ISSN:2249-5789
Shaziya Islam et al , International Journal of Computer Science & Communication Networks,Vol 3(3),199-206

ANALYSING THE PERFORMANCE OF TCP IN LINUX KERNEL

Shaziya islam(M.Tech (CS)) , Er.Deepak Bhatnagar (Head of computer science deptt.) CSE jhansi

ABSTRACT: This document reports the project performance study of the TCP/IP stack for the Linux kernel". We analyzed the packet processing time traversing each layer of the Linux kernel 2.6.9 TC stack (socket, TCP and Ethernet) and the influence of multi-threading and different packet sizes. The design is based on the idea of inserting probing points via hooks in the kernel code and export timing data to a user-space application. The results demonstrate a number of key concepts. In TCP networking, such as layering, user-system interface, connection versus datagram modes, processing routines and their overhead in different layers. Some preliminary results reveal the system has its bottlenecks in different situations.

1. INTRODUCTION

The objective of this project is to investigate the path TCP/IP packets take on a single node through each layer of the TCP/IP stack. Analyzed aspects are the queueing behavior in socket layer, UDP versus TCP processing, TCP fragmentation and IP and Ethernet layer processing. The performance metric we selected is the packet processing time, since it is the most noticeable behavior for a packet from an end-to-end point of view. Papadopoulos and Parulkar [Pap93] presented a performance study concerning UNIX IPC including the underlying TCP/IP protocol on SunOS 4.0.3 running on Sun Workstations with 10Mbps Ethernet.

With regards to TCP/IP protocol evaluation, they studied performance of queuing in layers, buffer requirements, protocol control mechanisms and the interaction with the operating system. In order to do so, several probes were inserted in the SunOS TCP/IP source code at different locations. However, they mainly focused on using IPC to trigger TCP/IP processing, with little consideration on UDP and the influence of other applications running directly above TCP. Moreover, the packet sizes used in their tests were rather small. Welte [Wel00] also described his method on how to determine the packet cross-layer processing times inside a router and a receiving host running Linux Kernel 2.4 in x86 architecture. In contrast, in this project we not only look at per-packet processing in these TCP/IP layers, but also study the difference between the uses of connection-less transport protocol UDP and connection-oriented TCP, as well as other scenarios such as larger packet size (which requires TCP fragmentation and reassembly), multiple TCP connections and multiple threads over a single TCP connection on a link. The operating system we used is the recent Linux kernel 2.6.9; all machines used are Via Eden 553MHz PCs with 256MB RAM and RTL-8139 100Mbps Ethernet NICs.

The Data Network traffic comprises of the packets flowing from a source to a destination. This traffic and the network behaviour are extremely unpredictable in nature. Before a message needs to be sent , it is broken down into small packets & transported that way from the source to the destination. The protocols responsible for this transport over TCP/IP networks are User Datagram Protocol (UDP) & Transmission Control protocol (TCP). On an average, about 90% of the Internet traffic use TCP which is a reliable, stream oriented protocol. TCP relies exclusively on the positive acknowledgement & retransmission when an acknowledgement does not arrive within given time out period.

TCP provide flow control by using sliding window mechanism. With the help of sliding window link control, the maximum possible throughput on TCP Connection may be determined. The throughput depends on the window size, propagation delay & data rate. To increase TCP throughput, the congestion has to be alleviated. It was the first steps towards to move bulk data quickly over high speed data network and when data arrives on a big pipe (A FAST LAN) and gets sent out a smaller pipe then bottleneck is occured which is known as Congestion.. The congestion can lengthen the response time, reduced availability and throughput. When network is overloaded with data then we say network is congested and to prevent that the sender overloads the network is called congestion avoidance.

To tackle the congestion at network layer, dynamic routing may be used but these routing deals only with the unbalanced load.

Ultimately congestion is controlled by limiting the total amount of data entering the Internet to the amount that the Internet can carry.In TCP flow & congestion control, the receiver will only acknowledge frames & expand the window to the extent that it has buffer space available. The rate at which a TCP object can send data is calculated by the rate of incoming ACKs to previous segments with new credit. In TCP, the rate of ACK arrival is calculated by the bottleneck the round trip way between source & destination, & that bottleneck may be either the destination or the Internet.

The data transfer of TCP starts from a *slow start*,in which TCP tries to increase its sending rate exponentially, until it encounters the first loss. It then switches to another stage,

called *congestion avoidance*, in which TCP employs the *Additive Increase ,Multiplicative decrease* mechanism to slowly adapt to the available bandwidth. On further congestion, the TCP goes into the *Fast Recovery &Fast Retransmission* stages.

In this scenario, when TCP do not receive an acknowledgment for a packet after some timeout period, it assumes that this packet is lost. & then retransmits that packet and doubles its retransmission timeout value(RTO) detecting packet loss. This process continues until the packet is successfully transmitted & acknowledged. TCP tries to clear congestion by cutting its sending rate in half. Out of the many UNIX like kernels, Linux is a matured product and has a significant share in worldwide server population dealing with TCP traffic under all kinds of traffic scenarios. Hence, the TCP implementation in Linux has been tuned enough to meet the requirements of heavy duty

2.PROBLEM DESCRIPTION

There are various kernels in which Linux is one of the matured product on the basis of security features in worldwide server population dealing with TCP traffic under all kinds of traffic scenarios.

Hence the TCPimplementation in Linux kernel has been tuned enough to meet the requirements of heavy duty applications depending on it. This study is a effort to analyze the strengths and weaknesses of tcp implementation in Linux kernel Basically there are at least two alternative approaches to study the per-packet,per-layer processing times. The first approach is to use APIs for different layer interfaces to filter and collect the performance data in the program level. Another approach is to follow the methods proposed in [Pap93], inserting certain probing points, placing timestamps and recompiling the kernel.Netfilter is one method belonging to the first group. It is a packet filtering engine of kernel version 2.4 and 2.6.

The netfilter code is executed via hooks that can be enabled during compilation at specific code locations that were made for packet filtering, but are not suitable for our project which targets at per-packet,per-layer and per-routine (if necessary) performance analysis. Furthermore,enabling netfilter creates unnecessary packet processing overhead which has an impact on the measurement results. These disadvantages can be avoided by following the second approach. Our investigation further shows that the probing point hooks can be selected with a focus on performance measurement only without introducing additional processing overhead, and our code can be flexibly extended for other analysis.

Before describing the detailed design, we identify the following project tasks which are critical to evaluate a TCP/IP packet processing across networking
layers and required functional routines:

• Probing points at each layer-to-layer transition (Christian Dickmann, Henning Peters and Bernd Schl¨or).

• Accurate and efficient measurement of processing with a minimal additional overhead (Jan Demter and Henning Peters), which implies the performance data that may be exported to the user-space, avoiding unnecessary data processing in the kernel space.

• Reliable packet identification at all probing points (Christian Dickmann and Henning Peters).

• Easy and automatic evaluation and illustration of the collected data (Christian Dickmann and Sebastian Vogelsang).

• Ability to generate packets in different modes (single process, single thread vs. multiple threads vs. multiple processes; data rate and size, etc.) (Niklas Steinleitner).

In addition, Ubbo Veentjer worked on a development environment usingUser-Mode Linux [UML]. Bernd Schl¨or was the project and development leader.Michael Sobol contributed to the investigation of previous approaches, general discussions and project presentation.

3.PROPOSED SOLUTION

Capture TCP Traffic on the network using capturing tools,analyse and compare practical results with theory of TCP traffic control

3.1. TCP CONGESTION CONTROL ALGORITHMS

Standard TCP

Standard TCP uses congestion control algorithms described in RFC2581 [1]. The algorithms used are:

Slow Start
Congestion Avoidance
Fast Retransmit
Fast Recovery

A TCP connection is always using one of these four algorithms throughout the life of the connection.

Slow start

TCP maintains a guess at the current reasonable window size, called the slow start threshold (or ssthresh). Whenever TCP starts sending after being idle (or timing out) it would like to send with a window of size ssthresh. It turns out to be a bad idea to send the entire window in a burst, which might force a nearby router to buffer the whole window; far better to spread the window over a round-trip time, so that they are stored in transit on the links. TCP accomplishes this using this algorithm, called “slow start”.

- Initialize the window size, CWND, to one segment
- Whenever an ACK that acknowledges new data arrives (a “positive” ACK). Increase CWND by one segment
- If the resulting CWND is less than ssthresh, stay in slow-start. Otherwise, enter congestion avoidance mode

This doubles CWND every round-trip time, so that TCP opens its window tcpssthresh in time proportional to log ssthresh instead of all at once. A typical initial ssthresh,

used when a TCP connection is first created, is 64Kilobytes. ssthresh is adjusted after segment loss as described below. The second event is receiving the duplicate ACKs for same data. Upon receiving three duplicate ACKs, the connection uses fast retransmit algorithm. The last event that can occur during slow start is a timeout. If a timeout occurs, congestion avoidance algorithm is used to adjust congestion window and slow start threshold

Congestion Avoidance

The data transfer of TCP starts from a *slow start*,in which TCP tries to increase its sending rate exponentially, until it encounters the first loss. It then switches to another stage, called *congestion avoidance*, in which TCP employs the *Additive Increase, Multiplicative decrease* mechanism to slowly adapt to the available bandwidth. On further congestion, the TCP goes into the *Fast Recovery &Fast Retransmission* stages . In this scenario, when TCP do not receive an acknowledgment for a packet after some timeout period, it assumes that this packet is lost. & then retransmits that packet and doubles its retransmission timeout value(RTO) detecting packet loss. This process continues until the packet is successfully transmitted & acknowledged. TCP tries to clear congestion by cutting its sending rate in half.

Out of the many UNIX like kernels, Linux is a matured product and has a significant share in worldwide server population dealing with TCP traffic under all kinds of traffic scenarios. Hence, the TCP implementation in Linux has been tuned enough to meet the requirements of heavy duty applications depending on it.

Additive Increment

After receiving an ACK for new data, congestion window is increment by (MSS)2/Cwnd, where MSS is maximum segment size, this formula is known as additive increment. The goal of additive increment is to open congestion window by a maximum of one MSS per RTT. Additive increment can be described by using the equation

(1): Cwnd = Cwnd + a*MSS2/Cwnd

where the value of a is a constant, a = 1.

Multiplicative Decrement

Multiplicative decrement occurs after a congestion event, such as a lost packet or a timeout. After a congestion event occurs, the slow start threshold is set to half current congestion window. This update to slow start threshold follows equation

(2): ssthresh = (1 – b)*CWND

CWND is equal to amount of data that has been sent but not yet ACKed and b is a constant, b = 0.5. The congestion window is adjusted accordingly. After a timeout occurs, congestion window is set to one MSS and slow start algorithm is reuse. The fast retransmit and fast

4. Technical approach

4. 1. Packet Generator

After a review of several packet generators, we found that none of them met all of our requirements. Therefore, we developed our own packet generator (including receiver). It consists of two parts. The packet generator is implemented in ANSI C.

• **Server**

The server accepts all incoming TCP connection requests and UDP data server-application part and a client-application part. The packet generator requests on a specify port. Then it opens a socket for each session and listen for packets on this socket. If packets come in over this socket, the application receives all packets and drops them.

• **Client**

We implemented a multithreaded client which can initiate both TCP and UDP sessions to a given ¡IP address, port number, pair if the server application is running at this port. The client provides the use of different numbers of threads and sockets, so we can cause the following cases:

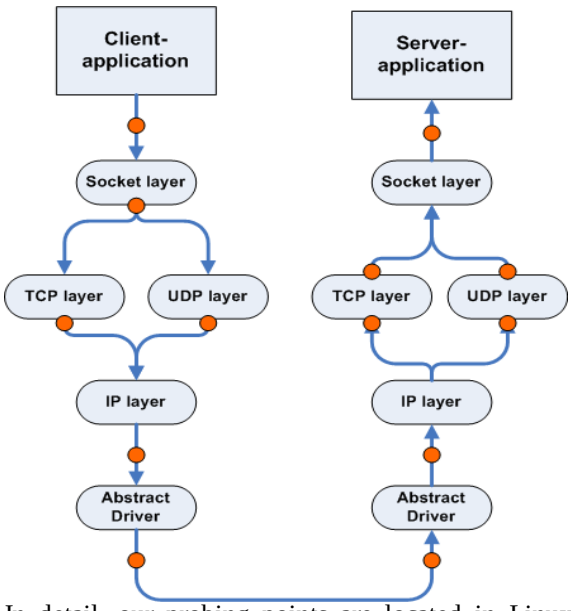
– each thread uses only one socket

– one thread uses many sockets

– one socket is shared by many threads

The packet generator basically does the following jobs: sends a userspecified number of packets, fills all these packets with our own 16 bytes frames payload and records threads and sockets numbers in each of these frames. For more realistic emulation of real traffic, the client can be further developed to wait a randomized idle time within a range between two consequent packets.

4.2. Probing points



In detail, our probing points are located in Linux kernel network stack at

Following positions:

• Net/socket.c:1554 TPS SOCK

• Net/ipv4/af inet.c:661 TPS SOCK TRANS

• Net/ipv4/tcp output.c:374 TPS TCP IP

• net/ipv4/udp.c:646 TPS UDP IP

• Net/ipv4/ip output.c:240 TPS IP NET

• Net/ipv4/ip output.c:224 TPS NET

• net/core/dev.c:1843, 1849 TPR NET
• Net/ipv4/ip input.c:293 TPR NET IP
• Net/ipv4/tcp ipv4.c:1744 TPR IP TCP
• net/ipv4/udp.c:1137 TPR IP UDP
• Net/ipv4/tcp input.c:4362, 4398, 4449 TPR TCP SOCK
• net/ipv4/udp.c:1161 TPR UDP SOCK
• Net/socket.c:1601 TPR SOCK

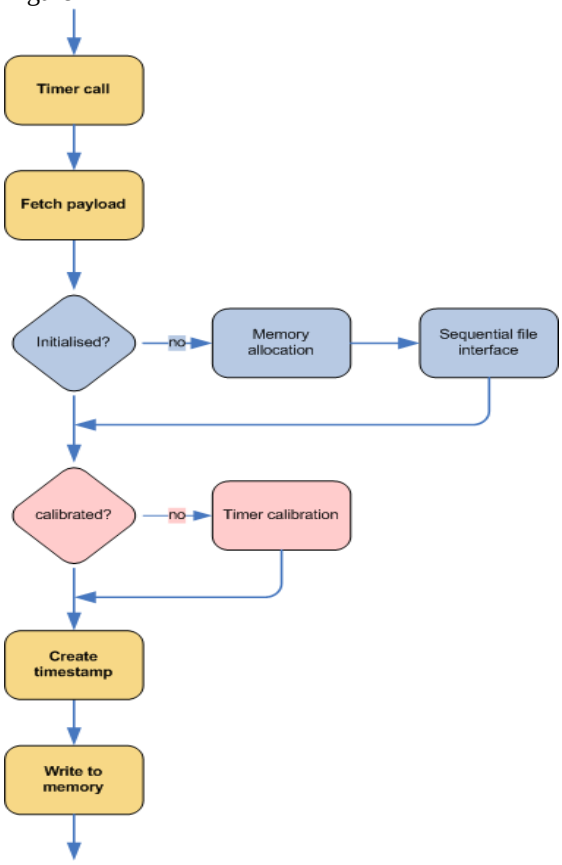
TPS_ SOCK_ TRANS is the same probing point for both, UDP and TCP protocol. At probing point TPR TCP SOCK there are three timestamps implemented,
at probing point TPS NET, we use a function pointer to get the timestamp when the device driver is called.

4.3. Timestamping code

Figure shows a sequence diagram of the timestamping code. After the first call, memory (space for 500000 measurement values) and user-space interface (seq file) is initialized implicitly and calibration is started. The calibration is finished after a certain amount of measured timestamps and is used for elimination of errors in measurement.

The timestamp functions are defined as follows:
• tp_ timer data: direct access to payload
• tp _timer seq: access to sk buf structure
• tp _timer: timestamp code (do gettimeofday)

Figure



5. Errors in measurement

Every measurement has an influence on the results because of its nature that the measurement code itself is taking up CPU cycles. Usually we invoke one, in some cases two (inline) functions at each measurement point. This depends on the direct availability of payload access. In some cases we have to prepare our accessible data (i.e. accessing payload at receiving socket layer when the network stack has not identified any protocol header, yet). This preparation is done in linear time and it only consists of pointer arithmetic (tp_timer_seq).In tests we found out that our measurement resolution (1 microsecond) is not good enough for this task. On the other hand, reading the data and counting the received fragments is much more expensive. That is the reason why we only focused on this single function (tp timer data). To make it more accurate, we implemented a so-called “timer calibration”.The time that is needed for our measurement code depends on the hardware and software configuration of our testbed. We were seeking an approach that would automatically calibrate our measurement depending on real-world testing data. In our current version, the first 100 function invocations are tracked and the time needed to run the measurement code is taken for a 5% trimmed-mean calculation. Finally, an informative message is sent to the kernel logger so that one knows when to start with real measurement. When calibration is finished,the calibration code is not called anymore. Only the calibration mean value is subtracted from each result.

Here is a shortened example of the kernel logger calibration message:

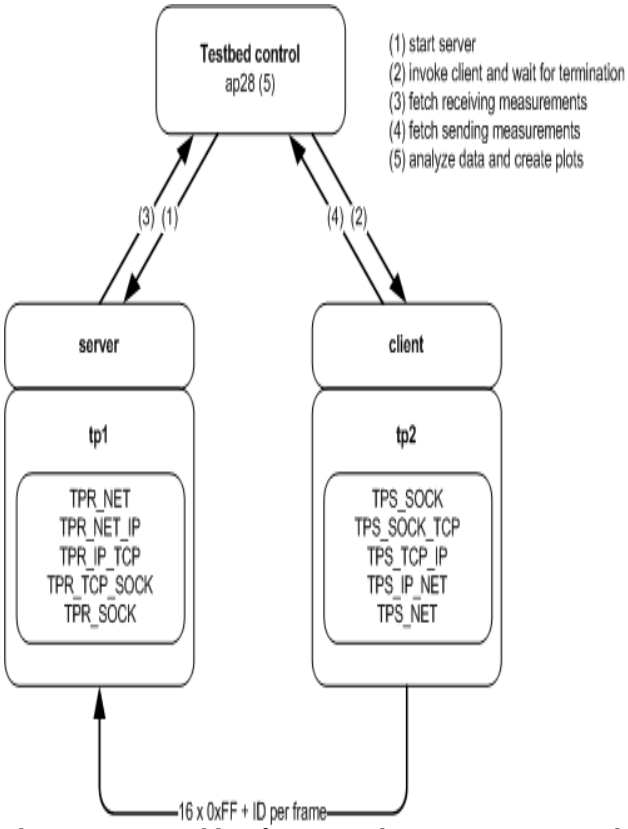
cal list: 3 3 3 3 3 3 3 ... 8 8 8 8 8 9 9 9 12 13 13 22 12738
calibration finished. runtime (5% trimmed mean): 4 µs

cal list is an array of measurement values. Each number represents the runtime of one measurement call in microseconds

.The reason for the long runtime of the outlier (last cal list value) is the implicit initialization (tp timer init) we are doing on first invocation. By using a trimmed mean, this value is ignored amongst others.Compared to the small size of the implementation we should keep in mind that the gained accuracy is probably not significant. In our development environment, the measurement code itself takes about 4-6 microseconds per invocation. Unfortunately, these small values have demonstrated their impact, sometimes comparable to the usual noise of our results. One thing we learned from this is that we do not have to pay too much attention to our implementation related measurement errors.

5.1. Automated measurement

We needed to run the packet generator for many times with different parameters to get the results we desired. So we implemented a script for automated measurement and evaluation of the results. For this task PHP was our first choice. The script uses two computers, one running the server, the other running the packet-generator. SSH is used to run these tools on the specified machines and SCP is used to download the measured timestamps. Figure 4.2 shows the operation of the automated measurement done in our test bed.

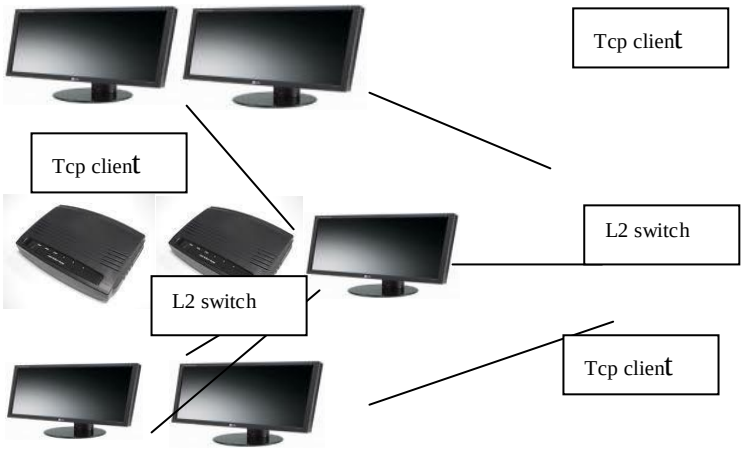


The script is capable of running the same scenario with a range of used packet sizes and threads. For each combination of packet size and thread count the measured timestamps are downloaded and processed into a PHP array by the script. This array is then used to generate plots and average times. The collected average times are used to create plots for packet size or thread ranges.We use gnuplot to generate the plots and as an additional feature even a HTML page is created.

6. EXPERIMENTAL SETUP

We construct an asymmetric dumbbell sort of topology where two L2 switches are located at the bottleneck between two end points. The end points consist of a set of HP Linux Systems running custom client and server applications dedicated to high-speed TCP variant flows and background traffic. Background traffic is generated by using various web based applications. We use Linux hosts as communication end points communicating over 100Mbps link with MTU of 1500 bytes. The RTT of each background traffic is random. The socket buffer size of some client machines is fixed to default 64KB while high-speed TCP machines are configured to have a very large buffer so that the transmission rates of high-speed flows are only limited by the congestion control algorithm. Two Layer 2 switches are deployed with four high-speed TCP machines which are tuned to generate or forward high traffic. Each TCP variant has been used individually to analyse the performance aspects.

The custom Java Client and Server programs are used to generate and receive high traffic end to end. The Server TCP suffers from high traffic ingress and has to take corrective and further preventive action evident from the analysis. As the Linux 2.6 TCP has pluggable modules now, we can inject and eject appropriate modules dynamically too. The analysis has been done for TCP Westwood and TCP CUBIC individually and the results have been compared. The packet sniffer tool „Wireshark" has been used to capture the live TCP traffic and generate logs/reports. A comparative study has also been done for competing TCP modules.



7. ANALYSING DIFFERENT SCENARIOS OF TCP TRAFFIC

TCP starts from a stage, called slow start, in which TCP tries to increase its sending rate exponentially, until it encounters the first loss. It then switches to another stage, called congestion-avoidance, in which TCP employs the Additive Increase, Multiplicative Decrease mechanism to slowly adapt to the available bandwidth. On further congestion , the TCP goes into the Fast Recovery and Fast Retransmission stages, When TCP doesn't receive an acknowledgement for a packet after some time out period, it assumes that this packet is lost, and then retransmits that packet and doubles its retransmission time out value(RTO) for detecting packet loss. TCP tries to clear congestion by cutting its sending rate in half.

7.1. Westwood:-TCP Westwood develop two basic concepts: the end to end estimation of the available bandwidth and the use of such estimate to set the slow start threshold and the congestion window. In TCP Westwood ,the sender continuously computes the connection Bandwidth Estimate (BWE) Which is defined as the share of bottleneck bandwidth used by the connection

BWE is equal to the rate at which data is received to the receiver or rate of acknowledgement received

After 3 duplicate packet received(packet loss indication) the sender resets the congestion window and the slow start threshold based on BWE ***cwin=BWE*RTT***. RTT is also required to compute the window that support the estimated rate BWE Initially congestion window increments during slow start and congestion avoidance remain the same as in Reno, that is they are exponential and linear, respectively. A packet loss is indicated by :
(a) the reception of 3 duplicate acknowledgements or
(b) a expiry of Round Trip Time. *TCPWestwood set cwin and ssthresh as follows*
If (3 DUPACKs are received) ssthresh=(BWE *RTTmin)/seg_size if(cwin>ssthresh) /* congestion avoidance*/ cwin=ssthresh Endif Endif *In* case a packet loss is initiated by a time out expiration. cwin and ssthresh are set as follows:
if(Coarse timeout expires)
cwin=1
ssthresh=(BWE * RTTmin)/seg_size;
If (sstresh<2)
ssthresh=2
endif
endif

Analysis Of Westwood:- Fig : RTT Graph for TCP Westwood(TCPW)

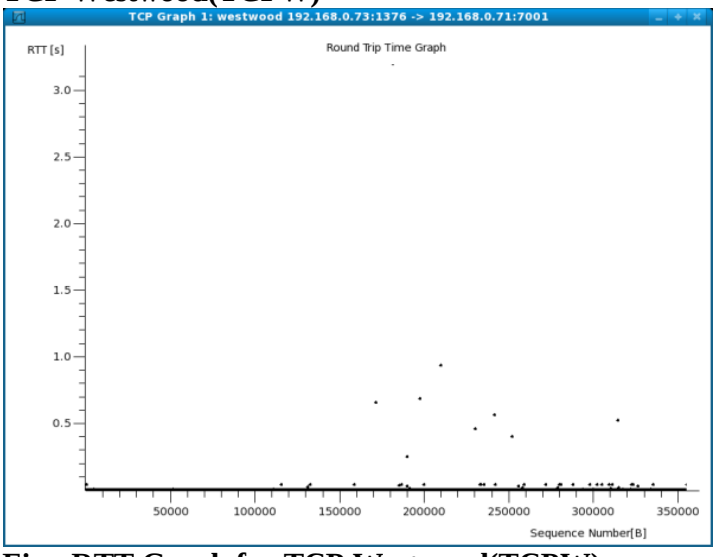


Fig : RTT Graph for TCP Westwood(TCPW)

In the analysis of TCPW Throughput, we use Linux hosts as communication end points communicating over 100Mbps link with MTU of 1500 bytes. The RTT of each background traffic is random. The socket buffer size of some client machines is fixed to default 64KB. The random bursts of data attack on the socket receive buffers and TCP enters into congestion avoidance mode. The graph shows that there is an effort to attain a steady state throughput due to Westwood algorithm. There are apparent traces showing the congestion window to become ¾ of the current congestion window. The throughput touches the peak of **312.5 kbps** with an immediate corrective congestion window size afterwards. The nature of the output traffic is almost steady state as there are no sharp increases like Reno and periodic wedges like vegas and it far better matches the objectives of the congestion control algorithm. The nature is less self similar in the trace received by us till the congestion collapse was finally achieved. TCP Westwood Congestion Control is based on –

(1) congestion window (*cwnd*) (2) slow start threshold (*ssthresh*) (3) round trip time of the connection (*RTT*) (4) minimum round trip time measured by the sender (*RTTmin*). The stream of returning ACK packets infers an estimate of connection available bandwidth (*BWE*). At the point of congestion, When 3 DUPACKs are received by the sender : ssthresh = (BWE * RTTmin) / MSS;

(2) if (ssthresh<2) ssthresh=2;

cwnd = ssthresh; Here, RTTmin = 0.001 sec (observed) MSS = 512 B BWE = 200 B (on an average)
ssthresh = (200*0.001)/512 < 2 Hence ssthresh = 2 cwnd = 2 This way TCPW in Linux handled a congestion scenario.
On the downside, the throughput does not seem to follow sharp 'additive increase' in congestion window like Reno and not even like pure AIMD+Vegas. This can be less useful in case of high speed networks. The congestion window has become more sensitive to congestion but less aggressive in following 'additive increase'. A peak is observed but a sharp fall follows due to decrease effect. The graph clearly shows that the congestion happened after 180 sec and the congestion window cautiously controlled it, finally showing a 'multiplicative decrease'. This performance is far better than all the previously analysed TCP versions. As the link bandwidth increases, the measurement-based nature of TCPW allows it to track the bandwidth variations of the bottleneck and to linearly build-up its performance. AIMD also improves its performance in same situation but in a less considerable way.

***7.2.* TCP CUBIC*:* -** As name suggests it implement cubic function. CUBIC is designed to simplify and enhance the window control of BIC .

***W*cubic =C(t-K)3 + *W*max,**

C = scaling factor t = elapsed time from the last window reduction.
Wmax = window size just before the last window reduction.
K= Wmaxß/C
where ß is a constant multiplicative decrease factor applied to window reduction at the time of loss event (i.e.the window reduces to ßWmax at the time of the last reduction). In this the window grows very fast upon a window reduction but as it gets closer to Wmax ,it slows down the growth. Around Wmax , the window increment becomes zero. Above that, CUBIC starts probing for more bandwidth

in which the window grows slowly initially, accelerating its growth as it moves away from Wmax .
K= Wmaxß/C where ß is a constant multiplicative decrease factor applied to window reduction at the time of loss event (i.e.the window reduces to ßWmax at the time of the last reduction) In this the window grows very fast upon a window reduction but as it gets closer to Wmax ,it slows down the growth. Around Wmax , the window increment becomes zero. Above that, CUBIC starts probing for more bandwidth in which the window grows slowly initially, accelerating its growth as it moves away from Wmax .

Analysis of TCP CUBIC:- In this section, we compare the performance of Linux CUBIC TCP w.r.t. AIMD. In the analysis of CUBIC,we use Linux hosts as communication end points communicating over 100Mbps link with MTU of 1500 bytes. The RTT of each background traffic is random. The socket buffer size of some client machines is fixed to default 64KB.We evaluate CUBIC-TCP and AIMD for the bandwidth utilization and RTT.

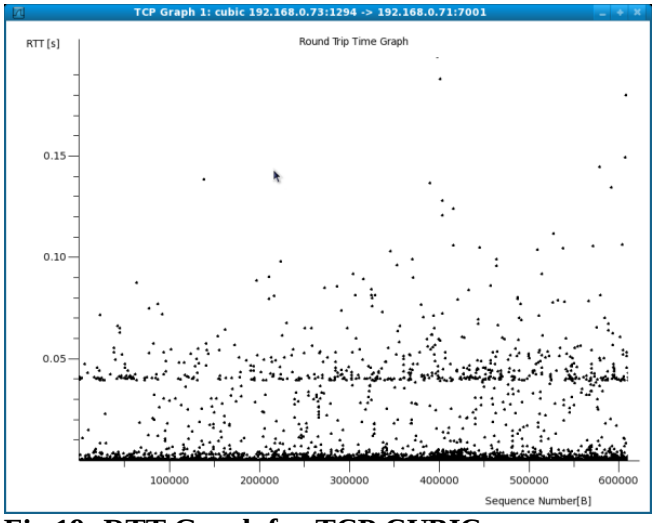


Fig 19: RTT Graph for TCP CUBIC

As per the graph shown ,the minimum RTT was around 0.001 sec and maximum RTT was around 0.18. The congestion

The congestion window of CUBIC is determined by $W_{cubic}=C\ (t-K)^3+W_{max}$ Where,

C=Scaling Factor

t=elapsed time from the last window reduction.

W_{max} =window size=β/C

K=3

β= Constant Multiplication window decrease factor.

t=0.18

C=0.4 and β=0.8[10]

$K=3\sqrt{65535*08/04}=50.7965$

Wcubic=0.4(0.18-50.7965)+65535 =13662.601 or 13663 approx. In this Graph we observe that, CUBIC starts probing for bandwidth in which the window grows slowly initially, accelerating its growth as it moves away from Wmax. This slows growth Wmax enhances the stability of the protocol,and increases the utilization of the network while the fast growth away from Wmax ensures the scalability of the protocol

8.CONCLUSION

In this paper we have analyzed the tcp scenarios with their transmission rate accordingly and we have seen different aspects of tcp in different forms from where we can conclude that we can use tcp in various forms for increasingly the transmission rate with more securing features also we can conclude this paper according to different scenarios of tcp i.e.

8.1. TCP WESTWOOD

In this work, we analyzed the TCP Westwood (TCPW) protocol in Linux. TCPW is a new TCP scheme, which requires modifications only in the TCP source stack and is thus compatible with TCP Reno and Tahoe destinations. Basically it differs from Reno in that it adjusts the *cwin*(congestion window) after a loss detection by setting it to the *measured rate* currently experienced by the connection, rather than using the conventional multiplicative decrease scheme.

We have analyzed with qualitative arguments and with experimental results that the Linux TCPW converges to "fair share." At steady state under uniform path conditions. One general concern with is compatibility towards current implementations. Linux TCPW exhibits some "aggressiveness" due to its unique window adjustment. However, if there is adequate buffering at the bottleneck, TCPW and Reno share the channel fairly.

The Linux implementation was developed in order to combat in presence of random errors and under different scenarios. However, unlike previous TCP versions, the TCPW addresses the bandwidth estimation mechanism and its impact on system behavior. The results show that, for a single connection case, Linux TCPW protocol performs better than or, at least, as well as Linux TCP Reno in terms of congestion avoidance. The results also show that TCPW is more robust under varying buffer size, round trip delays and bottleneck bandwidth. The multiple connections case is under investigation and will be considered in the near future.

8.2.TCP CUBIC

We analyzed Linux TCP CUBIC which simplifies the BIC-TCP window control and improves its RTT-fairness. CUBIC uses a cubic increase function in terms of the elapsed time since the last loss event. In order to provide fairness to Standard TCP, CUBIC also behaves like Standard TCP when the cubic window growth function is slower than Standard TCP. Furthermore, the real-time nature of the protocol keeps the window growth rate independent of RTT, which keeps the protocol TCP friendly under both short and long RTT paths. Through extensive

testing, we confirm that CUBIC tackles the shortcomings of BIC TCP and achieves fairly good congestion avoidance
Also we conclude that the TCP/IP implementation of the Linux Kernel 2.6.9 performed very well, in consideration of the fact that an increase of load had a linear impact on processing time. It should be noted, that this result
might be related to the machines we used. The machines were not able to use their 100 MBit/s connections, instead we managed to use only up to 40 MBit/s. This result is likely due to the relatively poor performance of the used Realtek RTL-8139 network cards. As a result our analysis could be done again on more powerful machines capable of using their connection speed. In order to get more realistic results the connection could be artificially congested. This way TCP's control mechanisms actually take place and its scaling under a more realistic scenario could be tested.
Additionally other implementation and versions of TCP can be tested. Older or future versions of the Linux Kernel along with other operating systems like FreeBSD and NetBSD could be an interesting subject to analyze.

References

[1]Allman, M.V. Paxson, and W.Stevens, (1999),‟ TCP congestion Control,‟ Request for Comments, RFC 2581.
[2] S. Y. Wang, "Decoupling Control from Data for TCP Congestion Control," Ph.D. Thesis, Harvard University, September 1999. (available at http://www.eecs.harvard.edu/networking/decoupling.html)
[3] Andrea Zanella, Gregorio Procissi, Mario Gerla,M.Y. "Medy" sanadidi „ TCP Westwood: Analytic Model and Performance Evaluation‟
[4]D.J.Leith(2003) „Linux Implementation issues in high speed networks.‟ Hamilton Institute technical Report www.hamilton.ie/net/LinuxHoghSpeed.pdf.
[5] Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation.
IEEE/ACM Transactions on Networking, 1(2): 199-216, Apr 1993.
[6] R. Morris, "TCP Behavior with Many Flows," IEEE ICNP'97, Atlanta USA, 1997.
[7] The journey of a packet through the Linux 2.4 network stack, Oct 2000.
URL: http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html
[8] A tool for Linux kernel performance analysis, Apr 2005.
URL: http://user.informatik.uni-goettingen.de/_kperf
[9] The TCP-Friendly Website, http://www.psc.edu/networking/tcp_friendly.html, 1999.
[10] S. Y. Wang and H.T. Kung, "A Simple Methodology for Constructing an Extensible and High-Fidelity TCP/IP Network Simulator," INFOCOM'99, New York, USA, 1999.
[11] FreeBSD web site at www.freebsd.org.
[12] D. Lin, and H. T. Kung, "TCP Fast Recovery Strategies: Analysis and Improvements," INFOCOM'98, San Francisco, USA, 1998.
[13] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," Transactions on Networking, Vol. 1, No. 4, August 1993.
[14] User-mode Linux Kernel (UML).
URL: http://user-mode-linux.sourceforge.net/
[15] Francois Baccelli and Dohy Hong, "TCP is Max-Plus Linear," ACMSIGCOMM'2000, Stockholm, Sweden, 2000.
[16] Eitan Altman, Kostia Avrachenkov, Chadi Barakat, "A Stochastic Model of TCP/IP with Stationary Random Losses," ACM SIGCOMM'2000, Stockholm, Sweden, 2000.
[17] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," ACM SIGCOMM'98, Vancouver, Canada, 1998.
[18] S. Floyd, "TCP and Explicit Congestion Notification," ACM Computer Communication Review, V. 24 N. 5, October 1994, p. 10-23.
[19] K. K. Ramakrishnan, and S. Floyd, "A Proposal to add Explicit Congestion Notification (ECN) to IP," RFC 2481, January 1999.
[20] K. K. Ramakrishnan and R. Jain, "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with Connectionless Network Layer," ACM Transactions on Computer Systems 8(2): 158-181, May 1990.
[21] D.-M. Chiu and R. Jain,(1989), „Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks,‟ *Computer Networks and ISDNSystems*, vol. 17, no. 1, pp. 1
[22] HabibullahJamal,Kiran Sultan (2008) „Performance Analysis of TCP Congestion Control Algorithims‟ International Journal of Computers and Communications. Issue 1, Vol 2.
[23] Kulvinder Singh,, Experimental Study of TCP Congestion Control Algorithms‟ IJCEM International Journal of Computational Engineering & Management, Vol. 14, October 2011
[24] M. Christiansen, K. Jeray, D. Ott, and F. D. Smith, Tuning RED for web traffic," in Proc. of ACM SIGCOMM'00, (Stockholm, Sweden), September 2000
[25] K.Fall and S.Floyd. (1996),, Simulation-based Comparisons of Tahoe, Reno, and SACK TCP ACM Computer Communication review 26(3)
[26] MarioGerla. M.Y.Sanadidi,Ren Wang and Andrea Zanella, Claudio Casetti „ TCP Westwood: Congestion Window Control Using Bandwidth Estimation‟ Computer Science Dept., University of California, Los Angeles (UCLA)
[27] S S.Shenker L. Zhang, and D. D. Clark, (1990),,Some observations on the dynamics of a congestion control algorithim‟ *ACM Computer Communication Review*, vol. 20, pp. 30–39
[28] SaverioMascolo and Francesco Vacirca (2001), „Issues in Performance Evaluation of New TCP Stacks in High Speed‟.
[29] Van Jacobson, (1988), „Congestion Avoidance and Control‟, *Proceedings of the Sigcomm '88 Symposium*, vol.18 (4): pp.314–329. Stanford, CA.