



**UNIVERSITÀ DEGLI STUDI  
DI TRENTO**

**Advanced Networking  
A.A. 2011-2012**

**Experimental analysis of the TCP Westwood+  
and TCP CUBIC congestion control algorithms**

**Davide Gerhard, Davide Girardi**

davide.gerhard@unitn.it  
davide.girardi-2@unitn.it

### **Abstract**

This report describes the experiments performed for the project assigned in the Advanced Networking course during the 2011-2012 academic year.

Our aim was to conduct some tests, in different scenarios, of the TCP Westwood+ congestion control algorithm. Then we will show its advantages and disadvantages and a comparison with the one described in the TCP CUBIC protocol.

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 TCP Westwood+</b>	<b>1</b>
<b>2 TCP CUBIC</b>	<b>2</b>
<b>3 Experimental Analysis</b>	<b>3</b>
3.1 Tools . . . . .	3
3.1.1 Iperf . . . . .	3
3.1.2 TCP probe . . . . .	3
3.1.3 DummyNet . . . . .	3
3.2 Infrastructure . . . . .	5
3.3 TCP parameters . . . . .	5
<b>4 Results</b>	<b>6</b>
4.1 Test 1 - Bandwidth: 10 Mbit/s, Delay: 0 ms, Packet loss: 0% . . . . .	6
4.2 Test 2 - Bandwidth: 10 Mbit/s, Delay: 500 ms, Packet loss: 0% . . . . .	7
4.3 Test 3 - Bandwidth: 10 Mbit/s, Delay: 0 ms, Packet loss: 5% . . . . .	8
<b>5 Comments</b>	<b>8</b>
<b>6 Future works</b>	<b>9</b>
6.1 Rate Adjustments . . . . .	9
6.2 Correlation . . . . .	10
<b>A Appendix: Problems</b>	<b>11</b>
A.1 Buffers, netem/tc . . . . .	11
A.2 TCP probe and 3.0+ linux kernels . . . . .	11

**Premise:** unless we are focusing on a specific version of TCP Westwood, we will use "TCP Westwood" meaning either TCP Westwood or TCP Westwood+.

## Introduction

In the base version of the TCP congestion control [1][2], they use an *additive increase/multiplicative decrease* (AIMD) sliding windows algorithm to avoid network congestion.

TCP uses a variable, called *congestion window* (*cwnd*), as senders estimation of how much data can be outstanding in the network without packets being lost. The *cwnd* is firstly set to one or two segments and then increased following either the *slow start* algorithm or the *congestion avoidance* phase. A variable called *slow start threshold* is used to determine whether the slow-start or congestion avoidance algorithm is used to control data transmission.

The two phases are described as follows:

1. **Slow-start phase** (when  $cwnd < ssthresh$ ) : the *cwnd* is increased exponentially, by one segment for each incoming acknowledgement (ACK).
2. **Congestion avoidance phase** (when  $cwnd \geq ssthresh$ ): the *cwnd* is increased linearly, at the rate of one segment per round-trip-time (RTT).

The sender becomes aware of a congestion when it receives 3 duplicate acknowledgements (DUPACK), which are generated by the receiver when it receives out-of-order segments, or when we are in the case of "heavy congestion" (e.g timeout expiration). In both cases, the sender retransmits a segment and sets *ssthresh* to half of the amount of currently outstanding data. The *cwnd* is then set to the value of *ssthresh* plus three segments in the first case or set equal to 1 in the other one. The retransmission due to incoming duplicate ACKs is called *fast retransmit*. After this phase, the TCP sender follows the fast recovery algorithm until all segments in the last window have been acknowledged. The number of outstanding segments is maintained by sending a new segment for each incoming ACK, if permitted by the congestion window value. The congestion window is temporarily increased for each incoming duplicate ACK to permit forward transmission of a segment. When the fast recovery phase is over its value is set to the number of segments at the beginning of the fast recovery.

Much research has been done about this topic and it has demonstrated how this algorithm could actually be enhanced, using other techniques or approaches, to fit all scenarios. The one on which we focused our project was TCP Westwood and in particular its improved version, Westwood+.

In most of the research papers which we have considered during our studies [4][5], the authors only performed a comparison between TCP Westwood and Reno. In these works, they performed various tests, trough either a link emulator or the real Internet, showing how Westwood is generally more efficient in networks with random rates of packet loss and how it is more fair in congested networks (e.g xDSL). Our purpose was updating these evaluations of TCP Westwood performing other tests in other scenarios and comparing TCP Westwood with the protocols which are mainly used nowadays.

In this report, we will describe TCP Westwood+ and TCP Cubic, the protocol used as its competitor in our tests, in the first two chapters. We will then show how we made our analysis in the 3rd chapter. All results and our reflection on them will continue in the following 4th and 5th chapters.

## 1 TCP Westwood+

TCP Westwood is a modification to TCP New Reno, which works only on the sender side, that aims to handle better those paths with high bandwidth-delay product (large pipes), with potential packet loss due to transmission or other errors (leaky pipes), and with dynamic load (dynamic pipes).

It uses an *additive increase/adaptive decrease* (AIAD) paradigm instead of the usual AIMD, which does not change the increasing window phase, but try to adaptively set the congestion window, when a congestion episode occurs, exploiting the stream of returning acknowledgement packets to estimate the available bandwidth for the TCP connection. It was developed in the 2001[4] and then improved in its current updated version Westwood+ [5].

The algorithm is the following:

- When 3 DUPACKs are received by the sender:

```
ssthresh = (B*RTTmin)/seg_size;
if ssthresh < 2 then ssthresh = 2;
cwnd = ssthresh;
```

- When a coarse timeout expires:

```
ssthresh = (B*RTTmin)/seg_size;
if ssthresh < 2 then ssthresh = 2;
cwnd = 1;
```

- When ACKs are successfully received:

cwnd increases following the Reno algorithm.

The estimated bandwidth is given as result of a filtering process, using the following discrete time-varying low-pass filter,

$$\hat{b}_k = \frac{2\tau - \Delta_k}{2\tau + \Delta_k} \hat{b}_{k-1} + \frac{\Delta_k}{2\tau + \Delta_k} (b_k + b_{k-1})$$

where  $\Delta_k$  is the inter-arrival time between the  $(k-1)_{th}$  and the  $k_{th}$  sample and  $\tau$  is the cut-off frequency of the filter.

In the original version of TCP Westwood, the sample  $b_k = d_k/t_k - t_{k-1}$  was computed every time an ACK is received. In [5], it is shown how the filter shown above overestimates the bandwidth in the presence of ACK compression provoked by reverse traffic. In TCP Westwood+, this problem is avoided computing  $b_k$  every RTT, instead of every time an ACK is received by the sender. In other words, we calculate  $b_k = D_k/\Delta_k$  as input for the filter, where  $D_k$  is the amount of data acknowledged during the last RTT ( $D_k$ ), to cut off the high frequency components due to the ACK compression.

## 2 TCP CUBIC

TCP CUBIC[3] is a different version of TCP with a congestion control algorithm optimized for high speed networks with high latency<sup>1</sup>. It is fundamentally an enhancement of TCP BIC which uses the following cubic window growth function instead of the linear one described in the TCP Reno/NewReno protocol.

The function is the following:

$$W_{cubic} = C(t - K)^3 + W_{max}$$

where  $C$  is a scaling factor,  $t$  is the elapsed time from the last window reduction,  $W_{max}$  is the window size just before the last window reduction and

$$K = \sqrt[3]{W_{max}\beta/C}$$

Since TCP CUBIC does not depend on the RTT but only on the last congestion event, it is more fair than the standard TCP. Indeed, if there are multiple standard TCP flows with different RTTs, those with very short RTTs will receive ACKs faster and therefore have their congestion windows grow faster than other flows with longer RTTs. This does not happen with TCP CUBIC because it is completely independent of ACKs reception and RTTs.

It is also the implementation of TCP currently used by default in Linux kernels 2.6.19 and above, and then used for our tests.

---

<sup>1</sup>Also called "Long Fat Networks" (LFN), they have a bandwidth-delay product which is significantly larger than  $10^5$  bits. See <http://tools.ietf.org/html/rfc1072>

## 3 Experimental Analysis

We will now describe how we performed our analysis, evaluating TCP Westwood+ alone and compared to TCP Cubic. We will firstly describe the various tools and scripts used during our tests and secondly the scheme of our infrastructure.

### 3.1 Tools

#### 3.1.1 Iperf

Iperf<sup>2</sup> is a spread network testing tool written in C++ used for network performance measurements. It creates TCP or UDP data streams and then measures the throughput of a network that is carrying them. It also permits to set a lot of parameters (e.g congestion control algorithm, static window size) in order to test a network under different configurations.

We used this tool during our tests executing it in the server mode on a computer with the following command:

```
iperf -s
```

In this way we obtain an Iperf server which is listening on the port 5001<sup>3</sup>. Therefore, depending on which test we were executing, we used the following general command to create the TCP flow at the sender:

```
iperf -c $SERVER_ADDRESS -i $TIME_INTERVAL -t $TEST_TIME -Z $CONGESTION -P $N_THREAD
```

#### 3.1.2 TCP probe

TCP probe<sup>4</sup> is a kernel module that saves the state of a TCP connection in response to incoming packets. It inserts a hook into the `tcp_recv` processing path using `kprobe` in order to capture the congestion window and sequence number, and other parameters such as, the slow start threshold. We used the following command to produce an output file with all the data captured by `tcpprobe`.

```
cat /proc/net/tcpprobe > $PATH/$DESTINATION_FILE
```

An example of the output file is then given by the following figure.

The tcp probe capture file will contain one line for each packet sent.



Figure 1: The output of the tcpprobe kernel module

#### 3.1.3 Dummynet

Dummynet[14] is a link emulator developed under FreeBSD, later imported into other BSD-derived operating systems, including Mac OS X, and currently also available on Linux, OpenWrt and Windows. It is a part of the operating system which allows to intercept network traffic and shape it, in order to simulate the behaviour of one or more network links with programmable features. It is composed by three parts:

<sup>2</sup>For more informations, look at <http://iperf.sourceforge.net/>

<sup>3</sup>The default port used by Iperf

<sup>4</sup><http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe>

- **dummynet:** the emulator itself.
- **ipfw:** the packet classifier.
- **/sbin/ipfw:** the user interface.

The first two components run in the kernel of the operating system, and communicate with the user interface through a control socket.

**The emulator - Dummynet** The emulator dummynet is used to create multiple "pipes". These objects model a network link with programmable bandwidth, delay and queue size.

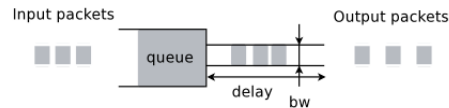


Figure 2: The basic components of a Dummynet pipe

There are also pipe configuration options which can be used to specify different queue management policies (e.g. RED), to model some MAC layer effects such as variable transmission times and link level overheads, and also to simulate very simple packet drop patterns. More advanced features allow connecting multiple queues to a packet scheduler running one of several scheduling algorithms, and then sending packets through a link with configurable features. Furthermore, Dummynet and the basic model shown above (fig. [2]) have been extended and they now provide a better emulation of wireless and other channels with peculiar MAC protocols, variable transmission rates or channel errors. In order to do this, the developers of Dummynet introduced two features: delay profiles and varying links.

Delay profiles use empirical profiles to allow the definition of additional MAC overheads (such as contention, framing, retransmissions). They compute a random time value, according to the distribution provided by the user, to extend the transmission time.

Varying links allow to model the variability of wireless channels (including loss rates and bandwidth) over time due to some factors such as external interference or mobility.

**The packet classifier - ipfw** Ipfw is the programmable packet classifier of Dummynet. Its purpose is intercepting packets in various points of the protocol stack, and decides how to handle them. The packets flow through the network stack, packet classifier and pipes is represented on Figure [3] below.

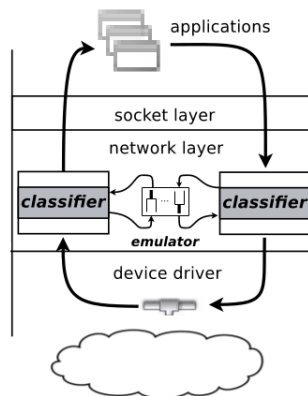


Figure 3: The flow of packets through network stack, packet classifier and pipes

Ipfw is configured by writing a set of numbered rules, each containing zero or more options used to match packets, and one action specifying what to do with matching packets. Matching options include addresses, ports, protocols,

protocol flags and various packets metadata, including the virtual server which the packet is associated to, in order to provide insulation between the different users. Traffic selection is performed by testing a packet against each of the rules, in numeric order, and performing the action associated to the first matching rule. For our purposes, the actions of interest are sending the packet to a pipe, which will in turn delaying or dropping the packet as appropriate, emulating the behaviour of the attached link. After the emulation, the non-dropped packets are sent back into the network stack for their regular processing.

### 3.2 Infrastructure

Our infrastructure was mainly composed by two laptops and one MikRotik RB750 router. The figure below shows a basic scheme of how these machines were linked together.

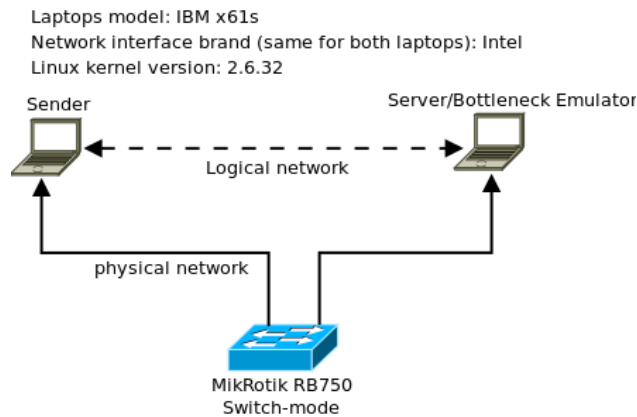


Figure 4: Main architecture used during our tests

The roles are explained as follows:

1. **Server:** it had to run iperf in the server mode as shown before.
2. **Bottleneck emulator:** it was used to simulate different links with different values of latency, maximum bandwidth, packet loss etc. between the sender and the server. We used the tool Dummynet shown before for this purpose.
3. **Sender:** the sender had to run iperf in the client mode, as shown before, and perform all tests. This includes using `tcpprobein` order to retrieve all the data to analyze.

The router MikRotik RB750<sup>5</sup> was used because it allows to set the value of the maximum bandwidth directly in the layer 2 of the stack ISO/OSI, avoiding the problems described in A.1, even though it would be possible to do the same with Dummynet.

Furthermore, as it can be noticed, we used a single laptop as server and bottleneck emulator.

### 3.3 TCP parameters

To ensure that the TCP stack had the same configuration on all machines, we used `sysctl`<sup>6</sup> to set some useful parameters for our evaluation. The most important ones which we changed are the following:

- **net.ipv4.tcp\_no\_metrics\_save=1:** by default, TCP saves various connection metrics in the route cache when the connection closes, so that connections established in the near future can use these to set initial conditions. Therefore, we have enabled this parameter to turn off the cache metrics on closing connections.
- **net.ipv4.tcp\_sack=1:** enable select acknowledgments (SACKS).

<sup>5</sup><http://routerboard.com/RB750>

<sup>6</sup><http://linux.die.net/man/8/sysctl>



- **net.ipv4.tcp\_dsack=1:** enable TCP to send "duplicate" SACKs.
- **net.ipv4.tcp\_window\_scaling=1:** enable window scaling as defined in RFC1323. TCP automatically adjusts the size of the socket receive window based on the amount of space used in the receive queue.

## 4 Results

We will now show the results<sup>7</sup> of our evaluation and, in particular, the graphics related to the variations of the congestion window and the slow start threshold.

### 4.1 Test 1 - Bandwidth: 10 Mbit/s, Delay: 0 ms, Packet loss: 0%

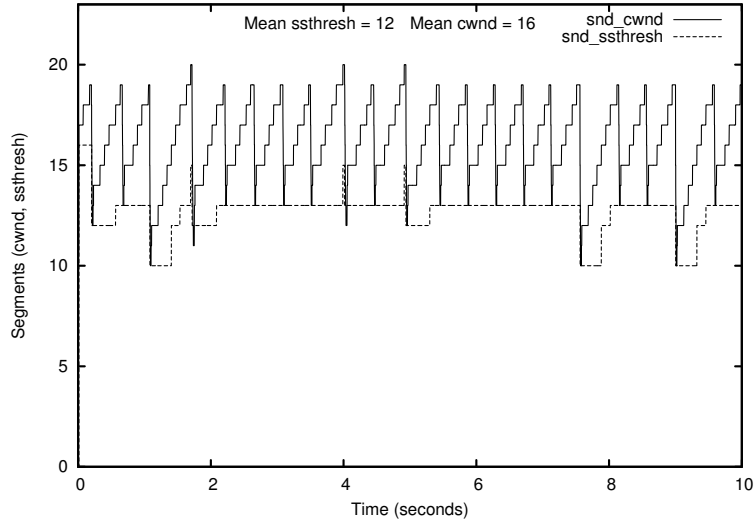


Figure 5: Congestion window and slow start threshold variations - Test 1 - TCP Cubic

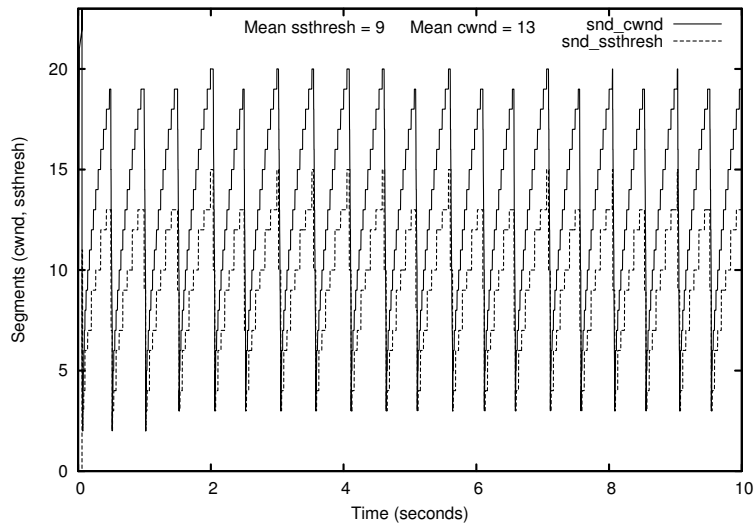


Figure 6: Congestion window and slow start threshold variations - Test 1 - TCP westwood

<sup>7</sup>Since we have produced a lot of data and results, we show only a summary of the most relevant and meaningful results

In both cases the average estimated throughput was about 9500 Kbits/s.

#### 4.2 Test 2 - Bandwidth: 10 Mbit/s, Delay: 500 ms, Packet loss: 0%

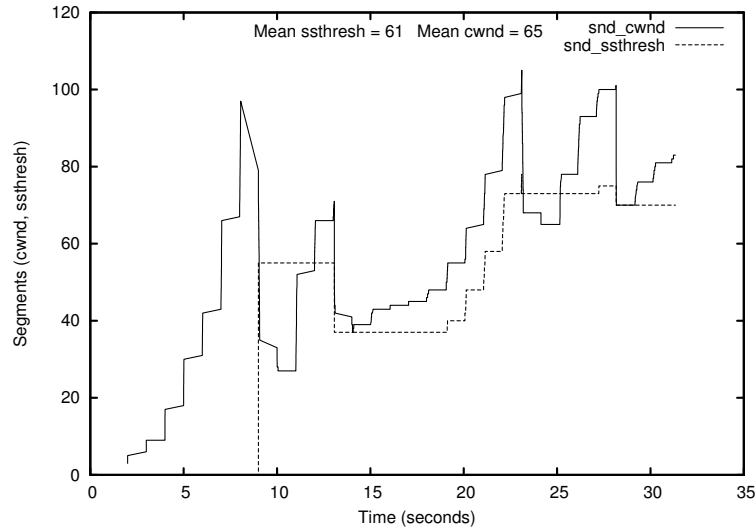


Figure 7: Congestion window and slow start threshold variations - Test 2 - TCP Cubic

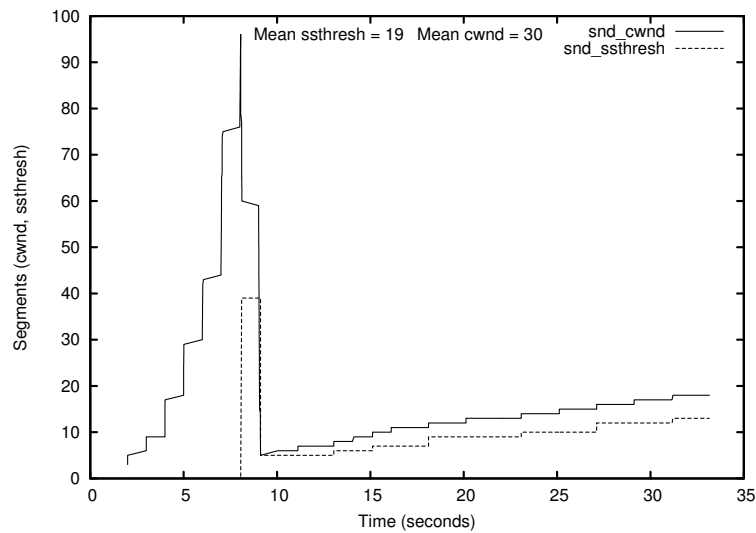


Figure 8: Congestion window and slow start threshold variations - Test 2 - TCP Westwood

According to the graphics, the average throughput detected using TCP Westwood was about the half of the one detected using TCP Cubic. The specific values were respectively 649 and 240 Kbit/s.

### 4.3 Test 3 - Bandwidth: 10 Mbit/s, Delay: 0 ms, Packet loss: 5%

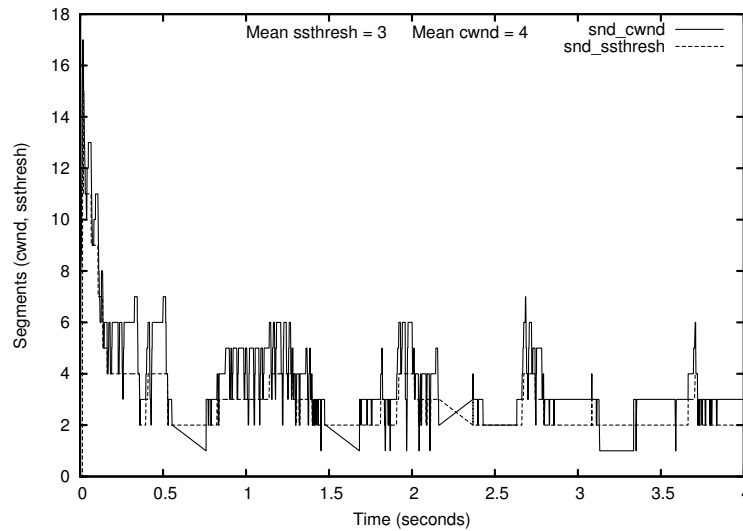


Figure 9: Congestion window and slow start threshold variations - Test 3 - TCP Cubic

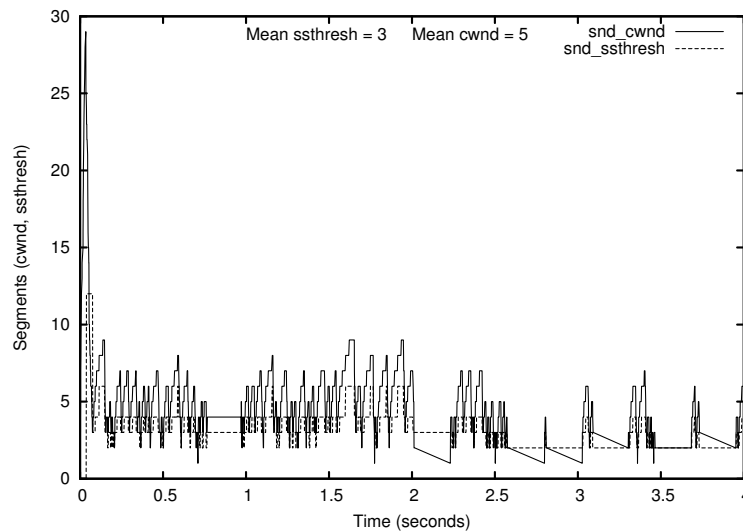


Figure 10: Congestion window and slow start threshold variations - Test 3 - TCP Westwood

Differently from the other cases, the average throughput of TCP Westwood was slightly better than TCP Cubic. The estimated values were respectively, 6156 Kbit/s for Westwood and 5637 Kbit/s for Cubic.

## 5 Comments

Reasoning about TCP behavior could be problematic sometimes. One of the reasons is that the results deriving from its evaluation does not always respect the theory behind it or its specifications because it is necessary to consider the different implementation, in particular in different operating systems. In addition, the protocol itself is very complex and it can be hard to model it and predict its behavior, such as the variations of the throughput and latency [15] and then the behavior of the congestion window, for its non-linearity. For these reasons and for others not strictly related

to the protocol (e.g benchmark tools, other layers, traffix shaper, hardware etc.), we encountered a broad variety of problems (see the appendix A) to retrieve relevant data that can be used to analyze TCP Cubic and Westwood during our experiments, and it was necessary to try several tools and configuration as well as perform several tests, before being able to state which version is more performant and why.

However, considering our results, in almost all cases the TCP Westwood's performances were not better than the TCP Cubic's ones, and it is possible to state that these results confirmed what it can be expected comparing the two protocols from a theoretical view.

The main issue behind the low performances of TCP Westwood could be related to the filtering process performed when it updates the value `cwnd`. As written in [5], considering a value of  $\tau = 0.5s$ , if the RTT is greater than  $\tau/4 = 0.125s$  we need to re-sample and create virtual samples in order to respect the Nyquist-Shannon sampling theorem and be conservative. This means that we solely need to have a RTT greater than  $125ms$ , and then a not negligible value of latency, to affect the sampling and then the bandwidth evaluation. In addition, the protocol relies on an additive increase strategy that does not allow TCP Westwood to grow fast when far from the maximum available bandwidth and respond effectively to the dynamic of the network with long delays, as showed in 4.2. On the other hand, TCP Cubic is completely not dependant to the RTT, and for this reason, it resulted more performant and stable even with huge delays as it can be expected.

The only case where we noticed a perceptible performance improvement is with a packet loss percentage greater than 0. In this case, TCP Westwood showed better performances than TCP Cubic, even though we had to increase the packet loss percentage up to 5%, that is not usual, to see some considerable changes.

Another aspect, not strictly related to the performances, that could be interesting to notice is the congestion window value's oscillation or more specifically, the difference between the highest and the lowest value in a single experiment. As it shown in the graphics (e.g figure 6), using TCP Westwood, the congestion window oscillates more around the average value than TCP Cubic.

## 6 Future works

Even though we retrieved enough results to state that TCP Westwood is less performant and how its dependancy to the Round Trip Time could affect his sampling process, there are a plenty of other scenarios and/or tests that a future and more complete evaluation of TCP Westwood+ should consider.

For example, it could be interesting to perform the same tests described in this report with multiple and independant connections but using different pc's/laptops rather than using the `-p` option offered by `iperf`, since we cannot retrieve multiple data related to different congestion windows with `tcpprobe` listening on a single port<sup>8</sup>. A solution for this problem could be using more servers listening on different ports. In this way, we can simply run `tcpprobe` with the option `port=0`, retrieving the data related to the traffic on every distinct port.

Another important addition to our work could be performing all the tests another time on different platform or operating systems (e.g FreeBSD, MacOSX, Windows) in order to obtain a more complete analysis and reason on how these factors might affect the behavior of TCP.

### 6.1 Rate Adjustments

The kernel developers have extended the TCP stack of Linux 3.2 to provide "Proportional Rate Reduction" (PRR) support. Introduced by a Google employee and described in an IETF draft<sup>9</sup>, this algorithm is designed to adapt transmission rates to the rates that can be processed by the recipient and by the routers along the way; especially after reducing the rate to prevent an imminent overload, the algorithm is designed to return to the maximum transfer rate faster than the previously used method, which is described in RFC 3517 ("A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP"). According to the developer's measurement results, the algorithm's faster transfer rates reduce HTTP response times by three to ten per cent.

It could be interesting to analyze how this improvement might influence a congestion control algorithm.

---

<sup>8</sup>`iperf` is able to work on a single port only

<sup>9</sup><http://tools.ietf.org/html/draft-ietf-tcpm-proportional-rate-reduction-00>

## 6.2 Correlation

Due to theoretical issues<sup>10</sup>, the generation of packet loss with correlation does not work properly in the current version of netem. For this reasons, some researchers at the University of Rome "Tor Vergata" developed a patch<sup>11</sup> for netem in order to solve this issue. To expand our evaluation, a future research on these topics could use this patch to perform more complete experiments.

## References

- [1] **V. Jacobson**, *Congestion avoidance and control*, In Proceedings of ACM SIG-COMM '88, pages 314-329, August 1988.
- [2] **M. Allman, V. Paxson, W. Stevens**, *TCP Congestion Control*, IETF, April 1999.
- [3] **Sangtae Ha, Injong Rhee, and Lisong Xu**, *CUBIC: a new TCP-friendly high-speed TCP variant*, SIGOPS Oper. Syst. Rev. 42, 5, July 2008.
- [4] **S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi and R. Wang**, *TCP westwood: Bandwidth estimation for enhanced transport over wireless links*, In Proceedings of the 7th annual international conference on Mobile computing and networking (MobiCom '01), 287-297, ACM, New York, NY, USA, 2001.
- [5] **S. Mascolo, L. A. Grieco, R. Ferorelli, P. Camarda, G. Piscitelli**, *Performance Evaluation of Westwood+ TCP Congestion Control*, Performance Evaluation, vol. 55, no. 12, pp. 931-111, January 2004.
- [6] **L. A. Grieco and S. Mascolo**, *Performance Evaluation and Comparison of Westwood+, New Reno and Vegas TCP Congestion Control*, ACM Computer Communication Review, vol. 34, no. 2, pp. 25-38, April, 2004.
- [7] **P. Sarolahti and A. Kuznetsov**, *Congestion Control in Linux TCP*, In Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, Chris G. Demetriou (Ed.). USENIX Association, Berkeley, CA, USA, 49-62. 2002.
- [8] **L. A. Grieco and S. Mascolo**, *Linux 2.4 Implementation of Westwood+ TCP with rate-halving: A Performance Evaluation over the Internet*, IEEE International Conference on Communications, ICC, 2004.
- [9] **W. Almesberger**, *Linux Network Traffic Control – Implementation Overview*, 5th Annual Linux Expo, p. 153-164, 1999.
- [10] **A. Keller**, *Manual tc Packet Filtering and netem*, ETH Zurich, July 20, 2006.
- [11] **S. Salsano, F. Salvatore**, *Performances of traffic control modules for QoS support in an IP router*, INFOCOM Department Report 002-004-1999.
- [12] **E. Altman, C. Barakat, S. Mascolo, N. Moller and J. Sun**, *Analysis of tcp westwood+ in high speed networks*.
- [13] *Linux Man pages and Linux Kernel Documentation*.
- [14] **M. Carbone, L. Rizzo**, *Dummnynet Revisited*, 2009.
- [15] **D. Zheng and G. Y. Lazarou** *A comprehensive stochastic model for tcp latency and throughput* 2008.

---

<sup>10</sup>A brief explanation was given at <http://lists.linux-foundation.org/pipermail/netem/2007-September/001156.html>

<sup>11</sup><http://netgroup.uniroma2.it/twiki/bin/view.cgi/Main/NetemCLG>

## A Appendix: Problems

### A.1 Buffers, netem/tc

Our first tests could have not be used to state anything about TCP Cubic and Westwood since the congestion window grew too slowly and we almost never end to a "real"<sup>12</sup> congestion. This was mainly due to a mismatch between the size of congestion window and the queue used in htb<sup>13</sup> and tbf<sup>14</sup>, the packet schedulers that we set while we were using netem<sup>15</sup> as bottleneck emulator. The figure [11] shows the result of one of these tests performed on TCP Westwood<sup>16</sup> as example of what could occur and how the congestion window grows in this case. As it can be noticed, there is no congestion episode and the sender sees no congestion in the link because the queue is never filled and then no packet is dropped. These tests were performed using either WanEm<sup>17</sup>, a linux distribution which allows to build a wide area network emulator based on netem and htb, or netem and htb/tbf alone.

We tried to use these tools on different kernel versions but the problem has not been solved yet and, the queue management and netem have not been improved. For example, with the version 3.3<sup>18</sup> of the linux kernel, setting the maximum bandwidth of the link at 10Mbit/s, the troughput was about 160kbit/s.

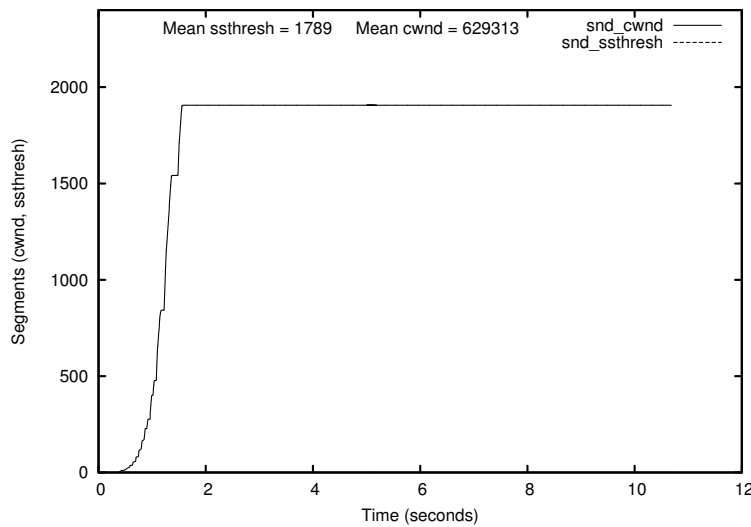


Figure 11: Congestion window and slow start threshold variations using htb as traffic shaper - TCP Westwood

### A.2 TCP probe and 3.0+ linux kernels

At the first steps of our work, we used the 3.0+ versions of the kernel to perform our tests. Unfortunately, `tcpprobe` has crashed everytime during our experiments. To solve this we tried an enhanced version of `tcpprobe` called TCP flow spy<sup>19</sup> developed by Soheil Hassas Yeganeh for collecting aggregated flow level information, but we ended up using the normal `tcpprobe`, since we figured it out that it works properly in the 2.6.32 kernels.

<sup>12</sup>Understanding if there is a real congestion or there is only a mismatch between the congestion window and the queue size is still an open problem.

<sup>13</sup><http://luxik.cdi.cz/~devik/qos/htb/>

<sup>14</sup><http://linux.die.net/man/8/tc-tbf>

<sup>15</sup><http://www.linuxfoundation.org/collaborate/workgroups/networking/netem>

<sup>16</sup>TCP Cubic showed the same behavior. We chose to not show its graphic since it was really similar to this one and therefore not meaningful.

<sup>17</sup><http://wanem.sourceforge.net/>

<sup>18</sup>We explained these issues in an e-mail that we sent to the netdev linux kernel mailing list. They suggested us to use this particular version.

<sup>19</sup>For more informations, look at <http://www.cs.toronto.edu/~soheil/>

This document is released under the Creative Commons license.  
Click on the image below for more details.

