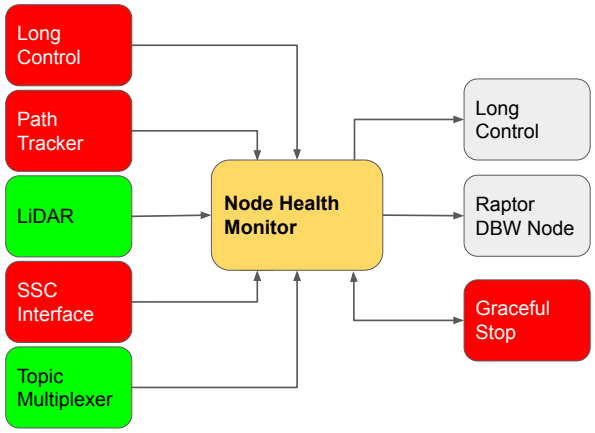

Fig. 7. The severity of a node health fault occurring. Red indicates a node crash that leads to an unrecoverable emergency stop, while green represents a node crash that may be able to be ignored depending on the circumstances. Grey nodes are those used to perform an emergency stop if a graceful stop is not possible

B. HALO Node II: Node Health Monitor

The Node Health Monitor node is responsible for handling node health faults in our stack. This node monitors the safety-critical nodes in our stack, which can be seen in Figure 7. The nodes that are considered safety-critical in our architecture are Long Control, Path Tracker, LiDAR, SSC Interface, Topic Multiplexer, and Graceful Stop. Depending on the particular node that failed, the node health monitor is capable of stopping the vehicle in different ways. The preferred way of stopping the vehicle is to send a stop request to the Graceful Stop Node. For instance, this is the procedure in case the Path Tracker Node dies. However, the graceful stop routine itself relies on several safety-critical nodes being operational in order to bring the vehicle to a stop. If any of the nodes that the graceful stop routine relies on has died, the Node Health Monitor must stop the vehicle in another way. For example, if the Graceful Stop Node has died, the Node Health Monitor will mimic the Graceful Stop Node's behavior by publishing a desired velocity of zero to the Long Control Node, allowing the Long Control Node to bring the vehicle to a stop. In the event that the Long Control Node has failed, then it will be unable to control the brakes to stop the vehicle. In this case, the Node Health Monitor will instead publish a moderate brake command of its own to the SSC Interface to bring the vehicle to a stop. If the SSC Interface has also died, then the Node Health Monitor can no longer communicate control commands to the low level actuators. In this case there is no way to stop the vehicle in a graceful manner, and the only recourse is to stop the vehicle by shutting off the engine (Purple race flag, or through the base station joystick).

In addition to stopping the vehicle in response to a node failure, the Node Health Monitor can provide other functionality as a response. For example, if the Topic Multiplexer (Section IX-C) dies, the Node Health Monitor will assume the duties of the Topic Multiplexer by publishing the odometry data coming from the GNSS units. Another case of a node crash that does not stop the vehicle is the LiDAR Node. In the event that the LiDAR data stops getting reported, the Node Health Monitor informs the operator at the base station that this data was not received, allowing the human to determine whether to allow the vehicle to continue driving. If the vehicle is the only vehicle on the track, it may be safe to continue driving. If there is another vehicle present, however, a stop may be necessary to keep all vehicles safe.

If the Long Control, Path Tracker, SSC Interface, or Graceful Stop Nodes crash, the vehicle can no longer continue to run, and must be manually recovered and taken back to the pits. A crash of the topic multiplexer node is less severe, because the node health monitor will take over its role of providing localization data to the control module.

C. HALO Node III:Topic Multiplexer

The Topic Multiplexer node is essentially a data health monitor that chooses between different data sources and provides whichever is most accurate. The data monitored by this node can be seen in Figure 8. The node continuously determines the most accurate source of localization between the two GNSS units and the EKF estimation. Because the EKF position is reported at a higher rate of 100 Hz, the topic multiplexer defaults to EKF as the mian source of localization. However, the EKF could become inaccurate even while the GPS remains accurate, such as in the case where the vehicle makes an unexpected sharp turn. Therefore, if the EKF covariance is above a threshold of 0.1225, then the topic multiplexer falls back on the slower 20 Hz GNSS as the source of localization. To determine which GNSS is more accurate, the node monitors the covariance, which is derived from the standard deviation reported with the latitude and longitude. In order to switch back from the lower rate GNSS to the faster EKF estimation, the node requires some number of EKF messages in a row which all have a covariance below the defined threshold. If any source stops reporting data, the topic multiplexer will disregard that source when determining the most accurate data. If all sources stop reporting data at the same time, the topic multiplexer will send a flag to the graceful stop node to bring

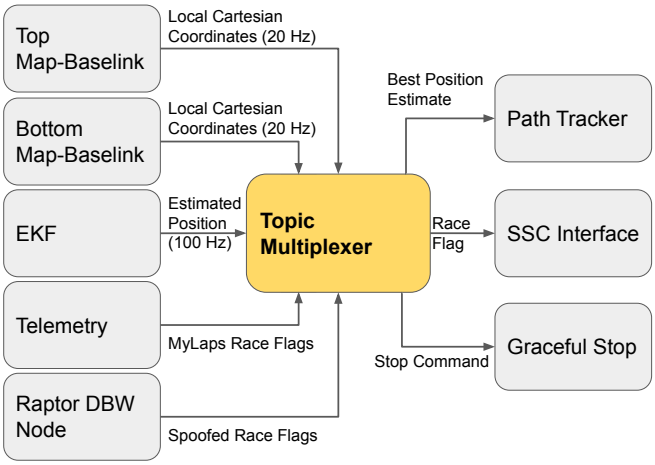

Fig. 8. The Topic Multiplexer Node chooses between multiple data sources to provide the remaining nodes with the best data available. It also detects data health faults in the data it processes.

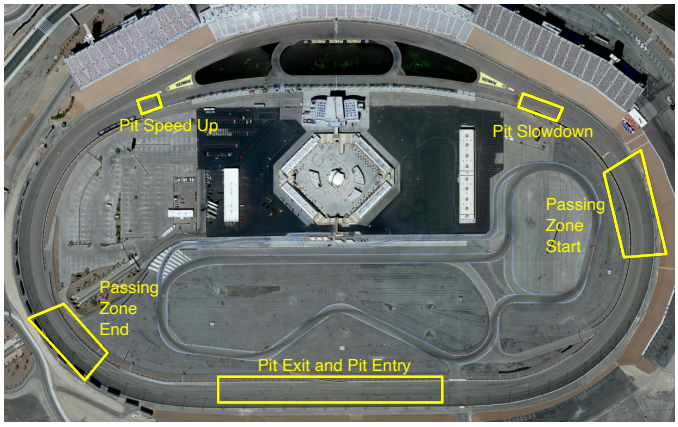

Fig. 9. The transition polygons defined on the Las Vegas Motor Speedway Track. Using these polygons, the vehicle can detect whether it is in the pits or on the track (Pit Exit and Pit Entry), beside the pit boxes (Pit Slowdown and Speed Up), or if it is in the passing zone (Passing Zone Start and End).

the vehicle to a stop. This could be the case if both GNSS units lose connection with the satellites.

D. HALO Node IV: Behavioral-Safety Monitor

The behavioral-safety monitor in our stack is implemented in the SSC Interface Node. This node is responsible for changing the desired velocity of the vehicle in response to flags received from race control. It also determines which reference path is provide to the Path Tracker Node (Section IV-B).

Polygon-Based Transitions for Race Flags The SSC Interface Node determines the desired velocity of the vehicle based the current race flag and the vehicle's location. For example, the vehicle should be slower while driving on pit lane than while driving on the track. The vehicle's position in this context is a coarse-grained determination of which part of the race track the vehicle is in, rather than an x,y coordinate value. To achieve this coarse-grained localization, polygons that mark the entrance to a new section of the track have been defined in the local coordinate frame. Figure 9 Shows the polygons that were used for the Las Vegas Motor Speedway track. The polygon at the bottom center of the image was used to identify when the vehicle had left the pits and entered onto the track, as well as when the vehicle was leaving the track to return to the pits. Additional polygons are used to define the start and end of the passing zone, and the start and end of the pit lane. This polygon-based transition method allows the node to know whether the vehicle is in the pit lane, in an overtaking zone, or elsewhere the track. Using this knowledge, whenever the node receives a new flag it uses a lookup table to determine how fast the vehicle should be traveling for the part of the track. The SSC Interface Node is also responsible for changing the reference path for the path tracker to follow.

Closing-the-Door Maneuver If the ego vehicle receives the signal from race control that it can overtake the opponent (Waving Green flag), it changes from the "raceline" path on the inside of the track to the "overtake" path on the outside of the track. The vehicle will then remain on the "overtake" path until it detects that it has passed the opponent to the point where it has maintained the required longitudinal separation of 30m to the opponent (as defined in Section III). At which time it may merge to the "raceline" path. This maneuver to return to the "raceline" path is referred to as "Closing-the-Door." Due to uncertainty associated with the obstacle distance reported by the perception module, it is not prudent to immediately switch paths on the first indication of the 30m distance measurement. This could be a false positive reading. Instead, the SSC interface implements a sliding window over the reported distances, and only closes the door if at least n out of the latest k distance measurements are reported to be outside of the safety distance requirement.

X. Results and Evaluation

During the Indy Autonomous Challenge, we recorded ROS Bags containing all of the ROS topics and sensor data from each run at the IMS and LVMS racetracks. We use this data evaluate the effectiveness of the HALO architecture to successfully mitigate the different failure modes. Specifically, we present three results, each corresponding to a different failure mode described in Section VIII.

A. Mitigating Data Health Faults

Figure 10 shows an example of a data health fault. Each block represents a node or module with a vertical arrow representing time. Each arrow going across between these node timelines denotes a message sent between the nodes. Here, the GNSS is reporting the vehicle's position to the EKF node and the Topic Multiplexer at 20 Hz. The EKF Node fuses GNSS data with IMU data and velocity data to produce localization estimates at 100 Hz, and reports them to the HALO Node III: Topic Multiplexer (Section IX-C). The Topic Multiplexer determines the best (accurate and fast) source of localization and reports it to the Control Module. Here it chooses the EKF source because the reported data covariance is below the defined accuracy threshold. This is nominal operation of the vehicle as it drives around the track. At some point, the covariance of the EKF localization estimates begin

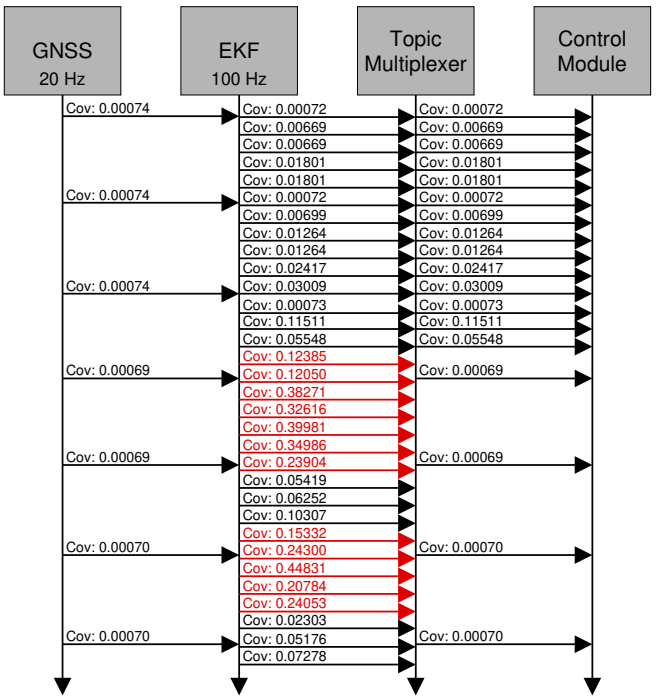

Fig. 10. Data showing a data health fault. The EKF localization estimate begins to drift, but the GNSS data remains accurate. In this case, the HALO Topic Multiplexer switches from the high frequency EKF to the lower frequency, but higher accuracy, GNSS data.

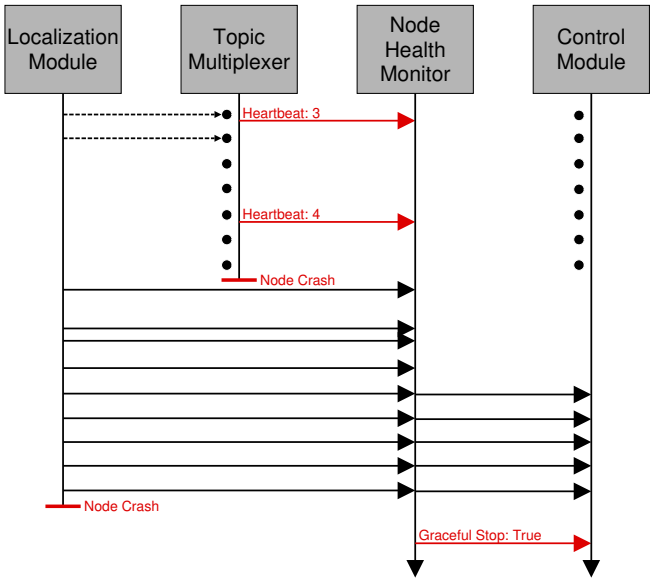

Fig. 11. Data showing a node health fault. The Topic Multiplexer node crashes, so the HALO Node Health Monitor instead begins providing localization data to the control module. Then the localization node crashes, so the Node Health Monitor tells the vehicle to perform a graceful stop.

to rise. This may occur due to errors in the velocity estimates or IMU data. In Figure 10, this is shown by the red arrows emanating from the EKF, which have a high covariance value (0.12385). During this time, even though the EKF becomes inaccurate, the GNSS data continues to be accurate, as shown by the low covariance value (0.000069). The topic multiplexer detects the deteriorating EKF accuracy, and switches to the lower rate (but higher accuracy) GNSS data. As seen in Figure 10, after the EKF's covariance rises, the rate of data going to the control module slows to the 20 Hz rate of the GNSS.

This example shows successful mitigation of a data health fault by the HALO Node III: Topic Multiplexer. Without the intervention of the Topic Multiplexer, the localization covariance would have continued to increase resulting in a risk of a vehicle crash. Instead, the HALO Topic Multiplexer switches to the more accurate GPS data, allowing the vehicle to continue to drive safely, albeit slightly more slowly due to the lower GNSS rate of 20 Hz.

B. Mitigating Node Health Faults

Figure 11 shows an example of a node health fault, correctly mitigated by the HALO architecture. In this example, the localization module is periodically reporting its position to the topic multiplexer, and the topic multiplexer is sending that data to the control module. This data flow is shown by the black dots in Figure 11. During this process, the topic multiplexer is publishing heartbeats in the form of a rolling counter to the HALO Node II: Node Health Monitor (Section IX-B). At some point the Topic Multiplexer ROS node experiences a fatal error and crashes, shown by the red bar at the end of the Topic Multiplexer's timeline. After receiving the last heartbeat from the Topic Multiplexer, the Node Health Monitor waits to see if the node is dead or if the heartbeat message was merely delayed. After waiting a defined time threshold (500 milliseconds), and still not receiving a heartbeat from the Topic Multiplexer, the Node Health Monitor begins its safety protocol. Since the Topic Multiplexer was the node which has crashed, the Node Health Monitor takes over reporting localization data to the control module. In Figure 11 this is shown by the arrows coming from node health monitor to the control module. Following this switch, the node providing localization also has a fatal error and crashes. Due to a complete lack of localization data, the Node Health Monitor tells Long Control to bring the vehicle to a stop. In Figure 11 this is shown by the red arrow indicating the graceful stop flag being set to *true*. This example shows the mitigation of a node health fault by the HALO Node Health Monitor. When the Topic Multiplexer dies, the Node Health Monitor is able to keep the vehicle operational by itself providing the localization data. Only when the localization node crashes and there is no localization data to provide to the control module does the vehicle enter into a graceful stop.

C. Mitigating Behavioral-Safety Faults

In this example, the Ego vehicle is overtaking an opponent. As discussed in Section III, the Ego needs to maintain a longitudinal separation of 30m from the opponent before closing-the-door. Figure 12 shows distance of the ego from the opponent as reported by the perception module. Positive numbers indicate that the ego is behind the opponent, while negative numbers indicate that the ego is in front of the opponent. During the beginning of the overtake maneuver,

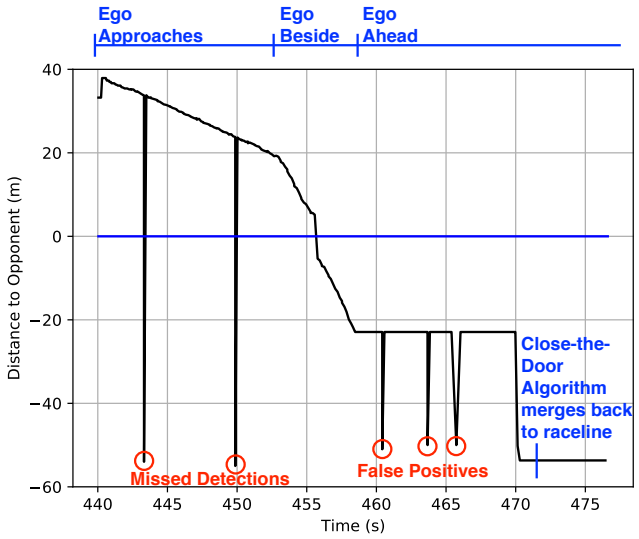

Fig. 12. Data showing a HALO behavioral-safety fault. The ego vehicle approaches an opponent for an overtake. The perception stack misses several detections of the opponent vehicle. Without a behavioral monitor, these missed detections would cause the vehicle to merge to the inner race line too early.

the perception stack erroneously reports observations of the opponent far behind the ego. Without the HALO Behavioral-Safety Monitoring, the ego vehicle would attempt to move to the inner line as soon as the first missed detection occurs, and would merge behind the opponent's vehicle. Since the ego is traveling faster than the opponent, merging behind the opponent could result in an increased risk of crashing into the opponent. Similarly, after the ego has pulled ahead of the opponent, there are several false positives, where the obstacle is reported as being far behind the ego. Without the Close-the-Door algorithm described in Section IX-D, the ego vehicle would attempt to close-the-dooronthe opponent too early. Not until 470 seconds is the opponent clearly behind the ego, and the perception stack reports the opponent's distance as consistently around -53m. At this time, the HALO Behavioral-Safety Monitor can confidently initiates the merging maneuver.

XI. Conclusion and Future Work

This paper describes HALO, a fault-tolerant architecture for an autonomous racing software stack. This is among the first papers to present such a deep dive into the various faults that were experienced on a real fully autonomous race car. We implemented HALO based on our software used on a full-scale fully-autonomous AV-21 race car in the Indy Autonomous Challenge. We first conduct a failure mode, effects, and criticality analysis of the control, localization, communication, and perception modules of the stack. Based on this, we present three different failure modes: data health, node health, and behavioral-safety faults. Our HALO architecture comprises of a composition of nodes that are designed to handle these faults and minimize the risk of a vehicle crash, even in the presence of poor data, node crashes, and faulty perception. We demonstrate the effectiveness of HALO on data gathered from the vehicle during the real-world race runs. Our ongoing and future work involves extending the architecture to include additional failure modes, and to integrate HALO with existing open-source self-driving stacks outside of autonomous racing.

References

[1] [Online]. Available: https://www.indyautonomouschallenge.com/
[2] M. Buehler, K. Iagnemma, and S. Singh, *The 2005 DARPA grand challenge: the great robot race*. Springer, 2007, vol. 36.
[3] ——, *The DARPA urban challenge: autonomous vehicles in city traffic*. springer, 2009, vol. 56.
[4] (2020) Open platform. [Online]. Available: https://apollo.auto/developer.html
[5] (2021) comma.ai. [Online]. Available: https://comma.ai/
[6] (2021) Autoware.auto. [Online]. Available: https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/
[7] (2021) System monitor for autoware. [Online]. Available: https://tier4.github.io/autoware.iv/tree/main/system/system_monitor/
[8] K. Cheng, Y. Zhou, B. Chen, R. Wang, Y. Bai, and Y. Liu, "Guardauto: A decentralized runtime protection system for autonomous driving," *CoRR*, vol. abs/2003.12359, 2020.
[9] Y. Jiang and S. Yin, "Recursive total principle component regression based fault detection and its application to vehicular cyber-physical systems," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 4, pp. 1415–1423, 2018.
[10] S. Safavi, M. A. Safavi, H. Hamid, and S. Fallah, "Multi-sensor fault detection, identification, isolation, and health forecasting for autonomous vehicles," *Sensors (Basel)*, vol. 21, no. 7, pp. 2547–2569, Apr 2021.
[11] C. Alippi, S. Ntalampiras, and M. Roveri, "Model-free fault detection and isolation in large-scale cyber-physical systems," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 1, pp. 61–71, 2017.
[12] F. Y. Chemashkin and A. A. Zhilenkov, "Fault tolerance control in cyber-physical systems," in *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2019, pp. 1169–1171.
[13] K. Ding, S. Ding, A. Morozov, T. Fabarisov, and K. Janschek, "Online error detection and mitigation for time-series data of cyber-physical systems using deep learning based methods," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 7–14.
[14] P. Kaveti and H. Singh, "ROS rescue : Fault tolerance system for robot operating system," *CoRR*, vol. abs/1910.01078, 2019.
[15] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, ser. EMSOFT '16. New York, NY, USA: Association for Computing Machinery, 2016.
[16] R. Gupta, Z. T. Kurtz, S. A. Scherer, and J. M. Smereka, "Open problems in robotic anomaly detection," *CoRR*, vol. abs/1809.03565, 2018.
[17] H. Tabani, L. Kosmidis, J. Abella, F. J. Cazorla, and G. Bernat, "Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.
[18] T. Ishigooka, S. Otsuka, K. Serizawa, R. Tsuchiya, and F. Narisawa, "Graceful degradation design process for autonomous driving system," in *Computer Safety, Reliability, and Security*, A. Romanovsky, E. Troubitsyna, and F. Bitsch, Eds. Cham: Springer International Publishing, 2019, pp. 19–34.
[19] M. Park, S. Lee, and W. Han, "Development of steering control system for autonomous vehicle using geometry-based path tracking algorithm," *Etri Journal*, vol. 37, no. 3, pp. 617–625, 2015.
[20] M. Nolte, M. Rose, T. Stolte, and M. Maurer, "Model predictive control based trajectory generation for autonomous vehicles — an architectural approach," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 798–805.
[21] R. E. Kopp and R. J. Orford, "Linear regression applied to system identification for adaptive control systems," *AIAA JOURNAL*, vol. 1, no. 10, pp. 2300–2306, Oct 1963.
[22] J. Poppinga, N. Vaskevicius, A. Birk, and K. Pathak, "Fast plane detection and polygonalization in noisy 3d range images," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008, pp. 3378–3383.
[23] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, p. 226–231.