TCP-L: Allowing Bit Errors in Wireless TCP

Stefan Alfredsson and Anna Brunstrom
Department of Computer Science, Karlstad University, S-65188 Karlstad, Sweden
email: {Stefan.Alfredsson,Anna.Brunstrom}@kau.se

***Abstract*— This paper presents a technique to improve the performance of TCP and the utilization of wireless networks. Wireless links exhibit high rates of bit errors, compared to communication over wireline or fiber. Since TCP cannot separate packet losses due to bit errors versus congestion, all losses are treated as signs of congestion and congestion avoidance is initiated. This paper explores the possibility of accepting TCP packets with an erroneous checksum, to improve network performance for those applications that can tolerate bit errors. Since errors may be in the TCP header as well as the payload, the possibility of recovering the header is discussed. An algorithm for this recovery is also presented. Experiments with an implementation have been performed, which show that large improvements in throughput can be achieved, depending on link and error characteristics.**

I. Introduction

The ubiquitous transport protocol on the Internet, TCP, was designed to operate over a large span of heterogenous network technologies. However, wireless links provide new challenges. The bit error rate is much higher over wireless links compared to wireline links. This results in corrupted frames that are discarded at the link layer or at the TCP receiver if the link layer delivers erroneous frames. This poses problems for both TCP and the wireless link. TCP uses lost packets as an indication of congestion, which is not the cause of loss in this case. Thus, TCP initiates congestion control unnecessarily. This causes TCP performance to degrade, and the wireless link to be sub-optimally utilized. Also, the damaged packet is retransmitted over the link, leading to less spectral efficiency. The problems of TCP over wireless networks are noted by [1], [2], [3] and others. The solutions proposed so far to support TCP over wireless can be coarsely classified into three categories. The *split-connection/proxy* approach uses one TCP connection to the base station, and another TCP connection from the base station to the mobile unit. This is for instance used by Indirect-TCP[4]. The *transparent inspection/link-layer* approach is located at the base station and keeps track of the packets flowing by. This is used by Snoop[1] and others to do local retransmission when it is determined that the receiver has not received all packets the base station has forwarded. The third approach, *end-host modification*, requires modifications in one or both of the communicating end-points. This is done for example in FreezeTCP[5] and TCP-HACK[6].

Most of the work to date optimizes the performance of TCP while keeping the paradigm of fully reliable transport. The *Partially Reliable Transport Protocol*[7] (PRTP) instead allows controlled loss in a modified version of TCP. The application can set a reliability level which indicates the amount of data loss the application can accept. Another example of accepting possibly erroneous data is the *UDP-Lite*[8] protocol. In UDP, the checksum covers either the entire packet or nothing at all. With UDP-Lite, the UDP length field semantic is changed to instead mean checksum coverage. This means that the checksum can cover for example UDP/RTP headers, while allowing errors in the RTP payload.

Building on these ideas, this paper presents a technique to accept TCP packets that have been damaged by bit errors. Accepting and acknowledging, instead of discarding, damaged packets means that losses due to bit errors and congestion are separated. Especially multimedia applications may benefit from trading bit errors to get better network performance[1]. For example, some pixels wrong in a picture will be compensated for by the human eye. Likewise for audio, the human ear will compensate for a few wrong sound samples. These applications could then benefit from an increased throughput and reduced jitter by doing a tradeoff for quality.

UDP is often used for transporting multimedia, depending on the needs of real time interactive communication. However, UDP lacks for example the congestion control features of TCP, and may therefore take an unfair amount of the available bandwidth, compared to TCP-friendly streams. This makes TCP an alternative for multimedia applications with soft real time constraints, such as streaming. TCP can be used for multimedia streaming by for example RealAudio[9] and Microsoft Windows Media Player[10]. The case for multimedia streaming with TCP is further argued in [11], [12].

To determine what gains can be made by accepting bit errors, an initial implementation of the idea has been performed in the Linux kernel. This modification is called "TCP-L", as an indication of a more lightweight (fewer retransmissions) version of TCP.

For the modification to be useful, it should be easy to employ. Therefore it only requires modification on the receiving side. Compared to having to modify both the sender and receiver, it is easier for the end-user to deploy a solution which is not dependent upon the service provider to make changes. This approach can then be classified as an end-host modification, relating to the classifications

[1]The underlying assumption is that the application using this modified TCP can handle errors, but this is subject to other research areas such as image coding. Also, the link layer must be able to deliver erroneous data to the transport layer.

discussed earlier. The advantages of this class of modifications are analogous to the "end-to-end" arguments presented in [13]; end-points have the best knowledge of its communication, so that is where logic should be implemented. Applications need to inform TCP-L whether bit errors can be accepted or not, which is another reason for modifying the end-point. At the same time, it could be a disadvantage to have to modify end-user terminals if they are many and require manual intervention. Implementing a modification as an update to the wireless infrastructure could sometimes be easier.

The remainder of this paper is organized as follows. Section II discusses the possibilities of recovering the header, by classifying the header content. The chapter further presents a recovery technique and the possible consequences of making a bad decision. Section III presents experiments done with a simple (no header recovery) implementation of TCP-L, from the setup of the emulation environment to the results. Finally, Section IV presents the conclusions.

II. Header decoding/recovery

The idea to make use of damaged TCP packets is at a first glance easily realized by accepting invalid TCP checksums. However, since the checksum covers both the TCP header and the data payload, it is not obvious if the error/errors is/are in the header or payload. This separation is important because the header contains (among other) information to deliver the data to the correct application, and in the correct order. For the initial experiments described in the next section, the header was kept intact except for the checksum (meaning that no header recovery was necessary). However, if the technique is to be used over a real network, then the problems with potential errors in the header must be taken into account. This section discusses the issues with decoding and recovery of a possibly corrupt TCP header.

A. TCP header overview

The TCP header (Figure 1) contains information to deliver the data to the correct application; when an application wishes to communicate with TCP, it uses the concept of *ports* which multiplex the communication endpoints at a host. The header also contains information (the sequence number) to fit its payload into the data stream delivered to the application, control information to establish/close a connection, and a checksum covering the header and payload.

As earlier header compression research has determined[14], [15], many fields in the header are constant or can be derived from other packets belonging to the same stream. The terminology **nochange**, **inferred**, **delta** and **random** is used in [15]. They refer to fields that are constant throughout the session, can be derived from other information, can be calculated from an earlier value plus a delta value, and fields whose content vary in a random fashion, respectively.

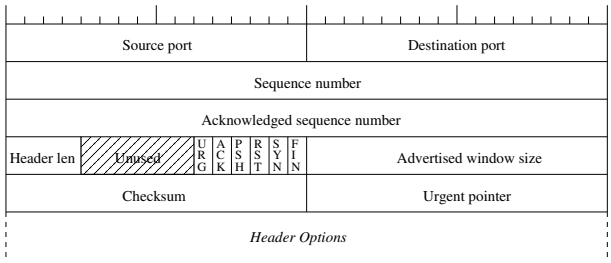

Fig. 1. The TCP header

We reuse these definitions[2] in Table I. It depicts the initial classification of the TCP header fields into the mentioned categories. The assumptions behind this classification are that every packet carries the same amount of data (to have delta-spaced sequence numbers), and the meaning of "random" is more like "unpredictable". Further, to simplify the discussion, it is assumed that TCP options are not used (which gives a constant header length).

nochange	source port, destination port, header length
delta	sequence number, ack number, adv. window
random	checksum, URG, ACK, PSH, RST, SYN, FIN, urg. pointer

TABLE I
CLASSIFICATION OF TCP HEADER FIELDS

There seems to be a lot of randomly varying fields which will cause problems when errors occur. However, by making some assumptions, the table can be extended with an additional category, **dontcare**, and reconstructed as in Table II.

nochange	source port, destination port, header length, URG, ACK, RST, SYN
delta	sequence number
random	
dontcare	ack number, adv. window, checksum, PSH, FIN, urgent pointer

TABLE II
A RE-CLASSIFICATION OF TCP HEADER FIELDS

The reasoning behind the additions to the **nochange** category is the following. The URG flag is used mostly by terminal traffic type applications, i.e. not the multimedia application target. It is also the choice of the application to use this flag, so it can be assumed that it is not used. Only packets with non-empty data payloads are recovered[3]. This means that the SYN flag will always be set to zero, and the ACK flag always set to one[16] since the connection has been established and data is being communicated. When

[2]Except "inferred", because this is e.g. used for frame sizes derived from link layer information.

[3]If a packet has an empty payload, it is of no use to the application. Further, if the checksum is incorrect, it means that all errors are in the header. Therefore it would be better to get a retransmission of this packet than to try to recover it.

the RST flag is set, the packet contains no data. Therefore it is safe to always clear it if there is data in the payload.

The reasoning for the classification of the **dontcare** header fields is the following. The cumulative acknowledgement number in a broken packet can be safely replaced with the most recently correctly received acknowledgement number. Data is assumed to flow mainly in one direction, to the receiver, which acknowledges packets. This causes the advertised window to change minimally, as the sender does not get any data to fill its buffers[4]. The TCP checksum has already failed verification, so it is irrelevant to the decoding. PSH is a recommendation to the receiver to deliver any buffered data to the application, and it is reasonable to believe that an error in this flag would not cause harm. The FIN flag should always be zero until the connection is to be closed down. If it is set to one in a damaged packet, it could either be in error, or it could be a true FIN. In both cases, it is safe to set it to zero, since it consumes one sequence number. A true FIN will then not be acknowledged, causing a retransmission of the FIN which hopefully will be delivered correctly. Since we assumed that no urgent data would be sent, the urgent pointer then becomes irrelevant as well.

The reconstructed classification makes it easier to handle errors in the header, since only the sequence number need to be recovered. As long as the corresponding connection can be found, the other fields can be found (or set to safe values) through other means.

B. Phases of Recovery

This section presents the steps that are taken in order to detect errors, locate connections and salvage errors in the header. A graphical overview is presented in Figure 2, and a discussion follows below.

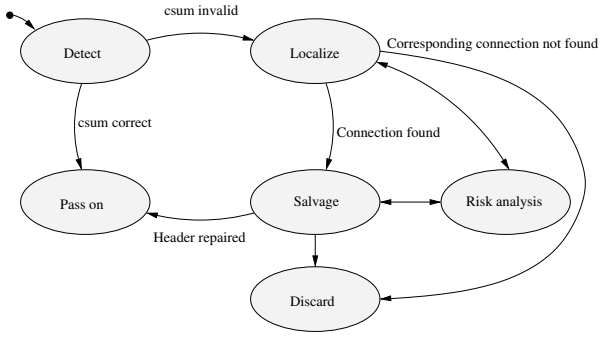

Fig. 2. Recovery procedure overview

a) Detect: The first step is to detect that an error has occurred. This is normally done by verifying the checksum of the segment, but may also be determined from the link layer, depending on the possibilities of information exchange between the layers. If the checksum is correct, the segment is passed on normally. Otherwise, the segment is processed to localize the corresponding TCP connection.

[4]Consideration has not been taken to other causes of advertised window changes, such as Freeze-TCP[5].

b) Localize: When a segment has been determined to contain errors, the corresponding connection for the packet must be found. This is necessary to make use of constant and temporal information for the connection, such as ports and previous sequence numbers. This localization can be carried out with help of the header characteristics discussed in Section II-A. This is done for example by trying to match port and sequence numbers to the active connections in the system. However, too much effort should not be spent trying to find a match. If a lot of the header has been damaged, it is probable that the payload is also severely damaged and not that useful to the application.

c) Risk analysis: If a header field does not correspond to an expected value, there are two options. Either the value is kept, or a presumably better value is chosen. This choosing implies a risk, and the risk analysis must consider the consequences of changing a given header.

The recovery algorithm works by first matching the packet to an existing connection. This is a critical step, because choosing the wrong connection may cause data to be injected into a completely foreign stream. If there are few streams to different hosts and varying port numbers, the chances of mapping a packet to the wrong connection is minimized. However, if there is a set of connections that share one end point and have consecutive local port numbers, the risk of mapping to the wrong connection is increased.

Another critical issue is the sequence number. This determines where in the stream the payload belongs. How critical depends on the applications. Can it tolerate “data reordering”, or “data truncation” for example?

Some of these choices may have to be controlled by the application, because it knows what kind of errors that are acceptable.

d) Salvage: In this step, the packet has been determined to contain errors, and if they are in the header they should be salvaged. The actual algorithm is described in section II-C below. After the header has been salvaged, the segment is passed on as if the checksum was correct.

e) Discard: In the salvation phase, chances are that the header could not be satisfactorily salvaged. For example, when doing the risk analysis a change may be deemed too risky to perform, but also too risky to not perform. One solution that can always be used in these cases is to simply discard the packet. This is what happens normally when the checksum is invalid, and causes a retransmission of the packet.

C. Recovery Algorithm

Given the assumptions outlined earlier with regard to the header field properties, a possible way to implement the recovery phases is outlined below:

1) A faulty packet (tcp segment) is detected by an invalid checksum.
2) The packet is mapped to an existing connection, identified by *{src addr, dest addr, src port, dest port}*.

If no exact match is found, an approximate matching is tried. That is, removing *src port* from the matching or doing bit-level matching. If several connections are found to be possible candidates, sequence and acknowledgement numbers are used to choose the most probable connection.
If no connection is found to be a close enough match, the packet is discarded.

3) If the constant header fields (Table II) are incorrect, they are changed to match those of the found connection. The "dontcare" fields are set to safe defaults.
4) The sequence number is checked to see if it is the expected one, in relation to earlier received packets. If it is not expected, there may be two causes.
   a) The sequence number may be corrupt
   b) The packet may have arrived out of sequence

   These causes could also be combined, i.e. a packet out of order has a corrupt sequence number.
   If the sequence number equals the last sent acknowledgement, we assume it is correct. From the assumption that the sequence number is delta spaced, multiples of packet sizes can be added to the last sent acknowledgement to see if it matches. Then, the packet may be assumed to be delivered out of order. However, if the sequence number does not meet the criteria above, the packet should be discarded. This is to minimize the risk of placing data at the wrong position in the data stream, which may be harder to handle for the application compared to bit errors.

III. Experiments

Experiments aimed to compare the performance of regular TCP to TCP-L, from a protocol point of view, have been performed. The performance was believed beforehand to be improved, but the degree of improvement would be shown through experiments. It was also the intention to get an indication if performance improvement differed between different types of networks. Therefore, a number of *link profiles* with different bandwidth and delay characteristics were used. To see the impact of different error characteristics, a number of *loss profiles* were defined. This implementation of TCP-L did not include the header recovery algorithm mentioned above.

A. Experiment Setup

The physical experiment setup consisted of three networked computers (Figure 3). One sender, one gateway which emulated the wireless link models and created errors in passing packets, and one receiver containing the TCP-L modification.

The wireless link emulator was set to emulate three different link profiles. To get a more concrete vision of the links, they may be thought of as a GSM telephone link, a third generation telephone system link and finally a WiFi 802.11b LAN network.

- Low bandwidth and high delay; 9.6kbit/s, 150ms delay ("GSM")

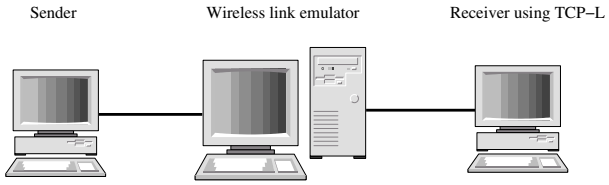

Fig. 3. Experiment setup

- Medium bandwidth and medium delay; 384kbit/s, 70ms delay ("UMTS")
- High bandwidth and low delay; 11Mbit/s, 10ms delay ("802.11b")

These link profiles were used together with a range of loss profiles (Table III). The loss profiles used a two-state markov model to simulate error bursts, with different timing values for a "good" state (no errors occurring) and a "bad" state where errors hit every packet. This is a standard model used by [8], [17], [18], and others to simulate errors on wireless links. The resulting packet error rates ranged from 0 to about 5 percent. Note that in these experiments the position of the corruption is not taken into account. Except for the checksum, the header was always correct.

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

TABLE III
LOSS PROFILES USED IN THE EXPERIMENTS

The experiment consisted of sending bulk data from the sender to the receiver, while capturing the network traffic with the tcpdump network packet capturer. This data was then analyzed with the tcptrace[19] tool. The results of this analysis is summarized in the next section. Further analysis is available in [20].

B. Experiment Results

Figures 4-6 show the results in a graphical form, and they are constructed as follows. On the y-axis, the throughput is shown. The x-axis is marked 0 to 6, corresponding to the loss profiles in Table III. Two values are shown within each loss profile, the leftmost representing the results from the experiments with TCP-L enabled and the rightmost with TCP-L disabled (i.e. an unmodified TCP). Each result is then presented as a box plot, to also show the extreme values and quartiles of measurements instead of only the mean and median value.

f) GSM Profile: The throughput graph of the GSM profile experiments can be found in Figure 4. We see that for loss profile 0, the results are the same whether TCP-L is enabled or not. This is the expected result, as this profile should generate zero errors. This shows that

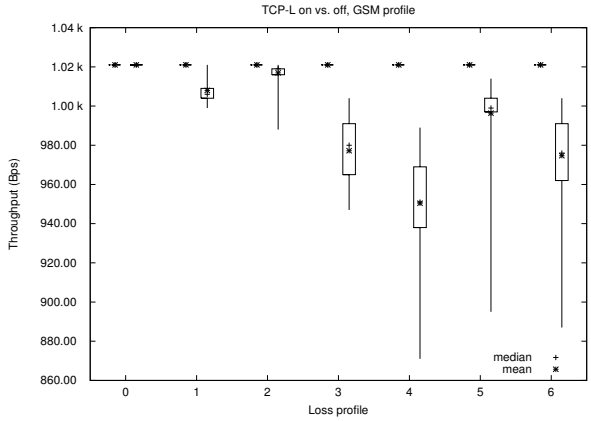

Fig. 4. GSM profile results

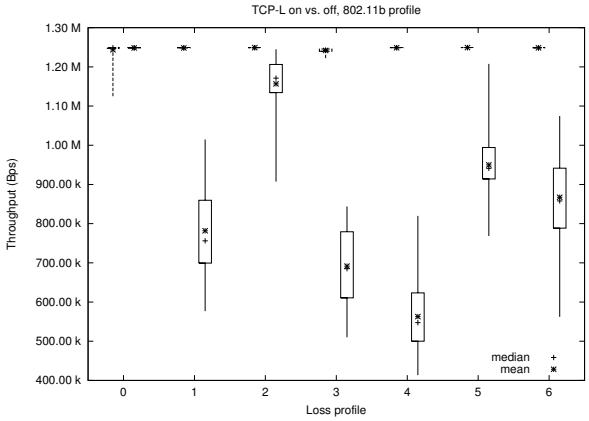

Fig. 6. 802.11b profile results

TCP-L behaves like unmodified TCP when there are no errors. When errors are introduced, TCP-L continues to maintain the same throughput, whereas regular TCP starts to degrade.

We note that in the experiments where TCP experiences packet loss, the mean/median degradations vary from about 1 to 6 percent. The performance degradations can be seen proportional to the "bad packet rate" presented in Table III. A final observation of the results within each loss profile reveals that TCP-L enabled shows no variation, while the variation in throughput degradation with original TCP is large. For example, in loss profile 4 it varies from 3 to 15 percent.

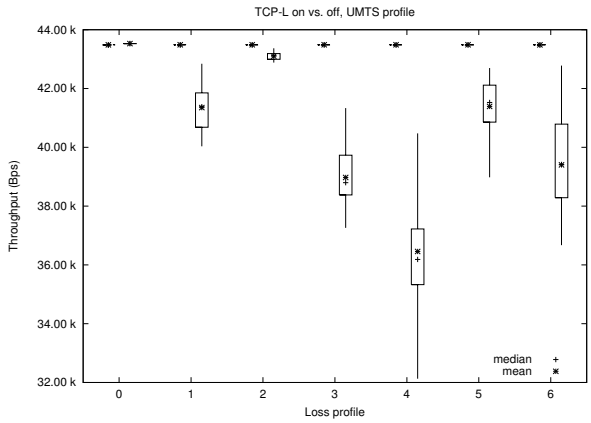

Fig. 5. UMTS profile results

g) UMTS Profile: The throughput graph for the UMTS profile is displayed in Figure 5. As for the GSM profile, enabled TCP-L delivers constant throughput and no variation. When regular TCP experiences packet loss, the degradations of mean/median throughputs varies from 3 to 17 percent. Also, we note that the throughput variations within the loss profiles are large. For example, a throughput degradation between 6 to 26 percent is seen in loss profile 4.

h) 802.11b Profile: Figure 6 shows the results for the 802.11b profile. As before, loss profile 0 is the same for TCP-L enabled and disabled, as it should. Using TCP-L gives a constant throughput with almost no variation. When TCP-L is not used, we see a degradation in average throughput in the range of 20 to 56 percent, for loss profiles 1 to 6. This means that in some cases, enabling TCP-L yields a 100% throughput increase.

However, it should be noted that these experiments were done with the assumption that all packets could be used by TCP-L. It is likely that an error burst destroys the sensitive header information, rendering the packet useless. It must then be discarded, and a retransmission would occur. This retransmission would then cause a performance degradation for TCP-L as well. The experiments do, however, indicate the maximum gains that can be made, if all headers can be recovered.

IV. Conclusions

This paper explores the idea of allowing bit errors to improve the performance of TCP in a wireless environment. From the presented experiments we conclude that large improvements in network performance may be gained, depending on link and error characteristics. For this to be possible, some conditions must be met. The link layer should deliver erroneous data to the transport layer. The application should also be able to handle errors, and to communicate to TCP-L when errors are acceptable or not. We are currently extending our TCP-L implementation to handle header recovery, using the presented recovery algorithm. Future work involves improving the implementation by utilizing "soft information" from the link layer frame decoding process to correct as many errors as possible.

Acknowledgements

This work has been sponsored in part by Vinnova (the Swedish Agency for Innovation Systems) and the Swedish graduate school PCC++.

REFERENCES

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

[2] S. Dawkins, G. Montenegro, M. Kojo, V. Magret, and N. Vaidya. RFC 3155: End-to-end performance implications of links with errors. August 2001.

[3] H. Elaarag. Improving TCP performance over mobile networks. *ACM Computing Surveys*, 34(3):357–374, August 2002.

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

[5] T. Goff, J. Moronski, D. Phatak, and V. Vipul Gupta. Freeze-TCP: A true end-to-end enhancement mechanism for mobile environments. *Proceedings of IEEE INFOCOM*, March 2000.

[6] R. K. Balan, B. P. Lee, K. R. R. Kumar, L. Jacob, W. K. G. Seah, and A. L. Ananda. TCP HACK: TCP header checksum option to improve performance over lossy links. *Proceedings of 20th IEEE Conference on Computer Communications (INFOCOM)*, April 2001.

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

[8] L-Å. Larzon, M. Degermark, and S. Pink. UDP Lite for real-time multimedia applications. *Proceedings of the IEEE International Conference of Communications (ICC)*, June 1999.

[9] RealNetworks Inc. Real player. *http://www.real.com*, November 1998.

[10] Microsoft. Microsoft windows media player. *http://www.microsoft.com/windows/mediaplayer*.

[11] C. Krasic, K. Li, and J. Walpole. The case for streaming multimedia with TCP. *8th International Workshop on Interactive Distributed Multimedia Systems (IDMS2001)*, September 2001.

[12] A. Goel, C. Krasic, K. Li, and J. Walpole. Supporting low latency TCP-based media streams. *Proceedings of IWQoS*, May 2002.

[13] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[14] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links. February 1990.

[15] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. *Wireless Networks*, 3(5):375–387, October 1997.

[16] J. Postel. RFC 793: Transmission control protocol, September 1981.

[17] 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.

[18] C. Parsa and J. Garcia-Luna-Aceves. Improving TCP performance over wireless networks at the link layer. *ACM Mobile Networks and Applications Journal, vol 5, issue 1.*, 2000.

[19] S. Ostermann. Tcptrace. *http://www.tcptrace.org/*.

[20] S. Alfredsson. TCP Lite - a bit error transparent modification of TCP. *Master's Thesis 2001:06, Karlstad University*, June 2001.