Towards CANLay: User-centered Overlay Design for In-Vehicle Data Dissemination In a Network Virtualized Testbed

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction and Background

In recent years security of the Controller Area Network (CAN) has been a much talked about topic of research. CAN is a broadcast media that enables reliable and low-latency communication between in-vehicle devices, also referred to as Electronic Control Units (ECU). This broadcast nature of CAN, along with the fact that it is inherently unauthenticated, makes it susceptible to network-wide cyber threats. Security researchers have shown [**?**, **?**, **?**] that remote interfaces on modern vehicles can be used to intrude into internal CAN networks and inject messages to control and/or disrupt the operations of the vehicle. At the same time, the development of security solutions can be pursued to detect and/or prevent this scenario from occurring. To evaluate the effectiveness of their methods, researchers have typically experimented on real-vehicles or homegrown testbed setups that mimic real vehicles. While most households in the United States have at least one passenger car[1], this is not the same for medium and heavy-duty (MHD) vehicles. Moreover, creating homegrown testbeds is both logistically and economically challenging. To that end, the need for a publicly accessible testbench is imminent. This is where the concept of the Software Define Truck (SDT) [**?**] is critical to the in-vehicle networking community. It aims to provide a distributed network virtualized platform on which in-vehicle security experiments can be performed. Although proposed primarily for the heavy-trucks, SDT can easily be adapted for lightweight passenger vehicles. SDT's information exchange goal is shown in figure **??**. CAN frames and physical signals need to be exchanged between ECUs and

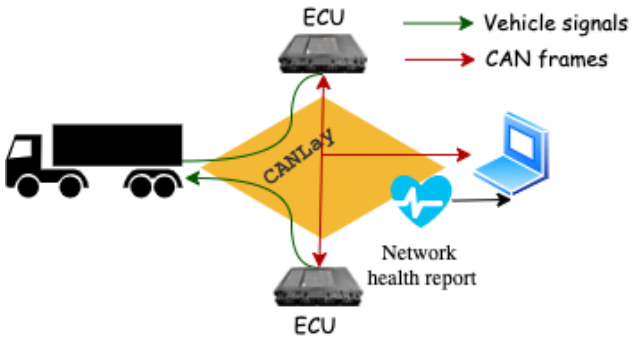

Figure 1: Scope of CANLay

vehicle simulators located in different subnetworks around the globe. CANLay is the networking backbone of the SDT and aims to provide the necessary infrastructure to enable this service.

Previous research [**?**] has established two critical criteria for quality evaluation of automotive networking testbeds: fidelity and adaptability. Fidelity is the ability to emulate a real-world in-vehicle networking infrastructure. Adaptability is the ability to simulate different real-world in-vehicle networking infrastructures. As such, it may be difficult to optimize both at the same time. To make a system adaptable, the underlying components need to be virtualized so they can be reconfigured to suit user needs. Albeit, this hampers the fidelity of the system. While CANLay provides the means to configure experiment networks on-demand, it also provides a real-time health report for the underlying network. This allows the user to assess the fidelity of the overlay in terms of standard networking metrics like latency, rate of packet drop, etc.

Although, CAN is a relatively new communication technology and has a smaller application scope than TCP/IP, there has been some proposals to virtualize its operations. First, there has been the attempt to adapt the software-defined networking paradigm for CAN [**?**, **?**, **?**]. This approach is largely

[1] https://www.statista.com/statistics/551403/number-of-vehicles-per-household-in-the-united-states/

hardware-based and is catered for in-vehicle networking on CAN physical channels, not over long-range overlays. For range relaying of CAN frames, there has been the CAN-to-ethernet direction of research [**?**, **?**]. The goal is not to enable ECU-to-ECU communication, rather transportation of data logged from one network to a remote endpoint. Configurability and network performance are usually not addressed. Neither is the CAN-to-ethernet paradigm designed to transport physical signals over long distances. X-in-the-loop (hardware, driver, vehicle etc.) simulation-based in-vehicle testbeds [**?**] have been proposed, but the signals from the simulators have been transported over physical connections, not over reconfigurable, long-range network overlays.

In summary, CANLay provides the following features:

- Transport of CAN frames and physical signals of the vehicle to a distributed network of electronic control units that can be located in different subnets
- Creation of these overlay CAN networks on demand
- Provision on runtime metrics to estimate the health of the network during the ongoing experiment.

In the rest of this paper we describe the design of CANLay (section **??**), provide a usability analysis (**??**), and finish with conclusive remarks and future works.

2 Design and Current Development

Figure **??** shows the proposed system design of CANLay. The system serves three functions: offline configuration of the network overlay and CAN frame and vehicle signal exchange at runtime. A description of the components and their roles in the system is provided next. Following that, a description of the behavioral aspects of the system is provided. Together, these aspects combine to accomplish the functional objectives.

2.1 Component Descriptions

2.1.1 Smart Sensor Simulator and Forwarder (SSSF)

The Smart Sensor Simulator and Forwarder acts as a gateway enabling the ECU to access and, more importantly, to be accessed by the CANLay system. In an active experiment, it acts as a forwarder between the Controller and the ECU through User datagram protocol (UDP) channels and CAN interfaces. SSSFs can forward two types of messages. The first type carries signals from the vehicle simulator. Eventually, these signals may have to be transmitted on the analog wiring that is shown using a dashed line figure **??**. The second type is CAN data carriers from the ECUs as well from other SSSFs in the current experiment. Through the SSSFs, multiple ECUs can actively communicate with each other to create a rich testing environment.

The SSSF is developed a built on a Teesny 3.6 .... a paragraph describing the SSS2's. Talk about SD cards. PLease include a blcok diagram etc. CAN Forwarding The real time clock on the SSF is synchronized through the network time protocol (NTP).

2.1.2 Controller

The Controller is the user's interface to the CANLay system and enables vehicle simulators to communicate with the CANLay network. The Controller's user interface is used to assist the user in building their virtual testbed. It does so by communicating with the central Server over hypertext transfer protocol secure (HTTPS). Once the experiment setup is completed the Controller transitions to acting as a gateway for a graphical vehicle simulator to communicate bi-directionally with the CANLay system. It forwards simulator outputs to the publish/subscribe (pub/sub) endpoint and listens for CAN messages from the same.

The Controller is a multi-threaded graphical application built using python. It is in charge of managing three main subcomponents: the vehicle simulator, the Network Designer, and the Network Health Monitor. The Controller manages the execution of the Vehicle Simulator so that it stay "in-step" with the flow of signals the Vehicle Simulator produces. The Controller ensures that it stays "in-step" with the Vehicle Simulator by managing all its Inter-Process Communication sockets with Selectors. Selectors enable the Controller to immediately know once the Vehicle Simulator has sent a new set of signals. By managing all the various sockets the Controller uses with Selectors its able to multiplex the streams of information effectively. In addition, the Controller manages the Network Designer and the associated request of the user. To do so the Controller maintains a persistent TCP connection with the Server. Using the same Selectors as mentioned before its able to efficiently relay network design requests to the Server and displaying the Server's responses. Also, the Contoller is in charge of maintaining the overall network health report for the system and displaying the statistics to the user. The Controller implements NTP to ensure that timestamps used for network analysis are accurate among the other devices in the session. In turn this ensures that the network statistics collected are accurate as well. Unlike the other devices in the session the Controller is in charge of aggregating the network reports from every device in a session and presenting them to the user.

2.1.3 Server

The Server helps in setting up the publishers and subscribers for an experiment. Each device opens and must maintain a persistent transmission control protocol (TCP) connection with the Server while they participate in the CANLay system. Once the TCP connection is established the devices

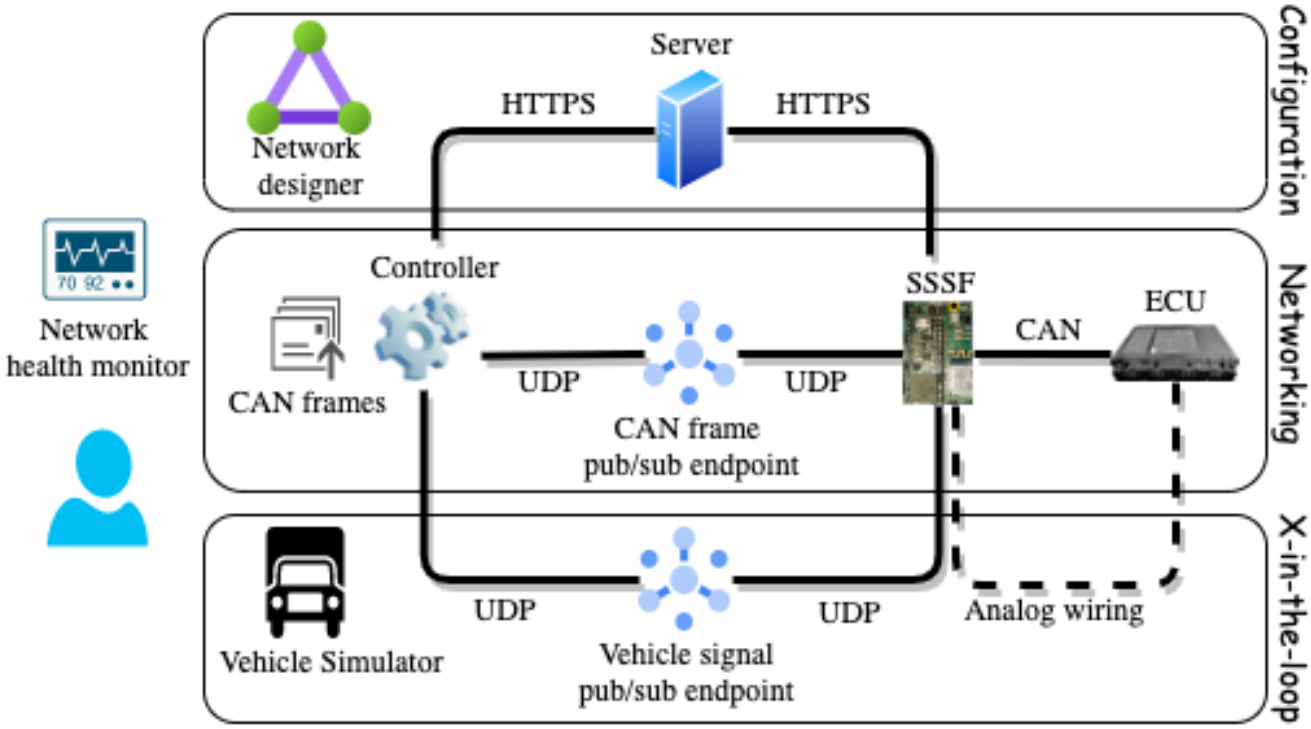

Figure 2: Proposed System

communicate with the Server through HTTP application programming interfaces (API). The Server can monitor the health of the devices and take actions if a device is malfunctioning or goes offline. This also allows the Server to keep track of free devices and free pub/sub endpoints, so it can validate new experiment requests and allocate the requested devices and endpoints without running into race conditions or double use issues that may arise if each Controller was in charge of allocating its own experiment. Finally, the Server keeps track of ongoing experiments and ensures the proper closure of a experiment in the event a device is experiencing issues.

The Server is a single-threaded session broker than multiplexes the handling of HTTP API calls through the use of Selectors. HTTP API calls were chosen because they clearly define the object to invoke and the manner in which to invoke it. The devices perform all necessary setup and teardown actions such as registering, deregistering, starting and stoping a session, by sending HTTP requests to different API endpoints. The Server responds to these requests using standard HTTP response messages and codes which can be easily interpreted by both the devices and users.

2.1.4 Publish/Subscribe Endpoints

UDP is used to connect the publishers and subscribers in the CANLay system. The pub/sub model was chosen because it can easily emulate the broadcast nature CAN [**?**] in that an ECU is subscribed to all other ECUs on the same CAN bus and all other ECUs on that CAN bus are subscribed to that ECU.

To find a suitable pub/sub mechanism that closely resembles a CAN network, we used a few criteria. The first is that the transport mechanism must support some form of message broadcasting that enables a sender to send one message that can be received by many receivers without significant duplication and delay-related overheads [**?**]. The next requirement is that the transport mechanism must enable the devices to receive messages from one or more devices while having to maintain only one connection.

At this time we have chosen UDP multicasting as a suitable pub/sub mechanism as it does not require a message broker with high-performance requirements. We realize that multicasting outside a local network may lead to increased cost for the implementers, but the current goal was to test its usability and make future decisions based on the observed performance. At this time, we are also exploring other potential pub/sub implementations such as MQTT.

2.1.5 Front-End Components

CANLay provides two front-end components: network designer and network health monitor. Figures – and – show

the snapshots of each ofn these components. Caption for the network designer UI: Currently the network designer is controlled through the command line but in the future a GUI will be added that performs the same function.

Due to the broadcast nature of CAN, the Network Designer consists only of a Catalog of available devices and the ability to select the requested devices. Upon start up the Controller contacts the Server and requests the lastest set of avaialble devices. The Controller then presents these options to the user and allows them to select the request devices.

The Network Health monitor shows real time network statistics that describe the current state of the network. These statistics are presented using heat matrices. Each cell in the matric represents a direct edge of the network. The heat matrix colors in the cells depending on their values. For packet loss, latency, and jitter the lower the number the better and the lighter the color will be. For goodput the higher the number the better and the lighter the color will be. The contrast between the light and dark colors allows the user to quickly spot parts of the network that are underperforming.

2.2 Behavior Descriptions

2.2.1 Setup (ref. figure ??)

While the Server is up and running SSSFs connect to it. SSSFs perform a setup(the) procedure by reading their inbuilt SD card. The type, year, make, model and a serial number of the connected ECUs are required to be included in a predefined file stored on the SD card. Next, the SSSF gathers its MAC address and list of attached devices into a JSON and sends it to the Server via a POST to the HTTP API endpoint $\backslash$ *SSSF* $\backslash$ *Register*. If the registration fails, the Server responds with a HTTP 4XX error code which indicates to the device why its registration failed. Otherwise, the Server responses with a HTTP 202 code indicating the SSSF was accepted. Once successfully registered the SSSF waits for further instructions from the Server on its TCP/HTTPS port.

The Controller begins by registering with the Server in a similar manner to the SSSF except that the Controller has no attached devices so it only sends its MAC address to the Server. Once successfully registered the controller asks the Server for a list of all of the available SSSF devices. Please note that if a SSSF device is currently being used in another experiment it is not considered available. After the Controller has received the list of available devices it presents the available ECUs to the user via the Network Designer described above. Notice that while the Server deals with the SSSF devices a user will typically only be interested in the ECUs that the SSSF is acting as a gateway for. After the user finalizes their selection the Controller sends the selected devices to the Server via a HTTP POST.

When the Server receives an experiment request it first checks to make sure that the request is coming from a registered Controller. Next the Server confirms that the devices are still available. If any of the devices are no longer available or become unavailable during the experiment setup process, the Server responds to the Controller with the error code 409 indicating there's a conflict in the selection. If the Controller receives this message it starts the experiment selection processes over again. If all of the devices are still available the Server then selects an available pub/sub endpoint for the experiment and assigns an index to each device. The index is used in the collection of network statistics which will be explained later on. At this time the Server sends the connection data to the Controller and selected SSSFs. Connection data contains the unique ID and index of the device, a multicast IP address and port acting as the pub/sub endpoint, and a list containing the ID and attached devices of other nodes in the experiment.

Once endpoints receives the multicast IP addresses they resync with NTP, allocate space for the required data structures, and begin listening for and forwarding messages to and from the pub/sub endpoint. At this point the experiment setup is completed.

2.3 Network Setup

2.4 Data Transport

The CANLay framework lays the foundation for X-in-the-loop simulation and testing. Before we can transform graphical vehicle simulator signals into CAN messages, there must first be a way for the in-game signals to reach the SSSF simulator and the CANLay framework achieves just that.

Once an active session has been established the Controller will begin forwarding sensor frames to the multicast endpoint. Before sending out the sensor frame the Controller will add a frame number to the message and increment it by 1. When SSSFs send out CAN messages they'll include the last frame number they saw in the message. This creates an acknowledgment feedback loop which enables the Controller to tell which devices have received a frame and when a frame has been completely lost without having to send any extra messages.

Before starting the session the Controller lets the user select the maximum number of retransmissions which is then used to calculate the timeout value of a Sensor frame. The timeout value is calculated as follows: $1/(simulator\_frame\_rate * max\_retransmitions)$. Once the controller has detected that the last frame has timed out, it'll repeat the process until it either receives a CAN message where the message frame number is equal to the current frame number or it reaches the maximum number of retransmissions. In the later case, the frame is considered to be lost and the number of lost simulator frames is incremented by 1.

On the SSSF side, when it receives a frame it first sets its last seen frame number to the message frame number. Next the SSSF will apply any necessary transformations to the

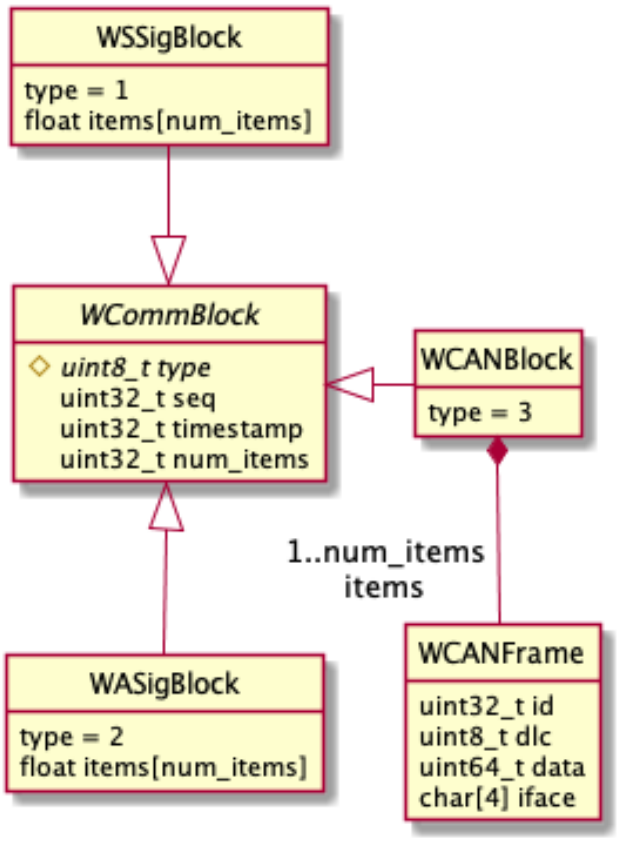

Figure 3: Transport Data Structures

Sensor frame before sending it on to the CAN network. For the purposes of this paper the transformations served mainly as a proof of concept and were kept simple, often sending just the raw sensor value in with the close matching PGN CAN frame.

2.4.1 CAN Communication (ref. figure ??)

When the SSSF is in an active experiment it attempts to read a message from the CAN network. If it reads a message from the CAN network, it proceeds to create the COMMBlock data structure (ref. figure **??**) that will be written to the CAN-Lay network. In the current scenario we set `num_frames` to 1. Therefore, a COMMBlock data structure is transmitted after the receipt of every message. COMMBlock requires some additional information before it can be written to the network. First, a type is added to indicate the subclass it is carrying. In this case the type will be 1, indicating that it is carrying a WCANFrame. It also adds the current millisecond timestamp at which it is sending the message , a sequence number that is incremented every time a CAN message is sent and a frame number of the last frame it received from the Controller. While the first two are used for network health monitoring (described in section **??**), the last one is used to for retransmission of retranmission of WSignalBlock structures as alread seen in section **??**. The SSSF also marks whether this message requires a response. The CAN message requires a response if its CAN frame's PGN matches a list received from the user.

After the SSSF is done checking for CAN messages from the CAN network, it moves on to check for messages from the pub/sub network. Again, CAN messages sent to the pub/sub endpoint are considered type 1 messages. So if an SSSF device receives a type 1 message it first checks if the message requires a response. If so, it replies to the pub/sub endpoint with a type 5 message with the frame number equal to the sequence number of the message it just received. Next it updates its network statistics about the device it came from using the sequence number, timestamp, and other metrics from the COMMBlock and then it writes the FlexCAN CAN frame onto its available CAN networks. if (msg.type == 1) if (msg.needsResponse) writeToMcastEndpoint(5. msg.sequenceNumber); networkHealth->update(msg); if (can0BaudRate) can0.write(msg.canFrame.can); if (can1BaudRate) can1.write(msg.canFrame.can);

2.4.2 Network health monitoring

As discussed earlier, monitoring the health of the network is key to ensuring that bad delays or large amounts of packet loss are not affecting your test results. In order to enable the devices to collect network statistics during an active experiment, each message is loaded with additional information. The first piece of additional information is a frame number. The Controller increments the frame number everytime it sends out new signals. When the SSSFs receive a message with a frame number higher than the previous frame number they saw, they send their frame number to the new frame number. Whenever SSSF devices send a message they add that frame number to the top of it. Therefore when a Controller receives a CAN message it can check the last frame the device had received when it sent the CAN message by checking the frame number in the COMMBlock. This system enables acknowledgement of messages without having to send any additional messages. In addition if an SSSF spots a gap in the frame numbers that is larger than 1, then it knows that a frame has been lost.

```
NetworkStats::update(index, packetSize, timestamp, sequenc
{
    now = timeClient->getEpochTimeMS();
    delay = now - timestamp;
    ellapsedSeconds = (now - HealthBasics[index].lastMessa

    calculate(HealthReport[index].latency, abs(delay));
    calculate(HealthReport[index].jitter, HealthReport[ind

    // If no packet loss then sequence number = last seque
    packetsLost = sequenceNumber - (Basics[index].lastSequ
    // If packetsLost is negative then this indicates dupl
```

Figure 4: CAN Communication Activity

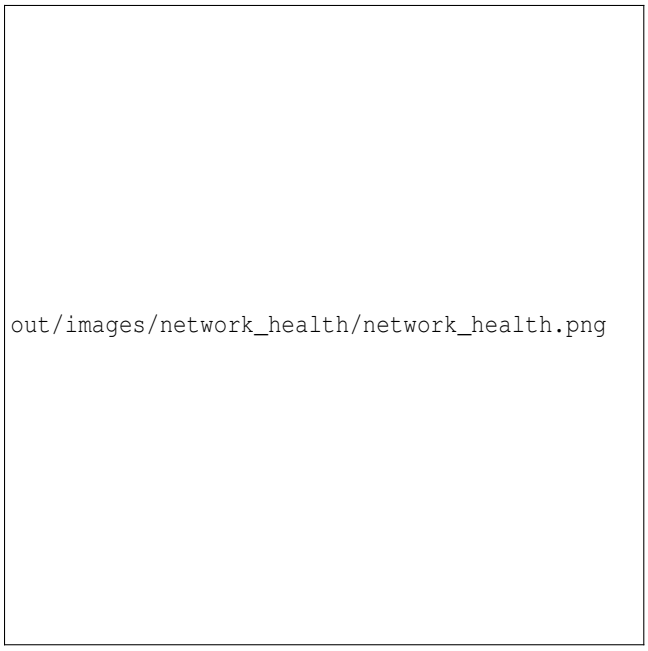

Figure 5:

3 Current Implementation

Figure **??** shows CANLay at work. The windows in the figure display CAN frames on left, the vehicle simulator on the bottom right and CANLay's network health monitoring on the top right. Each of these components were already introduced in figure **??**. For the current purpose we have been using the CARLA graphical vehicle simulator [**?**]. Although the Carla project mainly focuses on autonomous driving research it exposes its in-game signals through an easy-to-use python API and pays close attention to the scientific details represented in its simulator. While this is not required, the more realistic and accurate the signals are, the easier it will be to transform them into CAN messages. In this case, a specific CAN frame is printed as they are broadcasted on the overlay for this particular experiment. The ID of this frame is defined by the SAE-J1939 standards [**?**] and identifies engine parameters transmitted by an engine control module (ECM). The data bytes carried in the CAN frame are shown next. Of these, the third byte is shown to be changing. This particular byte carries the percentage throttle demanded by the driver. The value is also non-zero on the simulator frontend provided by the CARLA simulator.

```
    HealthReport[index].packetLoss += (packetsLost > 0) ? packetsLost : 0;

    calculate(HealthReport[index].goodput, (packetSize * 8), index);

    HealthBasics[index].lastMessageTime = now;
    HealthBasics[index].lastSequenceNumber = sequenceNumber;
}

NetworkStats::calculate(HealthCore &edge, n)
{// From: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Online_algorithm
    edge.min = min(edge.min, n);
    edge.max = max(edge.max, n);
    edge.count++;
    delta = n - edge.mean;
    edge.mean += delta / edge.count;
    delta2 = n - edge.mean;
    edge.sumOfSquaredDifferences += delta * delta2;
    edge.variance = edge.sumOfSquaredDifferences / edge.count;
}
```

On the top right is the network health monitoring window. It shows four matrices showing four different metrics to estimate network health: packet loss, latency, jitter, and goodput. The significance of each of these metrics and their calculation methods were already described in the previous section. In this case, the experiment is performed over a gigabit local area network with a layer 3 switch in between an SSSF and a Controller. The figure shows no packets were lost while the latency in the last cycle of health report collection was about 4 milliseconds between the endpoints. Although CANLay does not explicitly perform any latency reducing functions, the general latency of 4 milliseconds is considered to be sustainable for seamless CARLA emulation at standard frame rates. In this particular example, the CARLA emulation frame rate was chosen to be 60 frames per second. The jitter is also fairly low in comparison to the latency. The goodput, i.e. the application data rate is understandably higher for the Controller as it sends WSignalBlock frames that are slightly larger than the WCANBlock frames.

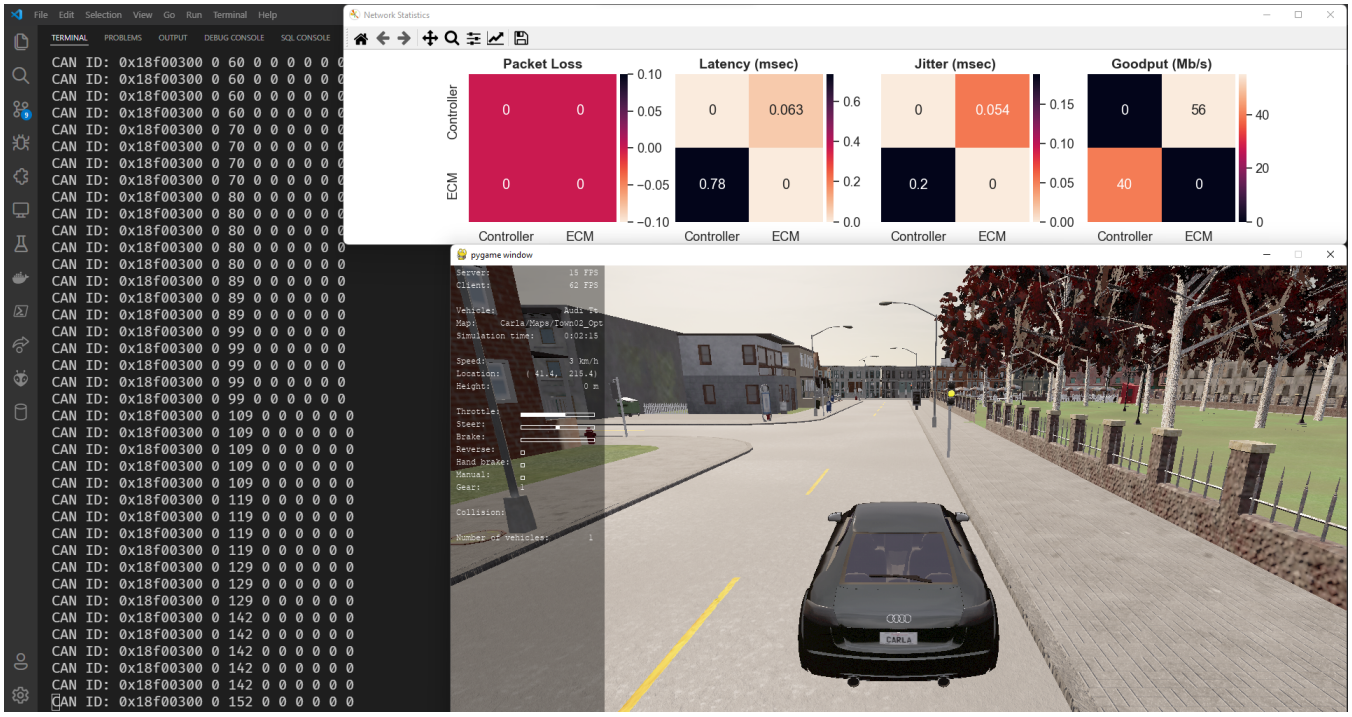

Figure 6: CANLay at work in the Software Defined Truck

4 Conclusion and Future Work

In this paper, we described the concepts behind the design of CANLay, the networking backbone for the Software Defined Truck. SDT is a virtualization based experimentation framework for CAN-based security experiments and CANLay is the carrier of physical control and CAN data over long distance networks. Essentially CANLay enables network virtualization for SDT. CAN is a reliable and low-latency network. CANLay does not explicitly ensure reliability and low latency, but provides a health monitoring service that provides real-time measures of network parameters to the user. This allows the user to make critical decisions about the state of the experiment they are in.

We believe more than one additional works can still be done on CANLay. Need response Dynamic buffer adjustment