Analysis the Different TCP Traffic Scenario

[1]Nidhi Gaur, [2]Sachin Shrivastava, [3]Dr. Sarika Agarwal, [4]Raj Ranjan Prasad
[1]Department of Computer Science Engineering
[1]Satya College of Engineering and Technology, Palwal, India

***ABSTRACT*- The Purpose of this paper is to evaluate and comparison in the transmission control protocol introduced in linux kernel 2.6. Tahoe and Reno are the classic TCP congestion control from the last decades but now there are many congestion control algorithm to improve them. In this paper simulation scenarios are carefully designed in order to investigate the output of Reno, Tohoe, and Vegas, High speed TCP, TCP westwood, TCP BIC, TCP Cubic. In order to investigate good output, fairness and friendliness provided by each of the algorithms the traffic grows exponentially and the service rate of the server decreases accordingly.**

Index Terms—TCP TRAFFIC, CUBIC, WESTWOOD, VEGAS, RENO.

I. INTRODUCTON

TCP provides the mechanisms that provide data to be transferred across networks that are dynamic and have a large variety of resources. For instance congestion control keeps the network resources from being overloaded without the need for specified information about network resources. This allows for the network to be very scalable and autonomous which has most likely been the reason for the success of the web. Without transmission control protocol, network resources like core links could easily get congested or underutilized. The Data Network traffic comprises of the packets flowing from a source to a destination. This traffic and the network behavior 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 occurred 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 in 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 applications depending on it.

II. TCP CONGESTION CONTROL ALGORITHMS

Standard TCP

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

i. Slow Start
ii. Congestion Avoidance
iii. Fast Retransmit
iv. 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 tcpsstresh in time proportional to log sstresh 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.

Fast Retransmit

Fast Retransmit is an enhancement to TCP which reduces the time a sender waits before retransmitting a lost segment. The fast retransmit enhancement works as follows: if a TCP sender receives a specified number of acknowledgements which is usually set to three duplicate acknowledgements with the same acknowledge number (that is, a total of four acknowledgements with the same acknowledgement number), the sender can be reasonably confident that the segment with the next higher sequence number was dropped, and will not arrive out of order. The sender will then retransmit the packet that was presumed dropped before waiting for its timeout.

Fast Recovery

The Fast-Recovery algorithm is implemented together with the Fast-Retransmit algorithm in the so-called Fast-Retransmit/Fast-Recovery algorithm

The Fast-Retransmit/Fast-Recovery algorithm was introduced in 4.3BSD Reno release and is described in the RFC 2001 as follows:

After receiving three duplicated ACKs in a row:

1. Set ssthresh to half the current send window.
2. Retransmit the missing segment
3. Set cwnd=sshtresh+3.
4. Each time the same duplicated ACK arrives, set cwnd++. Transmit a new packet, if allowed by cwnd.
5. If a non-duplicated ACK arrives, then set cwnd = ssthresh and continue with a linear increase of the cwnd (Congestion-Avoidance)

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 (1)
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(2)

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

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 analyze 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.

Fig: Experimental Linux 2.6 Testbed Layout

This paper is organized as follows. Section II presents different scenarios of TCP Traffic. Section III deals comparative study between these scenarios. Possible future work and concluding remarks are presented in section IV.

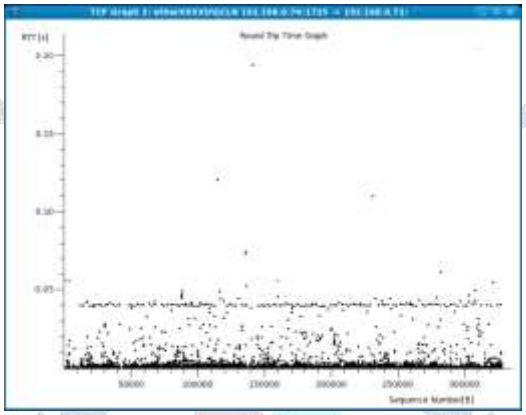
III. DIFFERENT SCENARIOS OF TCP TRAFFIC

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.
i) a)TCP Tahoe (1998, Free BSD 4.3 Tahoe) and TCP Reno (1990, FreeBSD 4.3 Reno) In Tahoe : Loss is detected when a timeout expires before an Acknowledgement is received or duplicate acknowledgement is received. Tahoe will then reduce congestion window

to 1 MSS and reset to slow start state. The fast retransmit algorithm first appeared in the 4.3 BSD Tahoe but it was incorrectly followed by slow start.

b) TCP RENO: It comes with fast recovery algorithm. When duplicate acknowledgement received by the receiver, it do the Fast Re - transmit the data without waiting timer to expire and enter in congestion avoidance but not slow start is performed that mean it perform Fast Recovery.

Fast Recovery: It avoids slow start after a faster retransmit, the reason behind is that the receipts of duplicate acknowledgement tell us more than just a packet has been lost. Since the receiver can only generate the duplicate acknowledgement. When another segment is received, that segment has left the network and is in the receiver's buffer that is there is still data flowing between the two ends we don't want to reduce the flow abruptly by going into slow start. In Fast Recovery, after three duplicate Acknowledgement it retransmit the lost packet and setssthresh =CWND/2, CWND=ssthresh+3 and then enters in Enter congestion avoidance phase and then Increment CWND by one for each additional duplicate ACK. When the next Ack. arrives that acknowledges new data set CWND=ssthresh and enters in Congestion avoidance phase.



ANALYSIS OF RENO

In the analysis of TCP Reno, 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 As per the graph shown, the maximum RTT was around 0.195 sec and a large number of segments got delays on microsecond scale not visible on minisecond scale. This is due to the LAN based connectivity with no logical intervention from L2 switch side. Since RTT estimations don"t infer the state of congestion in TCP Vegas, these values cant be deployed for any „practical" advantages.

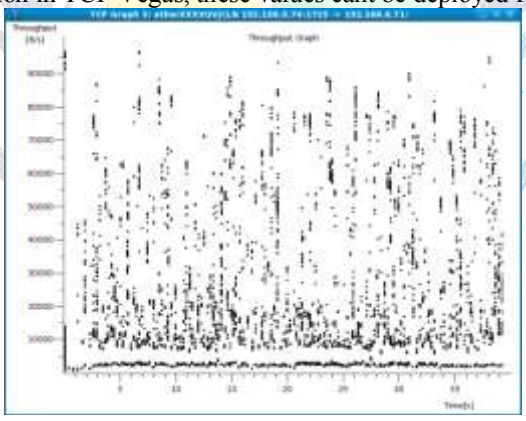


Fig: Throughput Graph

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 as per – ssthresh = CWND/2 CWND = ssthresh + 3 The graph shows that there is sharp increase and successive decrease in the throughput due to AIMD algorithm. There are a large number of apparent traces showing ssthresh = CWND/2. The throughput touches the peak of 757.8125 kbps with an immediate corrective congestion window size afterwards. The nature of the traffic is spiky as per AIMD and it clearly matches the objectives of the algorithm. The nature is also self similar since every peak is repeated at almost regular intervals. The peak of 750 kbps and above is repeated after almost 32 seconds. TCP is able to deal with excessive traffic using AIMD pretty well. The sizing constraints of CWND and ssthresh recursively cut short the possibilities of congestion

ii) TCP Vegas:- from the mid 1990s, all of TCP`s set timeouts and measured round-trip delays were based upon only the last transmitted packet in the transmit buffer. University of Arizona introduced TCP Vegas, in which timeouts were set and round-trip delays were measured for every packet in the transmit buffer. In addition, TCP Vegas uses additive increases in the congestion window.

TCP Vegas was the first attempt to depart from the loss-driven paradigm of the TCP by introducing a mechanism of congestion detection before packet losses. The major drawback in TCP Reno is that it does not receive fair share of bandwidth. In TCP Reno while a source does not detect any congestion, it continues to increase its window size by one during one round trip time obviously the connections with shorter delays can update the connections with shorter delay can update their window sizes faster than those with longer delays and thus faster than those with longer delays and thus steal higher bandwidth. It is harmful to the other version of TCP Connection with longer delays. It takes the difference between expected and actual flow rates to estimate the available bandwidth in the network. *Difference = (Expected Rate – Actual Rate) Base RTT.*
Base RTT = Min. Round Trip Time.

Expected Rate=CWND/Base RTT. Actual Rate=CWND/RTT CWND= Current Congestion window RTT= Actual RTT when network is not congested, the actual flow rate will be very close to expected flow rate otherwise Actual rate is smaller than Expected. **TCP Vegas take the difference and on this basis of this difference in flow rate, estimate congestion level and adjust/update the window size accordingly.**

1. First, the source computes the expected flow rate

Expected = CWND/BaseRTT, where CWND is the current window size and BaseRTT is the minimum round trip time. 2. Second, the source estimates the current ow rate by using the actual round trip time according to Actual = CWND/RTT , where RTT is the actual round trip time of a packet. 3. The source, using the expected and actual ow rates, computes the estimated backlog in the queue from Diff = (Expected Rate-Actual Rate) BaseRTT. 4 Based on Diff , the source updates its window size

Analysis of Vegas
In the analysis of TCP Vegas, 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.

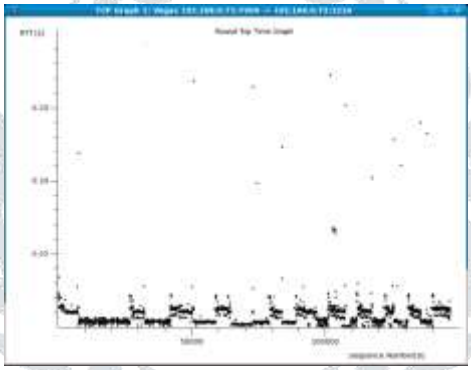


Fig: RTT Graph of Vegas

As per the graph shown, the minimum RTT was around 0.001 sec and maximum RTT was around 0.175 we can calculate the capacity of the pipe as Capacity (bits) = bandwidth (bits/sec) × round-trip time (sec)

This is normally called the bandwidth-delay product. This value can vary widely, depending on the network speed and the RTT between the two ends. On similar pattern, TCP Vegas Congestion Control is based on two parameters representing „expected" and „actual" rate calculated as –: Difference = (Expected Rate – Actual Rate) * Base RTT. Base RTT = Min. Round Trip Time. Expected Rate = CWND/Base RTT. Actual Rate = CWND/RTT CWND = Current Congestion window RTT = Actual RTT If we analyze our graph that show following parameters. Here, Base RTT = 0.001 Current Window size = 65535 bits. Expected Rate=65535/0.001=65535000 Actual Rate=65535/0.005=13107000 Difference = (65535000-13107000)*0.001 = 51828. The congestion window is adjusted depending upon the difference between expected and actual sending rates. Also two thresholds α and β are defined, such that α > β correspond to having too little and too much extra traffic in the network, respectively. When Difference < α, TCP Vegas increases the congestion window linearly during the next RTT, and when Difference > β, TCP Vegas decrease the congestion window linearly during the next RTT. The Congestion window is left unchanged when α <Difference < β. Here α =1and β =3, As our Difference > β, TCP Vegas decrease the congestion window linearly during the next RTT Therefore, we can say that TCP Vegas computes Difference and compares it with a unique threshold α =1; as long as Difference is less than α! And congestion window size is less than the slow start threshold, the TCP Vegas congestion window is doubled every other round trip delay. After slow start, TCP Vegas performs the congestion avoidance algorithm. When TCP Vegas source receives three duplicate acks, it performs fast retransmit and fast recovery as TCP Reno. Actually, TCP Vegas develops a more refined fast retransmit mechanism based on a fine-grain clock. After fast retransmit TCP Vegas sets the congestion window to ¾ of the current congestion window and performs again the congestion avoidance algorithm.

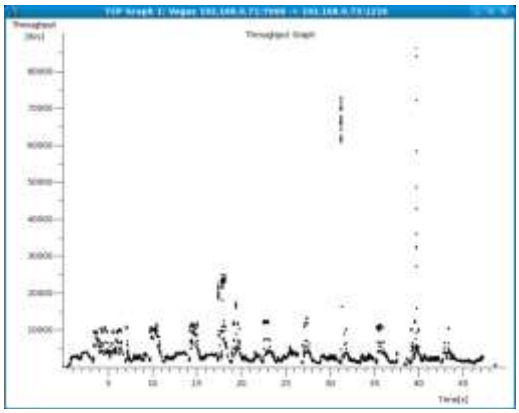


Fig: Throughput graph of TCP Vegas

iii) Westwood:- TCP Westwood estimates the available bandwidth by counting and filtering the flow of returning ACKs and adaptively sets the cwnd and the sshtresh after congestion by taking into account the estimated bandwidth. TCP Westwood develops 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. TCP Westwood 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
```

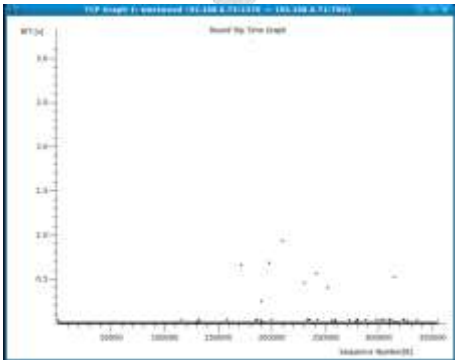


Fig : RTT Graph for TCP Westwood(TCPW)

Analysis of Westwood:-

TCP Westwood is a new congestion algorithm that is based on end-to-end bandwidth estimate. The estimate is obtained by filtering the stream of returning ACK packets and it is used to adaptively set the control windows when network congestion is experienced. In the

analysis of TCPW, 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. As per the graph shown, the minimum RTT was around 0.001 sec and maximum RTT was around 1sec

We can calculate the capacity of the pipe ascapacity (bits) = bandwidth (bits/sec) × round-trip time (sec) This is normally called the bandwidth-delay product. This value can vary widely, depending on the network speed and the RTT between the two ends. On similar pattern, 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). We compare two high-speed flows with a different RTT i.e. AIMD and Westwood. We observe the RTT of both cases as different and check which one is better RTT fair.

As per our analysis around small window sizes, TCPW shows the RTT unfairness. TCPW has window sizes around 200 for 100Mbps. Nonetheless, its RTT unfairness is much better than AIMD where we get a random RTT scenario. TCPW in Linux can better handle the congestion scenario under excess traffic.

The RTT observed is less but the numbers of segments are also low. This shows the combative state of TCP while the socket receive buffers get continuously overflowed. It shows as if a large number of SYN segments have been dropped either by socket receive buffer or the congestion window sizing.

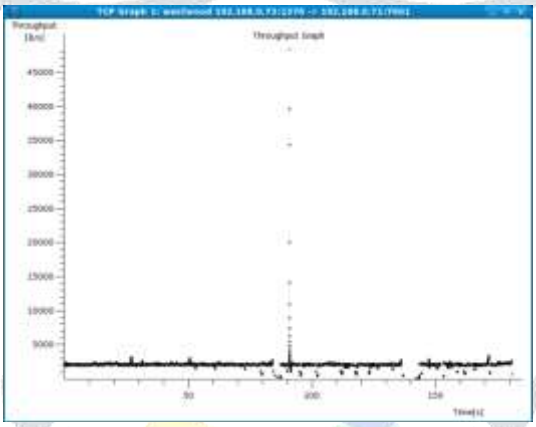


Fig: Throughput Graph for TCP Westwood(TCPW)

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 infer an estimate of connection available bandwidth (BWE). At the point of congestion,

When 3 DUPACKs are received by the sender :

ssthresh = (BWE * RTTmin) / MSS;
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 analyzed 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.

v) TCP CUBIC:- As name suggest 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.

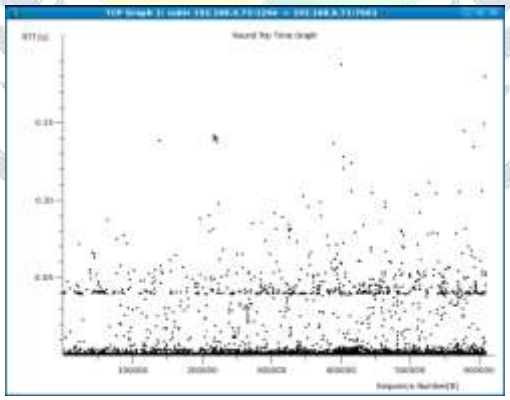


Fig: RTT Graph for TCP CUBIC

As per the graph shown, the minimum RTT was around 0.001 sec and maximum RTT was around 0.18.
The congestion window of CUBIC is determined by
Wcubic=C (t-K)3+Wmax
Where,
C=Scaling Factor
t=elapsed time from the last window reduction.
Wmax =window size=β/C
K=3
β= Constatnt Multiplication window decrease factor.
t=0.18
C=0.4 and β=0.8[10]
K=3√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.

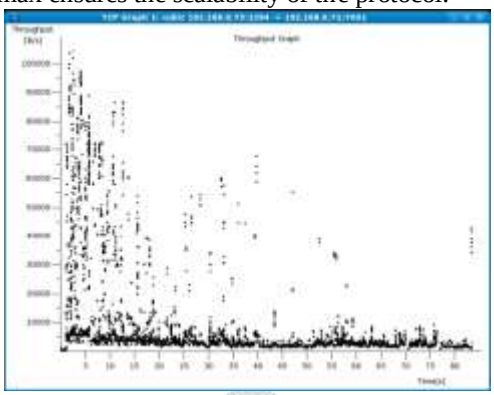


Fig: Throughput Graph for TCP CUBIC

CUBIC TCP achieves greater utilization then standard duTCP . The Graph shows that there is almost steady state due to CUBIC. The highest peak of throughput goes to 105000B/s and with an immediate corrective congestion window size afterwards. Earlier there was some sharp increase and then CUBIC maintains steady state. Unlike AIMD,Cubic increases the window Wmax very quickly and then holds the window for a long time . This keeps the scalibilty of the protocol high, while keeping the epoch long and utilization high.

IV. COMPARATIVE STUDY BETWEEN THESE SCENARIOS

In this section, we compare the performance of Linux CUBIC, vegas, Reno w.r.t. AIMD. In the analysis of cubic, we use Linux hosts as communication endpoints communicating over 100Mbps link with MTU of 1500 bytes.

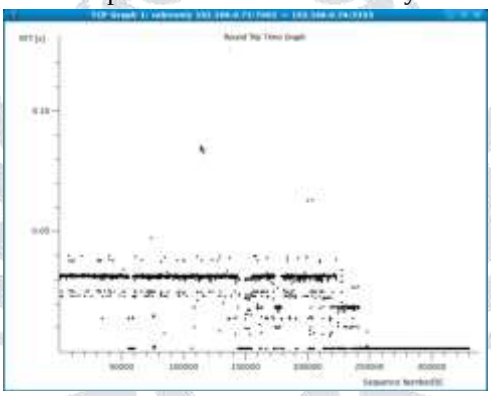


Fig:RTT Graph for TCP CUBIC+Vegas+Reno

The RTT of each background traffic is random. The socket buffer size of some client machine is fixed to default 64KB. We evaluate CUBIC-TCP and AIMD for the bandwidth utilization and RTT.

The Throughput touches the peak of 35000 B/s 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 and it far better matches the objective of the congestion control algorithim . The nature is less self similar in the trace received by us till the congestion collapse was finally achieved.

V. CONCLUSION

When a new set of protocol is proposed, it is necessary to collect a large set of simulation and experimental results in order to assess its validity and advantages of its deployment in the real Internet.

The average goodput of a measurements session is obtained by averaging the goodputs of the session uploads.

To summarize, changes to TCP are in progress that would continue to bring TCP's congestion control behavior closer to the goal of AIMD for larger congestion windows, and exponential backoff of the retransmit timer for regimes of higher congestion.

TCP RENO

We have analyzed a simple setup of the Linux TCP Reno protocol. It captures the essence of TCP's congestion avoidance behavior and expresses send rate as a function of loss rate. We analyzed the behavior of the protocol in the presence of time-outs, and is valid over the entire range of loss probabilities. We have compared it with the theoretical model and found that most of these connections suffered from a significant number of time-outs. Thus, we conclude that time-outs have a significant impact on the performance of the TCP

protocol, and that our model is able to account for this impact. A number of avenues for future work remain. First, the implementation can be enhanced to account for the effects of fast recovery and fast retransmit. Second, we have observed that once a packet in a given round is lost, all remaining packets in that round are lost as well. The Linux Reno can be modified to incorporate a loss distribution function. Third, it is interesting to further investigate the behavior of TCP Reno over slow links with dedicated buffers (such as modem lines).

TCP VEGAS

We analyzed the TCP Vegas behavior and found a need of the modification of the basic TCP Vegas congestion control in Linux to overcome the mixed TCP Vegas and TCP Reno network scenarios

Linux TCP Vegas reaches better congestion avoidance performance with respect to TCP Reno in quite a number of tests. Linux TCP Vegas preserves the fairness characteristic of the original TCP Vegas in several ways. When Linux TCP Vegas cannot reach the exact fairness degree of TCP Vegas (actually TCP Vegas needs slightly larger buffers to reach the fairness, since it naturally tries to force the stability), it performs a higher throughput, posing itself as a good trade-off between the features of theory and practice. A still open issue, not addressed in the present work, is to investigate the behavior of Linux TCP Vegas in network scenarios with non-persistent sources. Leaving the slow start mechanism of Linux TCP Vegas unchanged; Linux TCP Reno still appears to take advantage from its much more aggressive start-up. To allow Linux TCP Vegas to implement TCP Reno's slow start would reduce this disadvantage but it could generate a consequent loss of some TCP Vegas features, like stability and fairness.

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.

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

The future researches will continue in direction to discoveries of new mechanisms and creation of new version of TCP protocols aiming in combination of existing version of TCP.

REFERENCES

[1] Allman, M.V. Paxson, and W.Stevens, (1999),‟ TCP congestion Control,‟ Request for Comments, RFC 2581.
[2] Andrea Zanella, Gregorio Procissi, Mario Gerla,M.Y. "Medy" sanadidi „ TCP Westwood: Analytic Model and Performance Evaluation‟
[3] D.J.Leith and R.N.Shorten(2006) „ Impact of drop synchronization on TCP fairness in high bandwidth-delay product networks.‟ Workshop on Protocols for Fast Long Distance Networks, Nara, Japan.
[4] D.J.Leith(2003) „Linux Implementation issues in high speed networks.‟ Hamilton Institute technical Report www.hamilton.ie/net/LinuxHoghSpeed.pdf.
[5] D.J.Leith, R.N. Shorten, G.McCullagh, (2005) „Experimental evaluation of CUBIC-TCP‟ Hamilton Institute,Ireland.
[6] 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
[7] Evandro de Souza,debAgarwal „ A Highspeed TCP study:Characteristics and Deployment Issues‟ Lawrence Berkeley National Lab, Berkeley, CA, USA.
[8] FEDORA 11. [Online]. Available: http://fedora.redhat.com

[9] HabibullahJamal,Kiran Sultan (2008) „Performance Analysis of TCP Congestion Control Algorithims‟ International Journal of Computers and Communications. Issue 1, Vol 2.
[10] Injong Rhee, and LisongXu „ CUBIC: A New TCP-Friendly High Speed TCP Variant‟ LincolnNE68588-0115USA
[11] J. C. Hoe,(1996) Improving the start-up behavior of a congestion control scheme of TCP," ACMComputer Communication Review, vol. 26, pp. 270–280, Oct. 1996.
[12] J. Nagle, (1984), „Congestion Control in IP/TCP Internetworks," IETF RFC 896
[13] JeonghoonMo, Richard J. La, VenkatAnantharam, and Jean Warland (1998) „Analysis and comparison of TCP Reno and Vegas‟ eecs.berleley.edu.
[14] JitendraPadhye, victorFiroiu, Donald F. Towsley, James F.Kurose(2000). Modeling TCP RenoPerformance : A Simple Model and its Emperical Validation‟ IEEE/ACM Transaction on Networking Vol.8.No.2. April 2000
[15] Joachim charzinksi,(2000) „HTTP /TCP connection and flow characteristics.Vol.42 issues 2-3
[16] K.Fall and S.Floyd. (1996)„ Simulation-based Comparisons of Tahoe, Reno, and SACK TCP ACM Computer Communication review 26(3)
[17] L. S. Brakmo, S. W.O‟Malley, and L. L. Peterson, (1994) „TCP Vegas: New techniques for congestion detection and avoidance‟ ACM SIGCOMM ’94.