CANLay: A User-centered Overlay for In-Vehicle Data Dissemination in a Network Virtualized Testbed

Abstract

Testing and evaluating security solutions for devices that utilize in-vehicle networks often require setting up different configurations and topologies for tests. An efficient strategy to rapidly create multiple testing setups is to employ a software-defined vehicle and encapsulate in-vehicle networking messages in a switched packet network, like Ethernet. The purpose of this paper is to present the design of CANLay, the networking backbone of a software-defined vehicle, which is a network virtualized test bed for in-vehicle network security testing. CANLay, is responsible for carrying controller area network data as well as sensor signals emanating from the vehicle control units and sensor systems. CANLay acts as a communication media between a vehicle simulator and real-world electronic control units located on different network segments. The measurement of network performance for CANLay show the technique has high reliability and low latency for local network connections. Applications related to heavy vehicle networks are demonstrated to effectively show utilization of CANLay for testing.

1 Introduction and Background

In recent years, security of in-vehicle networks using the Controller Area Network (CAN) has been widely discussed. CAN is a broadcast media that enables reliable and low-latency communication between in-vehicle devices, also referred to as electronic control units (ECUs). The broadcast nature of CAN, along with the fact that it is inherently unauthenticated, makes it susceptible to network based cyber-attacks. Security researchers have shown [3,4,14] that remote interfaces on modern vehicles can be used to invade 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 have been pursued to detect and/or prevent this scenario from occurring.

To evaluate the effectiveness of their proposed techniques, researchers have typically experimented on real-vehicles or homegrown test bed setups that mimic real vehicles. While most households in the United States have at least one passenger car [12], this is not the same for medium and heavy-duty (MHD) vehicles. Moreover, creating homegrown test beds is both logistically and economically challenging. To that end, the need for a publicly accessible test bed is imminent. This is where the concept of the Software Define Truck (SDT) [13] is useful for researchers in 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 1. CAN frames and physical signals need to be exchanged between ECUs and vehicle simulators located in different subnetworks

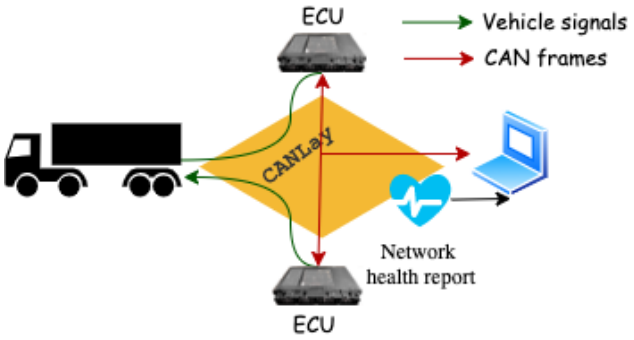

Figure 1: Purpose of CANLay

around the globe. CANLay is the networking backbone of the Software Defined Truck and provides the necessary infrastructure to enable this service.

Previous research has established two critical criteria for quality evaluation of automotive networking test beds: fidelity and adaptability [11]. 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 challenging to optimize both. An adaptable system requires virtualized components that adjust quickly to suit users' 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 feature allows the user to assess the fidelity of the overlay in terms of standard networking metrics like latency, rate of packet drop, etc.

CAN is newer than the TCP/IP communication technology and has a smaller application scope than TCP/IP. However, there have been some proposals to virtualize its operations. First, there has been the attempt to adapt the software-defined networking paradigm for CAN [5, 8, 16]. This approach is largely 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 [7, 9]. 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 test beds [2] have been proposed, but the signals from the simulators have been transported over physical connections, not over reconfigurable, long-range network overlays.

CANLay has the following functional objectives:

- Transport of CAN frames over a distributed overlay network of electronic control units
- Transport of sensor signals to a distributed network of electronic control units
- Supporting the creation of these overlays on-demand
- Provision of runtime metrics to estimate the network health during the ongoing experiment.

In the rest of this paper, we describe the design of CANLay (section 2), provide a discussion on its usage in an example scenario (3), and finish with conclusive remarks and future directions.

2 Design and Development

Figure 2 shows the proposed system design of CANLay. The system serves three functions: 1) offline configuration of the network overlay, 2) CAN frame exchange at runtime, and 3) vehicle signal exchange at runtime. Next, we will describe the components and their roles in the system. Following that, we will describe the behavioral aspects of the system. Together, these aspects combine to accomplish the functional objectives.

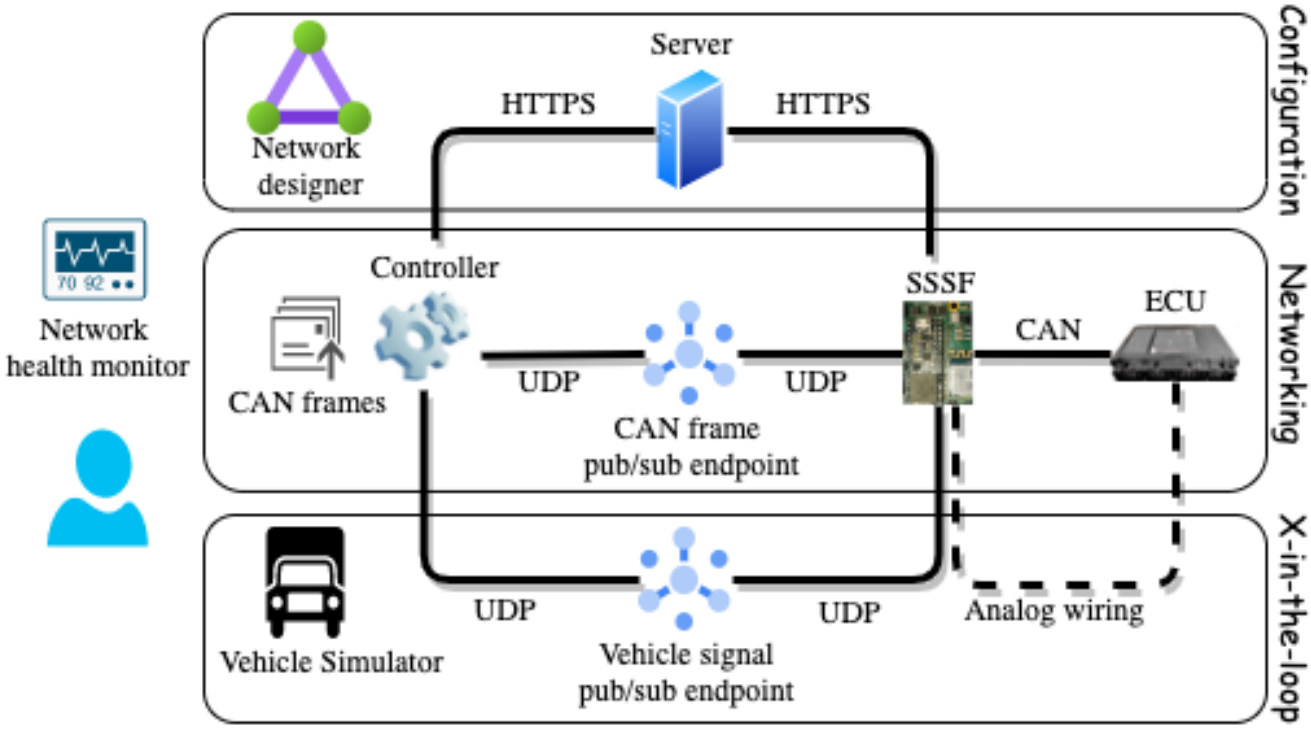

Figure 2: Proposed System

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 to be accessed by the CANLay system. In an active experiment, the SSSF 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 transmit along the analog wiring shown using a dashed line Figure 2. The second type is CAN data carried between the ECUs in the current experiment. Through the SSSFs, multiple ECUs can communicate with each other to create a rich testing environment.

The SSSF is an open hardware device designed to interface with the in-vehicle network and simulate sensor inputs to an ECU at the same time. It utilizes the NXP K66 processor with two CAN channels. This processor was used in the Teensy 3.6 development board, which has an add-in for the Arduino Integrated Development Environment (IDE). Description and evolution of the Smart Sensor Simulator can be found in reference [15]. The hardware and firmware are hosted on Github [1]. Part of the firmware includes operations for syncronizing the real-time clock on the 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 User Interface (UI) is used to assist the user in building their virtual test bed. It does so by communicating with the central Server over hypertext transfer protocol secure (HTTPS). Once the experiment setup finishes, the Controller transitions to acting as a gateway for a graphical vehicle simula-

tor to communicate with the CANLay system by forwarding the simulator's outputs to the publish/-subscribe (pub/sub) endpoint and listening for CAN messages from the same.

The Controller is a multithreaded graphical application built using Python. It exposes two UI components, namely, the Network Designer and the Network Health Monitor. The Controller manages the execution of the Vehicle Simulator to ensure that it stays "in-step" with the flow of signals produced. In addition, the Controller is in charge of updating the Network Designer's catalog of available devices and relaying the virtual network designs of the user to the Server. The Controller efficiently manages the multiple streams of incoming and outgoing data using Selectors. Selectors enable the Controller to know when a socket is available to read or write. Finally, once a session has begun, the Controller manages the aggregation and presentation of network health reports. The Controller collects the current statistics from all devices in the session and displays the results using heat matrices. Further discussion of the heat matrices is included in the description of Figure 3.

2.1.3 Server

The Server helps in setting up the publishers and subscribers for an experiment. Each device opens and maintains a persistent transmission control protocol (TCP) connection with the Server while participating in the CANLay system. After establishing the TCP connection, the devices communicate with the Server through HTTP application programming interfaces (API). The Server can monitor the health of the devices and take action if a device is malfunctioning or goes offline. This mechanism also allows the Server to keep track of free SSSFs and pub/-sub endpoints to validate new experiment requests and allocate the requested devices and endpoints. This technique avoids the race conditions or double use issues that may arise if each Controller was in charge of gathering the requested SSSFs for its experiment. Finally, the Server keeps track of ongoing sessions and ensures the proper closure of sessions when a device is experiencing issues.

The Server is a single-threaded session broker that multiplexes the handling of HTTP API calls using Selectors. We chose HTTP API calls because they clearly define the target object to invoke and how to invoke it. The devices perform all necessary setup and teardown actions such as registering, deregistering, starting and stopping a session by sending HTTP requests to different API endpoints. The Server responds to these requests using standard HTTP response messages and codes that are easily interpretable 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. We chose the pub-/sub model because it can easily emulate the broadcast nature of CAN [10] 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.

We used two criteria to find a suitable pub/sub mechanism that resembles a CAN network. First, 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 [10]. Next, the transport mechanism must permit the devices to receive messages from one or more devices while maintaining 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 costs for the implementers, but the current goal was to test its usability and make future decisions based on the observed performance. Other potential pub/sub implementations, such as MQTT, may also achieve the goals for the pub/sub system.

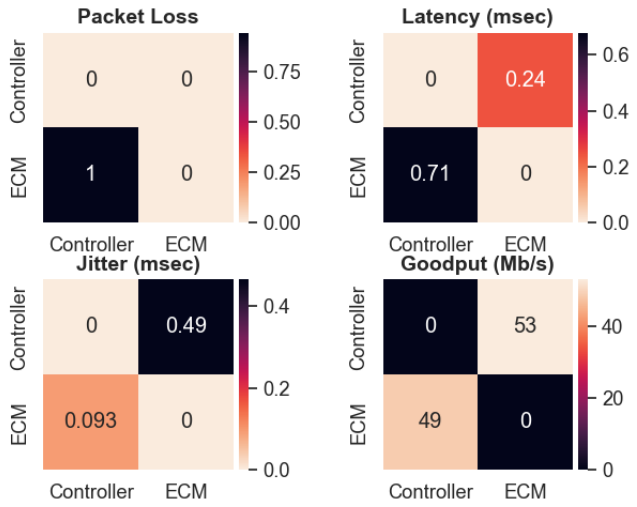

Figure 3: Network Matrices showing packet loss, latency, jitter, and goodput.

2.1.5 Front-End Components

As seen in Figure 4, the Network Designer consists of a catalog of available devices and the ability to select the requested devices. Upon startup, the Controller contacts the Server and solicits the set of available devices. The Controller then presents these options to the user and allows them to select the requested devices from the catalog of available ECUs.

As seen in Figure 3, 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 matrix represents a directed communication channel of the network. The color of the cells depends on their values. For packet loss, latency, and jitter (significance and method of evaluation for these metrics are described later) 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 the underperforming parts of the network.

[1]The network designer is controlled through the command line but in the future, a GUI could be built to make the system more user-friendly.

2.2 Behavior Descriptions

2.2.1 Network Setup (ref. figure 5)

While the Server is up and running, SSSFs connect to it. SSSFs perform a setup procedure by reading their inbuilt SD card. The type, year, make, model, and the serial number of the connected ECUs are read from 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 */SSSF/Register*. If the registration fails, the Server responds with an HTTP 4XX error code indicating why the registration was unsuccessful. Otherwise, the Server responds with an HTTP 202 code indicating the SSSF was accepted. Once successfully registered, the SSSF waits for further instructions from the Server on its HTTPS port.

The Controller begins by registering with the Server similarly 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 requests a list of the available[2] ECUs. After receiving the list of available devices, it presents the available ECUs to the user via the Network Designer described earlier. Notice that while the Server deals with SSSFs, 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 an HTTP POST and waits for the Server's reply.

The Server receives the list of requested devices and performs three checks. First, it checks that the request is coming from a registered Controller. Controllers are the only devices allowed to start sessions. Next, the Server double-checks that the devices are still available. If any of the devices are no longer available or become unavailable during the setup process, the Server responds to the Controller with the error code 409, indicating a conflict in the selection. If all the devices are still available, the

[2]If an SSSF device is currently being used in another experiment it is not considered available.

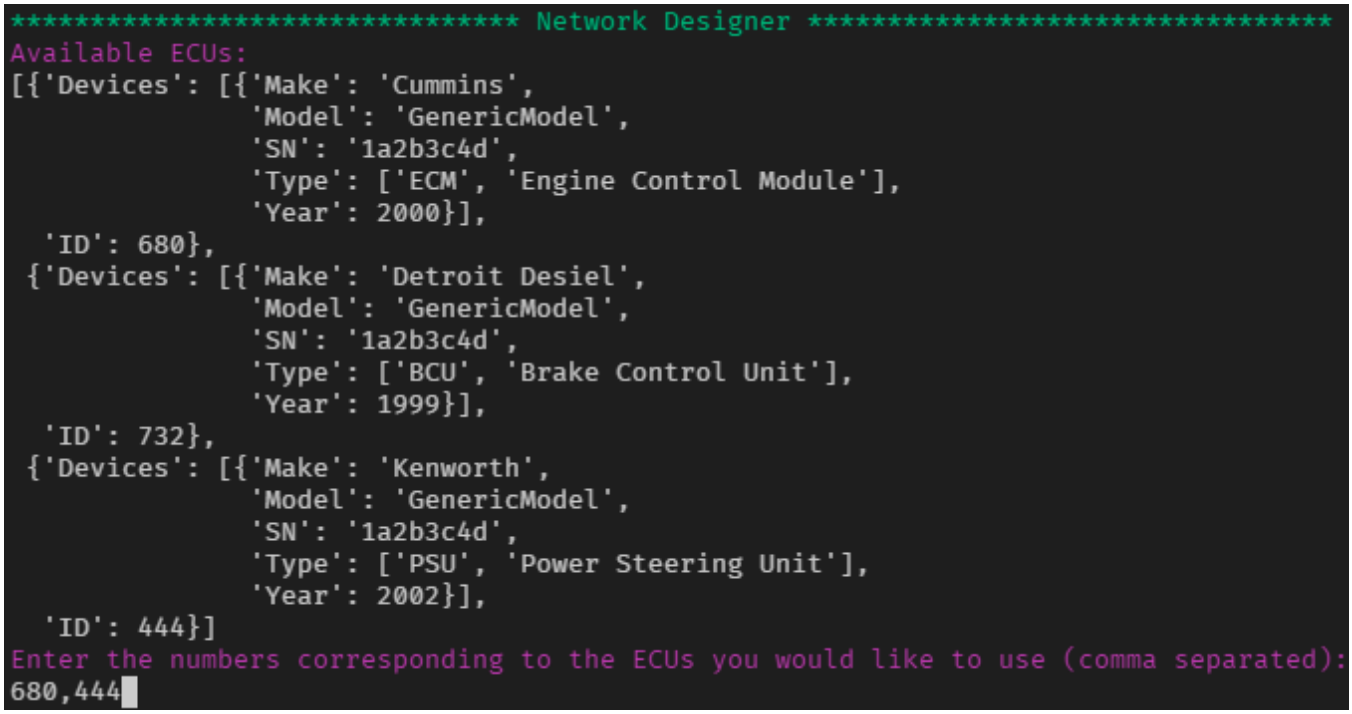

Figure 4: Network Designer displaying available ECUs[1]

Server then selects a vacant pub/sub endpoint for the experiment and assigns an index to each device. The index aids the collection of network statistics which is explained later on. Now the Server sends the connection data to the Controller and SSSFs. Connection data contains a unique identifier (ID), the index, a multicast IP address and port acting as the pub/sub endpoint, and a list containing the IDs and attached devices of other nodes in the experiment.

Once endpoints receive the connection data, they resyncronize time 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.

2.2.2 Sensor Signal Communication (ref. figure 7)

Once the Controller establishes an active session, it begins forwarding sensor signals to the pub/sub endpoint. Before sending out the sensor signals, the Controller wraps them in a WsensorBlock and then a COMMBlock. There are several additional pieces of information the COMMBlock requires, but we will focus on type and frame number for now. The type indicates the subclass that the COMMBlock is carrying. In this case, the type will be 2, implying that it is transporting a WsensorBlock. The frame number is added to the COMMBlock and incremented by one every time the Controller forwards a sensor message to the CANLay network. When SSSFs send out CAN messages, they will set their outgoing COMMBlock's frame number to the last received frame number from the Controller. When the Controller receives a CAN message, it will read the frame number field and know the latest sensor message that the SSSF received. This technique creates an acknowledgment feedback loop enabling the Controller to tell which devices have received a frame and when a frame was lost. This acknowledgment feedback loop is possible because CAN messages get sent at a higher rate than the sensor message are. The benefit of this acknowledgment mechanism is that it does not require additional

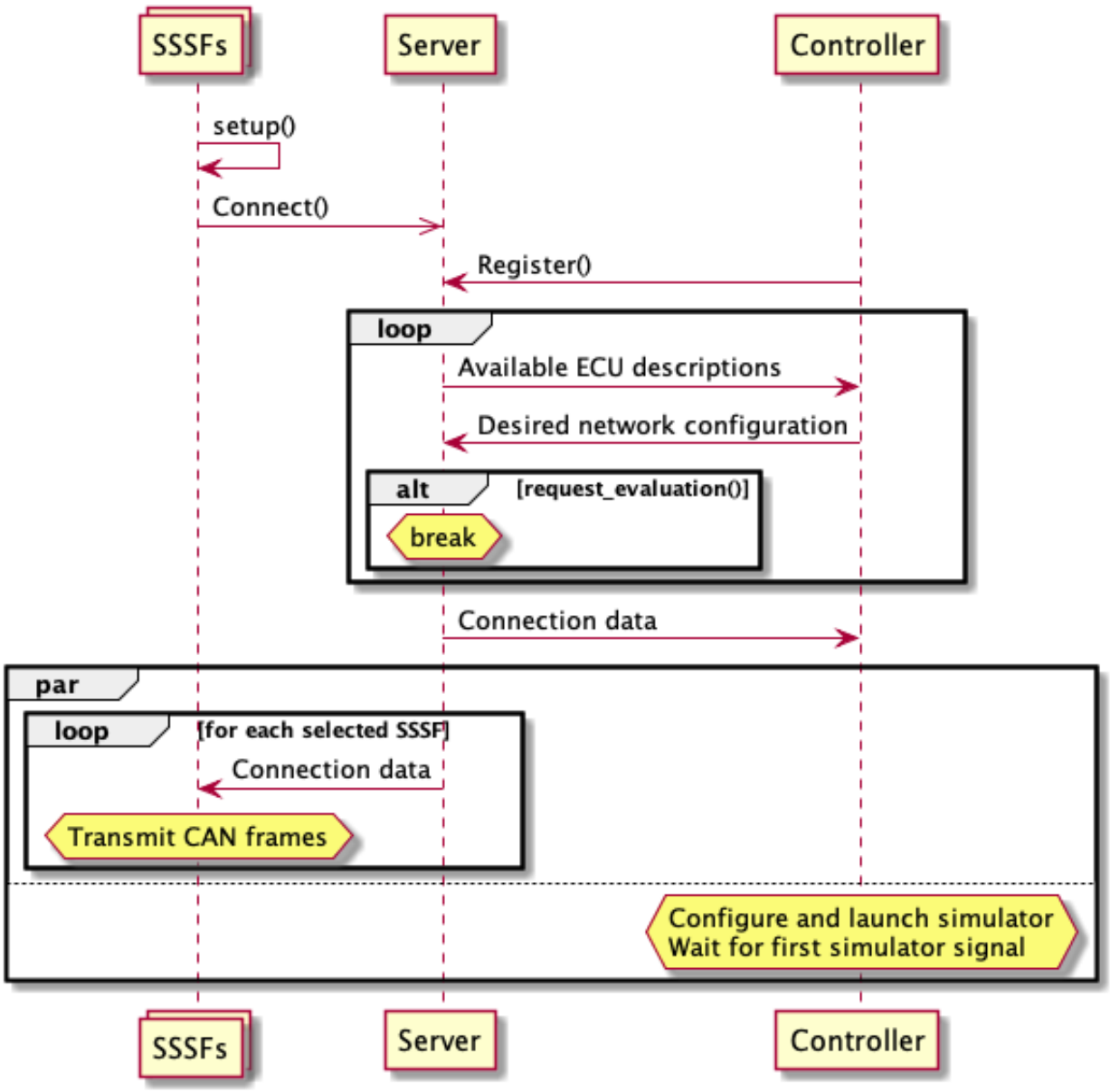

Figure 5: Network Setup Activity

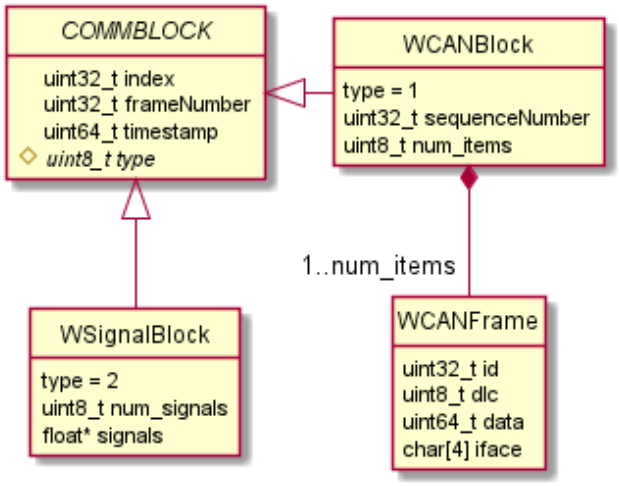

Figure 6: Transport Data Structures

replies from the receiver.

Before starting the session, the Controller lets the user select the maximum number of retransmissions. This number gets factored into the timeout value of a sensor frame. The timeout value is calculated as follows: $1/(simulator\_frame\_rate * max\_retransmitions)$. When the Controller detects that the last frame has timed out, it first checks if it has reached its maximum number of retransmissions. If not, it will resend the sensor message, increment the number of retransmissions it performed, and reset the timeout timer. It will repeat this process until it receives a CAN message with a frame number equal to the Controller's current frame number or the Controller reaches the maximum number of retransmissions. In the latter case, the frame is considered lost, and a counter `lost_simulator_frames` is updated to later show to the user.

As mentioned before, when an SSSF receives a sensor frame, it first sets its last seen frame number equal to the message's frame number. Next, the SSSF applies any necessary transformations to the sensor frame before forwarding it onto its device's CAN network(s). For 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 closest matching PGN CAN frame.

```
1 networkEdge.min = min(networkEdge.min, n);
2 networkEdge.max = max(networkEdge.max, n);
3 networkEdge.count++;
4 delta = n − networkEdge.mean;
5 networkEdge.mean += delta / networkEdge.count;
6 delta2 = n − networkEdge.mean;
7 networkEdge.sumOfSqrdDiffs += delta ∗ delta2;
8 networkEdge.variance = networkEdge.
      ↪ sumOfSqrdDiffs / networkEdge.count;
```

Listing 1: calculate_health(HealthCore &networkEdge, n)

2.2.3 CAN Communication (ref. figure 8)

When the SSSF is in an active experiment, it attempts to read a message from the CAN network. Upon doing so, it creates the COMMBlock data structure (ref. figure 6) that will get written to the CANLay network. Additional information is added to the COMMBlock before its written to the network. Several pieces of information are required, but we will focus on two right now, namely, type and sequence number. The type indicates the subclass that the COMMBlock is carrying. In this case, the type will be 1, implying that it is carrying a WCANBlock. The sequence number is added to the WCANBlock and incremented every time a CAN message gets sent from the device. The packet loss mechanism described later depends on this sequence number.

After the SSSF finishes checking for CAN messages from the CAN network, it checks for messages from the pub/sub network. Upon receiving a CAN message wrapped in the `w_can_block`, the SSSF updates its network statistics, extracts the WCANFrame, and writes it onto its available CAN networks.

2.2.4 Network health monitoring (ref. listings 1 and 2)

Monitoring the health of the network is key to ensuring that unnecessary delay and packet loss are not affecting the output of the experiment. Each message structure from figure 6 gets loaded with ad-

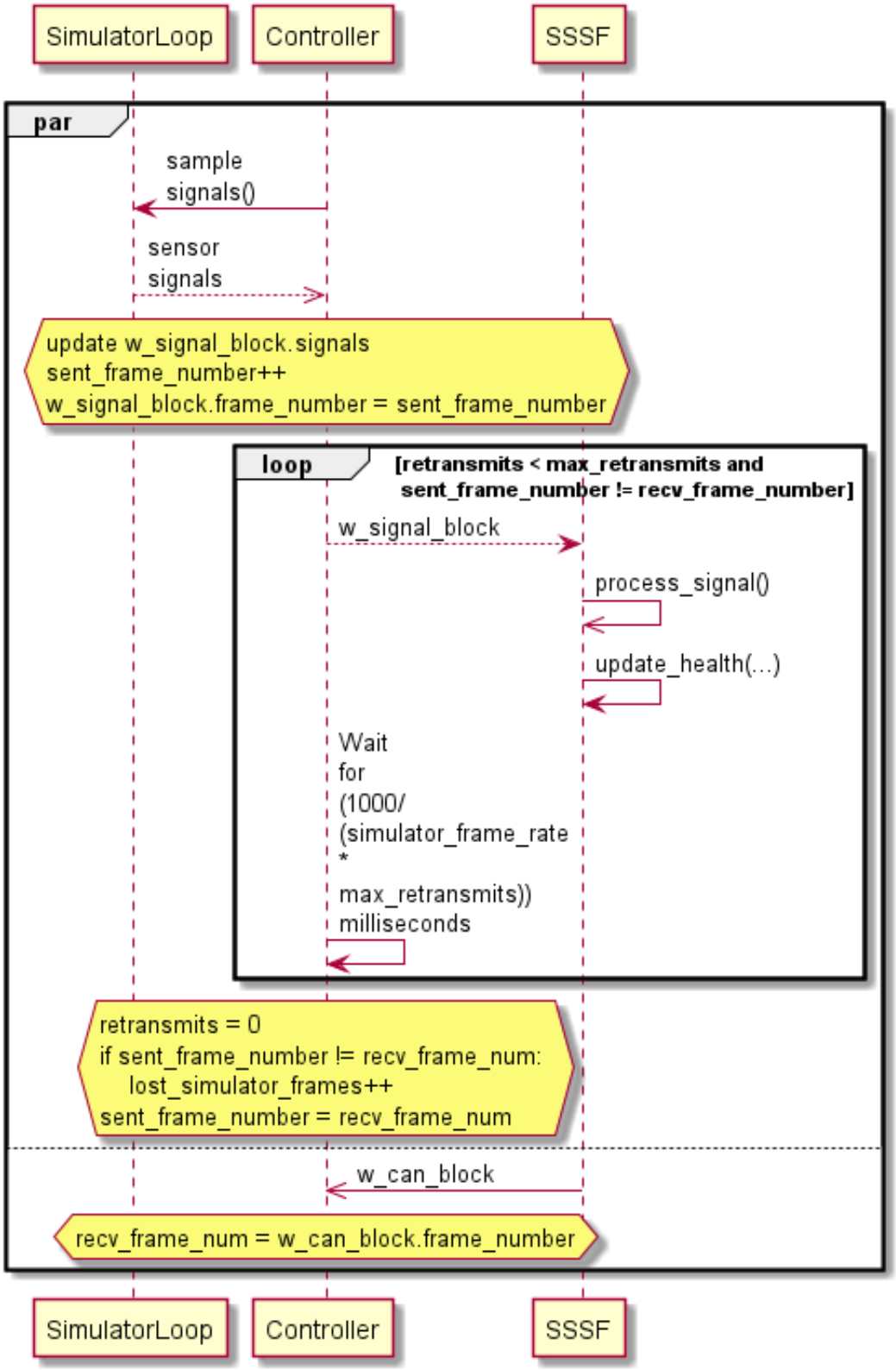

Figure 7: Signal Tranmission Activity

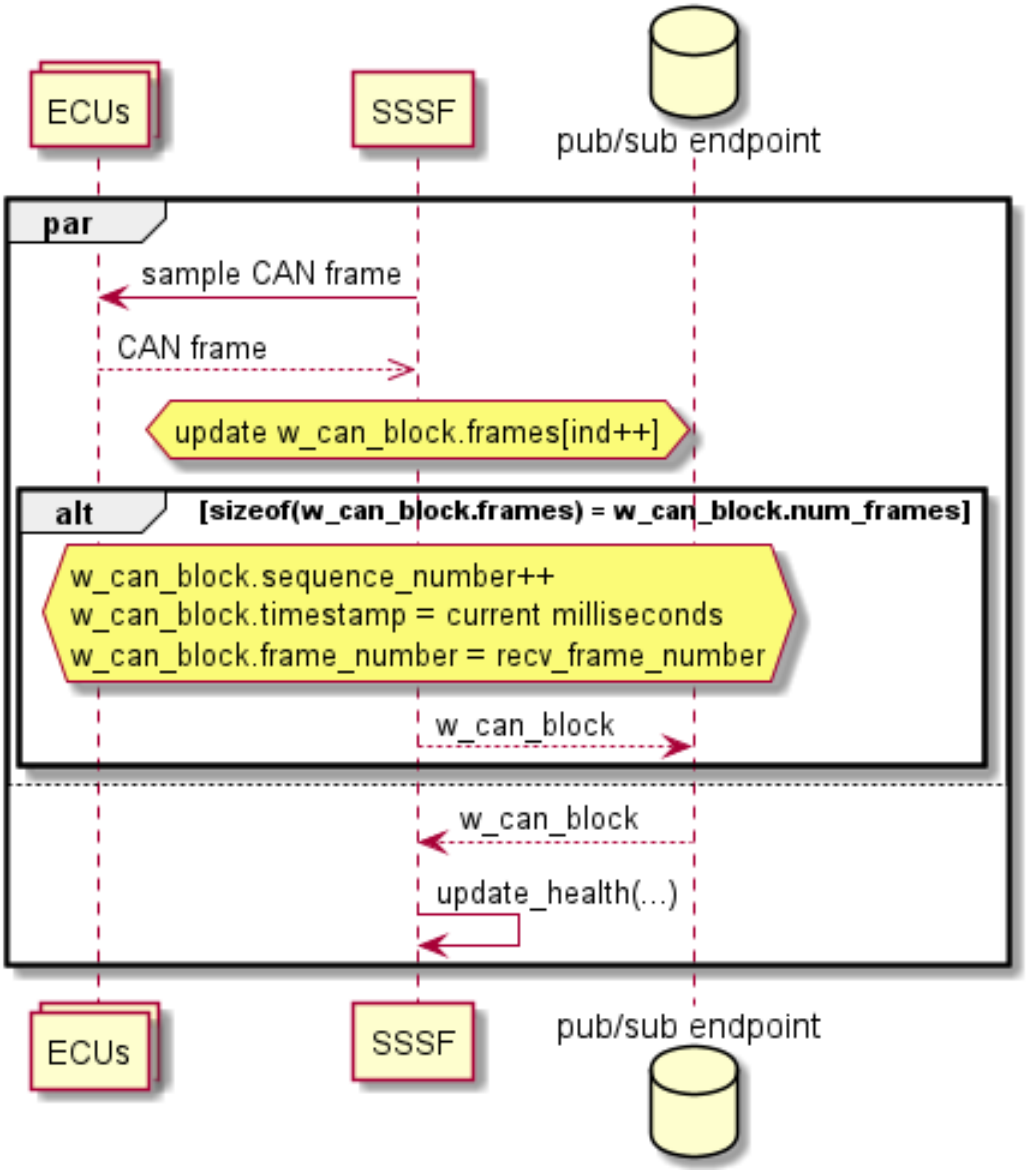

Figure 8: CAN Communication Activity

```
1  now = timeClient->getEpochTimeMS();
2  delay = now - ts;
3  ellapsedSeconds = (now - HealthBasics[ind].
       ↪ lastMessageTime);
4  ellapsedSeconds /= 1000.0;
5  calculate(HealthReport[ind].latency, abs(delay));
6  calculate(HealthReport[ind].jitter,HealthReport[ind].
       ↪ latency.variance);
7  pcktsLost = seq - (HealthBasics[ind].
       ↪ lastSequenceNumber + 1);
8  HealthReport[i].pcktLoss += (pcktsLost > 0) ?
       ↪ pcktsLost : 0;
9  calculate(HealthReport[ind].goodput,(packetsz * 8)/
       ↪ ellapsedSeconds);
10 HealthBasics[i].lastMessageTime = now;
11 HealthBasics[i].lastSequenceNumber = seq;
```

Listing 2: update_health(ind, packetsz, ts, seq)

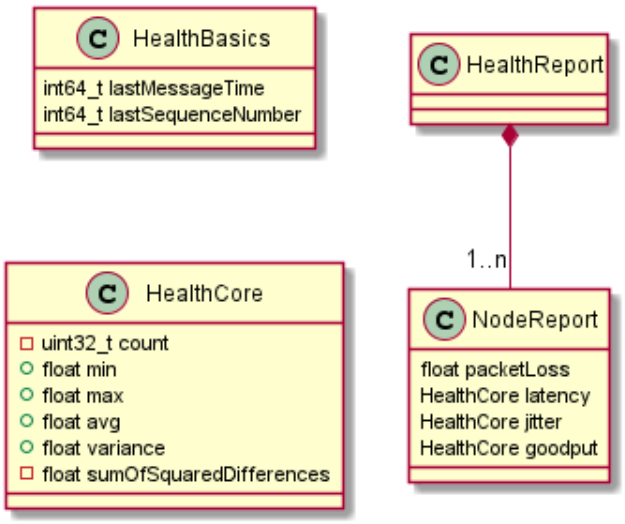

Figure 9: Network Health Data Structures

ditional information enabling the devices to collect network statistics during an active experiment. The first piece of additional information is a frame or sequence number. When other devices spot gaps in these numbers greater than 1, they know that a message was lost. Next, the timestamp gets included to allow devices to calculate the latency along the network edge from the sending device to the receiving device without interrupting the testing. The latency is calculated by subtracting the time at which the message was sent from the time at which the message was received. To ensure accuracy every device on the CANLay network implements NTP to synchronize their clocks.

These indicators enable each device to calculate four network statistics for every other device on the network. Currently, these statistics include *packet loss*, *latency*, *jitter*, and *goodput*. Packet loss is the number of packets determined to be lost along a network edge. Latency is the time it takes a message to travel from the sender to the receiver. Jitter is the variance in latency. There are different types of network jitter measures, but we use the simplest form, which is often called packet jitter or constant jitter, which is "the variation in latency as measured in the variability over time of the end-to-end delay across a network"[3]. Finally, goodput is the measurement of application-level throughput. In our case, it is calculated in megabits per second.

Functions for calculating the network statistics are shown through the code listings 1 and 2, the latter being called upon receipt of every COMMBLOCK. Associated data structures are shown in figure 9. The algorithm shown in listing 1 is based on Welford's Online Algorithm [18], a numerically stable algorithm. Such an algorithm is required when computing the running mean and variance so floating-point errors do not grow without bound. We chose to calculate the mean and variance in one pass to keep the time required for calculation within several microseconds. Storing each message's metrics and looping through the previous metrics to recalculate the mean and variance would take too long. The latency is calculated on line 5 of listing 2 using the delay between when the message was sent and when the message was received. Once we've updated the statistics for latency, we can calculate jitter using the variance in the latency as shown on line 6. Next, packet loss is calculated on lines 7-8. A packet is considered lost if the gap between the new sequence number and the old sequence number is greater than one. If `pcktsLost` is negative, it

[3] https://networkencyclopedia.com/jitter/

indicates a duplicate or an out-of-order frame. In that case, we neglect it. Afterward, we calculate goodput using the size of the packet received and the number of seconds that have elapsed since that last message.

To display these captured statistics to the user, the Controller needs to aggregate the health reports from every device. The Controller does so every second by sending out a *health request* message to the pub/sub endpoint. Each device responds to the request if they receive it. After responding, they reset their local state variables, keeping only the last message timestamp and the latest seen sequence number. This technique creates a measurement period of one second. As the Controller receives the health reports, it updates its internal memory and displays the results to the user through the network monitoring window.

3 Exemplary Usage Scenario

Figure 10 shows CANLay at work. The windows in the figure display CAN frames on the left, the vehicle simulator on the bottom right, and CANLay's network health monitoring on the top right. Each of these components was already introduced in figure 2. For the current purpose, we have been using the CARLA graphical vehicle simulator [6]. 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 [17] 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.

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. Each WCANBlock wraps exactly one CAN frame. The figure shows no packets were lost while the latency in the last cycle of health report collection was less than a milliseconds between the endpoints. Although CANLay does not explicitly perform any latency-reducing functions, during this and other test scenarios, we noticed that the SSF and Controller forward information in the sub-milliseconds. Researchers before us have observed the same when designing CAN-to-ethernet gateways [7]. In that light, the delay is purely due to transmission in the overlay. A latency of less than a millisecond 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 ECM as it sends many more frames compared to the Controller.

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

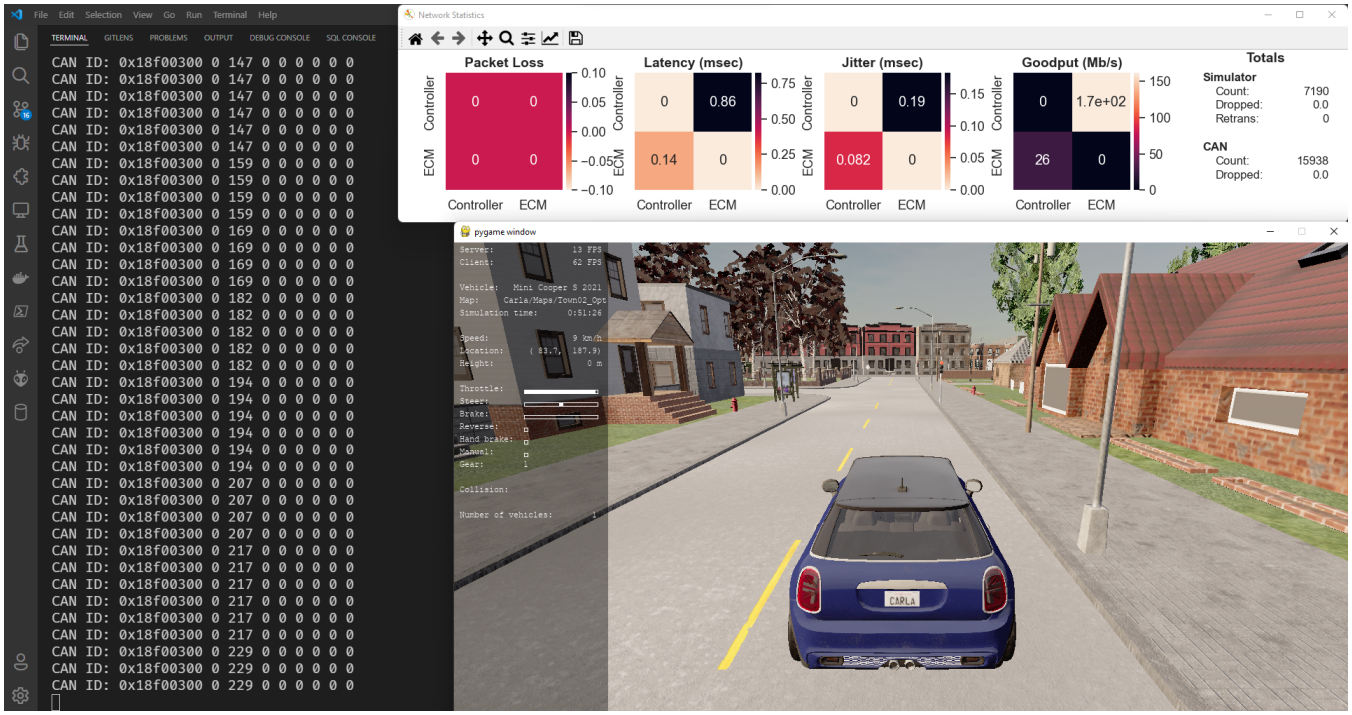

Figure 10: CANLay at work in the Software Defined Truck

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 have identified at least three enhancements to CANLay's design that are to be pursued next. Firstly, we want to enable routing of actuation signals from the ECU to the vehicle simulator such that the effect ECU actuation can be visualized. We believe that the main challenge in implementing this functionality will be to enable retransmissions. Recall that at this time, CANLay piggybacks acknowledgments on CAN frames and relies on the fact that CAN frames are transmitted by the recipient ECU at a faster rate than which signals are produced by the simulator. The recipient for the actuation signals will be the Controller, but it may not transmit periodic CAN frames at a reasonably high rate. Secondly, we aim to enable a selective acknowledgment scheme for CAN frames. That is, enforce acknowledgment for some frames identified critical by the user, such as a frame requesting engine control in real-time. Finally, we aim to introduce a dynamic buffer adjustment scheme when accumulating CAN frames in a WCANBlock. This is to make optimal use of the available bandwidth while supporting high CAN busloads.

References

[1] Mini-sss3: https://github.com/systemscyber/mini-sss3.

[2] Matthew Appel, Pradeep Sharma Oruganti, Qadeer Ahmed, Jaxon Wilkerson, and Rubanraj Sekar. A Safety and Security Testbed for Assured Autonomy in Vehicles. In *Proceedings of the WCX SAE World Congress Experience*, page 8, 2020.

[3] Yelizaveta Burakova, Bill Hass, Leif Millar, and Andre Weimerskirch. Truck Hacking: An Experimental Analysis of the SAE J1939 Standard. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*, pages 211–220, Austin, TX, USA, 2016. USENIX Association.

[4] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462, San Francisco, CA, USA, 2011. USENIX Association.

[5] Michael Doering and Marco Wagner. Retrofitting SDN to classical in-vehicle networks: SDN4CAN. Technical report, Universitat Tubinge.

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[7] Florian Polzlbauer and Allan Tengg. Experience Report: Lightweight Implementation of a Controller Area Network to Ethernet Gateway. In *Proceedings of the Brief Presentation Track of the RTAS'19 Conference*, MONTREAL, CANADA, 2019. IEEE.

[8] Dennis Grewe, Naresh Nayak, Deeban Babu, Wenwen Chen, Sebastian Schildt, and Clemens Schroff. BloomyCAN: Probabilistic Data Structures for Software-defined Controller Area Networks. In *2021 IEEE 94th Vehicular Technology Conference (VTC2021-Fall)*, pages 1–6, 2021.

[9] Mathias Johanson, Lennart Karlsson, and Tore Risch. Relaying Controller Area Network Frames over Wireless Internetworks for Automotive Testing Applications. In *2009 Fourth International Conference on Systems and Networks Communications*, pages 1–5, 2009.

[10] J. Kaiser and M. Mock. Implementing the real-time publisher/subscriber model on the controller area network (CAN). In *Proceedings 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99) (Cat. No.99-61702)*, pages 172–181, 1999.

[11] Shahid Mahmood, Hoang Nga Nguyen, and Siraj A. Shaikh. Automotive Cybersecurity Testing: Survey of Testbeds and Methods. In *Digital Transformation, Cyber Security and Resilience of Modern Societies*, volume 84, pages 219–243. Springer International Publishing, Cham, 2021.

[12] Mathilde Carlier. Number of vehicles per household in the United States from 2001 to 2017, August 2021.

[13] Subhojeet Mukherjee and Jeremy Daily. Towards a Software Defined Truck. In *Proceedings of the 31st Annual INCOSE International Symposium*, page 16, Online, 2021. INCOSE.

[14] Subhojeet Mukherjee, Hossein Shirazi, Indrakshi Ray, Jeremy Daily, and Rose Gamble. Practical DoS Attacks on Embedded Networks in Commercial Vehicles. In *International Conference on Information Systems Security*, pages 23–42, Jaipur, Rajasthan, India, 2016. Springer.

[15] Ram Rohit Gannavarapu. Secure remote sensor simulator for heavy vehicle electronic control units, December 2021.

[16] Randolf Rotermund, Timo Häckel, Philipp Meyer, Franz Korf, and Thomas C. Schmidt. Requirements Analysis and Performance Evaluation of SDN Controllers for Automotive Use Cases. In *2020 IEEE Vehicular Networking Conference (VNC)*, pages 1–8, 2020.

[17] Society of Automotive Engineers. SAE J1939 Standards Collection.

[18] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.