NASA/TM—2012-216047

A Performance Evaluation of NACK-Oriented Protocols as the Foundation of Reliable Delay-Tolerant Networking Convergence Layers

Dennis Iannicca, Alan Hylton, and Joseph Ishac
Glenn Research Center, Cleveland, Ohio

December 2012

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 Aeronautics and Space Database and its public interface, the NASA Technical Reports Server, 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 counterpart 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 cosponsored 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.

Specialized services also include creating custom thesauri, building customized databases, organizing and publishing research results.

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 443–757–5803

- Phone the NASA STI Information Desk at 443–757–5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/TM—2012-216047

A Performance Evaluation of NACK-Oriented Protocols as the Foundation of Reliable Delay-Tolerant Networking Convergence Layers

Dennis Iannicca, Alan Hylton, and Joseph Ishac
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

December 2012

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

Available from

NASA Center for Aerospace Information
7115 Standard Drive
Hanover, MD 21076–1320

National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Available electronically at http://www.sti.nasa.gov

A Performance Evaluation of NACK-Oriented Protocols as the Foundation of Reliable Delay-Tolerant Networking Convergence Layers

Dennis Iannicca, Alan Hylton, and Joseph Ishac
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

Delay-Tolerant Networking (DTN) is an active area of research in the space communications community. DTN uses a standard layered approach with the Bundle Protocol operating on top of transport layer protocols known as convergence layers that actually transmit the data between nodes. Several different common transport layer protocols have been implemented as convergence layers in DTN implementations including User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and Licklider Transmission Protocol (LTP). The purpose of this paper is to evaluate several standalone implementations of negative-acknowledgment based transport layer protocols to determine how they perform in a variety of different link conditions. The transport protocols chosen for this evaluation include Consultative Committee for Space Data Systems (CCSDS) File Delivery Protocol (CFDP), Licklider Transmission Protocol (LTP), NACK-Oriented Reliable Multicast (NORM), and Saratoga. The test parameters that the protocols were subjected to are characteristic of common communications links ranging from terrestrial to cis-lunar and apply different levels of delay, line rate, and error.

1.0 Introduction

Delay Tolerant Networking (DTN) was created by the research community as a means of experimenting with the concept of standardized store and forward techniques (Ref. 1). Convergence Layers are a set of network and transport protocols utilized by the Bundle Protocol (Ref. 2) to connect various DTN nodes (forwarding agents). To facilitate communication across the DTN, these convergence layers must be selected based on their expected performance across network conditions between DTN agents. One set of conditions often found in space networks is links that have high latency, high error, frequent disruption, and asymmetrical rates. Frequent disruption is handled via placement of the DTN agents across the network. The remaining conditions are best handled by choosing an appropriate transport, or convergence layer.

A variety of transmission protocols have been developed to handle the range of conditions found in space communication. We focus on the Consultative Committee for Space Data Systems (CCSDS) File Delivery Protocol (CFDP) (Ref. 3), Licklider Transmission Protocol (LTP) (Refs. 4 and 5), Negative-Acknowledgement (NACK) Oriented Reliable Multicast (NORM) (Ref. 6), and Saratoga (Ref. 7). Our goal is to test the performance of each of these protocols in a set of varying link conditions. In doing so, we hope that DTN agents may be configured to use the most appropriate transport mechanism available to them.

2.0 Protocols

The protocols discussed in this document are all designed to be reliable in the face of suboptimal conditions. A common trait of space links is asymmetry, where the link capacity differs greatly depending on the direction of communication. Space communication links are often highly asymmetric with Mb/s data transmission from a spacecraft and only a few Kb/s data transmission to a spacecraft. This restricts the ability to periodically transmit positive acknowledgments (ACK) of data reception, a typical feedback mechanism utilized in many terrestrial protocols.

To prevent flooding these links with ACKs, we identified protocols that utilize negative acknowledgments (NACK). NACK based protocols transmit focus on sending status messages when data is considered missing or corrupt. This greatly reduces the amount of messages transmitted when the link is moderately prone to errors and loss. While space links can be considered fairly noisy and prone to large error rates, link layer coding techniques are often used to correct those errors reducing the impact to the transport layer protocols. Burst errors can still be prevalent; however, sequential loss of data is well accommodated by NACK based algorithms.

As we will discuss, NORM is purely NACK based. Saratoga uses NACKs throughout a file transfer but sends an acknowledgment at the completion of a file transfer. CFDP uses both ACKs and NACKs. LTP offers both unreliable transmission and ACK-based reliable transmission.

The convergence layers studied in this paper can be run in conjunction with several typical transports found on all major operating systems. In our studies, all of our tests were conducted over the User Datagram Protocol (UDP), which provides the best environment to observe the behavior of each convergence layer under study. In addition, the convergence layers automatically handle fragmentation and reassembly of any over sized data payloads that exceed the transports MTU.

UDP is unreliable, and thus timeout mechanisms are left to the convergence layer. Not all links in a disconnected network have high latencies. Therefore, sometimes timers are either short or unnecessary. This level of control varies. In the chosen implementation of LTP, the timer values are determined automatically. The timeout values in the version of CFDP used may be compiled in. Timeout values in NORM are not configurable. Saratoga timeouts are specifically unspecified. The version of Saratoga used for testing enabled manual control of timeout values.

With NORM and with the version and implementation of Saratoga used in testing, reliability is always present. In CFDP, reliability is optional dependent on the class of service selected. In LTP, the user may select some or all of a given payload to be transmitted reliably. While both transmission error and link disruption may cause packet loss, only losses due to transmission error are studied in this paper.

2.1 CCSDS File Delivery Protocol (CFDP)

The CCSDS File Delivery Protocol (CFDP) is a recommended standard for transmission of files to and from spacecraft data storage. In addition to file transfer functionality, the protocol includes simple file management capabilities including the ability to create and remove directories. It is capable of operating in a wide variety of mission configurations ranging from relatively simple low earth orbit spacecraft to complex arrangements of orbiters and landers supported by multiple ground facilities and transmission links. It is independent of the data storage technology and makes no assumptions regarding the type of information being transferred so it can be used in a wide variety of scenarios involving loading, dumping, and control of spacecraft data storage. It is scalable and has been designed to minimize the required operational resources (Ref. 3). CFDP has the ability to cancel, suspend (freeze), and resume (thaw) an in-progress file transfer in order to deal with the intermittent connectivity associated with spacecraft. CFDP supports four classes of operations: Class 1—Unreliable Transfer; Class 2—Reliable Transfer, Class 3—Unreliable Transfer Via One Or More Waypoints In Series, and Class 4—Reliable Transfer Via One Or More Waypoints In Series (Ref. 3).

For the purposes of this evaluation, the Class 2 Reliable Transfer mode was used to allow a file to be reliably transferred between two nodes using NACKs. A class 2 CFDP file transmission can be broken down into four phases. In phase 1, the file metadata followed by the file data are transmitted down to the receiver one block at a time until the transfer is complete. In phase 2, or handoff phase, the sender transmits an end of file notification to the receiver, which in turn is expected to acknowledge it. In phase 3, the receiver attempts to fill any missing gaps in the transmission by sending a negative acknowledgement to the sender to request that it retransmits the missing blocks. In phase 4, the receiver transmits that it has finished successfully receiving the file and the sender should acknowledge the end of transmission.

2.2 Licklider Transmission Protocol (LTP)

The Licklider Transmission Protocol (LTP) is designed to provide reliable connectivity over single-hop deep-space RF links with extremely long round-trip times and frequent interruptions in connectivity by using an Automatic Repeat reQuest (ARQ) error-control method that solicits selective acknowledgments (SACK). (Refs. 4 and 5) LTP treats user data as blocks, which may be comprised of two parts: a “red” part that must be acknowledged and retransmitted if lost, and a “green” part that is a best-effort delivery. Either part of the block may be omitted such that a packet consists of only a single color. (Ref. 5) The “red” and “green” parts of the block are not intended to denote any priority and “red” blocks will thus not be delivered with a higher priority than “green” blocks. While LTP is designed to run directly over a data-link layer protocol it may be deployed over UDP in software development or testing purposes to form a “local data-link layer” (Ref. 5). During these evaluation tests, the LTP implementation was configured to transmit wholly “red” parts in the data blocks over UDP, thus providing reliable retransmission of blocks dropped due to errors.

2.3 Negative-Acknowledgement (NACK) Oriented Reliable Multicast (NORM)

The Negative-Acknowledgement (NACK) Oriented Reliable Multicast (NORM) protocol is designed to provide reliable transfer of data from one or more senders to a group of receivers of an IP multicast network. However, NORM can be used for unicast transmission (multicast group of one). The primary design goal is to provide efficient, scalable, and robust bulk data transfer across possibly heterogeneous IP networks (Ref. 6). It is able to adapt to a variety of network conditions autonomously with little or no pre-configuration. It is tolerant of mobile and wireless networks that provide unreliable connectivity including situations where there is heavy packet loss and large transmission delays. NORM is designed mainly under the assumption of a single sender transmitting data content out to a group of receivers. Unlike the other protocols evaluated in this paper, NORM is not inherently designed to target space communications. However, its ability to tolerate large propagation delays makes it an interesting comparison to the others.

2.4 Saratoga

Saratoga is a peer-to-peer file transfer protocol capable of efficiently transferring small files, large files, and streaming

continuous data. It was originally developed as a low-overhead alternative to CFDP and is used today to transfer data from Surrey Satellite Technology Ltd (SSTL) Disaster Monitoring Constellation (DMC) remote-sensing satellites in low-Earth orbit. The satellites feature a very fast downlink ranging from 8 Mbps to over 200 Mbps with uplink speeds of only 9.6 kbps. The Saratoga protocol was thus designed with these highly asymmetric links in mind in order to efficiently transfer the files to the ground (Ref. 7).

Saratoga uses a Selective Negative Acknowledgement (SNACK) mechanism to provide reliable retransmission of data (Ref. 7). Like CFDP, the newest specification of Saratoga supports a number of file management operations including "get" downloads, "put" uploads, directory listing, and deletion requests. The "put" transactions allow the immediate sending of packets without waiting for a status acknowledgement from the receiver. This unidirectional behavior is ideal for deep-space scenarios with large propagation delays where SYN/ACK style communications protocols are undesirable. The "get" transactions are better suited for scenarios with low propagation delays such as the transmission from LEO satellites to the Earth ground stations and were the only type of file transfer transaction supported by the original version 0 protocol implementation.

3.0 Test Environment and Setup

3.1 Test Systems

The experimental lab setup consists of three servers each running 64-bit Ubuntu GNU/Linux 10.04 and containing a 2 GHz AMD Opteron 246 CPU, 1 GB of RAM, 80 GB SATA disk drive, and multiple Broadcom Tigon3 Gb Ethernet interfaces. Two of the systems were utilized to send and receive test data, while the third system was used to control and monitor the link between the source and destination hosts. The test path was constructed by directly cabling the three servers, configured in a simple dumbbell topology as show in Figure 1.

The channel emulator system uses a software-based network emulator written for the NASA Compatibility Test Set (CTS) project that allows channels to be defined between interfaces with varying amounts of delay, bit error per packet, and rate constraints applied to it. The software provides an XML-RPC-based API to interact and control the channel parameters dynamically based upon the requirements of the testing phase (Ref. 8).

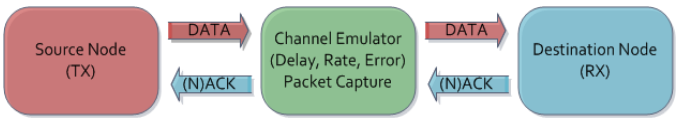


Figure 1.—Test environment network topology.

In addition to the test path, each machine was also connected to an internal lab network providing a means of managing and monitoring the nodes without negatively impacting the test results.

3.2 Network Topology

The IP addressing scheme of the environment employed standard RFC 1918 private addresses in the 192.168.0.0/16 range. The IP address of the source test interface was configured as 192.168.100.1 and the destination test interface as 192.168.100.2. The channel emulator acted as a bridge between the source and destination nodes and had no IP addresses accessible on its interfaces facing those systems. The second management interface for all three systems was connected to the internal laboratory network and used for Secure Shell (SSH) access and out-of-band configuration including instantiating the test runs. All test data was thus isolated between the 192.168.100.1 and 192.168.100.2 interfaces of the test hosts.

3.3 Protocol Implementations

The protocol implementations ultimately chosen for testing were standalone protocol implementations that did not require any additional dependencies or frameworks to operate. For these experiments, we chose the following implementations for testing:

- CFDP—NASA GSFC CFDP Engine 3.1a1 (Ref. 9)
- LTPlib—Mercurial snapshot (4-11-2011) (Ref. 10)
- NORM—Subversion snapshot (3-25-2011) (Ref. 11)
- Saratoga—v0 Perl snapshot (4-11-2011) (Ref. 12)

We first evaluated versions of CFDP obtained from JPL; however, efforts to compile and run them in our environment were met with only limited success and the stability and results were erratic. We were able to later obtain the GSFC CFDP Engine. Upon evaluation, we found that the GSFC code compiled cleanly and ran very well in our environment. Due to limited time available for debugging the JPL CFDP code, the GSFC CFDP Engine was selected for these tests.

For LTP, we found that there were two commonly available implementations available: a Java version from Ohio University (Ref. 13) and LTPlib from Trinity College in Dublin, Ireland (Ref. 10). The Ohio University Java version, lasted updated in late 2006, was deemed too dated to use. Unlike the Ohio University implementation, the LTPlib implementation was still actively maintained and provided LTP support to the DTN2 reference implementation. We initially faced a few variable type portability problems related to LTPlib being developed on a 32-bit architecture while we were using a 64-bit architecture, but these were patched.

Another problem we faced was that attempting to transfer files larger than a few megabytes would result in the ltpd daemon crashing. The author was able to make some simple fixes to the code that allowed us to transfer up to about 10 MB files successfully, but we continued to face problems transferring files much larger than that without the process crashing. The large file stability problem in the LTPlib code at the time of testing led us to constrain all the tests to use 10 MB files as the maximum size across all the protocols for consistency. If this limitation had not existed, it would have been preferable to use much larger files in the 100 MB to 1 GB range as the maximum size.

The version of NORM we obtained to use for testing was the latest Subversion snapshot available at the time and was recommended over the stable version by the NORM development team.

While there are various different versions and flavors of Saratoga implementations available, we decided to test the standalone version 0 implementation developed in Perl at GRC. While there are currently version 1 implementations being developed in both Perl and C++, it was determined they were not ready for a proper evaluation at the time of testing. Other versions of Saratoga available were custom-designed for a specific environment and could not be easily run in our environment without additional development effort to customize it. Ideally during these tests a "put" operation would have been used to transfer the data in order to more accurately model against the three other protocols. The other protocols operate by pushing data from the sender to the recipient immediately without waiting for acknowledgment, but the Saratoga's v0 protocol PERL implementation used did not support a "put" operation so we were constrained to use the "get" operation to initiate the file transfer instead.

For the test setup, a set of UNIX shell and Expect scripts was designed to automate the testing and ensure each run was handled consistently. The Expect scripts began by configuring the network emulator, via its XML-RPC interface, to the desired delay, line rate constraints, and bit error per packet rate. The Expect script would then spawn a tcpdump packet capture of all the test traffic by listening on the network emulator interface's connected to the receiver. It would then remotely connect to the source and destination nodes via SSH and execute the protocol executables for the desired protocol implementation run. Once the run was complete, it would tear down the spawned processes and loop back around for another run if necessary.

Due to an oversight during the setup of the test environment, only a single tcpdump packet capture point on the receiver side of the channel emulator was in place resulting in traffic only being captured from the receiver's point of view. In order to try to compensate for not having a packet capture running on the sender's side of the channel we added the round-trip delay time to the total transmission time and sender-to-receiver transmission times in order to estimate the goodput[1] and throughput[2] from the sender's point of view. The receiver-to-sender times and throughput statistics calculated during post-processing did not need to be adjusted.

4.0 Performance Tests

Each of the protocol implementations were put through 10 runs of the 144 different permutations of line rates, delays, errors, and file sizes in Table I during testing to provide a pool of data for performance analysis. For each protocol, a series of test data files was created and put into their working directories to be made available for the sender to transmit. For the purposes of this evaluation, all the implementations were tested over symmetric links configured at the line rates specified in that permutation of the test. In all cases, attempts were made to configure the protocols that had a configurable MTU size, if possible, to send a packet of approximately 1 KB as a common size to match the implementations where the size was hard-coded at compile time like CFDP.

TABLE I.—TEST PARAMETERS

Line rates, Kb/s	1000, 10000
Delay, ms	25, 250, 1250
Bit error rate	0, 10×10^{-6}, 10×10^{-5}, 10×10^{-4}, 10×10^{-3}, 10×10^{-2}
File size, KB [a]	10, 100, 1000, 10000

The Saratoga v0 Perl implementation was not able to consistently transfer files successfully at 10×10^{-6} b error rate due to implementation bugs and was excluded at this level from the results.

[a] For the purposes of the creation of the test files and in all references to file sizes, 10 KB = 10×10^{3}, 100 KB = 100×10^{3}, 1 MB = 1×10^{6}, 10 MB = 10×10^{6}.

4.1 CFDP

The GSFC CFDP Engine was configured and compiled using its default "flight" profile as distributed. The default outgoing file chunk size and max chunk size were kept at 997 B. The maximum concurrent transactions parameter was also set at the default of 1000 with 20 max gaps per transaction. Other default parameters of interest include the ACK limit of 250, ACK timeout of 50, an inactivity timeout of 86400, NAK limit of 250 and the NAK timeout of 50. According to the tcpdump captures, the average packet size during the test runs was 1011 B, including protocol headers, keeping it near our 1024 B target for packet size.

The command line execution of the sender side was as followed including sample output from a run:

```
Sender> ./cfdp_flight 101
CFDP Sample Application Version 2007_06_08 (static allocation
only)
   MAX_CONCURRENT_TRANSACTIONS=1000.
```

[1] Goodput is defined as the throughput of the payload file without regard to the protocol overhead. It is calculated by taking the total number of bits of the file size and dividing it by the total time of transmission.

[2] Throughput is defined as the total number of bits transmitted divided by the total time of the transmission.

```
   MAX_GAPS_PER_TRANSACTION=20.
   MAX_FILE_CHUNK_SIZE=997
   MAX_PDU_LENGTH=1029.
   MAX_DATA_LENGTH=1029.
   Uses 672 bytes per open transaction.
cfdp_engine: entity-id set to '101'.

*** MIB parameter settings ***
   Entity-ID = 101
   Issue EOF-Sent Indication? = 1
   Issue EOF-Received Indication? = 1
   Issue File-Segment-Sent Indication? = 0
   Issue File-Segment-Received Indication? = 0
   (default) Ack-limit = 250
   (default) Ack-timeout = 50
   (default) Nak-limit = 250
   (default) Nak-timeout = 50
   (default) Inactivity-timeout = 86400
   (default) Outgoing file chunk size = 997 (bytes)
   (default) Save-incomplete-files = 0
```

A single command was input into the sender's session to initiate the transfer:

```
Sender> put testfile.$size 102
<Link_Type> We now have a two-way link.
cfdp_engine: version = 3.1a1
```

The receiver side command line execution was as follows including sample output from the corresponding test run:

```
Receiver> ./cfdp_flight 102
CFDP Sample Application Version 2007_06_08 (static allocation
only)
   MAX_CONCURRENT_TRANSACTIONS=1000.
   MAX_GAPS_PER_TRANSACTION=20.
   MAX_FILE_CHUNK_SIZE=997
   MAX_PDU_LENGTH=1029.
   MAX_DATA_LENGTH=1029.
   Uses 672 bytes per open transaction.
cfdp_engine: entity-id set to '102'.

*** MIB parameter settings ***
   Entity-ID = 102
   Issue EOF-Sent Indication? = 1
   Issue EOF-Received Indication? = 1
   Issue File-Segment-Sent Indication? = 0
   Issue File-Segment-Received Indication? = 0
   (default) Ack-limit = 250
   (default) Ack-timeout = 50
   (default) Nak-limit = 250
   (default) Nak-timeout = 50
   (default) Inactivity-timeout = 86400
   (default) Outgoing file chunk size = 997 (bytes)
   (default) Save-incomplete-files = 0

<Link_Type> We now have a two-way link.
cfdp_engine: version = 3.1a1.
----------> (101_1) MD:  'testfile.1m'.
:::Machine_Allocated  trans 101_1.
>>> Transaction 101_1 started (receiving; file name not yet
known)...
:::MD_Recv  trans 101_1, class 2, receiving 'testfile.1m' (1000000
bytes).
----------> (101_1) EOF: xsum=00000000, file-size=1000000.
:::EOF_Recv  trans 101_1.
<---------- (101_1) Ack-EOF:  .
<---------- (101_1) Fin:  .
----------> (101_1) Ack-Fin:  .
:::Trans_Finished  trans 101_1 successful.
:::Machine_Deallocated  trans 101_1.
>>> Transaction 101_1 (testfile.1m -> testfile.1m):
```

4.2 LTP

The LTPlib build was configured using the default parameters using the GNU auto-configure script (./configure) and compiled. No source code or header files were modified during the build process and all parameters were configured on the command line during run time. LTP's MTU size was configured to be restricted to 1024 B. The resulting packets, according to the tcpdump capture, were on average 1011 B including all protocol headers, keeping it near our 1024 B target for packet size.

The receiver side was started first in server mode and the command line execution was as follows, including sample output from the corresponding run:

```
Receiver> ./ltpd -L 192.168.100.2:1113 -D 192.168.100.2:1113 -S
192.168.100.1:1113 -m server -g -v -2 1024 -b 12000000

ltpd running in verbose mode
LTPD: It seems silly to set a keyid () for the off or NULL
ciphersuite (254)!
LTPD: Local parameters are:
LTPD:   crypto-cfg=ltpd.crypt
LTPD:   disksync=false
LTPD:   daemon=false
LTPD:   input=ltpd.in
LTPD:   log=ltpd.log
LTPD:   mode=1
LTPD:   output=ltpd.out
LTPD:   verbose=true
LTPD:   l2mtu=1024
LTPD:   LTP-T cfg=ltpt.cfg
LTPD:   dowait=true,sleeptime=10
LTPD:   rxbufsize=12000000
LTPD:   blocking=false
LTPD: Protocol parameters are:
LTPD:   dest=192.168.100.2:1113
LTPD:   source=192.168.100.1:1113
LTPD:   listen=192.168.100.2:1113
LTPD:   redlen=-1
LTPD:   cookies=0
LTPD:   cookie_grace=5
LTPD:   Cipher=254,KeyID=
LTPD: Starting run at: 2011-09-21 15:14:46.553
LTPD: ltpd running in process: 17433
LTPD: calling recvfrom 2011-09-21 15:14:46.654
LTPD: Got 1000000 bytes back (0-th time) from 192.168.100.1:1113
at 2011-09-21 15:15:05.536
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: getsockopt(LTP_SO_SOP) returned error -1
LTPD: too many getsockopt errors - 10
LTPD: You have compiled ltpd as a DODGY FILE SERVER take a look at
ltpd.cc:416
LTPD: Request is
LTPD: Wrote 1000000 bytes to ltpd.out-192.168.100.1.1113
LTPD: calling recvfrom
```

The sender side was started in client mode and the command line execution was as follows including sample output from the corresponding test run:

```
Sender> ./ltpd -D 192.168.100.2:1113 -S 192.168.100.1:1113 -m
client -v -2 1024 -b 12000000 -i testfile.$size

ltpd: no process found
ltpd running in verbose mode
LTPD: It seems silly to set a keyid () for the off or NULL
ciphersuite (254)!
LTPD: Local parameters are:
LTPD:   crypto-cfg=ltpd.crypt
LTPD:   disksync=false
LTPD:   daemon=false
LTPD:   input=testfile.1m
LTPD:   log=ltpd.log
LTPD:   mode=0
LTPD:   output=ltpd.out
LTPD:   verbose=true
LTPD:   l2mtu=1024
LTPD:   LTP-T cfg=ltpt.cfg
LTPD:   dowait=true,sleeptime=10
LTPD:   rxbufsize=12000000
LTPD:   blocking=false
LTPD: Protocol parameters are:
LTPD:   dest=192.168.100.2:1113
LTPD:   source=192.168.100.1:1113
LTPD:   listen=127.0.0.1:1113
LTPD:   redlen=-1
LTPD:   cookies=0
LTPD:   cookie_grace=5
LTPD:   Cipher=254,KeyID=
```

```
LTPD: Starting run at: 2011-09-21 15:14:48.632
LTPD: ltpd running in process: 7208
LTPD: Read 1000000 bytes from testfile.1m OK
LTPD: sending 1000000 bytes from file testfile.1m at 2011-09-21
15:14:48.734 to 192.168.100.2:1113
LTPD: finished sending
```

4.3 NORM

In addition to the NORM source code itself, the ProtoLib library had to be obtained from the same site as NORM depends on it in order to compile. After NORM is unpacked into a directory, the ProtoLib library must be unpacked into the NORM subdirectory as a dependency for the configuration and compilation to be successful.

Both ProtoLib and NORM are configured using their WAF configuration infrastructure by calling "./waf configure". It is an autoconfiguration framework similar to GNU autoconf used by other programs like LTPlib. The default values were chosen wherever necessary. ProtoLib should be configured and compiled before the process is repeated within the NORM directory.

Configuration of NORM is available entirely on the command line. According to the tcpdump packet captures of the test runs, the average NORM packet size was 1056 B, including protocol headers, keeping it near our 1024 B target for packet size.

The Expect scripts first setup the receiver side. It is shown below with the executable command line and corresponding run output:

```
Receiver> ./norm address 224.1.2.3/12347 interface eth2 rxcachedir
. debug 10 id 8472

Proto Info: ProtoDebug>SetDebugLevel: debug level changed from 1
to 10
Proto Debug: NormApp::Notify() unhandled event: 10
Proto Debug: NormSession::ReceiverHandleCommand() node>8472 new
remote sender:8471 ...
Proto Debug: NormApp::Notify(REMOTE_SENDER_ACTIVE) ...
Proto Trace: NormSenderNode::HandleCommand() node>8472 begin CC
back-off: 0.785881 sec)...
Proto Debug: NormApp::Notify(RX_OBJECT_NEW) ...
Proto Info: 08:18:04.211756 start rx object>59332
sender>15956196919349354775
Proto Detail: NormSenderNode::HandleObjectMessage() node>8472
sender>8471 new obj>59332
Proto Debug: NormApp::Notify(RX_OBJECT_INFO) ...
Proto Debug: NormSenderNode::HandleObjectMessage() node>8472
allocating sender>8471 buffers ...
Proto Trace: NormApp::Notify(RX_OBJECT_UPDATED) ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>0 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>1 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>2 completed block ...
Proto Debug: NormSenderNode::UpdateGrttEstimate() node>8472
sender>8471 new grtt: 0.246600 sec
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>3 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>4 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>5 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>6 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>7 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>8 completed block ...
Proto Trace: NormSenderNode::HandleCommand() node>8472 begin CC
back-off: 0.672307 sec)...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>9 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>10 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>11 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>12 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>13 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>14 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>15 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>16 completed block ...
Proto Debug: NormSenderNode::UpdateGrttEstimate() node>8472
sender>8471 new grtt: 0.228400 sec
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Trace: NormSenderNode::HandleCommand() node>8472 begin CC
back-off: 0.792389 sec)...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>17 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>18 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>19 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>20 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>21 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>22 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>23 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>24 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>25 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>26 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>27 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>28 completed block ...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>29 completed block ...
Proto Debug: NormSenderNode::UpdateGrttEstimate() node>8472
sender>8471 new grtt: 0.195800 sec
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Trace: NormSenderNode::HandleCommand() node>8472 begin CC
back-off: 0.686003 sec)...
Proto Detail: NormObject::HandleObjectMessage() node>8472
sender>8471 obj>59332 blk>30 completed block ...
Proto Debug: NormApp::Notify(RX_OBJECT_COMPLETED)
```

The Expect script then executed the corresponding sender-side call with the following command line parameters and run output. NOTE: $ratek is defined as the number of bits per second for the run and the intention was to have it match the network emulator line rate setting. So for this run for instance, $ratek = 1000000. This does differ slightly from the way the emulator was configured though as its configuration uses Kbps and was configured as 1000 Kbps, which works out to 1024000 bps.

```
Sender> ./norm address 224.1.2.3/12347 interface eth2 sendfile
testfile.$size id 8471 debug 10 rate $ratek

Proto Info: ProtoDebug>SetDebugLevel: debug level changed from 1
to 10
Proto Info: 08:18:04.248854 enqueued tx object>59332 sender>8471
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::OnProbeTimeout() node>8471 decreased to
new grtt to: 0.246600 sec
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::OnProbeTimeout() node>8471 decreased to
new grtt to: 0.228400 sec
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::OnProbeTimeout() node>8471 decreased to
new grtt to: 0.195800 sec
Proto Debug: NormApp::Notify(TX_OBJECT_SENT) ...
Proto Debug: NormApp::Notify(TX_QUEUE_EMPTY) ...
Proto Fatal: norm: End of tx file list reached.
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:1)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:2)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:3)...
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::SenderUpdateGrttEstimate() node>8471
```

```
increased to new grtt>0.228400 sec
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:4)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:5)...
Proto Info: REPORT time>08:18:14.246113 node>8471
************************************
Proto Info: Local status:
Proto Info:    txRate> 1000.000 kbps sentRate>  825.234
grtt>0.228400
Proto Info:
**********************************************************************
*********
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:6)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:7)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:8)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:9)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:10)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:11)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:12)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:13)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:14)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:15)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:16)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:17)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:18)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:19)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush
queued (flush_count:20)...
Proto Trace: NormSession::Serve() node>8471 sender flush complete
...
Proto Debug: NormApp::Notify(TX_FLUSH_COMPLETED)
```

4.4 Saratoga

The Saratoga implementation was Perl-based and being an interpreted language, there are no source code configuration and compilation steps necessary. Saratoga v0 is configured via a single configuration file named sara.conf. Different versions of these were created with different line rate values contained within and the Expect script was responsible for adjusting symbolic links to these so that the Saratoga binary would read the right configuration file depending on the line rate associated with the test parameters on the network emulator. A sample sara.conf looks like:

```
$C_hostaddr = "192.168.100.1"; #hostname of this machine Saratoga
is running on
$C_peeraddr = "192.168.100.2";#address of the peer machine
$C_verbose  = 0; # set to 1 to turn debugging on
$C_rate = 1000;  # servers transmission data rate in Kbps. 0 =
line rate.
```

Saratoga was then started on the sender side (verbosity and debugging was turned off for performance reasons so there is no further output during the test run):

```
Sender> ./saratoga_v0.pl
info: listening at 192.168.100.1:4000
```

With Saratoga, the receiver side initiates the file transfer via a "get" command to the sender. The following is the receiver executing, followed by the test run output:

```
Receiver> ./saratoga_v0.pl
info: listening at 192.168.100.2:4000
saratoga> get testfile.1m
saratoga> Transferring file: testfile.1m, length is 1000.0 Kbytes.
saratoga> writing file testfile.1m
1000000 byte file was received in 8 seconds. 929.2 kbits per
second.
```

According to tcpdump packet captures at the time of the test runs, the average Saratoga packet during transmission was 1034 B including protocol overhead, keeping it near our 1024 B target for packet size.

4.5 Maximum Performance

In addition to the test runs of the protocols that were constrained by the network emulator, a series of "maximum performance" test runs were run to provide a performance baseline for each of the convergence layer protocols. While the network topology stayed the same physically, the network emulator software was disabled and replaced with a simple network bridge interface between the 2 Gb-Ethernet interfaces on the channel emulator system resulting in the link being completely unrestricted of any delay, packet error, or rate constraints. Traffic was again captured at the same location using tcpdump as each of the protocol implementations ran through their maximum performance runs with an unconstrained gigabit-Ethernet link, no delay, and no artificial bit error applied with results shown in Figure 2. To verify the unconstrained link, iperf was run on the sender (client mode) and receiver side (server mode) to show the link capable of approximately 940 Mb/s of TCP traffic.

```
Sender> iperf -c 192.168.100.2
------------------------------------------------------------
Client connecting to 192.168.100.2, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.100.1 port 56424 connected with 192.168.100.2
port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.10 GBytes    941 Mbits/sec


Receiver> iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.100.2 port 5001 connected with 192.168.100.1
port 56424
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  1.10 GBytes    939 Mbits/sec
```

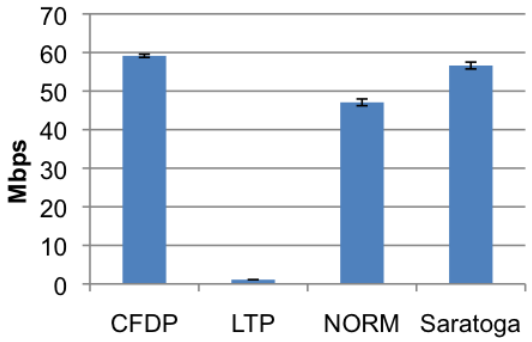


Figure 2.—Baseline protocol goodput on an unrestricted link.

Command line and configuration file parameters were the same as the regular test runs with the exception of the two implementations that had to be passed the line rate as a configuration parameter, NORM and Saratoga. During initial performance testing it was found that NORM tended to perform unreliably if its rate option was set to more than 100000000 bps (about 100 Mbps) and would actually result in longer transfer times than if set to a lower rate. For the sake of reliability, NORM was told to constrain itself to approximately 100 Mbps as it seemed to top out around 60 Mbps in our environment. It was not clear whether this was due to hardware processing constraints or timing problems with the software itself related to its rate constraints.

Likewise, Saratoga had problems when its line rate was set to a value above 85000 Kbps (approximately 85 Mbps) and so for the sake of reliable performance testing it was configured to constrain itself to a maximum of 85 Mbps. It is unclear whether the performance-related problems are related to the fact that the code is running in the interpreted Perl language or some other reason related to protocol timing or hardware processing constraints. While the Saratoga implementation is supposed to have a line rate option of "0", meaning unlimited, it was found to not work reliably and resulted in bugs popping up in the code related to uninitialized variables and division by zero errors. Since this was not a code development effort, the bugs were noted so they could be passed along to its developers while we attempted to work around them as best as possible. As with NORM, Saratoga tended to top out less than 60 Mbps in our environment so the artificial line rate constraint in its configuration did not appear to be an extra hindrance to its performance during this testing phase.

5.0 Test Results

5.1 Baseline

The purpose of these protocols is to make the most out of unreliable links. Thus, the analysis will focus on how well the protocols cope with various error rates. Due to time constraints and the number of tests run there are ten runs for each permutation. This does not lend the tests to parametric statistical analysis and therefore box plots are employed for visualization purposes. Several types of measurements were taken during each test. The focus in this paper will be on the goodput, the number of packets sent from the sender to the receiver, and the number of packets sent from the receiver to the sender. As mentioned in the previous section, a baseline performance test was performed to determine the maximum goodput of the protocol implementations over an unrestricted link. A 100 MB file was sent without the channel emulator in place and the rate limiting options in Saratoga and NORM set to provide the maximum performance.

As we see from the figure above, CFDP, NORM, and Saratoga all performed within the same order of magnitude when unrestricted by channel emulation. Despite many attempts to tweak the parameters of LTPlib, it was consistently sending at a lower rate than the other protocols. In Figure 3, the dot on LTP represents an outlier that is not visible on the scale that includes all protocols simultaneously. As can be seen in Figure 4, the number of packets from the sender to the receiver (STR) for the baseline is displayed.

We would expect a 100 MB file transfer would require approximately 100,000 packets to be sent without factoring in protocol overhead, but as can be seen in Figure 5 the number of packets sent varied a bit due to the differences in payload size not exactly matching up with the desired 1024 B MTU size. For example, the average maximum packet sizes recorded in the captures despite tweaking the configurations of the implementations were:

- CFDP—1011 B
- LTP—1010 B
- NORM—1056 B
- Saratoga—1034 B

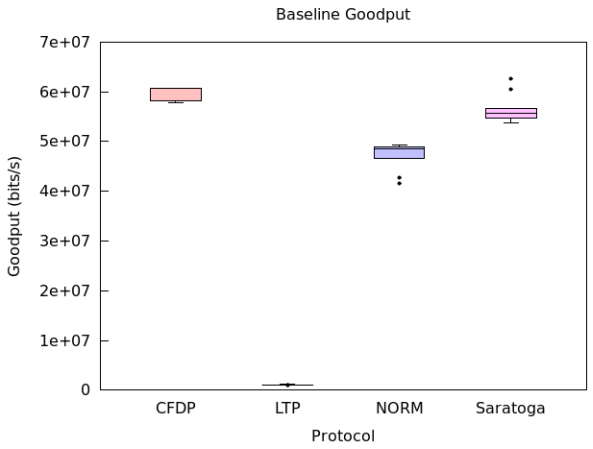


Figure 3.—Baseline goodput of protocols on unrestricted link.

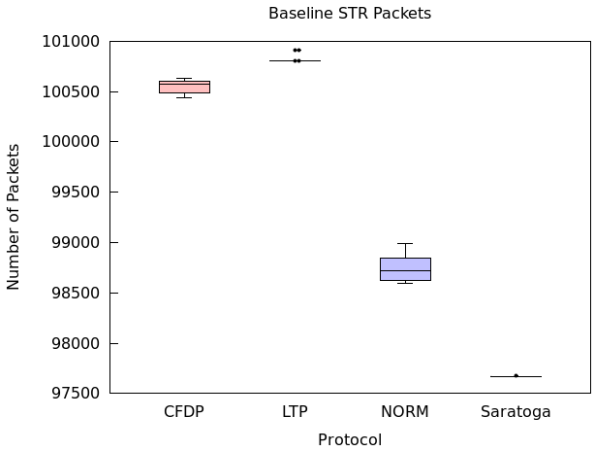


Figure 4.—Number of baseline sender-to-receiver packets captured on the unrestricted link.

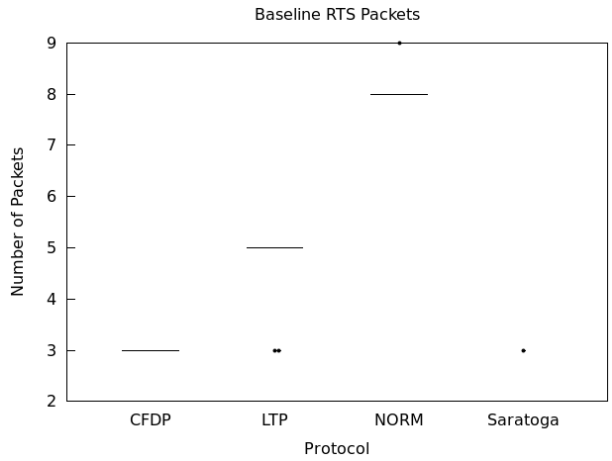


Figure 5.—Number of baseline receiver-to-sender packets captured on the unrestricted link.

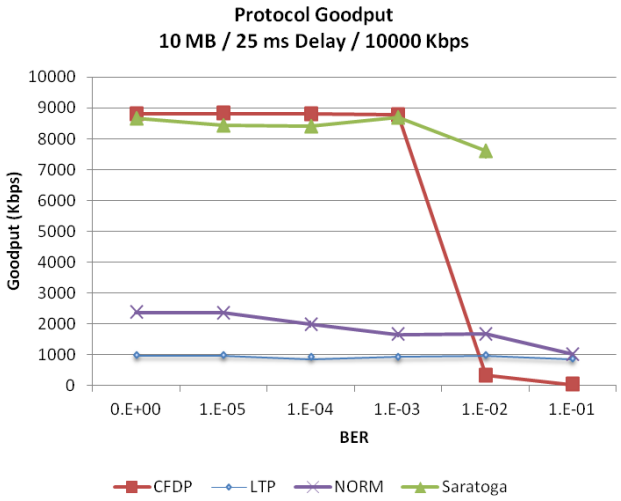


Figure 6.—Average protocol goodput rates of the protocols referenced in the analysis in Sections 4.2 to 4.5.

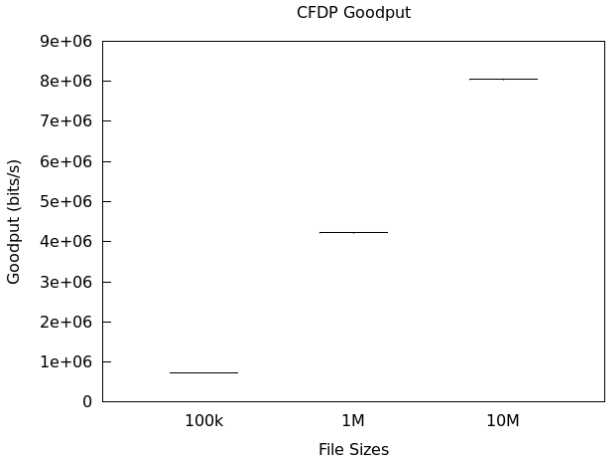


Figure 7.—The effects on the goodput of transferring different amounts of data using CFDP over the 10 Mbps constrained link.

As links to space assets are often asymmetrical, it is beneficial to see how many packets are sent from the receiver to the sender (RTS) when our link condition is ideal. The baseline RTS packet count follows.

With no errors, the RTS packet count is between 2 and 9 on our ideal link with NORM and LTP requiring many more feedback packets to the sender than CFDP or Saratoga. The average goodput rates at 25ms can be seen in Figure 6.

5.2 CFDP

Intuitively, as the payload size increases we would expect the goodput to also increase because less of the transmission is consumed by protocol overhead and NACKs. To illustrate CFDP's behavior as the payload increases, the error rate is set to zero and 100K, 1M, and 10M files are sent. All non-baseline tests (CFDP and otherwise) have the latency set to 25 ms and line rate set to 10 Mbps.

As we can see, the link is more efficiently used when the file size increases resulting in higher goodput as expected.

Since we now know to send larger files we will discover what happens when the channel emulator induces errors at various rates. For the remainder of the figures in this section the file size is fixed at 10 MB, the line rate is set to 10 Mbps, and the probability of a bit error in a packet is set to 0, 0.00001, 0.0001, 0.001, 0.01, and 0.1. As before, the goodput is used to show how much longer it takes to get payload data across a lossy link.

The purpose of showing the case where ten percent of the packets have a bit error is to show how gracefully the protocol drops off. Naturally, we would expect the link would either improve at this point or drop off completely. It is remarkable how little variance there is in the goodput until it suddenly starts to steeply drop off.

With the higher error rates, we would obviously expect the need for more retransmission of lost or corrupted packets. As the error varies, we plot the STR packet count. Figure 7 shows the effects on goodput by varying the overall data size. Figure 8 shows the effects on goodput by varying the error rate on the link. Figure 9 shows the effects of the number of sender-to-receiver packets transmitted by varying the error rate.

Figure 10 is intuitive after seeing Figure 9—however it more clearly illustrates the direct relation between error probability and variance as they affect the goodput.

Retransmission of lost or corrupted packets will occur when the receiver asks for it. The RTS packet count follows.

Even with an unrealistically high error rate, only about 70 KB worth of data are sent from the receiver to the sender to successfully transfer the 10 MB file.

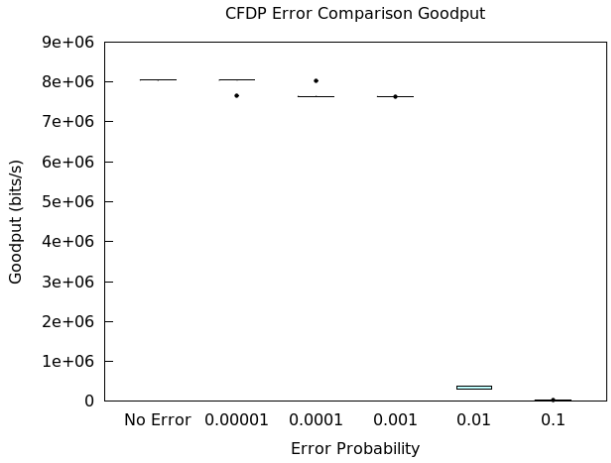


Figure 8.—The effects on goodput of CFDP by varying the error rate on the 10 Mbps constrained link.

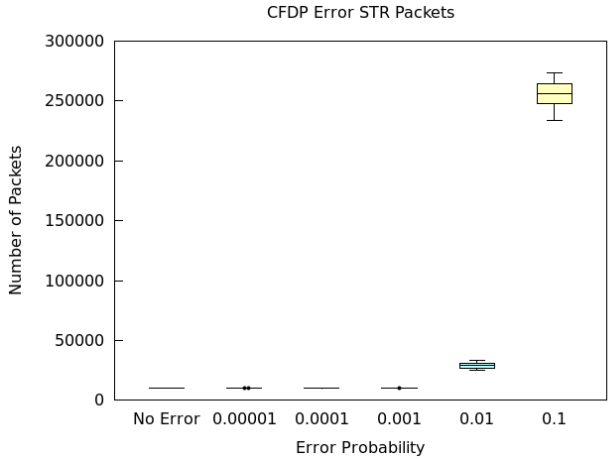


Figure 9.—The effects on the number of sender-to-receiver packets transmitted by CFDP by varying the error rate on a 10 Mbps constrained link.

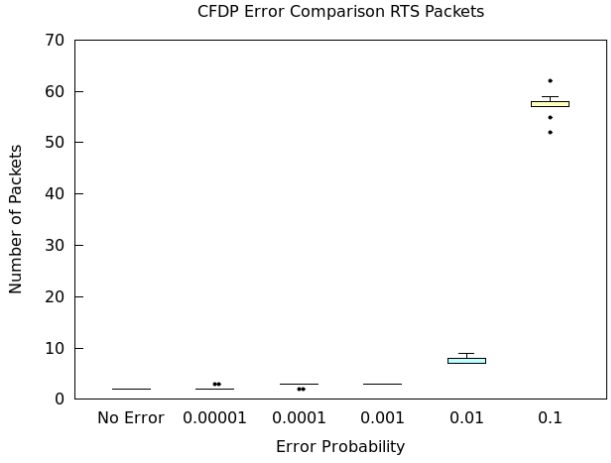


Figure 10.—This chart shows the effects of the number of receiver-to-sender packets transmitted by CFDP by varying the error rate on a 10 Mbps constrained link.

5.3 LTP

We begin the study of LTP with goodput versus file size plots shown in Figure 11 as we did with CFDP.

While the same relation holds, that is, the larger the payload the higher the goodput, the variance is much higher with LTP than CFDP. This appears characteristic of either LTPlib or LTP itself. In Figure 12, goodput is shown for the transmission of a 10 MB file with varying error rates.

This plot is very hard to interpret by itself since the best performing test runs of each variety are very similar. The 0.1 error rate case shows LTP outperforming CFDP in comparison charts. To try to understand what is happening in Figure 13 we plot STR packets versus error rates.

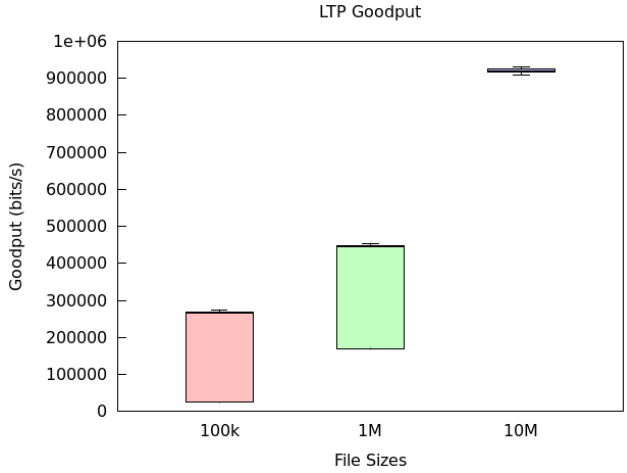


Figure 11.—The effects on the goodput of transferring different amounts of data using LTP over the 10 Mbps constrained link.

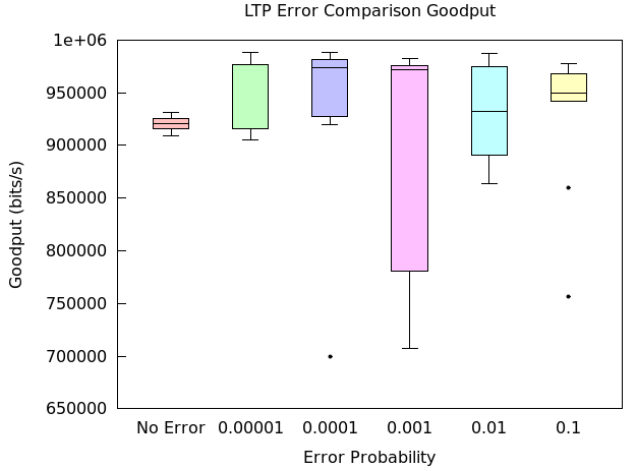


Figure 12.—The effects on goodput of LTP by varying the error rate on the 10 Mbps constrained link.

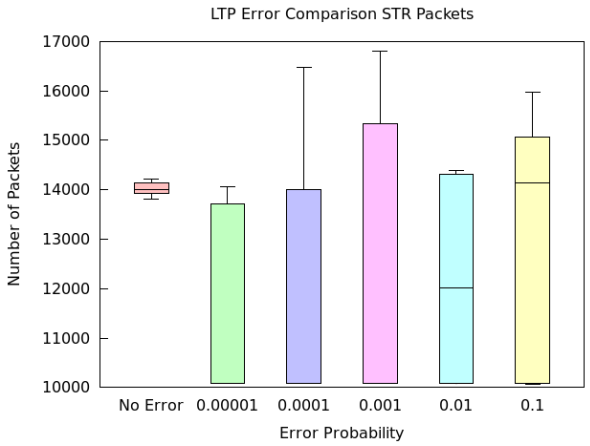


Figure 13.—The effects on the number of sender-to-receiver packets transmitted by LTP by varying the error rate on a 10 Mbps constrained link.

The case where there is no error exhibits comparatively low variance but shows thousands more packets are being sent than all other cases. When sending data over LTP in reliable mode, acknowledgments are used. To get the complete picture of the transmission we also need to visualize the RTS packet count.

LTP divides the data (including the header) into two parts: the *red* part and the *green* part. The red part denotes the data to be sent reliably with acknowledgment and in a typical LTP transfer, at least the header is marked in this manner. The green part is sent without positive or negative acknowledgment and sent as a best effort. Figure 14 shows that two ACKs are sent when there is no error. At the next lowest error-level, between 2 and 6 ACKs are sent. We see that with more two-way communication LTP retransmits less. This corresponds with Figure 12 where the goodput with an error probability of 0.00001 is at least as high as the goodput of the no-error case.

The behavior in Figure 12, Figure 13, and Figure 14 indicate that LTP is optimized for a lossy link.

5.4 NORM

As expected, NORM shows the same performance trend as CFDP and LTP. As file size increases, so does the goodput as shown in Figure 15.

Notice that the goodput of NORM is on the same order as CFDP for larger files but CFDP outperforms NORM when sending smaller 1 MB files. Figure 16 shows how NORM's goodput is affected by the error rate.

As expected, the variance increases with error and the goodput decreases with error. In Figure 17 the STR count is compared for various error rates.

The variance is minimal until the worst case. Even so, the progression is very clearly defined—unlike LTP. However, LTP still sends fewer packets from the sender to the receiver when the packet error rate is at 0.1. Since the communication is bi-directional, we have to consider the RTS packet traffic in our analysis.

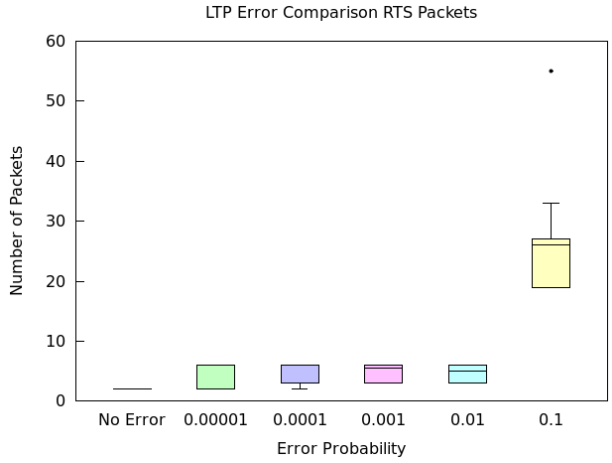


Figure 14.—The effects on the number of receiver-to-sender packets transmitted by LTP by varying the error rate on a 10 Mbps constrained link.

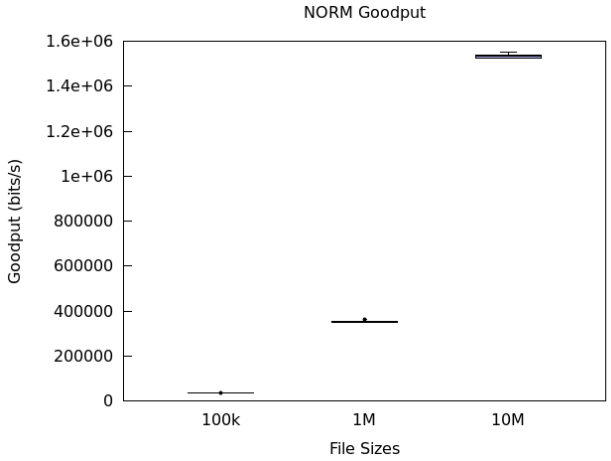


Figure 15.—The effects on the goodput of transferring different amounts of data using NORM over the 10 Mbps constrained link.

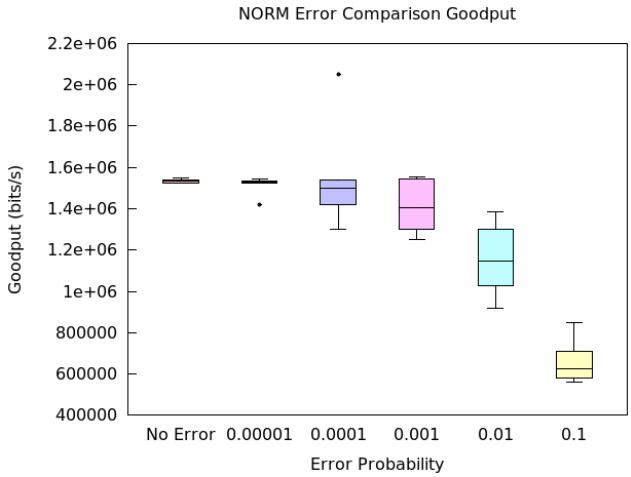


Figure 16.—The effects on goodput of NORM by varying the error rate on the 10 Mbps constrained link.

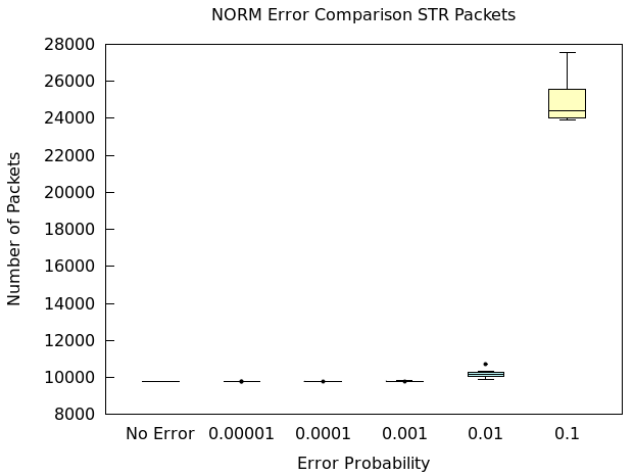


Figure 17.—The effects on the number of sender-to-receiver packets transmitted by NORM by varying the error rate on a 10 Mbps constrained link.

In Figure 18, the RTS numbers are similar between LTP and NORM (disregarding the outlier in LTP's worst-case) and they are both less chatty than CFDP.

5.5 Saratoga

As we have shown in the previous three protocol examples, Figure 19 shows Saratoga's goodput given a variety of file sizes.

Saratoga performs in the same class as CFDP and LTP in error-free conditions. A clear logarithmic relationship is depicted. Figure 20 illustrates the goodput roll-off as error rate increases. It is interesting that Saratoga outperformed the other protocols for error rates from 0 to 0.01 but it would not work with an error rate of 0.1 due to a decoding bug in the Perl implementation that arose when the packets became too corrupted. As such, we could not test the Saratoga v0 implementation at the 0.1 error rate reliably. It should be stressed that this is not an problem with the protocol itself, but rather the use of the Perl language in this particular implementation that gave rise a bug with highly corrupted data packets.

The variances in Figure 20 appear wider than the competition, but keep the scale in mind since the 0.1-error case is not included. In Figure 21 we see the STR traffic for Saratoga.

The high error case shows a STR packet increase of roughly 100 packets. CFDP's STR packet count increased nearly 20000 from 0 error to 0.01 error.

Figure 22 illustrates the RTS packet count for Saratoga.

It is seen that Saratoga is minimally chatty for error under 0.01. At the 0.01 error level, CFDP sends out between 7 and 9 packets from the receiver to the sender. Similarly, NORM and LTP send fewer packets back to the sender at Saratoga's maximum error rate.

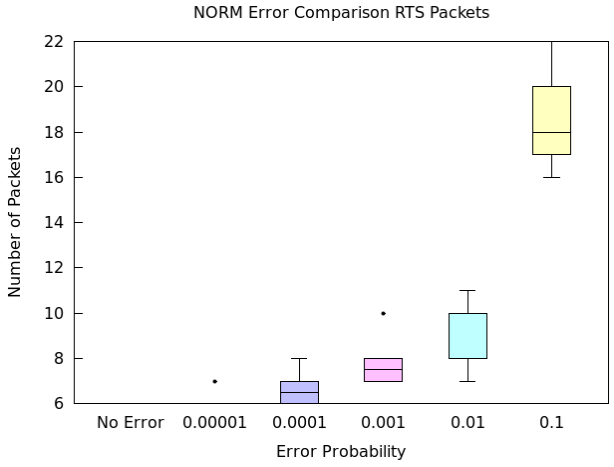


Figure 18.—The effects on the number of receiver-to-sender packets transmitted by NORM by varying the error rate on a 10 Mbps constrained link.

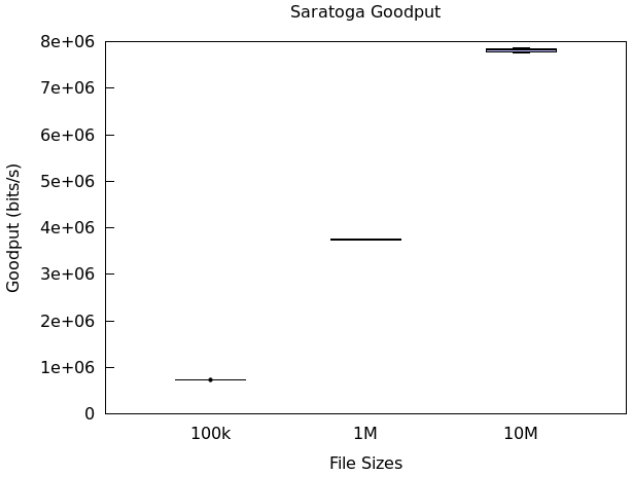


Figure 19.—The effects on the goodput of transferring different amounts of data using Saratoga over the 10 Mbps constrained link.

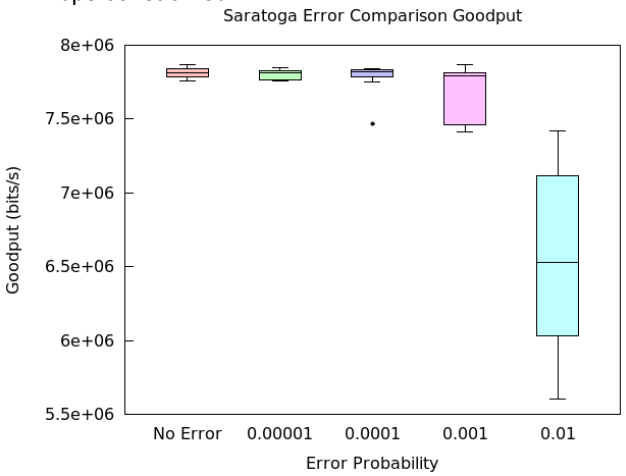


Figure 20.—The effects on goodput of Saratoga by varying the error rate on the 10 Mbps constrained link.

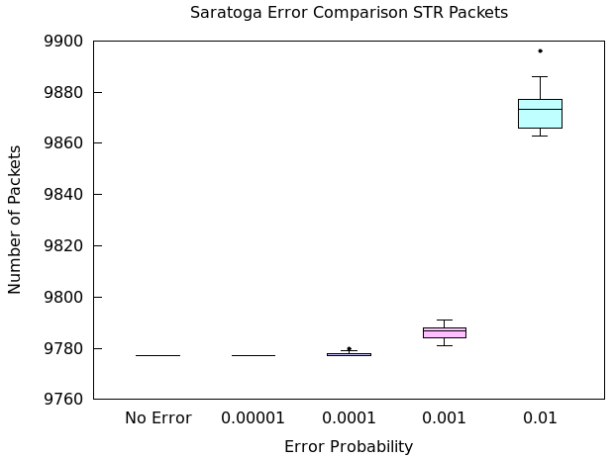


Figure 21.—The effects on the number of sender-to-receiver packets transmitted by Saratoga by varying the error rate on a 10 Mbps constrained link.

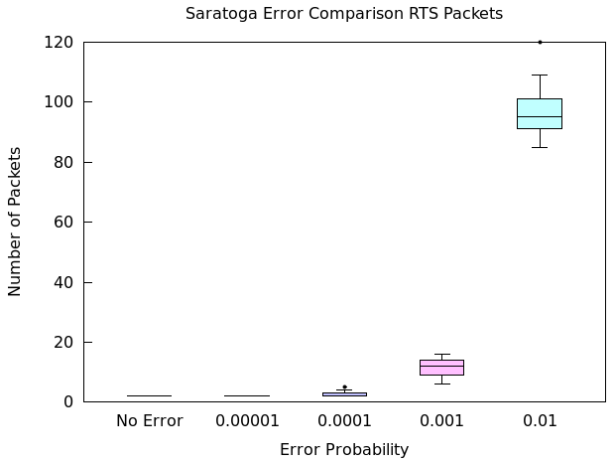


Figure 22.—The effects on the number of receiver-to-sender packets transmitted by Saratoga by varying the error rate on a 10 Mbps constrained link.

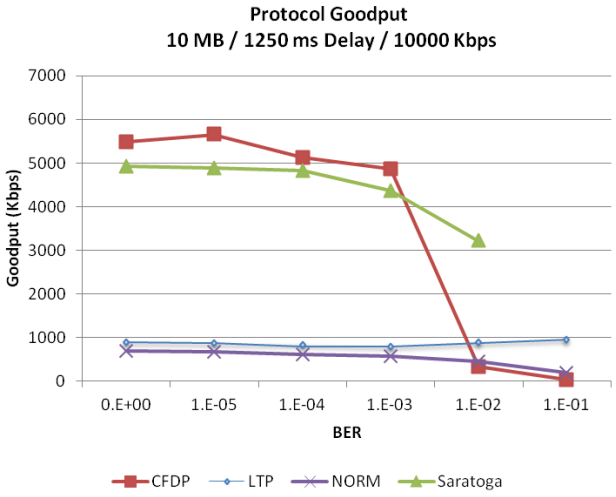


Figure 23.—Protocol goodput comparison of 10 MB file transfer at 1250 ms one-way delay at 10 Mbps.

5.6 Protocol Packet Analysis

In the following sections we took a closer look at a couple of example file transfer scenarios for each transport protocol using tcpdump packet captures recorded during a selected run to illustrate the variation in the number of packets needed to successfully transmit the file and cleanly terminate the session. We attempted to determine why some protocol implementations show a wide disparity in their performance in different link conditions compared to similar protocols to determine whether it was characteristic of the protocol itself or the implementation of the particular implementation we were testing.

5.6.1 Example 1, Large Files, High Bandwidth

The following example file transfers occur at a 10 Mbps line rate, with a 1.25 s one-way delay, and 0.0001 error rate on a 10 MB file. Figure 23 illustrates the goodput results of a high bandwidth, large file size transfer across varying error rates with a high delay.

5.6.1.1 CFDP

CFDP's file transfer termination phase finishes much quicker than LTP or NORM and results in overall better performance than these protocols due to the small amount of time it waits before it concludes that the file was successfully transmitted.

```
19:11:47.011739 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
19:11:47.012622 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
19:11:47.013507 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
19:11:47.014390 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 104
19:11:47.014562 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 20
19:11:47.014831 IP 192.168.100.2.2502 > 192.168.100.1.2501: UDP, length 13
19:11:47.015065 IP 192.168.100.2.2502 > 192.168.100.1.2501: UDP, length 43
19:11:49.518932 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
19:11:49.523781 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
19:11:49.523821 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
19:11:49.587337 IP 192.168.100.2.2502 > 192.168.100.1.2501: UDP, length 12
19:11:52.087918 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 13
```

5.6.1.2 Saratoga

Saratoga, like CFDP, concludes the file transfer phase of waiting for NACKs very quickly upon finishing sending the data from the sender to the receiver.

```
13:12:40.716883 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
13:12:40.717785 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
13:12:40.718692 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 650
13:12:40.783605 IP 192.168.100.2.4000 > 192.168.100.1.4000: UDP, length 7
13:12:40.888240 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 18
13:12:41.889693 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 18
13:12:42.891122 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 18
```

5.6.1.3 LTP

LTP's file transfer termination phase stays at the WAIT_RP_ACK state waiting for acknowledgment from the receiver that it has received all of the "red" parts before the transfer can conclude. (Ref. 5) This appears to contribute to poor performance when the file transfer is very small or there is a large amount of latency. It may be possible to adjust the default timers in LTPlib to help offset this to improve performance.

```
12:12:13.683679 IP 192.168.100.1.52059 > 192.168.100.2.1113: UDP, length 1012
12:12:13.704069 IP 192.168.100.1.55879 > 192.168.100.2.1113: UDP, length 1012
12:12:13.724461 IP 192.168.100.1.46087 > 192.168.100.2.1113: UDP, length 1012
12:12:13.725345 IP 192.168.100.1.56090 > 192.168.100.2.1113: UDP, length 1012
12:12:13.745081 IP 192.168.100.1.54561 > 192.168.100.2.1113: UDP, length 663
12:12:23.267041 IP 192.168.100.1.39980 > 192.168.100.2.1113: UDP, length 663
12:12:33.244261 IP 192.168.100.1.50456 > 192.168.100.2.1113: UDP, length 663
12:12:37.648531 IP 192.168.100.2.60750 > 192.168.100.1.1113: UDP, length 73
12:12:37.648764 IP 192.168.100.2.47909 > 192.168.100.1.1113: UDP, length 73
12:12:37.700821 IP 192.168.100.2.36164 > 192.168.100.1.1113: UDP, length 73
12:12:43.242798 IP 192.168.100.1.43766 > 192.168.100.2.1113: UDP, length 13
12:12:43.280137 IP 192.168.100.2.48385 > 192.168.100.1.1113: UDP, length 12
12:12:47.030608 IP 192.168.100.2.51587 > 192.168.100.1.1113: UDP, length 73
12:12:47.030753 IP 192.168.100.2.33117 > 192.168.100.1.1113: UDP, length 73
12:12:47.051047 IP 192.168.100.2.51706 > 192.168.100.1.1113: UDP, length 73
```

5.6.1.4 NORM

With NORM, the file transfer termination phase is quite a bit longer and occurs over the span of several minutes. In example 2, we detail how this is the result of the random back-off timers to prevent flooding the multicast sender with feedback (Note: NORM is optimized for multicast operations, not unicast.). This obviously has a large impact on the goodput when we include this end-of-transfer wait in our calculations. If we were to ignore the nearly 2-minute wait, NORM would be much closer to Saratoga and CFDP in terms of speed of the actual transfer as it is just blasting UDP packets over the channel.

```
21:43:30.081301 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 1056
21:43:30.082220 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 1056
21:43:30.083141 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 1056
21:43:30.084061 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 1056
21:43:30.084981 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 672
21:43:30.085599 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:43:34.280349 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
21:43:34.519335 IP 192.168.100.2.37829 > 224.1.2.3.12347: UDP, length 48
21:43:34.689731 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:43:41.874831 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
21:43:50.414888 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 1056
21:43:50.415801 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:43:53.266676 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
21:43:55.770422 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:01.124635 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:02.209320 IP 192.168.100.2.37829 > 224.1.2.3.12347: UDP, length 36
21:44:06.480576 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:10.357683 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
21:44:11.835213 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:17.190228 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:22.544591 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:27.899326 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:33.253640 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:35.989226 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
21:44:38.608427 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:42.246225 IP 192.168.100.2.37829 > 224.1.2.3.12347: UDP, length 36
21:44:43.963575 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:49.318088 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:44:54.672966 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:00.028188 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:05.383626 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:05.991198 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
21:45:10.738577 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:12.850224 IP 192.168.100.2.37829 > 224.1.2.3.12347: UDP, length 36
21:45:16.092833 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:21.447353 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:26.802354 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:32.156632 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 20
21:45:35.994465 IP 192.168.100.1.51946 > 224.1.2.3.12347: UDP, length 28
```

5.6.2 Example 2, Small Files, Low Bandwidth

In comparison to the large file transfer, using a very small file of 10 KB, transferred at 1 Mbps with an error rate of 0.0001, we can see the entire packet capture to illustrate the effects of the 1.25 s one-way delay on the file transfer with each of the protocols. Figure 24 illustrates the goodput results of a low bandwidth, small file size transfer across varying error rates with a high delay.

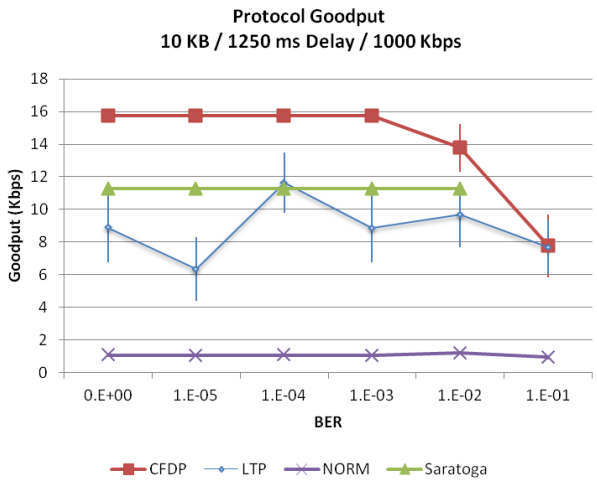


Figure 24.—Protocol goodput comparison of 10 KB file transfer at 1250 ms one-way delay at 1 Mbps.

5.6.2.5 CFDP

```
05:11:09.129333 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 42
05:11:09.130082 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.138363 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.146649 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.154933 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.163215 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.171500 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.179785 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.188067 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.196357 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.204637 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 1011
05:11:09.212921 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 44
05:11:09.213653 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 20
05:11:09.214258 IP 192.168.100.2.2502 > 192.168.100.1.2501: UDP, length 13
05:11:09.214436 IP 192.168.100.2.2502 > 192.168.100.1.2501: UDP, length 12
05:11:11.715268 IP 192.168.100.1.2501 > 192.168.100.2.2502: UDP, length 13
```

As shown, CFDP is very efficient at transferring the file as fast as possible over channel starting with the file metadata and then the contents of the file itself followed by acknowledgements from the receiver.

5.6.2.6 Saratoga

```
01:34:06.898516 IP 192.168.100.2.4000 > 192.168.100.1.4000: UDP, length 32
01:34:09.400220 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 10
01:34:09.408768 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 266
01:34:09.417388 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.426008 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.434617 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.443226 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.451836 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.460470 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.469085 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.477693 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.486307 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 1034
01:34:09.494914 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 794
01:34:09.495976 IP 192.168.100.2.4000 > 192.168.100.1.4000: UDP, length 7
01:34:10.496302 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 18
01:34:11.497755 IP 192.168.100.1.4000 > 192.168.100.2.4000: UDP, length 18
```

As observed, Saratoga and CFDP's performance is almost entirely impacted by the small number of packets transmitted vs. the large round trip latency for the NACK processing in these examples. Saratoga's file transfer is very similar to CFDP with the exception of requiring the initial "get" packet from the receiver due to the limitations of the v0 protocol. Once the sender receives the "get" request, it sends the metadata and contents of the file in the ensuing packets in a similar manner as CFDP. The file transfer is then concluded

by the receiver acknowledging the transmission and, if necessary, requesting the retransmission of any missing blocks of data via NACKs.

5.6.2.7 LTP

```
00:20:27.634931 IP 192.168.100.1.42791 > 192.168.100.2.1113: UDP, length 1009
00:20:27.643204 IP 192.168.100.1.60433 > 192.168.100.2.1113: UDP, length 1010
00:20:27.655298 IP 192.168.100.1.59690 > 192.168.100.2.1113: UDP, length 1010
00:20:27.675556 IP 192.168.100.1.51936 > 192.168.100.2.1113: UDP, length 1010
00:20:27.695793 IP 192.168.100.1.41218 > 192.168.100.2.1113: UDP, length 1010
00:20:27.716054 IP 192.168.100.1.47606 > 192.168.100.2.1113: UDP, length 1010
00:20:27.736278 IP 192.168.100.1.36030 > 192.168.100.2.1113: UDP, length 1010
00:20:27.756563 IP 192.168.100.1.58628 > 192.168.100.2.1113: UDP, length 1010
00:20:27.776804 IP 192.168.100.1.39097 > 192.168.100.2.1113: UDP, length 1010
00:20:27.797062 IP 192.168.100.1.59594 > 192.168.100.2.1113: UDP, length 1010
00:20:27.817272 IP 192.168.100.1.39914 > 192.168.100.2.1113: UDP, length 100
00:20:27.845487 IP 192.168.100.2.33163 > 192.168.100.1.1113: UDP, length 24
00:20:30.368747 IP 192.168.100.1.47307 > 192.168.100.2.1113: UDP, length 15
```

LTP is similar to CFDP and Saratoga in that it is waiting for the NACK to be processed and to receive the reply from the sender to conclude that the transmission is finished and is mostly affected by network latency at these parameters.

5.6.2.8 NORM

```
16:57:21.057915 IP 192.168.100.2 > 224.0.0.22: igmp v3 report, 1 group record(s)
16:57:22.525799 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:22.526493 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 40
16:57:22.527145 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.535765 IP 192.168.100.1 > 224.0.0.22: igmp v3 report, 1 group record(s)
16:57:22.536294 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.544934 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.553574 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.562211 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.570849 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.579488 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.588123 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.596762 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 1056
16:57:22.605405 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 816
16:57:22.612178 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:23.141711 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:23.275457 IP 192.168.100.2.42335 > 224.1.2.3.12347: UDP, length 36
16:57:23.525241 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:23.674569 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:24.127864 IP 192.168.100.2 > 224.0.0.22: igmp v3 report, 1 group record(s)
16:57:24.207837 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:24.443204 IP 192.168.100.1 > 224.0.0.22: igmp v3 report, 1 group record(s)
16:57:24.740769 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:25.025455 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:25.273794 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:25.799631 IP 192.168.100.2.42335 > 224.1.2.3.12347: UDP, length 36
16:57:25.806569 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:27.276628 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:29.743414 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:30.651056 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:32.500036 IP 192.168.100.2.42335 > 224.1.2.3.12347: UDP, length 36
16:57:34.701698 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:35.712811 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:39.663751 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:43.307130 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:44.621940 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:48.983011 IP 192.168.100.2.42335 > 224.1.2.3.12347: UDP, length 36
16:57:49.582124 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:54.540280 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:57:54.698902 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:57:59.894738 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:05.249571 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:10.608750 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:11.786995 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
16:58:15.963021 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:19.537548 IP 192.168.100.2.42335 > 224.1.2.3.12347: UDP, length 36
16:58:21.317683 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:26.672691 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:32.029043 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 20
16:58:37.416924 IP 192.168.100.1.35454 > 224.1.2.3.12347: UDP, length 28
```

Note that there is approximately 1 min and 15 s between the conclusion of the file transfer and the end of the NACK processing phase, at which point there is no further communication between nodes.

The NORM console output on the sender side during the file transfer shows:

```
Proto Info: ProtoDebug>SetDebugLevel: debug level changed from 1 to 10
Proto Info: 21:57:21.273860 enqueued tx object>41904 sender>8471
Proto Debug: NormApp::Notify(TX_OBJECT_SENT) ...
Proto Debug: NormApp::Notify(TX_QUEUE_EMPTY) ...
Proto Fatal: norm: End of tx file list reached.
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:1)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:2)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:3)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:4)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:5)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:6)...
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::SenderUpdateGrttEstimate() node>8471 increased to new grtt>1.968000 sec
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:7)...
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::SenderUpdateGrttEstimate() node>8471 increased to new grtt>2.479000 sec
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:8)...
Proto Info: REPORT time>21:57:31.273127 node>8471 ****************************************
Proto Info: Local status:
Proto Info:   txRate> 1000.000 kbps sentRate>   8.528 grtt>2.479000
Proto Info: ************************************************************************
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:9)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:10)...
Proto Info: REPORT time>21:57:41.277039 node>8471 ****************************************
Proto Info: Local status:
Proto Info:   txRate> 1000.000 kbps sentRate>   0.054 grtt>2.479000
Proto Info: ************************************************************************
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:11)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:12)...
Proto Debug: NormApp::Notify() unhandled event: 20
Proto Debug: NormSession::SenderUpdateGrttEstimate() node>8471 increased to new grtt>2.677000 sec
Proto Info: REPORT time>21:57:51.279509 node>8471 ****************************************
Proto Info: Local status:
Proto Info:   txRate> 1000.000 kbps sentRate>   0.054 grtt>2.677000
Proto Info: ************************************************************************
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:13)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:14)...
Proto Info: REPORT time>21:58:01.281004 node>8471 ****************************************
Proto Info: Local status:
Proto Info:   txRate> 1000.000 kbps sentRate>   0.054 grtt>2.677000
Proto Info: ************************************************************************
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:15)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:16)...
Proto Info: REPORT time>21:58:11.282546 node>8471 ****************************************
Proto Info: Local status:
Proto Info:   txRate> 1000.000 kbps sentRate>   0.054 grtt>2.677000
Proto Info: ************************************************************************
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:17)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:18)...
Proto Info: REPORT time>21:58:21.284552 node>8471 ****************************************
Proto Info: Local status:
Proto Info:   txRate> 1000.000 kbps sentRate>   0.032 grtt>2.677000
Proto Info: ************************************************************************
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:19)...
Proto Debug: NormSession::SenderQueueFlush() node>8471, flush queued (flush_count:20)...
Proto Trace: NormSession::Serve() node>8471 sender flush complete ...
Proto Debug: NormApp::Notify(TX_FLUSH_COMPLETED)
```

On the NORM receiver during the file transfer, the console output shows:

```
Proto Info: ProtoDebug>SetDebugLevel: debug level changed from 1 to 10
Proto Debug: NormApp::Notify() unhandled event: 10
Proto Debug: NormSession::ReceiverHandleCommand() node>8472 new remote sender:8471 ...
Proto Debug: NormApp::Notify(REMOTE_SENDER_ACTIVE) ...
Proto Trace: NormSenderNode::HandleCommand() node>8472 begin CC back-off: 0.748614 sec)...
Proto Debug: NormApp::Notify(RX_OBJECT_NEW) ...
Proto Info: 21:57:22.507198 start rx object>41904 sender>14319463051248541975
Proto Detail: NormSenderNode::HandleObjectMessage() node>8472 sender>8471 new obj>41904
Proto Debug: NormApp::Notify(RX_OBJECT_INFO) ...
Proto Debug: NormSenderNode::HandleObjectMessage() node>8472 allocating sender>8471 buffers ...
Proto Trace: NormApp::Notify(RX_OBJECT_UPDATED) ...
Proto Detail: NormObject::HandleObjectMessage() node>8472 sender>8471 obj>41904 blk>0 completed
block ...
Proto Debug: NormApp::Notify(RX_OBJECT_COMPLETED)
```

The NORM NACK delays are apparently part of the protocol and are caused by the receiver scheduling random back-off timeouts before generating more NACK messages. It uses probabilistic suppression of redundant feedback based on exponentially distributed random back-off timers (Ref. 6). This behavior is implemented due to its nature as a multicast

protocol in order to avoid overwhelming the sender with potentially similar NACK replies from large groups of recipients at the same time requesting the retransmission of the same corrupted or lost parts of the file.

6.0 Conclusion

The purpose of this paper was to perform a comparative analysis of several transport layer protocol implementations to determine the ideal operating conditions to maximize performance. The various environments that the protocols were subjected to are characteristic of those found in terrestrial, geostationary orbit, and cis-lunar communications links. Communications over the latter two types of links are often subjected to restrictions that do not apply to typical terrestrial links such as limited contact time, high delay, and limited power. Therefore, while mitigating the problems inherent in the reliability, integrity, and availability of the link itself, the protocols must also be able to maximize performance while the link is available.

CFDP performed consistently well across the entire collection of tests and satisfied our expectations that it would be able to handle the problems related to typical cis-lunar space links well considering that was the intent of its original design by CCSDS.

Saratoga likewise performed very well across most of the test scenarios matching and occasionally even exceeding CFDP's performance in many of the tests. We expected that due to the design of the Saratoga v0 protocol and lack of a "put" option, the higher-delay links would pose a performance problem considering the file transmission could not begin until the sender transmitted a "get" packet to the receiver. As seen in the appendix, this did in fact seem to affect performance slightly across the range of tests as Saratoga usually suffered a performance penalty from having to process the "get" request to initiate the transfer. If a future version of Saratoga such as version 1 is able to successfully implement a push-based file transfer initiated by the sender then it is very likely Saratoga will continue to meet and even exceed the performance of CFDP in higher-delay communications links. As it stands now, Saratoga appears best suited for low and moderate-delay communications links below geostationary orbit. One particularly frustrating problem faced with Saratoga v0's Perl implementation was a bug that exhibited itself during the 10 percent packet error tests. Saratoga could not properly handle the highly corrupted packets and the implementation would crash. Due to this problem we had to exclude the handful of successful runs captured at 10 percent packet error rate because they were not common enough and were too difficult to reproduce in a consistent fashion to find 10 successful runs in a row.

NORM was also highly reliable and its performance was consistent across a wide variety of link conditions, however the multicast-oriented design best suits those particular environments where data must be transferred to a large amount of receiver nodes at once. The random back-off timer used to avoid flooding the sender with NACKs did cause a hit to goodput performance during testing, but in a typical multicast environment where this protocol would be deployed this behavior is both expected and desired. If we were to exclude the delays associated with handling the NACK processing phase of the file transfer and instead concentrated solely on the unidirectional transfer of the file data itself, NORM would have performed just as well as CFDP or Saratoga considering it was simply blasting out UDP packets filled with data. It is the protocol's designs to accommodate the multicast environment that ultimately limit its performance in a field of unicast-oriented protocols.

LTP was expected to perform similarly to CFDP since the protocols are related and the design of the file transfer process is very similar, however this did not seem to be the case in practice with our LTPlib (Ref. 10) implementation. The variances in LTP observed using LTPlib are significantly higher than the other protocols tested and seem indicative of a problem with the version of the software implementation rather than the protocol design itself. It would be interesting to go back and compare the results of our testing against another independent standalone implementation of LTP such as the one from Ohio University written in Java (Ref. 13).

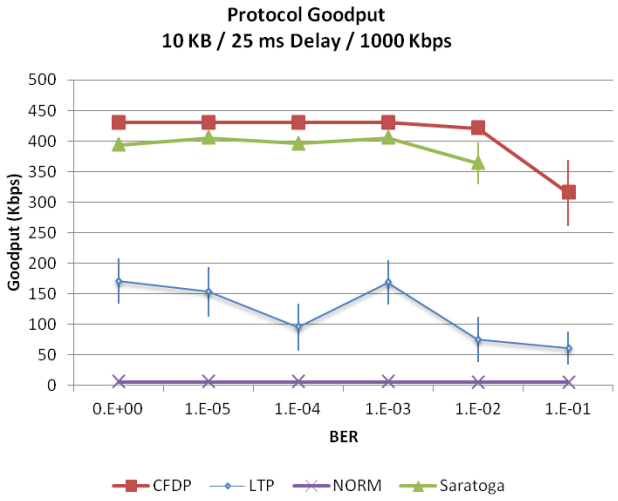
Individual implementation problems aside, the design and performance aspects of each of these protocols seems like they would be suitable for use as a convergence layer in concert with the Bundle Protocol under the kinds of DTNs deployed in space communications links. Convergence layer implementations of several of these transport protocols already exist for popular DTN implementations including ION (LTP) (Ref. 14) and DTN2 (NORM, LTP) (Ref. 15). An experimental Internet-draft document for explaining how a DTN bundle agent could be implemented with Saratoga as a convergence layer has been proposed as well (Ref. 16).

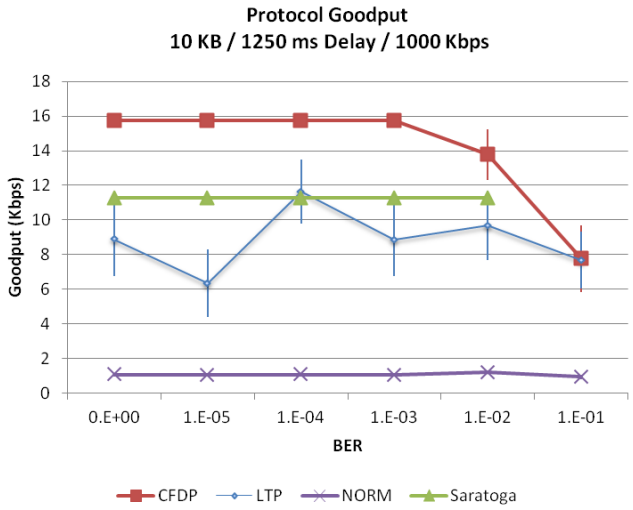
These tests show the importance of understanding and testing the particular convergence layer protocol and implementation prior to doing DTN network tests and the convergence layer performance will likely significantly effect to overall DTN network performance.

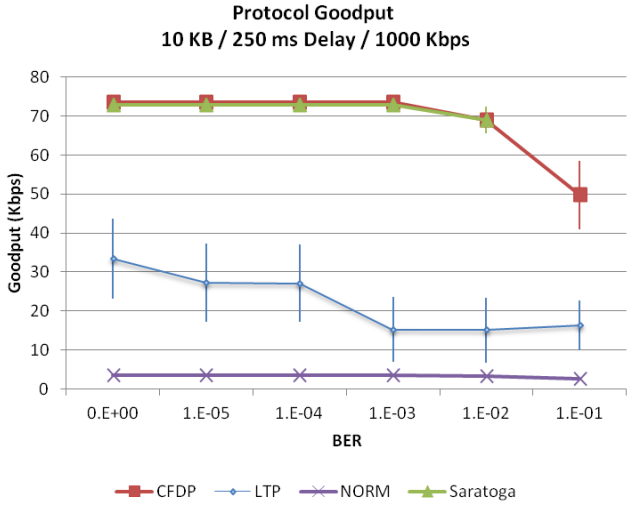
Appendix

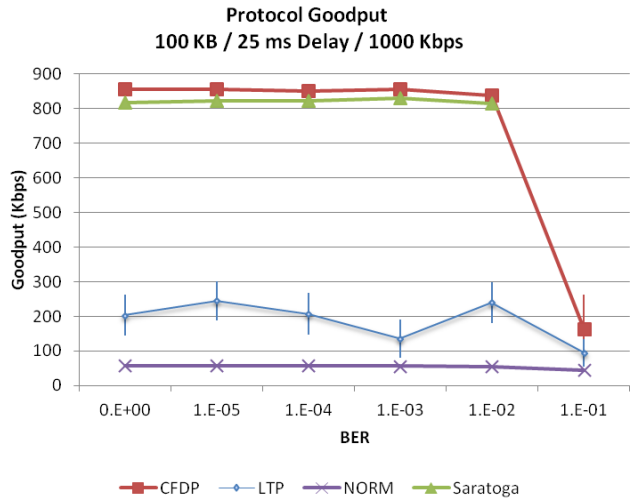
The following graphs were included to show the breadth of our tests keeping in mind it is not feasible to individually analyze every permutation of test run. The graphs may be used to supplement the analysis in order to satisfy the particular interests of the reader. The charts below show the average results of the protocols across a set of different file sizes (10 KB, 100 KB, 1 MB, and 10 MB) in a variety of link conditions where we vary the latency, throughput, and error rate. These charts were created using the average of the runs to give a closer look at the performance trends as the link conditions vary. Error bars were included to show the standard error of the mean for the test runs.

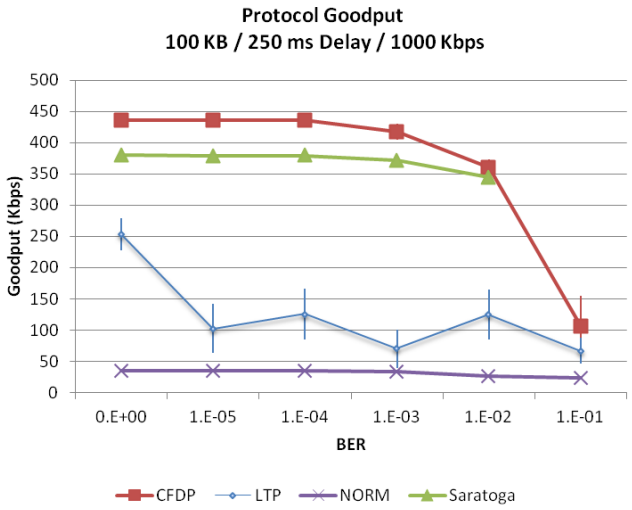
Keeping in mind that no data was collected for Saratoga when the error was at 0.1, we see that Saratoga and CFDP performed very similarly. The random back-off timers that are employed for the NACK phase of the file transfer impact NORM clearly here. The variance and lack of a trend in LTP is illustrated.

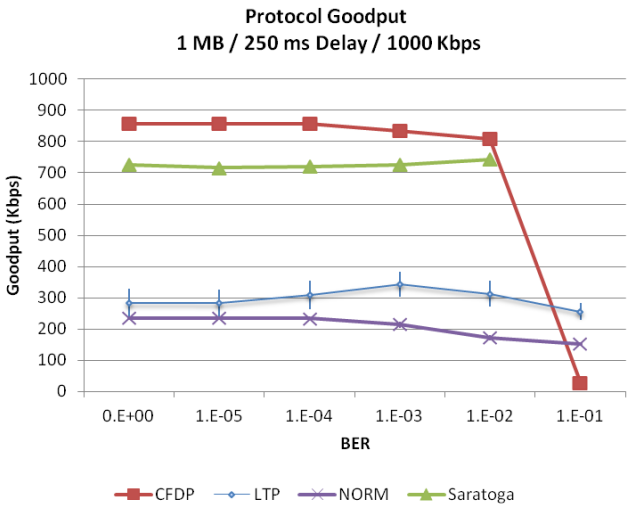


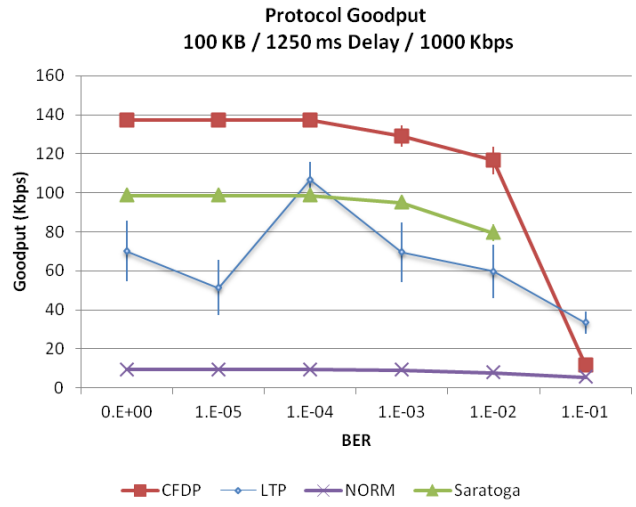


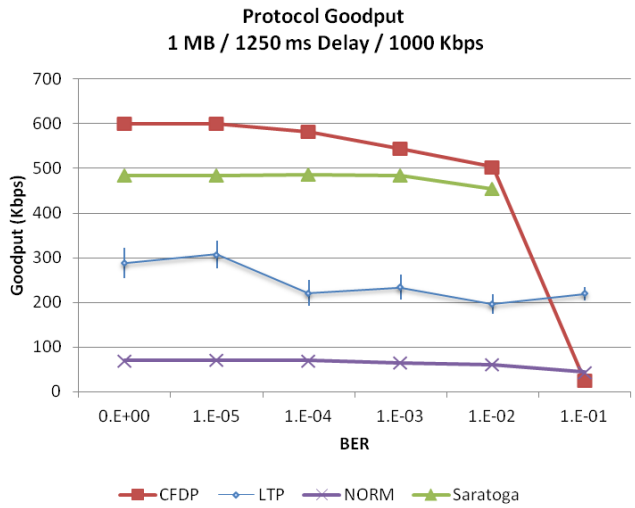


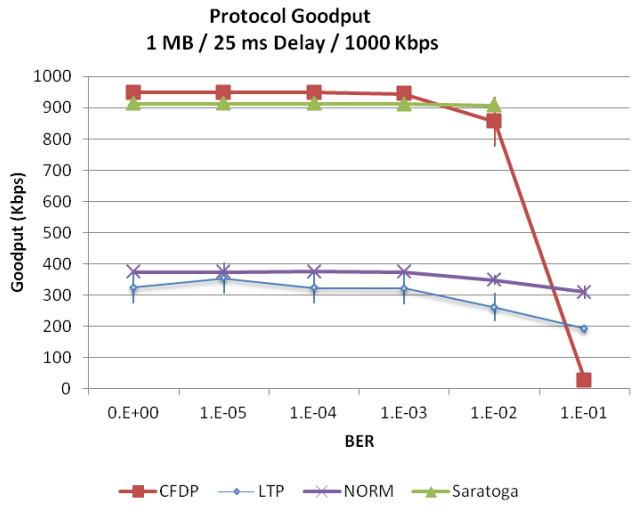


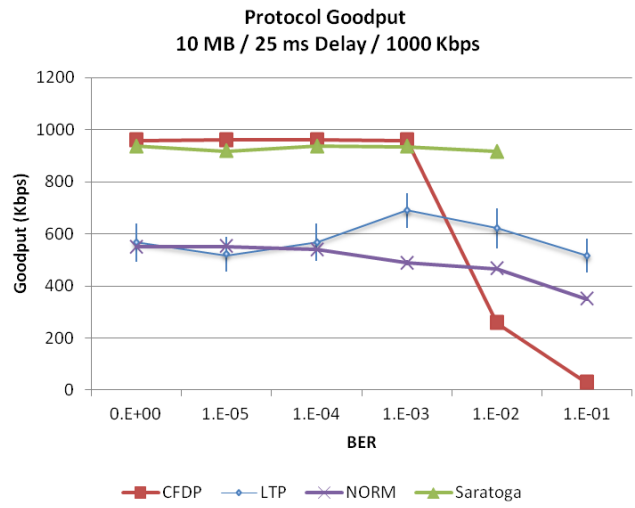

Protocol Goodput
100 KB / 250 ms Delay / 1000 Kbps
Goodput (Kbps)
500
450
400
350
300
250
200
150
100
50
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

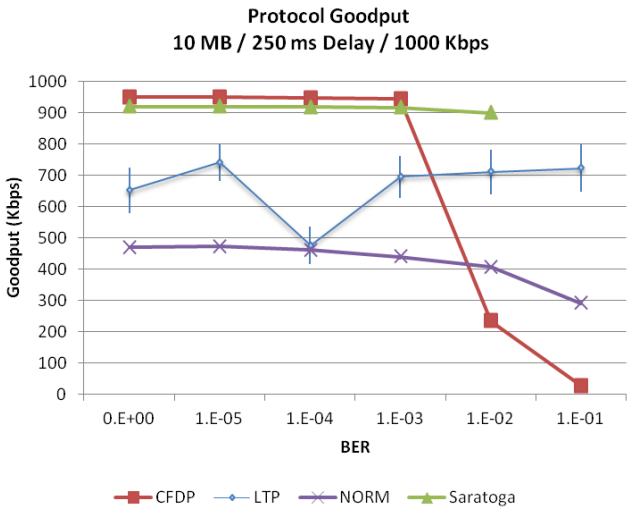

Protocol Goodput
1 MB / 250 ms Delay / 1000 Kbps
Goodput (Kbps)
1000
900
800
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

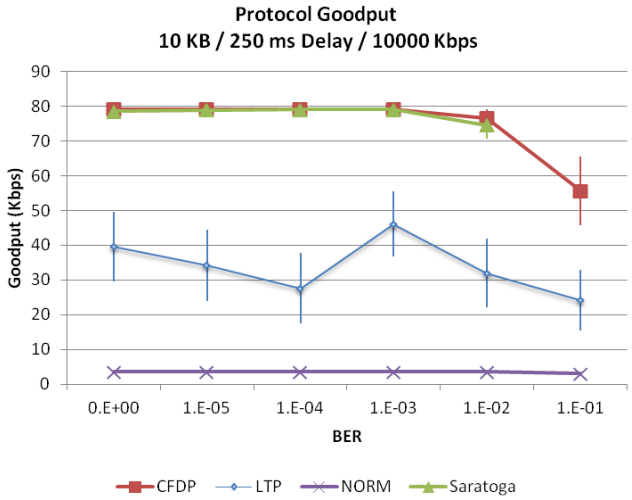

Protocol Goodput
100 KB / 1250 ms Delay / 1000 Kbps
Goodput (Kbps)
160
140
120
100
80
60
40
20
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

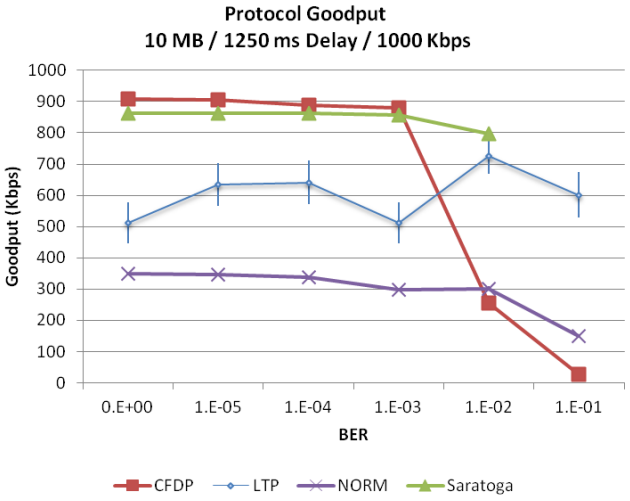

Protocol Goodput
1 MB / 1250 ms Delay / 1000 Kbps
Goodput (Kbps)
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

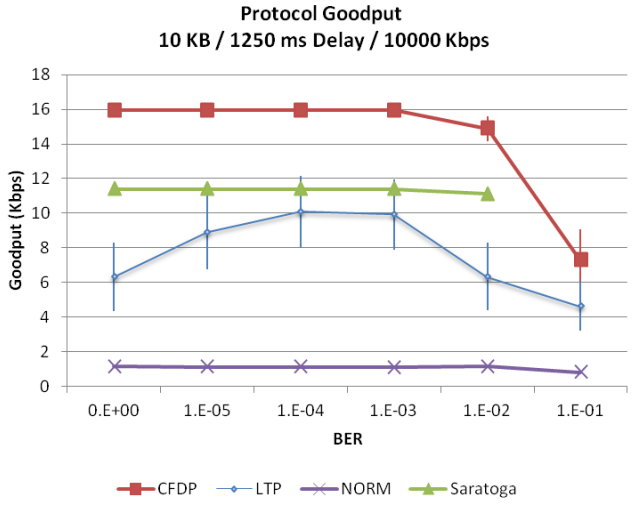

Protocol Goodput
1 MB / 25 ms Delay / 1000 Kbps
Goodput (Kbps)
1000
900
800
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

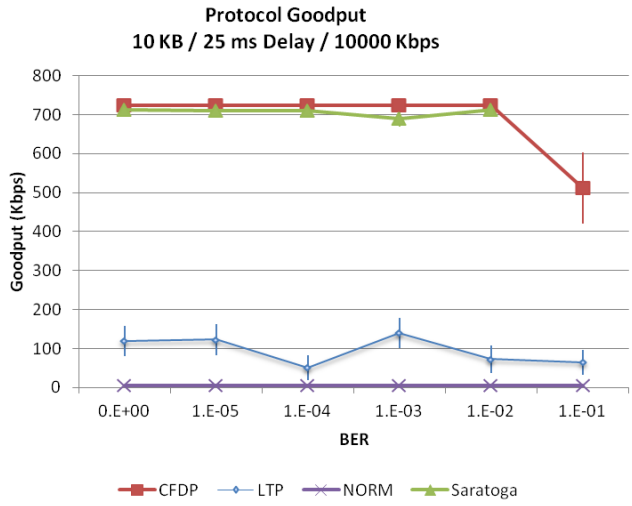

Protocol Goodput
10 MB / 25 ms Delay / 1000 Kbps
Goodput (Kbps)
1200
1000
800
600
400
200
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

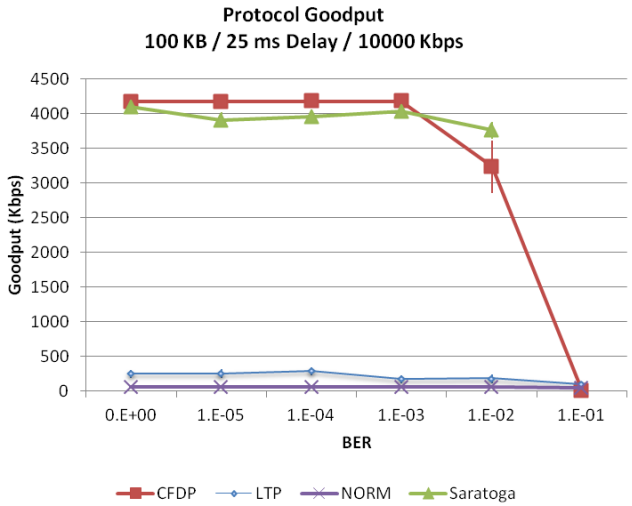

Protocol Goodput
10 MB / 250 ms Delay / 1000 Kbps
Goodput (Kbps)
1000
900
800
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

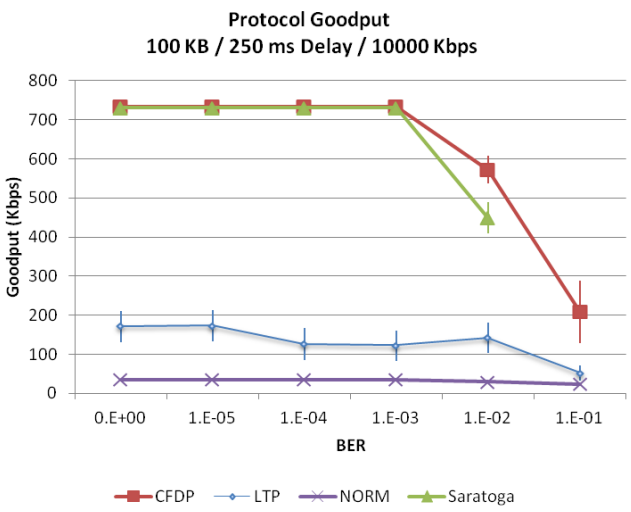

Protocol Goodput
10 KB / 250 ms Delay / 10000 Kbps
Goodput (Kbps)
90
80
70
60
50
40
30
20
10
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

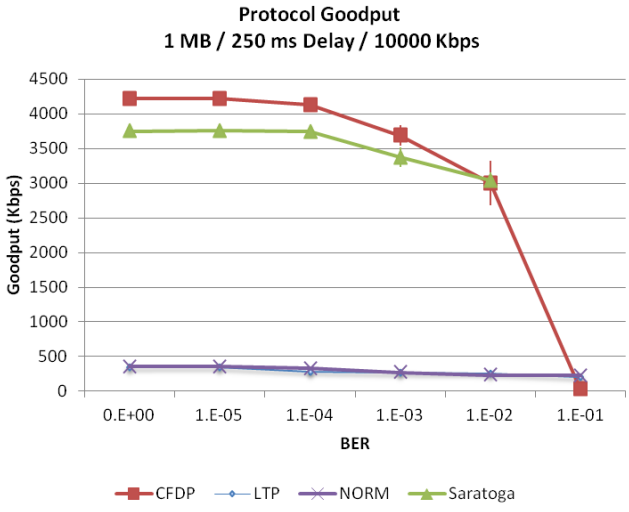

Protocol Goodput
10 MB / 1250 ms Delay / 1000 Kbps
Goodput (Kbps)
1000
900
800
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

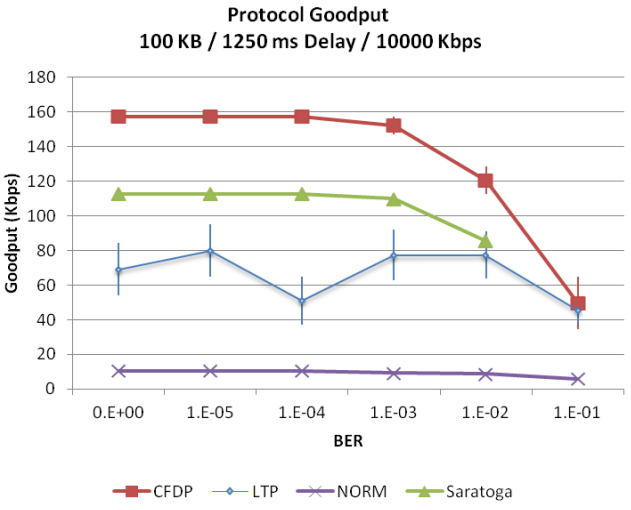

Protocol Goodput
10 KB / 1250 ms Delay / 10000 Kbps
Goodput (Kbps)
18
16
14
12
10
8
6
4
2
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

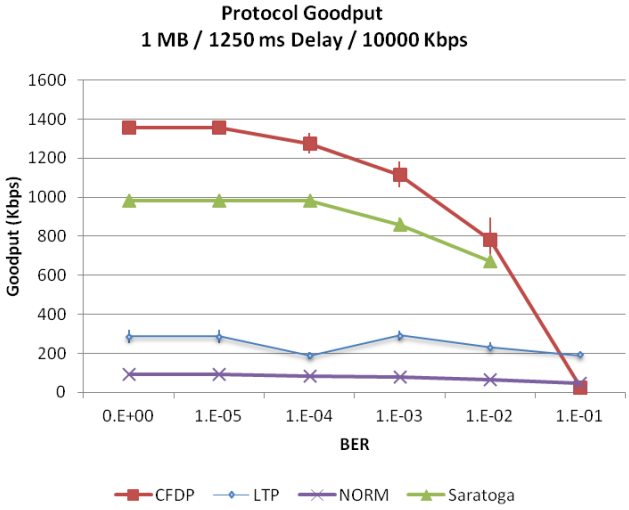

Protocol Goodput
10 KB / 25 ms Delay / 10000 Kbps
Goodput (Kbps)
800
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

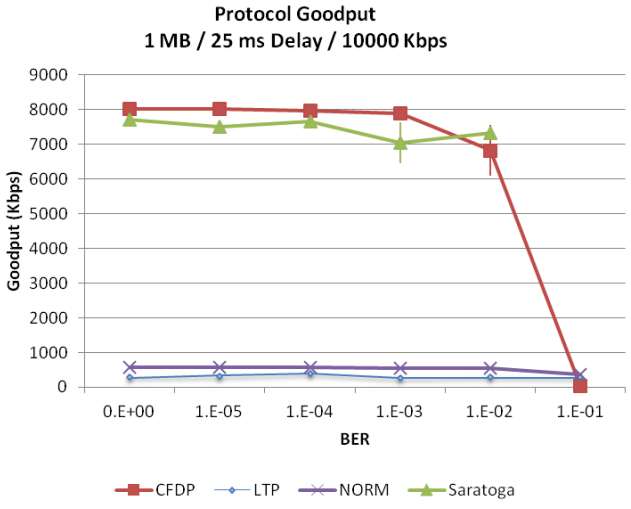

Protocol Goodput
100 KB / 25 ms Delay / 10000 Kbps
Goodput (Kbps)
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

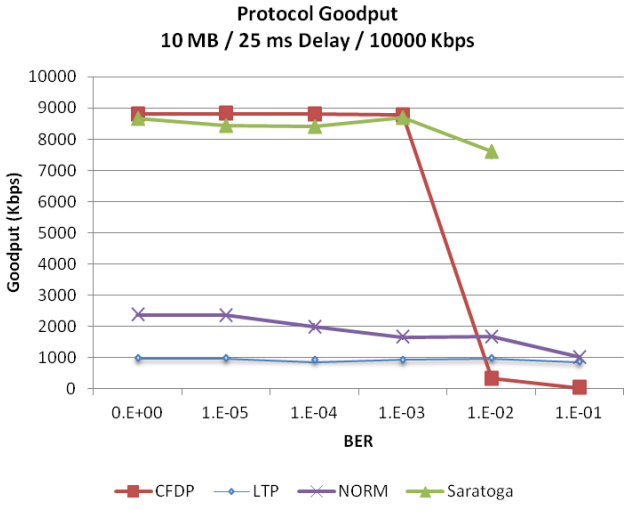

Protocol Goodput
100 KB / 250 ms Delay / 10000 Kbps
Goodput (Kbps)
800
700
600
500
400
300
200
100
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

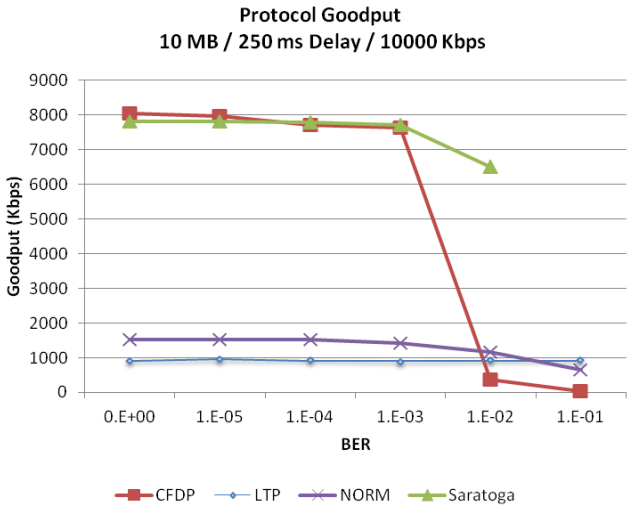

Protocol Goodput
1 MB / 250 ms Delay / 10000 Kbps
Goodput (Kbps)
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

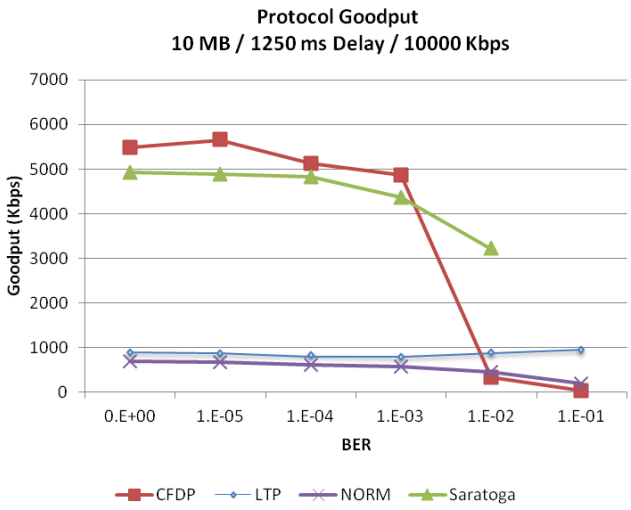

Protocol Goodput
100 KB / 1250 ms Delay / 10000 Kbps
Goodput (Kbps)
180
160
140
120
100
80
60
40
20
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga


Protocol Goodput
1 MB / 1250 ms Delay / 10000 Kbps
Goodput (Kbps)
1600
1400
1200
1000
800
600
400
200
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga


Protocol Goodput
1 MB / 25 ms Delay / 10000 Kbps
Goodput (Kbps)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga


Protocol Goodput
10 MB / 25 ms Delay / 10000 Kbps
Goodput (Kbps)
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga


Protocol Goodput
10 MB / 250 ms Delay / 10000 Kbps
Goodput (Kbps)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga


Protocol Goodput
10 MB / 1250 ms Delay / 10000 Kbps
Goodput (Kbps)
7000
6000
5000
4000
3000
2000
1000
0
0.E+00
1.E-05
1.E-04
1.E-03
1.E-02
1.E-01
BER
CFDP
LTP
NORM
Saratoga

References

1. V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss, "Delay-Tolerant Networking Architecture," IETF Request for Comments RFC 4838, Apr. 2007, . Available: http://tools.ietf.org/html/rfc4838
2. K. Scott, S. Burleigh, "Bundle Protocol Specification," IETF Request for Comments RFC 5050, Nov. 2007, Available: http://tools.ietf.org/html/rfc5050
3. Consultative Committee for Space Data Systems (CCSDS) File Delivery Protocol (CFDP). Blue Book. Issue 4. Jan. 2007.
4. S. Burleigh, M. Ramadas and S. Farrell, "Licklider Transmission Protocol – Motivation," IETF Request for Comments RFC 5325, Sep. 2008, Available: http://www.ietf.org/rfc/rfc5325.txt
5. S. Burleigh, M. Ramadas and S. Farrell, "Licklider Transmission Protocol – Specification," IETF Request for Comments RFC 5326, Sep. 2008, Available: http://www.ietf.org/rfc/rfc5326.txt
6. B. Adamson, C. Bormann, M. Handley, J. Macker, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol," IETF Request for Comments RFC 5740, Nov. 2009, Available: http://www.ietf.org/rfc/rfc5740.txt
7. L. Wood, W. Eddy, C. Smith, W. Ivancic, and C. Jackson, "Saratoga: A Scalable File Transfer Protocol," IETF Internet draft, work in progress, Sep. 2011.
8. NASA CTS Channel Emulator. Available: http://channel-emulator.grc.nasa.gov/
9. NASA CFDP Engine version 3.1a1, Greenbelt, Maryland: NASA Goddard Space Flight Center.
10. S. Farrell, LTPlib. Available: http://dtn.dsg.cs.tcd.ie/sft/ltplib/
11. Naval Research Laboratory, NORM. Available: http://cs.itd.nrl.navy.mil/work/norm/
12. Saratoga version 0, Brook Park, Ohio: NASA Glenn Research Center
13. M. Ramadas, LTP Reference Implementation. Available: http://irg.cs.ohiou.edu/ocp/ltp.html
14. ION Working Group, Interplanetary Overlay Network. Available: http://sourceforge.net/projects/ion-dtn/
15. DTN2 Reference Implementation. Available: http://www.dtnrg.org/wiki/Code
16. L. Wood, J. McKim, W. Eddy, W. Ivancic, and C. Jackson, "Using Saratoga with a Bundle Agent as a Convergence Layer for Delay-Tolerant Networking," IETF Internet draft, work in progress, May 2011.