Department of Computer Science

Stefan Alfredsson

TCP Lite - A Bit Error Transparent Modification of TCP

Master's Thesis

2001:06

TCP Lite -

A Bit Error Transparent Modification of TCP

Stefan Alfredsson

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Stefan Alfredsson

Approved, June 21, 2001

Advisor: Anna Brunström

Examiner: Donald Ross

Abstract

TCP is a reliable transport protocol designed for heterogenous networks. To provide a reliable service over possibly unreliable networks, retransmission of lost or damaged data is performed. These retransmissions incur a delay and increases the total transmission time. However, certain applications can make use of damaged data, while taking advantage of the decreased delay created by fewer retransmissions. Currently there is no way to allow the applications to access this data.

This thesis proposes a modification to TCP which would allow applications to decide when damaged data can be accepted and not. The idea has been implemented in the Linux operating system. As errors often occur over wireless links, the implementation has been tested with a number of emulated wireless links. The experiments showed that there are gains to be made by letting errors through.

Contents

1 Introduction **1**

2 Background **5**

2.1 TCP 5

2.1.1 Reliability Achievement Technique 7

2.1.2 The Problem with Congestion Control 8

2.2 PRTP 9

2.3 UDP Lite 10

2.4 Leaky ARQ 11

2.5 Snoop and Split Connection 12

2.6 Robust Checksum-based Header Compression 13

3 Problem Analysis **15**

3.1 Modification at the Sender and/or Receiver Side 15

3.2 Application to TCP Stack Communication 16

3.3 Link-layer Internal Checksum 18

3.4 Security Considerations 18

3.5 Proposed Solution 19

4 TCP Lite Implementation **21**

4.1 Linux Kernel Overview 21

4.2 The Modification 22

4.3 Kernel Modification 24

4.4 Socket Controlling 24

5 Experimental Environment **27**

5.1 Experiment Motivation 27

5.2 Characteristics of a Wireless Link 28

5.2.1 Bit Error Rate 28

5.2.2 Error Distribution 29

5.2.3 Bit Errors and Interleaving 30

5.3 Experimental Parameters 30

5.3.1 Selected Link Profiles 30

5.3.2 Loss Profile Parameters 31

5.3.3 Experiment Network Topology 33

5.4 Error Emulation Setup 34

5.4.1 Introduction to Dummynet 34

5.4.2 Bit Errors in Dummynet 35

5.4.3 Error Distribution Implementation Problem 36

5.4.4 Bandwidth Limitation Issues 38

5.5 Measurement Collection 39

6 Experimental Results **41**

6.1 Expected Results 41

6.2 Experiment Execution 41

6.3 Experiment Analysis 43

6.3.1 Throughput Graphs 44

6.3.2 Retransmission Graphs 48

6.3.3 Throughput vs Retransmission Graphs 51

6.3.4 General Analysis 54
6.4 Conclusion 56

7 Conclusion and Future Work **57**

References **59**

A Experiment Comparisons **63**
A.1 Disabled TCP Lite versus Unmodified TCP Stack 63
A.2 Two Runs of TCP Lite Disabled 66

B Experiment Anomaly Discussion **69**

C Example Application Program **73**

D Tcptrace Example Output **77**

List of Figures

2.1 The IP header 6

2.2 The TCP header 6

2.3 A successful (a) and a problematic (b) transmission 8

2.4 How congestion occurs 9

4.1 Biterrors let through 23

5.1 The error distribution model as a markov chain 29

5.2 The experiment network topology 33

5.3 How dummynet operates 35

5.4 Differences in error distribution depending upon time 37

5.5 Bandwidth limitation problem 38

5.6 One solution to the bandwidth limitation problem 38

6.1 Visual view of the experiment execution at the sender 42

6.2 Example graph with explanations 44

6.3 Throughput results for the GSM profile 45

6.4 Throughput results for the UMTS profile 46

6.5 Throughput results for the 802.11b profile 47

6.6 Retransmission results for the GSM profile 48

6.7 Retransmission results for the UMTS profile 49

6.8 Retransmission results for the 802.11b profile 50

6.9 Throughput vs. retransmissions results for the GSM profile 52
6.10 Throughput vs. retransmissions results for the UMTS profile 52
6.11 Throughput vs. retransmissions results for the 802.11b profile 53
6.12 Examination of errors in the GSM profile transmission 54
6.13 Examination of errors in the 802.11b profile transmission 55

A.1 Throughput comparison for the GSM profile 64
A.2 Throughput comparison for the UMTS profile 64
A.3 Throughput comparison for the 802.11b profile 65
A.4 Throughput comparisons for the GSM profile 66
A.5 Throughput comparisons for the UMTS profile 67
A.6 Throughput comparisons for the 802.11b profile 67

B.1 Retransmission problem 71

List of Tables

5.1 Parameter values used in the experiments 32
5.2 Characteristics of the link profiles 33

Chapter 1

Introduction

Since the invention of the telephone by Graham Bell, most communication has been transmitted over wired networks. The networks were first used for voice communication. When the computer was invented, people wanted it to communicate as well. Reusing the established infrastructure, the computers could communicate long distances over the wired networks by using modems. Dedicated networks were built to be used only by the computers, which lessened the need for analog modulation of the digital signal and provided higher reliability. As technology progressed, ways to transmit signals without wires were invented. The *cellular*[1] phone provided wireless communication to the general public. As in the case of the wired telephone network, people wanted their computers to be able to communicate without having to be connected with a wire to the wall all the time. Solutions for this problem have been developed as well, but some difficulties still remain. Wireless communication mostly uses radio waves as carrier. The physical properties of radio waves can be likened to dropping a stone in calm water. The water rings spread in circles, reflects against rocks and fades in amplitude. The same problems occur with radio waves [2]. They are reflected against walls and objects in the environment, and fades in amplitude relative to the distance from the source. These processes make it hard to distinguish what was sent, i.e. a "1"

[1]Also called *mobile* phone in Europe.
[2]Assuming lower frequencies. At higher frequencies the waves become more and more directed.

can be interpreted as a “0” and the other way around. These problems have however been worked around pretty well with techniques such as Forward Error Correction interleaving and adjusting the encoding of information to the radio environment. Upon this foundation, higher networking layers have been built. Some errors slip through the lower layer anyway (depending on the signal quality). This then affects the upper protocols, which receive erroneous data from the link layer. One protocol affected is TCP (see Section 2.1), which is the most used transport layer protocol on the Internet today. TCP was designed to operate in heterogeneous environments. It adapts to long round trip times and retransmits lost information. However, the problems described above with wireless communication can severely degrade TCP performance and utilization of wireless links. This is because TCP misinterprets the cause of lost packets in a wireless environment, compared to a wired one. Many ideas have been proposed to solve the problem, but one has yet to be generally accepted.

More and more of the traffic transmitted over the Internet can be classified as multimedia data. Usually it means video and sound streams. Compared to “regular data” such as documents or compressed information, multimedia data has a greater user tolerance for quality degradation. For example, some incorrect pixels in a picture or an incorrect sound sample is harder to perceive than missing text in a document. Thus, a small degradation in quality can be accepted. Multimedia data also has real-time like demands, compared to opening a document. A delay in a sound or video presentation is usually quite noticeable and distracting. The cause of the delay can depend on different reasons, with the network being one of them. In the case of TCP, it provides a reliable transmission, meaning that lost or damaged data will be retransmitted. Logically, a retransmission incurs a delay. Concluding the two discussions above, we find that delays can be reduced at the cost of reliability. As for the data that needs to be retransmitted; it can either be lost in the network, or reach the destination, but be corrupted. If the data is lost, there is no option but to request a retransmission, or accept that it is lost and manage without it. However, if the data is received but corrupted, this gives us a third option. Instead of discarding it, the application might make use of the corrupted data.

The objective of this masters thesis is to examine the possibilities of a TCP/IP stack that does not discard corrupted packets, but can deliver them anyway if so desired. The application can then control the communication stack to prioritize either quality or transmission delay. The modified TCP/IP stack is implemented in the Linux operating system. Emulations[3] are performed to show the gains that can be made in terms of delay versus acceptable loss.

The modified TCP protocol is named “TCP Lite”. The reasoning behind the name is that the protocol builds on TCP, while providing an option to make it more lightweight by accepting errors. Hence the suffix “Lite”. Parallels can also be drawn to the UDP Lite protocol (discussed in Chapter 2), which similarly accepts packets with errors but builds upon UDP. It should be noted that the TCP version in the 4.4BSD Lite operating system is sometimes referred to as TCPLite, but the two have no connection whatsoever.

The rest of this thesis is structured as follows. Chapter 2 presents background information on TCP as well as covering other work done in this area: PRTP, UDP Lite, Leaky ARQ, the Snoop protocol and robust header compression. Chapter 3 analyzes how the problem of accepting errors could be solved, and proposes to use a receiver-based modification of the TCP stack. Chapter 4 describes the implementation. It begins with an overview of Linux, how and where the modification was done and discusses how the application can control TCP Lite. To test the implementation, we built an experimental environment, described in Chapter 5. The chapter motivates the experiment, discusses characteristics of wireless links and defines three profile links used for the experiment. The software used for emulation and implementation of these characteristics is also detailed. The chapter concludes with a section on how measurements are collected for the experiments. Chapter 6 discusses and analyzes the results obtained in the experiments. Finally, Chapter 7 presents a summary of the work done, a conclusion of the TCP Lite experiments and areas for the future work.

[3]There is a fine line between emulation and simulation. Our experiments are however classified as emulations, because of the mixed hardware and software used.

Chapter 2

Background

This chapter presents background information and details other similar projects. Topics include TCP, PRTP, UDP Lite, Leaky ARQ, the Snoop protocol and robust checksums. Key points and differences with TCP Lite are discussed where applicable.

2.1 TCP

The Transmission Control Protocol[Pos81b] is the most used transport protocol on the Internet. It provides reliable stream communication between two processes, usually via the BSD socket abstraction[Ste98]. To achieve this, TCP has mechanisms for setting up and closing down connections, detecting lost or damaged packets, and requesting retransmission of the lost or damaged data. More advanced features include the ability to send out-of-band data, round-trip time calculation, congestion control and negotiating what features are to be used. Features are negotiated trough an "option" field in the TCP header, making the protocol extensible while keeping backwards compatibility.

TCP's primary carrier is the Internet Protocol[Pos81a], which delivers packets to the correct host, as determined by the destination address in the IP header, see Figure 2.1. The TCP header then contains a port number to further distinguish to which stream/process the packet[1] belongs, as depicted in Figure 2.2 [2]. In order to detect missing packets, sequence numbers are used, which are counters of the number of bytes transmitted. The sequence numbers are sent back to the sender by the receiver, to acknowledge that the packet has been received. If an acknowledgment is not received within a certain time-frame, the packet is assumed to be lost and is retransmitted.

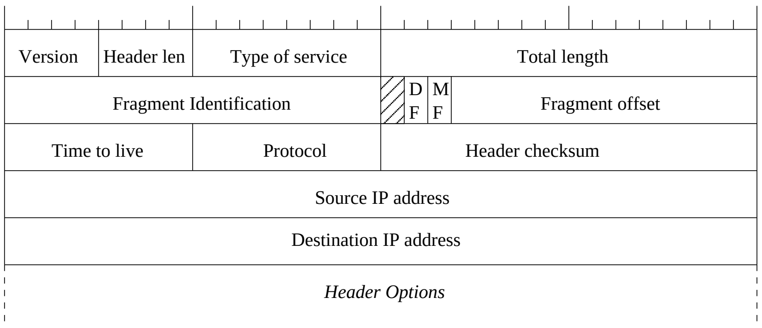

Figure 2.1: The IP header

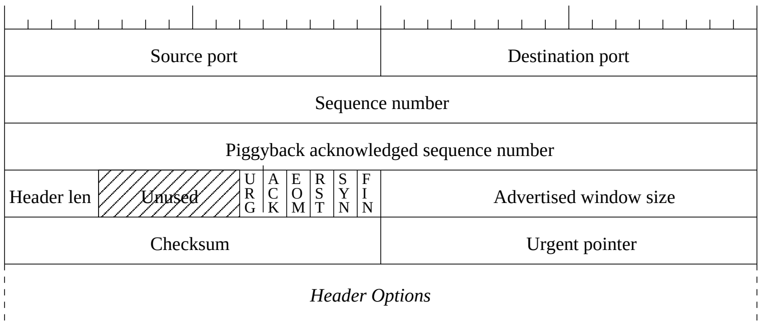

Figure 2.2: The TCP header

To detect corrupted packets, TCP uses a checksum. When the packet is sent, the checksum is calculated and set by the sender. When the packet is received, the checksum is checked again, and if it matches the packet is deemed to be correct. On busy hosts, these calculations can be performed a significant number of times. Therefore, it is designed to be as efficient as possible,

[1]Actually TCP segment, however the term ’packet’ is used throughout this thesis.
[2]The header figures are created by Johan Garcia at Karlstad University, and used with his permission.

by using 1-complement calculations. The specification [BBP88] also lists a number of assembler implementations that can be used.

2.1.1 Reliability Achievement Technique

To give a better understanding of the functionality of TCP Lite, TCPs reliability mechanisms are discussed in more detail. As said in the previous section, each byte sent by TCP is given a *sequence number*, which is a counter of the number of bytes that has been transmitted. The counter does not start at 0, but at an offset called the initial sequence number. The initial sequence number is chosen randomly in order to provide better security. For each TCP packet constructed, the packet is marked with the lowest sequence number, that is, the number of the first byte. When the receiver gets the packet, it sends an *acknowledgement* packet back to the sender. The acknowledgement number field of the reply contains the sum of the received sequence number and the number of bytes just received. This tells the sender that all bytes up to this number have been received.

Since TCP is often used over unreliable[3] networks, packets are sometimes lost. Either the packet from the sender or the acknowledgement from the receiver could be lost. In both cases the sender notices that it does not get an acknowledgement for data that has been sent, and starts a retransmission after a certain timeout period. This is illustrated[4] in Figure 2.3. In 2.3(a), a successful transmission is shown. In 2.3(b), an acknowledgement is lost at 2.3(b.1). This causes the packet to be retransmitted, since the sender did not receive an acknowledgement of the sent packet. This retransmission is lost in 2.3(b.2), causing another timeout at 2.3(b.3). This time the transmission is successful. However, since the receiver already got the data with the first transmission, it is discarded this time. From the figure it is easily seen that retransmissions incur a significant delay, compared to when no packets are lost. In the next section, causes of packet losses and how TCP Lite reduces the amount of losses in a wireless environment are discussed.

[3] With no guarantee about packet ordering or integrity.

[4] Note that this example only gives a simple illustration of a transmission with TCP. Connection setup, piggy-backed ACKs, advertised windows and exponential back-off have been left out.

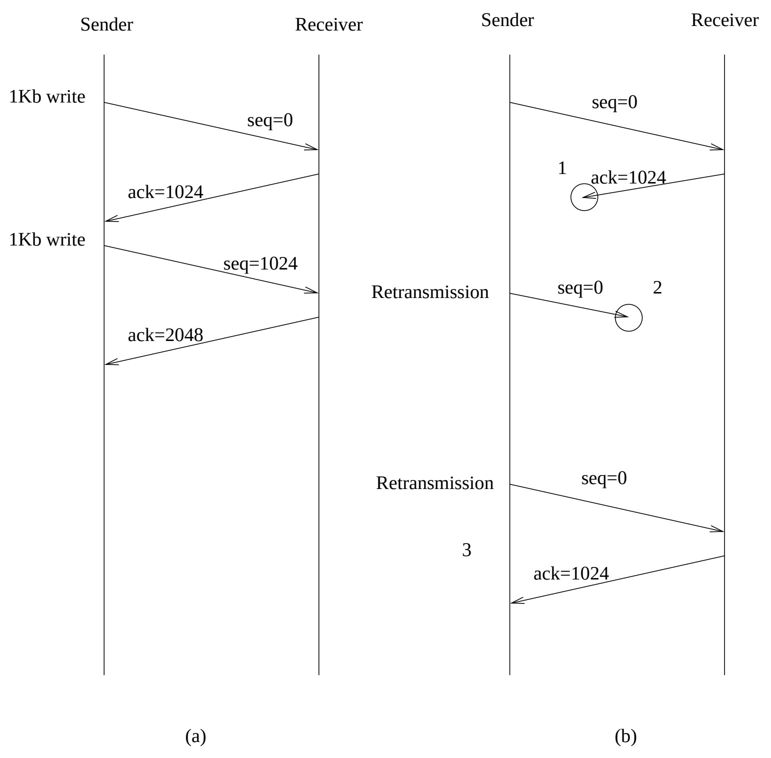

Figure 2.3: A successful (a) and a problematic (b) transmission

2.1.2 The Problem with Congestion Control

The common cause of lost packets in wired networks are congested routers. In an ideal world, the bandwidth limitation would consist of the hardware or software limitations of the peers involved in the communication. However, in order to utilize bandwidth over links as much as possible, a single link is shared by a number of clients. Looking at Figure 2.4, one sees a setup where clients on two local networks are connected through a shared network, which is assumed to be slow compared to the speed of the LANs. One can easily see that if many hosts were to be communicating with the other side, the network cannot deliver the traffic at the maximum local speed. This state is called *congestion*, and because the buffers in routers are filled to the maximum, the newly incoming packets are dropped (unless they have a special priority, forcing a less important packet to be dropped).

Since wired network links seldom cause dropped packets by themselves, packet loss is inter-

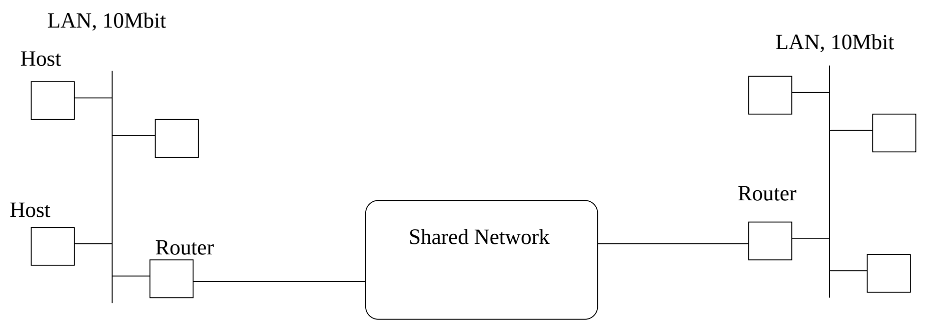

Figure 2.4: How congestion occurs

preted as a sign of congestion by TCP. This is detected by the lack of acknowledgements from the receiver. In order to get out of this situation, the sender lowers it transmission rate to ease the load on the network. However, there is no way to distinguish between a packet dropped due to congestion in a router and a packet dropped by the receiver due to a bad checksum. Therefore, it is argued that a TCP stack, like TCP Lite, that can make a difference between those two cases is more efficient[5] than one that does not. The problem of differentiation of loss due to congestion or errors is also an area of active research, for example the work done in [BV98, BV99].

2.2 PRTP

The *Partially Reliable Transport Protocol*[AGBS00] is developed by the data communications research group at Karlstad University. It is an extension to TCP, where the application can specify which reliability level it requires. PRTP then detects packet losses, and sends a fake acknowledgement if the packet is not needed to maintain the specified reliability level. This leads to a reduced total transmission time at the cost of lost packets. In addition to the reliability level parameter, an *aging factor* determines the time locality that will be considered. For example, with an aging factor of 1, all packet losses from the beginning of the session are considered when determining whether to send a fake acknowledgement or let it time out. A factor less than one will make the algorithm pay less attention to the status of older transmissions (both lost and

[5]While keeping in mind the tradeoff done in TCP Lite.

received data).

PRTP can be seen as a coarser version of TCP Lite, in the way that it works on a packet basis, and TCP Lite works on a bit by bit basis. Also, PRTP will not attempt to deliver damaged packets, either the data in the packet is delivered correctly or not delivered at all (with undetectable gaps to the application), as where TCP Lite will deliver damaged data. It is easy to see a future integration of the two ideas, complementing each other.

2.3 UDP Lite

UDP Lite[Lar00] shares many of the goals of TCP Lite, but operates on UDP rather than TCP. It is developed by researchers at Luleå University, Sweden, and allows an application to divide a UDP packet into a sensitive and an insensitive part. An error in the sensitive part will cause the packet to be dropped, while an error in the insensitive part will still be delivered. This is useful for example if the Real Time Protocol[SCFJ96] (RTP) is used on top of UDP. RTP provides functionality suitable for carrying real-time content, and has mechanisms for synchronizing streams with timing properties. The RTP header in the beginning of the UDP payload provides, among other information, a timestamp which needs to be correct. However, the rest of the UDP payload might be allowed to contain errors.

Comparing UDP Lite to TCP Lite, the formers packet abstraction provides a good starting point; since the application is aware of the packets it can control them individually. In TCP, the stack itself splits the byte stream into packets, with no control from the application, and the data is delivered to the receiving application in the same way, i.e. just a stream of bytes. The implication of this is that it is hard for the application to specify the level of importance for a range of bytes. Despite this, TCP Lite could be a better choice than UDP Lite depending on the needs of the application. This is because TCP (and therefore inherently TCP Lite) guarantees that data eventually will be delivered, whereas UDP provides no such mechanism in itself. Also, the inherent differences between the TCP and UDP protocol, such as lack of congestion control in UDP, should be taken into consideration.

2.4 Leaky ARQ

Han and Messerschmitt has devised a protocol called Leaky ARQ[6][HM99]. It is a progressively reliable transport protocol for interactive wireless multimedia and enables an application to quickly present a coarse representation[7] of the data, and later request retransmission of the damaged packets in order to progressively refine the data, if needed. The term “leaky” comes from the idea that the protocol leaks corrupt data to the application, instead of discarding it which is normally done. The requirements and limitations of an ARQ protocol to make it leaky is outlined in [HM99], and the use of TCP and UDP as a base is discussed. UDP could be used to send the initial data, and then TCP could be used to send the refinements. However, problems are noted when TCP is involved, because there is no way to cancel retransmissions of stale data. These retransmissions then incur delays on time-sensitive transmissions with UDP (where bandwidth is low). Another option is to use only UDP, but it has problems as well. If checksumming is enabled, corrupt packets are discarded. With a disabled checksum, the headers can be corrupt instead. Therefore, a need is seen to separate header and payload error detection. UDP Lite (discussed above) is listed as an alternative protocol to handle this separation. Another option is for Leaky ARQ to do header-only error correction by itself.

Relating Leaky ARQ to TCP Lite we see the common behavior of leaking corrupt data to the application. However, the later has no method of re-requesting corrupt data later, whereas this is thought of in Leaky ARQ. This can be done since both sides involved in the communication are aware that Leaky ARQ is used, whereas TCP Lite requires changes only to the recieiver.

[6]Automatic repeat-request
[7]Assuming packets are damaged

2.5 Snoop and Split Connection

In [BSAK95], the design and implementation of a protocol to improve the TCP/IP performance over wireless networks is described. The protocol is called the *snoop* protocol, and is deployed at the wireless base station. It listens on the communication to the client without the client knowing, therefore the name *snoop*. Snooped packets are cached and locally retransmitted over the wireless link, instead of having to be retransmitted over the full path. This means that TCP's congestion mechanisms work as intended, and that unnecessary reductions in the links bandwidth utilization are avoided. Instead of letting the sender assume congestion because of bit errors on the wireless link, packet loss is detected via ACK snooping and locally retransmitted.

The paper further discusses the desirability of only modifying components of administrative control, i.e. base stations and mobile hosts. Since it can not be expected of the origin server (e.g. web or ftp server) to be modified, the modification needs to be done in the other end of the communication. The evaluation of the protocol via experiments over an AT&T WaveLan showed performance improvements of up to 20 times over regular, non-snooped, TCP/IP communication.

Snoop tries to compensate for problems over the wireless link at the base station. Another approach similar to this is called *split-connection*. Instead of having one connection between the wired and wireless host, the base station acts as a proxy between the two and resolves problems locally. An example of this approach is *Indirect TCP*[BB95].

The development of these approaches confirms that there indeed exists a problem with wireless links and bit errors, and that it is interesting to make a distinction between packets dropped in a router due to congestion and those dropped at the receiver endpoint due to bad checksums. This distinction-feature is shared with TCP Lite, but the fundamental difference is that the snoop protocol and split-connection will recover corrupt packets locally, while TCP Lite delivers errors as is.

2.6 Robust Checksum-based Header Compression

As traditional tele and data communication converges, the demand to have IP communication to the end points increases. Having one network protocol used all the way decreases complexity and costs incurred by using bridges that convert IP to a proprietary protocol. One can envision future telephony as a subset of data communication, where voice samples are digitized and sent in packets, in this case IP packets. The packets must be kept small, in order to achieve “real-time” characteristics; larger packets mean more buffering, leading to delays in communication, and gives the sense that the other party takes a long time to think before answering.

In wired networks, IP telephony works quite well, but for wireless networks with low bandwidth and high delay the overhead incurred by headers is significant, because of the small payload in each packet. This problem is addressed by the IETF ROHC working group, which have produced an Internet draft for a technique named ROCCO[JDHS00]. Their proposal is to have a close look at how the header fields are changing over time, and classify them accordingly. For example, the IP version field, header length, protocol and source and destination address will be the same in all packets of a packet stream. Hence, they need only be sent once. Values might vary within certain limits, for example the sequence numbers in TCP varies within a certain amount, and it is thus unnecessary to use a 32 bit counter when a 16 bit counter would suffice[8].

In order to handle different types of packets, such as UDP/UDP+RTP/TCP, *compression profiles* have been defined. Thanks to the profiles, headers have been compressed to a minimal size of 1 octet, with an average of 1.25 octets. This compression have been shown to tolerate severe (10^{-2}) bit error rates. The framework also introduces a CRC checksum to ensure that the decompressed header is the correct one.

This header compression technique would be quite useful in a protocol like TCP Lite, where, in the current design, there is no checksum covering the headers. Also, the decreased size of the header will mean less probability of an error in the header.

Being an Internet draft, it should be noted that [JDHS00] is only valid for a maximum of six

[8]For a detailed discussion about the header fields, see the appendix of [JDHS00].

months and may be updated, replaced or obsoleted at any time, and is to be regarded as “work in progress”.

Chapter 3

Problem Analysis

In this chapter, problematic issues with TCP Lite are discussed and analyzed. The chapter discusses where the “lite” modification is to be done and how the application can communicate with the transport layer.

3.1 Modification at the Sender and/or Receiver Side

When a protocol is implemented, it can be divided into a receiver and a sender side (even if the two are interchangeable roles of the same implementation). However, when a modification is to be done to a protocol, it might not be necessary to modify both sides, but only the receiving or sending side may suffice.

Hence, one decision that has to be made is where the “lite” modification is to be done. There are three options; Either modify only the receiving side, both the sending and receiving side, or only the sending side. The last option implies that the receiver will request retransmissions for missing packets in order to reassemble the stream. Since this is in conflict with our goal to accept packets with bad checksums, we are left with the other two options.

With a sender/receiver based modification, the protocol can be made more robust; partial checksums can be used to protect important data, such as packet headers. An application where a server side modification can be deployed quite rapidly is in proxy situations, i.e. a proxy sits on each side of the erroneous link.

The advantage of only modifying the receiver side is that those wanting the functionality can install it without having to care about their peers interest in updating their operating systems. For example, content providers using proprietary or old, unsupported operating systems do not need to depend on their vendor to implement TCP Lite. The overhead of a proxy is also avoided, but we are left with the problem of errors in the header. Since bit-errors can occur anywhere in a packet, it is possible that the port numbers and sequence numbers will be damaged. IP addresses are protected by the IP header checksum, and if the destination address is damaged it will not even reach the correct host[1], so the issue is only about the TCP headers. This needs to be solved somehow. Since one can always fall back to discarding a packet (it will be retransmitted when it is not getting ACKed), it will be the easiest thing to do if the header fields do not make sense. Headers can also be guessed upon, for example if there is only one TCP connection setup and the port number does not match, we can be fairly sure where the packet is destined. These heuristic can be elaborated further, while keeping in mind the risks of passing a packet to the wrong socket. For example, if a `RST` packet is misjudged, the wrong socket will be closed.

3.2 Application to TCP Stack Communication

The application usually knows when it can and can not accept damaged data, and therefore needs a way to activate and deactivate this behavior in the underlying communication layer, in this case TCP. It would also be interesting for the application to know that errors have occurred in the stream.

The usual way of communicating with the kernel in a POSIX[2] compliant operating system is via *system calls*. When manipulating file descriptors, three system calls can be used, `ioctl()`,

[1]Depending upon where the damage was inflicted. If it happened on a link connected directly to the receiver, it would reach the host anyway.

[2]Portable Operating System based on unIX

`fcntl()` and `setsockopt()`. These take a file descriptor, a command and arguments used by the command as arguments. To implement new functionality, the kernel must be modified to accept the command number (which is defined in a header file). When the actual command is used, the new function corresponding to the command will be called.

The issue of knowing where errors have occurred is harder because of the stream abstraction. The TCP stack knows what packets contain errors, because of the broken checksum. However, the `read` and `recv` system calls have no notion of communicating such information. As the application uses these systems calls to transfer data, it therefore has no way of knowing when there is an error or not. An option might be to use OOB (Out Of Band) signalling locally (i.e. the TCP stack inserts OOB messages that the application can understand), but then the application would have to poll this information every so often. Packet based communication, like UDP, is easier to handle because the application is aware of the packet abstraction as discussed earlier.

A solution for this is to keep the application in full control of what is going on. It can dynamically enable or disable ’lite-mode’ depending on its current error tolerance. This enables the application to receive a correct HTTP header, notice that the content-type is `image/jpeg`, for which it can tolerate errors,and enable ’lite-mode’. It would also be possible to tell the stack that errors can be accepted in the next 23kb of data, so ’lite-mode’ is automatically turned off after that amount has been received.

However, there are problems with the enable/disable approach as well, most notably buffering. After the HTTP header has been received, much data might have been buffered in the network stack and might have been resent without need. This is only an issue of performance, but disabling lite mode is more serious. For example, if lite-mode has been enabled, data with errors is likely to have been buffered. When the application has disabled lite-mode, it expects the data to be correct. But there is no way to get the data already buffered and acknowledged to be retransmitted without the application knowing about it. In this thesis, this behavior of TCP Lite is not considered, but is subject to future research.

3.3 Link-layer Internal Checksum

The wireless links can have their own checksumming, so that the point to point connection is reliable, i.e. bit errors will not be passed further.

This poses a problem for TCP Lite, since the fundamental idea is to take advantage of bit errors. This behavior must be controlled somewhere, in order for TCP Lite to be effective. For example, in the GSM mobile system, the “reliable link protocol” can be enabled or disabled. This way bit errors can be corrected by the link or delivered to the transport layer.

To have a more flexible and controllable environment (instead of using a real wireless link), the link is emulated using a PC with two network cards, running appropriate software to introduce errors. This is discussed further in the chapter about the experiment environment.

Over some links, more than one link layer is used. For example, the *point to point* protocol (PPP) is a common way of encapsulating IP packets between two endpoints. Since PPP has its own checksum, it would have to be disabled in these situations for TCP Lite to be effective. Otherwise PPP would detect the broken packets and request retransmission.

3.4 Security Considerations

It could be argued that by relaxing the checksum control, an opportunity for unauthorized alteration of the transmission could be created. That is, it would be easier to get fake packets accepted by the receiver than it would originally. We do not believe that this is the case when only considering a broken checksum. Since the checksum can be calculated by any party and does not have any properties of a cryptographic signature, an attacker could just as well calculate a correct checksum.

However, there are other implications. To alter a TCP transmission, an attacker needs to know certain information such as who is communicating and what ports they use. Also, to inject packets into an existing connection, the actual sequence numbers currently used must be known. If a packet is injected with the wrong sequence number, it will be rejected by the receiver. This is the reason why the initial sequence number should be randomly generated, as discussed in Section 2.1.1. Suppose the initial sequence number always started at a fixed value (or an easily predictable value, as in earlier versions of the Windows NT operating system). An attacker could then try the numbers after this value, and eventually inject a packet which gets accepted. With a randomized initial sequence number, the attacker would have to analyze the network traffic between the hosts to determine what sequence numbers are used. This is related to TCP Lite in the following way. To be efficient, TCP Lite should try to reconstruct corrupt headers to deduce what stream the packet belongs to. Assuming the sequence number was corrupted, it could be[3] repaired to fit back into the stream. Therefore, an attacker could more easily inject a packet into a stream controlled by TCP Lite. The solution for this is to have separate checksums for the header and the payload.

3.5 Proposed Solution

In the light of the above discussion and to limit the scope of this thesis, the proposed solution is to use a receiver-based modification with an optimistic approach towards errors in the headers, i.e. headers are assumed to be correct or can be corrected easily. This is achieved by only modifying the checksum of TCP packets in the experiments. Further development of the protocol should introduce more intelligent header decoding. Negotiation about checksum coverage between communicating parties, preferably in combination with a header compression scheme such as ROCCO to achieve lower overhead should be considered as well. For this thesis, security has not been considered, but issues described in the previous section should be kept in mind during further development.

[3]Subject to future research

Chapter 4

TCP Lite Implementation

This chapter discusses the method used to implement the theories presented in earlier chapters. The target platform for the implementation is the Linux kernel. Therefore, the chapter begins with an overview of the Linux kernel. The chapter continues with a description of what needs to be modified in order to achieve the “lite” behavior, and ends with a description of how the application controls the behavior.

4.1 Linux Kernel Overview

Linux[Lin00] is a free[Sta92, Fou91] operating system originally created by Linus Torvalds in the early 1990s. Since then, numerous volunteers have contributed code and ideas to the project. The name “Linux” originally only meant the kernel, but is used nowadays to reflect both the kernel and the applications around it. Some people want to call this “GNU/Linux” instead[Sta97], because many of the applications belong to the GNU[Sta92] project. The kernel and applications are bundled by different companies. These bundles are called distributions. The reader might have heard about Debian, RedHat, Mandrake, SuSE or Slackware. Most of the software is the same in all the distributions. The differences are in the way it is packaged and in the installation procedures. Some companies also provide commercial support for their distribution, which is said to be the primary source of income (as anyone is free to copy the software, money needs to be made somewhere else).

The core of Linux, the kernel, is distributed in source code from[The00] and mirrors, and is about 75Mb unpacked[1]. The code is divided into many directories, for example `Documentation`, `arch`, `drivers`, `fs`, `include`, `kernel`, `mm` and `net`. Most of the names are self-explanatory, but `fs` means file systems and `mm` means memory management. Since we are interested in the networking area, the relevant files are found in `net/ipv4`, which contains the IP, TCP, UDP and ICMP functionality.

4.2 The Modification

The modification is done at the point in the TCP stack where checksums of incoming packets are compared to the calculated checksums. Figure 4.1 depicts a scenario where a byte stream is divided into TCP segments (or packets) by the stack, sent over the Internet, and at some point transmitted over an unreliable wireless link.

This link introduces bit errors at random positions, which are retained by the TCP Lite stack, so the receiver will experience these in its re-assembled byte stream. The original code looks something like this:

```
if (pkt->csum == calc_csum(pkt)) {
  /* accept packet */
} else {
  /* discard packet */
}
```

Adding 'lite' functionality, this code will look like this:

[1]Valid for the 2.2.x series of the kernel.

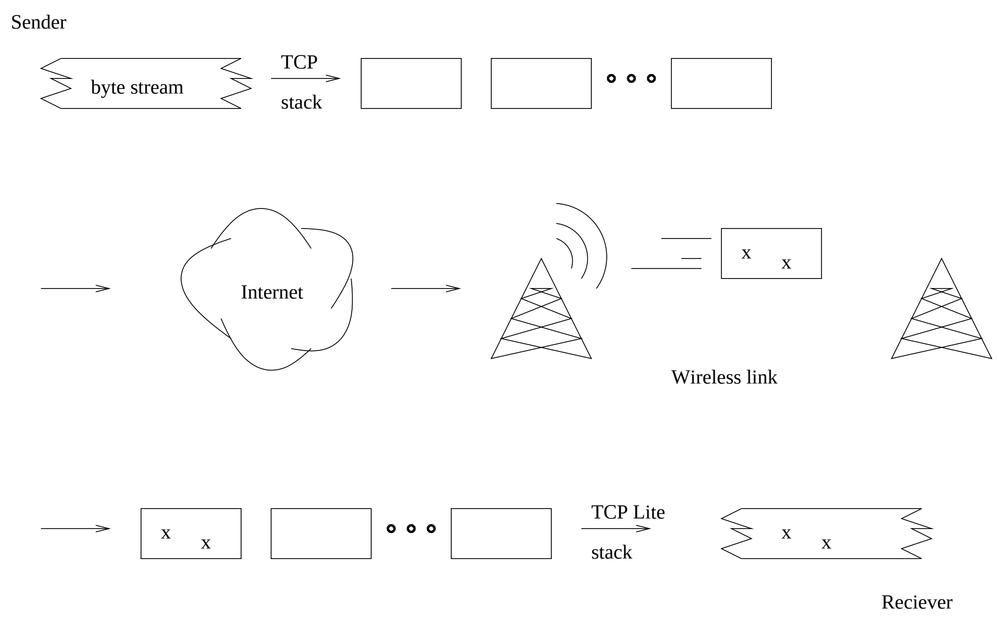

Figure 4.1: Biterrors let through

```
   if (pkt->csum == calc_csum(pkt) ||
       litecheck(pkt)) {
/* accept packet */
   } else {
/* discard packet */
   }
```

where the `litecheck` function looks up the socket belonging to the packet, decides whether it is acceptable and possible to deliver the packet although the checksum is wrong[2].

Note that this is a general discussion about where the modification should take place and not limited to Linux. Other operating systems are expected to have a similar structure and hence be easy to modify.

[2]It must be wrong because of C-style short-circuit evaluation of boolean expressions.

4.3 Kernel Modification

As mentioned in Section 4.2, the place to introduce TCP Lite in the kernel is where the checksum of the packet is calculated. By enabling the network debug functionality of the Linux kernel and sending a packet with wrong checksum to the machine, a “bad checksum” message is logged. Searching for where this message is printed from reveals that the checksum is verified in `net/ipv4/tcp_ipv4.c` by the `tcp_v4_check()` function. If this function returns zero, the checksum is correct, otherwise it is not.

The ’lite’ functionality is added right after this verification. If the ’lite’ flag is set for the active socket, then the packet is treated as if the checksum had been correct. If the ’lite’ flag is not set, the packet is discarded as would have been done originally.

4.4 Socket Controlling

To enable/disable ’lite-mode’, the system call `setsockopt` is used. ’Lite’-mode is implemented in the following steps, using setsockopt;

- The code to handle socket options is located in `net/ipv4/tcp.c`. Essentially it is one big switch/case statement, so the technique to add a new socket option is to define a new magic number and add a `case` statement here.
- The socket option magic numbers are defined in `include/linux/socket.h`.
- There must exist a place to keep the ’lite’ flag on a per socket basis. The `sock` struct holds meta information about individual tcp streams, for example the remote/local address and port, and is chosen to contain the ’lite’ flag.

Setsockopt is used in the following way by the application. First, a socket is opened and connected as usual. Then `setsockopt()` is called:

```
int val = 1; /* 1 = on, 0 = off */
setsockopt(socket, SOL_TCP, TCP_LITE, &val, sizeof(val));
```

A full program which opens a socket, turns on TCP Lite and just discards the data it receives can be found in Appendix C.

Chapter 5

Experimental Environment

This chapter describes the environment in which the TCP Lite implementation has been tested. It begins with a motivation for doing the experiments, followed by descriptions of how the wireless link is emulated. Then the setup and parameters of the emulated network are explained, and the chapter ends with a description of how measurements on the experiments have been collected.

5.1 Experiment Motivation

As mentioned in earlier chapters, the idea behind TCP Lite is that it should provide better throughput and delay than unmodified TCP. To support this statement, a number of experiments has been performed. A prerequisite for the experiments is that the TCP stacks get erroneous data. This is detected with the use of the TCP checksum field. The checksum is calculated and set by the sender, and verified by the receiver. If the checksum is wrong it is an indication that the packet content has changed along the way. With this in mind we find two ways to create false checksums:

- Change the payload of the packet.
- Change the checksum while keeping the payload intact.

The first approach can be used to measure the user perceived quality loss versus delay time. This also requires applications that can tolerate erroneous data to an extent. However, this is very application dependent, as different applications have varying error tolerance. Therefore the second approach where only the checksum is modified was chosen. This focuses more on the protocol behavior than the application, which better fits the scope of this thesis. Its implementation is described in Section 5.4.2.

5.2 Characteristics of a Wireless Link

Compared to a wired connection, a wireless connection can not offer the same bandwidth nor quality because of the limited frequency band allowed. The physical characteristics of radio waves are also a source of disruption. Radio waves are not directed[1], but spread in circles around the omnidirectional antenna, like rings on water. One wave will reach the receiver first, followed by fainter reflections caused by items in the surroundings. This phenomenon is called *multipath fading*, i.e. the signal reaches the receiver via different paths. Another source of signal degradation is the natural dampening. Logically, the greater the distance between the sender and receiver, the harder it is to detect the signal transmitted. This is called *path loss*.

These issues lead to problems for the receiver, if the signal quality is not good enough. A '1' might be interpreted as a '0' and the other way around. This is called a bit error. The following subsections discuss how the errors are measured, how they are distributed and what is done to lessen the impact of errors.

5.2.1 Bit Error Rate

When discussing the behavior of radio based wireless links, the amount of errors occurring are measured as the number of erroneous bits divided by the total number of bits transmitted over some time period. This is called the *Bit Error Rate/Ratio* (hereafter written BER), and is usually

[1]Assuming relatively low frequencies.

expressed as a power of 10 expression. For example, 5 erroneous bits out of 10,000 transmitted are 5 out of 10^4, which yields a BER of $5 * 10^{-4}$. Common BERs for radio based wireless links are between 10^{-3} to 10^{-6}[BKVP96, KK99]. This corresponds to an erroneous bit approximately every 0.1 kbyte and 100 kbytes, respectively.

5.2.2 Error Distribution

When emulating a wireless link, the distribution of the errors is important. This is because they are normally not randomly distributed, but depends on for example fading and interference, assuming radio based communication. This gives a bursty behavior, which could be emulated with a finite state machine with probabilities for each transition, also known as a Markov chain. This technique is used by [EW99, Lar99, WM95, PGLA00], and others to emulate bit errors on wireless links. Using a Markov model, two states with different BERs are defined. One state has a low BER, while the other has a high BER. As seen in Figure 5.1, the two states are marked G and B. The state G is the "good" state that represents successful transmissions and therefore has a low BER. The state B is the "bad" state which represents error bursts and therefore has a high BER. Two state transitions are then defined, from the good state to the bad, and vice versa. A transition to the bad state is decided by the probability p, and corresponds to the beginning of an error burst. The probability q corresponds to the end of the burst. The parameters, p, q, and the BERs, are dependent upon the wireless link emulated.

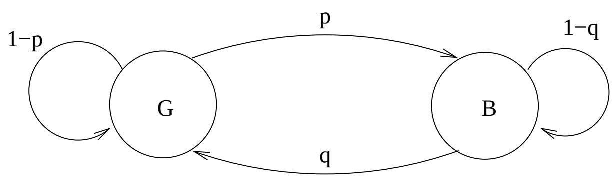

Figure 5.1: The error distribution model as a markov chain

5.2.3 Bit Errors and Interleaving

Radio based wireless links often use forward error correction to be able to correct errors at the link end-points. In order for the error correction to be as efficient as possible, the data is interleaved. This means that bits are rearranged in such a way that bursts are distributed over several frames instead of a single frame. For example, instead of sending ten frames in a row, one tenth of each frame is constructed into a new one. The loss of one such constructed frame will result in one tenth loss of each of the ten frames. However, this loss can be recovered with the error correcting codes. If interleaving had not been performed, one complete frame would have been lost instead.

5.3 Experimental Parameters

Different links exhibit different characteristics, for example different bandwidth and propagation delay. To evaluate the behavior of TCP Lite in differing environments, three link profiles representing low, medium and high bandwidth have been chosen. Note that we are not modeling their exact behavior, but focuses on the differing features. A number of loss profiles, i.e. different error burst characteristics, have also been defined. This is introduced in the following sections, along with a description of the physical network layout and configuration variables.

5.3.1 Selected Link Profiles

GSM

The Global System for Mobile Communications[MP92], GSM, is the most used system for cellular phones in the world. Normally GSM is used for voice communication, but it can also transmit data and fax. The data rate is 9.6kbit/s which is very slow compared to a dedicated wired network, which usually delivers 10 to 100Mbit/s. The delay is quite significant because of the low bandwidth. Internal experiments have shown them to be 150ms on an unloaded link. The delay rises sharply as the load increases, and has been measured to be as much as 7 seconds. GSM has a transparent and non-transparent mode. The non-transparent mode uses an error correction facility called Radio Link Protocol, RLP, to ensure more robust transmission. The transparent mode provides no error correction over the air interface. Therefore, disabling RLP may be a good way to test TCP Lite in a real environment. GSM is used as a profile because of its wide deployment.

UMTS

The Universal Mobile Telecommunications System[Mur00] is the third generation telecommunication system. It is supposed to replace GSM within the next decade, and provides both circuit- and packet-switched data communication. UMTS is also part of the IMT-2000 system. In the beginning, data rates from 64kbit/s are offered, but the technology supports up to 2Mbit/s. No figures on UMTS TCP delays have been found, but we have estimated them to half of the GSM delay. UMTS will presumably be commercially launched later this year (2001). The UMTS parameters are used as a profile in this experiment because it is believed to be the most widely deployed mobile system in the following years.

IEEE 802.11b

One of the issues that has slowed down wireless LAN adoption is the limited throughput. This has changed with the IEEE 802.11b standard[Gro01]. It provides up to 11Mbit/s transmission rate, and builds upon 802.11 which was limited to 2Mbit/s. The delay for this profile was chosen to be 10ms. 802.11b is included as a profile because of its wide deployment and to contrast the lower bandwidth profiles with a much higher bandwidth.

5.3.2 Loss Profile Parameters

In the experiment, the three link profiles mentioned above defines the bandwidth and delay. In addition to this, different bit error patterns are chosen and tested against each profile. Although the BER is an average value calculated over a number of bits, the actual errors occur in bursts. The error bursts lasts some time, and then stops. The bursts give rise to erroneous bits in the packet, causing the whole packet to be classified as having a bad checksum. Therefore, instead of choosing and emulating specific BERs, we specify the average time spent in the "good" and "bad" state of the markov model, during which individual packets are having their checksum changed. Also, since there is no difference between the "damage" caused by one wrong bit compared to some hundred bits (either the checksum is good or bad), the derived BER becomes somewhat insignificant in this particular case. A discussion about this can be found later in this section. Table 5.1 lists the chosen time parameters for the markov model. These are derived from other work in the wireless emulation field, [EW99, BKVP96, CLM99, KK99] among others. The table also shows a calculated "bad packet rate", which is the average percentage of the packets that will be affected by a burst, assuming a constant packet flow.

Loss profile	T_B	T_G	Bad Packet Rate
0	0	∞	0
1	30ms	2.0s	1.5%
2	30ms	10.0s	0.3%
3	60ms	2.0s	2.9%
4	100ms	2.0s	4.8%
5	100ms	5.0s	2.0%
6	200ms	5.0s	3.8%

Table 5.1: Parameter values used in the experiments

As link errors are normally specified in a number of bits, although we are more focused on the number of packets with errors, a reflection over the relation between packet error rate and bit error rate is appropriate. The maximum transmission unit, that is, the maximum size of each TCP packet was set to 576 bytes. This corresponds to 4608 bits. Assume a 5 percent packet loss and that one bit caused corruption in these 5 percent. This could then correspond to $0.05/4608 = .0000108506 = 1.1 * 10^{-5}$ BER, which is not that much. On the other extreme we assume that all bits in the packet caused corruption. This leads to a BER of $0.05 = 5 * 10^{-2}$, which is quite severe.

5.3.3 Experiment Network Topology

The network topology used in the experiments can be seen in Figure 5.2. The network consists of a sending host and a receiving host, whose traffic is routed through a FreeBSD "dummynet" router, carrying the modifications discussed in Section 5.4.2.

The packet length (MTU) used in all experiments was set to 576 bytes, as this is de facto used over slower links, and also specified by [Pos83]. The TCP receiver window at the receiver was not modified, and defaulted to 32kb. This is more than twice as large as the largest bandwidth-delay product presented in Table 5.2. The table also shows the amount of data transferred in each profile. It differs from profile to profile, because a common size would be impractical. For example, if all profiles had used a 60Mb transmission, it would take a very long time for the GSM profile to complete (about 14 hours). At the other end, the 802.11b profile would process a 50kb transmission in tenths of a second, which would make the results unreliable. That is, small natural variations would cause big impact compared to the total transmission time. The queue size in "dummynet" (described in the next section) was not changed, and defaulted to 50 slots.

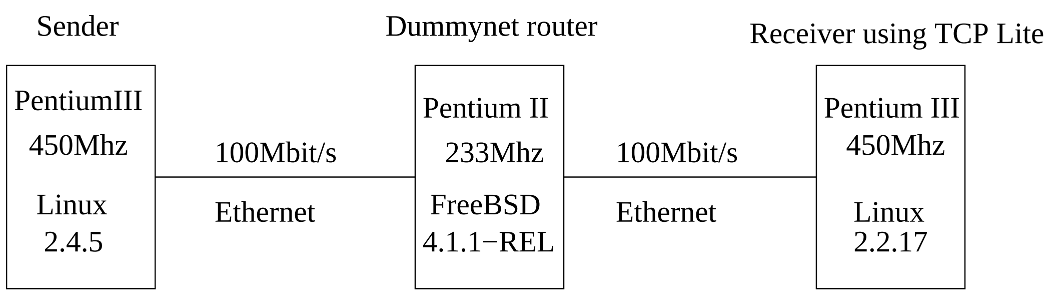

Figure 5.2: The experiment network topology

Profile	Bandwidth (kbit/s)	Delay (ms)	BW*Delay	Data transferred
GSM	9.6	150	0.2kb	50kb
UMTS	384	70	3.3kb	3Mb
802.11b	11000	10	13kb	60Mb

Table 5.2: Characteristics of the link profiles

5.4 Error Emulation Setup

An interesting scenario for the experiments would be to run on real hardware in a realistic environment. However, when doing initial experiments it is usually better to be able to have close control over the environment. This enables us to better control the parameters used, and better understand the implications of modifying parameters. Another reason for doing emulations was that a real environment was not available to us. Emulation/Simulation can be done purely in software[2], or in a mixture of software and hardware. The software then simulates the wireless environment. The latter approach was chosen, as this has been used earlier in the data communications research group at Karlstad University. The research group has most experience from the network emulation software NIST Net[Gro00]. However, recently the *dummynet*[Riz97] network emulator included in FreeBSD has been used instead, since it uses a better technique to achieve bandwidth throttling than NIST Net. This, along with the need to gain more experience with dummynet, led us to choose it for the experiments. The following subsections introduce dummynet and how bit errors are is implemented in dummynet.

5.4.1 Introduction to Dummynet

Dummynet[Riz97] is a network emulator, which operates on a physical network. This means that it will be more accurate than a simulated network (i.e. software only), while being easier to setup and maintain than a full network. It works by using a concept named *pipes* where traffic routed through dummynet enters one or more pipes and is modified according to the rules of the pipe. A pipe is identified by an order number and parameters to match against incoming packets. The order number decides the order in which packets are matched against the rules. For example, a pipe can be configured to a specific delay and bandwidth, to emulate a modem connection or a congested network. Figure 5.3 shows two example pipes; the leftmost pipe has a long delay and has a constraint on the bandwidth, whereas the rightmost pipe has a shorter

[2]For example NS, the Network Simulator[MF99].

delay and greater bandwidth. Since dummynet is deployed on a real network, ordinary traffic generators and packet inducers can be used. An implementation of the dummynet emulator is integrated in the FreeBSD[Fre00] operating system kernel, and is used in the experiment setup. However, the basic functionality does not cover bit errors. Bit errors were implemented in the dummynet module, as described in the next section.

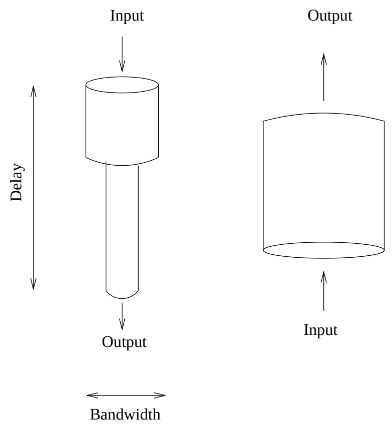

Figure 5.3: How dummynet operates

5.4.2 Bit Errors in Dummynet

Without modifications, dummynet provides an 'all or nothing' approach to losses, in the sense that it either drops a packet or does not. We want a behavior where instead of dropping the packet, bit errors would be created in the packet. To implement this, changes need to be made in the file `/sys/netinet/ip_dummynet.c`, which provides the dummynet functionality. The code is quite straightforward to understand and modify, as intended by its author.

Incoming and outgoing packets are sent through the `dummynet_io()` function, which determines whether it is to be dropped or not. Instead of sending packets to the usual `drop` routine, they are sent to a new `biterror` routine. The routine checks the state of the markov model, and creates an error accordingly. The packet is then passed along. To be able to control the parameters of the markov model without recompiling the kernel, these can be set via the `sysctl` interface.

The function `dummynet_io()` gets access to the packet data via a variable of type `struct mbuf`, defined in `/sys/sys/mbuf.h`. We will not go into detail about this structure, since it is very well described in [SW95], but it can be thought of as a general memory buffer.If one buffer is not enough to hold the data, several memory buffers are linked together and forms a linked list. Since an `mbuf` can contain any kind of data, it must be mangled into the right structure. This type must be known, and in our case we know it is a TCP packet because this is the only protocol allowed to enter the pipe, as determined when configuring it. The function `mtod`, mbuf to data, does this conversion by essentially casting a pointer to a specified type. So in our case, to get access to the TCP header, this technique is used:

```
mytcpheader = mtod(m, struct tcpiphdr*);
mytcpheader->ti_t.th_sum = 0; /* create false checksum */
```

The TCP header is extracted from the memory buffer, and the checksum in the header is changed to emulate that an error has occurred in the packet. Further development of the error generator will do it the real way instead, by modifying the data contained in the mbuf chains. However, changing just the checksum allows us to test the modification in a controlled manner and make sure the TCP Lite modification works as intended. This is possible since we are investigating how the protocol behaves, and not individual applications. In order to test applications on top of TCP Lite (or to test a more advanced version of TCP Lite), a full implementation that actually changes the data is needed.

5.4.3 Error Distribution Implementation Problem

In the initial implementation, the packets and their sizes were used to determine when state transitions in the markov chain should occur, meaning that large packets had a greater chance to change state than small ones. This had the effect that small packets, used for example in the TCP connection setup, did not have a high probability to change state. In the case where TCP Lite was not used, and assuming we were in the bad state, this led to a retransmission of the packet. Since the packet was small, it was unlikely to change to the good state. Therefore the packet was dropped, and the procedure repeated. Due to the binary back-off of TCP to avoid congestion, this can lead to very long transmission times. Eventually the state changes, packet sizes increase and also the probability to change state more often. As these fluctuations were not considered realistic behavior, other approaches were looked into.

So, how should the error distribution be implemented? To decide this, we need to know *why* errors occur. One possible cause is mobility, which gives radio interference and fading. Even if the device is in a fixed position, changes in the surroundings such as moving cars, can be a source of problems. This means that the error distribution does not depend upon the amount of data transmitted, but is a function of time. Hence, depending upon when packets are sent, different BERs can be experienced. Figure 5.4 shows a scenario where all the packets are unaffected, while in the second scenario, many packets gets errors, because the first packet was sent somewhat later.

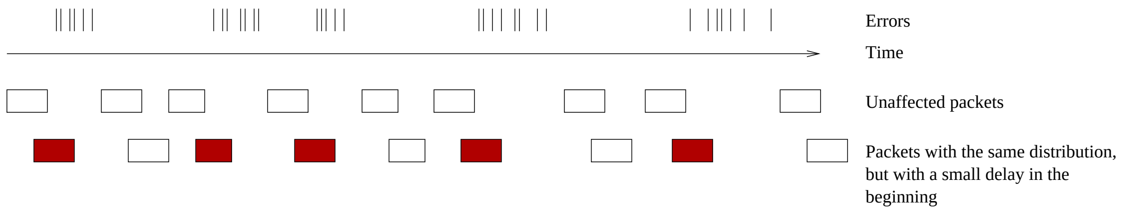
Errors
Time
Unaffected packets
Packets with the same distribution, but with a small delay in the beginning

Figure 5.4: Differences in error distribution depending upon time

To emulate the time dependency approach, the probability of changing state must be calculated continuously at discrete time intervals. This gives an overhead, especially if few packets are being transmitted. It is therefore desirable to calculate the current state at the arrival of each packet. The time of the last state transition and time of the packet arrival could be used to determine if a transition should take place or not. We thought this calculation would be too complex. For example, if a long time period has passed between two packets, how can the number of transitions be estimated? Should we keep the last state, and calculate a probability to change state, depending upon the last state and elapsed time? Instead, the first approach was chosen, despite the overhead. A periodic calculation is performed at each timer interrupt in the kernel. This interval defaults to 100hz, i.e. every 10ms.

5.4.4 Bandwidth Limitation Issues

One thing to consider when introducing errors and bandwidth limitations within the same machine is to be careful on which “side” (input or output) the errors are induced. Assuming the errors are dependent on time, the scenario in figure 5.5 can occur. The error burst length is the same, but if errors are generated on the input side with high bandwidth, more packets will be affected compared to if errors were generated on the output side.

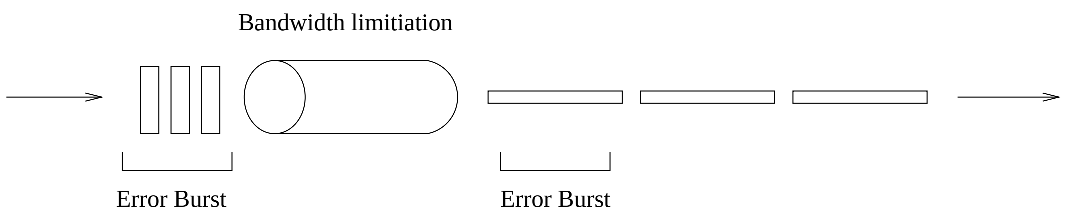

Figure 5.5: Bandwidth limitation problem

This issue arises because a high-bandwidth link can deliver more packets during a time period than a low-bandwidth link. The solution is to make sure that errors are not generated until after the bandwidth limitation has taken place. This can be done by separating the error generation and bandwidth limitation into different physical machines, as illustrated in Figure 5.6.

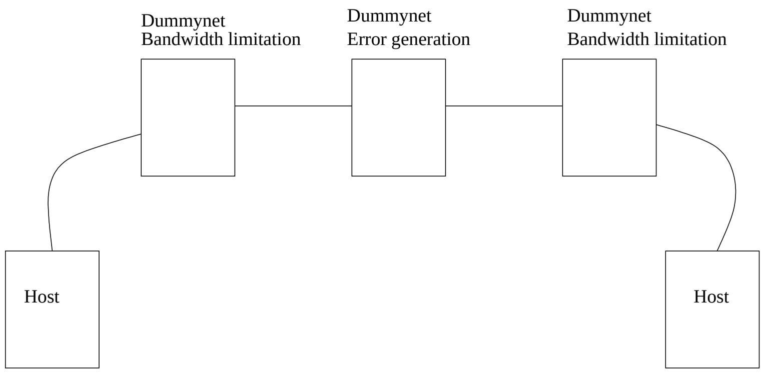

Figure 5.6: One solution to the bandwidth limitation problem

A second option is to set up the software to explicitly limit the bandwidth first, and then apply error generation. For example, dummynet has an option to let a packet pass through more than one matching pipe. Using this approach, we first create a pipe whose only purpose is to incur bandwidth limitation. Then, the pipe which induces the errors is created with a higher order number. Finally, dummynet is told to use all matching pipes by setting the `sysctl` variable `net.inet.ip.fw.one_pass` to zero.

A third option is to differentiate the pipes, by determining incoming and outgoing traffic. The qualifiers `in` and `out` can be appended onto any rule. This way, the bandwidth limit rule is applied to incoming traffic, and the error inducing rule is applied to outgoing traffic. This method was chosen for the setup, as it was deemed the easiest to configure and maintain.

5.5 Measurement Collection

When doing experiments of any kind, measurements must be taken to examine the outcome. Otherwise, there would be no point in doing the experiment in the first place. In our case, we are interested in measures related to the transmission of data over the TCP protocol. One idea is to measure the time to transmit a fixed amount of data at the sender side. Another idea is to use a network analyzer to examine the traffic sent over the network. This would provide a better measurement compared to the timing method, because buffering in the network stack makes it hard for the application to know exactly when all data has been received after the connection has been closed.

We have decided to use the `tcpdump` utility to capture the traffic at the receiving host. This collection is later analyzed with the `tcptrace` utility[Ost01] that provides thorough information about captured traffic. An example of a tcptrace analysis can be found in Appendix D. These analyzes are then processed by custom written scripts and fed to a plotting utility to create graphs.

Chapter 6

Experimental Results

This chapter presents the results obtained from the experiments. First, the expected results are stated. Following this is a description of how the experiments were carried out in a more technical sense. Then the results are presented and discussed in the form of graphs. Finally the conclusions are drawn.

6.1 Expected Results

The results of the experiment are expected to show that connections utilizing TCP Lite will have fewer retransmissions than unmodified TCP. It is further expected that the throughput will be higher because the sender does not reduce its sending rate in response to bit errors, when talking to a TCP Lite enabled receiver.

6.2 Experiment Execution

As mentioned in the earlier chapter, the network topology consists of a sender, an error inducer and a receiver. The sender is in control of setting the various parameters by logging on to the other machines. The error inducer has ten scripts: three which control the link profile, and seven that control the error model parameters. The receiver runs an application[1] that is able to enable or disable TCP Lite in the TCP stack. The purpose of the application is to wait for connections on a socket, and to read and discard data sent to it. The receiver also runs the network packet capturer `tcpdump` for logging of the traffic.

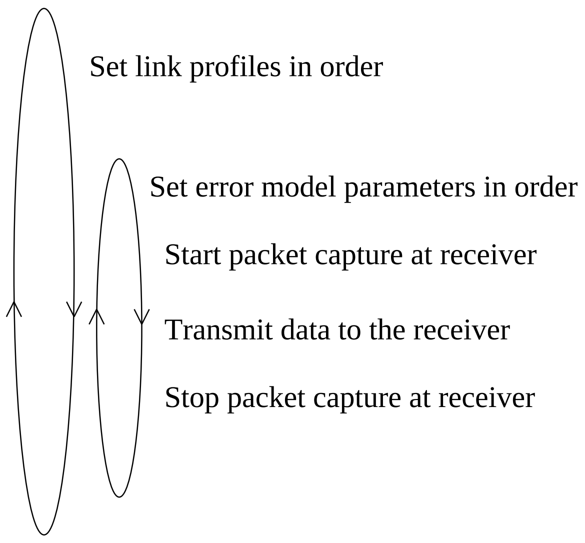
Set link profiles in order
Set error model parameters in order
Start packet capture at receiver
Transmit data to the receiver
Stop packet capture at receiver

Figure 6.1: Visual view of the experiment execution at the sender

As seen in Figure 6.1, the experiment execution consists of a script with two loops running on the sender. In the outer loop, the link profile is set by logging onto the error inducer and executing the corresponding script. The link profile contains the parameters shown in Table 5.2. In the inner loop, the loss profile parameters (Table 5.1) are set on the error inducer, and the packet capturer is started on the receiver. Depending on the link profile, a given amount[2] of data is transmitted to the application. This transmission is repeated 30 times to collect data on more than one transmission, in order to get a better statistical measure. The packet capturer is then stopped, and the loop continues. To be able to compare TCP Lite to regular TCP, two runs of the process described above is performed. One with TCP Lite enabled in the application, and one with TCP Lite disabled. When the process is finished, there is a set of packet capture logs. The

[1]The applications can be found in Appendix C.
[2]See Table 5.2 for the specific numbers.

logs are then examined and the results are presented in the sections below. To show that the TCP Lite modification does not affect TCP if it is disabled, experiments comparing TCP Lite disabled to an unmodified kernel has been conducted as well. They can be found in Appendix A.

6.3 Experiment Analysis

Usually when comparing two experiments, a number of runs are made of each experiment which yields different values, *samples*. The mean value of each experiment can then be compared, and conclusions drawn. However, this technique does not reveal anything about the distribution of the samples. Are perhaps all equal to the mean value, or are they spread out? Therefore, we wanted a better way to present the data. We think that showing all samples provides a better understanding of the distribution, but lots of dots can be hard to perceive. Instead, we use the *quartile* concept from descriptive statistics. This means that all the samples are ordered and partitioned into four quartiles. The first quartile contains the lower 25 percent of the samples, the second quartile is the median value, and the third quartile contains 75 percent of the samples. This gives a better understanding of how the samples are distributed, compared to the mean value. As seen in Figure 6.2, the line at the bottom represents the first quartile (marked 25%), the "box" represents 25-75 percent of the samples, and the upper line represents the third quartile (also marked 25%). The median and mean values are also indicated in the graph. With this representation, lines and boxes centered around the median reveals that the variance of the samples is small. A small box and a long line indicates that a few (or even one) experiment run differed a lot from the other, which in turn had a low variance.

The x-axis in Figure 6.2 is numbered from 0 to 6. These numbers correspond to the loss profiles in Table 5.1. Within each loss profile there are two bars that correspond to the description in the title. In this example, the leftmost bar within each loss profile corresponds to measurements with TCP Lite enabled. The rightmost bar corresponds to measurements with TCP Lite disabled.

As mentioned in the measurement collection section above, the traffic of each experiment was

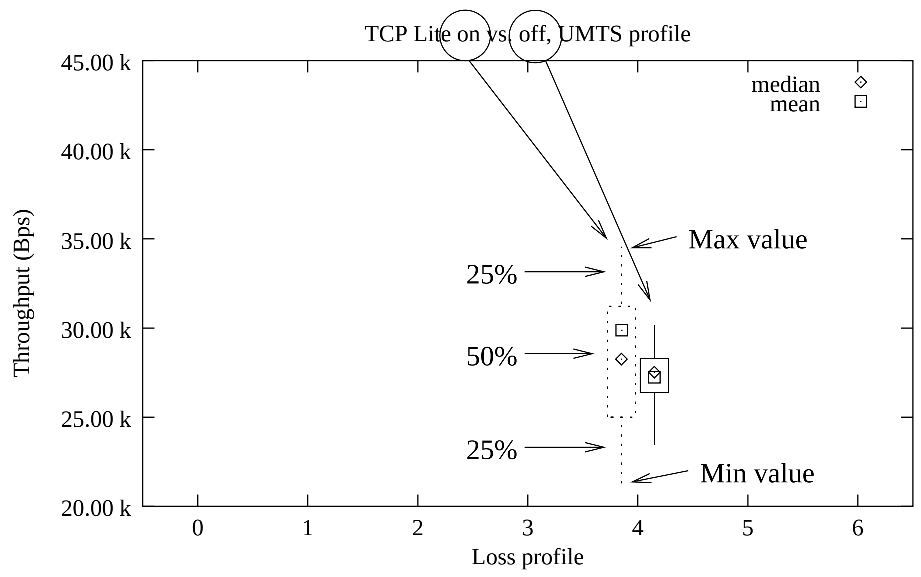

Figure 6.2: Example graph with explanations

captured at the receiver. Each of these logs were then analyzed with the `tcptrace` utility. The output was processed with custom written scripts to produce the basis for the graphs presented below.

6.3.1 Throughput Graphs

This subsection presents an analysis from a throughput perspective.

GSM Profile The throughput graph of the GSM profile experiments can be found in Figure 6.3. We see that for loss profile 0, the results are the same whether TCP Lite is enabled or not. This is the expected result, as this profile should generate zero errors. This shows that TCP Lite behaves like unmodified TCP when there are no errors. Looking at the graph we note that the mean/median degradation varies from about 1 to 6 percent. Further, the performance degradations can be seen proportional to the "bad packet rate" presented in Table 5.1. A final observation is that the results with TCP Lite enabled shows no variance, while the throughput degradation of original TCP varies a lot. For example, in loss profile 4 it varies from 3 to 15 percent.

As a side note, in the initial experiments we experienced a case where the retransmission endured an exceptionally long time, but it has not been reproduced since. For reference, a discussion about this can be found in Appendix B.

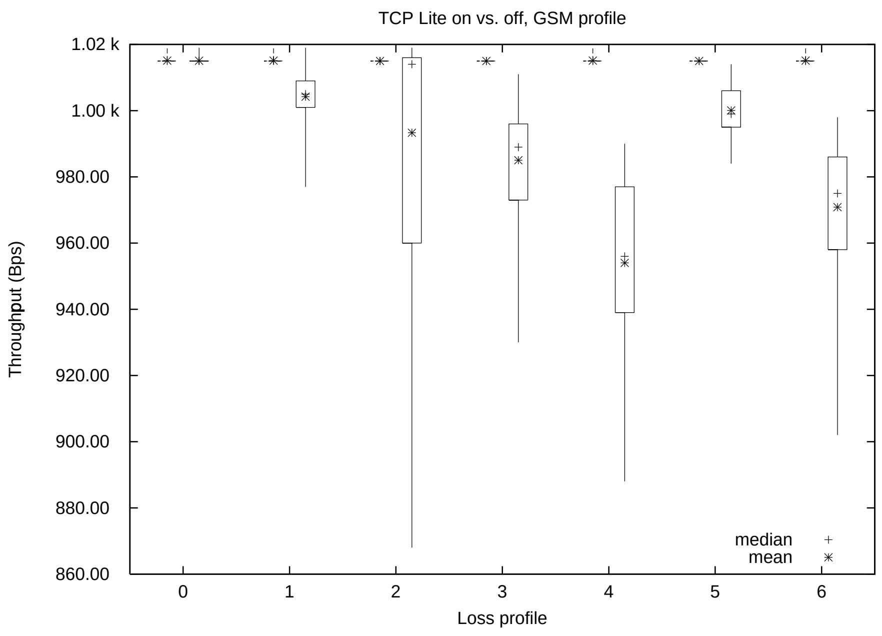

Figure 6.3: Throughput results for the GSM profile

UMTS Profile The throughput graph for the UMTS profile is displayed in Figure 6.4. As for the GSM profile, enabled TCP Lite delivers constant throughput and no variance. When disabled, the throughput degradation varies from 3 to 17 percent[3]. As in the GSM profile, we also note that the variance is large in this case. For example, a throughput degradation between 6 to 26 percent is seen in loss profile 4.

Notable is that loss profile 2 has a small degradation and variance, and does not deviate much from the throughput of an enabled TCP Lite stack. This means that this bandwidth/delay/loss profile would not benefit much by using TCP Lite[4].

[3]Considering the mean/median values.

[4]It is hard to tell how valid this statement is in a real environment, since many more factors could have an effect

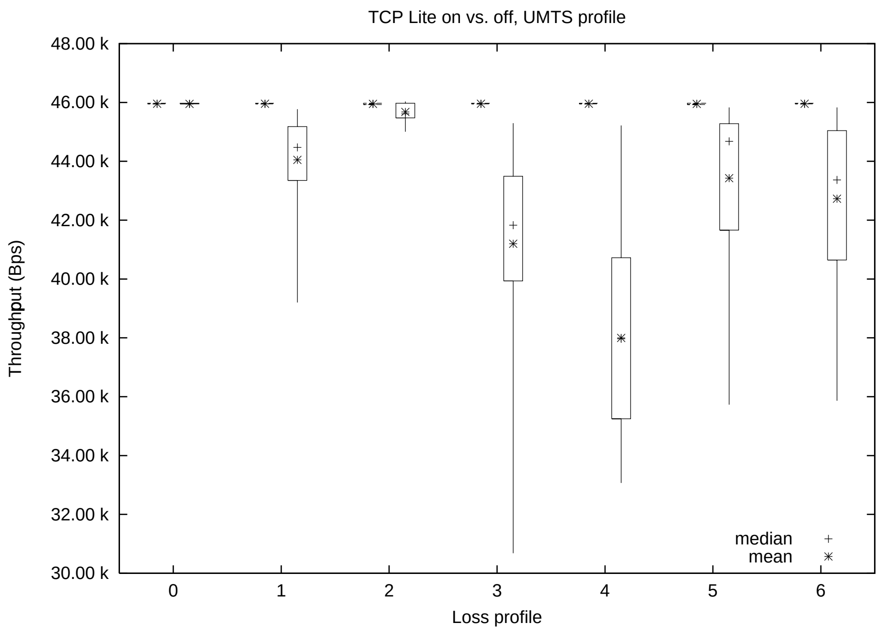

Figure 6.4: Throughput results for the UMTS profile

802.11b Profile Figure 6.5 shows the results for the 802.11b profile. As before, loss profile 0 is the same for TCP Lite enabled and disabled, as it should. Using TCP Lite gives a constant throughput with almost no variance[5]. When TCP Lite is not used, we see a general throughput degradation in the range of 20 to 56 percent. This means that in some cases, enabling TCP Lite yields a 100% throughput increase. This is surprisingly high, and is further analyzed in a later section.

These observations must be put in relation to how many retransmissions that have actually occurred. For example, a 100 percent throughput increase would not be much of a revolution if at the same time 50 percent of the packets were retransmitted (in this case equal to being corrupted). In the next subsection we will examine just how many retransmissions are being made when these improvements are achieved.

upon the result.

[5]Compared to the other link profiles, there is some variation. This is believed to occur because of the much higher throughput which is more sensitive to other traffic on the path between the machines.

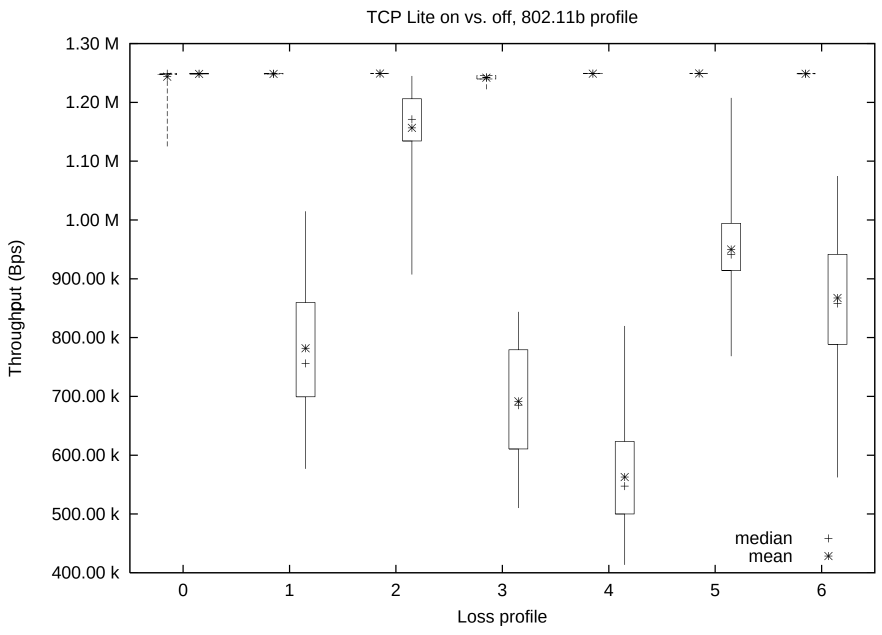

Figure 6.5: Throughput results for the 802.11b profile

6.3.2 Retransmission Graphs

This subsection presents an analysis of the experiments based upon the number of retransmissions performed. Reflections are made to the discussion about throughput above.

GSM Profile Figure 6.6 shows the number of retransmissions, i.e. the number of corrupted packets, for the GSM profile. By design, when TCP Lite is enabled there should be no packet retransmissions[6], and this is verified in the graph. In the other loss profiles, the packet loss ranges from about 0 to 5 percent. The mean and median values can be clearly correlated to the packet error rate presented in Table 5.1. Relating this graph to the corresponding throughput graph, Figure 6.3, we find that by accepting 0 to 5 percent packet loss, a throughput degradation of 1 to 6 percent is avoided.

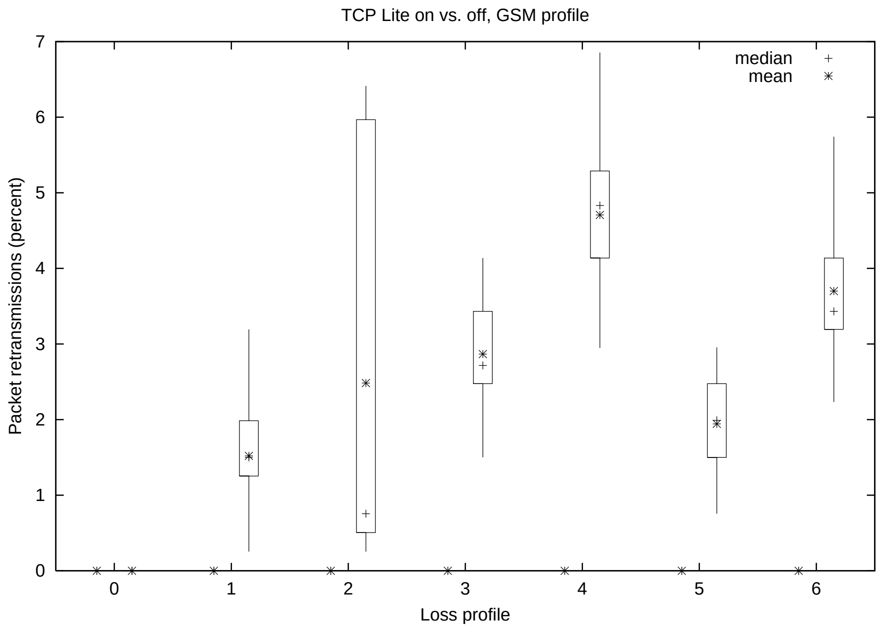

Figure 6.6: Retransmission results for the GSM profile

[6]Packet losses can of course occur from other causes in other parts of the network, for example due to congestion.

UMTS Profile The retransmission graph of the UMTS profile is shown in Figure 6.7. When TCP Lite is enabled, there are no retransmissions. When TCP Lite is disabled, the relation to the packet error rates are also clearly visible, as in the GSM profile. However as discussed in the throughput analysis for the UMTS profile, the throughput degradation ranges between 6 to 26 percent. The difference in degradation between the GSM and this profile has been found to be that due to the higher bandwidth, sometimes a whole window (congestion/receiver window) is lost, causing TCP to use the slow start mechanism. When a low bandwidth is used, not many packets are affected by an error burst, and the packet is resent with selective acknowledgements. As seen in the next paragraph, 802.11b is even more affected by the lost window phenomena.

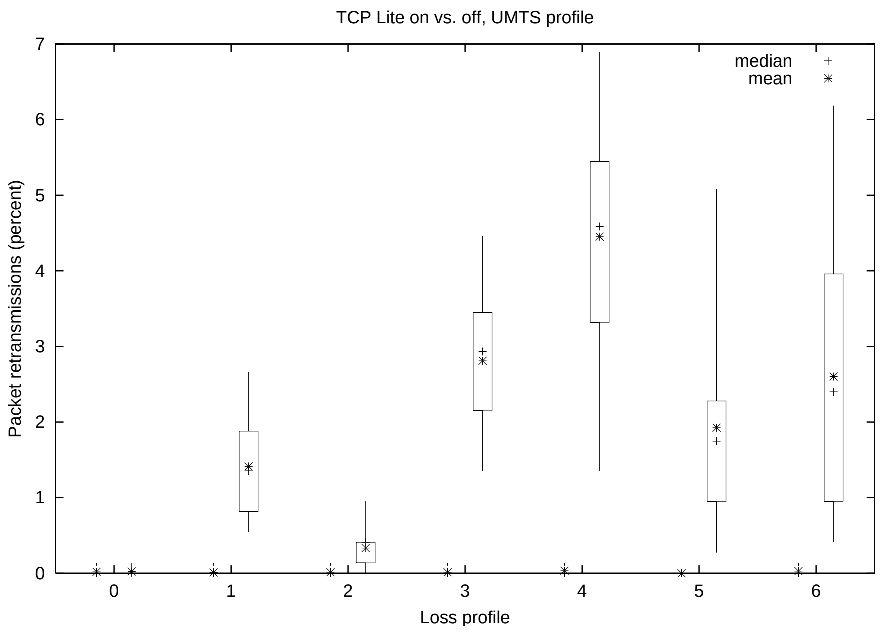

Figure 6.7: Retransmission results for the UMTS profile

802.11b Profile In the previous section, we concluded that the throughput in the 802.11b profile was severely degraded. We questioned how many retransmissions that would cause such a degradation. As seen in the retransmission graph for the 802.11b profile, Figure 6.8, we have about 0.2 to 1.0 percent packet losses. This should be compared to 0 to 5 percent for GSM and UMTS. However a bit surprising, the conclusion is therefore that in this profile, TCP Lite would be quite useful, as a low percentage of packet loss causes a large degradation in throughput. It should be noted however that when TCP Lite is enabled, higher packet losses will be experienced[7], because the losses contribute to their own reduction. Analysis in a later subsection explains this behavior.

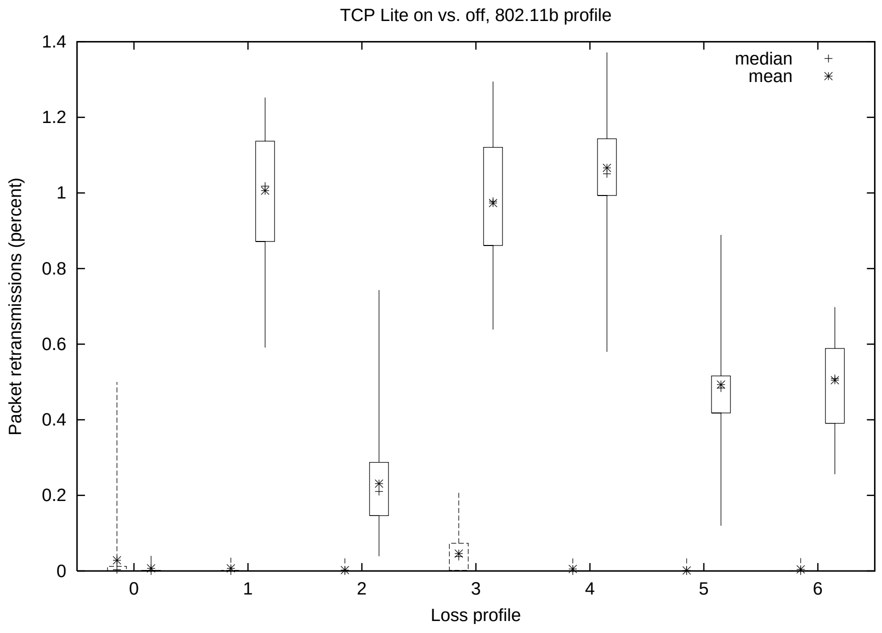

Figure 6.8: Retransmission results for the 802.11b profile

[7]Probably 0 to 5 percent.

6.3.3 Throughput vs Retransmission Graphs

In this subsection, the throughput and the number of retransmissions (equal to the packet loss) are plotted against each other. This gives a better view of how throughput is degraded when more and more packets are lost. For reference, the dots are marked with their respective loss profile, but since the discussion is on a more general level this is not further analyzed. The graphs do not include TCP Lite enabled transmissions, as they are independent of the packet loss. This can be seen in the previous retransmission graphs, or thought of as belonging to the highest throughput.

GSM Profile Figure 6.9 shows the throughput versus the number of retransmissions for GSM. We note that there seems to be discrete intervals on the percentage axis. This is found to result from the relatively low number of packets transmitted (about 400). This means that a packet loss percentage can only occur in quarters of percent, hence the feeling of discrete intervals. Further examining the graph, we see that throughput is not affected much by packet loss. At about 5 percent packet loss, we have 6-7 percent decrease in throughput. Another interesting phenomena is the vertical spread. For example, at 4.5 percent packet loss we have a span from 4 to 11 percent in throughput degradation (counting extremes).

UMTS Profile Figure 6.10, showing the throughput and retransmissions for the UMTS profile, again confirms that throughput decreases as packet loss increases. Compared to the GSM discussion above, this profile is more sensitive to packet loss, with about 23 percent decrease in throughput at 5 percent packet loss (extrapolated value).

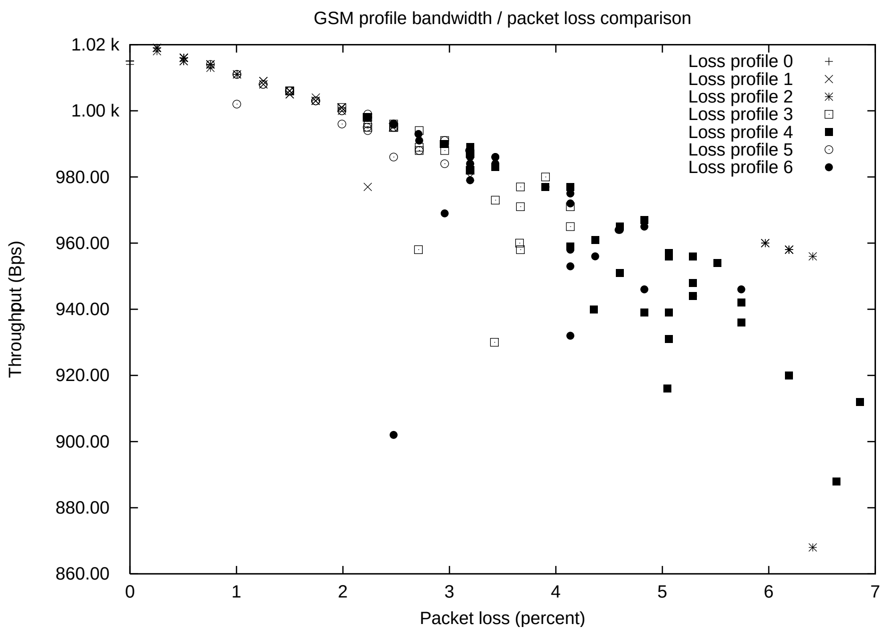

Figure 6.9: Throughput vs. retransmissions results for the GSM profile

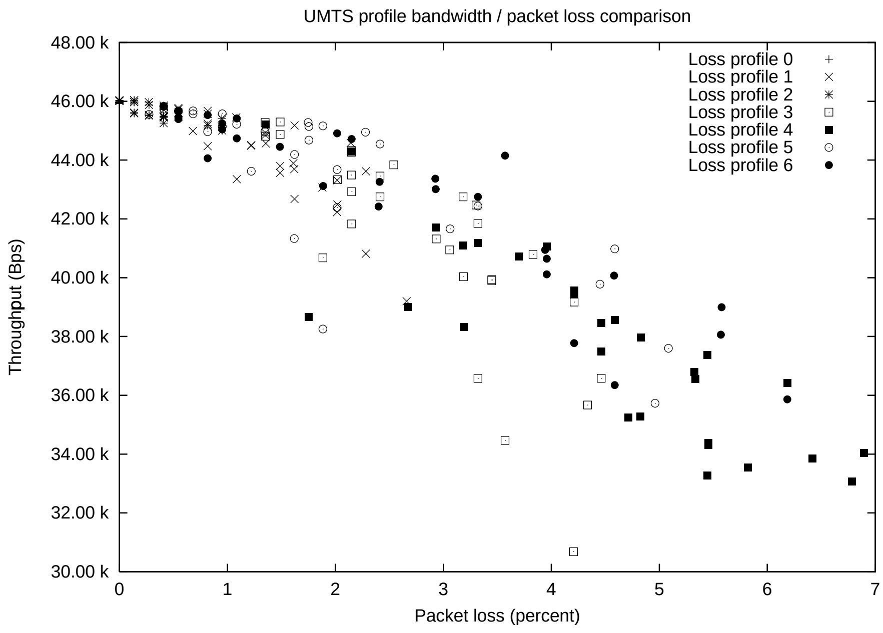

Figure 6.10: Throughput vs. retransmissions results for the UMTS profile

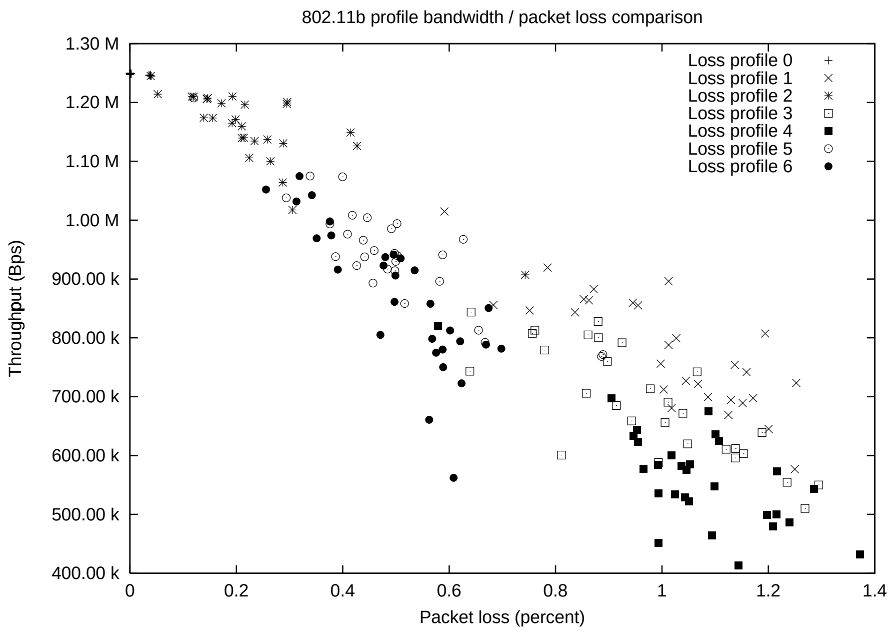

Figure 6.11: Throughput vs. retransmissions results for the 802.11b profile

802.11b Profile In Figure 6.11, which shows the throughput and retransmissions for the 802.11b profile, we see that already at 1 percent packet loss, regular TCP has half the throughput of TCP Lite. Compared to the other two profiles, this means that the throughput in this profile is very sensitive to packet losses. Again worth noting is the spread of the throughput at the same packet loss. For example, at 1 percent packet loss the degradation in throughput ranges from about 28 to 64 percent. As previously stated, the major degradation with low retransmissions was a bit surprising, and is analyzed in the next section.

6.3.4 General Analysis

A general conclusion when examining the graphs is that TCP Lite performs optimally in every loss profile. This is not so strange, since TCP Lite never performs retransmissions or lowers its throughput if the packet is corrupt. We can also see that higher bandwidth is more sensitive to packet losses than lower bandwidth. For example, the throughput in the highest bandwidth profile is halved at around 1 percent packet loss, which was a bit surprising. To explain this behavior, the dump files have been examined, to see what happened with the transmission when an error burst occurred.

Figure 6.12: Examination of errors in the GSM profile transmission

In Figure 6.12, the `xplot` graph of a GSM profile transmission is shown. The capture was done locally at the receiver, and the selected time frame shown in the figure is about 20 seconds. The vertical bars with arrows represents a segment received, and the “stairs” represent an acknowledgement sent for all segments up to a specific sequence number. The “S” marking represents a selective acknowledgement, that is, information about a missing segment at the receiver. The segment marked with an “R” is a retransmitted segment. Even though we see no reason for a retransmission from the graph, it still takes place. This is because the `tcpdump` tool captures the packets from the ethernet device. The packet has instead been discarded at the TCP layer, due to an erroneous checksum. This cannot be seen directly, but is understood from the selective acknowledgements sent and the later retransmission.

From the discussion above, we conclude that the error burst in relation to the bandwidth causes one packet to be corrupted and retransmitted. As can be seen in the figure, this single retransmission does not really affect the throughput, because of selective acknowledgements.

Figure 6.13: Examination of errors in the 802.11b profile transmission

On the other hand we have the 802.11b profile, seen in Figure 6.13. The time frame shown, about 1.5 seconds is shorter than in the other example, which should be taken into consideration when looking at the figures. Because of the high throughput, many packets are hit by the error burst. This causes a whole window (receiver or congestion window) to be lost, and the sender stops transmitting because no acknowledgements are sent back. After a timeout, the sender begins a “slow start” phase, which causes a delay in the transmission, and therefore lowers the total throughput. Therefore, the distribution of the errors cause the throughput of the higher bandwidth profile to degrade more than the low bandwidth profile, despite having fewer retransmissions. It could be argued whether the combination of this high bandwidth and these types of errors are a realistic combination, so care should be taken if the results are to be interpreted in a real environment.

Looking at the throughput graphs, we see that there generally is a large spread of the samples. The quartiles are about the same size, which indicates that samples are very spread out in the interval, rather than centered around the mean or median value.

What also needs to be kept in mind is that we are dealing with *packet losses*. As discussed earlier, even one single bit error will lose the whole packet. Therefore, perspective should be kept when relating the above percentages to actual bit error rates. See also the discussion in section 5.3.2.

6.4 Conclusion

Based on the analysis in the earlier section, we conclude that that our expectations set out in the first section have been met. TCP Lite achieves higher throughput in all the selected profiles, and is generally more efficient at higher bandwidths. However, it was a bit surprising that a lower bandwidth had a higher tolerance towards packet loss compared to a higher bandwidth, as this had not been thought of before performing the experiments. As found in the analysis, the cause of this was that a whole window was lost, which forced the sender to initiate a slow start. If part of the window was delivered, we believe that the result could have turned out a bit different, due to the feedback of selective acknowledgements.

Chapter 7

Conclusion and Future Work

This thesis has introduced a new approach to enhance the performance of TCP over wireless links. The technique is based upon breaking TCP semantics by allowing data that has been damaged while in transit between a sender and receiver to be delivered from the transport layer to the application layer. Since this can be seen as a more lightweight version of TCP, the protocol has been named “TCP Lite”. The requirements and limitations of TCP Lite have been analyzed. A receiver based approach was decided upon, since this has less impact on the other participants of the network compared to a full sender/receiver solution. TCP Lite has been implemented in the Linux operating system kernel. To test the implementation, an experimental environment was built, consisting of a physical network and a modified network emulator. The environment emulated wireless links with different bandwidth and delay parameters. A number of link and packet loss profile combinations were defined and tested in the environment. The results have been analyzed and suggests that TCP Lite gives higher throughput and fewer retransmissions than regular TCP. However it should be kept in mind that these improvements are at the cost of accepting damaged data.

During the work on this thesis, some issues have come up that would be good to investigate further. First off, extended measurements examining the behavior of bit errors in combination with congestion losses is needed, since a wireless receiver often talks to the sender over a wired network. As discussed in the background chapter, an integration of TCP Lite and PRTP is a natural evolution of both protocols, complementing each other. Emulations using a demo application would be a logical next step. This could for example be done with a JPEG image transcoder[GB00] that was used in the PRTP experiments[Gar00]. A solution to the problem with errors occurring in the header must also be found. Two paths are seen. The first is to try to correct the headers at the receiver, which could prove tricky depending on the location of the errors. The second is to use two checksums, one for the payload and one for the header. This would however require a modification at both ends of the communication channel.

References

[AGBS00] K. Asplund, J. Garcia, A. Brunstrom, and Sean Schneyer. Decreasing Transfer Delay Through Partial Reliability. *Proceedings of PROMS 2000*, October 2000.

[BB95] Ajay Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. *15th International Conference on Distributed Computing Systems*, May 1995.

[BBP88] Braden, Borman, and Partridge. RFC 1071: Computing the internet checksum, September 1988.

[BKVP96] B. Bakshi, P. Krishna, N. Vaidya, and D. Pradhan. Improving performance of TCP over wireless networks. *Technical Report 96-014, Texas A & M University*, May 1996.

[BSAK95] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP performance over wireless networks. *In proc. 1st ACM Int'l Conf. on Mobile Computing and Networking (Mobicom)*, November 1995.

[BV98] S. Biaz and N. H. Vaidya. Distinguishing congestion losses from wireless transmission losses: A negative result. *Seventh International Conference on Computer Communications and Networks (IC3N), New Orleans*, October 1998.

[BV99] S. Biaz and N. Vaidya. Discriminating congestion losses from wireless losses using inter-arrival times at the receiver. *IEEE Symposium ASSET'99, Richardson, TX, USA*, March 1999.

[CLM99] H. M. Chaskar, T. V. Lakshman, and U. Madhow. TCP over wireless with link level error control: Analysis and design methodology. *IEEE/ACM Transactions on Networking*, 7(5):605 – 615, October 1999.

[EW99] Jean-Pierre Ebert and Andreas Willig. A Gilbert-Elliot bit error model and the efficient use in packet level simulation. Technical Report TKN-99-002, EE, TU Berlin, Germany, 1999.

[Fou91] Free Software Foundation. GNU General Public License, available from `http://www.fsf.org/copyleft/gpl.txt`. June 1991.

[Fre00] FreeBSD. http://www.freebsd.org, Oct 2000.

[Gar00] Johan Garcia. Integrated testing - test setup. *Working Report, Karlstad University*, October 2000.

[GB00] Johan Garcia and Anna Brunstrom. Progressive parsing transcoding of JPEG images. *Proc. 7th Int. Workshop on Mobile Multimedia Communications (MoMuC2000), Tokyo, Japan*, October 2000.

[Gro00] NIST Internetworking Technology Group. NISTNet network emulation package. *http://www.antd.nist.gov/itg/nistnet/*, June 2000.

[Gro01] 802.11 Working Group. http://grouper.ieee.org/groups/802/11/, May 2001.

[HM99] R. Han and D. Messerschmitt. A progressively reliable transport protocol for interactive wireless multimedia. *Multimedia Systems, Springer-Verlag*, 7(2):141–156, 1999.

[JDHS00] L-E Jonsson, M. Degermark, H. Hannu, and K. Svanbro. RObust Checksum-based header COmpression (ROCCO). *Internet-Draft, draft-ietf-rohc-rtp-rocco-01.txt (work in progress)*, June 2000.

[KK99] J. Kim and M. Krunz. Fluid analysis of delay and packet discard performance for QoS support in wireless networks. Technical Report CENG-TR-99-119, Department of ECE, University of Arizona, August 1999.

[Lar99] L. Larzon. Efficient use of wireless bandwidth for multimedia applications. *Proceedings of IEEE workshop of Mobile Multimedia Communications, San Diego*, November 1999.

[Lar00] L. Larzon. A Lighter UDP. *Licenciate Thesis, Luleå University, Sweden*, February 2000.

[Lin00] Linux Online. http://www.linux.org, http://www.linux.com, Oct 2000.

[MF99] Steven McCanne and Sally Floyd. UCB/LBNL/VINT Network Simulator - ns (version 2). *http://www-mash.CS.Berkeley.EDU/ns/*, April 1999.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgment options, October 1996.

[MP92] Michel Mouly and Marie-Bernadette Pautet. The GSM system for mobile communications. *Cell & Sys*, 1992.

[Mur00] Flavio Muratore. *UMTS - Mobile Communications for the Future*. Wiley, 2000.

[Ost01] Shawn Ostermann. Tcptrace. *http://www.tcptrace.org/*, February 2001.

[PGLA00] C. Parsa and J. Garcia-Luna-Aceves. Improving TCP performance over wireless networks at the link layer. *ACM Mobile Networks and Applications Journal*, 2000.

[Pos81a] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760. Status: STANDARD.

[Pos81b] J. Postel. RFC 793: Transmission control protocol, September 1981. Status: STANDARD.

[Pos83] J. Postel. RFC 879: The TCP Maximum Segment Size and Related Topics, November 1983.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications, January 1996.

[Sta92] R. M. Stallman. Why software should be free, available from `http://www.fsf.org/philosophy/shouldbefree.html`. April 1992.

[Sta97] R. M. Stallman. Linux and the GNU Project, available from `http://www.fsf.org/gnu/linux-and-gnu.html`. 1997.

[Ste98] W. Richard Stevens. *UNIX Network Programming, Interprocess Communications*, volume 2. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1998.

[SW95] R. W. Stevens and G. R. Wright. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison Wesley, 1995.

[The00] The Linux Kernel Archives. `http://www.kernel.org/pub/linux/kernel/`, Oct 2000.

[WM95] H. Wang and N. Moayeri. Finite-state markov channel: A useful model for radio communication channels. *IEEE Trans. on Veh. Tech., 44(1):163–171*, February 1995.

Appendix A

Experiment Comparisons

A.1 Disabled TCP Lite versus Unmodified TCP Stack

To show that the TCP Lite modification does not change the behavior of TCP when it is turned off, reference emulations were performed. They were carried out in the same way as the original experiments, but with one difference. A TCP Lite enabled TCP stack versus an unmodified TCP stack were compared, instead of comparing with TCP Lite enabled and disabled. As seen in Figures A.1, A.2 and A.3, both measures in each profile resembles the other. As can be expected, since the errors are randomly generated, there are some minor differences.

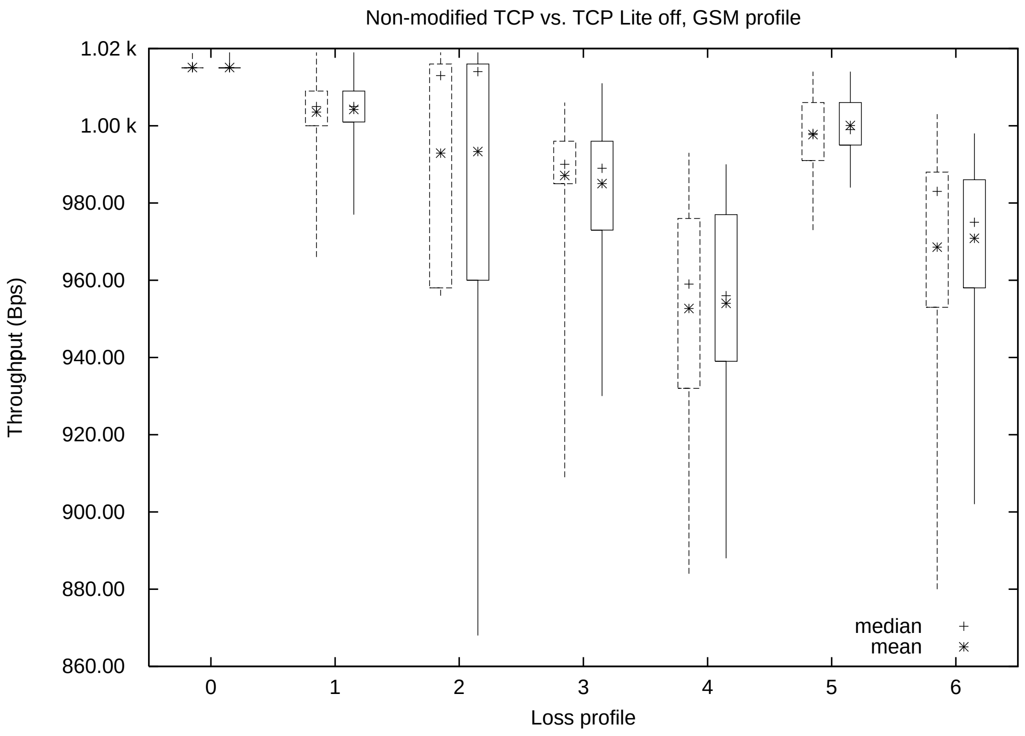

Figure A.1: Throughput comparison for the GSM profile

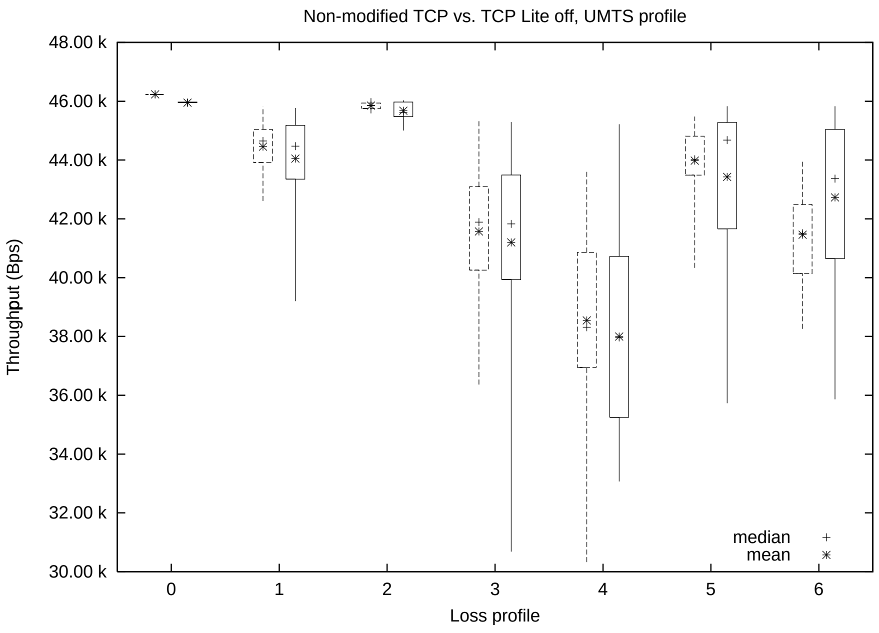

Figure A.2: Throughput comparison for the UMTS profile

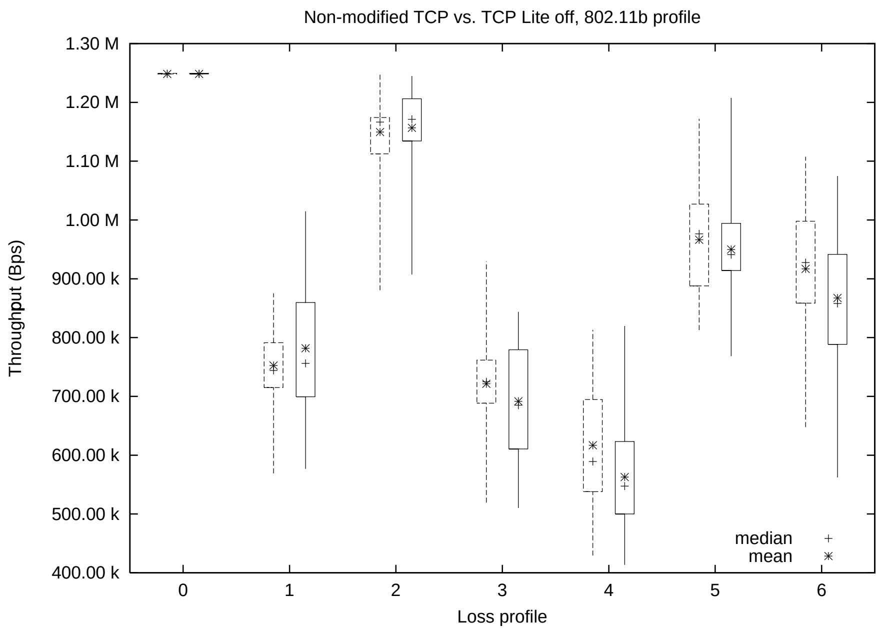

Figure A.3: Throughput comparison for the 802.11b profile

A.2 Two Runs of TCP Lite Disabled

This section presents comparisons of runs with TCP Lite disabled. This is done to show how the randomness incurs small differences in the comparisons. This serves to justify the differences in the previous section.

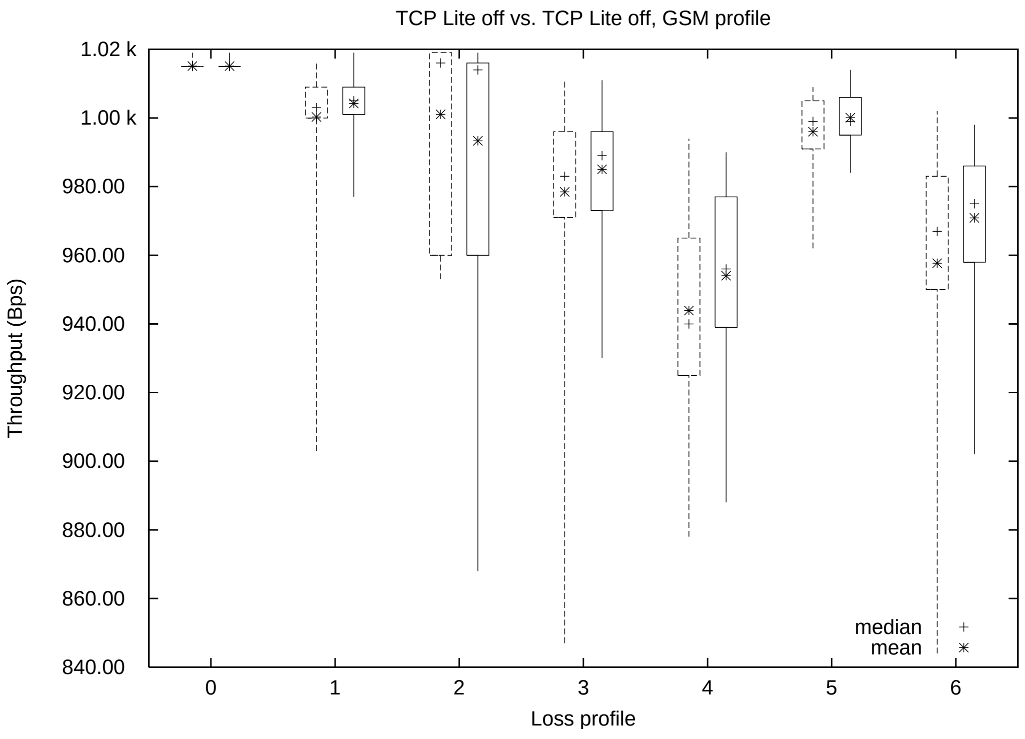

Figure A.4: Throughput comparisons for the GSM profile

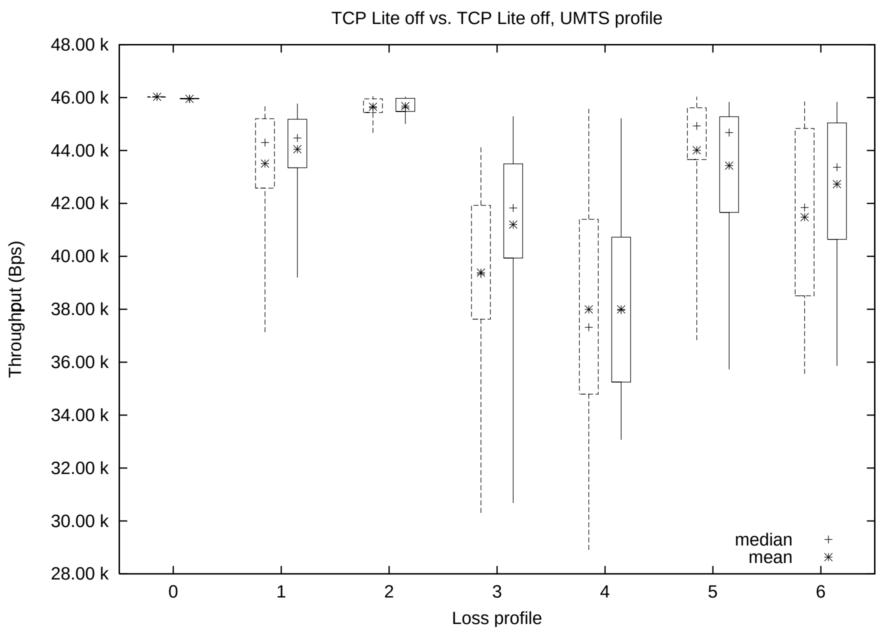

Figure A.5: Throughput comparisons for the UMTS profile

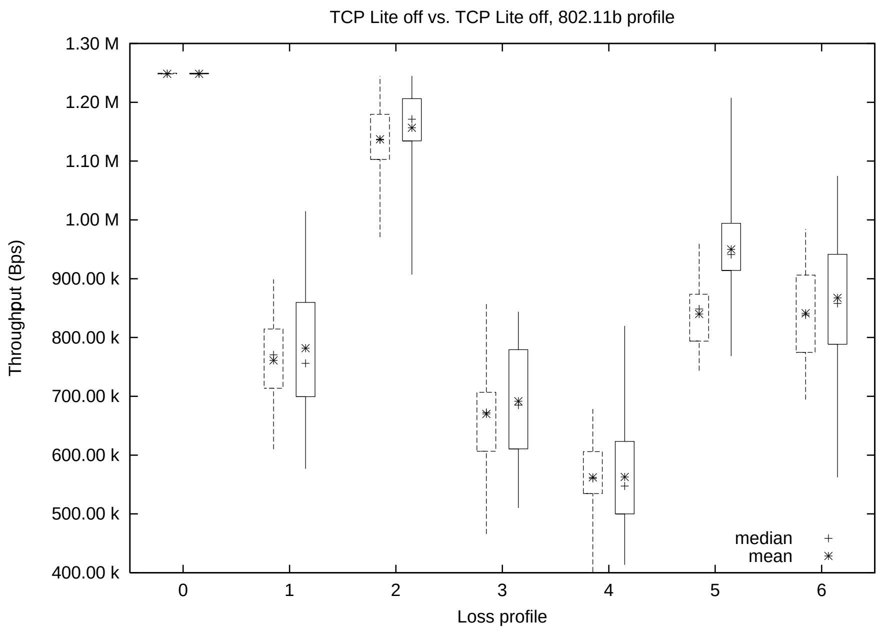

Figure A.6: Throughput comparisons for the 802.11b profile

Appendix B

Experiment Anomaly Discussion

In an earlier experiment (using the same setup) we found an anomaly that could not be reproduced. However it is worth discussing if it should occur again. We saw a major deviation of the minimum value of loss profile 3 (GSM profile), and decided to examine more closely why it had occurred. We found that it was caused by one sample with a very low throughput. If it is removed, the new minimum value would consistent with the other profiles. To determine the cause of the low throughput, we used the graphing functionality of `tcptrace`. Along with the textual information found in Appendix D, graphs of the communication can also be made with tcptrace, for example *time-sequence* graphs which show the relation of time and sequence numbers. Consider Figure B.1, which shows the initial phase of the deviating experiment. After the initial three-way handshake, a packet with data is received. However, no acknowledgement is sent, indicating that the packet was corrupted. This is further evident by the retransmission received seven seconds later. Meanwhile, packets are continued to be received, which are acknowledged with Selective ACKnowlegements[MMFR96], SACK, to indicate that not all packets have been received. Note that `tcptrace` detects a triple dupack with the third SACK acknowledgement. This should normally trigger a *fast retransmit*, but this is not the case. Later, the eleventh packet also becomes corrupted, as seen by the lack of an acknowledgement and by a later retransmission. Eventually, the sender will time out waiting for an acknowledgement for the first packet and retransmit it.

What is believed to happen next is that the sender reduces its congestion window in response to the retransmission, as congestion is believed to have occurred. Since packet eleven still has not been acknowledged, the senders window is full and therefore no more packets can be sent until the window "opens up". This happens when an acknowledgment is received. But the receiver can not send any acknowledgement since it still awaits one packet. After about 32 seconds, the senders retransmission time out triggers a retransmission, packet 14 is acknowledged, and the transmission continues.

The cause of the low throughput is the very long retransmission timeout. Its cause is not fully understood, but some reasons have been thought of:

- The round-trip time estimation have not had time to calibrate since the first data packet was lost.
- The trace is from the receiver. This means that there could be additional retransmissions that have disappeared earlier in the network. Note that errors induced by dummynet are still visible, as evident of the two packets not acknowledged.
- The TCP implementation could contain bugs.

If the reader has any ideas about the cause of this, the author would like to hear them.

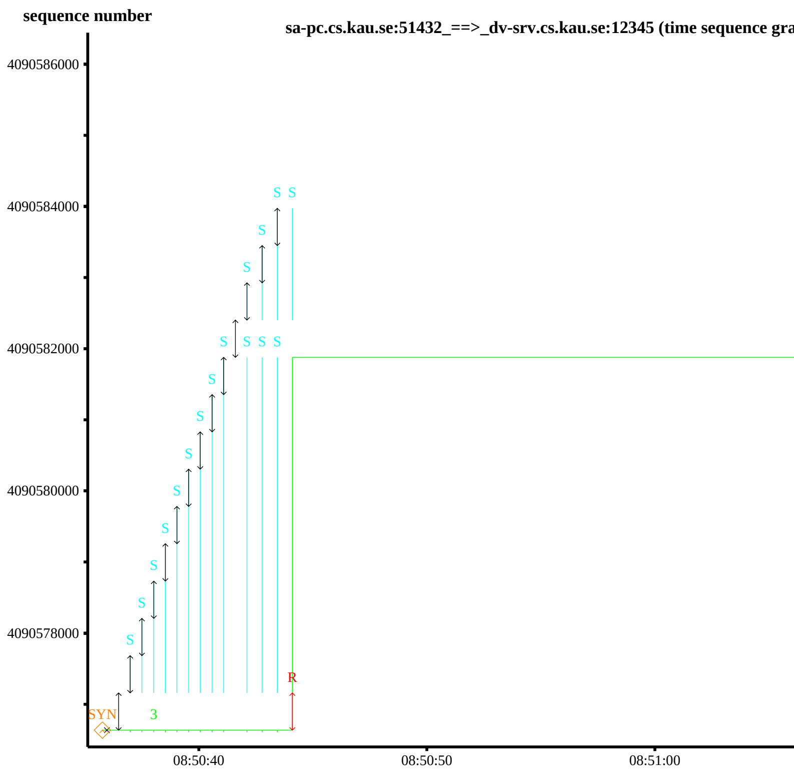

Figure B.1: Retransmission problem

Appendix C

Example Application Program

This is an example of how an application would use TCP Lite. This program was also used in the experiments, on the receiver.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>

#define TCP_LITE 4  /* should be defined in tcp.h */

#define ON 1
#define OFF 0

int create_socket() {

  struct sockaddr_in my_addr;
  int sockfd;

  if((sockfd = socket(AF_INET,  SOCK_STREAM, 0)) < 0) {
    perror("socket");
  }

  my_addr.sin_family = AF_INET;     /* host byte order */
  my_addr.sin_port = htons(12345); /* short, network byte order */
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  bzero(&(my_addr.sin_zero), 8);    /* zero the rest of the struct */

  if (bind(sockfd,
           (struct sockaddr *)&my_addr,
   sizeof(struct sockaddr)) < 0)
    perror("bind");

  if (listen(sockfd, 0) < 0)
    perror("listen");


  return sockfd;
}

void lite(int socket, int val) {
  if (setsockopt(socket, SOL_TCP, TCP_LITE,
                 &val, sizeof(val)) < 0) {
    perror("setsockopt failed");
  }
}

int lite_status(int socket) {
  int val=0, len;
  if (getsockopt(socket, SOL_TCP, TCP_LITE,
                 &val, &len) < 0) {
        perror("getsockopt failed");
  }
  return val;
}

int main(int argc, char **argv) {
  int s, cs, len;
  struct sockaddr addr;

  s = create_socket();
  if (s < 0) {
    perror("socket");
    /* exit(-1); */
  }
  if (argc < 2) {
    printf("Turning lite-mode off.\n");
    lite(s, OFF);
  }
  else {
    printf("Turning lite-mode on.\n");
    lite(s, ON);
  }


  printf("lite-mode is now %d\n", lite_status(s));

  while (cs = accept(s, &addr, &len)) {
        char buf[4096];
        int count;
        printf("Koppel startat.\n");
        while( (count = read(cs, buf, 4096)) > 0);
        close(cs);
  }


return 0;
}
```

Appendix D

Tcptrace Example Output

This shows what information that can be produced with the tcptrace utility.

```
TCP connection info:
30 TCP connections traced:
TCP connection 1:
host a:        sa-pc.cs.kau.se:33440
host b:        dv-srv.cs.kau.se:12345
complete conn: yes
first packet:  Tue Apr 24 23:10:41.854876 2001
last packet:   Tue Apr 24 23:11:32.282353 2001
elapsed time:  0:00:50.427476
total packets: 200
filename:      exp15_dv-srv_100mbit_30_iter/dump-gsm-m0-off.pcap
   a->b:       b->a:
     total packets:           101           total packets:            99
     ack pkts sent:           100           ack pkts sent:            99
     pure acks sent:            2           pure acks sent:           97
     unique bytes sent:     51200           unique bytes sent:         0
     actual data pkts:         98           actual data pkts:          0
     actual data bytes:     51200           actual data bytes:         0
     rexmt data pkts:           0           rexmt data pkts:           0
     rexmt data bytes:          0           rexmt data bytes:          0
     outoforder pkts:           0           outoforder pkts:           0
     pushed data pkts:          7           pushed data pkts:          0
     SYN/FIN pkts sent:       1/1           SYN/FIN pkts sent:       1/1
     req 1323 ws/ts:          Y/Y           req 1323 ws/ts:          Y/Y
     adv wind scale:            0           adv wind scale:            0
     req sack:                  Y           req sack:                  Y
     sacks sent:                0           sacks sent:                0
     mss requested:           536 bytes     mss requested:           536 bytes
     max segm size:           524 bytes     max segm size:             0 bytes
     min segm size:           372 bytes     min segm size:             0 bytes
     avg segm size:           522 bytes     avg segm size:             0 bytes
     max win adv:            2144 bytes     max win adv:           32696 bytes
     min win adv:            2144 bytes     min win adv:           31964 bytes
     zero win adv:              0 times     zero win adv:              0 times
     avg win adv:            2165 bytes     avg win adv:           32681 bytes
     initial window:          524 bytes     initial window:            0 bytes
     initial window:            1 pkts      initial window:            0 pkts
     ttl stream length:     51200 bytes     ttl stream length:         0 bytes
     missed data:               0 bytes     missed data:               0 bytes
     truncated data:        48260 bytes     truncated data:            0 bytes
     truncated packets:        98 pkts      truncated packets:         0 pkts
     data xmit time:       49.527 secs      data xmit time:        0.000 secs
     idletime max:          522.3 ms        idletime max:         1002.9 ms
     throughput:             1015 Bps       throughput:                0 Bps
================================
```