Feasibility of Docker for Emulation of Satellite Communication Network

Austin Liu
Department of Electrical and Computer Engineering
Duke University, Durham, NC, 27708, USA

Jeffrey Gilbert
Strategic Center for Education, Networks, Integration, and Communication
NASA Glenn Research Center, Cleveland, OH, 44135, USA

Robyn Atkins
Strategic Center for Education, Networks, Integration, and Communication
NASA Glenn Research Center, Cleveland, OH, 44135, USA

As the size of telecommunication systems in the Near Earth Network (NEN) and Space Network (SN) rapidly increases, the practice of integrating diverse analytical software into a single system model is desired in order to anticipate long term effects of technology on the existing infrastructure. The objective of this investigation was to determine if a suitable environment with distinct software processes could be merged to provide unambiguous collaboration. We confirmed in this paper that Docker, an application to merge distributed processes, can host an extensive emulation environment for satellite communication that ensures a robust collaboration system. This paper described in detail the CPU abd memory of computationally heavy applications, such as GNURadio, inside containers during parallel processing. We went on to conduct consistency tests to determine the extent to which containers share CPU processor cores, which is indicative of latency and processing capabilities. Through modeling a Near Earth Network (NEN) topology via Linux containers (LXC) in Docker, we compared the data transfer efficiency of serialization, piping, and socketing methodologies. To emulate bulk data transfer, we also explored using shared memory to reliably model larger bandwidths and data rates. From these experiments, we confirmed the feasibility of Docker as a reliable framework for interprocess communication.

I. Introduction

Simulation and emulation of satellite communication systems are critical for mitigating the risk of launching satellites into orbit. Current evaluations of space and terrestrial networks use a variety of software to determine orbital dynamics, channel and link impairments, and data traffic behavior in order to provide high-level modeling and simulation for dynamic communication systems.[1]

The number of satellites in two of the major satellite communication systems, the Near Earth Network (NEN) and the Space Network (SN), has been constantly increasing. Accurately modeling different aspects of both systems currently requires the use of many distributed applications to examine simultaneous orbits, simulate satellite dynamics, and coordinate sequences of operations.

Existing tools for distributed simulations are currently being developed through the SCaN integrated DSIL operation.[2] Yet, as collaboration between teams from different companies and labs has become more prominent as the models and tools have become more accurate, a lack of a uniform environment to run distributed applications causes friction between experimentation, quality assurance, and development.

Several tools currently exist in industry to assist the easy construction of distributed applications, one of which is a native Linux application called Docker. Research by Paolo Di Tommaso and his team has indicated that there is minimal impact of Docker containers on large amounts of data transfer through

singular genomic pipelines.[3] With this in mind, we sought to determine the feasibility of expanding this concept into a larger network topology. We will demonstrate that Docker provides a suitable environment for satellite network emulation, which will also enable a robust collaborative environment.

I.A. Docker Environment and Statistics

To describe the network environment that Docker can create, we must first explain the components that the Docker application provides. The building block of the Docker environment is the Docker container, a Linux shared kernel that initially runs isolated from other containers and processes through an abstraction layer. This allows direct communication between the OS host, hardware, and applications residing inside the containers without sources of conflicts, such as CPU steal, memory contention, or port conflicts.

Docker provides functions that link containers together, expose ports, and assign specific IP addresses. Combined with Linux networking tools, Docker provides a stable and well-documented environment through its open source API for creating experimental network topologies.

I.A.1. Containers vs Virtual Machines

In abstraction, isolation aspects through network containerization from the Linux kernel make containers very similar to virtual machines. There are, however, a few key differences:

- Docker uses Linux Containers (LXC), where any particular container runs in the same OS as the host application.
- Docker containers have less isolation than VMs but are extremely lightweight, allowing for quick building of large network architectures. A host capable of running tens of VMs would be capable of running hundreds of containers.
- Container initialization is extremely rapid in comparison to VM initialization.
- Containers do not necessarily need an operating OS, and have fewer overhead processes running in the background.
- Docker containers have image-saving functionalities, which allows users to commit and update changes to containers. This quality also makes parallel processing ideal.

To better understand the statistics of container initialization, experimental tests were conducted to obtain startup metrics, in which the results are displayed in 1. It is important to note that the results displayed in table 1 are particular to the environment in which we initialized the containers. Specifically, we conducted our experiments on a commercial off-the-shelf (COTS) entry level server. Table 2 below identifies the specifications of the server we used for the study.

Table 1. Container Statistics

Container Parameters	Average Experimental Values
Startup time	0.5 to 5 s
Exit time	1 to 4 s
Memory overhead at start-up	7.268 MB
Network throughput (iperf)	17.5 Gbits/s

In terms of performance, the startup times for VMs were more than 10 times slower than those of containers based on the OS the virtual machine was running and its hardware specifications.[5] With the extra overhead that the experimental VMs came with, containers were ideal for the network nodes that we created, which in our environment had only one or two processes running in parallel.

Figure 1 indicates three successive startup processes of containers based on a CentOS7 image. Container startup procedures on a typical machine use a range of 0.20 to 0.45 processor cores or about 4% CPU for approximately one second, and drops off immediately afterwards. However, the ultimate advantage of containers lies in their low overhead. For detached containers, total consumption tended to be $< 1\%$ CPU, meaning that hundreds of containers could be initialized without much cost to the system.

Container exit processes used about twice the number of CPU process cores as startup processes, which can be seen in figure 2 below. The resource draw for exit process was therefore less intensive than a standard

Table 2. Platform Specifications

Parameter	Specification
Docker Version	1.6.2
Kernel Version	3.10.0-229.7.2.el7.x86_64
OS Version	CentOS Linux 7 (Core)
Number of Processors in System	8
Processor CPU Clock Freq	1600 MHz
System Memory Capacity	2 GB
Model Name	Intel (R) Xeon (R) CPU E5620 @ 2.40 GHz

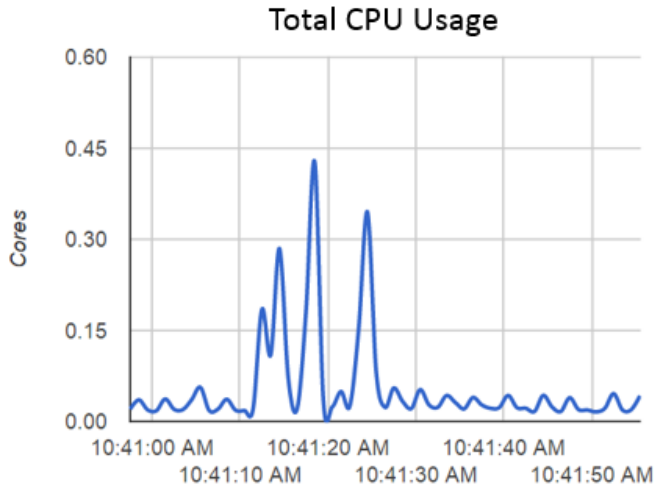

Figure 1. Typical CPU Strain on Three Successive Docker Container Startup Processes

virtual machine. However, there may be contention for CPU if large numbers of containers exit at the exact same time, which would undoubtedly increase the time needed to clear a congested network.

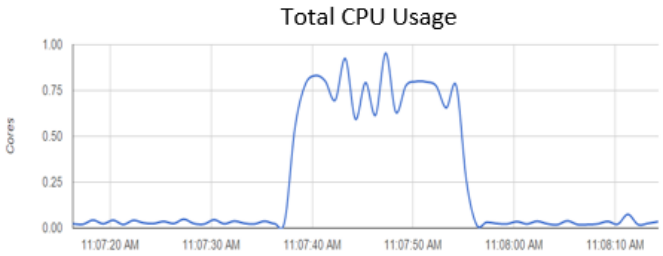

Figure 2. Typical CPU Strain on Eight Successive Docker Container Exit Processes

II. Methods and Procedures

II.A. Initial Satellite Emulation Environment

We have considered multiple models based on an idealized network topology that represents the Near Earth Network. Specifically, we used a finite state automation methodology[6] to demonstrate that all possible geometric arrangements of satellite orbits could be created by modifying resource allocations and maximizing allowable throughput in containers interior to the Docker environment described in 3.

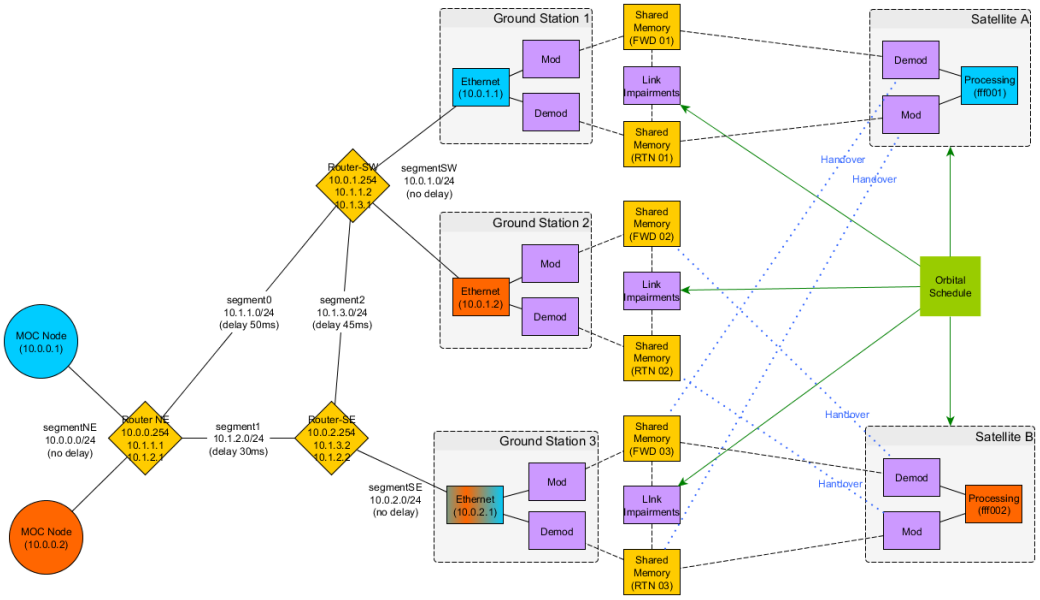

Figure 3. Docker Network Topology of Terrestrial and Satellite Network

To emulate the visibility changes of network nodes, we containerized each individual process and created Docker templates called images for rapid startup of the following components:

- Mission Operation Centers
- Ground Stations
- Link Impairment Nodes
- Satellite Nodes
- Orbital Schedulers
- Terrestrial Routing Nodes

For communication between terrestrial and space nodes, data is transferred through electromagnetic waves. Because Docker does not currently have any inherent tools that allow for signal modulation, we adopted a heuristic approach and constructed, modulated, and demodulated signals interior to the software radio development toolkit: GNURadio. Instances of GNURadio resided inside the ground station, satellite, and link impairment containers. Terrestrial communications between ground stations and mission operation centers implemented standard Linux networking commands such as *netcat*, *socat*, and *iperf*.

One of the benefits of including an orbital scheduler node in the topology is that parameters of each node can be modified before, during, and after emulation. When Docker containers are created, they are initially isolated and there exists no network connectivity between any individual container instance. However, the Docker environment provides linking functionalities, allowing for containers to not only communicate information, but also share files and memory locations. Since almost any process can run inside of a Docker container, it is very possible to implement an orbital scheduler or allow external programs to assign parameters for a specific satellite orbit. Since our focus was on the performance of Docker, we decided to use a static timing of a single handover, without contention, and with varying levels of processing overlap.

For this model, traffic was generated on the host, was channeled through the docker ethernet bridge (docker0), and was directly passed through GNURadio modules inside signal processing containers. However, standard networking inside container clusters is already commonplace.[7] Although not currently implemented, an additional expansion to the network capabilities of the Docker environment would be to create a template for a specialized router container, where an example can be seen in figure 4. For terrestrial communication, router containers would provide the functionality to modify datalink layer communication, as well as filter and manipulate IP traffic.

Using this topology, metrics were obtained from an overseer container which was external to the network. The overseer container obtained information about other containers from the central data source on the

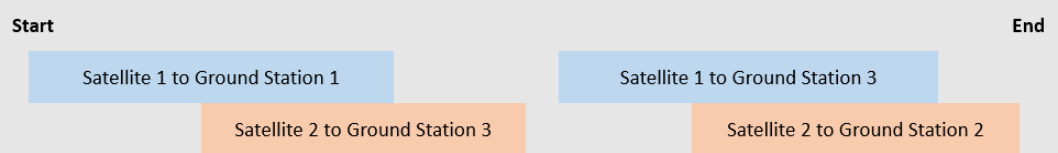

Figure 4. Simple Satellite Connection Schedule

host using the Docker *stats* API and was offloaded on a private log server using *cAdvisor*. As the collection of data was completely done inside of an isolated container, multiple instances of the container could have existed simultaneously.

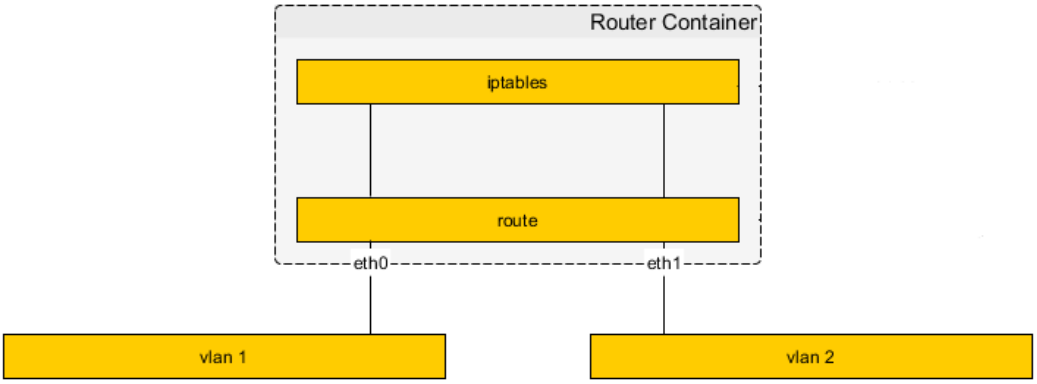

Figure 5. Hypothetical Specialized Routing Container

II.B. Signal Processing through GNURadio

Signal Processing nodes in our topology used GNURadio exclusively to generate software radio signals. However, the flexibility of Docker containers allowed implementation of other commonly used tools such as Matlab, Octave, and pre-compiled code written by other engineers. The signal transfer process that we conducted was based on on the following steps:

1. The host sent IP traffic over the Docker bridge from the Mission Operation Center (MOC) nodes towards the clusters of ground stations in Figure 3.
2. Ground stations obtained traffic from MOC nodes, implemented PSK modulation on input signals, and broadcasted the modulated message across shared memory, sockets, or pipes. The output medium depended on the method of data transfer that was chosen.
3. Link impairment nodes obtained the modulated signal from the ground station output and aimed to model impairments introduced by the propagation channel such as Doppler effects or noise additions. Complex effects such as attenuation, scintillation, or Ionospheric effects also exist,[8] and could be introduced as parameters using an external scheduler node. In this experiment, we used a simple noise source.
4. Satellite nodes obtained the output of the link impairment nodes, demodulated the signal, and reversed the process back to the MOC nodes. Visibility of the satellite nodes was based on parameters specified by the orbital scheduler node.

One of the experimental modem diagrams is shown in Figure 6 below, which shows a modulation and demodulation schema using QPSK. The experimental setup also includes other phase key shifting modulation methods such as BPSK and 8-PSK.

Because visibility of satellite nodes are based on orbital parameters, many signal transfer tests using GNURadio were required to be burst tests. Fortunately, GNURadio provided block functionality that dealt with insertion of preambles, encoding, and decoding.

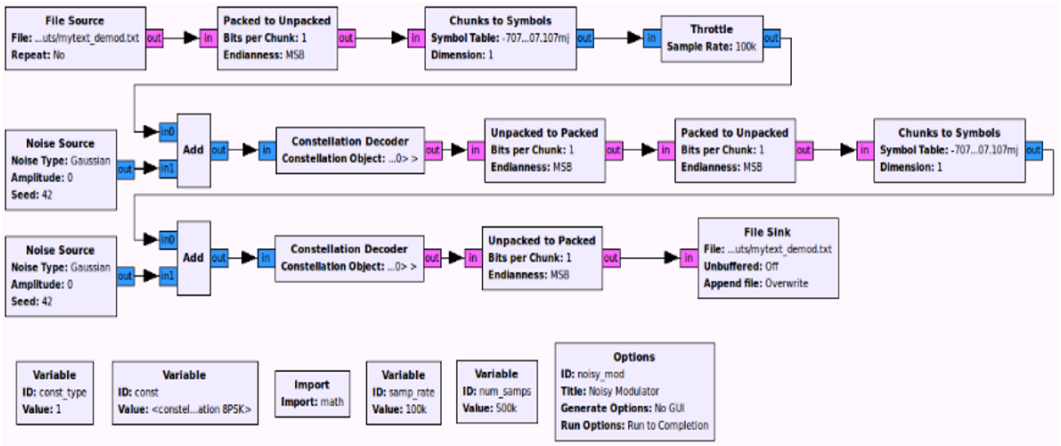

Figure 6. Modulation and Demodulation Block Diagram using QPSK

Both throughput and CPU usage were controllable via throttles provided by the GNURadio module. Hence, the primary bottleneck in performance comes from memory consumption. To determine the feasibility of using computationally heavy software interior to Docker containers, we analyzed the memory overhead of two processes of GNURadio inside a single container. Specifically, one processes was kept on and the other turned on and off in sequence over a 60 second time period, and can be seen in Figure 7.

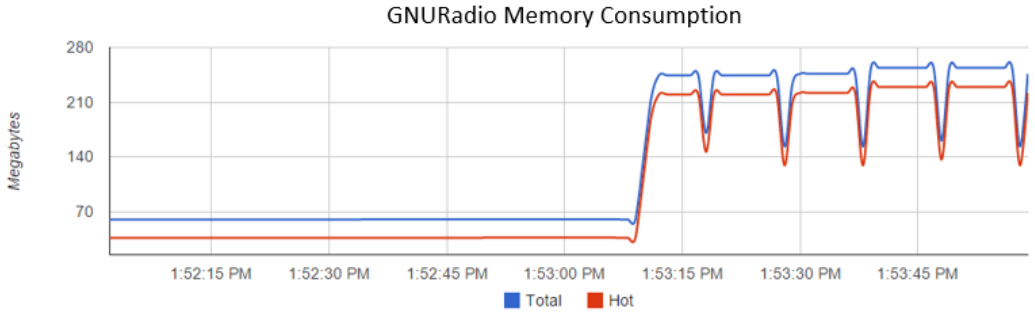

Figure 7. Containerized GNURadio Memory Consumption

While idle, each GNURadio process used 20-30 MB of memory. At peak consumption, each GNURadio process consumed 90-120 MB of memory. As can be seen in Figure 7, memory consumption as well as processing speed for individual processes was consistent. Therefore, with the experimental network in Figure 3, memory bottlenecks posed a minor issue that could be worked around. However, with a realistically-sized network, a more memory efficient processing method would have needed to be used. Determining where the inefficiencies are and how to compensate for streamlining containers to process larger networks would allow a good indication of larger network feasibility.

The GNURadio repository provided many helpful blocks including those for data sinks and sources. The flexibility of the transferring traffic allowed limitless options for the format of the input and output data. Some processing blocks were not provided by GNURadio initially; however, GNURadio provided for custom block creation and code modification using Python modules. This was useful in experimenting with different methods of data transfer.

III. Results and Discussion

III.A. Data Transfer Metrics

In order to emulate bulk transfer between terrestrial and satellite components and to model larger bandwidths and data rates, it was extremely important to determine the most efficient method of data transfer between containers. Using the network topology shown in Figure 3, we determine throughput, CPU, and memory constraints of the following methods on the Docker host:

- Serialization/JSON encoding over linked file locations
- Socketing
- Transfer via host shared memory

III.A.1. Data Transfer through Serialized File Objects

GNURadio provides vector sinks which convert streams of data into tuples of data. By serializing the output of tuples, it became possible to write the converted object to a file descriptor, which could then be read on the receiving container. Docker has linking capabilities and is able to share directories between specified containers. Hence, transferring data via a shared directory would bypass communication over the bridge. Throughput was completely determine by serialization.

Average network throughput, which can be seen in Figure 8 was measured for the entire communication chain. In order to preserve congruity between tests of different data transfers, a throttle cap of 10 MB/s was introduced in the GNURadio block diagram. The data source was a random stream of data converted to binary to be read by the GNURadio module.

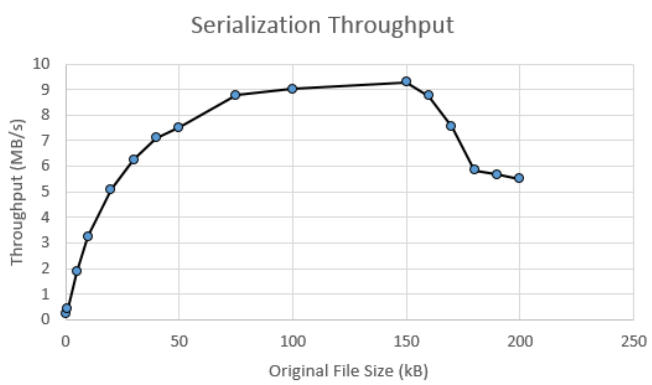

Figure 8. Average Throughput for Single Process Modulation and Demodulation using Serialization

Serialization was completed primarily through the Python module: *pickle*. Encoding using Python's JSON modules in principle transmit traffic using the same process. It was found that serialization efficiency gradually increased up to 9 MB/s for packet sizes below 150 kB. Lower throughput for smaller packet sizes was expected because the serialization function itself had startup time. However, for larger packets (> 150 kB), performance tended to decrease down to 5 MB/s for a 200 kB file size.

The reason for a decrease in throughput for larger data packets is not readily apparent. Yet, by examining the memory consumption during serialization, it becomes apparent that the pickling process consumed large amounts of memory. Normally, pickling is fast and efficient for storing small-sized data on disk. However, the limitation was that the *pickle* module required a large amount of memory allocation (> 2 GB) to save a data structure to file. This proved to be a bottleneck, especially when attempting to emulate bulk data transfer. This result pointed us towards determining a method in which to transfer data without writing to disk.

III.A.2. Data Transfer through Sockets

GNURadio also provided specific source and sink blocks for TCP and UDP connections. For this data transfer test, the preferred protocol to use was UDP in order to reduce complexity. By implementing UDP

sources and sinks in different containers, data traveled from one container through the Docker network bridge to another container using networking tools. Yet, although data traveled longer to reach the destination, the data was never written to disk. Memory consumption was not expected to be a bottleneck for socketing.

Socketing data packets was completed using standard Linux networking tools (*socat* and *netcat*). Once data was transferred to ground station nodes, socketing was completed entirely inside GNURadio. It was found that socketing followed the same increase in throughput efficiency for data packets, except for data packets below 40 kB, which can be shown in Figure 9. Data packets below 40 kB tended to decrease transfer efficiency because of function startup times. However, socketing throughput decreased exponentially for larger packet sizes. The reason for this arose from the processing speed of GNURadio blocks. Because the GNURadio modulator converted a binary stream into a complex modulated stream, sent packets needed to be larger than those that were received. Hence, the UDP listener throughput was faster than the UDP chatter, which caused interior buffers to get overloaded and data packets to get dropped.

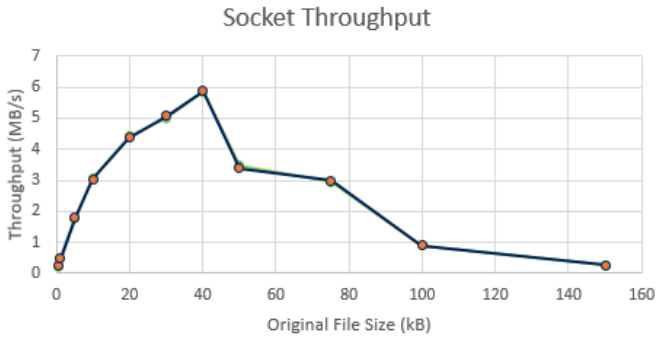

Figure 9. Average Throughput for Single Process Modulation and Demodulation using Sockets

Data transfer through sharing filesystems proved to be an extremely efficient method of transferring data. However, as memory proves to be a large bottleneck for process functionality, bypassing disk-writing is essential. We next explore the efficiency of data transfer through shared memory modules.

III.A.3. Data Transfer through Host Shared Memory

Using shared memory for interprocess communication (IPC) is commonly used where a single process creates an area in the RAM, in which other processes can rapidly obtain access. Typically, using shared memory as an IPC facility is common when large data structures need to be transferred.[9] For this analysis, the POSIX IPC shared memory module in Python was used to communicate between containers. Interior to GNURadio, file source and sink blocks were modified to directly interact with a shared memory location: */dev/shm*.

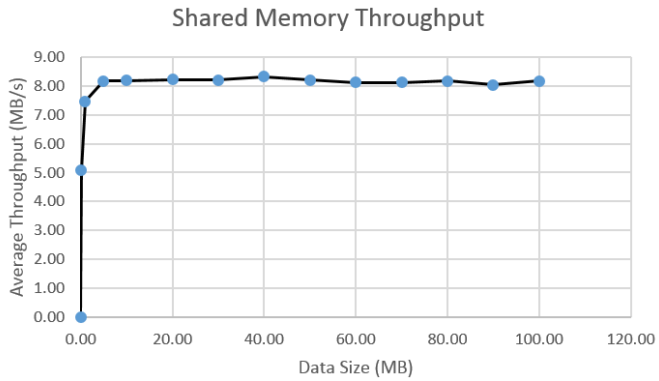

Figure 10. Average Throughput for Single Process Modulation and Demodulation using Shared Memory

Figure 10 illustrates that data throughput did not experience the same decrease as socketing or serialization. It was found that shared memory throughput for packets with a size greater than 5 MB had little

fluctuation, while maintaining an average throughput of 9 MB/s that was significantly higher than that of other methods.

Shared memory tended to perform well for large blocks of data transfer, but for blocks of data less than 1 MB, throughput tends to decrease dramatically, with an average throughput of 5 MB/s for a packet size of less than 1 MB. This could be attributed to the fact that it took time for the Python shared memory module to initialize, configure semaphores, and begin transferring data. Hence, shared memory was useful for burst transferring, and less useful for streaming.

III.A.4. Throughput Stress Tests

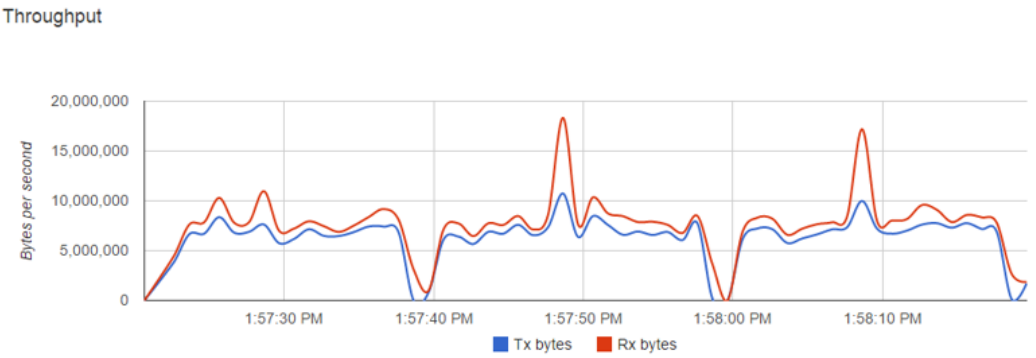

Figure 11. Throughput without Signal Processing

In Figure 11, we ran baseline traffic without DSP to show the Docker internal traffic capacity. The graph shows a satellite coming in and out of view from various ground station nodes. Areas of no visibility accounted for the dips in throughput shown on the graph. This traffic was constrained using the traffic control (*tc*) tool to add delay and traffic limits to links between router nodes, which we set to 10 MBps for simplicity. System limits seemed to be around 8 Gbps across the bridges when no traffic controlling was in place.

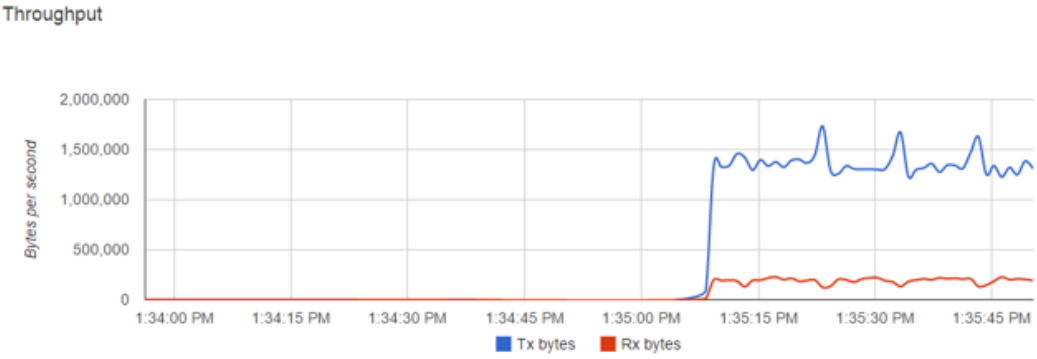

Figure 12. Early Throughput with Signal Processing

The large discrepancy between transmission and receiving throughput shown in Figure 12 can be attributed to dropped packets. In the early stages of testing we found buffer overflow issues in our GNURadio processing. This can be attested to the use of serialization to pass data between containers. This overhead of serialization, as expected, was costly and started dropping packets.

Once we removed all serialization processing between containers, we were able to process two full streams at 600 KBps consistently without packet loss. Our constraining factor became CPU usage of the signal processing.

III.B. CPU and Memory Strain Running Distributed Applications

Usage of CPU and memory is important in determining Docker feasibility because they are directly correlated to maximum network capability. A feasible environment would allow for scalable consumption of resources.

III.B.1. CPU Strain

CPU strain is especially important when running the same application on different devices. In this case, it was important to determine whether CPU consumption for GNURadio remained consistent when running in different containers, and whether running multiple instances caused too much strain on the system processors.

The system used for satellite emulation used 8 cores, all of which are labeled in figure 13. For a baseline the system was running a minimal headless system, and with all containers running the CPU usage was about 8%.

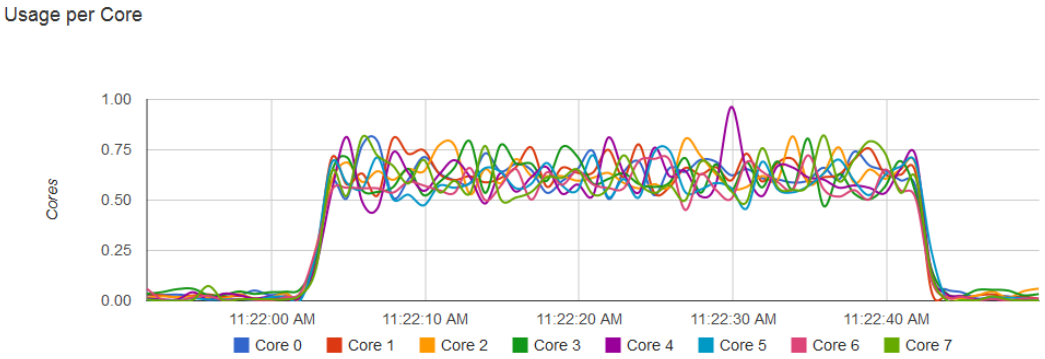

Figure 13. CPU Usage Per Core during Network Emulation Single Satellite

Figure 14 shows CPU consumption of the Docker host with a single satellite processing data. However, analyzing CPU usage per core in individual containers shows the same behavior. The CPU usage gives a good sign that Docker containers were used efficiently, and that the shares of CPU time for each container are evenly distributed. With a single satellite processing data the CPU is being utilized at 75%.

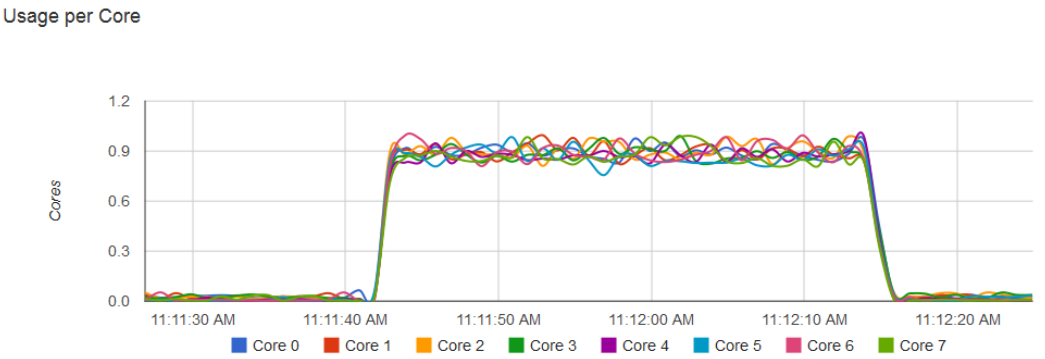

Figure 14. CPU Usage Per Core during Network Emulation of Two Satellites

Running processing on two satellites pressed the CPU usage to between 95 and 100%. In both cases, container CPU usage scaled linearly with the number of instances of GNURadio running. For Docker to emulate larger networks, we could:

- Scale hardware to increase the number of processors
- Introduce throttles and specialized CPU allocation with some cost to throughput
- Optimize signal processing with GNURadio
- Switch to more resource efficient processing methods
- Schedule to ensure that only a certain number of processes may run at the same time

III.B.2. Memory Strain

Memory constraints become major bottlenecks when the combined memory of all processes in Docker exceeds the system memory allocation. Thus, it is important to determine whether memory consumption remains relatively low, whether there exists memory leaks between data transfers, and whether process of GNURadio can run on different interfaces with consistent memory consumption.

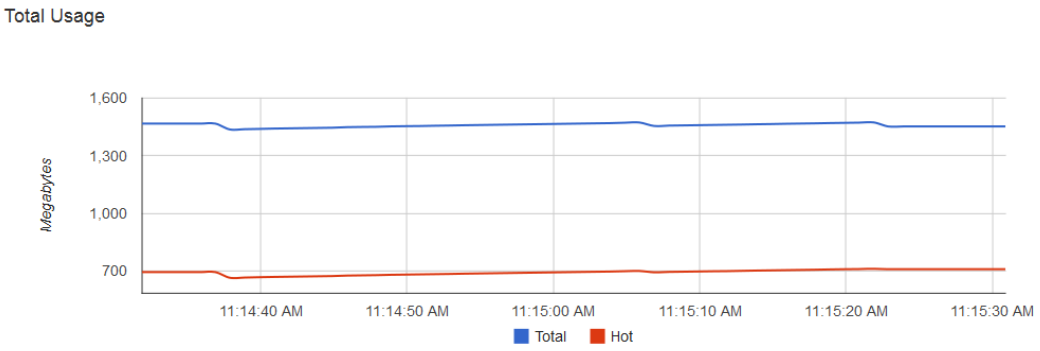

Figure 15. Memory Usage during Network Emulation 2 Satellite

As can be seen in Figure 15, the Docker system allocates memory for each running container. During the emulation, changes to memory consumption were minor. Memory consumption scaled linearly with the number of containers running at once. Docker also provided mechanism through command line tags to limit memory per containers so it is possible to manage the elements being used to a greater degree.

III.C. Model Network Experiment of Near Earth Network (NEN)

The topology in figure 3 provided a solid exploratory environment for metrics gathering. However, a more robust environment shown in Figure 16 provides better decomposition of functions into containers, a best practice for working with Docker.[10] This decoupling of the problem space would allow us to drop in and out different components in the signal chain without disruption of the other components.

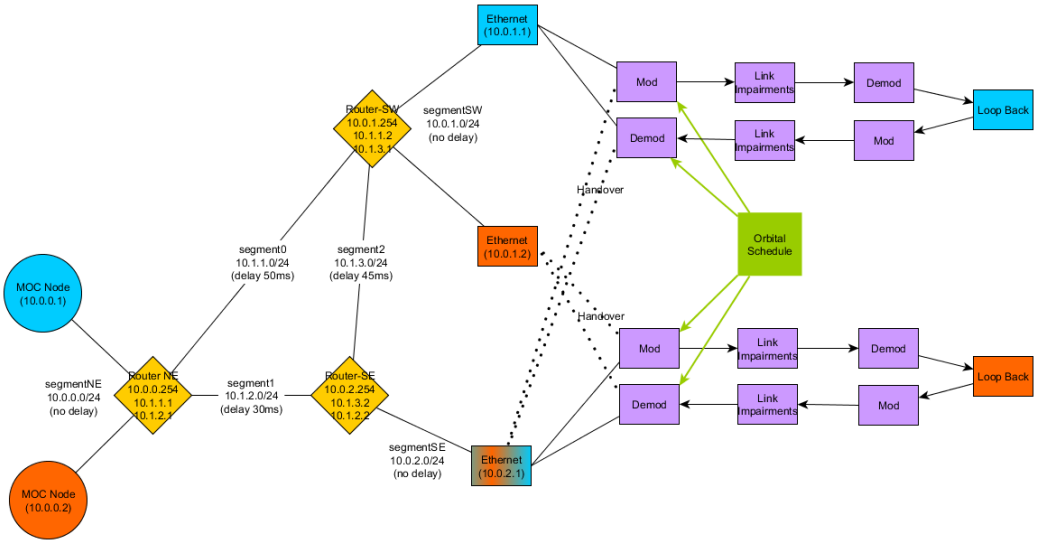

Figure 16. Sample Near Earth Network Topology

Between Figure 3, the primary difference in using Figure 16 occurred between terrestrial and satellite nodes. Specifically, satellite nodes were no longer limited to a single container, but consisted of a cluster

of containers: unidirectional modulators, demodulators, and link impairment nodes. Ground station nodes had their own IP addresses and contained metric analysis tools instead of *cadvisor*.

Communication existed between containers residing on their own network bridges using a Docker extension called *Pipeworks* and Linux bridge tools. The purpose behind isolating the terrestrial containers was because it provided a logical way to control network traffic without using network masking on the Docker *bridge0*.[11]

III.C.1. Data Transfer via FIFO

The final solution that provided consistent data transfer was to use common host FIFOs, in which each container is given access to the FIFO that resides on the host shared between the containers. The ingesting container established a listener on shared FIFO processes. The incoming stream then outputted to a shared FIFO that the next container was listening on. GNURadio file descriptor blocks provided stream processing entry and exit points while using standard OS commands, which redirected to get data to and from FIFOs to process.

Each ground station had an eternal UDP connection which redirected to the entry FIFO structures using *socat* listening commands. A *socat* command redirects the returning FIFO back to the MOC. Within each of the satellite processing containers, we are able to bring up and reset the FIFO pointers as a method to establish when a satellite was connected to a ground station. This also had the added benefit that retained satellite process containers without restarting.

III.C.2. Fifo Network Metrics

When implementing host FIFO piping, the limiting factor for signal processing was constrained by CPU usage. We were able to process consistent 600 Kb/s throughout the system with all containers running.

We ran full traffic through the system bypassing the signal processing blocks as an initial baseline to determine underlying CPU usage using the system level FIFO processes. We redirected the incoming FIFO through the containers directly to the exiting FIFO. CPU usage in this case was approximately 15%. This included the processing for the twp traffic generators, all containers, system overhead, direct FIFO redirecting, and monitoring applications.

III.D. Collaboration Environment

Docker provides ease of collaboration through container utilities. Beginning with Docker's efforts in evolve the Open Container Initiative (OCI), multi-industry collaboration in code development has boomed. The SCENIC lab aims to expand the capabilities of its network emulation environment by possibly implementing applications in Docker.

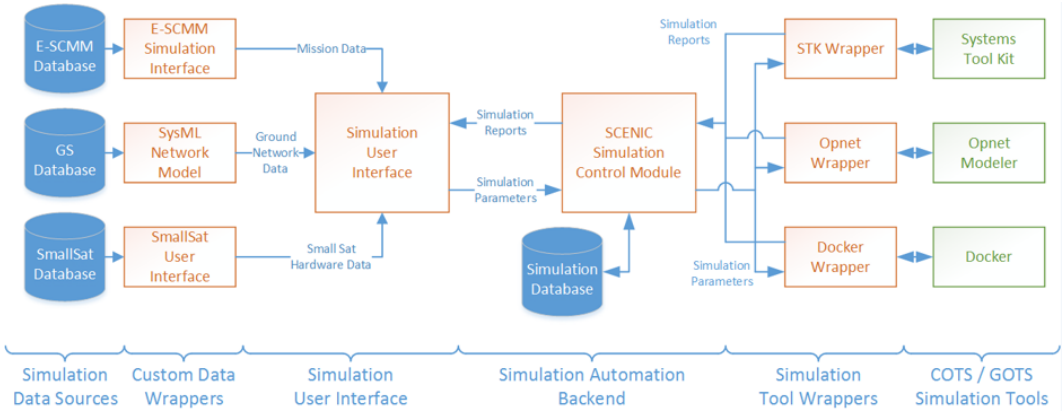

Figure 17. Sample Docker Implementation into the SCENIC Mode Control (SMC)

Ideally, Docker would act as a COTS emulation tool in the SCENIC Model Control. A sample configuration can be seen in Figure 17. Collaboration could occur in the following process:

- The Docker environment obtains emulation parameters or special containers from an external module or the SCENIC Simulation Control Module. Parameters, models, and closed form equations transfer to the orbital scheduler container.
- The orbital scheduler container formats obtained data and sends parameters such as attenuation rate and propagation delay to network nodes such as link impairment nodes.
- The orbital scheduler begins environmental simulation by initializing, pausing, and stopping containers based on their network visibility. Signal processing and network traffic control modules run in initialized containers.
- Simulation is conducted based on parameters. Host-level and Container-level metrics such as bit error rate, throughput, and resource strain are obtained either via a logging service or a metric gathering container.
- Simulation reports, graphs, and raw data are returned back to the SCENIC Simulation Control Module.

Current developments in Docker aim to provide even more robust functionality in the future such as MicroSoft Windows Server Containers[12] to expand the collaborative process even more.

IV. Conclusion

Our research has shown that the Docker application provides a solid infrastructure for network emulation. We were able to get reasonable throughput and processing metrics to continue to pursue more efficient and more robust configurations. The benefit of our approach is that it provided a rudimentary analysis showing that network emulation is indeed possible using Docker. However, the approach in using resource heavy applications caused a shortcoming in full scalability analysis. Future work would deal specifically with optimizing the Docker structure to provide a more base-level approach to signal processing.

The approach in implementing Docker command line functions allows for rapid testing of topologies. However, a more detailed method would be to work directly with the Docker API, controlling functionality from the ground level. It was confirmed, nonetheless, that containers are more suitable to large networks than virtual machines. From analysis of containers, it was shown that typical usage of containers without running computationally heavy software provided negligible overhead, which allows for enormous scalability.

Different satellite emulation environments were created to demonstrate the flexibility of Docker's network capabilities. Through signal processing implementation with containerized GNURadio processes, resource consumption was analyzed to show that the typical usage of distributed applications scales linearly with the number of processes running at once, rather than the number of containers running. Although the procedure in this paper sacrificed network robustness for simplicity, the architecture of Docker allows for future advancements in increasing signal processing efficiency.

Various direct data transfer methods between non–network containers were experimented with. The most viable transfer methods allowing for the maximum attainable throughput without enormous resource draw was transfer through shared memory locations and transfer through host FIFOs. While the simplest form of transfer was through a generic network connection between containers, the overhead of that data passage was too costly.

With future developments into Docker's networking environment, the influence of the Docker application on providing a collaborative emulation environment is immense, especially as Docker is constantly being improved by the open source community. Future areas of research into the Docker environment would be to explore running Docker on multiple machines, multiple hosts, and larger networks. This study emphasized simplicity, but future efforts would improve the efficiency of topology setups. In terms of SCENIC goals, integration with the SCENIC Simulation Control Module is currently a priority. With more processing cores and larger memory allocations, emulation of large topologies is certainly feasible.

Acknowledgment

The authors would like to thank the Glenn Research Center for supporting the student internship program in the department of Space Communications and Navigation (SCaN).

References

[1]Johnson, Mark A. (2006) "Modeling, simulation, and analysis of satellite communications in nuclear disturbed environments with OPNET." *Defense and Security Symposium.* International Society for Optics and Photonics.

[2]Jennings, Esther, et al. (2009) "Space Communications and Navigation (SCaN) network simulation tool development and its use cases." *AIAA Modeling and Simulation Conference, Chicago, Illinois.*

[3]Di Tommaso, Paolo, et al. (2015) *The impact of Docker containers on the performance of genomic pipelines.* No. e1426. PeerJ PrePrints.

[4]D. Bernstein, *"Containers and Cloud: From LXC to Docker to Kubernetes"*, *IEEE Cloud Computing*, vol.1, no. 3, pp. 81-84, Sept. 2014, doi:10.1109/MCC.2014.51

[5]Bernstein, David. (2014) "Containers and Cloud: From LXC to Docker to Kubernetes." *IEEE Cloud Computing 3*, pp. 81-84.

[6]Chang, Hong Seong, et al. (1995) "Topological design and routing for low-earth orbit satellite networks." *Global Telecommunications Conference, 1995. GLOBECOM'95., IEEE.* Vol. 1. IEEE.

[7]V. Marmol, et al. (2015) "Networking in Containers and Container Clusters", *Google, Inc.*, Mountain View, CA, pp. 1-4

[8]Scalise, S., et al. (2007) "Satellite Channel Impairments." *Digital Satellite Communications.* Springer US, pp. 65-115.

[9]Tevanian, Avadis, et al. (1987) "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach." *USENIX Summer.*

[10]"Best practices for writing Dockerfiles", *https://docs.docker.com/articles/dockerfile_best–practices/*

[11]"Network Configuration", *https://docs.docker.com/articles/networking/*

[12]Zander, J. "New Windows Server containers and Azure support for Docker"
https://azure.microsoft.com/blog/2014/10/15/new–windows–server–containers–and–azure–support–for–docker/