18$^{th}$ International Conference on Space Operations, Montreal, Canada, 26-30 May 2025.
Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

SpaceOps-2025, ID # 492

# Integrating ROS 2 with the Flight Core System

Ana C. Huamán Quispe[a*] Tod Milam[b] Andrew Harris[c] David Edell[d]

[a]*TRACLabs, 1331 Gemini St. Suite 100, Houston, United States*, ana@traclabs.com
[b]*TRACLabs, 1331 Gemini St. Suite 100, Houston, United States*, tmilam@traclabs.com
[c]*Johns Hopkins University Applied Physics Laboratory, United States*, andrew.harris@jhuapl.edu
[d]*Johns Hopkins University Applied Physics Laboratory, United States*, david.edell@jhuapl.edu
*Corresponding Author

**Abstract**

Advanced robotic applications have the potential to revolutionize space exploration. As these systems incorporate state-of-the-art advances in sensing, manipulation, and AI, it is important to consider how these components will be integrated while still ensuring reliable operations in flight systems. Recent efforts in standardizing flight software development have allowed for code reuse and standard practices among the aerospace community – e.g., NASA's cFS and JPL's F' – but these efforts have largely been independent of similar efforts in the larger robotics community via efforts such as the development of the Robot Operating System, currently in its second iteration (ROS 2).

ROS 2 brings together a common and reliable middleware standard (DDS) with a wealth of robotic application, integration, and development tools. Through the Space ROS efforts of NASA, Blue Origin, and Open Robotics, there has been a recent effort to bridge the gap and bring the advantages of ROS2 to the space community. Ultimately, Space ROS hopes to ensure that ROS2 can be used as the primary FSW system by maintaining reliability and safety requirements. In the interim, however, it is possible that ROS2 components should be made to work together with FSW tools such as cFS—either on the flight hardware or as part of the ground system.

To these ends, the authors have developed a set of software libraries for ROS2 interoperability with flight software. In this paper, we present the tools developed with the main goal of *bridging* both flight and robot frameworks, thus allowing the creation of hybrid software applications that integrate both the ROS 2 and the cFS software frameworks. Our toolkit is available open source and has been originallh developed under a NASA Phase II STTR effort and a recently concluded NASA Phase I SBIR project [1].

## 1 Introduction

Over the coming decades, NASA expects to establish a sustained presence on the Moon, in preparation to further exploration into deep space[1]. As missions get farther from Earth, robots will be a fundamental component to support a variety of activities such as: Maintenance and caretaking of spacecraft for human inhabiting, long-term uncrewed terrain exploration, cargo offload, and building large-scale infrastructures on planetary surfaces. To this purpose, it is desirable for robot systems to leverage the state of the art in terrestrial robot technology and, when applicable, use it on space missions. During the last decade, most of the advances in robotics have been assisted by the development of the Robot Operating System (ROS)[2], an Open Source framework that allowed researchers to more effectively develop their work - by avoiding redundant implementation of core utilities - as well as enabling researchers to quickly make their work accessible. At this point in time, a significant share of robotics researchers use ROS for developing, testing and releasing their algorithms. We posit that in order to facilitate the use of SoA advances for space missions, robot researchers should have available a framework that allows them to develop new technology using their default backend (ROS), while providing them with tools that allow the integration of such ROS-based code with flightsystem software.

The robotics and flight software communities have been aware of the need for closer integration between robot and flight software; as such, during the last years, a number of teams have developed diverse communication approaches between both backends; by developing such tools, the hope is that the high-level users might be spared of having to write project-specific code to integrate both sides. Our team at TRACLabs, in collaboration with Johns Hopkins APL, and with NASA support, has developed a number of Open Source tools to provide software tools to this end. Our first effort consisted on a software bridge that allowed the automatic mapping of cFS structures onto ROS 2 messages, these tools, named BRASH, are particularly useful in exposing information shared within a cFS application and making it easily available to ROS 2 graphical UI tools. Our second effort built up on the lessons learned during BRASH development and focused on creating bridging tools that enable the automatic mapping from any 2 message into a generic cFS structure, thus allowing a flight software developer to easily integrate existing terrestrial robotic software into cFS applications, with minimal adaptation. In this paper, we'll describe our latter efforts in more detail.

[1]NASA STTR Phase II contract #80NSSC22CA02 and SBIR Phase I contract #80NSSC24PB469

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

## 2 Background

In the presented work, our goal is to provide tools to simplify the development of flight-capable robotic systems that integrate ROS 2 and flight software, specifically cFS. In this section we'll succintly present a description of both frameworks and the currently existing technology that allows the integration of both, including our own initial work (BRASH)

### 2.1 Flight software

In [3], NASA defines Flight Software a follows:

*The Flight Software (FSW) is, at a fundamental level, the instructions for the spacecraft to perform all operations necessary for the mission. These include all the science objectives as regular tasks (commands) to keep the spacecraft functioning and ensure the storage and communication of data (telemetry). The FSW is usually thought of as the programs that run on the Command & Data Handling (C&DH) avionics, but should also include all software running on the various subsystems and payload(s).*

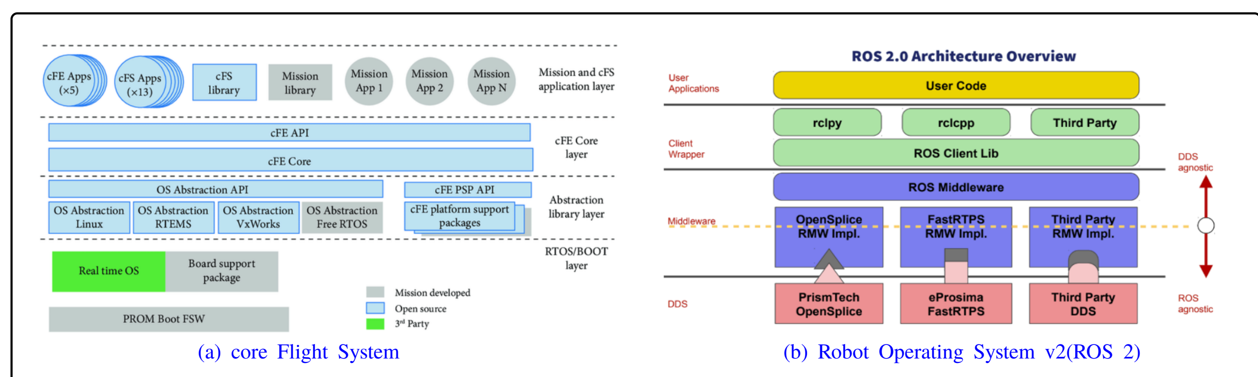
A number of recent efforts have aimed to develop open source FSW architectures—including NASA GSFC's cFS [4], JPL's F' (F Prime) [5], among others. cFS in particular has gained tremendous popularity within the NASA software community, as well as the Open Source community. As noted in [6], the NASA Gateway program has decided to adopt cFS as part of its level-2 sofware commonality requirements. Furthermore, NASA's international partners, such as the Canada Space Agency (CSA) and the European Space Agency (ESA) have shown interest on evaluating cFS; these reasons motivated us to focus on this library in particular.

**cFS:** The core Flight System (cFS) was designed by engineers with the Flight Software Systems branch at Goddard Space Flight Center to be a reusable, component-based software product line that supports desktop development of FSW that is agnostic to the underlying embedded hardware that might be used during flight (Figure 1(a)). Initially designed to support Class-B systems (Non-Human Space Rated Software), cFS supports real-time operating systems, health monitoring, robust startup, and run-time application discovery. It is built on a publish-subscribe architecture of CCSDS-standard messages, which uses shared memory across the single cFE (core Flight Executive) middleware (abstracted communications bus) within a cFS process. The Software Bus Network (SBN) is a cFE plugin that allows cFS applications running across distributed processes to communicate messages across TCP or UDP as well. Since it was certified for flight in 2009, cFS has been adopted in some part by a wide variety of aerospace entities including GSFC, JSC, Ames, JPL, APL, KARI and Astrobotic.

### 2.2 Robot Operating System

At the same time cFS was being developed, so was the popular Robot Operating System (ROS) [2] — which, despite its name, is simply a Linux-based, distributed-computing middleware with some robotics-relevant libraries, utilities, GUIs, and compiler tools (Figure 1(b)). ROS was started at Stanford, taken over and popularized by the robotics incubator Willow Garage, and currently maintained by Open Robotics. Similar to cFS, ROS was created to eliminate the need to reinvent shared, baseline technologies for each new robotic testbed.

ROS, initially oriented towards the research community, saw a commercial rise as the result of the development of several flagship projects providing autonomous navigation, simulation, visualization and control capabilities to robot in a (mostly) robot-agnostic manner. As commercial opportunities transitioned into products, ROS's foundation as a research platform began to show its limitations [7]. Security, reliability and support for large-scale embedded systems became essential requirements that needed to be addressed to further ROS adoption into industry. In response to this need, the second generation of ROS was born.



(a) core Flight System (b) Robot Operating System v2(ROS 2)

**Figure 1:** cFS and ROS 2 are both programming frameworks that layer user-level components on top of utility libraries and a publish/subscribe middleware.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

**ROS 2:** ROS 2 was redesigned from the ground up to address the challenges aforementioned. 2 is based on the Data Distribution Service (DDS). DDS enables ROS 2 to obtain best-in-class security, embedded and real-time support and operations in challenging networking environments. ROS 2 supports a broad range of software components used to develop robotic applications. These components can be divided into three categories: (1) *Middleware:* Encompassing communication among components, (2) *Algorithms:* Such as perception, SLAM, motion planning, robot coordination, among others; and (3) *Developer tools:* A suite of command-line and graphical tools for instrospection, visualization, simulation and logging.

**Space ROS:** Space ROS[2] is an open-source spacecraft flight software framework for developing robotic applications for space being developed by NASA, Open Robotics, Blue Origin and others. Space ROS is a fork of the ROS 2 framework, and conforms to the ROS 2 API that has been hardened to be compatible with the demands of safety-critical space robotic applications [8]. The intent of Space ROS is to **eventually** provide a robust framework for space robotic applications where ROS 2 applications can be reused with little to no modification, enabling the space community to take advantage of the innovation of the ROS community for Earth-based applications. At the time of writing, Space ROS continues in active development and is not yet certification-ready.

Given that Space ROS is a fork of ROS 2, we'll mainly refer to the latter through this document, noting here that the presented tools should be equally applicable for both.

### 2.3 Approaches for communication between ROS 2 nodes and cFS

As mentioned earlier, there are efforts to implement reusable software bridges that allow the integration of ROS 2 and FSW such as cFS. We briefly describe two of these works, each implementing different architecture approaches:

- **Compile-time conversion of ROS 2 nodes to cFS applications:** This approach has been implemented by JAXA in the form of their ROS/cFS converter tool [9, 10]. This tool receives as input node code written with ROS (C++), alongside references to the ROS 2 messages used in the node. The converter generates cFE's header and source files that can thus be used for integration. The main disadvantage, from our perspective, is that this converter only provides a means of wrapping a small set of basic system and message passing utilities, which is certainly useful for some applications, but unlikely to support more complex nodes.

- **ROS 2 as cFE middleware:** This architecture considers to implement a new ROS 2-based software bus for cFE that handles "under the hood" passing cFE message data around using ROS 2 transport mechanisms. Such an approach - focusing on DDS in general, rather than ROS 2 specifically - is currently being investigated by the Distributed Spacecraft Autonomy (DSA) project [11] and by the NSF Center for Space, High-performance, and Resilient Computing (SHREC) at the University of Pittsburgh[3]. While these efforts are commendable and will serve to "make transparent" the use of DDS when deploying distributed systems, we believe that this approach is not as appropriate for single-agent cFE architectures, nor would it be appropriate when communicating to Ground Systems that are best interfaced through CI and TO cFE applications

### 2.4 BRASH

TRACLabs and the John Hopkins University's Applied Physics Laboratory (APL) developed, under a NASA Phase II STTR effort [12, 13], a toolkit for ROS 2 interoperability with flight software, named BRASH (Bridge for ROS 2 Applications to Space Hardware). BRASH consists of a plugin-based bridge that *converts FSW messages into ROS 2 messages*, and transmits data between the systems accordingly (see Figure 2). To date, we have developed plugins for connecting ROS 2 with cFS systems via the Software Bus Network (SBN) application; for connecting a ground-side ROS 2 system to flight-side cFE applications via the Telemetry Output (TO) and Command Ingest (CI) lab utilities; and for connecting with a simple F' GDS via web services. Implemented as a Python ROS 2 node using a simple factory interface, our bridge framework provides useful tools for both automatic ROS 2 message construction and translation to make the integration as seamless and maintainable as possible. BRASH is available as Open Source with an MIT license and with growing documentation.

## 3 Design of Bridge tools ROS 2 $\rightarrow$ cFS

The main goal of this project was to develop a conversion tool that allows a user to use existing ROS2 messages to send and receive information from a cFS application. We designed our prototype for a simple user case (Section 3.1) involving controlling a single robot; afterwards, we address slightly more complex scenarios involving multiple robots that use ROS2 on the flight side (Section 3.4).

[2]Space ROS: https://space.ros.org/
[3]https://github.com/CHREC/openSBD

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

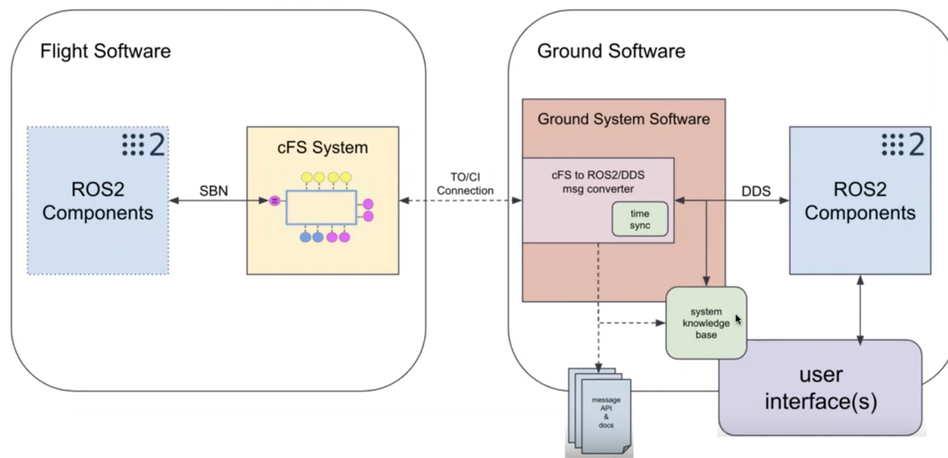



**Figure 2:** Generalized Architecture for BRASH: On the ground side (right) a cFS/ROS 2 message converter allows data mapping between their corresponding message types. On the flight side (left), a similar plugin based on the SBN application allows - if needed - a ROS 2-enabled robot to communicate directly with cFS.

### 3.1 Single-robot scenario

We defined a baseline test case, shown in Figure 3, which consists on 2 distinct segments:

- *Ground:* An operator in mission control who sends commands to a remote robotic asset, such as a robot arm, to direct its motion. It receives telemetry data back with information on the latest state of the system. The operator uses ROS2/SpaceROS tools, such as User Interfaces (e.g. Rviz) to visualize said data.
- *Flight:* Remotely, a processor is running cFS and a task-specific cFS application to command the robotic asset. In this scenario, the robot is commanded directly by cFS.

Ground and flight are expected to communicate using standard cFS channels of communication such as the CI/TO applications (using the UDP protocol).

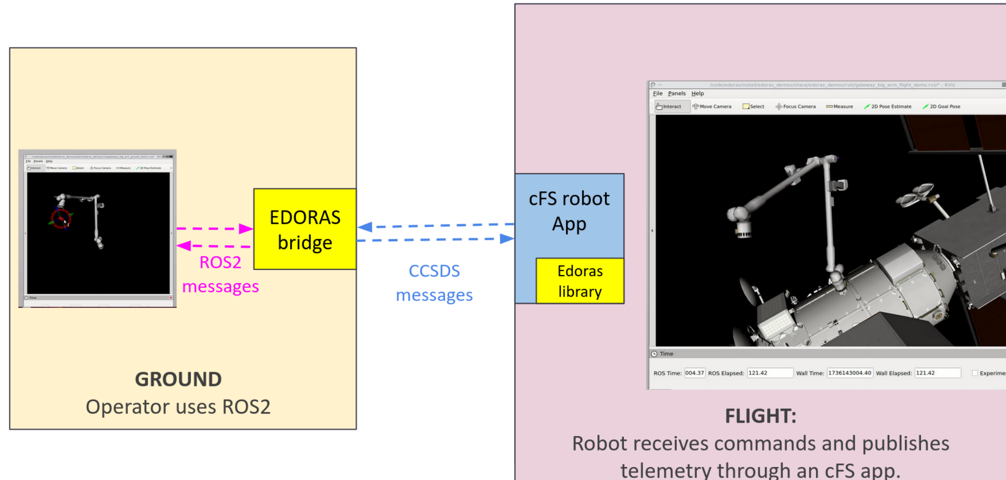



**Figure 3:** Single-robot application scenario considered

### 3.2 Ground-side bridge

The bridge's main goal is to perform a mapping between ROS2 messages and a data structure that can be parsed by a cFS application. As noted in Section 2, our previous work has already addressed the automatic mapping of cFS messages into ROS2 messages. This process took advantage of the fact that cFS messages are well-defined C structures that consist on 2 parts: (1) A CCSDS header, with identifying information such as message id, and message length; and (2) Actual data to be transmitted, normally defined as a C structure of size known at compile time.

The goal of the presented work is to map messages in the opposite direction; that is, ROS2 messages onto data structures that can be used in cFS applications. This process is not straightforward as ROS2 messages have less constraints in their construction, such as:

- ROS2 messages allow for message nesting, that is, messages can contain fields that are themselves messages.
- ROS2 messages allow for different types of arrays: (a) Fixed-size array, for which their capacity is always known, (b) Dynamically-sized array, with an arbitrary size, (c) Dynamically-sized array with a maximum size restriction.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

- ROS2 messages support a number of primitive types including strings of arbitrary length.

The last two points are of particular interest, given that most cFS applications are written using C, which does not natively support dynamically-sized arrays, as opposed to C++ does with the use of structures such as vectors. While in theory, a user could write a cFS application in C++, we wanted to investigate whether we could also support the most common user case of using C.

Our original approach to this Objective was to develop a tool to automatically generate data structures in C (or C++) that mapped the information from the ROS2 fields accordingly. While looking for strategies for code generation, we came across the *introspection libraries* used by ROS2. These libraries get as input the ROS2 messages files (with .msg extension) and automatically generate C and C++ structures to hold the message information. These structures follow a pre-determined templated structure that allows a user to write / read data from them by using the message's field names.

**Bridge Configuration** The bridge is implemented as a C++ ROS2 node. It accepts configuration parameters in the form of a yaml file such as the one depicted in Fig.4.

```yaml
conversion_node:
  ros__parameters:

    communication:
      bridge_port: 1235
      fsw_port: 1235
      fsw_ip: "10.5.0.3"
      telemetry_ip: "0.0.0.0"

    command: ['cmd_twist']
    telemetry: ['tlm_pose']

    cmd_twist:
      type: "geometry_msgs/msg/Twist"
      topic: "cmd_vel"
      mid: 0x1827 # EDORAS_APP_CMD_MID

    tlm_pose:
      type: "geometry_msgs/msg/PoseStamped"
      topic: "telemetry_pose"
      mid: 0x827 # EDORAS_APP_TLM_MID
```

**Figure 4:** Structure where data needed to serialize command data sent back to cFS is stored

The configuration file contains parameters of two types: (1) Communication parameters, and (2) Message mapping parameters. The *communication parameters* contain information regarding the IP addresses and ports associated to the ground and the flight machine. The IPs are specific to a user's particular setup, whereas the ports (`bridge_port` and `fsw_port`) are the ones used by the TO cFS app, hence it is a fixed value. The `telemetry_ip` value (0.0.0.0) indicates that our bridge is open to receive data of any other agent in our network.

The *Message mapping* parameters associate the ROS2 channels of communication with their respective cFS counterparts. These parameters are grouped in 2 lists of strings: **command** and **telemetry**. Each string map to a dictionary of 3 values:

- *type:* The type of ROS2 message to be processed.
- *topic:* The name of the ROS2 topic associated with this stream of data.
- *mid:* The message ID that identifies data on the cFS side.

This information is stored in two maps within the bridge, where the key is the topic name. These maps are aptly named `cmd_info_` and `tlm_info_`. The next subsections explain how the stored data is used for the parsing/decoding of the ROS2 messages into/from CCSDS packets.

**Command processing on the Ground** The ground bridge iterates over all the elements in `cmd_info_` and creates ROS2 subscribers to their corresponding `topic` fields. Figure 5 shows the data structure that is stored as a value for the `cmd_info_` map.

Note that the bridge is not aware of the type of ROS2 message that will be received by these subscribers at compile time, so we use *GenericSubscription* instances, which allow the publisher to have the message type given at creation time. Figure 6 shows the function used to add a subscriber in the ground bridge. The argument *_message_type* contains the ROS2 message type to be received (e.g. `geometry_msgs/msg/Pose`).

Any time a ROS2 message is published in one of the subscribed topics, the bridge will invoke the *subscriberCallback* function, which is responsible of actually performing the conversion of the ROS2 message received into a CCSDS packet. Figure 7 shows a diagram of the process. A CCSDS packet is an array of bytes composed of 2 distinct parts: The CCSDS header and the actual data we wish to transmit. The CCSDS header

$18^{th}$ International Conference on Space Operations, Montreal, Canada, 26-30 May 2025.
Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

```
struct CmdInfo_t {
  std::string msg_type;
  std::string topic;
  uint16_t mid;
  std::string interface_name;
  std::string interface_type;
  std::shared_ptr<rcpputils::SharedLibrary> library;
  const TypeSupport_t* type_support;
  const TypeInfo_t* type_info;
};
```

**Figure 5:** Structure where data needed to serialize command data sent back to cFS is stored

```
bool GroundConversion::addSubscriber(const std::string &_topic_name,
                                     const std::string &_message_type)
{
 auto sub = this->create_generic_subscription(_topic_name, _message_type,
      rclcpp::QoS(1),
      [this, _topic_name](std::shared_ptr<const rclcpp::SerializedMessage> _msg)
      {
         this->subscriberCallback(_msg, _topic_name);
      } );
  subscribers_[_topic_name] = sub;
  return true;
}
```

**Figure 6:** Snippet from ground bridge showing how a generic subsciption is created to accommodate the runtime configuration of the bridge using the configuration file.

is defined by the CCSDS protocol as consisting of **8 bytes** for messages that carry command information. These 8 bytes contain:

- Bytes[0:2]: Stream ID
- Bytes [2:4]: Sequence number
- Bytes [4:6]: Length of message
- Byte [6]: Code
- Byte [7]: Checksum

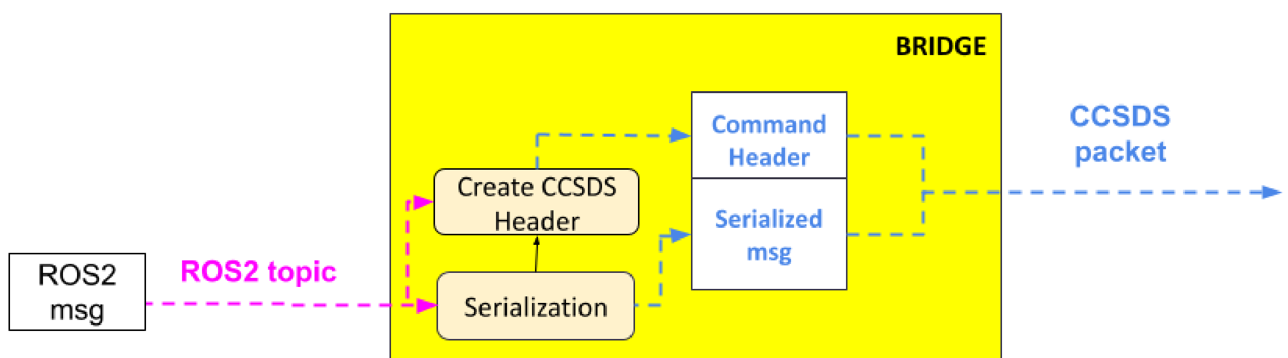



**Figure 7:** Flow of information for commands going from ROS2 to cFS using the ground bridge

Our bridge, upon processing the data from the ROS2 message into an array of bytes, will attach a CCSDS header on top of the array, thus creating a CCSDS packet. Figure 8 shows the function used by the ground bridge to accomplish this. The function takes as input, the `mid` code provided in the configuration file for each command, as well as the array of bytes containing the ROS2 messsage data (`_data_buffer`) and its size (`_data_size`). The function returns the CCSDS packet (`_cmd_packet`) and its size.

As previously described, the ROS2 message data is stored as an array of bytes. We could choose multiple ways to do this, using third-party libraries such as protobuf or using an ad-hoc serialization library. For this project, we chose to simply use the default serialization provided by ROS2 at a lower level of communication. As the reader might be aware of, ROS2 provides the user to choose among a number of middleware representations, each of which serializes the data on its own way. For this proof-of-concept, we used the default `fastrtps` library[4], which in turn uses the CDR[5] protocol for serialization.

[4] https://github.com/ros2/rmw_fastrtps?tab=readme-ov-file
[5] https://github.com/eProsima/Fast-CDR/tree/master

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

```
size_t createCmdPacket(const uint16_t &_mid,
                       const uint8_t &_code,
                       const uint16_t &_seq,
                       const size_t &_data_size,
                       unsigned char** _data_buffer,
                       unsigned char** _cmd_packet)
{
   const uint8_t header_size = 8;
   uint16_t packet_length = header_size + _data_size;
   uint16_t packet_length_ccsds_offset = packet_length - 7;

   unsigned char  cmd_header[header_size];

   cmd_header[0] = (_mid >> 8) & 0xFF;
   cmd_header[1] = _mid & 0xFF;
   cmd_header[2] = (_seq >> 8) & 0xFF;
   cmd_header[3] = _seq & 0xFF;
   cmd_header[4] = (packet_length_ccsds_offset >> 8) & 0xFF;
   cmd_header[5] = packet_length_ccsds_offset & 0xFF;
   cmd_header[6] = _code;
   cmd_header[7] = 0;

   // Add header bytes on top of the serialized data
   size_t offset = 0;

   memcpy(*_cmd_packet + offset, &cmd_header, sizeof(cmd_header));
   offset += sizeof(cmd_header);
   memcpy(*_cmd_packet + offset, *_data_buffer, _data_size);

  return packet_length;
}
```

**Figure 8:** Snippet from ground bridge that adds a CCSDS command header on top of message data to be sent back to the cFS application.

As noted earlier, the ground bridge uses generic subscribers to handle message types being defined at runtime. An advantage of this implementation is that the subscriber callback receives the ROS2 message as a shared point of an `rclcpp::SerializedMessage` instance. This class has a member variable named `serialized_message_` which is a structure of type `rcutils_uint8_array_t`, and shown in Figure 9. The field `buffer` stores an array of bytes of size `buffer_length` containing the ROS2 message data. Our bridge, upon reception of a new message, will simply extract the buffer array, length and capacity, and will pack them up in an array of bytes that can then be joined to the command header and sent back to cFS.

```
uint8_t* buffer
size_t buffer_length
size_t buffer_capacity
rcutils_allocator_t allocator
```

**Figure 9:** Structure `rcutils_uint8_array_t`.

Figure 10 show a diagram of the bytes that are sent to cFS. We use the UDP protocol (through the Command Ingest cFS application) to send the data from the ground to the spacecraft machine running cFS.

**Figure 10:** Diagram of all the bytes that constitute the data being sent to cFS

**Telemetry processing on the ground** Figure 11 shows the flow of information from a CCSDS packet being received by the bridge till its published as a ROS2 topic. Unlike for the command case, in which the arrival of a new command was an event that invoked a callback, the telemetry data does not produce such event. Instead, the bridge runs a timer at a given rate, such as 20Hz, and it constantly queries whether a new message from the cFS side has been received. We use the `recvfrom` function for this purpose.

The CCSDS packets are received from the TO (telemetry output) cFS application. Per the CCSDS protocol, the packet consists on a primary header (8 bytes), a secondary header (4 bytes), padding (4 bytes)

18$^{th}$ International Conference on Space Operations, Montreal, Canada, 26-30 May 2025.
Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

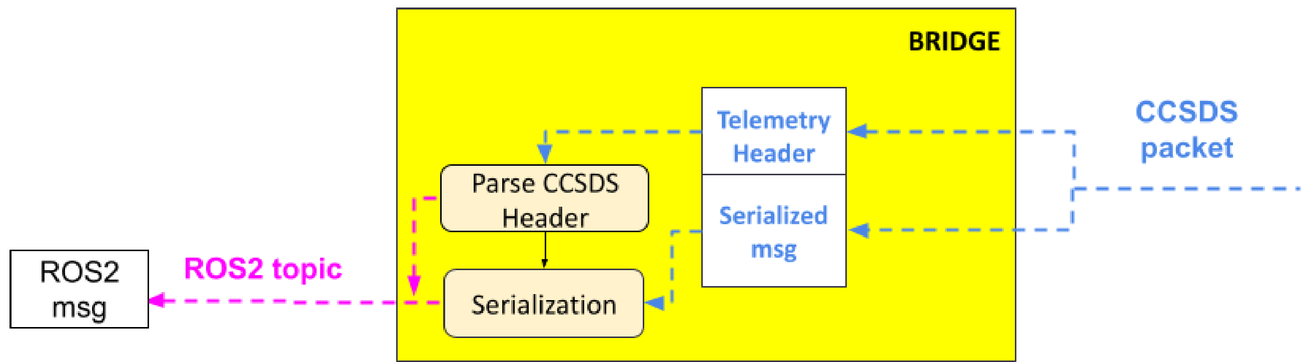



**Figure 11:** Flow of information for commands going from ROS2 to cFS using the ground bridge

and then the message data. Figure 12 shows a diagram of the bytes in the packet. In the same manner as for the command message, the data sent in this packet is stored by having 1 byte storing the buffer length, 1 byte for buffer capacity and the rest of bytes contain the serialized ROS2 message information.

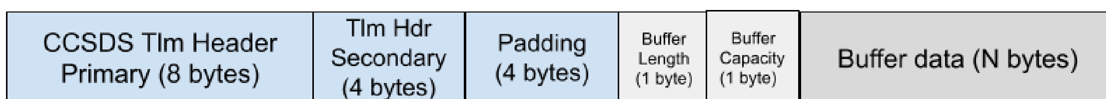



**Figure 12:** Diagram of all the bytes that constitute the data being sent to cFS

Figure 13 shows the function called by the bridge to check for incoming messages. As a first step, the function `receiveTlmPacket` calls `recvfrom`. If a message is received, and its size is bigger than that of a telemetry header size, this function extracts the MID and the message size information from the header, as well as the array of bytes with the message data. Afterwards, the ground bridge evaluates whether the MID corresponds to one of the telemetry MIDs provided in the configuration file at runtime. If this is the case, then the bridge proceeds and creates a ROS2 serialized message (of type `rclcpp::SerializedMessage`) with the data received and publish it to the corresponding ROS2 topic that maps to the received MID.

```
void GroundConversion::receiveTelemetry()
{
  uint16_t mid;
  uint8_t* buffer = NULL;
  size_t buffer_size;
  bool received_data;

  do
  {
     received_data = receiveTlmPacket(mid, &buffer, buffer_size);
     if( received_data )
     {
        // Check if this telemetry's MID is one the bridge is subscribed to
        std::string topic_name;
        if( hasMid(mid, topic_name) )
        {
           // Publish data
           rcutils_uint8_array_t* serialized_array = nullptr;
           serialized_array = make_serialized_array(buffer);
           rclcpp::SerializedMessage serialized_msg(*serialized_array);
           publishers_[topic_name]->publish(serialized_msg);

           // Clean up
           free(buffer);
           rmw_ret_t res = rcutils_uint8_array_fini(serialized_array);
        } // if hasMid

     } // if received_data

  } while(received_data);
}
```

**Figure 13:** Snippet from ground bridge that reads the telemetry data from the specified port that is listening for the TO cFS app.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

### 3.3 Flight Parse library

Up till the previous section, we have explained how the ground bridge operates to create/receive CCSDS packets to send/receive to/from cFS. In this section, we describe the tools we created so a user can decode/encode the messages that are communicated by the bridge.

**edoras_core library** The data that is transferred between the ground bridge and cFS is an array of bytes containing the buffer length, capacity and the serialized ROS2 message. We created a C++ library, named `edoras_core`[6] that contains a number of helper functions that allow a user to extract values of the different parameter fields of a ROS2 message, as well as functions to assign values given a parameter name and its type.

Figure 14 shows a few of the functions from the `edoras_core` library used on a cFS application – in this case, the app responsible for receiving a 3D pose command from the ground representing a desired pose for the end effector of a robot arm. These functions are:

- `from_uint_buffer_to_msg_pointer`: Transforms an array of bytes into a pointer to a C structure storing the ROS2 message data.
- `get_float64`: Receives as input the message pointer aforementioned and the name of the field to be queried. It returns true if the field exists or false otherwise, while updating a parameter passed by reference with the obtained value.

If a cFS app needs to access to data from the ground bridge, it simply needs to subscribe to their respective MIDs (using the `CFE_SB_Subscribe` function), and once the message is received, use the aforementioned functions to extract the required information. Besides the `get_float64` method, our library has a number of *getter* functions for the set of primitive types supported by ROS2 and that are compatible with C, such as uint8_t, char, float32_t, a string (defined as a const char*) among others. It is relevant to point out that, as ROS2 messages support nesting of parameters, the parameter field can accept the nested name of a parameter (e.g. `position.x`).

The edoras_core library has been implemented with dependencies to ROS2 introspection libraries [7], which allows us to do the querying of the message pointer using the field names. We took inspiration for this approach from a public repository from OSRF that illustrate how a user can leverage the introspection libraries to parse and extract data from ROS2 messages to yaml files [8].

```
void EdorasAppProcessGroundCommand(CFE_SB_Buffer_t *SBBufPtr)
{
   offset = 8;
   uint8_t* msg_pointer = NULL;
   size_t buffer_size;
   msg_pointer = from_uint_buffer_to_msg_pointer( (uint8_t*)SBBufPtr, offset,
                 parse_pose_.ts, parse_pose_.ti, &buffer_size);

   // Get data
   double pos_x, pos_y, pos_z, orient_x, orient_y, orient_z, orient_w;
   get_float64(msg_pointer, parse_pose_.ti, "position.x", &pos_x);
   get_float64(msg_pointer, parse_pose_.ti, "position.y", &pos_y);
   get_float64(msg_pointer, parse_pose_.ti, "position.z", &pos_z);
   get_float64(msg_pointer, parse_pose_.ti, "orientation.x", &orient_x);
   get_float64(msg_pointer, parse_pose_.ti, "orientation.y", &orient_y);
   get_float64(msg_pointer, parse_pose_.ti, "orientation.z", &orient_z);
   get_float64(msg_pointer, parse_pose_.ti, "orientation.w", &orient_w);
   // Send to robot
   sendPoseCmd(&commData, pos_x, pos_y, pos_z, orient_x, orient_y, orient_z, orient_w);
 }
```

**Figure 14:** (Simplified) Snippet from ground bridge that reads the telemetry data from the specified port that is listening for the TO cFS app.

The process to create telemetry messages to be sent back to the ground bridge is fairly similar to the one described to decode command messages. The main difference is that we need to create a structure that will be used to store the telemetry message. An example of this structure is show in Figure 15. It requires 2 fields: A telemetry header and an array of bytes (unsigned int type). The header is a type defined within cFE core libraries. As for the array of bytes, the size does not need to be exact, as long as it is bigger than the message size.

[6] https://github.com/traclabs/edoras_core
[7] https://github.com/ros2/rosidl/tree/humble
[8] https://github.com/osrf/dynamic_message_introspection

18$^{th}$ International Conference on Space Operations, Montreal, Canada, 26-30 May 2025.
Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

```
typedef struct
{
    CFE_MSG_TelemetryHeader_t  TlmHeader;
    uint8_t data[260];
} JointStateData_t;
```

**Figure 15:** Example of a telemetry structure used for storing joint state data

```
int32 EdorasAppReportHousekeeping(const CFE_MSG_CommandHeader_t *Msg)
{
  //>>> snip, snip, snip

   // If data received from robot update telemetry data to send back to ground
   uint8_t* js_msg = create_msg(parse_joint_state_.ti);

   // Fill data
   resize_sequence(js_msg, parse_joint_state_.ti, "position", 7);

   set_float64(js_msg, parse_joint_state_.ti, "position.0", joints[0]);
   set_float64(js_msg, parse_joint_state_.ti, "position.1", joints[1]);
   set_float64(js_msg, parse_joint_state_.ti, "position.2", joints[2]);
   >>> snip, snip, snip

   resize_sequence(js_msg, parse_joint_state_.ti, "name", 7);

   //>>> snip, snip, snip
   set_const_char(js_msg, parse_joint_state_.ti, "name.4", "big_arm_joint_6");
   set_const_char(js_msg, parse_joint_state_.ti, "name.5", "big_arm_joint_7");
   set_const_char(js_msg, parse_joint_state_.ti, "name.6", "big_arm_joint_8");

   set_int32(js_msg, parse_joint_state_.ti, "header.stamp.sec", sec);
   set_uint32(js_msg, parse_joint_state_.ti, "header.stamp.nanosec", nanosec);

   // Convert data to serialized version
   uint8_t* tlm_data = NULL;
   size_t tlm_data_size;
   tlm_data = from_msg_pointer_to_uint_buffer(js_msg, parse_joint_state_.ts,
              parse_joint_state_.ti, &tlm_data_size);

   // Fill telemetry message
   for(size_t i = 0; i < tlm_data_size; i++)
      memcpy(&tlm_joint_state.data[i], (uint8_t*)(tlm_data) + i, sizeof(uint8_t));

   CFE_SB_TimeStampMsg(&tlm_joint_state.TlmHeader.Msg);
   CFE_SB_TransmitMsg(&tlm_joint_state.TlmHeader.Msg, true);

   // >> snip, snip, snip
    return CFE_SUCCESS;
}
```

**Figure 16:** (Simplified) Snippet from cFS app filling a telemetry message with data to be read by the ROS2 bridge

Figure 16 shows how the data structure described can be filled with ROS2 data. This snippet uses some of the helper functions available in `edoras_core` to create a message:

- `create_msg`: Function that creates a pointer to a C structure that contains the ROS2 mesage information. The function requires as input a rosidl typesupport structure loaded using the name of the message to be encoded (e.g. `sensor_msgs/msg/JointState`. This input has also been initialized using a helper function.
- `resize_sequence`: This function sets the size of a message field that is a dynamic array. In this case we are filling a JointState message, which has array fields for `name` and `position`. You need to provide the name of the field to be resized as argument.
- `set_float64`: Set the value of a field with a primitive type of type double (float64). Note that we are setting values corresponding to elements of an array (e.g.`positions[0]`), so we must provide the name of the field including the index (e.g. `position.0`).
- `set_const_char`: Set the value of a field that is a string.
- `from_msg_pointer_to_uint_buffer`: This function creates an array of bytes from serializing the C structure that has been filled with the previous steps. After this step, all that is needed to send the telemetry back is to attach a telemetry header on top of it.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

### 3.4 Multi-robot scenario

After developing the ground bridge and testing it with single-robot testing setups, we received some feedback regarding alternative user cases, one of them being multi-robot scenarios, where each robot is already equipped with a ROS2 control interface, which is a reasonable assumption, as robots developed for ground applications often have a ROS2 driver available, hence it is expected that in the future, commercial robots developed for remote applications will likely have ROS2 interfaces . An example of such scenario is shown in Figure 17. This setup has 3 distinct segments:

- `Ground`: The machine running on the ground; operator uses ROS2 to send commands and visualize telemetry back from the spacecraft. This machine has a ground bridge running, which allows for communication between the bridge and cFS.
- `Spacecraft`: The processor running cFS, with a task-specific application to control the robot.
- `Flight`: A processor running on a robot, such as a wheeled rover. This processor is running ROS2 software to control the robot, and a *flight bridge* that allows it to communicate with cFS.

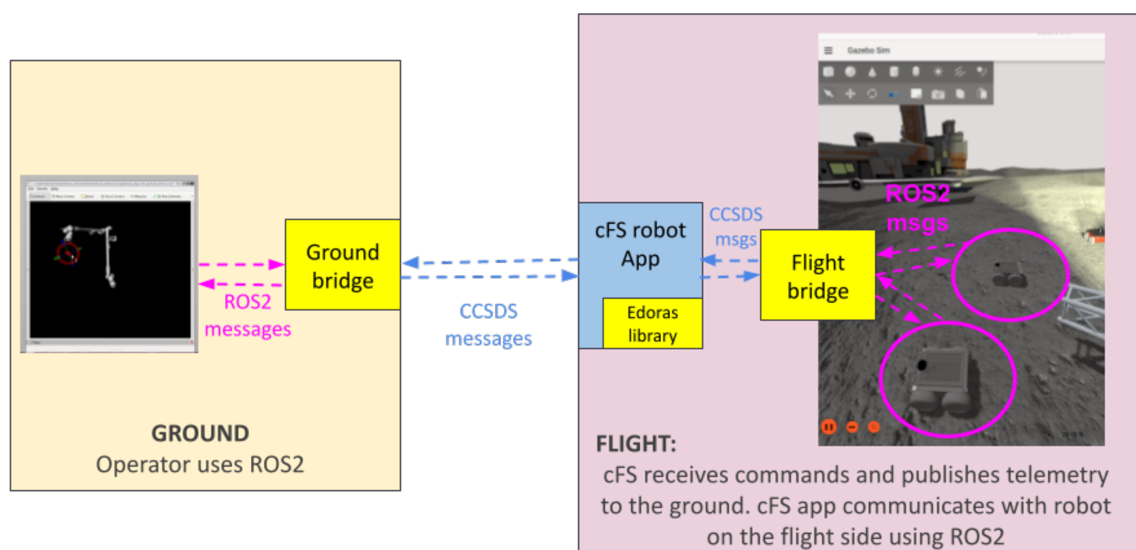



**Figure 17:** Multi-robot application scenario considered

The flight bridge cannot be the same as the ground bridge; this is due to the fact that the ground bridge uses CI/TO to send and receive CCSDS packets. To the best of our knowledge, the TO app can only send data back to **one IP at a time**. This means that if we were to use multiple ground bridges (say, one on the ground and one on the flight side), then the telemetry data from cFS would only be sent to one of the bridges. There are workarounds for this, such as sending a request to TO to change the receiving IP, but those would involve additional communication traffic that could delay the regular transmission of data.

In light of this, we decided to develop a bridge specific for the flight side. In a manner similar to the BRASH bridge, we chose to use the SBN cFS application to communicate with the bridge. This also gives us the opportunity of potentially have a number of flight bridges, as opposed to one - as in the ground case - given that the SBN app considers that there are potentially multiple *peers* in the network, hence multiple flight bridges / cFS executables are a possibility.

### 3.5 Flight bridge using the SBN app

The flight bridge is implemented as an *SBN Peer*, which follows a similar pattern as the ground bridge, with the main differences being 2: (1) The messages exchanged need an additional `SBN` header on top, and (2) A SBN Peer needs to interchange additional messages – other than command / telemetry – with cFS to maintain the communication active. These additional types of messages include:

- SBN_UDP_HEARTBEAT_MSG (0xA0): Message requesting a heartbeat to be sent back.
- SBN_SUB_MSG (0x01): Message where payload is subscription request.
- SBN_UNSUB_MSG (0x02): Message where payload is an unsubscription request.
- SBN_APP_MSG (0x03): Message with actual data to be transferred from cFS.
- SBN_PROTO_MSG (0x04): Message where payload is the SBN protocol (for verification).

The flight bridge has the necessary logic to process to manage the low-level messages that are not specific to data transfer (e.g. heartbeat, protocol). For our bridging purposes, the messages with type `0x03` are of main interest, as they carry actual data. The bridge identifies the message type by parsing the header of any incoming packet. Figure 18 shows the structure of the messages being received. As you can observe,

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

the last half of the message has the same structure as the one used for the ground bridge (CCSDS header + buffer information + data serialized). The 11 first bytes, however, are now the SBN header, which contains information about the message size, the type and other identifying information of the message source.

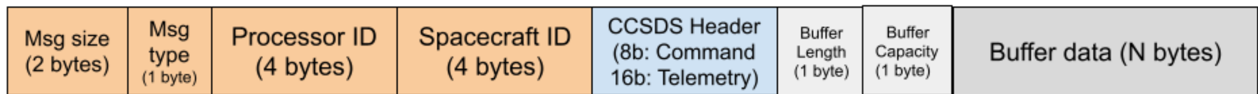



**Figure 18:** Message structure for flight bridge using SBN. The message type – which is used to identify which message carries actual data – is found in the third most-significant byte.

Figure 19 shows the configuration file for the flight bridge used for a 2-rover scenario. In a manner similar to the ground bridge, the file contains information of 2 types: (1) Communication, which involves IP and port addresses, as well as additional ids that identify a SBN peer within the software bus, and (2) ROS2 mapping, which is contained within the *telemetry* and *command* lists. These parameters have the information of how to map ROS2 topics with their corresponding MID on the cFS app. Note that in this particular example, we have command and telemetry mappings corresponding to 2 robots under different namespaces (`robot_1` and `robot_2`). We could have created 2 bridges, one per each robot on the flight side, but for simplicity, we just used a unique flight bridge for this setup.

```
ros__parameters:

  communication:

    # FSW information
    peer_ip: "10.5.0.3"   # IP for fsw
    peer_port: 2235     # 2234 for single-host (cpu1), 2235 for cpu2
    peer_processor_id: 2  # Processor id
    peer_spacecraft_id: 0x42 # Spacecraft id

    # rosfsw information
    udp_receive_port: 2236
    udp_receive_ip: '0.0.0.0'
    processor_id: 3
    spacecraft_id: 0x42 # 0x42 == 66

  command: ['pose_tlm_1', 'pose_tlm_2']
  telemetry: ['twist_cmd_1', 'twist_cmd_2']

  pose_tlm_1:
    type: "geometry_msgs/msg/PoseStamped"
    topic: "/robot_1/pose"
    mid: 0x1829 # EDORAS_APP_POSE_1_FLIGHT_MID

  pose_tlm_2:
    type: "geometry_msgs/msg/PoseStamped"
    topic: "/robot_2/pose"
    mid: 0x182A # EDORAS_APP_POSE_2_FLIGHT_MID

  twist_cmd_1:
    type: "geometry_msgs/msg/Twist"
    topic: "/robot_1/cmd_vel"
    mid: 0x82A # EDORAS_APP_TWIST_1_FLIGHT_MID

  twist_cmd_2:
    type: "geometry_msgs/msg/Twist"
    topic: "/robot_2/cmd_vel"
    mid: 0x82B # EDORAS_APP_TWIST_2_FLIGHT_MI
```

**Figure 19:** Configuration file for flight bridge that controls 2 rovers

# 4 Demonstrations

We developed 4 demonstation scenarios to test our developed bridge tools. All these setups use simulated robots:

1. Control a wheeled-rover in Rviz, using the ground bridge on the operator side, and cFS on the spacecraft processor communicating with the rover via serial communication.
2. Control a 7-DOF robot arm in Rviz, using the ground bridge on the operator side, and cFS on the spacecraft processor communicating with the arm via serial communication.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

3. Control 2 rovers on a remote station in Gazebo, using the ground bridge on the operator side, cFS on a spacecraft processor, and the flight bridge on a ROS2 machine simulating the robots.
4. Control 2 robot arms on a simulated Gateway station in Rviz, using the ground bridge on the operator side, cFS on a spacecraft processor, and the flight bridge on a ROS2 machine simulating both robots.

We'll describe each of these in detail in the following subsections:

### 4.1 Control a single rover

This demo has the following characteristics:

- Demo is run using 2 Docker containers, one simulating a ground machine, the other the spacecraft processor.
- The ground machine has ROS2 installed and it has a ground bridge running.
- The flight machine has a cFS executable running. In this same machine, the rover is being simulated. The cFS process and the executable simulating the rover communicate using serial control
- The ground machine and the flight machine are only expected to communicate through the CI/TO cFS applications, using UDP.

The demo consists on an operator publishing Twist commands (`geometry_msgs/msg/Twist`) in a ROS2 topic, and subscribing to a ROS2 topic carrying the rover location (`geometry_msgs/msg/Pose`). The command is being sent using a Qt tool (`robot_steering`) and the telemetry obtained is visualized using a Display of type Pose. A link to a video of this demonstration is available in the image's description.

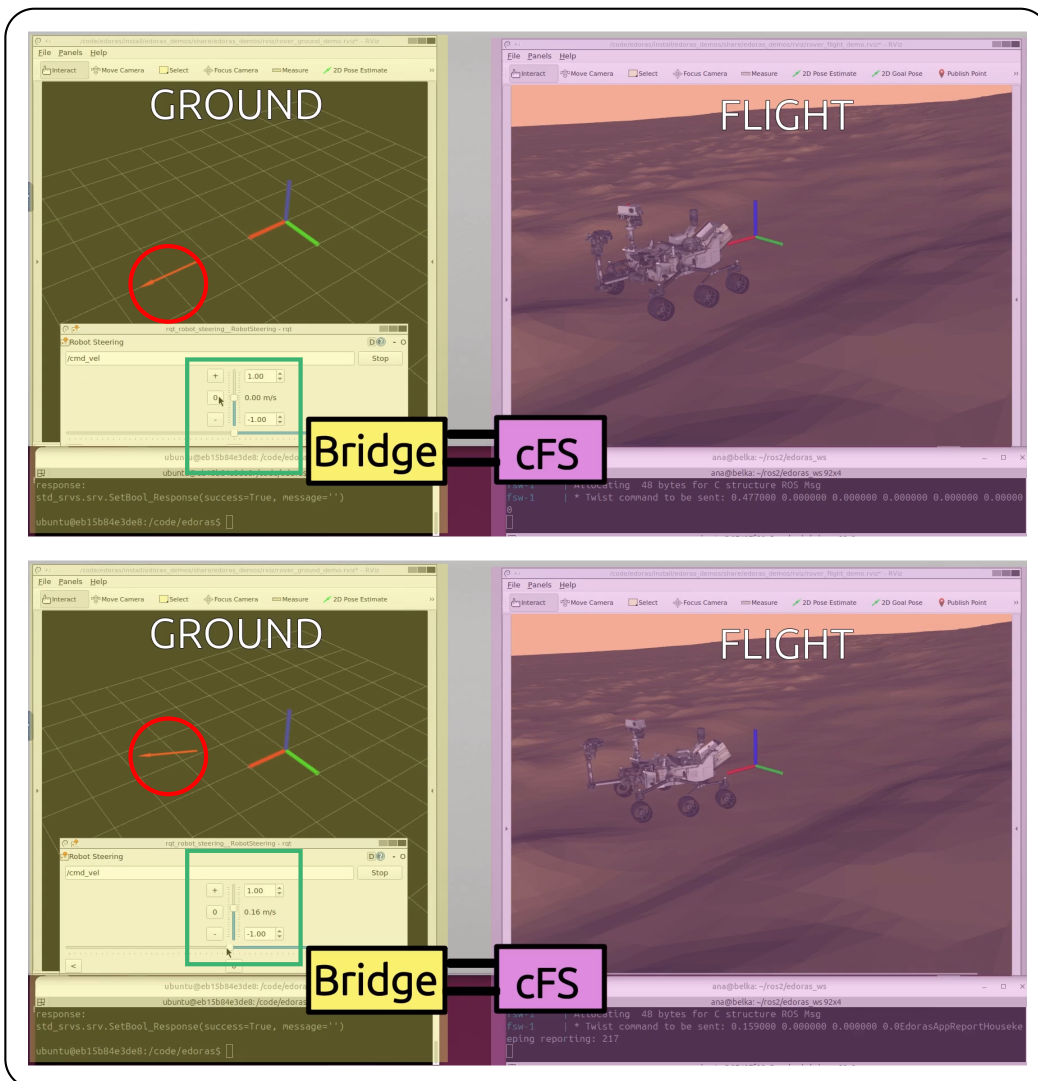



**Figure 20:** Demo: Controlling a wheeled-rover using the ground bridge on the ground, cFS on the flight side. **Top:** Task state after commanding the robot with a positive linear velocity (green box highlights the Rviz tool to send the Twist message, right circle shows the telemetry data received through the bridge). **Bottom:** Task state after adding angular velocity to the command message. Robot is now turning towards its right.
**Video:** https://youtu.be/hkcide0oox4?si=03nP0WlNpKjkAyiL

While testing this demostration, we didn't observe any communication issues. Note that the Twists commands are being sent to cFS at a relatively low frequency (10Hz), as this is the default rate for the `robot_steering` tool we use. As for the telemetry, we are sending it back at low speed (1Hz), as it is customary for sending data back to Earth. With higher speeds (e.g. 100Hz) we did receive warning messages from the TO application, indicating that the default configuration could not acommodate such speeds.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

## 4.2 Control a single robot arm

This demo has the following characteristics:

- Demo is run using 2 Docker containers, one simulating a ground machine, the other the spacecraft processor.
- The ground machine has ROS2 installed and it has a ground bridge running.
- The flight machine has a cFS executable running. In this same machine, the robot arm is being simulated. The cFS process and the executable simulating the robot arm communicate using serial communication.
- The ground machine and the flight machine are only expected to communicate through the CI/TO cFS applications, using UDP.

The demo consists on an operator publishing 3D Pose commands (`geometry_msgs/msg/Pose`) in a ROS2 topic, and subscribing to a ROS2 topic carrying the arm's joint state (`sensor_msgs/msg/JointState`). The command is being sent using an Interactive Marker's gimbal published by a simple ROS2 node we created for this purpose, and the telemetry obtained is visualized using a Display of type RobotModel. A link to a video of this demonstration is available in the image's description. For this demo, we created a custom ROS2 node on the flight side; this node waits for a 3D Pose to be received. Upon reception, the inverse kinematics solver calculates a joint state that moves the end effector to reach the desired 3D pose, then commands the robot to move towards that pose using simple interpolation.

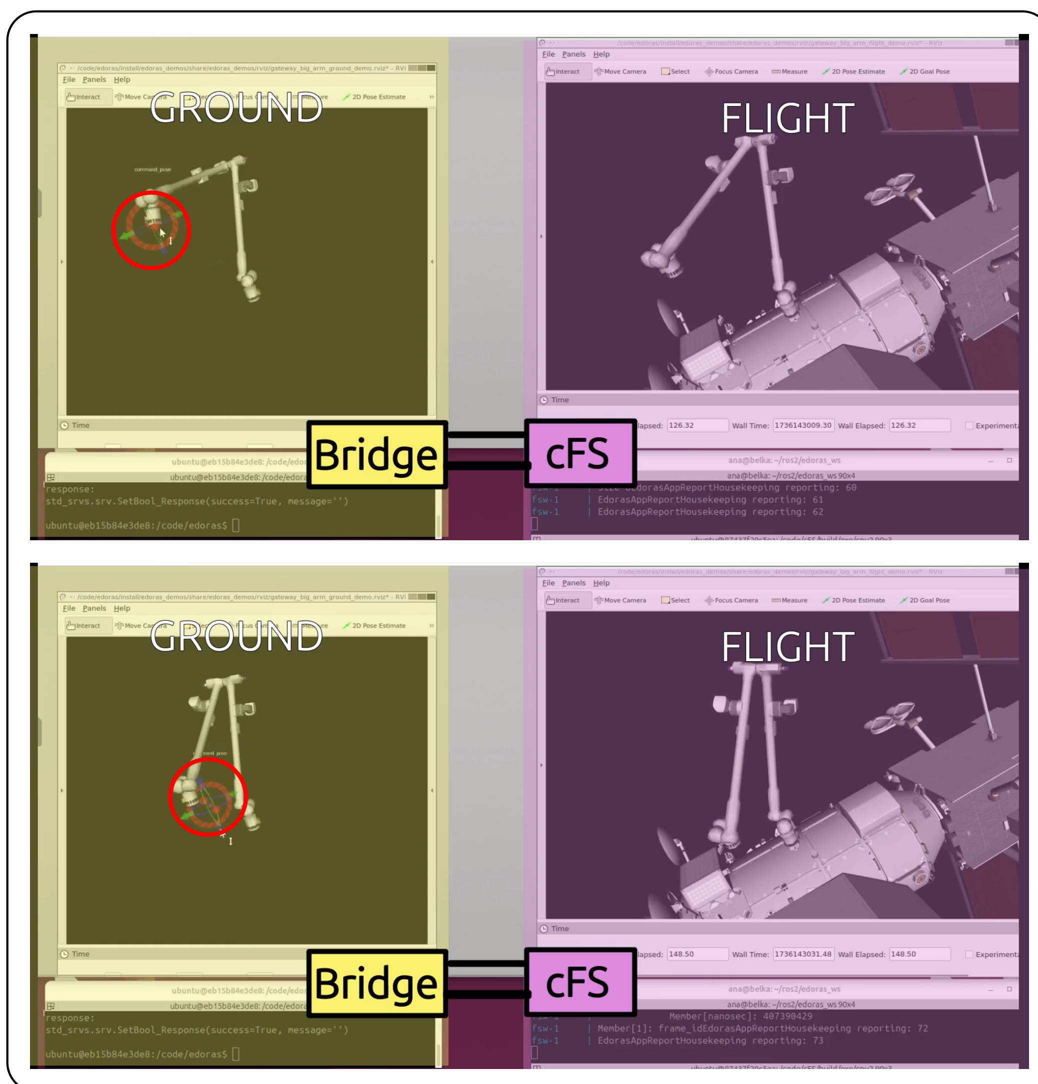



**Figure 21:** Demo: Controlling a 7-DOF arm using the ground bridge on the ground, cFS on the flight side. **Top:** Task state after commanding the arm to move such that its end effector reaches the pose where the 6D gimbal is (red circle at the left) **Bottom:** Task state after the operator sends a second pose command to the robot on the flight side.
**Video:** https://youtu.be/Bkz7cD1LA6w?si=XQwV8p9GxOpAz7U8

## 4.3 Control of two ROS2-enabled rovers in Gazebo using a pair of ground/flight bridges

This demo has the following characteristics:

- Demo is run using 3 Docker containers, one simulating the ground machine, the second one the spacecraft processor, and the third one simulates the processor onboard the rovers.
- The *ground* machine has ROS2 installed and it has a ground bridge running.

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

- The *spacecraft* machine has a cFS executable running. cFS is running with the default apps (CI/TO/SBN) as well as with a demo-specific application (`edoras_app`[9] )
- The *flight* machine is running ROS2 and a Gazebo simulation with the 2 rovers. In addition, this machine is also running a flight bridge.
- The ground machine and the spacecraft machine are only expected to communicate through the CI/TO cFS applications, using UDP. The spacecraft machine and the flight machines will communicate using the cFS software bus.

The demo, similar to its single-robot counterpart - consists on an operator publishing Twist commands (`geometry_msgs/msg/Twist`) in two ROS2 topics, one per each robot. The operator also subscribes to two ROS2 topics carrying each rover location (`geometry_msgs/msg/Pose`). Figure 22 shows a couple of snapshots of the demonstration.

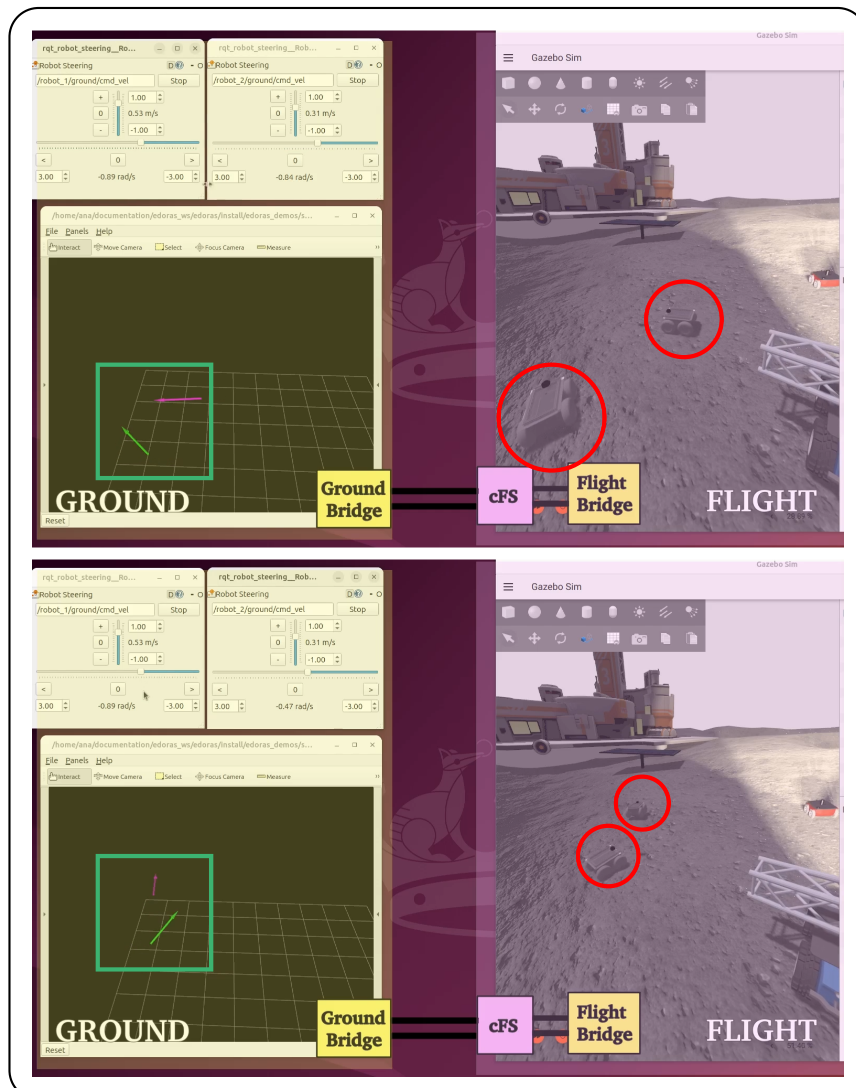



**Figure 22:** Demo: Controlling two rovers using the ground bridge on the ground, cFS on the spacecraft and a flight bridge on the robots' side. **Top:** Task state after commanding the robots. **Bottom:** Task state after sending different twist commands. The green and magenta arrows show the poses of the rovers being received back on the ground. **Video:** https://youtu.be/NfyMaSZ8ams?si=HWtlTkWEw54iO9DX

While testing this demostration, we observed some issues regarding frequency of message publication: The telemetry data from the robots (3D pose) is published by the Gazebo simulation, and it is then consumed first by the flight bridge, which then converts it to a CCSDS packet that is sent to cFS, which then sends it back to the ground, and finally this data is being republished back as a ROS2 topic. Gazebo by default runs at high frequencies (such as 250 Hz), hence the 3D pose being consumed by the flight bridge was being republished at very high speeds as well. We got a number of SBN error messages indicating that these speeds were too elevated.

For the purposes of this Phase I, we worked around this issue by manually reducing the frequency at which the corresponding Gazebo plugin publishes this data (30Hz) and this worked fine. Another alternative that we considered was to use the `throttle` utility, part of the `topic_tools` package [10]. While this was doable for these demonstrations, it is cumbersome having to add additional processes as intermediary steps between the application and the bridge. A simpler (for the user) approach would be to add a parameter in the flight bridge that limits the rate at which data is sent back to cFS, either as a global parameter for all ROS2 topics or as an individual (per-topic) value. We intend to explore this in our Phase II proposal.

[9] https://github.com/traclabs/edoras_app/tree/edoras_dual_small_rover
[10] http://wiki.ros.org/topic_tools/throttle

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

### 4.4 Control of two ROS2-enabled robot arms using a pair of ground/flight bridges

This demo has the following characteristics:

- Demo is run using 3 Docker containers, one simulating a ground machine, another simulates the spacecraft processor and the last one simulates the flight machine that interfaces with the robots using ROS2.
- The ground machine has ROS2 installed and it has a ground bridge running.
- The flight machine has a cFS executable running. Like in the previous case, cFS is running the default apps plus one task-specific application [11].
- The ground machine and the flight machine are only expected to communicate through the CI/TO cFS applications, using UDP.

The demo consists – similarly to its single-arm counterpart – on an operator publishing a pair of 3D Pose commands (`geometry_msgs/msg/Pose`) in two ROS2 topics, each of them namespaced for each arm (big_arm and little_arm, respectively). The operator on the ground also subscribes to two ROS2 topics that receive the arm's joint states (`sensor_msgs/msg/JointState`). Figure 23 shows two snapshots, each of them depicting each arm on the flight side moving after receiving a command from the ground.

While developing this demonstration, we only used a single flight bridge to link the cFS executable with the ROS2 topics on the flight side. By using namespaced topics, we were able to send/receive data from 2 different robots without issues. It should be possible, if needed, to create multiple flight bridges, one per each robot, although that would need some additional configuration in the cFS configuration files in order to add more SBN peers.

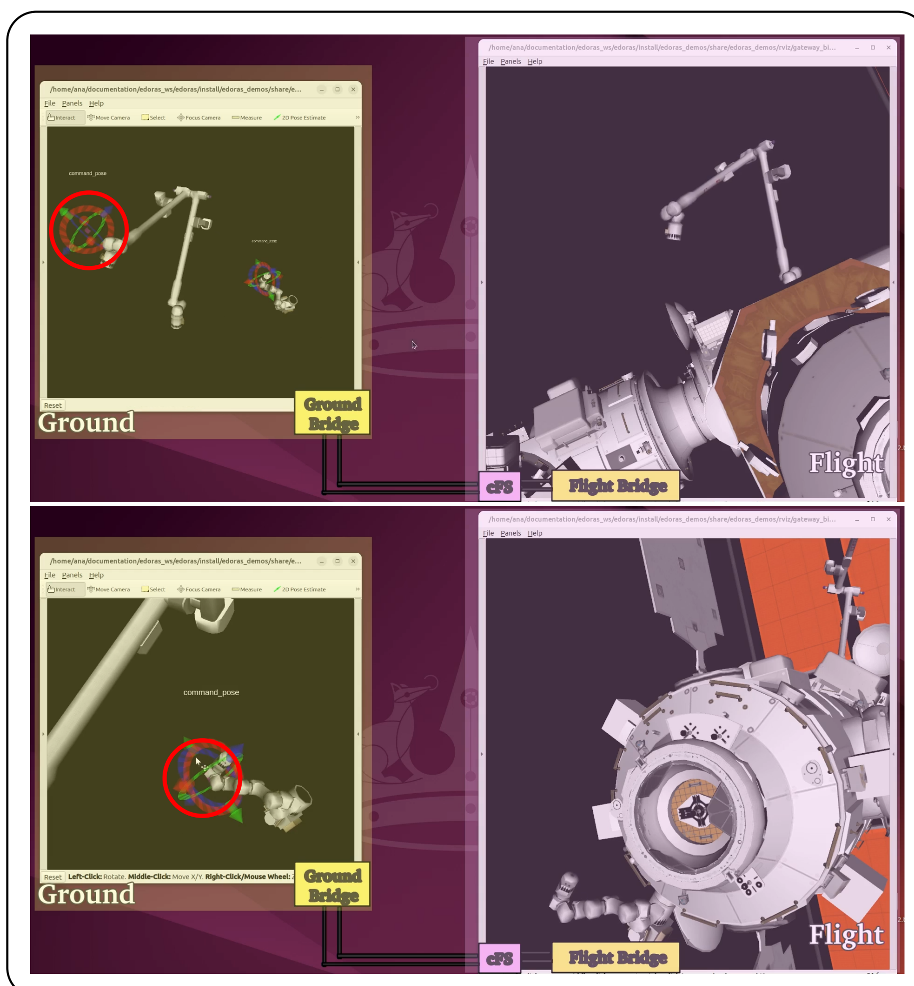



**Figure 23:** Demo: Controlling two 7-DOF arms using the ground bridge on the ground, cFS on the spacecraft processor and the robots being simulated on a flight machine. **Top:** Task state after commanding the big arm to move such that its end effector reaches the pose where the 6D gimbal is (red circle at the left) **Bottom:** Task state after the operator sends a second pose command to the small robot on the flight side.
**Video:** https://youtu.be/2KZ2urETX9Y?si=fKQUfyLizpDp3fiQ

## 5 Conclusion

In this paper we presented a set of tools we have developed to enable users to create hybrid systems that integrate flight and robotic software. Our tools allow the automatic mapping of ROS 2 messages and makes their information available to cFS applications. These tools were tested using a number of scenarios involving

[11] https://github.com/traclabs/edoras_app/tree/edoras_gateway_dual_arm

Copyright 2025 by TRACLabs. Published by the Canadian Space Agency (CSA) on behalf of SpaceOps, with permission and released to the CSA to publish in all forms.

single and multi-robot setups in simulation and running in a distributed manner using Docker containers. Our initial tests demonstrated that our tools were easily customizable to be used under these different conditions.

Several improvements can be made upon our current work. The serialization procedure used to encode the ROS 2 messages depends on a ROS 2-based serialization library, which - although effective - adds a dependency on ROS 2 that should not be necessary. As future work, we'd like to replace the currently used serialization library and instead develop our own tool that, in addition to serializing a message, also creates on the fly C classes that further facilitates the access to ROS 2 information in the cFS side.

# References


[1] NASA. *NASA's Plan for Sustained Lunar Exploration and Development*. https://www.nasa.gov/wp-content/uploads/2020/08/a_sustained_lunar_presence_nspc_report4220final.pdf. 2020.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. "ROS: an Open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[3] *2023 State-of-the-Art of Small Spacecraft Technology*. https://www.nasa.gov/smallsat-institute/sst-soa/small-spacecraft-avionics/#8.4. Accessed: 2024-03-09.

[4] D. McComas. "NASA/GSFC's Flight Software Core Flight System". In: *Flight Software Workshop*. GSFC. CPR. 7525.2013. 2012.

[5] *F': Flight Software & Embedded Systems Framework*. https://nasa.github.io/fprime/. Accessed: 2024-03-09.

[6] T. M. Ngo. "NASA class A certification of core flight software (cFS)". In: *Virtual 2021 Flight Software Workshop*. 2021.

[7] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[8] A. Probe, A. Oyake, S. W. Chambers, M. Deans, G. Brat, N. B. Cramer, B. Kempa, B. Roberts, and K. Hambuchen. "Space ROS: An open-source framework for space robotics and flight software". In: *AIAA SCITECH 2023 Forum*. 2023, p. 2709.

[9] T. Saito, H. Kato, D. Hirano, K. Iwabuchi, and S. Kawaguchi. "For the Use of ROS Applications on Spacecraft Systems". In: *The Proceedings of JSME annual Conference on Robotics and Mechatronics (Robomec)* 2018 (Dec. 2018).

[10] H. Kato, D. Hirano, S. Mitani, T. Saito, and S. Kawaguchi. "ROS and cFS system (RACS): Easing space robotic development". In: *2021 IEEE Aerospace Conference (50100)*. IEEE. 2021, pp. 1–8.

[11] J. Fugate. "Distributed Spacecraft Autonomy-Development of Swarm Autonomy Capability and Scalability for Spacecraft". In: *2021 JPL AI & Data Science Workshop*. 2021.

[12] A. Huamán. *The BRASH Integration Toolkit for ROS2 and Flight Software Interoperability*. STTR Phase II. NASA Contract # 80NSSC22CA020. TRACLabs Inc.

[13] S. Hart, T. Milam, A. Harris, and D. Edell. "Tools for ROS2 and Flight Software Interoperability". In: *Flight Software Workshop*. GSFC. CPR. 7525.2013. 2012.