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

Abstract

Controller Area Network ...

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 [2, 3, 12] 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 proposed techniques, 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) [11] 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 1. CAN frames and physical signals need to be exchanged between ECUs and 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.

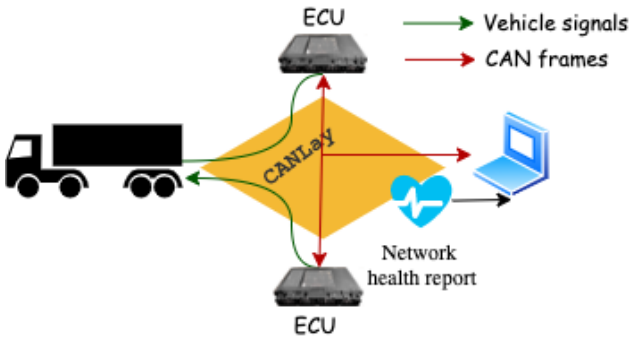

Figure 1: Purpose of CANLay

Previous research [10] 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

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

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 have been some proposals to virtualize its operations. First, there has been the attempt to adapt the software-defined networking paradigm for CAN [4, 7, 13]. 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 [6, 8]. 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 [1] have been proposed, but the signals from the simulators have been transported over physical connections, not over reconfigurable, long-range network overlays.

At this time, 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 health of the network 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 works.

2 Design and Current Development

Figure 2 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 2. The second type is CAN data carriers from the ECUs as well as 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 block diagram etc. CAN Forwarding The real-time clock on the SSF

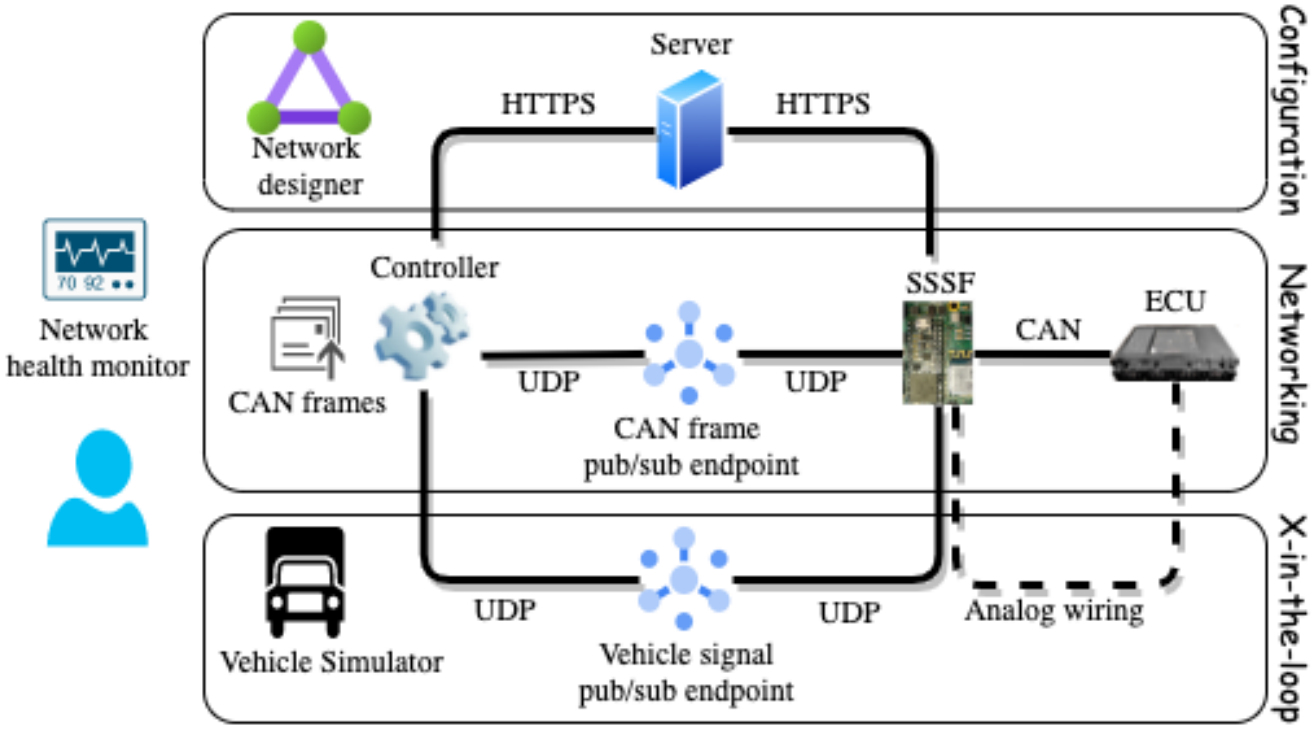

Figure 2: Proposed System

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 (UI) 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 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 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 ensuring that it stays "in-step" with the flow of signals being produced. In addition, the Controller is in charge of updating the Network Designer's catalog of available devices as well as relaying the virtual network designs of the user to the Server. The Controller efficiently manages the multiple streams of incoming and outgoing data through the use of Selectors. Selectors enable the Controller to know when a socket is available to read from or write to. 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 in the form of heat matrices. These heat matrices will be discussed in-depth later in the paper.

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 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 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 an experiment in the event a device is experiencing issues.

The Server is a single-threaded session broker that 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 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 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 [9] 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 two 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 [9]. The next

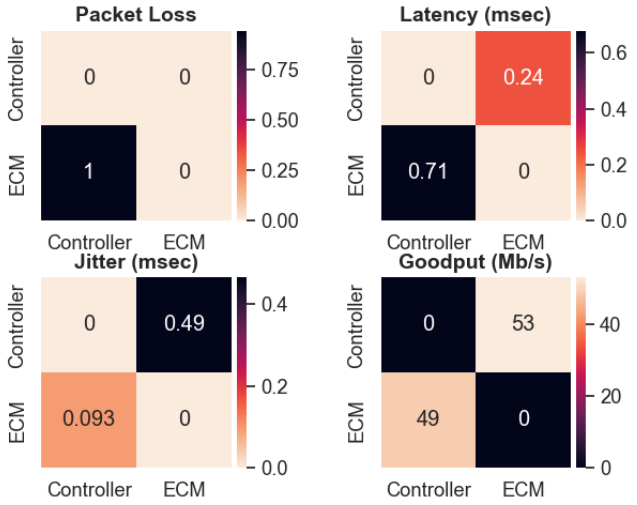

Figure 3: Network Matrices showing packet loss, latency, jitter, and goodput.

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 costs 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 another potential pub/sub implementations such as MQTT.

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 requests the latest 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.

[2]Currently, the network designer is controlled through the command line but in the future, a GUI will be added that

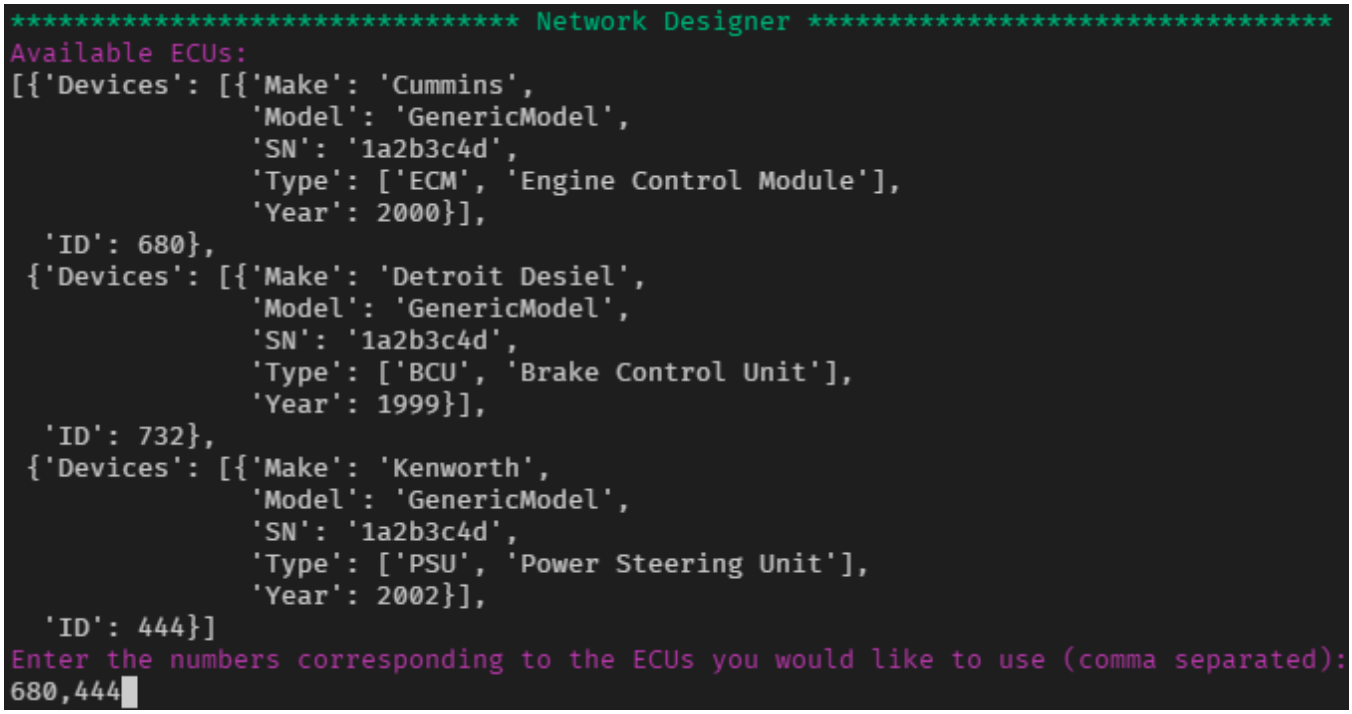

Figure 4: Network Designer displaying available ECUs[2]

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 matric 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.

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 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 */SSSF/Register*. If the registration fails, the Server responds with an HTTP 4XX error code indicating why the registration failed. 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 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 requests a list of the available[3] 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 the SSSF

performs the same function.

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

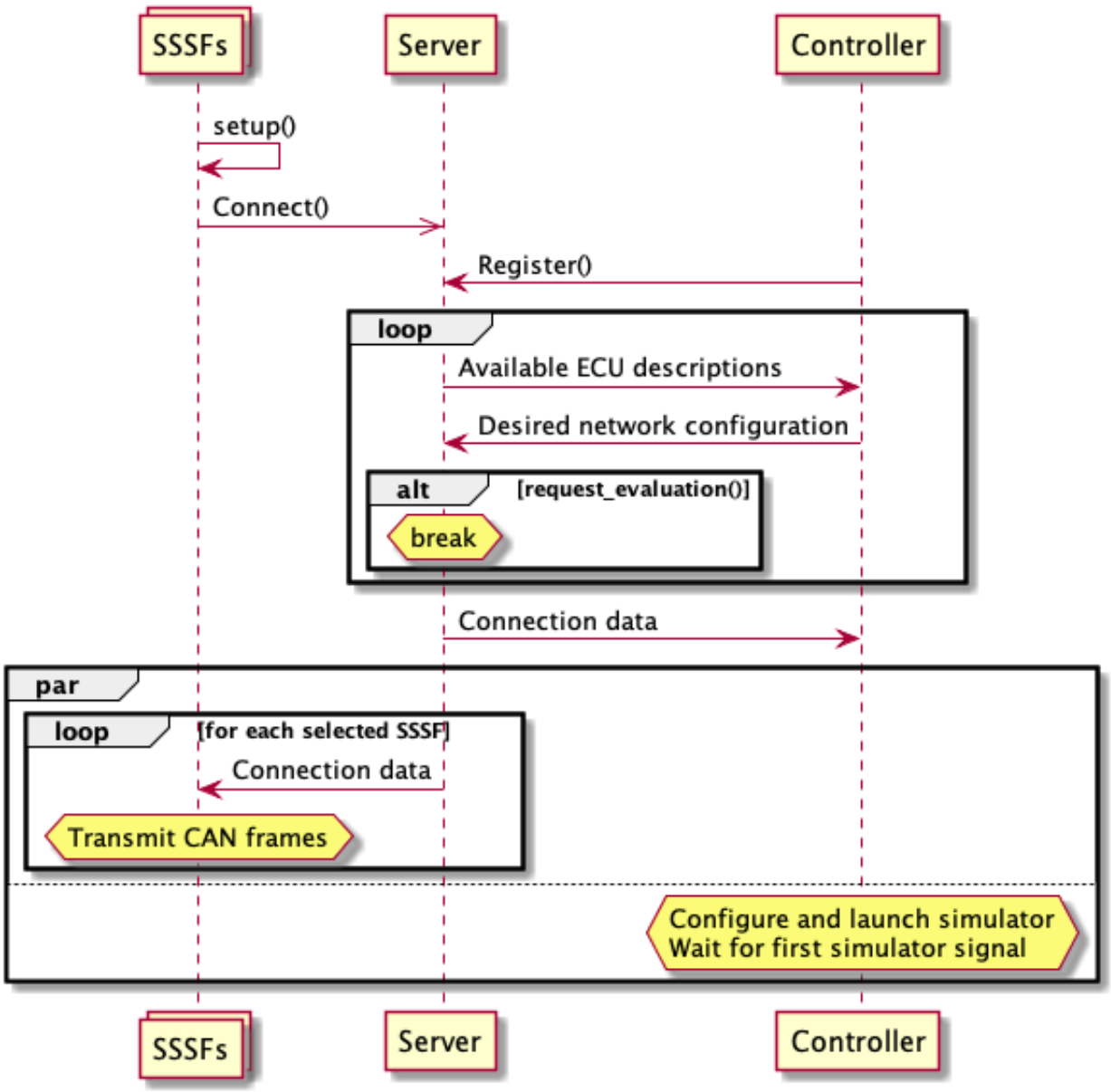

Figure 5: Network Setup Activity

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 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 there's a conflict in the selection. If all 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 aids 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 a unique identifier (ID) and 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 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.

2.2.2 sensor Signal Communication (ref. figure 7)

Once an active session has been established the Controller 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'll 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, indicating that it is carrying a WsensorBlock. The frame number is added to the COMMBlock

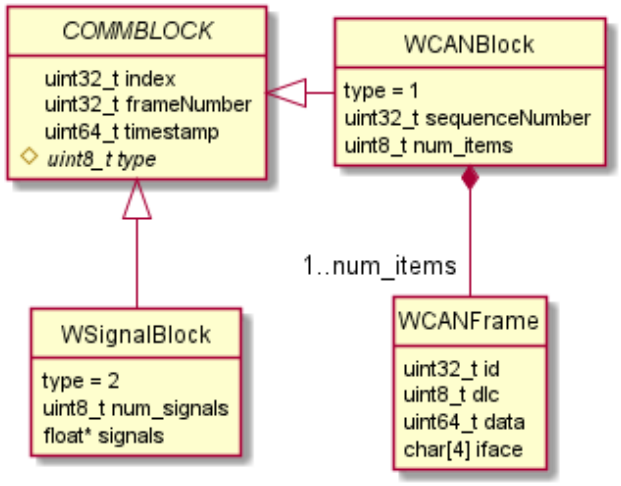

Figure 6: Transport Data Structures

and is incremented by 1 every time the Controller forwards a sensor message to the CANLay network. When SSSFs send out CAN messages they'll set their outgoing COMMBlock's frame number to the last received frame number from the Controller. When the Controller receives CAN messages, it'll read the frame number field and know the last sensor message that the SSSF received. This creates an acknowledgment feedback loop enabling the Controller to tell which devices have received a frame and when a frame has been lost. This acknowledge feedback loop is possible because CAN messages are sent at a higher rate than the sensor message are. This enables an acknowledgment mechanism that does not require any additional 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: $1/(simulator\_frame\_rate * max\_retransmitions)$. 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'll resend the sensor message, increment the number of retransmissions its performed, and reset the timeout timer. It'll repeat this process until it either receives a CAN message with a frame number equal to the Controller's current frame num-

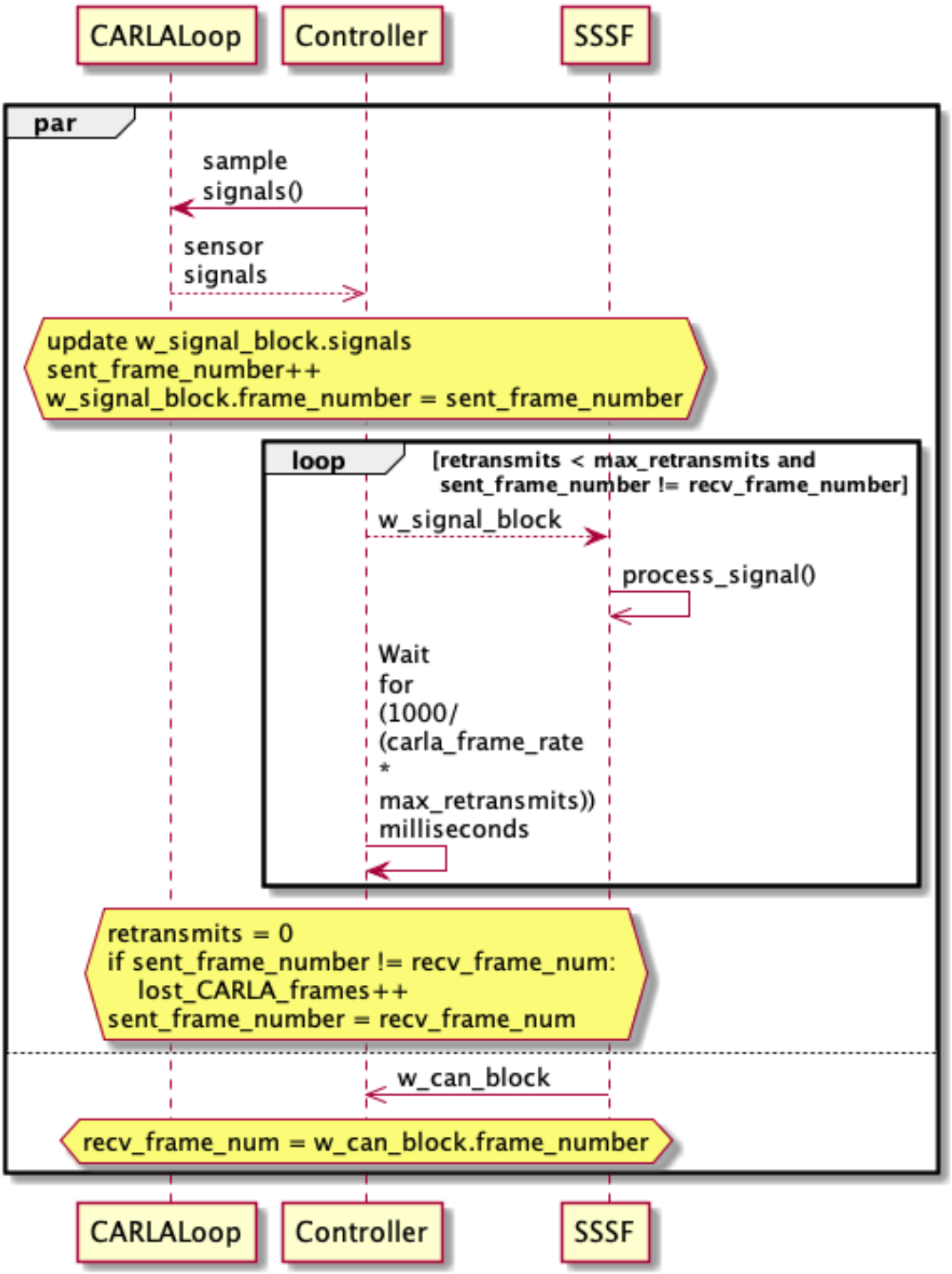

Figure 7: Signal Tranmission Activity

ber, or it reaches the maximum number of retransmissions. In the later case, the frame is considered lost and a counter `lost_simulator_frames` is updated to later show it 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 the purpose 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 closest matching PGN CAN frame.

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 be written to the CANLay network. Before the COMMBlock is written to the network additional information is added. There are several additional pieces of information required, but we'll 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, indicating that it is carrying a WCANBlock. The sequence number is added to the WCANBlock and is incremented every time a CAN message is sent from the device. It is included to detect pack loss, a mechanism that is described later.

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. Upon reciept of 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

Monitoring the health of the network is key to ensuring that unnessecary delay and packet loss are not affecting the output of the experiment. In order to enable the devices to collect network statistics during an active experiment, each message strcuture from figure 6 is loaded with additional information. The first piece of additional information is a frame or sequence number. When other devices spot gaps in these numbers larger than 1 they know that a message has been lost. The next important metric included in the COMMBlock messages is a timestamp. The timestamp is included to allow devices to calculate the latency along the network edge from the sending device to the receiving device. The timestamp is included with each message that is sent so that network health metrics can be calculated 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.

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

Listing 1: calculate_health(HealthCore &edge, n)

```
1  now = timeClient−>getEpochTimeMS();
2  delay = now − timestamp;
3  ellapsedSeconds = (now − HealthBasics[i].
       ↪ lastMessageTime);
4  ellapsedSeconds /= 1000.0;
5  calculate(Report[i].latency, abs(delay));
6  calculate(Report[i].jitter,Report[i].latency.variance);
7  pcktsLost = seqNum − (Basics[i].lastseqNum + 1);
8  Report[i].pcktLoss += (pcktsLost > 0) ? pcktsLost : 0;
9  calculate(Report[i].goodput,(packetSize ∗ 8)/
       ↪ ellapsedSeconds);
10 HealthBasics[i].lastMessageTime = now;
11 HealthBasics[i].lastseqNum = seqNum;
```

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

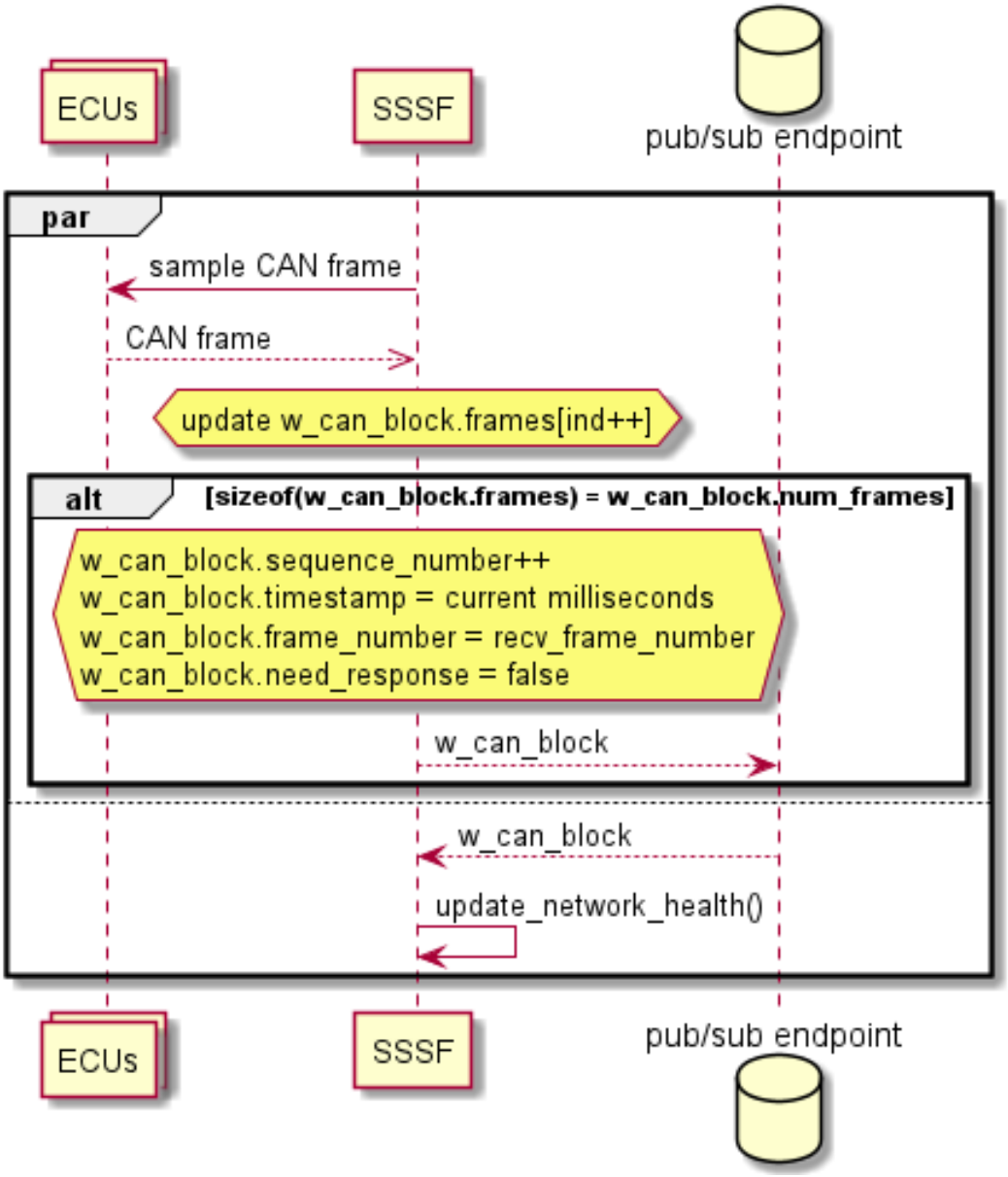

Figure 8: CAN Communication Activity

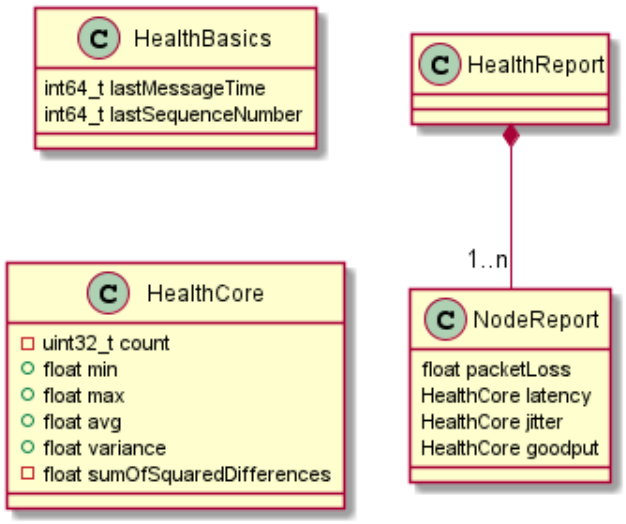

Figure 9: Network Health Data Structures

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"[4]. Finally, goodput is the measurement of application-level throughput. In our case, it is calculated as megabits per second.

The process of calculating the statisics is shown through the code listings 1, 2. Associated data structures are shown in figure 9.

To display these captured statistics to the user the Controller needs to first 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 each device resets its local state variables, keeping only the last message timestamp and the last seen sequence number. This 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.

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

3 Exemplary Usage Scenario

Figure 10 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 2. For the current purpose we have been using the CARLA graphical vehicle simulator [5]. 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 [14] 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. The figure shows no packets were lost while the latency in the last cycle of health report collection was about 4 milliseconds

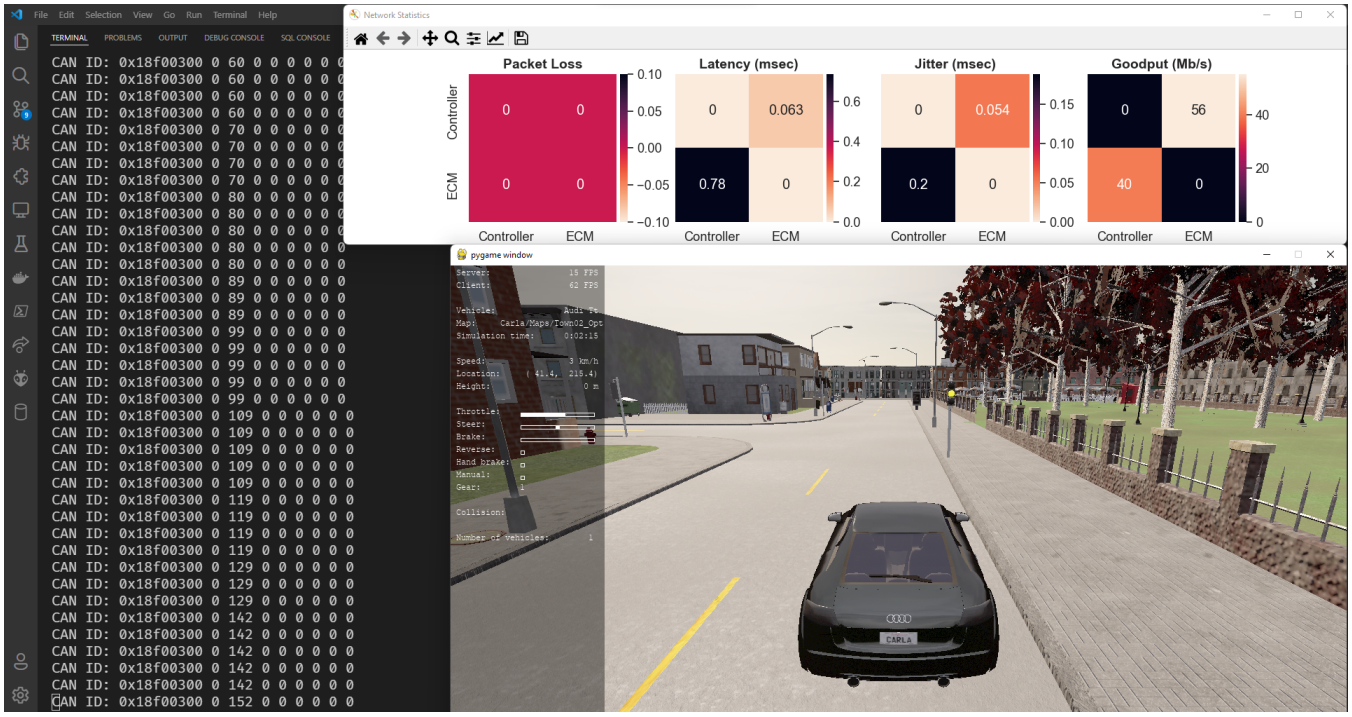

Figure 10: CANLay at work in the Software Defined Truck

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.

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

References

[1] 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.

[2] 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.

[3] 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.

[4] Michael Doering and Marco Wagner. Retrofitting SDN to classical in-vehicle networks:. Technical report, Universit{\"a}t T{\"u}binge.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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