STUDY OF A PROTOCOL AND A PRIORITY PARADIGM FOR DEEP SPACE DATA COMMUNICATION

A dissertation presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Doctor of Philosophy

Manikantam Ramadas
June 2007
This dissertation titled

STUDY OF A PROTOCOL AND A PRIORITY PARADIGM FOR DEEP SPACE DATA COMMUNICATION

by

MANIKANTAN RAMADAS

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Shawn D. Ostermann
Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, College of Engineering and Technology
Communicating data in deep-space entails the following constraints: large signal propagation delays - on the order of minutes or hours, coupled with the disruptions due to planetary orbital dynamics; high channel error characteristics; scarce and asymmetric data bandwidth availability, etc. Communication protocols developed for the terrestrial Internet perform poorly, or not at all, in this environment. We have co-designed a protocol, namely the Licklider Transmission Protocol (LTP), for reliable data communications in deep-space. We compare its performance to that of the well-known terrestrial Internet protocol - TCP - for various communication channel characteristics.

Deep-space missions often generate much more data for communication to Earth than what is allowed by the channel data rate. Therefore, it becomes important to decide which data to send, and when; this is done largely as a manual process at present. To help address this problem, we propose a two-dimensional priority paradigm for applications, aimed at optimizing the overall data communication performance. The two dimensions are: Intrinsic Value, a measure of how innately valuable the application data is, and Immediacy, a measure of how urgently a unit of application data needs
to be communicated. We integrate this priority paradigm with LTP, and study candidate Forward Error Correction (FEC) mechanisms for implementing the paradigm such as Convolutional codes, Reed-Solomon codes, and Digital Fountain codes, for various channel characteristics. Finally, we recommend appropriate FEC mechanisms for the priority requirements of applications under different channel characteristics to optimize the volume and value of the data received.

Approved: 

Shawn D. Ostermann
Associate Professor of Electrical Engineering and Computer Science
Acknowledgments

I acknowledge God’s infinite grace for the love and enthusiasm to do this work. I thank my parents, family members, and friends for all their love and affection, with special thanks to my room-mate and good friend Narendran for all the wonderful music.

Thanks are due to my advisor Dr. Shawn Ostermann for making me consider doing my PhD. Without him as a catalyst, this chemical reaction would not have even started. I am grateful to him for keeping me financially secure via research assistantships through the period of this study, for helping me gain more exposure in the domain by letting me attend many conferences, and most of all for encouraging in me a scientific temper and a cool spirit.

Thanks are due to my second advisor Dr. Hans Kruse for his words of encouragement, and for enlightening the path forward when it seemed headed for a dead-end.

Thanks to Scott Burleigh of NASA/JPL, for welcoming me into this research domain, for his patient listening, thoughtful comments and insights, and for his words of encouragement and affection.

Thanks to the CETS team members for their help, with special thanks to Kavitha Chayanam and Josh Blanton for their help in setting up the research test-bed.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>5</td>
</tr>
<tr>
<td>List of Figures</td>
<td>9</td>
</tr>
<tr>
<td>List of Tables</td>
<td>11</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>2 Protocol Study</td>
<td>22</td>
</tr>
<tr>
<td>2.1 TCP Overview</td>
<td>23</td>
</tr>
<tr>
<td>2.2 TCP for Space and Error-prone Channels</td>
<td>29</td>
</tr>
<tr>
<td>2.3 LTP</td>
<td>36</td>
</tr>
<tr>
<td>2.3.1 Basic Operation</td>
<td>38</td>
</tr>
<tr>
<td>2.3.2 Other Features</td>
<td>42</td>
</tr>
<tr>
<td>2.4 Testbed Setup</td>
<td>46</td>
</tr>
<tr>
<td>2.5 Experiments</td>
<td>51</td>
</tr>
<tr>
<td>2.5.1 Geo-synchronous Orbit Distances</td>
<td>51</td>
</tr>
<tr>
<td>2.5.2 Lunar Distances</td>
<td>59</td>
</tr>
<tr>
<td>2.5.3 Sub Geo-synchronous Orbit Distances</td>
<td>65</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>69</td>
</tr>
<tr>
<td>3 Priority Paradigm Study</td>
<td>72</td>
</tr>
<tr>
<td>3.1 FEC Schemes</td>
<td>74</td>
</tr>
</tbody>
</table>
A.1.2 Configuring the Error-emulator .......... 157

A.2 Priority Paradigm Study ................. 159
## List of Figures

1.1 A View of the Cassini Spacecraft [Image Courtesy : NASA/JPL Caltech] 17

2.1 LTP Operation - No Channel Errors ............................. 39

2.2 LTP Operation - With Channel Errors ............................. 41

2.3 Experiment Setup .................................................. 47

2.4 Protocol Stacks .................................................... 50

2.5 Geo-synchronous Orbit, Ideal Channel, TCP & LTP Throughput ........................ 52

2.6 Geo-synchronous Orbit, Ideal Channel, TCP & LTP Duration ........................ 53

2.7 Geo-synchronous Orbit, Noisy Channel, TCP Throughput ........................ 56

2.8 Geo-synchronous Orbit, Noisy Channel, LTP Throughput ........................ 58

2.9 Lunar Distance, Ideal Channel, TCP & LTP Throughput ........................ 60

2.10 Lunar Distance, Ideal Channel, TCP & LTP Duration ........................ 60

2.11 Lunar Distances, Noisy Channel, TCP Throughput ........................ 63

2.12 Lunar Distances, Noisy Channel, LTP Throughput ........................ 64

2.13 Sub Geo-synchronous Orbit, Ideal Channel, TCP & LTP Throughput 66

2.14 Sub Geo-synchronous Orbit, Ideal Channel, TCP & LTP Duration 66

2.15 Sub Geo-synchronous Orbit, Noisy Channel, TCP Throughput ........................ 67

2.16 Sub Geo-synchronous Orbit, Noisy Channel, LTP Throughput ........................ 68

2.17 TCP vs LTP Protocol Boundary: Distance; 1 Mbps Channel ........................ 71

3.1 LT Codes: Operation ................................................ 88
3.2 LTP-PP Protocol Stack ........................................... 92
3.3 An LTP-PP Packet .................................................. 95
3.4 PP Mechanism Performance: Uniform BER 5x10^{-3} .............. 108
3.5 PP Mechanism Overhead: Uniform BER 5x10^{-3} .............. 108
3.6 PP Mechanism Performance: Uniform BER 0.01 .................. 113
3.7 PP Mechanism Overhead: Uniform BER 0.01 .................. 113
3.8 PP Mechanism Performance: Uniform BER 1x10^{-3} .............. 115
3.9 PP Mechanism Overhead: Uniform BER 1x10^{-3} .............. 115
3.10 PP Mechanism Performance: Uniform BER 1x10^{-4} .......... 117
3.11 PP Mechanism Overhead: Uniform BER 1x10^{-4} .......... 118
3.12 PP Mechanism Performance: Uniform BER 1x10^{-5} .............. 121
3.13 PP Mechanism Overhead: Uniform BER 1x10^{-5} .............. 122
3.14 PP Mechanism Performance: Uniform BER 1x10^{-6} .............. 123
3.15 PP Mechanism Overhead: Uniform BER 1x10^{-6} .............. 124
3.16 Gilbert-Elliot Channel Model ..................................... 126
3.17 PP Mechanism Performance, Burst Errors: φ_{good} = 10^{-5}; φ_{bad} = 10^{-3} 128
3.18 PP Mechanism Overhead, Burst Errors: φ_{good} = 10^{-5}; φ_{bad} = 10^{-3} 129
3.19 PP Mechanism Performance, Burst Errors: φ_{good} = 10^{-6}; φ_{bad} = 10^{-4} 130
3.20 PP Mechanism Overhead, Burst Errors: φ_{good} = 10^{-6}; φ_{bad} = 10^{-4} 130
4.1 Lunar Communication System: Network of Internetworks ........ 142
List of Tables

1.1 Cassini Science Instrument Data Rates . . . . . . . . . . . . . . . . . . 18
2.1 Testbed Machine Information . . . . . . . . . . . . . . . . . . . . . . 48
3.1 Bandwidth Consumption (in bytes) by TC, Uniform BER 5*10^{-3} . . 136
3.2 Bandwidth Consumption (in bytes) by TC, Uniform BER 10^{-5} . . . 138
**Chapter 1**

**Introduction**

Our dissertation work presented here concerns with the design, study, and optimization of data communication protocols for operating in the Space and Deep-Space communication environments. We loosely define the “Space” environment to refer to the satellite communications environment in near-Earth distances, such as those involved with Low Earth Orbit (LEO), and Geo-Synchronous Orbit (GEO) satellites, where signal propagation latencies are at most in the order of seconds, and Deep-Space to refer to communicating in Inter-planetary distances such as those involved with missions to Mars, and Saturn, where communication latencies are in the order of minutes and hours. Communication protocols designed for operating in our terrestrial Internet (the TCP/IP suite of protocols) perform poorly, or not at all, in the Space and Deep-Space environment. This is due to the following constraints that are innate to this environment:

- **High Signal Propagation Delays**

  The enormous distances involved between the communicating entities, and the relativistic constraint restricting signal transmissions to the speed of light, cause high signal propagation delays. For example, Saturn is at least a billion kilometres from Earth (approximately), and one-way signal propagation delays for
the current Cassini mission to Saturn are in the range of 1 hour and 8 minutes to 1 hour and 24 minutes [Spi97].

- High Data Corruption Rates

Radio signal strength reduces proportionate to the square of the distance between the communicating entities; thus the large distances involved in the interplanetary communication environment cause the signal to be received at extremely low strengths. Further, this reception is subject to high probability of bit-errors in the channel due to random thermal noise, burst errors due to solar flares, etc. [Hem97].

- Disruption Events

Since communicating entities in deep-space almost always are in motion relative to one another, the communication channel between them is prone to disruption. For example, a rover on the surface of Saturn’s moon Titan communicating directly with Earth could experience disruption due to the rotation of Titan on its own axis - when it goes to the night side of Titan, due to the revolution of Titan around Saturn - when it passes under Saturn’s shadow, and when other moons, planets, or the Sun itself block the line of sight[FBMT97].

Moreover, communicating with an entity in deep-space requires expensive, specialized equipment. The Deep-Space Network system of antennas [DSN] used
for current missions tends to be tightly scheduled for usage by multiple missions and hence the communication window is likely to be restricted from Earth too.

• **Meager, Asymmetric Bandwidth**

The bandwidth capacities are meager, and asymmetric in the deep-space environment. The uplink channel (Earth to Destination) tends to have much lower bandwidth than the downlink channel (Destination to Earth), typically by one or two orders of magnitude. This is normally by design, due to signal power considerations and the fact that the uplink channel is expected to carry mostly light-weight command traffic while the more interesting data collection, analysis, reports etc. are expected in the downlink channel. Therefore the uplink channel tends to be designed as a reliable, lightweight communication channel. For example, the Cassini spacecraft has an uplink bandwidth of 1 Kbps while the maximum downlink bandwidth is 166 Kbps [Spi97]

Communication protocols used in the Internet for reliable data transfer such as TCP, were designed with very different assumptions on the communication channel, and therefore exhibit characteristics that make them unsuitable for the deep-space environment. TCP, the common transport-layer protocol used for reliable data delivery in the Internet:

1. has innate congestion control mechanisms to protect against congestion collapse of the Internet [Jac88, APS99];
2. assumes every packet loss to be an indicator of congestion in the network (unless
the Explicit Congestion Notification mechanism [RFB01] is in use);

3. has handshaking procedures for connection setup and tear-down - which means
that it takes a few round-trip times before the actual data transfer begins.

These assumptions make it unsuitable as a deep-space communication protocol. Other
terrestrial Internet communication protocols have similar, or related problems when
operated over deep-space datalinks because of the channel assumptions they make.

Thus, realizing the need for a deep-space communication protocol for reliable data
delivery, we have co-designed the Licklider Transmission Protocol (LTP) ¹; LTP is
designed to operate optimally over single hop long-haul deep-space datalinks (LTP
also stands for the pseudonym Long-haul Transmission Protocol). LTP:

1. assumes operation over a point-to-point datalink;

2. has no connection establishment / tear-down procedures that add chattiness;

3. perceives packet loss as only an indication of corruption, and has no congestion-
control mechanisms;

4. guarantees reliable data delivery by doing Automated Repeat Requests (ARQ)
by following the Selective Acknowledgment model;

5. has variable length protocol headers for better bit economy.

¹Named in honor of ARPA pioneer JCR Licklider
From our discussion above, we see that TCP would clearly be unsuitable in the deep-space context. We would similarly expect TCP to perform poorly in the near-space context; however, the performance of TCP versus LTP for near-space under different channel characteristics is not fully clear. We describe TCP and LTP in more detail, and explore their relative performance in the near-space context in Chapter 2.

Next, we present our work concerning the development and study of a priority paradigm for deep-space applications. We observe that spacecrafts on deep-space missions carry a host of scientific instruments for performing various scientific endeavours. The net spacecraft downlink bandwidth tends to be significantly lesser than the peak data generation rate of all the instruments put together, creating a communication bottle-neck at the sender. Although on-board data storage is generally available to mitigate this, it itself could become a resource prone to contention, especially when working under very short communication windows. We provide data from the Cassini spacecraft currently orbiting Saturn as a concrete example.

The Cassini-Huygens spacecraft illustrated in Figure 1.1 carries 18 science instruments in total - 12 on the Cassini orbiter and 6 on the Huygens probe to Titan. Some of the science instruments on board Cassini and their estimated data rates [Cas] are shown in Table 1.1.

Cassini operates in the X-band (7-12 GHz) radio channel with a downlink data bandwidth (while using its High-Gain Antenna) in the range of 14.2 to 165.9 Kbps, significantly lesser than the total bandwidth required to support all the instruments
Figure 1.1: A View of the Cassini Spacecraft [Image Courtesy: NASA/JPL Caltech]
<table>
<thead>
<tr>
<th>Science Instrument</th>
<th>Estimated Data Rate Approx. (Kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassini Plasma Spectrometer (CAPS)</td>
<td>8.0</td>
</tr>
<tr>
<td>Cosmic Dust Analyzer (CDA)</td>
<td>0.5</td>
</tr>
<tr>
<td>Composite Infra-Red Spectrometer (CIRS)</td>
<td>6.0</td>
</tr>
<tr>
<td>Ion and Neutral Mass Spectrometer (INMS)</td>
<td>1.5</td>
</tr>
<tr>
<td>Imaging Science Subsystem (ISS)</td>
<td>365.6</td>
</tr>
<tr>
<td>Magnetometer (MAG)</td>
<td>3.6</td>
</tr>
<tr>
<td>Magnetospheric Imaging Subsystem (MIMI)</td>
<td>7.0</td>
</tr>
<tr>
<td>Cassini RADAR</td>
<td>364.8</td>
</tr>
<tr>
<td>Radio and Plasma Wave Science Instrument (RPWS)</td>
<td>0.9</td>
</tr>
<tr>
<td>Ultra-Violet Imaging Subsystem (UVIS)</td>
<td>32.1</td>
</tr>
<tr>
<td>Visual and Infra-red Mapping Spectrometer (VIMS)</td>
<td>182.8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>972.8</strong></td>
</tr>
<tr>
<td><strong>Maximum Spacecraft Downlink Bandwidth</strong></td>
<td><strong>165.9</strong></td>
</tr>
</tbody>
</table>

Table 1.1: Cassini Science Instrument Data Rates

Data simultaneously. It is conceivable that based on the science experimental results, data from the instruments could be of varying degrees of interest and importance during various times in the mission. For example, the Cosmic Dust Analyzer might have an interesting experimental observation that may be of more immediate critical value compared to the last image captured by the wide-angle camera of the Imaging Science Subsystem - which presumably may be stored and transmitted in the next communication window.

Similarly, the Huygens probe carried six science instruments, and its science mission life after being detached from Cassini into the atmosphere of Titan, was only 3 hours (the extreme physical environment within the atmosphere of Titan was not expected to be kind on the communication equipment and batteries). In this short
communication window, science data was beamed up to the orbiting Cassini spacecraft on S-band radio at a maximum data rate of 16 Kbps. It is clear that such a short communication window must be optimally used, and preferentially by the on-board applications having the most critical, interesting data.

In addition to the data generated by the science instruments on board a spacecraft, regular housekeeping and telemetry data pertaining to the physical operation and well-being of the spacecraft also needs to be transmitted periodically. It is conceivable that such data is typically not extremely critical, and that healthy operation could be perceived even under the transmission environment where a few telemetry data updates are lost, from the subsequently received updates.

The paper [CDF+03] presents a proposal for applying an Adaptive Resource Management framework for optimal utilization of resources on the Swift Gamma-Ray Burst Alert Telescope spacecraft. The three science-instruments on-board: the Burst Alert Telescope, the X-Ray Telescope, and the Ultra-Violet/Optical Telescope; the on-board storage; and other spacecraft house-keeping resources could be managed by the Adaptive Resource Management Middleware (ARM) system. The ARM system represents the requirements on the resources set by the science-investigators as service attributes. It then attempts to optimize the overall system utility based on the relative importance of the tasks requiring the resources. Here, we propose a priority paradigm for optimizing the volume and value of the data transferred on the communication
channel. Our proposal can be visualized as a simple Resource Management approach, that is concerned with optimizing the single resource: channel bandwidth.

The work cited in [CWSA05], discusses a mechanism for prioritizing images returned by the NASA Mars missions and indicates the need for application level data prioritization. Our priority paradigm however, is designed to be generic, suitable to all deep-space application data.

It provides two dimensions to applications: Intrinsic Value, and Immediacy.

- **Intrinsic Value**: It is a measure of how valuable a unit of application data (called a ‘job’) is, relative to the rest of the applications. For example, the picture of a microbe found on the surface of Titan may be much more valuable to the mission compared to regular house-keeping telemetry data, and thus may need appropriate care in delivery.

- **Immediacy**: Immediacy is a notion of how urgently a job needs to be received compared to other jobs. Some jobs have tight time constraints on them, and they are valuable only if received within the constraints. For example, a message from an on-board science instrument entering a critical state such as “Instrument too hot”, “Low battery condition”, or commands sent from Earth to a rover such as “Stop! Don’t go down that crater”, may need to be received ahead of all other experimental results / commands. Some classes of streaming telemetry data could also have similar immediacy requirements; if the job is not
received within the time constraints, it is probably not useful to receive it at all.

We then study mechanisms to guarantee the two-dimensional Priority Paradigm (PP) policy values chosen by a job. We study the efficacy of Forward Error Correction (FEC) schemes such as Convolutional Codes, Reed-Solomon Codes, Raptor Codes - a member of the family of Digital Fountain Codes, and few combinations of them as mechanisms to guarantee the PP requirements of a job, for various channel error characteristics. Our results on the Priority Paradigm are presented in Chapter 3.

In Chapter 4, we conclude with our recommendations for the appropriate FEC schemes to use for various channel characteristics to achieve our goal of optimizing both the volume and value of the data transmitted on the deep-space communication channel.
CHAPTER 2

Protocol Study

In this chapter, we present our study of TCP and LTP for near-space environments, operating under various channel settings for delay and error. We outlined the different channel assumptions made by the designers of TCP and our assumptions in designing LTP in the previous chapter; therefore, we would naturally expect TCP to perform poorer than LTP in our tests, in general. However, it is not fully clear if there is a realm of operation in near-space where the performance of TCP may match LTP; if there is such a realm, it may not be necessary to incur the overhead of implementing and installing a new protocol - LTP, while operating within it. We conjecture that such a realm might exist, and that it may be dependent on signal propagation delay and channel error characteristics.

The goal of this study is two-fold:

- Study the performance of LTP in the near-space environment to make sure that LTP performance is as expected.

- Compare the performance of LTP with TCP under identical conditions, and determine the boundary of the realm within which TCP performs reasonably well, and beyond which the use of a protocol specifically designed for space - such as LTP - becomes advisable.
We first give an overview of TCP, describing its basic operation, and outline related work in the literature on optimizing TCP for space communications. This is followed by a similar overview of LTP, describing its features and operational procedure. Next, we describe our setup of testbed machines where our experiments were conducted, followed by the experimental results and conclusions.

2.1 TCP Overview

TCP - Transmission Control Protocol is the standard end-to-end protocol used in the Internet for reliable, in-order delivery of data between a pair of hosts. It was originally specified in RFC 793 [Pos81]. Early TCP implementations based on this specification led to the congestion collapse event on the Internet in October 1986. After analyzing the causes for congestion collapse, Van Jacobson proposed the “conservation of packets principle” [Jac88] as a remedy. This principle states that when a TCP connection reaches its steady-state, a packet should be added to the network only when another packet (sent earlier) is known to have left the network. To implement this principle, he proposed the slow-start and congestion-avoidance algorithms [Jac88]. These two algorithms, along with the Fast Retransmit algorithm were implemented in 4.3 BSD Unix Tahoe TCP in 1988; the Fast Recovery algorithm was added in 4.3 BSD Unix Reno TCP in 1990 [FF96]. All four algorithms: Slow-Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery are now part of the IETF TCP standard, per RFC 2581 [APS99].
Before describing the above four algorithms, we describe the following key TCP variables maintained by TCP implementations:

- **rwnd** (Receive Window) is the amount of buffer-space available at the TCP receiver to store outstanding, unacknowledged data. This governs the maximum outstanding data a TCP sender may transmit, and is the transmission rate limit imposed by the TCP receiver.

- **cwnd** (Congestion Window) is the amount of unacknowledged outstanding data a TCP sender is allowed to transmit. This is transmission rate restriction imposed by the sender itself to protect against congestion.

- **ssthresh** (Slow Start Threshold) determines whether the sending TCP is in the slow-start phase or the congestion-avoidance phase.

The initial value of ssthresh may be arbitrarily high; it might be set to rwnd, for example. The cwnd is set to a low initial value: RFC 2581 permits initial cwnd value of at most 2 MSS (Maximum Segment Size); however, an update to this RFC - RFC 3390 [AFP02] permits an initial congestion window of 4 MSS, with a maximum value of 4380 bytes. The sender can transmit the minimum\{cwnd, rwnd\} outstanding bytes of data every Round Trip Time (RTT).

A TCP receiver normally sends an acknowledgment segment (ACK) upon reception of a data segment; the ACK identifies the sequence number of the next byte of data expected at the receiver. The hosts requirements RFC - RFC 1122 [Bra89]
relaxes this requirement and specifies that the receiver should send delayed ACKs, by waiting for the second data-segment and acknowledging the two data segments together, but waiting at most 0.5 seconds for the second data-segment to arrive. Note that TCP ACKs are cumulative; i.e., an ACK for the data byte with sequence number 100 mentions that all data until sequence number 99 are received, and sequence number 100 is the next expected sequence number.

Now, the purpose of the slow-start algorithm is to let TCP probe the channel for the appropriate bandwidth to use. As long as the cwnd is less than ssthresh, the TCP sender is in the slow-start phase, and cwnd is incremented by 1 segment for every received ACK; this approximately doubles both the cwnd and the transmission rate (if permitted by rwnd), for every RTT. The TCP sender stays in slow-start unless a congestion event is noticed, whence it enters congestion avoidance; TCP treats packet-loss as the sole indication of congestion in the network, and reacts differently depending on how exactly packet-loss is detected:

- **If three duplicate ACKs (dupacks) are received:**

  For example, if the TCP sender transmitted 1 byte segments with sequence numbers 10, 11, 12, 13, 14, and segment 11 alone is dropped in the network, the receiver would generate the first ACK for segment 11 upon reception of segment 10 and three dupacks upon reception of segments 12, 13, and 14, and now the TCP sender infers that segment 11 was dropped in the network. Note that the threshold of 3 dupacks is to be robust to reception of dupacks due to packet
re-ordering in the network. The TCP sender now follows the Fast Retransmit algorithm and retransmits segment 11, and then enters the Fast Recovery phase.

In fast recovery, the following steps are followed:

- ssthresh is set to \( \text{minimum}\left\{ \frac{\text{FlightSize}}{2}, 2 \times MSS \right\} \) (Flight size is the amount of outstanding unacknowledged data in transit).
- cwnd is set to \( ssthresh + 3 \) (adding 3 for the 3 segments that left the network).
- cwnd is incremented by 1 for each arriving dupack thereafter, and a new segment is transmitted if allowed by rwnd. Note that dupacks for approximately half the earlier Flight Size must be received before cwnd reaches its original value again and any data can be transmitted; this lets half the data in Flight to drain from the network before any further transmission.

When the cumulative ACK covering the retransmitted segment is received, the TCP sender exits Fast Recovery \(^1\), sets cwnd to the ssthresh value set upon entering Fast Recovery, and continues operation in congestion-avoidance mode, incrementing the transmission by at most one MSS per RTT. If the segment retransmitted in Fast Retransmit is lost again, it is taken as an indication of severe congestion in the network; the Retransmission Timer times out eventually.

---

\(^1\)This is as per RFC 2581 [APS99]. Future recommendations, such as RFC 3517 [BAFW03] exit Recovery only when the entire Flight size of data (at the time Fast Retransmit procedure is invoked) is cumulatively acknowledged.
ally, and the slow-start procedure is invoked again to determine the appropriate bandwidth to use in the current state of the network.

• *If the Retransmission Timer fires:*

If there was severe packet loss in the forward path, say for example that the entire window of data in flight is lost, or many ACKs are lost in the reverse path, 3 dupacks may not be received, but the TCP Retransmission Timer may timeout causing an RTO (Retransmission Time Out) event. If packet loss is inferred via a time-out, TCP assumes that the network must be severely congested, sets ssthresh to \( \frac{cwnd}{2} \), and begins data recovery by invoking slow-start again to determine the appropriate bandwidth to use afresh. The procedure for computing the TCP retransmission timer is specified in RFC 2988 [PA00]; the RTO is recommended to be at least 1 second to be conservative, avoiding spurious RTO events, because an RTO event has severe impact on TCP performance due to the RTTs spent in slow-start and the beginning of congestion-avoidance at half the value of existing cwnd.

If multiple segments are lost in the window of data, Fast Retransmit / Fast Recovery procedures do not recover all such missing segments, and often an RTO event is incurred and recovery continues via Slow-Start. To address this, and make TCP loss recovery more informed in general, Selective Acknowledgments (SACKs) were introduced; they are specified in RFC 2018 [MMFR96]. With SACK, a TCP receiver conveys the sequence number range of data received in its out-of-sequence queue.
Two strategies for modifying the Fast Recovery phase with the additional information received via SACKs have been proposed; they are both functionally equivalent, and are allowed by RFC 2581 [APS99]:

- The first strategy, informally called the pipe algorithm was proposed in [FF96] and has hence been standardized by RFC 3517 [BAFW03]. The pipe algorithm, in addition to keeping track of the amount of data in flight after receiving 3 dupacks, performs the Fast Retransmit and enters the Fast Recovery procedure as always. The cwnd and ssthresh variables are set to half the Flight size, and the pipe variable keeps track of the amount of data in Flight. The pipe variable is incremented for the segment retransmitted in Fast Retransmit, and decremented appropriately upon arrival of each new SACK block by decrementing the amount of new data SACK-ed. When \((cwnd - pipe) \geq 1MSS\), a new segment is eligible for retransmission; at this point another segment inferred lost in Flight based on the SACK information received, is retransmitted. See RFC 3517 [BAFW03] for more details.

- The second strategy, called the FACK (Forward Acknowledgment) algorithm is specified in [MM96a, MM96b]. It works by keeping track of the forward most data (highest data sequence number) SACK-ed; this value is stored in the variable \(snd.fack\), and is used to make retransmission decisions during Fast Recovery. Besides invoking Fast Retransmit/Fast Recovery procedures based on 3 dupacks, the FACK algorithm also invokes Fast Recovery if the \((snd.fack - \)
snd.una > 3MSS where snd.una is the standard TCP variable that tracks the last sequence number cumulative acknowledged (i.e., recovery is also invoked when the sender knows that more than 3 segments beyond snd.una are known to have left the network via SACKs). FACK algorithm also introduces Rate-halving which causes the sender to retransmit a packet for every two ACKs received during Fast recovery. See [MM96a, MM96b] for more details.

This concludes our discussion of the standard features of TCP. Addition of SACK to TCP makes the recovery of multiple dropped segments more efficient in the Fast Recovery phase. However, Fast Recovery still reduces sending-rate by half and expects the sender to continue operation in the congestion avoidance mode. This is ideal for wired, congestion-prone networks where packet corruption is a rare occurrence; in domains where packet corruption is more common, as in wireless channels and space channels, TCP performs degrades rapidly with increasing packet error rate probabilities. This is a well-known and well-studied problem for TCP; In the following section we describe related research work for near-space communications.

### 2.2 TCP for Space and Error-prone Channels

If packet loss is not a reliable indicator of congestion then some other form of congestion notification must be provided so that TCP can distinguish packet losses
due to congestion from those due to channel errors, and invoke congestion response only in the former case.

The RFC 2760 [ADG+00] describes research directions for optimizing TCP performance in Satellite communication channels. RFC 2760 mentions the use of an ICMP “corruption experienced” message. Routers experiencing packet corruption inform the TCP receivers with the ICMP message; the TCP receivers then convey this message in their TCP ACKs; the TCP senders are then expected to use this information in their congestion avoidance procedures. RFC 2760 also suggests link-layer usage of Forward Error Correction (FEC) in the presence of channel errors. Another approach to gather information from the network about corruption based losses is suggested in the paper on CETEN (Cumulative Explicit Transport Error Notification) [EOA04]. This paper starts with the fundamental premise that packet loss rate $p$ is the sum of the loss rate due to congestion $c$ and the loss rate due to corruption $e$; since the TCP sender directly measures $p$ via packet-losses, knowledge of $e$ would help measure $c$; thereafter, congestion response could be based upon $c$ alone. This paper proposes the following mechanism to gather $e$ from the network:

- An IP option carries a new Time To Live field $TTL'$, which must be decremented by all CETEN capable routers in the path.

- The IP option also carries a cumulative packet erasure probability, initialized to 1 by the sender. Each CETEN capable router multiplies the average packet
erasure rate observed in its recent history to this field, decrements TTL’, and then forwards it along.

- The receiving TCP echoes the cumulative error-rate back to the TCP sender in TCP ACKs.

The paper further recommends congestion response procedures for the TCP as a function of the cumulative $e$ value, and $p$.

The Explicit Congestion Notification (ECN) procedure, on the other hand, is a mechanism for the network to inform incidence of congestion. ECN, standardized in RFC 3168 [RFB01], provides a mechanism for the network to inform the TCP end-points about congestion explicitly, avoiding packet-loss. Implementation of this mechanism requires ECN to be implemented in all the routers in the network traversed by the TCP connection, and requires the routers to implement some form of Active Queue Management (AQM). Traditionally, routers have had drop-tail queue, i.e., the router has a queue of finite buffers to store packets received from one network temporarily before forwarding onto the other network; if the arrival rate of packets is higher than the departure rate, the queue builds up and eventually becomes full; a router with drop-tail queuing policy would drop all future packet arrivals when the queue is full. There are many proposed schemes and variants for AQM; Random Early Detection (RED) [FJ93] is a prominent scheme. A router implementing RED has two thresholds: a low-water-mark $L$ and a high-water-mark $H$ where $0 < L \geq H \geq N$ where $N$ is the maximum router queue size. If the average queue size $A$ is below $L$
the router works normally and forwards the incoming packet; if $A$ is greater than $H$, the router drops the incoming packet; if $L \geq A \geq H$, the router probabilistically marks the packet with a “Congestion Experienced” notification, with greater and greater probabilities as $A$ gets closer to $H$, and then forwards the packet. The ECN specification describes the mechanism by which this marking is done in the IP layer; a TCP receiver implementing ECN is required to echo back this ECN message via TCP so that the sending TCP receives the message and invokes the appropriate congestion avoidance procedure. If ECN and AQM were implemented in all the routers of the networks, a TCP connection passes through, a TCP sender may depend on ECN for congestion notification and perceive packet-drops as due to channel errors. All routers on the Internet may not implement ECN as yet; so relying on ECN alone for congestion notification can be done only on private networks at present. The paper on LT-TCP (Loss-Tolerant TCP) [TSKR05] makes this assumption, and describes a mechanism for using FEC schemes adaptively upon detecting packet-losses to mitigate channel errors, and thus help TCP perform better on error-prone channels.

RFC 2760 also suggests using T/TCP (TCP Extensions for Transactions) [Bra94] for short messages, wherein application data is carried along in the TCP SYN segment itself to save the 1.5 RTTs spent in the TCP 3-way handshake. However, there are security issues associated with using T/TCP on the Internet, as noted by RFC 2760.

Another related experimental research work in progress is the Quick-Start algorithm for TCP [FAJS06]. This work presents a procedure by which the TCP sender
negotiates the appropriate sending-rate with the goal of by-passing the multiple RTTs spent in the slow-start phase, which are expensive in long delay environments. This procedure again needs support from both TCP and IP layers. The TCP sender transmits the sending-rate it wishes to send data at, in an IP option; another IP option field Quick-Start TTL’ is also added; all routers that understand Quick-Start approve or modify the sending rate requested, decrement the Quick-Start TTL’, and forward the packet along; the TCP receiver transmits the approved sending-rate and the difference between the standard IP TTL and Quick-Start TTL’, via TCP ACKs; the TCP sender receives this ACK, and verifies if all routers along the path have approved the sending rate (the TTL, TTL’ difference must be zero), and if so, uses the product of the sending-rate and RTT to seed its cwnd. This mechanism, however, may fail to function correctly in the presence of middle-boxes, IP tunnels, etc., and therefore this research work (work in progress) is recommended only for controlled private networks.

Finally, we briefly describe the SCPS (Space Communications Protocol Standards) protocol suite [SCP]. The SCPS protocol suite is standardized by the International Space Standards organization CCSDS - Consultative Committee for Space Data Systems [CCS]. In general, the SCPS protocol standards have followed the approach of adapting established standards from other domains - the Internet in particular, to operate optimally in Space communication channels. The SCPS protocols are designed for near-space communications involving satellite channels such as Low Earth Orbit
(LEO) and Geo-stationary Orbit (GEO) satellites, and could be applied to at most lunar distances. The suite comprises of four layered protocols [SCP]:

- **SCPS-FP** - SCPS File Protocol is based on the File Transfer Protocol (FTP) used in the Internet. It is used for uploading commands and retrieving telemetry data from spacecrafts.

- **SCPS-TP** - SCPS Transport Protocol is an adaptation of TCP for space. We describe it in more detail in the following paragraphs.

- **SCPS-SP** - SCPS Security Protocol is based on the IPSec protocol - used in the Internet for secure IP level communications, and other standard network-layer security protocols.

- **SCPS-NP** - SCPS Network Protocol is an adaptation of IP.

The SCPS-TP protocol specified in [SCP06], adds or modifies features from TCP; we cite some here:

- **MFX**: Multiple Forward Transmissions. A configurable parameter MFX governs the number of multiple forward transmissions made for each TCP data segment along with their original transmission, to be robust to channel erasures.

- **BETS**: Best-Effort Transport Service introduces a configurable parameter R2. If the number of retransmissions of a segment crosses this threshold, SCPS-TP gives up on the segment and adjusts the TCP sender-side variables as if the segment was cumulatively ACK-ed by the TCP receiver.
- Congestion Control: SCPS-TP lets the TCP connection peers negotiate whether to use Congestion Control at all, and if so, whether to use the TCP standard congestion control algorithms or the TCP Vegas congestion control algorithm [BOP94].

SCPS-TP options including the ones mentioned above are negotiated between TCP peers as part of the three-way handshake, via the mechanism outlined in [SCP06]. Thus SCPS-TP provides a space mission with a tunable transport layer configuration settings, and mission communication planners may set them according to their needs.

It may not always be feasible or optimal to tune the SCPS-TP settings (or standard TCP in general for that matter), on the end-hosts in an end-to-end path having a single space communication link; to address this, TCP Performance Enhancing Proxies (PEPs) are often deployed on the end-points of the space link. The PEPs on either end of the space link proxy the TCP connection between the end-hosts by terminating the connection at the space-link entrance and exit; a single end-to-end TCP connection is thus split into three connections: the first, between the TCP sender to the space-link entrance; the second, on the space link; and the third between the space-link exit and the TCP receiver. The PEPs strive to be transparent to the TCP sender and receiver; in such a proxied environment, SCPS-TP may be employed with no-congestion control mode on the space-link for example, with regular TCP connections on the first and third segments of the path. PEPs help improve
end-to-end performance, but break the end-to-end transport layer principle, making debugging and troubleshooting harder.

In summary, many mechanisms have been proposed, and studied to extend, or adapt TCP/IP for the near-space environment. However, none of these mechanisms address the innate chattiness of TCP (taking 1.5 RTTs to finish the hand-shakes, for example), which becomes very expensive in deep-space channels where one-way signal propagation delays could be in the order of minutes and hours. In the following section, we describe a reliable transmission protocol designed for deep-space channels - LTP.

2.3 LTP

Licklider Transmission Protocol (LTP) is designed as a reliable data delivery protocol for deep-space datalinks. It is a work in progress, specified in three Internet-Drafts describing: the motivation for LTP, the protocol specification, and optional protocol extensions\(^2\). In the protocol stack, it operates over the deep-space datalink layer (typically RF datalink), providing a reliable datalink abstraction to the upper layers. It offers retransmission-based reliability, performing ARQ (Automated Repeat reQuest) of lost data by following the selective- acknowledgment model. It makes the following basic assumptions in its design:

\(^2\)The LTP Motivation, Specification, and Extensions Internet-Drafts are work-in-progress items. The current versions of these documents [BRF07], [RBF07], and [FRB07] have reached general research community consensus; Following final IETF review procedures, they are set to be published as RFC standards.
The datalink is a point-to-point link, with segments arriving at the receiver in their transmission order, if at all. If segments arrive out-of-order, those segments missing in sequence are considered lost. Note that segment losses are assumed to be caused only by erasures due to errors in the channel.

Data flow is unidirectional, as is typical of deep-space communications today; data flows in the forward direction (\textit{sender} \(\Rightarrow\) \textit{receiver}), while the acknowledgment traffic flows in the reverse direction (\textit{receiver} \(\Rightarrow\) \textit{sender}); LTP sender is the LTP node initiating the thread of data communication (called a “session”), and its peer node is the LTP receiver. Hence, bidirectional data-flow entails starting two independent sessions.

When an LTP node has data to send to its peer, it can open a session unilaterally, and start sending segments to the receiver. The sender is required to choose a unique Session ID to identify the session, and identify the receiver LTP application by its Client Service ID in all its data segments. The Client Service ID demultiplexes the LTP application from other active applications at the receiver; its usage is analogous to the way port numbers identify TCP applications. In the course of an LTP session, different types of LTP segments may be transmitted; the segment types fall in the following five classes:

- **DS** Data Segment: carries original transmissions and retransmissions of application data.
• RS Report Segment: carries a reception report from the receiver, selectively acknowledging the ranges of data received within a specified scope.

• RA Report Acknowledgment: carries the acknowledgment sent from the sender upon receipt of an RS.

• Cx Cancel Segment: sent to initiate session cancellation procedures (from either sender or receiver).

• CAx Cancel Acknowledgment: sent as a response by the LTP peer upon reception of a Cx segment.

2.3.1 Basic Operation

The operation of LTP in the simple case, i.e., assuming no channel errors occur, is illustrated in Figure 2.1.

The LTP sender starts, and transmits Data Segments 0, 1, …, 9. DS 9 is marked as a Checkpoint segment (CP), and End-of-Block (EOB); CP indicates that a Reception Report is solicited from the receiver, and EOB indicates that the end of original data transmission is reached for this session. A CP segment is identified by a serial number; in this example, the sender chooses serial number 79. After transmitting the CP segment, the sender starts a timer for a certain interval within which the Reception Report is expected to arrive. If the CP timer expires and the Reception Report has not been received yet, it is assumed to be lost and the CP segment is
retransmitted. If the Reception Report is received within the timer interval, as in this example, the timer is turned off immediately.

In this example, the receiver issues a Reception Report: Report Segment (RS) with serial number 18 acknowledging the CP 79. This Report Segment (RS) acknowledges successful reception of all data within the scope of the Checkpoint (segments with block sequence numbers [0:10]). The receiver then starts a timer for a certain interval within which the sender is expected to transmit the corresponding Reception-Acknowledgment (RA) segment. If the RA segment is not received within the timer interval, the RS segment is assumed to be lost and is retransmitted. If the RA seg-
ment is received within the timer interval, as in this example, the timer is turned off immediately.

Since timers are started to guarantee their delivery, CP and RS segments are said to be transmitted \textit{reliably}.

In our example, since the sender knows that all data is successfully received, it closes the session after transmitting the RA segment. Similarly, when the receiver gets the RA, it closes the session.

Next, we describe LTP behaviour in the presence of channel errors, as illustrated in Figure 2.2.

Here again, the sender transmits data segments 0, 1, \ldots, 9, but segments 2 and 3 are lost due to errors in the channel. The reception report transmitted upon reception of CP 79 - RS 18, selectively acknowledges data reception, indicating that segments in range 0:2 (0 inclusive, 2 exclusive), and similarly in range 4:10 are received. The sender receives this RS, transmits the acknowledging RA, and retransmits the missing segments. The last retransmitted segment, DS 3 is marked as a CP (CP 80, which also indicates that the CP corresponds to RS 18) to solicit a new Reception Report. The receiver happens to receive all the retransmissions, and therefore sends a new RS (RS 19) that conveys that all data in scope 0:10 are received. The sender receives RS 19, sends an RA in response, and then closes the session; the receiver closes the session after receiving the RA. Note that the timer management procedures (turning off the timers upon receiving a response for the reliably transmitted segments) is followed
as in the description earlier. The loss of reliably transmitted segments themselves is not shown in the previous illustrations; but if such an event occurs, the timers would expire eventually, and the reliably transmitted segment alone would be retransmitted.
2.3.2 Other Features

LTP needs help from the local network management module to set the timers for the reliably transmitted segments appropriately; specifically, the one-way signal propagation delay to the LTP peer must be given. The retransmission timers are set to be twice the one-way signal propagation delay plus the Additional Anticipated Latency (AAL) to cover queueing, and processing delays.

LTP incorporates support for link disruptions in its design, but again needs support from the network management module in the form of “link-state-cue”s to inform it when the link goes down and when it comes back up. The network management module may pass such link-state-cues based on spacecraft, and antenna operational schedules, dynamic link-layer auto-detection procedures, etc. When a link-state-cue indicating the loss of link is received, LTP suspends retransmission timers for all active sessions; when the link-state-cue indicating link availability is received, the frozen timers are resumed for all active sessions.

LTP strives for bit economy, as bits are expensive to transmit and receive in deep-space. In the common practice of fixed length encoding of header fields, the protocol designer predicts the largest possible value for the field and makes the encoding size large enough to accommodate it. This however could mean wasted higher-order bits that carry no useful information if the typical value was significantly smaller than the maximum. Also, in the future, if it was found that the maximum value supported by a field was insufficient and affected protocol performance, expensive
protocol revisions may be required; this fact may also cause the designer to overestimate the maximum expected value in the first place. To address this problem, many of the LTP protocol header fields are encoded in a variable-length encoding scheme called the “Self-Delimiting Numeric Value” (SDNV) encoding scheme; the SDNV scheme is based on the Abstract Syntax Notation encoding rules [ASN02]. The SDNV encoding attempts to be bit-economical, while retaining scalability to accommodate arbitrarily larger values. It works as follows: A positive integral value (the only kind supported by SDNV encoding) is stored in the least-significant 7 bits of each octet; the most significant bit (MSB) of each octet is the discriminant, distinguishing the last octet of the sequence of octets of the encoding from the rest. MSB 1 indicates that the octet is either the first octet or one of the middle octets in the entire encoding, and MSB 0 indicates that the octet is the last octet of the SDNV encoding; the encoded value can be decoded back by concatenating the 7 least significant bits of all octets back-to-back, from the first to the last, and interpreting it as a positive integer. For example:

- 127 (decimal)=
  
  7F (hexa-decimal)=

  0111 1111 (binary)=

  0111 1111 (SDNV)

- 1000 (decimal)=
  
  3E8 (hexa-decimal)=

0000 0011 1110 1000 (binary)=

1000 0111 0110 1000 (SDNV)

- 16383 (decimal)=
  3F FF (hexa-decimal)=
  0011 1111 1111 1111 (binary)=
  1111 1111 0111 1111 (SDNV)

- 32767 (decimal)=
  7F FF (hexa-decimal)=
  0111 1111 1111 1111 (binary)=
  1000 0001 1111 1111 0111 1111 (SDNV)

SDNV encoding entails an overhead of 1/8 for the discriminant bit; the LTP specification uses SDNV encoding for fields that are expected to be at most 64 bit quantities (the 8 octet value could be stored in at most 9 octet SDNV encoding). For storing larger integral values such as public / private encryption keys, the SDNV overhead may become non-negligible, and one might choose an SDNV scheme for storing the length of the parameter, with the parameter of the specified length octets following; this scheme is recommended for some fields in LTP Extensions that typically take larger values.

LTP also offers a “best-effort” mode of operation with RED — GREEN block transmission semantics. The description of LTP basic operation earlier covered fully-
RED transmission of the block of application data, meaning that all of the block is required to be delivered reliably, subject to LTP’s ARQ procedure. A block may also be transmitted fully GREEN and sent in the best-effort mode, i.e., segments are transmitted only once (similar to the UDP protocol), and no ARQ procedure is invoked. The segment type field in the DS header indicates whether the segment is being transmitted RED or GREEN. It might often be the case that a block of data sent all GREEN is composed of application specific meta-data and data, and that the application data, or portions of it, might be useful only if the application meta-data was received reliably. For example, assume that a picture is transmitted from a spacecraft in a fully GREEN LTP transmission; the meta-data covering the dimensions, and related picture settings might be necessary to interpret the actual picture data or parts of it. To support this, LTP provides a red-part marker - the End of Red-Part (EORP) marker - for a mixed RED-GREEN transmission with a RED block-prefix followed by the GREEN block-suffix, with the EORP marker indicating the block offset from which GREEN data begins. The last DS of the RED block-prefix has the segment type RDS_EORP, which is also a CP segment sent reliably, soliciting a Reception Report from the receiver; thus the RED block-prefix alone is subject to ARQ procedures. The interface offered by LTP to its applications for block transfer must also require the RED block-prefix length to be given by the application, which may be 0 (fully GREEN mode), the same as the block size (fully RED mode), or a value in between (for RED-GREEN mode).
2.4 Testbed Setup

We describe the procedures followed in setting up the hardware and software components of our testbed here.

All our experiments were carried out in a testbed of GNU/Linux workstations running Linux kernel version 2.6.X; our TCP study was done with the Linux kernel 2.6.14 TCP implementation. Linux TCP implementations (beginning with higher 2.6.X kernel versions) support few experimental congestion control algorithms besides the standard TCP congestion control algorithms; we have to choose the standard congestion control algorithms by setting the kernel parameter:

```
net.ipv4.tcp_congestion_control = "reno"
```

The TCP implementation also supports TCP SACKs, and uses the FACK algorithm [MM96a, MM96b] for SACK-based Fast Recovery. It also includes support for larger initial windows as specified by RFC 3390 [AFP02].

Our TCP sender and receiver applications are implemented in Java 1.5. To make sure that application performance is not limited by TCP socket buffer size, we set the kernel parameters:

```
net.core.rmem_max, and net.core.wmem_max
```

that control maximum socket buffer size limits appropriately, and requested sufficient socket buffer sizes from the applications. Note that TCP Window Scaling options, and Timestamp options are enabled by default in the kernel.
For LTP tests, we used our Java 1.5 LTP implementation. Unlike TCP which runs in kernel space (i.e., as part of the Operating System), our LTP implementation runs in user space (i.e., as a user application from the Operating System’s perspective). Our LTP implementation is designed as a resident daemon program, exporting its interface to LTP applications via Java Remote Method Invocation (RMI). LTP applications - sender and receiver - effect transmission and reception of application data by invoking RMI calls on the LTP daemon, and themselves export methods to get informed of session events such as “Red-part data reception”, “Transmission completion”, etc., by the daemon, asynchronously.

The experiment testbed is illustrated in Figure 2.3. Our tests require three machines: the data sender, the data receiver, and the error emulator. As the names imply, the data sender machine models the data transmitter such as a spacecraft
Testbed #1:

<table>
<thead>
<tr>
<th>Role</th>
<th>Machine Name</th>
<th>CPU, Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Sender</td>
<td>Ram</td>
<td>Intel Celeron 431 MHz, 256 MB RAM</td>
</tr>
<tr>
<td>Error Emulator</td>
<td>Hanuman</td>
<td>Intel Pentium-4 1.6 GHz, 128 MB RAM</td>
</tr>
<tr>
<td>Data Receiver</td>
<td>Lakshman</td>
<td>Intel Celeron 565 MHz, 256 MB RAM</td>
</tr>
</tbody>
</table>

Table 2.1: Testbed Machine Information

Testbed #2:

<table>
<thead>
<tr>
<th>Role</th>
<th>Machine Name</th>
<th>CPU, Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Sender</td>
<td>Sambadhi</td>
<td>Intel Celeron 431 MHz, 256 MB RAM</td>
</tr>
<tr>
<td>Error Emulator</td>
<td>Jambhavan</td>
<td>Intel Pentium-4 1.6 GHz, 128 MB RAM</td>
</tr>
<tr>
<td>Data Receiver</td>
<td>Jatayu</td>
<td>Intel Celeron 431 MHz, 256 MB RAM</td>
</tr>
</tbody>
</table>

sending data on the downlink; the data receiver models the data receiver such as an antenna-site receiving the data beamed to it; and the error emulator machine models channel error characteristics by emulating a configured error-rate on the channel. Bandwidth and delay are emulated directly on the end-nodes (data sender and data receiver).

We had two such testbed setups so two independent tests could be done in parallel. The hardware details of the two testbeds are given in Table 2.1. All machines ran the GNU/Linux (kernel 2.6.x) operating system. The error emulator machines ran our errorem program to emulate a configurable error-rate; since errorem operates in user-space, we chose relatively more powerful workstations for error emulation to make sure that error emulation itself did not become a performance bottleneck.  

\[3^{\text{The machines are named after the author’s favourite characters from the Indian epic Ramayanam.}}\]
All three machines in a testbed are physically connected in a 100 Mbps switched Ethernet. We added host-specific routes on the data sender and data receiver machines for their data reception peers, so that the next-hop for their traffic was via the error-emulator. The `errorem` program uses the `tun` virtual network device to do the error emulation of packets in user-space. The `errorem` program opens the `tun` device upon initialization; this causes the operating system to associate the program with the `tun` device; depending upon system configuration, the operating system hands over certain packets from the real network device via the `tun` device to the program for processing. The `errorem` goes into an infinite loop reading packets from the `tun` device, applying errors to packets according to the user-specified error characteristics, and writing the packets back, which are then dispatched by the operating system via the real network device. Refer to Section A.1.2 for the full configuration procedure. We configured `errorem` to emulate the following error characteristics: no-errors; uniform, random, bit error-rates - $10^{-7}, 10^{-6}, 10^{-5}$.

While errors are emulated in the middle node, delay and bandwidth are emulated in the end-nodes themselves, via the Linux Traffic Control module `tc`. In our tests, we emulated 1 Mbps end-to-end bandwidth, with delays emulating geo-synchronous orbit, lunar, sub-geo-synchronous orbit distances. Refer to Section A.1.1 for the full configuration procedure.

The TCP and LTP protocol stacks in our tests are illustrated in Figure 2.4. The TCP protocol stack shows the standard TCP/IP over Ethernet protocol stack, and
Figure 2.4: Protocol Stacks

the LTP protocol stack shows LTP running over UDP over IP over Ethernet. Although in theory LTP could be operated directly over IP, or even over Ethernet, operating directly over UDP is much easier for implementation and use (no superuser privileges are required to run LTP, for example). Moreover, LTP can now utilize the UDP checksum for error detection. LTP specification does recommend an optional authentication extension that would also provide error-detection capability for LTP segments; however, our LTP implementation does not support this, and hence we relied on UDP checksums for detecting errors in the protocol-study tests. Our LTP implementation uses UDP port# 1113, which is reserved for LTP experimental use over UDP with the Internet Assigned Numbers Authority (IANA).

The standard Ethernet Maximum Transmission Unit (MTU) of 1500 bytes is used. TCP segments carried at most 1448 bytes of application data:
1500 - 20 (IP header) - 32 (TCP header with the timestamp option). LTP segments (variable length LTP headers + LTP application data) on the other-hand, were at most 1472 bytes in size:

1500 - 20 (IP header) - 8 (UDP header); total variable-length header size of data segments (Segment header + DS header) is typically below 20 bytes, and thus LTP application data size is seen close to 1452 bytes.

2.5 Experiments

We begin our protocol performance tests with our testbed set to model the communication latency of geo-synchronous satellite orbit.

2.5.1 Geo-synchronous Orbit Distances

Light travels a distance of 300 km in a milli-second, and the geo-synchronous orbit is at a mean distance of approximately 35000 kms from the mean sea level; the one-way signal propagation delay at the speed-of-light is thus approximately 116 milli-seconds (msecs). We emulated this one-way signal propagation delay of 116 msec at both of our end-hosts.

In the following analysis, we present throughput and duration plots to measure performance. Throughput and Duration are both measured at the data-sender. Throughput is the rate at which application data is transferred, i.e., the Application
Figure 2.5: Geo-synchronous Orbit, Ideal Channel, TCP & LTP Throughput

Data Size (in bits) divided by the Time to transfer (in seconds). Duration is defined as follows: for TCP, this duration is the total time the TCP session lasted at the sender-side (including the TCP initial and final hand-shakes); for LTP, it is measured as the time between the initiation of a new session for the sending LTP Application by the LTP daemon, and the time the LTP daemon closes the session (this is immediately after the RS segment indicating successful reception of all application-data is received, and the acknowledging RA segment is sent).

Figure 2.5 shows the sender-side throughput for transmitting files of various sizes in an ideal, no-error channel. Note:

1. This plot shows averages of 25 trials with 99% confidence-interval (the same is true for all other plots to follow).
2. All LTP transmissions are sent in fully RED mode [Section 2.3.2].

For LTP, the throughput seems farther from the channel bandwidth of 1 Mbps for smaller file sizes; for 100 KB, this is close to 80% of the bandwidth. This is so, because for smaller file sizes, the “xmit” time (time taken to clock-out bits on the channel) is significantly lesser than the signal propagation delay; since the sender necessarily waits for the Reception Report indicating successful reception from the receiver, one RTT is always added to the session duration. For example, for 100 KB file: 102400 bytes at 1 Mbps takes a xmit time of 0.8192 sec, but the RTT is 2 x 0.116 sec = 0.232 sec; these are approximately 80% and 20% of the total session duration, and hence we see 80% of the channel-bandwidth as throughput from LTP. Thus LTP is indeed performing close to optimal.
We notice that TCP takes longer than LTP to converge to the channel bandwidth; this is primarily due to the extra time expended by TCP in its slow-start algorithm. We present the session duration plot in Figure 2.6, to illustrate this further. In this figure, we also include the theoretical minimum value of the session duration as a base-line. This value is calculated as the sum of the “xmit” time (to clock out the application data on the channel) and the channel RTT. TCP stays in slow-start until a packet-loss event is noticed; in our set up, this never happens, and so TCP stays in slow-start for the entire session. Although the TCP congestion window (cwnd) keeps growing until it reaches its maximum value in slow-start, the expense due to slow-start is only in the number of RTTs necessary to get the cwnd large enough to fill the channel bandwidth and thus reaching steady-state. At steady state, the channel “bent-pipe” is kept full, and the Bandwidth-Delay-Product (BDP) (measuring the total pipe capacity) worth of data is sent every RTT. Here, the BDP =

\[ 1Mbp \times 0.232\, sec = \]

\[ 232,000\, bits = 29000\, bytes = \]

approximately 20 TCP segments (each carrying 1448 bytes of data).

Starting with the initial window of 3 TCP segments, and doubling it every RTT, this takes 4 RTTs (3, 6, 12, and 24 segments being sent progressively in every successive RTT). Note that two additional RTTs are also expended for the SYN and FIN hand-shakes. For shorter files, the RTTs spent in slow-start to reach steady-state may be significant relative to the number of RTTs spent in steady-state, and thus slow-start
may affect throughput significantly; for larger files however, the number of RTTs spent in steady-state is much larger than the few RTTs spent in slow-start, and thus the effect of slow-start becomes negligible. This fact is affirmed by the TCP duration plot shown, which comes closer and closer to the theoretical minimum for larger and larger file sizes.

Note: For smaller file-sizes (<100 KB), we notice that LTP duration is lesser than even the theoretical minimum. This is due to the inaccuracy in bandwidth emulation at shorter time-scales. We emulate a 1 Mbps channel on a 100 Mbps physical Ethernet channel via the token-bucket-filter (tbf) model. Implementations based on this model are bursty (transmitting multiple packets contiguously) in short time-scales. This burstiness is more noticeable for smaller file-transfers, where a bandwidth rate of 1.1 to 1.25 Mbps gets emulated instead of the configured bandwidth of 1 Mbps. This causes LTP performance to be slightly better than the theoretical maximum. For larger file-sizes however, the effect of burstiness becomes negligible, and near-perfect channel bandwidth emulation is obtained; consequently LTP’s performance falls within the theoretical maximum as expected.

Next, we present the performance of the two protocols in the presence of channel errors. We emulated uniform, random, Bit Error Rates (BERs) of $10^{-7}$, $10^{-6}$, and $10^{-5}$; the results are presented in Figure 2.7; TCP performance in the ideal channel is also shown as an upper-bound.
Figure 2.7: Geo-synchronous Orbit, Noisy Channel, TCP Throughput

- BER $10^{-7}$: Packet losses in the forward (data-path) and reverse (ack-path) paths are possible, but rare. Although lost packets are recovered via SACK-based Fast Recovery with the FACK algorithm, TCP does cut its sending rate by half, and this affects performance. For shorter files, such as those around 100 KB, this reduction in sending-rate has a noticeable impact because TCP does not have enough data to transmit, and hence not enough time (RTTs) to recover back to steady-state. However, for larger file sizes, such as those around 1 MB, this is not the case, and TCP is able to recover back to its original sending rate even after getting into Fast Recovery. Loss of ACKs in the reverse path also contributes to poorer TCP performance, since data packets are clocked out only upon incoming ACKs; we noticed this in a 100 KB file transfer: although not a
single data packet was lost in the forward path, due to ACK losses, TCP was not able to match its performance from the “no-errors” plot.

- **BER 10⁻⁶**: Packet losses are a much more common occurrence than in the previous case. For shorter file-sizes (< 100KB), however, the actual number of loss-events tends to be closer to that from the BER 10⁻⁷ case, and hence the two plots coincide. However, for larger file-sizes, this is no longer true; the larger number of packet loss events is noticeable, and TCP stays almost always in Fast Recovery mode. Just as soon as a Fast Recovery event ends and TCP begins congestion avoidance to gain back the original sending-rate, another loss event occurs, and it gets back to Fast Recovery. Thus on an average TCP sends at approximately half the channel bandwidth (500Kbps), as is seen from the plots for larger file-sizes.

- **BER 10⁻⁵**: Packet losses are plentiful here. Often, segments retransmitted in Fast Recovery are themselves lost, causing a TCP Retransmission Timeout (RTO) event, making it enter slow-start all over again. Further, RFC 3390 mandates that TCP should interpret an RTO as an indication of severe congestion in the network and therefore that TCP should begin its slow-start at the sending rate of 1 MSS (Maximum Segment Size) per RTT instead of the initial slow-start window of approximately 3 MSS per RTT. This design characteristic of TCP to enter slow-start upon an RTO event, although providing it with
Figure 2.8: Geo-synchronous Orbit, Noisy Channel, LTP Throughput

The much needed conservative and restrained behaviour on a shared network such as the Internet, hurts performance severely on a single-hop space-link with high error-rates. TCP almost always remains in slow-start, with an average outstanding window of approximately 5000 bytes (for the 1 MB case); sending 5000 bytes per RTT (0.232 sec) is approximately 40 Kbits per RTT, which is 172 Kbps. TCP plot thus seems to flatten-out at a throughput below this value, remaining constant with increasing file-sizes.

LTP performance for the same channel conditions is shown in Figure 2.8. LTP performs significantly better than TCP for BERs of $10^{-6}, 10^{-5}$. For 1 MB file transfers, throughputs of approximately 900 Kbps, and 750 Kbps are achieved at these
error-rates; TCP throughputs are approximately 500 Kbps and 120 Kbps for the same. As shown earlier, this is primarily because of the “packet-loss is due to corruption” assumption of LTP. LTP sends all of the file’s data in its first transmission and hears back a report on it from the receiver - forming the first transmission cycle, then all data reported as missing are retransmitted and a report on them is received - forming the second transmission cycle, and so on. For 1 MB file transfers, we observe the following average number of Transmission Cycles (TC): 1 (no-errors), 1.44 (BER $10^{-7}$), 2.04 (BER $10^{-6}$), and 3.6 (BER $10^{-5}$). LTP session-duration, in theory, is simply $TC \times (RTT + xmittime)$, where xmit time is time required to transmit the amount of data sent in that TC; from the plots, we find that LTP performance degrades gracefully with errors, depending on TC, matching expectations.

Next, we present results from our experiments for lunar distances.

### 2.5.2 Lunar Distances

The moon is approximately 384,000 kms from Earth; light travels this distance in 1.3 sec. Hence, we emulated this one-way delay from both the end-hosts and repeated our previous experiment.

We present upper-bound protocol performance to be expected from TCP and LTP, with the results from tests on the ideal (no-error) channel. Figures 2.9, and 2.10 show the throughput and session-duration graphs.
Figure 2.9: Lunar Distance, Ideal Channel, TCP & LTP Throughput

Figure 2.10: Lunar Distance, Ideal Channel, TCP & LTP Duration
The relative progress of the TCP and LTP throughput plots for increasing file-sizes is similar to the performance from the geo-synchronous orbit experiments 2.5, with LTP converging quicker to the channel bandwidth than TCP. But the convergence rate for LTP is slower than in the geo-synchronous orbit case; while LTP reaches around 90% of the channel bandwidth at 512 KB file-size for geo-synchronous orbit, it reaches the same mark only at 5 MB file-size for lunar orbit. This is to be expected because of the larger signal propagation delay at lunar orbit; while the xmit time for a given file-size remains constant for both cases, the ratio of xmit time to RTT for the lunar case is much lower than that of the geo-synchronous orbit case, and it takes a bigger file-size for the xmit time to become significantly larger than the RTT in the lunar case. To illustrate this, we present the following theoretical analysis:

- Geo-synchronous orbit case, reaching 90% bandwidth at 512 KB file-size: xmit time (@ 1 Mbps) = 4.096 sec, RTT = 0.232 sec; total session-duration = 4.328 sec; fraction of RTT in the session-duration is approximately 5%.

- For the same 512 KB file-size in the lunar case: xmit time = 4.096 sec, RTT = 2.6 sec; total session-duration = 6.696; fraction of RTT in the session-duration = 38%. Only for a 5 MB file, requiring a xmit-time of 40.96 sec and thus a total session-duration of 43.56 seconds, does the fraction of RTT in the session-duration drop to a comparable value of approximately 6%. Hence, it is for 5 MB file-size transfers at lunar distances that LTP reaches its matching throughput for 512 KB files at geo-synchronous orbit.
Note however that this only affects the sender-side throughput; an LTP receiver receives data at channel bandwidth in an ideal channel, independent of the RTT.

The duration plots also affirm this behaviour of LTP.

The BDP for this channel is:

\[ 1 \text{Mbps} \times 2.6 \text{sec} = \]

\[ 2,600,000 \text{bits} = 325,000 \text{bytes} = \]

approximately 224 TCP segments (each carrying 1448 bytes of data).

Starting with an initial window of 3 segments, TCP takes 8 RTTs (with a sending-rate of 3, 6, 12, 24, 48, 96, 192, 384 segments for every successive RTT) to reach steady-state via slow-start, and 2 RTTs (for SYN and FIN handshakes), for a total of 10 RTTs before it reaches steady-state. Since more RTTs are required (compared to the total of 6 RTTs required for the geo-synchronous orbit case), and because the RTT value itself is much higher, TCP converges to steady-state only for much larger file-sizes. This fact is also corroborated in the duration plots; it takes larger file-sizes for TCP to stay in steady-state for a relatively longer fraction of the total session-duration than the fraction of session duration spent in slow-start to get to steady-state.

We now present the performance of TCP in the presence of channel errors. The results are presented in Figure 2.11. As with the geo-synchronous orbit tests, we emulated uniform BERs of \(10^{-7}\), \(10^{-6}\), and \(10^{-5}\). The relative performance of TCP under these different error-rates is similar to the performance in the geo-synchronous
orbit tests in general. However, since 224 TCP segments are required to keep the sender in steady-state using full channel bandwidth, a packet loss event, even though infrequent in the BER $10^{-7}$ case, could have a much more severe performance impact. If a packet loss event occurs when TCP is in steady-state, TCP reduces its sending rate to 112 segments per RTT, and takes 112 RTTs to reach back to steady-state. Thus unlike TCP performance for geo-synchronous orbits, where it almost coincided with the no-errors case for larger file sizes, TCP performance is significantly poor at lunar distances for BER $10^{-7}$. At BER $10^{-6}$, loss events are much more common; TCP recovers them with Fast Recovery, but stays almost always in Fast Recovery phase, sending at half the sending-rate after every event. For BER $10^{-5}$, as with the geo-synchronous orbit tests, due to the plentiful occurrences of loss events, TCP
stays almost always in slow-start, sending on an average approximately 5000 bytes of outstanding data every RTT; but since the RTT value is higher, this constant throughput rate happens to be much lesser than its value from the geo-synchronous orbit tests. 5000 bytes every 2.6 seconds, gives us an upper-bound of only 15000 bps approximately.

LTP performance in the presence of channel errors is shown in Figure 2.12. LTP takes similar number of transmission cycles (TC) as with the geo-synchronous tests for 1 MB file-transfers: 1 (no-errors), 1.72 (BER $10^{-7}$), 2.04 (BER $10^{-6}$), and 3.76 (BER $10^{-5}$), and overall session-duration expected is given by $TC \times (RTT + xmittime)$, where xmit time is the time required to transmit the amount of data sent in that
TC. LTP performance is significantly better than TCP all through, becoming more significant with increasing error-rates.

2.5.3 Sub Geo-synchronous Orbit Distances

As the final set of experiments, we wished to study protocol performance at a distance lesser than the geo-synchronous orbital distance. The International Space Station (ISS) orbits in a low-earth orbit at approximately 350 kms from the surface of Earth; but since at the speed of light this is a mere 1.17 msec away, a TCP connection to the ISS via the Internet, taking a single space-hop, is likely to exhibit no different performance due to delay alone (error-rates, and link disruption may make it different than other terrestrial Internet connections, however). We could not model any realistic space entities for our studies below the geo-synchronous orbit; but for the purpose of this study to see how performance changed with nearer space realms, we arbitrarily chose a one-way delay of 50 msec. The results from this study are presented in the same format used in the previous sections.

TCP and LTP performance in the ideal channel are presented in the throughput graph in Figure 2.13 and the session-duration graph in Figure 2.14. The plots are similar in nature to the plots for the geo-synchronous orbit, except that we notice both TCP and LTP converging to the channel bandwidth much quicker. To illustrate, if we consider the file-size at which a throughput value worth 90% of the bandwidth (900 Kbps) is achieved:
Figure 2.13: Sub Geo-synchronous Orbit, Ideal Channel, TCP & LTP Throughput

Figure 2.14: Sub Geo-synchronous Orbit, Ideal Channel, TCP & LTP Duration
Figure 2.15: Sub Geo-synchronous Orbit, Noisy Channel, TCP Throughput

- TCP achieves this at 5 MB file-size for geo-synchronous orbit, but now achieves this at just 512 KB.

- LTP does even better. While this throughput threshold was achieved at 512 KB for geo-synchronous orbit, it is now achieved at just 100 KB.

This is primarily due to the shorter RTT and the shorter BDP for this setup. The session-duration plot also illustrates that the TCP overhead due to slow-start becomes negligible quicker, since only a few short RTTs are required to reach steady-state. Performance of TCP and LTP in the presence of channel errors show similar improvements, for the same reasons. TCP performance is presented in Figure 2.15, and LTP performance is presented in Figure 2.16.
Figure 2.16: Sub Geo-synchronous Orbit, Noisy Channel, LTP Throughput

We notice that TCP performance at BER $10^{-7}$ coincides with the no-errors case. This indicates that TCP is able to recover from the infrequent Fast Recovery phases back to its original sending-rate in negligibly small amounts of time, thus having limited performance impact. Performance is significantly better for the BER $10^{-6}$ case too, when compared with the geo-synchronous orbit tests; for 1 MB file transfers, while 520 Kbps throughput was achieved in the geo-synchronous tests, a throughput of 860 Kbps is now achieved. Although the same number of loss events would be encountered on both the cases, recovery is swifter now, because fewer number of segments are required to keep TCP in steady-state because of the smaller BDP of

$$BDP = 1 Mbps \times 100 \text{msec} = 100000 \text{bits} = 12500 \text{bytes}$$
about 9 TCP segments, compared to the 20 segments necessary in the geo-synchronous orbit case; further the RTTs are much shorter. Performance at BER $10^{-5}$ flattens out at a shorter throughput value, because TCP stays in slow-start almost always, due to plentiful errors, as with previous cases, albeit at a higher constant value than the previous experiments.

LTP performance, measured in the number of TCs required is almost the same as the previous tests, as expected. LTP takes 1 TC (no-errors), 1.68 TC (BER $10^{-7}$), 2.24 TC (BER $10^{-6}$), and 3.8 TC (BER $10^{-5}$); but since the RTTs are much smaller, this affects throughput performance even lesser than its performance from our previous tests.

2.6 Summary

We have studied the performance of the two protocols TCP and LTP under different channel delays (one-way delays of 50 msec, 116 msec, 1.3 sec) and error-rates (no-errors, uniform and random bit error rates of $10^{-7}$, $10^{-6}$, and $10^{-5}$, for transferring a range of file-sizes (from 10 KB to 10 MB).

- From our near-space environment experiments we have demonstrated that LTP performs optimally, matching theoretical expectations.
- We notice that the performance of TCP degrades noticeably in the geo-synchronous orbital distances, and extremely poorly for distances beyond, such as lunar
distances. TCP’s performance depends on the bandwidth delay product, the round-trip time, the size of the file being transferred, and the channel error-rates. TCP begins to perform poorly under channel error-rates higher than $10^{-6}$. We observe that TCP performance begins to match LTP’s performance when operated within the following realm of near-space, characterized by three dimensions:

- BER of $10^{-7}$ or lesser. If the raw channel BER is higher, some form of channel Forward Error Correction must be provided so that the TCP visible BER is in this range.
- One-way signal propagation delays of around the 50 msec range; since light travels 300 kms a msec, this would be around 15000 kms from the ground station on Earth.
- File sizes of at least 512 KB or higher.

The protocol boundary marking the realm as measured in the distance dimension is illustrated in Figure 2.17.

Note that this is one possible realm of near-space where TCP can be expected to perform reasonably well, and not the only such realm imaginable. All our tests were done for a 1 Mbps bandwidth, and our recommended realm boundary is dependent on this given bandwidth. For a 1 Gbps space channel for example, the same 50 msec experiments should be expected to give significantly different
Figure 2.17: TCP vs LTP Protocol Boundary: Distance; 1 Mbps Channel

results because the BDP is now much higher. We leave the near-space realm
dependency on the channel bandwidth as an interesting future work item.
In Chapter 1, we presented the motivation for a Priority Paradigm (PP) for deep-space applications. In the absence of a job-by-job priority paradigm, such as the one proposed, the other option is to configure the communication settings of the channel as a whole, presenting a “one-size fits all” approach for all jobs. Often this causes system engineers to “over-code” the channel (i.e., expend additional error-protection overhead for the data bits) for the average job to meet the expected performance level for all jobs. This approach has the benefit of being simple to implement and use, but tends to be wasteful of bandwidth. Over-coding the channel expends additional bandwidth overhead for applications that do not require the extra coding overhead, and may yet be insufficient for critical data commands and highly valuable science data. Our priority paradigm approach presents an alternative model in which channel coding and data handling procedures are done on a job-by-job basis, adaptive to the priority needs of the job. Each job can be tagged by the application with the required two dimensional priority paradigm values of Intrinsic-Value and Immediacy. Intrinsic Value parameter lets the application specify the innate value (for example, science or mission value) of the application data relative to other application data, and the Immediacy parameter lets the application specify the delivery time-constraints on the job relative to other applications (i.e., how urgently this job must be delivered).
The Intrinsic Value and Immediacy parameters may be expressed in many different units. For example, since Intrinsic Value specifies the degree of care required in successfully delivering the job, the natural units may be the probability of successful delivery required. Note that in an error-prone channel, no communication protocol can make a 100% successful delivery guarantee for a job within any given time constraint. However, one can get arbitrarily closer to 100% (stochastically speaking), as in 99%, 99.9%, 99.99%, 99.999%, etc., by expending more and more error-correction-coding overhead. Since Immediacy specifies time-constraints, the natural units for it are absolute time values such as “20070318-12:00:00 EST”, or relative time values from the time the job is initiated, such as “within the next hour”, “by tomorrow morning”, “within this communication window”, etc. If a reliable ARQ protocol such as LTP is handling the data transfer, an application may also express its Immediacy requirements as the number of Transmission Cycles it can afford for successful transfer. Each of the parameters may also be expressed simply in either of “HIGH”, “MEDIUM”, “LOW” settings, for example, if this coarse degree of resolution is sufficient. The relative efficacy of the differing priority units merits further study, and it may vary depending on the specific deployment environment (deep-space or near-space, for example). For simplicity, we assume in our work here that Intrinsic Value and Immediacy are always expressed in opaque integer units in the range [0-10], 0 being the least setting of the parameter, and 10 the highest. We assume that a lookup
table may always be constructed, independent of our choice of natural units, mapping those natural units to these opaque units.

The Intrinsic Value and Immediacy parameter settings specify the priority policy sought by the application. Various mechanisms may be necessary to implement the policy chosen, such as a choice from a suite of Forward Error Correction (FEC) Coding schemes, adaptive transmission rate control, adaptive packet sizing, etc.

In this chapter, we present our study of various FEC mechanisms, and their efficacy in implementing priority policies for different channel error characteristics. The layout of this chapter is as follows. We first present a overview of the FEC schemes studied; next we present the design of our Priority Paradigm in more detail, followed by implementation details; finally we present our experimental results, and our conclusions.

3.1 FEC Schemes

Forward Error Correction schemes add robustness to communication on an error-prone channel. Within a limit, that is determined by the error-correction capacity of a code, errors may be automatically detected and corrected at the receiver. An FEC scheme generates a set of encoded bits from a set of data bits, and transmits the encoded bits on the channel. From $k$ data bits, $n, (n \geq k)$ encoded bits are generated, with the $n - k$ extra bits providing the redundancy required for error correction. The FEC instance at the receiver processes the encoded bits received
from the channel, looks for the presence of error in the bits, corrects the bit errors in the error-locations detected (assuming that the number of errors is within the error detection and error correction capability of the code), and presents the correct decoded bits to the application. The amount of redundancy added for a set of $k$ bits, and the maximum number of errors that can be detected and corrected from a set of $n$ coded bits, are the basic properties of an FEC scheme.

FEC schemes may be broadly classified into stream codes and block codes. In stream codes, a contiguous stream of data bits are processed to generate a contiguous stream of encoded data bits for transmission. Convolutional codes are a good example of this class. In block codes, the data bits are segmented into blocks of fixed size; encoding is performed on one block at a time, and the encoded block is transmitted. Reed-Solomon codes are a good example of this class. FEC codes may also be classified as systematic or non-systematic. In a systematic code, the $k$ data bits appear directly in the $n$ encoded bits; the set of encoded bits thus comprise the original data bits and the error-correction bits (called parity bits). In a non-systematic code, the original data bits do not appear directly in the set of encoded bits.

### 3.1.1 Convolutional Codes

Convolutional codes were originally proposed by Elias [Eli55]. A Convolutional code is specified by the 3-tuple $(k, n, m)$, where:

- $k$ is the number of input bits presented at every discrete time unit to the encoder.
• \( n \) is the number of output bits generated at every discrete time unit by the encoder.

• \( m \) is the size (in bits) of a shift-register used to hold the current bit and the most recent \( m-1 \) input bits. The \( n \) output bits are generated as a function of the \( m \) bits in the shift register. The value \( m-1 \) specifying the memory size of the register is also called as the constraint length of the code (\( c \)).

The encoding procedure used to generate each of the \( n \) output bits is specified by the coefficients of a polynomial of degree at most \( c \) (comprising thus of at most \( m \) terms) that define how the output bit is generated from the \( m \) bits in the shift register. For example, with a \((1, 3, 3)\) convolutional code, upon each new input bit, 3 output bits are generated, using a 3-bit shift register. This code would be specified by 3 polynomials, specifying how each of the 3 output bits are to be generated. For a binary convolutional code, the output bits are generated by adders performing XOR operations on the specific bits from the shift register. For example, the polynomial \( 1 + x^2 \) has coefficients \((1, 0, 1)\); the corresponding output bit is generated by an adder which XORs bits from register positions 0 (LSB, the most recent bit), and 2 (MSB, the oldest bit). The operation of the encoder may also be seen as the behaviour of a linear time-invariant control system whose output is given by the convolution of its impulse response with the input; hence the name convolutional codes.

Note that redundancy is provided via:
• the number of encoded bits generated for each input bit. (the ratio k:n is known to take typical values of 1:2, 1:3, and even 1:6 in various convolutional coding implementations), and

• the persistence of a data bit in the shift register, i.e., an input bit continues to have its effect on the output of the encoder for a total of m time units.

The performance of a specific convolutional code depends on the polynomials generating the output bits; “good” polynomials are often found by computer simulation [CC82].

The decoding procedure is notably more complex than encoding, and custom-made hardware circuits are often used to improve decoding speed. Decoding can be performed via two approaches: Sequential decoding, and Maximum Likelihood decoding. The latter approach, called Viterbi decoding [Vit67] after its inventor Andrew J. Viterbi, is more commonly used. This approach is presented here.

We first observe that the convolutional coding process can be visualized as the operation of a finite state machine that traverses a set of $2^m$ states. Upon arrival of each new bit the state machine changes to a new state specified by the current contents of the m-bit shift register, and generates n output bits from the n adders.

Let the list of states traversed by the state machine constitute a state path. For an input sequence consisting of $N_i$ bits, we have a path of length $l = N_i + (m - 1)$ states, where $m - 1$ counts the final states visited by the encoder (after the last bit is processed) when the remaining contents of the shift register are flushed). The total
number of encoded bits transmitted is thus:

\[ N_t = l \times (n/k). \]

Now, assuming that the decoder is presented with the \( N_r \) received bits (same number of channel bits transmitted are assumed to be received, i.e., there are no channel erasures), it has to detect the most likely state path traversed by the finite state machine at the encoder, and from this, the most likely original input bits. There is no guarantee however, that the \( N_r \) received bits are the same as the \( N_t \) transmitted bits. The decoder has to detect the correct encoder state path from the potential \( 2^{l+m} \) paths and then decode back the original \( N_i \) data bits that correspond to this path. The interesting property of the Viterbi algorithm is that instead of the exponential-time search of \( 2^{l+m} \) paths, it retains at most \( 2^{m-1} \) potential paths at each of \( l \) steps in the decoding process, and runs in time \( O(l \cdot 2^{m-1}) \) (note that \( m-1 \) depends on the shift-register size, and is constant for a given convolutional code).

The following discussion is presented for a \((1, 2, 3)\) convolutional code; the decoder processes 2 encoded bits at a time and keeps track of \( 2^{3-1} = 4 \) states in every step. The decoder finite state machine begins in the state 00 (i.e., zero shift register contents). It considers the two choices for the input bits - 0 and 1, and the two encoded bits that must have been generated for either of these choices; both pairs of encoded bits are compared with the first two bits from the received sequence \( N_r \), and a distance metric, such as the Hamming distance metric, is calculated for both the choices. Now, both the potential input paths are retained (i.e., the original input bits may be 0 or 1)
with the decoder allowing the possibility of both the potential shift-register states: 00 (lets call this state A) or 01 (state B). The decoder now repeats the above procedure, allowing the possibility of the next input bit being 0 or 1. After this step, the decoder has four potential paths $A, A_1, A, A_2, B, B_1, B, B_2$ where $A_1$ and $A_2$ are the next two potential states from state A, and $B_1$ and $B_2$ are the next two potential states from state B. The decoder now calculates the distance metric between the first four bits in $N_r$ and the four encoded bits expected for each of the four potential paths. The tree of such paths thus grows as this procedure gets repeated, and for all the paths, the decoder keeps track of the distance metric, and the predicted set of input bits. However, the number of paths can grow to at most $2^{m-1}$ before two paths converge on a single state in the next step. At this point the decoder chooses the path with the least metric and its corresponding set of potential input bits, and discards the other path. This procedure is carried out until all paths converge into a single path; at this point the decoder can release the predicted set of input bits corresponding to this path as output, and continue on.

We refer the interested reader to [CC82, Con] for more details.

### 3.1.2 Reed-Solomon Codes

Reed-Solomon (R-S) codes [RS60] are systematic block codes. They belong to a family of error-correcting codes called Bose-Chaudhuri-Hocquenghem (BCH) codes. In the following section we describe BCH codes; much of this discussion applies to
R-S codes, except that R-S codes impose additional requirements on the generic BCH codes, which are noted. These codes have their foundations in abstract algebra and algebraic coding theory, and operate on a category of finite fields called Galois Fields (GF). A finite field may be loosely defined as follows:

- it comprises a finite set of n elements over which two operations: addition (+) and multiplication(.) are defined
- the result of addition or multiplication is always another element within the field
- the field has two elements that serve as the additive identity 0 and multiplicative identity 1
- every element ’a’ (besides 0), has additive inverse -a and multiplicative inverse $a^{-1}$
- the field is commutative (a+b = b+a; a.b = b.a), associative (a+(b+c) = (a+b)+c; a.(b.c) = (a.b).c), and distributive (a.(b+c) = (a.b)+(a.c)).

Note that in general, the number of elements in a GF, n, is either a prime number q or a power of the prime $q^m$ for some integer $m > 1$.

The BCH code-words are elements of a Galois Field of q elements (GF(q)), and the code is generally specified as (n, k) where k input data-symbols are suffixed with n-k parity symbols to give the n symbol code-word. A BCH code with n-k=2t parity
symbols can correct at most \( t \) symbol errors in the code-word of length \( n \) symbols. The functioning of a BCH code depends on the generator polynomial which defines how the parity symbols are generated: the input polynomial \( d(x) \) of degree \( k-1 \) (the \( k \)-symbol data-word \( d_0, d_1, \ldots, d_{k-1} \) provides its \( k \) coefficients) is multiplied by the generator polynomial \( g(x) \) of degree \( 2t \), to generate the code-word polynomial \( c(x) \) of degree \( n-1 \) with coefficients \( c_0, c_1, \ldots, c_{n-1} \). By choosing parity-blocks of various sizes, with their corresponding generator polynomials, many configurations of BCH codes such as \((7, 4), (15, 11), (31, 26), (255, 223)\) are possible, and each such instance provides a code with different error correction capacity. For example, the \((15, 11)\) code can correct at most 2 symbol errors in a 15 symbol code-word (since \( 2t=4 \)), and the \((255, 223)\) code can correct at most 16 symbol errors in the 255 symbol code-word (here \( 2t=32 \)). Note that although symbol sizes can be chosen to be arbitrarily large or small, it is typical to find implementations with symbol size of one byte (8-bit).

The BCH encoding process is simple - each incoming \( k \)-symbol code-word is multiplied by the generator polynomial to give the \( n \)-symbol code-word. The decoding process however, is more complex and more interesting. The BCH generator polynomial has the property that its \( 2t \) roots are consecutive powers of \( \alpha \), the primitive element of \( GF(q^m) \) (Note that the primitive element \( \alpha \) has the property that \( \alpha^n = 1 \), and all \( n \) elements can be uniquely expressed as a power of \( \alpha \)), as in \( \alpha^{m_0}, \alpha^{m_0+1}, \alpha^{m_0+2}, \ldots, \alpha^{m_0+2t-1} \). Note that for R-S codes, \( m_0 = 1 \).
Let us assume that a code-word \( \mathbf{c} \) was transmitted; now \( \mathbf{c} \) may be thought of as a vector comprising \( n \) elements: \( c_0, c_1, \ldots, c_{n-1} \), where each of the \( c_i \in GF(q) \). We define the finite field transform \( \mathbf{C} = C_0, C_1, \ldots, C_{n-1} \) where

\[
C_j = \sum_{i=0}^{n-1} c_i \alpha^{ij}, \quad \text{and each of the } C_j \in GF(q^m).
\]

Note that for R-S codes, the transform is defined such that \( C_j \in GF(q) \). The finite field transform has the property that \( C_j = 0 \) iff \( \alpha^j \) is a root of the generator polynomial; therefore the transform of any valid code-word has \( C_j = 0 \) for \( j = m_0, m_0 + 1, m_0 + 2, \ldots, m_0 + 2t - 1 \). The errors added to the transmitted code-word \( \mathbf{c} \) may be represented by a vector \( \mathbf{e} \), such that the received code-word \( \mathbf{r} = \mathbf{c} + \mathbf{e} \).

The decoder’s task is to predict the most-likely transmitted code-word \( \mathbf{c} \) given \( \mathbf{r} \). Representing the transforms of the vectors \( \mathbf{r}, \mathbf{c}, \) and \( \mathbf{e} \), by \( \mathbf{R}, \mathbf{C}, \) and \( \mathbf{E} \) respectively, we note that

\[
\mathbf{R} = \mathbf{C} + \mathbf{E}
\]

If there were no channel errors, \( R_j \) must be 0 for \( j = m_0, m_0 + 1, m_0 + 2, \ldots, m_0 + 2t - 1 \), and we know that the transmitted code-word \( \mathbf{c} \) was received without any errors. If not, we know that one or more errors exist. We specify the “ Syndrome” of the polynomial to comprise the vector \( E_j, j = m_0, m_0 + 1, m_0 + 2, \ldots, m_0 + 2t - 1 \). We know that if there were \( v \leq t \) errors, \( \mathbf{e} \) must have a 1 in at most \( v \) positions, and 0s everywhere else. We now construct a polynomial \( \Lambda(z) \) that has an inverse transform \( \lambda \) such that \( \lambda \) has a 0 at the \( v \) error positions and 1s everywhere else (i.e., \( \lambda \) is the 1’s complement of \( \mathbf{e} \)). Therefore,
\[ \lambda_i e_i = 0, \ (i = 0, 1, \ldots, n - 1). \]

Since multiplication in this domain is equivalent to convolution in the transform domain, this means that,

\[ \sum_{k=0}^{n-1} \Lambda_k E_{j-k} = 0 (1). \]

(1) is called the Key equation. The decoding task comprises the following steps:

- **Solve the key equation**: Determine the \( n \) coefficients of \( \Lambda \) to construct the error-locator polynomial \( \Lambda(z) \). Note that \( \Lambda(z) \) has at most \( v \) terms, with a degree of at most \( v-1 \). Since we know \( 2t \) values of \( E_{j-k} \), substituting them back in (1) yields a set of \( v \) linear equations. Solving them yields the coefficients of \( \Lambda(z) \). The Euclid’s algorithm for determining the greatest common divisor may be adapted for this; a more efficient practical implementation is the Berlekamp algorithm, along with optimizations given by Massey, that is together designated as the Berlekamp-Massey algorithm.

- **Determine the error-locations**: Once the coefficients of the error-evaluator polynomial \( \Lambda(z) \) are determined, we solve it to obtain the \( v \) roots, which then directly give the error positions. The Chien search procedure does this efficiently.

- **Correct the errors**: If the individual symbols constituting the code-words were single bits, error correction is merely toggling the bit in the error position. For larger symbol sizes, we need to construct the error-evaluator polynomial
\( \Omega(z) \), evaluate it at each of the error-positions to determine the error value, and then correct the symbol appropriately.

We refer the interested reader to Chapter 5 of [CC82] where the above steps are described in detail.

### 3.1.3 Raptor Codes

Raptor codes belong to the family of Digital Fountain codes (referred to as fountain codes from here on). Before we describe fountain codes any further, we note that they are designed to operate over a binary erasure channel. The Binary Erasure Channel was proposed by Elias [Eli55], with the following basic properties: the channel never outputs any symbols with errors; the symbols known to be error-free are released (the binary channel can be thought of being in the state “1” for this symbol), while the symbols known to contain errors are silently discarded and considered erased (channel state being “0”). In the Internet, packets are generally tagged with checksums to detect errors, and packets with errors are silently discarded; thus the Internet is a good example of a binary erasure channel. Note that other error-correction codes such as convolutional codes and Reed-Solomon codes do not require a binary erasure channel; they accept symbols with errors for error detection and correction.

The basic behaviour of fountain codes is as follows: From a set of \( k \) input symbols \( x_1, x_2, \ldots, x_k \), a potentially infinite stream of encoded symbols \( z_1, z_2, \ldots \) may be generated. All encoded symbols are equivalent in decoding, and the input symbols may
be decoded, with high probability, from any $n = k(1 + \epsilon)$ distinct encoded symbols. Here, $\epsilon$ is any positive real number greater than 0; it is typically close to 0, and the codes typically perform near-optimal. Note however, that decoding success is probabilistic. While the codes exhibit a high probability of decoding success even with a small value of $\epsilon$, increasing $\epsilon$ and providing a larger number of encoded symbols only helps increase the probability of decoding success. As a metaphor, the codes are like a water fountain that continuously spouts water droplets. Collecting a full glass of water quenches our thirst (with high probability), and it does not matter which exact droplets we collect. Any extra water droplets we may collect only increases the likelihood of quenching our thirst.

The input symbols and encoded symbols are of the same size. They could be just one bit, or a bit vector of any arbitrarily length $k$. In theory, symbols could be of arbitrary length (to a certain extent) in other error-correction schemes too; but in practice, traditional FEC schemes tend to be expensive to compute over large units of data at a time. On the other hand, the core operations involved with fountain codes are XOR operations over a small set of symbols to generate an output symbol, and are easier to implement with modern CPUs, even for large symbol sizes. It is more natural to think of fountain encoding as operating over a group of symbols at a time, while other traditional coding schemes such as convolutional and Reed-Solomon codes are better thought of as operating over a symbol at a time. Also, since fountain decoding is probabilistic and the codes are rateless, we have the flexibility to choose
the degree of error-protection offered by the code. For example, for a regular unit of application data, one may choose to generate just the minimal number of encoded symbols required to provide the baseline probability of decoding success; whereas for a unit of critical application data, one may choose to generate some extra encoded symbols to be more sure of decoding success, and to be more robust to channel errors.

The digital fountain concept was first expounded in [BLMR98] as a good mechanism for multicast data distribution. A class of fountain codes called Tornado codes were presented; “Tornado codes are similar to Low Density Parity Check (LDPC) Codes presented by Gallager, except that they use an irregular weight distribution”[Ami06]. Tornado codes offer good encoding and decoding performance (i.e., they require only a constant number of XOR operations in encoding an decoding). However, their decoding overhead (i.e., the number of encoded symbols required to decode back the input symbols with high probability) is a function of \( n \) (where \( n \) is the number of encoded symbols generated from the \( k \) input symbols), and thus for low-rate settings (low \( \frac{k}{n}, k << n \)), could become unreasonably high. This disadvantage was overcome with the Luby-Transform codes (LT Codes) presented by Luby in [Lub02]. LT codes use the “Robust-Soliton” function as the degree distribution function, where the degree distribution function specifies how many input symbols get XOR-ed to generate an encoded symbol, and is the heart of LT Codes. LT Codes offer better asymptotic decoding overhead, given by

\[ k + O(\sqrt{k \ln^2(\frac{k}{\delta})}), \]

where \( 1 - \delta \) is the probability of decoding success. Note that the
decoding overhead depends only on $k$ - the number of input symbols, for a chosen $\delta$, unlike Tornado codes. The degree distribution is logarithmic in $k$, and thus the encoding and decoding performance is given by $O(ln(\frac{k}{\delta}))$, which is poorer than that of Tornado codes. The next class of fountain codes, called Raptor codes, builds on LT codes and provides constant time encoding and decoding performance, with constant decoding overhead. Before describing Raptor codes, we briefly describe the basic operation of LT codes, which applies to Raptor codes as well.

The basic operation of LT codes is illustrated in Figure 3.1. The encoding and decoding processes are characterized by the degree distribution function $\Omega(x)$, and the number of input symbols, $k$. For generating each output symbol:

- $\Omega(x)$ is queried to give its degree $(d)$
- $d$ input symbols are chosen at random based on a uniform distribution
- these input symbols are then XOR-ed to give the output symbol.

For example, output symbol $z_1$ chose a degree 3, and then chose 3 input symbols $x_1, x_2, and x_4$; these input symbols are then XOR-ed to give the value of $z_1$.

At the decoding side, for every received encoded symbol, the decoder needs to be told of the encoding degree $(d)$, and the identity of the $d$ input symbols that were XOR-ed to generate it. This information may be communicated to the peer in many ways. All this meta-data could be piggy-backed to the encoded symbol itself and transmitted in a packet header. However, this could entail a large overhead. The
Figure 3.1: LT Codes: Operation
meta-data could be made even shorter, if it can be assumed that the decoder shares knowledge of the pseudo-random number generator function used by the encoder. In this case, for each encoded symbol, the meta-data comprises only: the symbol sequence number (in the range 0 to n-1), symbol size (in bytes), k, and n. The decoding process works as follows:

- When an encoded symbol with degree 1 is received, the corresponding input symbol is known directly from it. For example, from $z_4$, we trivially derive the value of $x_3$ ($z_4 = x_3$).

- Once the value of an input symbol is derived, it is applied back to all output symbols it is connected to. For example, we know that $z_3 = x_2 + x_3$; we know the value of $z_3$, and just derived the value $x_3$; the value of $x_2$ is thus obtained by $x_2 = z_3 + x_3$. Thus the value of an input symbol is applied back to the encoded symbols it is connected to, and these edges are erased from the graph; in this process we might release new encoded symbols (i.e., reduce the degree of some encoded symbol to 1), which are then used to keep the decoding process alive.

The degree of $z_1$ for example would have been reduced to 2 when the value of $x_2$ is applied back; once the value of either $x_1$ or $x_4$ is derived, the value of the other ($x_4$ or $x_1$, respectively) could be derived using $z_1$.

Raptor codes [Ami06] build upon LT codes by using a pre-coding process on the input symbols before LT coding. The k input symbols are first pre-coded using
another standard erasure protection code to generate \( m \) intermediate symbols, and
the \( m \) intermediate symbols are then LT-coded to obtain the set of \( n \) encoded symbols
(for any arbitrary \( n \)). The decoding process, in concept, is just the reverse of this;
the set of encoded symbols are decoded with the LT decoder to derive the set of
intermediate symbols, which are then decoded with the pre-code to give the set of
\( k \) input symbols. Raptor codes have optimal asymptotic performance; the encoding
and decoding process take a constant time of \( O(ln(\frac{1}{\epsilon})) \), and the decoding overhead
is given by \( O(k(1 + \epsilon)) \) where \( \epsilon \) is any real number greater than 0.

The procedure for implementing Raptor Codes is given in the Internet-Draft
[LSWS07], which is a version 05 document at the time of this writing. This doc-
ument describes systematic Raptor codes, along with an efficient encoding/decoding
algorithm. The pre-coding process involves generation of LDPC [Gal63] codes and
half-symbols.

- From the set of \( k \) input symbols, \( s \) LDPC symbols are generated based on an
LDPC degree distribution, where \( s \) is derived as follows [LSWS07]: Choose the
smallest integer \( x \) such that \( x.(x - 1) \geq 2.k \). Now, \( s \) is the smallest prime
number such that
\[
\left\lceil \frac{K}{100} \right\rceil + x.
\]

- The \( h \) half-symbols are calculated from the \( k + s \) available symbols using the
gray sequence. \( h \) is derived as follows [LSWS07]: \( h \) is the smallest integer such
that \( h^{C_{\left\lceil \frac{h}{2} \right\rceil}} \geq (k + s) \). Let \( h' = \left\lceil \frac{h}{2} \right\rceil \). The gray-sequence is a sequence of positive
integers with the property that each element in the sequence differs from its subsequent number in only one bit position in their binary representation. The $i^{th}$ element in the gray sequence, $g(i)$ is given by $g(i) = i + \lfloor \frac{i}{2} \rfloor$, where $+$ is the XOR operation. The half-symbols are generated by choosing the first $k+s$ elements from the gray sequence such that they have exactly $\lceil \frac{n}{2} \rceil$ 1s in their binary representation; let us call this set of gray-sequence elements, $G$. Now, each half-symbol corresponds one-on-one to a distinct element $a$ from $G$, and is generated by XOR-ing those $k+s$ symbols whose respective indices have a '1' in $a$.

Thus, $m=k+s+h$ intermediate symbols are generated by precoding the $k$ input symbols; now, a potentially infinite number of encoded symbols may be generated from these intermediate symbols using the LT Coding algorithm. The encoding, and decoding algorithms involves solving a system of equations represented in a sparse matrix. The encoder and decoder share the same pseudo-random generator and the degree distribution functions. As mentioned earlier, an implementation only needs to communicate the following meta-data for each encoded symbol via packet headers or other means: $k$, $n$, encoding symbol sequence number (in the range 0 to $n-1$), and the symbol size (in bytes) in use.
3.2 Design

We now present the design of our Priority Paradigm in detail. The task of the Priority Paradigm layer is to choose a PP Mechanism that would adapt the error characteristics of the channel - as visible to the LTP Application, in a manner most appropriate for the priority policy sought. This requires the choosing, and if applicable also the tuning, of an error-protection scheme for the job before its transmission on the Datalink.

We modified our protocol stack so that the PP layer operates below the LTP layer and above the Datalink layer, as illustrated in Figure 3.2.

The operation of the PP layer is designed to be transparent to LTP; any addition or deletion of PP Mechanisms can be done in the PP layer without any change to the LTP layer implementation. LTP notices a difference in channel error-characteristics due
to various PP mechanisms only via the different amount of retransmissions required: sessions whose data have a higher quality PP Mechanism (such as a stronger FEC scheme) might see lesser amounts of retransmission, while those with lesser quality PP mechanisms (a weaker FEC scheme for example) might see more. The PP layer chooses PP mechanisms based on the PP policy given; since the policy parameters (i.e., the actual values for the PP dimensions) are chosen by the Application, the following minimal change is required in the LTP layer to trickle down this information to the PP layer. The main LTP Application Programming Interface (API) method for data dispatch - the sendBlock() routine - normally takes the following parameters from the application: Destination LTP Engine ID, Destination Client Service ID, the Block of Application Data to send, and the End-of-Red-Part offset marker. We added a new argument to this method for PP policy settings. This argument is used by the Application to pass its policy settings; the policy settings are given as a two-dimensional vector, with the first value specifying Intrinsic Value and the second value specifying Immediacy. Both components of the vector need to be in the range [0-10], with 0 being the least setting, and 10 the highest. The policy values are opaque to LTP, which just passes them to the main PP dispatch routine along with other parameters. The task of the dispatch routine is first to choose the best PP Mechanism to use for the PP policy values, apply the appropriate PP Mechanism encoding procedures, and then effect transfer of the encoded segments via the datalink. The dispatch routine may use the predicted error-characteristics of
the channel (based on the recent history of channel errors, for example), if they are available, in choosing the PP Mechanism.

We entail the problem of signaling the chosen PP Mechanism to the communicating peer. Note that this is the inevitable overhead of doing adaptive FEC on a job-by-job basis versus having a constant FEC scheme for all jobs. At the minimum, we need to signal the following:

- **PP\_Mech** - the specific PP Mechanism type in use for the job, such as Convolutional codes, Reed-Solomon codes, etc.

- **JobID** - a unique identifier for the job.

Since the communication channel is prone to errors, this signaling information needs to be protected from channel errors as well. This signaling must be afforded reasonable error-correction capabilities to automatically correct any channel errors encountered. It needs even stronger error-detection capabilities to reliably detect corrupted signaling information, especially when the number of channel errors overwhelms the error-correction capability provided. Besides the above generic signaling information, some PP Mechanisms may require additional mechanism-specific parameters to be signaled; for example, Raptor Codes need the number of input symbols, the number of encoded symbols, the Encoded Symbol Identifier for every encoded symbol, etc., to be conveyed for each encoded symbol, while other mechanisms such as Convolutional codes and Reed-Solomon codes may need to append a Cyclic Re-
dundancy Check (CRC) to the data before encoding, so that the decoder may use the CRC to verify successful decoding. Thus, we notice that the signaling information comprises the generic signaling part and the PP mechanism-specific signaling part, and packets generated by the LTP-PP system are encapsulated as shown in Figure 3.3.

### 3.3 PP Implementation

We support the following PP Mechanisms in our implementation:
• *Conv 1:2* Convolutional codes with rate $\frac{1}{2}$, and constraint length $m = 7$. The two coding polynomials used are those recommended by CCSDS, given by the decimal values 109 and 79.

• *R-S* Reed-Solomon codes with parameters (255, 223). Symbols are 8-bits in length, and the generator polynomial as recommended by CCSDS is given by the decimal value 391.

• *R-S & Conv 1:2* Inner Reed-Solomon (255-223) codes, and outer Convolutional (rate $\frac{1}{2}$, constraint length 7) codes. Each LTP segment is first encoded in Reed-Solomon codes, and the resulting output is then encoded in convolutional codes.

• *RPTR* Raptor codes implementation based on the Internet-Draft [LSWS07].

• *RPTR & R-S* Inner Raptor codes, and outer Reed-Solomon codes with parameters (255, 223). Here, groups of LTP segments are first encoded via Raptor Codes of any chosen code-rate, and each of the Raptor encoded segments are then encoded again in Reed-Solomon codes.

• *NO FEC* This is the trivial PP Mechanism type where no PP Mechanism is applied on the data, and reliability is provided only by the retransmission capabilities of LTP.

Our implementation of Convolutional codes and Reed-Solomon codes were based on the public domain, open source implementations of them by Phil Karn [Kar]. We
ported his C implementation to Java. We implemented Raptor Codes based on the procedure specified in the Internet-Draft [LSWS07] ¹.

Our generic PP header comprises the following fields:

- \( PP_{Mech} \): a 1-byte value that uniquely identifies the specific PP Mechanism in use. For example, we use 3 for \( RPTR \), 4 for \( R-S \).

- \( JobID \): a 4-byte integer uniquely identifying the job.

- \( CRC \): a 4-byte CRC calculated on the previous 5 bytes.

The JobID field in every packet allows the PP receiver to gather all packets belonging to a job and then decode the job as a whole. The LTP layer makes a dispatch request with the PP layer whenever it has a related set of segments to transmit, such as: original application data transmissions; retransmissions; LTP control information such as Report Segments, Report-Acknowledgment segments, etc. Based on the specific PP Mechanism chosen, the PP layer generates one or more jobs for each such dispatch request. For convolutional codes and Reed-Solomon codes, encoding and decoding are actually done on a segment by segment basis, and the PP layer creates a single job comprising as many LTP segments as are part of the dispatch request. For Raptor codes, however, our implementation is configured to operate on 100 input symbols at a time, i.e., \( k=100 \), and any \( n \geq k \) encoded symbols are generated based on the chosen code-rate. For example, if say 402 segments are presented in the PP

¹This is version 05. Our implementation was however made from the earlier version 04; we notice only minor changes between the two versions, though
layer dispatch request, 5 jobs are created: the first 4 jobs process 100 LTP segments each ($k=100$) and the last job processes 2 LTP segments ($k=2$).

The CRC helps detect errors in the generic PP header; if there are errors in it, the whole packet is discarded. In our experiments, the Datalink layer was UDP/IP over Ethernet, and we found our original error-detection strategy based on UDP checksums to be insufficient. The UDP checksum algorithm based on calculating the 1’s complement of the 16-bit XOR-sum of the packet contents gives occasional false-positives under larger error-rates (such as those greater than $10^{-3}$). Therefore, we added a CRC to the generic header itself, though it indeed is a large percentage overhead (4 bytes for 5 bytes of data).

Since having errors in the generic header is expensive, it is important to add error-correction capability to it. The safe strategy for doing this depends on the worst-case channel error-rate. Note that while we can adaptively change the error-protection strategy for the application data, it is necessary to have a fixed way to encode the generic header or we have to signal the adaptive strategy to the peer as well, creating a cyclic problem. The highest error-rate we studied was $10^{-2}$, and we found convolutional coding to be the safe strategy in our error-range. The 9 byte header is convolutionally encoded with our rate $\frac{1}{2}$, $m=7$, convolutional encoder to give a 20 byte encoded header. Thus this constant 20 byte PP overhead is present in all packets generated by the PP layer.
The PP receiver processes the generic PP header on all incoming packets as follows. It first decodes the 20 byte header to get the original 9 byte header; the CRC is computed afresh on the first 5 bytes of the header and verified with the received CRC. Packets failing the CRC check are discarded, while those clearing it are accepted.

If the JobID of a packet is greater than the active JobID, it indicates complete reception of all packets belonging to the active job. This follows from the in-order delivery requirement on the channel, and the requirement that jobs not be preempted during transmission (as is true with our implementation). Upon complete reception, an active job may be processed by applying the appropriate PP Mechanism’s decoding algorithm, and some or all of the original LTP segments may then be recovered for delivery to the LTP layer. The packet with the new JobID is now saved as the first packet of the new job. If the job being transmitted is the last job from the PP sender, no packets with a different JobID will arrive to indicate job completion. In this case, as also when there is a noticeable pause between jobs, the PP receiver depends on a Datalink timeout to indicate lack of channel activity, and thus to infer complete reception of the current job. The Datalink timeout therefore needs to be configured correctly. One important criteria is to make sure that it is large enough to cover any jitters introduced due to processing, queuing, or other delays, lest we infer job completion prematurely. We found 4 seconds to be a conservative choice for our UDP datalink timeout.
Now, assuming that all packets belonging to a given job are available, processing may continue based on the specific PP Mechanism, as follows.

- Convolutional codes \((Conv\ 1:2)\): In the encoding process at the sender, a CRC is calculated on the LTP segment, and tagged as prefix to the LTP segment. The ensemble is then convolutionally encoded to give the encoded packet. This CRC aids in verifying the success of convolutional decoding: after convolutionally decoding a received packet, the PP receiver verifies the CRC and only upon success clears it for delivery to the LTP layer.

- Reed-Solomon codes \((R-S)\): Here again, a CRC is added to verify decoding success. LTP segments at most 219 bytes long are prefixed with a 4-byte CRC over the segment contents to give the 223 byte input block for the code. In the encoding process, 32 parity symbols (32 bytes) are calculated and suffixed to the input block to give the 255 byte block for transmission. At the decoder, the R-S decoding procedure is invoked on the incoming block, and after decoding, the CRC is verified. The LTP segment contents from only those packets clearing the CRC check are cleared for delivery to the LTP layer.

- Inner Reed-Solomon, Outer Convolutional codes \((R-S \& Conv\ 1:2)\): The encoding and decoding procedures are straightforward applications of the above two procedures. The LTP segment is prefixed with a CRC (Reed-Solomon CRC) and is Reed-Solomon encoded to give a 255 byte block. A CRC is now
tagged to the front of this Reed-Solomon output block again (Convolutional CRC), and the 259 byte ensemble is then convolutionally encoded to give the encoded packet. At the decoder, the incoming packet is first convolutionally decoded; if the Convolutional-CRC verifies, the 219 byte block of R-S data following the Reed-Solomon-CRC is written right-away to the LTP layer. If the Convolutional-CRC fails, Reed-Solomon decoding is invoked on the convolutional decoder output (excluding the Convolutional-CRC). After this decoding, the Reed-Solomon-CRC is verified, and if it succeeds, the 219 byte block of R-S data following this CRC is deemed fit for delivery to the LTP layer. If the Reed-Solomon CRC fails to verify, the packet is discarded.

- Raptor codes ($RPTR$): Raptor codes require each packet to carry a mechanism specific header carrying the following information.

1. $SymSZ$ The Raptor symbol size (in bytes) in use.

2. $K$ The total number of input symbols.

3. $N$ The total number of encoded symbols generated.

4. $ESID$ Encoded Symbol ID, a number in the range $0, 1, \ldots, N - 1$, identifying the encoding symbol.

Each of the above fields take values that are at most 2 bytes long, and thus the net $RPTR$ header is 8 bytes long. Note that in our implementation, we only
carry a single raptor-encoded symbol per packet (though in theory, there is no such requirement).

We supported a default symbol size of 256 bytes, and processed $RPTR$ jobs taking at most 100 input symbols (LTP segments) at a time for the encoding procedure (i.e., $K = 100$, at most); but based on the requested code-rate, an arbitrary number of encoded symbols ($N \geq 100$) may be generated. All $N$ encoded symbols also have the CRC as a prefix to the 8-byte header, with the CRC (Raptor-CRC) following the generic PP header and covering the remainder of the packet: the 8 byte $RPTR$ header and the encoded symbol contents. At the receiver end, Raptor-CRC is verified first on all of the $N' \leq N$ encoded symbols received (it is assumed that $N - N'$ symbols failed the CRC on the PP generic header and were already discarded); only the $M \leq N'$ segments clearing the Raptor-CRC are taken for processing. Using the the $RPTR$ header from each of the $M$ segments, Raptor decoding is done, and some or all of the $K$ original input symbols (LTP segments) are obtained.

- Inner RAPTOR, outer Reed-Solomon codes ($RPTR \& R-S$): Since RAPTOR codes require the erasure channel requirement, even a single bit error on an encoded packet causes CRC failure, and the packet is discarded. The motivation behind $RPTR \& R-S$ codes is to extend the range of operation of $RPTR$ codes to domains with high error-rates, such as $10^{-3}$, when almost all packets (of size 256 bytes each) are expected to take at least one bit error. The outer Reed-Solomon
codes offer error-correction on a packet-by-packet basis, while the $RPT R$ codes provide error-correction over a group of packets. At the encoder, first $N$ Raptor-encoded symbols are generated from $K$ input symbols (LTP segments). Next, each of the $N$ encoded symbols (each of which are at most 219 bytes in size) are prefixed with a 4 byte CRC covering it, and the 223 byte input block thus obtained is Reed-Solomon encoded to give a 255 byte output block. At the receiver-end, the reverse procedure is done. Each of the $N'$ ($N' \leq N$) packets received are first Reed-Solomon decoded and only packets clearing the Reed-Solomon CRC are kept; $M$ ($M \leq N'$) Raptor-encoded symbols are recovered, which are then Raptor-decoded to give some or all of the $K$ original input symbols (LTP segments). These are then written to the upper LTP layer.

- **NO FEC**: The coding procedure is trivial: a 4 byte CRC is calculated on each LTP segment and prefixed to the segment to give the PP encoded symbol; at the receiver side, CRC is verified on each arriving packet, and only the LTP segments from the packets clearing CRC are delivered to the LTP layer.

### 3.4 Experimental Results

In our experiments, we studied the performance of our PP Mechanisms under various channel error conditions.
Our testbed hardware setup was similar to the one described earlier in Section 2.4 and illustrated in Figure 2.3. We emulated a 1 Mbps channel, with the Data Sender and Data Receiver hosts communicating via the Error-emulator host. The given error-rate was emulated with the `errorem` program on the error-emulator host. The protocol stack on the end-hosts was as illustrated in Figure 3.2, with the UDP/IP-over-Ethernet datalink layer.

Our goal here was to study our Priority-Paradigm model on the deep-space channel in general. We avoided emulating specific deep-space channels such as Earth-to-Mars (closest one-way light-time: 4 min) or Earth-to-Jupiter (closest one-way light-time: 35 min) towards this goal because:

- Emulating such actual, but large, values for RTT means that our experiments would take a larger time to complete (take days, typically).

- Moreover, we observe that, once we emulate a channel delay that is sufficiently larger than the transmission time (the time it takes to clock out bits on the channel at the channel data-rate), our performance results should, in general, be applicable to any larger deep-space channel delay. Note that performance metrics based on actual time units, such as throughput, may differ between different actual deep-space links since the actual RTT values might be different on each; but instead, if we measure performance in abstract units such as number of RTTs, for example, the results could be applicable directly for any
deep-space channel by merely substituting the actual channel RTT value in the results.

In our study, we measure the performance of various PP Mechanisms at the LTP layer in the abstract Transmission Cycle (TC) units. The actual time units corresponding to a TC in our experiments is given by the TC-Eqn:

\[ TC = RTT_{emul} + (k_1 \cdot FileSize) + k_2 \]

where \( RTT_{emul} \) is the emulated round-trip time; \( k_1 \) is a constant such that \( (k_1 \cdot FileSize) \) gives the upper bound on the PP processing delay (encoding or decoding time); and \( k_2 \) is a constant covering the time necessary for the PP receiver to infer job completion \((k_2 \) is upper-bounded by the Datalink timeout value used to indicate job completion, and is 4 seconds in our tests).

We expected to test a file-size of at most 1 MB in our experiments; since our emulated channel bandwidth was 1 Mbps, this takes a transmission-time of slightly higher than 8 sec (when the extra overhead due to the protocol headers is also included). We chose to emulate a conservative one-way delay of 10 sec on both end-hosts, resulting in a 20 sec \( RTT_{emul} \). Since our test measurements are in the LTP layer on the sender-side, we had to seed the LTP Retransmission timers appropriately to avoid premature retransmission. The one-way light time to the destination was given as 10 sec in our LTP link-configuration file, and the LTP Additional Anticipated Latency (AAL) was set to 15 seconds to cover encoding/decoding processing delays and job-completion-inference delay (the last two addition terms in the RHS of the TC-Eqn).
Thus the LTP retransmission timers were always set to 20+15=35 seconds. Note that for a specific deep-space channel, one may interpret our performance results in units of TC (say $x$ TCs), in actual time units as follows: Let the actual deep-space channel RTT be $RTT_{ds}$; calculate the actual time units ($t_{TC}$) corresponding to one TC by replacing $RTT_{emul}$ by $RTT_{ds}$ in the $TC-Eqn$, and replacing the last two addition terms of $TC-Eqn$ with the actual LTP AAL value used to cover these: 15 sec; the performance in actual time units is given by the product $x.t_{TC}$.

Another important configuration parameter is the Datalink MTU. We are operating over the UDP—IP—Ethernet datalink; the UDP and IP MTU are 65535 bytes, and the Ethernet MTU is 1500 bytes. To avoid additional performance overhead due to IP fragmentation and reassembly, we are restricted based on the Ethernet MTU to 1500 - 20 (IP header) - 8 (UDP header) = 1472 bytes. But, our Reed-Solomon code implementation supports code-blocks of 255 bytes each, and for the sake of simplicity we encode just one code-block per packet. This implies that our LTP visible MTU is restricted to 219 bytes, which with 4 more bytes for the CRC gives the 223 byte Reed-Solomon input block in the 255 byte Reed-Solomon code-block. For the sake of fairness, we presented the same LTP visible datalink MTU of 219 bytes for other PP mechanisms (convolutional codes and Raptor codes) as well, though they have no similar block-size limits. We retained the same LTP MTU for the mixed Reed-Solomon(inner)-and-Convolutional(outer) coding strategy: the 255 byte Reed-Solomon block was then convolutionally encoded. But the LTP visible MTU for the
mixed Raptor(inner)-and-Reed-Solomon(outer) coding strategy was set to 219-8=211 bytes, since the 8 byte space was needed for the Raptor PP mechanism-specific header. Note that the LTP-visible datalink MTU is the total space available to the LTP layer for storing a full-size LTP segment: LTP headers and LTP Application data.

We now report the results of our first set of experiments, performed while emulating uniform bit-error rate of $5 \times 10^{-3}$.

### 3.4.1 Uniform BER $5 \times 10^{-3}$

The performance of various PP Mechanisms were studied for transferring files of size 100 KB. Each data-point represents an average of 25 trials, with 99% confidence-interval on the mean. Note that these criteria are applicable to the way other sets of experimental results are presented as well.

The PP performance plot is given in Figure 3.4 and the corresponding PP overhead is given in Figure 3.5.

We first observe that our Datalink-visible packet size is 255 Bytes, which is approximately 2000 bits. At the channel error-rate of 5 bit-errors every 1000 bits, on an average a packet would have approximately 10 bit errors; thus there is a very high probability that almost all transmitted packets would have at least a single bit in error. Without some form of FEC, i.e., for example if we were operating plain LTP without any PP mechanism, no data would get through.
Figure 3.4: PP Mechanism Performance: Uniform BER $5 \times 10^{-3}$

Figure 3.5: PP Mechanism Overhead: Uniform BER $5 \times 10^{-3}$
Now, we note from the performance plot that the Conv 1:2 (Convolutional coding) mechanism is powerful enough to correct all errored-packets at the decoder. At the Data receiver, all packets in error are corrected and all the encapsulated LTP segments are recovered and presented to the LTP layer. The LTP session ends when the LTP Reception-Report indicating complete reception is received successfully at the Data Sender, prompting the issue of a Report-Acknowledgment for this, which is received successfully at the Data receiver as well. Thus the strong error-correction capability of Conv 1:2 lets the LTP session complete in a single TC.

Now, we seem to be operating near the threshold of the R-S (Reed-Solomon coding) mechanism’s error-correction capacity, however. Note that R-S codes can correct 16 byte-errors in the packet, and on average we expect to take 10 bit-errors (which in the worst-case may also mean 10 byte-errors when the errors occur on 10 distinct bytes) on a packet. We seem to occasionally shoot above the average and overwhelm the error-correction capacity on a few PP packets. This results in few discarded packets, and thus few lost LTP segments. Hence, we never finish in a single TC, and one or two LTP retransmission cycles are required before the receiver successfully receives all data. This causes the average value of TCs to be 2.32.

The inner R-S and outer Convolutional codes (R-S; Conv 1:2) are powerful enough to correct all packets in error as well. But the presence of inner R-S codes and the extra overhead it entails seem redundant, since the same best-case performance is achieved with plain Convolutional codes (Conv 1:2) as well.
The interesting results from this plot are for the different instances of the inner-Raptor-and outer-Reed-Solomon PP mechanism (Raptor $X.Y \& R-S$). Note that the plain Raptor mechanism will not work because without some form of packet-level FEC, packets with errors would be discarded to preserve the erasure channel notion (which in this error-setting would amount to discarding almost all packets). The $X.Y$ value gives the average Raptor-coding overhead per input symbol; $K$ original input symbols are Raptor-coded to give $N$ coded symbols, where $N$ is given by the product of $K$ and $X.Y$. The value of 1.1 implies 10% overhead, 1.2 implies 20% overhead, and so on. Since our maximal Raptor input job size was configured to be 100 input symbols, $X.Y$ values of 1.1, 1.2, 1.3, correspond to the generation of 110, 120, 130 encoded symbols, respectively, from it. Each of the Raptor encoded symbols are then R-S coded to give the final coded symbols. We note from the performance plot that with this mechanism we can bring the average number of TCs down significantly, to 1.12, 1.04, and 1 TCs for Raptor-overhead of 1.1, 1.2, and 1.3 respectively. Although the occasional R-S coded packet overwhelms the R-S error-correction capacity and is discarded, the original 100 Raptor input-symbols (for a maximal-size Raptor job) are still recovered from the Raptor encoded symbols obtained from those R-S code-blocks which had the number of errors within the R-S error-correction threshold (and hence, were corrected). Generating more Raptor-coded symbols at the Data Sender (130 for a full-size job with the Raptor-overhead of 1.3 seems sufficient) means that despite losing the packets discarded due to R-S decoding failure, we still have sufficient
number of Raptor-encoded symbols \((100 \leq N' \leq 130\) for Raptor overhead of 1.3) to allow successful Raptor-decoding. Thus we see the performance of 1 TC for \(Rptr 1.3 \& R-S\).

The overhead plot illustrates the channel overhead incurred with each of the PP mechanisms. The PP overhead for an LTP session is defined as the ratio of the total size of all PP packets (generated from all the LTP data segments presented to the PP layer), to the total size of all LTP data segments (headers + data) presented to the PP layer (including the segments sent as part of both the original transmission and any retransmissions). In other words, it is the ratio \(\frac{y}{x}\), where \(x\) is the net size of LTP data segments entering the PP layer, and \(y\) is the net size of all resultant PP packets leaving the PP layer.

We notice that \(R-S\) takes the least PP overhead of 1.22, whereas \(Conv 1:2\), and \(R-S \& Conv 1:2\) mechanisms take larger overheads of 2.1 and 2.4 respectively. We notice that with the lesser overhead of 1.64, \(Rptr 1.3 \& R-S\) offers equivalent performance to both \(Conv 1:2\), and \(R-S \& Conv 1:2\), and thus presents itself as a better alternative to them. \(Rptr 1.2 \& R-S\), and \(Rptr 1.1 \& R-S\) take lesser overhead than \(Rptr 1.3 \& R-S\).

Assuming that 100 KB is a representative size of file transfer, and that the deep-space channel exhibits the error-characteristic emulated, we can make the following Recommendations for the channel.
Recommendations

- Rptr 1.3 & R-S is a better alternative than Conv 1:2 and R-S & Conv 1:2, and is a good mechanism for jobs with high Immediacy requirements.

- R-S offers the least overhead (amongst the PP mechanisms we study), and hence is a good default PP mechanism for jobs.

- Rptr 1.2 & R-S, and Rptr 1.1 & R-S may be applied for jobs with medium levels of Immediacy.

As a future work, it might be interesting to study Rptr 1.05; R-S.

3.4.2 Uniform BER 0.01

The uniform BER of 0.01 was the highest bit-error rate tested in our experiments. Bit error probability of 1 every 100 bits, implies approximately 20 errored bits in our 2000 bit packet on average. This error-probability exceeds the error-correction threshold of Reed-Solomon codes, and thus both the R-S, and Rptr X.Y; R-S mechanisms are moot. We are now left with the remaining two mechanisms: Conv 1:2 and R-S & Conv 1:2. The PP performance and the PP overhead plots are given in Figure 3.6 and Figure 3.7.

We infer from the plots that we are operating slightly above the error-correcting boundary of the two mechanisms. Occasionally, Conv 1:2 fails, resulting in a few lost PP packets, which then require an additional LTP TC for retransmissions. This
Figure 3.6: PP Mechanism Performance: Uniform BER 0.01

Figure 3.7: PP Mechanism Overhead: Uniform BER 0.01
is not a very frequent occurrence though, as is illustrated by the close-to-1 average number of TCs: 1.16.

Adding in inner R-S codes to recover PP packets when the outer Convolutional codes fail on them, does help, as is observed from the smaller average TC value of 1.08. But even now, though less frequently, the decoder fails on PP packets (both the inner and outer codes fail to correct errors), and needs an additional LTP retransmission cycle.

The overhead plots illustrate the same behaviour as illustrated earlier with Uniform BER $5 \times 10^{-3}$: Conv 1:2 has the PP overhead of 2.1, while R-S & Conv 1:2 takes the PP overhead of 2.4.

**Recommendations**

We recommend Conv 1:2 as the default PP mechanism, and R-S & Conv 1:2 as the mechanism for jobs with high immediacy.

We expect the much lower code-rate Convolutional codes, rate $\frac{1}{6}$ and constraint length $m = 15$ to be sufficient to cover this error-rate probability. As a future work, we need to study this as a PP Mechanism.

**3.4.3 Uniform BER 1x10$^{-3}$**

The PP performance and the PP overhead plots are given in Figure 3.8 and Figure 3.9.
Figure 3.8: PP Mechanism Performance: Uniform BER $1 \times 10^{-3}$

Figure 3.9: PP Mechanism Overhead: Uniform BER $1 \times 10^{-3}$
This bit-error rate of 1 in every 1000, is on average 2 bit errors per PP packet. This is still a high enough BER that some packet-level FEC mechanism is necessary, since otherwise, almost all packets transmitted would be discarded due to errors. The error-rate is well within the R-S mechanism’s error-correction threshold, and thus the mechanism helps LTP sessions complete in 1 TC. The corresponding PP overhead for the mechanism is 1.22.

Since Conv 1:2 was shown to perform in 1 TC in the higher BER of 5x10^{-3}, it must perform in 1 TC here as well. However, it takes a much larger PP overhead of 2.1, and thus it is effectively replaced by R-S. Since we get optimal performance from R-S codes themselves, double-FEC schemes incorporating them, namely R-S; Conv 1:2 and Rptr X.Y & R-S, provide excessive FEC performance, that is only wasted overhead here.

Recommendations

We recommend the R-S PP Mechanism as both the default and optimal choice.

3.4.4 Uniform BER 1x10^{-4}

We have 1 in 10,000 bits in error. We now are in an error-domain where a reasonable fraction of packets get through without any bit-errors. With our packet size of 2000 bits, the expected packet error-rate is approximately $\frac{1}{5}$, and 4 out of 5 packets are expected to be received without any errors. Hence we can study the NO FEC PP
Mechanism, i.e., the trivial PP Mechanism adding no FEC capability, and Raptor codes, which does not require packet-level error-correction capability.

The PP performance and the PP overhead plots are given in Figure 3.10 and Figure 3.11.

The performance graphs also include the RTO (Retransmission TimeOut) parameter. This is the average number of LTP Retransmission Timeout events observed, also with a 99% confidence interval on its mean. We recall that, at the LTP Data Sender, the feedback-loop of an LTP Transmission Cycle is closed by the arrival of the Reception Report generated by the LTP Data Receiver upon reception of the Checkpoint Data Segment (Refer Section 2.3.1). If the Checkpoint Data segment, or one or more of the Report segments belonging to a Reception Report are lost (the
PP packets encapsulating these LTP segments are discarded due to errors), an RTO event occurs when the corresponding retransmission timer expires. The loss of an LTP CP segment or an LTP Report segment therefore causes the associated TC to drain, and adds another TC when the associated segment gets retransmitted.

Thus the net performance of a PP mechanism is actually given by the total number of TCs $\tau$ such that

$\tau = x + y$, where $x$ is original TCs observed, and $y$ is the number of TCs resulting from RTO events. Note that in the sets of experiments presented so far (Uniform BERs $5 \times 10^{-3}$, 0.01, and $1 \times 10^{-3}$), the probability of RTO events was very low (due to the availability of some form of packet-level FEC); we noticed no RTO events in them, and hence, the associated plots did not have the RTO values illustrated.
Using the retransmission capability of LTP alone for loss-recovery (*NO FEC*), we get a performance of approximately 5 TCs (4.24 original TCs + 0.72 RTO events, on average). Raptor Codes with overheads 1.1, and 1.2 help reduce the original number of TCs, but the number of RTO events is higher than even the amount seen in *NO FEC*, and thus the net performance $\tau$ is poorer than *NO FEC*. With Raptor-codes, *Rptr 1.1* for example, we generated 110 symbols from 100 input symbols; but with a packet error probability of $\frac{1}{5}$, the receiver is expected to receive, on average, only 88 symbols from the 110 symbols, and Raptor-decoding fails because at least the minimum number of 100 input-symbols are required to even attempt decoding. Similar argument holds for *Rptr 1.2*, where we are expected to receive, on average, only 96 symbols from the 120 symbols. Thus in both cases, raptor-decoding fails to recover all 100 original input symbols. However, since the codes are systematic, the 100 original input symbols constitute the first 100 of the 110 (*Rptr 1.1*) or 120 (*Rptr 1.2*) encoded symbols, and partial raptor-decoding success can be achieved by using the original input symbols that were received error-free. However, we observe that the probability of losing the Checkpoint segment or the Report segment is higher when we run Raptor codes *Rptr 1.1*, and *Rptr 1.2*, and the number of RTO events are higher with them than even *No FEC*. This is because in the *No FEC* case, the CP segment, being always the last segment of a job, may be of arbitrary size between the minimal size containing the headers and just 1 byte of data, to the maximal LTP MTU size; on average this is only half the LTP MTU. With Raptor codes, all input
symbols need to be of the full segment size (to be of the fixed Raptor symbol size), and hence all CP segments are padded with zeroes to be full-size Raptor-symbols (Note that LTP headers are not modified; the length field in the LTP header still covers just the exact amount of LTP application data present; the padding is only in the PP layer). Since the probability of losing a packet increases with size, we notice the regular packet drop-rate (based on full packet size) to hold for CP segments sent in Raptor codes, while this rate is serendipitously lesser with the NO FEC case. This issue is only exacerbated for Report Segments, which may be even smaller when there is less to report (when almost all data are received, for example), and the probability of losing a Report Segment becomes even lesser in the No FEC case compared to Raptor codes. Thus the number of RTOs are significantly lesser with No FEC than Rptr 1.1 or Rptr 1.2.

Raptor codes with overhead 1.3, perform well however, and improve the total performance $\tau$ to approximately 1.7 TCs.

We observe from the overhead plot however, that all of our Raptor PP Mechanism perform with higher overhead than that offered by R-S codes, 1.22, which we know from its performance on the higher channel error-rate of $1 \times 10^{-3}$ to perform optimally at 1 TC.
Recommendations

Jobs with low immediacy may be sent without any FEC (the NO-FEC PP Mechanism), while all other jobs may be sent encoded in R-S codes.

3.4.5 Uniform BER $1 \times 10^{-5}$

The PP performance and the PP overhead plots are given in Figure 3.12 and Figure 3.13.

Here we tested the following three PP Mechanisms: NO FEC, Rptr 1.05, and Rptr 1.10. Since the error-rate is an order of magnitude lesser than our earlier experiments with Uniform BER $1 \times 10^{-4}$, we tested Raptor-codes with lesser overheads of 1.05
Figure 3.13: PP Mechanism Overhead: Uniform BER $1 \times 10^{-5}$

and 1.10. The net performance $\tau$ is seen to be 2.2 TCs, down from the 5 TCs observed for BER $1 \times 10^{-4}$. Raptor-codes help improve performance: $Rptr\ 1.05$ has net performance of 1.5 TCs, while $Rptr\ 1.1$ yields 1.04 TCs.

The overhead plot illustrates the worth of the $Rptr\ 1.05$ PP mechanism. It takes an overhead of 1.17, higher than 1.07 ($NO\ FEC$), yet lesser than the overhead of 1.22 expected from $R-S$. Its performance $\tau$ of 1.5 TCs is in the middle of $NO-FEC$ (2.2 TCs) and $R-S$ (1 TC), as well. The overhead of $Rptr\ 1.10$, is the same as $R-S$ codes, with the value of 1.22.
Figure 3.14: PP Mechanism Performance: Uniform BER $1 \times 10^{-6}$

**Recommendations**

By default, jobs may be sent with no FEC (*NO FEC*). Jobs with medium immediacy may be sent in *Rptr 1.05*, while those with high immediacy may be sent in *R-S*.

### 3.4.6 Uniform BER $1 \times 10^{-6}$

The PP performance and the PP overhead plots are given in Figure 3.14 and Figure 3.15.

We now emulate an even lesser uniform BER value of 1 error in a million bits, and studied the same three PP Mechanisms studied earlier with Uniform BER $1 \times 10^{-5}$. 
The general relative performance of the mechanisms is similar to BER $1 \times 10^{-5}$, except that the performance of all mechanisms are better, given the order-of-magnitude lesser BER. No RTO events were observed with any of the mechanisms. The NO FEC scheme performs in 1.72 TCs on average, lesser than the 2.2 TCs observed with BER $1 \times 10^{-5}$. This illustrates that we are operating in a relatively low error-regime where it takes at most 2 TCs even when No FECs are applied. Applying Rptr 1.05, improves the performance, bringing the number of TCs down to 1.12 (note that this is better than the 1.5 TCs offered by the mechanism in BER $1 \times 10^{-5}$). Rptr 1.10 offers absolutely optimal performance of 1 TC (from the closer, but not fully optimal value of 1.04 TCs observed in BER $1 \times 10^{-5}$).

The overhead plot illustrates the same results obtained in BER $1 \times 10^{-5}$.
Recommendations

Our recommendations are similar to those from the BER 1x10^-5 experiments. By default, jobs may be sent with no FEC (NO FEC). Jobs with medium immediacy may be sent in $Rptr\ 1.05$, while those with high immediacy may be sent in either $R-S$ or $Rptr\ 1.10$ (since we noticed that they both offer optimal performance with equivalent overhead).

3.4.7 Burst Error Channels

The error characteristics we have studied so far are representative of random thermal noise, ever-present on the channel. We now present our study of burst-error channels.

Errors due to events such as solar flares and Coronal Mass Ejections from the Sun and the prevailing atmospheric weather conditions on Earth such as clouds, rain, and thunder-storms, are generally more time-variant, lasting only for a specific duration depending on the nature of the event. These errors occur as bursts on the channel, and are typically modeled with a two-state Markov Chain model called the Gilbert-Elliot model, as illustrated in Figure 3.16.

The channel is considered to be in the GOOD state normally, with a bit-error probability given by $\phi_{good}$. The probability of staying in the GOOD state is given by $p_{gg}$. The probability of beginning a new burst-error event is given by the probability of switching from the GOOD state to the BAD state ($p_{gb}$). The probability of staying
in the burst-error event, i.e., the probability of staying in the BAD state, is given by $p_{bb}$. The bit error probability in the BAD state is given by $\phi_{bad}$. The probability of reaching the end of the burst-error event is given by the probability of returning back to the GOOD state, $p_{bg}$. Note that $p_{gg} + p_{gb} = 1$, $p_{bb} + p_{bg} = 1$, and $\phi_{bad} \gg \phi_{good}$.

Note that within a state: GOOD or BAD, errors are assumed to occur independently.

The paper [Sun06] uses the Gilbert-Elliot model to study the effects of weather on the performance of CFDP (CCSDS File Delivery Protocol) over Ka-band channel. The Ka-band channel operates in the 32 GHz frequency band, which is of a higher radio-wave frequency compared to the more-commonly used X-band channel that operates at 8.4 GHz. The larger operating frequency allows for a larger downlink data rate, but also makes the channel more susceptible to atmospheric conditions such as cloud-cover, rain, etc. at the receiving antenna site [Sun06]. This paper uses the weather statistics from the Deep-Space Network antenna site at Madrid, Spain, and uses the Gilbert-Elliot model to characterize burst-errors on the channel. The system

![Gilbert-Elliot Channel Model](image)
noise temperature data from the antenna-site is divided into time-slots of 40 minute
duration each (maximal round-trip duration to Mars), and the channel is deemed to be
in the GOOD state if the average receiving-system noise-temperature from a time-slot
is lesser than or equal to 20 K, and in the BAD state if it is greater than 20 K. The
paper also considers the minimum and maximum system noise-temperatures from
each time-slot to construct two different Gilbert-Elliot channel models to bound the
performance of the model using the average noise-temperature. We use their Gilbert-
Elliot model based on the average noise-temperature per time-slot in our work here.
The paper suggests a set of possible values for $\phi_{\text{good}}$: $10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}$ and $\phi_{\text{bad}}$: $10^{-4}, 10^{-3}$.

We modified our error-emulator `errorem` to use the Gilbert-Elliot model to em-
ulate burst-errors, and used it to study the performance of our PP Mechanisms.
We applied the following model parameter values cited in the paper: $p_{gg} = 0.9773$,
$p_{gb} = 0.0227$; $p_{bb} = 0.8333$, $p_{bg} = 0.1667$. We then performed two different experiments:
the first with $\{\phi_{\text{good}} = 10^{-5}; \phi_{\text{bad}} = 10^{-3}\}$, and the second with $\{\phi_{\text{good}} = 10^{-6}; \phi_{\text{bad}} = 10^{-4}\}$. Since we emulated an RTT of 20 seconds in our experiments, we configured `errorem` to choose the GOOD/BAD state-transitions every 20 seconds, to remain
true to the assumption on the time between state-transition events made in the paper.
Thus, our experimental results are indicative of the performance of PP mechanisms
on the Ka-band channel with 40 min RTT.
Experimental results for the first set \( \{ \phi_{\text{good}} = 10^{-5}; \phi_{\text{bad}} = 10^{-3} \} \), are given in Figures 3.17, and 3.18.

As we would expect, the PP performance plots indicate that the performance results are upper-bounded by the performance from the Uniform BER \( 10^{-5} \) tests, and lower-bounded by the performance from the Uniform BER \( 10^{-3} \) tests. For example, the \textit{NO FEC} mechanism was not viable with the Uniform BER \( 10^{-3} \) channel, because almost all packets would get errored with high probability and an FEC mechanism was necessary. But here, since the channel gets into Uniform BER \( 10^{-3} \) only in the BAD state, which tends to last for relatively short periods, the \textit{NO FEC} mechanism becomes viable, and its performance (TC=2.28) is comparable to its performance with the Uniform BER \( 10^{-5} \) case (TC=2.08). Reed-Solomon codes give the optimal
Recommendations Our recommendations are the same as with the Uniform BER $10^{-5}$ tests: Jobs with lower immediacy, and any intrinsic value requirement may be sent with the No FEC mechanism. Those with medium immediacy may be sent in Rptr 1.05, while jobs with high immediacy may be sent in R-S.

Experimental results for the second set $\{\phi_{good} = 10^{-6}; \phi_{bad} = 10^{-4}\}$, are given in Figures 3.19, and 3.20.

The results from this set, are analogously upper-bounded by the performance at Uniform BER $10^{-6}$, and lower-bounded by the performance at Uniform BER $10^{-4}$. The relative performance of PP mechanisms is in general similar to the Uniform BER $10^{-5}$. Raptor codes (Rptr 1.05) help improve the performance to (TC=1.6) with lesser overhead (1.17) than R-S (1.22).
Figure 3.19: PP Mechanism Performance, Burst Errors: $\phi_{\text{good}} = 10^{-6}; \phi_{\text{bad}} = 10^{-4}$

Figure 3.20: PP Mechanism Overhead, Burst Errors: $\phi_{\text{good}} = 10^{-6}; \phi_{\text{bad}} = 10^{-4}$
BER $10^{-6}$ tests. The *NO FEC* mechanism performs at TC=1.92 compared to the performance of TC=1.72 in the Uniform BER $10^{-6}$ tests. Similarly, the *Rptr 1.05* mechanism performs at TC=1.4 compared to the performance of TC=1.12 from the Uniform BER $10^{-6}$ tests. *Rptr 1.10* performs with TC=1.28, compared to the optimal TC=1 performance from the Uniform BER $10^{-6}$ tests.

**Recommendations** Here we make the same recommendations as with the previous burst-error test: Jobs with lower immediacy requirements may be sent with the *No FEC* mechanism. Jobs with medium immediacy may be sent in *Rptr 1.05*, while jobs with high immediacy may be sent in *R-S*.

**Discussion**

The burst error study we have presented does not corroborate with our intuitive sense of error-bursts very much. For example, even in the BAD state with $\phi_{BAD} = 10^{-3}$, the chances of getting many contiguous bits in error is meager; since errors are expected to occur independently, the chances of even 3 contiguous bits in error is a low $(10^{-3})^3 = 10^{-9}$. Thus, our tests so far may not have reached the burst-error correction limits of the PP Mechanisms.

It is possible that solar flares and causes other than the atmospheric effects of Earth might present a more interesting burst-error model to study. However we were unable to find any such model characterizing the effects of such events on the deep-space communication channel. Anecdotal evidence suggests that this may be so
because, solar flares and similar burst-error events tend to be observed or predicted before-hand and as a safety precaution communication systems are normally shut-down in these periods. Thus there is probably a lack of study of communication channels under solar-flares and similar burst-error events, resulting in the lack of models characterizing their effects on the communication channel.

However, we resorted to some off-line testing to see the burst-error correction limits of various PP mechanisms. We take one packet’s (256 bytes to remain comparable to our other experiments) worth of random data, and encode it with a specific PP mechanism. Then we corrupt contiguous bit-sequences of various lengths in the encoded buffer, and attempt decoding. We start with a burst-length of 1 and iterate the process, increasing burst-lengths until we reach the length where decoding fails. We repeated the tests for bursts beginning at various byte-offsets in the encoded buffer, such as 0, 10, 25, 50, 100, 200, etc.

- **Conv 1:2** This PP mechanism corrects burst at most 5 bits long.

- **R-S**: (255, 223) R-S codes, we know, can correct at most 16 symbol errors. Since the symbol size is 1 octet, our R-S code implementation can correct bursts at most 16 octets (128 bits) long within a single 255 byte R-S code-block.

- **R-S & Conv 1:2**: Here we observe that bursts in the range 220 to 250 bits in length are correctable. Although the outer convolutional codes do not correct all the burst-errors added, they do correct some of them, and reduce the number
of remaining errors to within the threshold of the number of errored-symbols correctable by the inner Reed-Solomon codes. Thus the burst-error correction limits are higher with this mechanism.

- *Rptr* We know that Raptor codes operate over an erasure channel: an encoded symbol is discarded even if it has a single bit in error. It does not matter if the burst is one bit long or the packet length in bits, the penalty is the same: the encoded symbol (in our implementation - the packet itself) is discarded. However, keeping in mind that Raptor codes encode groups of packets at a time, we note that the burst-error length correctable is conceptually infinite, provided that the inter-burst arrival time is relatively high. For example, let us say that we have bursts of full packet length (256 bytes in our implementation), and these bursts occur once every 100 packets; *Rptr 1.10* would generate 110 encoded packets from every 100 data packets, of which at most 2 packets are expected to be corrupted. This still leaves the Raptor decoder with 108 packets, which is generally sufficient to decode back the 100 original symbols with high probability of success. As a further example, if we had bursts of length 256 bytes long each, and even if the inter-burst arrival time was one in every 10 packets, we could apply *Rptr 1.20* to generate 120 encoded packets for every 100 data packets of which 12 would be expected to be lost to burst errors, leaving the Raptor decoder with 108 encoded packets to guarantee decoding success with high probability. Thus, as long as we are willing to expend additional
bandwidth, any given burst-error characteristic can be corrected with Raptor codes.

**Signaling Issues**

Note that the generic PP header needs to be recovered error-free, or the entire packet is discarded. Our current implementation encodes the generic PP header with \( \text{Conv 1:2} \) to generate a fixed 20 byte (160 bit) header for all packets, and as we noted earlier that \( \text{Conv 1:2} \) can correct burst at most 5 bits long. We mentioned earlier in our description of the generic PP header that the conservative strategy to error-protect the header is to provide an FEC scheme that is robust to the worst-case error expected on the channel. Therefore, it is important to consider other FEC schemes such as inner R-S codes (31, 25) and outer Convolutional codes (\( \text{Conv 1:2} \)), or stronger Convolutional codes such as \( \text{Conv 1:6} \) to encode the generic PP header for large burst-error rates.

### 3.5 Performance Comparison

We illustrate the performance comparison between our Priority Paradigm based adaptive FEC strategy, and the contemporary, conservative fixed FEC strategy, with a hypothetical set of jobs. We assume that we have 10 jobs with each job comprising 100 KB of application data. The jobs are assumed to fall into one of the following four priority categories:
• Category A: High Intrinsic Value, High Immediacy

• Category B: Low Intrinsic Value, High Immediacy

• Category C: High Intrinsic Value, Low Immediacy

• Category D: Low Intrinsic Value, Low Immediacy

Of our 10 jobs, we assume that 4 are Category D, 3 are Category C, 2 are Category B, and 1 is Category A.

**Uniform BER 5x10^{-3}**

We now calculate the data-bandwidth consumption of the two strategies, for the Uniform BER 5x10^{-3}. We presented the PP performance results and recommendations for this channel in Section 3.4.1. Let us assume that we use the $Rptr\ 1.3 \& R-S$, $Rptr\ 1.2 \& R-S$, $Rptr\ 1.1 \& R-S$, and $R-S$ mechanisms for jobs in Category A, B, C, and D respectively. We know [Figure 3.4] that the average number of TCs expected for Categories A, B, C, and D are 1, 1.04, 1.12, and 2.32 respectively, and that the average overhead (Figure 3.5) for the corresponding PP mechanisms are 1.64, 1.52, 1.39, and 1.22 respectively. Note that the PP mechanisms applied for priority categories B, C, and D (as indicated by their greater-than-1 TC average) take one or two more LTP TCs to retransmit lost-data. We take this into account by assuming that two equivalent sets of jobs (each set comprising of 10 jobs - each 100 KB in size, and having the same overall priority requirements as the current set of jobs) were
Table 3.1: Bandwidth Consumption (in bytes) by TC, Uniform BER $5 \times 10^{-3}$

<table>
<thead>
<tr>
<th>TC</th>
<th>R-S</th>
<th>$Rptr\ 1.10\ &amp;\ R-S$</th>
<th>$Rptr\ 1.20\ &amp;\ R-S$</th>
<th>$Rptr\ 1.30\ &amp;\ R-S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>137,975</td>
<td>158,267</td>
<td>172,706</td>
<td>187,039</td>
</tr>
<tr>
<td>2</td>
<td>3,812</td>
<td>182</td>
<td>53</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>214</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

presented to the PP layer in the time units corresponding to the past two TCs, and add the data sent for the jobs from these recent sets to the current calculations.

The total average number of PP bytes transferred for each job, based on the PP mechanism used (including the application data, LTP headers and PP headers) per TC is shown in Table 3.1.

The net bandwidth consumption for our PP model in the current TC is the sum of:

- Category-D jobs from the current set, and two most recent sets:
  \[ 4 \times 137,975 + 4 \times 3,812 + 4 \times 214 = 568,004 \]

- Category-C jobs from the current set and the most recent set:
  \[ 3 \times 158,267 + 3 \times 182 = 475,347 \]

- Category-B jobs from the current set and the most recent set:
  \[ 2 \times 172,706 + 2 \times 53 = 345,518 \]

- Category-A job from the current set:
  \[ 1 \times 187,039 = 187,039 \]
and is found to be 1,575,908 bytes.

The conservative fixed FEC model would use the strongest FEC scheme amongst our choices: $Rptr \ 1.3 \ & \ R-S$ for all jobs. The total bandwidth consumed is $187,039 \times 10 = 1,870,390$ bytes. However, the 20 byte generic PP header present in all packets to primarily indicate the PP mechanism in use, is not required in this case. On an average, 700 packets were transmitted with the $Rptr \ 1.3 \ & \ R-S$ PP mechanism. Deducting 20 bytes for each packet, for 10 jobs, we deduct, $20 \times 700 \times 10 = 140,000$ bytes, and get the net bandwidth consumption of $1,870,390 - 140,000 = 1,730,390$ bytes.

Thus using the PP approach, saves us $1,730,390 - 1,575,908 = 154,482$ bytes, corresponding to a comparative bandwidth savings of approximately 9%. Note that this relative-bandwidth savings may be used for transmitting another full Category-D job, or about 80% of a Category-A job.

**Uniform BER $10^{-5}$**

We apply the same set of 10 jobs, for the Uniform BER $10^{-5}$ channel. Let us assume that we apply the PP Mechanism $No \ FEC$ for category D, $Rptr \ 1.05$ for categories B, and C, and $R-S$ for category A.

The bandwidth consumption per TC for each of the PP mechanisms is given in Table 3.2. Repeating our calculations as we did for the Uniform BER $5 \times 10^{-3}$, we find that the total bandwidth consumption for our PP based adaptive strategy
was 1,295,570 bytes. The bandwidth consumption for the conservative fixed-priority scheme, always using the $R-S$ PP mechanism is 1,379,110 bytes, which after deducting the overhead due to generic PP headers (20 bytes of header overhead, with 630 PP packets for each job, and 10 such jobs), 126,000 bytes reduces to 1,253,110 bytes! The savings obtained by using the PP mechanism are offset by the overhead necessary for the PP mechanism itself (the generic PP headers).

The above result is of course tied to the job mix chosen by us. As an extreme case, considering the following job-mix of 10 jobs, each carrying 100 KB of application data, such that 9 jobs are of category D and 1 job is of category A, we note that the bandwidth consumption from our PP approach would be 1,249,690 bytes. Although the savings of 3,420 bytes gained with the PP approach is meager, we at least break-even and do slightly better than the fixed-priority approach.

**Uniform BER $10^{-5}$**

We recommended that only one of our PP mechanisms - $R-S$, which gives the optimal 1-TC performance, to be used in this case. All other PP mechanisms from our suite give poorer, or equivalent performance compared to $R-S$, and consume more

<table>
<thead>
<tr>
<th>TC</th>
<th>No FEC</th>
<th>$Rptr\ 1.05$</th>
<th>$R-S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121,290</td>
<td>131,959</td>
<td>137,911</td>
</tr>
<tr>
<td>2</td>
<td>2,221</td>
<td>748</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.2: Bandwidth Consumption (in bytes) by TC, Uniform BER $10^{-5}$
PP overhead than the $R$-$S$ mechanism’s overhead. Therefore there are no advantages to using them.

Since we have only one PP mechanism to use, the PP based job dispatching only adds overhead due to PP headers.
CHAPTER 4

Conclusions

In this closing chapter, we present our conclusions and give recommendations for future work.

4.1 Protocol Study

We find that, as expected, LTP performs better than TCP, and becomes more and more appealing for larger delay and/or larger error channels. This is to be expected because TCP’s innate assumptions on the end-to-end path it operates in, such as short communication latency, low probability of channel errors, etc., fail to hold in the space channel; LTP, on the other hand, was designed with the space channel characteristics in mind.

TCP is an end-to-end protocol. For optimal performance, it expects a certain degree of homogeneity from the network-hops constituting the end-to-end path. If one or more network-hops are significantly different from the rest of the end-to-end path (in their delay, errors, etc.), TCP performance suffers.

Let us consider an end-to-end path in the Internet that takes a geo-synchronous satellite hop; this adds a delay of approximately 500 msec to the RTT (116+116 msec up and down the satellite in the forward path, and the same in the return path, plus
a small margin for processing delay), and is more prone to channel errors than in the terrestrial Internet hops. On the satellite hop, if feasible, channel errors could be mitigated by deploying an FEC scheme, and the effects of the extra delay could be mitigated with protocol optimizers such as TCP Performance Enhancing Proxies [Section 2.2]. Since the satellite-hop heterogeneity, although noticeable, is not severe, the above solutions suffice in practice.

However, when the heterogeneity becomes more severe due to higher channel error-rates and delay, as with the lunar communication channel for example, such patchwork solutions do not suffice. Optimal end-to-end performance is best achieved by deploying a heterogeneous set of Internetwork protocol stacks, where each protocol stack is optimal for the Internetwork it is operating on. We illustrate this for a lunar communication system in Figure 4.1.

Here the Bundle Protocol [SB06] serves as the internetwork overlay protocol connecting the heterogeneous internetworks: the terrestrial Internet, the long-haul Space internetwork, and the Lunar internetwork. The Bundle protocol glues disparate Internetworks just as IP glues disparate networks in the terrestrial Internet. The application uses the services of the Bundle protocol to effect transfer of its data. The Bundle protocol performs routing across internetworks, and uses the protocol layers below - together termed as the Internetwork convergence layer - to transfer data. In the terrestrial Internet, this is the TCP/IP protocol suite. In the lunar internetwork, Transport, Network, Datalink, and Physical layers comprising the convergence
Figure 4.1: Lunar Communication System: Network of Internetworks
layer may be chosen depending on the specifics of the deployment. For example, either UDP, or the recently developed Datagram Congestion Control Protocol [KHF06] may be a good choice as the Transport layer; IPv6 may be ideal as the Network layer; CCSDS Proximity-1 [Pro06] may be used as the Datalink layer for the Orbiter-Lander network hop, while WiFi or Bluetooth may be used as the Datalink layer for network hops amongst rovers, robots, and lander.

LTP, along with the Radio-Frequency Datalink, is ideally suited as the convergence layer in the long-haul space internetwork. Since communication is point-to-point between a pair of antennas, the network layer itself may be redundant here. LTP offers the necessary Transport-layer functions and provides reliable data transfer capability for the long-haul space-hop.

4.2 Priority Paradigm Study

A typical space-craft carries a host of science instruments on-board, each capable of generating data at a different maximum data rate, and the net bandwidth downlink capability is often less than the maximum data rates of all instruments put together. We conceive that there may at least be a few broad classes of jobs, such as regular spacecraft telemetry jobs, regular science data, and the class of critical telecommands / telemetry / science data. Our study was aimed at studying various PP mechanisms, and using them adaptively on a job-by-job basis to see the gain in channel utilization.
From our experiments, and the basic performance comparison for a simple job-mix [Section 3.5], we draw the following conclusions:

The gain in bandwidth usage depends upon the channel error characteristics, and the nature of the job mix (i.e., how different the priority requirements of the jobs are).

- For channels with better error characteristics (for example uniform BER $10^{-5}$ or better), there is little gain from using the PP based approach. This is because the channels are relatively good, and the need for having any FEC scheme at all, is relatively low. For channels with poorer error-characteristics (such as Uniform BER 0.005), the need for having any FEC scheme is relatively high, and larger bandwidth savings may be obtained by using a suite of FEC schemes and applying them adaptively to jobs based on their priority needs. This saved bandwidth may be used for transmitting additional jobs on the channel, and thus the *volume* of application data transmitted on the channel may be increased. If job scheduling is done judiciously, and jobs of greater value are scheduled carefully, the *value* of application data transferred may be increased as well.

- The nature of the job mix also determines how valuable the PP approach is. If all jobs are of high priority, or all jobs are of low priority, we are better off applying the same FEC scheme to all jobs on the channel. However, we note that given our assumption of the existence of at least a few broad classes of
jobs from a typical space-craft, and assuming that there may at least be certain critical time-spans in the duration of a deep-space mission, when bits on the channel are extremely expensive, the need for the PP-based handling of jobs is paramount. For example, using the PP approach during the comet-impact phase of the Deep-Impact mission, or during the 3-hour long mission-life of the Huygens probe when it descended into the atmosphere of Saturn’s moon Titan could have greatly improved the volume and value of the data received.

- The PP-based scheme needs two or more PP mechanisms (the more, the better) such that they may be used appropriately depending on the job priority policy.

**Future Work**

Since our study indicates that the PP approach tends to be more valuable for poorer channel error characteristics, it would be valuable to study error-rates poorer than Uniform BER $10^{-2}$. The PP mechanisms we studied would be inadequate in such channels. Therefore it would be valuable to implement and study the performance of Turbo Codes, and much stronger Convolutional codes (rate $\frac{1}{6}$) as PP mechanisms.

We have not studied the *value* of jobs transmitted quantitatively. Study of one or more utility functions, and their relative merits in characterizing the net utility gained from the channel is an interesting future-work item.

Finally, algorithms for scheduling the jobs such that the overall utility of the channel is maximized were not studied in our work. Under severely constrained
bandwidth channels and/or a deep queue of jobs to process, job scheduling could have an important role in maximizing the utility of the channel.


**Bibliography**


[Pro06] Proximity 1 Space Link Protocol - Data Link Layer, Recommended Standard, CCSDS 211.0-B-4, BLUE BOOK. July 2006.


APPENDIX A

Testbed Configuration

A sequence of configuration-procedures needs to be followed to initialize our GNU/Linux testbed machines as required for our experiments. We present them here for reference.

A.1 Protocols Study

The discussion here covers testbed configuration for our protocol study experiments comparing TCP versus LTP, as illustrated in the testbed setup in Figure 2.3, comprising the hosts from Testbed #1 (Table 2.1). Hosts *Ram* and *Lakshman* are the data sender and receiver respectively, while the host *Hanuman* serves as the error emulator. All three hosts have two physical ethernet interfaces on them; interface eth0 is assigned a public IP address and is used to login to the hosts remotely, whereas interface eth1 is assigned a private IP address (10.x.x.x) and is used solely for our experiments. Host-names *ram-l, lakshman-l, hanuman-l* correspond to the private IP addresses assigned to the eth1 devices of the hosts: 10.1.1.1, 10.1.1.2, 10.1.1.9 respectively, and all three hosts are part of the 10.1.1.0/24 network.
A.1.1 Configuring End-hosts

We illustrate configuration on the data sender Ram; similar procedure is followed on the data receiver Lakshman.

1. First, a permanent ARP table entry is added for hanuman-l so that ARP lookup overhead is never incurred in the middle of the experiments. If the ethernet hardware address of hanuman-l’s eth1 interface is 00:01:02:03:04:05, we give:

   ```
   sudo arp -s hanuman-l 00:01:02:03:04:05
   ```

2. Route traffic for lakshman-l via hanuman-l:

   ```
   sudo route add -host lakshman-l gw hanuman-l
   ```

3. Configure a Token-Bucket-Filter traffic control policy to emulate 1 Mbps bandwidth on eth1:

   ```
   sudo tc qdisc add dev eth1 root handle 1: tbf
   burst 12500 limit 25000 rate 1mbit
   ```

   The above command adds a new queuing discipline (qdisc) handle #1 as the root for eth1; the qdisc is of type token-bucket-filter (tbf) with a bandwidth rate emulation of 1 Mbps; the burst parameter chooses the token-bucket size as 12500 bytes, and the limit parameter sets the size of data queued awaiting tokens to be 25000 bytes. See [Hub03] for more details on configuring a tbf policy.
4. Configure the one-way propagation delay required:

```
sudo tc qdisc add dev eth1 parent 1: handle 10: netem
delay 110msec limit 160000
```

Here, we are adding the qdisc handle #10 under the parent handle #1; this handle emulates a delay of 110 msec, i.e., packets departing out of `eth1` are held for 110 msec and then transmitted on the wire. The limit parameter specifies the maximum number of packets that may be delayed this way; since any more packets would be dropped silently, we over-provisioned this limit by giving a large value of 160000 packets.

5. Set the maximum UDP, TCP socket buffer size:

```
sudo sysctl -w net.core.rmem_max=10000000
sudo sysctl -w net.core.wmem_max=10000000
```

`rmem_max` refers to the maximum socket “read” (reception) buffer size, while `wmem_max` refers to the maximum socket “write” (transmission) buffer size. We set them both to the large value of 10 MB. Note that these parameters only set the maximum value the operating system would grant to applications; an application still has to request the socket buffer size to be larger than the default size given by the operating system, if necessary.
6. Finally, making sure that the standard “reno” TCP Congestion Control algorithm is set:

```
sudo sysctl -w net.ipv4.tcp_congestion_control=reno
```

### A.1.2 Configuring the Error-emulator

Here, we illustrate our configuration procedure on the error-emulator *Hanuman*.

1. First we add permanent ARP table entries for *ram-l* and *lakshman-l*. This is similar to the ARP configuration procedure in Step 1 from Section A.1.1.

2. Create our own custom routing table *errorem-tbl* to be used along with the standard kernel routing table. First an entry for the table is added in the file `/etc/iproute2/rt_tables` as in:

```
200 errorem-tbl
```

200 is an arbitrary, but unique numeric identifier for *errorem-tbl*.

3. We then add the following rules to tell the operating system *when* this routing table must be consulted:

```
sudo ip rule add from ram-l dev eth1 table errorem-tbl
sudo ip rule add from lakshman-l dev eth1 table errorem-tbl
```

The above rules mention that if a packet is received from *ram-l* or *lakshman-l* via *eth1*, the *errorem-tbl* (and not the standard kernel routing table) must be referred to route the packet.
4. The error-emulator program `errorem` is started, which opens the `tun` device;

   Linux kernel now recognizes the presence of the `tun0` virtual network device.

5. We assign a private IP address 10.1.2.9 to the tun interface:

   ```
   sudo ifconfig tun0 10.1.2.9 netmask 255.255.255.0 up
   ```

6. Add routing entries to `errorem-tbl`:

   ```
   sudo ip route add ram-l via 10.1.2.9 dev tun0 table errorem-tbl
   sudo ip route add lakshman-l via 10.1.2.9 dev tun0 table errorem-tbl
   ```

   When a packet arrives from `ram-l` or `lakshman-l` via `eth1`, the operating system
   first notices that the rules from Step 3 apply to it, and hence looks up `errorem-
tbl`. The routes we just added to `errorem-tbl` cause the packets to be handed
   over to `tun0` for the `errorem` program to process. Note that it is important for
   the operating system to distinguish packet arrivals via `eth1` from those arrivals
   via `tun0`. Packets are originally received by the operating system via `eth1`; but
   when the error-emulator program `errorem` processes the packet and writes it
   back, the packet would be received via `tun0`. The `dev eth1` phrase in the rules
   in Step 3 causes the operating system to lookup `errorem-tbl` only for packet
   arrivals on `eth1`; in the latter case (packet arrivals on `tun0`), the standard kernel
   routing table is consulted, causing the packet to be sent out via `eth1`. 
7. We make sure that the `tun0` interface’s packet queue is long enough so that no packets are dropped:

```
sudo ifconfig tun0 txqueuelen 160000
```

### A.2 Priority Paradigm Study

The end-host configuration procedure is similar to the procedure followed in our Protocols Study, outlined in Section A.1. To configure the end-hosts, Steps 1 to 5 from Section A.1.1 are followed: Steps 1, 2, 3, and 5 apply as is, but Step 4 is replaced with the following to emulate a 10 second delay:

```
sudo tc qdisc add dev eth1 parent 1: handle 10: netem
delay 10sec limit 160000
```

The error-emulator host configuration procedure is the same as with the Protocols study, outlined in Section A.1.2.