Towards CANLay: An X-in-the-loop Virtual Testbench for In-Vehicle Security Testing with Real-time Network Performance Monitoring

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 inherrently unauthenticaed, makes it susseptible to network-wide cyber threats. It has been shown [1, 2, 6] that remote interfaces on modern vehicles can be used to intrude into CAN networks and inject message to control and/or disrtupt the operations of the vehicle. To evaluate the effectiveness of their approaches, 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 test-bench is imminent. This is where the concept of the

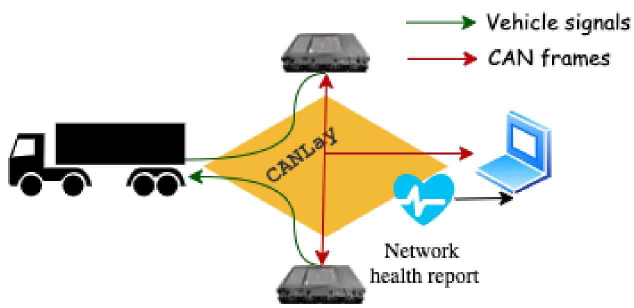

Figure 1: Scope of CANLay

Software Define Truck (SDT) [5] is critical to the in-vehicle networking community. It aims to provide a distributed virtualization infrastructure on which in-vehicle security experiments can be performed. Although proposed primarily for the heavy-trucks, SDT can easily be adapted for light-weight passenger vehicles as well. 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 accross the globe. CANLay is the designed to be the networking backbone of SDT and provide the neccesseary infrastructure to enable this service.

Previous research [4] has established two critical quality evaluation criteria for testbeds namely, fidelity and adaptability. Fidelity is the ability to emulate a real-world in-vehicle networking infrastructure to the best. Adaptability is the ability to emulate different real-world in-vehicle networking infrastructures. As such, it may be difficult to optimize both these criteria at the same time. To make a

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

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 allows on-demand network configuring, 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.

Network virtualization for CAN has been approached in different ways. First, there has been the software-defined-networking adapted for CAN. This approach is largely hardware basd and is targetted primarily for in-vehicle networking, not over long range networking. For range relaying of CAN frames, there has been the CAN-to-ethernet direction of research. Reconfigurability and network performance has not been a cornerstone for these systems. Then again, neither of these paradigms have been designed to transport vehicle signals over long distances. A few research works have ventured into that area. Albeit, they have not resorted to network virtualization for the same.

In the rest of this paper.

2 Design and Current Development

Figure 2 shows the proposed system design of CANLay. Based on the goals established in the previous section, we identified three different functional objectives of the system: 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 acomplish the functional objectives of the 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 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 is sensor messages from the vehicle simulator, but, in accordance with the current design goals, this feature is not used and the analog connection between the SSSF and the ECU is shown in a dashed line figure 2. The second type is CAN data carriers from the ECUs connected to them and 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 essentially the smart sensor simulator with additional CAN forwarding capabilities. It is a built on a Teesny 3.6 .... a paragraph describing the SSSF's ability... talk about SD cards 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.

Add some implementation details - like python, thread etc. The real time clock on the Controller

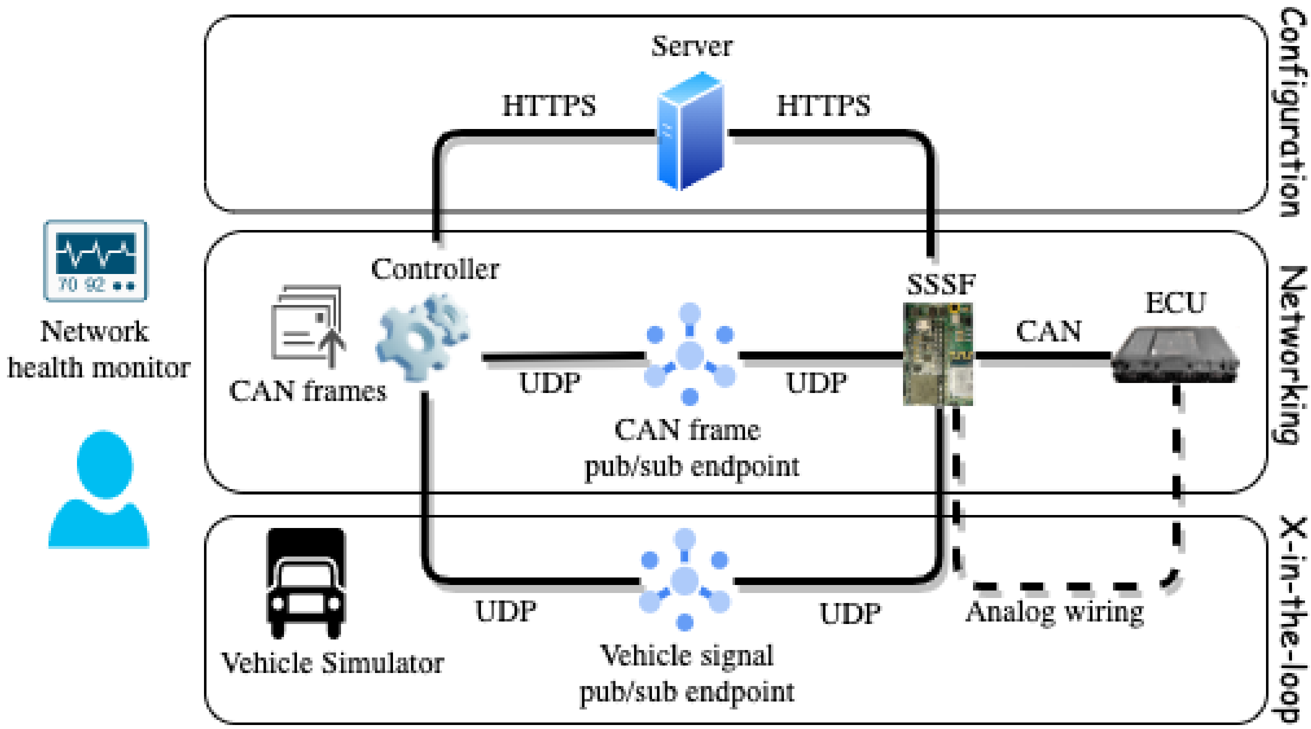

Figure 2: Proposed System

is synchronized through the network time protocol (NTP).

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 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.

Add some implementation details - like python, thread etc. Sockets. The Server accepts HTTP API calls. API calls were chosen because they clearly define the object to invoke and the manner in which to invoke it.

2.1.4 Publish/Subscribe Endpoints

UDO 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 resembled that of a CAN network we used a few criteria. The first is that the transport mechanism must support some form of message broadcasting which enables a sender to send one message that can be received by many receivers without signficant 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 throughout. At this time, we are also exploring other potential pub/sub implementations such as MQTT.

2.1.5 Network health monitor

A large drawback of connecting devices over a shared/multipurpose network is the increase of networking delays. When a user creates a virtual test bed using CARLay they are logically forming a virtual network for that experiment, but physically each device is still connected to the same network as before. As a result, communication between devices in a virtual test bed could suffer from network delays that are out of the user's control. The Network Health Monitor provides the user with useful information about the current status of the network so they can ensure that the current network conditions are not affecting their results. The network information is gathered from the perspective of all devices in a experiment rather than collecting it from just the perspective of the Controller. This provides a more full view of the current state of the network.

2.2 Behavior Descriptions

2.2.1 Setup (ref. figure 3)

While the Server is up and running SSSFs connect to it. SSSFs perform a setup() procedure by reading their inbuit 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 202 error code. Otherwise, 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. This asked 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. 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 via HTTP POST.

When the Server receives a 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

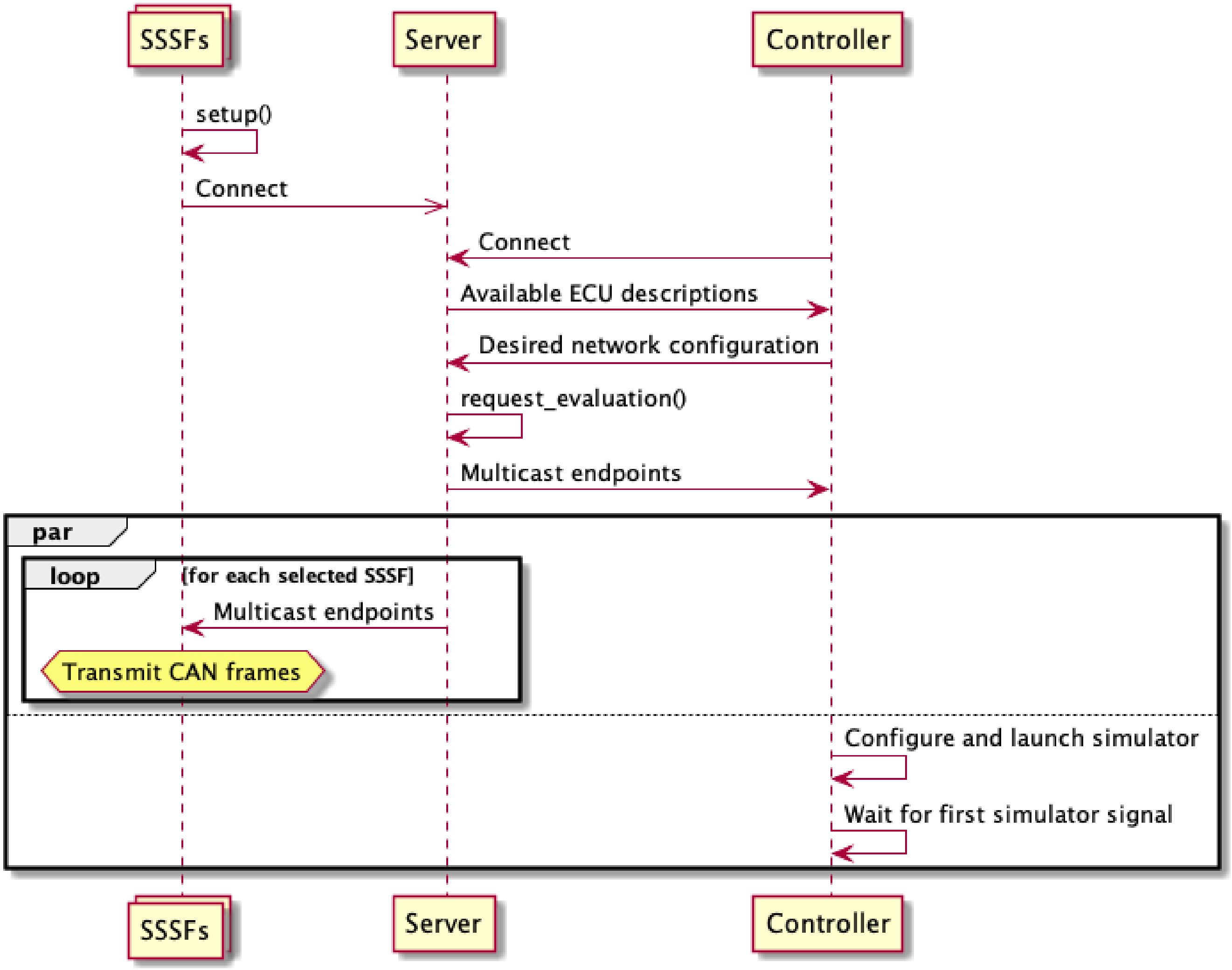

Figure 3: Setup Activity

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.

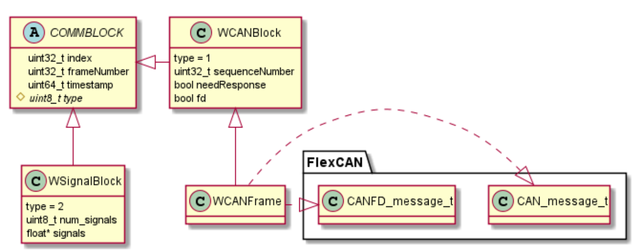

Figure 4: Useful Data Structures

2.2.2 Signal Transmission (ref. 5)

2.2.3 CAN Communication (ref. figure 6)

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 4) that will be written to the CANLay network. In the current scenario we set `num_frames` to 1. Therefore, a COMMBlock data structure is trans-

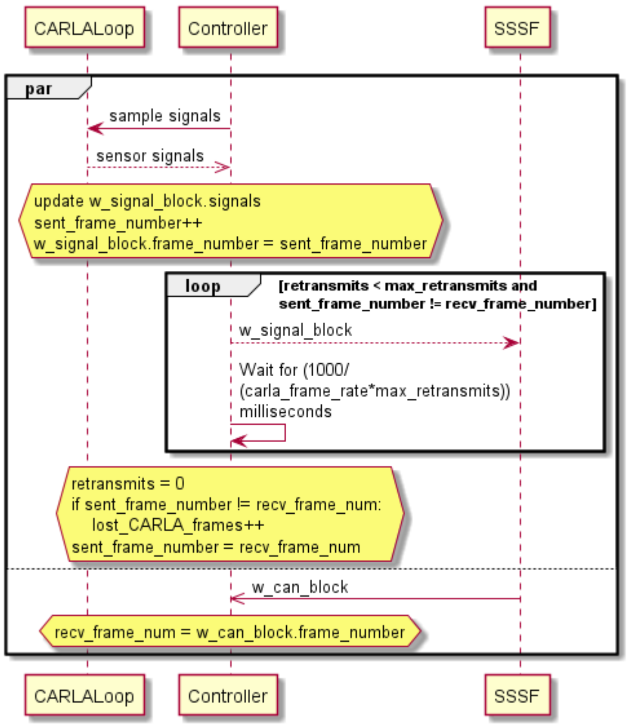

Figure 5: Signal Tranmission Activity

mitted 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 2.2.4), the last one is used to for retransmission of retranmission of WSignalBlock structures as alread seen in section 2.2.2. 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);

2.2.4 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.

The type 1 COMMBlock messages (messages containing CAN frames) include a sequence number in addition to the frame number discussed earlier. Whenever an SSSF device sends out a CAN message, it adds a sequence number and increments

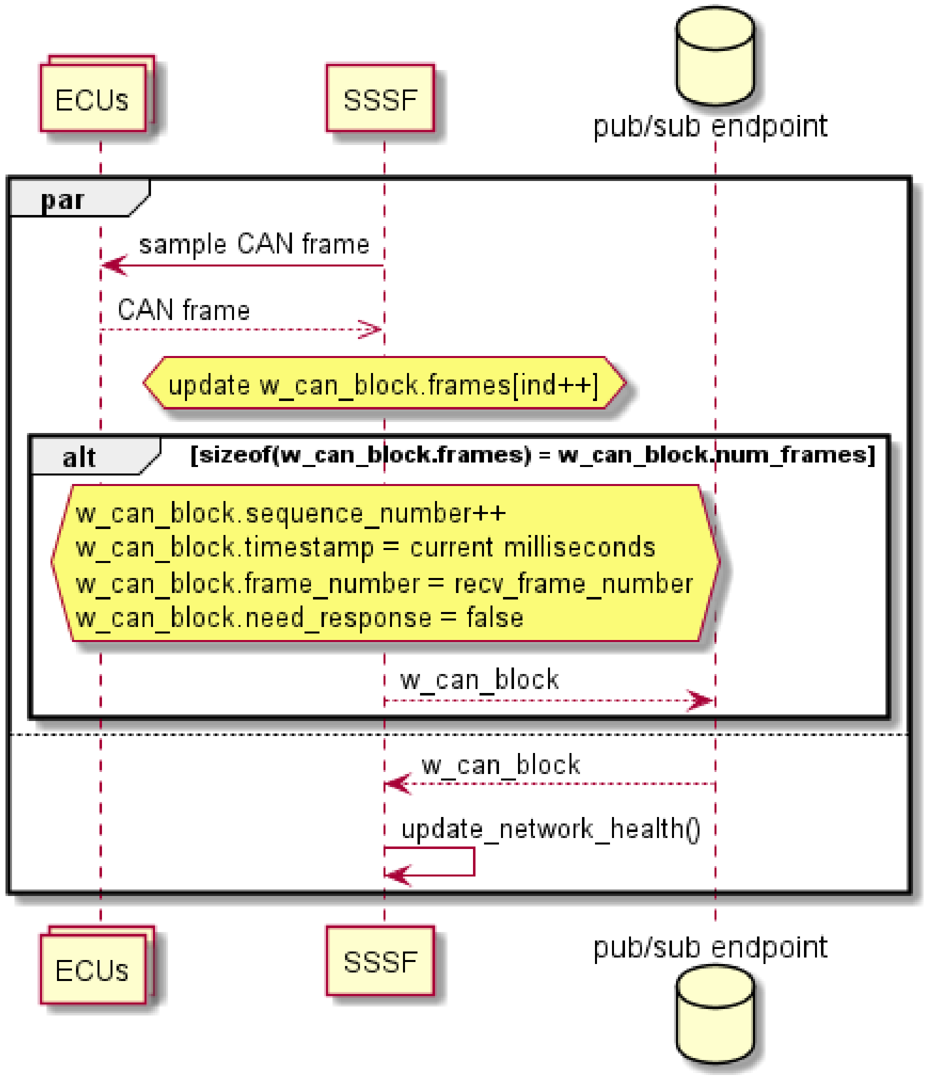

Figure 6: CAN Communication Activity

it by 1. Signals from the controller are sent at the frame rate of the game. This means that with a frame rate of 60fps the Controller is sending messages about 3-4x slower than the message rate seen in on a 250,000 baud ECU operating at nominal bus load. As such we cannot expect the acknowledgement mechanism used for the Controller messages to work for the CAN messages. However other devices can still use the sequence numbers to detect dropped CAN messages by looking for gaps larger than 1.

The next important metric included in the COMMBlock messages is a timestamp. Timestamp is included to allow devices to calculate the latency along the network edge from the sending device to the receiving device. Of course this could be done by performing network tests during an active experiment but unless the latency is guaranteed to always be low, this testing could interrupt the normal flow of the other messages being sent in the experiment. So instead the timestamp is included with each message that is sent so that network health metrics can be calculated on the fly and performed 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.

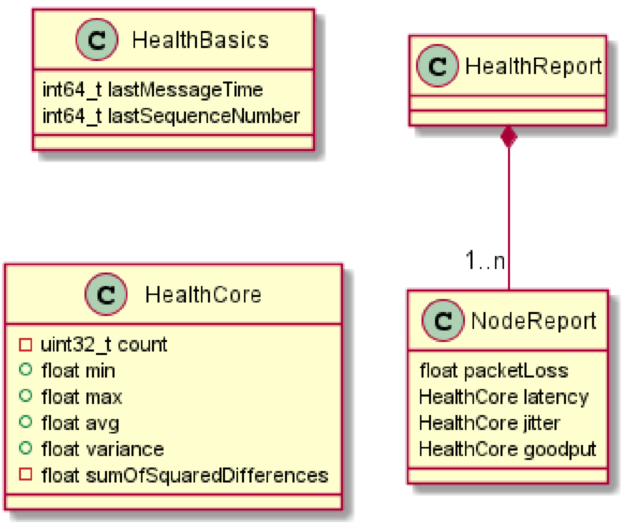

Figure 7:

The downside to this method is that it requires all of the device's clocks to be synchronized as close as possible. With the current basic implementation of the NTP protocol, the devices on average stay within a millisecond or two of each other when they are all referencing an NTP Server on the same network. Unfortunately when the devices use different NTP Servers that are much farther away the devices tend to stray between 5-15 milliseconds from each other. This creates issues when trying to measure the latency of devices that are very close to each other. It's important to note that more advanced implementations of NTP may be able to shrink these numbers.

Overall, these indicators enable each device to calculate four network statistics for every other device on the network namely 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 for a message to go from the sender to its receiver. Jitter is the variance in latency. There are many different types of network jitter but we use the simplest form which is often called packet jitter or constant jitter which is defined as "packet jitter or packet delay variation (PDV) is the variation in latency as measured in the variability over time of the end-to-end delay across a network" (cite https://networkencyclopedia.com/jitter/). Goodput is the measurement of application level throughput. In our case it is calculated in bytes per second.

Every second the Controller sends out a type 3 message which requests the health report from each device. After each device has sent their health report to the Controller they reset their statistics, keeping only the last message timestamp and the last seen sequence number. This effectively creates a measurement period of 1 second. As the Controller receives the health reports, it updates the network statistic data structure and displays the new results to the user. By collecting health reports from each network node we are able to get the statistics about a network edge from devices on each side of that network edge (how do I explain the importance of the network stats matrix better?).

3 Usage Example

Figure 8 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 described in figure 2. For the current purpose we have been using the CARLA graphical vehicle simulator [3]. 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 frames is being printed as they are broadcasted on the overlay for this particular experiment. The ID of this frame is defined by the SAE-J1939 standards [7] and identifies engine parameters transmitted by an engine control module (ECM). The data bytes carried in the CAN frame are showed next. Of these, the third byte is shown to be changing. This particular byte carries the percentage throttle demanded by the driver. The

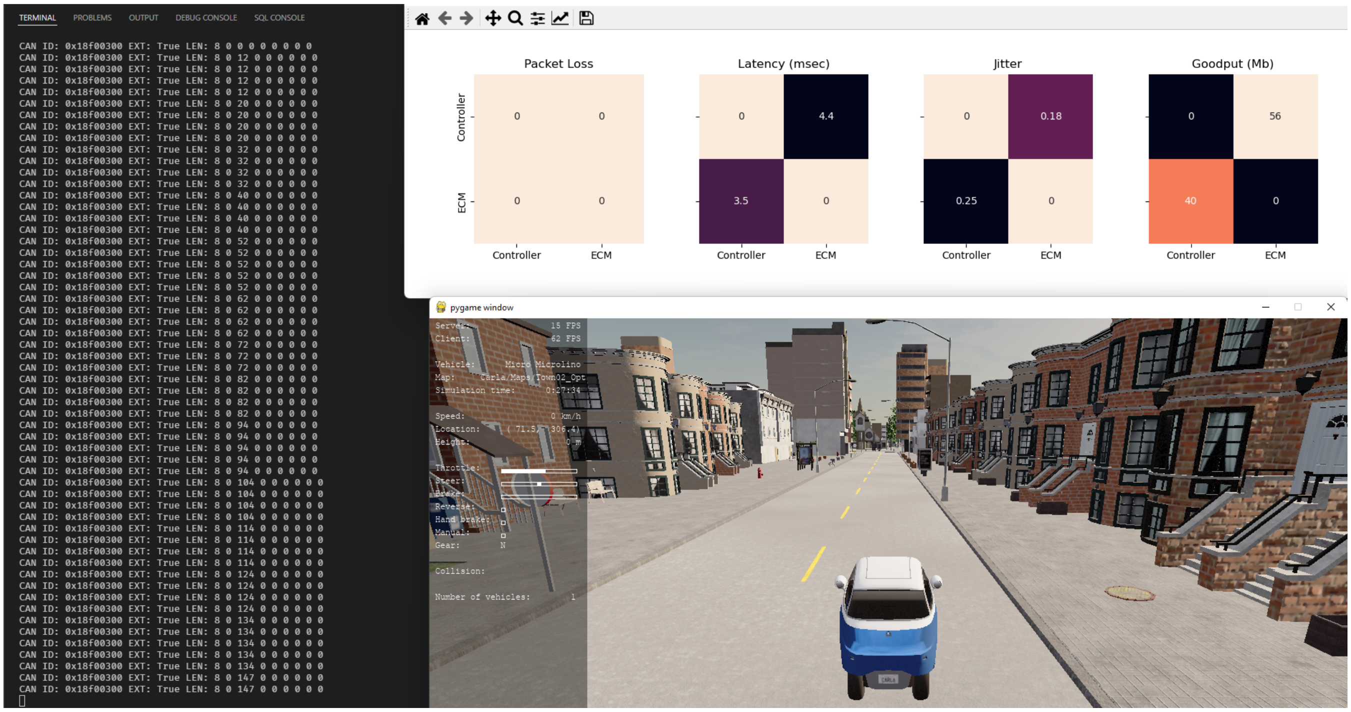

Figure 8: CANLay at work in the Software Defined Truck

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 differnet metrics to estimate network health: packet loss, latency in milliseconds, 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 substainable 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 comparsion to the latency. The goodput, i.e. the application data rate is understandibly higher for the controller as it sends WSignalBlock frames that are slightly larger in size 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] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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