SSC22-WKP2-14

Evaluating Network Performance of Containerized Test Framework for Distributed Space Systems

Walter Vaughan, Alan George
NSF Center for Space, High-Performance, and Resilient Computing
University of Pittsburgh
Pittsburgh, PA, USA
walter.vaughan@nsf-shrec.org

Brian Kempa, Daniel Cellucci, Nicholas Cramer
Intelligent Systems Division
NASA Ames Research Center
Moffet Field, CA, USA
brian.c.kempa@nasa.gov

ABSTRACT

Distributed space systems are a mission architecture consisting of multiple spacecraft as a cohesive system which provide multipoint sampling, increased mission coverage, or improved sample resolution, while reducing mission risk through redundancy. To fully realize the potential of these systems, eventually scaling to hundreds or thousands of spacecraft, distributed space systems need to be operated as a single entity, which will enable a variety of novel scientific space missions. The Distributed Spacecraft Autonomy (DSA) project is a software project which aims to mature the technology needed for those systems, namely autonomous decision-making and swarm networking. The DSA project leverages a containerized swarm test framework to simulate spacecraft software, which can identify emergent behavior early in development. Container virtualization allows distributed spacecraft systems to be simulated entirely in software on a single computer, avoiding the overhead associated with conventional approaches like hardware facsimiles and virtual machines. For this approach to be effective, the simulated system behavior must not be artificially influenced by the swarm test framework itself. To address this, we present a series of benchmarks to quantify virtual network bandwidth available on a single-host computer and contextualize this against the network and application behavior of the DSA swarm test framework.

INTRODUCTION

The proliferation and popularity of small satellites in use for scientific missions has driven an increased interest in distributed space missions (DSMs). In particular, more complex and dynamic DSMs will need to rely increasingly on autonomous commanding, communication, and cooperation in order to realize sophisticated mission goals.[1] The physical devices designed and built to actually perform these DSMs are referred to as distributed spacecraft systems (DSS) and are the focal technology for this paper.

Testing of space systems at the algorithmic, software, and physical levels of functionality is a critical element of the development process for virtually any space mission.[2] Historically, testing has relied on facsimile devices such as flatsats, or similar dedicated hardware, meant to match the system used in-flight as closely as possible.[3] Unlike the traditionally single-purpose nature of spacecraft software systems, newer spacecraft systems are more often composed of reused software components from previous missions.[4] Additionally, large-scale software deployment necessarily means reuse of code across many spacecraft at the same time. These effects combine to form a larger potential failure area, further justifying increased attention to the development of those software components. For distributed spacecraft design, this means that performing repeatable and detailed testing and simulations of space systems continuously throughout their development is inherently more important. This paper sets out to define common motivations and challenges behind DSMs and the development of their spacecraft software, describe an approach for testing a DSS early in development, and evaluate the fidelity and applicability of this testing system to both the use case it was originally designed for and other generic DSS.

BACKGROUND

This section covers the technology components which form the motivational building blocks for this research. The sections are organized in order of specificity, starting with the general problem space and ending in the domain of the specific focus of this research.

Distributed Space Systems

Broadly speaking, DSS comprise a family of system architectures wherein several spacecraft operate to achieve a singular goal. Many basic examples of DSS are currently active in space missions serving functions across multiple scientific domains. For example, the following categories of missions include launches spanning several decades into the past: communications networks, such the Iridium or SpaceX Starlink constellations; position and navigation, such as the GPS or GLONASS systems; earth sensing and imaging, such as Techsat-21,[5] Pléiades Neo,[6] or Tandem-X;[7] and ionospheric or heliophysics missions such as GRACE[8] or MMS.[9] Many more scientific DSS are under active development for heliophysics,[10–14] planetary missions,[15] and radio astronomy[16,17] use cases.

While some DSS already serve functions which may not be feasible or even possible using a monolithic spacecraft, the addition of certain features can fundamentally augment DSS abilities by facilitating different forms of autonomous operation. Specifically, if given sufficient compute capability and intersatellite network links (ISLs), a DSS can be designed to perform automatic workload balancing, respond to sudden opportunities or operational faults, improve ground-to-swarm network availability, intelligently share data between spacecraft, and avoid communication delays.[18] Given the scientific potential offered by these capabilities, this research focuses on high-computational-performance DSS with ISLs.

DSS with ISLs can be further categorized in terms of their mission goals, homogeneity, relative spatial proximity, collaborative abilities, and mission interdependencies.[19] Satellite constellations use spatially distant spacecraft in fixed orbits, but seldom include ISLs for collective system decision-making due to their long communication distance.[1] Federated satellite systems are heterogenous in spacecraft composition and do not operate towards common system goals, and fractionated satellite systems are explicitly heterogenous components of a singular system. Satellite clusters are formed by collections of homogenous spacecraft in fixed, nearby positions, operating on some common goal. Satellite trains are similar to clusters, but conventionally share the same orbital path, and do not necessarily function similarly or towards the same mission goal. Swarms are the most dynamic and flexible taxonomy of DSS, conceptually encompassing both constellations and clusters, incorporating dynamic spatial distances and optional spacecraft heterogeneity. Crucially, spacecraft swarms represent the most generic range of network topologies for an autonomous DSS and are therefore of prime interest for this research. A simplified summary of these taxonomies is given in Table 1.

Swarm Communication

Communication between spacecraft in a swarm configuration happens in dynamic topologies, with some connections between spacecraft behaving intermittently, whether intentionally part of a mission or incidentally due to flight conditions. These demanding network conditions require that flight software is capable of dynamically maintaining a useful understanding of its present network state, conceptually operating as a Mobile Ad-Hoc NETwork (MANET). Little flight heritage exists around MANET technologies in space, with only sparse simulation and theoretical exploration of its applications in a DSS.[20]

Even with a MANET in place, spacecraft swarms still require additional functionality for relaying information across the network in a way which ensures delivery to all connected systems, regardless of topology, ideally with some guarantee of quality-of-service (QoS). In terrestrial computing, one approach to solving this communication problem is through the use of a Data Distribution Service (DDS) standards-compliant networking middleware. DDS is a platform- and language-agnostic specification that implements a publish-subscribe model for communication in a dependable manner.[21] This technology lends itself well to the goals of autonomous DSS and is featured in the mission use case described below.

Distributed Spacecraft Autonomy

The motivating mission context for this paper is the Distributed Spacecraft Autonomy project (DSA) at NASA Ames Research Center, which seeks to develop and mature technologies and methods for autonomous coordination, adaptive reconfiguration, planning, and swarm commanding.[18] This mission is structured to specifically advance capabilities in the areas of swarm scale, complexity, and human-

Table 1: Feature Comparison of DSS Simplified[19]

DSS Architecture	Mission Goals	Cooperation	Homogeneity	Inter-Satellite Distance	Autonomous/Co-dependent
Constellations	Shared	Required	Homogeneous	Regional	Autonomous
Trains	Independent	Both	Heterogeneous	Local	Autonomous
Clusters	Shared	Required	Homogeneous	Local	Both
Swarms	Shared	Required	Both	Both	Both
Fractionated Satellites	Shared	Both	Heterogeneous	Local	Both
Federated Satellites	Independent	Both	Heterogeneous	Both	Autonomous

swarm interaction. Those capabilities are visualized in Figure 1.

The project is divided into two phases of technology demonstration: initially, a software payload on a flight mission consisting of a swarm of 4 spacecraft operated as part of the Starling technical demonstration;[20] and later, a simulation mission involving a much larger swarm of 100 facsimile spacecraft in a hardware-in-the-loop (HiL) configuration. The flight software used in each spacecraft is based on NASA's Core Flight Software. Each spacecraft is equipped with a uniform set of mission applications designed to handle incoming sensor information, interface with the publish-subscribe inter-satellite network interface, and autonomously compute an execution plan for how to use sensor data in the next cycle of data collection based only on data received from other spacecraft in the swarm.[24]

Core Flight System

The DSA project uses NASA's Core Flight System (cFS), previously known as core Flight Software under the same acronym. cFS is an open-source, reusable software framework for space missions that is distributed by NASA and used broadly in the space community for various missions.[25] cFS is a reusable framework derived from the codebase of historical NASA missions and maintained agency-wide as a set of centrally managed open-source components and interfaces, available for reuse and extension across the broader space science domain. cFS includes a set of commonly needed applications; storage, command, & data-handling utilities; the core Flight Executive (cFE); and an explicit application programming interface (API) between each programming layer and component.[26] The layout of cFS is shown in Figure 2 for reference. By defining a standard, layered API for components of cFS, scientists can develop new applications, platform support packages (PSPs), and operating system abstraction layers (OSALs), while maintaining functional compatibility within the rest of the cFS ecosystem. Since the DSA project is concerned with software reuse and improvement in future missions, cFS is well suited as a base framework.

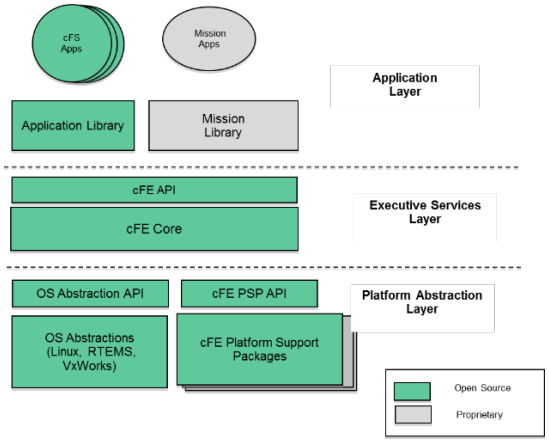

Figure 2: General Architecture of cFS Mission Software[26]

DSA Flight Software Design

The Starling concept of operation calls for four CubeSats to be operating in relatively close proximity (< 100km), allowing them to always be within crosslink radio range of each other. This persistent but potentially lossy link was a core design consideration for the DSA software architecture, which can be broken down into three primary design components:

- Intelligent Sensor — components that take raw sensor input and translate them to reward values
- Autonomous Planner — takes the reward states from itself and other spacecraft and plans the next observations to process
- Communication Manager — manages communication from the spacecraft to other assets, including other spacecraft and the ground

Figure 3 shows the applications that make up these components and the specific hardware devices that they interact with. The intelligent sensor component

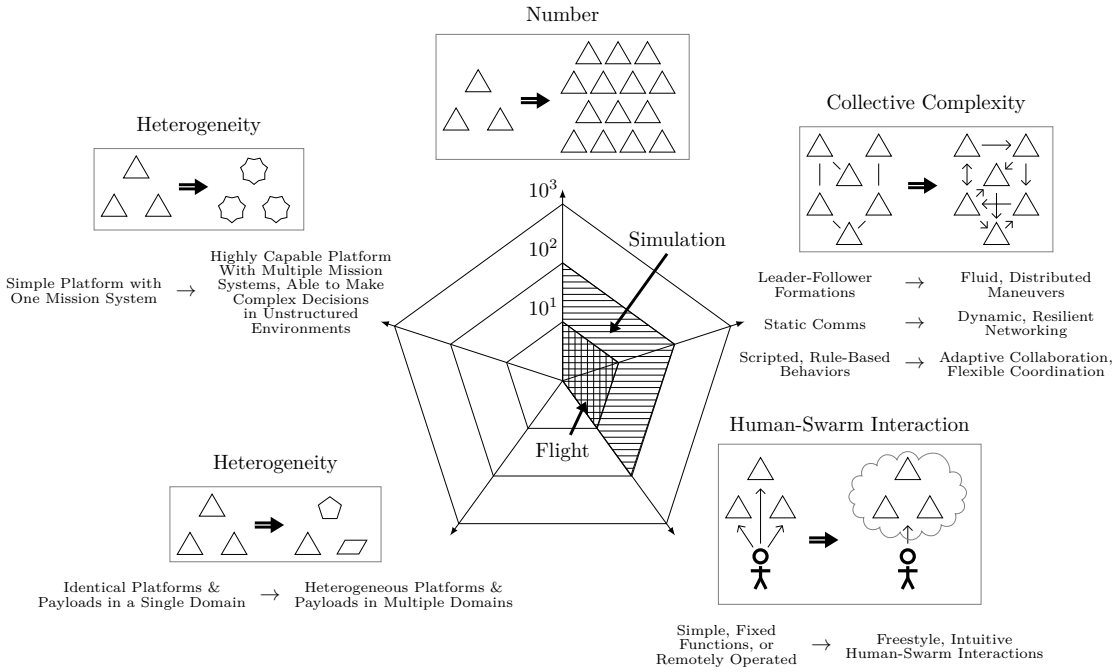

Figure 1: Challenges in Swarm Development[22,23]

uses the GPS application to unpack compressed Novatel messages. At the same time, the Total Electron Count (TEC) application takes the L1 and L2 frequencies and uses them to calculate the relative total electron count. This count is then translated into a decision reward, which is used by the Autonomous Planner component. The Autonomous Planner component comprises the cFS scheduler application and the AUTO application. The scheduler provides a trigger for plans to be generated. The AUTO application takes in the rewards from across the DSS to create an observation plan shared with the TEC application for execution.

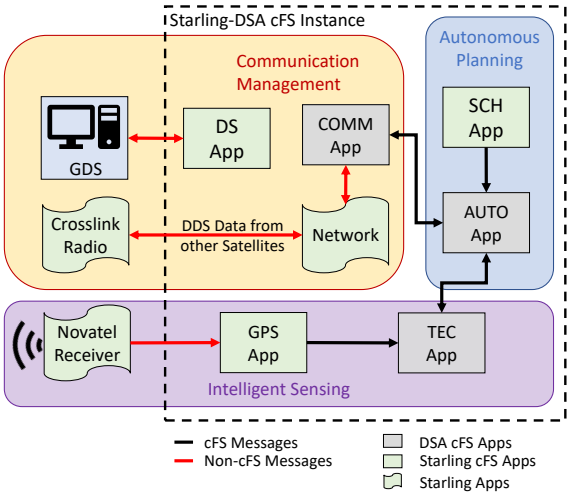

Figure 3: DSA Flight Application Architecture

The communication manager comprises the crosslink radios, virtual network interface, COMM application, and cFS Data Store application. The COMM application is built on top of DDS and acts as a translator between the cFS middleware and the DDS middleware. The operational network stack is displayed in Figure 4.

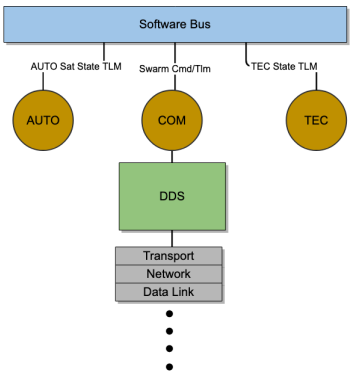

Figure 4: DSA Network Stack Diagram

The DSA software is designed to be "best-effort." The AUTO application will generate a plan with its information, making the communication infrastructure a pivotal component to maintain consistency and maximize performance. This is why the testing of these interactions is so critical to the project's success.

Testing & Development

Flight software is tested at as many different levels of abstraction as feasible, starting from unit testing of individual functions of code, through integration tests of larger components, up to complete testing of the finished system through different scenarios. Typically, as a project progresses, larger and more complex components of a system are developed, and, upon testing, these components are sometimes found to be unsound or otherwise require design revisions. By testing the interaction and integration of these complex components at earlier points in development, those necessary changes can occur while reducing impact to a project's schedule or budget.[27] An increased investment in early testing is one of the central requirements for building complex flight software at scale.[28]

Swarm Test Framework

Early into development of the first phase of the DSA mission, a software testing framework based around the concept of containerization was designed and implemented, consisting of four containers intending to simulate each spacecraft of the flight mission. Each container is populated by the cFS applications and cFE compiled for the native architecture of the developer's machine, as well as all other runtime dependencies for the flight software to function. The ability to compile flight software for native execution in the host architecture is a feature already built into the cFS framework.[26]

The swarm test framework is used to perform functional tests of the DSA applications at different levels of abstraction and functionality. Tests range from basic aliveness tests to more complicated measures of behavior and performance requirements. The more complicated tests are referred to in DSA as *scenario tests*. Scenario tests are tied to project software requirements and typically test applications close to real operating conditions.

While the research presented in this paper is primarily coupled with the development efforts of the DSA project, similar work is also being done elsewhere on developing test frameworks for simulating DSS behavior through containers.[29] This similar work focuses on common elements of container-based abstractions for spacecraft systems but also leverages technology popular in cloud computing to perform software-defined networking, metrics collection, and network disruption and failure testing. Their application under test is designed around a fixed constellation network topology but still shares some design elements of the DSA flight experiment.

The specific communication mechanisms used in the DSA project are formed from a multi-layered networking stack. At the lowest level of abstraction, a MANET is operating using the Better Approach To Mobile Ad-Hoc Networking (B.A.T.M.A.N.) protocol. This tooling operates the ISL network interface for each spacecraft at the ISO/OSI network layer 2, which avoids a dependence on IP addresses.[30] Above the MANET, a publish-subscribe DDS middleware transparently handles the distribution of data at scale, abstracting away the process of routing traffic to other spacecraft. Specifically, the RTI Connext Micro DDS software package is used in the DSA flight software. This package is specially designed for resource-constrained environments.[18,31]

APPROACH

This section describes the approach used to test the abilities and limits of containers as representative simulated spacecraft. Both the DSA project and generalized, scalable autonomous DSS are considered as potential contexts for containerized testing. We showcase an initial set of experiments designed to quantify the expected performance limits of a generic simulated DSS being developed in the context of a communication-intensive distributed spacecraft mission.

Although processing capability is an inherent limiting factor of the performance of scalable systems, the computational requirements of a scalable system tend to be highly mission-dependent. Accordingly, this experimental approach ignores the effects of computational load and focuses primarily on a particular component of autonomous DSS: the ISLs. Specifically, this paper evaluates the network performance of multiple containers on a single host system, a context which is realized in the DSA project software test framework.

Container Virtualization

Virtualization technologies can be described by a variety of mechanisms, but the central premise is that certain portions of a computing system's resources are isolated and managed by an intermediary tool in order to enhance system security, limit the execution environment and available system resources, provide a consistent and artificial interface for system software, or some combination of each.[32] While container virtualization shares some similarity to conventional virtualization (VMs), a fundamental difference is that containers abstract and isolate the operating system from the containerized applications while still allowing access to the host operating system's functions and computing resources. The architectural differences between containers and VMs are shown in Figure 5.

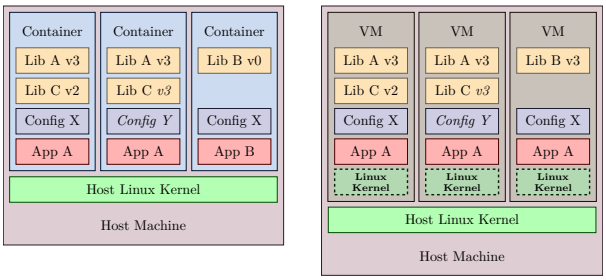

Figure 5: Architectural Differences of VMs and Containers

On Linux systems, container virtualization is implemented through mechanisms built into the kernel, such as `cgroups` and namespaces. These mechanisms can give processes their own hierarchical visibility or control of devices, memory, CPU, and network interfaces.[33] Multiple processes can run inside the same container, and just like a normal kernel process tree, exactly one process is at the "root" of a container's process tree. From the perspective of the kernel, processes inside a container are running alongside normal processes on a system and make the same system calls on the same kernel, but the kernel exposes different filesystems, process trees, network interfaces, and other kernel resources to processes inside a container.

This approach to virtualization avoids the need for a host system to spend time translating a system call from a virtual system or to maintain a virtual state and set of emulated interfaces. In effect, then, starting a container is no different than simply starting any other process on a machine.[33] This means that, compared to traditional VMs, containers offer superior image-generation speed and startup time, while also exhibiting less processing and memory overhead.[32] This presents an opportunity for improvement upon the status quo, as historically, the standard approach for flight software testing has been to use machine virtualization (VMs).[34]

Container Runtime & Image Building

With many different container virtualization technologies being actively developed, there are ample combinations of container builders and container runtimes which support all the necessary features for isolating spacecraft software from a host operating system. Since container virtualization broadly works through simple tooling on top of operating system features, different container runtimes generally have negligible differences in runtime performance.[35] Therefore, without a compelling reason to compare the performance of the runtimes themselves, the remaining considerations in choice of a container runtime center around features and compatibility.

The most widely used container runtime at the time of writing, Docker, conforms to a standard specification called the Open Container Initiative (OCI) specification.[35,36] This allows compliant tools to benefit from being inter-operable with other tooling in the container ecosystem.[37] By using OCI-compliant container tooling, DSA containers and containerization tools can be more easily integrated or adapted into other projects and experiments.

Another major component to containerization is the process of building the filesystem image and execution configuration, which comprise what is known as a container image. In considering the image-building process, lower build times result in faster iterative testing during development cycles and could therefore have a positive effect on development productivity. Different tooling exists for the image-generation process, in addition to the default tool in Docker, `docker build`. When comparing each of

these tools, build times for small images vary by significant margins, particularly if intermediary images are produced.[38] One of the fastest OCI specification-compliant image builders is BuildKit, which is now incorporated directly into the Docker software package.[39]

Given these considerations, as well as the rich tooling and features of Docker software and its intrinsic compatibility with the OCI runtime and image specifications, the DSA project uses the freely available Docker suite for container building and running. Containers are built starting from an official Docker image based on the same Linux distribution used for compiling the flight software. This ensures application binary interface (ABI) compatibility with system libraries used to compile and run cFS software (i.e. libc/libc++).

Container Networking & Orchestration

Many software packages exist to support the scalable systems envisioned by the original designers of container virtualization.[35] These tools perform what is referred to as *orchestration*, which is the process of configuring and managing a collection of containers through a unified system, rather than interacting with each container individually. For this research, the complexity of orchestration required is relatively simple. Since orchestration tools are not in any critical performance path, the DSA project uses *Docker Compose*, as it is already tightly bundled with the Docker suite.[40]

Docker also provides a simplified interface for creating the virtual networks used by their container runtime. The resulting virtual networks can support different models of operation, giving container interfaces transparent access to a host's network interfaces, containers on other Docker runtimes, including on different systems, or simply other designated networks on a single host.[41] Containers can be added to or removed from networks while running at any time, offering a way to change network topology on-the-fly. However, this offers only rudimentary network control, falling short of the more complicated disruptions characteristic of a DSS swarm network.

The network configuration used in this research is the *bridge* mode which forms a virtual network of every container attached to it, allowing the host to send and receive traffic through that network interface but providing no explicit routes outside the network to the containers. For the DSA test framework, Docker Compose is used to specify the container runtime configuration required for the full flight software, including network device settings for the MANET and DDS networking stack. Each simulated spacecraft is assigned a unique network ID and placed on a network bridged to the host, through which commands and telemetry can pass.

Traffic Control

Another important feature of the swarm test framework is the use of traffic shaping through the netem ("Network Emulator") kernel component. The Linux kernel comes with a variety of quality-of-service and network traffic shaping mechanisms by default. Netem specifically provides emulation of packet loss, delay, corruption, and other network failures. This mechanism was introduced originally as part of the NetEm tools.[42] Netem allows individual Linux network interfaces to behave more similarly to imperfect, real-world interfaces such as those seen in extreme network environments like autonomous DSS swarms. DSA uses netem to artificially induce network failures and change the apparent topology of the network of the containers at any point during simulation.

Network Profiling

The DSA swarm test framework is capable of monitoring container network performance metrics while tests are running. To accomplish this, network metrics are logged as reported directly to the kernel by each network interface. These metrics are recorded by the network drivers themselves. This kernel network data is collected for all active container network interfaces and logged to a file associated with each run of a scenario test in the swarm test framework.[43]

This tool is incorporated into the swarm test framework scenario testing infrastructure such that, when any scenario test is run, network performance is automatically saved alongside other telemetry and test artifacts. The sampling frequency for this data is configurable, but a high-resolution rate of 20 Hz was chosen due its negligible CPU usage impact. This mechanism allows developers to inspect burst traffic speeds, evaluate expected application bandwidth, and observe traffic patterns across containers. An example of the network data collected is shown in Figure 6. Here, the bitrate of traffic for each sample period is shown for each spacecraft, in both transmitted (TX) and received (RX) traffic.

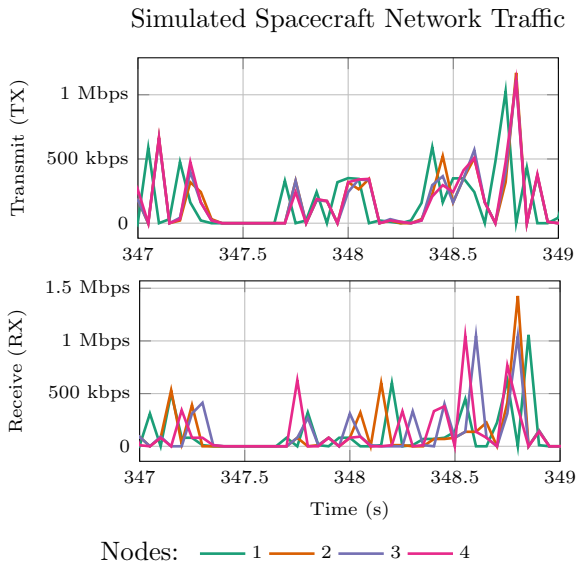

Figure 6: Container Network Traffic During a Simulated Scenario Involving Four Fully Connected Nodes

Network Benchmarking

To understand the theoretical bandwidth limits of the swarm test framework, a series of benchmarks was designed to examine the behavior of the network under the heaviest possible load for a swarm of a given size. Theoretically, the maximum load occurs when all spacecraft are connected to each other and are simultaneously attempting to transmit to every other spacecraft, forming a fully connected bidirectional network. These benchmarks were designed to measure per-connection bandwidth, evaluating how much traffic each connection is capable of theoretically supporting under a worst-case network load.

To measure these bandwidth limits, a lightweight network speed-testing tool was chosen to both generate artificial network traffic and measure it. *iPerf3* is a commonly-used tool for measuring unidirectional throughput and other properties of a single client-server network connection. It also supports both TCP and UDP traffic, as well as bandwidth targets and other, more advanced options.[44] When running in TCP mode, iPerf automatically tries to run at maximum network speed. However, to test UDP speed, the desired bandwidth must be specified, and traffic which cannot be sent or delivered across the virtual interface is dropped, which is reported as packet loss. Therefore, to measure the bandwidth of UDP traffic, we have to instead consider packet loss at a variety of speeds and observe where loss occurs.

Each container image for the benchmarks was built starting from an official Ubuntu Linux 18.04 Docker image, the operating system used in the DSA test framework. The container image was then augmented with the iperf3 Ubuntu package and an custom script used for orchestration of each container's iperf3 processes. The script acts as an entry point to the container and starts a parameterized number of server instances as child processes at startup.

For each benchmark, the parameters of network size, traffic type, and–for UDP tests–target bandwidth are passed into a test script. This script starts the specified number of containers n. Then inside each container, the scripts runs $n-1$ instances of the iPerf3 process in server mode, followed by $n-1$ instances of the iPerf3 process in client mode, configured such that each client connects to a server on every other container. This configuration is visualized for $n = 3$ containers in Figure 7. Connection tests are run for the default period of 10 seconds. These steps happen in immediate succession, with the intention that tests start close enough in proximity and run over a long enough duration that they can be considered concurrent.

In the context of the DSA project, the most interesting specific swarm size is $n = 4$, which reflects the flight experiment hardware configuration. However, the second-phase experiment of the DSA project includes up to $n = 100$ facsimile spacecraft on the same network. While it would be ideal to simulate simultaneous 100-spacecraft communication on a single host, the actual communication patterns used by the DSA network stack can use multicast communication and use broadcast patterns rather than fully connected patterns. In an ideal multicast network of size n, the effective network traffic needed to communicate one packet from each craft to every other craft is simply n. The fully connected network size with an equivalent amount of traffic would be $\approx \sqrt{n}$. Assuming that the simulation network patterns are imperfect but still somewhat optimized, being able to handle $10 < n \leq 20$ reasonably captures the expected network loads for the DSA experiments.

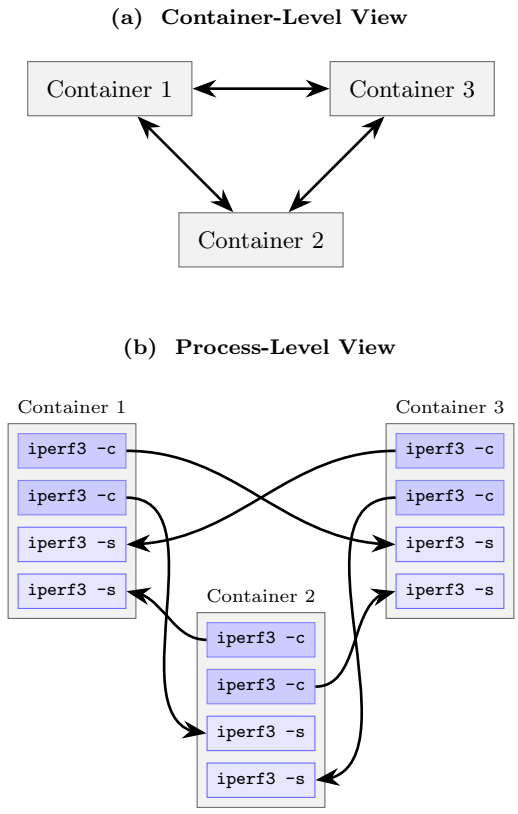

Figure 7: Container Configuration for Network Benchmarks, Swarm Size of $n = 3$ Shown

RESULTS

The results are broken into sections demonstrating the expected behavior of the test framework and limits and the results of using the test framework to assess the DSA software.

Test Framework Analysis

The testing framework needed to have its expected performance limits quantified and verified to confirm that the results of the software testing were from the software performance, not the underlying testing framework. To do this, the network testing was performed as described in the earlier sections and are reported here.

For the iPerf benchmarks, the test suites were run inside an Ubuntu Linux 18.04 LTS virtual machine with 16 dedicated Intel Xeon vCPU cores running at 2.2GHz and 80 GiB of physical memory. While the hardware configuration used was chosen out of convenience, this paper assumes that the performance of this machine is representative of the expected performance from a development server available to organizations with the resources to build DSMs similar to those described in this paper.

TCP Bandwidth

For the first set of benchmarks, using TCP traffic with no restriction on bandwidth, the total throughput for each iPerf3 server process was recorded as a single connection's bandwidth. Note that this includes throughput in only one direction; data sent in the opposite direction is considered a distinct connection. From these data points, average bandwidth for each connection was computed, and the lowest effective bandwidth observed for a single connection was also recorded for each benchmark. These tests were performed in increasing scale for $2 \leq n \leq 25$ containers.

As shown in Figure 8, the test framework sustained a minimum throughput above 200 Mbps for each connection in the system for all tested swarm sizes, which supports the notion that this test framework can support a high ceiling of network performance and complexity. At a swarm size of $n = 4$, matching the DSA flight configuration, the system sustained a minimum of 12.9 Gbps on each connection, which is far beyond the capabilities of the flight hardware network devices, which are rated for only 50 Mbps of traffic. Based on this observation, the test framework has the potential to support the traffic capacity needed for effective simulation fidelity of the DSA flight mission.

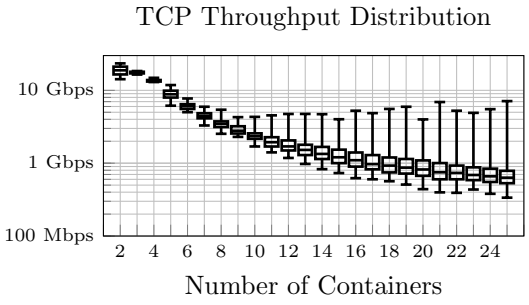

Figure 8: TCP Throughput of Unidirectional Connections

In the total throughput of all connections (total system bandwidth) in Figure 9, there is a sharp rise in system bandwidth between $2 \leq n \leq 4$ followed by a slow increase in bandwidth for swarm sizes $n > 4$. This result somewhat matches expectations; given that the experiment hardware has a fixed number of

CPU cores, it was expected that the network performance would improve as more CPU cores were used, saturating once the number of processes communicating exceeds the number of available CPU cores. However, once the available cores are saturated, total system throughput should stay roughly the same or decrease, yet this bottleneck was not observed. System bandwidth appeared to steadily increase by over a factor of two from the approximate saturation point of $n = 4$ to the maximum swarm of $n = 25$. This unexpected result suggests a weakness in this experiment design, which is discussed in more detail later.

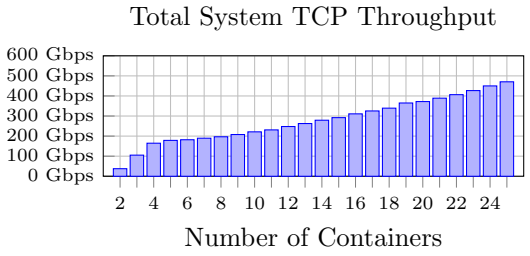

Figure 9: Total System TCP Throughput vs. Scale

UDP Packet Loss

In the second set of experiments, using UDP traffic at specified bit rates, the total packet loss from each iPerf3 server process was recorded as a single connection's loss percentage. As in the TCP experiments, each recorded loss value pertains to only one direction of the duplex communication paths between each container. Unlike the TCP experiments, every network scale size from 2 containers up through 32 containers was tested, and for each network size, 10 different target bit rates were selected heuristically across a range of feasible network interface rates. The selected bit rates span from 10 kbps to 10 Gbps.

As shown in Table 2, the test framework was capable of sustained UDP traffic of up 1 Mbps without any observed packet loss for all swarm sizes tested ($n \leq 32$). Sustained traffic of up to 10 Mbps was seen with no observed packet loss for swarm sizes of $n \leq 15$, but only a minimum swarm size of $n = 2$ could support up to 100 Mbps of sustained traffic with no packet loss. As expected, the test framework could not sustain the same network throughput with UDP traffic as it could with TCP traffic. Even when operating at lower target bit rates, the system dropped substantial proportions of network traffic.

Table 2: Largest Network with No Packet Loss per UDP Bit Rate

UDP Bit Rate	Largest Network with No Packet Loss
10 kbps	(≥ 32)
100 kbps	(≥ 32)
500 kbps	(≥ 32)
1 Mbps	(≥ 32)
5 Mbps	31
10 Mbps	15
50 Mbps	2
100 Mbps	2
1 Gbps	(n/a)
10 Gbps	(n/a)

Although results of the TCP experiments suggest that the virtual network interfaces could support the quantity of traffic for at least 200 Mbps of traffic for all swarm sizes, this test demonstrated total packet loss for most connections at swarm sizes of $n > 19$ at 100 Mbps. This should not be surprising, however, due to the lack of synchronization mechanisms in UDP traffic. Network interfaces communicating over UDP will attempt to send packets even when a receiving interface might be forced to drop the traffic, unlike TCP, which can use congestion control schemes to negotiate rate with the sender.[42]

Table 3: Maximum Packet Loss (%) vs. Network Speed (bps) and Size

size	10k	100k	500k	1m	5m	10m	50m	100m	1g	10g
2	0	0	0	0	0	0	0	0	0.02	0.1
3	0	0	0	0	0	0	7.17	0.34	0.42	0.83
4	0	0	0	0	0	0	19.62	16.23	7.06	8.62
5	0	0	0	0	0	0	31.41	15.82	21.31	52.56
6	0	0	0	0	0	0	30.48	19.76	38.04	100
7	0	0	0	0	0	0	24.44	37.79	100	100
8	0	0	0	0	0	0	43.86	32.63	100	100
9	0	0	0	0	0	0	43.65	28.47	100	100
10	0	0	0	0	0	0	32.34	20.58	100	100
11	0	0	0	0	0	0	30.4	22.15	100	100
12	0	0	0	0	0	0	25.11	32.16	99.39	100
13	0	0	0	0	0	0	30.62	29.85	100	100
14	0	0	0	0	0	0.07	31.09	100	100	100
15	0	0	0	0	0	0	31.67	100	100	100
16	0	0	0	0	0	0.07	31.95	100	100	100
17	0	0	0	0	0	0.07	62.44	100	100	100
18	0	0	0	0	0	0.07	39.6	100	100	100
19	0	0	0	0	0	0.07	58.85	100	100	100
20	0	0	0	0	0.13	0.13	100	100	100	100
21	0	0	0	0	0	0.13	100	100	100	100
22	0	0	0	0	0	0.13	100	100	100	100
23	0	0	0	0	0	0.86	100	100	100	100
24	0	0	0	0	0	0.2	100	100	100	100
25	0	0	0	0	0.4	0.79	100	100	100	100
26	0	0	0	0	0	0.86	100	100	100	100
27	0	0	0	0	0	0.79	100	100	100	100
28	0	0	0	0	0.26	0.66	100	100	100	100
29	0	0	0	0	0	0.46	100	100	100	100
30	0	0	0	0.66	0	0.33	100	100	100	100
31	0	0	0	0.66	0	0.73	100	100	100	100
32	0	0	0	0	0.79	1.06	100	100	100	100

Scenario Test Comparison

Fortunately, as shown in Table 3, all UDP traffic was still delivered with maximum observed loss rates below 20% for a swarm size of $n = 4$. The DDS protocol implemented by the spacecraft networking stack for DSA supports traffic delivery reliability through retry mechanisms, and therefore some amount of packet loss may be acceptable in the swarm test network, even if the loss is not intentionally induced. Indeed, when looking at the detailed scenario test network data, average bit rates for each device stay well below 1 Mbps. Swarm communication, as visualized in Figure 6, tends to generate traffic only in bursts rather than sustained data transfers like those used in the benchmarks.

Furthermore, for the $n = 4$ swarm, the measured network speed of an interface actually corresponds to the sum of all outgoing connections from a node, so a per-connection speed s corresponds to a $s \cdot (n - 1)$ limit on interface speed. For $n = 4$, a hypothetical 10 Mbps per-connection limit would correspond to an interface limit of $3 \cdot 10 = 30$ Mbps. These observations lend credibility to the notion that the swarm test framework is capable of effectively handling the amount of traffic generated.

Application Performance With Packet Loss

As mentioned above, the DSA applications are designed to function in a non-trivial network topology with restricted bandwidth and lossy connections. To analyze the impact of network disturbances on application effectiveness, we performed a scenario test on a lossy network of $n = 4$ spacecraft. We measured the proportion of simulated satellites which successfully received the shared satellite planing states throughout a scenario while introducing a uniform amount of packet loss between each satellite. This can be thought of as a loss of plan consistency between members of the DSS. The resulting performance of degradation of swarm collaboration is visualized in Figure 10. As expected, the system is robust to small amounts of packet loss, with performance quickly degrading after roughly 15% packet loss.

These results illustrate the expected performance ranges on-orbit and help identify risk reduction plans. Software developers can use this information to determine if the current software design and configuration will be sufficient under various conditions. Furthermore, it allows for additional analysis to see if the provided control settings can cover the expected operational performance range.

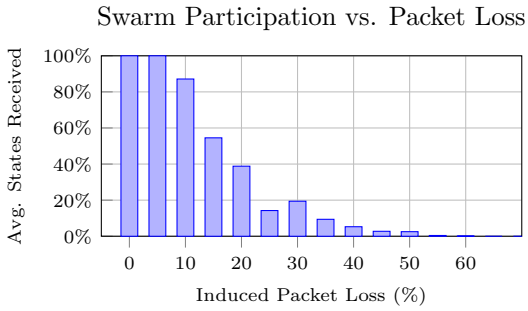

Figure 10: Degredation of Neighbor Participation from Simulated Packet Loss

DISCUSSION

The results obtained from these tests add valuable insight to the behavior and limitations of the swarm test framework as well as guidance for other containerized DSS development. While several issues emerged from the test design, these issues also provide insight to a category of simulation failures which may affect testing of other DSS software. By evaluating the behavior of the virtual network interfaces intrinsic to containerized DSS testing, this research is able to describe expected performance limits for testing DSS in general, in addition to the system designed in the DSA project.

When performing tests using TCP traffic, the kernel is afforded control over how much data to send at a time and can therefore shape traffic sent between different processes to manage congestion and achieve higher throughput.[42] This optimization is not present for UDP traffic, leading to scenarios where processes send large packets much faster than they can be processed. When looking only at TCP traffic, each connection was shown to sustain well over 100 Mbps of traffic at every swarm size tested, but this throughput was only observed with low loss ($< 20\%$) for networks sizes of $n \leq 13$ using UDP traffic.

This finding highlights an important consideration for simulation of containerized DSS. In contrast to TCP traffic, the performance of equivalently large swarm networks ($n \geq 20$) transmitting UDP traffic at speeds ≥ 50 Mbps showed packet losses so large that most packets were either dropped or never transmitted at all. The underlying network mechanisms of a spacecraft in a DSS as implemented at the protocol level can therefore have a significant effect on the ability for the host system to deliver the traffic in time. If an application designed for a DSS, such as

an event-driven call-and-response across the swarm, could produce a large (e.g. 100 Mbps) burst of traffic across a significant portion of the swarm and the traffic was not being delivered with any mechanisms of traffic congestion in place, then the simulation environment could introduce dropped packets.

As shown with small swarm sizes ($n \leq 4$), if the number of active connections is fewer than the number of logical CPU cores on the test machine, then the containerized DSS simulation may still be able to handle those large traffic bursts by matching the transmission speed of data. On the other hand, if the traffic produced by DSS applications is effectively limited at the application level, burst traffic across the entire swarm can be sustained effectively even when the swarm size exceeds the number of available CPU cores. The exact degree to which burst traffic could be handled is not addressed in this paper, but it is worthy of consideration for further research.

Memory & Network Interface Limitations

Through the course of this research, several key insights stemming from the design of the experiment brought to light other considerations for containerized testing of DSS. These issues potentially impacted the scalability of the tests, particularly at higher network throughput, and reflect limitations which may not be seen in some real DSS. Each issue is discussed in detail below.

First, each test connection required two iPerf3 processes: one for the server and one for the client. For a fully connected network, the number of processes required to run an experiment increases quadratically as a function of network size, an effect which introduces significant computational and memory overhead. While an individual spacecraft designed as part of a DSS may operate multiple communication interfaces or channels simultaneously, this experiment explicitly models network communication through a multiprocessing lens. In contrast, missions involving lightweight flight software might only generate network traffic on one network interface at a time, meaning a much larger swarm size could be simulated than was demonstrated here.

The second issue encountered in this experiment arose as a consequence of modeling the worst-case, fully connected network topology. Each iPerf3 test was started sequentially, so an interval of time was present between the first test and the last test starting. For low numbers of connections, this interval was negligible relative to the duration of the connection tests. But for larger swarms, the interval grew significantly. As a consequence, some connection tests were necessarily running while others were waiting to be initiated for at least some portion of the intended window of simultaneous test, which in turn meant that the duration of the tests was longer, and therefore the true bandwidth sustained by the test system was lower than indicated in Figures 8 and 9.

Swarm Test Framework Performance

When contextualized against the results of the iPerf3 benchmarks, the network load of the DSA experiment in the swarm test framework is well within the theoretical performance limits during simulated tests. Even burst speeds observed on individual interfaces were orders of magnitude below the maximum sustained speeds obtained in the benchmarks. While the network monitoring feature of the swarm test framework was useful for contextualizing the benchmarks, it also serves a practical function during regular development by allowing developers to gauge what kind of network strain changes might impose. Observing network traffic patterns can also give insight to the otherwise opaque layers of the network stack, which in the case of the DSA network stack (shown in Figure 4) include DDS on top of B.A.T.M.A.N. mesh networking.

CONCLUSION

Industry trends in spacecraft mission design are driving an increased focus on the development of autonomous, distributed spacecraft systems. These systems present a way to perform scientific and technical missions which are otherwise infeasible with monolithic spacecraft architecture. With both DSS design and the categories of missions enabled by DSS comes inherent flight software complexity. Early testing of these systems is critical because complex flight software requires a proportionally higher investment in software testing architecture to avoid delays and excess development later on.

While it is straightforward to simulate a single spacecraft's flight applications directly, DSS application testing requires simulating multiple spacecraft instances interacting over a network. Recent efforts in the space industry have made headway on addressing this gap in early testing by using virtualization technologies as part of the simulation environment for flight software. We investigate a promising approach to this challenge: container virtualization, which offers performant, lightweight abstraction of an application from its operating system. However,

very little literature currently exists on the limitations and considerations relevant to using containers as a platform for simulating DSS.

To better understand containerized DSS testing, this research presents an experimental evaluation of containerized DSS performance by measuring the bandwidth available in a containerized environment. The results of this research showed that a containerized DSS test framework can simulate the network behavior of DSS with up to 32 containerized spacecraft exchanging 1 Mbps of sustained UDP traffic across the entire swarm without introducing artificial packet loss. Additionally, the experiments demonstrated the effects of traffic congestion control mechanisms on network performance and highlight other important considerations for containerized DSS test frameworks. Finally, we were able to observe the detailed network speeds of our containerized DSS test framework and verify they did not approach or the exceed the limits determined by the benchmarks. We also demonstrated a basic measure of distributed application robustness in the presence of packet loss.

FUTURE WORK

There are several promising future directions for this work given the results of this research. Currently, the swarm test framework primarily tests the swarm configuration matching the starling experiment of 4 spacecraft, but it would be worth seeing how actual swarm network speed and behavior changes as spacecraft are introduced. These tests could also be done with more carefully-designed network disruptions and spacecraft topologies.

At a more general level, the behavior of DDS across virtual networks in particular is also worth specific consideration. While literature exists evaluating the relative performance of different implementations,[21] understanding performance under the context of publish-subscribe data delivery mechanisms would yield a more precise point of comparison for DSS which use those DDS for swarm communication. DDS middleware offers a powerful network abstraction across distributed systems, which will likely be of importance in other many other DSS missions.

Finally, to further evaluate other categories of DSS, these experiments could be modified to explicitly analyze broadcast communications, where data sent from a spacecraft is delivered directly to the entire swarm rather than repeated across each connection.

Acknowledgments

This research was funded by industry and government members of the National Science Foundation (NSF) Center for Space, High Performance, and Resilient Computing (SHREC), under Grant No. CNS-1738783, and by NASA's Game Changing Development Program.

REFERENCES

[1] Carles Araguz, Elisenda Bou-Balust, and Eduard Alarcón. Applying autonomy to distributed satellite systems: Trends, challenges, and future prospects. *Systems Engineering*, 21(5):401–416, 2018.

[2] L. F. Nozhenkova, O. S. Isaeva, and Rodion V. Vogorovskiy. Scenario approach to testing spacecraft's onboard equipment command and software management. *DEStech Transactions on Engineering and Technology Research*, 2017.

[3] Michael Wright. Lunar reconnaissance orbiter flatsat. In *13th European Test Symposium IEEE Computer Society and Test Technology Technical Council*, 2008.

[4] M. Annette Mirantes, Maria Spezio, and Kristin A. Wortman. Confidence in spacecraft software: Continuous process improvement in requirements verification. In *2014 IEEE Aerospace Conference*, pages 1–11, 2014.

[5] Marco D'Errico and Et Alii. *Distributed Space Missions for Earth System Monitoring*. Springer Science and Business Media, 01 2013.

[6] Soubirane Jérôme. Shaping the future of earth observation with pléiades neo. In *2019 9th International Conference on Recent Advances in Space Technologies (RAST)*, pages 399–401. IEEE, 2019.

[7] Jean-Sébastien Ardaens, Ralph Kahle, and Daniel Schulze. In-flight performance validation of the tandem-x autonomous formation flying system. *International Journal of Space Science and Engineering 5*, 2(2):157–170, 2014.

[8] BD Tapley, C Reigber, and JC Ries. The grace mission: status and early results. In *AAS/Division of Dynamical Astronomy Meeting# 34*, volume 34, pages 07–01, 2003.

[9] JL Burch, TE Moore, RB Torbert, and BLhttps Giles. Magnetospheric multiscale overview

and science objectives. *Space Science Reviews*, 199(1):5–21, 2016.

[10] Scott E Palo, Marcin Pilinski, Jeffrey P Thayer, Whitney Quinne Lohmeyer, Kristina M Lemmer, Simone D'Amico, E Glenn Lightsey, and Saeed Latif. The space weather atmospheric reconfigurable multiscale experiment (swarm-ex): A new nsf supported cubesat project. In *AGU Fall Meeting Abstracts*, volume 2020, pages SA023–11, 2020.

[11] Laura Plice, Andres Dono Perez, and Stephen West. Helioswarm: Swarm mission design in high altitude orbit for heliophysics. In *AAS/AIAA Astrodynamics Specialist Conference*, number AAS 19-831, 2019.

[12] Justin Kasper, Joseph Lazio, Andrew Romero-Wolf, James Lux, and Tim Neilsen. The sun radio interferometer space experiment (sunrise) mission concept. In *2020 IEEE Aerospace Conference*, pages 1–12. IEEE, 2020.

[13] Adam Koenig, Simone D'Amico, and E Glenn Lightsey. Formation flying orbit and control concept for the visors mission. In *AIAA Scitech 2021 Forum*, page 0423, 2021.

[14] Karl Magnus Laundal, Jeng-Hwa Yee, Viacheslav G Merkin, Jesper W Gjerloev, Heikki Vanhamäki, Jone Peter Reistad, Michael Madelaire, Kareem Sorathia, and Patrick Joseph Espy. Electrojet estimates from mesospheric magnetic field measurements. *Journal of Geophysical Research: Space Physics*, 126(5):e2020JA028644, 2021.

[15] RJ Lillis, SM Curry, CT Russell, D Curtis, E Taylor, J Parker, JG Luhmann, A Barjatya, D Larson, R Livi, et al. Escapade: Coordinated multipoint observations of ion and sputtered escape from mars. In *Lunar and Planetary Science Conference*, number 2326, page 2470, 2020.

[16] Sung-Hoon Mok, Jian Guo, Eberhard Gill, and Raj Thilak Rajan. Lunar orbit design of a satellite swarm for radio astronomy. In *2020 IEEE Aerospace Conference*, pages 1–9. IEEE, 2020.

[17] Noah Saks, Albert-Jan Boonstra, Raj Thilak Rajan, MJ Bentum, Frederik Beliën, and Kees van't Klooster. Daris, a fleet of passive formation flying small satellites for low frequency radio astronomy. In *The 4S Symposium (Small Satellites Systems & Services Symposium), Madeira, Portugal, 31 May-4 June 2010 (ESA and CNES conference)*, 2010.

[18] Nicholas Cramer, Daniel Cellucci, Caleb Adams, Adam Sweet, Mohammad Hejase, Jeremy Frank, Richard Levinson, Sergei Gridnev, and Lara Brown. Design and testing of autonomous distributed space systems. In *Proceedings of the AIAA/USU Conference on Small Satellites*, 2021.

[19] Armen H. Poghosyan, Ignasi Lluch, Hripsime Matevosyan, Alex Lamb, C. Moreno, Christianna Taylor, Alessandro Golkar, Judith Cote, Sébastien Mathieu, Stephane Pierotti, Jean-Pierre Magalhaes Grave, Janusz Narkiewicz, Sebastian Topczewski, Mateusz Sochacki, Estefany Lancheros, H. Park, Adriano Camps, and Skolkovo. Unified classification for distributed satellite systems. In *3rd Federated Satellite Systems Workshop*, 2016.

[20] Hugo Sanchez, Dawn M. McIntosh, Howard N. Cannon, Craig Pires, Joshua Sullivan, Simone D'Amico, and Brendan H. O'Connor. Starling1: Swarm technology demonstration. In *Proceedings of the AIAA/USU Conference on Small Satellites*, 2018.

[21] Paolo Bellavista, Antonio Corradi, Luca Foschini, and Alessandro Pernafini. Data distribution service (dds): A performance comparison of opensplice and rti implementations. *2013 IEEE Symposium on Computers and Communications (ISCC)*, pages 000377–000383, 2013.

[22] Daniel Cellucci, Nicholas Cramer, and Jeremy Frank. Distributed spacecraft autonomy. In *2020 AIAA Ascend Conference*. AIAA, 2020.

[23] TH Chung. Offensive swarm-enabled tactics. In *DARPA Briefing*. DARPA, 2017.

[24] Jason Fugate. Distributed spacecraft autonomy (dsa): Development of swarm autonomy capability and scalability for spacecraft. In *Second AI and Data Science Workshop for Earth and Space Sciences*, 2021.

[25] David McComas, Jonathan Wilmot, and Alan Cudmore. The core flight system (cfs) community: Providing low cost solutions for small spacecraft. In *Proceedings of the AIAA/USU Conference on Small Satellites*, 2016.

[26] David McComas. Increasing flight software reuse with opensatkit. *2018 IEEE Aerospace Conference*, pages 1–8, 2018.

[27] Christopher Heistand, Justin Thomas, Nigel H. Tzeng, Andrew Badger, Luis M Rodriguez, Aaron Dalton, Jesse Pai, Austin Bodzas, and D. Thompson. Devops for spacecraft flight software. *2019 IEEE Aerospace Conference*, pages 1–16, 2019.

[28] Daniel Dvorak. *NASA Study on Flight Software Complexity*. NASA, 2009.

[29] Eric D. Yuan, Jon Neff, and Jeffrey Won. Evaluation of a distributed kalman filter for autonomous satellite navigation using dasee. In *2018 IEEE Aerospace Conference*, 03 2021.

[30] batman-adv. Accessed 2022-03-18.

[31] Rti connext micro. Accessed 2022-03-18.

[32] Michael Eder. Hypervisor- vs. container-based virtualization. *Seminar Future Internet, Technical University of Munich*, 2016. Accessed 2022-03-12.

[33] Rami Rosen. Namespaces and cgroups, the basis of linux containers. In *NetDev 1.1: The Technical Conference on Linux Networking*, Seville, Spain, 02 2016.

[34] Dustin M. Geletko, Matthew D. Grubb, John P. Lucas, Justin R. Morris, Max Spolaor, Mark D. Suder, Steven C. Yokum, and Scott A. Zemerick. NASA Operational Simulator for Small Satellites (NOS3): The STF-1 CubeSat Case Study. *Journal of Small Satellites*, 7(3):789–800, December 2018.

[35] Naylor Garcia Bachiega, Paulo Sergio Lopes de Souza, Sarita Mazzini Bruschi, and Simone do Rocio Senger de Souza. Container-based performance evaluation: A survey and challenges. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 398–403, 2018.

[36] Mikael Koskinen, Tommi Mikkonen, and Pekka Abrahamsson. Containers in software development: A systematic mapping study. In Xavier Franch, Tomi Männistö, and Silverio Martínez-Fernández, editors, *International Conference on Product-Focused Software Process Improvement*, pages 176–191, Cham, 2019. Springer International Publishing.

[37] About the open container initiative. Accessed 2022-03-15.

[38] Akihiro Sudo. Comparing next-generation container image building tools. In *Open Source Summit Japan*, 2018.

[39] Build images with buildkit. Accessed 2022-03-10.

[40] Overview of docker compose. Accessed 2022-03-19.

[41] Networking overview. Accessed 2022-03-19.

[42] Stephen Hemminger. Network emulation with netem. In *Proceedings of the 6th Australia's National Linux Conference (LCA2005)*, pages 18–26, 2005.

[43] tgraf/bmon: bandwidth monitor and rate estimator. Accessed 2022-04-29.

[44] iperf3 – iperf3 3.10.1 documentation. Accessed 2022-03-18.