
INSTALLATION AND TESTING GUIDE FOR COMMUNICATION SETUP BETWEEN HMS AND SIMULINK

DATE: MAY 9, 2023



Funded by the National Aeronautics and Space Administration under award number 80NSSC19K1076.

Table of Contents

1	Required Software Installation	2
1.1	Matlab	2
1.2	Docker Desktop	3
1.2.1	Creating a Docker Container	3
1.2.2	Applying Changes to the Container	6
2	Communication Network Emulator	7
2.1	Communication Network Front-end	7
2.1.1	Accessing Data Through Docker Container	10
2.1.2	Accessing and Storing Data Using REQUEST Service	10
2.2	Telemetry Data Visualization	11
3	MCVT-Docker Communication	14
3.1	Running Test Scenario	14
3.2	Communication Between the Docker and a Separate MCVT Computer	15
3.3	Communication Between the Docker and Speedgoat	15
4	Communication Performance Requirement Verification	16
4.1	Sending Data from Simulink Model	16
4.2	Verifying Performance Requirements	19
4.2.1	Packet Loss and Time Delay	19
4.2.2	Misordered Data	21
4.2.3	Multi-source, multi-signal	22
5	Appendix I	24
6	Appendix II	26

1 Required Software Installation

To successfully run the communication network emulator and establish a communication with the MCVT, MATLAB and Docker Desktop will be required. This section serves the purpose of an installation guide for the two software.

1.1 Matlab

For all the toolboxes needed to be supported, please avoid using a Mac device. Additionally, to make sure that the newest version of MCVT can be ran without any compilation error, please install MATLAB **2020b** ([here](#)) to support the model's compatibility with Speedgoat real-time machine. During installation, please select the following products to be installed along with MATLAB:

1. Simulink
2. Control System Toolbox
3. Deep Learning Toolbox
4. DSP System Toolbox
5. Signal Processing Toolbox
6. System Identification Toolbox
7. Statistics and Machine Learning Toolbox

After MATLAB 2020b is successfully installed, please navigate to the Add-Ons drop-down menu shown in the figure below, and select Get Add-Ons.

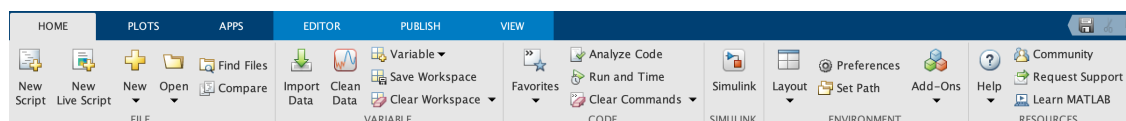


Figure 1. MATLAB Toolbar

On the Get Add-Ons page, please search and install **Simulink Real Time**. *Note: Simulink Real Time toolbox is not supported by Mac devices, so please install MCVT on a Windows or Linux machine.*

Now all the toolboxes have been installed, please download the MCVT repository and run MCVT_v6_RunScenarios.m to check if the program can be run without any errors.

1.2 Docker Desktop

The communication network emulator is packaged as a Docker container, and it needs to be run through **Docker Desktop**. The Docker Desktop software can be installed either on the same device as the MCVT or on a separate device ([here](#)).

Note: For Windows users, please install wsl_update_x64.msi ([here](#)) after installing Docker Desktop

1.2.1 Creating a Docker Container

To create the Communication Network Emulator, first start up the Docker Desktop application and make sure the application starts successfully. Then, clone the shared Github repository for data repository and data service and communication network to your computer. Open the new folder containing the data repository and data service code, note there should be a file named docker-compose.yml in the folder, then open a terminal/command prompt at this folder.

For **Windows**, open a new Command Prompt at this newly created folder by first navigating to the folder using Windows Explorer, and then clicking on the top bar and typing in cmd.exe as shown below.

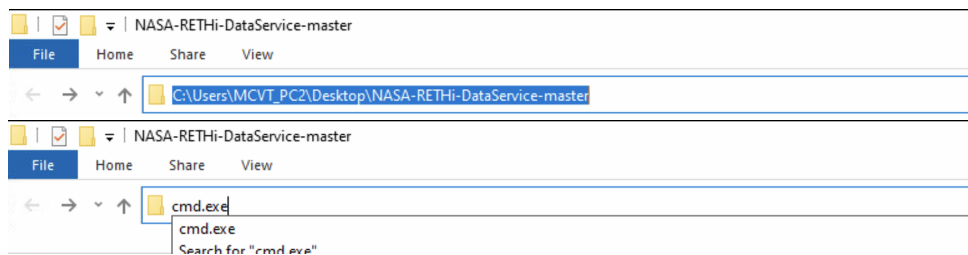


Figure 2. Windows Explorer File Path

For **MacOS** and **Linux**, right click on the selected folder and click "Open Terminal at Folder" as shown below.

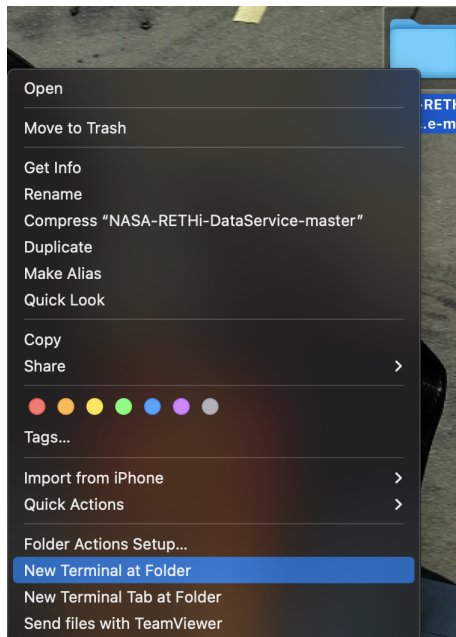


Figure 3. Openng Terminal at Folder

Make sure Docker Desktop has started. Type in **docker compose up** in the terminal to create the docker container. This process might take a few minutes, please wait til the following messages are shown in the terminal indicating the communication network emulator is now running.

```
nasa-rethi-dataservice-master-data_service-1 | Database has been initialized
nasa-rethi-dataservice-master-data_service-1 | Database habitat has been connected!
nasa-rethi-dataservice-master-data_service-1 | Habitat Data-Service Started
nasa-rethi-dataservice-master-data_service-1 | Habitat Web-Service Started
nasa-rethi-dataservice-master-data_service-1 | Ground Server Started
nasa-rethi-dataservice-master-data_service-1 | Database ground has been connected!
nasa-rethi-dataservice-master-data_service-1 | Database habitat has been connected!
nasa-rethi-dataservice-master-data_service-1 | *SGo* -- Listen on :9999
nasa-rethi-dataservice-master-data_service-1 | Ground Server subscribed Habitat server
```

Figure 4. Terminal Prompt for Docker

If an error occurs during the installation of the Docker containers, click on the **Troubleshoot** button on the top right corner of the dashboard, and select **Clean / Purge Data** to start over.

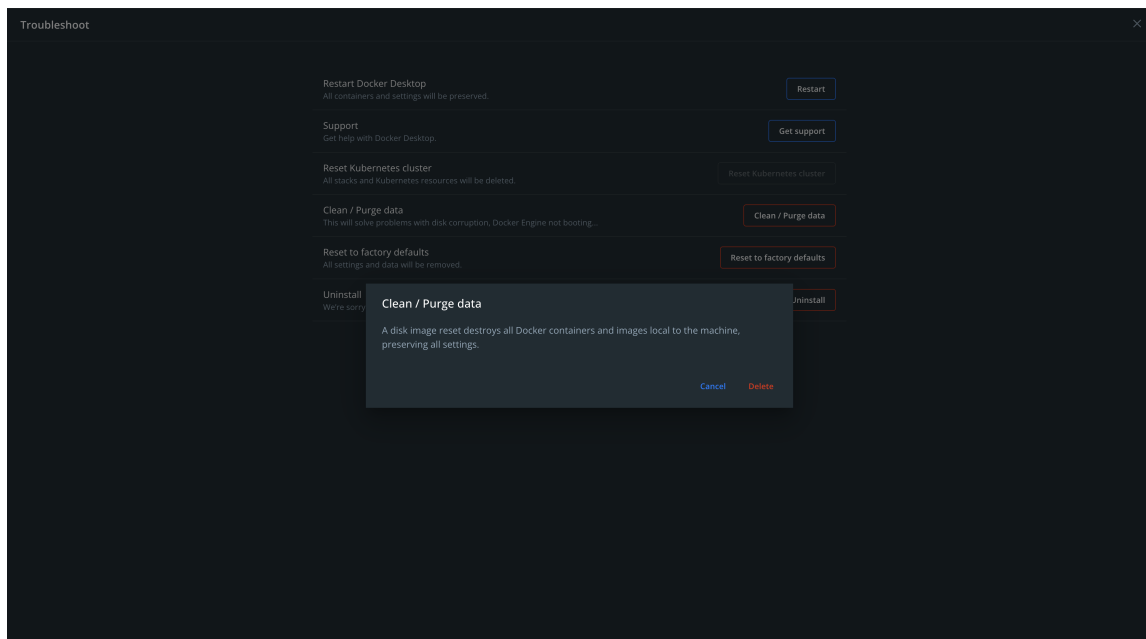


Figure 5. Docker Troubleshoot

Now in the Docker Desktop, you should see a new container now running. Under status you should see **Running (6/6)** as there are total of 6 images in the container, and you can click on the arrow to the left of the green icon to expand the container to see all images.

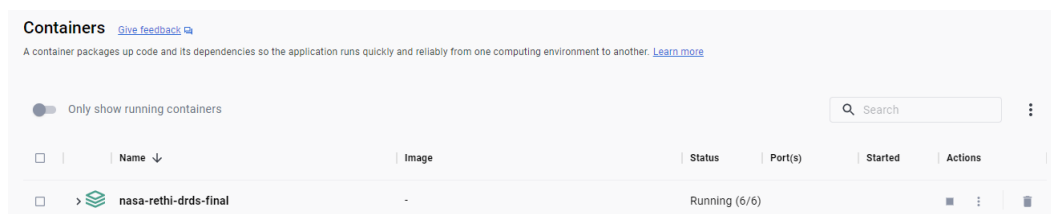


Figure 6. Docker Desktop: Containers Page

It is important to make sure that all images in the container are the latest versions, and to do so, navigate to the **Images** tab on the left side of the Docker Desktop page. On the **Images** page, you are able to gain more insights on each of the images, including the name, image ID, data created, etc. Please check the creation dates of the images and consult to make sure the latest versions are being used. If a particular image is not up to date, please click on the three dots under the Actions column and click on **Pull** as shown in the figure below.
















<input type="checkbox"/>	Name	Tag	Status	Created	Size	Actions
<input type="checkbox"/>	comm 9790700c6d7 	latest	In use	about 23 hours ago	33.31 MB	  
<input type="checkbox"/>	mariadb a3871bf45d8a 	latest	In use	5 days ago	400.93 MB	  
<input type="checkbox"/>	lovablemahira/rethi_hms_vis 3863df70dcf5 	latest	In use	5 days ago	684.48 MB	  
<input type="checkbox"/>	xuechuanyu/rethi-drds 5f646eb9cab7 	latest	In use	13 days ago	15.49 MB	  

Figure 7. Docker Desktop: Images Page

1.2.2 Applying Changes to the Container

we provide an alternative method for creating local images (the method above pulls the docker images from GitHub) so you can make local changes to the .go files used to construct the images: First, open up the docker-compose.yml file, under **services:** are the six images, examine the image that you want to create a local image for, for example, the **comm** image. Under **comm**, the line starting with **image:** is what we want to modify. Change the address of the image to an arbitrary name as shown below.

```
comm:
  #image: amyangxyz111/rethi-comm
  image: comm
  environment:
    ADDR_LOCAL_GCC: ":10000"
```

Figure 8. docker-compose.yml file

Then, navigate to the folder where the image is stored and change the necessary changes. After the changes have been applied, open a terminal at this folder using the method described earlier in this section and type: **docker build -t comm .** to build this folder into a local docker image named **comm**. Lastly, delete the original docker container and recreate the container using **docker compose up**.

2 Communication Network Emulator

The emulator contains two major human interfaces, a front-end application for the communication network and OpenMCT, a visualization of all the telemetry data received.

2.1 Communication Network Front-end

The front-end visualization of the communication network can be accessed by typing in "localhost:8000" in your choice of browser.

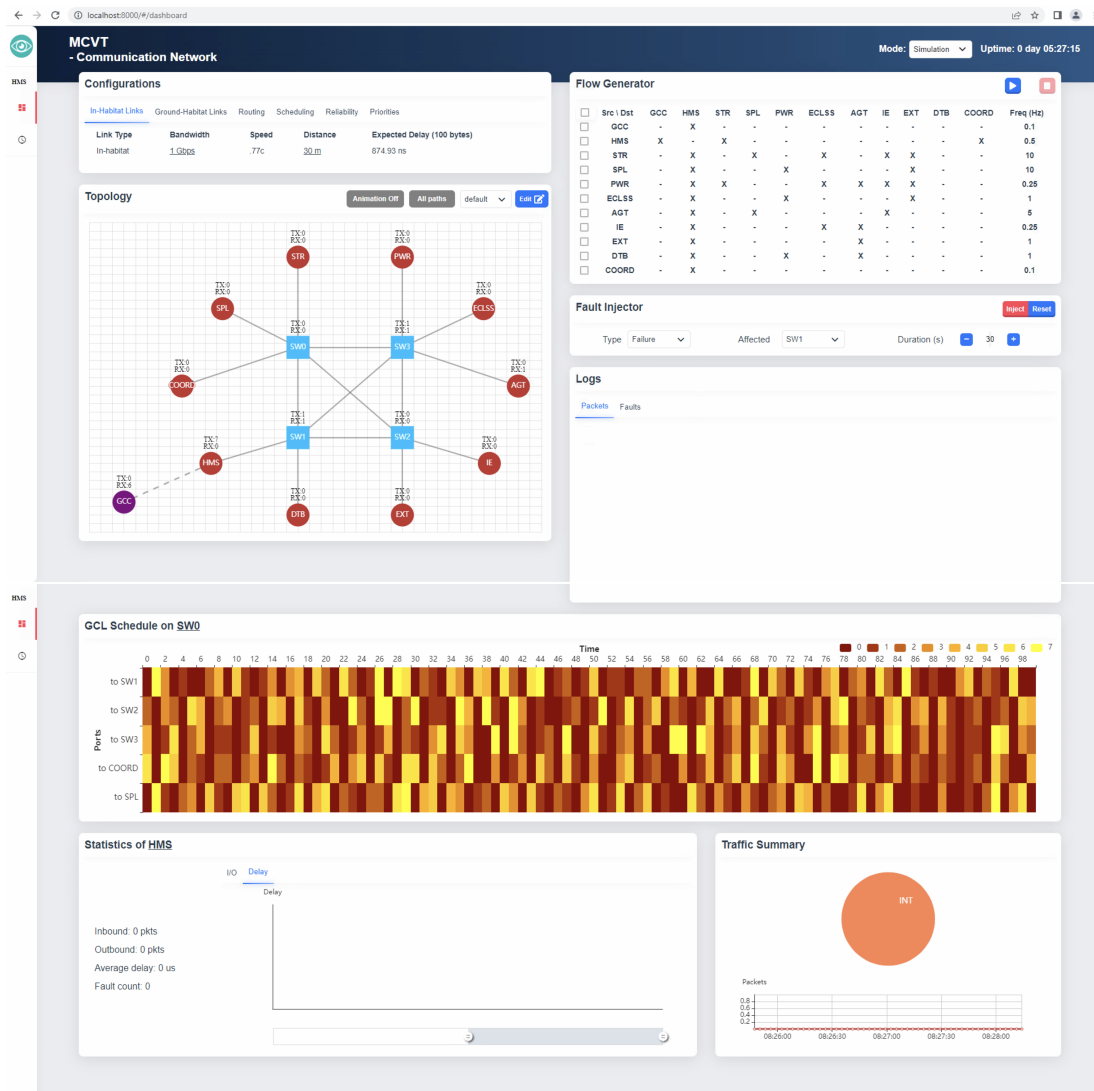


Figure 9. Communication Network Front-End

This is the main interface to visualize internal data flow and test fault injections in the communication network. There are a few ways to generate data flow between subsystems:

1. Using the **Flow Generator** panel on the user interface to generate data flow through a set of preset paths.
2. Generating synthetic data and sending it to HMS from subsystems by writing a *packetgenerator* program using the `pyapi.api`
3. Using the MCVT Simulink model and sending the generated data through **UDP** blocks (Please see the document for more detail)

Fault Injector panel can be used to inject fault or traffic into the communication network. Based on the current topology of the network, if one switch faults, there is no other possible rerouting since the subsystems are all only connected to one of the four switches.

Communication Delay between the ground and habitat can also be injected by selecting the *Ground-Habitat Links* under the *Configuration* tab.

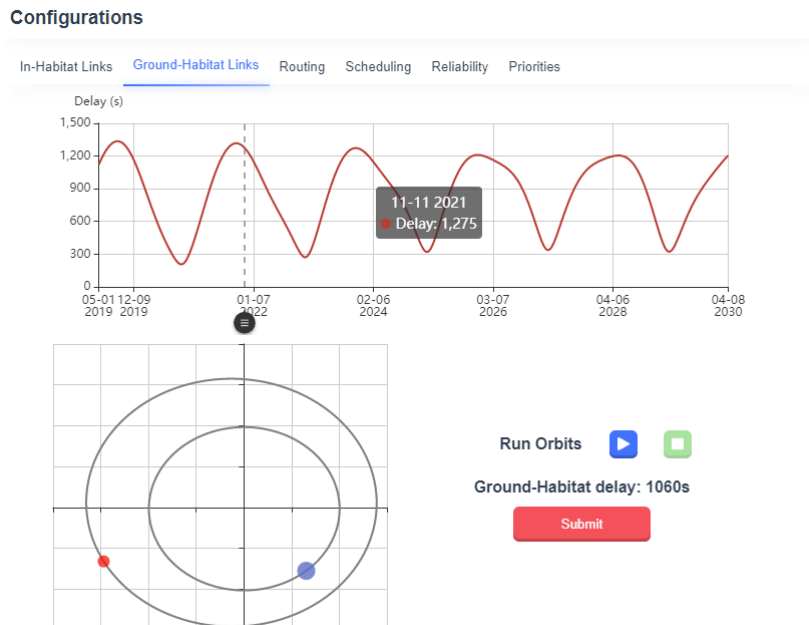


Figure 10. Ground-Habitat Links

A sample program for generating synthetic data can be found in Appendix 1. The randomly generated data is sent to the HMS using the .send function. The "id" field corresponds to the failure mode id of each subsystem found in the Failure Modes document. Once the program has started generating and sending synthetic data, the internal data flow can be verified by the Logs panel as well as the Traffic Summary panel, Figures 9 and 10 on the user interface.

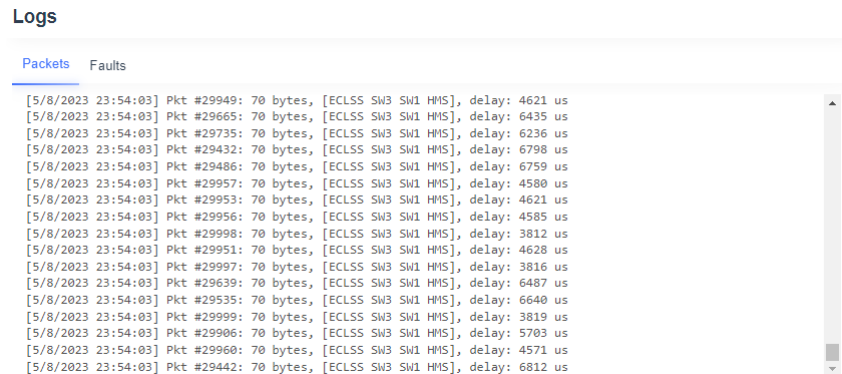


Figure 11. Data Flow Log

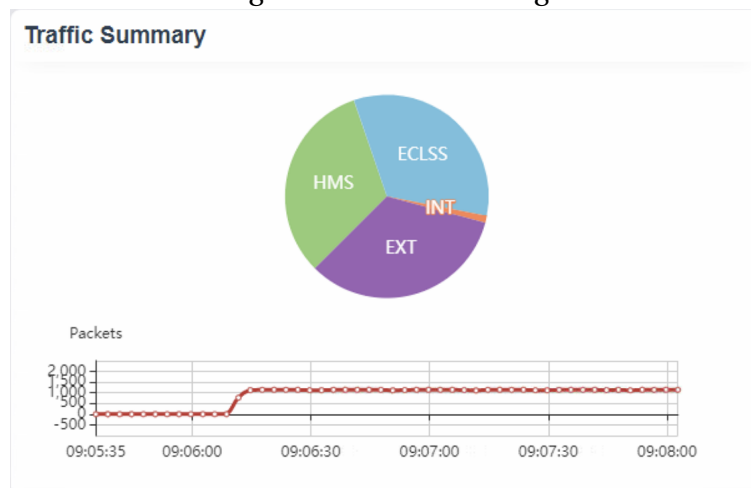


Figure 12. Traffic Summary

All the data received by different subsystems are then stored in the SQL database which can be accessed in two different ways: 1. through the Docker Desktop, 2. Using the REQUEST service which is part of the pyapi.api.

2.1.1 Accessing Data Through Docker Container

For quick debugging and verification purposes, you can access the database and view all data through the Docker Desktop as follows:

First, click on the arrow to the left of the green running container icon on the Docker Desktop main menu to see all running images. Then, double click on **habitat_db-1** image to open the image and select **Terminal** on the top bar.



```
< nasa-rethi-drds-final-habitat_db-1 mariadb RUNNING
Logs Inspect Terminal Stats
# mysql
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 5
Server version: 10.11.2-MariaDB-1:10.11.2+maria-ubu2204 mariadb.org binary distribution
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
MariaDB [(none)]> use habitat;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
MariaDB [habitat]> select * from record5001;
```

Figure 13. Habitat Database Image Terminal

In the terminal, first type in **mysql** after the # sign, then **use habitat;** to enter the habitat database monitor. Lastly, to select a specific database to view, use command **select * from record" id";**, where "id" is the specific failure modes signal id for the subsystem. For example, for "ECLSS/ECLSS_Exterior/Radiator_Panels_Paint-5001", the id is 5001, so to view the data received, command **select * from record5001;** will be used.

*Note: All commands after # **mysql** needs to end with a semicolon*

IMPORTANT NOTE: All data in the database will be lost when you stop running the Docker Container that contains the database image. Please save all relevant data on your local computer after an experiment run.

2.1.2 Accessing and Storing Data Using REQUEST Service

Due to the fact that the SQL database data structure is inherently different than the typical file system we use on our computers, the data received by the DRDS is

not directly stored in separate files. However, in the CPT, it is important to store all historical data in a system so it is easy to retrieve in the future, and to do that we have an API that allows to "request" data from the database and save it to various files using a simple python code that can be run after the experiment is finished. A sample code is shown below, and it has been tested:

```
ins = api.API(local_ip="0.0.0.0",
              local_port=65533,
              to_ip="127.0.0.1",
              to_port=65531,
              client_id=1,
              server_id=1)

re = ins.request(synt=0xffffffff, id=129)
print(re.header.simulink_time)
print(re.subpackets[0].header.length)
print(list(re.subpackets[0].payload))
```

In the code snippet above, the parameter *synt* is where you would identify the specific index of the data you want to request, and the *id* field is where you would identify which record you are requesting the data from. All the record id's can be found in the *db_info.json* file.

2.2 Telemetry Data Visualization

The telemetry data visualization, named OpenMCT, can be accessed by typing in "localhost:8080" in your choice of a web browser. The OpenMCT visualizes the data received from either the MCVT or real sensors in the cyber-physical test bed. In the case of this document, data is generated by the MCVT.

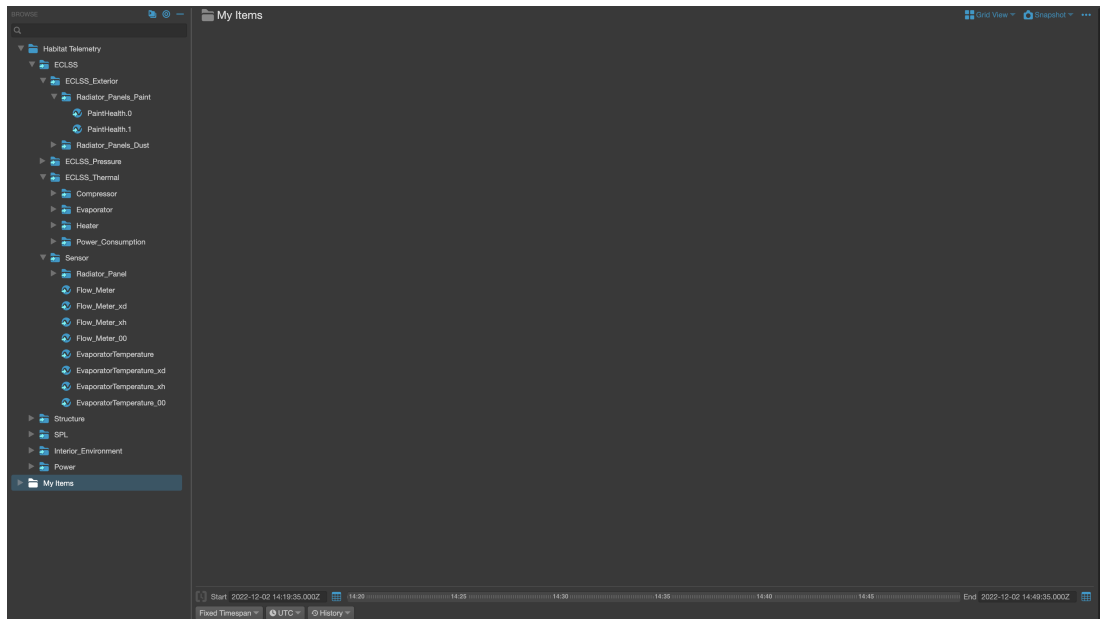


Figure 14. OpenMCT

From the drop-down menu on the left, select the specific sensor data to view. *Note: to view live-time data, please click on the Fixed Timespan button on the bottom and change it to Local Clock*

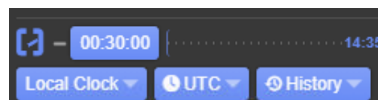


Figure 15. Local Clock Button

Once the MCVT (please see Section 3.1 for instructions on how to run the MCVT test scenario) is running and the Local Clock option is selected, the graphs should start to update. An example of such data visualization is shown below.

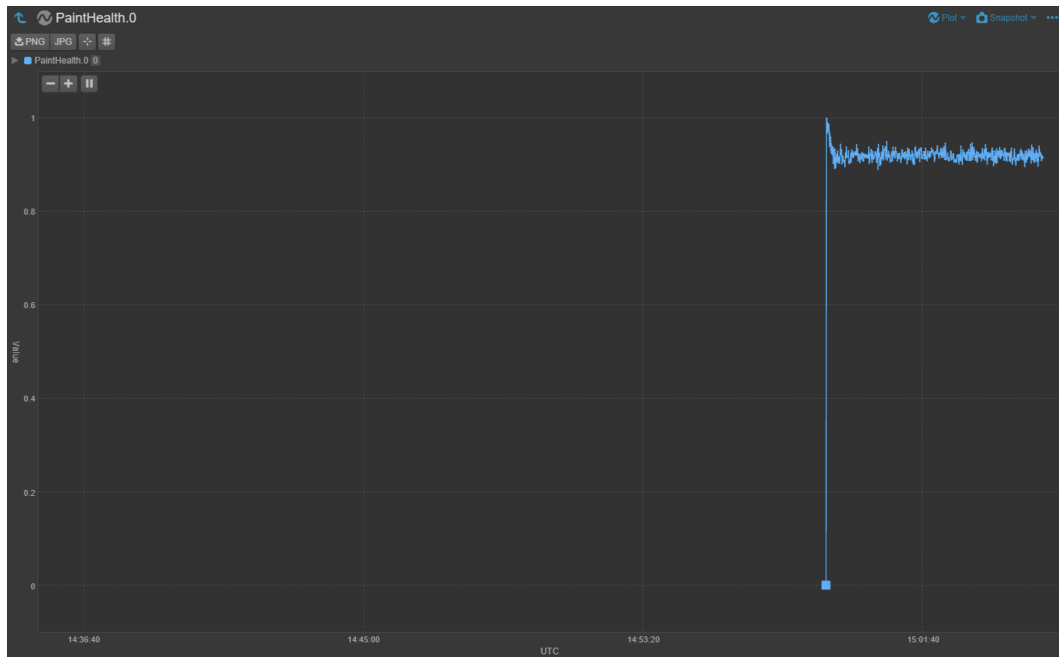


Figure 16. Live Data Visualization

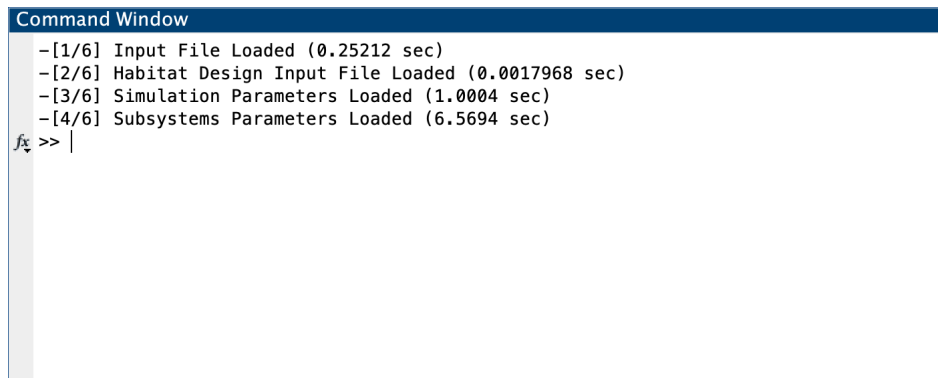
Additional functionality of set-point control using the OpenMCT will be discussed in Section 3.

3 MCVT-Docker Communication

3.1 Running Test Scenario

For testing the functionality of the communication network emulator, a test scenario of the MCVT with set values is run to generate the data sent to the emulator.

To run the test scenario, first download the MCVT repository, and run **MCVT_v6_RunScenarios.m** to load all parameters for the scenario. Please check Section 1.1 for the installation guide for all required MATLAB toolboxes if the program returns any error. Proceed to the Simulink after the following messages are printed in the terminal.



```
Command Window
-[1/6] Input File Loaded (0.25212 sec)
-[2/6] Habitat Design Input File Loaded (0.0017968 sec)
-[3/6] Simulation Parameters Loaded (1.0004 sec)
-[4/6] Subsystems Parameters Loaded (6.5694 sec)
fx >> |
```

Figure 17. RunScenarios.m Terminal Output

After the Simulink model is loaded, on the Simulation toolbar change the Stop Time to a large number so MCVT will be kept running long enough to test the functionalities of OpenMCT. Then press **Run**. It might take a few minutes to compile depending on your device, but once it starts running, go to "localhost:8080" on your browser of choice. Following the procedure described in Section 2.2, the synthetic observables will start to be plotted in real-time. These synthetic observables are critical in determining the health status of the components of the habitat.

3.2 Communication Between the Docker and a Separate MCVT Computer

If MCVT and the communication network emulator are run on two separate devices, the two devices can be connected via Ethernet. On the Docker computer, go to the **docker-compose.yml** file, and inside, replace **host.docker.internal** in quotation marks with the IP address of the MCVT computer for the following entries:

```
ADDR_REMOTE_STR: "host.docker.internal:20002"  
ADDR_REMOTE_SPL: "host.docker.internal:20011"  
ADDR_REMOTE_PWR: "host.docker.internal:20003"  
ADDR_REMOTE_ECLSS: "host.docker.internal:20005"  
ADDR_REMOTE_AGT: "host.docker.internal:20006"  
ADDR_REMOTE_EXT: "host.docker.internal:20007"  
ADDR_REMOTE_IE: "host.docker.internal:20008"  
ADDR_REMOTE_DTB: "host.docker.internal:20009"
```

Figure 18. Docker-compose file Entries

On the MCVT computer, navigate to SysFiles/Comms/Comms_Config.c and change the IP address to the IP address of the Docker computer.

3.3 Communication Between the Docker and Speedgoat

The setup for communication between the docker computer and Speedgoat real-time machine is similar to the previous section. Before uploading the Simulink model onto Speedgoat, make sure to change the IP address in the *UDP Send* block to the IP address of the Docker computer (see section 4.1 for more information). And modify the *docker-compose.yml* file with the ID address of the Speedgoat machine.

4 Communication Performance Requirement Verification

4.1 Sending Data from Simulink Model

Any data generated in your Simulink model is sent to the Docker container through a UDP Send block in Simulink, and before sending the data into the UDP Send block, you would have to prepare the data into the format desired by the database. The format and standard procedure are shown in the figures below.

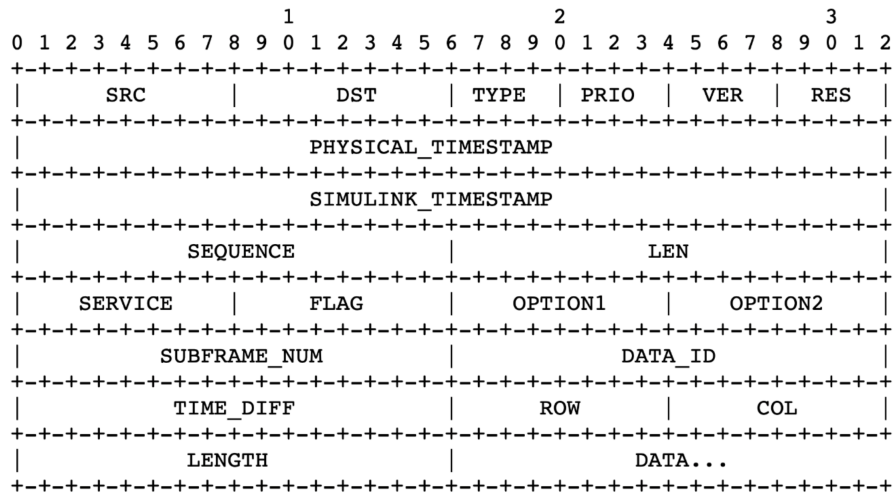


Figure 19. Data Service Protocol Format

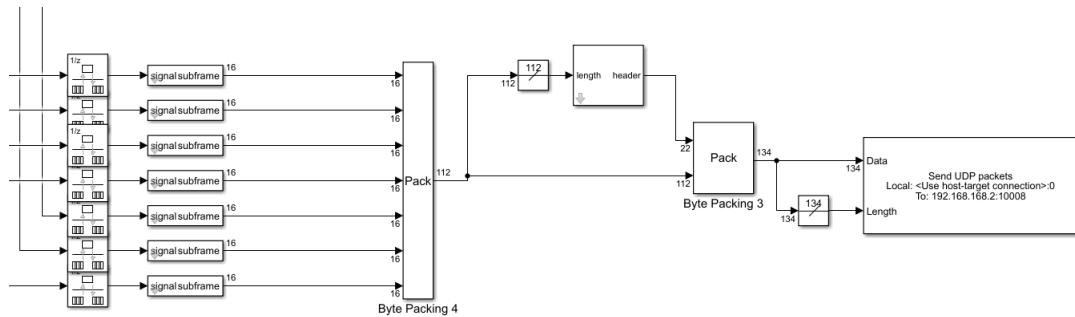


Figure 20. MCVT Communication API

In this figure, we can see we have 7 signals coming from the left, these signals are first sent into rate transition blocks so the output signals have the desired frequency

(the frequency at which the Byte Packing and UDP Send are operating under). Then the signals are first sent into a custom-made block called **signal subframe**. If you double-click on this block, you will be prompted to enter the `data_id`, which is the id of the record that you wish to send this data to in the database (you can find all the record id's in the `db_info.json` file). As you can see in the diagram above, the vector coming out of the block has a length of 16. The number 16 bytes can be justified when you further inspect the block by opening it up. In the figure below we can see what additional information has been added to the signal:

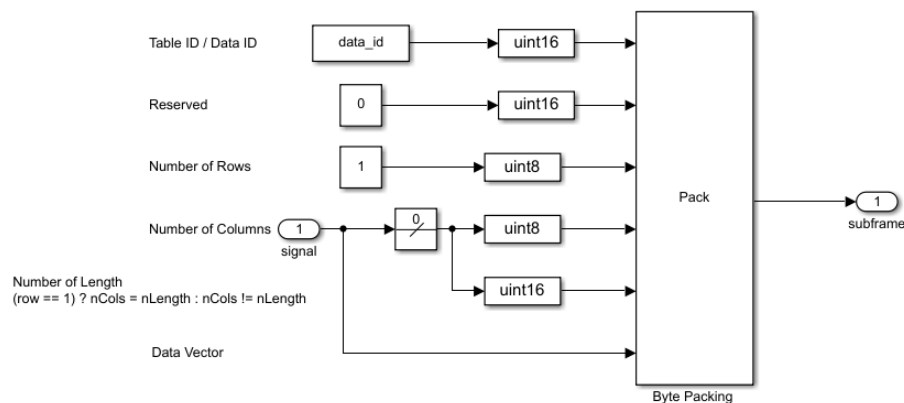


Figure 21. Signal Subframe

Each piece of information is converted into `uint16` which is 2 bytes, or `uint8` which is one byte, and then all stitched up with the data (double data type has a length of 8 bytes) to be 16 bytes in total.

Then all the signals, now labeled with the `data_id`, length, etc., are all stitched into one vector using the *Byte Packing 4* block. The head of the second signal is connected with the tail of the first signal, and so on. It is important to double-click on the *Byte Packing 4* block to make sure the number of signals going into the block matches the number of 'uint8' in the *input port data type* field. This vector is then sent into the top block to create more labels such as the Simulink time when the signal is sent, SRC, DST, etc. These labels are called the header of the entire packet.

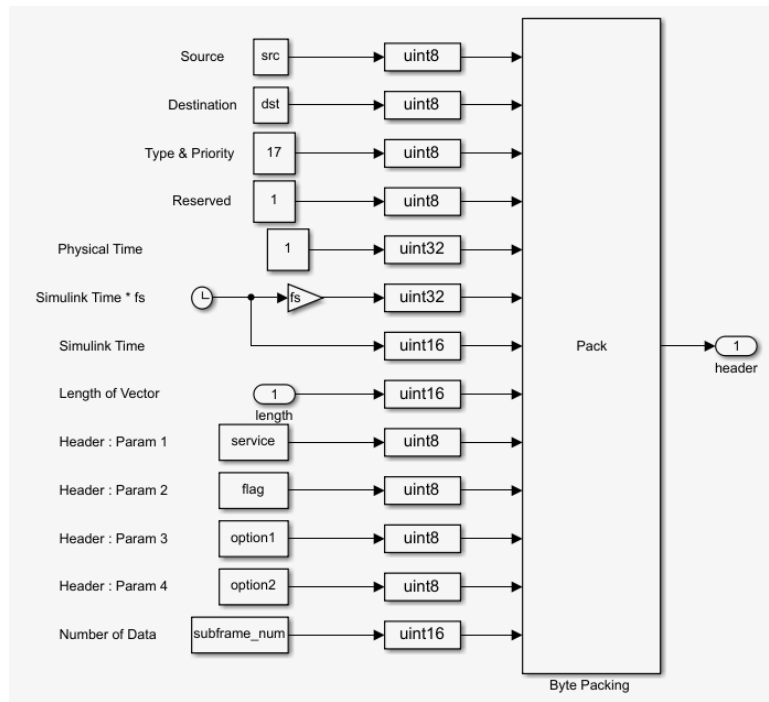


Figure 22. PacketHeader

In the figure above, you can see there is one line consisting of a clock and a gain, then converted into uint32 (4 bytes). This is where the index of each data is created, the same index used in the database. And when you double-click on both the clock and gain you can see that there are two parameters that you can change. The parameter inside of the gain block is predetermined by the *RunScenario.m*, but the decimation parameter inside the clock is free to change. **To ensure there is no packet loss**, which is due to multiple data points having the same index caused by the precision of the clock decimation, make sure the decimation is set to 1 here. Going up a level, back to the block diagrams of the entire API, double-click on the *PacketHeader* block and make sure the field *SubframeNum* equals the number of signals going into *Byte Packing 4* block. This ensures that all data will be sent. Lastly, this newly created packet header is then stitched with the signal using *Byte Packing 3* block, and lastly, this packet goes to the *UDP Send* block which sends it to the database in Docker. To make sure the packet is sent to the correct location, double-click on the *UDP Send* block,

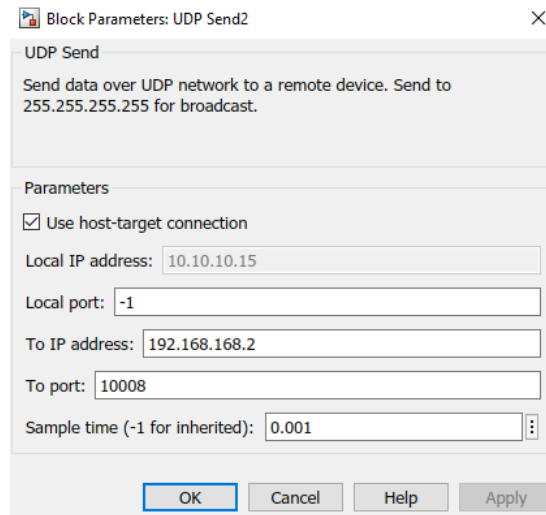


Figure 23. UDP Send Block Parameters

For the field *To IP Address*, enter the IP address of the Docker computer, and you can also change the *To port* field to indicate which subsystem this packet is sent from, in this case, port 10008 indicates the data is sent from IE. Lastly, make sure the sample time is consistent throughout all blocks.

It is important to understand that all the signals going into one UDP block are sent as **one** packet. Note that depending on the number of signals, each packet sent from the different UDP blocks will have different length as well. The maximum number of bytes a UDP block can have is 65535 bytes.

Now you should be able to create your own API and send arbitrary data for various tests, and in the next section, we will present a few key tests done to verify the requirements of the communication network.

4.2 Verifying Performance Requirements

4.2.1 Packet Loss and Time Delay

Originally, the issue of packet loss was due to the indexing of each data point in the database. And the large time delay was caused by an infinite loop within the code taking up all the CPU of the Docker computer. Now that these two have been fixed (make sure to set the decimation of the clock inside the *PacketHeader* block to 1),

there is no packet loss in the database or any significant time delay. The received data is then plotted with the original data obtained from Simulink, the difference between the sent and received data is on the order of 10^{-7} at largest.

In the Simulink API and within the DRDS, there were checkpoints where the sent time and received time can be recorded, and logically, these two values can be used to calculate the time delay of the data transmission and recording. However, a problem was that when cross-device communication is performed, Speedgoat to Docker computer, the clocks on the two devices are not synchronized therefore the times recorded become meaningless.

In Simulink, you can use a profiler to profile the execution time of your communication network API with various numbers of signals and UDP blocks. To run your model on Speedgoat with such a profiler active, use the following Matlab script

```
% Defining the target machine
tg = slrealtime('Baseline_RETHi');
% Put in the name of your real-time machine above

% Building and Loading the Simulink model
rtwbuild('test'); % Put in your file name here and the line below
load(tg, 'test');

% Starting the profiler application
startProfiler(tg);

% Check the profiler status and start running the machine with
% the application loaded
tg.ProfilerStatus
start(tg);
tg.ProfilerStatus
```

After the simulation is finished, run **tg.getProfilerData** in the Command Window to open the profiling report where the execution time, as well as turnaround time, will be displayed. It is found that the time between the data generation and sending the data through UDP blocks, including preemption time is on the order of microseconds for a simulation rate of 1000 Hz.

4.2.2 Misordered Data

It is noticed that there are a number of times when the order in which 2 or more data are recorded in the database is not in chronological order. However, when the clock decimation is set to 1, the effect of the misordering is minimum since it only happens for a few thousandths of a second at a time. Additionally, the data values corresponding to each of the misordered indices are verified to be correct after reordering. When the REQUEST service is used, the data will be re-ordered to make sure of its chronological order. The Command and Control (C2) uses the SUBSCRIBE service which bypasses the database, therefore, the misordering in the database would not affect its decision-making. The presence of misordered data is inevitable as the data is stored in the DRDS in a parallel fashion, and it is possible for some future data coming in from one connection to be stored before previous is stored from a different connection. If only one connection to store data in the database is used instead to prevent the misordering, the operation is slow, and it leads to the potential of packet loss. A checkpoint is set before the data is stored in the DRDS (inside of the server.go file, recreation of the image and compilation of the Docker container is needed) to print out the iteration number, time, and value of every packet, and it can be verified that all the packets going into the database are in order. The misordering only happens when the data is being stored into the database.

4.2.3 Multi-source, multi-signal

To demonstrate this capability, a few UDP blocks each taking a various number of signals are put in one Simulink model and run on Speedgoat. A partial screenshot of the model is shown:

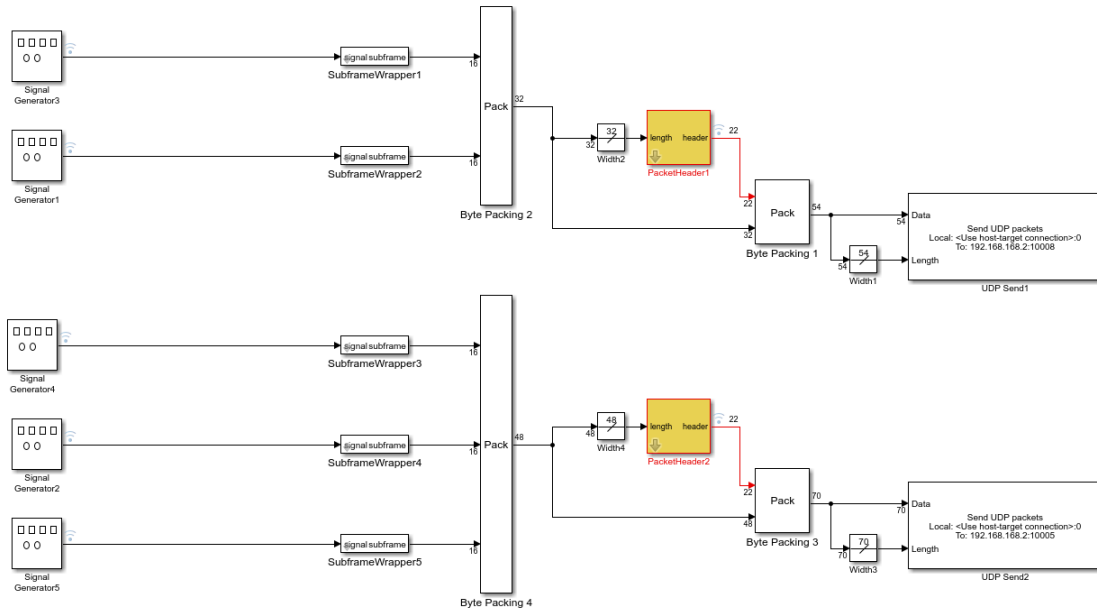


Figure 24. Multi-signal Multi-source Testing

As you can see in the *UDP Send* blocks, the port numbers to which the packets are sent are all different, simulating the multi-source aspect of this testing. The integrity of the data transmission under traffic can be verified in two ways: through the communication front-end and by checking the values in the database. In the front end, we are able to inspect the number of packets sent and transmitted through each node and switch, and the numbers all match expectations.

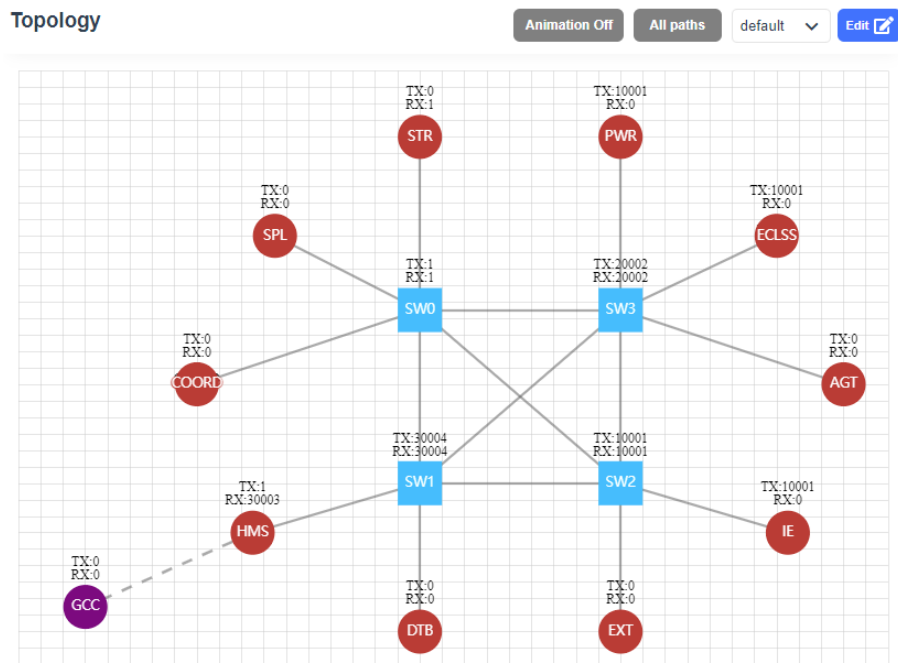


Figure 25. Communication Network Front End Network Topology

And once again, after reordering, by taking the difference between the received and sent data, the largest difference is on the order of 10^{-7} . So the communication network and DRDS have the capability of handling multi-signal, multi-source traffic.

5 Appendix I

Sample pkt_generator.py code for communication network testing.

```
import pyapi.api as api

import time

import json

import random

import json, random

with open("../db_info_v6.json") as f:

    data_discript = json.load(f)

# Simulation for 600 seconds

for synt in range(99, 99 * 600 + 1, 100):

    for name, data in data_discript.items():

        if name != "Health_Management_System":

            ins = api.API(

                local_ip="0.0.0.0",

                local_port=61234,

                to_ip="localhost",

                to_port=10000 + int(data['data_subtype1']),

                # to_port=65533, # for local testing

                # to_port=65531, # for data service testing

                client_id=int(data["data_subtype1"]),

                server_id=1,

                # set_blocking=

                # False, # Setting it to "True" causes issues when testing

            )
```

```

value = [random.random() for i in range(data["data_size"])]
ins.send(
    synt=synt,
    id=data["data_id"],
    value=value,
    priority=3,
)
time.sleep(1e-3)
ins.close()
time.sleep(1e-6)
time.sleep(1)
print("Simulation time -----", synt)

```

6 Appendix II

Figure 24 provides an overview of the new Docker-MCVT communication schematic.

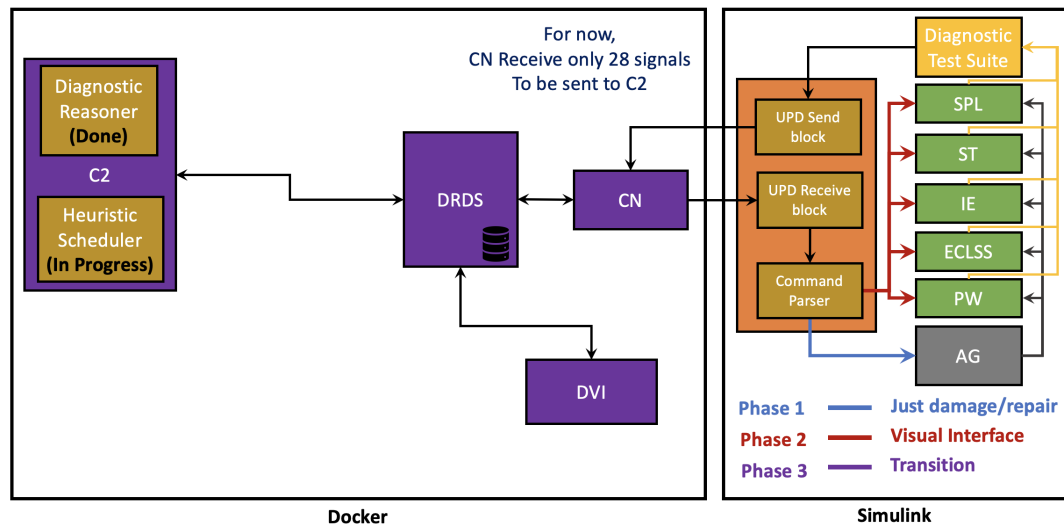


Figure 26. Docker-MCVT Communication