LOSS-TOLERANT TCP (LT-TCP): ADAPTING TCP TO MODERN WIRELESS NETWORKS

By

Nathaniel Hourt

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

_____

Boleslaw Szymanski, Thesis Adviser

_____

Koushik Kar, Member

_____

Barbara Cutler, Member

_____

Bishwaroop Ganguly, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2014
(For Graduation August 2014)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

Thanks to Herbert Holzbauer for answering questions about the LT-TCP reference implementation.

Thanks to Professor Boleslaw Szymanski for being an encouraging research advisor.

Thanks to Professor Koushik Kar for being a supportive and understanding mentor.

Many thanks to Dr. Bishwaroop Ganguly for his invaluable advice and aid during development and testing.

Colossians 2:2-3

ABSTRACT

Demand for widespread, robust wireless computer networking is increasing rapidly, and unlike wired networks, wireless networks frequently drop packets due to interference or signal attenuation. To compound this issue, mobile, ad-hoc networks have no reliable structure, and routes or adjacent nodes may change without notice. The transport protocol most widely adopted for reliable network communication, TCP, does not fully utilize wireless networks where packet losses and topological changes are prevalent. To address this issue, Loss-Tolerant TCP (LT-TCP) has been implemented to provide robust and efficient communications over potentially lossy networks without assuming the presence of a static network structure.

This thesis examines the current implementation of LT-TCP and compares it with other transport layer solutions on network profiles with tunable loss and delay properties. In particular, tests were conducted with correlated losses, where contiguous groups of packets are dropped, and high latencies. Results show that LT-TCP performance is significantly higher than that of TCP on networks with correlated losses or networks with losses and high latency.

This testing identified some weaknesses in the current LT-TCP implementation, and this thesis describes the subsequent revisions to the protocol to address these weaknesses and provides experimental data demonstrating the effects of these improvements.

Finally, this thesis explores some architectural problems with the current LT-TCP implementation, which limit its portability to modern operating systems, and proposes a new architecture which will facilitate integration of LT-TCP into modern Linux-based operating systems as well as simplify the development and testing of new algorithms to support LT-TCP functionality.

Experiments have shown LT-TCP to yield throughput speedups of greather than 262 over TCP on high bandwidth networks which experience packet loss bursts of average length 5, causing 10% of total packets to be lost.

1. Introduction

Computer networks provide a rapid transit mechanism for many kinds of information, but while these networks are fast, they are not necessarily reliable. Inherent to computer networking are problems that impede reliable transfer of information, among which are the potential reordering of data packets sent over the network, the possibility that a packet may be lost in transit, and congestion on heavily-trafficked nodes within the network.

To combat these issues and provide a reliable network communication platform to application developers, the Transmission Control Protocol was created with facilities to assure complete and in-order delivery of network transmissions [24] while detecting and controlling congestion within the network [2]. To provide this reliable network platform, TCP makes some assumptions about the behavior of the underlying network links. In particular, TCP assumes that a lost packet is most likely an indication of network congestion and therefore uses packet losses as an indicator that congestion is present and the data sender should reduce the send rate to alleviate the congestion.

1.1 Motivation for LT-TCP

As demand for widely accessible, robust networks has increased, wireless networking technologies have been developed, and the introduction of these technologies invalidates the assumption that packet loss is a reliable indicator of congestion. Physical characteristics of the environment in which wireless networks are deployed may cause airborne packets to be lost although no congestion is present. This causes TCP to invoke spurious congestion control, which results in under-utilization of available network bandwidth [22].

In addition to unnecessarily applying congestion control, TCP also relies solely on retransmissions to correct for lost packets, which may take multiple round-trip times to detect. While updates to TCP, such as TCP-SACK, have improved the time required to detect losses and reduced unnecessary retransmissions [14], there is

further room for improvement by eliminating the need for retransmissions altogether if the packet loss rate can be measured with sufficient accuracy.

Loss-Tolerant TCP (LT-TCP) was created to address these weaknesses of TCP on wireless networks [27]. By utilizing Explicit Congestion Notification (ECN) [7], LT-TCP eliminates the assumption that losses are a reliable indicator of congestion and processes these events separately, enabling it to fully utilize networks where physical losses are present without neglecting congestion control. Furthermore, LT-TCP estimates the network loss rate and utilizes Forward Error Correction (FEC) packet encoding to proactively repair losses and avoid retransmissions. If the Proactive FEC (PFEC) is not sufficient for the receiver to completely recover the transmitted data, Reactive FEC (RFEC) is sent, which is analogous to TCP retransmission [27]. By using a combination of proactive and reactive FEC, LT-TCP is able to correct for losses and avoid retransmissions [28].

LT-TCP is designed as a drop-in replacement of TCP, to be implemented in the kernel using the same API that TCP uses [18], thus requiring minimal changes (if any) to applications in order to use LT-TCP instead of TCP. A kernel implementation also enables LT-TCP to accurately record timing statistics from the network and facilitates efficient implementations of the coding algorithms, which can run in the kernel with minimal overhead [6].

LT-TCP does require ECN to be deployed on networks before it can be used on them. Although ECN is defined in an RFC and supported by most major operating systems, its deployment has been somewhat slowed by network appliances such as firewalls and routers which process it incorrectly and either clear the ECN flags in packets, or drop the packets entirely; nevertheless, ECN adoption is increasing. ECN is enabled by default on Linux servers running kernel version 2.6.31 or later [19]; furthermore, as IPv6 adoption has increased, so also has ECN support [21].

1.2 Related Work

TCP is widely deployed in many application domains, and its performance limitations with regard to wireless networks are well understood [28]. In light of this, it is to be expected that other efforts to improve performance on lossy networks are

underway. Existing efforts to improve reliable communication over wireless networks generally employ one of two possible strategies: one is to make the underlying network technology more reliable (physical or data-link layer solutions), and the other is to improve or replace TCP (transport layer solutions).

Although correcting for packet losses is historically a transport layer concern, it also seems logical to address the physical loss problem at its root, in the lower layers. Efforts on this front range from simply increasing the wireless transmission power to adding redundancy or retransmissions at the data-link layer (as 802.11 WiFi does) [8], [26] to cross-layer solutions such as the Snoop protocol, which uses proxies to inject and manipulate TCP packets at the network layer [4]. While these solutions can be helpful, they are unable to completely eliminate losses apparent at the transport layer, and these solutions necessitate more complex networking hardware which increases cost and power consumption, which is a significant concern on mobile platforms, where losses are the most likely.

Since lower layer solutions cannot adequately address the issue of random packet loss, there has also been a great deal of research on how best to address these issues in the transport layer. Some solutions, such as TCP Veno [15] and TCP Westwood [9], aim to improve TCP using sender-side only changes that rely on RTT and inter-packet delays to guess at the state of the network to avoid invoking unnecessary congestion control. These solutions still make assumptions about the state of the network, and they do nothing to mitigate the losses or avoid retransmissions. Other solutions attempt to resolve the loss issue by using Forward Error Correction (FEC) coding to repair the losses, but while this improves TCP performance, it is not alone as effective as the above send-side modifications to the congestion control algorithms [5].

Combining the two methods of improving the transport layer can allow protocols to harness the advantages of both methods. This is the approach used by NACK Oriented Reliable Multicast (NORM) [1] and Network Coded TCP (CTCP) [10]. These projects both implement a reliable transport based on UDP and use FEC to mitigate losses. Both are implemented in user-space, however, and so they are not suitable as a direct replacement to TCP but instead require applications to

be rebuilt to use them. To compound this issue, NORM has a significantly different programming interface from TCP, which would make it costly to redesign existing TCP applications to use NORM instead.

CTCP has a more TCP-like programming interface, but it relies to timing information to guess whether losses are due to random errors or congestion. Timing information is not necessarily a reliable indicator of congestion (especially in user-space, where interference from other programs competing for CPU time is an issue) as it requires the underlying network to be stable, which is not a safe assumption in mobile and ad-hoc networks, where routes may change rapidly as nodes enter, exit, or move within the network.

1.3 Research Objectives

There are three objectives of the research detailed in this thesis. These objectives will be addressed respectively in the next three chapters.

Expand Scope of Performance Testing Previous performance testing of LT-TCP has been conducted in an isolated, high-bandwidth and low-latency network with variable, uniformly random losses. This thesis presents performance tests on additional network profiles, including high-latency networks, networks which group losses into continuous runs (referred to as correlated losses), and networks which mark ECN instead of dropping packets. In addition, some performance baselines are presented, including an ideal scenario where LT-TCP has oracle access to the rate of loss of the network, which gives insight as to the performance of the loss rate estimator; and tests of TCP-SACK with congestion control disabled, which provides a control case for the efficacy of LT-TCP's FEC system.

Enhance LT-TCP for Performance and Applicability The wider range of performance tests provided good performance results, but also illuminated some areas for improvement in the LT-TCP protocol. This thesis presents these issues and the revisions to the protocol to address them. The specific issues are as follows:

- Loss rate estimator yielded poor performance under correlated losses

- Connection closing algorithm was unreliable under losses
- Loss detection algorithm counted packets delivered out-of-order as lost
- RFEC scheduling could be improved due to more stable loss detection
- Packets in flight counter was inaccurately tracking network utilization

Update LT-TCP Implementation for Portability The LT-TCP reference implementation is currently a monolithic design which is implemented directly within the Linux kernel. The Linux kernel version used in the implementation is now outdated, and is not compatible with modern software; therefore, a new implementation must be created to enable testing in modern hardware and software environments. This thesis describes a new, modular implementation which will enable easier development of future versions of LT-TCP, and will also allow the implementation to be ported up to later versions of Linux with minimal effort.

2. Performance Investigation

This chapter describes the network used for performance testing of LT-TCP, the network profiles used, and the results of those tests. TCP and NORM were also tested in these same network configurations, and experimental results comparing the performance of all three protocols are provided.

In previous work, LT-TCP was tested in comparison with only TCP, and only on networks experiencing independent, uncorrelated losses. Section 2.2 presents experimental results of LT-TCP, TCP, and NORM on an expanded set of network profiles; namely, networks with bursty losses, high latencies, or ECN marking.

The reference implementation of LT-TCP uses Gentoo Linux with a modified version of Linux Kernel 2.6.26. LT-TCP is implemented alongside TCP in this kernel, allowing applications to specify which one to use via the protocol number [18]. To promote a fair comparison, all three protocols under consideration were tested on these Gentoo systems running Linux 2.6.26.

2.1 Experimental Network Configuration

The network used for testing consists of three computers on a 1Gbps ethernet LAN. There are two endpoint machines which send and receive data, and one machine used as a bridge between the endpoints which applies network conditions according to a specified configuration. All three machines are also connected to an independent control LAN, which is used to send control messages between the machines to schedule experiments and report their results. These machines are controlled by an operator machine on the LAN, as shown in Figure 2.1.

The TCP and LT-TCP protocols perform handshakes both to establish a connection before transmitting data and to terminate a connection after all data has been delivered to the application layer. To prevent packet losses during these handshakes from creating unnecessary noise in the experimental results, the connection

Portions of this chapter previously appeared as: B. Ganguly *et al.*, "Performance of Loss-Tolerant TCP (LT-TCP) in the Presence of Correlated Losses," in *Proc. MILCOM 2013, IEEE Military Commun. Conf.*, San Diego, CA, 2013, pp. 1341-1346.

and disconnection handshakes are performed without any losses applied to the network and are excluded from timing results. An experiment with losses and high latency proceeds as follows:

1. Experiment configuration is sent to bridge
2. Bridge applies high latency
3. Endpoints perform connection handshake
4. Bridge applies packet losses
5. Endpoints perform data transfer
6. Bridge disables packet losses
7. Endpoints perform disconnection handshake
8. Bridge disables high latency

Timing results are collected only during step 5 above. These results are collected at the application layer on the system receiving the data. The time result for the transfer is measured from the time the first byte is received to the time the entire transfer has been received.

The number of steps involved in each experimental transfer necessitated the construction of an automated test harness. The harness is composed of three separate programs: a graphical interface which runs on the operator's computer, a network controller daemon which runs on the bridge, and an endpoint daemon which runs on each endpoint system.

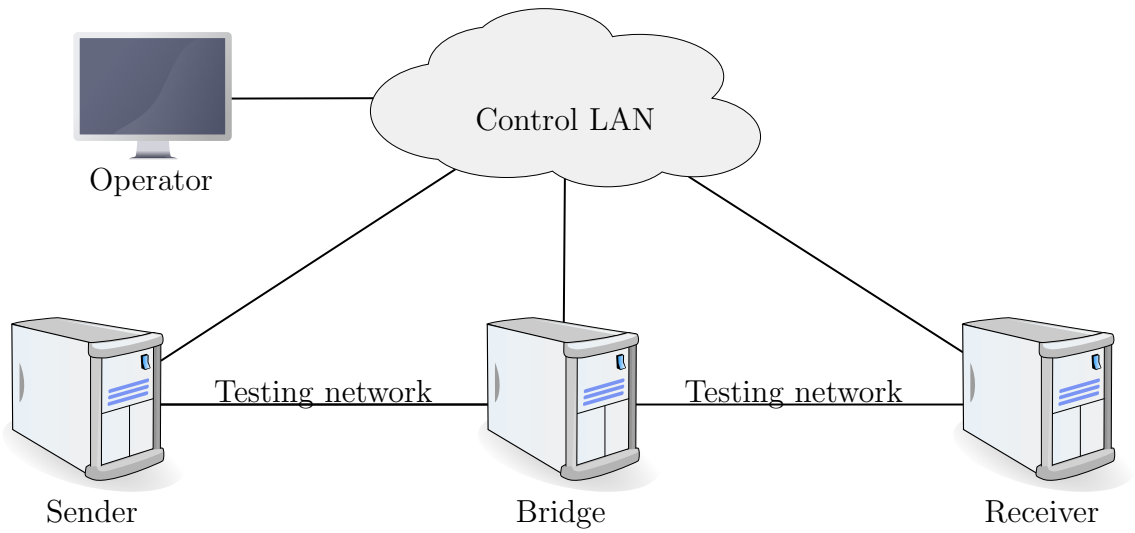

Figure 2.1: Experimental Network Diagram

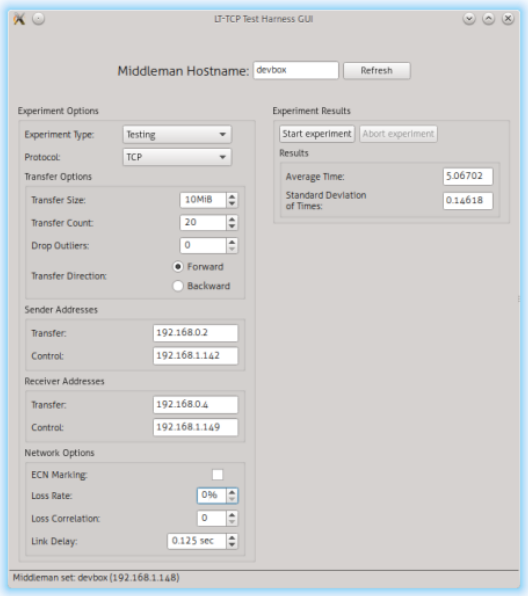

Figure 2.2: Test harness GUI in Testing mode

The majority of the logic is in the network controller daemon. It is responsible for applying the network conditions (packet loss profile, link delay, ECN marking) using Network Emulator [17], a feature of the Linux kernel which allows the application of various network conditions on packets passing through the system; configuring the endpoint daemons; coordinating transfers; and relaying the status and results of experiments back to the user interface. For greatest accuracy, the final timing results for individual transfers are generated at the endpoint receiving the data transfer, then sent to the controller daemon over the control LAN for statistical analysis. When a round of tests is complete, the results are returned to the user interface.

The user interface to the test harness does not do any processing directly; it simply informs the network controller as to what kind of experiment is desired and reports the statistics received from the controller to the user. The interface has two modes. The first is Testing mode, shown in Figure 2.2, which allows a series of individual transfers to be run and the average and standard deviation of their times to be reported back. The other mode is Demonstration, shown in Figure 2.3, which requires the user to provide an image to be transferred once. As the image is transferred, a real-time visualization of the partially received image is displayed.

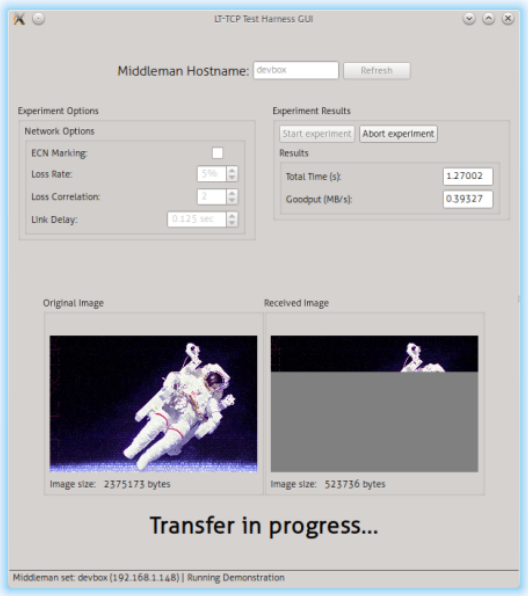

Figure 2.3: Test harness GUI in Demonstration mode

The test harness supports performing transfers over various network profiles based on tuneable packet loss, ECN marking, and/or delay options. Symmetric delays can be applied to the network such that packets moving over the test network in either direction between endpoints are delayed by a specified time period. The delay profile can be applied alongside either packet loss or ECN marking, but due to limitations in Network Emulator, a single bridge system cannot effect both packet losses and ECN marking on the network simultaneously. A loss profile can be applied and set to either drop packets or mark them with ECN, but not both.

The test harness allows one of two loss profiles to be applied to the network. The first is a simple uniform loss actuator using a configurable loss probability which performs a Bernoulli trial for each packet to determine whether to drop it or not. The second is a correlated loss process, which allows losses to be aggregated into contiguous runs which simulates brief network outages, such as would be expected on a wireless network.

The correlated loss process is configured by two parameters: an overall percentage of total packets to be dropped, and an average drop run length. For example, if the user specified a 5% loss rate with an expected drop run length of 7, then in a 1000 packet transfer one would expect 50 packets to be dropped in groups of av-

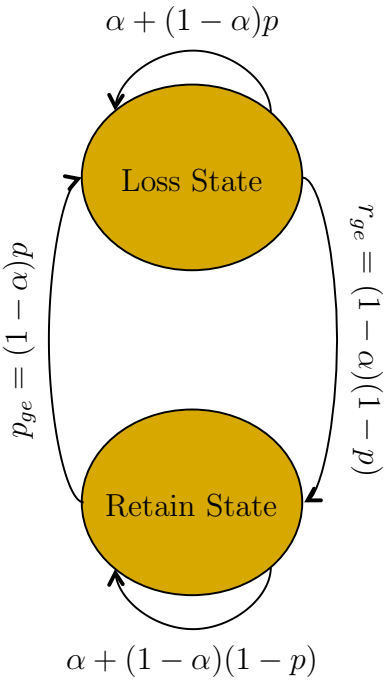

Figure 2.4: Two-State Markov loss model

erage size 7. This is achieved using a Gilbert-Elliott model [11], [12] as shown in Figure 2.4.

It can be shown that that the stationary probability of being in the loss state is p, regardless of the value of α; therefore, it is possible to tune α to set the correlation factor as desired for a given p. Furthermore, the expected number of packets dropped in a single burst can be expressed as:

$$\begin{aligned} E &= 1 + \frac{\alpha + (1-\alpha)p}{(1-\alpha)(1-p)} && : p < 1 \\ &= \frac{1}{(1-\alpha)(1-p)} && : p < 1 \end{aligned}$$

Intuitively, this formula is derived as one packet already dropped, plus the expected number of self-transitions to the loss state before transitioning to the retain state.

The values of E and p are provided to the test harness, so it is necessary to express α as a function of these two values by solving the above formula for α:

$$\alpha = 1 - \frac{1}{E(1-p)} \qquad : p < 1$$

The operator can specify a loss or ECN profile and a link delay to the test harness, and the harness will automatically perform a series of tests and yield the average and standard deviation of transfer times. The results of these experiments at various network configurations are presented in the following sections.

2.2 Testing Results on Extended Set of Network Profiles

This section presents test results of LT-TCP in comparison with TCP-SACK and, where appropriate, NORM, on networks exhibiting correlated packet losses, high latency, and ECN marking.

There are some outstanding issues in the LT-TCP reference implementation used to collect these results that should be understood. First, the implementation currently has some faults in its RTT estimation algorithm which cause it to overestimate the round trip time of the network and sometimes calculate extremely large values. This causes LT-TCP to select the maxmimum timeout (this is 120 seconds in Linux) before continuing transmission although no acknowledgments have returned. This is normally not an issue, but under correlated packet losses it is common for an entire send window of packets to be dropped, forcing the sender to wait for the timeout before sending any more data, which may be dropped as well, delaying for another timeout. This problem is particularly prevalent early in the slow-start phase when the send window is only a few packets in size.

To collect the results in this chapter in a timely manner, the maximum timeout for LT-TCP was set to five seconds, which allowed transfers to complete in reasonable time. Setting the maximum timeout this small does violate RFC 6298 which dictates a maximum retransmission timeout of no less than 60 seconds [23], but it was deemed acceptable given the known and controlled loss and congestion factors in the testing environment. It is worth noting that the presence of this fault intuitively makes the

results presented for LT-TCP here a lower bound, when compared with a correct implementation.

The second outstanding issue is that LT-TCP's packets in flight counter is overly simplified, and should be developed further. The packets in flight counter keeps track of how many packets the connection has placed on the network, and is compared with the congestion window before sending new packets. If the packets in flight is greater than or equal to the size of the congestion window, no new packets are sent until some packets leave the network. This is how congestion control is enforced. At present, the LT-TCP packets in flight counter is reset to zero if no acknowledgments are received, the congestion window is full, and the retransmission timeout expires with outstanding packets waiting to be sent. It is assumed that if the timeout expires with a full congestion window, an entire send window has been lost, which is fairly common with correlated losses, especially early in the connection when the congestion window is small. This solution ensures completion of transfers, but introduces other issues. Further discussion of this issue is provided in Section 3.5.

Unless otherwise stated, the following methodology was used for the collection of LT-TCP results presented below: timings of 20 transfers were collected, and the top and bottom n outliers were discarded, where n does not exceed 5, until the standard deviation of transfer times was less than the mean. In some cases, no outliers needed to be eliminated.

This methodology was chosen because of the outstanding faults in the reference implementation, especially the RTT overestimation, which can cause some transfers to require an inordinately long time, particularly in correlated loss cases where several send windows are dropped at the beginning of the connection requiring several consecutive maximum timeouts before the congestion window can be grown to large enough to continue sending in spite of the losses. In a correct implementation, the RTT would be estimated accurately, which would allow transmission to continue sooner because the timeouts would reflect the real link instead of being the maximum. This and other remaining transient defects in the reference implementation introduce noise which skews the results, so the methodology of dropping outliers

until the standard deviation no longer exceeds the mean was adopted to produce results believed to be most consistent with a correct implementation.

2.2.1 Correlated Packet Losses

In real mobile, ad-hoc networks, packet losses tend to occur in bursts, such as those caused by moving network nodes, changing routes, or interfering signals. Previously, LT-TCP was tested using a loss model which dropped packets uniformly at random. In order to gauge performance in more realistic networks, data transfers were run over a network exhibiting correlated losses as described in the previous section. Each of LT-TCP, TCP, and NORM were tested with data transfers at several different loss rates and correlation factors. The average transfer rates for LT-TCP, TCP and NORM are presented respectively in Tables 2.1, 2.2, and 2.3. Note that NORM was run with congestion control disabled, as its performance was similar to that of TCP with it enabled. NORM also requires the transmission rate of the link to be specified to it by the application.

Table 2.1: LT-TCP transfer rates (KiB/s) for correlated losses

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
0	48762	N/A*	N/A	N/A
5	16516	14222	9309	8533
10	11378	5120	4655	2626
20	3938	3793	2276	528

*Correlation of losses is not defined where no losses are present

Table 2.2: TCP transfer rates (KiB/s) for correlated losses

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
0	111304	N/A	N/A	N/A
5	861	366	59	29
10	233	92	20*	<10*
20	67	0	0	0

*Average of successful trials; some did not complete
Value of zero indicates no trials completed in reasonable time

The results indicate that TCP performance degrades as loss correlation increases. This is consistent with other research [3], [30], and is believed to be the

Table 2.3: NORM transfer rates (KiB/s) for correlated losses

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
0	30118	N/A	N/A	N/A
5	13128	11253	14423	13653
10	7877	9309	10240	10240
20	6024	6400	5389	4655

result of TCP halving the congestion window multiple times. The effect of correlation on LT-TCP and NORM is not as pronounced, and both protocols perform acceptably in the presence of correlated packet losses. In particular, correlation of losses does not appear to have any reliable effect on NORM's performance.

One issue of interest is that TCP significantly outperforms LT-TCP at 0% losses. This has also been observed in previous work [6], and is believed to be due to inefficiencies in the reference implementation rather than the LT-TCP protocol itself. Nevertheless, this is a serious issue and efforts to optimize the LT-TCP implementation to produce equivalent performance to TCP on lossless networks are ongoing.

2.2.2 High Latency Networks

Another network profile which TCP is known to under-utilize is networks with a high Round Trip Time (RTT), such as networks including one or more satellite links. TCP becomes highly sensitive to losses on networks with a high RTT because TCP's congestion window is not able to grow to fill the network once TCP enters the Congestion Avoidance state, at which point it will only increase the congestion window once per RTT.

Throughputs for LT-TCP, TCP, and NORM on a network with an RTT of 250 milliseconds are presented respectively in Tables 2.4, 2.5, and 2.6. The 250ms RTT was selected because satellite network links impose a 125ms delay in each direction. As can be seen, TCP performance is severely impeded by any losses in a high-latency environment. The effects of correlation on TCP are less pronounced at this latency, but are still observable.

Table 2.4: LT-TCP transfer rates (KiB/s) for 250ms RTT links

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
0	1766	N/A	N/A	N/A
5	1796	2179	2179	1969
10	1625	2090	2133	1600
20	1652	1707	1484	1384

Table 2.5: TCP transfer rates (KiB/s) for 250ms RTT links

*Average of successful trials; some did not complete

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
0	2301	N/A	N/A	N/A
5	40	60	68	51*
10	21	19*	∞	∞
20	∞	∞	∞	∞

Somewhat counter-intuitively, in these results, LT-TCP performance is sometimes increased slightly at higher uncorrelated loss rates. This is believed to be caused by Network Emulator imposing packet reordering when it applies delays to a network. The loss detector presently employed in the LT-TCP implementation incorrectly detects losses when significant packet reordering is present. This behavior is discussed in greater detail in Section 3.3. Because LT-TCP is spuriously detecting losses, it generates and sends RFEC which incurs overhead at lower loss rates, but at higher loss rates the extra redundancy is helpful, particularly as the loss rate estimation algorithm currently used tends to underestimate the loss rate, as discussed in Section 3.1.

Another interesting effect of higher delays is the tendency of TCP and LT-TCP to yield higher performance in the presence of correlated losses as compared with uncorrelated losses at a given loss rate. TCP's congestion window does not grow to utilize all bandwidth on high-latency networks [13], particularly in the presence of losses. If the window never grows far above the minimum, then the effect of multiple halvings due to correlated losses is mitigated, as the window reaches the minimum sooner, making further reduction impossible. Increasing correlation at a given loss rate makes bursts less frequent, meaning TCP can grow its window slightly more

Table 2.6: NORM transfer rates (KiB/s) for 250ms RTT links

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
0	1510	N/A	N/A	N/A
5	887	952	1047	926
10	818	869	681	880
20	679	583	478	493

between bursts, giving, on average, greater throughput, if the total rate of losses is low. At higher loss rates, however, the rate of entire windows being dropped grows too high, especially in increased correlation, and the data transmission rate becomes unacceptably low.

NORM does not perform as well as LT-TCP or TCP with high latency and no loss, but it maintains its performance much better than TCP as losses increase. LT-TCP continues to outperform NORM in these tests; however, socket parameters to NORM can be tuned to optimize it for high-latency networks. In the interest of fairness with TCP and LT-TCP, this custom optimization was not performed, but it is anticipated that NORM performance would be improved by more careful tuning.

2.2.3 ECN-Marking Networks

A final network profile of interest for LT-TCP is one which marks packets with ECN. On a network which does not drop packets, but does mark ECN, LT-TCP should perform similarly to TCP, as both protocols should respond to ECN by reducing their send rate. In congested networks where both TCP and LT-TCP streams are present, the protocols should both back off to ensure a fair distribution of the available bandwidth.

Because ECN is marked randomly by routers with increasing per-packet probability as the queues are filled, and a congested router is likely to be processing traffic from many flows, it would be highly improbable for ECN markings to be correlated. The result of correlated ECN marking is an exponential increase in transfer time due to repeated halving of the congestion window. For these reasons, correlated ECN marking was not tested. Average transfer rates for LT-TCP and TCP over

ECN-marking networks are presented together in Table 2.7. NORM is implemented over UDP, which cannot support ECN, so it could not be tested with ECN.

Table 2.7: LT-TCP and TCP transfer rates (KiB/s) for ECN-marking links

Loss Rate (%)	LT-TCP	TCP
0	48762	69508
5	6827	4440
10	3938	1539
20	2626	427

While LT-TCP is reacting to congestion, it does not react as strongly as TCP does. This could be due to error in the packets in flight counter (see Section 3.5), or it may be caused by LT-TCP not adjusting its congestion window by the same rules TCP does. Further investigation on this issue is necessary.

2.3 Performance Baselines

In addition to testing LT-TCP alongside other transport protocols on more network profiles, it is also useful to perform some special-case tests which can form objective benchmarks to compare LT-TCP performance against to determine whether its various components are as beneficial as expected. In particular, this section presents two special tests: an oracle loss estimator, to determine how effective LT-TCP's loss estimator is at avoiding retransmissions; and standard TCP-SACK without congestion control, which determines how advantageous LT-TCP's error-correction coding is, excluding the benefit of differentiating between physical loss and congestion.

Oracle loss estimator The oracle loss estimator sits in place of LT-TCP's standard packet loss rate estimation code. This code normally takes samples from the SACK maps returning from the receiver and maintains a weighted average of the overall packet loss rate, which is then used to determine how much PFEC to generate and send in a block. The oracle does not use the SACK maps, but instead returns a fixed value which matches the real loss rate configured by the test harness. Comparing the throughput of transfers using the oracle estimator against the throughput

using the standard loss estimator indicates the overall effect of the loss estimator code on transfer speeds, and bounds the improvement which can be gained by improving the loss estimator. The oracle estimator was tested under correlated loss conditions and its results are presented in Table 2.8. These results can be compared with those in Table 2.1, which were generated using the standard estimator.

Table 2.8: LT-TCP transfer rates (KiB/s) with oracle loss estimator

Loss Rate (%)	Uncorr.	E=2	E=5	E=10
5	18301	18273	16412	16131
10	12171	11624	11086	9894
20	6787	6794	6136	4683

The oracle yields significantly improved performance in many of the tests, which indicates that there is significant improvement to be gained by further improving the loss estimator. Further discussion on the loss estimator is provided in Section 3.1, where two loss estimation algorithms are compared. It can be seen that the current loss estimator frequently underestimates the loss rate, which explains the oracle's higher performance.

TCP without congestion control One of LT-TCP's advantages over TCP is the ability to distinguish between loss and congestion by using ECN. Another advantage is the usage of network coding to proactively mitigate packet losses and avoid retransmissions. It is useful to evaluate the significance of these advantages separately, to demonstrate the value of having both.

To accomplish this, a null operation congestion control module for TCP was constructed, which never shrinks the congestion window, only grows it. This module was enabled in some congestion-free experiments to provide a baseline for TCP performance where losses are not interpreted as indicators of congestion.

These results are presented in Table 2.9. New LT-TCP results were taken, as these tests were performed on a separate instance of the experimental network. The design of this network is the same as the one used in other results.

The results in Table 2.9 show that LT-TCP's Forward Error Correction coding is highly advantageous at higher loss rates, but at 5% loss, TCP without congestion

Table 2.9: LT-TCP and TCP transfer rates (KiB/s) with no TCP congestion control

Loss Rate (%)	LT-TCP	TCP
5	11674	25366
10	7574	1068
20	3613	88

control is faster. It is important to note that LT-TCP is known to underestimate the loss rate with uncorrelated loss of 5% of packets (see Section 3.1), which would render LT-TCP's FEC coding much less effective. This fact, combined with the performance issues which cause LT-TCP's performance to fall behind TCP's on lossless networks, provides a compelling explanation as to why TCP's performance exceeds that of LT-TCP in this test.

At higher loss rates, the advantages of proactively mitigating losses are more significant. This is partly due to the more accurate loss estimation which reduces LT-TCP's need for RFEC, but also the increase in losses means that TCP will lose more time waiting to detect each loss, then sending a retransmission before it can resume sending new data.

3. Protocol Enhancements

The results presented in the previous chapter reflect the current state of LT-TCP, following many iterations of testing and improvement. In the process of collecting these results, several issues in LT-TCP were highlighted, and the protocol was revised to improve performance and stability in the new network profiles as well as the old ones.

This chapter presents the most significant of these revisions and discusses the rationale behind the changes made as well as other solutions which were tried and rejected in the process.

3.1 Loss Rate Estimation Algorithm

The loss rate estimator is an integral part of LT-TCP's network coding system. The loss rate estimator runs on the sender and uses data acquired by inspecting the returning SACK maps to estimate the packet loss rate of the network as a percentage of packets dropped. This estimate is then factored into the block coding parameters in an effort to minimize the probability that RFEC will be needed for the receiver to decode the blocks.

The loss rate estimation algorithm needs to be as accurate as possible. If the loss rate is underestimated, then the receiving endpoint will be unable to decode the original data, and RFEC must be sent to correct for the missing data, similar to a retransmission in TCP. If the loss rate is overestimated, then unnecessary error correction coding will be sent, which wastes network resources and reduces throughput.

In previous work, LT-TCP used a dynamic Exponentially Weighted Moving Average (EWMA) model to estimate the loss rate. In this original model, the loss estimate at iteration n, denoted r_n was calculated using the new sample of the loss rate s_n as follows:

$$\alpha = \frac{s_n}{r_{n-1} + s_n}$$
$$r_n = \alpha s_n + (1 - \alpha) r_{n-1}$$

This method of calculating α based on the relationship between the new sample and the last estimate causes α to approach 1 if $s_n >> r_{n-1}$ so a single sample indicating a high loss rate has a large effect on the estimate; however, if $s_n << r_{n-1}$ then α approaches 0, meaning a single sample indicating a low loss rate has little effect on the overall estimate. In short, this gives an estimator which will immediately yield a high loss estimate given a high sample, but requires many iterations of low samples to lower the estimate again.

Under uncorrelated losses, this estimator works reasonably well because all samples are expected to match the overall loss estimate. The introduction of correlated losses, however, introduces a few samples with high loss rates interspersed with many samples of low loss rates, which caused the estimator to quickly return an estimate close to 1 after the first high sample, and subsequent low samples had little effect to bring it back down.

To resolve this situation, the dynamic EWMA algorithm was replaced with a static EWMA algorithm, which uses a fixed value for α, which gives new samples an equal weight on the estimate, regardless of the difference between the new sample and the previous estimate. The value for α was chosen to be 0.6 after experimenting with several different values and choosing the one which yielded the most accurate results. The time-weighted averages of the the loss estimate for transfers run using the original dynamic EWMA estimator can be found in Table 3.1 and those using the static EWMA are in Table 3.2.

In light of these results, the static EWMA estimator is generally more accurate. Although it does have a tendency to underestimate, especially in uncorrelated losses, as correlation increases, the static EWMA estimator yields consistently more accurate results.

Table 3.1: Time-weighted average of loss estimates (%) given by dynamic EWMA at various loss rates and correlations

True Loss Rate	Estimated Loss Rates			
	Uncorr.	E=2	E=5	E=10
5	4.65	5.84	9.18	18.27
10	9.11	9.72	14.35	20.46
15	14.22	16.24	18.79	24.66
20	18.05	18.94	30.78	34.65

Table 3.2: Time-weighted average of loss estimates (%) given by newly-designed EWMA using $\alpha = 0.6$ at various loss rates and correlations

True Loss Rate	Estimated Loss Rates			
	Uncorr.	E=2	E=5	E=10
5	2.57	4.60	3.27	8.69
10	7.97	8.68	9.80	10.07
15	14.31	14.93	12.21	13.20
20	18.88	18.35	18.11	18.65

3.2 Connection Closure Handshake

Like TCP, LT-TCP performs handshakes to both establish and close connections, and in previous work, LT-TCP has used the same handshake procedures as TCP. The connection establishment handshake requires no adjustments for LT-TCP, save that the endpoints must agree to use LT-TCP instead of TCP. The TCP connection closure handshake, however, does not function properly in LT-TCP.

In TCP, the connection closure procedure begins when one or both sides' application decides that it will send no further data. At this point, an application which has finished sending data directs its transport layer to begin closing the connection. TCP dictates that the closing endpoint then send a segment with the FIN flag set, to notify the other endpoint that no further data is forthcoming and it should also begin the closing handshake.

When the application notifies TCP to close the connection, it is possible that TCP still has some segments queued to be sent; however, it is guaranteed that no further data segments will be created, and TCP always guarantees that, unless the underlying network fails entirely, all data segments will be delivered. Because of

these guarantees, TCP can safely mark the last segment in the queue with the FIN flag. If the queue is empty, TCP creates an empty segment with the FIN flag set and transmit it.

LT-TCP does not guarantee that all data segments will be delivered, as its use of redundancy corrects for any missing data. Any segment can be lost and LT-TCP will not attempt to retransmit it; if the receiver is unable to decode the original application data, RFEC will be sent until the receiver can decode. Furthermore, LT-TCP cannot guarantee that when the application notifies it of connection closure, no further data segments will be created; RFEC data segments may be created after this point. Because these guarantees cannot be made, LT-TCP cannot rely on a single data segment to carry the FIN flag. If that segment is dropped, the opposing endpoint may never initiate the handshake to close the connection.

In summary, the TCP concept of a FIN segment does not port to LT-TCP, as the segment may be lost and will not be retransmitted as it would in TCP. It is possible to mark the subsequent RFEC segments, which must be sent to correct for the lost segment, with FIN flags, but this creates further problems because the FIN segment is no longer unique, nor does it indicate that no new data segments will be transmitted.

The proposed solution is to replace the TCP concept of a FIN segment, which is the unique segment after which no further data segments will be created, with the LT-TCP concept of a FIN block. In TCP, a segment is an atomic unit of transmission which TCP guarantees will be delivered intact, complete, and in order. LT-TCP does not treat segments this way, but it does treat blocks this way; therefore, it is natural to replace the TCP FIN segment with an LT-TCP FIN block.

In the revised LT-TCP closing procedure, when the application notifies LT-TCP that it has finished sending data, LT-TCP will check if any outstanding blocks are still in the queue to be sent. If so, it will mark the last outstanding block as a FIN block, similar to the way TCP marks the last outstanding segment as the FIN segment. Then all segments associated with that block will be marked with the FIN flag, guaranteeing that the receiving endpoint will be notified that the relevant block is the FIN block and no further blocks are forthcoming. The receiving endpoint will

mark all ACKs associated with that block as FIN/ACKs as TCP would with the ACK for the FIN segment.

If there are no outstanding blocks to be sent, LT-TCP will construct an empty FIN segment and send it, exactly the same as TCP would.

One final possibility is that LT-TCP is notified to close with a partially-transmitted block still in the queue. In this case, LT-TCP will mark this partially-transmitted block as the FIN block if and only if all packets sent so far in that block are not sufficient to decode it. If it is possible that the receiver could decode the block with only the packets already sent, the sender shall instead queue an empty FIN segment to be sent when the partially-transmitted block has been fully received, and will proceed from that point as though there had been no outstanding blocks in the queue. This final behavior is analogous to TCP behavior if the application closes the connection with the last data segment having been transmitted, but not yet acknowledged.

While it is believed that this revision to the protocol will allow the connection closure handshake to cleanly and reliably end the connection, the revision has not yet been applied to the reference implementation for testing.

3.3 Loss Detection Algorithm

Closely related to, but distinct from the loss rate estimator is the loss detector. This is the code which parses incoming SACK maps and tallies up how many packets are missing. This code works by finding the last acknowledged packet in a SACK map and counting how many packets previous to that packet are unacknowledged. The loss detector then updates two counters: the total packets up to and including the last received packet, and the number of unacknowledged packets prior to the last received packet. When the loss rate estimator runs, it calculates the newest loss rate sample as $s_n = \frac{\text{unacked}}{\text{total}}$, then it resets both counters to zero.

Unfortunately, this algorithm does not factor in packet reordering. If a packet is delivered out of order, the loss detector regards it as lost. This leads to an overestimate of packet loss rate when packet reordering is prevalent.

A revision to the protocol has been applied to address this issue, which is to detect packets which were previously counted as lost in older SACK maps but are now delivered, and decrement the count of lost packets accordingly. This approach has improved the overestimation, but it is limited due to the fact that if the loss rate estimator runs before reordered packets are acknowledged, it will calculate a high sample of the loss rate and then reset the counters, thereby preventing the loss detector from repairing the mistake when the packets are acknowledged. This revision has been implemented, and the results in Chapter 2 were obtained using this algorithm.

A better solution is to implement a configurable reordering threshold, such as TCP uses, so that a packet is not regarded as lost immediately if the next packet is delivered first, but instead is regarded as lost only when a certain number of packets following it are delivered before it is. This will greatly reduce the probability of the loss estimator running with an inflated loss rate sample, which in turn avoids the situation where a packet which was prematurely declared lost cannot be removed from the loss counter after the loss rate estimator resets it. At present, this solution has not been implemented nor tested in the reference LT-TCP implementation.

3.4 RFEC Scheduling

The more stable loss detection algorithm described in Section 3.3 facilitates some improvements to the algorithm controlling when RFEC is created and sent for a block. Presently, blocks for which all data and PFEC packets have been transmitted are tracked using an idle counter, and if a given block has not been updated after a certain number of ACKs, RFEC is generated and sent.

This idle counter approach does ensure that all blocks will be delivered eventually, but if packets are reordered near block boundaries, RFEC may be sent unnecessarily if too many acknowledgments for some block $n+1$ are received before block n is completely acknowledged. Furthermore, it does not consider a block eligible for RFEC transmission until the block has been completely transmitted.

The more stable threshold-based loss detector can be leveraged for a more streamlined block delivery. A block is initially composed of n packets: k data

packets and $n - k$ PFEC packets. The block cannot be decoded with less than k packets, so if more than $n - k$ packets are lost, the receiver will be unable to decode the block without RFEC. Because a threshold-based loss detector is unlikely to spuriously declare packets as lost, if more than $n - k$ packets in a given block are known to be lost, the RFEC scheduler can generate RFEC to be sent for that block as soon as the initial n packets are sent. This avoids the need to wait for several acknowledgments (or far worse, timeouts, if there are no other blocks in transit) before sending the RFEC the receiver will need to decode the block.

At present, this revision has not been made in the LT-TCP reference implementation, and so it is not reflected in the results in Chapter 2.

3.5 Packets in Flight Counting

The packets in flight counter is a crucial component of congestion control. Congestion control is managed in two parts. First is the widely known congestion window, which tracks how many packets a connection may have on the network at any give time. This necessitates a second, but often less recognized component, which is the packets in flight counter, the estimate of how many packets are actually in flight, which must be compared with the congestion window to determine if any new packets can be produced.

In networks where any packets may be lost at any time, counting packets in flight is far from trivial. The Linux TCP implementation's packets in flight counter is implemented as a complex confluence of four different counters, each of which may be adjusted by several different pieces of code for various reasons. The final packets in flight count is calculated by adding and subtracting these four counters into a single result.

Much of this complexity in the TCP counter is due to TCP's need to deal with retransmissions, and can be eliminated in LT-TCP, as LT-TCP does not retransmit packets; instead, it generates RFEC. For this reason, TCP's packets in flight counter is not a natural fit for LT-TCP and a new one had to be constructed.

The counter used for the results presented in Chapter 2 use a simple counting algorithm which uses a single packets-in-flight counter which is incremented when a

packet is sent and decremented when it is acknowledged. The counter can be reset to zero only if packets are waiting to be sent, but the congestion window is full, and a timeout occurs due to no returning acknowledgments. In this event it is assumed that, due to the lack of returning acknowledgments for an entire timeout, all packets in the network have been lost.

This prevents the connection from hanging, but it has other issues. It is possible that acknowledgments are en route, but delayed until past the timeout (TCP has facilities for this contingency as well) which drives the LT-TCP counter to a negative value. To avoid this, the counter was lower bounded at zero.

Another shortcoming of this solution is that the count is known to drift from reality due to a few packets being added to the counter which will never be removed unless a timeout occurs. These packets accumulate as blocks are fully received with packets belonging to them still in flight. As long as the congestion window keeps growing, these do not hang the connection, but they do reduce performance. In a network which both drops packets and marks ECN, this could trigger unnecessary timeouts and severely hamper performance.

The proposed solution to this problem is to implement a block-based packets in flight counter, which tracks the total packets in flight as a sum of counters for all outstanding blocks. Each block counter is implemented as the difference between a count of packets sent for that block and the packets for that block which have left the network, either due to acknowledgment or having been lost (as detected by the loss detector discussed in Section 3.3). Retired blocks' counters do not factor into the overall packets in flight count, so the problem of drift is eliminated.

As before, if a timeout occurs due to no acknowledgments being received, the packets in the network are all declared lost. If some acknowledgment for a packet declared lost is received after this point, it is counted towards block completeness, but not towards the block's counter of packets which have left the network. This could be implemented by setting a marker on the last packet sent for the block when the timeout occurs, which denotes that all prior packets are regarded as having left the network, regardless of their lost or acknowledged status.

This solution does temporarily allow a few extra packets to be placed on the network while waiting for the packets from the retired block to all leave; however, the number of extra packets is bounded by the block size, and should be small as LT-TCP tries to avoid excess transmissions whenever possible.

4. Portable Reference Implementation

The present LT-TCP reference implementation was developed via direct modifications of the Linux 2.6.26.5 kernel source code. The TCP code was copied and modified in some places, but in other instances it was directly altered in place to comply with the LT-TCP protocol. This created a massive code base with many lines of unused, duplicated code, some of which were modified and later replaced by changes elsewhere, but never deleted. As a result, the implementation is tightly bound to an old version of Linux (as of this writing, the current stable Linux kernel is version 3.14.2 and the oldest supported version is 2.6.32.61) and is difficult to extract and move to a modern operating system.

This chapter presents a design for an updated reference implementation developed with modularity and portability in mind, to facilitate faster and easier development, testing, and updating as new versions of the Linux kernel are released.

4.1 Motivation

The current LT-TCP reference implementation is complex. Its code is divided among several files in the Linux source tree, including some dedicated LT-TCP files and some mixed TCP and LT-TCP files. There are no clear indications whether any given code section belongs to TCP, LT-TCP, both or neither. This yields an implementation which is illustrated in Figure 4.1.

This creates several problems when testing, debugging, and extending the implementation. Because the code is dependent on a deprecated version of the Linux kernel, it cannot be tested with modern applications that utilize modern kernel functionality, such as virtualization systems which could test LT-TCP in more complex and realistic scenarios than Network Emulator [17] is capable of. Furthermore, when testing does reveal a fault in the implementation or a weakness in the protocol, determining what code is responsible for undesired behavior and how to resolve the issue without breaking other sections of the code is significantly more laborious than would be the case in a modular implementation, which could be updated and tested

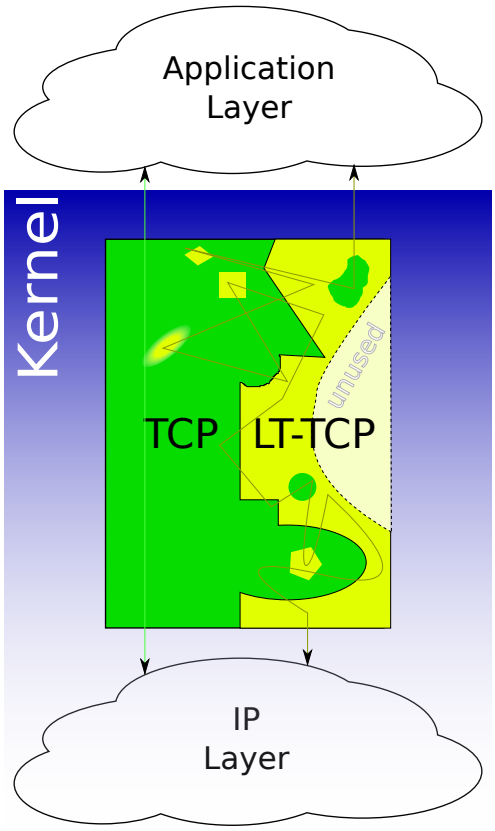

Figure 4.1: Illustration of Current LT-TCP Implementation

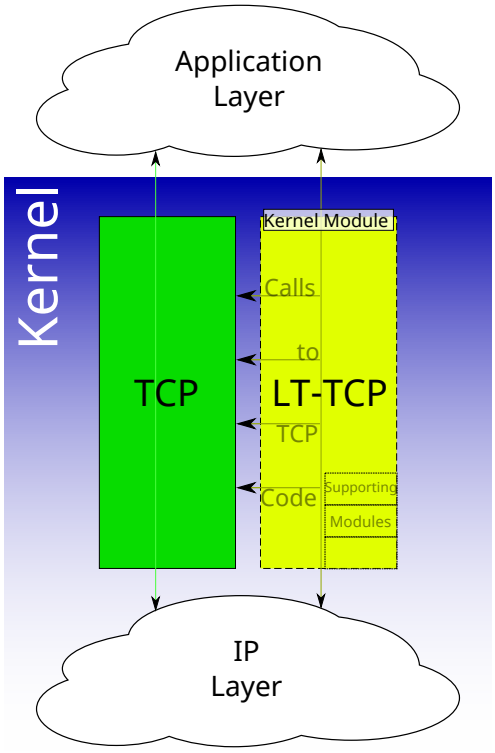

Figure 4.2: Illustration of Modular LT-TCP Implementation

within the running kernel, as opposed to the current implementation which requires the entire kernel to be recompiled and rebooted for each change to be tested.

To address these issues, the following sections describe a new reference implementation designed as a set of Linux Kernel Modules, which are sections of code developed outside of the Linux kernel but can be loaded and unloaded dynamically into a running kernel [25]. The proposed reference implementation is illustrated in Figure 4.2.

By developing the LT-TCP code outside of the Linux code, dependencies of LT-TCP on a particular version of the Linux source are minimized, greatly simplifying the task of updating LT-TCP to newer versions of Linux as they are released. Furthermore, it eliminates confusion as to whether a given section of code is part of the Linux TCP implementation, or part of LT-TCP, which expedites the process of debugging and extending LT-TCP. Finally, the modular architecture allows rapid development and testing of submodules of LT-TCP, such as loss estimators, block encoders/decoders, and RTT calculators; all without even needing to rebuild or reload the LT-TCP implementation as a whole. Only changes to the module under development would be required, and only that module would need to be reloaded.

4.2 Architecture

The modular LT-TCP architecture will entail no changes to the kernel source code itself; rather, all of LT-TCP will be implemented as dynamically loadable Linux Kernel Modules (LKMs). An LKM is a compiled C binary with a special .ko (kernel object) filename extension, which can be loaded and run in a Linux kernel. LKMs can be compiled directly into a kernel, so that they are available at all times and require a slightly smaller memory allocation, but cannot be unloaded or modified without recompiling the kernel; or they can be built outside the kernel and loaded and unloaded dynamically. The latter option is preferable during development, while the former option can be used to deploy a stable implementation. This development methodology is used for other transport layer protocols in Linux, such as SCTP [29].

LT-TCP can be logically divided into several different modules. The core module will register the LT-TCP protocol with the kernel and provide handlers for

all of the socket events that LT-TCP must handle, such as an application creating an LT-TCP socket, connecting it, reading and writing data on it, and closing it; as well as the relevant IP layer events, such as an incoming LT-TCP packet or an error packet for an LT-TCP connection. The block encoding and decoding functionality can be extracted into another module, congestion control into another (Linux's TCP implementation does this), loss estimation into another, and so on. Implementing the logical components of LT-TCP in separate modules allows each component to be developed independently, which simplifies the code base by reducing and enumerating dependencies, and also allows different algorithms for each component to be tested easily. An added benefit is that a given LT-TCP deployment can be optimized for its environment by selecting component implementations which best fit that environment.

In addition to the LT-TCP Core module, the following components have been identified to be implemented in external modules:

- Block encoder and decoder
- Congestion control
- Packet and block size calculator
- Loss rate estimator
- Round trip time estimator

4.3 Magnitude of Task

An implementation of LT-TCP is a sizeable project, and re-implementing LT-TCP in the modular architecture detailed above is a nontrivial undertaking. The Linux TCP implementation features many options that are not relevant to LT-TCP, but even excluding those, the relevant code is approximately 10,000 lines of C code. It is difficult to count how many lines of code are in the current LT-TCP implementation, as it is mixed in with the TCP code, but the number is greater than 6,300 lines. Although the time investment into rebuilding LT-TCP from the ground up is significant, the benefits and advantages outlined in this chapter justify the difficulty.

The process of rebuilding the LT-TCP implementation will be somewhat eased by the existence of the current reference implementation; however, because that im-

plementation is so tightly coupled with Linux 2.6.26.5 and its TCP implementation, it is difficult to determine what code is relevant to LT-TCP and whether that code can be used in a different kernel or not. Because of this, the current implementation is suitable as a reference to determine completeness and to test the new implementation against, but most of it cannot be copied directly into the new implementation and must instead be rewritten completely.

4.4 Implementation Milestones

The process of reimplementing LT-TCP has been mapped out into five milestones. These milestones are defined below.

Milestone 1: Protocol Registration Create a module which registers the LT-TCP protocol with the Linux kernel when loaded, and unregisters it when unloaded. This module will need to provide empty implementation functions for the various kernel events which LT-TCP must handle. Upon completion of this milestone, an application will be able to create an LT-TCP socket, but will not yet be able to do anything with it.

Milestone 2: Data Segmentation and Transmission Add facilities to the module to establish and close connections, as well as segment data from the application layer and send and receive it. Upon completion of this milestone, an application will be able to send and receive data on an LT-TCP socket similarly to a UDP socket. Correct, in-order delivery will not be guaranteed.

Milestone 3: Modular Architecture Add facilities to load and unload LT-TCP supporting modules, and create the modules as minimal functional implementations, which return static results (for example, the loss rate estimator may always report 5% losses). Upon completion of this milestone, LT-TCP will be capable of loading and unloading its supporting modules at run-time, but these modules will not yet be functional.

Milestone 4: Block Segmentation and Transmission Implement segmentation of application data into blocks and implement transmission and selective acknowledgment of these blocks based on the information generated by the supporting modules. Upon completion of this milestone, an application can expect LT-TCP to correctly send and receive data, with guarantees of complete, in-order delivery, but performance will be poor as the supporting modules are still placeholders which do not account for observed network conditions.

Milestone 5: Complete Implementation Fully implement remaining supporting modules so that LT-TCP reacts to the observed network conditions and adjusts its parameters accordingly. Upon completion of this milestone, the modular implementation should have feature parity with the previous implementation.

5. Conclusion and Future Work

In this thesis, the LT-TCP protocol has been tested on a variety of network profiles, including various loss rates, correlated losses, networks which mark packets with ECN, and networks with high round trip times of 250ms. It has been tested in comparison with TCP, as well as an alternative transport layer solution, NORM. In many cases, LT-TCP offers the best performance of these solutions, generally outperforming TCP by orders of magnitude on networks with high losses, and giving competitive results with NORM while offering a more convenient programming interface.

This testing has exposed several faults in the reference implementation including crashes, corruption of application data, and performance issues. Many of these faults have been corrected, and the implementation has been updated to improve stability and ensure data correctness and improve performance on this widened scope of network environments.

The testing has also illuminated some areas for improvement in the protocol, such as the loss detection algorithm, the loss rate estimation algorithm, the connection closure process, RFEC scheduling, and packets in flight tracking. This thesis has proposed improvements to the protocol to address all of these issues, some of which have also been implemented and tested.

Even so, there are some outstanding issues in LT-TCP which must be addressed. As in previous work [6], LT-TCP performance is lower than TCP's when no losses are present. While this is not particularly surprising, since Linux's TCP implementation is the product of decades of refinement by professional developers well familiarized with low level software optimization, it does somewhat diminish LT-TCP's attractiveness for implementation in real-world applications, so further optimization of LT-TCP is necessary. Furthermore, there continue to be defects in the reference implementation, and some protocol revisions proposed in this thesis have yet to be tested.

In addition, the current reference implementation is becoming dated, and is no longer suitable for the kinds of testing which is needed for the protocol. Modern virtualization software could unlock a host of new network profiles and testing environments, but the current implementation is tightly coupled with a Linux kernel which predates these solutions and is not compatible with them.

To address this issue, this thesis has presented a design for an updated, modular implementation of LT-TCP which would be simpler to maintain, facilitate easier development and comparison of different revisions and algorithms for the various components of the protocol, and would greatly reduce the labor required to update LT-TCP to future versions of the Linux kernel as they are created.

Future Work

LT-TCP shows promise as a candidate to replace TCP, but before it can be seriously considered for this role, further refinement is necessary. Perhaps the clearest issue is that it does not match TCP performance on lossless networks. This continues to be an issue which must be addressed, as it is unlikely that many would consider downgrading a system's performance in wired networks, which rarely lose data, in order to improve performance in wireless networks.

Additionally, the testing presented in Chapter 2 demonstrates numerous areas for improvement, such as Subsection 2.2.3 which establishes that LT-TCP is reacting less strongly to congestion than TCP, which requires further investigation.

Although a new loss rate estimation algorithm was proposed, implemented, and demonstrated significant improvements as per the results in Section 3.1, these results as well as the oracle loss estimator results in Section 2.3 indicate that there are further benefits to be gained by improving the loss rate estimator. One avenue to explore would be to replace the Exponentially Weighted Moving Average model with an Auto-Regressive model, which reportedly yields more accurate results [20].

Section 3.3 details a more accurate algorithm for detecting packets which have been lost as opposed to merely delivered out of order, and this algorithm should be implemented and tuned. Having this more stable loss detection algorithm will provide a foundation for the revisions discussed in Sections 3.4 and 3.5 as well. All

of these revisions should have a favorable impact on performance, particularly in networks which reorder packets.

Further work is needed testing LT-TCP in real networks, where multiple traffic flows are present and routers mark ECN packets. Testing LT-TCP with a more diverse set of applications and protocols such as HTTP, SSH, and FTP is needed as well.

Finally, the largest and perhaps most important outstanding task is the completion of the modular reference implementation. As of this writing, milestone 1 is complete, and milestone 2 is underway, so there is much work yet to be done. When completed, the modular implementation should facilitate much easier development and testing of the LT-TCP protocol.

Loss-Tolerant TCP continues to show promise as a transport layer protocol applicable for a wide range of network environments. It maintains its performance advantages over TCP in lossy network environments; furthermore, it has proven to be competitive with an alternative transport layer solution, NORM, and the path forward promises further performance improvements yet to come. In an age of ever-expanding wireless networks, LT-TCP promises to provide an efficient and reliable transport layer for all applications that currently rely on TCP, as well as many of those applications whose requirements necessitate alternative solutions to TCP to maintain performance in challenging network environments.

REFERENCES

[1] *NACK-Oriented Reliable Multicast (NORM) Transport Protocol*, RFC Editor 5740, 2009.

[2] *TCP Congestion Control*, RFC Editor 5681, 2009.

[3] E. Altman *et al.*, "TCP in Presence of Bursty Losses," in *Proc. 2000 ACM SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems*, New York, NY, 2000, pp. 124-133.

[4] H. Balakrish *et al.*, "Improving TCP/IP Performance over Wireless Networks," in *Proc. 1st Annu. Int. Conf. Mobile Computing and Networking*, New York, NY, 1995, pp. 2-11.

[5] L. Baldantoni *et al.*, "Adaptive End-to-End FEC for Improving TCP Performance over Wireless Links," in *2004 IEEE Int. Conf. on Communications*, Paris, FR, 2004, pp. 4023-4027.

[6] K. Battle, "Optimization of an Implementation of Loss Tolerant Transmission Control Protocol (LT-TCP) and Its Evaluation," M.S. Thesis, Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2012.

[7] *The Addition of Explicit Congestion Notification (ECN) to IP*, RFC Editor 3168, 2001.

[8] P. Brenner. (1996, July 18). *A Technical Tutorial on the IEEE 802.11 Protocol* [Online]. Available: http://www-users.cselabs.umn.edu/classes/Fall-2010/csci4211/Lec-Notes/802.11-tutorial.pdf (Retrieved on 29, July, 2014)

[9] C. Casetti *et al.*, "TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links," in *Proc. ACM MobiCom 2001*, Rome, ITA, 2001, pp. 287-297.

[10] J. Cloud *et al.*, "Network Coded TCP (CTCP) Performance over Satellite Networks," in *Computing Research Repository*, 2013. arXiv: 1310.6635

[11] E. Gilbert, "Capacity of a burst-noise channel," *Bell Syst. Tech. J.*, vol. 39, no. 5, pp. 1253-1265, Sept. 1960.

[12] E. Elliott, "Estimates of error rates for codes on burst-noise channels," *Bell Syst. Tech. J.*, vol. 42, no. 5, pp. 1977-1997, Sept. 1963.

[13] *HighSpeed TCP for Large Congestion Windows*, RFC Editor 3649, 2003.

[14] *TCP Selective Acknowledgement Options*, RFC Editor 2018, 1996.

[15] C. P. Fu and S. C. Liew, "TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 2, pp. 216-228, Feb. 2003.

[16] B. Ganguly, *et al.*, "Performance of Loss-Tolerant TCP (LT-TCP) in the Presence of Correlated Losses," in *Proc. MILCOM 2013, IEEE Military Commun. Conf.*, San Diego, CA, 2013, pp. 1341-1346.

[17] S. Hemminger, "Network Emulation with NetEm," presented at the Linux Conference Australia, Canberra, AUS, 2005.

[18] H. Holzbauer, "An Implementation of Loss Tolerant Transmission Control Protocol (LT-TCP)," MS Thesis, Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2012.

[19] I. Järvinen and D. Miller. (2009, May 4). *TCP: Extend ECN Sysctl to Allow Server-Side Only ECN* [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=255cac91c3c9ce7dca7713b93ab03c75b7902e0e (Retrieved on July, 12, 2014)

[20] D. Jeske *et al.*, "QoS With an Edge-Based Call Admission Control in IP Networks," in *Proc. 2nd Int. IFIP-TC6 Networking Conf. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; and Mobile and Wireless Communications*, Pisa, ITA, 2002, pp. 178-189.

[21] M. Kühlewind *et al.*, "On the State of ECN and TCP Options on the Internet," in *Proc. 14th Int. Conf. Passive and Active Measurement*, Hong Kong, CHN, 2013, pp. 135-144.

[22] A. Kumar, "Comparative Performance Analysis of Versions of TCP in a Local Network with a Lossy Link," *IEEE/ACM Trans. Netw.*, vol. 6, no. 4, pp. 485-498, Aug. 1998.

[23] *Computing TCP's Retransmission Timer*, RFC Editor 6298, 2011.

[24] *Transmission Control Protocol*, RFC Editor 793, 1981.

[25] P. Salzman *et al.*. (2007, May, 18). *The Linux Kernel Module Programming Guide (2.6.4th ed.)* [Online]. Available: http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf (Retrieved on July, 17, 2014)

[26] A. Sheth *et al.*, "Packet Loss Characterization in WiFi-Based Long Distance Networks," in *Proc. INFOCOM 2007. 26th IEEE Int. Conf. Computer Communications*, Anchorage, AK, 2007, pp. 312-320.

[27] V. Subramanian *et al.*, "An End-to-End Transport Protocol for Extreme Wireless Network Environments," in *Proc. MILCOM 06, IEEE Military Communications Conf.*, Washington D.C., 2006, pp. 1-7.

[28] O. Tickoo *et al.*, "LT-TCP: End-to-end framework to improve TCP performance over networks with lossy channels," *Int. Workshop Quality of Service*, vol. 3552, Passau, GER, 2005, pp. 81-93.

[29] L. M. Yarroll and K. Knutson. (2001, Jul. 16). *Linux Kernel SCTP: The Third Transport* [Online]. Available: http://old.lwn.net/2001/features/OLS/pdf/pdf/sctp.pdf (Retrieved on July, 12, 2014)

[30] Y. Zhang *et al.*, "TCP over OBS: Impact of Consecutive Multiple Packet Losses and Improvements," *Photonic Network Commun.*, vol. 16, no. 3, pp. 203-220, Dec. 2008.