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

In recent years automotive security has been a much talked about topic of research. Researchers [2] have shown that remote interfaces on modern passenger vehicles can be used to intrude into the in-vehicle network of embedded controllers, also referred to as Electronic Control Units (ECU). MHD vehicles expose similar interfaces that can be used to control/disrupt critical functions [1, 5] typically operated by ECUs using sensors and actuators. To evaluate the effectiveness of their approaches, researchers have traditionally 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.

There are two important criteria that reseach in this area has established for this type of testbench. The first is fidelity, i.e. the ability of the setup to replicate a real-world in-vehicle networking infrastructure. The second is adapability i.e. the ability to be reconfigured to suit different needs. It may be difficult to maximixe the extent to which both these criteria is achieved. The most adaptible testbed is the one in which ECUs as well as the network configuration can be programmed on the fly. Existing solutions have enabled ECU virtualization but not network virtualization for real-world ECUs. In those setups, ECUs from different physical locations cannot be used in the same testbed, neither can networks be configured on demand, unless the ECU is virtualized. We believe that having real-world ECUs in the testing setup is critical. This not only provides greater fidelity but also alleviates any concerns with intellectual privacy and availability on the ECUs. Another aspect to fidelity and adaptibility is the realization of vehicular conquality of the network Existing solutions

In this

2 Related Work

A recent survey presents a broad overview. THe main concerns are fidelity and adaptbility. More you virtualize, more you loose fidelity. There is no general about to an optimum measure but it depends on the type of experimentation the testbed aims to support. Experiments could be reliant on subtle physical features or the data being transmitted. Virtualization is suited for the first kind, while definitely providing greater adaptbility. Virtualiza-

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

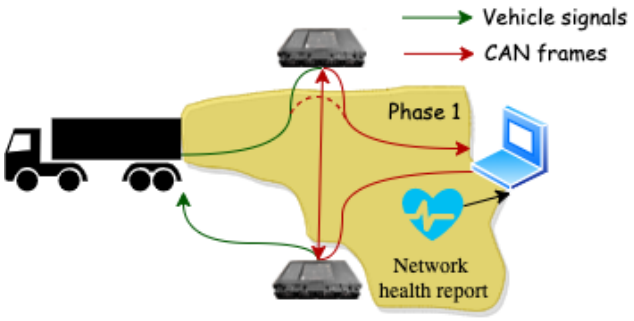

Figure 1: Design Goal of CANLay

tion can be applied at three levels namely, ECUs, network, and physical. ECU virtualization may be critical but extremely challenging to acheive due to intellectual property regulations. Network virtualization may be a more viable route but, to the best of our knowledge, most testbed proposals have eluded this feature. Albeit, experiments that rely on low-level features of networking, like signaling schemes and timing patterns may be difficult to acompolish here, but it may be a more concrete ground for data oriented reseach with certainly a more diverse specturm of testing setups.

3 Design Goals

The overarching design goal of this project is elucidated through figure 1. To design a platform that simulates a CAN internetwork within a running vehicle using ECUs from separate subnets within the internet a. la. the software-defined truck [4]. Clearly, this has to be achieved over in-vehicle communication overlays on top of protocols of the internet.

The distributed overlay nature of the project introduces a set of fidelity-related challenges when emulating the tightly integrated infrastructure within the vehicle. Firstly, CAN provides a low-latency, strong reliability communication media that may be difficult to emulate over long-range communication channels. This requirement is even stronger for the physical signals that are usually transmitted over direct wiring in a real-world setting. Secondly, the broadcast nature of CAN may be difficult to realize in a distributed manner over the (typically) unicast internet protocols. Using unicast packets for multi/broadcast may require duplication in linear time. This is both space and time-intensive, especially if performed in a smaller scale local network that has limited bandwidth. The final challenge is to realize the complexity of interactions between the different components of the vehicle that normally operate in the same physical setting.

Albeit, all of these challenges can be difficult to realize in full in an adaptability-centric setting like ours. Even so, a major goal of this project is to optimize and report the quality of experimentation, thereby creating a transparent and usable environment for the experimenter.

4 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.

4.1 Component Descriptions

4.1.1 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

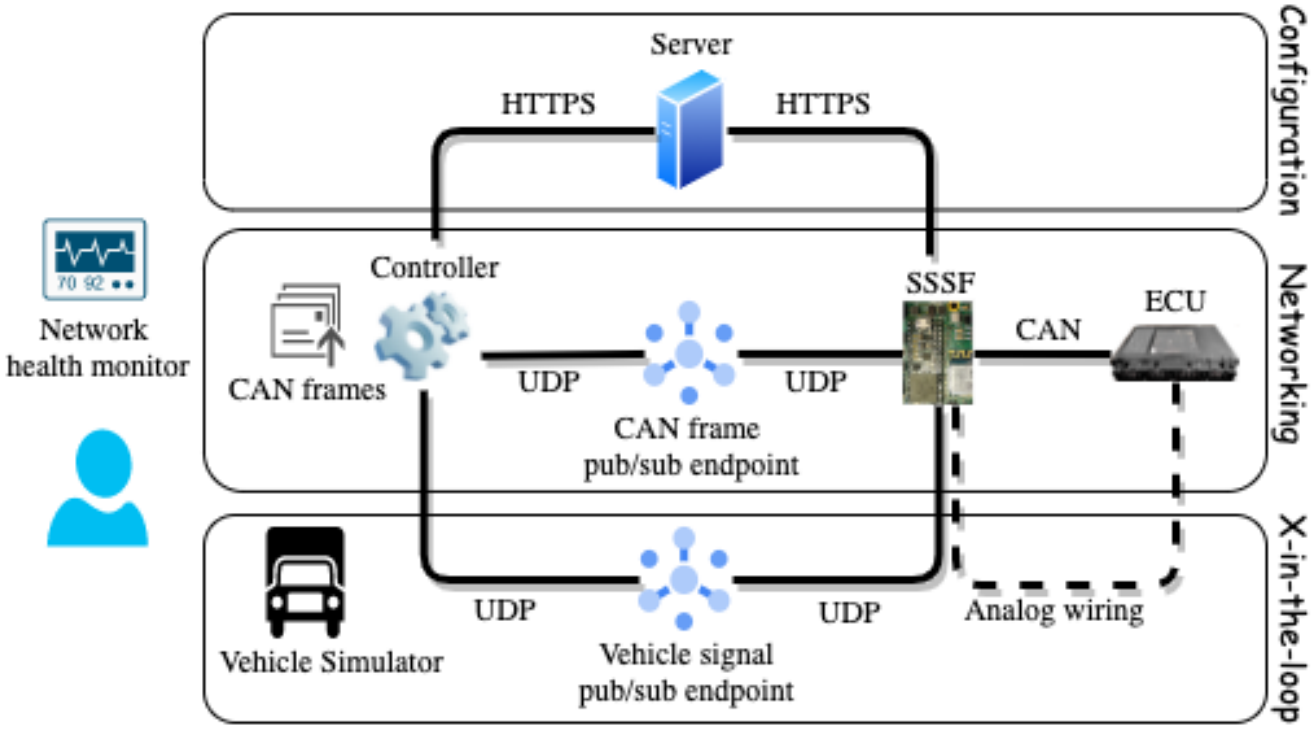

Figure 2: Proposed System

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).

4.1.2 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 is synchronized through the network time protocol (NTP).

4.1.3 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.

4.1.4 Vehicle Simulator

In theory, the Vehicle Simulator can be any software that provides the user with the ability to control a vehicle in a digital world. To be compatible with CANLay, the Vehicle Simulator must expose its operating signals in some manner. For the purpose of this paper, we chose 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. Furthermore, CARLA is highly configurarble, robust and has been used for purposes similar to ours.

4.1.5 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.

4.1.6 User Interface to CAN

We forward the recieved CAN frame on a virtual CAN (VCAN) interface that is provided by the user at startup. A user can access these frames via socketCAN [2] and assosiated tools.

4.1.7 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

[2] https://www.kernel.org/doc/html/latest/networking/can.html

from just the perspective of the Controller. This provides a more full view of the current state of the network.

4.2 Behavior Descriptions

4.2.1 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 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.

4.2.2 X-in-the-loop Simulation

4.2.3 CAN communication

In order to create a virtual test bench of ECUs the devices must be able to communicate with each other as if they were talking via a CAN bus. Most ECUs do not come with built in ethernet ports and the ability to broadcast CAN frames via Ethernet. So in order to connect an ECU to Ethernet we need a device that can interface with a CAN network and interface with an Ethernet Network. This is why the Smart Sensor Simulator and Forwarder is necessary. It acts as a programmable gateway that can transfer messages between the CAN network and the Ethernet Network.

When the SSSF is in an active experiment it first checks if one or both of its CAN networks were assigned baud rates. If the CAN network was assigned a baud rate, 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 message that will be written to the CANLay network. The COMMBlock message requires some additional information before it can be written to

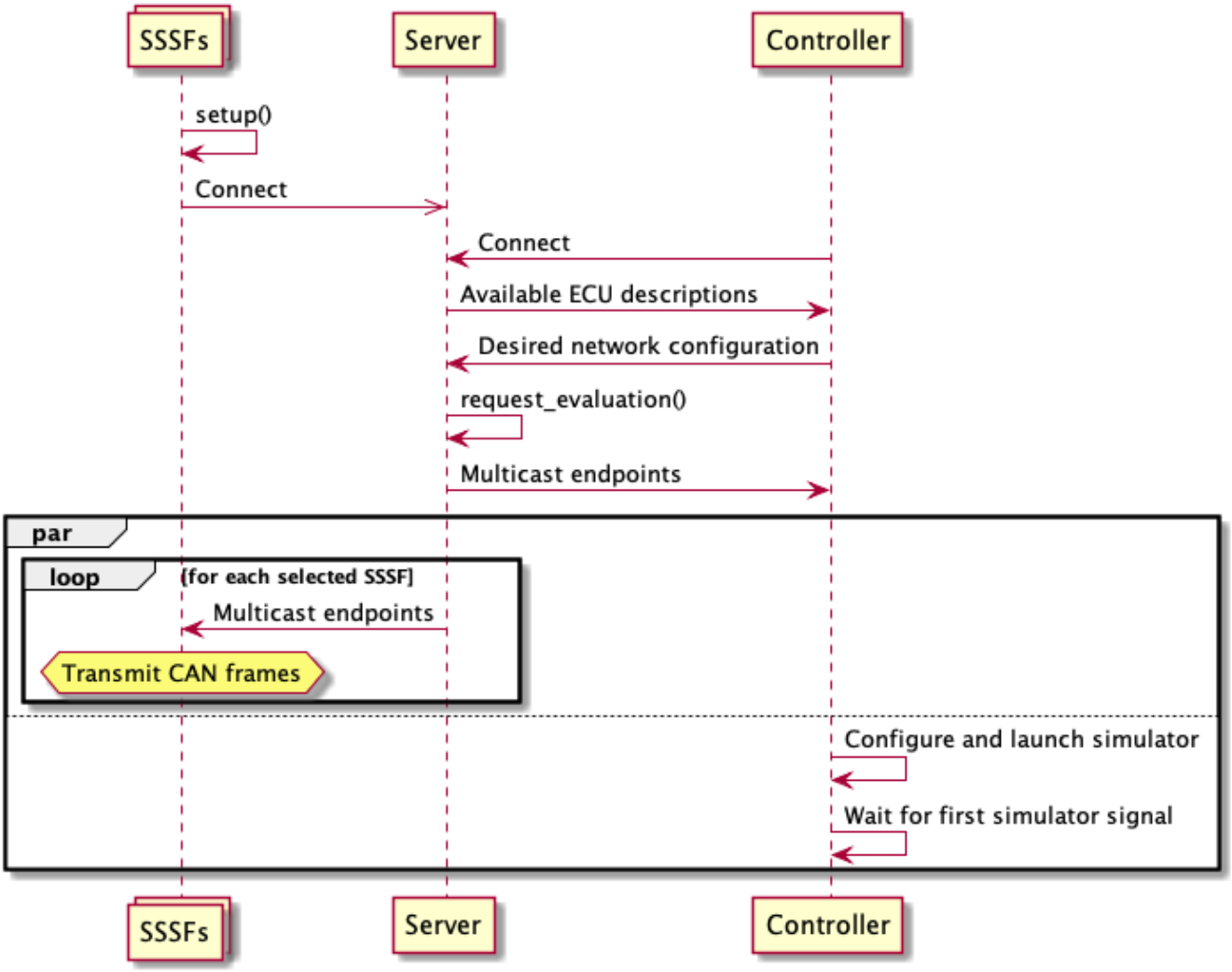

Figure 3: Setup Activity

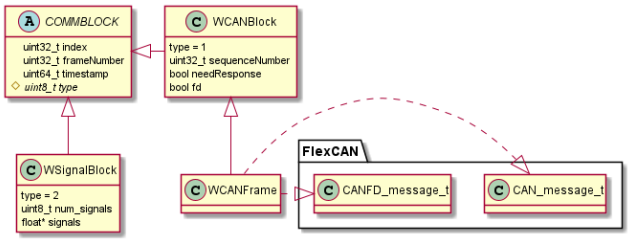

Figure 4:

the network. First a type ID is added which indicates the type of message the COMMBlock is carrying. In this case the type will be 1, indicating that it is carrying a CAN message. It will also need to add the current millisecond timestamp at which it is sending the message. This is used by other devices to calculate the network latency for the network path from this device. Also, a sequence number is added and incremented every time a CAN message is sent. If other devices on the network detect gaps in the sequence numbers they know that a message has been lost. In addition it will also append the frame number of the last frame it received from the Controller. If it has not received any frames from the Controller then this number will be zero. The last seen frame number indicates to the Controller which SSSFs have and have not received the last sent frame. Also, the SSSF will mark whether this

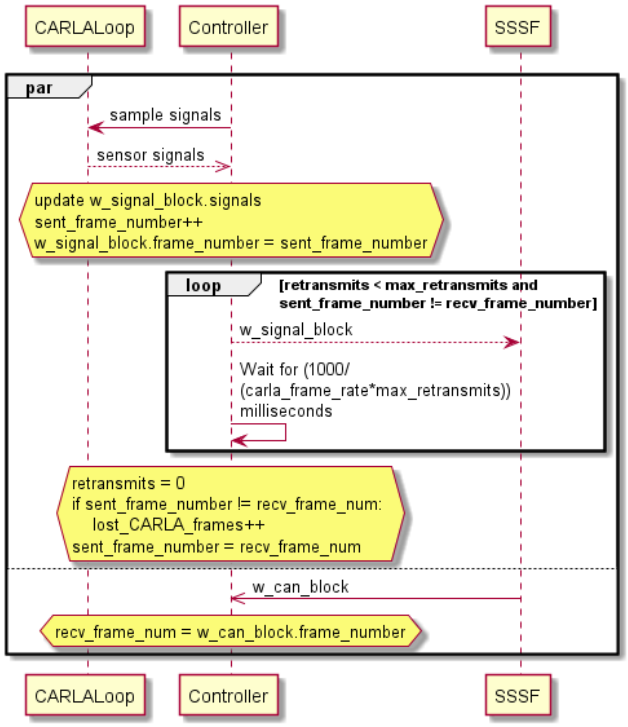

Figure 5:

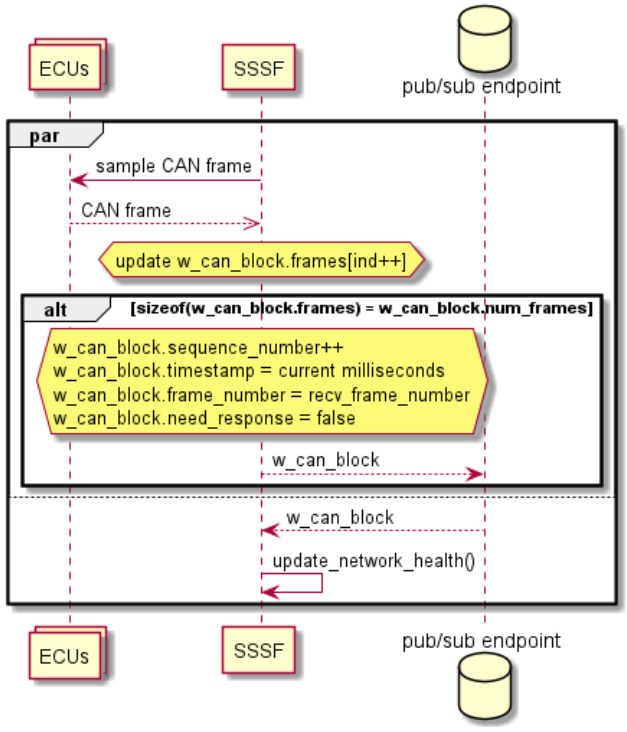

Figure 6:

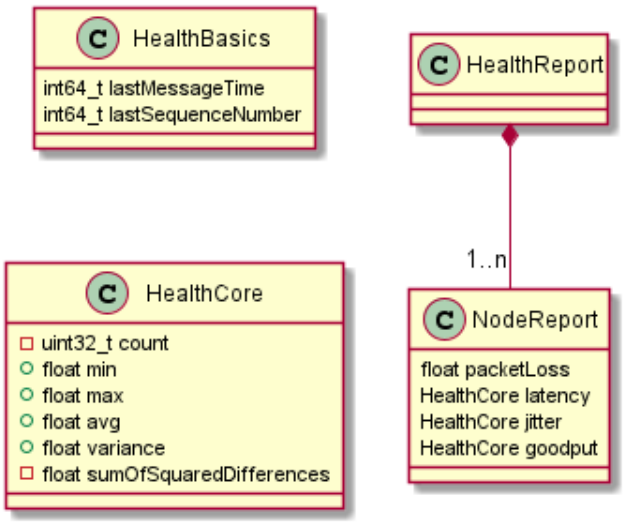

Figure 7:

CAN message is a canFD CAN frame. Finally, the SSSF will mark whether or not 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);

4.2.4 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 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. 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, keep-

ing 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?).

5 Analysis

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

[5] 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.