Comparative Study of Congestion Control Mechanisms

Project Report Proposal

Tushar Agarwal
1783910056
agarwal.tushar2905@gmail.com

January 31, 2020

1 Introduction

Congestion refers to a stage in a network in which a node or link carries more data than it can handle which ultimately leads to delay increases, performance degradation. Congestion occurs when the load on the communication network sends packets in excess of its capacity. TCP is a transport layer protocol which used to interconnect network devices on the internet. It is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which supports a variety network application.

To avoid or reduce the congestion problem, congestion control techniques are used.

Congestion control refers to the mechanism or algorithm used to ensuring the network stability and robustness. To control the congestion, in a communication network, there can be two ways, either stopping before congestion or removing congestion after this happens. A congestion window is maintained, which transmits the maximum number of data packets over the communications network without being accepted. Update this congestion window and various congestion control algorithms used efficiently to transmit data. Various algorithm and mechanisms are following:

- Tahoe.
- Reno.
- Vegas
- Cubic
- BIC
- BBR

2. Project phases

The work of this project is divided in following three phases:

Phase1:

In the phases one, we will be studying about the TCP/IP transport layer protocol. After that a step wise deep study of various congestion control mechanisms will be done. A report of study will be prepared and it will be submitted by 28 February, 2020.

Phase 2:

In the second phase, we will be simulating the mechanisms that we had studied in the previous phase. Simulation will be done by using a software network simulating tool 'Netsim'. Simulation report will be submitted by 22 March,2020.

Phase 3:

In the last phase of project, we will analyze and conclude the results of phase 2 and will prepare a cumulative report. We will try to conclude best possible mechanisms for Congestion control after the practical implementation of each congestion control mechanism. Report for phase 3 will be submitted by 12 April, 2020.

Final demo:

Final demo of project and final report will be submitted on 20 April, 2020.

3 References

1. Brakmo, Lawrence S., Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas:New techniquesforcongestiondetectionandavoidance."Proceedingsoftheconferenceon Communications architectures, protocols and applications.1994.
2. M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.
3. Hua, Wu, and Gong Jian. "Analysis of TCP BIC Congestion Control Implemen- tation." 2012 International Conference on Computer Science and Service System. IEEE,2012.
4. Scholz, Dominik, et al. "Towards a deeper understanding of tcpbbr congestion con- trol." 2018 IFIP Networking Conference (IFIP Networking) and Workshops. IEEE, 2018.

Phase 1: Report of study of Tcp/Ip and congestion control mechanism.

1 Introduction

The TCP/IP model, it was designed and developed by Department of Defense (DoD) in 1960s and is based on standard protocols. It stands for Transmission Control Protocol/Internet Protocol. The TCP/IP model is a concise version of the OSI model. It contains four layers, unlike seven layers in the OSI model. The layers are:
1. Process/Application Layer.
2. Host-to-Host/Transport Layer.
3. Internet Layer
4. Network Access/Link Layer

1. Network Access Layer {
This layer corresponds to the combination of Data Link Layer and Physical Layer of the OSI model. It looks out for hardware addressing and the protocols present in this layer allows for the physical transmission of data. We just talked about ARP being a protocol of Internet layer, but there is a conflicts about declaring it as a protocol of Internet Layer or Network access layer. It is described as residing in layer 3, being encapsulated by layer 2 protocols.

2**. Internet Layer** {
This layer parallels the functions of OSI's Network layer. It defines the protocols which are responsible for logical transmission of data over the entire network. The main protocols residing at this layer are:

1. IP {stands for Internet Protocol and it is responsible for delivering packets from the source host to the destination host by looking at the IP addresses in the packet headers. IP has 2 versions:
2. IPv4 and IPv6. IPv4 is the one that most of the websites are using currently. But IPv6 is growing as the number of IPv4 addresses are limited in number when compared to the number of users.
3. ICMP {stands for Internet Control Message Protocol. It is encapsulated within IP datagrams and is responsible for providing hosts with information about network problems.
4. ARP {stands for Address Resolution Protocol. Its job is to find the hardware address of a host from a known IP address. ARP has several types: Reverse ARP, Proxy ARP, Gratuitous ARP and Inverse ARP.

3. **Host-to-Host Layer** {
This layer is analogous to the transport layer of the OSI model. It is responsible for end-to-end communication and error-free delivery of data. It shields the upper-layer applications from the complexities of data. The two main protocols present in this layer are:

1. Transmission Control Protocol (TCP) {It is known to provide reliable and error-free communication between end systems. It performs sequencing and segmentation

of data. It also has acknowledgment feature and controls the ow of the data through ow control mechanism. It is a very effective protocol but has a lot of overhead due to such features. Increased overhead leads to increased cost.

2. User Datagram Protocol (UDP) {On the other hand does not provide any such features. It is the go-to protocol if your application does not require reliable transport as it is very cost-effective. Unlike TCP, which is connection-oriented protocol, UDP is connectionless.

4. Process Layer {

This layer performs the functions of top three layers of the OSI model: Application, Presentation and Session Layer. It is responsible for node-to-node communication and controls user-interface specifications. Some of the protocols present in this layer are: HTTP, HTTPS, FTP, TFTP, Telnet, SSH, SMTP, SNMP, NTP, DNS, DHCP, NFS, X Window, LPD. Have a look at Protocols in Application Layer for some information about these protocols. Protocols other than those present in the linked article are:

1. HTTP and HTTPS {HTTP stands for Hypertext transfer protocol. It is used by the World Wide Web to manage communications between web browsers and servers. HTTPS stands for HTTP-Secure. It is a combination of HTTP with SSL (Secure Socket Layer). It is efficient in cases where the browser need to fill out forms, sign in, authenticate and carry out bank transactions.

2. SSH {SSH stands for Secure Shell. It is a terminal emulations software similar to Telnet. The reason SSH is more preferred is because of its ability to maintain the encrypted connection. It sets up a secure session over a TCP/IP connection.

3. NTP {NTP stands for Network Time Protocol. It is used to synchronize the clocks on our computer to one standard time source. It is very useful in situations like bank transactions. Assume the following situation without the presence of NTP. Suppose you carry out a transaction, where your computer reads the time at 2:30 PM while the server records it at 2:28 PM. The server can crash very badly if it's out of sync.

2 TCP Congestion Control

TCP uses a congestion window and a congestion policy that avoid congestion. Previously, we assumed that only receiver can dictate the sender's window size. We ignored another entity here, the network. If the network cannot deliver the data as fast as it is created by the sender, it must tell the sender to slow down. In other words, in addition to the receiver, the network is a second entity that determines the size of the sender's window.
Congestion policy in TCP {
1. Slow Start Phase: starts slowly increment is exponential to threshold
2. Congestion Avoidance Phase: After reaching the threshold increment is by 1
3. Congestion Detection Phase: Sender goes back to Slow start phase or Congestion avoidance phase.

Slow Start Phase: exponential increment {In this phase after every RTT the congestion window size increments exponentially.
Congestion Avoidance Phase: additive increment {This phase starts after the threshold value also denoted as ssthresh. The size of cwnd (congestion window) increases additive. After each *RTT cwnd = cwnd + 1.*

Congestion Detection Phase: multiplicative decrement {If congestion occurs, the congestion window size is decreased. The only way a sender can guess that congestion has occurred is the need to retransmit a segment. Retransmission is needed to recover a missing packet which is assumed to have been dropped by a router due to congestion. Retransmitssion can occur in one of two cases: when the RTO timer times out or when three duplicates ACKs are received.

Case 1: Retransmission due to Timeout { In this case congestion possibility is high.
(a) ssthresh is reduced to half of the current window size. (b) set cwnd = 1 (c) start with slow start phase again.

Case 2: Retransmission due to 3 Acknowledgement Duplicates { In this case congestion possibility is less.
(a) ssthresh value reduces to half of the current window size.
(b) set cwnd= ssthresh
(c) start with congestion avoidance phase

4 Study of Congestion Control Mechanisms

Network congestion may occur when a sender overawes the network with too many packets. At the time of congestion, the network cannot handle this traffic properly, which results in a degraded quality of service (QoS). The typical symptoms of a congestion are: excessive packet delay, packet loss and retransmission.

The function of TCP (Transmission Control Protocol) is to control the transfer of data so that it is reliable. The main TCP features are connection management, reliability, ow control and congestion control. Connection management includes connection initialization (a 3-way handshake) and its termination. The source and destination TCP ports are used for creating multiple virtual connections.

A reliable P2P transfer between hosts is achieved with the sequence numbers (used for segments reordering) and retransmission. A retransmission of the TCP segments occurs after a timeout, when the acknowledgement (ACK) is not received by the sender or when there are three duplicate ACKs received (it is called fast retransmission when a sender is not waiting until the timeout expires). Flow control ensures that a sender does not overflow a receiving host. The receiver informs a sender on how much data it can send without receiving ACK from the receiver inside of the receiver's ACK message. This option is called the sliding window and it's amount is defined in Bytes.

Different variants of TCP use different approaches to calculate cwnd, based on the amount of congestion on the link. For instance, the oldest TCP variant {the Old Tahoe initially sets cnwd to one Maximum Segment Size (MSS). After each ACK packet is received, the sender increases the cwnd size by one MSS. Cwnd is exponentially increased, following the formula*: cwnd = cwnd + MSS.* This phase is known as a \slow start" where the cnwd value is less than the ssthresh value.

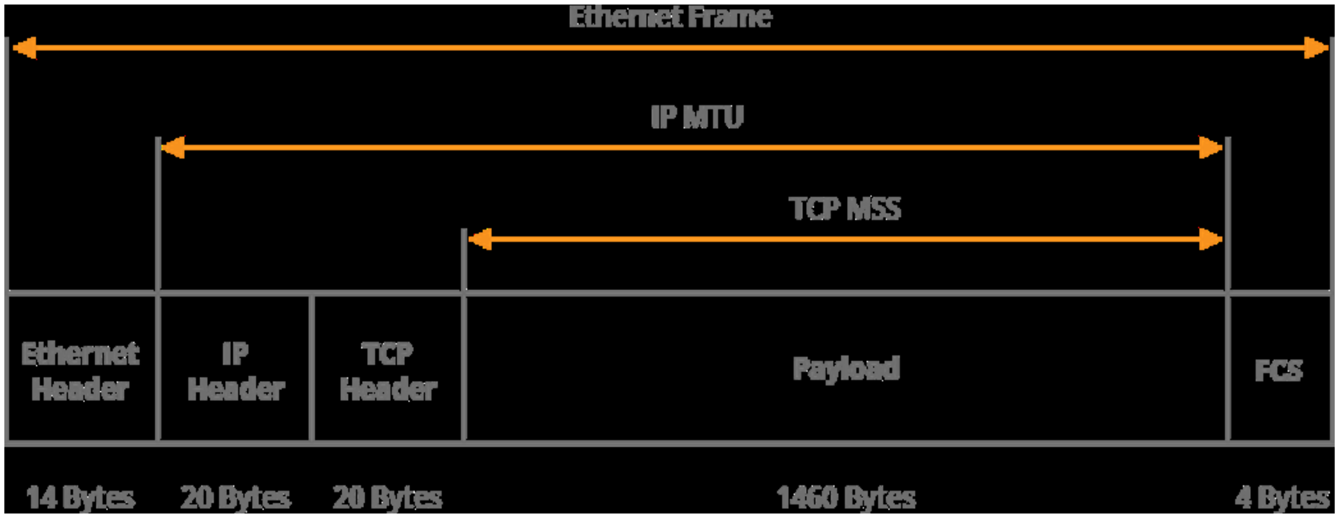

Figure 1: TCP MSS 1460 Bytes Inside Ethernet Frame

Insufficient link bandwidth, legacy network devices, greedy network applications or poorly designed or configured network infrastructure are among the common causes of congestion. For instance, a large number of hosts in a LAN can cause a broadcast storm, which in its turn saturates the network and increases the CPU load of hosts. On the Internet, traffic may be routed via the shortest but not the optimal AS PATH, with the bandwidth of links not being taken into account. Legacy or outdated network device may represent a bottleneck for packets, increasing the time that the packets spend waiting in buffer. Greedy network applications or services, such as file sharing, video streaming using UDP, etc., lacking TCP ow or congestion control mechanisms can significantly contribute to congestion as well.

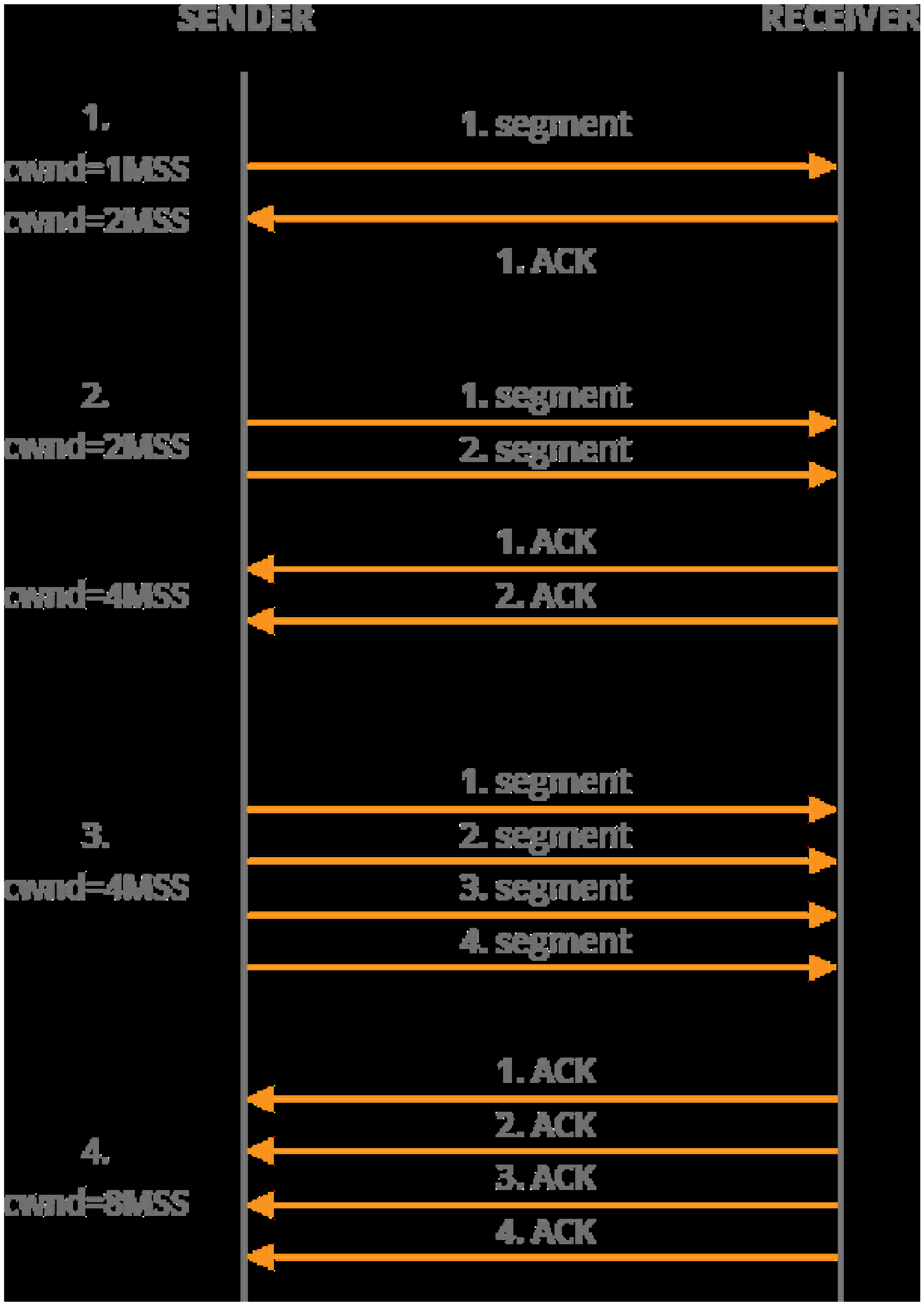

Figure 2: Old Tahoe Slow Start Algorithm

When the slow start threshold (ssthresh) is reached, TCP switches from the slow start phase to the congestion avoidance phase. The cwnd is changed according to the formula: *cwnd = cwnd + MSS /cwnd* after each received ACK packet. It ensures that the cwnd growth is linear, thus increased slower than during the slow start phase. However, when TCP sender detects packet loss (receipt of duplicate ACKs or the retransmission timeout when ACK is not received), cwnd is decreased to one MSS. Slow start threshold is then set to half of the current cwnd size and TCP resumes the slow start phase.

Phase 2: Simulation of Various Mechanism

Note: we doesn't have netsim software so we have done on ns3.

1 Introduction

The topology in the wired network is set-up using the node and link creation APIs. The bottleneck is a duplex link that has router in both directions representing the traffic flow from multiple sources. The congestion in the wired network can be created using the bottleneck link.
The tcl script in test5.tcl creates the congestion in which each link is configured with the specific bandwidth, propagation delay and Drop- Tail queue. Data transmission between the sender1 and receiver1 is created using the tcp connection and FTP application. Data transmission between the sender2 and receiver2 is created using the udp agent and CBR traffic.
Congestion occurs at the link between router1 and router2 (bottleneck link) due to the high data flow rate that exceeds the link capacity. The length of the queue is limited. TCP enforces congestion control mechanism automatically by adjusting the sending data rate according to the acknowledgment arrival time. TCP is a transport layer protocol used by applications that require guaranteed delivery. It is a sliding window protocol that provides:

1. Handling for both timeouts and retransmissions.
2. Communication service at an intermediate level between an application program and the Internet Protocol
3. Host-to-host connectivity at the Transport Layer of the Internet model.
4. Full duplex virtual connection between two endpoints. Each endpoint is defined by an IP address and a TCP port number. In TCP, the congestion window is one of the factors that determines the number of bytes that can be outstanding at any time. Above diagram is an example in which congestion may occur when all clients sends data at the same time to the server.

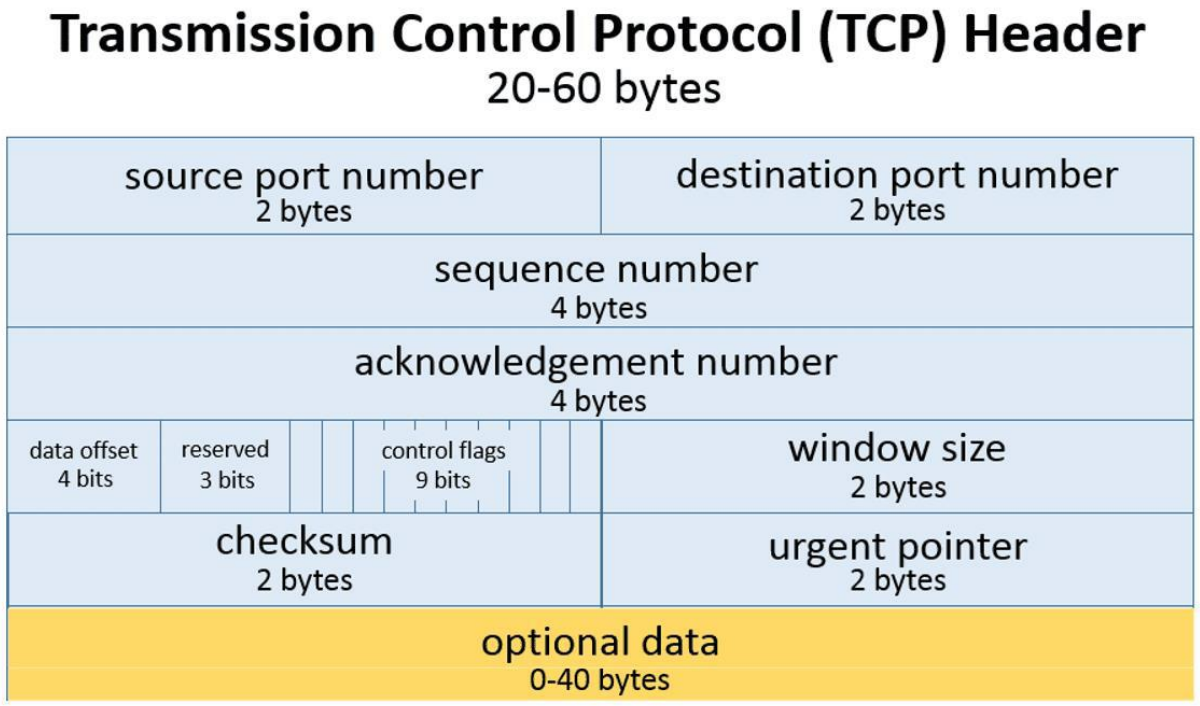

Figure 1: Above is the header format of TCP:

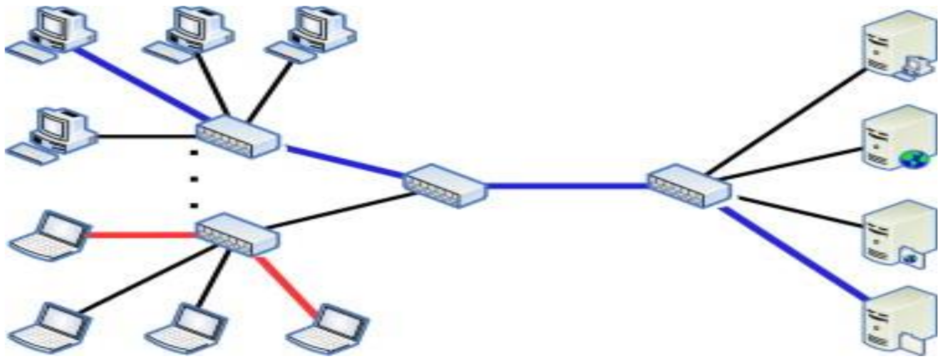

Figure 2: All clients sends data at the same time to the server.

2 Evaluation Features / Parameters

TCL SCRIPT FOR CONGESTION CONTROL IN TCP
create links between the nodes
$ns duplex-link n0n2 2Mb 10ms DropTail
$ns duplex-link n1n2 2Mb 10ms DropTail
$ns simplex-link n2n3 0.3Mb 100ms DropTail
$ns simplex-link n3n2 0.3Mb 100ms DropTail
$ns duplex-link n3n4 0.5Mb 40ms DropTail
$ns duplex-link n3n5 0.5Mb 30ms DropTail
Set Queue Size of link (n2-n3) to 10
$ns queue-limit n2n3 20
Setup a TCP connection
set tcp [new Agent/TCP/Newreno]
$ns attach-agent n0tcp
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent n4sink
$ns connect tcpsink
$tcp set fid1
$tcp set window8000
$tcp set packetSize552
Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent tcp
$ftp set typeFTP
Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent n1udp
set null [new Agent/Null]
$ns attach-agent n5null
$ns connect ud pnull
$udp set fid2
4 Study of Congestion Control Mechanisms
Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent ud p
$cbr set typeCBR
$cbr set packetsize1000

```
$cbr set rate0:01mb
$cbr set randomf alse
$ns at 0.1 ”cbrstart”
$ns at 1.0 ” f t pstart”
$ns at 10.0 ” f t pstop”
$ns at 10.5 ”cbrstop”
```

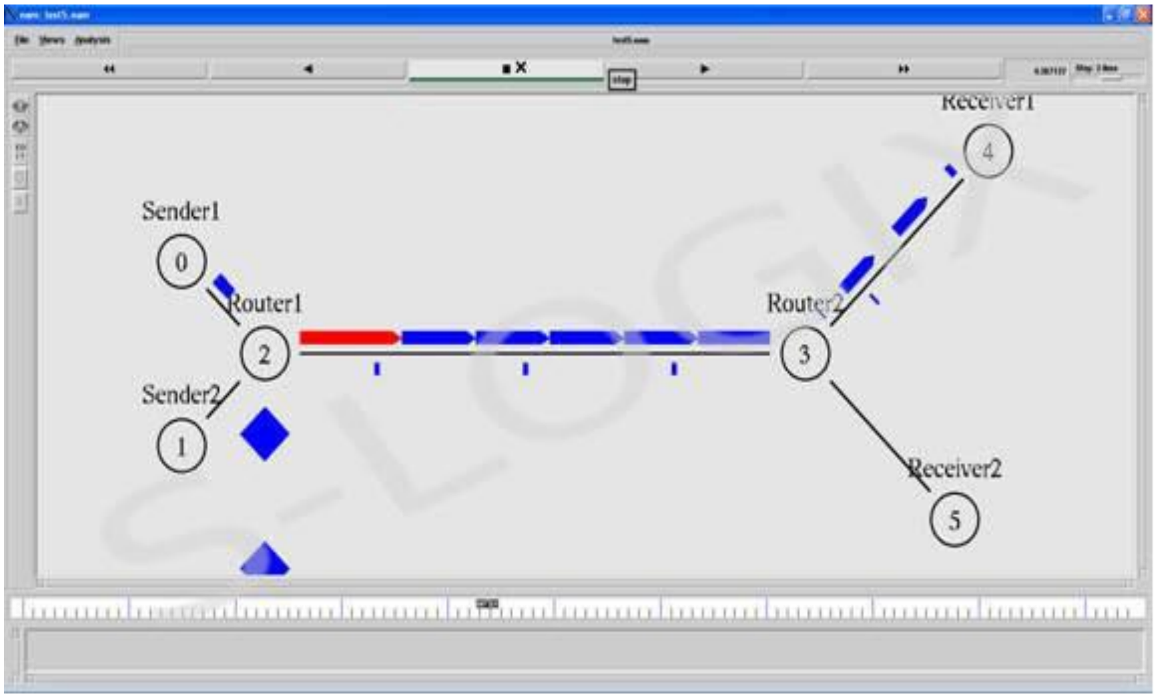

Figure 3: Congestion.

1: Congestion window

TCP and its Algorithms (Slow-Start, Congestion Avoidance, Fast Retransmit and Fast Recovery) :

TCP is a complex transport layer protocol containing four interwined algorithms: Slow-start, congestion avoidance, fast retransmit and fast recovery. In Slow-start phase, TCP increases the congestion window each time an acknowledgement is received, by number of packets acknowledged. This strategy effectively doubles the TCP congestion window for every round trip time (RTT).

Calculation:

Used TCP Reno for window size calculation. *cwndvariablewasusedtogetthewindowsize f romthetcp*
reno:cc f ile:Startswithslowstart;inwhich : cwnd =cwnd+1;Thisincreasesexponentiallyuntil packetstartsdropping:Whenpacketsdrops

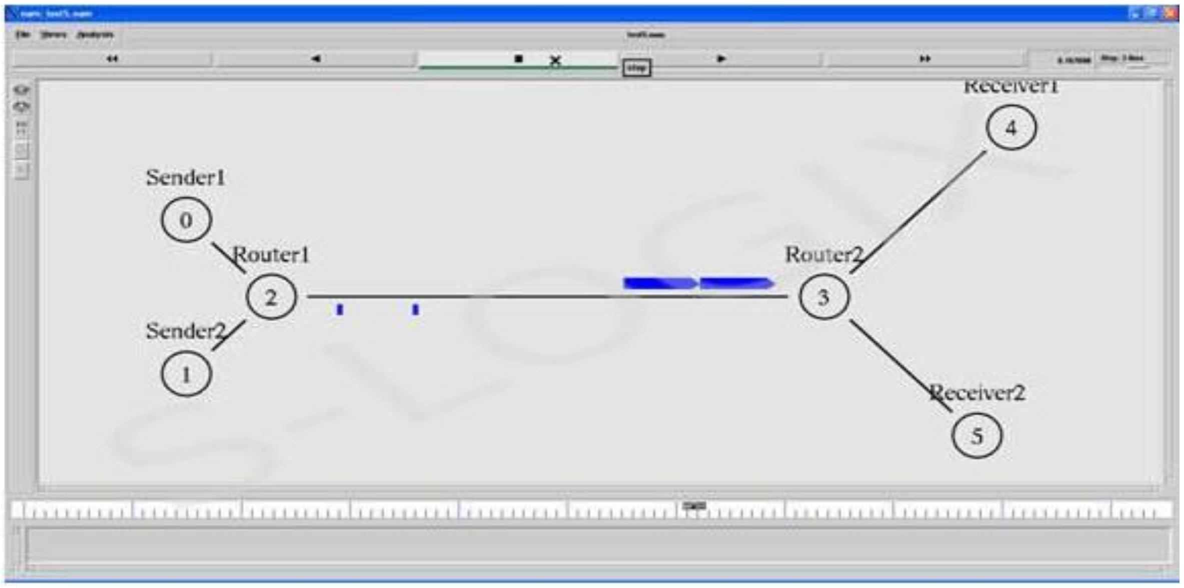

Figure 4: Congestion control mechanism of TCP.

ssthresh = cwnd/2;
Then, cwnd = cwnd + 1/cwnd;
i.e., increases almost linearly.

2. Round Trip Time

Round-trip time (RTT), also called round-trip delay, is the time required for a signal pulse or packet to travel from a specific source to a specific destination and back again. This time delay includes the propagation times for the paths between the two communication endpoints.

Calculation:

Ping Agent was used to calculate RTT. A method of ping.cc called “recv” was being called for the same.
RTT = avg (2 x propagation time)
It helps in calculation of Timeout, which is calculated by:
Timeout = RTT + Back-off time

3 **Graphical Analysis of Protocol**

Congestion window:

Throughput:

Drop-Ratio:

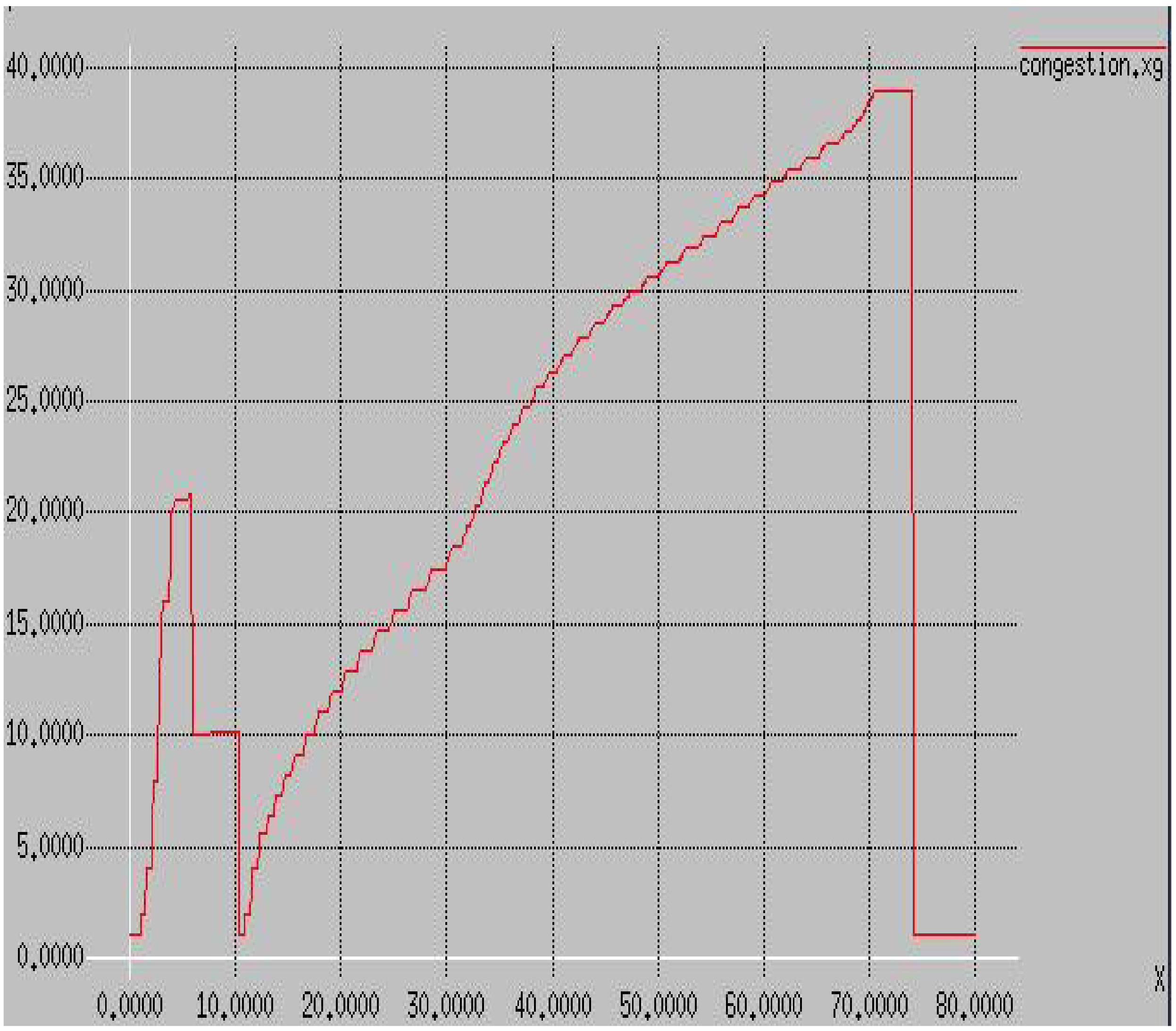

Figure 5: Congestion window:

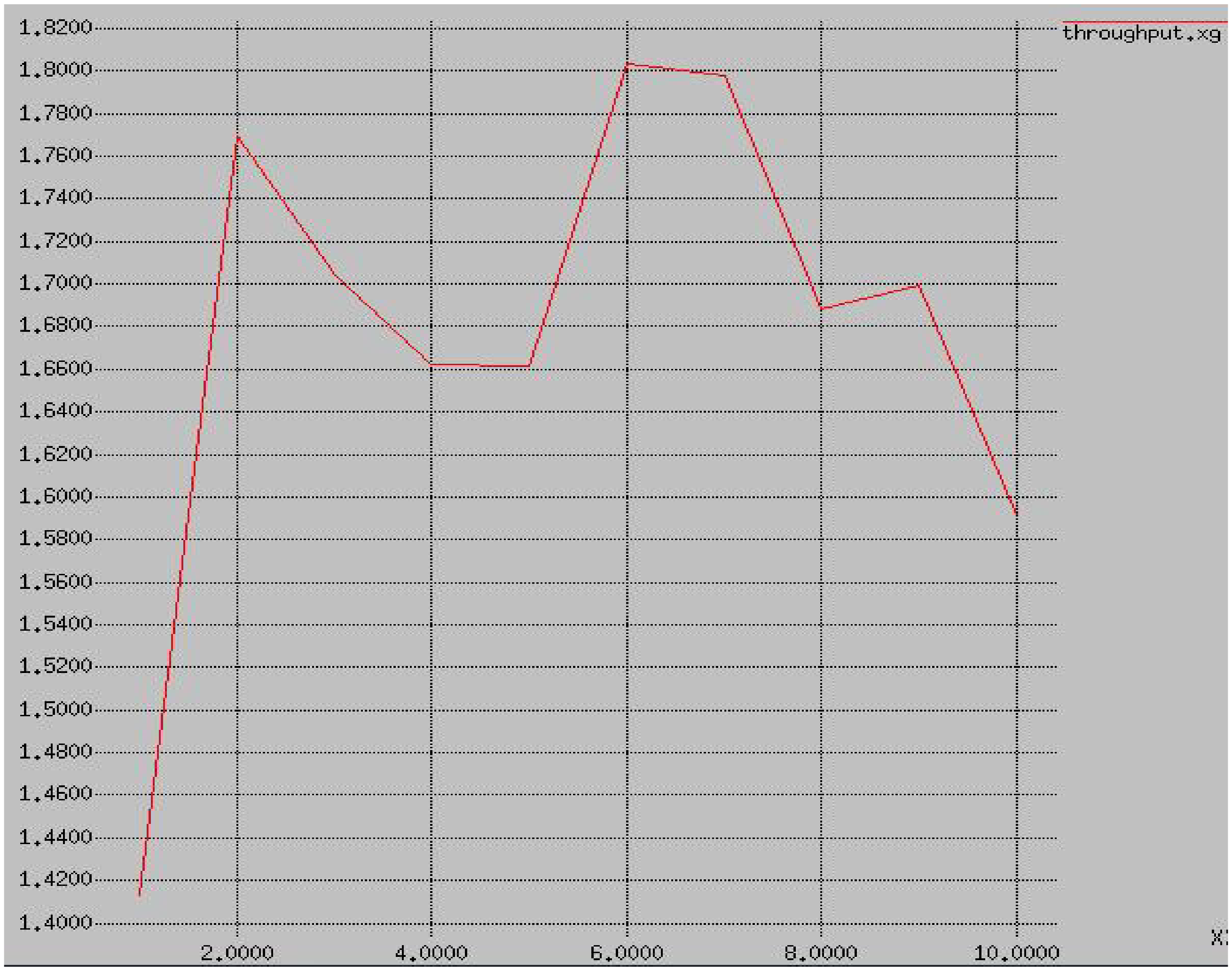

Figure 6: Throughput:

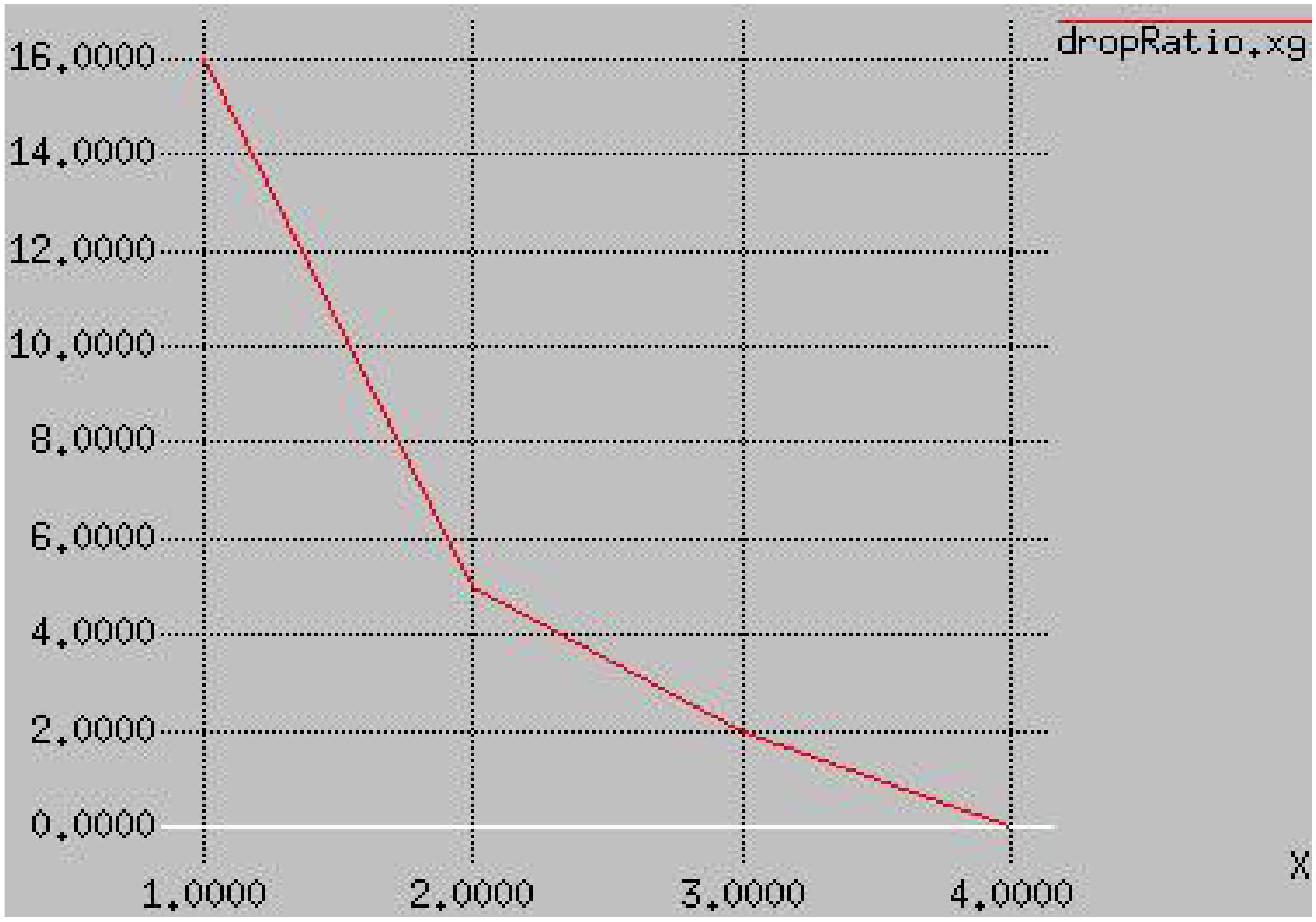

Figure 7: Dropratio:

Phase 3: Practical implementation

1 Introduction

In this section we return to our study of TCP. TCP provides a reliable transport service between two processes running on different hosts. Another extremely important component of TCP is its congestion control mechanism.

As we indicated in the previous section, TCP must use end-to-end congestion control rather than network-assisted congestion control, since the IP layer provides no feedback to the end systems regarding network congestion. Before diving into the details of TCP congestion control, let's first get a high-level view of TCP's congestion control mechanism, as well as the overall goal that TCP strives for when multiple TCP connections must share the bandwidth of a congested link.

An important measure of the performance of a TCP connection is its throughput - the rate at which it transmits data from the sender to the receiver. Clearly, throughput will depend on the value of w. W. If a TCP sender transmits allow segments back-to-back, it must then wait for one round trip time (RTT) until it receives acknowledgments for these segments, at which point it can send w additional segments.

If a connection transmits segments of size MSS bytes every RTT seconds, then the connection's throughput, or transmission rate, is (w*MSS)/RTT bytes per second.
Suppose now that K TCP connections are traversing a link of capacity R. Suppose also that there are no UDP packets flowing over this link, that each TCP connection is transferring a very large amount of data, and that none of these TCP connections traverse any other congested link. Ideally, the window sizes in the TCP connections traversing this link should be such that each connection achieves a throughput of R/K.

More generally, if a connection passes through N links, with link n having transmission rate Rn and supporting a total of Kn TCP connections, then ideally this connection should achieve a rate of Rn/Kn on the nth link. However, this connection's end-to-end average rate cannot exceed the minimum rate achieved at all of the links along the end-to-end path. That is, the end-to-end transmission rate for this connection *is r = minR1/K1,...,RN/KN.* The goal of TCP is to provide this connection with this end-toend rate, r. (In actuality, the formula for r is more complicated, as we should take into account the fact that one or more of the intervening connections may be bottlenecked at some other link that is not on this end-to-end path and hence can not use their bandwidth share, Rn/Kn. In this case, the value of r would be higher than *minR1/K1,...,RN/KN.*)

2 Overview of TCP Congestion Control

In Section we saw that each side of a TCP connection consists of a receive buffer, a send buffer, and several variables (LastByteRead, RcvWin, etc.) The TCP congestion control mechanism has each side of the connection keep track of two additional variables: the congestion window and the threshold. The congestion window, denoted CongWin, imposes an additional constraint on how much traffic a host can send into a connection. Specifically, the amount of unacknowledged data that a host can have within a TCP connection may not exceed the minimum of CongWin and RcvWin, i.e*. LastByteSent - LastByteAcked ¡= minCongWin, RcvWin*. The threshold, which we discuss in detail below, is a variable that effects how CongWin grows. Let us now look at how the congestion window evolves throughout the lifetime of a TCP connection. In order to focus on congestion control (as opposed to flow control), let us assume that the TCP receive buffer is so large that the receive window constraint can be ignored. In this case, the amount of unacknowledged data hat a host can have within a TCP connection is solely limited by CongWin. Further let's assume that a sender has a very large amount of data to send to a receiver.

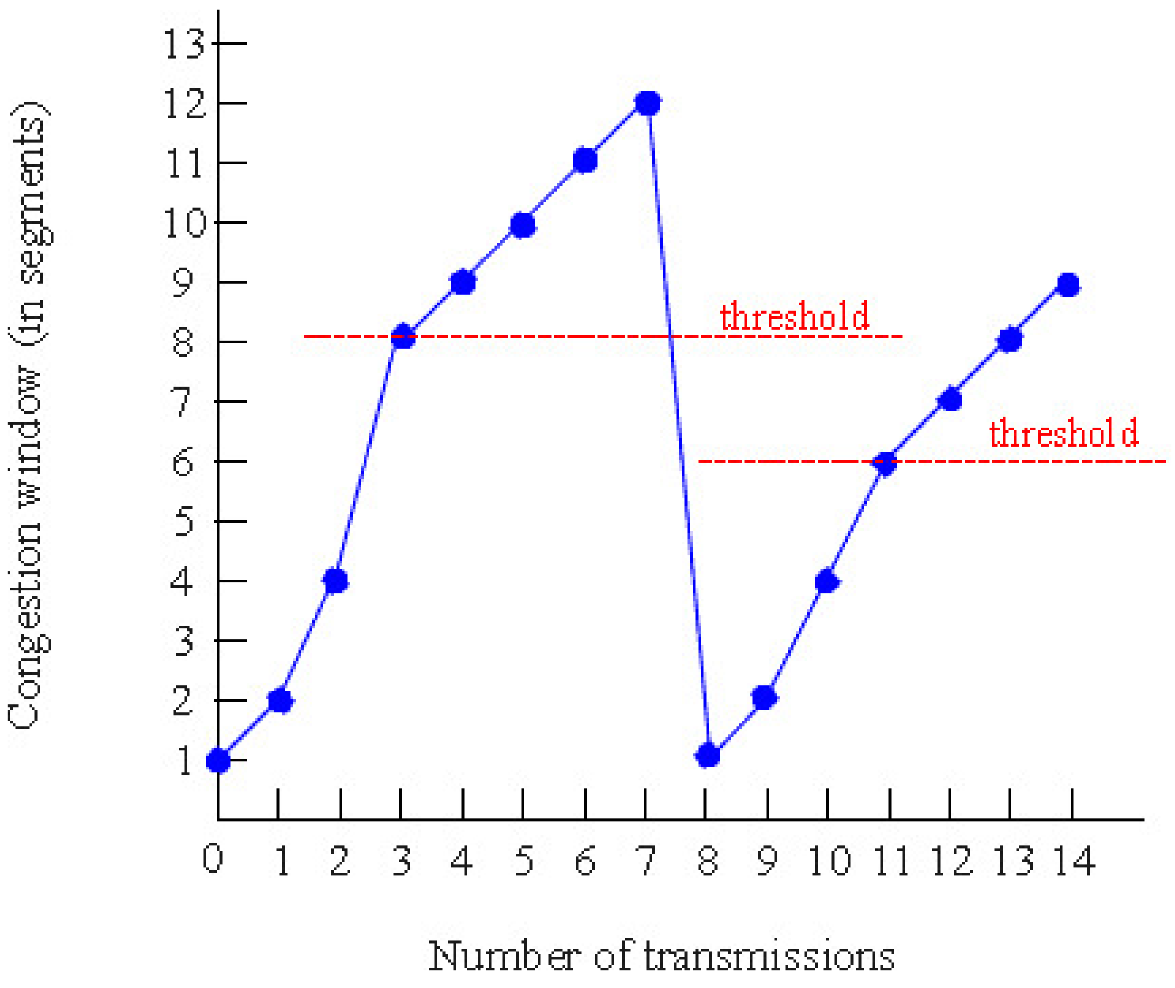

Figure 1: Evolution of TCP's congestion window:

Working of TCP Tahoe, NewReno and Vegas

TCP Tahoe:

Before TCP Tahoe, TCP used go back n model to control network congestion, then Tahoe added slow
start, congestion voidance and fast transmit.

Slow Start:
The congestion window of sender increases exponentially as a result of every acknowledgement. Once a thresh hold is achieved, the congestion window increases linearly i.e. congestion window increases 1 by each RTT and then congestion avoidance begins. A packet loss is a sign of congestion as soon as congestion occurs congestion window is reduced to 1 and start over again (start slow start until reached threshold).

Problem:
Tahoe is not very much efficient because every time a packet is lost it waits for the timeout and then
retransmits the packet. It reduces the size of congestion window to 1 just because of 1 packet loss, this inefficiency cost a lost in high bandwidth delay product links.

TCP Reno:
TCP Reno includes algorithms named Fast Retransmit and Fast Recovery for congestion control.

Fast Retransmit:
When the sender receives three duplicate acknowledgements of a sent packet then sender retransmit it without waiting for the time out.

Fast Recovery:
After retransmission Reno enters into the Fast Recovery. In Fast Recovery after the packet loss, the
congestion window size does not reduce to 1 instead it reduced to half of current window size.

Problem:
TCP Reno is helpful when only 1 packet is lost, in case of multiple packet loss it acts as Tahoe. Then evolved TCP New Reno which is a modification of TCP Reno and deals with multiple packets loss.

TCP New Reno:
TCP New Reno is efficient as compare to Tahoe and Reno. In Reno Fast Recovery exits when three duplicate acknowledgements are received but in New Reno it does not exist until all outstanding data is acknowledged or ends when retransmission timeout occurs. In this way it avoids unnecessary multiple fast retransmit from single window data.

TCP Vegas:

TCP Tahoe,Reno and New Reno detects and controls congestion after congestion occurs but still there
is a better way to overcome congestion problem i.e TCP Vegas.TCP Vegas detects congestion without
causing congestion.
It differs from TCP Reno concerning

- ✓ Slow Start
- ✓ Packet loss detection
- ✓ Detection of available bandwidth

4 TCP TAHOE:

Tahoe refers to the TCP congestion control algorithm which was suggested by Van Jacobson in his paper
[1]. TCP is based on a principle of ‘conservation of packets’, i.e. if the connection is running at the available bandwidth capacity then a packet is not injected into the network unless a packet is taken out as well. TCP implements this principle by using the acknowledgements to clock outgoing packets because an acknowledgement means that a packet was taken off the wire by the receiver. It also maintains a congestion window CWD to reflect the network

capacity [1]. However there are certain issues, which need to be resolved to ensure this equilibrium.
1) Determination of the available bandwidth.
2) Ensuring that equilibrium is maintained.
3) How to react to congestion.

Slow Start:
TCP packet transmissions are clocked by the incoming acknowledgements. However, there is a

problem
when a connection first starts up cause to have acknowledgements you need to have data in the network and to put data in the network you need acknowledgements. To get around this circularity

Tahoe
suggests that whenever a TCP connection starts or re-starts after a packet loss it should go through a procedure called 'slow-start'. The reason for this procedure is that an initial burst might overwhelm the network and the connection might never get started. Slow starts suggests that the sender set the congestion window to 1 and then for each ACK received it increase the CWD by 1. so in the first round trip time (RTT) we send 1 packet, in the second we send 2 and in the third we send 4. Thus we increase
exponentially until we lose a packet which is a sign of congestion. When we encounter congestion.
we decreases our sending rate and we reduce congestion window to one. And start over again. The important thing is that Tahoe detects packet losses by timeouts. In usual implementations, repeated interrupts are expensive so we have coarse grain time-outs which occasionally checks for time outs.
Thus it might be some time before we notice a packet loss and then re-transmit that packet. Congestion Avoidance: For congestion avoidance Tahoe uses 'Additive Increase Multiplicative Decrease'. A packet loss is taken as a sign of congestion and Tahoe saves the half of the current window as a threshold. value. It then set CWD to one and starts slow start until it reaches the threshold value. After that it increments linearly until it encounters a packet loss. Thus it increase it window slowly as it approaches the bandwidth capacity.

Problems:
The problem with Tahoe is that it take a complete timeout interval to detect a packet loss and in fact, in most implementations it takes even longer because of the coarse grain timeout. Also since it doesn't send immediate ACK's, it sends cumulative acknowledgements, there fore it follows a 'go back n approach. Thus every time a packet is lost it waits for a timeout and the pipeline is emptied. This offers a major cost in high band-width delay product links.

5 TCP RENO:

This Reno retains the basic principle of Tahoe, such as slow starts and the coarse grain re-transmit timer. However it adds some intelligence over it so that lost packets are detected earlier and the pipeline is not emptied every time a packet is lost. Reno requires that we receive immediate acknowledgement whenever a segment is received. The logic behind this is that whenever we receive a duplicate acknowledgment, then his duplicate acknowledgment could have been received if the next segment in sequence expected, has been delayed in the network and the segments reached there out of order or else that the packet is lost. If we receive a number of duplicate acknowledgements then that means that sufficient time has passed and even if the segment had taken a longer path, it should have gotten to the receiver by now. There is a very high probability that it was lost. So Reno suggest an algorithm called 'Fast Retransmit'. Whenever we receive 3 duplicate ACK's we take it as a sign that the segment was lost, so we re-transmit the segment without waiting for timeout. Thus we manage to re-transmit the segment with the pipe almost full.

Another modification that RENO makes is in that after a packet loss, it does not reduce the congestion window to 1. Since this empties the pipe. It enters into a algorithm which we call 'Fast-ReTransmit'[2].
The basic algorithm is presented as under:

1)Each time we receive 3 duplicate ACK's we take that to mean that the segment was lost and we re-transmit the segment immediately and enter 'FastRecovery'
2)Set SSthresh to half the current window size and also set CWD to the same value.
3)For each duplicate ACK receive increase CWD by one. If the increase CWD is greater than the amount of data in the pipe then transmit a new segment else wait. If there are 'w' segments in the window and one is lost, the we will receive (w-1) duplicate ACK's. Since CWD is reduced to W/2, therefore half a window of data is acknowledged before we can send a new segment. Once we retransmit a segment, we would have to wait for atlease one RTT before we would receive a fresh acknowledgement. Whenever we receive a fresh ACK we reduce the CWND to SSthresh.

If we had previously received (w-1) duplicate ACK's then at this point we should have exactly w/2 segments in the pipe which is equal to what we set the CWND to be at the end of fast recovery. Thus we don't empty the pipe, we just reduce the flow. We continue with congestion avoidance phase of Tahoe after that.

Problems:
Reno perform very well over TCP when the packet losses are small. But when we have multiple packet losses in one window then RENO doesn't perform too well and it's performance is almost the same as Tahoe under conditions of high packet loss. The reason is that it can only detect a single packet losses. If there is multiple packet drop then the first info about the packet loss comes when we receive the duplicate ACK's. But the information about the second packet which was lost will come only after the ACK for the retransmitted first segment reaches the sender after one RTT. Also it is possible that the CWD is reduced twice for packet losses which occurred in one window. Suppose we send packets 1,2,3,4,5,6,7,8,9 in that order. Suppose packets 1, and 2 are lost. The ACK's generated by 2,4,5 will cause the re-transmission of 1 and the CWD is reduced to 7. Then when we receive ACK for 6,7,8,9 our CWD is sufficiently large to allow to us to send 10,11. When the re-transmitted segment 1 reaches the receiver we get a fresh ACK and we exit fast-recovery and set CWD to 4. Then we get two more ACK's for 2(due to 10,11) so once again we enter fast-retransmit and re-transmit 2 and then enter fast recovery. Thus when we exit fast recovery for the second time our window size is set to 2. Thus we reduced our window size twice for packets lost in one window. Another problem is that if the widow is very small when the loss occurs then we would never receive enough duplicate acknowledgements for a faster transmit and we would have to wait for a coarse grained timeout. Thus is cannot effectively detect multiple packet losses.

6 NEW-RENO:

New RENO is a slight modification over TCP-RENO. It is able to detect multiple packet losses and thus is much more efficient that RENO in the event of multiple packet losses. Like Reno, New-Reno also enters into fast-retransmit when it receives multiple duplicate packets, however it differs from RENO in that it doesn't exit fast-recovery until all the data which was out standing at the time it entered fast recovery is acknowledged. Thus it overcomes the problem faced by Reno of reducing the CWD multiples times. The fast-transmit phase is the same as in Reno. The difference in the fastrecovery phase which allows for multiple re-transmissions in new-Reno. Whenever new-Reno enters fastrecovery it notes the maximums segment which is outstanding. The fast-recovery phase proceeds as in Reno, however when a fresh ACK is received then there are two cases: If it ACK's all the segments which were outstanding when we entered fastrecovery then it exits fast recovery and sets CWD to ssthresh and continues congestion avoidance like Tahoe. If the ACK is a partial ACK then it deduces that the next segment in line was lost and it retransmits that segment and sets the number of duplicate ACKS received to zero. It exits Fast recovery when all the data in the window is acknowledged[3].

Problems:
New-Reno suffers from the fact that its take one RTT to detect each packet loss. When the ACK for the first retransmitted segment is received only then can we deduce which other segment was lost.

7 VEGAS:

Vegas is a TCP implementation which is a modification of Reno. It builds on the fact that proactive measure to encounter congestion are much more efficient than reactive ones. It tried to get around the problem of coarse grain timeouts by suggesting an algorithm which checks for timeouts at a very efficient schedule. Also it overcomes the problem of requiring enough duplicate acknowledgements to detect a packet loss, and it also suggest a modified slow start algorithm which prevent it from congesting the network. It does not depend solely on packet loss as a sign of congestion. It detects congestion before the packet losses occur. However it still retains the other mechanism of Reno and Tahoe, and a packet loss can still be detected by the coarse grain timeout of the other mechanisms fail. The three major changes induced by Vegas are:

New Re-Transmission Mechanism:

Vegas extends on the re-transmission mechanism of Reno. It keeps track of when each segment was sent and it also calculates an estimate of the RTT by keeping track of how long it takes for the acknowledgment to get back. Whenever a duplicate acknowledgement is received it checks to see if the (current timesegment transmission time)¿ RTT estimate; if it is then it immediately retransmits the segment without waiting for 3 duplicate acknowledgements or a coarse timeout[6]. Thus it gets around the problem faced by Reno of not being able to detect lost packets when it had a small window and it didn't receive enough duplicate Ack's. To catch any other segments that may have been lost prior to the retransmission, when a non duplicate acknowledgment is received, if it is the first or second one after a fresh acknowledgement then it again checks the timeout values and if the segment time since it was sent exceeds the timeout value then it re-transmits the segment without waiting for a duplicate acknowledgment[6]. Thus in this way Vegas can detect multipple packet losses. Also it only reduces its window if the re-transmitted segment was sent after the last decrease. Thus it also overcome Reno's shortcoming of reducing the congestion window multiple time when multiple packets are lost.

Congestion avoidance:
TCP Vegas is different from all the other implementation in its behavior during congestion avoidance.

It does not use the loss of segment to signal that there is congestion. It determines congestion by a decrease in sending rate as compared to the expected rate, as result of large queues building up in the routers. It uses a variation of Wang and Crowcroft;s Tri-S scheme. The details can found in [6].

Thus whenever the calculated rate is too far away from the expected rate it increases transmissions to make use of the available bandwidth, whenever the calculated rate comes too close to the expected value it decreases its transmission to prevent over saturating the bandwidth. Thus Vegas combats congestion quite effectively and doesn't waste bandwidth by transmitting at too high a data rate and creating congestion and then cutting back, which the other algorithms do.

Modified Slow-start:

TCP Vegas differs from the other algorithms during it's slow-start phase. The reason for this modification is that when a connection first starts it has no idea of the available bandwidth and it is possible that during exponential increase it over shoots the bandwidth by a big amount and thus induces congestion. To this end Vegas increases exponentially only every other RTT, between that it calculates the actual sending through put to the expected and when the difference goes above a certain threshold it exits slow start and enters the congestion avoidance phase.

8 A TCP CUBIC Implementation

TCP is the dominant transport protocol on the Internet, accounting for the vast majority of the number of flows and bytes transferred. To deal with network congestion, TCP uses congestion control algorithms to vary its transmission rate and manage on lost packets. There have been numerous variants proposed for TCP [5], but the current default used in the Linux kernel is CUBIC [4]. Along with its predecessor BIC [7], CUBIC and the Microsoft Windows Compound [6] have been the dominant three TCP algorithms for years. A 2011 study [8] of about 30,000Web servers show about 25use TCP CUBIC, 20TCP Compound. An important step in the development of these congestion control algorithms, as well as for developing new TCP variants and other aspects of computer networking, is simulation. This paper presents the design and implementation of CU- BIC in ns-3. In order to match the ns-3 TCP CUBIC with the most widely deployed Internet version, our implementation focuses on the CUBIC algorithm found in Linux. How- ever, the original CUBIC algorithm introduced by Ha et al. [4] differs somewhat from the current Linux kernel implementation. To deal with these discrepancies, our implementation follows the basic details of the CUBIC algorithm as much as possible, using the Linux-specific details provided in the kernel code when appropriate. Verification of our ns-3 CUBIC implementation is provided via a demonstration of the functioning ns-3 code, and validation is provided by com- paring performance for TCP CUBIC in Linux TCP CUBIC in ns-3 with our ns-3 implementation. A final comparison of TCP CUBIC to TCP New Reno, the default TCP for ns-3, is provided to highlight the differences in performance between these two TCP variants.

The rest of this paper is organized as follows: Section 2 reviews basic TCP congestion control concepts; Section 3 briefly discusses prior related work; Section 4 gives an overview of CUBIC; Section 5 describes the CUBIC implementation in Linux; Section 6 presents our ns-3 implementation of CU- BIC; Section 7 details the verification and validation of our implementation; and Section 8 summarizes our conclusions and presents possible future work.

CUBIC BASICS

New technologies have allowed network architects to in- crease network link capacities, accommodating more traffic and faster transmission rates. Unfortunately, legacy TCP algorithms like Reno and New Reno are not well suited for large capacity links. Since the congestion window in older TCP variants grows by one each RTT, it takes too long for a TCP flow to expand to meet the capacities available on some of today's Internet links. An example given by Ha et al. [4] is that traditional TCP, using 1250 byte packets on a 10 Gb/s link with a

100 ms RTT, requires 1.4 hours just to reach half the bandwidth delay product. To more efficiently utilize the capacities of modern Inter- net links, a new algorithm was needed to grow the TCP congestion control window (cwnd) faster. This resulted in the Binary Increase Congestion (BIC) congestion control algorithm for TCP [7]. In TCP BIC, a binary search algorithm is used to grow cwnd from its current position to the mid-point of the cwnd position when the last loss occurred, called Wmax. When cwnd is far away from Wmax, cwnd grows quickly, but as cwnd approaches Wmax, cwnd grows slowly. After passing Wmax, cwnd continues to grow slowly for a short time before continuing a more rapid growth until the next loss event occurs.

The pattern of cwnd growth when graphed over time creates a curved line with a concave region followed by convex region. The BIC algorithm described by Ha et al. [4] for managing the concave and convex regions of cwnd growth is complicated. Since BIC depends on the RTT to change cwnd, controlling growth is problematic when a network has a low RTT. Each time an ACK for a packet is received, BIC grows cwnd by only small increments to compensate for the short RTT. This strategy magnifies the inherent unfairness experienced by TCP flows with long RTTs. CUBIC reduces this complexity by using a cubic function, an odd order polynomial which naturally handles concave and convex growth. Being independent of RTT, CUBIC relies instead on the time between consecutive congestion events to adjust cwnd.

Hence, CUBIC maintains the same concave and convex patterns found in BIC, shown in Figure 2. The horizontal axis represents time and the vertical axis represents cwnd. The left side of the figure shows the concave region where the rate of cwnd growth slows as it approaches the position where it last lost packets, midway along the line. After flat growth for some time around this mid-point, the right side shows the convex region where cwnd grows more rapidly as TCP CUBIC probes for a new loss region. Like BIC, CUBIC keeps track of Wmax when a loss event occurs.

To keep track of time CUBIC uses the variables epoch start, K and t. epoch start is the time when the first new TCP acknowledgment arrives after the loss event. At this point, cwnd is at its lowest in the curve and CUBIC needs to start growing cwnd to Wmax. The variable K is the time when cwnd should reach Wmax. The variable t keeps track of how long it has been since epoch start. With these three variables, CUBIC tracks where the cwnd growth should be in comparison to Wmax. When just starting cwnd growth, t is small as little time has passed since epoch start. This causes large updates to cwnd. As t grows larger it starts to approach the value K and the increases to cwnd become smaller. After t grows larger than K, CUBIC changes from convex to concave where the updates to cwnd start small but increase the further t grows from K.

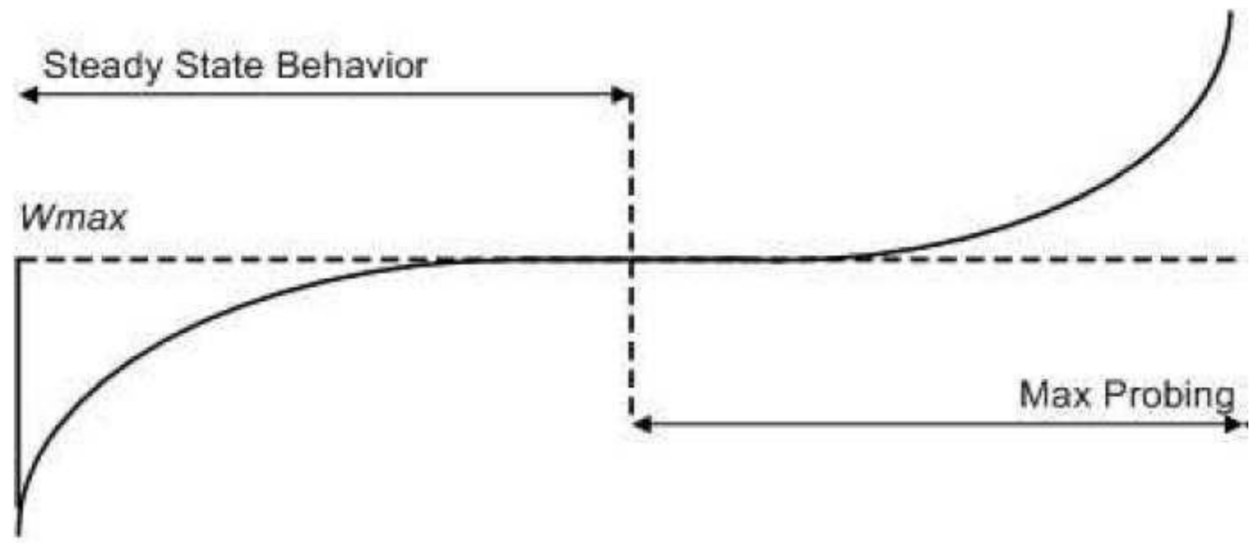

Figure 2: CUBIC growth function [4]:

Linux Constants and Variables

The Linux implementation of CUBIC has the same constants and variables used in the original CUBIC paper [4], but with some notable differences. For example, when CU- BIC was implemented in Linux adjustments had to be made for tracking time. CUBIC uses TCP timestamps, represented in the Linux kernel as jiffies.

The actual amount of time in 1 jiffy ranges from 1 ms to 10 ms, depending upon the computer's system clock – not consistent across platforms. Other differences in hardware architectures mean that the Linux implementation of CUBIC needs additional parameters and different values for constants than those presented in the original CUBIC paper [4]. The following is a list of the main constants and variables used in Linux and in our ns-3 implementation of CUBIC. big scale – Constant scaling factor used to set the Linux version of CUBIC's C and cube rtt scale. BICTCP BETA SCALE – Constant scaling factor used with the CUBIC constant b.

C – Constant used for comparing time and congestion window size in the CUBIC equations. cube rtt scale – Scaling factor used in place of the CU- BIC constant C when calculating recommended CU- BIC update, but not when calculating the value of K. BICTCP HZ – Constant used to convert units of time. epoch start – Time of last loss event – from this point, time is tracked to find values of t and K. cnt – Count set by CUBIC to indicate when next cwnd growth can occur. cwnd cnt – Global variable incremented on every ACK received. Once *cwnd cnt == cnt,* cwnd is incremented and cwnd cnt is reset.

b– Constant used as a multiplicative factor for decreasing cwnd during loss events.t – Elapsed time
since last loss event.
W max – cwnd position at last loss event.
CUBIC Packet Loss

The method bictcp recalc ssthresh in the Linux implementation of CUBIC handles lowering cwnd and resetting ssthresh. epoch start is reset to 0 since the congestion window is being lowered. If *CUBIC is in fast convergence, Equation 7 is used to calculate Wmax.*
Wmax = cwnd (BICTCP BETA SCALE +) 2 BICTCP BETA SCALE (7)
However, when fast convergence is not in use, Wmax is set to cwnd.
The reduction of cwnd and ssthresh uses Equation 8 (note the max() function ensures cwnd and ssthresh
are at least 2): cwnd = ssthresh = max(cwnd
BICTCP BETA SCALE , 2) (8)

9 Best possible mechanism: BBR

Into this mix comes a more recent TCP delay-controlled TCP flow control algorithm from Google, called BBR. BBR is very similar to TCP Vegas, in that it is attempting to operate the TCP session at the point of onset of queuing at the path bottleneck. The specific issues being addressed by BBR is that the determination of both the underlying bottleneck available bandwidth and path RTT is influenced by a number of factors in addition to the data being passed through the network for this particular flow, and once BBR has determined its sustainable capacity for the flow, it attempts to actively defend it in order to prevent it from being crowded out by the concurrent operation of conventional AIMD protocols.

Like TCP Vegas, BBR calculates a continuous estimate of the flow's RTT and the flow's sustainable bottleneck capacity. The RTT is the minimum of all RTT measurements over some time window that is described as "tens of seconds to minutes". The bottleneck capacity is the maximum data delivery rate to the receiver, as measured by the correlation of the data stream to the ACK stream, over a sliding time window of the most recent six to 10 RTT intervals. These values of RTT and bottleneck bandwidth are independently managed, in that either can change without necessarily impacting on the other. par For every sent packet, BBR marks whether the data packet is part of a stream or whether the application stream has paused, in which case the data is marked as “application limited”. Importantly, packets to be sent are paced at the estimated bottleneck rate, which is intended to avoid network queuing that would otherwise be encountered when the network performs rate adaptation at the bottleneck point.

The intended operational model here is that the sender is passing packets into the network at a rate that is anticipated not to encounter queuing within the entire path. This is a significant contrast to protocols such as Reno, which tends to send packet bursts at the epoch of the RTT and relies on the network’s queues to perform rate adaptation in the interior of the network if the burst sending rate is higher than the bottleneck capacity.

For every received ACK, the BBR sender checks if the original sent data was application limited. If
not, the sender incorporates the calculation of the path RTT and the path bandwidth into the current flow estimates.

The flow adaptation behavior in BBR differs markedly from TCP Vegas, however. BBR will periodically spend one RTT interval deliberately sending at a rate that is a multiple of the bandwidth delay product of the network path. This multiple is 1.25, so the higher rate is not aggressively so, but enough over an RTT interval to push a fully occupied link into a queueing state. If the available bottleneck bandwidth has not changed, then the increased sending rate will cause a queue to form at the bottleneck. This will cause the ACK signalling to reveal an increased RTT, but the bottleneck bandwidth estimate should be unaltered. If this is the case, then the sender will subsequently send at a compensating reduced sending rate for an RTT interval, allowing the bottleneck queue to drain. If the available bottleneck bandwidth estimate has increased because of this probe, then the sender will operate according to this new bottleneck bandwidth estimate.

Successive probe operations will continue to increase the sending rate by the same gain factor until the estimated bottleneck bandwidth no longer changes because of these probes. Because this probe gain factor is a multiple of the bandwidth delay product, BBR’s overall adaptation to increased bandwidth on the path is exponential rather than the linear adaptation used by TCP Vegas.

This regular probing of the path to reveal any changes in the path’s characteristics is a technique borrowed from the drop-based flow control algorithms. Informally, the control algorithm is placing increased flow pressure on the path for an RTT interval by raising the data sending rate by a factor of 25 There is also the possibility that this increased flow pressure will cause other concurrent flows to back off, and in that case BBR will react quickly to occupy this resource by sending a steady rate of packets equal to the new bottleneck bandwidth estimate. Session startup is relatively challenging, and the relevant observation here is that on today’s Internet link bandwidths span 12 orders of magnitude, and the startup procedure must rapidly converge to the available bandwidth irrespective of its capacity. With BBR, the sending rate doubles each RTT, which implies that the bottleneck bandwidth is encountered within log2 RTT intervals.

This is similar to the rate doubling used by Reno, but here the end of this phase of the algorithm is within an RTT of the onset of queuing, rather than Reno’s condition of within one RTT of the saturation of the queue and the onset of packet drop.

At the point where the estimated RTT starts to increase, BBR assumes that it now has filled the network queues, so it keeps this bandwidth estimate and drains the network queues by backing off the sending rate by the same gain factor for one RTT. Now the sender has estimates for the RTT and the bottleneck bandwidth, so it also has the link Bandwidth Delay Product (BDP). Once the server has just this quantity of data unacknowledged, then it will resume sending at the estimated bottleneck bandwidth rate. The overall profile of BBR behavior, as compared to Reno and CUBIC, is shown in Figure.

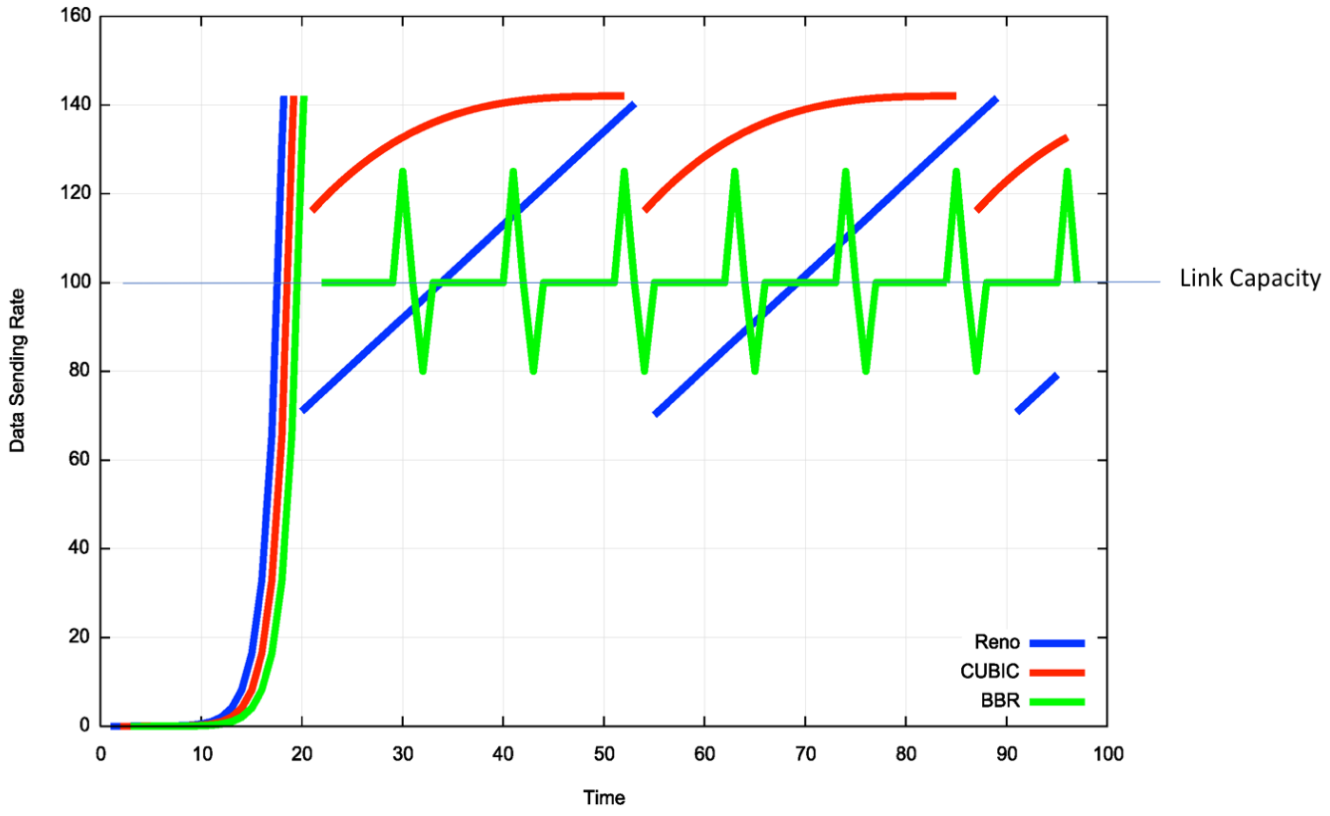

Figure 3: Comparison of model behaviors of Reno, CUBIC and BBR:

Sharing
The noted problem with TCP Vegas was that it ‘ceded’ flow space to concurrent drop-based TCP flow control algorithms. When another session was using the same bottleneck link, then when Vegas backed off its sending rate, the drop-based TCP would in effect occupy the released space.

Because drop-based TCP sessions have the bottleneck queue occupied more than half the time, Vegas would continue to drop its sending rate, passing the freed bandwidth to the drop-based TCP session(s). BBR appears to have been designed to avoid this form of behaviour. The reason BBR will claim its “fair share” is the periodic probing at an elevated sending rate.

This behaviour will “push” against the concurrent drop-based TCP sessions and allow BBR to stabilize on its fair share bottleneck bandwidth
estimate. However, this works effectively when the internal network buffers are sized according to the delay bandwidth product of the link they are driving.

If the queues are over-provisioned, the BBR probe phase may not create sufficient pressure against the drop-based TCP sessions that occupy the queue and BBR might not be able to make an impact on these sessions, and may risk losing its fair share of the bottleneck bandwidth. On the other hand, there is the distinct risk that if BBR overestimates the RTT it will stabilise at a level that has a permanent queue occupancy. In this case, it may starve the drop-based flow algorithms and crowd them out. Google report on experiments that show that concurrent BBR and CUBIC flows will approximately equilibrate, with CUBIC obtaining a somewhat greater share of the available resource, but not outrageously so. It points to a reasonable conclusion that BBR can coexist in a RENO/CUBIC TCP world without either losing or completely dominating all other TCP flows.

Our limited experiments to date point to a somewhat different conclusion. The experiment was of the form of a 1:1 test with concurrent single CUBIC and Reno flows passed over a 15.2ms RTT uncongested 10Gbps circuit (Figure). CUBIC was started initially, and at the 20-second point a BBR session was started between the same two endpoints. BBR's initial start algorithm pushes CUBIC back to a point where it appears unable to re-establish a fair share against BBR. This is a somewhat unexpected result, but may be illustrative of an outcome where the internal buffer sizes are far lower than the delay bandwidth product of the bottleneck circuit.

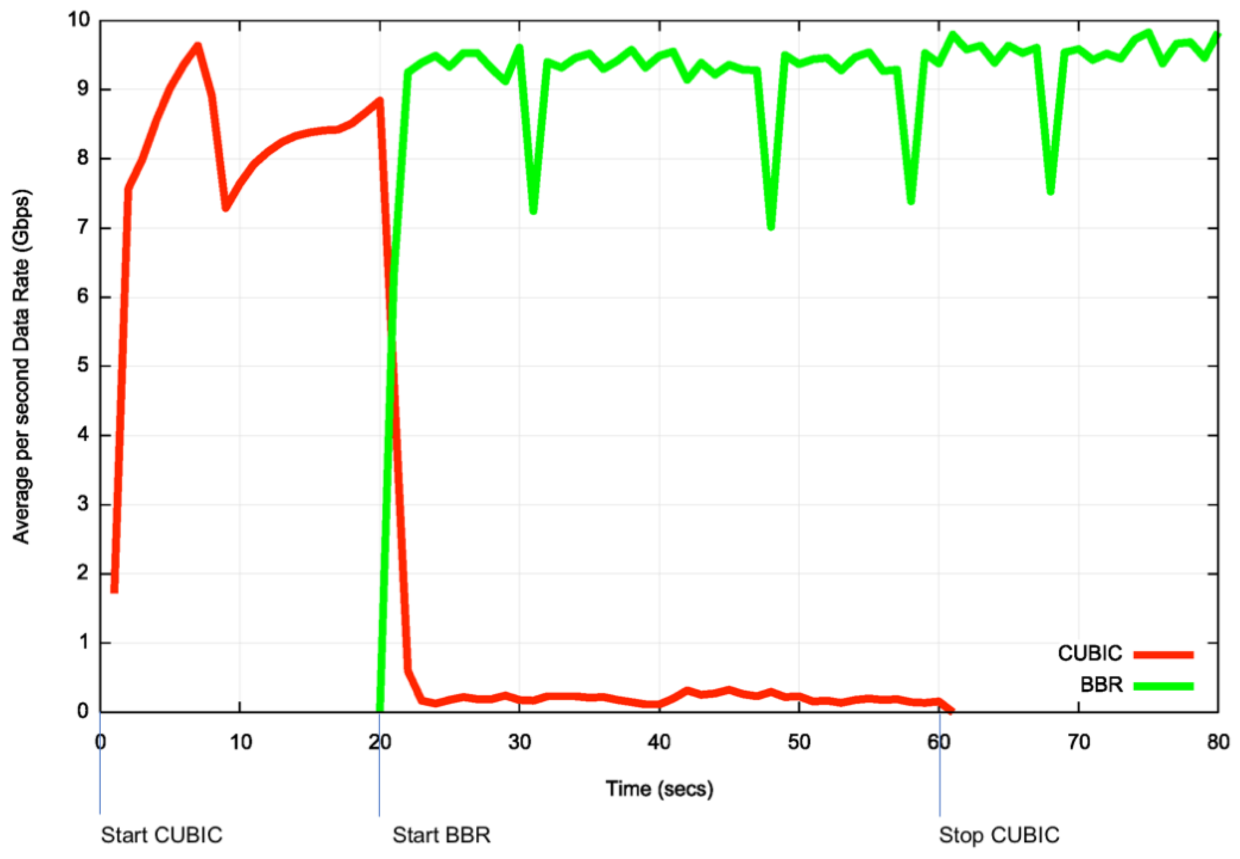

Figure 4: Test of CUBIC vs BBR

The second experiment used a 298ms RTT path across a large span of the Internet between end nodes located in Germany and Australia. This is a standard Internet path across commercial ISPs, as one would expect to encounter in the Internet. These are not the only two streams that exist on the 16 forward and 21 reverse direction component links in this extended path, so the two TCP sessions are not only vying with each other for network resources on all of these links, but variously competing with cross traffic on each component link.

Again, BBR appears to be more successful in claiming path resources, and defending them against other flows for the duration of the BBR session. This is perhaps a less surprising outcome,

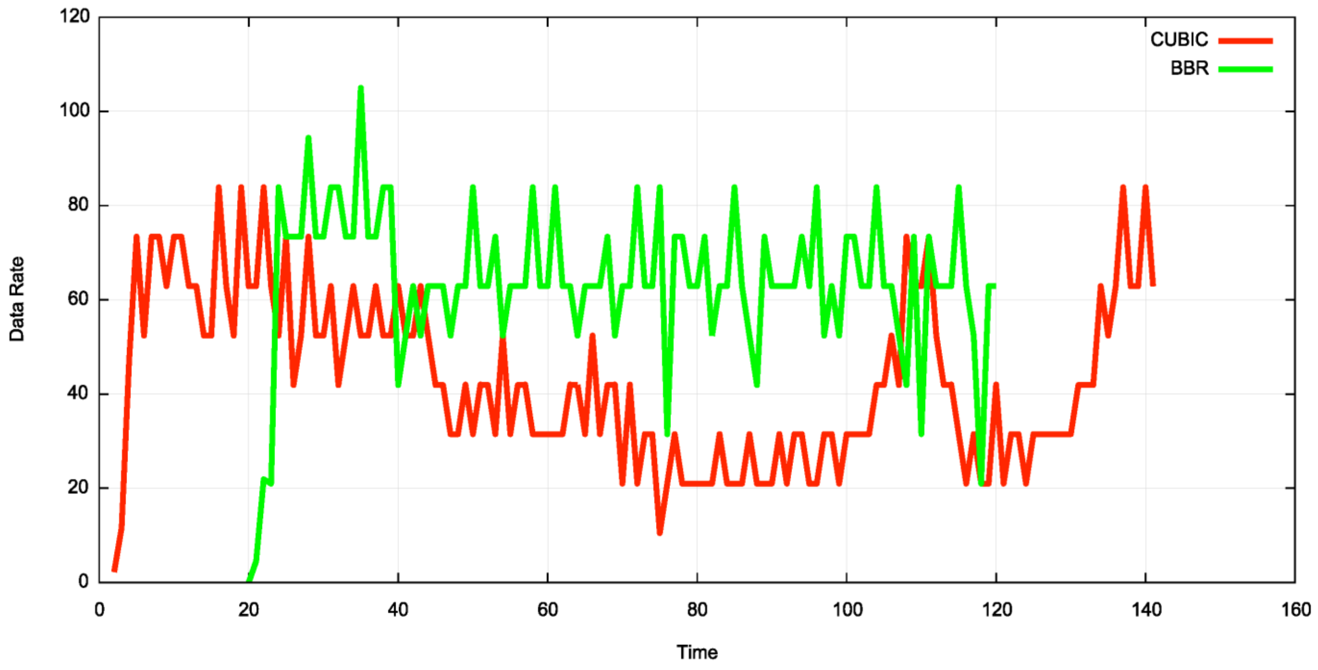

Figure 5: Second Test of CUBIC vs BBR

in that the extended RTT apparently posed some issues for CUBIC, while BBR was able to maintain its
path bandwidth estimate in this period. This results point to some level of co-existence between BBR and CUBIC, but the outcome appears be biased to BBR gaining the greater share of the path capacity.

10 CONCLUSIONS

The current ns-3 TCP implementations lack the newer TCP congestion control algorithms used by most compute- 6TCP NewReno is currently the default in ns-3. ers today [8]. For realistic network simulations that more closely model Internet behavior, additional implementations of TCP congestion control algorithms are needed in ns-3. This paper introduces a TCP CUBIC implementation for ns-3,7 based
on the original CUBIC algorithm [4] while using many of the implementation details found in Linux [1].

Evaluation of our ns-3 CUBIC verifies the implementation, and comparisons with Linux measurements provide validation. In particular, the ns-3 implementation produces a CUBIC-like cwnd growth pattern which is critical for modern TCP implementations to effectively utilize available network capacities. Simulated performance compared with TCP NewReno implies a significant bitrate advantage gained by employing TCP CUBIC. Existing issues with our ns-3 CUBIC include lack of sup- port for TCP timestamps in the base TCP implementation of ns-3, making it unable to exactly match CUBIC simple- mentation in the Linux kernel or ns-2. Future work includes implementing TCP timestamps in ns-3 and running more validation tests involving multiple flows and different net- work conditions, such as low and high congestion as well as a wide range of capacities and RTTs, that stress TCP functionality. Additionally, native implementations of other TCP protocols, such as TCP BIC and TCP Compound, with corresponding verification and validation, could be under- taken of each algorithm.