Invited Paper:

TCP Performance - CUBIC, Vegas & Reno

Ing. Luis Marrone
lmarrone@linti.unlp.edu.ar
Lic. Andrés Barbieri
barbieri@cespi.unlp.edu.ar
Mg. Matías Robles
mrobles@info.unlp.edu.ar

LINTI - Facultad de Informática
Universidad Nacional de La Plata
La Plata, Buenos Aires, Argentina

Abstract—At present there are different TCP versions providing different performances. In this work the three of them: Reno, CUBIC and Vegas are considered. We simulate a WAN type network analyzing the throughput and performance of these TCP variants in order to discover which of them has a better performance.

Keywords—TCP, Congestion Control, Reno, CUBIC, Vegas.

1 Introduction

Although it is one of the first protocols developed for Internet its study and analysis has not stopped. The present features in the new data networks make further protocol developments and modifications. Since Jacobson and Karels presented, in 1988, its congestion control mechanism, called TCP Tahoe, many amendments have been done to TCP, resulting in different implementations of the same. The main changes point to the Congestion Control mechanism and it is the aim of this study to compare the performance in high speed networks.

In order to make efficient use of network bandwidth, TCP controls its flow rate using feedback provided by the network (the feedback is generated through message loss, delays or through network itself as in the case of Explicit Congestion Notification (ECN)), thereby preventing the collapse of Internet due to congestion. In addition, TCP has achieved considerable success in maximizing link efficiency through a balance in all existing flows. However, it has been found that TCP underutilizes the digital bandwidth large BDP(Bandwidth Delay Product) networks.

As speeds and lengths of networks increase, the performance of TCP has been challenged. These networks are characterized by having a BDP high, which implies a significant number of packets in transit necessary to maintain the bandwidth of the network completely used, that is, the congestion window size should be high in order to make efficient use of network resources.

This paper compares the performance of three variants of the TCP protocol. The selected Implementations are: TCP Reno [1] , CUBIC TCP Reno and TCP Vegas.TCP Reno as it represents the evolution of the original implementation; TCP CUBIC for being present in most Linux platforms and TCP Vegas by carrying out a proactive congestion control model, moving away from traditional approaches.

2 Congestion Control Algorithms Analyzed

2.1 TCP Reno

It emerged in 1990 and was implemented initially on a BSD Unix operating system. It is the successor of TCP Tahoe and basically proposes improvements thereon. Maintains all its main features (Slow Start, Congestion Avoidance and Fast Retransmit) but adds Fast Recovery. Both mechanisms were proposed by Van Jacobson.

TCP Reno, as TCP Tahoe, implements an Additive Increase Multiplicative Decrease (AIMD)algorithm. That is, increments or decrements the size of the window depending on whether the packet is acknowledged or lost (its time-out has expired or the source has received over a three repeated ACKs).

TCP Reno adjusts its window according to the phase in which it is:

Slow Start (SS):

In this phase, which is the initial , the window size increases exponentially. The session begins with a unitary window which is increased in one segment by each segment validated through an ACK. This phase also is initiated after a loss is detected by time-out. The limit of this phase is found when the window reaches a threshold called Slow

Start Threshold (ssthresh), that at the beginning of the connection is established to a high value.

Congestion Avoidance (CA):
This phase begins at the end of Slow Start or after a loss is detected by the reception of duplicate ACKs. TCP continues to increase its congestion window as 1/CWnd (CWnd = Congestion Window) each time it receives an ACK.

Fast Retransmit:
Packet loss is detected, as well as when the time-out expires and the package has not been confirmed (corresponding ACK received), for the reception of three duplicate ACKs (ie 4 equal ACK packets). In this case, it must be retransmitted only the package supposed to be lost, but not the rest of the packages, without waiting for the time-out to expire. Congestion Avoidance phase is performed again. This is TCP Tahoe.

Fast Recovery:
This phase goes on as follows:

- Slow Start Threshold (ssthresh) is set half the current value of the congestion window
- The congestion window, cwnd is set to ssthresh value plus 3 times the package size
- Each time the sender receives a duplicate ACK, cwnd increases in one packet and sends a new packet
- When the first non-duplicate ACK arrives, cwnd is set to ssthresh value

All these steps define the Fast Recovery mechanism.

2.2 TCP CUBIC

THIS is the default TCP algorithm in GNU / Linux. The differences with the other variants considered are scalability and mainly the fact that the window updates are independent of RTT, achieving a better distribution of bandwidth among all active sessions.
It is equivalent to Reno in stages Slow Start (in [4] there is a slight variation at this stage), Fast Retransmit and Fast Recovery. It differs from the standard TCP solely on the adjustment of the congestion window, replacing the linear growth function of a standard TCP for a cubic function in order to achieve the improvements mentioned above.
On loss, reason for setting the congestion window (cwnd), the value reached is recorded as W_{max} and the window is decreased by a factor $\beta = 0.2$.
From that time a new window is calculated as:

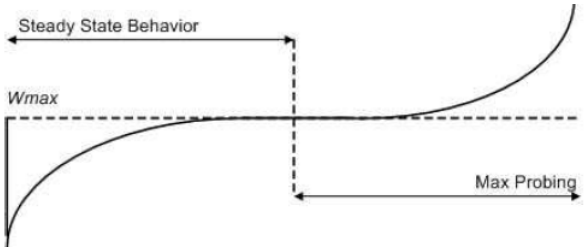
$$W(t) = C(t-K)^3 + W_{max} \quad (1)$$

The values of C and K are to be determined such that W_{max} be the turning point of the function $W(t)$, t is the time since the last window reduction .

K be the time required by the window to reaches the value W_{max} with no loss:

$K = \sqrt[3]{W_{max} * (\beta/C)}$
$C = 0,4$ usually.

It is shown in the following figure, extracted from [4], the behaviour of the cubic function defined for this implementation:



It is then set the value of $W(t)$ and the value of the window that would have come with standard TCP, $W_{TCP}(t)$

$$W_{TCP}(t) = W_{max} * (1-\beta) + 3 * \frac{\beta}{2-\beta} * \frac{t}{RTT}$$

The window grows as follow for each ACK received

- If $W(t) < W_{TCP}(t)$, then CUBIC TCP operates in support or friendly mode, setting the window according to the value of $W_{TCP}(t)$
- If $W(t) < W_{max}$, then it is in the concave region of the curve or stable mode [5]. The window is adjusted according to $W(t)$.
- If $W(t) > W_{max}$, then it is in the convex region of the curve or exploratory mode. The window is adjusted according to $W(t)$.

2.3 TCP Vegas

THE novelty of Vegas is its proactive character because determines if there is an incipient congestion by observing the difference between the actual throughput (that calculated from RTT measurement) and the maximum (which is calculated at the start of the session). TCP Vegas adjusts the delivery rate of the source node (i.e. its congestion window) in order to maintain a small number of packets in the buffers of the routers. It was developed at the University of Arizona by Larry L. and Lorenzo Brakmo Peterson in 1994 [2], [3]. Today it is implemented on various platforms, but due to the competition with other algorithms such as Reno or CUBIC, is not widely used.
While conceptually it contrasts with Reno it runs the same mechanisms, with differences in their implementations. Obviously, the congestion window is updated according the phase in which it is currently running.
Let's see the differences:

- The RTT, key parameter to trigger the various mechanisms, is measured with a finer granularity, away from Reno 500 msec. RTT value is directly determined by the difference in time of

sending a packet and receiving the corresponding ACK for each packet sent.
- This more accurate RTT value is to be used to determine the retransmission in two stages:
  - When receiving a duplicate ACK, if the RTT is larger than RTO.
  - When receiving a non duplicate ACK that is not the first or second after a retransmission, if the RTT is larger than the RTO.
- In the TCP Congestion Avoidance phase Vegas estimates the amount of data that can exist in the buffers of the switches and / or routers and according to that, decide whether to increase or decrease its congestion window by one packet. This estimation is performed for each of the measured RTT as follows:

$$CWnd = \begin{cases} CWnd + 1, & \textbf{if Diff} < \alpha \\ CWnd - 1, & \textbf{if Diff} > \beta \\ CWnd, & \textbf{otherwise} \end{cases}$$

Where:
$Diff = (Expected - Actual)$
$Expected = CWnd/BaseRTT$. Maximum Throughput expected.
$Actual = CWnd/RTT$. Measured Throughput.
$BaseRTT$. Minimum RTT.
The thresholds, α and β, correspond approximately to have little and much backlog of packets in the network, respectively. α must be less than β.
Obviously, the further the current throughput is from the expected more congestion exists in the network, which means that the sender should reduce its sending rate. The threshold β triggers this decrease of the congestion window. On the other hand, if the actual throughput gets too close to the expected , the connection may not be using the available bandwidth. The threshold α triggers this increase.
- The mechanism of Slow Start exponentially increases CWND every two RTT, (Reno does in all RTT). At the other RTT, the window does not change. This phase ends when a growth is found in the buffer queues. This occurs when the congestion window reaches a value for which $Diff = (Expected - Actual)$ exceeds a threshold δ . When this happens, TCP Vegas enters the Congestion Avoidance phase.

3 Lab Tests

The test environment, Figure 1, simulates the characteristics of a WAN network , in which the central link fulfills that role setting its digital bandwidth to 100Mbps and with several different delays. About the end links between routers and stations, we did not make any changes being the same with a speed of 1Gbps and usual delay of a LAN.

In much of the literature tests are conducted on simulators like NS-2 [9], in this case the approach seeks to make them on a real environment. In previous tests it has been observed that simulators often

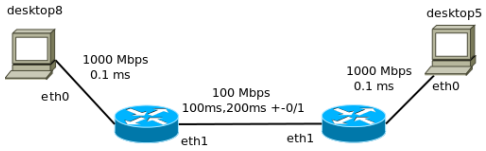


Fig. 1: *Test Environment*

lose characteristics that become visible in real environments, giving different results. The tests were conducted on physical machines with enough memory and CPU, AMD with 2 cores and 3GB of RAM running OS GNU / Linux Red Hat Enterprise Server 6 with kernel version 2.6.32-71-7-1.el6 of 64bits for all devices, both routers and workstations. To make changes to link parameters we used Linux tools Tc (Traffic Control) and Linux Netem (Network Emulation) [10]. The configurations for testing the link between routers are:

- Bandwidth 100Mbps, LAN delay (no changes), test results were taken only as references, these were not analyzed.
- Bandwidth 100Mbps, delay $d = 100ms$ (50ms + 50ms) without jitter , $j = 0ms$.
- Bandwidth 100Mbps, delay $d = 100ms$ (50ms + 50ms) with jitter: $j = +/- 1ms$ on each end-router.
- Bandwidth 100Mbps, delay $d = 200ms$ (100ms + 100ms) without jitter, $j = 0ms$.
- Bandwidth 100Mbps, delay $d = 200ms$ (100ms + 100ms) with jitter: $j = +/- 1ms$ on each end-router.

The jitter (j) has a uniform variation of +/-1ms. For each set of tests there were not made competitive analysis with another traffic.

Each test was run five or more times with variations in the TCP congestion control algorithms mentioned in the introduction.

The tool used to generate traffic at the end nodes was `iperf(1)` [6]. To capture traffic behavior and congestion control algorithm we took messages from the physical interfaces from the sending host with tcpdump/Wireshark [7]. The kernel module `tcp_probe` (TCP cwnd snooper) [8] allowed to follow the behaviour of the congestion window, cwnd, and SSTH threshold, for different TCP sessions. For the value of CWND and SSTH, in every case, all values were monitored over time and not just the changes. This fact is clarified since the tool can also be used to monitor changes only.

The test environment with respect to the kernel parameters configured on computers for the first tests is the default and configured to not cache results of previous connections. Linux TCP normally remembers some parameters of the last links in a flow cache. This configuration is recommended for benchmarks (tests).

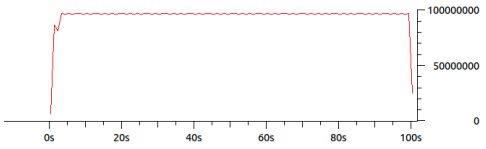


Fig. 2: *Reno throughput in bps, for 100Mbps d=100ms, j=0ms*

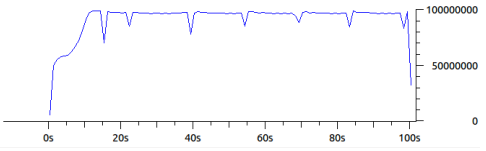


Fig. 3: *CUBIC throughput in bps for 100Mbps, d=100ms, j=0ms*

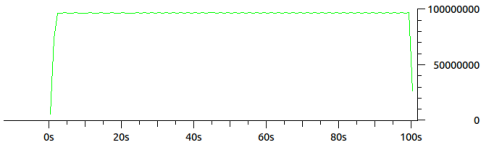


Fig. 4: *Vegas throughput in bps for 100Mbps, d=100ms, j=0ms*

4 Results

4.1 TCP Tests: 100Mbps, 100ms delay without jitter

The tests were conducted for 100 seconds and were repeated several times for each algorithm. Not any extension in time was required because the results were as expected. Each variant of congestion control algorithm of TCP obtained almost the highest possible performance in test time.

Figures 2, 3 and 4 show obtained throughput.

Figures 5, 6 and 7 show the evolution of the congestion window and SSTH for each one.

During the initial phases the curves show that CUBIC is a bit slower, but this time in which works somewhat below the rest, in the total test is almost negligible.

4.2 TCP Tests: 100Mbps, 100ms delay with jitter +/-1ms

We started with a jitter of $j = +/-1ms$, testing for 100 seconds and later, for a better analysis a little more time. In the case of Reno in 30% of the cases it

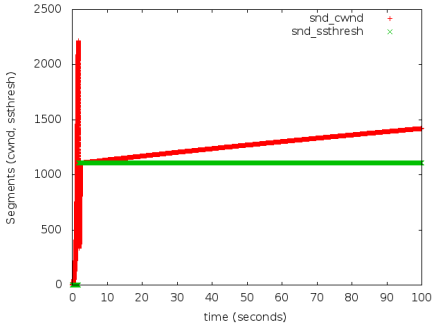


Fig. 5: *Reno CWND/SSTH evolution for 100Mbps, d=100ms, j=0ms*

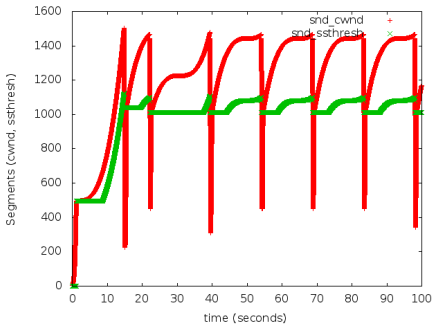


Fig. 6: *CUBIC CWND/SSTH evolution for 100Mbps, d=100ms, j=0ms*

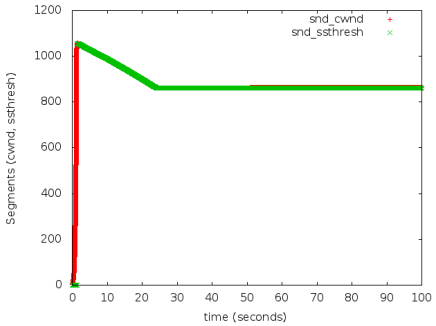


Fig. 7: *Vegas CWND/SSTH evolution for 100Mbps, d=100ms, j=0ms*

gets a throughput over the average (70Mbps, approx), in the remaining 70% because of selecting low value of SSTH, it goes immediately into phase CA and gets the maximum throughput slowly; approximately in 300 seconds. Figure 8 shows the evolution of CWND/SSTH with a low performance in the short-term.

Figure 11 shows in pulse lines the low performance of Reno in most of the cases.

Figure 10 shows the evolution of the window and threshold in the few cases where the algorithm starts well and quickly reaches the ceiling. A feature seen in the Linux implementation is that it rises the threshold according to the window during phase CA.

If the tests run longer it shows that the algorithm is stable in the long term, behaviour that is seen in Figure 9 showing the evolution of the parameters that control performance.

In contrast, with CUBIC the results were more even reaching better results in the short term. Figures 11 and 12 shows the test results with this algorithm, both performance and CWND / SSTH evolution. The performance is compared with that obtained with Reno.

Vegas behaviour was completely outside the expected since the same curves as in the case of Reno were obtained. Figure 13 shows CWND evolution. It seems that this implementation of Vegas behaves like Reno in the presence of jitter.

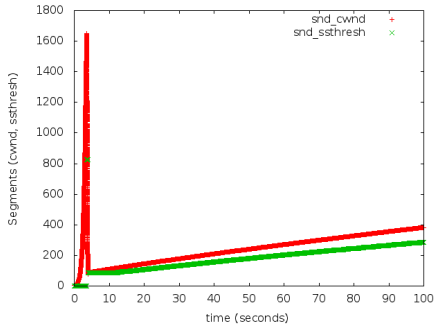


Fig. 8: *Reno CWND/SSTH evolution for 100Mbps, d=100ms, j=+/-1ms in 100s*

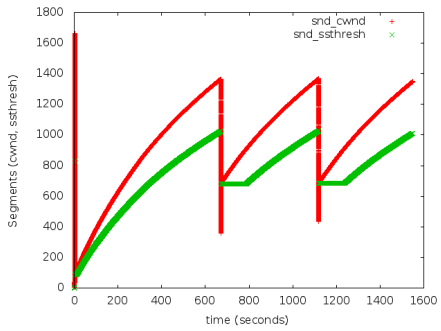


Fig. 9: *Reno CWND/SSTH evolution for 100Mbps, d=100ms, j=+/-1ms with extended time*

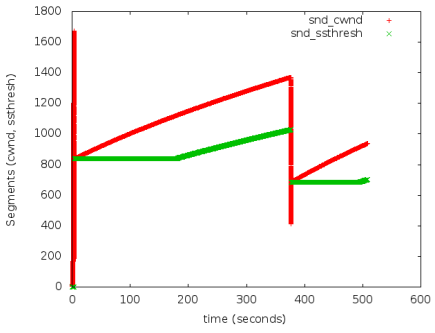


Fig. 10: *Reno CWND/SSTH evolution for 100Mbps, d=100ms, j=+/-1ms good start*

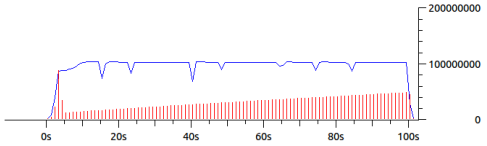


Fig. 11: *CUBIC throughput(line) vs. Reno(pulse) for 100Mbps, d=100ms, j=+/-1ms*

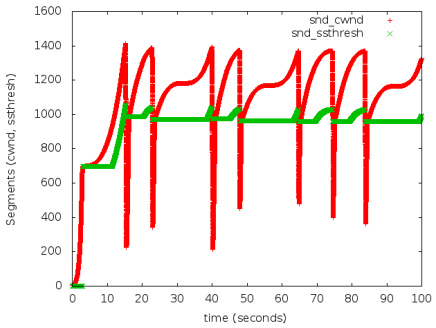


Fig. 12: *CUBIC CWND/SSTH evolution for 100Mbps, d=100ms, j=+/-1ms*

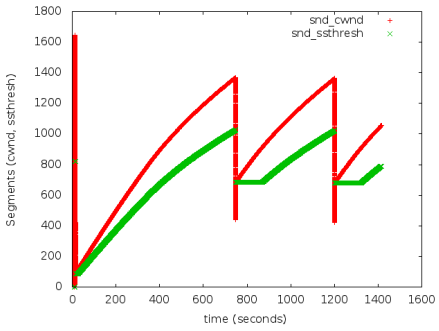


Fig. 13: *Vegas CWND/SSTH evolution for 100Mbps, d=100ms, j=+/-1ms, extended execution*

4.3 TCP Tests : 100Mbps, 200ms delay without jitter

In this case, the tests have a double BDP (Bandwidth Delay Product), causing the usual buffer's parameters, directly associated with the maximum CWND are not adequate and by having a large RTT, there is a slower growth.

Reno, in tests of 100 seconds, reaches an average of 65Mbps without optimization. The performance is similar to CUBIC and Vegas in the same environment. While CUBIC is much more aggressive in growth at the beginning, showing peaks at the end all reach similar performance.

Performing the following optimization, the results are different.

```
sysctl net.ipv4.tcp_rmem=''4098 87380
46777216''
sysctl net.core.rmem_max=46777216
sysctl net.ipv4.tcp_wmem=''4098 87380
46777216''
sysctl net.core.wmem_max=46777216
```

With these modifications, Reno achieves better results. The second phase of CA does not begin from below but does so from the middle of SSTH, that in the short-term with tests lasting 100ms, gives an average of 80Mbps. This can be seen in Figures 14 and 15 showing the performance and evolution of the parameters for congestion control running in an optimized way.

In CUBIC, for its part, the peaks do not return to

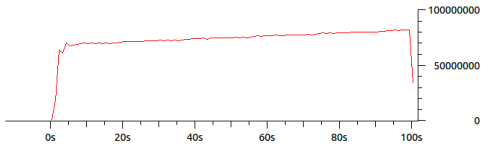


Fig. 14: *Reno throughput in bps for 100Mbps, d=200ms, j=0ms, optimized*

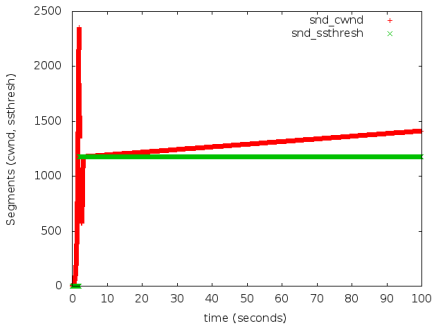


Fig. 15: *Reno CWND/SSTH evolution for 100Mbps, d=200ms, j=0ms, optimized*

zero, therefore, a better final performance is obtained, which can be seen in Figures 16 and 17.

The results obtained with Vegas optimized are quite similar to those of Reno once again. The graphs are not provided.

Extending the time of testing the final digital bandwidths averages are somewhat better and the performance of each algorithm is observed correctly. CUBIC has similar behaviour in both the short and the long term because the cycles are much shorter. In the long term, Reno works as expected, which can be seen in Figures 18 and 19. The graph of the throughput is generated using a different option in wireshark that shortens the amount of data considering that the expected behaviour was observed. In the evolution of

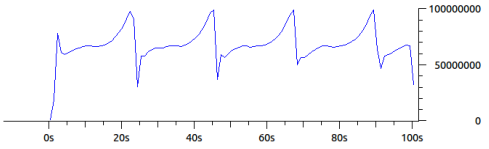


Fig. 16: *CUBIC throughput in bps for 100Mbps, d=200ms, j=0ms, optimized*

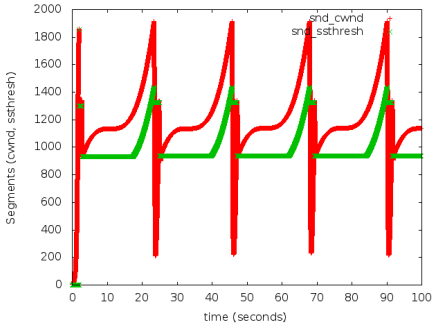


Fig. 17: *CUBIC CWND/SSTH evolution for 100Mbps, d=200ms, j=0ms, optimized*

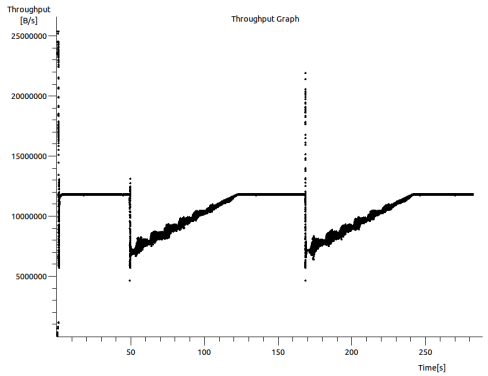


Fig. 18: *Reno throughput in bps for 100Mbps, d=200ms, j=0ms, optimized, extended time*

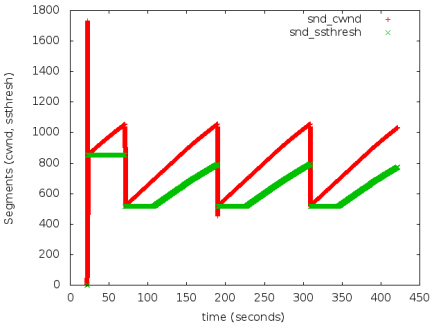


Fig. 19: *Reno CWND/SSTH evolution for 100Mbps, d=200ms, j=0ms, optimized, extended time*

the congestion window, time t = 0 is shifted because the capture began moments before traffic generation.

Vegas, meanwhile, manages to stabilize and work according to its features, which can be seen in Figures 20 and 21. The graphs are generated in the same way as for Reno on the same test bench.

Although in some tests the growth was much slower, which can be seen in Figures 22 and 23, Vegas ends getting somewhat better results in an environment with a fixed delay and without competition with other flows.

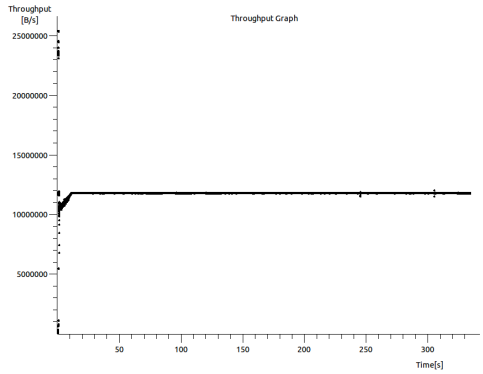


Fig. 20: *Vegas throughput in bps for 100Mbps, d=200ms, j=0ms, optimized, extended time*

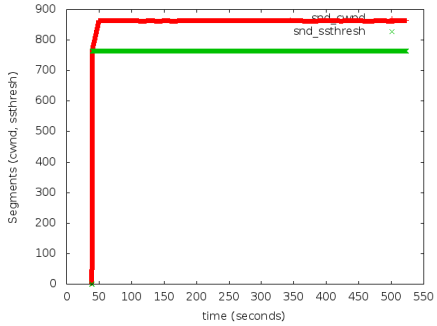


Fig. 21: *Vegas CWND/SSTH evolution for 100Mbps, d=200ms, j=0ms, optimized, extended time*

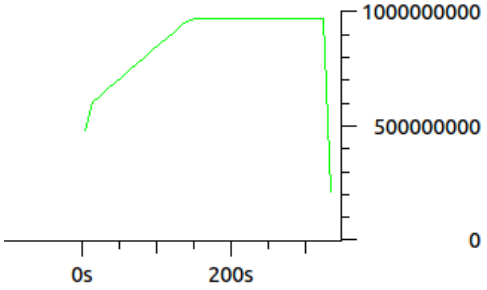


Fig. 22: *Vegas throughput in bps for 100Mbps, d=200ms, j=0ms, optimized, extended time, another case*

4.3.1 TCP Tests: 100Mbps, 200ms delay with jitter +/-1ms

In these tests the behaviour of the algorithms is similar to that obtained with 100ms delay and jitter +/-1. To study the behaviour of the Reno and Vegas the test should be extended for much longer, of course, because the growth is slower. Once Vegas reaches the roof the behaviour is like Reno doing slow cycles of ups and abrupt downs , giving a bad performance. We find that for these algorithms that the throughput in the short term is much worse, reaching the maximum at approx 1200s. CUBIC, because of its aggressiveness gets better results in the early, short-cycles. Vegas, now with jitter behaves like Reno. Figure 24 is the evolution of Reno and Figure 25 that of Vegas. Regarding CUBIC in smaller duration tests can be seen Figure 26. To compare performances, in Figures 27 and 28 Reno (pulses) is compared with CUBIC (line), in 300s and the first 100s. You can

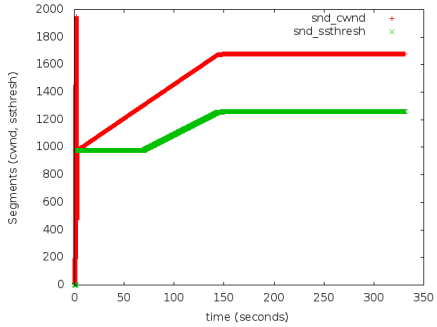


Fig. 23: *Vegas CWND/SSTH evolution for 100Mbps, d=200ms, j=0ms, optimized, extended time, another case*

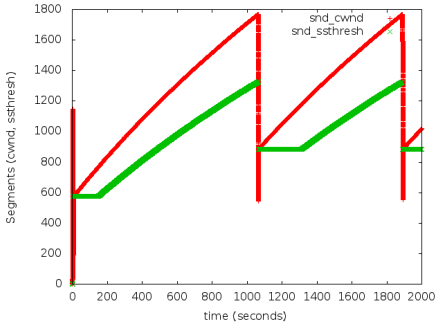


Fig. 24: *Reno CWND/SSTH evolution for 100Mbps, d=200ms, j=+/-1ms in 2000s*

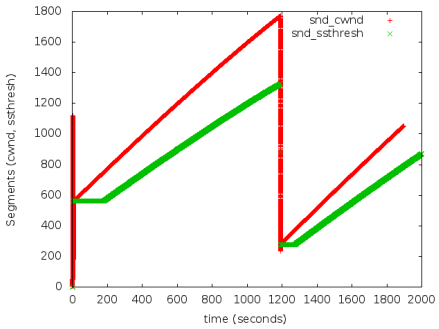


Fig. 25: *Vegas CWND/SSTH evolution for 100Mbps, d=200ms, j=+/-1ms in 2000s*

notice the slow growth of Reno, which according to tests, is similar to Vegas quite different from CUBIC on the fast attack . In a long-term test, 2000s, the results are more even and the two algorithms, Reno and Vegas, perform not so badly compared to CUBIC.

5 Conclusions

For the tests performed with different WAN parameters, CUBIC algorithm seems best suited to the different conditions tested. While sometimes Vegas was the one that obtained the best performance for short sessions, it often takes a long time to reach the maximum and is not efficient. It is also observed that Vegas sometimes does not behave like it is expected but works as Reno on GNU / Linux. Reno and Vegas in a great number of tests gave a low yield in the short term because they quickly enter the

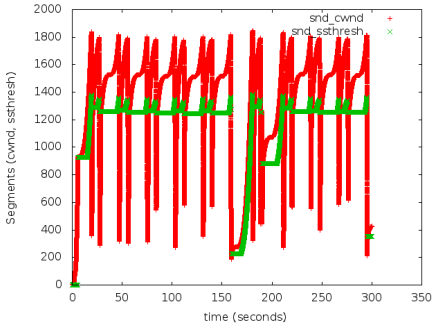


Fig. 26: *CUBIC CWND/SSTH evolution for 100Mbps, d=200ms, j=+/-1ms in 300s*

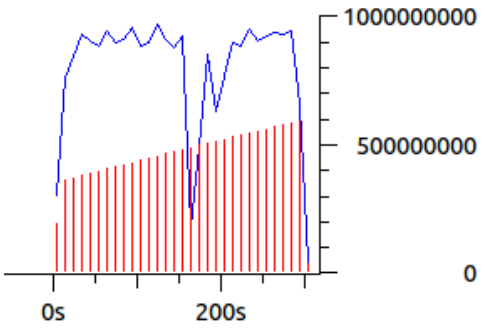


Fig. 27: *Throughput: CUBIC(line) vs. Reno(Pulses) for 100Mbps, d=200ms, j=+/-1ms in 300s*

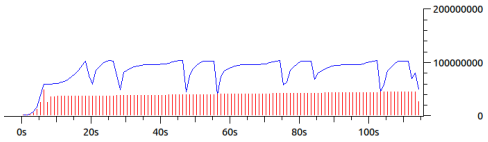


Fig. 28: *Throughput: CUBIC(line) vs. Reno(pulse) for 100Mbps, d=200ms, j=+/-1ms in 100s, initial phase*

stage of CA (Congestion Avoidance) having reached a low SSTH. For links with larger values of BDP in the case of Reno and Vegas it was necessary to accomplish the optimizations and modifications in the window configuration for them to achieve acceptable performance. Regarding CUBIC, optimizing kernel improved the performance although without it the algorithm even works much better.

6 Future Work

As counterpart to WAN test it could be simulated LAN environments where the features are quite different and optimize the parameters that come into play. These tests should obtain a comparison of the algorithms in stand-alone environments. Also it would be interesting to set up tests environments with the algorithms competing between themselves. Finally, Vegas unexpected performance with jitter in GNU/Linux platforms should be investigated.

References

[1] *S. Floyd and T. Henderson, The New Reno Modification to TCP's Fast Recovery Algorithm*, 1999, http://www.ietf.org/rfc/rfc2582.txt

[2] *L.S. Brakmo and S.W. O'Malley and L.L. Peterson, TCP Vegas:New Techniques for Congestion Detection and Avoidance*, 1994. Department of Computer Science, University of Arizona.

[3] *L.S. Brakmo and S.W. O'Malley and L.L. Peterson, TCP Vegas:New Techniques for Congestion Detection and Avoidance*, 1994. Proceedings of the SIGCOMM '94 Symposium, pages 24-35.

[4] *D.J. Leith and R.N. Shorten and G. McCullagh, Experimental evaluation of Cubic-TCP*, 2007. Fifth International Workshop on Protocols for Fast, Long Distance networks,(PFLDnet2007).

[5] *L. Goytezuela, Cubic TCP Analysis Using a Deterministic Model of Loss*, 2009. Facultad de Ingeniería Electrónica, Universidad Mayor de San Marcos, Lima, Perú, vol 24, pages 33-42.

[6] *http://sourceforge.net/projects/iperf/, Iperf - The TCP/UDP Bandwidth Measurement Tool*, 2013.

[7] *Lamping, Ulf and Sharpe, Richard and Warnicke, Ed, The Wireshark User's Guide*, 2008, http://www.wireshark.org/download/docs/user-guide-a4.pdf

[8] *http://www.linuxfoundation.org, tcp Probe : TCP cwnd snooper*, 2013.

[9] *The Network Simulator NS-2*, 2013, http://www.isi.edu/nsnam/ns/.

[10] *Netem: Network Emulation*, 2013, http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.