



High-Rate Delay Tolerant Networking (HDTN) User Guide Version 1.1.0

*Stephanie Booth, Rachel Dudukovich, Nadia Kortas, Ethan Schweinsberg,
Brian Tomko, Blake LaFuenta, Evan Danish, Timothy Recker, and Prash Choksi
Glenn Research Center, Cleveland, Ohio*

This Revised Copy, numbered as NASA/TM-20230000826/REV1, December 2023, supersedes the previous version, NASA/TM-20230000826, April 2023, in its entirety.

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Technical Report Server—Registered (NTRS Reg) and NASA Technical Report Server—Public (NTRS) thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers, but has less stringent limitations on manuscript length and extent of graphic presentations.
- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., “quick-release” reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.
- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- **TECHNICAL TRANSLATION.** English-language translations of foreign scientific and technical material pertinent to NASA's mission.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at <http://www.sti.nasa.gov>
- E-mail your question to help@sti.nasa.gov
- Fax your question to the NASA STI Information Desk at 757-864-6500
- Telephone the NASA STI Information Desk at 757-864-9658
- Write to:
NASA STI Program
Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199



High-Rate Delay Tolerant Networking (HDTN) User Guide Version 1.1.0

*Stephanie Booth, Rachel Dudukovich, Nadia Kortas, Ethan Schweinsberg,
Brian Tomko, Blake LaFuente, Evan Danish, Timothy Recker, and Prash Choksi
Glenn Research Center, Cleveland, Ohio*

This Revised Copy, numbered as NASA/TM-20230000826/REV1, December 2023, supersedes the previous version, NASA/TM-20230000826, April 2023, in its entirety.

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

Acknowledgments

This work is funded by the programmatic SCA_N office in the SCA_N Network Services. On behalf of the HDTN team, we would like to thank the following who have contributed to the project since its infancy: Monty Andro, Aimee Bogner, Gilbert Clark, Skylar Hoffert, Alan Hylton, Dennis Iannicca, Katherine King, Blake LaFuente, Lisa Lambert, Gary Pease, Norman Prokkop, Nick Tollis, and Dale Walter.

Revised Copy

This Revised Copy, numbered as NASA/TM-20230000826/REV1, December 2023, supersedes the previous version, NASA/TM-20230000826, April 2023, in its entirety.

Extensive modifications have been made to the report title, author list, and text.

Level of Review: This material has been technically reviewed by technical management.

Contents

1	HDTN Version 1.1 Description	1
1.1	HDTN Version 1.1	1
1.2	Changes from Version 1.0.0 to 1.1.0	1
1.2.1	Fixed	1
1.2.2	Added	1
1.2.3	Changed	2
1.2.4	Removed	2
2	High-rate Delay Tolerant Networking Overview	2
3	Architecture	3
3.1	Ingress	3
3.2	Storage	4
3.3	Router	4
3.4	Egress	4
3.5	Telemetry Command Interface	4
3.6	Libraries	4
4	Requirements	5
4.1	Tested Platforms	5
4.2	Dependencies	5
4.2.1	Linux Dependencies	5
4.2.2	Windows Dependencies	6
4.3	Known Issues	6
5	Build HDTN	6
5.1	Notes on HDTN CMake	6
5.2	Build HDTN on Linux	6
5.3	Optional x86 Hardware Acceleration	7
5.4	Storage Capacity Compilation Parameters	8
5.5	Build HDTN on Windows with its Dependencies	8
5.5.1	HDTN Developers	9
5.5.2	Setup Instructions for Developers Using Installed HDTN Libraries within their own Projects	10
5.6	Build HDTN on ARM	11
5.6.1	Debugging Errors/Problems	11
5.7	Building for ARM on x86	11
5.7.1	Setting up ARM Chroot on x86 Desktop	11
5.7.2	Setting up HDTN Dependencies in the Chroot Environment	12
5.7.3	Compiling HDTN	12
5.7.4	Useful Commands	12
6	Running HDTN	12
6.1	Directory Structure	13
6.2	Unit Tests	13
6.3	Integrated Tests	13
7	Graphical User Interface	13
7.1	Running the Web Interface	13
7.2	Statistics Page	13
7.3	System View Page	14
7.4	Config Page	14
7.5	Statistics Logging	14

8	Getting Started with the API	16
8.1	API Calls	16
8.1.1	HDTN Version	16
8.1.2	HDTN Configuration	17
8.1.3	Storage	17
8.1.4	Expiring Storage	17
8.1.5	Inducts	18
8.1.6	Outducts	18
8.1.7	Maximum Send Rate for an Outduct	18
8.1.8	BP Security Configuration	19
8.1.9	Upload Contact Plan	19
8.1.10	Ping	20
9	Simulations	21
10	HDTN Applications	21
10.1	BPGen	21
10.2	BPSink	21
10.3	BPSendFile	22
10.4	BPReceiveFile	22
10.5	BPing	22
10.6	Fprime Applications	22
11	Runscript	22
11.1	Path Variables	23
11.2	BpSink	23
11.3	BpReceiveFile	23
11.4	Egress	23
11.5	Router	24
11.6	Ingress	24
11.7	Storage	24
11.8	HDTN One Process	24
11.9	BpGen	24
11.10	BpSendFile	25
11.11	Bping	25
11.12	BpSendPacket	25
11.13	BpReceivePacket	25
11.14	CleanUp	26
12	Config Files	26
12.1	hdtm.config	26
12.2	sink.config	29
12.3	gen.config	29
12.4	bpsec.config	30
12.5	distributed.config	32
13	Contact Plans	34
13.1	JSON Fields	34
14	Convergence Layers and Routing Protocols	34
14.1	Overview of Compatible Convergence Layers	34
14.2	Additions to Config Files	35
14.2.1	TCPCLv3	35
14.2.2	TCPCLv4	36
14.2.3	UDPCL	37

14.2.4 LTP	37
14.2.5 STCP	38
15 Test Configurations and Instructions	39
15.1 TCP Loopback Test	39
15.2 Two Node LTP Test	39
15.3 Four Nodes STCP Test	39
15.4 File Transfer Test	40
15.5 Integrated Tests	41
16 Containerization	41
16.1 Docker Instructions	41
16.2 Docker Compose Instructions	42
16.3 Kubernetes Instructions	42
17 Troubleshooting	43
17.1 Logging	43
17.2 LTP Tuning Recommendations	43
18 Notes	44
18.1 TLS Support for TCPCL Version 4	44
18.2 BP Version 6 and Version 7	45
18.2.1 Bundle Protocol Version 6	45
18.2.2 Bundle Protocol Version 7	48

High-Rate Delay Tolerant Networking (HDTN) User Guide

Version 1.1.0

Stephanie Booth, Rachel Dudukovich, Nadia Kortas, Ethan Schweinsberg, Brian Tomko, Blake LaFuente,
Evan Danish, Timothy Recker, and Prash Choksi
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

1 HDTN Version 1.1 Description

1.1 HDTN Version 1.1

HDTN Version 1.1.0 is a minor release with several features promoted out of the HDTN development branch to the master branch. The most notable additions are BPSec, consolidation of the scheduler and router modules, and beginning work on the HDTN API. At the time of the HDTN v1.1.0 minor release, the software and engineering processes are undergoing improvements to reach NASA Procedural Requirements 7150.2D Class B compliance. HDTN is currently Class D. Full Class B compliance is planned for HDTN v2.0.0 in the first half of 2024. New features added to the master branch are considered stable, however additional testing will be conducted prior to the release of HDTN v2.0.0.

1.2 Changes from Version 1.0.0 to 1.1.0

1.2.1 Fixed

- All “volatile” variables, especially “volatile bool”, have been replaced with “std::atomic” with proper memory-ordering semantics.
- All convergence layer telemetry variables/counters use “std::atomic” with proper memory-ordering semantics.
- Routing library now uses reloaded contact plans from the telemetry API.
- Fixed bug where router link state information would become out of sync with actual link state.
- Addressed minor custody transfer bug that was found during testing.

1.2.2 Added

- Bundle Protocol security (BPSec) support. “-bpsec-config-file” command-line argument was added to bpgen, bpsink, bpsendfile, bpreceivefile, and hdtm-one-process. This argument is used to specify BPSec config file location.
- Added “slip_over_uart” convergence layer, allowing bidirectional communication and framing of bundles over a COM/Serial port.
- Added “-cla-rate” command-line argument to bpgen. This argument, defaulting to zero for unlimited, can be used to set the rate for LTP and UDP connections.
- Added config option “neighborDepletedStorageDelaySeconds” allowing for optional rerouting around neighboring nodes with depleted storage. Zero disables; otherwise, the value is interpreted as the amount of time to wait before forwarding to the neighbor after a depleted storage message is received. Requires that custody be enabled.
- Added functionality to support API calls for retrieving configuration and statistics information of HDTN (refer to section 8)
- Added FPrime support by adding BpSendPacket and BpReceivePacket tools which can convert Fprime data to bundles and vice versa.

1.2.3 Changed

- All bundle data types use “padded_vector_uint8_t” instead of “std::vector<uint8_t>” to remain uniform across both inducts, outducts, and BundleView. This results in API changes for:
 - “LtpClientServiceDataToSend”
 - the outduct “Forward” calls
 - the internal buffers of “BundleViewV6” and “BundleViewV7”
- Added support for “BundleViewV6” and “BundleViewV7” to recycle their canonical block header objects whenever the bundle view object is reused in the creation or loading of bundles.
- Combined router and scheduler into one module.
- Updated routing logic. Minor bug fixes and improved handling of interrupted/failed contacts. Upon a failed contact, HDTN will attempt to calculate a new route avoiding the failed node.
- Changed “-bundle-rate” argument to floating point type to allow for rates slower than one bundle-per-second.
- Enabling BP version 6 style priority is optional in BP version 7.

1.2.4 Removed

- Removed “finalDestinationEidUris”, “udpRateBps”, and “ltpMaxSendRateBitsPerSecOrZeroToDisable” fields from the HDTN configuration. These values now come from the contact plan or command-line arguments.

2 High-rate Delay Tolerant Networking Overview

Delay Tolerant Networking (DTN) has been identified as a key technology to enable and facilitate the development and growth of future space networks. Classically, space communications networks are collections of disparate links that are manually managed either point-to-point or use space relays. The accelerating accessibility of space enables a new scaling of space nodes, yet both the manual management of configurations and scheduling and the lack of structure connecting links precisely prohibit scaling. This challenge gives rise to newer and larger classes of communications needs that are met by DTN, which must overcome the disconnection, disruption, latency, and mobility featured in space communications systems.

DTN joins the underlying links as an overlay, and can be made to communicate over any protocol stack. The core actions of DTN are *store*, *carry*, and *forward*, where data are stored instead of dropped if there is no immediately available outduct. It does this by taking the DTN unit of data, *bundles*, and providing necessary layers to adapt these bundles to the underlying transport protocols of choice; these are called *convergence layers*. DTN’s Bundle Protocol (BP) can then be used on top of terrestrial protocol stacks, such as TCP/IP, as well as protocols for space, such as LTP/AOS, all in the same network. For emphasis it is noted that bundles can be of essentially any size, and hence this convergence to lower layers of choice is necessary.

Existing DTN implementations have operated in constrained environments with limited resources, resulting in low data speeds. However, as various technologies have advanced, data transfer rates and efficiency have advanced, which has pushed the need for a DTN implementation for ground systems and for spacecraft that is performance-oriented in order to not impose an unnecessary bottleneck.

High-rate Delay Tolerant Networking (HDTN) takes advantage of modern hardware platforms to substantially reduce latency and improve throughput compared to today’s DTN operations. The HDTN implementation maintains interoperability with existing deployments of DTN that conform to IETF RFCs 4838, 5050, and 9171. At the same time, HDTN defines a new data format better suited to higher-rate operation. It defines and adopts a massively parallel pipelined and message-oriented architecture, allowing the system to scale gracefully as its resources increase. HDTN’s architecture also supports hooks to replace various

processing pipeline elements with specialized hardware accelerators. This offers improved Size, Weight, and Power (SWaP) characteristics while reducing development complexity and cost.

For questions and comments on this project, feel free to reach out to the contributors found on the Github page at <https://github.com/nasa/HDTN>.

3 Architecture

HDTN is written in C++, and is designed to be modular. These modules include:

- Ingress - Processes incoming bundles.
- Storage - Stores bundles to disk.
- Router - Calculates routes for bundles and determines when bundles may be forwarded.
- Egress - Forwards bundles to the proper outduct and next hop.
- Telemetry Command Interface - Displays the operations and data for HDTN.
- Libraries - implement common algorithms and functions needed by the HDTN core modules such as Routing and BPsec libraries

Figure 1 shows the HDTN modules and their interactions:

3.1 Ingress

The Ingress module intakes bundles and decodes the header fields to determine the source and destination of the bundles. If the link is available, Ingress will send the bundles in a cut-through mode straight to Egress, and if the link is down or custody transfer is enabled it sends the bundles to the Storage module. Even if an immediate forwarding opportunity exists, Storage is always required when custody transfer is enabled. The bundle layer must be prepared to re-transmit the bundle if it does not receive an acknowledgment within the time-to-acknowledge that the subsequent custodian has received and accepted the bundle.

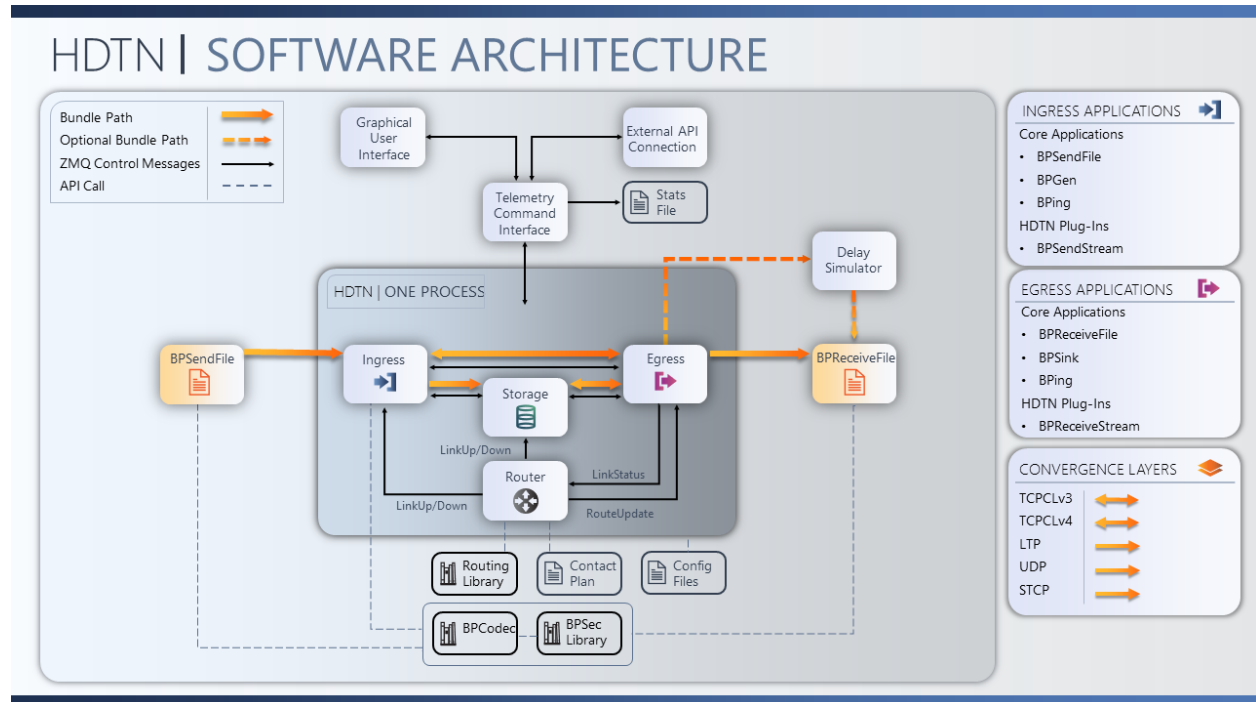


Figure 1.—HDTN architecture.

3.2 Storage

Storage is a multi-threaded implementation distributed across multiple disks that also handles custody transfer. It receives messages from the Router to determine when stored bundles can be released and forwarded to Egress.

3.3 Router

The router calculates routes and tracks link (outduct) state. It is responsible for determining whether neighboring nodes are “up” or “down”, recomputing routes as necessary, and providing that information to the other modules. The router sends `LinkUp` and `LinkDown` messages to indicate when bundles may be forwarded on a given link. The Router also sends `RouteUpdate` messages to Egress to update the route and next hop. The Router uses contact plans in JavaScript Object Notation (JSON) to determine when contacts start and end, and to calculate routes. The Router reloads the contact plan upon receipt of a `NEW_CONTACT_PLAN` request. The Router dynamically handles changes in network state including physical link changes and (optionally) depleted storage on neighboring nodes. If a link unexpectedly goes down, the Router will attempt to reroute around that node. The Router module gets the next hop and best route to the final destination using one of the algorithms in the routing library. HDTN currently supports Contact Graph Routing (CGR), Dijkstra’s algorithm (default algorithm used), and also Contact Multigraph routing (CMR).

3.4 Egress

The Egress module is responsible for forwarding bundles received from Storage or Ingress to the correct outduct and next hop based on the optimal route computed by the Router. HDTN uses an event-driven approach based on ZeroMQ pub-sub sockets for sending unexpected link updates and contact plan changes from Egress to Router. When the connection is lost unexpectedly, Egress will send a `LinkStatus` change message to the Router, which triggers the Router to send `LinkUp` or `LinkDown` events to Ingress and Storage. In addition, the Router will recompute routes and send a `RouteUpdate` message to Egress.

3.5 Telemetry Command Interface

The Telemetry Command Interface module exchanges messages with the Graphical User Interface (GUI) which has a page that displays a live system view of HDTN node with bundles flowing in real time, and pages that display the current configuration settings, data rates graph and bundles statistics for network troubleshooting. The GUI is also used for updating configuration, routes, and contact plans. Telemetry Command Interface module will also be able to communicate with External APIs to display stats and configuration via a command Line Interface (CLI).

3.6 Libraries

- The Routing Library implements the routing algorithms in C++ that currently supports Contact Graph Routing (CGR), Dijkstra’s algorithm (the default algorithm used), and Contact Multigraph Routing (CMR), which is a modified version of Dijkstra’s algorithm using a multigraph structure instead of a contact graph and providing a significant performance improvement.
- BPCodec is a stand-alone library designed to support encoding and decoding the Bundle Protocol format. Both version 6 and version 7 of the bundle protocol are supported
- Bundle Protocol Security (BPsec) Library includes the implementation of the cryptographic functions using OpenSSL APIs and it requires OpenSSL FIPS module to comply with NASA cybersecurity requirements. BPsec is enabled by default with Bundle Protocol Version 7 with Open SSL support. BPsec has the methods for adding and processing BCB confidentiality and BIB integrity blocks based on the security policy Rules. It also defines the methods for searching and processing security policy rules and handling of security events failures. BPsec Library has a dependency BPCodec library as it is using the corresponding class `BundleViewV7` to add and remove security blocks. This is a helper

class for reading/parsing/validating all blocks within Bundle Protocol Version 7 bundles and it makes it easy to modify specific blocks and rewrite bundles to new memory. Bundles may be rendered in-place if the bundle is decreasing in size or if there is enough padding available if the bundle is growing in size.

4 Requirements

In this section, the run environments, including tested platforms, architectures, and dependencies are detailed.

4.1 Tested Platforms

- Linux
 - Ubuntu Desktop 18.04, 18.10, 20.04.2, 20.10
 - Ubuntu Server 20.04, 20.10
 - Debian 10
 - RHEL (Red Hat Enterprise Linux) 8
 - Fedora 38
- Windows
 - Windows 10 (64-bit)
 - Windows Server 2022 (64-bit)
 - Windows Server 2019 (64-bit)
- Raspbian
- ARM on x86

4.2 Dependencies

4.2.1 Linux Dependencies

The HDTN build environment requires:

- CMake version 3.16.3
- Boost library version 1.66.0 minimum, version 1.69.0 for TCPCLv4 TLS version 1.3 support, version 1.70 is required for the Web User Interface to support HTTPS/WSS.
- ZeroMQ version 4.34
- gcc version 9.3.0 (Debian 8.3.0-6)
- OpenSSL version 1.1.1f (Optional - needed to support BPSec)

These can be installed using the following command(s):

On Debian/Ubuntu

```
sudo apt-get install cmake build-essential libzmq3-dev
sudo apt-get install libboost-dev libboost-all-dev openssl libssl-dev
```

On RHEL

```
sudo dnf install epel-release
sudo yum install cmake boost-devel zeromq zeromq-devel
```

On Debian

```
sudo apt-get install cmake build-essential openssl libssl-dev
sudo apt-get install libboost-dev libboost-all-dev libzmq3-dev python3-zmq
```

```
On Fedora sudo dnf install gcc gcc-c++ kernel-devel make cmake boost boost-devel
zeromq-devel openssl-devel
```

4.2.2 Windows Dependencies

HDTN supports 9 permutations of the Visual Studio compilers on Windows:

- Versions: 2022, 2019, and 2017 (note: for 2017, only versions 15.7 and 15.9 have been tested)
- Editions: Enterprise, Professional, and Community

HDTN build environment on Windows requires:

- One of the supported Visual Studio compilers listed in the Overview section. Visual Studio must be installed for C++ Development during setup.
- PowerShell (recommended Visual Studio Code with the PowerShell extension installed)
- 7-Zip
- Perl (needed for building OpenSSL) with perl.exe in the Path environmental variable (Strawberry Perl for Windows has been tested)

4.3 Known Issues

- Ubuntu distributions have been known to install older non-compatible versions of -CMake.
- Some processors do not support hardware acceleration or the RDSEED instruction. (Note: In the CMake file, both are set to “ON” by default.) This support will be auto-detected by CMake if not cross-compiling.
- Mac OS may not support recvmmsg and sendmmsg functions, recvmsg and sendmsg could be used instead.

5 Build HDTN

5.1 Notes on HDTN CMake

All of HDTN’s directories of modules/libraries contain their own CMakeLists.txt file. The root CMakeLists.txt adds all those modules/libraries to the HDTN project using the CMake `add_subdirectory` command. It should be noted that the HDTN CMake files are written using modern CMake paradigms, such as “dependencies as targets” which makes it much easier and cleaner to manage a multi-platform library like HDTN. In addition, package config information gets exported to the installation (`install_root/lib/cmake`) whenever a user wants to do a “make install”. There is an example in `tests/unit_tests/import_installation/CMakeLists.txt` which is a copy of the regular `tests/unit_tests/CMakeLists.txt` except that the former uses the `find_package` package config information from the `install_root/lib/cmake` directory. The package config information is great for users that may want to write custom software projects that only use portions of the HDTN codebase such as a library of a particular convergence layer. The HDTN CMake tries to optimize the build as much as possible; it will test the compiler for more recent C++ standards, and it will test the compiler and the CPU for specific x86 hardware instructions and utilize those if available. Finally, the HDTN CMake supports building its libraries as either static or shared using CMake’s `GENERATE_EXPORT_HEADER` which is required for building or using `.dll` files on Windows and for using GCC’s new C++ visibility support.

5.2 Build HDTN on Linux

To build HDTN in “Release mode”, perform the following steps. (Note: If the `-DCMAKE_BUILD_TYPE` is not specified, HDTN is built in “Release mode” by default).

- `git clone https://github.com/nasa/HDTN.git`
- `export HDTN_SOURCE_ROOT=/home/username/HDTN` (set to filepath containing HDTN)

- `cd $HDTN_SOURCE_ROOT`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make`
 - Adding `-j8` (i.e. `make -j8`) to the make will speed up the processing time but requires a system with at least 8 cores.

Note: By Default, `BUILD_SHARED_LIBS` is OFF and HDTN is built as static. To use shared libs, edit `CMakeCache.txt`, set `BUILD_SHARED_LIBS:BOOL=ON` and add `fPIC` to the `CMakeCache` variable: `CMAKE_CXX_FLAGS_RELEASE:STRING=-O3 -DNDEBUG -fPIC`

5.3 Optional x86 Hardware Acceleration

HDTN build environment sets the following `CMakeCache` variables to “On” by default:

`USE_X86_HARDWARE_ACCELERATION` and `LTP_RNG_USE_RDSEED`.

Notes:

- If building natively (i.e. not cross-compiling), the HDTN `CMake` build environment will check the processor’s CPU instruction set and the compiler to determine which HDTN hardware accelerated functions will build and run on the native host. `CMake` automatically sets various compiler definitions to enable supported HDTN hardware accelerated features.
- If cross-compiling, the HDTN `CMake` build environment will check the compiler to determine if the HDTN hardware accelerated functions will build. It is up to the user to determine if the target processor can support/run those instructions. `CMake` will automatically set various compiler definitions to enable supported HDTN hardware accelerated features only if they compile.
- Hardware accelerated functions can be turned off by setting `USE_X86_HARDWARE_ACCELERATION` and/or `LTP_RNG_USE_RDSEED` to “Off” in the `CMakeCache.txt`.
- If building for ARM or any non X86-64 platform, `USE_X86_HARDWARE_ACCELERATION` and `LTP_RNG_USE_RDSEED` must be set to “Off”.

If `USE_X86_HARDWARE_ACCELERATION` is turned “On” some or all of the following features will be enabled if `CMake` finds support for these CPU instructions:

- Fast SDNV encode/decode (BPv6, TCPCLv3, and LTP) requires SSE, SSE2, SSE3, SSSE3, SSE4.1, POPCNT, BMI1, and BMI2.
- Fast batch 32-byte SDNV decode (not yet implemented into HDTN but available in the `common/util/Sdnv` library) requires AVX, AVX2, and the above “Fast SDNV” support.
- Fast CBOR encode/decode (BPv7) requires SSE and SSE2.
- Some optimized loads and stores for TCPCLv4 requires SSE and SSE2.
- Fast CRC32C (BPv7 and a storage hash function) requires SSE4.2.
- The HDTN storage controller will use BITTEST if available. If BITTEST is unavailable, it will use ANDN if BMI1 is available.

If `LTP_RNG_USE_RDSEED` is turned “On”, this feature will be enabled if `CMake` finds support for this CPU instruction:

- An additional randomness source for LTP’s random number generator requires RDSEED. This feature can be disabled for potentially faster LTP performance.

5.4 Storage Capacity Compilation Parameters

HDTN build environment sets two CMake cache variables by default: `STORAGE_SEGMENT_ID_SIZE_BITS` and `STORAGE_SEGMENT_SIZE_MULTIPLE_OF_4KB`.

- The `STORAGE_SEGMENT_ID_SIZE_BITS` flag must be set to the recommended default, 32 or 64. It determines the size/type of the storage module's `segment_id.t`. Setting the flag to 32-bit significantly decreases memory usage because we don't need to use the RAM.
 - If this value is 32, the formula for the max segments (S) is given by

$$S = \min(UINT32_MAX, 64^6) \approx 4.3 \text{ billion}$$

segments since `segment_id.t` is a `uint32.t`. A segment allocator using 4.3 Billion segments uses about 533 MByte RAM), and multiplying by the minimum 4KB block size gives 17TB bundle storage capacity. Make sure to appropriately set the `totalStorageCapacityBytes` variable in the HDTN JSON config so that only the required amount of memory is used for the segment allocator.

- If this value is 64, the formula for the max segments (S) is given by

$$S = \min(UINT64_MAX, 64^6) \approx 68.7 \text{ billion}$$

segments since `segment_id.t` is a `uint64.t`. Using a segment allocator with 68.7 Billion segments, when multiplying by the minimum 4KB block size gives ~ 281TB bundle storage capacity.

- The flag `STORAGE_SEGMENT_SIZE_MULTIPLE_OF_4KB` must be set to an integer of 1 or greater. It determines the minimum increment of bundle storage based on the standard block size of 4096 bytes. (Note: One is the default and recommended.) Example:
 - If `STORAGE_SEGMENT_SIZE_MULTIPLE_OF_4KB=1` , a `4KB*1=4KB` block size is used. A bundle size of 1KB would require 4KB of storage. A bundle size of 6KB would require 8KB of storage.
 - If `STORAGE_SEGMENT_SIZE_MULTIPLE_OF_4KB=2`, a `4KB*2=8KB` block size is used. A bundle size of 1KB would require 8KB of storage. A bundle size of 6KB would require 8KB of storage. A bundle size of 9KB would require 16KB of storage. If

`STORAGE_SEGMENT_ID_SIZE_BITS=32`

then bundle storage capacity could potentially double from ~ 17TB to ~ 34TB.

For information on how the Storage works, see `module/storage/doc/storage.pptx` in the HDTN main repository.

5.5 Build HDTN on Windows with its Dependencies

To build HDTN and its dependencies in Release mode and as shared libraries (shared .dll files for both HDTN and its dependencies), simply run the PowerShell script in `building_on_windows\hdtn_windows_cicd_unit_test.ps1` from any working directory. The working directory does not matter. Once finished, HDTN and its dependencies will be installed to `C:\hdtn_build_x64_release_vs2022` (suffix will be 2019 or 2017 if that's the Visual Studio compiler installed). The script will also run HDTN's unit tests after the build. Once completed, you will see the following message:

```
"Remember, HDTN was built as a shared library, so you must prepend the following to your Path so that Windows can find the DLL's of HDTN and its dependencies:"
```

It will print four directory locations to be added to your Path environmental variable to facilitate use of HDTN outside this PowerShell script.

- From the Windows Start Menu, type `env`.
- Open Edit environmental variables for your account

- double click Path
- Add the four directories. (Omit the directory containing `hdtm_install\lib` if modifying HDTN source code within Visual Studio. You will later build and install your HDTN binaries within Visual Studio.)
- If you are a user of HDTN and you are NOT going to modify HDTN source code within Visual Studio, also add this directory to your Path: `C:\hdtm_build_x64_release_vs2022\hdtm_install\bin`
- Click OK
- Click New
- Add the following new variable: `HDTM_SOURCE_ROOT`
- Set the variable value to your source root (the folder that contains `README.md`).
Example `C:\path\to\hdtm`
- Click OK
- Click OK

If you are a user of HDTN and you are NOT going to modify HDTN source code within Visual Studio, you can reference any of the .bat file example tests located in `HDTM_SOURCE_ROOT\tests\test_script_s_windows`. Note that these scripts were intended for developers, so you will have to modify the scripts, fixing any lines that reference `HDTM_BUILD_ROOT`, so, for example, if you see `%HDTM_BUILD_ROOT%\common\bpcodec\apps\bpgen-async.exe`, replace it with `bpgen-async.exe`. Also note that these .bat files reference config files located in `HDTM_SOURCE_ROOT\config_files`, so feel free to modify those .json configs to meet your needs.

5.5.1 HDTN Developers

If you are a developer and you are going to modify HDTN source code within Visual Studio, you may delete the directory `C:\hdtm_build_x64_release_vs2022\hdtm_install` and continue on with the next set of instructions.

Launch Visual Studio 2022 and open HDTN as a project with these steps:

- File >> open >> cmake
- Open HDTN root CMakeLists.txt
- Make sure drop down configuration at the top is set to x64-Release. You may need to go to Manage Configurations if not.

Then click Project >> view CMakeCache.txt Add these lines (change `_vs2022` directory suffix if different):

- `BOOST_INCLUDEDIR:PATH=C:\hdtm_build_x64_release_vs2022\boost_1.78.0_install`
- `BOOST_LIBRARYDIR:PATH=C:\hdtm_build_x64_release_vs2022\boost_1.78.0_install\lib64`
- `BOOST_ROOT:PATH=C:\hdtm_build_x64_release_vs2022\boost_1.78.0_install`
- `OPENSSL_INCLUDE_DIR:PATH=C:\hdtm_build_x64_release_vs2022\openssl-1.1.1s_install\include`
- `OPENSSL_ROOT_DIR:PATH=C:\hdtm_build_x64_release_vs2022\openssl-1.1.1s_install`
- `libzmq_INCLUDE:PATH=C:\hdtm_build_x64_release_vs2022\libzmq.v4.3.4_install\include`

- `libzmq_LIB:FILEPATH=C:\hdtb_build_x64_release_vs2022\libzmq-v4.3.4-install\lib\libzmq-v143-mt-4_3_4.lib` (note: may be v141 or v142)
- `BUILD_SHARED_LIBS:BOOL=ON`

Then click Project >> configure cache

It is now time to set up additional environmental variables in order to be able to run the .bat file tests located in `HDTN_SOURCE_ROOT\tests\test_scripts_windows`:

- Right click on the open tab within Visual Studio titled `CMakeCache.txt` and then click "Open Containing Folder"
- Copy the path at the top of the Windows Explorer window
- From the Windows Start Menu, type "env", open "Edit environmental variables for your account"
- Click New
- Add the following new variable: `HDTN_BUILD_ROOT`. The variable value will look something like `C:\Users\username\CMakeBuilds\17e7ec0d-5e2f-4956-8a91-1b32467252b0\build\x64-Release`
- Click OK
- Click New
- Add the following new variable: `HDTN_INSTALL_ROOT`; the value will look similar to `HDTN_BUILD_ROOT` except change "build" to "install".
something like `C:\Users\username\CMakeBuilds\17e7ec0d-5e2f-4956-8a91-1b32467252b0\install\x64-Release`
- Click OK
- Double click the Path variable, add the `HDTN_INSTALL_ROOT\lib` folder to your Path, something like `C:\Users\username\CMakeBuilds\17e7ec0d-5e2f-4956-8a91-1b32467252b0\install\x64-Release\lib`. This step is needed because HDTN is built as a shared library with multiple .dll files, so this step allows Windows to find those .dll files when running any HDTN binaries.

Relaunch Visual studio so that it get's loaded with the updated environmental variables. Now build HDTN:

- Build >> Build All
- Build >> Install HDTN
- Run `unit_tests.bat` located in `HDTN_SOURCE_ROOT\tests\test_scripts_windows`
- For a Web GUI example, run `test_tcpcl_fast_cutthrough_oneprocess.bat` and then navigate to `http://localhost:8086` (note: to exit cleanly, do a ctrl-c in each cmd window before closing)

NOTE: Since CMake is currently configured to build HDTN as a shared library (because the CMake cache variable `BUILD_SHARED_LIBS` is set to ON), any time you make a source code change to HDTN, for it to be reflected in the binaries, don't forget to Build >> Install HDTN after the Build >> Build All step.

5.5.2 Setup Instructions for Developers Using Installed HDTN Libraries within their own Projects

HDTN utilizes modern CMake. When HDTN is installed, it installs the appropriate CMake Packages that can be imported. For an example of this use case, see `HDTN_SOURCE_ROOT\tests\unit_tests_import_installation\CMakeLists.txt` for a project that imports the libraries and headers from an HDTN installation and builds HDTN's unit tests from that installation.

5.6 Build HDTN on ARM

HDTN is compiled with x86 hardware optimizations by defaults. To compile for ARM, these optimizations must be disabled. To build HDTN on ARM running Ubuntu (follow Ubuntu dependencies in Section 4.2):

1. `cd HDTN`
2. `mkdir build && cd build`
3. `cmake .. -DCMAKE_SYSTEM_PROCESSOR=arm`
4. `make -j`
5. `sudo make install -j`

These commands should work for any ARM platform running Ubuntu such as Raspberry Pi 4 or NVidia Jetson Nano. Compiling on a Raspberry Pi 4 takes anywhere from 30 minutes to an hour. It is recommended to compile with only one or two cores (`make -j2`) on memory constrained devices to ensure `gcc` does not consume all RAM.

5.6.1 Debugging Errors/Problems

- For errors reporting similar to: `cpuid.h is not found for file HDTN/common/util/src/CpuFlagDetection.cpp`
 - Double check `CMakeList.txt` edits
- For errors reporting similar to: recompile with `-fPIC`
 - Double check `CMakeCache.txt` edits
- For errors reporting `./runscript.sh not found`
 - Run: `export HDTN_SOURCE_ROOT=/home/user/HDTN`
- For errors reporting similar to: `no tcpdump`
 - Run: `sudo apt install tcpdump`
- For runtime errors:
 - Check the log files under `HDTN/logs`. These are not created by default but can be created following the instructions in Section 17.1.

5.7 Building for ARM on x86

5.7.1 Setting up ARM Chroot on x86 Desktop

Run the following commands:

- `sudo apt install qemu-user-static`
- `sudo apt install debootstrap`
- `sudo qemu-debootstrap--variant=buildd--archarm64focal/var/chroot/http://ports.ubuntu.com/`
 - `focal` in the above command is the name of the Ubuntu Release and may need to be changed.
 - This final command creates an `armhf` operating system located at `/var/chroot`. Users can move it elsewhere, but it is recommended to keep it out of the `/home/` and `user` directories.
- To get into chroot, use the command: `sudo chroot /var/chroot`

At this point, users are now in an ARM environment. Users should run the commands in the following sections AFTER they enter the ARM environment.

5.7.2 Setting up HDTN Dependencies in the Chroot Environment

Note: Sudo does not exist in chroot.

- `apt install make cmake build-essential software-properties-common`
- `add-apt-repository universe`
- `apt update`
- `apt install libboost-dev libboost-all-dev libzmq3-dev openssl libssl-dev`

5.7.3 Compiling HDTN

- Download the latest HDTN from Github.
- Unzip the file in your home directory.
 - Note: Users cannot write directly to ARM emulator directories in Windows Subsystem for Linux.
- `sudo mv HDTN /var/chroot/home`
- `sudo chroot /var/chroot`
- `cd home/HDTN`
- `mkdir build`
- `cd build`
- `cmake .. -DCMAKE_SYSTEM_PROCESSOR=arm`
- `make -j 1`
 - Multiple threads will cause a race condition.

5.7.4 Useful Commands

- `readelf -h executable`
 - Read executable header.
- `apt install cmake-curses-gui`
 - Installs CMakeCache.txt editor install for static builds.
- `ccmake ..`
 - Runs the CMakeCache.txt editor.

6 Running HDTN

Note: Ensure your config files are correct, e.g., check that the outduct `remotePort` is the same as the induct `boundPort`, a constant `convergenceLayer`, and the outduct's `remoteHostname` is pointed to the correct IP adress. `tcpdump` can be used to test the HDTN ingress storage and egress. The generated `pcap` file can be read using `wireshark`: `sudo tcpdump -i lo -vv -s0 port 4558 -w hdtm-traffic.pcap`
In another terminal, run: `./runscript.sh`

Note: The Contact Plan, which lists future contacts for each node, is located under `module/router/-contact_plans/contactPlan.json` and includes the source and destination nodes, the start and end times, and the data rates. Based on the schedule in the Contact Plan the router sends events on link availability to Ingress and Storage. When the Ingress receives the `Link Available` event for a given destination,

it sends the bundles directly to egress. When the Link is Unavailable it sends the bundles to storage. Upon receiving Link Available event, Storage releases the bundle(s) for the corresponding destination. When a Link Down event is received, Storage stops releasing the bundles.

There are additional test scripts located under the directories `test_scripts_linux` and `test_scripts_windows` that can be used to test different scenarios for all convergence layers.

6.1 Directory Structure

<code>common/</code>	Common Libraries and Utils
<code>module/</code>	HDTN Core Modules
– <code>egress</code>	CL adapter(s) that forwards bundle traffic
– <code>ingress</code>	CL adapter(s) that accepts traffic in bundle format
– <code>storage</code>	Stores bundles
– <code>router</code>	Sends link state and routes to other modules
– <code>hdtm_one_process</code>	Combines the main processes into one HDTN process
– <code>udp_delay_sim</code>	Proxy that simulates long delays
– <code>telem_cmd_interface</code>	Web interface for stats display and configuration
<code>config_files/</code>	HDTN config files
<code>tests/</code>	Example Test cases and experiments

6.2 Unit Tests

After building HDTN (see Section 5), unit tests can be run using the following command within the build directory:

```
./tests/unit_tests/unit-tests
```

6.3 Integrated Tests

After building HDTN (see Section 5), integrated tests can be run using the following command within the build directory:

```
./tests/integrated_tests/integrated-tests
```

7 Graphical User Interface

7.1 Running the Web Interface

This repository comes equipped with code to launch a web-based user interface to display statistics for the HDTN engine. It relies on a dependency called Boost Beast which is packaged as a header-only library that comes with a standard Boost installation. The web interface requires OpenSSL since the web interface supports both http as well as https, and hence both ws (WebSocket) and wss (WebSocket Secure). The web interface is compiled by default. Anytime that HDTNOneProcess runs, the web page will be accessible at `http://localhost:8086`

To prevent the web interface from running, follow the normal build instructions for Linux. The only difference will be in the cmake command will now be: `cmake -DUSE_WEB_INTERFACE:BOOL=OFF ..`

7.2 Statistics Page

This page, displayed in Figure 2, displays real-time telemetry of HDTN. At the top are three boxes displaying the current Data Rate in Mega-Bits Per Second, the Average Data Rate, and the Maximum Data Rate reached. All of these are measured in the Ingress module.

Beneath are three graphs. The first two display the data rate of the Ingress and Egress Modules in Mega-bits Per Second. The third graph is a pie chart displaying the location of data bundles received by Ingress - either sent to the Storage module or directly to Egress. If a bundle is sent from Storage to Egress, it will be measured on the pie chart as having gone to Egress.

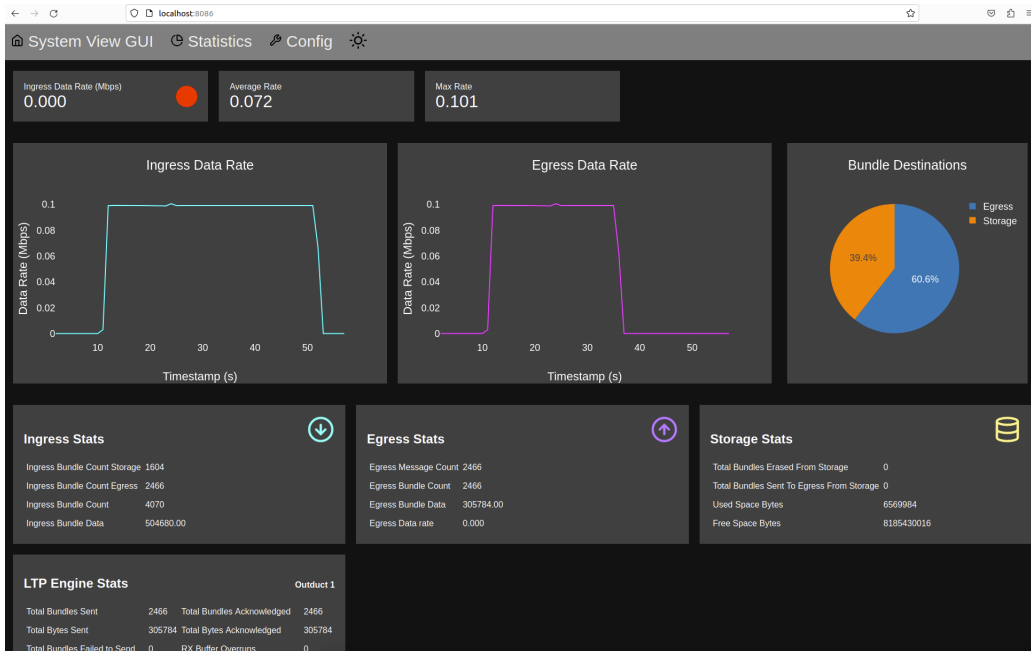


Figure 2.—Statistics Page of the Web Interface.

Beneath the graphs, cards display statistics for different parts of HDTN. At this time only Ingress and Egress are displayed.

7.3 System View Page

This page can be accessed by clicking `System View GUI` in the top left-hand corner of the Statistics page. As shown in Figure 3, this page displays a graphic of the different modules of HDTN as well as information on where the bundled data is coming from and where it is going. In the top row are adjustable settings for users to make the information more legible. On the left of the page are the IP addresses and IPN numbers from which data is being received. On the right is displayed the Nodes and IPN numbers to which data is being sent from this HDTN node. The graphic displays the data rate as data comes into into Ingress, between the different modules of HDTN (Ingress, Storage, and Egress), and the rate as it leaves Egress. The Storage graphic displays the percentage and amount of storage space being used.

Users can also hover over each HDTN module and a pop-up graphic will appear displaying data for that module. An example is shown in Figure 4 which shows this information for the Storage module.

7.4 Config Page

This page is not configured yet.

7.5 Statistics Logging

HDTN telemetry can be automatically logged to CSV files by compiling HDTN with the `DO_STATS_LOGGING` CMake option. The command for enabling this is: `cmake -DDO_STATS_LOGGING:BOOL=ON`. Files will be created in the `/stats` directory of the source code root. Statistics are logged on a 1 second interval. The following statistics are currently supported:

- `ingress_data_rate_mbps`
- `ingress_total_bytes_sent`
- `ingress_bytes_sent_egress`

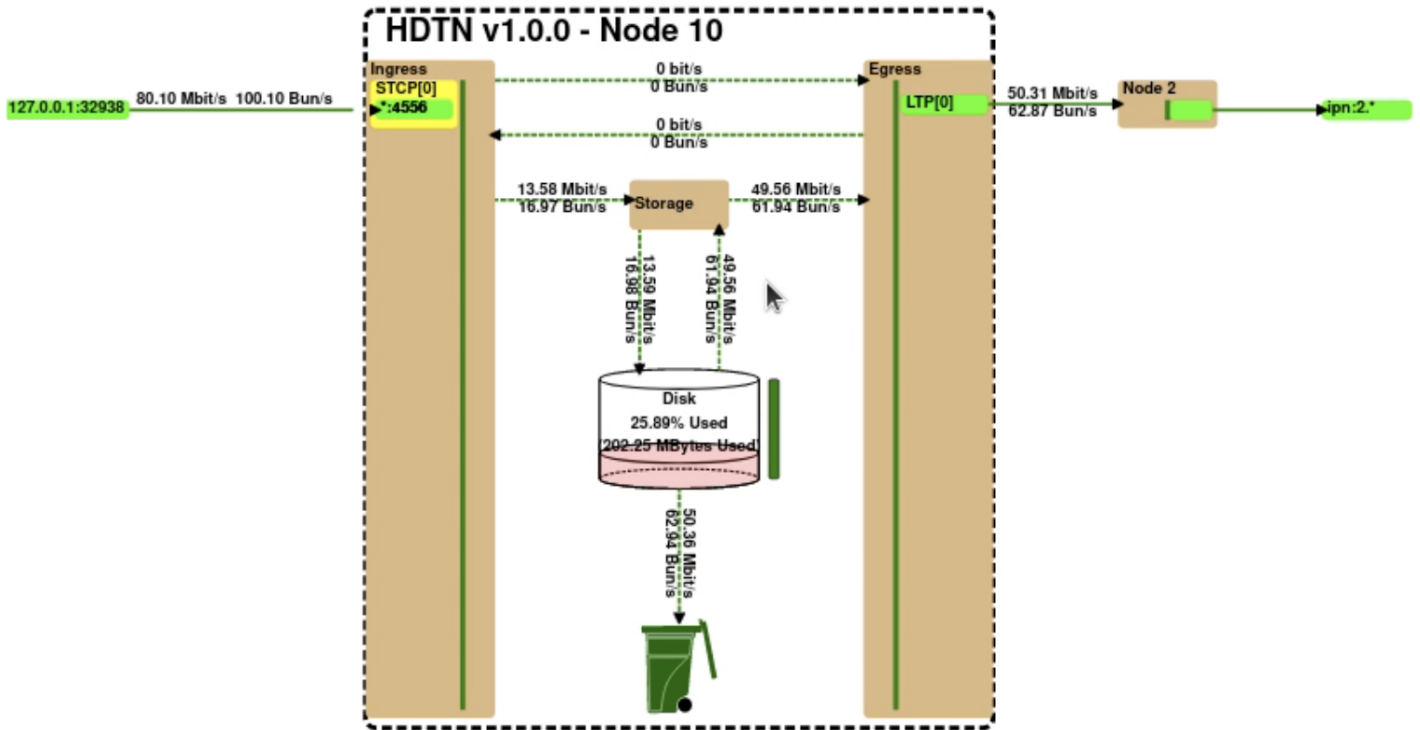


Figure 3.—System View Page of the Web Interface.

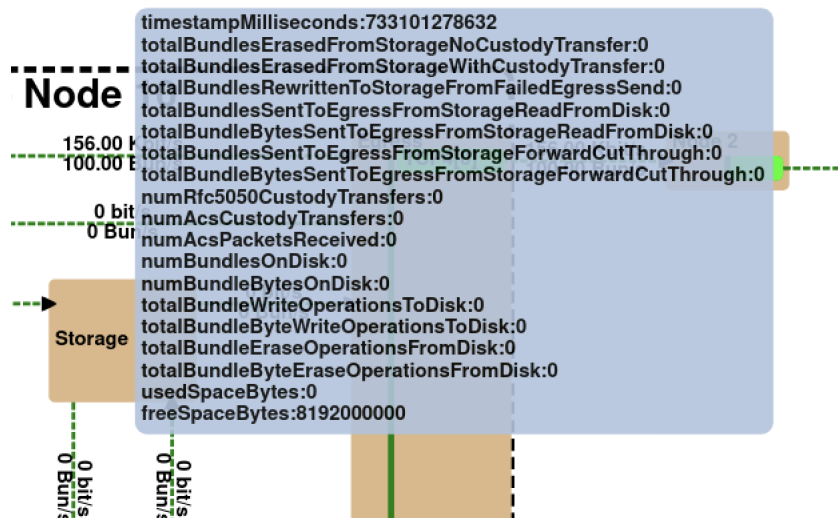


Figure 4.—Pop-up Graphic Displaying Data from the Storage Module.

- ingress_bytes_sent_storage
- storage_used_space_bytes
- storage_free_space_bytes
- storage_bundle_bytes_on_disk
- storage_bundles_erased
- storage_bundles_rewritten_from_failed_egress_send
- storage_bytes_sent_to_egress_cutthrough
- storage_bytes_sent_to_egress_from_dis
- egress_data_rate_mbps
- egress_total_bytes_sent_success
- egress_total_bytes_attempted

8 Getting Started with the API

The HDTN API is a ZeroMQ-Based Messaging API that follows a request-reply messaging pattern. The HDTN API leverages the ZeroMQ messaging library to facilitate efficient and flexible communication between HDTN and other software systems. It enable users to request information on the configuration and statistics of HDTN.

A simple python script named `run_api_commands.py` under `tests/tests_scripts_linux` has been provided as a reference. This script contains all the supported API calls. The API calls can be utilized in any programming language that support the ZeroMQ library. The HDTN API accepts and responds with JSON-encoded strings.

8.1 API Calls

8.1.1 HDTN Version

Call: `get_hdtm_version`

Description: This call is used to obtain the current HDTN version.

Syntax:

```
{
  "apiCall": "get_hdtm_version",
}
```

Parameter(s): None

Return Value: If the call succeeds, then the current HDTN version is returned as a JSON string.

Example:

```
json = {
  "apiCall": "get_hdtm_version",
}
```

8.1.2 HDTN Configuration

Call: get_hdtm_config

Description: This call is used to obtain the currently active configuration information for HDTN.

Syntax:

```
{
  "apiCall": "get_hdtm_config",
}
```

Parameter(s): None

Return Value: If the call succeeds, then the currently active configuration information for HDTN is returned as a JSON string.

Example:

```
json = {
  "apiCall": "get_hdtm_config",
}
```

8.1.3 Storage

Call: get_storage

Description: This call is used to obtain storage statistics information of HDTN.

Syntax:

```
{
  "apiCall": "get_storage",
}
```

Parameter(s): None

Return Value: If the call succeeds, then the storage statistics information of HDTN is returned as a JSON string.

Example:

```
json = {
  "apiCall": "get_storage",
}
```

8.1.4 Expiring Storage

Call: get_expiring_storage

Description: This call is used to obtain expiring storage statistics information of HDTN.

Syntax:

```
{
  "apiCall": "get_expiring_storage",
  "priority": INTEGER,
  "thresholdSecondsFromNow": INTEGER
}
```

Parameter(s):

- **priority** specifies which bundles to look for based on priority when retrieving expired bundle.
 - Input: 0,1, or 2
- **thresholdSecondsFromNow** specifies how far in the future to look for expiring bundles in seconds.
 - Input: positive integer

Return Value: If the call succeeds, then the expiring storage statistics information of HDTN is returned as a JSON string.

Example:

```
jSON = {  
    "apiCall": "get_expiring_storage",  
    "priority": 1,  
    "thresholdSecondsFromNow": 600  
}
```

8.1.5 Inducts

Call: get_inducts

Description: This call is used to obtain all inducts statistics information of HDTN.

Syntax:

```
{  
    "apiCall": "get_inducts",  
}
```

Parameter(s): None

Return Value: If the call succeeds, then all the inducts statistics information of HDTN is returned as a JSON string.

Example:

```
jSON = {  
    "apiCall": "get_inducts",  
}
```

8.1.6 Outducts

Call: get_outducts

Description: This call is used to obtain all outducts statistics information of HDTN.

Syntax:

```
{  
    "apiCall": "get_outducts",  
}
```

Parameter(s): None

Return Value: If the call succeeds, then all the outducts statistics information of HDTN is returned as a JSON string.

Example:

```
jSON = {  
    "apiCall": "get_outducts",  
}
```

8.1.7 Maximum Send Rate for an Outduct

Call: set_max_send_rate

Description: This call is used to set the maximum send rate in bit per second for a specific outduct of HDTN.

Syntax:

```
{  
    "apiCall": "set_max_send_rate",  
    "rateBitsPerSec": INTEGER,  
    "outduct": INTEGER  
}
```

Parameter(s):

- **rateBitsPerSec** specifies the maximum rate in bits per second. When the input is 0, there is no upper limit imposed on the maximum rate.
 - Input: 0 or positive integer
- **outduct** specifies the number of the outduct to which the maximum rate should be applied.
 - Input: positive integer

Return Value: If the call succeeds, then a success response is returned as a JSON string.

Example:

```
json = {
  "apiCall": "set_max_send_rate",
  "rateBitsPerSec": 64000,
  "outduct": 1,
}
```

8.1.8 BP Security Configuration

Call: get_bpsec_config

Description: This call is used to obtain BP Security statistics information of HDTN.

Syntax:

```
{
  "apiCall": "get_bpsec_config",
}
```

Parameter(s): None

Return Value: If the call succeeds, then the BP Security statistics information of HDTN is returned as a JSON string.

Example:

```
json = {
  "apiCall": "get_bpsec_config",
}
```

8.1.9 Upload Contact Plan

Call: upload_contact_plan

Description: This call is used to upload a contact plan for HDTN.

Syntax:

```
{
  "apiCall": "upload_contact_plan",
  "contactPlanJson": JSON STRING
}
```

Parameter(s):

- **contactPlanJson** specifies the contact plan to apply.
 - Input: json string

Return Value: If the call succeeds, then a success response is returned as a JSON string.

Notes: The contact plan presented in the example illustrates a scenario with two nodes

Example:

```

jSON = {
  "apiCall": "upload_contact_plan",
  "contactPlanJson": {
    "contacts": [
      {
        "contact": 0,
        "source": 1,
        "dest": 10,
        "startTime": 0,
        "endTime": 2000,
        "rateBitsPerSec": 0,
        "owlt": 1
      },
      {
        "contact": 1,
        "source": 10,
        "dest": 20,
        "startTime": 0,
        "endTime": 2000,
        "rateBitsPerSec": 0,
        "owlt": 1
      },
      {
        "contact": 2,
        "source": 20,
        "dest": 2,
        "startTime": 0,
        "endTime": 2000,
        "rateBitsPerSec": 0,
        "owlt": 1
      }
    ]
  }
}

```

8.1.10 Ping

Call: ping

Description: This call is used to ping a specific service of a HDTN node.

Syntax:

```

{
  "apiCall": "ping",
  "bpVersion": INTEGER,
  "nodeId": INTEGER,
  "serviceId": INTEGER
}

```

Parameter(s):

- **bpVersion** specifies which Bundle Protocol Version to use for the ping.
 - Input: 6 or 7
- **nodeID** specifies the node number.
 - Input: positive integer

- **serviceId** specifies which service to ping.
 - Input: positive integer

Return Value: If the call succeeds, then a ping reply is returned as a JSON string.

Example:

```
JSON = {
  "apiCall": "ping",
  "bpVersion": 7,
  "nodeId": 2,
  "serviceId": 1
}
```

9 Simulations

HDTN can be simulated using DtnSim, a simulator for delay tolerant networks built on the OMNeT++ simulation framework. Use the “support-hdtn” branch of DtnSim which can be found in the official DtnSim repository¹. HDTN simulation with DtnSim has only been tested on Linux (Debian and Ubuntu). Follow the readme instructions for HDTN and DtnSim to install the software. Alternatively, a pre-configured Ubuntu VM is available for download here² (the username is hdtnsim-user, and the password is grc). More Details about the installation steps can be found here³.

10 HDTN Applications

10.1 BPGen

BPGen is a tool that generates bundles of any specified size, and it is intended to be used with its receiving tool BPSink. It is not a component of HDTN, but it does share the same codebase and libraries as HDTN; hence it can use any of the convergence layers available to HDTN. Additionally, the config file that BPGen uses shares the outductsConfig part of the HDTN config file. If a user desires BPv6 custody transfer support, BPGen supports passing in an additional separate config file containing the inductsConfig part of an HDTN config file; this additional config file is necessary when using unidirectional (non-bidirectional) convergence layers. The BPGen source code is very small because it derives from a helper C++ class called BPSourcePattern and overrides two virtual methods. Users who wish to write a utility that generates bundles as fast as possible and then terminates can model after the BPGen source code.

10.2 BPSink

BPSink receives and validates the bundles sent from BPGen (and possibly bundles that were forwarded from the HDTN Egress). It is not a component of HDTN, but it does share the same codebase and libraries as HDTN; hence it can use the convergence layers available to HDTN. Additionally, the config files that BPSink uses shares the inductsConfig part of the HDTN config file. If a user desires BPv6 custody transfer support, BPSink supports passing in an additional separate config file containing the outductsConfig part of an HDTN config file; this additional config file is necessary when using unidirectional (non-bidirectional) convergence layers. The BPSink source code is very small because it derives from a helper C++ class called BPSinkPattern and overrides one virtual method; any users that want to write their own utility that receives bundles indefinitely and terminates cleanly with a SIGINT signal can simply model after the BPSink source code.

¹<https://bitbucket.org/lcd-unc-ar/dtnsim/src/master/>

²https://drive.google.com/file/d/1dSjxKIZ03U-gsAnMMzcizHAW_gFkDZDe/view

³https://docs.google.com/document/d/1KrKyO_pr-v9CeS5n_ectpfWtwkYL40PdLICGh-24zZY/edit#

10.3 BPSendFile

BPSendFile is a tool that sends either a single file or a directory of files (with recursion), and it takes those file(s) and breaks them into max specified size bundles, and it is intended to be used with its receiving tool BPreceiveFile. It can remain open after all files have been sent to monitor those directories (up to a user-specified max recursion depth) for newly added files. It (like BPGen) is not a component of HDTN, but it does share the same codebase and libraries as HDTN; hence it can use any of the convergence layers available to HDTN. The config files used are the same as BPGen. The BPSendFile source code is very small because it derives from a helper C++ class called BPSourcePattern and overrides three virtual methods; any users that want to write their utility that generates bundles as fast as possible and then remains open and episodically sends new bundles can model after the BPSendFile source code.

10.4 BPreceiveFile

BPreceiveFile tool receives the bundles sent from BPSendFile in any order and reassembles the file fragments, closes the file when all file fragments have been received and writes them to a user-specified directory. It is not a component of HDTN, but it does share the same codebase and libraries as HDTN; hence it can use any of the convergence layers available to HDTN. The config files used and the derived C++ classes used are the same as BPSink.

10.5 BPing

BPing is a tool that generates ping bundles intended to be used with any bundling agent that supports an echo service. HDTN has an echo service with a service number specified in its config as myBpEchoServiceId. All HDTN apps (BPSink and BPreceiveFile) that inherit from BPSinkPattern have an echo service number of 2047. BPing is not a component of HDTN, but it does share the same codebase and libraries as HDTN; hence it can use any of the convergence layers available to HDTN. The config files used are the same as BPGen. BPing does not support custody transfer, but it must also use the same config file BPGen would use for BPv6 custody transfer support in order for BPing to receive an echo bundle back; this additional config file is necessary when using unidirectional (non-bidirectional) convergence layers. The BPing source code is very small because it derives from a helper C++ class called BPSourcePattern and overrides two virtual methods; any user that wants to write their own utility that generates a single bundle, waits for a response, and then either terminates or continues can simply model after the BPing source code.

10.6 Fprime Applications

The BpSendPacket tool receives a data payload over UDP or STCP, extracts the payload, bundles it, and sends it over any convergence layer supported by HDTN. The BpReceivePacket tool does the inverse. By configuring these tools and connecting them to an application, HDTN can provide bundle sending and receiving services to any application that can communicate over a UDP or STCP socket, including an Fprime application. Connecting one of these tools to an Fprime application via STCP requires using an Fprime TCP client configured to use STCP framing or deframing.

11 Runscript

One run script file (in JSON format) is required per each instance of HDTN. Each file starts with assigning path variables, starting up Egress (the outduct), and the bpsink (the method of where bundles will be received). Each file ends with starting up Ingress, assigning bpgen (the process by which bundles can be generated), and a cleanup procedure (optional). Within each file, two different setups can be found depending if the setup is cut-through, i.e. by-passes Storage or utilizes Storage and, in turn, the Router. The cut-through option requires the link to be up and no BPv6 custody. Note: To avoid starting up Egress and Ingress separately, a method called hdtm-one-process combines the two with one command. An example runscript can be found at `HDTN/runscript.sh`.

11.1 Path Variables

The Path Variables section of each run script file points to the locations of all required configuration files. This way, one needs only to edit this section instead of each section if only changing a configuration file. The path variables contain locations of:

- `config_files`
- `hdtm_config`
- `bpsec_config`
- `sink_config`
- `gen_config`

The section ends with a `cd $HDTN_SOURCE_ROOT` an already declared variable in your linux path. This command is here if the run script is not in the HDTN source root directory (where HDTN runs).

11.2 BpSink

A typical instantiation of BpSink is:

```
./build/common/bpcodec/apps/bpsink-async --my-uri-eid=ipn:2.1 --inducts-config-file=$sink_config --bpsec-config-file=$bpsec_config &
```

The command has three main parts: (1) the location of the bpsink code within HDTN, (2) the endpoint ID number, and (3) the inducts configuration file location. The `ipn:2.1` shows, in this particular case, the receiving node is the second endpoint. The `$sink_config` response ties to the respectable Path Variable. For more information about the sink configuration file see section 12.2. `bpsec-config-file` is an optional parameter that is only required if BPsec is enabled and bpsink node is a security acceptor. This option specifies the location of the BPsec Config file which has the policy rules and security operation failure events handling. To end bpsink, it is recommended to insert a 3 second pause command to initialize before the configuration file starts the next section, i.e. `sleep 3`.

11.3 BpReceiveFile

A typical instantiation of BpReceiveFile on the receiving node is: `./build/common/bpcodec/apps/bpreceivefile --save-directory=received --my-uri-eid=ipn:2.1 --inducts-config-file=$sink_config --bpsec-config-file=$bpsec_config &`

The command has five main parts: (1) the new directory where the received data is saved(2) (3) the receiving node endpoint id, (5) the inducts configuration file location. The `$gen_config` response ties to the respectable Path Variable. For more information about the gen configuration file see section 12.2.

`bpsec-config-file` is an optional parameter that is only required if BPsec is enabled and bpreceivefile node is a security acceptor. This option specifies the location of the BPsec Config file which has the policy rules and security operation failure events handling.

To end bpreceivefile, it is recommended to insert an 8 seconds pause command to initialize before the configuration file starts the next section, i.e. `sleep 8`.

11.4 Egress

A typical instantiation of Egress is:

```
./build/module/egress/hdtm-egress-async --hdtm-config-file=$hdtm_config &
```

The command has two main parts: (1) the location of the egress code within HDTN and (2) the HDTN configuration file that the egress code requires. For more information about the HDTN configuration file see section 12.1. To end Egress, it is recommended to insert a 3 second pause command to initialize before the configuration file starts the next section, i.e. `sleep 3`.

11.5 Router

A typical instantiation of the Router in a runscript is: `./build/module/router/hdtn-router --contact-plan-file=contactPlan.json --hdtn-config-file=$hdtn_config &`

The command has three main parts: (1) the location of the Router code within HDTN, (2) the contact plan and (3) the same HDTN configuration file that the Egress required. For more information about the HDTN configuration file see section 12.1. To end the Router, it is recommended to insert a 1 second pause command to initialize before the configuration file starts up the next section, i.e. `sleep 1`. An example of a Routing scenario test with 4 HDTN nodes was added under `$HDTN_SOURCE_ROOT/test/test_script_s_linux/Routing_Test`

11.6 Ingress

A typical instantiation of Ingress is:

```
./build/module/ingress/hdtn-ingress --hdtn-config-file=$hdtn_config --bpsec-config-file=$bpsec_config &
```

The command has two main parts: (1) the location of the Ingress code within HDTN and (2) the HDTN configuration file that the ingress code requires. For more information about the HDTN configuration file see section 12.1. `bpsec-config-file` is an optional parameter that is only required if BPsec is enabled and this HDTN node is a security source, verifier or acceptor. This option specifies the location of the BPsec Config file which has the policy rules and security operation failure events handling. To end Ingress, it is recommended to insert a 3 second pause command to initialize before the configuration file starts the next section, i.e. `sleep 3`.

11.7 Storage

A typical instantiation of Storage is:

```
./build/module/storage/hdtn-storage --hdtn-config-file=$hdtn_config &
```

The command has two main parts: (1) the location of the storage code within HDTN and (2) the HDTN configuration file that the storage code requires. For more information about the HDTN configuration file see section 12.1. To end Storage, it is recommended to insert a 3 second pause command to initialize before the configuration file starts the next section, i.e. `sleep 3`.

11.8 HDTN One Process

A typical instantiation of `hdtn-one-process` is:

```
./build/module/hdtn_one_process/hdtn-one-process --hdtn-config-file=$hdtn_config --contact-plan-file=contactPlan.json --bpsec-config-file=$bpsec_config &
```

The command has three main parts: (1) the location of the `hdtn-one-process` code within HDTN, (2) the HDTN configuration file that the `hdtn-one-process` code requires and (3) the contact plan. For more information about the HDTN configuration file see section 12.1. `bpsec-config-file` is an optional parameter that is only required if BPsec is enabled and this HDTN node is a security source, verifier or acceptor. This option specifies the location of the BPsec Config file which has the policy rules and security operation failure events handling.

11.9 BpGen

A typical instantiation of BpGen is:

```
./build/common/bpcodec/apps/bpgen-async --bundle-rate=100 --my-uri-eid=ipn:1.1 --dest-uri-eid=ipn:2.1 --duration=40 --outducts-config-file=$gen_config &
```

The command has six main parts: (1) the location of the `bpgen` application code within HDTN, (2) the bundle rate, (3) the endpoint ID, (4) the destination endpoint ID, (5) the duration, and (6) the gen configuration file that the `bpgen` code requires. For more information about the gen configuration file see section 12.3. In the example instantiation, the bundle rate was designated for 100 bundles per second with the `bpgen`'s endpoint ID being the first node (`ipn:1.1`) and is sending to the second node (`ipn:2.1`), which

matches the bpsink's endpoint ID. The duration value is in seconds, and in this case, 40 seconds. To end bpgen, it is recommended to insert a 8 pause command to initialize before the configuration file starts the next section, i.e. `sleep 8`.

if BPsec is enabled and BpGen node is a security source, it's required to add the option `--bpsec-config-file=$bpsec_config` to specify the BPsec Config file which has the policy rules and security operation events handling.

11.10 BpSendFile

A typical instantiation of BpSendFile on the sending node is: `./build/common/bpcodec/apps/bpsend-file --max-bundle-size-bytes=500000 --file-or-folder-path=./flightdata --my-uri-eid=ipn:1.1 --dest-uri-eid=ipn:2.1 --outducts-config-file=$gen_config --bpsec-config-file=$bpsec_config --upload-new-files &`

The command has five main parts: (1) the maximum bundle size (2) the location of the folder or file to be sent, (3) the current node endpoint id, (4) the destination node endpoint id, (5) the outducts configuration file location. The `$gen_config` response ties to the respectable Path Variable. For more information about the gen configuration file see section 12.3.

`bpsec-config-file` is an optional parameter that is only required if BPsec is enabled and bpsendfile node is a security source. This option specifies the location of the BPsec Config file which has the policy rules and security operation failure events handling.

`upload-new-files` is an optional parameter that can be used to continue sending files when more files are added to the folder being sent.

To end BpSendFile, it is recommended to insert an 8 seconds pause command to initialize before the configuration file starts the next section, i.e. `sleep 8`.

11.11 Bping

A typical instantiation of Bping is:

```
./build/common/bpcodec/apps/bping --my-uri-eid=ipn:1.1 --dest-uri-eid=ipn:2.2047 --outducts-config-file=$ping_config &
```

The command has six main parts: (1) the location of the bping application code within HDTN, (2) the endpoint ID, (3) the destination endpoint ID, and (4) the configuration file that the bping code requires is the same as BpGen config file. For more information about the gen configuration file see section 12.3. In the example instantiation, node is sending ping bundles to the node 2 (using the echo service number 2047).

11.12 BpSendPacket

A typical instantiation of BpSendPacket is:

```
./build/common/bpcodec/apps/bpsendpacket --use-bp-version-7 --my-uri-eid=ipn:1.1 --dest-uri-eid=ipn:2.1 --outducts-config-file=$gen_config --packet-inducts-config-file=$send_config &
```

The arguments are repeated from the other applications in this section and have the same meaning, except for the `packet-inducts-config-file` argument, which is new. It refers to a config for an induct that will be used to receive a data payload from a local UDP or STCP client.

11.13 BpReceivePacket

A typical instantiation of BpReceivePacket is:

```
./build/common/bpcodec/apps/bpreceivepacket --my-uri-eid=ipn:2.1 --inducts-config-file=$sink_config --packet-outducts-config-file=$receive_config &
```

The `packet-outducts-config-file` argument is new and refers to config for an outduct that will be used to deliver a data payload to a local application listening on a UDP or STCP socket.

11.14 CleanUp

If HDTN only needs to run for a certain amount of time and then end, add a line under all other sections (minus the path variables) after the instantiation command in the format of `< SectionName > _PID = $!` For example, after `bpngen`'s instantiation, the cleanup command will be: `bpngen_PID=$!`

Within the clean-up section, wait for HDTN to run. Then, from the bottom to the top of the configuration file sections, end them via format of `kill -2 $ < PID name >`. A wait statement for at least 2 seconds between each kill command is included. Clean-up script example:

```
sleep 30
echo "\nKilling bpngen..." && kill -2 $bpngen_PID
sleep 2
echo "\nKilling HDTN storage..." && kill -2 $storage_PID
sleep 2
echo "\nKilling HDTN ingress..." && kill -2 $ingress_PID
sleep 2
echo "\nKilling router..." && kill -9 $router_PID
sleep 2
echo "\nKilling egress..." && kill -2 $egress_PID
sleep 2
echo "\nKilling bpsink..." && kill -2 $bpsink_PID
```

This example will run HDTN for 30 seconds before closing HDTN.

The following options can be added: (1) `--bpsec-config-file=$bpsec.config`: if BPSec is enabled and this HDTN node is a security source, acceptor or verifier it's required to add this option to specify the BPSec Config file which has the policy rules and security operation events handling. (2) `use-unix-timestamp` to use a contact plan with unix timestamp and (3) `use-mgr` to use Multigraph Routing Algorithm instead of the default CGR Dijkstra routing Algorithm.

To end `hdtm-one-process`, it is recommended that users insert a 10 second pause command to initialize before the configuration file starts the next section, i.e. `sleep 10`.

Note: When using the `hdtm-one-process`, the runsript does not need to instantiate the Egress, Storage, Ingress, and Router separately.

12 Config Files

12.1 hdtm_config

The typical HDTN configuration file instantiation can be seen below. All values are default and changeable. More information on each line can be seen bulleted.

```
"hdtmConfigName": my hdtm config,
  - User description of config file
"userInterfaceOn": true,
  - When compiled determines if the interface is displayed
"mySchemeName": "unused_scheme_name",
  - DTN scheme name
  - Vestigial, still needs to be defined
"myNodeId": 10,
  - Node running ID
  - Must be an integer
"myBpEchoServiceId": 2047,
  - Service number to ping if user wants to ping HDTN
  - Must be an integer
"myCustodialSsp": "unused_custodial_ssp",
  - Custodial scheme specific part
  - Vestigial, still needs to be defined
"myCustodialServiceId": 0,
```

- Service ID where custody reports are sent to HDTN
 - Must be an integer
-

"isAcsAware": true,
- Aggregate Custody Signals (ACS)
- Specifies if HDTN is to use ACS
- Must be a Boolean

"acsMaxFillsPerAcsPacket": 100,
- How many custody signals to be packed into one ACS packet
- Must be an integer

"acsSendPeriodMilliseconds": 1000,
- Aggregation time in ms
- Must be an integer

"retransmitBundleAfterNoCustodySignalMilliseconds": 10000,

"maxBundleSizeBytes": 10000000,
- The maximum size of the bundle HDTN can receive or send in Bytes
- NOTE: if the bundle is larger than this, the bundle will be dropped.

"bufferRxToStorageOnLinkUpSaturation": false,
-
- Must be a Boolean

"maxIngressBundleWaitOnEgressMilliseconds": 2000,
- During Cut-through, if egress cannot finish a bundle within this time it will give up cut-through and send to storage; in ms

"maxLtpReceiveUdpPacketSizeBytes": 65536,
- Maximum packet size in bytes that can be received by HDTN
- Set to the largest datagram the protocol will see on the network.
- 65536 is the typical max size of a local UDP packet will support
- This is unneeded if not using LTP as an irrelevant value

"neighborDepletedStorageDelaySeconds": 0,
- If non-zero, re-route around neighbors with depleted storage
- Attempt to send bundles to depleted neighbor after specified value in seconds
- Requires custody to detect depleted storage on neighboring nodes
- Setting to zero disables

"enforceBundlePriority": false,
- If set to true, enables strict priority ordering of forwarded bundles
- Lower priority bundles cannot be sent via cutthrough queues before higher priority bundles in storage
- Disables ingress-to-egress cutthrough queue, likely decreasing performance

"fragmentBundlesLargerThanBytes": 0,
- If non-zero, HDTN will attempt to fragment BPv6 bundles with payloads larger than value
- Does not fragment BPv7 bundles nor those with the NOFRAGMENT flag set

"zmqBoundRouterPubSubPortPath": 10200,
- ZMQ bound port of the router ZMQ pub-sub socket
- Must be an integer

"zmqBoundTelemApiPortPath": 10305,
- ZMQ bound port of the API socket
- Must be an integer

"inductsConfig": {
 "inductConfigName": "myconfig",
 "inductVector": [
 {

```

        "name": "stcp_ingress",
        "convergenceLayer": "stcp",
        "boundPort": 4556,
        "numRxCircularBufferElements": 200,
    }
]
},

```

- inductConfigName and name are for user comment
- Can choose within the convergence layer: stcp, tcpcl_v3, tcpcl_v4, udp, ltp_over_udp
- boundPort and numRxCircularBufferElements must be integers
- NOTE: numRxCircularBufferElements differs for each convergence layer. STCP this represents the number of bundles to buffer up; TCPCL this is the number of bundles or data fragments; LTP this is the number of UDP packets; UDP this is the number of packets/bundles.

```

"outductsConfig": {
  "outductConfigName": "myconfig",
  "outductVector": [
    {
      "name": "stcp_egress",
      "convergenceLayer": "stcp",
      "nextHopNodeId": 2,
      "remoteHostname": "localhost",
      "remotePort": 4558,
      "maxNumberOfBundlesInPipeline": 50,
      "maxSumOfBundleBytesInPipeline": 50000000,
    }
  ]
},

```

- outductConfigName and name are for user comment
- Can choose within the convergence layer: stcp, tcpcl_v3, tcpcl_v4, udp, ltp_over_udp
- remoteHostname is the IP or hostname that HDTN is sending bundles to. remotePort is the port HDTN is sending bundles to.
- Final destination IDs can be multiple or one IPN URIs. IPN service number can be an * for a service wildcard.
- nextHopNodeID, remotePort, maxNumberOfBundlesInPipeline, and maxSumOfBundleBytesInPipeline must be integers.

```

"storageConfig": {
  "storageImplementation": "asio_single_threaded",
  "tryToRestoreFromDisk": false,
  "autoDeleteFilesOnExit": true,
  "totalStorageCapacityBytes": 8192000000,
  "storageDeletionPolicy": "never",
  "storageDiskConfigVector": [
    {
      "name": "d1",
      "storeFilePath": ".\store1.bin"
    },
    {
      "name": "d2",
      "storeFilePath": ".\store2.bin"
    }
  ]
},

```

- storageImplementation has 2 options: asio_single_threaded and stdio_multi_threaded. Default to asio_single_threaded.
- tryToRestoreFromDisk: if the bundle storage was left used, when HDTN reloads it can restore the state
- autoDeleteFilesOnExit: tells HDTN, when cleanly closed, to delete or save all storage bundles
- totalStorageCapacityBytes: storage module capacity or quota for bundles, in Bytes. NOTE: each storage item in the storageDiskConfigVector must be able to hold totalStorageCapacityBytes divided by the number items in the total storageDiskConfigVector
- storageDeletionPolicy: policy for deleting expired bundles from disk. Options are “never”: never delete expired bundles, “on_expiration”: delete bundles as the expire, and “on_storage_full”: delete all expired bundles when storage reaches 90% capacity.
- storageDiskConfigVector is a striping scheme similar to RAID 0 but not using RAID itself. NOTE: can have unlimited storage vector size i.e. number of storeFilePath(s).

Miscellaneous notes for the HDTN configuration file:

- Depending on the convergence layer there may be additions to the “inductVector”, “outductVector”. See section 14 for details.
- Ingress, Egress, and/or storage are optional additions to the HDTN configuration file depending on the HDTN node need.

12.2 sink_config

A typical sink configuration file includes an “inductConfigName” and an “inductVector”. The “inductVector” requires the name, convergence layer, the bound port number, and the number of received circular buffer elements. For details about this section, see 12.1 since the bp_sink config file is a copy of the inductConfigName/inductVector of the hdtm_config file. This is because the bp_sink config file goes to the BPSink application detailed in section 10.2. Note: Depending on the convergence layer there may be additions to the “inductVector”. See section 14 for details. Structure example:

```
"inductsConfig": {
  "inductConfigName": "myconfig",
  "inductVector": [
    {
      "name": "stcp-ingress",
      "convergenceLayer": "stcp",
      "boundPort": 4556,
      "numRxCircularBufferElements": 200,
    }
  ]
},
```

12.3 gen_config

A typical gen configuration file includes an “outductConfigName” and an “outductVector”. The “outductVector” requires: the name, convergence layer, next hop, remote hostname, remote port, bundle pipeline limit, and the final destination endpoint ID. For details about this section see 12.1 since the gen_config file is a copy of the outductConfigName/outductVector of the hdtm_config file. This is due to the gen_config file going to the BPGen application detailed in section 10.1 Note: depending on the convergence layer there may be additions to the “outductVector”. See section 14 for details. Structure example:

```
{
  "outductConfigName": "myconfig",
  "outductVector": [
    {
```

```

        "name": "bpgen",
        "convergenceLayer": "tcpcl_v3",
        "nextHopNodeId": 10,
        "remoteHostname": "localhost",
        "remotePort": 4556,
        "maxNumberOfBundlesInPipeline": 5,
        "maxSumOfBundleBytesInPipeline": 50000000,
    }
]
}

```

12.4 bpsec_config

A typical BPsec configuration file includes a “bpSecConfigName” which is user description of the config file, “policyRules” and “securityFailureEventSets” vectors which include the BPsec security policy rules and the security operation failure events handling.

Here’s the overall structure example:

```

{
  "bpsecConfigName": "my BPsec Config",
  "policyRules": [
    {
      "description": "Bpsource confidentiality",
      "securityPolicyRuleId": 1,
      "securityRole": "source",
      "securitySource": "ipn:1.1",
      "bundleSource": [
        "ipn:1.1"
      ],
      "bundleFinalDestination": [
        "ipn:2.1"
      ],
      "securityTargetBlockTypes": [
        1
      ],
      "securityService": "confidentiality",
      "securityContext": "aesGcm",
      "securityFailureEventSetReference": "default_confidentiality",
      "securityContextParams": [
        {
          "paramName": "aesVariant",
          "value": 256
        },
        {
          "paramName": "ivSizeBytes",
          "value": 12
        },
        {
          "paramName": "keyFile",
          "value": "config_files/bpsec/ipn1.1_confidentiality.key"
        },
        {
          "paramName": "securityBlockCrc",
          "value": 0
        }
      ],
    }
  ],
}

```

```

        {
            "paramName": "scopeFlags",
            "value": 7
        }
    ]
}
],
"securityFailureEventSets": [
    {
        "name": "default_confidentiality",
        "description": "default bcb confidentiality security operations event set",
        "securityOperationEvents": [
            {
                "eventId": "sopCorruptedAtAcceptor",
                "actions": [
                    "removeSecurityOperationTargetBlock"
                ]
            }
        ]
    }
]
}
]
}

```

More information on each line of the policy rules vector is detailed below:

```

"description": "Bpsource confidentiality",
    - User description of bpsec policy rule file
"securityPolicyRuleId": 1,
    - Policy rule Id
"securityRole": "source",
    - security role can be configured as "source", "verifier" or "acceptor"
"securitySource": "ipn:1.1",
    - security source Endpoint ID

```

```

"bundleSource": [
    "ipn:1.1"
],

```

- a vector of bundle source endpoint IDs (wildcard can also be used).

```

"bundleFinalDestination": [
    "ipn:2.1"
],

```

- a vector of bundle final destination endpoint IDs (wildcard can also be used).

```

"securityTargetBlockTypes": [
    1
],

```

- a vector of all the security Target Block Types, in this example 1 corresponds to the payload block
 - bundle Source Endpoint ID

```

"securityService": "confidentiality",
    - Security service could be either "confidentiality" or "integrity"
"securityContext": "aesGcm",
    - Security Context can be "aesGcm" for confidentiality or "hmacSha" for integrity

```

```
"securityContextParams": [
  {
    "paramName": "aesVariant",
    "value": 256
  },

```

- Security Context parameters is a vector of security context parameters and their values. The parameters and their possible values are listed below:

- "aesVariant" possible values are 128, 256
- "shaVariant" possible values are 256, 512 or 384
- "ivSizeBytes" possible values are 12 or 16
- "keyFile" specifies the location of the key
- "keyEncryptionKeyFile" specifies the location of the key-encryption key
- "securityBlockCrc" defines optional Cyclic Redundancy Check (CRC) to identify block corruption
- "scopeFlags" defines any additional data to be included along with the block-type-specific data

```
"securityFailureEventSetReference": "default_confidentiality",
  - Reference to the name of the security Failure Event Set to be used
```

More information on each line of the security Failure Event Sets vector is detailed below:

```
"name": "default_confidentiality",
  - name/reference of the security Failure Event Set
"name": "description",
  - user description of the event set
```

```
"securityOperationEvents": [
  {
    "eventId": "sopCorruptedAtAcceptor",
    "actions": [
      "removeSecurityOperationTargetBlock"
    ]
  }

```

```
"name": "securityOperationEvents",
  - Security operation events vector which has security event Ids and their corresponding actions
  - The supported security events are "sopMisconfiguredAtVerifier", "sopMissingAtVerifier", "sopCorruptedAtVerifier", "sopMissingAtAcceptor", "sopCorruptedAtAcceptor"
  - The supported actions are "removeSecurityOperation", "removeSecurityOperationTargetBlock", "removeAllSecurityTargetOperations", "failBundleForwarding", "requestBundleStorage", "reportReasonCode", "overrideSecurityTargetBlockBpcf", "overrideSecurityBlockBpcf"
```

12.5 distributed_config

If you are running HDTN in distributed mode, you will need to add a command line argument `--hdtndistributed-config-file` as shown in `runscript distributed`⁴ and `hdtndistributed defaults`⁵.

```
"zmqIngressAddress": "localhost",
  - IP or hostname of the machine running the Ingress module of HDTN
"zmqEgressAddress": "localhost",
  - IP or hostname of the machine running the Egress module of HDTN
"zmqStorageAddress": "localhost",
  - IP or hostname of the machine running the Storage module of HDTN
"zmqRouterAddress": "localhost",
  - IP or hostname of the machine running the Router module of HDTN
```

⁴https://github.com/nasa/HDTN/blob/master/tests/test_scripts_linux/runscript_distributed.sh

⁵https://github.com/nasa/HDTN/blob/master/config_files/hdtndistributed_defaults.json

"zmqBoundIngressToConnectingEgressPortPath": 10100,
 - ZMQ bound TCP port of the Ingress module for internal messages sent from Ingress to Egress
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingEgressToBoundIngressPortPath": 10160,
 - ZMQ bound TCP port of the Ingress module for internal messages sent from Egress to Ingress
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqBoundEgressToConnectingRouterPortPath": 10162,
 - ZMQ bound TCP port of the Router module for internal link down messages sent from Egress to Router
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingEgressBundlesOnlyToBoundIngressPortPath": 10161,
 - ZMQ bound TCP port of the Ingress module for internal TCPCL opportunistic bundles sent from Egress to Ingress
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqBoundIngressToConnectingStoragePortPath": 10110,
 - ZMQ bound TCP port of the Ingress module for internal messages sent from Ingress to Storage
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingStorageToBoundIngressPortPath": 10150,
 - ZMQ bound TCP port of the Ingress module for internal messages sent from Storage to Ingress
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingStorageToBoundEgressPortPath": 10120,
 - ZMQ bound TCP port of the Egress module for internal messages sent from Storage to Egress
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqBoundEgressToConnectingStoragePortPath": 10130,
 - ZMQ bound TCP port of the Egress module for internal messages sent from Egress to Storage
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingRouterToBoundEgressPortPath": 10210,
 - ZMQ bound TCP port of the Egress module for internal messages sent from Router to Egress
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingTelemToFromBoundIngressPortPath": 10301,
 - ZMQ bound TCP port of the Ingress module for internal messages sent from Ingress to Telemetry module (GUI)
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

"zmqConnectingTelemToFromBoundEgressPortPath": 10302,

- ZMQ bound TCP port of the Egress module for internal messages sent from Egress to Telemetry module
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer
- "zmqConnectingTelemToFromBoundStoragePortPath": 10303
- ZMQ bound TCP port of the Storage module for internal messages sent from Storage to Telemetry module
 - NOTE: This value is unused when using hdtm-one-process; still needs to be defined.
 - NOTE: TCP is unidirectional in ZMQ
 - Must be an integer

13 Contact Plans

The contact plan is a JSON file which has a list of all forthcoming contacts for all the nodes in the network. The contact plans are accessible under HDTN/module/router/contact_plans/src

13.1 JSON Fields

- "contact": 1
- Identification number of the contact.
 - Integer
- "source": 10
- Source/Sending node number for that contact.
 - Integer
- "dest": 2
- Destination/Receiving node number for that contact or next hop.
 - Integer
- "startTime": 25
- This is the time, in seconds, after which the link is UP for that contact (start transmission time)
 - Integer
- "endTime": 38
- This is the time, in seconds, after which link will be DOWN for that contact (end transmission time)
 - Integer
- "rateBitsPerSec": 1000
- The data rate in bits per second
 - Integer
- "owlt": 1
- One Way Light Time which is the distance (range) expressed in light-seconds
 - Integer

14 Convergence Layers and Routing Protocols

14.1 Overview of Compatible Convergence Layers

BP (Bundle Protocol) is used in space and other areas that experience intermittent connectivity and/or long latencies. Figure 5 shows how BP fits into the protocol stack. The BP software included two major revisions between version 6 and version 7. The HDTN software can understand both versions depending if it is being used with current and legacy assets or future assets slated to use BPv7. Users may select from the convergence layers listed below based on several factors, including the estimated round trip time, estimated link rates, need for reliable transport, security requirements, and underlying network protocol stack specific to their use-case.

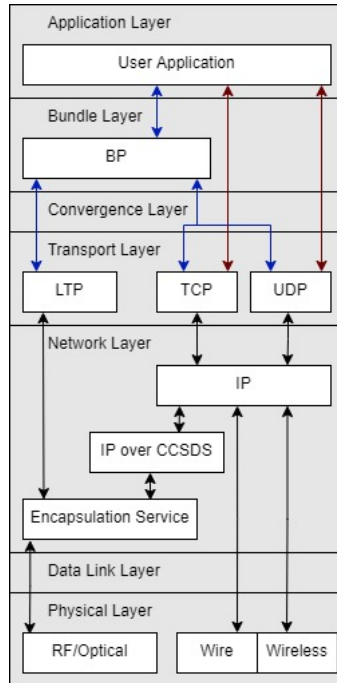


Figure 5.—Simplified Protocol Stack.

TCPCL (Transmission Control Protocol (TCP) Convergence Layer) is utilized in space applications for users using Internet Protocol (IP). In addition, this convergence layer provides a bridge from the Bundle Protocol (BP), if hops are required, to get to the destination. TCP requires acknowledgment before the message is sent and, for space, can be inefficient. HDTN currently uses TCP version 4.

UDP (User Datagram Protocol) is like TCP, except UDP does not require acknowledgment before sending a message. This protocol is over IP but can be used with BP to enable hops, like TCP.

LTP (Licklider Transmission Protocol) is the main transport layer for BP. Therefore, no matter how many hops or delays the message goes through to arrive at its destination, the message will remain intact. HDTN's LTP is compatible with BP version 6 with and without custody.

STCP (Simple TCP) is a DTN simplified TCP convergence-layer adapter. This means STCP utilizes standard TCP connections but is topologically adjacent in the BP network to transmit BP bundles between nodes.

14.2 Additions to Config Files

Each convergence layer has additional "inductVector" and "outductVector" configuration fields. These additions apply to all config files that utilize "inductVector" and "outductVector", e.g. hdtm_config, sink_config, and gen_config. The additions are listed in the sections below.

14.2.1 TCPCLv3

Common induct and outduct fields:

"keepAliveIntervalSeconds": 15

- This is the minimum interval, in seconds, to negotiate as the Session Keepalive. See RFC7242 Section 5.6.
- Integer

"tcpclV3MyMaxTxSegmentSizeBytes": 200000

- This is the maximum segment size, in bytes, to use for transmitting data segments.
- Integer

Induct fields:

"numRxCircularBufferBytesPerElement": 100
- This is the maximum size, in bytes, of each element in the circular receive buffer.
- Integer

Outduct fields:

"tcpclAllowOpportunisticReceiveBundle": false
- This is whether to allow receiving opportunistic bundles.
- Boolean

14.2.2 TCPCLv4

Common induct and outduct fields:

"keepAliveIntervalSeconds": 15
- This is the minimum interval, in seconds, to negotiate as the Session Keepalive. See RFC7242 Section 5.6.
- Integer
"tcpcl4MyMaxTxSegmentSizeBytes": 200000
- This is the maximum segment size, in bytes, to use for transmitting data segments.
- Integer
"tlsIsRequired": false
- This is whether TLS (Transport Layer Security) is required.
- Boolean

Induct fields:

"numRxCircularBufferBytesPerElement": 100
- This is the maximum size, in bytes, of each element in the circular receive buffer.
- Integer
"tcpclV4MyMaxRxSegmentSizeBytes": 20000,
- This is the maximum segment size, in bytes, to use for transmitting data segments.
- Integer
"certificatePemFile": "C:hdtm_ssl_certificatescert.pem"
- This is a path to the file containing the certificate for SSL (Secure Socket Layer).
"privateKeyPemFile": "C:hdtm_ssl_certificatesprivatekey.pem"
- This is a path to the file containing the private key for SSL.
"diffieHellmanParametersPemFile": "C:hdtm_ssl_certificatesdh4096.pem"
- This is a path to the file containing the Diffie-Hellman parameters for TLS.

Outduct fields:

"tryUseTls": false
- This is whether TLS is required.
- Boolean
"useTlsVersion1.3": false
- This is whether TLS version 1.3 is required. If not specified, version 1.2 will be used.
- Boolean
"doX509CertificateVerification": false
- This is whether to do X.509 certificate validation is required.
- Boolean
"verifySubjectAltNameInX509Certificate": false
- This is whether to verify the subject alternative name in the X.509 certificate is required.
- Boolean
"certificationAuthorityPemFileForVerification":
"C:hdtm_ssl_certificatescert.pem"

- This is a path to the file containing the certificate authority.

14.2.3 UDPCL

Induct fields:

"numRxCircularBufferBytesPerElement": 100

- This is the maximum size, in bytes, of each element in the circular receive buffer.
- Integer

14.2.4 LTP

Common induct and outduct fields:

"clientServiceId": 1

- This is the ID of the client service.
- Integer

"ltpDataSegmentMtu": 1360

- This is the maximum size of the data portion (excluding LTP headers and UDP headers and IP headers) of an LTP sender's Red data segment being sent. Set this low enough to avoid exceeding ethernet MTU to avoid IP fragmentation.
- Integer

"ltpMaxRetriesPerSerialNumber": 500

- This is the maximum number of retries/resends of a single LTP packet with a serial number before the session is terminated.
- Integer

"ltpMaxUdpPacketsToSendPerSystemCall": 1

- This is the maximum number of UDP packets to send per system call.
- Integer

"ltpRandomNumberSizeBits": 64

- This is whether to use a 32 or 64 bit random number is required.
- Integer (32 or 64)

"oneWayLightTimeMs": 1000

- This is the one way light time. Round trip time (retransmission time) is computed by $(2 * (\text{oneWayLightTime} + \text{oneWayMarginTime}))$.
- millisecond - Integer

"oneWayMarginTimeMs": 200

- This is the one way margin (packet processing) time. Round trip time (retransmission time) is computed by $(2 * (\text{oneWayLightTime} + \text{oneWayMarginTime}))$.
- millisecond - Integer

"remoteLtpEngineId": 20

- This is the ID of the remote LTP engine.
- Integer

"thisLtpEngineId": 10

- This is the ID of this LTP engine.
- Integer

"delaySendingOfReportSegmentsTimeMsOrZeroToDisable": 20

- Time in milliseconds to defer data retransmission in order to efficiently handle out-of-order report segments.
- Integer

"keepActiveSessionDataOnDisk": false

- Supports the running of LTP sessions (both for receivers and senders) from a solid-state disk drive in lieu of keeping session data-segments in memory.
- If this feature is enabled, it also uses the added configuration values `activeSessionDataOnDiskNewFileDurationMs` and `activeSessionDataOnDiskDirectory` to determine where on the drive to temporarily store sessions.

- As this is still experimental, if a LTP link goes down, bundles don't yet get transferred to storage and get dropped.
- Boolean

Induct fields:

- "ltpMaxExpectedSimultaneousSessions": 500
 - This is the number of expected simultaneous LTP sessions for this engine.
 - Integer
- "ltpRemoteUdpHostname": "localhost"
 - This is the remote IP address or hostname.
- "ltpRemoteUdpPort": 4556
 - The remote UDP port
 - Integer
- "ltpRxDataSegmentSessionNumberRecreationPreventerHistorySize": 1000
 - This is the number of recent LTP receiver history of session numbers to remember. If an LTP receiver's session has been closed and it receives a session number, within the history, the receiver will refuse the session to prevent a potentially old session from being reopened, which has been known to happen with IP fragmentation enabled.
 - Integer
- "preallocatedRedDataBytes": 200000
 - ESTIMATED_BYTES_TO_RECEIVE_PER_SESSION
 - The number of Red data contiguous bytes to initialized on a receiver. Make this large enough to accommodate the max Red data size so that the LTP receiver does not have to reallocate, copy, and/or delete data while it is receiving Red data. Make this small enough so that the system does not have to allocate too much extra memory per receiving session.
 - Integer

Outduct fields:

- "ltpCheckpointEveryNthDataSegment": 0
 - This enables accelerated retransmission for an LTP sender by making every Nth UDP packet a checkpoint.
 - Integer
- "ltpSenderBoundPort": 1113
 - This is the bound port of the LTP sender.
 - Integer
- "ltpSenderPingSecondsOrZeroToDisable": 15
 - This is the number of seconds between LTP session sender pings during times of zero data segment activity. An LTP ping is defined as a sender sending a cancel segment of a known non-existent session number to a receiver, in which the receiver shall respond with a cancel ACK in order to determine if the link is active.
 - Integer

14.2.5 STCP

Common induct and outduct fields:

- "keepAliveIntervalSeconds": 17
 - This is the minimum interval, in seconds, to negotiate as the Session Keepalive.
 - Integer

15 Test Configurations and Instructions

15.1 TCP Loopback Test

To run this simple Loopback Test as shown in Figure 6, from the HDTN source directory run the command:

```
./runscript.sh
```

This works by running 3 modules for about 30 seconds:

- BPGen - Generates the bundles and sends them to the Ingress module.
- HDTN One Process - Launches the modules for HDTN as a single process. Since this is a cutthrough mode test, Storage is not used. Ingress, Egress, and Router are run.
- BPSink - Receives the bundle data from Egress.

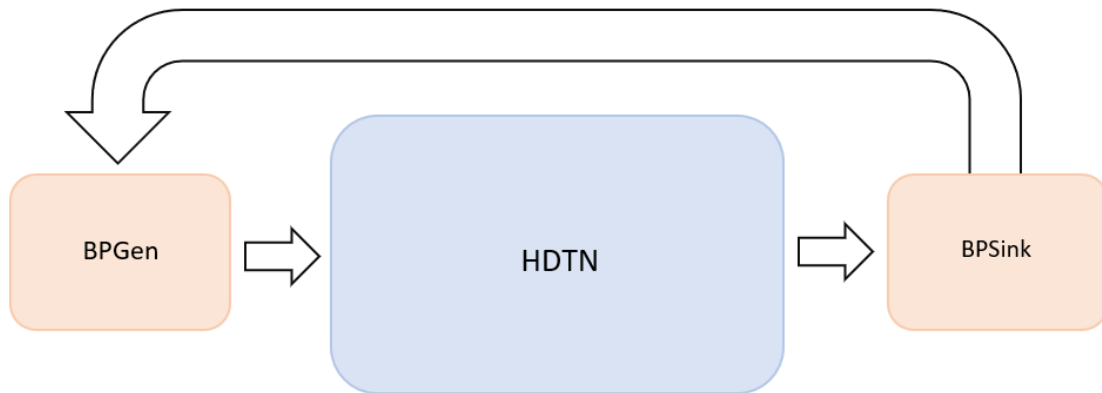


Figure 6.—HDTN Loopback test.

15.2 Two Node LTP Test

This test, shown in Figure 7 relies on having two nodes running HDTN: the sender and the receiver. The sender will run BPGen, and HDTN One Process. The receiver will run BPSink, and HDTN One Process. Example scripts for this can be found under `HDTN/tests/test_scripts_linux/LTP_2Nodes_Test`.

Separate machines with HDTN installed can each run either the sender or receiver. If using these runscripts this way, users should update the `remoteHostname` field in each config file to the proper IP addresses. The config files for these scripts can be found under `HDTN/config_files/hdtN/hdtN_Node1_ltp.json` and `HDTN/config_files/hdtN/hdtN_Node2_ltp.json`.

When running this test, users should start the receiver script before the sender script.

15.3 Four Nodes STCP Test

Shown in Figure 8, this test⁶ relies on having four nodes running HDTN and uses the router module. Node 1 runs BPGen and HDTN One Process. Node 2 and 3 only run HDTN One Process. The final destination Node 4 will run HDTN One Process and BPSink.

⁶https://github.com/nasa/HDTN/tree/master/tests/test_scripts_linux/Routering_Test

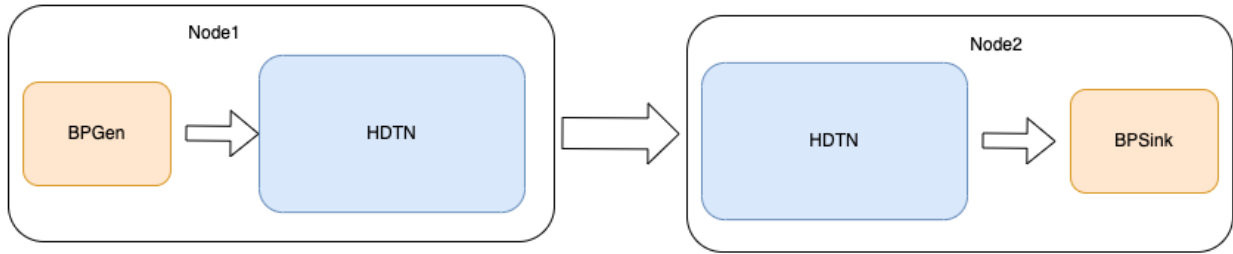


Figure 7.—HDTN two nodes test.

In Node 1’s HDTN config file, the next hop is configured to node 3 originally. After the router computes the optimal route to the final destination, the outduct will select node 2 as next hop instead. At initialization, the HDTN json config file for each node has all possible next hops. If we have multiple hops leading to the same final destination, only one Outduct should be initialized with the final destinations vector, and the other next hops Outducts should be dormant, ie having no values initialized in their final destinations json fields. Router will compute the best route and send an event to Egress to update the Outduct to use the nextHop for that optimal route leading to the final destination.

The runscripts for each node can be found under `HDTN/tests/test_scripts_linux/Routing_Test`. The config files for Node 1 can be found at `HDTN/config_files/hdt/hdt_node1_cfg.json` with the other node config files immediately following it. If running on separate machines, make sure to update the `remoteHostname` field in each config file to the proper IP addresses. Users should start the runscript for each node ordered from receiver to sender, i.e start Node 4, then Nodes 3, 2, and 1.

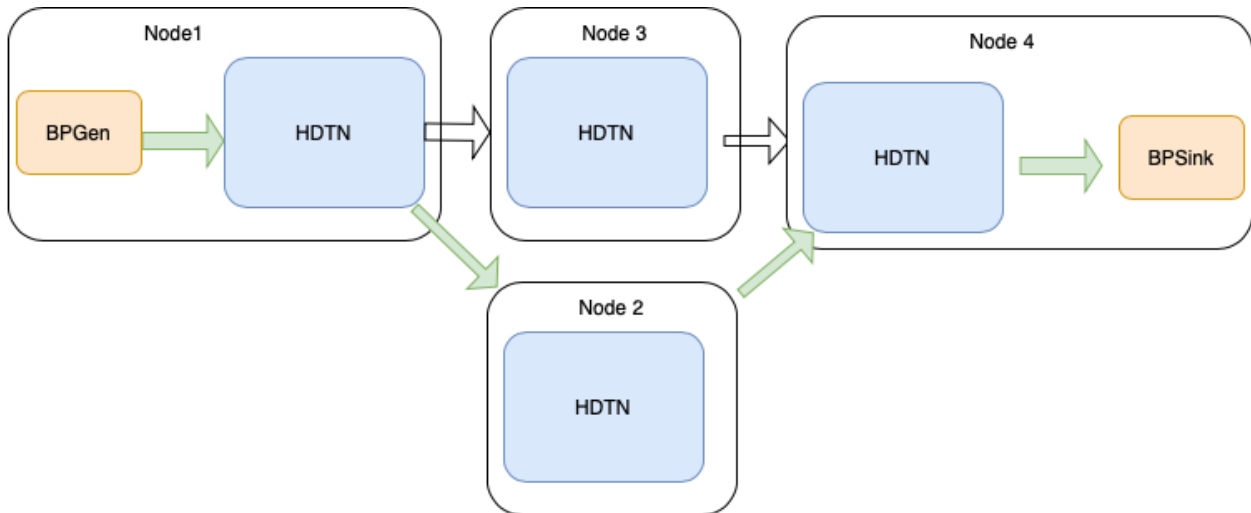


Figure 8.—HDTN Four Nodes STCP Routing test.

15.4 File Transfer Test

This test relies on having two machines running HDTN: the sender and the receiver. The sender will run `BPSendFile`, and `HDTN One Process`. The receiver will run `BPReceiveFile`, and `HDTN One Process`. Example scripts were added to send files using LTP and TCPCL convergence layers. They can be found under `HDTN/tests/test_scripts_linux/LCRD_File_Transfer_Test`. Both BPv6 with and without custody transfer and BPv7 with and without BPsec are supported in this example. Figure 9 shows the file transfer test with BPsec enabled where the source encrypts the plaintext associated with the payload and the destination does the security verification and decryption of the payload. Separate machines with

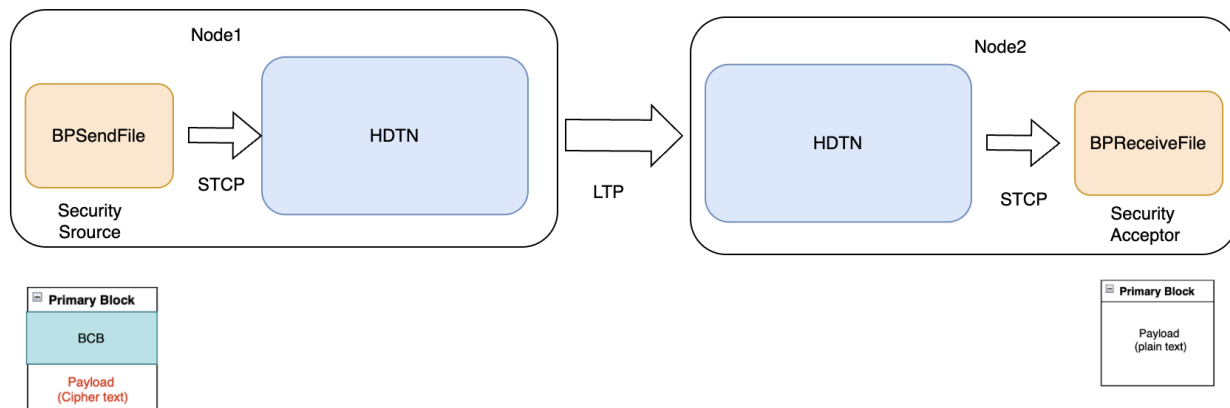


Figure 9.—HDTN File transfer with confidentiality test

HDTN installed can each run either the sender or receiver. User should update the `remoteHostname` field in each HDTN config file in this directory to the proper IP addresses.

When running this test, users should start the receiver script before the sender script.

15.5 Integrated Tests

A series of integrated tests⁷ were added using Boost Test Framework. These tests are automatically run as part of our CI/CD pipeline. The main tests currently included are HDTN Loopback tests in cutthrough and storage modes (using contact plans with or without link disruptions).

16 Containerization

HDTN currently supports the use of Docker and Kubernetes to deploy containers with HDTN built with all of its required dependencies.

16.1 Docker Instructions

First make sure docker is installed.

- `apt-get install docker`

Check the service is running.

- `systemctl start docker`

There are currently two Dockerfiles for building HDTN, one for building an Oracle Linux container and the other for building an Ubuntu. This command will build the Ubuntu one:

- `docker build -t hdtntest containers/docker/ubuntu/.`

The `-t` sets the name of the image, in this case `hdtntest`. Check the image was built with the command:

- `docker images`

Now to run the container use the command:

⁷https://github.com/nasa/HDTN/blob/master/tests/integrated_tests/src/integrated_tests.cpp

- `docker run -d -t hdtm_ubuntu`

Check that it is running with:

- `docker ps`

To access it, you'll need the `CONTAINER_ID` listed with the `ps` command

- `docker exec -it container_id bash`

Stop the container with

- `docker stop container_id`

The same container can either be restarted or removed. To see all the containers Use:

- `docker ps -a`

These can still be restarted with the run command above. To remove one that will no longer be used:

- `docker rm container_id`

16.2 Docker Compose Instructions

Docker compose can be used to spin-up and configure multiple nodes at the same time. This is done using the docker compose file found under `HDTN/containers/docker/docker_compose`.

- `cd containers/docker/docker_compose`

This file contains instructions to spin up two containers using Oracle Linux. One is called `hdtm_sender` and the other `hdtm_receiver`. Start them with the following command:

- `docker compose up`

On another bash terminal these can be accessed using the command:

- `docker exec -it hdtm_sender bash`
- `docker exec -it hdtm_receiver bash`

This setup is perfect for running a test between two hdtm nodes. An example script for each node can be found under `HDTN/tests/test_scripts.linux/LTP_2Nodes.Test/`. Be sure to run the receiver script first, otherwise the sender will have nowhere to send to at launch.

16.3 Kubernetes Instructions

Download the dependencies

- `sudo apt-get install docker microk8s`

The first step is to create a docker image to be pushed locally for kubernetes to pull:

- `docker build docker/ubuntu/. -t myhdtm:local`

Check that it was built:

- `docker images`

Next we build the image locally and inject it into the microk8s image cache

- `docker save myhdtm > myhdtm.tar`
- `microk8s ctr image import myhdtm.tar`

Confirm this with:

- `microk8s ctr images ls`

Now we deploy the cluster, the yaml must reference the injected image name

- `microk8s kubectl apply -f containers/kubernetes/hdtn_10_node_cluster.yaml`

There should now be ten kubernetes pods running with HDTN. See them with:

- `microk8s kubectl get pods`

To access a container in a pod, enter the following command:

- `microk8s kubectl exec -it container_name -- bash`

When you're finished working with this deployment, delete it using:

- `microk8s kubectl delete deployment hdtn-deployment`

Use the `get pods` command to confirm they've been deleted

- `microk8s kubectl get pods`

17 Troubleshooting

By default HDTN is built in Release mode. To enable DEBUG mode during build use: `cmake .. -DCMAKE_BUILD_TYPE=Debug`

17.1 Logging

Logging is controlled by CMake cache variables. Build (or rebuild) HDTN after making the following changes under `HDTN/build/CMakeCache.txt`. By default logging to a file is turned off to reduce resource draw.

- `LOG_LEVEL_TYPE` controls which messages are logged. The options, from most verbose to least verbose, are `TRACE`, `DEBUG`, `INFO`, `WARNING`, `ERROR`, `FATAL`, and `NONE`. All log statements using a level more verbose than the provided level will be compiled out of the application. The default value is `INFO`.
- `LOG_TO_CONSOLE` controls whether log messages are sent to the console. The default value is `ON`.
- `LOG_TO_ERROR_FILE` controls whether all error messages are written to a single `error.log` file. The default value is `OFF`.
- `LOG_TO_PROCESS_FILE` controls whether each process writes to their own log file. The default value is `OFF`.
- `LOG_TO_SUBPROCESS_FILE` controls whether each subprocess writes to their own log file. The default value is `OFF`.

17.2 LTP Tuning Recommendations

There are several fields in the LTP configuration that will impact performance.

- Client service id corresponds to the LTP Client Service Identifiers as described in RFC7116. In general, select 1 for compatibility with most DTN implementations. See <https://www.iana.org/assignments/ltp-parameters/ltp-parameters.xhtml>. The following values are common for the client service id.
 - 0 - Reserved
 - 1 - Bundle Protocol
 - 2 - LTP Service Data Aggregation

- 3 - CCSDS File Delivery Service
- The following parameters must match on the sender and receiver:
 - OneWayLightTimeMs
 - OneWayMarginTimeMs
 - LtpMaxRetriesPerSerialNumber
- To properly tune LTP for a system with appreciable delay, make sure $2 \times (\text{oneWayLightTimeMs} + \text{oneWayMarginTimeMs})$ is slightly larger than the expected round trip time.
- Set `ltpRandomNumberSizeBits` to 32 for compatibility with DTNME. For HDTN to HDTN testing use 64.
- Set `ltpMaxRetriesPerSerialNumber` to a larger number (around 100) on a system that has significant disruptions. For a system that does not experience significant loss, 5 or less should be acceptable.
- If LTP is being used on a lower-rate communication system that does not provide flow control (for example a radio that supports several Mbps) it is important to set the LTP rate (either in the contact plan or via `--cla-rate`) to slightly lower than the expected link rate. Failure to do so may cause dropped packets since LTP is based on UDP. Alternatively, the max rate can be set per contact in the contact plan by setting `rateBitsPerSec` to a nonzero value. If both fields are set, `rateBitsPerSec` in the contact plan will take precedence.
- For a particular outduct, the max data it can hold in its sending pipeline shall not exceed, whatever comes first, either
 - More bundles than `maxNumberOfBundlesInPipeline`
 - More total bytes of bundles than `maxSumOfBundleBytesInPipeline`
- An error is thrown on startup if $(\text{maxBundleSizeBytes} * 2)$ is greater than `maxSumOfBundleBytesInPipeline`.
- Worst case RAM memory usage is given by summation of all outduct `maxSumOfBundleBytesInPipeline`. The sum should not exceed the system memory.
- If using Ethernet small frames, it is recommended to set the LTP MTU to 1360 to prevent IP fragmentation.
- Please see `LtpEngineConfig.h` docstrings for specific details related to LTP configuration.
 - <https://github.com/nasa/HDTN/blob/master/common/ltp/include/LtpEngineConfig.h>

18 Notes

18.1 TLS Support for TCPCL Version 4

TLS Versions 1.2 and 1.3 are supported for the TCPCL Version 4 convergence layer. The X.509 certificates must be version 3 in order to validate IPN URIs using the X.509 “Subject Alternative Name” field. HDTN must be compiled with `ENABLE_OPENSSL_SUPPORT` turned on in CMake. To generate (using a single command) a certificate (which is installed on both an outduct and an induct) and a private key (which is installed on an induct only), such that the induct has a Node Id of 10, use the following command:

```
openssl req -x509 -newkey rsa:4096 -nodes -keyout privatekey.pem -out cert.pem -
sha256 -days 365 -extensions v3_req -extensions v3_ca -subj "/C=US/ST=Ohio/
L=Cleveland/O=NASA/OU=HDTN/CN=localhost" -addext "subjectAltName =
otherName:1.3.6.1.5.5.7.8.11;IA5:ipn:10.0" -config /path/to/openssl.cnf
```

Note: RFC 9174 changed from the earlier -26 draft in that the Subject Alternative Name changed from a URI to an otherName with ID 1.3.6.1.5.5.7.8.11 (id-on-bundleEID).

- Therefore, do NOT use: -addext “subjectAltName = URI:ipn:10.0”
- Instead, use: -addext “subjectAltName = otherName:1.3.6.1.5.5.7.8.11;IA5:ipn:10.0”

To generate the Diffie-Hellman parameters PEM file (which is installed on an induct only), use the following command:

```
openssl dhparam -outform PEM -out dh4096.pem 4096
```

18.2 BP Version 6 and Version 7

Both versions of BP, BP version 6 (BPv6) and BP version 7 (BPv7), are similar in their core layout. Bundles are made up of various *blocks* of information that are necessary for nodes in a DTN network to execute store-and-forward and routing behavior. There are only two required blocks in a bundle: a primary block (the beginning of a bundle), and a payload block (the end of a bundle). Besides these required blocks, there are optional extension blocks between the primary and payload block. These extension blocks can include information such as hop limits, information about the previous/sender node, and class of service. While BPv6 and BPv7 bundles have the same bundle block structure, the details of included fields and field encoding within these blocks varies greatly. This section will discuss major differences relevant to processing both Bundle Protocol versions.⁸

18.2.1 Bundle Protocol Version 6

This section highlights unique characteristics of BPv6 and details relevant to parsing BPv6 bundles in a resource-constrained parser. Figure 10 provides a general diagram of the layout of BPv6 bundles.

⁸This section is an excerpt from (<https://www.mdpi.com/2673-8732/3/1/2>).

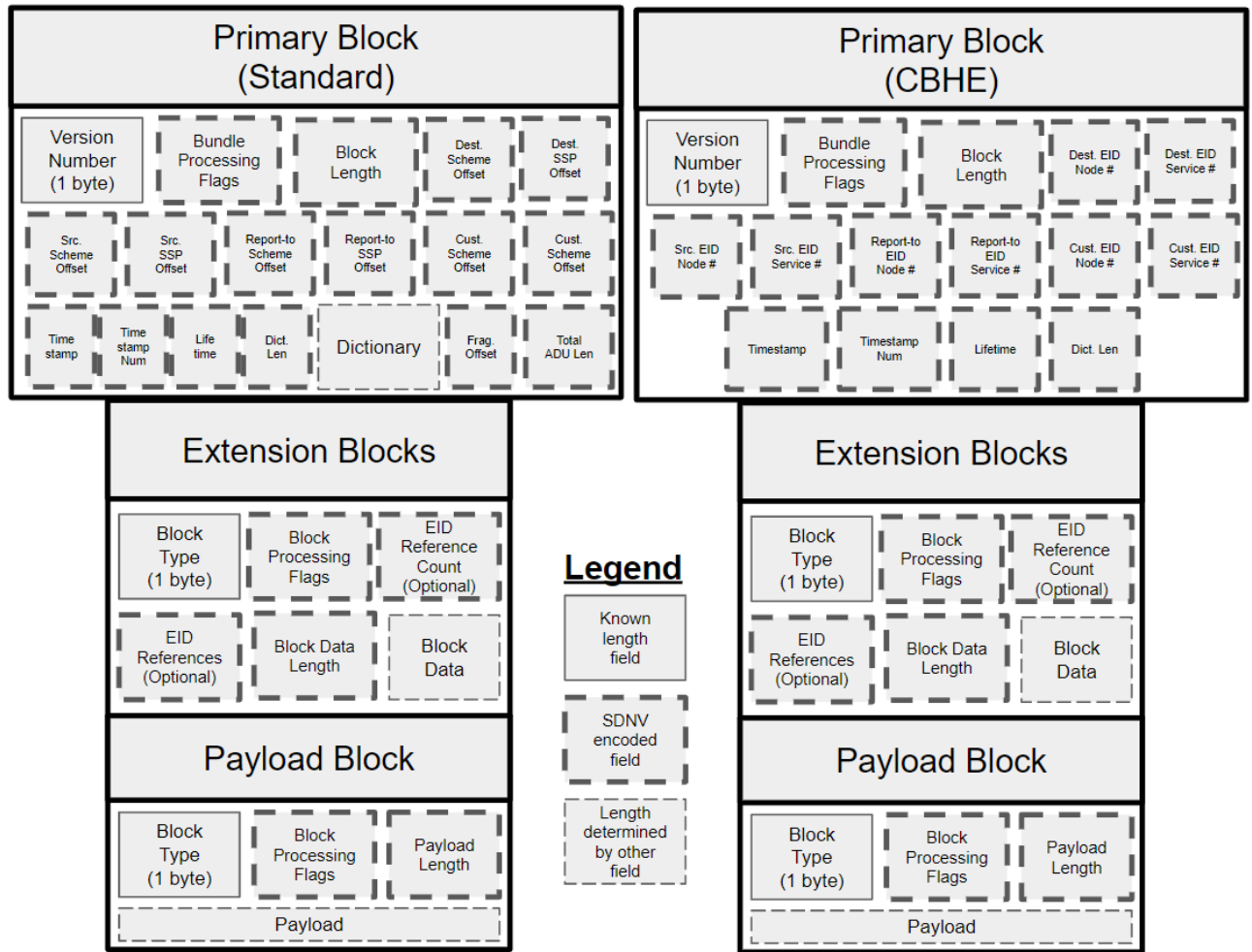


Figure 10.—Layout of BPv6 bundles.

BPv6 uses the Self-Delimiting Numeric Value (SDNV) encoding scheme. Compared to network protocols that have fields with pre-defined bit lengths, SDNV preserves bandwidth (by avoiding a minimum length), and allows for future extensibility and scalability (by avoiding a maximum length). The SDNV encoding scheme encodes any data into several octets of bits, where the 7 least significant bits (LSB) encodes the original data, and the most significant bit (MSB) of an octet determines whether or not it is the last octet of data. A MSB with a value of 0 indicates that it is the last octet of data, with all other octets having a MSB of 1. Figure 11 shows the decimal value 86,400 encoded using SDNV.

```

Original Decimal:
86400
Original Hexadecimal:
15180
Original Binary:
00010101000110000000
Original Binary (Groups of 7, starting from LSB):
00010101000110000000
SDNV-Encoded Binary:
10000101101000110000 0000
SDNV-Encoded Hexadecimal:
85a300

```

Figure 11.—Example SDNV encoding. The Most Significant Bit (MSB) or leftmost bit in the first two octets are a 1 indicating that this is not the last octet within this piece of data.

BPv6’s SDNV encoding adds significant complexity to a parser compared to other networking protocols (e.g. IP, UDP, and TCP) that have fields with known fixed-length with additional unknown options. For example, IP includes optional fields, but the length of these optional fields can be determined by examining the Internet Header Length (IHL) field which has a pre-determined position and length. With BPv6 and SDNV encoding, it is impossible to parse subsequent blocks or fields without examining all fields in order. Additionally, when parsing an individual SDNV-encoded field, it is impossible to know the length of this field until all bits of the field are examined. This prevents a hardware-constrained parser from having known compile-time behavior about a protocol, which introduces potential bugs and sub-optimal runtime performance.

Another complexity in parsing BPv6 are endpoint IDs (EID), which are names for destinations of bundles. EIDs are represented as Uniform Resource Identifiers (URI), of which there are currently two standardized URI schemes for DTN EIDs: the `dtm` scheme and the `ipn` scheme. The `dtm` scheme is more permissive and allows for arbitrarily complex character strings, looking similar to web URLs. However, the `ipn` scheme uses pairs of unsigned integers. Both of these schemes are able to represent an identifier for a bundle node and a demultiplexing token.

As EIDs can potentially be arbitrarily long, BPv6 utilizes *EID dictionaries*, an array of value pairs representing EIDs (pair of scheme name and scheme-specific value) in the primary block, that specifies all relevant EIDs for a given bundle. This can minimize bandwidth by allowing fields that need to reference an EID to simply reference *offsets* or indices in this dictionary, rather than encoding the actual endpoint ID.

Introduced four years after the BPv6 specification, Compressed Bundle Header Encoding (CBHE) is a mechanism to further preserve bandwidth, by avoiding encoding EID dictionaries. In a situation that meets the requirements for CBHE, encoding a dictionary can be skipped, with important EID information encoded as `ipn` scheme EIDs in the primary block’s source EID, destination EID, report-to EID, and custody EID offset fields. With CBHE, bundles can be transmitted without a dictionary, and the dictionary can then be rebuilt at the receiving node. This introduces complexity to a resource-constrained parser, as a parser must now be able to handle two different versions of a primary block. The CBHE version not only removes the EID dictionary, but changes the *meaning* of the EID fields, requiring significantly more logic to be programmed into a parser to properly handle both versions of the primary block.

18.2.2 Bundle Protocol Version 7

This section highlights unique characteristics of BPv7 and details relevant to parsing BPv7 bundles in a resource-constrained parser. A general diagram of the layout of BPv7 bundles is shown in Figure 12.

BPv7 changes the encoding of bundles from SDNV-formatted to “Concise Binary Object Representation” (CBOR). CBOR provides a structured serialization format, while maintaining flexibility and compactness. The layout of the CBOR data model is a superset of JavaScript Object Notation (JSON), containing several data types (e.g. integers, strings, maps, and arrays). An instance of a CBOR data type is a *data item*. In BPv7, a single bundle is a CBOR indefinite-length array, comprised of an indefinite number of blocks which are encoded as CBOR definite-length arrays. The end of a bundle is terminated by a stop code (0xff). An example BPv7 bundle and a decoding of its primary block is shown in Figure 13. This CBOR encoding introduces complexity to a parser as it must now maintain more metadata about the bundle, and additional logic is required to parse each unique possible CBOR data type.

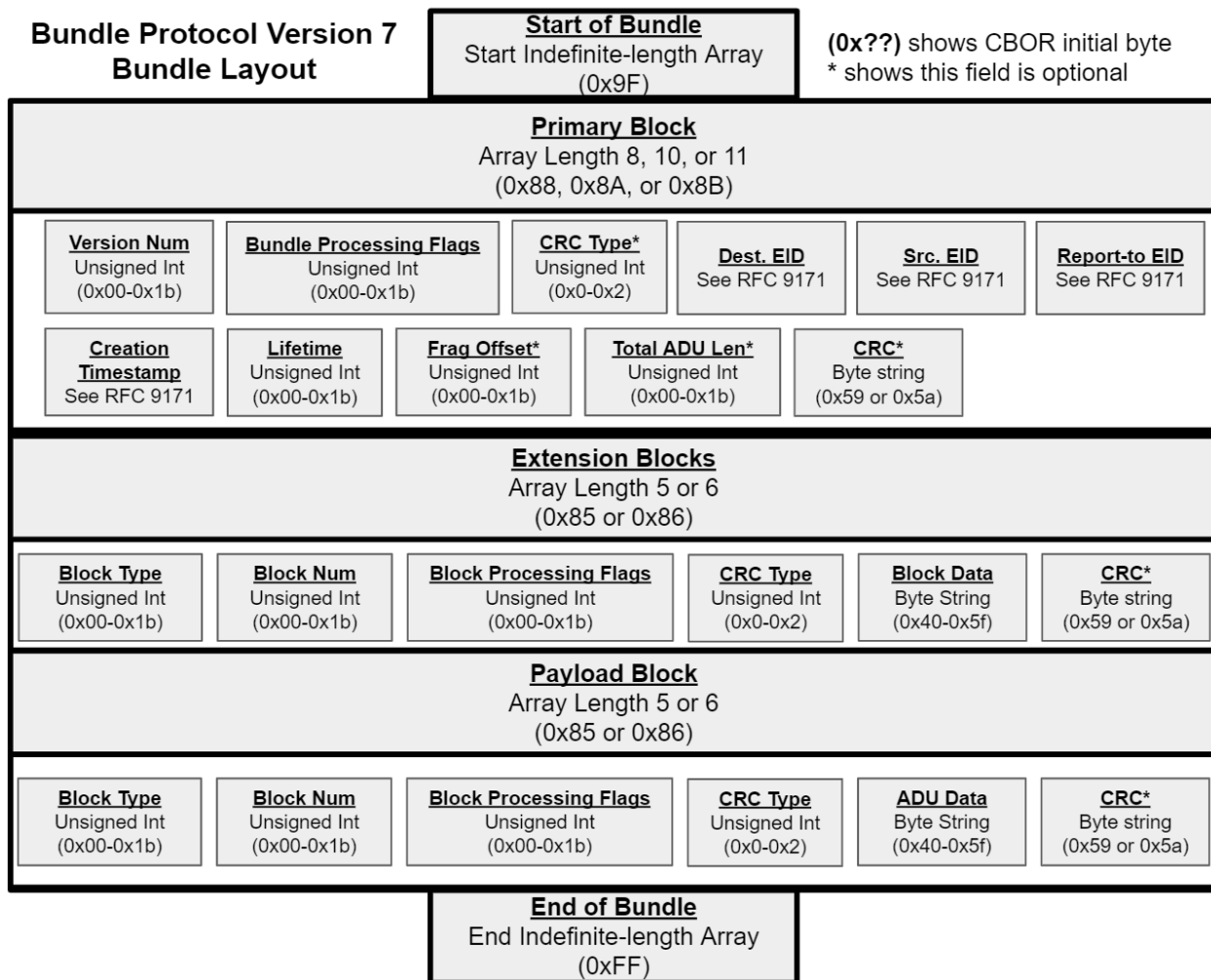


Figure 12.—Layout of BPv7 bundles.

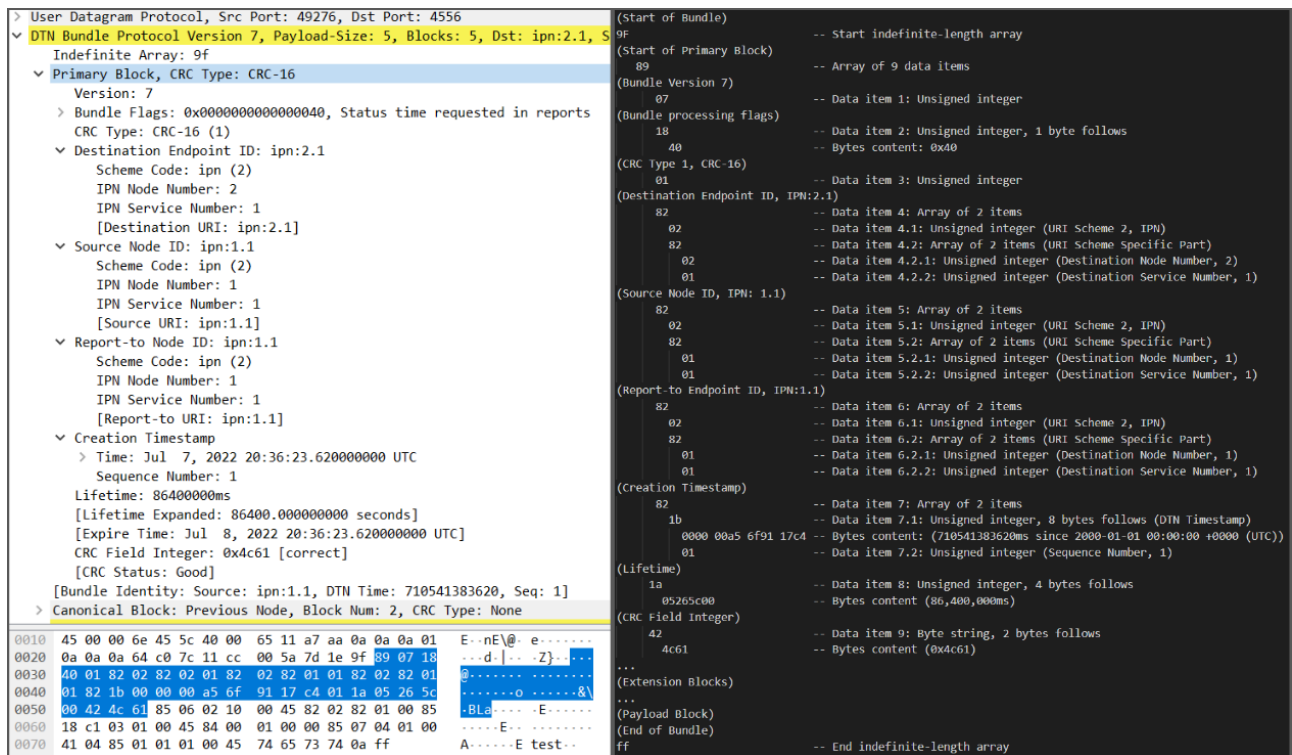


Figure 13.—Decoding of BPv7 bundle (left: Wireshark capture of BPv7 bundle with primary block bytes highlighted; right: CBOR decoding).

Alongside an encoding change, a couple important components were removed from the BP specification in BPv7. A removed feature in BPv7 is class of service. Through bundle processing flags, a BPv6 bundle requests either bulk, normal, or expedited service throughout the network. Another removed feature is custody transfer, also implemented through bundle processing flags, which is used to request another node to take responsibility for a given bundle. Successful custody transfer allows DTN nodes to clear space that was used for a bundle, knowing that another node is now responsible for its end-to-end delivery. BPv7 removes both of these features from the BP specification, however, they are now handled in different locations. In the future, class of service may be handled as an extension block, and custody transfer in the bundle-in-bundle encapsulation (BIBE) specification. Removing these features from the BP specification adds complexity for a BP parser and translator. A BP parser must recognize that the bundle processing flag bits for these removed features are still present in BPv7 but not used, and that BPv7 no longer retains information about a custody EID. Lastly, BPv7 adds optional error detection to bundle blocks through the form of Cyclic Redundancy Check (CRC) error-detecting codes (CRC-16 and CRC-32). This enables DTN nodes to ensure the data integrity of received bundle.

