Contents

Chapter 1

Software Architecture and Data Pipeline

This chapter covers how the WATonomous Automated Driving Stack (ADS) is architected in simulation and on Bolty.

Acknowledgement: The design and implementation of the technology covered here was done primarily by Rowan Dempster, with secondary contributions by Charles Zhang and Chuan Tian (Ben) Zhang.

1.1 Requirements

The requirements for the software architecture design were:

1. Support rapid prototyping of new software modules, and the rapid onboarding of new WATonomous team members. By rapid prototyping it is meant that when a new software library dependency or and entire new software module is introduced, that process should be painless. Developers should not have to worry about incompatible versioning. By rapid onboarding it is meant that new team members should only have to install a small set of additional software (limited to Integrated Development Environment (IDE) software and visualization software like Virtual Network Computing (VNC)), and any modern laptop should be able to run such required software.

2. Support the concurrent use of a single Virtual Machine (VM) in the WATonomous server cluster by multiple developers. This means multiple WATonomous team members can do development work simultaneously using the CARLA simulator on the same VM. Thus, data coming from CARLA and data being exchanged between ADS modules must be isolated for every VM user.

3. Have the ability to deploy the ADS developed in simulation to the production Bolty vehicle with no modifications to the software modules themselves.

1.2 Solutions Implemented

In light of these requirements, the *watod* ecosytem was designed. In *watod*, each software module (e.g. the detector, tracker, environment model, occupancy grid, etc...) is containerized into its own Docker container which only has that module's software dependencies installed (see right side of Fig. 1.1). Thus, when a new dependency is required for a specific ADS module, only the that module's Docker image needs to be updated, without having to worry about affecting the whole system.

watod also supports inter-module communication via ROS. A new module can be added to *watod* as a new Docker container and hook-in to the existing data pipeline using the ROS communication protocols.

An instance of the *watod* ecosystem can be run remotely on any of WATonomous' server cluster VMs and connected to via the Visual Studio Code IDE. Thus, there is no hardware requirements for new members' laptops to interact with *watod*. New team members can get up and running just by connecting to a VM and executing the *watod* commands necessary to start the Docker containers they need to do development (see top left side of Fig. 1.1).

The CARLA simulator is also Dockerized and available via *watod*. CARLA provides sensor inputs to the software modules via the Carla ROS Bridge package, allowing for simulation based development of the ADS. *watod*, via Docker network isolation, ensures that data coming from CARLA stays within each VM user's separate ADS instance, allowing many simulators and ADS instances to be running concurrently on a single VM. Thus, there is no software-enforced limit to how many concurrent users the VMs in the WATonomous server cluster can support. There are limits imposed by the hardware resources needed to run the simulations. At the time of writing, the WATonomous server cluster can support 50+ concurrent team members remotely developing using CARLA.

Furthermore, the Carla ROS Bridge package is configured to publish information in the same format as the actual sensors on Bolty. Therefore, the ADS modules need not be

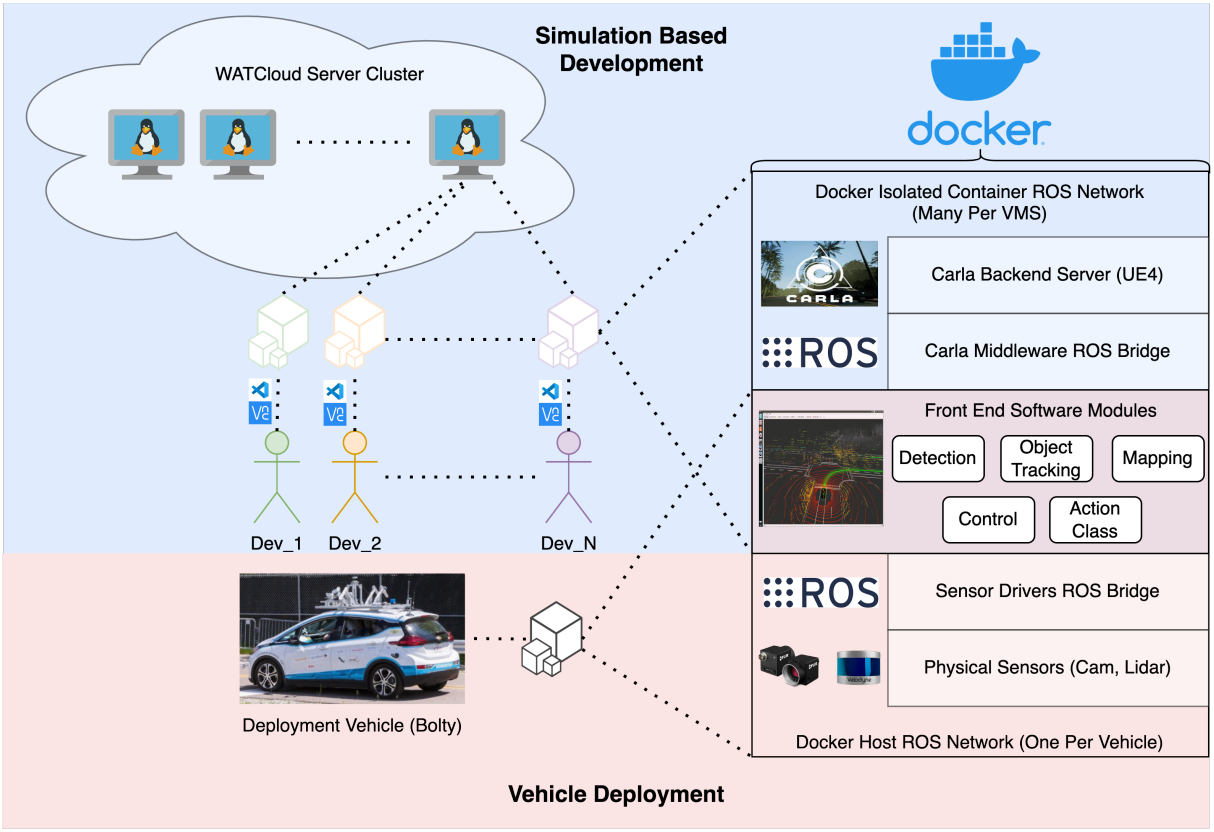

Figure 1.1: Diagram of WATonomous' approach to software development and deployment. On the simulation side (top), the penguin monitors represent abstracted Ubuntu VMs running on some physical hardware in the WATonomous server cluster. Multiple developers can run isolated ADS instances on a single VM, and can interact with their instance using Visual Studio Code and a VNC client. Each isolated ADS instance is spun up using *watod* utilities, and includes an instance of CARLA, the Carla ROS Bridge, and however many "front-end" software modules the developer needs for their work. On the deployment side, there is only ever a single instance of the ADS running on Bolty, and the CARLA simulator inputs are replaced by the physical sensors. However, from the perspective of the "front-end" software modules, nothing has changed.

aware if they are running in the context of the CARLA simulator, or the real world. This makes deployment to the research vehicle simple. The only thing that changes is where the sensor input is coming from and thus no changes to the software modules are needed for deployment of the ADS.

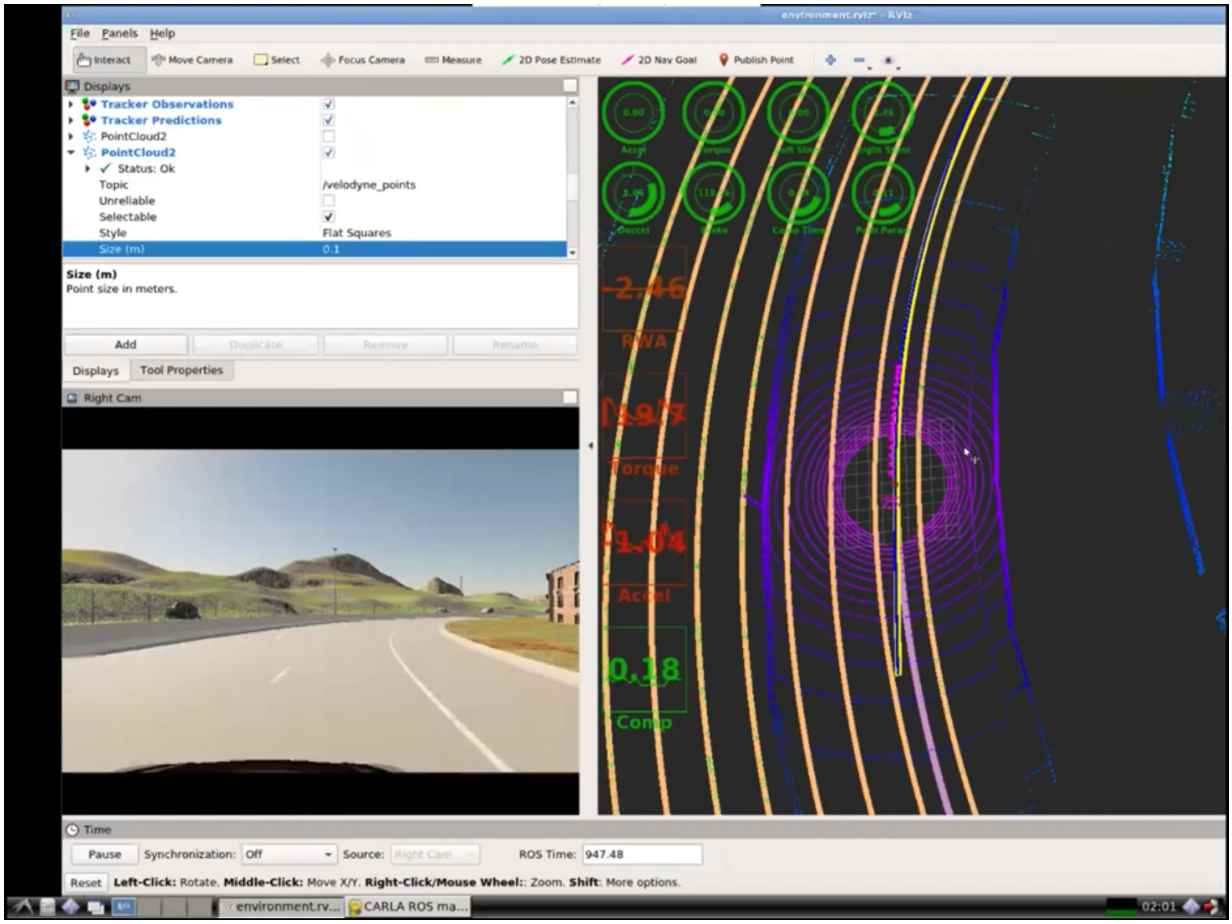

Figure 1.2: Screenshot RViz running in the GUI Tools container, accessible to remote developers via a VNC client software.

One downside of a fully Dockerized simulator and ADS is that any desktop based visualization (like the popular RViz software for robotics) is not immediately supported. To move past this limitation, the GUI Tools container was added to the *watod* ecosystem. The purpose of this container is solely to run desktop based visualization software such as RViz and expose a VNC server. This way, remote developers are able to interact with any desktop software via a VNC client application on the developer's local laptop. Fig. 1.2 shows the RViz instance that developers see when they connect to the VNC server exposed by the GUI Tools container.

An in-depth tutorial of how to use the *watod* ecosystem in conjunction with ROS and

the WATonomous server cluster can be found at:
https://drive.google.com/file/d/17N2Zrw1cjUfZgALg3ImMrmcRMzwkOGVz/view.
Additionally, an in-depth tutorial of how to use the CARLA simulation functionality built into *watod* can be found at:
https://drive.google.com/file/d/1F32Ui6QaegQ8wSl-ga5ifP1fNKGzba09/view.

1.3 ROS Software Module Data Pipeline Design

The *Front End Software Module* block on the right side of Fig. 1.1 is where the majority of the design work of the ADS was done. A more detailed view of that block is shown in Fig. 1.3. See Appendix A for the custom ROS message definitions mentioned in this section.

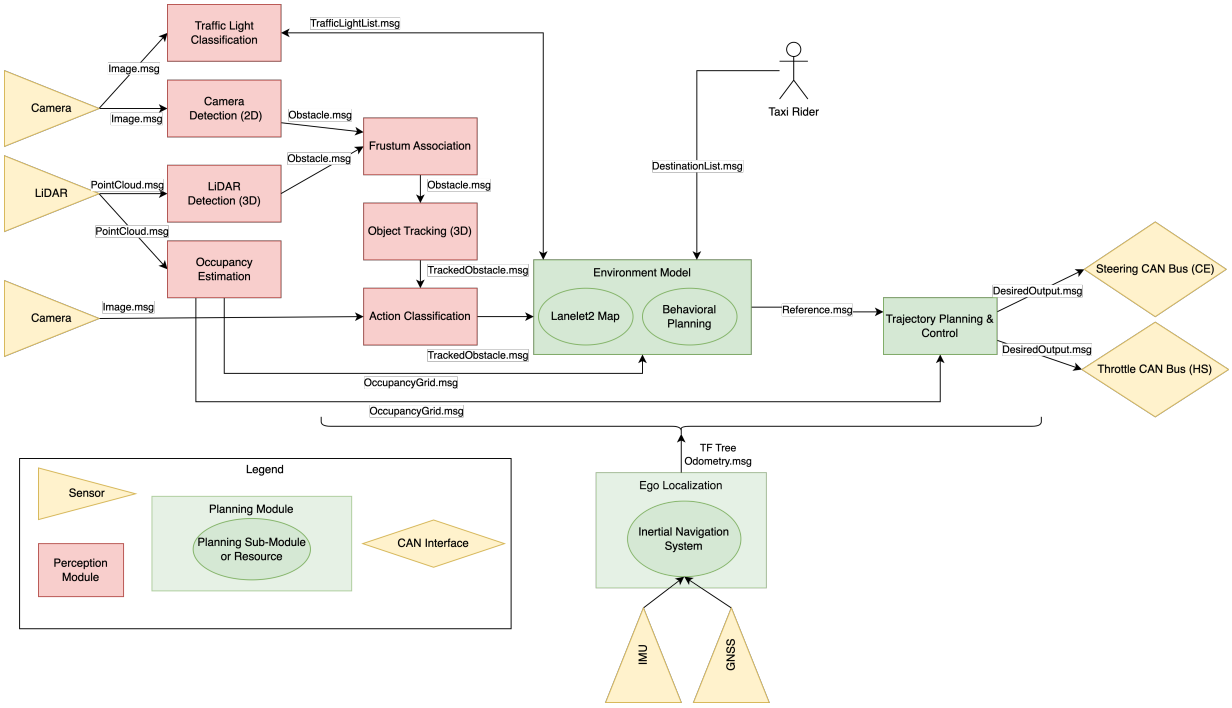

Figure 1.3: The WATonomous ADS data pipeline. Uni-directional arrows represent "topics" in the publisher-subscriber ROS communication paradigm. Bi-directional arrows represent topics in the client-server ROS communication paradigm. Arrows are labeled with the data type (ROS message) being transmitted over that topic.

Sensor Interfacing

WATonomous uses cameras (Blackfly S Color 3.2 MP GigE Vision) and LiDARs (Velodyne Ultra Puck) to perceive the world, and a Global Navigation Satellite System (GNSS) + Inertial Measurement Unit (IMU) Span Device (Novatel SPAN PwrPak7-E1) to provide odometry. First, raw sensor readings are received from the sensors and processed into ROS messages by the sensor drivers (Pointgrey Camera Driver, Velodyne LiDAR Driver, and Novatel SPAN Driver), and then are published into the ROS network so that the other software modules can further process the information.

Localization

The Novatel SPAN Driver implements a complete inertial navigation system, and publishes an *Odometry.msg* messages at 20Hz into the ROS network, which many software modules subscribe to. The localization stack also sets up the ROS transform tree.

Perception

Next, the perception modules perform detection, classification, and tracking to transform the high dimensional sensor modalities (images and point clouds) into low dimensional representations that the environment model can understand. YOLOv5 [3] is used for 2D object and sign detection in camera images, and publishes *Obstacle.msg* messages. The euclidean clustering package from Autoware [2] is used for 3D obstacle detection on the LiDAR point clouds, and also publishes *Obstacle.msg* messages. A custom frustum-based heuristic is used for 2D-3D association. The occupancy grid estimation module used was developed by Autonomoose [1], and is based on the well-known log odds formulation from Probabilistic Robotics by Thrun *et al.* [5]. The environment model sends the 3D positions of upcoming traffic lights to the traffic light state classification module, where they are projected into the camera 2D frame and then classified in 2D using Hue, Saturation, Value (HSV) thresholds. The traffic light state classification module publishes *TrafficLight.msg* messages.

Object tracking is done in 3D after the frustum-based association. A modified version of the AB3DMOT [6] algorithm (a benchmark 3D multi-object tracker which uses a linear Kalman Filter and Hungarian matching for association) is used, and publishes *TrackedObstacle.msg* messages. These tracks as well as the camera stream are then sent to the action classification module which appends a history of actions to each track using the neural design described in Chapter **??**.

Mapping and Environment Modeling

Offline high definition Lanelet2 maps [4], which describe lane geometry and topology, are used for navigation and motion planning. These maps are crafted by hand using the JOSM map editing software [7].

The environment modeling module unifies the prior Lanelet2 map with the online perception outputs, creating a combined relational graph that is used for behavioral planning. This process is described in detail in Chapter **??**. The behavioral plan is expressed as a *Reference.msg* message, which is continuously sent to the controller.

Motion Planning and Control

The controller consumes the *Reference.msg* message as well as a description of free space (*OccupancyGrid.msg*) from the occupancy estimation module. These messages are then used as parameters for a non-linear program which is continuously solved as part of a Model Predictive Control (MPC) design that optimizes trajectory and control actions (longitudinal acceleration and road wheel angle). These control actions are then sent to the vehicle's Controlled Area Network Bus (CAN Bus). Details of the controller implementation are given in Chapter **??**.

CAN Bus Interfacing

The control actions are sent to the CAN Bus modules, or the CARLA simulator in the context of remote simulation development, via the *DesiredOutput.msg* message. The control actions are then actuated and thus the robotic feedback loop is closed

Human-Computer Interface

When a user of the system enters the vehicle, they select a point on the Lanelet2 map which is displayed via RViz. This point gets translated into a destination lane, which then gets sent to the environment model as a *DestinationList.msg* message. The environment model next runs a search over its internal graphical representation of a world to find a route from the Autonomous Vehicle (AV)'s current location to the user's destination, and the autonomous drive begins.

References

[1] Wise automated driving system, Sep 2022.

[2] Autoware. lidar_euclidean_cluster_detect.

[3] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, NanoCode012, Yonghye Kwon, TaoXie, Jiacong Fang, imyhxy, Kalen Michael, Lorna, Abhiram V, Diego Montes, Jebastin Nadar, Laughing, tkianai, yxNONG, Piotr Skalski, Zhiqiang Wang, Adam Hogan, Cristi Fati, Lorenzo Mammana, AlexWang1900, Deep Patel, Ding Yiwei, Felix You, Jan Hajek, Laurentiu Diaconu, and Mai Thanh Minh. ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference, February 2022.

[4] Fabian Poggenhans, Jan-Hendrik Pauls, Johannes Janosovits, Stefan Orf, Maximilian Naumann, Florian Kuhnt, and Matthias Mayr. Lanelet2: A high-definition map framework for the future of automated driving. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 1672–1679. IEEE, 2018.

[5] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[6] Xinshuo Weng, Jianren Wang, David Held, and Kris Kitani. 3d multi-object tracking: A baseline and new evaluation metrics. *arXiv preprint arXiv:1907.03961*, 2020.

[7] OpenStreetMap Wiki. Josm — openstreetmap wiki,, 2022. [Online; accessed 29-September-2022].

APPENDICES

Appendix A

ROS Message Definitions

A.1 Sensor Messages

Messages produced by sensors.

A.1.1 PointCloud.msg

See http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/PointCloud.html

A.1.2 Image.msg

See http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/Image.html.

A.2 Perception Messages

Messages produced by the perception stack.

A.2.1 Obstacle.msg

```
Header header
```

```
# Obstacle Type Enums
string OBS_TP_UNKNOWN=UNKNOWN
string OBS_TP_PED=PEDISTRIAN
string OBS_TP_CYC=CYCLIST
string OBS_TP_VCL=VEHICLE
string label # see Obstacle Type Enums

# Detection confidence
float32 confidence

# Position and its uncertainty
# For 3d bounding boxes, the (x, y, z) is the center point of the 3d bounding box
# For 2d bounding boxes, the (x, y) is the top left point of the 2d bounding box
geometry_msgs/PoseWithCovariance pose

# Velocity and its uncertainty
geometry_msgs/TwistWithCovariance twist

# Dimensions of bounding box assuming BEV perspective
# x=width, y=height, z=depth
# For example, a vehicle with its bumper facing North that's
# encapsulated by a rectangular prism defines width as the
# E/W measurement, height as the N/S measurement.
# Obstacle.msg is also used as 2d bounding boxes
# In that case width_along_x_axis is the width of the 2d bounding box
#   in image coordinates
# and height_along_y_axis is the height of the 2d bounding box
#   in image coordinates
# z axis is not used with 2d bounding boxes
float64 width_along_x_axis
float64 height_along_y_axis
float64 depth_along_z_axis

# Unique ID number
uint32 object_id
```

A.2.2 TrafficLight.msg

```
Header header

# Traffic Light State Enums
string TL_ST_NON=NON
string TL_ST_RED=RED
string TL_ST_YEL=YELLOW
string TL_ST_GRE=GREEN
string TL_ST_FLA=FLASHING_RED
string left
string forward
string right

# Traffic Light Sign Direction Enums
string TL_DIR_NON=NON
string TL_DIR_LT=LEFT
string TL_DIR_RT=RIGHT
string TL_DIR_FD=FORWARD
string TL_DIR_LT_FD=LEFT_FORWARD
string TL_DIR_RT_FD=RIGHT_FORWARD
string sign_dir

geometry_msgs/Pose pose
geometry_msgs/Vector3 dimensions

geometry_msgs/Point p1
geometry_msgs/Point p2

common_msgs/StopLine stop_line

# Unique ID number
uint64 id
```

A.2.3 OccupancyGrid.msg

See http://docs.ros.org/en/melodic/api/nav_msgs/html/msg/OccupancyGrid.html.

A.2.4 TrackedObstacle.msg

```
Header header

common_msgs/Obstacle obstacle

common_msgs/TrackedObstacleState[] observation_history
common_msgs/TrackedObstacleState[] predicted_states
```

TrackedObstacleState.msg

```
# Attributes of the TrackedObject at every observation/prediction
# To be published in the Odom frame

Header header
# bounding box centroid (m) and orientation
geometry_msgs/Pose pose
# Velocity (m/s)
geometry_msgs/Twist velocity
```

A.3 Navigation Messages

Messages used by the navigation stack.

A.3.1 DestinationList.msg

```
# Ordered sequence of destinations that the ego vehicle should proceed through
# In competition this list will be hardcoded and published via the command line
# In RVIZ the 2D Nav Goal Tool is converted to a single "x,y" destintation
string[] destination_list
```

A.3.2 Odometry.msg

See http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html.

A.3.3 Reference.msg

```
Header header

geometry_msgs/Point[] ref_line
geometry_msgs/Point[] left_bound
geometry_msgs/Point[] right_bound

path_planning_msgs/PredictedTraj[] trajectories
```

A.3.4 DesiredOutput.msg

```
Header header
float32 theta
float32 torque
# Two states: 0 = stopped, 1 = driving
int8 state
```