Tel Aviv University

The Raymond and Beverly Sackler

Faculty of Exact Science

Improvement of TCP connection throughput over noisy environment

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in Tel-Aviv University, School of Computer Science

by

Alexander Cheskis

Prepared under the supervision of Prof. Yishay Mansour

September, 2006

Abstract

An algorithm for the improvement of the speedup of TCP connections in modern networks with multiple packet drops is proposed, studied, and analyzed. The modification of the TCP protocol, is aimed to improve the congestion control between TCP client and server. While numerous TCP improvements algorithms and TCP modifications require to change both TCP client and TCP server, our TCP Taba modification changes only the TCP client, which gives the possibility to implement it on the new-comming wireless networks, which support smart-phones, PDA devices and laptop computers, without need of altering existing TCP servers.

The implications of our modification in the congestion control algorithm is studied. We show that in order to best implement our modification, the TCP client needs to learn the status of the TCP server, and for this purpose a "Slow-start server detector" is introduced.

We have implemented TCP Taba client on a Linux OS, and used it in the evaluation of the TCP Taba performance. To facilitate the testing we used a self-made Quality of Server (QoS) router, which allows us to simulate traffic with different delay and loss rate. The problems of coexistence of our TCP Taba client with different Operation Systems is addressed, and our tests show that the vast majority of Operation Systems work well with the TCP Taba client.

Keywords: Computer network architecture, TCP protocol, Tahoe, Reno, Vegas, ACK division, Daytona

Acknowledgments

I would like to thank my advisor, Prof. Yishay Mansour. His advise, patience and support made this work possible. I also would like to thank my wife, Inna. Her support is unfailing and her patience and understanding during many long nights working was unselfish and appreciated. Finally, I would like to thank my parents for their support over the years.

Contents

List of Figures

List of Tables

Chapter 1

Background

The Internet has changed dramatically during the last decade. Today powerful mainframes and workstations can communicate with wireless smartphones, PDA devices and laptop computers not only via existing wired networks, but also via various wireless, radio or infrared networks. In spite of the fact that underling physical protocols can be different (for example Bluetooth, Wi-Fi, ZigBee or GPRS) the network layer for all of them uses the same IP protocol.

TCP is the dominant protocol, which form the basis for the reliable communication to the web browsing, and for the file and email transfer. According to several researches [1] it takes about 90% of all Internet traffic. However, TCP protocol was designed [2] and later was modified [3] assuming wired connection in the network. Thus, the network congestion and server overload was considered as the main reason for the packet losses. This assumption is not true for the wireless networks, where due to the fading channels and user mobility, transmission losses are much more frequent. The current TCP implementations do not perform well in such environments since these assumptions do not hold for such transmission-heterogeneous media. One of the first researches analyzing these problems were made as early as in 1995 [4]. At present this problem started to be of a great importance, since the cost of the packet drops is much higher in the current high-speed wireless networks. Numerous works [5] were done to make new client-server extensions of TCP to answer this challenge.

The goal of this introduction is not to give a complete overview of the fascinating area of computer communication; more detailed descriptions of network architectures and protocols can be found in various textbooks such

as: Stallings [6] (hardware signaling and networking concepts), Comer [7] (general networking concepts), Stevens [8] (detailed description of the Internet protocols). In this introduction we will focus on the background required to introduce our ideas in TCP Taba.

1.1 TCP

The Transport Control Protocol, TCP, is a protocol that runs on the top of the the IP network layer. In TCP, the receiver acknowledges all received data with an ACK message, and the sender buffers the sent data until an ACK packet is received. If no ACK packet is received, the sender retransmits the data.

The TCP sender uses two parameters, also referred as "windows", to control the send rate. Both parameters limit the number of the packets that may be sent by the sender without receiving the acknowledgment. The sender uses the minimal value which provides by these two parameters. One parameter is the receiver-controlled *offered window* (also called advertised window), written in the ACK packet, that tells the sender how many bytes can be sent without overflowing the receivers buffer. This parameter inhibits the sender from flooding the received buffer of a slow receiver. (See Fig. 1.1).

The other parameter is the sender-controlled *congestion window,* which limits the number of packets that may be sent by the sender without receiving the acknowledgment. This parameter prevents network congestion.

The packet losses are detected in TCP by using a timer that triggers after the time which is twice that network round-trip time. TCP protocol assumes that losses are mainly due to congestion rather than to transmission errors [9]. Therefore after detection of the packet losses, the sender voluntarily reduces the congestion window, which in its turn decreases the transmission rate avoiding in such a way a packet congestion. The senders decrease their send rate in two ways. First, when a packet loss occurs, the congestion window size is reduced to some threshold, and afterwards it increases slowly. Second, for every consecutive loss, the retransmission interval is doubled, in order to prevent the network from being flooded by retransmissions.

Different actual TCP implementations have various ways to increase the window size again. The implementations are often named after their corresponding BSD Unix releases, with names like Tahoe, Reno, or Vegas. The different versions work together, since the protocol does not require a partic-

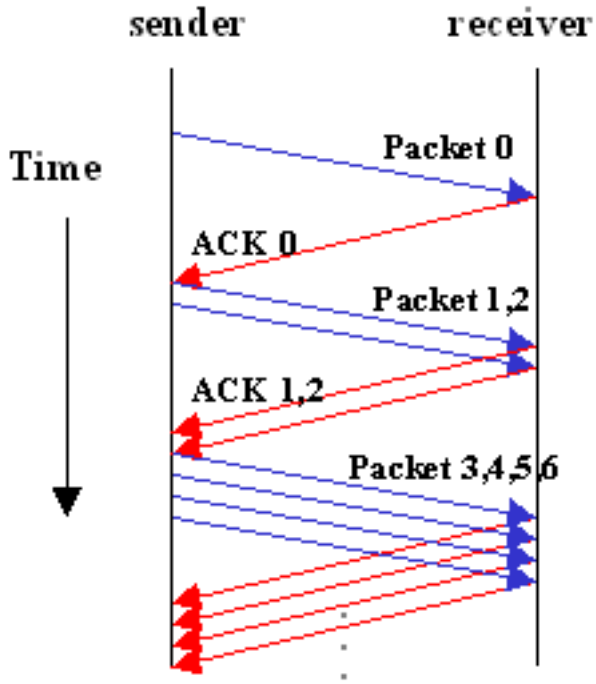


Figure 1.1: Slow start algorithm in TCP protocol. Above the sender increase the window, every time ACK is arrived.

ular sender behavior.

1.2 TCP congestion control

Let us define the following TCP parameters in more details below, in order to use them later in this section:

- a sender maximum segment size (*smss*) represents the maximum amount of the data that can be sent in a single TCP segment, without including the header.
- a sender window (*swnd*) represents the maximum number of bytes that the can be sent. Its value is the lowest between receiver's window and congestion window.
- a receiver's window (*rwnd*) is the latest window advertised by the receiver.
- a congestion window (*cwnd*) is a TCP state variable, limiting the amount of the data that can be sent.
- a loss window (*lw*) is the value of the congestion window when a packet loss has been detected.
- a slow-start threshold (*ssthresh*) is another TCP state variable that determines which of the congestion control algorithms to be employed: either the slow-start algorithm (if $cwnd \leq ssthresh$) , or the congestion avoidance algorithm(if $cwnd \geq ssthresh$). The basic TCP congestion control which uses the Slow-Start and Congestion Avoidance algorithms, is based on the work initiated by Van Jacobson [9].

1.2.1 The TCP Slow-Start and Congestion Avoidance Algorithms

The changes of congestion window in the TCP protocol is realized by use of two algorithms, which control the amount of data sent over the network.

The word "slow" in the name of Slow-Start algorithm has a historical base, and the algorithm is sufficiently fast; for example to reach 1Mbps (100 kBytes per second) it needs only 8 RTT packets (with a minimal default *smss*

is equals to 512 bytes). The purpose of the Slow-Start algorithm is to fill as soon as possible the transmission channel up to the available maximum, with fairness between the competing TCP streams. In the beginning of transmission or after detection of the multiple packet loss, the TCP entities use so called "Slow start algorithm", which actually increases exponentially fast the size of the congestion window (See Fig. 1.1). The algorithm begins in the exponential growth phase initially with 1 or 2 *smss* (depending on the TCP implementation), and increases the congestion window size exponentially every round trip time (*cwnd = cwnd * 2*) until a predefined *ssthresh* is reached and $cwnd \leq ssthresh$.

Once the *ssthresh* is reached, and $cwnd \geq ssthresh$, the Congestion Avoidance Algorithm is started and the congestion window size is increased linearly (*cwnd = cwnd + 1*) after every RTT. When segment acknowledgements are not received, the *ssthresh* is set to half of the current congestion window size (*cwnd*), and the algorithm restarts. The Congestion Avoidance Algorithm is used to control the congestion window, from the time when the available maximum rate was reached.

1.2.2 The Fast Retransmission and The Fast Recovery Algorithms

Implementing the TCP Slow-Start and Congestion Avoidance algorithms, which were described in Section 1.2.1 creates new problems. The first one is related to the packet loss detection. Normally, a packet loss is recognized when the timeout of the retransmission timer is detected. This, however, may lead to significant delays in the data transmissions, so another way to determine packet loss has been added to TCP. Under normal circumstances the TCP entity must send an acknowledgment (ACK) for every received data packet. It should also send ACK for every packet that arrives out of sequence. But in this case the ACK contains the reference to the data, which was received before the packet loss. A packet may be received out of sequence due to packet duplication by the network, packet delays or packet loss.

The Fast Retransmission algorithm assumes that a packet has been lost when it receives 3 duplicate ACKs (namely 4 ACK packets which contains same reference to the same data), before the timeout of the retransmission timer. In this way valuable time is saved.

The second problem is related to a drastic decrease of the congestion

window (*cwnd*) after the packet loss detection. In earlier versions of TCP, if a packet is lost the value of *cwnd* is set to the value of the minimal *cwnd* window (equals to 1 *smss*).

The Fast Recovery algorithm tackles this problem. If the packet loss is detected according to the Fast Retransmission protocol, the new value of the congestion window is set to *ssthresh + 3*smss*. This is called as *artificial inflation* of the congestion window. For every duplicate ACK the congestion window (*cwnd*) is incremented by 1, when a new ACK is received the Fast Recovery algorithm terminates, and the congestion avoidance algorithm is continued.

1.3 TCP implementations

According to [10] the most popular TCP implementations today are the following:

1. TCP Tahoe: includes Slow-Start, Congestion Avoidance and Fast Retransmit. The Tahoe implementation of TCP (1988) introduced significant improvements for working over a shared network. An algorithm Slow Start (for congestion control) and multiplicative decrease (for congestion avoidance) was introduce to control transmission following any detected congestion.

2. TCP Reno: adds Fast Recovery to TCP Tahoe.

   (a) The new algorithm (1990) prevents the communication path from going empty after Fast Retransmit, thereby avoiding the need to Slow-Start to re-fill it after a single packet loss. Fast Recovery is entered by a TCP client after receiving an initial threshold of duplicated ACKs. This threshold, is generally set to three. Once the threshold of duplicated ACKs is received, the server retransmits one packet and reduces its congestion window by one half.

   (b) Instead of slow-starting after detection a packet loss, as is performed by a TCP Tahoe server, the TCP Reno server uses additional incoming duplicated ACKs to clock subsequent outgoing packets. TCP Reno's Fast Recovery algorithm is optimized for the case when a single packet is dropped during a window of data.

The TCP Reno server retransmits at most one dropped packet per round-trip time. TCP Reno significantly improves upon the behavior of TCP Tahoe when a single packet is dropped from a window of data, but can suffer from performance problems when multiple packets are dropped during a window of data.

3. TCP New-Reno[11]: enhanced TCP Reno using a modified version of Fast Recovery. Janey Hoe proposed a modification to TCP Reno usually called New-Reno, which addressed two problems in TCP Reno, these ideas are gradually finding acceptance within the IETF. First Hoe noted that a smaller value for *ssthresh* causes premature termination of the slow start phase and subsequent slow increase of the *cwnd* (i.e. the linear increase phase). A larger value causes the sender to over-feed packets to the network (i.e. , transmit too long a burst of data packets) causing congestion. The second problem occures since most TCP sessions last only for a short period of time, the initial slow start period is significant for the over all performance. Hoe proposed a method to estimate an optimum *ssthresh* value by calculating the byte equivalent of bandwidth delay product of the network, when a new connection is made. The bandwidth is calculated using the Packet-Pair algorithm (measuring the arrival time of closely spaced ACKs at the sender).

4. TCP Vegas: uses a smart "Additive Increase Multiplicative Decrease" (AIMD) technique, was proposed by Lawrence Brakmo. This primarily add rate control to avoid congestion (rather than react after detection of congestion). Vegas has not been widely implemented and is not universally accepted by the Internet community and is still a subject of much controversy.

5. SACK (Selective acknowledgment): The probability of multiple packets loss in a window is much greater for a fast network, where many more packets are in transmit. Although TCP is able to recovering multiple packet losses without waiting for expiry of the retransmission timer, frequent packet loss may still not be efficiently recovered. The SACK extension (1996-98) improves TCP performance over such a network, and has been included in some recent TCP implementations. The SACK option is triggered when the receiver buffer holds in-sequence data segments following a packet loss. The receiver then

sends duplicate ACKs bearing the SACK option to inform the transmitter which segments have been correctly received. When the third duplicate ACK is received, a SACK TCP transmitter retransmits only the missing packets starting with the sequence number acknowledged by the duplicate ACKs, and followed by any subsequent unacknowledged segments. The Fast Retransmission and Recovery algorithms are also modified to avoid retransmitting already SACKed segments. The explicit information carried by SACKs enables the transmitter to also accurately estimate the number of transmitted data packets that have left the network (this procedure is known as Forward Acknowledgment (FACK)), allowing transmission of new data to continue during retransmission. The SACK option is able to sustain high throughput over a network subject to high packet loss and is therefore desirable for bulk transfers over a network. Optimization of the algorithms which govern use of the SACK information are still the subject of research, however the basic algorithms are now widely implemented.

Other current TCP optional implementations are Peach, ATCP etc. As we can see, the differences between versions are related to the congestion control algorithms involved. We can exploit this observation in order to determine the TCP implementation on a certain machine. More information on the differences between the versions is found in[12]. Table 1.1, describes TCP versions used by different Operation Systems:

We tested several operating systems in order to determine the TCP implementation. Some old editions of tested systems used Reno (FreeBSD 3.5.1 and 4.2), while the latest versions evolved toward New Reno (FreeBSD 4.3, 4.4, 4.5, RedHat 7.2). Windows 95/NT Professional are currently using Tahoe NoFR (Tahoe without Fast Retransmit).

1.4 Our Contribution

As it was said before one of the current challenges is to change TCP protocol for wireless heterogeneous media. We propose a method and algorithms which allows to improve dramatically the throughput over wireless media, with only changes of the client side of the TCP protocol.

Our approach is different from other attempts, mostly theoretical, which try to change both server and client TCP stacks [13], or server TCP side

OS TCP	Implementation
FreeBSD 3.5.1	TCP Reno
FreeBSD 4.2	TCP Reno
FreeBSD4.3	TCP New Reno
FreeBSD 4.4	TCP New Reno
FreeBSD 4.5	TCP New Reno
Linux 2.4.9 (RedHat 7.2, 9.0)	TCP New Reno
MS Windows 95	TCP Tahoe NoFR
MS Windows NT	TCP Tahoe NOFR
MS Windows 98	TCP New Reno
MS Windows ME	TCP New Reno
MS Windows 2000	TCP New Reno
MS Windows XP	TCP New Reno

Table 1.1: TCP versions, on different Operation Systems
NoFR - No Fast Recovery

only [14]. It is obvious that big companies, like eBAY or CNN use robust and proved TCP stacks, which serves millions of "wired" connected users, and will not change their servers because of the small amount wireless connected clients. Thus there is only tiny probability that TCP server's software will be changed in the near future especially to experimental and not-robust software. On the other hand, TCP client's software on the new created devices changed quicker. Almost every new device has its own modification of the TCP stack. Therefore, TCP stack on new-designed wireless devices such as smart-phone or PDA will be changed rather easy.

In the present work the implication of this modification in the congestion control algorithm is studied. The status of the TCP server state machine should be known to best implement our modification only on TCP client. Since there is no explicit notification from the network about TCP server state machine, implicitly detection of packets loss as congestion events was introduced. We called this algorithm "Slow-start server detector" (See Section: 3.1.1)

The majority of TCP modification algorithms were studied previously only in simulation. For our tests we implemented TCP Taba client on Linux OS. To fusilitate the testing we use self-made QoS router (See Section: A.3), which allow us to simulate traffic with different delay and loss rates. We

point out the problems of coexistence of our TCP Taba client with different Operation Systems and show that the vast majority of them work well with TCP Taba.

1.5 Related work

As it was said in Section 1.4 we are changing TCP client stack in order to boost throughput over non-reliable physical media. Thus we can divide the related work, into two parts: the first part of them, is general TCP optimization research of the traffic over non-reliable media, and the second part is the research on algorithms for throughput optimization, by TCP client.

TCP improvements algorithms for non-reliable media. There is a lot of research in the recent years on this area. Some of them [15] are focused on *additive increase, and multiplicative decrease* (AIMD) algorithms and relate to TCP's congestion control mechanism. They suggest to change congestion algorithm to return to a fair level quicker. Other research [16] deal with changing of the Non-reliable Physical Layer in order to improve its behavior by entering "TCP-look like" protocols.

TCP congestion control algorithms for speed boosting on TCP Client Surprisingly the one of the interesting methods, which we used for our optimization, was "discovered" by hackers, and analyzed in the following research work [17]. The authors studied the consequences of some possible hacker attacks:

- ACK division - multiplexing of the ACK acknowledges to destroy fairness between TCP connections.
- DupACK spoofing - algorithm artificially inflate additional (more than 3) ACK acknowledges, to reflect the additional segment to be sent.
- Optimistic ACKing - algorithm sends successful acknowledge to "expecting" segment, which is still not delivered.

They show that all attacks are possible on most TCP stacks, and suggest to change future TCP stacks to minimize trust they place in the other parties. Needless to say that our changes in TCP stack are build to preserve fairness.

Chapter 2

TCP Taba[1]

2.1 Motivation

The mechanism of the TCP flow control, which based on the TCP Slow Start and Congestion – Avoidance algorithms (described in Chapter 1.2.1), was introduced to optimize a traffic load in the wired Internet. In conditions of the wired network, the main reason of the packet loss, was related to overload of the intermediate servers and routers in the network. This mechanism allows to avoid server congestion, and additionally it shares fairly the available bandwidth between flows. The mechanism also provide an efficient utilization of the available capacity under a wide range of the dynamic traffic loads. The TCP flow control mechanism uses a window and end-to-end acknowledgment scheme to provide reliable data transfer across a network.

The sending host (server) maintains a congestion window, *cwnd*, which places an upper limit on the number of segments that may be sent into the network awaiting acknowledgment by the receiver. Upon receiving a data packet the receiver (client) sends a cumulative acknowledgment, that covers all continuously received packets.

The TCP flow control needs a way to detect congestion. Since there is no explicit notification from the network it uses implicitily detection of packets loss as congestion events. The packet loss is detected either through a timeout of an unacknowledged packet, or the receipt of several duplicate acknowledgments. The loss is assumed to be caused by a buffer overflow at

[1]TCP versions (TCP Tahoe, Reno, Vegas) have names, according to the famous casino places. We call our flavor as TCP Taba

the client or at some intermediate router due to offered traffic exceeding the available capacity on the end-to-end path of the connection.

Increasing and decreasing *cwnd* allows TCP better utilization of the available bandwidth on a given end-to-end path. The *cwnd* is increased by 1 *smss* (normally 512 bytes) for each acknowledgment during the TCP Slow start phase (see Figure: 1.1), and by a fraction $1/cwnd$ of a *smss* during the TCP congestion avoidance phase; The *cwnd* is decreased to the *ssthresh smss* after the detection of the packet loss.

While this mechanism works fairly well, especially in wired networks, there is a problem in the case of the multiple packet drops (due, for exapmle, to RF or wireless networks interference). When such drop happens, the server initiates a Slow-Start phase, since obviously the server is unable to recognize the reason for the drops. From the obsolete "wired" point of view, the server believes that drop packets are due to the network congestion. In the "wired" world the chance for such drop event was so low, that the protocol designers didn't care about the consequences of such drops. In the "wireless" world, there is a high probability of the multiple packet drops, especially when wireless coverage is insufficient. These drops obviously do not require a decrease in the throughput (in other words, the congestion window should not be changed). Nevertheless the actual effect for the wireless packet drops, will cause the TCP to return to the Slow Start phase. This return will result in a significant decrease of overall network throughput.

Lets consider for example what happens after such a multiple packet drop in a standard wireless networks. Say that we have a connection with a round trip time (RTT) of a 200ms (say, from US to Europe) and a standard packet size of 1500 bytes. For wireless bandwidth of 20Mbps the ideal congestion window is of about 350 packets. Immediately after the detection of the multiple packet drop, *cwnd* will be set to 1 *smss*, which is equivalent to sending at 20 kbps. To reach the sending rate of 20Mbps again will take 8 round trip times, or about 1.6 seconds.

All the TCP algorithms (Tahoe, Reno, New Reno, or Vegas) are negatively influenced by such a *cwnd* drop. However in the case of obsolete Tahoe protocols, such a behavior will lead to espessialy low throughput utilization. As one can see from the Table 1.1, there is still a significant fraction of OS that use it.

The first solution for multiple packet drop behavior is to classify the packet drop reasons into two categories:

1. buffer overflow on the client side or on the route, or
2. a noisy environment, or non-reliable media (such as wireless, RF, Infrared, and so on)

To distinguish between the two categories requires changes in both server and client network stacks; one of the solutions of such protocol changes is ECN (Explicit Congestion Notification). A second solution, is to change only the server's part of the TCP stack[14], and to return to the previous *cwnd* quicker. The main drawbacks of these schemes is the fact that they do not work with existing the TCP servers.

We propose a third solution: "TCP Taba algorithm", which requires a change only in the TCP client stack, and works with the majority of the available TCP servers. Following is the description of our algorithm.

2.2 TCP Taba Algorithm

TCP Taba changes the state machine of the client TCP implementation. (The complete implementation details are given in Section A.1). The behavior of our client is different and can be described by the following state machine:

1: Upon receiving message *M(s)*
2: **if** *SlowStartDetected* **then**
3: *START TCP Taba*
4: **else**
5: *RETURN to normal TCP*(*CongestionAvoidance*)
6: **end if**

First we operate as usual, based on the current client TCP implementation. The changes starts, when we detect the Slow-Start mode on the server's side. We know that the reason of the Slow-Start are multiple packet drops. We also know that the server will increment its congestion window size, every time when we send it an ACK. In order to return its window to the previous value as soon as possible we send multiple ACKs to the server, based on the single data packet we received. For example if we get 512 bytes of the data, we can acknowledge them with two subsequent ACKs: the first on receiving 256 bytes, and the second on receiving the last 256 bytes of the data. The sender, instead of receiving of one ACK and increasing the congestion window once, receives two ACKs and therefore increases the window twice.

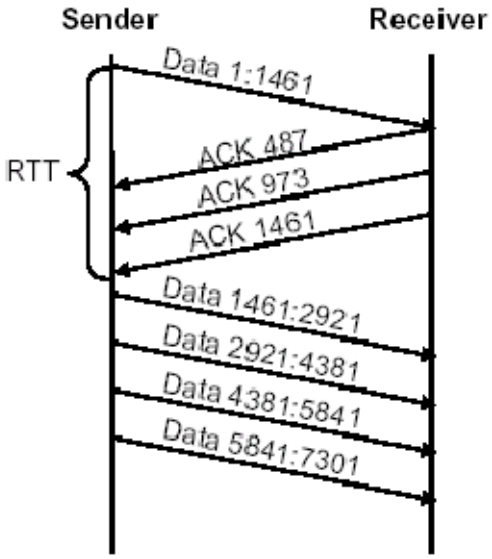


Figure 2.1: TCP trace under ACK amplification.

We call this phase an "amplification" phase. During this phase TCP client sends additional ACK packets. The amplification phase ends when we detect that the sender has entered the Congestion Avoidance mode on the server's side. In such an event we know that the server reaches its previous window size and we can return to the normal operation. (Algorithms for such a detection are described in Section 2.4).

2.3 Correctness

Even minor changes to the TCP state machine can result in the loss of utilization, increased congestion, or reduced fairness. Therefore we need to analyze the consequences of the multiple ACKs.

First of all, it is legitimate from the formal point of the TCP protocol. We can acknowledge the sender with the number, which is less than the packet size length. Historically, it was used by the low-buffered TCP machine, to indicate that the portion of the packet, was already received.

The second aspect of the ACK duplication, is a protocol behavior during

the loss of the the duplicated packets. The TCP acknowledges stream of the sequence numbers, rather the single packets. Therefore, if the first duplicated packet is lost, any server TCP windows are not changed. The second ACK, which successfully reach the sender, indicate to it that all data is received. On the other hand, in case that all ACK are lost, sender will behave, exactly as the one normal ACK is lost.

The only scenario with the different TCP behavior, is the case when the first ACK is delivered, and the second ACK is lost. In that case server knows the data position, to continue the stream. It is not important if the position indicates the first byte of the sent packet or not, since TCP works with streams, rather with packets.

The last aspect is a fairness. We obviously overrun other TCP flows, because we grow our window faster. But on the other hand, the reason for applying TCP Taba protocol was existence of multiple drops, which results in unfair TCP behavior. TCP Taba protocol stops when it reach the previous window value, and therefore it doesn't take the share of the other TCP streams. Its more accurate to say that we are fair "enough" to other TCP streams, but we are not give them a chance to overrun us because of the drop events.

2.4 Slow-start server detector

As we saw in the previous sections, our algorithm allows us to speed up the TCP sessions. It is important during the packet losses and the slow-start process. But this speedup process has its weakness: it generates additional ACK messages, and therefore increase the total throughput. In order to minimize this effect, we need to finish acceleration process as soon as server reachs congestion avoidance mode. The problem is to detect when this occurs, using only information available to the client. Our proposal for detection of this event is as follows:

First, lets analyze a typical sequence of packets that the client observes during server's slow start mode. The server starts with a small value for *cwnd* (typically between 1 and 4). After the client receives the data, and replies with ACKs, next the batch of packets from the server arrives at the client IP stack only after an RTT time. Due to the slow start mode the number of packets in the batch increases exponentially. Now let's see the interarrival times between packets - plotted Figure 2.2.

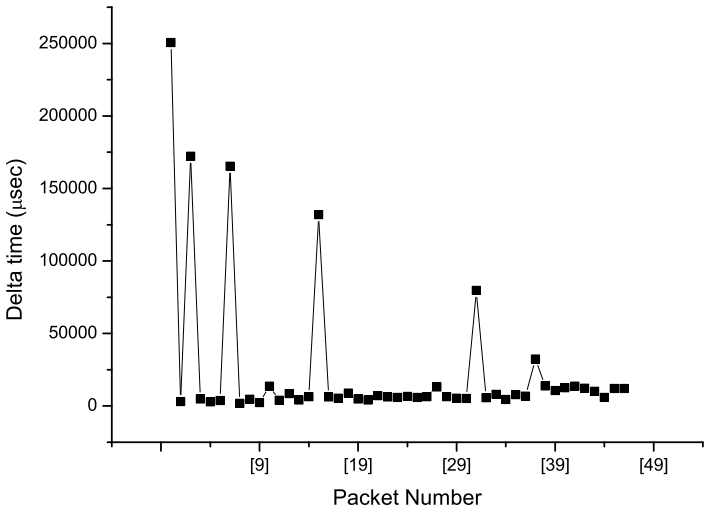


Figure 2.2: The duration between consecutive packets arrival on client's TCP stack.

As we can see the graph is a "decreasing exponent", with the variance which is caused by various components, such as the NIC adapter, the CPU or the routers in the path. Our aim is to find the time, where inter-arrival time between windows is roughly equals to the variance. TCP Taba algorithm uses *rtt* estimation, based on Van Jacobson algorithm [9]. This algorithm estimates the mean round trip time via the low-pass filter, as follows:

$$srtt_{i+1} = \alpha * srtt_i + (1 - \alpha) * rtt$$

where $srtt_i$ is the estimate, *rtt* is the round trip time measurement from the most recently acked data packet, and α is a filter gain constant with a suggested value of 0.9. Given the *rtt* time we can present our algorithms.

First we define floating-block size parameter (in our tests it was the 20 packets) and calculate local minimum and maximum for such block. Figure 2.3 shows maximum floating delta times, and Figure 2.4 shows minimum floating delta times, where the RTT time for this specific capture, measured on-line by the following algorithm based on the Van Jacobson algorithm as was shown above. Second, when a packet is sent over the TCP connection, the sender measures how long it takes for it to be acknowledged, producing

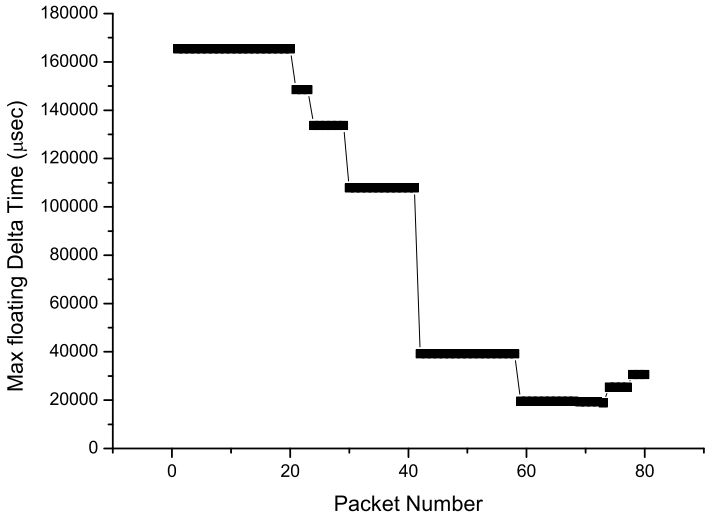


Figure 2.3: Maximum times between consecutive packets arrival on client's TCP stack.

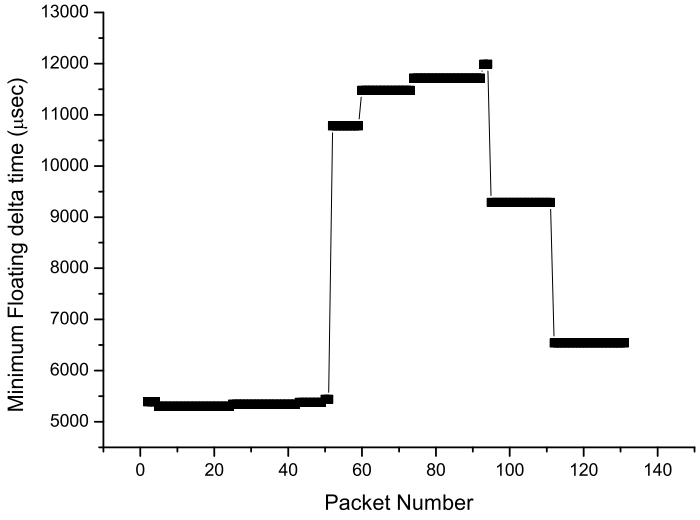


Figure 2.4: Minimum times between consecutive packets arrival on client's TCP stack.

a sequence, S, of the round-trip time samples: s_1, s_2, s_3.... With each new sample, s_i, the new *ssum* is computed by the formula:

$$ssum_{i+1} = \alpha * ssum_i + (1 - \alpha) * s_{i+1}$$

where $ssum_i$ is the previous estimation, $ssum_{i+1}$is the new estimation, and α is a constant between 0 and 1 that controls how rapidly the *ssum* adapts to the change. We used α=7/8 for our tests, which is similar to the value proposed by Van Jacobson value.

Lets define the local minimum for the block, which was decribed above as *MinFloat()*, and the local maximum for the block as *MaxFloat()*. Putting all together we are using the following algorithm to enable-acceleration:

1: **if** *MinFloat()* $\simeq$ *MaxFloat()* **then**
2: *RETURN to normal TCP*
3: **else**
4: **if** *MaxFloat()* $\geq \beta * ssum()$ **then**
5: *START Taba TCP*
6: **else**
7: *RETURN to normal TCP*
8: **end if**
9: **end if**

The maximum of the *MaxFloat()* is approximately the RTT time. The minimum of the *MaxFloat()* is slightly above 0. Therefore the parameterβ is a constant, which is set to a value between 0 up to 1. We tested this parameter and found it to be optimal if we set β=0.6 for all flows.

Chapter 3

Results and Discussion

3.1 Wired networks results

This section gives the results of our experiments. In the tests TCP Taba was tested and compared to standard TCP (RENO and new RENO) in real network conditions. Packet delay and packet loss were emulated using of our QoS router.

3.1.1 Slow start process acceleration

Our first goal was the creation of an environment, which emulates a long Slow-Start phase. Such long phases exist in modern high-bandwidth and high-latency links, and noisy wireless connections. Typical TCP stacks were created for low-bandwidth, high-latency links and therefore are very conservative in high-speed environments, and thus it takes for them a lot of time to fill in existing network pipe. We want to show that using TCP Taba technique we can achieve the saturation speed quicker, therefore boosting TCP performance.

3.1.1.1 Description of Testbed and the test procedure

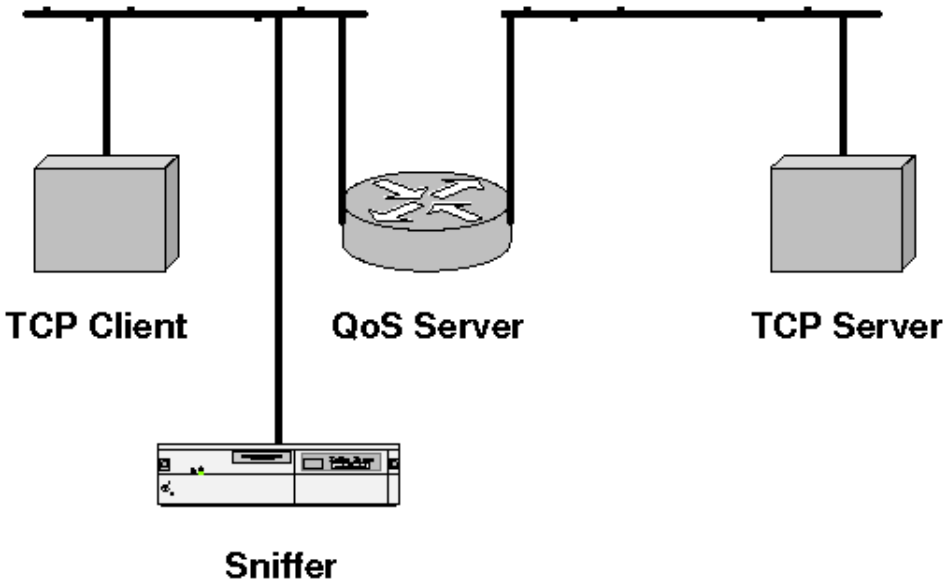


Figure 3.1: WAN emulation connection scheme

In order to emulate a high-bandwidth and high-latency links, we create the following setup: Linux Workstation with modified TCP Taba stack connected to QoS Router (for detailed explanation about QoS Router see Section A.3), which connected to FreeBSD 3.0 FTP Server.

We choose FreeBSD 3.0 FTP server, to be assure that TCP stack on the server part behave exactly according to widely used TCP New Reno protocol. All tests which will be shown below support this suggestion. In our case FreeBSD TCP stack increase *cwnd*, for every ACK received, according to standard.

The use of Linux workstation gives us possibility to modify the kernel and to perform changes. We also use special Linux programs to present the TCP flow results. TCPdump sniffer (see [18]) was connected to the same network, to achieve OS independent logs. A schematic figure of our setup is presented in Figure 3.1.

Our QoS router emulates WAN links (it adds both delay and loss). Therefore it was possible to produce all tests based on standard 100Mbps network,

and not use physically high-latency links. The second reason for using of our QoS router was possibility to create very simply different bandwidth, latency and loss rate conditions.

The disadvantage of using our QoS router is the nature of the losses and delays it adds, which are slightly different from the real ones. In section 3.2.3 we produce the real tests, and compare them to the emulated ones.

In our tests we defined the RTT time to be equal to 2 sec. We use 100Mbps local network, where latency was minimal and add a one second delay from the QoS router to the client and one second delay from the QoS router to the server.

For these tests we created two different FTP sessions - one with standard (new Reno) TCP stack on Linux, and the second session with TCP Taba stack on Linux.

During the first session we expected to see regular slow-start progress, i.e., doubling of window every RTT time. Because of the large RTT time, we can easily detect each TCP window transfer.

During the second session we expected to see that the enhanced slow start phase will increase the congestion window much faster.

3.1.1.2 Test results

First we tested the TCP sessions with defined RTT (2sec), and 200KB file transfer (which was approximately 150 packets) using TCP New Reno. As expected, the window size during the slow start phase doubles after each RTT.

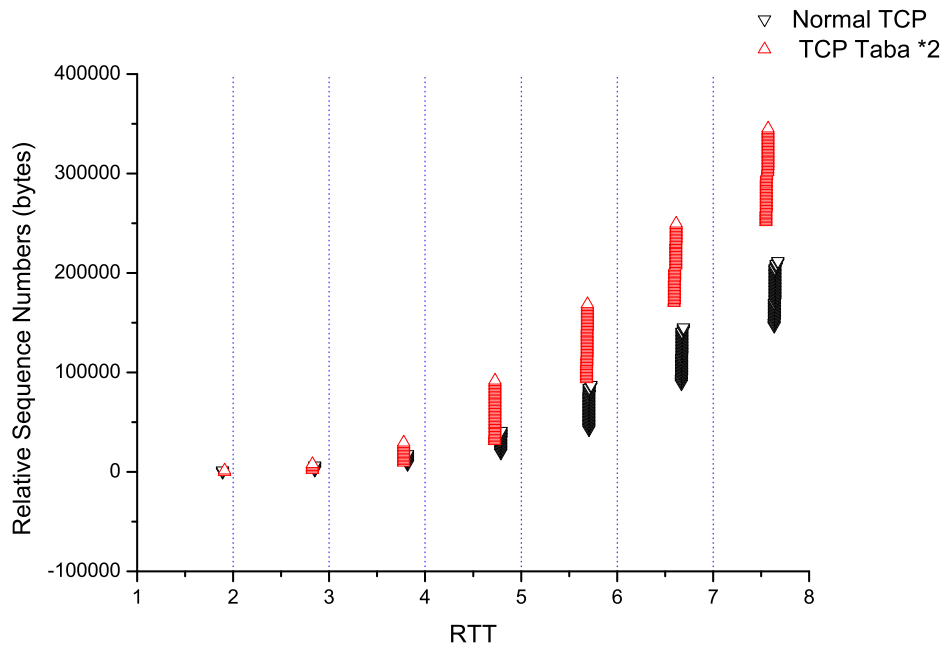


Figure 3.2: FTP session to FreeBSD FTP server with normal and TCP Taba stack. Normal TCP packets are marked as black triangles, and TCP Taba packets are marked as red triangles.

TCP starts with *cwnd* equals 1, and doubles *cwnd* every RTT time. In Figure 3.2 one can see that after successful SYN, and SYN-ACK packets (which don't carry data), therefore a first data packet was received after them (2RTT times). Immediately after that Normal-TCP client sends acknowledge, which results in three[1] data packets after approximately RTT time, sent by the server. From the third packet transfer *cwnd* doubles precisely twice every RTT time. In Figure 3.2 we show only 40 first data packets, therefore window sizes are 1, 3, 6, 12, 18. [2].

The second experiment was done with TCP Taba stack, which sends two acknowledges for every data packet received.

[1]This is a feature of FreeBSD TCP stack, to start TCP session with 3 data packets, rather 2 data packets according to TCP New Reno

[2]In Slow start the number of data packets, sent by server is equal to the *cwnd* of the previous step plus the number of currently received ACKs

The first TCP data packet was received after SYN and SYN-ACK (2 RTT) time like in a normal TCP session. TCP acknowledged with 1 packets like in normal session. After RTT time we receive 3 packets, which is the expected response in a normal session. After this TCP Taba starts an acceleration phase, and therefore send 6 ACK packets to acknowledge the 3 data packets, instead of required 3 by normal TCP protocol. We expected to see 9 packets after RTT time, which indeed happened. After 9 packets received, TCP Taba sent 18 acknowledges, for the data packages, thus we expects 27 packets (9+18), in the next RTT, which is indeed what happened.

We can conclude that we receive the empirical results about Normal-TCP and TCP Taba behavior, which fully supports our theoretical expectations. The amplification of TCP Taba protocol is considerably faster then Normal-TCP in "Slow-Start" part of TCP protocol.

3.1.2 Amplification parameter effect

Our second goal was compare different parameters of the TCP Taba protocol, where we use different amplification parameters, to send acknowledges to the server.

3.1.2.1 Description of Testbed and the test procedure

We use the same setup as in previous experiment, and captured all results with a sniffer. We expected to get different acceleration speed, using different amplification parameters.

The test procedure was following: we defined the RTT time to be equal to 2 sec. We use 100Mbps local network, where latency was minimal and added one second delay in the QoS router to the client and one second delay in QoS router to server.

For these tests we made 4 different FTP sessions - one with a Normal TCP, and three others with a TCP Taba modified stack on the Linux Workstation.

The first 2 tests were exactly the same, as in Section 3.1.1 - Normal TCP FTP transfer, and TCP Taba FTP transfer, where TCP Taba sent twice the ACK packets of a normal-TCP.

The third and the fourth tests were made with different amplification parameters (the third test with 3 ACK per packet, and the fourth test with 4 ACK per packet)

TCP \ Window	srv 1st	client 1st	srv 2nd	client 2nd	srv 3rd	client 3rd
Normal	1	1	3	3	6	6
TABA parameter 2	1	1	3	6	9	18
TABA parameter 3	1	1	3	9	12	36
TABA parameter 4	1	1	3	12	15	60

Table 3.1: TCP behavior due to TCP Taba amplification parameters. All tests were done with different TCP Taba amplification parameters (Normal, 2, 3, 4). Result TCP window sizes on Client (client) and Server (srv) are shown respectively.

We expect to see TCP speed improvements for higher amplification, and obviously that the results of amplification parameter 4 are better than amplification parameter 3 and amplification parameter 3 is better then amplification parameter 2.

3.1.2.2 Test results

Our results, show the acceleration speed for different amplification parameter (e.g. Normal, 2, 3, 4) .

Let us start with Table 3.1, which shows the difference in TCP window due to the difference between various amplification parameters. Please note that we start acceleration phase only after SYN, and SYN-ACK (2 RTT) data packets receive, therefore all types of TCP acknowledge with one ACK packet, after receiving one data packet. As shown, the result were as we predicted.

The sequence-time graph in Figure 3.3, shows normal TCP and TCP Taba with amplification parameter equal 2, 3, and 4. It it easy to see that amplification parameter 4 is better than, amplification parameter 3, and amplification parameter 3 is better than amplification parameter 2. All amplifications parameter were better than Normal-TCP.

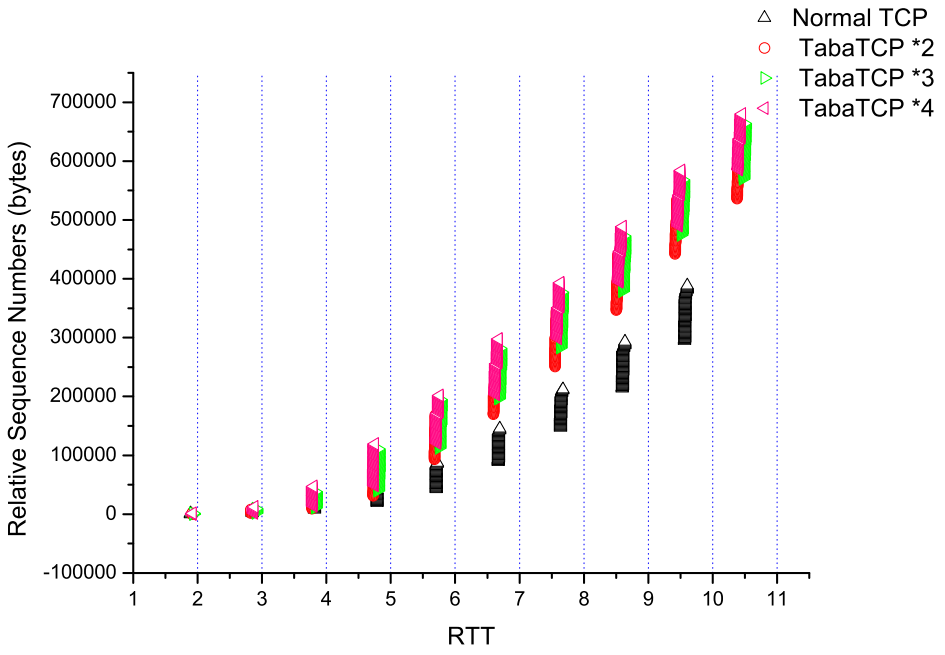


Figure 3.3: FTP session to FreeBSD with TCP Taba stack (Normal, parameter 2, parameter 3, parameter 4). Normal TCP data packets are marked as black triangles, amplification parameter 2 packets are marked as red circles, amplification parameter 3 packets are marked as green triangles, amplification parameter 4 packets are marked as pink triangles.

3.1.2.3 Remarks

There are two considerations, we made during these tests that should be mentioned:

The first observation is an interesting behavior, which was found during WinXP-SP2 tests. WinXP SP2 limits the number of "growth", for every RTT time. Table 3.2, is compares various amplification TCP Taba parameters (Normal, 2, 3 and 4) under Windows XP-SP2. From the table we can see that WinXP SP2 limits the growth with the amplification parameters.

The second observation is growth limitation phenomena due to reaching congestion avoidance phase after some time. Almost all OS systems we

TCP	1 Data	1 ACK	2 Data	2 ACK	3 Data-Exp	3 Data-Real	Limit
Normal	2	2	4	4	8	8	no
TABA param 2	2	4	6	12	18	18	no
TABA param 3	2	6	8	24	32	22	10
TABA param 4	2	8	10	40	50	28	22
TABA param 5	2	10	12	60	76	36	40

Table 3.2: WinXP limiting barriers. Data - actually received data packets (and equals the expected); ACK - expected and actually received ACK packets; Data-Exp - expected data packets; Data-Real - actually received data packets

checked still don't have scale-parameter for window size and are limited by a buffer size of 64KB.

In Figure 3.4, we can see that the fast window growth of TCP Taba with parameter 2 stops after the fifth RTT. The window rapid growth terminates after 11 packets received, and continue to grow slowly one packet per RTT. This imply that the server enters the congestion avoidance phase. In the congestion-avoidance phase, we resume the normal TCP.

The reason to return to the normal TCP is following: during congestion-avoidance phase of TCP session, the effect of the acceleration process is minor. We can't accelerate the growth of the window much by duplicating ACK packets. Moreover, we are consuming bandwidth with every additional ACK packet we send. Therefore we need to limit acknowledge packets in congestion-avoidance phase. (A detailed explanation of the reason to return to Normal TCP is found in Section 2.2). Thus, in congestion avoidance phase the behavior of parameter 2 and Normal Taba clients is similar.

How can we solve the short "slow-start" phenomena? One way is to increase the sender buffer size on Server's TCP/IP stack, which is impossible in our model, since we limit the change to the client TCP connection.

A second way to solve it, is to make sure that "window-scaling" factor is enabled, and was negotiated during TCP session handshake. We can make client's TCP stack aware, and expect that Server's TCP stack has "window-scaling" factor enabled.

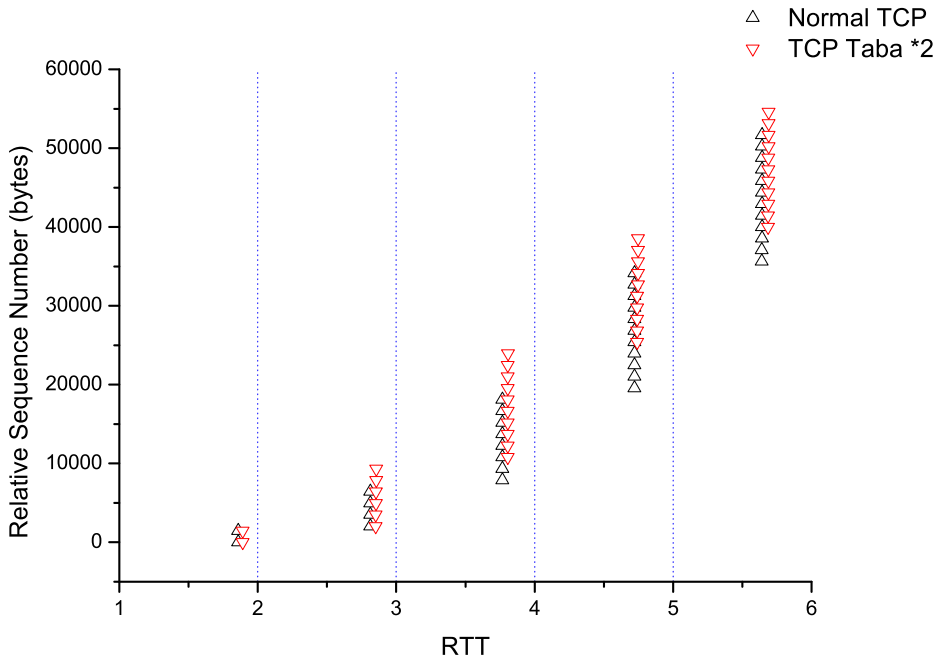


Figure 3.4: Congestion avoidance limit on FTP session to FreeBSD with TCP Taba stack (Normal and parameter 2). Slow start finishes after 5 RTT

3.1.3 OS Behavior

Our third goal was compare different Operation System behavior as Server OS, working with our TCP Taba client .

3.1.3.1 Description of Testbed and the test procedure

We use the same setup as in previous experiment, and capture all results with a sniffer. We expect to get different TCP behavior, using different OS systems. Regarding vast majority of Operational Systems (See Table 1.1), our algorithm works well.

Some of new OS have “TCP Taba” protection. They ignore additional ACK packets - and count both packet sizes and number of packets during a period of RTT time. Therefore we cannot achieve any acceleration in these

cases.

The test procedure was following: we defined the RTT time to be equal to 2 sec. We use 100Mbps local network, where latency was minimal and add one second delay from QoS router to client and one second delay from QoS router to server. We use TCP Taba FTP transfer, where TCP Taba used amplification parameter 2.

For these tests we made 2 different FTP sessions - the first one with WinXP SP2, and second one with SUSE Linux based on kernel 2.6.5

We expect to see TCP speed improvements for WinXP, similar to the results of FreeBSD 3.0. We also expected to see that Linux Workstation with Kernel 2.6 will ignore additional ACK packages, and that the performance of Normal TCP and TCP Taba would be identical.

3.1.3.2 Test results

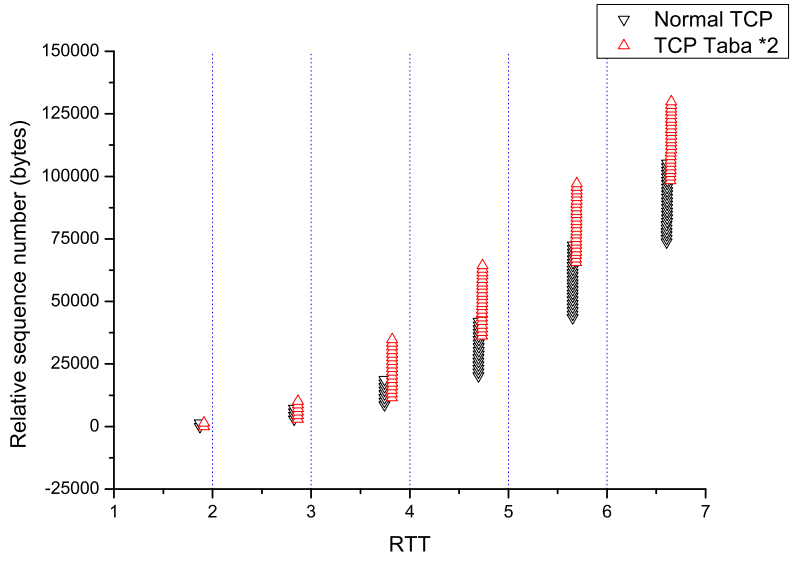
The results of the test show us, that we can classify the Operation Systems (See Table: 1.1) into three categories:

1. Standard TCP Tahoe, TCP Reno, TCP New Reno - majority of Operation Systems, like FreeBSD 3.0, work well with TCP Taba, and have no limits on window growth. We use this OS in tests we described above. (In Figure 3.2 there is example of such an acceleration).

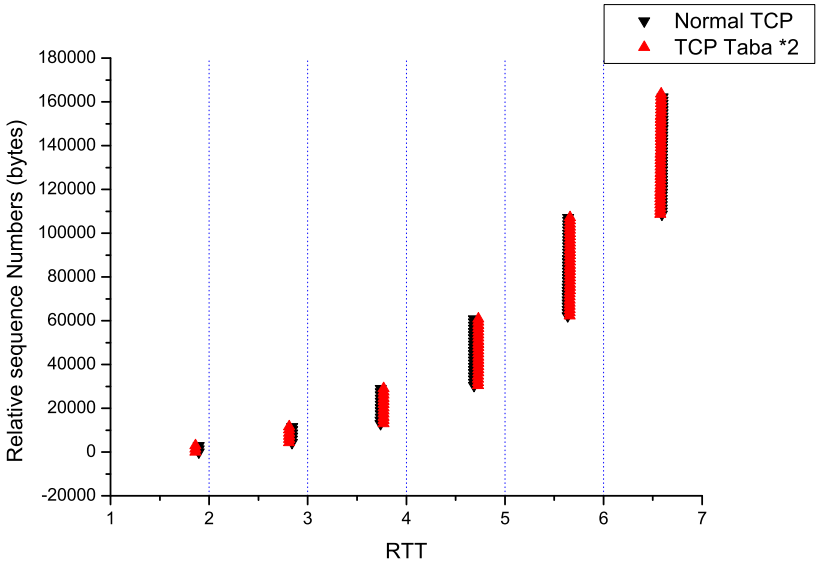
2. Proprietary modified TCP stacks - Windows based Operation Systems, like WinXP SP2, work well with TCP Taba, but have some limits on window growth. (See Figure 3.5 gives a comparison between WinXP SP2 workstation and Linux Workstation with Kernel 2.6).

3. New-coming Linux and *BSD stacks, like SUSE 10, with different levels of protection, to prevent our acceleration technique. (See Figure 3.5 for an example of such a protection.)

The conclusion, from the last set of experiments is that Linux 2.6.5 workstation, which operates as TCP server, doesn't allow us to accelerate the protocol. TCP behavior is similar for New-Reno and TCP Taba protocols. WinXP, as it was shown above, allows us to accelerate TCP traffic.



(a) WinXP



(b) Linux

Figure 3.5: Comparisson of FTP sessions between WinXP SP2 and Linux 2.6.5 with normal and modified TCP Taba (parameter 2)

3.2 Packet losses

Most interesting part of our tests is the analysis of TCP Taba performance when there is packet loss. As we mentioned before, standard TCP servers (Reno, New Reno) significantly decreases congestion window after multiple packet losses, and therefore our benefit should be larger, as we with increase loss rate.

The tests are divided into three parts: random packet loss simulation, burst packet loss simulation, and real network tests.

3.2.1 Random Packet Loss Simulation

3.2.1.1 Description of Testbed and the test procedure

We use our QoS router (see Section A.3) to simulate multiple random losses. We expect to see the differences between our TCP stack and normal stack behavior, because of the quicker return from Slow start.

We made several experiments using different loss ratios. The range of loss ratios was chosen between 0.5 to 5 percent. In experiments with larger random loss rate (more then 5%) the congestion window is almost always at its minimum size, and therefore the actual throughput is dominated by RTT time. Our QoS router allows us to set different loss ratios in each direction, therefore, another goal of our research was the effect of the loss ratio in different directions. We call the loss from Server to Client "*data losses*", and loss from Client to Server "*ACK losses*".

The test procedure of the experiments was as follows: we choose network delay between the TCP nodes (100 msec), we set the loss ratio for each direction, and TCP connection client type (TCP New Reno and TCP Taba). All tests were performed at least 3 times, and we report the average throughput.

We expect that TCP Taba accelerate traffic during Slow-Start phase, to enter "stop ACK duplication" during congestion avoidance phase, and therefore we expect to see the improvements of the TCP Taba protocol only during Slow Start phase. Also, we expect differences in the results due to RTT time, because relatively large RTT time causes every loss to have significant impact and prevent TCP to increase the congestion window quickly. It was expected that TCP Taba gives significant gains at high random loss rates and smaller gains at low random loss rates. We also expect that *data-losses* reduce the throughput significantly. In contrast *ACK-losses* only slightly

Num	Data loss (%)	ACK loss (%)	New Reno (kB/s)	TCP Taba (kB/s)
1	0	0	338	357
2	0.5	0	148	164
3	0	0.5	334	355
4	0	1.0	330	355
5	0.5	0.5	146	164
6	1.0	0	115	132
7	5.0	0	54	54

Table 3.3: Random losses test results; measured in kB/sec throughput, with FreeBSD 3.0 TCP stack; congestion window is 64KB; where RTT is 100msec during all tests; loss ratio from Client to Server (so-called ACK-loss), loss ratio from Server to Client (so-called data-loss).

reduce the throughput. Thus we expect that the performance of TCP Taba using combined *data-losses*, and *ACK-losses* would be similar to the effect of only *data-losses*.

3.2.1.2 Test results

In Table 3.3 we show the results of the difference between the behavior of normal TCP New Reno and TCP Taba with amplification parameter equals to 2. We use FTP transfer of the file with size is 3.81MB, delay of 100 msec, with different packet losses for each direction.

The first test has no-loss. It gave us the baseline for the following measurements, and also shows the basic advantage of the TCP Taba due to acceleration during the Slow Start phase.

The second test has *data-loss* of 0.5%. As expected, we see significant throughput fall (more then a half) during these tests. We can also see that TCP Taba gains are significant (approximately 10% in second test compared to 4% in first test), which show that TCP Taba returns to the congestion avoidance phase quicker than TCP New Reno.

The third and fourth tests have no data loss, and *ACK-loss* of 0.5% and 1.0% respectively. In general the results are similar to the first test results, with a very slight throughput decrease. We can also see the difference between the TCP Taba and TCP New Reno, because of the fact, that TCP Taba makes additional ACKs during Slow start phase, and therefore loss of such ACK packets are less influential than loss of the ACK packets during

TCP New Reno operation.

The fifth test has combined loss of 0.5% for data-loss and ACK-loss. As expected we can see that results are similar to the second test (data loss only).

The sixth test has 1% data loss and no ACK-loss. Degradation of the throughput after additional 0.5% of losses is less significant than after the first 0.5%. This is happens because of the fact that maximum TCP throughput degradation starts after first data-loss. We can see that TCP Taba gain 13%, which is slightly more them 10% with 0.5%loss, and much larger then 4% without loss.

Approximately at 5% loss ratio (seventh test) the throughput reaches its minimal level. This is due to *cwnd* window being at the minimal value most of the time. We note that TCP Taba results at loss ratio, above 5% are similar, to the TCP New Reno results, because of the collapse of the *cwnd* window to the minimal value. We want to mention that such high random loss rates are really rare in the real world.

3.2.2 Burst packet loss simulation

3.2.2.1 Description of Testbed and the test procedure

A burst loss is a sequential packet drop. Such drops can occurs due to lack of memory CPU resources, or due to physical media instability (wireless or radio networks). A burst loss is characterized by two parameters: the burst duration (# of consecutive packet loss), and the frequency of bursts.

We use our QoS router (see Section A.3) to simulate the burst losses. We expect to see the differences between our TCP Taba and normal TCP New Reno behavior, because of the quicker return from Slow start phase in TCP Taba.

There is a significant difference between "random" and "burst" losses, especially when considering the number of sequential packet drop. In Section 3.2.1 we varied loss ratio between 0.5% up to 5% losses. Even in case of 5% losses, the chance to find two sequential lost packet is only 0.25%. In case of burst losses the probability of sequential packet losses is very high. Recall that TCP is sensitive to sequential losses of packets. When the burst loss causes a timeout, the TCP connection starts new Slow Start Phase. We expect to see similar results to that of the Slow Start acceleration tests (See 3.1.1). Since such burst losses results in Slow Start initiation. As we see in

previous experiments, TCP Taba returns to the previous throughput quicker, and therefore has a significant throughput advantage in this case.

3.2.2.2 Test results

The explained results were in line with our predictions. Our burst duration is 5 RTT times (approximately 1 second), and burst frequency is 4 seconds, which mean 3 seconds between loss sequences. Such burst losses results in immediate Slow Start initiation. Various estimations of fade effects, talk about seconds of inactivity time, and therefore it is good estimation to predict at least 5 RTT of loss. In case of large losses (hundreds of packets) typical TCP Servers, and even more important, Application servers, close all TCP connections, in which case our improvement is not useful.

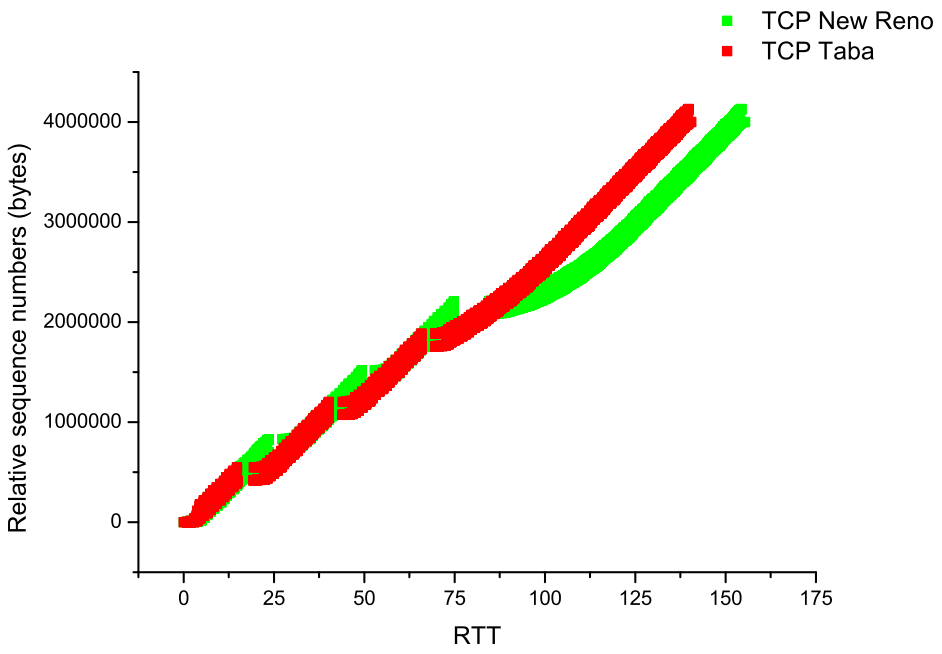


Figure 3.6: TCP session with 200 msec RTT, burst duration equals to 1 sec, no-burst duration equals to 3 sec.

Figure 3.6, shows the behavior of TCP Taba and TCP New Reno stacks

with 1 second loss out of a 4 second cycle. TCP Taba recovers quicker from the losses and therefore finishes the TCP tranfer before the TCP New Reno. Burst losses ends after 80 RTT, and we see return to normal TCP behaviour.

3.2.3 Real network tests

3.2.3.1 Description of Testbed and the test procedure

We use standard Bluetooth LAN PCMCIA Card and Bluetooth Access Point as wireless networks, and produce multiple losses by decreasing the signal strength between two Bluetooth peers.

First set of experiments was done using following setup.

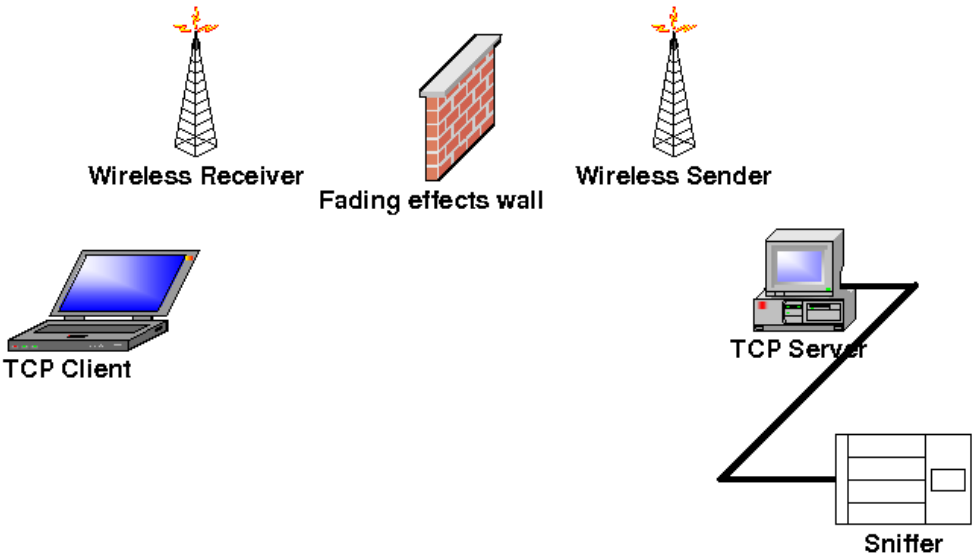


Figure 3.7: Real wireless connection scheme (fading effects)

The distance between source and destination was changed. Such a change results in immediate signal strength loss, add multiple losses and a TCP Slow Start initiation. During a single FTP session we kept unchanged distance between peers, and measured the session throughput (3 times each series). We made measurements using standard TCP client, and our TCP Taba client.

We choose FreeBSD 3.0 FTP server, to be assure that TCP stack on the server part behaves exactly according to widely used TCP New Reno

Num	Distance(m)	Noise (dB)	New Reno (kB/s)	TCP Taba (kB/s)
1	<1	0	53.36	58.7
2	4	-5	29.9	35.5
3	6	-9	23.06	31.7
4	8	-12	8.97	11.2
5	10	-16	3.35	3.64

Table 3.4: Real Bluetooth connection test results; measured in KBpsec throughput, with FreeBSD 3.0 TCP stack; congestion window is 64KB; Distance is measured in meters, and signal strength in dB.

protocol. In our case FreeBSD TCP stack increase, *cwnd*, for every ACK received, according to the standard.

In our tests we made FTP download a file of 3,999,607 Bytes, and we report the average throughput.

3.2.3.2 Test results

In Table 3.4 we show the results of the difference between the behavior of normal TCP New Reno and TCP Taba with amplification parameter equals to 2.

The first test has no-loss. It gave us the baseline for the following measurements, and also shows the basic advantage of the TCP Taba due to acceleration during the Slow Start phase.

The second test has signal strength at -5dB. As expected, we see significant throughput fall (more then a half) during these tests. We can also see that TCP Taba gains are significant (approximately 19% in second test compared to 10% in first test), which show that TCP Taba returns to the congestion avoidance phase quicker than TCP New Reno.

The third test has signal strength of -9 dB. We see additional difference between TCP stacks due to increase number of errors during the session.

The forth test, has signal strength as -12dB. We see major degradation for both TCP stacks.

Approximately at -16dB (the fifth test) the throughput reaches its minimal level. This is due to *cwnd* window being at the minimal value most of the time. We note that TCP Taba results at signal strength, above 16dB are similar, to the TCP New Reno results, because of the collapse of the *cwnd*

window to the minimal value. Such high noise ratio also results in regular Bluetooth disconnects, which prevent us from running consecutive tests.

Bibliography

[1] Tcp penetration. Available on http://www.cs.columbia.edu/ hgs/internet/traffic.html.

[2] J.B. Postel. RFC 793: Transmittion Control Protocol TCP. IETF, September 1981.

[3] John Nagle. RFC 896: Congestion Control in IP/TCP Internetwork. IETF, January 1984.

[4] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the 1st ACM Int'l Conf. On Mobile Computing and Networking (Mobicom)*, November 1995.

[5] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A Comparison of Mechanisms for Improving TCP Performance Over Wireless Links. *ACM/IEEE Transactions on Networking*, December 1997.

[6] William Stallings. *Data & Computer Communications*. Prentice Hall, 6th edition, 1999.

[7] Douglas E. Comer. *Computer Networs and Internets*. Prentice Hall, 3rd edition, 2001.

[8] W. Richard Stevens. *TCP/IP Illustrated, Volume 1, The Protocols*. Addison-Wesley, 1994.

[9] V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[10] Moraru etc al. Practical analysis of tcp implementations: Tahoe, reno, newreno.

[11] S. Floyd and T. Henderson. The New-Reno Modification to TCP's Fast Recovery Algorithm, April 1999.

[12] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. 26(3):5–21, July 1996.

[13] K. Ramakrishnan and S. Floyd. RFC 2481: A proposal to add explicit congestion notification (ECN) to IP, January 1999.

[14] T. Kelly. Scalable tcp: Improving performance in highspeed wide area networks, 2003.

[15] A. Maor and Y. Mansour. AdaVegas: Adaptive control for TCP vegas.

[16] H. Wu, Y. Peng, K. Long, S. Cheng, and J. Ma. Performance of reliable transport protocol over ieee 802.11 wireless lan: Analysis and enhancement, 2002.

[17] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion contorl with a misbehaving receiver. *Computer Communication Review*, 29(5), October 1999.

[18] Tcpdump. Available on http://tcpdump.org.

[19] Tcptrace. Available on http://tcptrace.org.

[20] Xplot. Available on http://xplot.org.

[21] Network emulator. Available on http://http://developer.osdl.org/shemminger/netem.

[22] Vmware. Available on http://www.vmware.com.

[23] User mode linux. Available on http://user-mode-linux.sf.net.

[24] Availalble on http://www.kernel.org.

[25] Ethereal. Avalible on http://www.ethereal.com.

Appendix A

Detailed description of TCP Taba

Our algorithm was implemented in the network stack of the Linux 2.4.24 kernel. Our TCP patch adds the congestion window algorithm changes, which operates only in the client , without interference with the server TCP stack. We use User Mode Linux, which is a version of the Linux kernel that runs as a process on top of Linux and VmWare, for rapid development. It gives us a full (although slow) Linux systems. We use ***ethereal*** and ***tcpdump*** packages for network capture, and ***tcptrace*** package for TCP data analysis.

A.1 Our changes in Linux Stack

As it was described in Section 2.2 we need to divide a single acknowledge into a few other ones, which every one of them will acknowledge the previous one, and the last will acknowledge the original data segment.

A.1.1 Enable/Disable TCP Taba

We need a effective way to enable and disable such improvement, and use proc file system (see Section B.4.4) as a trigger for that operation. We added new TCP parameter:

tcp_taba - How many times TCP will multiply outgoing ACK messages. The default is 0 - disable;

Important to note that we can change proc parameters even for the same TCP session, therefore making possible to change the way we're dealing with acknowledges without closing the session.

A.1.2 Changes in TCP core

We need to change original TCP stack, according to our algorithm. We already have enable/disable mechanism, and we need now to hook incoming TCP packages, wait until TCP is ready to acknowledge data, and instead of sending ACK, send multiple ACKs.

As it was described in Section B.4.3 we are waiting for the ACK, which generates by Linux stack, in tcp_send_ack routine. If all conditions are successfully, we should have:

- *tp->rcv_nxt* - the final ACK number we need to acknowledge.
- *last_int_ack* - the last acknowledged number for the same session.

All we need to do now, is to divide the number of unacknowledged bytes (*tp->rcv_nxt- last_int_ack*) by the *tcp_taba* amplification factor, and to create relevant ACK messages.

If *tcp_taba* is disabled, or we do not have enough unacknowledged bytes, we proceed with the original TCP flow.

A.2 Visualizing TCP protocol

There are several excellent tools, which helps to build readable TCP traces, and analyze its behavior. But the classical one is TCPdump[18], TCPtrace[19], Xplot [20] sequence, where TCPdump make capture, TCPtrace divide capture into TCP flows, and creates vector graphs, about almost all interesting TCP parameters, and Xplot display them.

A.3 QoS Server

One of the challenges we have to solve is how to simulate high latency, high bandwidth links without the need of expensive hardware. The solution was to use a Linux Workstation, with kernel 2.6.8, and enabled module of traffic shaper, and netem (NETwork EMulator) [21].

A.3.1 Emulating wide area network delays

This is the simplest example, it just adds a fixed amount of delay to all packets going out of the local Ethernet.

```
# tc qdisc add dev eth0 root netem delay 100ms
```

Now a simple ping test to host on the local network should show an increase of 100 milliseconds. Later examples just change parameters without reloading the qdisc.

A.3.1.1 Delay distribution

Typically, the delay in a network is not uniform. It is more common to use a something like a normal distribution to describe the variation in delay. The netem discipline can take a table to specify a non-uniform distribution.

```
# tc qdisc change dev eth0 root netem delay 100ms 20ms distribution normal
```

A.3.2 Packet loss

Random packet loss is specified in the 'tc' command in percent. The smallest possible non-zero value is: .0000000232%

```
# tc qdisc change dev eth0 root netem loss .1%
```

A correlation value can also be specified.

A.3.3 Packet duplication

Packet duplication is specified the same way as packet loss.

```
# tc qdisc change dev eth0 root netem duplicate 1%
```

A.3.4 Packet re-ordering

Packet re-ordering causes 1 out of N packets to be delayed.

```
# tc qdisc change dev eth0 root netem gap 5 delay 10ms
```

So the 5th (10th, 15th, ...) packet will get delayed by 10ms and the others will pass straight out.

Appendix B

Development environment.

B.1 Basic system.

B.1.1 The Linux kernel.

Linux is a free Unix-like operating system kernel originally created by Linus Torvalds who was later assisted by developers around the world. The source code for Linux is freely available to everyone under the GNU General Public License. Linux was chosen as a base for the implementation for the following reasons:

- free source code
- availability of extensive documentation
- existence of an extremely active community of developers and users around the world
- pre-dominance in the research community - others could use my code

The last stable version of the Linux kernel at the time was 2.4.24 and this version has been used for the project. There are some differences in the networking stack implementation between 2.2.x and 2.4.x kernels. Nevertheless, it should be easy to port the implementation to 2.2.x and possibly to the newest series of 2.6.x kernels.

B.1.2 Operating system.

Since Linux is only a kernel (often mistakenly called an operating system), it doesn't provide any user interface. A set of tools, utilities and libraries working together with the kernel make a usable operating system. These tools are mostly provided by the GNU project and every Linux distribution is built on top of them. The operating system used for this project was Red Hat v.9.0., but any distribution able to work with 2.4.x kernels could be used.

B.2 Development tools.

B.2.1 Basic tools.

- gcc GNU project C compiler was used along with make and other common development utilities.
- vim (vi clone improved) was used as an editor.

B.2.2 Advanced tools.

- VmWare [22]- Intel x86-based virtual machine. It gives an OS the illusion of running on standard PC hardware, but it isolates it from the real hardware and from other activities of the host OS. This setup allows for testing newly compiled kernels without re-booting the PC itself (normally a kernel can be loaded only during the boot process) thus making the development cycle shorter. Nevertheless, to 'boot' the VmWare virtual machine takes about as much time as to boot the PC itself. VmWare is a commercial product. The version used here was 4.52.
- UML (User-Mode Linux)[23]. It is essentially a kernel patch that allows for compiling a Linux kernel as an executable and running it as a process. This creates a great opportunity for using standard debugging tools (like gdb) on a kernel. It is also a safe way of performing any tests, because the kernel runs as a process with user privileges, thus preventing damage to hardware and software. To make use of it a loop-back file system was created and RedHat 9.0 was installed inside, thus making the complete GNU/Linux system running as a regular process. The

loop-back file-system is a file-system created inside a regular file with the *mkfs* command and then mounted with the *mount* command with *-o loop* option specified. Various virtual-devices (such as hard-drives) can be specified from the UML command line. UML will access them through standard system calls. The boot process is very quick and takes less than 10 seconds on an Intel 2.4GHz, 512MB ram, IDE HDD. User-Mode Linux and all the supporting documentation is available for free under the GPL license.

Setting up User-Mode Linux for purposes of kernel development:

1. Get the Linux kernel sources of your choice ([24]).
2. Download the User-Mode patch for the appropriate version of Linux kernel, root-file-system to boot it on and documentation[23]
3. Follow the User-Mode documentation. You will basically have to patch your kernel sources, compile them, create a loop-back file-system and install some Linux distribution on it (Slackware is recommended) and finally boot your kernel with this file-system as a root-file-system. All these steps are covered by documentation available from the User-Mode Linux website.
4. Run a compiled kernel passing as an argument the file-system created above (do not run it as root):

```
$ ./linux ubd0=<file-system's path> eth0=tuntap,,,192.168.0.1
```

ubd0 will become a first hard-drive; eth0 will be first ethernet adapter

If run under X-Windows, User-Mode linux will use xterms as consoles. To control the number of consoles edit /etc/inittab file. Once xterms with login prompt show up, you can log-in and run programs from the shell-prompt as usual.

From our experience User-Mode Linux is a better solution for this project than VmWare for the following reasons:

- much faster boot process

  - a very unique possibility of debugging the kernel with gdb
    * less resource-intensive solution (VmWare runs several daemons and a GUI front-end, User-Mode Linux uses 2 xterms)
    * ability to run with non-root privileges
    * it is not necessary to re-install the kernel using lilo after each re-compilation (because it is just a regular executable)

From the other side VmWare is better than UML in the following cases:

- ability to work with real ethernet driver, in bridge mode, which operates network traffic without network latency, faster than on UML machine.
- User-Mode Linux handle of a large amount of debugging information, which was sent to the klogd (Kernel Log Daemon). In extreme cases some of our output was lost, whereas it was always saved properly by VmWare.

Network tools.

- Ethereal[25]- Network sniffer and analyzer.
- Tcptrace [19] - TCP analyzer

B.3 Testing environment.

In order to test protocol we added TCP Taba code into the networking part of Linux kernel.
The development cycle basically consists of:

1. Writing the code and compiling the kernel.
2. Running the kernel (either by installing it with lilo and rebooting the machine/VmWare or by running User-Mode Linux).
3. Starting ethereal in capture mode
4. Running the test application in one of the scenarios. Scenarios are as follows (client always initiate the connection):

(a) server sends data, client receives, server closes first

5. Analyzing TCP data with tcptrace and plotting it with xplot

6. Reading the debugging info stored by klogd in '/var/log/messages'.

   One can run the User-Mode Linux in the 'debugging' mode

   ```
   $ ./linux ubd0=<root_fs> debug
   ```
   in which case an additional xterm will show up with gdb launched. gdb can be used as usual; the advanced debugging issues (debugging the kernel threads) are described in the User-Mode Linux documentation.

B.4 Basic structure of the Linux networking stack.

B.4.1 Sources arrangement.

The majority of the sources related to networking is located in the net/ subdirectory of the main source tree. net/core contains some generic routines (like generic socket support) while net/ipv4 contains all the network and above layer's protocols built on top of IP v.4. Appropriate header files can be found in include/net directory. Device drivers for network interfaces are located in drivers/net directory.

B.4.2 Process of receiving the packet from the network and delivering it to the user space.

- Device receives a packet and generates an interrupt.
- Device driver adds the packet to the global receiving queue (so it can be handled later on) and raises the software interrupt (so the data can be processed by any CPU later on).
- Software interrupt handler calls the appropriate network-protocol handler for the packet (here - IP).

- IP makes some checks, including recognition of the higher level protocol's id and calls the appropriate transport-level handler for the packet.
- Transport protocol eventually delivers the data to the user space (a socket).

B.4.3 Process of building TCP ACK and scheduling it into the queue

- Each element in Linux stack is a pointer to the *struct sk_buff* structure, which contains fields for various elements of the packet (like link, network and transport layer header). This struct is filled in by a device driver.

```
struct sk_buf is defined in include/linux/skbuff.h
```

- As we said above, interrupt function calls the protocol handler (using "func" pointer mentioned in (a) above) on a packet. The function registered for IP happens to be ip_rcv, defined in net/ipv4/ip_input.c as:

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev,
        struct packet_type *pt)
```

- ip_rcv is a main IP-handling function. It first makes some general checks on a packet and drops it if something is wrong.
- The packet is passed to a higher level protocol by extracting the appropriate protocol id from the packet and then calling ipprot->handler(skb) in the ip_local_deliver_finish. All the constants for protocols' ids (like IPPROTO_TCP=6, or IPPROTO_UDP=17) are defined in include/linux/in.h, according to RFC 790. It's a place where one could add a new protocol number. Now, before this could happen, the transport protocols must have been registered. This is done by inet_init function, called at the kernel's start-up.

```
net/ipv4/af_inet.c : static int __init inet_init(void)
```

- This function adds all the transport protocols to the list, calls initialization functions for IP, ARP, ICMP, creates /proc entries, etc. This is the place where stack receives TCP Taba initialization parameters.
- Data delivered to tcp handler, and after a while outgoing ACK is build:

  `net/ipv4/tcp_output.c : void tcp_send_ack(struct sock *sk)`
  The tcp_send_ack creates the ACK packet, finds a socket for it and queues the packet on the socket's receiving queue (so it can be picked next time the user process is woken). Here is the place, where we can add our amplification routines.

B.4.4 Proc file system

The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl).

A very interesting part of /proc is the directory /proc/sys. This is not only a source of information, it also allows you to change parameters within the kernel. To change a value, simply echo the new value into the file.

An example how to change IP TTL is given below in the section on the file system data. You need to be root to do this. You can create your own boot script to perform this every time your system boots.

```
#echo "205" > /proc/sys/net/ipv4/ip_default_ttl
$ ping localhost
PING localhost(127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=205 time=0.058 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=205 time=0.057 ms
```

The files in /proc/sys can be used to fine tune and monitor miscellaneous and general things in the operation of the Linux kernel. The interface to the networking parts of the kernel is located in /proc/sys/net.

We will concentrate on TCP networking here.

1. tcp_ecn - Enable ECN after RFC2481
2. tcp_keepalive_time - How often TCP sends out keep alive messages, when keep alive is enabled. The default is 2 hours.

3. tcp_sack - Enable select acknowledgments after RFC2018.

4. tcp_timestamps - Enable timestamps as defined in RFC1323.

5. tcp_stdurg - Enable the strict RFC793 interpretation of the TCP urgent pointer field. The default is to use the BSD compatible interpretation of the urgent pointer pointing to the first byte after the urgent data. The RFC793 interpretation is to have it point to the last byte of urgent data. Enabling this option may lead to interoperability problems. Disabled by default.

6. tcp_window_scaling - Enable window scaling as defined in RFC1323.