**DZ Notes on the DTNME 1.0.1-BETA**

**06/10/2021**

Contents

[Build Instructions 3](#_Toc74212126)

[New Executable Names 4](#_Toc74212127)

[New Commands 5](#_Toc74212128)

[LTP Convergence Layer Configuration 7](#_Toc74212129)

[LTP Performance Testing 10](#_Toc74212130)

[LTP Discretionary Checkpoints 18](#_Toc74212131)

[Experimental Bundle Payload Delivery Option for Applications 19](#_Toc74212132)

[Append A 21](#_Toc74212133)

# Build Instructions

The latest NASA version of the DTNME 1.0.1-BETA source code can be downloaded from the GitHub site:

<https://github.com/nasa/DTNME/tree/v1.0.1_Beta>

(As of 06/09/2021, the latest version is: **1.0.1-2021-06-01-BETA**)

I forked the NASA DTNME project so that I could experiment with it at home, and I have a fix for the experimental Release DB File delivery method for apps that has not made it into the official release yet. The experimental delivery method is only implemented as an option in the sink\_me app used for throughput testing so not a significant issue in the scheme of things. If you would like to pick up that fix at this time you can pull from my fork:

<https://github.com/dz-dtn/DTNME/tree/v1.0.1_Beta>

(Its version is: **1.0.1-2021-06-09-BETAz**)

From the README file:

Dependencies:

libdb-dev g++ automake autotools-dev tk tk-dev tcl tcl-dev

Building:

**./make\_dtnme.sh <#cores>**

<#cores> can be replaced with the number of cores you would like to use in conjunction with the make -j option to try to speed up the build process.

Installing:

**sudo make install**

To change the install location, edit the make\_dtnme.sh script and update the configure command for the DTNME build at or near line 58. There is also a configure command for the OASYS library at or near line 23 which does not need to be modified to change the delivery location.

# New Executable Names

DTNME 1.0.1 (and 1.0) uses different executable names from those used in DTN2 and DTNME0.1. The executables included in the DTNME 1.0.1 release are:

|  |  |  |  |
| --- | --- | --- | --- |
| **ME 1.0.1**  **Name** | **DTN2/ME0.1**  **Name** | **Similar ION**  **App** | **Description** |
| dtnme | dtnd | (distributed) | The DTN server application |
| send\_me | dtnsend | bpdriver / bpsendfile | An app used to send bundles in throughput testing (probably not compatible with bpsendfile/bprecvfile) |
| sink\_me | dtnsink | bpcounter | An app used to receive bundles in throughput testing (similar to bpcounter for those familiar with ION) |
| ping\_me | dtnping | bping | An app that sends a bundle that functions similar to the network ping |
| echo\_me | dtnecho | bpecho | An app that receives and replies (echoes) a bundle back to the sender |
| trace\_me | dtntrace | bptrace | An app similar to ping\_me but also requests status reports to attempt to trace the path of the bundle |
| deliver\_me |  | cfdptest | An app the implements the CFDP file transfer protocol |
| dtpc\_send\_me | dtpc\_send | dtpcsend | An app that sends data using the DTPC protocol |
| dtpc\_recv\_me | dtpc\_recv | dtpcrecv | An app that receives data using the DTPC protocol |
| recv\_me | dtnrecv | bprecvfile /  bpcounter | An app that can receive bundles and output the payloads to files or generate less accurate throughput statistics than sink\_me  (probably not compatible with bpsendfile/bprecvfile) |
| report\_me | dtnreporter |  | An app that displays received status reports |
| source\_me | dtnsource | bpsource | An app that can generate test bundles |
| tunnel\_me | dtntunnel |  | An app that can be used to “tunnel” UDP or TCP or raw IP packets through DTN nodes. For example, it can establish an ssh connection from a system through DTN nodes to a remote system. |
| sdnv\_convert\_me | sdnv2num |  | An app that converts between values and their SDNV encoded equivalent |
| test\_ehs\_router | test\_ehs\_router |  | A testbed for the external router used by MSFC to support payloads using DTN on the International Space Station |

# New Commands

Below is a table of the new commands available in DTNME 1.0.1 that were not in the DTNME 0.1 release. DTNME 1.0.1 (and 1.0) support both Bundle Protocol version 6 (BPv6) and Bundle Protocol version 7 (BPv7) and a few of the new commands are related to BPv7.

A Contact Plan implementation has been included in the release. Issue the command “help contact” on the DTNME console for additional information on the contact commands listed in the table.

For a full list of available commands, issue the “help” command in the DTNME console and then for details for a particular command issue “help <command>”.

New in DTNME 1.0.1 is the feature that parameters of type <U64>, <Rate> and <Size> can include a magnitude character (K, M or G): 125G = 125000000000) which is easier to interpret than trying to count a string of zeros.

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Data**  **Type** | **Default** | **Description** |
| bp set status\_rpts\_enabled | Bool | false | Whether to generate Bundle Status Reports when requested |
| bp set use\_age\_block | Bool | false | Whether to use the BPv7 Age Block |
| bp set default\_hop\_init | U8 | 0 | Max allowed hops for the BPv7 bundles |
| bp malloc\_trim | N/A | N/A | (this is a development command to test releasing heap space) |
|  |  |  |  |
| bundle set use\_age\_block | Bool | false | Whether to use the BPv7 Age Block (deprecated - will probably be deleted in the future) |
|  |  |  |  |
| contact load plan |  |  | Load a contact plan formatted as a CSV file |
| contact export plan |  |  | Export a contact plan for the specified Endpoint ID to a CSV file |
| contact view plan |  |  | View the contact plan for a specified Endpoint ID |
| contact reset plan |  |  | Delate all contact plans |
| contact add contact |  |  | Add a contact manually |
| contact delete contact |  |  | Delete contacts for an Endpoint ID |
| contact view contact |  |  | View contacts for an Endpoint ID |
|  |  |  |  |
| interface options [cla\_name] |  |  | Lists the available Convergence Layer Adaptors (CL or CLA) or list the options available for a specified CLA name |
|  |  |  |  |
| link options [cla\_name] |  |  | Lists the available Convergence Layer Adaptors (CL or CLA) or list the options available for a specified CLA name |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| param set api\_send\_bp7 | Bool | | true | Whether an application should default to sending BPv7 or BPv6 bundles.  (Most apps provide the option to specify which version to use) |
| param set api\_deliver\_max\_memory\_size | | Size | 1M | Specify the max allowed bundle payload size to deliver to an application via memory (the TCP connection). Larger payloads will be delivered via file. (range is 0 to 100M) |
| param set clear\_bundles\_when\_opp\_link\_unavailable | | Bool | true | Whether to clear bundles from an opportunistic link when it goes unavailable |
| param set ipn\_echo\_service\_number | | U64 | 0 | Specify an IPN scheme service number to use for an echo service.  (only valid in the startup config file) |
| param set ipn\_echo\_max\_return\_length | | U64 | 1024 | Max payload size for an echo response bundle |
| param set persistent\_links | | Bool | false | Whether links should be stored in the database for retention across restarts |
| param set recreate\_links\_on\_restart | | Bool | true | Whether to recreate non-opportunistic links added during previous runs of the DTN daemon on startup |
| param set file\_permissions | | Octal16 | 0775 | Specify the desired file and directory permissions  (only valid in the startup config file) |
|  | |  |  |  |
| registration set suppress\_duplicates | | Bool | true | Whether to suppress delivery of duplicate bundles to registrations |
|  | |  |  |  |
| route set auto\_deliver\_bundles | | Bool | true | Whether local bundles should be automatically delivered when received or the [external] router should handle it |
|  | |  |  |  |
| storage set db\_force\_sync\_to\_disk | | Bool | True | Whether Bundle payload files should be synced to disk or allow the OS to handle it |
|  | |  |  |  |
| version | |  |  | Display the version of the DTNME daemon |
|  | |  |  |  |

# LTP Convergence Layer Configuration

The DTNME 0.1 LTP implementation in in the “ltpudp” convergence layer only supported a single remote node and the “link add” command was used to configure both the receiver and the sender. In DTNME 1.0.1 (and 1.0) the LTP implementation has been updated to support multiple remote nodes and now the receiver(s) is configured using an “interface add” command while the “link add” command is used to configure each sender.

From the DTNME console, the “interface options ltpudp” command below shows the options available for configuring an LTP receiver. Note that a single receiver can support multiple senders. Additional receivers can be used if there is a need for different UDP port numbers to support the remote nodes.

node2 dtn% interface options ltpudp

LTP over UDP Convergence Layer [ltpudp] - valid Interface options:

CLA specific options:

UDP related params:

local\_addr <IP address> - IP address of interface on which to listen

(default: 0.0.0.0 = all interfaces)

local\_port <U16> - Port on which to listen (default: 1113)

recvbuf <U32> - socket receive buffer size

(default: 0 = operating system managed)

Example:

interface add in\_ltp ltpudp local\_addr=192.168.0.1 local\_port=1113

(create an interface named "in\_ltp" to listen for LTP over UDP packets on network interface with IP address 192.168.0.1 using port 1113)

NOTE: The local LTP Engine ID is the node number extracted from the configured "route local\_eid\_ipn" parameter if not specified using the "route set local\_lt\_engine\_id" command in the startup configuration file.

The table below contains a list of the LTP Link specific parameters with the new ones available in DTNME 1.0.1 highlighted in gold. These are specified on the “link add” command for the “ltpudp” convergence layer. Additional parameters and a sample usage can be viewed using the “link options ltpudp” command from the DTNME console.

New in DTNME 1.0.1 is the feature that parameters of type <U64>, <Rate> and <Size> can include a magnitude character (K, M or G): 125G = 125000000000) which is easier to interpret than trying to count a string of zeros.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter Name** | **Data Type** | **Default** | **Description** |
| remote\_engine\_id | U64 | 0 | LTP Engine ID of remote peer  (NOTE: DTNME 0.1 used engine\_id to specify the local LTP engine ID instead of the remote peer’s ID) |
| max\_sessions | U32 | 100 | Max number of outgoing sessions |
| agg\_size | U64 | 1M | Accumulate/aggregate bundles for an LTP session until this number of data bytes is reached  (not including header/trailer bytes) |
| agg\_time | U32 | 500 (ms) | Accumulate/agregate bundles for an LTP session for up to this number of milliseconds |
| ccsds\_compatible | Bool | False | Whether to use CCSDS 734.1-B-1 compatibility which specifies client service ID 2 for Data Aggregation  (default: false which is compatible with ION through at least 3.7.3 / 4.0.2) |
| inact\_intvl | U32 | 30 (secs) | Cancels an incoming LTP session if there has been no activity for this number of seconds |
| retran\_intvl | U32 | 7 (secs) | Retransmit interval in seconds for LTP segments if ACK or Report not received |
| retran\_retries | U32 | 7 | Number of retransmissions to make before cancelling an LTP session |
| queued\_bytes\_quota | Size | 2G | Max incoming bytes of LTP segments to queue for processing before starting to discard [only] Data Segments; 0 = no limit |
| bytes\_per\_checkpoint | Size | 0 | When to generate discretionary/intermediate checkpoints for outgoing sessions; 0 = never  (not compatible with ION through at least 3.7.3 / 4.0.2) |
| use\_files\_xmit | Bool | false | Whether to use files or memory to hold the data for outgoing LTP sessions |
| use\_files\_recv | Bool | false | Whether to use files or memory to hold the data for incoming LTP sessions |
| use\_files | Bool | False | Shorthand for setting both use\_files\_xmit and use\_files\_recv to the same value |
| dir\_path | String | <local dir> | Path to directory to use for storing LTP session data (defaults to using the current working directory if not specified) |
| keep\_aborted\_files | Bool | false | Whether to keep LTP session data files that had errors for analysis |
| use\_diskio\_kludge | Bool | false | Whether to use some disk I/O kludges that were necessary for a while on the development system; Hopefully, never needed again - details in LTPCommon.h. |
| hexdump | Bool | false | Whether to output all incoming and outgoing segments in hex to the log file |
| comm\_aos | Bool | true | Communications Acquisition of Signal flag (false pauses LTP processing) - "link reconfigure" can be used to change it |
| clear\_stats | Bool | false | Used with "link reconfigure" to clear LTP statistics between test runs |
| dump\_sessions | Bool | true | Whether to include a listing of the active LTP sessions when executing the “link dump” command |
| dump\_segs | Bool | false | Whether to include a listing of all of the segments of the active LTP sessions when executing the “link dump” command |
| xmit\_test | I32 | 0 | Transmit Test modes:  n <= 0 : normal operation  n >= 1 : drop every nth packet to transmit |
| recv\_test | I32 | 0 | Receive Test modes:  n <= 0 : normal operation  n >= 1 : drop every nth packet received |
|  |  |  |  |
|  |  |  |  |
| **UDP Parameters** |  |  |  |
| rate | Rate | 0 | UDP transmit rate throttle in bits per second (default: 0 = no throttle) |
| bucket\_type | 0 or 1 | 0 | Token bucket type: 0 = standard; 1 = leaky  (standard sort of averages out to the rate; leaky treats the rate as a hard ceiling to not exceed) |
| bucket\_depth | U64 | 524280 | Token bucket throttle depth in bits  (default = max sized UDP packet in bits) |
| sendbuf | U32 | 0 | UDP socket send buffer size to use  (default: 0 = operating system managed) |
|  |  |  |  |

# LTP Performance Testing

The updates in the DTNME 1.0.1 LTP implementation were done with the main goal of improving memory usage and hoping for a secondary performance benefit as well. The LTP links can now be configured to use memory or files to hold the LTP Session Data. First up is a table comparing the throughput rates for the various DTNME implementations using memory and DTNME 1.0.1 using the file option. The test environment and configurations used are provided in Appendix A. In general, there is minimal round trip time and packet loss between the two nodes so LTP sessions are quickly received, reported/acknowledged and released unless processing becomes backed up on the receiver.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Re-baselined DTNME 0.1 and DTNME 1.0 using the 1.0.1 version of sink\_me (dtnsink for 0.1) which has some tweaks to make it more accurate and to support payloads greater than 4 GB | | | | | | |
|
| **LTP Convergence Layer  Testing** | | | | | | |
| ME01 = DTNME 0.1.2020-03-18 (LTP in memory) | | | | | | |
| ME10 = DTNME 1.0.2021-03-26 (LTP in memory) | | | | | | |
| ME101 = DTNME 1.0.1-2021-06-01 (LTP in memory) | | | | | | |
| ME101-f = DTNME 1.0.1-2021-06-01 (LTP in files) | | | | | | |
|  | | | **Goodput Rate in Mbps** | | | |
| **Test** | **Num Bundles** | **Payload Size** | **ME01 -> ME01** | **ME10 -> ME10** | **ME101 -> ME101** | **ME101-f -> ME101-f** |
| **1** | **1M** | **100 KB** | 2348 | 2843 | 2878.834 | 2766.618 |
| **2** | **100K** | **1 MB** | 2996 | 5788 | 5977.422 | 5780.491 |
| **3** | **10K** | **10 MB** | 1818 | 6294 | 6292.381 | 4911.551 |
| **4** | **1K** | **100 MB** | 1385 | 6137 | 6044.747 | **2318.168** |
| **5** | **100** | **1 GB** | 1441 | 5748 | 5794.711 | 5301.961 |
| **6** | **25** | **4 GB** | FAIL | 4622 | 5710.274 | 3993.971 |
| **7** | **10** | **12 GB** |  | 499 | 2131.938 | 448.726 |
| **The blue 100M test with LTP in files experienced a double digit CPU wait percentage and processing was backed up such that 7 GB of LTP segments were queued for processing at one point** | | | | | | |
|
| The red 12GB tests required changing the max retransmits from 100 to 500 during the test as the receiver started cancelling LTP sessions; ME101-f peaked at 11.7 GB of LTP segments queued to be processed; | | | | | | |
|

The tables below provide statistics on the CPU and memory usage for the various tests.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 1: Sending 1M bundles of size 100K via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** | 340.654 | 2348 | 280 | < 0.1 | 1.5 | 356 | 0.6 | 1.7 |
| **1.0** | **Memory** | 281.363 | 2843 | 343 | 0.2 | 1.7 | 311 | 0.2 | 1.6 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 277.89 | 2879 | 370 | 0.6 | 2.2 | 291 | 0.3 | 1.6 |
| **1.0.1** | **Files** | 289.162 | 2767 | 404 | 0.6 | 2.2 | 365 | 0.3 | 1.6 |

A bit more CPU usage when using LTP data in files. I think the limiting factor on throughput here is that DTNME can only take in about 3000 – 4000 bundles per second from the send\_me application. LTP is transmitting them as fast as they are coming in. Prior testing with an aggregation size of 1 MB vs 100K did not appreciably increase throughput.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 2: Sending 100K bundles of size 1M via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** | 267.936 | 2996 | 287 | 0.2 | 1.5 | 314 | 0.6 | 1.7 |
| **1.0** | **Memory** | 138.221 | 5788 | 268 | 0.1 | 1.7 | 211 | 0.2 | 1.6 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 133.837 | 5977 | 341 | 0.6 | 2.1 | 221 | 0.2 | 1.6 |
| **1.0.1** | **Files** | 138.397 | 5780 | 353 | 0.6 | 2.2 | 293 | 0.2 | 1.6 |

DTNME 1.0.1 is using more CPU and resident memory than the earlier versions.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 3: Sending 10K bundles of size 10M via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** | 440.006 | 1818 | 289 | 0.3 | 1.7 | 254 | 0.5 | 1.8 |
| **1.0** | **Memory** | 127.104 | 6294 | 273 | 0.2 | 1.7 | 219 | 0.1 | 1.5 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 127.138 | 6292 | 333 | 0.8 | 2.3 | 252 | 0.3 | 1.6 |
| **1.0.1** | **Files** | 162.881 | 4912 | 507 | 0.6 | 2.2 | 304 | 0.3 | 1.6 |

DTNME 1.0.1 using LTP files is using quite a bit more CPU in this test.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 4: Sending 1K bundles of size 100M via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** | 577.459 | 1385 | 270 | 1.9 | 3.3 | 263 | 1.3 | 2.7 |
| **1.0** | **Memory** | 130.357 | 6137 | 283 | 1.7 | 3.2 | 249 | 0.3 | 1.6 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 132.346 | 6045 | 285 | 0.9 | 2.4 | 277 | 0.4 | 1.8 |
| **1.0.1** | **Files** | 345.1 | **2318** | 332 | 0.7 | 2.3 | 304 | 6.5 | 7.9 |
| **The blue 100M test with LTP in files experienced a double digit CPU wait percentage and processing was backed up such that 7 GB of LTP segments were queued for processing at one point** | | | | | | | | | |
|

The “link dump <link\_name>” provides statistics that can help determine where the bottleneck is occurring. This is from an earlier test in which the 1.0.1 Receiver using LTP data files had backed up even more than the one in the table:

Receiver threads: SegProcessor: queued: 0 bytes: 0 ( 0 ) **max bytes: 11057813174 ( 11G+)**

quota: 50000000000 ( 50G ) DS discards: 0

BundleProcessor: queued: 0 bytes: 0 ( 0 ) max bytes: 2200001386 ( 2G+)

To minimize delays reading the UDP socket, packets are read and immediately posted to another thread for the initial processing to determine the LTP segment type and which remote node it goes with. The LTP segments are then posted to the SegProcessor for the appropriate node for full processing.

The SegProcessor thread actually has two queues. One is for administrative type segments and the other is for Data Segments (DS). The stats are only for the DS queue since the admin queue should not grow very large. Queued administrative type segments are always processed first to speed up completing sessions and freeing resources for both nodes. The Data Segments are the bottleneck as writing them out to the LTP file(s) cannot keep up. This is observed when the system total CPU wait time reaches about 10% (4 CPUs worth of time in this environment). Both nodes are running on the same machine and using the same disk drive, which may be a factor.

The quota was set to 50G to see how high it might reach. If controlling memory usage is a necessity, then the queued\_bytes\_quota parameter can be used to limit that at the expense of dropping LTP Data Segments that will require retransmission later.

Once a complete LTP session has been received, the associated Data Segments are posted to the BundleProcessor where the bundles are extracted from the LTP data. Above it can be seen that 2.2 GBs of LTP data was queued for processing at its peak, which is 22 bundles/sessions in this 100M test. When using LTP files, the 2.2 GBs is actually in the file and not in memory, so processing requires reading from the file and writing to the database Bundle Payload file.

Here are the stats for the 1.0.1 Receiver using LTP in memory that shows no issue keeping up with the incoming LTP segments:

Receiver threads: SegProcessor: queued: 0 bytes: 0 ( 0 ) **max bytes: 3503080 ( 3M+)** quota: 50000000000 ( 50G ) DS discards: 0

BundleProcessor: queued: 0 bytes: 0 ( 0 ) max bytes: 100000125 (100M+)

Speaking of disk I/O, most of the areas in the source code dealing with reading and writing files have been updated to use 10 MB buffers where they were originally using buffers of about 8K or so. During testing, it was found that a 4K buffer was still in use when transferring a file from the send\_me application to the Bundle Payload file. Changing that one to use a 10 MB buffer resulted in the Sender overwhelming the Receiver and cutting throughput in half in some cases so in the spirit of the Volkswagen diesel engine engineers it was left at 4K for now. The disk I/O buffer sizing might be worth experimenting with in the future.

To avoid overrunning a receiver, it might be useful to implement a “bytes in flight quota” parameter in addition to the max sessions parameter to prevent new sessions while the size of the active sessions is >= to the quota to control how much data is thrown at a receiver. I may have to experiment with that sometime in the future.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 5: Sending 100 bundles of size 1G via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** | 555.345 | 1441 | 323 | 15.5 | 17 | 274 | 9.1 | 10.9 |
| **1.0** | **Memory** | 139.184 | 5748 | 252 | 3.8 | 5.4 | 236 | 2.4 | 3.8 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 138.057 | 5795 | 227 | 5.2 | 6.8 | 224 | 1.2 | 2.6 |
| **1.0.1** | **Files** | 150.888 | 5302 | 252 | 0.6 | 2.3 | 268 | 0.3 | 1.8 |

Here, some of the memory saving can finally be observed.

The 1.0.1 Receiver using LTP memory has to hold the entire set of Data Segments for a session in memory but it now incrementally extracts bundles from each segment. Earlier DTNME versions reconstituted the LTP session data in contiguous memory for processing which doubled the memory usage.

It is not obvious but the 1.0.1 Sender with LTP in memory is using about 1/3 of the memory compared to the 1.0 version per bundle but here the 1.0.1 Sender has five bundles processed for sending while the 1.0 Sender only has one.

The 0.1 Receiver is unable to keep up with the Sender and sessions are backing up on the Sender.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 6: Sending 25 bundles of size 4G via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** | FAIL | FAIL | 244 | 88.4 | 104.3 | 192 | 60.3 | 61.8 |
| **1.0** | **Memory** | 173.082 | 4622 | 251 | 15 | 16.6 | 251 | 11.3 | 12.8 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 140.098 | 5710 | 236 | 5 | 6.6 | 273 | 4.8 | 6.2 |
| **1.0.1** | **Files** | 200.302 | 3994 | 242 | 0.7 | 2.3 | 262 | 3.5 | 5 |

The “link dump” command output for the 1.0.1 Sender with LTP in memory shows that the transmit queue peaked at 3.9 GB of data backed up at one point in either this test or the next one (these stats were taken after the last test):

Transmit queues - admin: 0 resend DS: 0 send DS: 0 Total bytes: 0 ( 0 )

**Max: 3987558448 ( 3G+)**

The transmit thread utilizes three queues. The highest priority queue is for administrative type LTP segments. The second priority for transmission goes to LTP Data Segments that are being resent either due to checkpoints not receiving a Report Segment or in response to a Report Segment that indicated missed segments. The lowest priority queue is for LTP Data Segments that are being sent for the first time.

Once again, the thinking is to try to complete in-progress LTP sessions as quickly as possible before adding additional sessions to the mix and we do not want more important segments stuck in a queue behind a long line of first-time data segments.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test 7: Sending 10 bundles of size 12G via LTP** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **LTP Storage** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **0.1** | **Memory** |  |  |  |  |  |  |  |  |
| **1.0** | **Memory** | 1924.43 | 499 | 296 | 27.2 | 29 | 223 | 35 | 36.7 |
|  |  |  |  |  |  |  |  |  |  |
| **1.0.1** | **Memory** | 450.295 | 2132 | 274 | 11.9 | 13.5 | 263 | 14.3 | 15.7 |
| **1.0.1** | **Files** | 2139.39 | 449 | 324 | 0.8 | 2.5 | 216 | 11.3 | 12.7 |

Backed up processing on the Receivers is evident here again. The 1.0.1 test using LTP files and the 1.0 test using LTP in memory both required the retransmit retries limit to be raised from 100 to 500 to complete the test.

When the processing gets backed by several GBs of data, there will be many DS Checkpoints retransmitted that do not get processed until well after the LTP session has been completed and removed from the list of active sessions. In the case of using LTP files, logic had to be added to prevent a late DS from opening a “new” session and clobbering the LTP data file that has not yet had its bundle(s) extracted.

**NOTE:** When LTP sessions are removed from the active list, the session number and disposition of success and size or cancelled is now maintained for the inactivity interval (inact\_intvl) number of seconds or a minimum of 1 hour so that appropriate reply segments can be sent when processing catches up to these “late” segments. **The downside to this might be that if the Sender is recycled and starts using those session numbers again within that period there might be issues.**

# LTP Discretionary Checkpoints

New in DTNME 1.0.1 is the ability to configure to use LTP discretionary or intermediate checkpoints. It was explored as a possible memory saving mechanism but may only serve that purpose in this test environment with a minimal round trip time and packet loss. When using LTP in memory, Data Segments are queued for transmission as soon as they are generated. Including a discretionary checkpoint every 10 MBs makes it possible for the Receiver to signal (Report Segments) to the Sender that it can release 10 MB chunks of segments while it is still in the process of generating segments for a large session.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test: Sending 1 bundle of size 12G via LTP using memory** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **10MB Checkpoints** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **1.0.1** | **No** | 39.425 | 2435 | 138 | 11.3 | 12.9 | 92.9 | 11.4 | 12.7 |
| **1.0.1** | **Yes** | 40.852 | 2350 | 110 | 5.4 | 7.1 | 101 | 11.4 | 12.7 |

Definite memory savings for the Sender in this particular scenario.

It may be worth exploring whether or not there is an advantage to using discretionary checkpoints versus only making the last Data Segment a checkpoint. In the extreme case of a 12 GB LTP Session that drops every other packet, there are a lot of Report Segments that have to be generated in response to the one and only checkpoint at the end of the session.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test: Sending 1 bundle of size 12G via LTP using memory  & dropping every other packet in both directions (xmit\_test=2)** | | | | | | | | | |
|  | | | | **Sender** | | | **Receiver** | | |
| **DTNME Version** | **10MB Checkpoints** | **Seconds** | **Goodput (Mbps)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** | **Max CPU (%)** | **Max RES Mem (G)** | **Max VIRT Mem (G)** |
| **1.0.1** | **No** | 140.123 | 685 | 233 | 11.3 | 12.9 | 173 | 11.5 | 12.9 |
| **1.0.1** | **Yes** | 118.225 | 812 | 229 | 6.4 | 8 | 148 | 11.5 | 12.9 |

Is it better to be able to start filling in some of the gaps earlier or not in various scenarios? It appears that intermediate checkpoints may hinder performance a bit when there is a clean data flow but improve performance when the data flow drops packets.

# Experimental Bundle Payload Delivery Option for Applications

The standard method of delivering a bundle payload to an application is to copy the data from the DTN Server’s bundle payload file to a temporary file and provide the name of that new file to the app. Why not just release the DTN Server’s copy of the payload file to the app and skip all of the disk I/O required to copy the file? The sink\_me application includes an option (**-b**) to utilize the experimental Release DB File delivery method but the results are mixed and errors were encountered in one scenario so it is left as an experimental option at this point (a fix is available as noted above).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **LTP Convergence Layer  Testing** | | | | | | |
| **All tests use DTNME 1.0.1 and LTP in files** | | | | | | |
| Baseline = both nodes on drive A with temp file delivery results detailed earlier | | | | | | |
| Release/DriveB = Sender using drive A and Receiver using drive B, release DB file delivery | | | | | | |
| Release/DriveC = Sender using drive A and Receiver using drive C, release DB file delivery | | | | | | |
|  | | | **Goodput Rate in Mbps** | | | |
| **Test** | **Num Bundles** | **Payload Size** | **Baseline** | **Release DB** | **Release/DriveB** | **Release/DriveC** |
| **1** | **1M** | **100 KB** | 2767 | 2738 | 2749 | 2775 |
| **2** | **100K** | **1 MB** | 5780 | 5778 | 5833 | 5737 |
| **3** | **10K** | **10 MB** | 4912 | 5308 | 5244 | 5334 |
| **4** | **1K** | **100 MB** | **2318** | 5248 | 2322 | **ERRORS** |
| **5** | **100** | **1 GB** | 5302 | 5315 | 3376 | 5343 |
| **6** | **25** | **4 GB** | 3994 | 1389 | 719 | 5080 |
| **7** | **10** | **12 GB** | 449 | 396 | 739 | 1968 |

I was expecting that using two drives would improve performance over a single drive so when adding drive B to the mix did the opposite, I tested with drive C. It appears that not all drives are created equal even though they supposedly have the same specs.

Using drive C, the sink\_me application reported errors in the 100M test when a few of the file sizes did not match the payload size provided through the API. The DTN server log messages indicated bad file descriptor errors for some of the LTP data files and bundle payload files indicating that one thread probably closed a file while another was using it.

Note that payload sizes up to 1 MB are being delivered in memory instead of via file so the first two tests are generally not impacted by the delivery method.

If a sync to disk is in progress for a bundle’s payload file then the release routine waits for it to complete. If persistence is not a major concern then the sync to disk can be disabled with the command “storage set force\_sync\_to\_disk false” which might ease disk I/O a bit and improve throughput.

Bottom line is that this feature should not be used in production and there does not appear to be much performance improvement to be gained except when working with large bundles.

While touching on the subject of using multiple drives, do they help improve performance when LTP is in memory and only the database payload files are on disk? Here are the results using the same drives as above:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **LTP Convergence Layer  Testing** | | | | | | |
| **All tests use DTNME 1.0.1 and LTP in memory and temp file delivery method** | | | | | | |
| Baseline = both nodes on the same drive with temp file delivery | | | | | | |
|  | | | | | | |
| Receiver-DriveB = Sender using drive A and Receiver using drive B | | | | | | |
| Receiver-DriveC = Sender using drive A and Receiver using drive C | | | | | | |
|  | | | **Goodput Rate in Mbps** | | | |
| **Test** | **Num Bundles** | **Payload Size** | **Baseline** |  | **Receiver-DriveB** | **Receiver-DriveC** |
| **1** | **1M** | **100 KB** | 2879 |  | 2821 | 2751 |
| **2** | **100K** | **1 MB** | 5977 |  | 5869 | 5683 |
| **3** | **10K** | **10 MB** | 6292 |  | 5348 | 5145 |
| **4** | **1K** | **100 MB** | 6045 |  | 3028 | 5356 |
| **5** | **100** | **1 GB** | 5795 |  | 2490 | 5272 |
| **6** | **25** | **4 GB** | 5710 |  | 1335 | 4663 |
| **7** | **10** | **12 GB** | 2132 |  | 705 | 1246 |

Not exactly what I expected. Maybe drive A is just extremely fast??

# Append A

**Test Environment**

DELL R730: 40 CPUs (including hyper threading), 128 GB RAM, 10K RPM 1 TB drives;

Red Hat Enterprise Linux Server 7.5 running the Boeing Common Open Research Emulator (CORE)

Initial testing was done with both nodes utilizing a single hard drive. Additional tests were conducted with the two nodes using separate drives.

Bundle transmission was performed using the send\_me application varying the file size and count:

send\_me -s ipn:2.1 -d ipn:3.2 -Y -t f -p file\_100K -n 1000000

Bundle reception was performed using the sink\_me application varying the count:

sink\_me -e 0 -p -v 10000 -t 7777 -n 1000000 ipn:3.2

The configuration of the LTP receiver is simple:

interface add ltp\_rcvr ltpudp local\_addr=$local\_ip\_address local\_port=$local\_ltp\_port

Below is a typical configuration for the Sender (node2 transmitting to node3) with parameters that varied highlighted in yellow:

link add ${remote3\_link\_name} ${remote3\_ip\_address}:${remote3\_ltp\_port} ALWAYSON ltpudp inact\_intvl=30 retran\_intvl=3 retran\_retries=100 remote\_engine\_id=${remote3\_node\_number} rate=8G max\_sessions=100 agg\_size=100000 agg\_time=500 seg\_size=64000 keep\_aborted\_files=true bytes\_per\_checkpoint=0 queued\_bytes\_quota=50G use\_files=false xmit\_test=0

node2 dtn% link dump node3

Current link:

Link node3:

clayer: ltpudp

type: ALWAYSON

state: OPEN

bp6\_redirect:

bp7\_redirect:

deleted: false

nexthop: 10.0.1.12:1113

remote eid: dtn:none

mtu: 0

min\_retry\_interval: 5

max\_retry\_interval: 600

idle\_close\_time: 0

potential\_downtime: 30

prevhop\_hdr: true

reincarnated: false

used in fwdlog: false

queue limits disabled

contact: connected

contact duration: 0

contact elapsed: 610

contact planned state: active

remote\_engine\_id: 3

local\_addr: 255.255.255.255 local\_port: 0

remote\_addr: 10.0.1.12 remote\_port: 1113

rate: 8000000000 ( 8Gbps )

bucket\_type: Standard

bucket\_depth: 524280

inact\_intvl: 30 seconds

retran\_intvl: 3 seconds

retran\_retries: 100

agg\_size: 100000 (100K ) bytes (setting agg\_size=0 and/or agg\_time=0 would trigger sending the initial 3-byte

agg\_time: 500 milliseconds pilot bundle immediately and would improve accuracy in the rate calcs)

seg\_size: 64000 bytes

ccsds\_compatible: false

max\_sessions: 100

hexdump: false

use\_files\_xmit: false

use\_files\_recv: false

keep\_aborted\_files: true

use\_diskio\_kludge: false

dir\_path: [full paths: ./tmp\_ltp\_<engine\_id>/<files> ]

bytes\_per\_checkpoint: 0 ( 0 )

queued\_bytes\_quota: 50000000000 ( 50G )

xmit\_test: 0 (0=normal; >=1 = drop every nth packet to transmit))

recv\_test: 0 (0=normal; >=1 = drop every nth packet received))

dump\_sessions: true

dump\_segs: false

comm\_aos: true

Below is a typical configuration for the Receiver (node3 receiving from node2) with parameters that varied highlighted in yellow:

link add ${remote2\_link\_name} ${remote2\_ip\_address}:${remote2\_ltp\_port} ALWAYSON ltpudp inact\_intvl=30 retran\_intvl=3 retran\_retries=100 remote\_engine\_id=${remote2\_node\_number} rate=8000000000 max\_sessions=100 agg\_size=100000 agg\_time=500 seg\_size=64000 keep\_aborted\_files=true bytes\_per\_checkpoint=0 queued\_bytes\_quota=50G use\_files=false xmit\_test=0     dir\_path=../database

node3 dtn% link dump node2

Current link:

Link node2:

clayer: ltpudp

type: ALWAYSON

state: OPEN

bp6\_redirect:

bp7\_redirect:

deleted: false

nexthop: 10.0.1.11:1113

remote eid: dtn:none

mtu: 0

min\_retry\_interval: 5

max\_retry\_interval: 600

idle\_close\_time: 0

potential\_downtime: 30

prevhop\_hdr: true

reincarnated: false

used in fwdlog: false

queue limits disabled

contact: connected

contact duration: 0

contact elapsed: 452

contact planned state: active

remote\_engine\_id: 2

local\_addr: 255.255.255.255 local\_port: 0

remote\_addr: 10.0.1.11 remote\_port: 1113

rate: 8000000000 ( 8Gbps )

bucket\_type: Standard

bucket\_depth: 524280

inact\_intvl: 30 seconds

retran\_intvl: 3 seconds

retran\_retries: 100

agg\_size: 100000 (100K ) bytes

agg\_time: 500 milliseconds

seg\_size: 64000 bytes

ccsds\_compatible: false

max\_sessions: 100

hexdump: false

use\_files\_xmit: false

use\_files\_recv: false

keep\_aborted\_files: true

use\_diskio\_kludge: false

dir\_path: ../database [full paths: ../database/tmp\_ltp\_<engine\_id>/<files> ]

(a symbolic link was used to change the disk drive this node used)

bytes\_per\_checkpoint: 0 ( 0 )

queued\_bytes\_quota: 50000000000 ( 50G )

xmit\_test: 0 (0=normal; >=1 = drop every nth packet to transmit))

recv\_test: 0 (0=normal; >=1 = drop every nth packet received))

dump\_sessions: true

dump\_segs: false

comm\_aos: true