Practical Analysis of TCP Implementations: Tahoe, Reno, NewReno

Bogdan Moraru
Technical University of Cluj-Napoca
Bogdan.Moraru@com.utcluj.ro

Flavius Copaciu
Technical University of Cluj-Napoca
Flavius.Copaciu@com.utcluj.ro

Gabriel Lazar
Technical University of Cluj-Napoca
Gabriel.Lazar@com.utcluj.ro

Virgil Dobrota
Technical University of Cluj-Napoca
Virgil.Dobrota@com.utcluj.ro

Abstract

The paper presents the experimental evaluation of the existing TCP implementations: Tahoe without Fast Retransmit, Reno, New-Reno. The short time analysis involved a software tool called TBIT (TCP Behavior Inference Tool), which was designed by AT&T Center for Internet Research. It generates short TCP traffic (about 25 segments), with the 13th and the 16th segments intentionally dropped. Depending on the type of TCP implementation the behavior was different, due to the activation/missing of the following congestion control algorithms: "Slow-Start", "Congestion Avoidance", "Fast Recovery" and "Fast Retransmit". TCP segments were captured at both ends of the TCP connection using tcpdump tool and then the data was analyzed with several programs (tcptrace, xplot and proprietary programs developed for Linux Red Hat).

1. Introduction

During the last years, computer networks have experienced tremendous growth. More and more computers get connected to both private and public networks, the most common protocol stack used being TCP/IP.

Nowadays it is difficult to identify the congestion control algorithms that are currently implemented by various machines in Internet. The TCP header does not provide any information about them.

Another important issue is the way that these algorithms are implemented in different operating systems. By this time, the most frequent TCP implementation for clients is based on the Windows 2000 kernel. On the other hand, most Internet servers use various FreeBSD or Linux-based versions.

The related work on TCP congestion control covers at least two major issues. The first one includes simulations based on theoretical analysis of TCP implementations, such as in [1]. Although new ideas could be tested, this kind of work is not always close to real implementations from the operating system's kernel. For this reason, a second major issue is focused on real TCP implementations, such as in [2].

TCP is trying to provide reliable data transmission between two entities. It implies anyway to handle packet losses, that are due to transmission errors or traffic congestion.

2. TCP congestion control

Let us define the following parameters:

- *sender maximum segment size* (`smss`) represents the maximum amount of data that can be sent in a single TCP segment, without including the header.
- *sender's 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.
- *receiver's window* (`rwnd`) is the latest window advertised by the receiver.
- *congestion window* (`cwnd`) is a TCP state variable, limiting the amount of data that can be sent.
- *loss window* (`lw`) is the value of the congestion window after a packet loss has been detected.
- *slow-start threshold* (`ssthresh`) is another TCP state variable that determines the congestion control algorithm to be employed: either slow-start (if $\mathtt{cwnd} \leq \mathtt{ssthresh}$) , either congestion avoidance (if $\mathtt{cwnd} \geq \mathtt{ssthresh}$).

Basic TCP congestion control is done using Slow-Start and Congestion Avoidance algorithms, based on the work initiated by Van Jacobson.

According to [3] these algorithms are mandatory, but they could be accompanied by two new ones: Fast Retransmit and Fast Recovery.

2.1 Slow-Start and Congestion Avoidance

These two algorithms must be implemented by TCP entities in order to control the amount of data sent over the network.

The Slow-Start, improperly called like this, actually increases exponentially the size of the congestion window. It is used by a TCP entity at the beginning of a transmission or after detecting a packet loss. The purpose of Slow-Start is to fill as soon as possible a transmission channel.

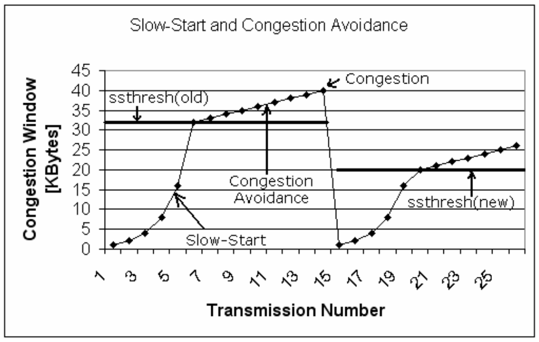

Figure 1. Slow-Start and Congestion Avoidance

After the congestion window has reached the threshold value, the Congestion Avoidance algorithm is employed. It continues to increase linearly the congestion window, adding up to one SMSS but not less then one byte. In both cases a retransmission timer is used for every packet. The timeout signals the loss of the packet. This leads to the retransmission of that packet and halving of the Slow-Start threshold. The congestion window is also set to the value of the loss window.

2.2 Fast Recovery and Fast Retransmit

After implementing the previous two algorithms, new problems arise. The first one is related to the packet loss detection. Normally a packet loss is inferred based on the timeout of the retransmission timer. This, however, may lead to significant delays in data transmissions, so another way to determine packet loss has been added to TCP.

Under normal circumstances a TCP entity must send a duplicate ACK for every packet that arrives out of sequence. A packet may be received out of sequence due to packet duplication by the network, packet delays or loss.

The Fast Retransmission algorithm considers that a packet has been lost when it receives 3 duplicate ACKs, before the timeout of the retransmission timer. In this way valuable time is saved.

The second problem is related to the drastic decrease of the congestion window after a packet loss detection. If a packet is lost during Slow-Start or Congestion Avoidance the value of the congestion window is set to the value of the loss window (1 SMSS). The Fast Recovery algorithm tackles this problem. The new value of the congestion window after a packet loss is detected by the Fast Retransmission is set to ssthresh + 3 SMSS. This is called "artificial inflation" of the congestion window. Beside that, for every new duplicate ACK received the congestion window is further increased with 1 SMSS.

3. Tools and Environment

3.1 Tools

In order to identify the TCP implementation within the operating system's kernel, an apache web server should run on the tested host. This software is free and there are ports available for all the systems we tested.

The TCP packets exchanged between the testing and the tested system were captured using `tcpdump` and stored for analysis. On Windows systems `ethereal` was preferred. The most important tool we used was TBIT (TCP Behavior Inference Tool).

This tool was developed at AT&T Center for Internet Research and it can be used to characterize the behavior of a TCP implementation from a distant machine running a web server.

Generally speaking TBIT works like a regular web browser: it establishes a TCP connection to the web server and requires a web page. TBIT builds it's own TCP packets and uses an IP socket to send them to the server. It also uses a Berkley Packet Filter to prevent the TCP packages received from the web server from reaching the operating systems kernel and to redirect them towards TBIT. Then TBIT creates controlled packet loss by confirming only certain received packets. The web server interprets those losses as a sign of congestion and reacts according to the congestion control algorithms it implements. This reaction can be analyzed, the algorithms used can be recognized and the TCP version estimated.

3.2 Network Configuration

In order to perform the experiments we used 2 machines: the testing system and the tested system. The testing machine was based on FreeBSD 4.5 with a recompiled kernel (according to TBIT specifications), TBIT and `tcpdump`. On the tested system we installed various versions of FreeBSD, Linux and Windows. A web server plus `tcpdump` or `ethereal` ran on the system. The testbed configuration is presented in Figure 2.

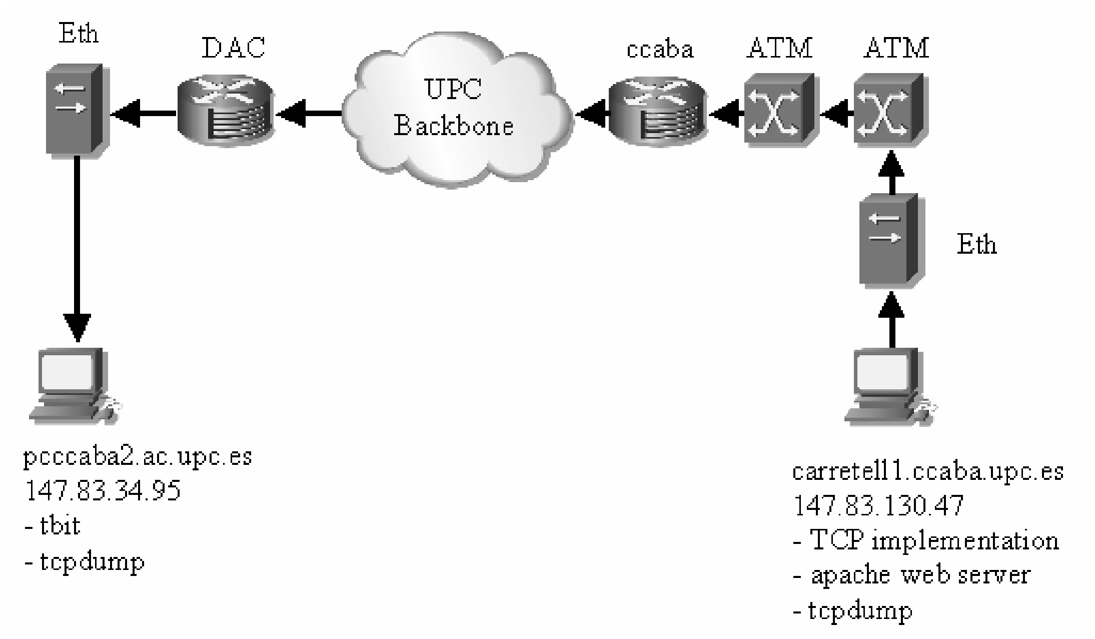

Figure 2. Testbed Configuration

The tests were performed as follows:

1. The http daemon was started on the tested system.
2. `tcpdump` was started on both systems. The captured packets were written into a file for future analysis.
3. TBIT was started with the required parameters on the testing system. The TBIT output was redirected to another file for analysis.
4. After the TBIT test was finished, both the `tcpdump` and `httpd` were stopped.

At the end of the tests we proved that both the TBIT results and those from the dump files converged. They are indicating the same TCP implementation.

4. Experimental Results

4.1. TCP Implementations

According to [2],[4], [5] the most popular TCP implementations are the following:

- *Tahoe without Fast Retransmit*: includes Slow-Start, Congestion Avoidance.
- *Tahoe*: includes also Fast Retransmit.
- *Reno*: adds Fast Recovery to Tahoe TCP.
- *New-Reno*: enhanced Reno TCP using a modified version of Fast Recovery.
- *Reno Plus*: on some Solaris systems.
- *SACK*: uses selective acknowledgements.

Other current TCP implementations are Vegas, 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.

4.2. Tahoe without Fast Retransmit

The TCP sender that implements Tahoe without Fast Retransmit does not count the duplicate ACKs in order to determine if a packet has been lost. The sender infers that a packet has been lost only when the retransmission timer expires.

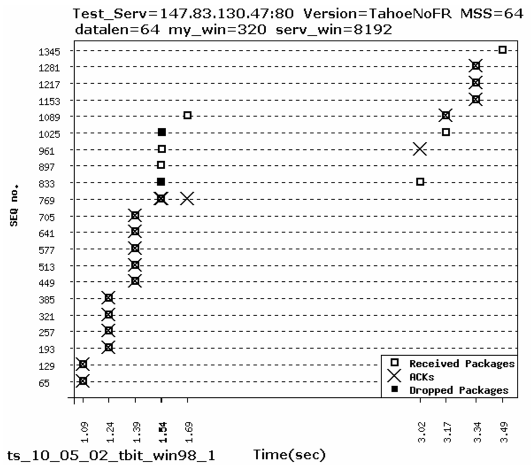

Figure 3. Tahoe without Fast Retransmit

This implementation includes two algorithms only: Slow Start and Congestion Avoidance. Figure 3 describes the working mode of this implementation:

1. The first 12 packets are acknowledged appropriately.
2. Packet 13 is dropped.
3. Packets 14 and 15 are acknowledged, but the ACKs sent are duplicate ACKs for segment 12
4. Packet 16 is dropped
5. Packet 17 is acknowledged, but the ACK sent is a duplicate ACK for segment 12
6. The last 5 segments were not acknowledged properly so the sender cannot send anymore packets.
7. The transmission restarts (with Slow Start algorithm) when the retransmission timer for packet 13 expires (timeout). Segment 13 is retransmitted.
8. The ACK generated because of the correct reception of packet 13 is the ACK for packet 15, because packets 14 and 15 are already in the receiver's buffer. This ACK segment acknowledges segments 13, 14 and 15.
9. Packet 16 is retransmitted, but there is also an useless retransmission of packet 17 because this packet is already in the receiver's buffer

TCP TahoeNoFR is characterized by a retransmission timeout for segment 13 and an useless retransmission of segment 17.

The situation when multiple packets are lost from one window is almost similar with the situation when there is only one packet lost from that window. The first lost packet will generate a retransmission timeout (a lot of time wasted), and all the lost packets will be retransmitted immediately afterwards.

This implementation performance is very poor especially when at least one packet per window is lost and packet loss happens very often. This would lead to many timeouts.

4.3. Tahoe TCP

Tahoe implementation added a number of new algorithms and refinements to earlier implementations (including TCP without Fast Retransmit). The algorithms suite included Slow-Start, Congestion Avoidance and Fast Retransmit. With the latter one, after receiving a 3 duplicate acknowledgments for the same TCP segment, the data sender inferred that a packet has been lost and retransmitted the packet. Note that this happened before the retransmission timer generated timeout, leading to a higher channel utilization and connection throughput. Unfortunately, in practice we were not able to find any workstations in the Internet currently using this TCP version.

4.4. Reno TCP

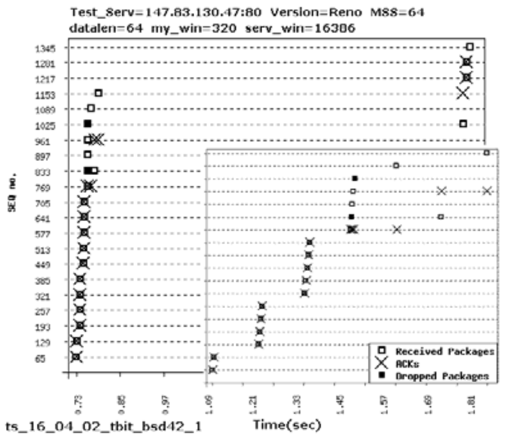

Figure 4. Reno TCP

Reno implements two new algorithms beside the those ones implemented by TahoeNoFR: Fast Retransmit and Fast Recovery.

In Figure 4 the area for the first 18 segments is zoomed because the initial figure might create the idea that there is no Slow Start, which is not true. The implementation works as follows:

1. The first 12 packets are acknowledged appropriately .
2. Packets 13 and 16 are dropped.
3. Segments 14, 15 and 17 generate duplicate ACKs for segment 12. Because of the 3 consecutive duplicate ACKs, Fast Retransmit and Fast Recovery algorithms are started.
4. Packet 13 is fast retransmitted.
5. The received ACK confirms packets 13, 14, and 15, and asks for segment 16. This is a new and distinct ACK and because of it Fast Retransmit algorithm ends and a new packet is transmitted: 18.
6. Packet 18 generates a duplicate ACK for packet 15.
7. Since there are no new and distinct ACKs, no more data can be sent. Because there aren't enough duplicate ACKs to start the Fast Retransmit algorithm for packet 16, transmission restarts only when the retransmission timer for packet 16 generates timeout.
8. When the timer expires, packet 16 is retransmitted, and because packets 17 and 18 are already in the receiver's buffer, an ACK for packet 18 will be generated.

Fast Retransmission algorithm solves one problem from TahoeNoFR: there is no timeout for the first packet lost for one window. But this happens when we have a multiple packet loss from the same window. Reno TCP works best for only one lost packet per window. Another problem of TahoeNoFR that is solved by Reno is the useless retransmission of packet 17.

This was a problem because we were loading the network with unnecessary packets since they are already in the receiver's buffer.

Reno TCP is characterized by a Fast Retransmit for packet 13, a Retransmit Timeout for packet 16, and no unnecessary retransmission of packet 17 (for the scenario described in Figure 5).

4.5. NewReno TCP

NewReno TCP is a variant of Reno with a little modification within Fast Recovery algorithm. This was done in order to solve the timeout problem when multiple packets are lost form the same window.

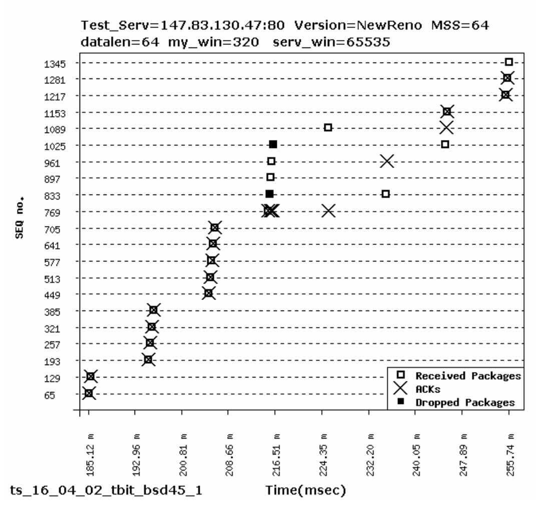

Figure 5. NewReno TCP

Figure 5 shows the way this implementation works:

1. The first 12 packets are acknowledged appropriately
2. Packets 13 and 16 are dropped
3. Segments 14, 15 and 17 generate duplicate ACKs duplicate for segment 12. Because of the 3 duplicate ACK Fast Retransmit and Fast Recovery algorithms are started.
4. Packet 13 is fast retransmitted
5. The received ACK confirms packets 13, 14, and 15, and it asks for segment 16. This is a new and distinct ACK, but an intermediate one (it acknowledges only some of the segments not all the segments that need to be acknowledged). Because of it Fast Retransmit algorithm does not stop, and is applied for segment 16
6. Segment 16 is fast retransmitted and it generates an ACK for segment 17, because packet 17 already in the receiver's buffer.
7. All the packets that needed to be acknowledged were acknowledged, so the Fast Retransmit algorithm stops.

Note that higher performances were obtained due to the little modification of Reno TCP. Although NewReno solves the timeout problem when multiple packets are lost form the same window, it can retransmit only one packet per Round Trip Time.

4.6. RenoPlus TCP

This implementation was found on Solaris 2.51.

In Figure 6 it can be observed the way this implementation works, and also that this implementation does not perform a correct Slow Start.

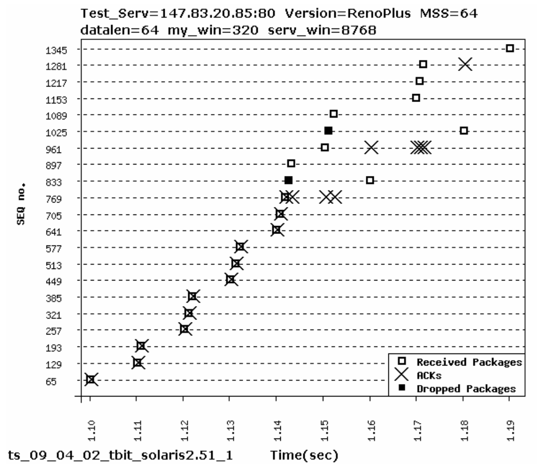

Figure 6. RenoPlus TCP

1. The first 12 packets are acknowledged appropriately.
2. Packets 13 and 16 are dropped.
3. Segments 14, 15 and 17 generate duplicate ACKs duplicate for segment 12. Because of the 3 duplicate ACK Fast Retransmit and Fast Recovery algorithms are started.
4. Packet 13 is fast retransmitted.
5. The received ACK confirms packets 13, 14, and 15, and it asks for segment 16. This new and distinct ACK is not considered as an intermediate ACK (like NewReno). Because of this ACK segments 18, 19 and 20 are transmitted.
6. These 3 segments will generate duplicate ACKs for segment 15. Fast Retransmit and Fast Recovery algorithms are started.
7. Packet 16 is fast retransmitted.

8. The received ACK (acknowledgement for segment 20) confirms packets 16, 17, 18, 19 and 20.

4.7. TCP Versions Used by Some of the Current Operating Systems

OS	TCP Implementation
FreeBSD 3.5.1	Reno
FreeBSD 4.2	Reno
FreeBSD 4.3	NewReno
FreeBSD 4.4	NewReno
FreeBSD 4.5	NewReno
Windows 98	TahoeNoFR
Windows 2000	TahoeNoFR
RedHat 7.2	NewReno

Table 1. TCP Versions

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), whilst the latest versions evolved towards NewReno (FreeBSD 4.3, 4.4, 4.5, RedHat 7.2). Surprisingly, Windows 98/2000 Professional are currently using TahoeNoFR (Tahoe without Fast Retransmit).

5. Conclusions and further work

The most reliable implementation is NewReno TCP.

1. It has no useless retransmissions and very low probability of retransmission timeouts.
2. Most web servers prefers NewReno.
3. To avoid the performance decreasing in case of a congested network, the selective ACK option should be enabled.

Several other tests, related to the new coming operating systems (Windows XP/2003, RedHat 8.0/9.0 etc.) are under progress. We want to extend also our study for the new congestion control algorithms used in the latest TCP implementations: Vegas, Peach, ATCP etc. Also we plan to study the dynamics of the congestion window in order to analyze TCP throughput for different implementations.

Acknowledgments

We would like to acknowledge the support from Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, Barcelona (Spain), led by Professor Jordi Domingo-Pascual. Special thanks go to Carles Kishimoto Bisbe and Roberto Borgione. The initial work was carried out within SOCRATES/ERASMUS 2001-2002 programme.

References

[1] K. Fall, and S. Floyd, "Simulation–Based Comparison of Tahoe, Reno and SACK TCP", *Computer Communications Review* ACM-SIGCOMM, Vol. 26, No. 3, July 1996
[2] J. Padhye, and S. Floyd, " On Inferring TCP Behavior", *Computer Communications Review* ACM-SIGCOMM, Vol. 31, August 2001
[3] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control", *RFC 2581*, IETF, April 1999
[4] Dobrota, V., *Digital Networks in Telecommunications. Volume III: OSI and TCP/IP*, Second Edition, Mediamira Science Publishers, Cluj-Napoca, 2003 (in Romanian)
[5] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgement Options", *RFC 2018*, IETF, October 1996