User Guide for Delay/Disruption Tolerant Networking Bundle Protocol Node (BPNode) v7.0

NASA GSFC DTN Project

Version 1.0, August 2025

Table of Contents

Signatures

The signatories below indicate with their signature their commitment to the implementation of this Plan.

```
Prepared by:




_____________________________________ _______________
Sara Garcia-Beech                     Date
DTN FSW Development Team Lead
Flight Software Systems Branch
Code 582, NASA GSFC




_____________________________________ _______________
Rebecca Besser                        Date
DTN Systems Engineering Lead
Software Systems Engineering Branch
Code 581, NASA GSFC



Approved by:




_____________________________________ _______________
Peyush Jain                           Date
Flight Segment Mission Manager
DTN Study and Pathfinder Lead
TEMPO, Code 450.2, NASA GSFC




_____________________________________ _______________
Ben Anderson                          Date
DTN Enterprise Lead
TEMPO, Code 450.2, NASA GSFC
```

Change History

Version	Date	Description	Affected Pages
1.0	08/2025	Initial Release	All

Chapter 1. Introduction

The Bundle Protocol (BP) application node (BPNode) is a core Flight System (cFS) application that provides RFC-9171 BP services to embedded space flight systems by integrating BP library (BPLib) with cFS framework and processes. The Goddard Space Flight Center (GSFC) Delay/Disruption Tolerant Network (DTN) Project designed and developed BPNode as a generic flight-capable node compliant with Bundle Protocol version 7 (BPv7).

BPNode 7.0 was developed in accordance with the NPR 7150 Class B engineering requirements as defined in NASA Procedural Requirements 7123.1D and 7150.2D, respectively. It relies on a subset of cFS utilities (such as fault handling, task scheduling, data recording, etc.) to avoid duplication of code and leverage a NASA Class B/A software framework common to many human and robotic missions. This framework allows BPNode to run without modifications on various hardware and software configurations. Missions can tailor their operational environment based on available cFS configurations.

BPNode also complies with the Consultative Committee for Space Data Systems (CCSDS) standards for international interoperability and conforms with the LunaNet Interoperability Specification. BPNode enables cFS to communicate with other DTN nodes using Bundle Protocol.

1.1. Purpose and Scope

This BPNode v7.0 User Guide provides its users with general understanding of node functions. It covers node installation, configuration, and operations. The guide is intentionally generic to enable customization.

This software is available as an open source distribution, released under the Apache 2.0 license. The open-source license enables the industry adoption of BPNode. Outside of major releases, updates to the open repositories are limited to bug fixes and minor enhancements to these components.

1.2. Document Review, Approval, and Update

This guide is developed and maintained by the DTN Project at NASA Goddard Space Flight Center. The document has been reviewed and approved by the team members as listed on the signature page. After initial approval, the document will be treated as a controlled document, placed under configuration management. Changes are listed in the document *Change History* log.

1.3. Support

The development team also offers training, support, and subject matter expertise to government and industry users, and can provide custom development and integration services for flight projects. A support agreement may be required for certain engagements. To contact the team, email gsfc-dtn@nasa.onmicrosoft.com. Additional information is available on the DTN website: https://www.nasa.gov/communicating-with-missions/delay-disruption-tolerant-networking/.

1.4. Assumptions

The guide assumes that node operators are already familiar with BP and DTN, as well as with cFS Framework and how it integrates with COSMOS - the chosen telemetry and command system for interfacing with BPNode.

Chapter 2. Software Overview

BPNode is a DTN node that implements BP as defined in RFC 9171. The node software runs on Linux and real-time operating systems in a context where user applications communicate directly with BPNode. BPNode expects a Publish/Subscribe pattern where user applications perform as follows:

- publish Application Data Units (ADUs) to which BPNode subscribes
- subscribe to expected ADUs that BPNode publishes.

The entire system is composed of several modular components grouped by function. They include the core framework, mission-specific applications, supporting libraries, and development and integration tools.

2.1. Supported Platforms

The BPNode application has been built and tested on Ubuntu 20.04.6 Long-Term Support.

2.2. cFS Framework

cFS is a generic flight software architecture framework used on NASA spacecraft and cubesats. The cFS Framework is a core subset of cFS that offers an extensive ecosystem of applications and tools. The Figure 1, “Framework Structure” figure below demonstrates the external cFS and Core Flight Executive (cFE) framework structure. BPNode interacts with these elements to receive and distribute data for DTN service:

Operating System Abstraction Layer (OSAL)
: provides features on the operating system level, including persistent storage and mutexes.

Platform Support Package (PSP)
: interfaces with hardware-specific elements like memory and device I/O.

Software Bus
: passes the data to BPNode on the application layer. Applications use it to communicate with BPNode and to exchange telemetry, directives, and events.

Application
: sends and receives Application Data Units (ADUs) to/from BPNode via the framework software bus.

Performance Services
: interacts with **cFE Performance Log** to provide logging functionality for BPNode by sending entries through the framework.

Event Logging
: interacts with **cFE Event Services (EVS)** to collect events generated by BPNode and distribute them via Telemetry Distribution.

Time Services

interacts with **cFE TIME** to provide BPNode with time (host time, clock state, etc.) to calculate and maintain the time within the node.

Command Source

interacts with **cFS Command Ingest (CI)** or **Stored Command (SC)** to provide directives to BPNode, such as stored commands or commands received from an external source for BPNode execution.

Telemetry Distribution

interactions with **cFS Telemetry Output (TO)** to collect telemetry from BPNode and send it to external entities.

Table Services

interacts with **cFE Table Services** to provide a mechanism for table updates which are used by BPNode for configuration purposes.

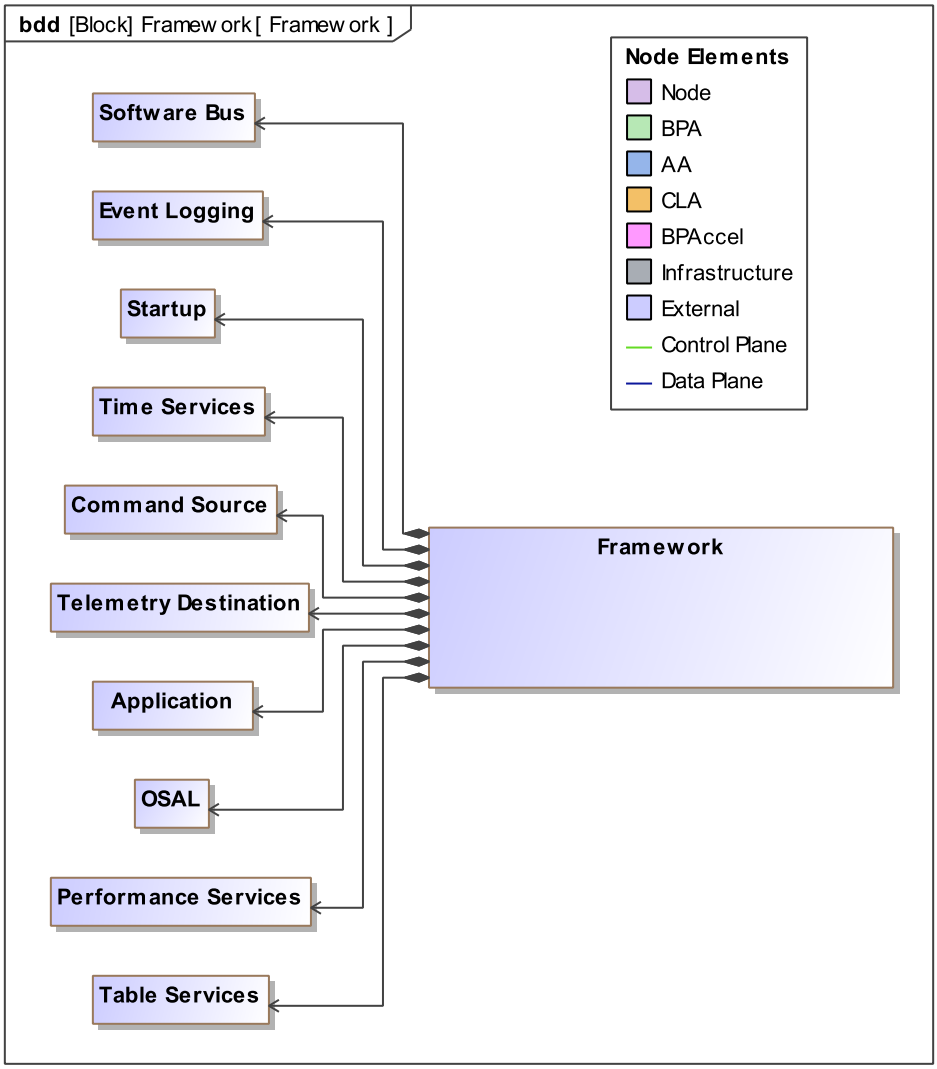

Figure 1. Framework Structure

See Chapter 6, *Directives, Telemetry, Tables, and Events* for the list of BPNode directives, tables, and telemetry.

2.3. BPNode and BPLib

BPNode and BPLib are complementary components that run on top of the framework:

BPLib
: serves as the core implementation library that provides BP functionality: https://github.com/nasa/bplib

BPNode
: integrates this library into the cFS framework and implements the BP node behavior: https://github.com/nasa/bp

Together they provide a shared library and support BP and related functions.

BPNode contains the following subsystems, modules, and components:

Application Agent (AA)
: provides application and administrative services associated with monitoring and control, such as status monitoring or configuration control:

- Admin Statistics
- Administrative Record Processor
- Framework Proxy
- Node Configuration

BP Agent (BPA)
: executes BP-related activities and functions:

- Bundle Interface
- Custody Transfer
- Extension Block Processor
- Payload Interface
- Policy Database
- Storage

Core Infrastructure (CI)
: functions as a common utility across modules for low-level functions and processing, such as memory, time, and events:

- Concise Binary Object Representation (CBOR)
- Cyclic Redundancy Check (CRC)
- Endpoint ID (Endpoint)
- Event Management
- Memory Allocator

- Performance Logger
- Queue Management
- Time Management

Convergence Layer Adapters (CLAs)
: interfaces with external Convergence Layers (CLs) such as UDP to send and receive bundles on behalf of the BPA

The following features only contain skeleton code in build 7.0 and are slated for future implementation:

- Administrative Record Processor
- Custody Transfer
- Policy Database

2.4. Dependencies

For both BPNode and BPLib, the following dependencies are required:

CMake (build system)
: version 3.22.1 used during testing

pkg-config
: version 0.29.1 used during testing

Gnu Compiler Collection toolchain
: version 9.4.0 used during testing

sqlite3
: version 3.31.1 used during testing

On Ubuntu 20.04.6, install the necessary packages with:

```
sudo apt-get install cmake pkg-config build-essential libsqlite3-dev
```

2.5. QCBOR

BPLib relies on CBOR to handle the serialization and deserialization of data structures transmitted over the protocol. BPLib uses version 1.5.1 of QCBOR for CBOR data encoding and decoding.

Follow these steps to install QCBOR:

1. Clone the QCBOR repository:

```
cd ~/ # Or any directory for installing QCBOR
git clone https://github.com/laurencelundblade/QCBOR.git
```

2. Navigate to the QCBOR repository:

```
cd QCBOR
```

3. Select the correct version:

```
git checkout v1.5.1
```

4. Build and install the library:

```
sudo make uninstall # Remove any previous installation
cmake -DBUILD_SHARED_LIBS=ON -S . -B build
cmake --build build
sudo make install
```

Chapter 3. Installation

3.1. Building BPNode and BPLib within cFS

3.1.1. Clone Repositories

Clone the cFS repository to the local system:

```
git clone https://github.com/nasa/cFS.git
```

Follow the instructions in the cFS README to initialize the sample cFS deployment: https://github.com/nasa/cfs?tab=readme-ov-file#build-and-run

cFS relies heavily on Git submodules, so to avoid errors during when updating code, they must be kept updated by running `git submodule update --init`.

Clone the BPNode and BPLib repos as submodules in the cFS bundle.

```
cd cFS/apps/
git submodule add https://github.com/nasa/bp.git bpnode
cd ../../libs/
git submodule add https://github.com/nasa/bplib.git
```

3.1.2. Add BPNode and BPLib to cFS

To add BPNode and BPLib to the cFS build system, modify sample_defs/targets.cmake to include them in the MISSION_GLOBAL_APPLIST variable.

```
LIST(APPEND MISSION_GLOBAL_APPLIST bplib bpnode)
```

BPNode has a PSP module for UDP socket interfacing called udpsock_intf that also needs to be added to the build system. Add the following to sample_defs/mission_build_custom.cmake.

```
set(MISSION_MODULE_SEARCH_PATH
    "apps"                  # general purpose ${top}/apps directory
    "libs"                  # general purpose ${top}/libs directory
    "psp/fsw/modules"       # modules for optional platform abstraction, associated
with PSP
    "cfe/modules"           # modules for optional core functions, associated with CFE
    "apps/bpnode/psp_mod"   # Custom PSP modules for BPNode, including udpsock_intf
)
```

To ensure that BPNode and BPLib are loaded automatically upon cFE startup, add them to

sample_defs/cpu1_cfe_es_startup.scr.

```
CFE_LIB, bplib,         BPLib_PrintVersion, BP_LIB,       0,    0,     0x0, 0;
CFE_APP, bpnode,        BPNode_AppMain,     BPNODE,       55,   16384, 0x0, 0;
```

To add BPNode wakeup messages at 10Hz and telemetry requests at 0.25Hz, the following sch_lab_table.c file will be needed.

```
#include "bpnode_msgids.h"

SCH_LAB_ScheduleTable_t SCH_LAB_ScheduleTable = {
            .TickRate = 10,
            .Config   = {
                {CFE_SB_MSGID_WRAP_VALUE(CFE_ES_SEND_HK_MID), 40, 0},
                {CFE_SB_MSGID_WRAP_VALUE(CFE_EVS_SEND_HK_MID), 40, 0},
                {CFE_SB_MSGID_WRAP_VALUE(CFE_TIME_SEND_HK_MID), 40, 0},
                {CFE_SB_MSGID_WRAP_VALUE(CFE_SB_SEND_HK_MID), 40, 0},
                {CFE_SB_MSGID_WRAP_VALUE(CFE_TBL_SEND_HK_MID), 40, 0},
                {CFE_SB_MSGID_WRAP_VALUE(CI_LAB_SEND_HK_MID), 40, 0},
                {CFE_SB_MSGID_WRAP_VALUE(TO_LAB_SEND_HK_MID), 40, 0},
#ifdef HAVE_SAMPLE_APP
                {CFE_SB_MSGID_WRAP_VALUE(SAMPLE_APP_SEND_HK_MID), 40, 0},
#endif
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 40}, /*
BPNODE_SEND_NODE_MIB_CONFIG_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 41}, /*
BPNODE_SEND_SOURCE_MIB_CONFIG_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 42}, /*
BPNODE_SEND_NODE_MIB_COUNTERS_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 43}, /*
BPNODE_SEND_SOURCE_MIB_COUNTERS_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 44}, /*
BPNODE_SEND_STORAGE_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 45}, /*
BPNODE_SEND_CHANNEL_CONTACT_STAT_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_CMD_MID),       40, 46}, /*
BPNODE_SEND_NODE_MIB_REPORTS_HK_CC */
                {CFE_SB_MSGID_WRAP_VALUE(BPNODE_WAKEUP_MID),    1, 0},
}};
```

The telemetry request rate can be customized entirely to a project's needs. The wakeup rate is recommended to be set to 10Hz and should not be set to slower than 1Hz. See BPNODE_MAX_EXP_WAKEUP_RATE in bpnode/config/default_bpnode_platform_cfg.h for more details.

To ensure BPNode telemetry packets are egressed by to_lab, the following configurations are needed in the to_lab_sub.c file:

```
#include "bpnode_msgids.h"

TO_LAB_Subs_t TO_LAB_Subs = {.Subs = {/* CFS App Subscriptions */
                                      {CFE_SB_MSGID_WRAP_VALUE(TO_LAB_HK_TLM_MID), {0,
0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(TO_LAB_DATA_TYPES_MID),
{0, 0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CI_LAB_HK_TLM_MID), {0,
0}, 4},
#ifdef HAVE_SAMPLE_APP
                                      {CFE_SB_MSGID_WRAP_VALUE(SAMPLE_APP_HK_TLM_MID),
{0, 0}, 4},
#endif
                                      /* cFE Core subscriptions */
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_ES_HK_TLM_MID), {0,
0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_EVS_HK_TLM_MID),
{0, 0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_SB_HK_TLM_MID), {0,
0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_TBL_HK_TLM_MID),
{0, 0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_TIME_HK_TLM_MID),
{0, 0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_TIME_DIAG_TLM_MID),
{0, 0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_SB_STATS_TLM_MID),
{0, 0}, 4},
                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_TBL_REG_TLM_MID),
{0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(CFE_EVS_LONG_EVENT_MSG_MID), {0, 0}, 32},

                                      {CFE_SB_MSGID_WRAP_VALUE(CFE_ES_APP_TLM_MID),
{0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(CFE_ES_MEMSTATS_TLM_MID), {0, 0}, 4},


{CFE_SB_MSGID_WRAP_VALUE(BPNODE_NODE_MIB_CONFIG_HK_TLM_MID), {0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(BPNODE_SOURCE_MIB_CONFIG_HK_TLM_MID), {0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(BPNODE_NODE_MIB_COUNTERS_HK_TLM_MID), {0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(BPNODE_NODE_MIB_REPORTS_HK_TLM_MID), {0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(BPNODE_SOURCE_MIB_COUNTERS_HK_TLM_MID), {0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(BPNODE_STORAGE_HK_TLM_MID), {0, 0}, 4},
```

```
{CFE_SB_MSGID_WRAP_VALUE(BPNODE_CHANNEL_CONTACT_STAT_HK_TLM_MID), {0, 0}, 4},

{CFE_SB_MSGID_WRAP_VALUE(BPNODE_ADU_OUT_SEND_TO_MID), {0, 0}, 4},

                                        {CFE_SB_MSGID_RESERVED, {0, 0}, 0}}};
```

3.1.3. Building and Deploying the DTN cFS

This method is ideal for simulation, unit testing, and general development work on a local Linux machine. Use the following commands from the top-level `cFS` directory:

```
make distclean
make SIMULATION=native prep
make
make install
cd build/exe/cpu1/
./core-cpu1
```

This compiles and launches the `cpu1` target in a native environment.

3.1.4. Run Unit Tests and Generate Coverage

Testing is a key part of validating the DTN cFS system. The sequence below starts by cleaning and configuring the native build environment to include unit test support, followed by compilation, testing, and coverage reporting.

```
make distclean
make SIMULATION=native ENABLE_UNIT_TESTS=true prep
make
make test
make lcov
```

The resulting report can be used to assess code coverage and identify gaps in test validation.

Always run `make distclean` before switching targets or simulation environments to avoid build conflicts. For advanced build configurations or debugging, consult the Makefiles and scripts included in the repository.

3.1.5. Troubleshooting

This section addresses common issues that may arise when working with the DTN cFS bundle.

Missing files or directories after cloning

Run `git submodule update --init` to ensure all nested submodules are pulled.

Unit tests not executing

Make sure to include `ENABLE_UNIT_TESTS=true` in the `make prep` step.

3.2. Standalone BPLib Build (bpcat)

BPLib can be built as a standalone executable called `bpcat`, suitable for a ground relay node or as a demo of bplib's features without the full CS architecture. This standalone build still depends on the OSAL library.

Note that `bpcat` does not implement the full suite of BPNode features, since the node relies on cFS for directive handling, telemetry routing, table updates, etc.

Build/run instructions for `bpcat`:

1. Clone the repositories

```
cd <chosen working directory>
export BPLIB_HOME="$(pwd)"  # Set working directory
git clone https://github.com/nasa/bplib "${BPLIB_HOME}"/bplib
git clone https://github.com/nasa/osal "${BPLIB_HOME}"/bplib/osal
```

2. Configure the build environment

```
# Choose a build configuration - Debug or Release
export MATRIX_BUILD_TYPE=Debug

# Indicate where the OSAL build files are
export NasaOsal_DIR="${BPLIB_HOME}/bplib/osal-staging/usr/local/lib/cmake"

# Define bplib build directory
export BPLIB_BUILD="${BPLIB_HOME}/bplib/bplib-build-matrix-${MATRIX_BUILD_TYPE}-
POSIX"
```

3. Build BPLib with OSAL

```
cd $BPLIB_HOME/bplib/osal
cmake -DCMAKE_INSTALL_PREFIX=/usr/local \
      -DOSAL_SYSTEM_BSPTYPE=generic-linux \
      -DCMAKE_BUILD_TYPE="${MATRIX_BUILD_TYPE}" \
      -DOSAL_OMIT_DEPRECATED=TRUE \
      -DENABLE_UNIT_TESTS=TRUE \
      -DOSAL_CONFIG_DEBUG_PERMISSIVE_MODE=ON \
      -B ../osal-build
cd ../osal-build
make DESTDIR=../osal-staging install
cd ..
cmake -DCMAKE_VERBOSE_MAKEFILE=ON \
      -DCMAKE_BUILD_TYPE="${MATRIX_BUILD_TYPE}" \
      -DBPLIB_OS_LAYER=OSAL \
      -DCMAKE_PREFIX_PATH=/usr/local/lib/cmake \
      -B "${BPLIB_BUILD}"
cd $BPLIB_BUILD
make all
```

4. Test and execute
   - To run the bplib unit tests:

     ```
     cd $BPLIB_BUILD
     ctest --output-on-failure 2>&1 | tee ctest.log
     ```

   - To run bpcat as a standalone:

     ```
     cd $BPLIB_BUILD
     cd app
     ./bpcat
     ```

In the default bpcat configurations, a single contact is started that listens on 0.0.0.0:4501 and forwards bundles with the EID ipn:200.64 to 127.0.0.1:4551. To change these configurations, modify the ContactsTbl in bplib/app/src/bpcat_nc.c and rebuild bplib.

Chapter 4. Configuration

4.1. Startup Configuration

On startup, the BPNode main task will initialize itself, the BPLib library data, and any child tasks. On first initialization, the node creates a SQLite database named `bplib-storage.db` in the same directory as the executable (typically `build/exe/cpu1/`) along with temporary files associated with it. On subsequent initializations BPLib reads from it and reports the total number of bundles stored (see the Node MIB Reports telemetry packet as described in Chapter 6, *Directives, Telemetry, Tables, and Events*).

The node's Endpoint Identifier (EID) is established during initialization by reading from the node MIB configuration table. This table contains an "instance EID" field that provides the base EID value used as the node's primary identity. Channels use this EID for bundles originating on local applications, modifying the service number to match the corresponding value from the channel configuration table.

See Chapter 6, *Directives, Telemetry, Tables, and Events* for more information.

The counts of tasks are determined by the following BPNode or BPLib macros:

BPNODE_NUM_GEN_WRKR_TASKS
: Generic Worker

BPLIB_MAX_NUM_CHANNELS
: ADU In/Out

BPLIB_MAX_NUM_CONTACTS
: CLA In/Out

4.1.1. ADU Channels

ADU tasks are implemented via virtual communication pathways for application data called channels. Each channel has a unique ID from 0 to `BPLIB_MAX_NUM_CHANNELS` - 1. All channels default to `REMOVED` state at initialization unless their configuration in the channel configuration table has `loadAutomatically` set to `true`, in which case they should initialize as `STARTED`. Channel ID indexes into:

- The ADU proxy table
  - This table has any cFS-specific channel information, including which message IDs a channel should subscribe to
- The channel configuration table
  - This table has general channel configurations, including the service number to assign to the channel, and how to configure new bundles created from this channel's ADUs
- The channel status array in the channel/contact status telemetry packet

See Chapter 6, *Directives, Telemetry, Tables, and Events* for more information on BPNode tables.

4.1.2. CLA Contacts

CLA tasks are implemented via network interface connections called contacts: Each contact has a unique ID from 0 to `BPLIB_MAX_NUM_CONTACTS` - 1. All contacts default to `TORNDOWN` state at initialization. Contact ID indexes into:

- The contact configuration table
  - This table has information including which destination EID(s) to associate with this contact and the needed IP information to set up a UDP contact
- The contact status array in the channel/contact status telemetry packet

Chapter 5. Running the Node

Main configurations that drive node performance:

Rate limits

see the channel and contact configuration tables (defaults located at `bpnode/fsw/tables/bpnode_contacts_tbl.c` and bpnode/fsw/tables/bpnode_channel_config_tbl.c`)

Task priorities

see `bpnode/config/default_bpnode_platform_cfg.h` and the cFE startup script

Memory pool size

see BPNODE_MEM_POOL_LEN in `bpnode/config/default_bpnode_platform_cfg.h`

Number of tasks in the system

based on BPLIB_MAX_NUM_CONTACTS, BPLIB_MAX_NUM_CHANNELS, and BPNODE_NUM_GEN_WRKR_TASKS in `bplib/inc/bplib-cfg.h` and `bpnode/config/default_bpnode_platform_cfg.h`

The maximum bundle bytes allowed in the storage database

based on BPLIB_MAX_STORED_BUNDLE_BYTES in `bplib/inc/bplib_cfg.h`

The batch size of bundles

loaded out of storage

BPLIB_STOR_LOADBATCHSIZE

loaded into storage

BPLIB_STOR_INSERTBATCHSIZE

discarded from storage

BPLIB_STOR_DISCARDBATCHSIZE

Note that configurations can affect each other. The storage bundle byte limit and the memory pool size should be at least as large as the maximum bundle size times the relevant load batch sizes into or out of storage. Otherwise error event messages will start to be regularly issued when either the storage limit or the memory limit is reached.

Configurations that affect how bundles move between nodes and how bundles are created/processed are primarily found in the ADU proxy, channel configuration, and contact configuration tables. Multiple BPNode instances can be deployed in a network as source, relay, or destination nodes, as shown in Figure 2, “BPNode deployment roles” below:

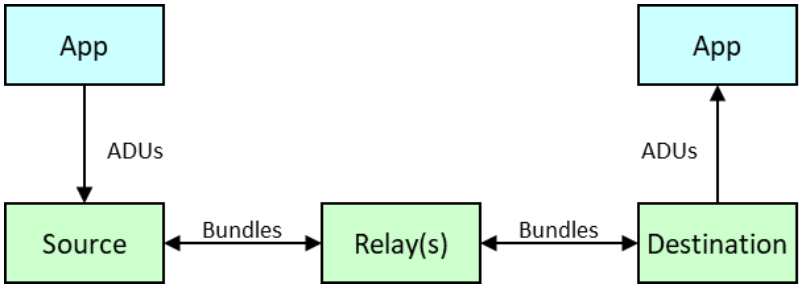

Figure 2. BPNode deployment roles

To deploy multiple BPNode instances, follow the standard cFS instructions for deploying multiple CPU instances (see here) and modify the BPNode tables for each CPU instance accordingly.

5.1. Managing Channel Applications

BPNode manages channel applications by separating configuration from activation, allowing for complete lifecycle control from setup through teardown. When an application needs to be integrated into the network, it must first be added to make its configuration available, then explicitly started to begin data flow. Applications can later be stopped to halt data processing or completely removed, which flushes any remaining data from both the egress queue and the cFE software bus pipe. These application lifecycle operations are controlled through the following directives:

`add-application`
: this directive loads an application's configuration by reading settings from both the ADU proxy table and channel configuration table using the specified channel ID. However, this step only makes the configuration available - no actual data processing begins until the application is started.

`start-application`
: this directive activates the configured application by enabling its ADU tasks to begin processing data flow. In the cFS environment, this activation involves subscribing to the relevant ADU message IDs and establishing the connection to post delivered bundle payloads to SB.

`stop-application`
: this directive stops the flow of ADUs in and out of the specified channel.

`remove-application`
: this directive flushes the egress queue of the specified channel and store any bundles found in there. It will also flush the cFE SB pipe of any new ADUs.

See Chapter 6, *Directives, Telemetry, Tables, and Events* for the list of BPNode directives.

Once active, applications participate in BP communication through two primary data flows:

ADU In
: this task handles data originating from local applications by converting received cFS packets into bundle payloads, applying the settings specified in the channel configuration table to create properly formatted bundles for network transmission.

ADU Out

this task handles inbound data bundles arriving from the network that are destined for local applications. When a bundle's destination EID matches an application running on the current node, the bundle is routed to the appropriate task, which extracts the payload data and posts it to the cFS SB where the target application can receive it.

This architecture allows BPNode to serve as a bridge between local cFS applications and the broader BP network, handling the translation between application-specific data formats and standardized bundle protocols.

5.2. Managing Contacts

BPNode manages network connections through contacts using a comprehensive lifecycle process similar to application management, allowing for complete control from initial setup through final teardown. Contacts represent network connections to other BP nodes and must be configured before they can actively participate in bundle routing. Like applications, contacts can be stopped to halt bundle traffic or completely torn down, which flushes any remaining bundles from the egress queue into storage. These contact lifecycle operations are controlled through the following directives:

`contact-setup`

this directive establishes a contact's network configuration by reading settings from the contact configuration table using the specified contact ID. This step creates the underlying CL connection infrastructure but leaves it in a dormant state with no bundle traffic flows until the contact is explicitly started.

`contact-start`

this directive activates the configured contact by enabling its CLA tasks to begin processing bundle traffic. Once started, the contact can participate in both receiving bundles from and transmitting bundles to other nodes in the BP network.

`contact-stop`

this directive stops the flow of bundles in and out of the specified contact.

`contact-teardown`

this directive flushes the egress queue of the specified contact and stores any bundles found in there.

See Chapter 6, *Directives, Telemetry, Tables, and Events* for the list of BPNode directives.

Active contacts handle BP communication through two distinct pathways:

CLA In

this task receives encoded bundles from incoming network traffic, decodes the CBOR formatting, and deserialize the data into a readable bundle format. Each received bundle undergoes validation to ensure data integrity. Bundles that fail validation are immediately discarded to prevent corrupt data from entering the system.

CLA Out

this tasks manages the outgoing traffic by routing bundles destined for remote nodes based on their destination EIDs. When a bundle's destination corresponds to a destination EID in the started contact, it is forwarded to the appropriate task, which encodes the bundle using CBOR formatting and transmits it to the next node in the network routing path.

This contact management system enables BPNode to maintain reliable connections with multiple network peers while ensuring data integrity through validation and proper encoding protocols.

5.3. Bundle Storage Management

When BPNode receives a bundle that cannot be immediately processed—either because its destination EID does not match any active local applications (channels) or available network connections (contacts), the system provides persistent storage to ensure reliable message delivery even across system restarts. Such bundles are written to the SQLite database `bplib-storage.db`, which provides persistent storage that survives node restarts, ensuring that bundles waiting for delivery or forwarding opportunities are not lost during system maintenance or unexpected shutdowns.

The main task continuously monitors stored bundles to manage their lifecycle effectively. It regularly checks each bundle's creation timestamp against its specified lifetime to identify expired bundles, which are automatically discarded to prevent indefinite storage of undeliverable messages. Simultaneously, the system monitors for new routing opportunities by checking whether previously unroutable bundles can now be forwarded to newly available contacts or delivered to recently started local applications. When such opportunities arise, bundles are moved from storage to the appropriate egress queue and removed from the database.

The system provides telemetry reporting on storage utilization through the storage telemetry packet, which includes both the total size of the `bplib-storage.db` file and the actual space consumed by stored bundles. The bundle storage size is typically smaller than the total database file size due to SQLite's internal database management overhead. Storage capacity is controlled by the `BPLIB_MAX_STORED_BUNDLE_BYTES` macro, which sets limits based on the actual byte size of bundle content rather than total database file size.

Chapter 6. Directives, Telemetry, Tables, and Events

For the list and descriptions of BPNode directives, telemetry, tables, and events, consult the Monitor & Control Interface Control Document (M&C ICD) in the BPNode repository.

Chapter 7. Known Issues

1. The `maxBundlePayloadSize` configuration in the channel configuration table conflicts with a similar configuration in the MIB node configuration table. Currently the value in the channel configuration table is used, but eventually the one in the MIB configuration will be used.
2. If the general rate of ingressing data is greater than the general rate of egressing data, memory will start to run out and ingressing bundles will start to be dropped.
3. If bundles start to expire from storage at the same time as they start to be egressed from storage, the `BUNDLE_COUNT_LIFETIME_EXPIRED` telemetry counter might erroneously count bundles that have been egressed as expired.
4. Large volumes of bundles in storage result in a general system slowdown and occasional spikes in egress rates beyond what the rate limit has been configured to allow.
5. Rate limits where the bits per wakeup is set to a value smaller than the expected bundle size will result in unusual behavior.
6. Storage database sizes larger than 4GB result in unexpected errors
7. Expiration of bundles in storage does not work when bundles have an age block but no valid creation timestamp.

For a full list of issues, see the BPNode 7.0 Version Description Document. These issues are all slated for fixing in build 7.1.

Chapter 8. DTN Tools Suite

Tools Suite is a toolkit that allows for:

- confirming whether bundles created by the implementation adhere to specifications such as RFC 9171, ensuring protocol compliance at a fundamental level
- rigorous testing of how the implementation handles invalid or off-nominal bundles
- verifying implementation's resistance to corrupted data

The toolkit provides comprehensive end-to-end tests within a single script for a consistent methodology for creating bundle content, covering the entire process from bundle creation through transmission to a DTN node, reception from a DTN node, and verification of received bundles.

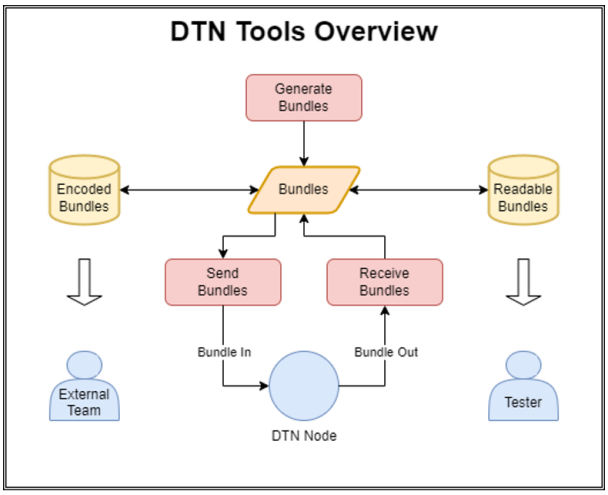

Figure 3. Tools Overview

It includes support for extension blocks like Custody Transfer Extension Block (CTEB) and Compressed Reporting Extension Block (CREB) as well as CBOR-encoded and decoded formats.

The suite contains two tools: DTNGEN and DTNCLA.

8.1. DTNGEN

DTNGEN is a Python library that enables the creation and interpretation of DTN bundles for compliance verification as well as error handling and interoperability testing between different DTN implementations. It can create both standards-compliant bundles and deliberately invalid bundles with incorrect data or block ordering.

The tool provides bundle manipulation capabilities through dedicated classes for primary blocks and canonical block types. It can generate:

- individual bundles with customizable components (primary, payload, and extension blocks)
- sets of bundles using:
  - script-based loops
  - specialized Bundle Set Generation feature, which allows for controlled variations in creation timestamps and payload sizes.

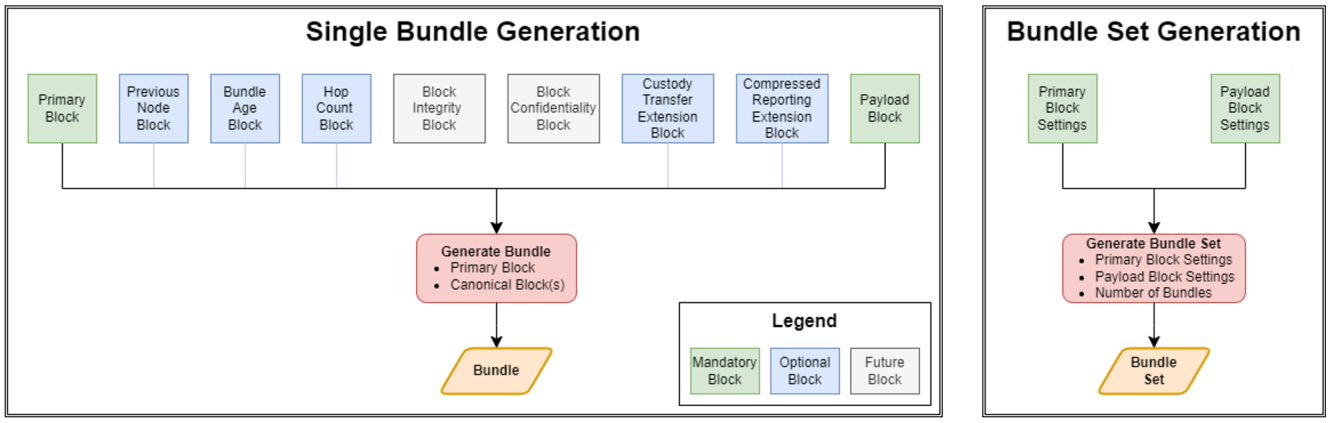

Figure 4. Single Bundle and Bundle Set Generation

DTNGEN offers flexible data format support, allowing bundles to be read from and written to both binary and JSON formats:

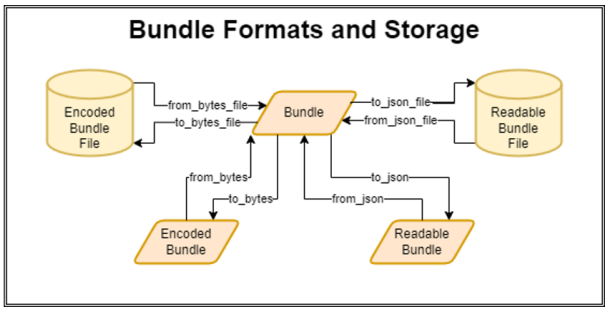

Figure 5. Conversion between CBOR and JSON

```
{
  "block_class": "PrimaryBlock",
  "version": 8,
  "control_flags": 4,
  "crc_type": 1,
  "dest_eid": {
    "type": "EID",
    "value": {
      "uri": 2,
      "ssp": {
        "node_num": 103,
        "service_num": 1
      }
    }
  },
  "src_eid": {
    "type": "EID",
    "value": {
      "uri": 2,
      "ssp": {
        "node_num": 101,
        "service_num": 1
      }
    }
  },
  "rpt_eid": {
    "type": "EID",
    "value": {
      "uri": 2,
      "ssp": {
        "node_num": 100,
        "service_num": 1
      }
    }
  },
  "creation_timestamp": {
    "type": "CreationTimestamp",
    "value": {
      "time": 755533883904,
      "sequence": 0
    }
  },
  "lifetime": 3600000,
  "crc": {
    "type": "hexbytes",
    "value": "af7c"
  }
}
```

8.2. DTNCLA

DTNCLA is a unified Python library that implements CLAs needed for DTN testing. It provides a consistent interface for communicating with DTN nodes, regardless of the underlying convergence layer protocol.

The library follows a modular design where most functionality is shared across different CLAs, promoting code reuse and consistent behavior. Only the network read/write functions are specialized, with implementations tailored to the specific requirements of each convergence layer protocol.

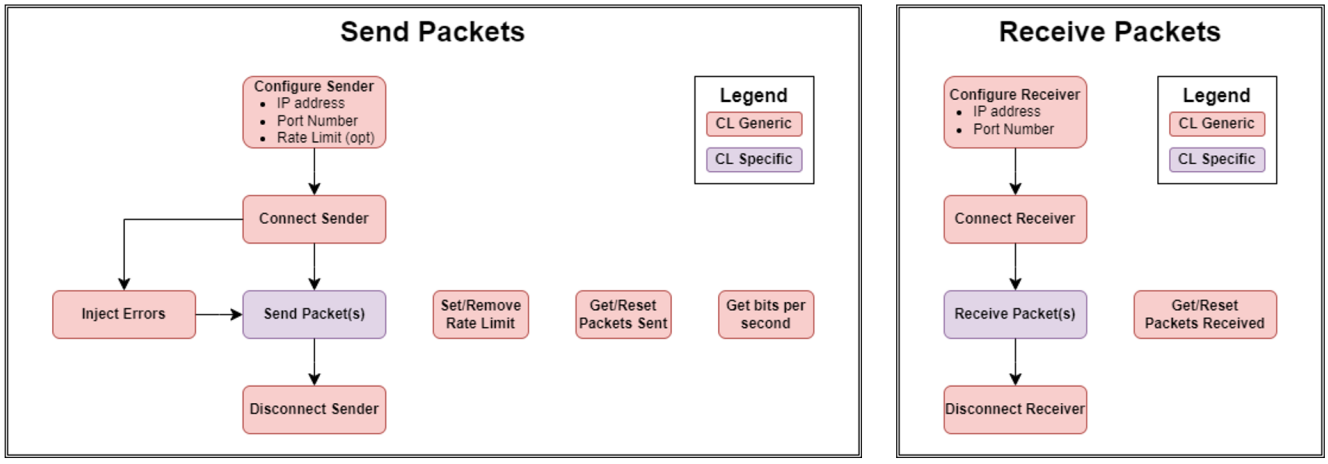

Figure 6. Send and Receive Packets

8.3. Dependencies

The suite requires the following dependencies:

- Python v3.8 or higher
- cbor2 v5.6.2 or higher
- crccheck v1.3.0 or higher

8.4. Tools Suite Installation

Each DTN Tools Suite release provides a complete package containing a built Python wheel file (.whl) for easy installation, source code, release notes, and documentation in HTML and PDF formats that covers all classes, variables, and usage examples that can be copied directly into test scripts.

To install for command line usage:

```
pip install dtntools-X.X.X-py3-none-any.whl
```

The source code is organized into four main folders:

dtntools

Contains the Python source code for DTNGEN and DTNCLA

docs

Houses all HTML documentation

tests

Contains scripts for testing DTN Tools, including unit tests and tests against existing DTN implementations

examples

Includes demonstration scripts for DTN Node testing:

bundle-generation

Shows how to generate bundles, send them to a DTN Node, receive them back, and verify content

multiple-contacts

Demonstrates using Python "subprocess" to simulate multiple concurrent contacts

performance-test

Illustrates creating and exchanging large bundle sets with a DTN node at configured bit rates

8.5. COSMOS Integration

The DTN Tools Suite seamlessly integrates with OpenC3 COSMOS version 5.12 and later, which supports native Python package installation. DTN Tools packages can be installed directly through the COSMOS Administrator Console, with required dependencies (cbor2 and crccheck) being automatically resolved and installed during the process.

When configuring test environments, any host ports or folders accessed by the DTN test scripts must be explicitly defined in the COSMOS compose.yaml file to ensure proper container communication. The integration enables calling DTN Tools functions directly from COSMOS script-runner, facilitating their incorporation into scripts and procedures.

```
 96 try:
 97     data_receiver.connect()
 98     data_sender.connect()
 99
100     for x in (bundle_data):
101         data_sender.write(x)
102
103     for x in range(1, len(bundle_data)+1):
104         received_bundle = data_receiver.read()
105         decoded_bundle = Bundle.from_bytes(received_bundle)
106         if decoded_bundle:
107             print(decoded_bundle.to_json())
108             decoded_bundle.to_json_file(f"/bundles/received_bundle_{x}.json")
109
110     print(f'Packets sent = {data_sender.get_packets_sent()}')
111     print(f'Packets received = {data_receiver.get_packets_received()}')
112
```

Figure 7. Bundle transmission and reception integrated into COSMOS script-runner

Within the COSMOS environment, some methods differ from standard Python constructs, e.g.: COSMOS uses "ask" rather than "input" for user prompts.

Appendix A: COSMOS Interface

COSMOS is a suite of software application for integration, testing, and operations of embedded systems that can range from power supplies to satellite constellations. BPNode has been tested against OpenC3 COSMOS versions 5.17.1 and 6.3.0. All files needed to run BPNode against COSMOS are provided in the bpnode/openc3-cosmos-dtnfsw directory.

To run COSMOS with the openc3-cosmos-dtnfsw plugin, first make the following modifications to COSMOS's compose.yaml file:

- If you are running to_lab, add "1235:1235/udp" to the ports list under the openc3-operator section

```
openc3-operator:
  ports:
    - "1235:1235/udp"
```

- If you are using one or more UDP configurations for the CLA tasks and wish to use the test tools in COSMOS to receive bundles, the CLA out port(s) need to be listed in the ports list under the openc3-cosmos-script-runner-api section For example, if you are setting the CLA Out port for contact 0 to 4551 (as in the default configurations), your compose.yaml changes should look like this:

```
openc3-cosmos-script-runner-api:
  ports:
    - "4551:4551/udp"
```

To build the openc3-dtnfsw-plugin, run the following command from within the `bpnode/cosmos` directory:

```
<Path to COSMOS installation>/openc3.sh cli rake build VERSION=7.0.0
```

COSMOS has a comprehensive user's guide with a dedicated section for cFS integration, maintained by the OpenC3 team, so further information about the installation and operation of COSMOS is outside the scope of this document.

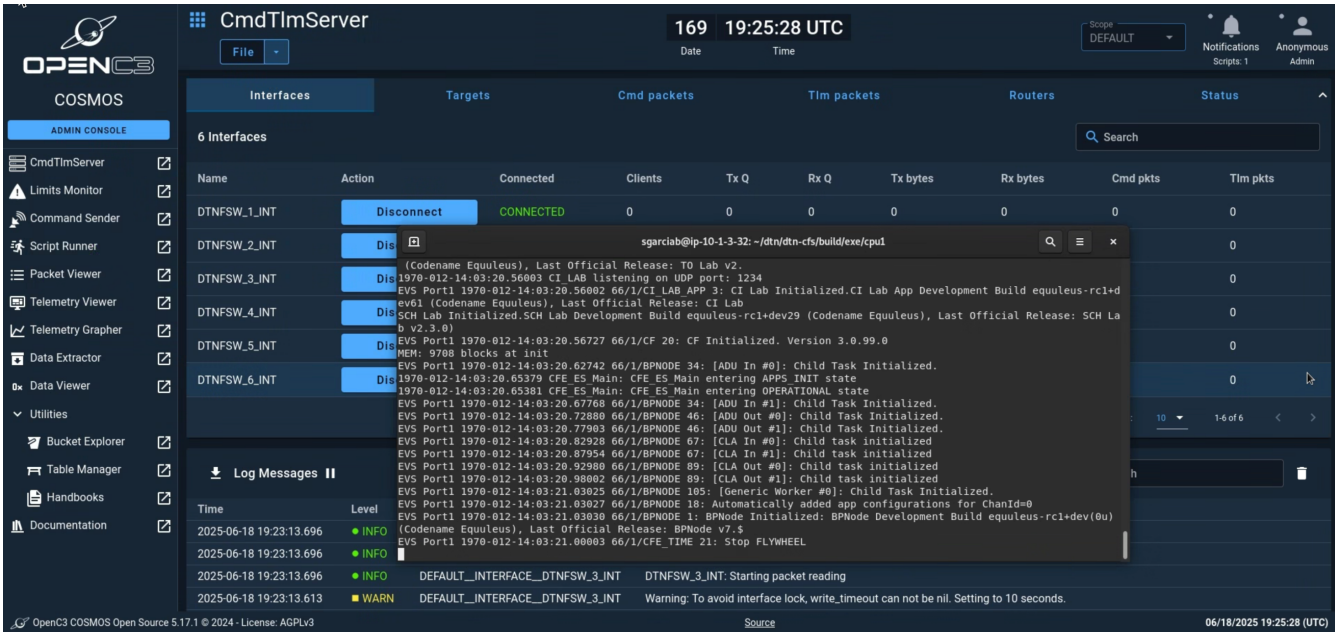

Figure 8. COSMOS Interface

Appendix B: Host Assumptions

BPNode operates under the following host assumptions:

ADU Proxy

DTN.A.00080

Host provides a mechanism for the DTN Node to exchange ADUs with local client applications.

DTN.A.00090

The mechanism by which ADUs are exchanged between the DTN Node and its client applications is reliable.

Directive Proxy

DTN.A.02120

Host provides a command ingest application that accepts and sends CCSDS telecommand packets in order.

DTN.A.02220

The host will authenticate the source of all directives.

Event Management

DTN.A.00020

Host provides event services.

DTN.A.00030

Host event services provide the capability to filter events by event ID.

DTN.A.00040

Host event services provide the capability to filter events by event type: critical, error, info, and debug.

DTN.A.00050

Host event services include in all event messages the time at which the event occurred.

DTN.A.00060

Host event services will send event messages in the form of CCSDS Space Packets.

Event Proxy

DTN.A.00020

Host provides event services.

DTN.A.00030

Host event services provide the capability to filter events by event ID.

DTN.A.00040

Host event services provide the capability to filter events by event type: critical, error, info, and debug.

DTN.A.00050

Host event services include in all event messages the time at which the event occurred.

DTN.A.00060

Host event services shall send event messages in in a form specified by the flight software framework.

TABLE

DTN.A.02030

Host Table Services supports initialization of the DTN Node configuration.

DTN.A.02031

Host Table Services performs table updates synchronously with the DTN Node application that owns the table to ensure data integrity and table activation atomicity.

DTN.A.02032

Host Table Services supports common viewing of DTN Node application configuration and policy data structures.

DTN.A.02200

The host will load configuration updates via table.

DTN.A.02210

The host will dump DTN Node configuration from tables.

TIME and Time Management

DTN.A.00000

Host provides a monotonic clock that gives a count of ticks since the most recent power-on reset.

DTN.A.00005

Host monotonic time will not rollover during the duration of the mission.

DTN.A.00010

Host monotonic clock reliably counts forward except when reset to zero.

DTN.A.00012

Host provides "host time", an absolute time kept synchronized by the host. Host time is a counter from set epoch known by the DTN Node.

DTN.A.00015

Host determines if host time is valid and provides a time quality indicator.

DTN.A.00017

Host will interface to Position, Navigation, and Timing (PNT) services, if necessary, to synchronize clock time.

Telemetry Proxy

DTN.A.02110

Host provides a telemetry output application for non-DTN communication channels, which transports CCSDS telemetry packets if applicable.

DTN.A.02195

Host provides a method to trigger periodic generation of telemetry.

Appendix C: Acronyms

Acronym
: Definition

AA
: Application Agent

ADU
: Application Data Units

BP
: Bundle Protocol

BPA
: Bundle Protocol Agent

BPLib
: Bundle Protocol Library

BPNode
: Bundle Protocol Node

CBOR
: Concise Binary Object Representation

CCSDS
: Consultative Committee for Space Data Systems

cFE
: core Flight Executive

CFDP
: CCSDS File Delivery Protocol

cFS
: code Flight Systems

CI
: Core Infrastructure (in the context of BPNode), Command Ingest (in cFS context)

CL
: Convergence Layer

CLA
: Convergence Layer Adaptor

DTN
: Delay/Disruption-Tolerante Network

EVS
: Event Services

FSW
: Flight Software

GSFC
: Goddard Space Flight Center

OSAL
: Operating System Abstraction Layer

PSP
: Platform Support Package

RFC
: Request for Comments

SB
: Software Bus

SC
: Stored Commands

TO
: Telemetry Output