Evaluating CAIA delay gradient as a Less-than-best-effort congestion control in Linux

Tor Christian Croff Tangenes

15th November 2016
Abstract

The Transmission Control Protocol (TCP) together with the Internet Protocol (IP) are the most widespread protocols used in the Internet. TCP controls data transmission between end-hosts in a network. One of its main components the Congestion Control (CC) which regulates the rate an end-host can inject packets into the network. Most applications, if they require it, wish to utilize as much available bit rate as possible. Other applications may have specific demands to timeliness. Some applications however do not have such requirements, other than the guarantee that the data eventually will be sent and received reliably. The Less-than-best-effort (LBE) service class is meant to cater for these types of applications, by preventing them from becoming disruptive towards standard best effort flows. Recently CAIA delay gradients (CDG), a candidate for providing a CC-based LBE service has been implemented in the Linux kernel. CDG has adjustable parameter that can control the degree of its aggressiveness. The aim of this thesis is to evaluate how tuning CDG parameters can provide a LBE service when running against the standard Linux congestion control CUBIC. To do this, we have set up a Linux based emulation test-bed where we run CDG and tune its parameters to see how it responds to CUBIC Best-effort flow. We have concluded that CDG is indeed able to provide such a service in a Linux environment, and that by setting its parameters to specific values, can guarantee that it will on average stay within a certain fraction of the available capacity.
List of selected abbreviations

**BE service** A network service, while not giving any guarantees, gives users what it can without discrimination.

**LBE service** Less-than-best effort is a term used for services providing lower quality in terms of bit rate and delay to users, in order to have little impact on standard Best-effort services.

**TCP** Transmission Control Protocol. One of the most important protocols in the Internet. Facilities machine-to-machine communication using Internet as a medium.

**IP** The Internet Protocol which is the main protocol in the Internet responsible for making machines addressable and reachable through IP addresses.

**CC** Congestion control is the main mechanism preventing hosts from over-utilizing network resources, as-well as allocation of capacity among peers at a host-level. Classical CC algorithms react to packet loss as a sign of congestion.

**ACK** Short for acknowledgement. A packet confirming to a sender host the reception of a packet with a certain sequence number.

**RTT** Short for Round trip time. The time it takes from when a packet is sent from a sender host to the time it takes for the sender to receive an ACK from the receiver.

**BRTT** Base RTT, the RTT of a path in absence of any queuing delay (no traffic).

**OWD** One-way delay, the time it takes for a packet from when it is sent from a sender host to when it arrives at the receiving host. Not the same as half and RTT because paths may be asymmetric.

**FTP** An application layer protocol using TCP to transfer files between hosts.

**LEDBAT** A delay-based CC algorithm, it tries to keep forward queuing delay below a threshold value.

**CDG** A type of congestion control algorithm using change in minimum and maximum RTT to infer congestion at a bottleneck. Its use of gradients distinguishes it from delay-threshold algorithms like TCP-Vegas and LEDBAT.

**SYN** A packet with its SYN flag set, used when TCP initiates a connection.

**FIN** A packet signaling the sender has terminated or wishes to terminate the connection.

**RTO** A timer used by TCP, which when expires triggers a retransmission of the last unacknowledged packet.
**BDP** Bandwidth-delay product. Calculated by the multiplying delay and bit rate. The BDP is a measure for how many bits can occupy a channel or path at once.

**AIMD** Additive Increase Multiplicative Decrease. The policy used by classic TCP to adjust its cwnd when additively increasing cwnd to probe for more capacity, or reducing cwnd by a factor in response to congestion indication.

**CUBIC** A high speed TCP variant optimized for utilizing paths with high BDP. As of today it is the default CC.
# Contents

1 Introduction 1  
1.1 Motivation ................................................. 1  
1.2 Related work ............................................... 2  
1.3 Problem statement ......................................... 2  
1.4 Outline ...................................................... 2  

2 Background 5  
2.1 IP networks .................................................. 5  
2.1.1 The Internet Protocol Suite .............................. 5  
2.1.2 The Internet Protocol ................................... 6  
2.1.3 Network buffers ......................................... 6  
2.1.4 Network queues sizing .................................. 6  
2.1.5 Internet bottlenecks ..................................... 8  
2.1.6 Unreliability of the Internet ............................ 8  
2.2 Transmission Control Protocol ............................ 9  
2.2.1 TCP overview ........................................... 9  
2.2.2 TCP basics ................................................. 10  
2.2.3 Lifetime of a TCP connection ........................... 11  
2.2.4 TCP Congestion Control ................................ 12  
2.3 TCP Fairness .................................................... 17  
2.3.1 Max-min fairness ........................................ 17  
2.3.2 Jain's Fairness Index .................................... 18  
2.4 Linux congestion control .................................. 19  
2.4.1 TCP CUBIC .............................................. 19  
2.4.2 Linux improvements to SS .............................. 20  
2.4.3 Linux improvements to FR .............................. 21  

3 Delay-based Congestion Control 23  
3.1 CARD ......................................................... 23  
3.2 TCP Vegas ................................................... 23  
3.3 TCP Nice and TCP Low Priority ......................... 24  
3.4 LEDBAT ....................................................... 24  
3.4.1 Problems with delay-threshold CC regarding LBE .... 25  
3.5 CAIA delay gradients ....................................... 26  
3.5.1 The CDG algorithm ...................................... 26  
3.5.2 CDG’s LBE capabilities ................................. 27  
3.5.3 Linux implementation ................................... 28
## A Plots of the results

<table>
<thead>
<tr>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 CUBIC / CDG Time series (CUBIC starts first)</td>
<td>79</td>
</tr>
<tr>
<td>A.2 CUBIC / CDG Time series (CDG starts first)</td>
<td>140</td>
</tr>
<tr>
<td>A.3 CDG LBE Homogeneous Capacity Sharing</td>
<td>201</td>
</tr>
</tbody>
</table>
List of Figures

2.2 TCP segment header ............................................. 10
2.3 Phases of a TCP connection ................................. 13
2.4 TCP Tahoe and TCP Reno response to congestion. Tahoe resets $cwnd$ and goes directly to SS, while Reno sets $cwnd$ to half the flight size and enters FR on reception of three dupacks. The $sshthresh$ line is set after congestion detection at RTT round 7 .............................................. 16
2.5 A max-min allocation between flows A, B, C and D. Flow B, C and share capacity equally between router R1 and R2. Flow B splits up from C and D at R2, and takes 1/3 of the capacity between R5 and R6 while Flow A gets the remaining 2/3 of capacity .................................................. 18
2.6 The $cwnd$ growth function of CUBIC. It first operates in the ‘steady-state’ region initially ramping swiftly up towards it target value. Later it tries to probe for more capacity in the ‘max probing’ region ........................................... 20
3.1 Illustrations of LEDBAT’s base delay estimation problems .. 31
4.1 The testbed topology ............................................. 34
5.1 CUBIC / CDG ...................................................... 44
5.5 Surface plot of CUBIC share of capacity against G and RTT. See figure 5.8a, to figure 5.10b ........................................ 49
5.11 Surface plot of CUBIC share of capacity against G and RTT . 56
5.13 ........................................................................ 58
5.14 ........................................................................ 58
5.15 ........................................................................ 59
5.16 ........................................................................ 59
5.17 ........................................................................ 60
5.25 Three CDG flows with G = 0.1 and Beta = 0.8 ................. 68
5.26 Three CDG flows with G = 0.1 and Beta= 1 ................. 69
A.133 ................................................................. 214
A.136 ................................................................. 217
A.137 ................................................................. 218
A.138 ................................................................. 218
A.139 ................................................................. 219
A.140 ................................................................. 219
Chapter 1

Introduction

1.1 Motivation

The Transmission Control Protocol (TCP) employs services provided by the Internet Protocol (IP) that operates in hosts and network devices to transmit and receive data from host-to-host. IP makes no guarantee as to how data it transmits is treated or whether it will arrive to the recipient at all. However it will do what it can to deliver data from one destination to another. This type of service provided by IP is termed Best Effort (BE).

A downside of having BE services for all applications is that it does not consider how much network resources the application actually needs. From a user perspective one might wish for an application $A$ to be prioritized over another application $B$ by having application $B$ use a service that provides less than what BE provides $A$. As an example let application $A$ be a video streaming application and $B$ be a system update manager. Let say the system update is rather large but not critical, and takes several hours to download. While the user wants the update to be downloaded and installed eventually, she may not care if it takes some extra hours if it means she can stream video simultaneously, unaffected by the download. The same may apply to automated system backups, or bulk data transfer applications like FTP. Consequently, the bit rate (we use this term instead of the common ‘bandwidth’ slang, which may have several meanings) requirement can be lowered for these application. A service called Less-than-best-effort (LBE) has been defined for this niche of applications. As the name suggests this service class makes few guarantees to the application with regards to the quality of its service (e.g. what bit rate is allocated to the application).

A proposal to use the Differentiated Services (DiffServ) architecture, which classifies traffic according to their types of service, to cater for LBE applications has been proposed in [16]. Alternatively LBE could be implemented in CC. LBE CC uses IP the same way as regular BE CC, but regulates its sending rate at the transport layer when co-existing with BE CC. In absence of BE traffic, LBE can scavenge spare bandwidth not used by BE, but should yield to the former when it’s present. This means that more capacity can be spent on applications more relevant to the user, while
still keeping LBE flows alive. Using LBE for low priority external traffic may also have potential in data centers, where other internal traffic must have bit rate availability and low latency. Further, it may also be possible to implement a deadline-aware LBE service, meaning a service that given a completion deadline and transfer size, will adapt to a more aggressive sending rate by lowering its ‘LBE-ness’ in order to maintain its deadline when necessary, but will otherwise exhibit LBE behavior.

1.2 Related work

In [35] a survey of transport layer approaches to achieve LBE-behaviour in end-hosts are presented. In particular it looks at how different delay-based Congestion Control (CC) algorithms can be used for this purpose. Most of the delay-based CC algorithms surveyed were found to have problems with providing LBE service, which was mostly related to their dependence on accurately estimate network delay. In 2011 [17] presented a delay-based CC algorithm called CAIA Delay Gradient (CDG). Its main purpose is to control sending rate by using change in network delay as a congestion signal to infer congestion, and employ heuristics to have a fair chance against the more aggressive traditional loss-based CC. It was originally implemented by the authors in FreeBSD. Recently in 2015, a master thesis by [23] made an implementation of CDG in the Linux kernel. It is similar to the FreeBSD version, except it can be tuned in way that makes it exhibit ‘LBE-ness’. We use this implementation to evaluate whether CDG is a suitable candidate to provide a CC-based LBE service in Linux.

1.3 Problem statement

CC behavior depends on feedback it receives from the network in form of signals such as packet loss or delay, and how it is tuned to the specific network setting. If CC is to be used as a means to implement a LBE service it will need to be tunable in a way that enables it to have little impact on BE TCP flows. We have chosen the Linux implementation of CDG as our candidate for CC-based LBE service. In particular CDG has a parameter, which if increased makes it back-off (throttling its sending rate) more aggressively in response to increasing delay as an indication of congestion. The objective of this thesis will be to study how adjusting this parameter can enable LBE behavior in CDG flows towards BE flows.

1.4 Outline

The rest of the thesis is structured as follows

Chapter 2 Introduces some fundamental concepts such as communication protocols and discuss problems related to data transmission over the Internet. We then proceed to discuss TCP and Congestion Control.
Finally we review some delay-based CC algorithms and discuss their suitability for providing a CC-based LBE service.

Chapter 3  Gives a description of CDG and its Linux implementation. We move on to discuss LEDBAT which is a well known algorithm used for providing LBE. Last we look at the potential advantages CDG has over LEDBAT

Chapter 4  Describes the test-bed environment used in the evaluation. We present the test-bed topology and emulation tools as-well as discussing experimental pitfalls related to emulated environments. Lastly we present the metrics we have chosen to use in our evaluation, and give a presentation of the tools used to conduct the experiments.

Chapter 5  Presents the results of our experiments and describes our parameter space

Chapter 6  Concludes the thesis and presents future work that the thesis did not cover
Chapter 2

Background

2.1 IP networks

Because of its size and complexity it is difficult to describe the Internet with a few sentences. A rough description could be that it is a world spanning network of networks interconnecting a large number of computers (hosts) around the world, with the main goal to provide connectivity among host machines at its edges by acting as their medium. Connectivity however is not enough. There must be semantically well-defined protocols between the connected machines in order to facilitate meaningful data exchange.

2.1.1 The Internet Protocol Suite

Hosts and network devices work together by using communication protocols. A communication protocol in the context of a data network is a strict definition of how the format of transmitted data must be, and rules for how data is to be transmitted between two network entities. The Internet Protocol Suite, also called the TCP/IP stack, is the most widely deployed family of protocols in the Internet and defines many of the most common protocols used in the Internet today. A common model used to describe it, is the TCP/IP reference model shown in Figure 2.1a. It consists of four layers, each layer provides a service to the layers above (except the application layer which provides services to user applications). A short description of the roles of each layer is listed below.

**Application Layer**: Provides Internet related services to applications at end-hosts (e.g. web-browsing, file transfers, Voice-over-IP).

**Transport Layer**: Facilitates host-to-host data transportation. It encapsulates data received from the Application Layer into its data unit. This can either be ‘segment’ or a ‘datagram’ depending on the transport layer protocol.

**Internet Layer**: Responsible for sending data over the Internet by routing them towards the destination based on an address. While the Transport Layer only operates between end-hosts, the Internet Layer
operate on both hosts and network devices. It encapsulates a segment or datagram into its own data unit called a ‘packet’.

**Link Layer:** Transfers packets over a physical channel between two network devices. It encapsulates packets into ‘frames’ before transmission.

The data unit of a layer consists of the encapsulated actual data from the layer above, together with the added protocol data (header) which is control information transmitted between peers of that layer. When data is encapsulated the protocol adds protocol specific data in the header before sending it to the protocol in the layer below, or in the case of a link layer protocol, transmits it over a channel. When a protocol receives a data unit from a peer protocol, it processes the header and then strips of all header data before sending the content to the next upper layer. A network device which implements a specific layer, must implement all layers below it as well (e.g. a router implementing the Internet layer must also implement the link layer). End-hosts usually implement all four. Figure 2.1b shows a simplified picture of data flows between layers when two end-hosts communicate [39].

### 2.1.2 The Internet Protocol

The main protocol that facilitates connectivity in the Internet among connected hosts is called the Internet Protocol (IP). Operating at the Network Layer of the TCP/IP stack on both hosts and routers, it routes packets across the Internet based on a destination address called the IP address, located in the header of every packet. Every network interface has an IP address associated to it. When packets are sent through routers, IP forwards them to the next node based on this address.

### 2.1.3 Network buffers

If packets arrive too fast for a router to forward, they are buffered in queues so that they can be forwarded later. Larger queues gives increased ability to maintain the utilization of the bit rate capacity and absorb traffic bursts (a large number of packets arriving at once). Many network queues use a scheme called FIFO tail-drop [4]. This means that excessive packets are enqueued and serviced in order of arrival. When the queue is full the router starts dropping arriving packets.

### 2.1.4 Network queues sizing

The decrease over time of the price of computer memory, makes it feasible to add more buffer space to network devices like router. Although this makes it possible to maintain utilization of capacity under higher traffic loads, there is a downside to it. Because sender hosts often will probe for available capacity by continually increasing their sending rate, larger network queues allow senders to an increasing degree to transmit more
(a) The TCP/IP protocol stack

(b) Data flow between two host using similar TCP/IP protocols

(c) Application data after being encapsulated by the TCP/IP protocols. Note the ‘footer’ appended at the end, which usually is a CRC error detection code.
data than the routers is able to process. As the queue size of a router increases so does the per packet queuing delay, because packets have to spend more time waiting in the queue before being processed. This condition of a queue having large queues and many packets delayed because of sender capacity probing is called buffer-bloat [8]. Buffer-bloat can be a considerable contributor to latency if left unchecked. Regulating the size of the network queues often mitigates this problem, at the cost of decreased capacity under high load. However if queues are undersized and traffic is sufficiently bursty, packet may be dropped. If a reliable transport protocol such as TCP is used, retransmission of lost packets will results in increased end-to-end delay.

2.1.5 Internet bottlenecks

For any given flow, the router along its path with the lowest available capacity is called an Internet bottleneck. The performance metrics associated to the flow, such as its bit rate and delay, will be limited by the available capacity of its bottleneck. A router can be a bottleneck of a flow for several reasons. The hardware (its interface card or memory size) may not support more than a certain bit rate while other routers along the path support larger bit rates. The router may be a gateway to the Internet for many other flows, simultaneously contending for its resources. If a router prioritizes traffic differently, certain flows may be limited due their having a lower priority.

2.1.6 Unreliability of the Internet

IP networks like the Internet are packet switched networks, meaning bit rate is allocated dynamically depending on demand (e.g. packets gets served in a first-come first-serve manner) The Internet makes no guarantees that the packets sent will be received at their destination. Following are listed some of the reasons

- One reason is packets that have to be dropped when a routers capacity is exceeded as discussed in Section 2.1.3
- A router may temporarily be unavailable due to malfunctions in either hardware or software.
- IP treats packets independently, and routing changes can make packets from the same flow traverse different paths, and be received out of order.
- Although most link layer protocols implement error detection, routers due to software or hardware malfunction may corrupt packets. The packet sent from a sender host can therefore be different from the one arriving at the receiving host.
- Malfunction in Internet devices can also cause packets to be duplicated, however the most common source of packet duplication is from transport layer protocols that may resend segments.
2.2 Transmission Control Protocol

The Transmission Control Protocol (TCP) is a transport layer protocol providing the following to the application layer.

Reliability: Ensures transmitted data is eventually delivered to the TCP receiver.

In-order delivery: The data received at the receiver side application layer are received in the order they were sent.

Error checking: End-to-end error checks are performed before passing segments on to the application layer.

Flow control: A TCP sender does not send at a bit rate higher than the receiver is able to store and process.

Congestion control: It is the responsibility of TCP to regulate it sending rate to avoid sending more than the network is able to absorb in order to avoid congestion.

2.2.1 TCP overview

TCP provides reliability by having the receiving host send back acknowledgement packets (ACK) to the sender, as a receipt confirming which packets have been correctly received. All segments are marked with sequence numbers. The sequence number indicates in which part of the byte stream a segment belongs. An ACK is a segment sent from the receiver which confirms the reception of data up to the sequence number in the ACK. Thus, ACKs are cumulative. The reception of an ACK with a given sequence number, means that the receiver has received all bytes up to the ACK sequence number. If a TCP sender does not receive an ACK (confirming reception of previously unacknowledged data) within a certain time interval, it resends its oldest unacknowledged data. A TCP receiver will send an ACK when receiving new data from the sender. This ensures that lost packets are eventually received. A TCP receiver will wait until sufficient data has been received reliably before passing it to the application layer.

TCP segments are also marked with an error checksum that is computed before being sent, which is used by the receiver to detect whether a packet has been corrupted during transmission by looking at discrepancies between the packet content and the checksum.

TCP flow control keeps a fast sender from overwhelming a slower receiver. Sometimes a receiver will have limited processing resources or buffer capacity. If the sender is faster than the receiver and the network paths capacity is larger than the limit of the receiver, TCP flow control must limit the senders rate. Another important task of TCP is to avoid network congestion. TCP’s mechanism to avoid this is called Congestion Control (CC). This will be explained in section ??
## 2.2.2 TCP basics

Every segment sent by TCP contain an initial part portion called a the TCP header. The TCP header format is depicted in Figure 2.2. The TCP header is used mainly for providing the above mentioned services, as-well as identification of the endpoints of a flow, by use of the port number.

**Source port:** Port number of the process/service sending the packet (2 bytes)

**Destination port:** Port number of the process/service recipient of the packet (2 bytes)

**Sequence number:** Sequence number of the first byte of data in this segment. Sequence numbers are accumulated and marked by the sending host (4 bytes)

**Acknowledgement number:** Acknowledges the number of bytes received correctly (in order) by the receiver. The number is set to the next byte the receiver expects and is only valid if the ACK flag is set (4 bytes)

**Data offset:** The byte number in the segment which contains the actual data sent. Varies with header size (4 bits)

**Reserved field:** Not assigned a role (3 bits)

**Flags:** A bit field used for signaling between hosts (9 bits)

**Window:** The maximum number of bytes the receiver is willing to receive (2 bytes)

**Checksum:** A checksum calculated by the sender to detect eventual segment corruption at the receiver

**Urgent pointer:** An offset from the sequence number indicating data to be considered urgent for the receiving application. Only valid if the URG flag is set (2 bytes)

---

![Figure 2.2: TCP segment header.](image)
TCP options

After these field there is the variable length TCP Options fields. The options fields can be up to 40 bytes and the number of bytes in the option field must be divisible by 4. This why the data offset field is relevant. TCP options are mainly used to extend the functionality of TCP. If the receiver is able to receive more than 64KB at a time, it can use the option ‘Window scale’ to specify receive windows up to 1GB. Another option called ‘Selective acknowledgements’ (SACK) [cit-55] allows the receiver to specify intervals of bytes which has been received correctly instead of only cumulative acknowledgements. This is more efficient when several bytes have been lost before reaching the receiver.

Payload considerations

The size of the data portion of the segment will vary according to application. In the SSH application layer protocol, only single characters may be sent at a time, so the data portion will be small. When downloading large files, the data portion will be equivalent the ‘Maximum Segment Size’ (MSS). The MSS is the largest size, in bytes, that can be sent between two hosts over a connection. It is usually set to avoid fragmenting segments into several packets when sent over the network. Fragmentation at the network layer can lead to more loss since a segment is distributed over several packets. If one of the packets carrying a part of the segment is lost, the whole segment has to be retransmitted. Fragmentation can be avoided by using the path MTU (Maximum Transmission Unit) discovery technique. This sets the ‘Don’t fragment’ flag in the IP packet header. If the packet reaches a router that cannot forward it without fragmentation, it will drop the packet and respond with a ICMP message telling the sender to further reduce the size of the packet.

2.2.3 Lifetime of a TCP connection

TCP is a connection-oriented protocol, meaning sessions between hosts have to be established as a connection before data can be transmitted, and the state of the connection must be stored on both hosts. In order to connect to another host, TCP uses special synchronization messages called SYN packets. A SYN packet is simply a TCP segment usually with a empty data portion and the SYN flag set. The process of establishing a TCP connection is called a ‘three-way handshake’. First the initiating host sends a SYN packet. If the recipient host is listening to the port specified in the SYN packet, it will respond with its own SYN packet, except the ACK flag is also set. When the initiator receives the recipients SYN packet with the ACK flags set (called SYN-ACK packets). It confirms this with a standard ACK packet. Then the connection can be considered established. During the handshaking process parameters of the connection, like whether to use SACK or the Window scale, are negotiated. Figure 2.3a shows a flow diagram describing the initiation phase.
After the handshaking process data can be exchanged between hosts as shown in 2.3b. All data segments sent by a host must be acknowledged by the receiving host by ACK packets. A TCP connection is considered duplex (data packets can travel in both directions), therefore ACK’s can be piggybacked in data packets if both hosts are sending data. Should a receiving host become unavailable, the sending host will usually detect this after a number of unsuccessful retransmission or by the reception of an ICMP message from a router telling that the destination host as become unreachable.

When one side (initiator) wishes to terminate the connection, it sends a FIN message to the other host (receiver). It is similar to the SYN message, but with the FIN flag set instead. The initiator then waits for the receiver to acknowledge the FIN message. When the receiver gets the FIN message it immediately sends back an ACK followed by its own FIN message and then passively waits for the initiator to ACK its FIN message. When the initiator receives the ACK from the receiver it waits for the FIN message from the receiver. When it arrives it replies by an ACK and then waits a certain time to make sure the receiver receives the ACK. This process is shown in Figure 2.3c. Should the ACK be lost, the receiver must send another FIN message to the initiator, where the initiator will reply with another ACK. When the receiver receives the ACK it closes the connection, while the initiator closes the connection on timeout.

2.2.4 TCP Congestion Control

Most commonly the TCP CC protocol working at the end-hosts have little knowledge of parameters such as the capacity of the links along the path, or topology changes due to re-routing. However it must, based on feedback in the form of signals (such as packet loss or delay), regulate its sending rate to avoid congesting the network.

The task of CC is to regulate the number of outstanding packets between hosts in a network. The goal is to maintain this number at an optimum. The most important rationale behind CC is the phenomenon of congestion collapse. The first network to deploy the TCP and IP protocols was the now historical ARPANET. In ARPANET IMPs (Interface Message Processor, predecessors to today’s routers) managed packet routing and were mostly homogeneous in hardware and capacity [5]. When the TCP/IP suite was deployed in other network where routing equipment and links had different capacities, the routers with least capacity would be susceptible to overload, called congestion, when being traversed by a sufficient amount of packets. Typically congestion would occur at gateway routers in wide area networks (WAN) [5]. These gateways would be interconnected by links slower than the Ethernet-based local area networks (LAN) connected to them.

Faced with congestion these routers had no choice but to drop excess packets, making TCP hosts assume the packet it sent had been lost. At that time TCP did not take this into account, caring chiefly about flow control,
Figure 2.3: Phases of a TCP connection

(a) TCP connection initiation

(b) TCP data exchange

(c) TCP connection tear down
reliability and error checking between hosts [20]. TCP would then respond by retransmitting the packet dropped by the network without adjusting their sending rate, further compounding congestion. Because higher end-to-end delay follows congestion, a TCP sender may also time-out waiting for an ACK and perform a retransmit even though the packet had not been lost. When a network is in this state it is characterized by high end-to-end delays and reduced capacity, mainly because of large and slow moving queues and packet drops. This phenomenon is called congestion collapse, which may happen whenever a build up of packets at bottleneck routers reaches a critical levels, and the incoming packet arrival rate is much higher than the outgoing one [11]. In addition hosts may have difficulty establishing new connections because SYN messages and their replies are more likely to be lost.

**Congestion Control**

The first remedy to address the congestion collapse problem are the algorithms outlined in [1] [cit-52]. These fundamental algorithms (collectively referred to as 'Congestion control') are the main mechanisms used to avoid congestion in the Internet without sacrificing efficiency. CC works in a distributed manner at end-hosts, and controls the number of bytes a host can inject into the network using a counter called 'congestion window' (cwnd). The cwnd represents how many unacknowledged packets / bytes of data a sender may have in transit at once. When initiated CC usually sets cwnd to a small size (usually 2MSS). To increase cwnd CC relies on the reception of ACK's. If an ACK, acknowledging previously unacknowledged data, is not received within a certain time interval specified by a timer called the 'Retransmission time-out' (RTO), CC will retransmit the most last unacknowledged segment, and infer congestion. We will now look at the elements common in most CC algorithms. Although CC algorithms vary with regards to how they detect and respond to loss, the following components are commonly implemented by many CC variants:

**Slow start (SS)** The algorithm used initially to probe the network path for available capacity. Contrary to its name it is exponential in nature, increasing cwnd by one SMSS for every acknowledgement received as shown in Equation 2.1. Thus for every round trip time the cwnd is doubled. When the capacity is reached, packets are lost. This will usually be detected by the RTO or the reception of three duplicate ACKs (dup-ACKs). A dup-ACK is the last ACK already sent by the receiver. It sends a dup-ACK when it receives data out of order, to indicate to the sender that a segment may have lost. When exceeding the capacity of the path the sender will have exceeded it by at most twice its size, because it takes the sender roughly one RTT to infer congestion, and the sender will only double its rate for each RTT. Slow start is exited when cwnd exceeds a variable called ssthresh (slow start threshold). After detecting loss, the sender sets ssthresh to half the number of packets in transit (flight size). And takes measures,
depending on the ‘Congestion avoidance’ algorithm in use, to reduce its sending rate. After congestion inference, \( ssthresh \) represents an estimate of the paths capacity.

\[
cwnd = cwnd + MSS
\]  
\( (2.1) \)

**Congestion avoidance** When \( ssthresh \) is exceeded, the sender is likely to be close to the path capacity (the capacity of the bottleneck along the path between sender and receiver). To avoid exceeding the paths capacity, \( cwnd \) must increased more carefully. In ‘Congestion avoidance’ (CA), which is used when \( cwnd > ssthresh \), the cwnd is increased by 1 MSS for each RTT. This is done by incrementing cwnd according to Equation 2.2 for each ACK received, or instead use a counter \( cwnd\_cnt \) incremented on every ACK which acknowledges new data, and then reset the \( cwnd\_cnt \) and increment \( cwnd \) when \( cwnd\_cnt = cwnd \). By doing this CC can keep on probing for more capacity should the network conditions have changed, but in a more conservative manner than SS. If a packet loss is detected by the reception of three dup-ACKs, CC will do a ‘Fast retransmit’ and switch to the ‘Fast recovery’ mode.

\[
cwnd = cwnd + \frac{MSS^2}{cwnd}
\]  
\( (2.2) \)

**Fast Retransmit and Fast Recovery** On reception of three dup-ACKs it is safe to assume that the corresponding segment has been lost, and that the dup-acks are not a result of packet re-ordering. Instead of waiting for the retransmission timer to expire, the fast retransmit algorithm retransmits the lost segment immediately. Having experience a loss means that the congestion window needs to be reduced. One way is for the sender to set \( cwnd \) to its initial value, and re-initiate SS. This means capacity probing will start anew, even though we have a rough estimate, namely \( ssthresh \). Instead the Fast Recovery (FR) algorithms sets \( cwnd \) and \( ssthresh \) according to Equation 2.3 , where flightsize is the estimated number of current unacknowledged packets in transit between the sender and receiver. It continues transmitting until receiving an ACK acknowledging the previous window. FR has the advantage that it maintains \( cwnd \) near its optimum as-well as maintaining the ‘ACK clock’ of the sender. When operating in FR mode, for each dup-ACK received it computes \( pipe \). \( pipe \) is an estimate of the current number of outstanding packets between the sender and receiver. After waiting for half an RTT (in order to let \( cwnd \) be halved) \( cwnd \) is incremented by 1 MSS for each dup-ACK received. This is because these packets, which triggered a dup-ACK at the receiver, are now buffered there and are no longer consuming network resources. If \( cwnd \) allows, it sends \( cwnd - pipe \) new packets.
Figure 2.4: TCP Tahoe and TCP Reno response to congestion. Tahoe resets \( cwnd \) and goes directly to SS, while Reno sets \( cwnd \) to half the flight size and enters FR on reception of three dup-acks. The \( sssthresh \) line is set after congestion detection at RTT round 7.

If an ACK arrives which acknowledges the whole sender window, \( cwnd \) is set to \( sssthresh \) and FR is exited. If the unacknowledged packet times out, FR is exited and SS entered.

\[
cwnd = sssthresh = \frac{\text{flightsize}}{2} \tag{2.3}
\]

These are the basic operations of the classic TCP CC algorithm New Reno. The policy of incrementing \( cwnd \) linearly in CA by a constant \( \alpha \), while reducing it multiplicatively by on loss by a factor \( \beta \) is called Additive Increase Multiplicative Decrease (AIMD). It results in the growth function of \( cwnd \) to have a sawtooth pattern over time. [39][1]

**Standard TCP congestion control**

The first CC algorithm designed to solve the problem of congestion collapse was dubbed TCP-Tahoe. It had all features mentioned above except FR. Instead it would do a fast retransmit and go directly to SS. TCP-Reno, improves Tahoe, by resetting the retransmission timer and entering FR after performing a fast retransmit. Figure 2.4 shows the
main differences between Tahoe and Reno. In situations where multiple
packets are lost, dup-ACKs of segments with different sequence number
may trigger halving of cwnd several times. This can be remedied by using
the SACK option in the TCP header so that the sender can selectively
retransmit all lost segments at once. In cases where the SACK option is
unavailable (either because of lack of implementation or the willingness
to use it), TCP New-Reno further improves retransmission in FR. When
first entering FR, New-Reno checks to see whether ‘acked’ is larger
than a variable ‘recover’ (initially set to 2MSS). If so, ‘recover’ is set
to the sequence number the of last unacknowledged segment sent by
recording the highest unacknowledged packet. Each ACK that makes
partial progress (acknowledging a sequence number less than ‘recover’)
resets to retransmission timeout and sets ‘recover’ to reflect the changes.
This allows for multiple holes in a window to be filled without unnecessary
reduction of cwnd [12].

2.3 TCP Fairness

While preventing congestion is the primary concern of CC, fairness among
flows, in the sense of allocation of bit rate to each flow also important.
Different flows may have a shared path that spans many links, or only a
few links. Fairness becomes relevant when two or more flows share a given
link or traverse the same bottleneck, where we want each flow to receive
a fair share. Mostly, the Internet does not guarantee fairness between
packet flows. IP treats packets independently regardless of which host
they originate from. Therefore, CC must implicitly perform allocation of
bit rate among themselves. This is why fairness is considered an important
property of a CC algorithm. Fairness among flows using the same CC is
called intra-protocol fairness, while fairness among flows with different CC
is called inter-protocol fairness. Flows using standard TCP, given the same
conditions, are fair towards one another, due to TCP’s AIMD policy to cwnd
growth which will eventually results in convergence among TCP flows.

How different kinds of CC regulate cwnd in response to network signals
varies. If two hosts use different CC, one host may overrun the other if its
CC grows cwnd more aggressively. A flow that doesn’t exceed the rate of
TCP, in a given network condition, is said to be TCP-friendly [10].

2.3.1 Max-min fairness

Formally an allocation a is max-min fair if and only if the following hold:

1. ∀f : f ∈ F_L ⇒ λ^a_f,L > 0 ∧ \sum_{i=1}^n λ^a_{f_i,L} ≤ C_L
2. ∀λ^b : λ^b_f,L > λ^a_f,L ⇒ λ^b_f,L < λ^a_f,L ≤ λ^a_f,L

Here λ^a_f,L and λ^b_f,L are the bit rates allocated by allocation a and b
respectively on link L to flow f, F_L is the set of all flows traversing link
L and C_L is the capacity of link L.
In other words, an allocation is said to be max-min fair if and only if it is feasible, and the increase of the bit rate allocation of a single flow results in the decrease of bit rate of another allocation with equal or less bit rate allocated. An allocation is feasible if it doesn’t allocate a bit rate to a flow so that it exceeds a links capacity, and it allocates some bit rate larger than zero to all flows. Max-min fairness ensures that minimal flows are maximized, so that a flow cannot increase its bit rate if that means a flow with equal or less bit rate than itself must be reduced.

An example of a max-min fair allocation is shown in figure 2.5. Since all flows are given a bit rate greater than 0 and no allocation exceed a channel, the allocation is feasible. The second property also holds, if B is allocated, say equal capacity as A between R5 and R6, it must necessarily result in a decrease in C or D which has an equal share of the capacity with B Due to its AIMD policy, allocations between standard TCP roughly approximates the max-min fairness property [39].

2.3.2 Jain’s Fairness Index

Raj Jain suggested measuring fairness using a quantitative measure, now called called Jain Fairness Index. The function is shown in Equation 2.4

\[
f(x_1, x_2, \ldots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2}
\]   

When capacity is shared among \( n \) flows the bit rate given to each flow is denoted as \( x_i \) where \( 0 < i \leq n \). If the result is equal to 1, each bit rate allocation is equal. If \( f \) is equal to \( 1/n \), then only one flow is given the whole capacity. If \( k < n \) flows are allocated equal bit rate and \( n - k \) flows are allocated none, then \( f = k/n \) [22]. Because the function gives a quantitative description of the fairness of the allocation of each flow, we will use this metric when looking at how different flows allocate capacity between themselves.
2.4 Linux congestion control

2.4.1 TCP CUBIC

Current releases of the Linux kernel have several flavors of CC available, but use a congestion control called CUBIC [33] by default. A problem with the previously mentioned New-Reno is its dependence on the RTT of the flow’s path for growing its cwnd. Paths with high RTTs and bit rates are commonly referred to as high bandwidth-delay products (BDP) paths. High BDP means that the path can more bits in transit at once than a path with a lower RTT or bit rate. New-Reno is known to underperform along high BDP paths, because of its incremental policy to cwnd growth. Moreover, cwnd growth depends on the the stability and rate of the ‘ACK clock’ and consequently, is affected by traffic load and RTT. This means flows traversing paths with different RTT grows cwnd at different rates, leading to an unfair allocation between them. Like TCP, CUBIC infers congestion when detecting a packet loss, but does not depend on RTT for growing cwnd in CA. Two great advantages of this kind of cwnd increase is that network paths with higher delay and bit rates are better utilized. Also RTT independence makes for fairer competition between flows of different RTT’s. CUBIC doesn’t use the AIMD policy to adjust it’s window during CA, but uses instead a function of the time interval between two congestion events to find a ‘target’ for cwnd to grow towards. When CUBIC infers loss in CA, it registers current cwnd as $W_{max}$, sets $cwnd = cwnd \times \beta$ and then does a fast retransmit followed by FR. When it returns to CA it adjusts cwnd growth every RTT according to the cubic function (hence the name “CUBIC”) in equation 2.5,

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

(2.5)

where $t$ is the elapsed time since the last congestion event, $C$ is a parameter constant that shapes the growth and $W_{max}$ is as mentioned the size of $cwnd$ before its reduction after the last congestion event. $K$ is equal to Equation 2.6,

$$K = \frac{3}{\sqrt[3]{W_{max} - cwnd}}$$

(2.6)

where $\beta$ is the multiplicative decrease factor (0.8 by default in Linux). $K$ is the time it takes for the cwnd to grow to $W_{max}$. Thus the first term of Equation 2.5 will be negative until $cwnd > W_{max}$, where it will be close to zero. During this growth towards $W_{max}$, cwnd will have a concave shape. When it reaches $W_{max}$, Equation 2.5 will grow larger than $W_{max}$ resulting in a convex shape of cwnd, as it tries to probe for more capacity. The cwnd growth function of CUBIC is shown in Figure 2.6 The capacity probing is necessary to avoid cwnd from stagnating. If some flows have left the bottleneck, CUBIC should grab the spare capacity. CUBIC is considered a relatively aggressive CC algorithm due to its very fast concave recovery after a loss. It also has problems with convergence between flows due to its low reduction of cwnd after packet loss. The fairness issue can be mitigated
Figure 2.6: The cwnd growth function of CUBIC. It first operates in the ‘steady-state’ region initially ramping swiftly up towards its target value. Later it tries to probe for more capacity in the ‘max probing’ region.

in Linux by tuning the C parameter. Although TCP CUBIC solves the issue with RTT dependency by having an adaptive cwnd growth function, it has been shown to have performance issues in wireless networks when coexisting with delay-based CC [3].

2.4.2 Linux improvements to SS

As of today, Linux offers several CC algorithms. Its uses CUBIC by default (which can be modularly replaced by other kinds CC algorithms). CUBIC in Linux implements SS and FR differently than in [1]. SS rapidly increases cwnd. This may lead to cwnd overshooting, especially along paths with bloated network buffer larger than the BDP of the path because the burst will be absorbed, fooling SS to double its sending rate to exceed the path capacity [cit-46]. In Linux CUBIC uses a SS scheme called Hybrid Slow Start (HyStart) [cit-46] [32] in place of traditional SS, to make SS exit before overshooting. HyStart employs two heuristics to decide when to exit, the ACK train length and delay estimation. It looks at the sum of the Inter-Arrival Times (IAT) (the time between packets arriving at a receiver) of ACKs each RTT round (called ACK train length) and compares them with an estimate of path one way delay (OWD). When the sum of IAT’s approaches the estimated OWD, SS is exited. If the path is already occupied or congested by other flows, the delay estimation heuristic complements the ACK train length. If the path is already occupied by other flows when a HyStart flow starts, it will detect congestion by looking at whether differences in RTT of its first packets (which at the initial low rate does not cause congestion by themselves) is larger than some threshold value.
2.4.3 Linux improvements to FR

Linux uses a modified FR called Proportional Rate Reduction (PRR) developed by Google [9]. As earlier mentioned standard FR reduced \( cwnd \) to half the current flight size. This means that nothing is transmitted during a period of half an RTT, resulting in lost transmission opportunities. If losses are bursty, transmission during FR may become bursty as a results of \( pipe \) being wrongly estimated. PRR solves these problems by operating in two modes. The first mode is active when \( pipe > ssthresh \). It then calculates the number of new packets to send using Equation

\[
\text{sndcnt} = \left\lceil \text{prr}_{\text{delivered}} \times \frac{ssthresh}{\text{RecoverFS}} - \text{prr}_{\text{out}} \right\rceil \tag{2.7}
\]

When \( pipe \leq ssthresh \) it operates in SS. Unlike standard FR, this ensures that \( cwnd \) is gradually converging towards \( ssthresh \) reclaiming much of the lost transmission opportunities lost and avoiding bursty transmission when losses are bursty.
Loss-based CC reacts to packets loss (retransmission time-outs or reception of dup-ACKs) as a congestion signal. But these signals are generated only after congestion itself has occurred. In order to find an optimal sending rate, meaning that the bottleneck link capacity is not over or under-utilized and other flows are treated fairly, loss-based CC must induce congestion. Because of this reactive approach, many packets are usually lost. Moreover, it is difficult for a loss-based CC to differentiate between packet loss caused by network congestion and loss caused by link errors like wireless links. AIMD Loss-based CC like NewReno often oscillates around an optimum value of cwnd, especially if the bandwidth-delay product (BDP) is large, due to the late response to congestion, in combination with larger decreases to cwnd in response to congestion.

3.1 CARD

In 1989 Jain proposed a scheme, ‘Congestion avoidance using round trip delay’ (CARD) using RTT measurements as a means for measuring congestion. It uses a normalized delay gradient \( \frac{dD}{dW} \) to control whether to increase or decrease cwnd. An advantage of using delay to control cwnd growth is that it allows cwnd to be more smoothly stabilized near an optimum value, instead of an oscillatory behavior typical of standard-TCP. The scheme is independent on the size of network buffers in a way that larger buffer doesn’t introduce more delay, since delay itself controls cwnd [21].

3.2 TCP Vegas

A well-known delay-based CC scheme is TCP-Vegas [6]. It uses fine-grained timers for each packet for precisely measuring per packet RTT. It tries to measure the base path delay of the path between hosts, called base-RTT (BRTT), which is the smallest RTT measurement made (e.g.
the network propagation delay when queues are empty). It compares
the actual throughput using measured packet delay, to the expected
throughput using \( BRTT \).

\[
diff = (\frac{cwnd}{BRTT} - \frac{cwnd}{RTT}) \times BRTT = \frac{cwnd}{RTT} \times (RTT - BRTT) \quad (3.1)
\]

\[
cwnd = \begin{cases} 
  \text{diff} \leq \alpha & cwnd = cwnd + 1 \\ 
  \alpha < \text{diff} < \beta & cwnd = cwnd \\ 
  \text{diff} > \beta & cwnd = cwnd - 1 
\end{cases} \quad (3.2)
\]

In equation 3.1, \( \frac{cwnd}{BRTT} \) and \( \frac{cwnd}{RTT} \) represents
the actual and estimated throughput respectively. \( RTT - BRTT \) is
the estimated queuing delay, thus \( \text{diff} \) represents Vegas’
estimate of congestion. Vegas uses two parameters \( \alpha \) and \( \beta \),
which are the lower and upper congestion thresholds respectively.
Vegas’ congestion avoidance behaviour is shown in equation 3.2. If \( \text{diff} \) is lower than \( \alpha \), \( cwnd \) is incremented since
queuing delay is low and the path is potentially under-utilized. If \( \text{diff} \) is larger than \( \beta \) (meaning an onset of congestion) it slowly
backs off to avoid congestion, instead of dramatically reducing \( cwnd \) after congestion has occurred like classic TCP does [36]. If
neither of these conditions are the case then \( cwnd \) maintains its size.

3.3 TCP Nice and TCP Low Priority

TCP-Nice [41] is another delay-based flow similar to TCP-Vegas, but is
designed for background transfer LBE services, whereas TCP-Vegas is
designed for maximizing throughput. Instead of the linear decrease of
\( cwnd \) in CA, Nice halves \( cwnd \) when exceeding \( \beta \) so as not to hurt
foreground flows. Nice can also increase \( cwnd \) by less than 1 MSS
during an RTT, enabling a more gradual increase. TCP Low Priority (TCP-LP) [25],
also a TCP protocol designed for LBE, uses OWD measurements which
are smoothed using Exponential Weighted Moving Average (EWMA) and
compares it to interval of the maximum and minimum measurements
observed as a congestion indication. When a congestion indication is
inferred it halves its window and sets a timer while waiting to see if another
congestion indication arrives, if so it sets \( cwnd \) to 1 MSS, if not it maintains
\( cwnd \). This is a compromise between an overly careful behavior, and of the
standard TCP loss-based approach.

3.4 LEDBAT

The Low Extra Delay Background Transfer (LEDBAT) [37] protocol tries
to solve the problem related to BRTT estimation. The goal of LEDBAT is
to keep forward queuing delay low beneath a threshold parameter value
\( TARGET \) (100 ms by default). LEDBAT is designed specifically to be used
as a LBE CC. It is meant to be used as a CC algorithm in general, meaning
it can be applied to application layer protocols. It has been adopted by both BitTorrent, and Apple in its software update manager. It uses OWD measurements and compares them with the minimum value of a windows of \textit{BRTT} estimates called \textit{BASE\_HISTORY} (10 entries by default) to estimate the queuing delay (QD). The purpose of \textit{BASE\_HISTORY} is to forget old \textit{BRTT} estimates that has become irrelevant due to routing changes, but avoid instability by keeping the window large. \textit{BRTT} estimates are made every minute and then added to \textit{BASE\_HISTORY}. Thus a \textit{BRTT} estimate is valid for only ten minutes. A normalized value off\_target is calculated for every ack received accordin to 3.3

\[ \text{off\_target} = \frac{QD - \text{TARGET}}{\text{TARGET}} \]  

The queuing delay is calculated by subtracting a filtered value from the minimum \textit{BASE\_HISTORY}. The filter is modular and can for instance be the EWMA of the OWD measurements. The growth of \textit{cwnd} is dictated by whether off\_target is negative or positive. LEDBAT requires the receiver to be configured to make OWD measurements. This can enabled by using the TCP timestamp option. LEDBAT adjusts \textit{cwnd} based on Equation 3.4,

\[ \text{cwnd} = \text{cwnd} + (\text{GAIN} \times \text{off\_target}) \times (\text{cwnd}_{\text{cnt}} \times \text{MSS} \times \text{cwnd}) \]  

where GAIN is an adjustable growth parameter and \textit{cwnd}_{\text{cnt}} is the number of bytes acked before the last \textit{cwnd} incrementation. Thus \textit{TARGET} plays an important role in determining the change of \textit{cwnd}. Adjusting \textit{TARGET} will have an effect on its LBE behaviour. It has been shown, using simulation, that in networks with large BDP values (e.g. 4G over satellite networks) LEDBAT yields more to foreground flows and utilize spare bandwidth better for smaller values of \textit{TARGET} (5ms) in moderately loaded networks. In fully loaded networks, there was little difference between 5ms or 100ms for the \textit{TARGET} parameter [24]. LEDBAT tries to solve the problem of unrepresentative \textit{BRTT} estimates that are outdated by using the more dynamic \textit{BASE\_HISTORY} to filter them out. However it has been shown in [34] that it may require certain conditions such as stable BE flows or pauses (for queues to drain), because LEDBAT may add its own self-induced queue delay to \textit{BRTT} when it forgets old \textit{BRTT} estimates, therefore it may become more aggressive than intended.

### 3.4.1 Problems with delay-threshold CC regarding LBE

TCP-Vegas and related CCs, use \textit{BRTT} as a measure of path delay. Since the queue size of a bottleneck router is not known by a sender measuring \textit{BRTT}, it must assume that the minimum RTT measured so far is the best estimate. A sender measuring a large \textit{BRTT}, even though the real \textit{BRTT} is much smaller, will believe that the propagation delay is large and can be more aggressive since this implies a larger BDP. If a newly established delay-threshold flow starts in this condition, the phenomenon known as latecomers advantage present a problem for LBE behaviour.
Latecomer’s advantage happens when newly established delay-threshold flows measure $BRTT$ to be high as a result of queuing delay induced by older flows. This gives them an advantage over other competing flows. On the other hand a sender measuring a smaller $BRTT$ will send more carefully, since it will infer congestion from much smaller throughput values. This means that in order for delay-based CC using threshold parameters to function optimally, the estimation of $BRTT$ must be accurate, which requires the queuing delay in the routers along the path to be small at some point. This in turn can depend on conditions in the network, like routing changes and traffic characteristics, of which the transport layer is largely unaware [17]. Paths through the Internet are unstable due to events like node failure and re-routing in the network layer. This may lead unstable delay between sender and receiver. [37].

3.5 CAIA delay gradients

A CC algorithm called CAIA Delay-Gradient (CDG) [17] avoids using threshold parameters, using instead delay gradients. A gradient is a measure of change of a particular variable, in this case a measure of how much RTT changes between intervals of RTTs. CDG measures change in maximum and minimum RTT. By looking at relative change in RTT, CDG infers the state of a bottleneck queue. The goals of CDG are to use gradients instead of threshold parameters as a congestion indicator, have a probabilistic back-off independent of RTT, co-exist with BE flows such as NewReno and CUBIC and differentiate between congestion related and non-congestion related losses.

3.5.1 The CDG algorithm

CDG uses gradients to infer the state of a bottleneck queue. To do this CDG keeps two variables $g_{min}$ and $g_{max}$ representing the change in minimum and maximum RTT measures of an RTT round respectively, which are updated every RTT according to Equations 3.5 and 3.6

$$g_{min,n} = \tau_{min,n} - \tau_{min,n-1}$$  \hspace{1cm} (3.5)

$$g_{max,n} = \tau_{max,n} - \tau_{max,n-1}$$  \hspace{1cm} (3.6)

where $n$ is the $n$’th RTT round. Even though these measures are less noisy than per packet RTT, they are filtered using moving averages shown in Equation 3.7.

$$\bar{g} = \frac{\sum_{i=n-a}^{n} g_i}{a}$$  \hspace{1cm} (3.7)

If the queue of the bottleneck router is increasing, both delay gradients will be positive on each measurement. CDG uses the following queue state heuristics. When the queue is about to get full, $g_{max}$ will stop increasing before $g_{min}$ stops. If the queue is draining $g_{min}$ will start to decrease, and then $g_{max}$ will start to decrease. The queue is emptying when $g_{min}$ stops
increasing before $g_{\text{max}}$ stops. CDG can use these heuristics to tell whether a packet loss is due to congestion or not (in the case of wireless channels).

CDG uses a probabilistic back-off in order to ensure fairness among other CDG flows. When the gradients increase, the likelihood of back-off increases. The probability function is exponential, in order to avoid larger RTT differences due to longer paths having higher probability to back-off. The $cwnd$ of CDG is adjusted by a constant increase of 1 if no incipient congestion is detected, else if the gradients indicate oncoming congestion and a random number $X > P[\text{Back-off}]$ is generated, where $P[\text{Back-off}]$ is shown in Equation 3.8,

$$P[\text{Back-off}] = 1 - e^{-\bar{g}_n/G}$$

$cwnd$ is set to $\beta \times cwnd$, where $\beta$ is the AIMD decrease factor. CDG can tell whether backing off is effectual by looking at the queue state heuristics (e.g. backing off should result in a decrease in $g_{\text{min}}$ and $g_{\text{max}}$). If after backing off a certain number of RTT’s has no effect on $g_{\text{min}}$ or $g_{\text{max}}$, CDG assumes another flow is consuming the bandwidth CDG is yielding. It then stops backing off a number of RTT’s, to avoid being out-competed by greedy flows. After a back off, CDG waits one RTT round before checking if it should back off again, because of the time it takes for the back off to become visibly effective. It also maintains a shadow window $s$ growing in a NewReno like manner. When backing off $s$ is initialized according to Equation 3.9.

$$s_{i+1} = \begin{cases} \max(cwnd_i, s_i) & \text{delay based backoff} \\ 0 & \text{queue is inferred empty} \\ s_i & \text{otherwise} \end{cases}$$

(3.9)

If a packet loss occurs and the shadow window heuristic is enabled, $cwnd$ is set to according to Equation 3.10.

$$cwnd_{i+1} = \begin{cases} \frac{\max(cwnd_i, s_i)}{2} & \text{queue is inferred full} \\ cwnd_i & \text{otherwise} \end{cases}$$

(3.10)

This means CDG won’t miss transmissions opportunities because of a packet loss. Another advantage of using a shadow window is that CDG can separate the back-off behaviour of CA from the back-off behaviour of a packet loss. It achieves the flexibility of a delay-based CA while avoiding being out-competed by greedy flows, like NewReno, causing packet losses.

### 3.5.2 CDG’s LBE capabilities

As a LBE transport service, CDG may be more suited than delay-threshold CC like LEDBAT and TCP-LP. The use of gradients to detect congestion avoids trouble with having to rely on unstable metrics like base delay which may actually cause non-LBE behaviour towards BE flows due to the latecomers advantage problem. LEDBAT avoids getting stuck with an outdated BRRT, but is still prone to errors caused by cases where
it considers self-induced delay as part of the base delay [21]. CDG’s delay gradient is more suited, since it is independent of precise BRRT measurements and setting of parameters, while still reflecting the state of the queue. It has been shown in [17] that CDG flows share bandwidth more fair among themselves than NewReno flows in scenarios where the probability of losses are 1% or more, when sharing a medium, and that CDG flows out-performs NewReno flows, with regards to delay and jitter, when sharing a medium. NewReno fill a bottleneck queue and maintains it until a loss is detected. This may induce more delay due to buffer-bloat. CDG using queuing heuristics can adapt its rate to keep queuing delay low. When co-existing with NewReno however, CDG tends to claim less than its fair share of bandwidth due to its careful approach to cwnd incrementation. A similar study [3] of CDG in a home WLAN environment shows that CDG exhibits LBE-behaviour towards BE traffic, and outperforms NewReno and CUBIC when considering delay and jitter. A test scenario using 2 CDG senders and 2 NewReno senders shows that CDG would have a median goodput of 1Mbps while NewReno would use 5Mbps. It was also shown that CDG imposes less and more consistent delay on latency-sensitive traffic in a wireless home network because of its proactive behaviour and smoother cwnd increases, while NewReno and CUBIC would impose higher and more varying delays, because of their tendency to let buffers be cyclically filled and drained. Another problem that was illustrated was how loss-based TCP would have a much more varying and smaller goodput upstream than CDG because of a higher degree of wireless slot contention. The probabilistic back-off of CDG helps to make CDG flows fair among themselves. Nice and LEDBAT mostly rely on every flow estimating BRRT to be the same for fairness. If buffer-bloat is a problem, fairness can be hard to achieve for Nice and LEDBAT because new flows will measure increasingly higher BRRT due to flows already inducing queuing delay. Although LEDBAT flows will eventually forget underestimated BRRT’s and adapt after ten minutes, LEDBAT requires other flows to stabilize in order to measure precise queuing delay, and will often not account for mice flows such as http request, because they may be too sporadic or temporarily spread to introduce the necessary base delay at the bottleneck queue for LEDBAT to measure. These mice flows may actually be out-competed by a LEDBAT flow even though they are regular BE flows. CDG does not seem to exhibit this behaviour, because it rely on gradients instead of parameters. Although CDG must be configured with a backoff probability parameter it does not impact how CDG detects the state of a bottleneck queue. We believe that the backoff probability parameter may be set to a single value (or a small set of values) to achieve LBE behavior in most cases. [21] [17]

3.5.3 Linux implementation

CDG is available in Linux from kernel 4.2 and onwards. It is based on the specification in [17]. The original implementation is due to [23]. Like other Linux CC’s it uses the PRR modification to FR, as-well as newly adopting
the HyStart SS mechanism used by CUBIC. It has features and switches to enable LBE-behaviour. In particular it has a parameter to change beta factor, backoff probability, and disable the shadow window, ineffectual back off detection and loss tolerance heuristics. It also differ from [17] by not resetting s on back off when the queue is empty. Instead s maintains a NewReno growth. We will base our evaluation on this implementation and use its parameters to achieve LBE-behaviour. Following are its configurable parameters.

- backoff_beta: Sets the AIMD $\beta$ value used when backing off, by default 0.7
- backoff_factor: Equivalent to the scaling parameter $G$ ($G = 1000 / \text{backoff_factor}$) Determines together with the delay gradients the probability of backing off
- hystart_detect: Toggles the use of the HyStart (Hybrid Slow Start) algorithm from CUBIC
- use_ineff: Toggles the use of the ineffectual backoff detection heuristic
- use_shadow: Toggles the use the shadow window heuristic
- use_tolerance: Toggles the use of the loss tolerance heuristic

Current Linux releases supports modular CC, in the form of a so-called loadable kernel module (LKM). LKM’s are object files that can be modularly inserted into the kernel to extend its functionality (e.g. device driver). The source code for the module is found in ‘net/ipv4/tcp_cdg.c’ in the kernel source archive. When compiling the kernel, the CDG code (if enabled in menuconfig) is compiled to ‘tcp_cdg.ko’. It can be enabled using the script in listing 3.1.

Listing 3.1: CDG module enable script

```
#!/bin/bash
if [ $# -lt 2]; then
    echo "missing arguments"
    exit (1)
endif
modprobe tcp_cdg use_shadow=${1} backoff_factor=${2}
if [ $? -neq 0 ]; then
    echo "error inserting module"
    exit (1)
endif
sysct1 net.ipv4.tcp_congestion_control=cdg
```

The modprobe command inserts the module into the kernel, while the following arguments are module parameters (corresponds to the CDG parameters above). Parameters not set in modprobe uses default values
defined in the kernel module code. A minimum requirement for a CC LKM is that it implements functions for setting the SS threshold and for performing CA. The hooks for these function are called ‘ssthresh’ and ‘cong_avoid’ in kernel code respectively. Listing ?? shows the CA algorithm used by Linux CDG. It is called by the ‘cong_avoid’ hook from ‘net/ipv4/tcp_input.c’ which handles the reception of packets (in this case an ACK) [31].

3.6 CDG’s potential advantage over LEDBAT

We will now look at some weaknesses inherent in LEDBAT, which we believe are not present (or to a lesser degree) in CDG.

3.6.1 Delay threshold setting

Being a delay-threshold CC LEDBAT has to be configured. As stated in the draft the recommended parameter value for the target queuing delay is 100. The argument is anecdotal, referring to experience with BitTorrent file transfers [37]. To our knowledge, no comprehensive research has been done to establish if this represent an optimal value. Even if it is now, this may not be true in 10 years. A study [24] has shown that LEDBAT may comprise both fairness and ‘LBE-ness’ over high BDP 4G satellite links if the target parameter is not set accordingly. In contrast to the draft, the study suggested that the target parameter should be set to 5ms in the satellite environment it was evaluated, but that it needs dynamically adjusted according to network conditions in particular when the number of flow increase. Thus, the LBE-ness of LEDBAT is depending in setting the right target value for the right conditions. CDG’s delay gradients are independent on BRTT because they measure change in minimum and maximum delay, and not delay itself. Tuned with a right G value, CDG may be able to give better guarantees to LBE-ness than LEDBAT.

3.6.2 Fairness issues

Another issues plaguing LEDBAT is its lack of guaranteed intra-protocol (and possible inter-protocol) fairness. In particular, the latecomers advantage scenario can be problematic for LEDBAT. Several LEDBAT flows in parallel, configured with the same target value will, in aggregate, not exceed the target queuing delay. However if one flow arrives, after other flows are established, the former may estimate a higher base delay, potentially starving other flows. Figure 3.1b illustrates this problem. First one LEDBAT flow started, then after 40 seconds another LEDBAT flow is started. Since flow 1 has already added to the queuing delay, flow 2 (assuming the queue is empty) will add it to its base delay, forcing flow 1 to yield. CDG has measures for ensuring inter-protocol fairness, it backs off randomly independent of RTT.

30
3.6.3 Possible confusion about base delay

The draft assumes that foreground flow are long-lived enough for LEDBAT to yield. If this is not the case, there may arise a condition where LEDBAT measures its own self-induced delay, when it forgets its lowest base delay measurement, and takes it to be a positive change in base delay. This may have serious impact on shorter TCP flows such as http or time critical UDP packets if the queuing delay becomes too large. Although not very likely, there’s no guarantee that this will not happen in a given context. Figure 3.1a illustrates this phenomenon. Here we set the base history length to 1. After each minute, LEDBAT forgets its oldest base delay recorded (because here there’s only one slot) and adds its own induced delay on top of the actual base delay (here near 0ms). This scenario can easily be extended to cases where the base history filter is longer. With k slots, it will forget its oldest (and possible correct) base delay in k minutes. We don’t expect CDG to exhibit this behaviour.
Chapter 4

Experimental test-bed design

We have chosen to use an emulated testbed environment as the setting for conducting experiments. Section 4 describes specifics about the emulation, and some considerations we had to make. Section 3.2 show how we have set up our test environment. Section 3.3 presents our choice of metrics and why we have chosen them. Section 3.4 gives a description of tools used to gather, analyze and plot data.

4.1 Emulation vs. simulation

Ideally it would be best to test new Internet protocols in the environment for which it is intended, namely the Internet itself. But this presents a range a problems relating to the fact that the Internet is difficult control, observe and reason about. The Internet is a collection of many independent networks, also called autonomous systems, which have their own rules and routing policies. Flow traversing several independent networks may have their packets subjected to different kind of treatments, and may be transmitted over nodes using different technologies. Because the Internet is so heterogeneous consisting of many technologies, internal network policies and traffic types, it is difficult to reason about data collected from it when testing new protocols.

Emulation and simulation allows control over most of these aspects, making it possible to draw more precise conclusion based on observations. Both network emulation and simulation tries to recreate specific network conditions. The fundamental difference between them is that emulation use actual hardware modified (usually with emulation software) to behave according to the target network setting acting as a substitute, while simulators, such as the ns-2 network simulator, employ network models with properties such as link capacities, network buffers and host dynamics to model the internal state of the network. Usually simulators are implemented as software in user space with few kernel interactions, while emulators are actually transmitted through the TCP/IP stack. The main advantage of using a simulation model is that it is easier to mathematically analyze and reason about. However the abstraction may filter out real world interaction which may have significant effect on the protocol in
the Internet [13]. In this aspect emulation is closer to reality, particularly because hardware and kernel effects are in play making it more realistic, although deterministic.

4.2 Testbed

4.2.1 Topology

The testbed depicted in Figure 4.1 consists of two subnets, each with of two hosts and a network bridge. A router machine connects the two subnets. Each host has a dedicated interface for management traffic. All are connected to a management machine via a switch, used to configure the test machines. The other interfaces are used only for experimental data. The machine ‘xsender’ sends cross BE traffic to ‘kreceiver’, while ‘sender’ sends LBE traffic to the ‘receiver’ host. All senders and receivers run linux servers with kernel version 4 or later. The bridges and the router run kernel version 3 or later.

4.2.2 Background traffic

Flows traversing the Internet are affected by various traffic types. In an emulated environment, its absence may have large impacts on results, because the state of the emulation network may be very different (initially empty network queues, little flow-level multiplexing, homogeneous traffic types) than a real network. To make the environment more realistic,
we decided to add congestion insensitive UDP streams. They a mean inter departure time following the exponential distribution so that take on average 10% of the available bit rate. Using tc we limited them to 15% of the available bit rate. Another alternative would be to capture live traffic and have them replayed by sender hosts using tools such as tcpreplay [40], however this does not take into account differences in the network where the capturing was made, and the emulated network. Packets will be sent at the rate they were sent when captured regardless of congestion.

4.2.3 Buffer sizing

Choosing the size of the router queue size is not trivial, because it affects the queuing delay of packets and may consequently affect the \( cwnd \) growth of the flow to which it belongs [18]. A common approach [cit-37] [7] is to size the queue according to Equation 4.1

\[
\text{Queue size} = \frac{\text{BDP}}{\text{MTU}} \tag{4.1}
\]

This allows a flow to fill the pipe without excessive delay or packet loss due to the bottleneck queue [28][2]. Networks queues does not only exits in router devices. Hosts must also buffer data being either sent or received because of processing limitations. If these queues are too small, throughput may suffer regardless of path capacity. In Linux the size of the TCP buffers can be found in ‘/proc/sys/net/core/wmem_max’ and ‘/proc/sys/net/core/rmem_max’, corresponding to max write and read buffer size respectively. To increase to size of these buffers we can use the Linux command sysctl. A common approach to maximize throughput for bulk data transfer is to set the sender and receiver tcp buffers to twice the BDP of the path [15]. However, this assumes the path is not flow multiplexed, which is rarely the case. Since Linux 2.6, the kernel has facilities for dynamically tune the socket buffer, by accounting for efficiency and delay. The maximum buffer size used by TCP autotuning can be changed by sysctl and is stored in ‘/proc/sys/net/ipv4/tcp_wmem’ and ‘/proc/sys/net/ipv4/tcp_rmem’. We will let Linux do the socket buffer tuning by setting the max size to twice the BDP.

4.2.4 Router and bridge set-up

The bridge machines are configured using the utility ‘brctl’. We use it to bundle together several physical ethernet interfaces into one logical interface, transparent to the network. When frames arrive at one of the bridged interfaces, it is directly forwarded through the link layer (not passing through any upper layers) based on the frame MAC destination address.

The router machine works by enabling ip forwarding. This is accomplished using the command ‘sysctl net.ipv4.ip_forward=1’. When ip forwarding is enabled, incoming packets are forwarded by looking up their destination IP address in the network layer routing table.
4.3 Tools

4.3.1 Linux traffic control

Although it’s possible to limit bandwidth in hardware using tools like ethtool [26] there is a limited range of bandwidth values to choose from (usually 10, 100, 1000Mbps for most NIC’s). With the tool tc (traffic control) [19], bandwidth limitation can be done by traffic shaping. To shape the traffic to a given rate, we employ a data structure called a queuing discipline (qdisc) at egress interfaces. A qdisc controls how packets sent down from the IP layer are queued into the hardware driver queue (the queue dequeued by the NIC for transmission). Each interface has a qdisc associated to it. The driver queue contains pointers to kernel socket buffers (SKB), which is the common data structure used for packets either being sent or received. After a packet has traversed the IP layer, the kernel first enqueues it in the root qdisc. Eventually, assuming that it hasn’t been filtered away or dropped) the kernel will dequeue it from the root qdisc and store it in the driver queue [38]. In other words, qdisc works as packet schedulers. tc also has the ability the create class hierarchies of qdiscs. Such qdisc are referred to as classful qdiscs, otherwise they are called classless. Each classful qdisc contains a class with reference to its subclasses. The kernel always enqueues packets at the root qdisc, which in turn may enqueue it to its child classes. When the kernel calls a dequeue on the root it will, given that it is classful and has children, call dequeue on each if its children (which in turn may call on their own child classes). Packets are never stored in a parent qdisc. Thus a packet enqueued at the root, will traverse down the qdisc tree to a leaf qdisc. A filter can be attached to a classfull qdisc to classify traffic towards different leaf qdiscs. These objects, namely qdiscs, classes and filter, are the basic components of tc, and make advanced control of traffic, like shaping, filtering, policing and scheduling possible. To limit the queue length of the router we use the leaf qdisc pfifo. The pfifo is a regular FIFO queue with a length bounded by a given number of packets. When it is full, incoming packets will be discarded. Queuing delay depends largely on bandwidth and queue size. Therefore we can use pfifo to limit the queuing delay of the router. The bash script in listing 4.1 is used to configure the bottleneck bandwidth as well as the queue size at the router machine. The queue size is calculated using equation 4.1 which is explained in section 3.1.3. The script is used for both experimental traffic interfaces, creating a symmetric path between senders and receivers.

Listing 4.1: script for traffic shaping at the router

```
#!/bin/bash
#$1, $2 and $3 are substituted for the interface, rate and queue length respectively
tc qdisc del dev $1 root
tc qdisc add dev $1 root handle 1: htb default 10
tc class add dev $1 parent 1: classid 1:10 htb rate $2Mbit
```
4.3.2 netem

Besides queuing delay, we need to vary the network path delay. In our testbed propagation delay from senders to receivers is very small (sub millisecond) and does not represent realistic conditions of large spanning networks like WANs. Packets must therefore be artificially delayed. To do this we use the network emulation module netem [14]. Netem comes as an extension to tc, and is therefore used in conjunction with it. Netem is used to artificially add delay to every packet or increase loss rate, reordering or duplication to packet flows. Although it can emulate other WAN factors like packet loss, we don’t use this facility, because we are interested in losses incurred from congestion as opposed to losses incurred from unreliable media. The bash script in 4.2 is used on the bridge interfaces connected to the router. Delay is symmetric in both directions.

Listing 4.2: script for delay emulation at bridge machines

```
#!/bin/bash
#$1 and $2 are substituted for the interface and
#emulation delay respectively

tc qdisc del dev $1 root

tc qdisc add dev $1 root netem delay $2 ms
```

4.4 Pitfalls

An emulated environment must be properly configured if it is to work as a reasonable WAN substitute for experimental traffic. Following are the issues we identified which might have affected our results.

4.4.1 Netem and tc

According to [34, 35] netem doesn’t work well with other tc features and may give unreliable results when combined, this is why we use the bridge machines to do delay emulation.

4.4.2 Hardware offloading

Another issue is hardware offloading. Segmentation offloading is a hardware optimization to give the CPU less overhead placing data in segments headers. Instead it can pass off data to the NIC in chunks to let it do job. At the receiver side segmentation offloading combines packet from the same stream before it passes further up the protocol stack. This reduces the number of interrupts to the CPU. We have decided to disable all NIC offloading that can affect TCP. In particular we disabled TSO at the sender side and GRO at the receiver side. The bash script in listing disables segmentation offloading for both sent and received segments.
4.4.3 Kernel clock frequency

When compiling the Linux kernel it is possible to configure the granularity of the kernel system timer. The system timer interrupts the kernel at regular intervals to do process scheduling. Increasing the HZ configuration value incurs more overhead on the CPU, but makes the system more responsive. Having a HZ value of 250, which is default in many architectures, gives a precision of \( \frac{1}{250} = 4 \) milliseconds. This may influence delay emulation like netem where the tolerance will be +/- 4ms. We have compiled the kernels on the bridges and router to 1000HZ to minimize such variation.

4.4.4 clock drift

The internal clock in computers tend to ‘drift’ away from real time (and consequently each other) because they run at rates different than the definition of a second.\(^1\) While this makes little difference in the short term, the longer a machine runs the more it will drift away from real time. To mitigate this problem, all machines run the Network Time Protocol (NTP) [27] to have system time updated regularly. NTP works by having NTP client query servers for local time. The servers are hierarchically organized in stratums, which a number indication their connection level proximity to stratum 1 servers. Stratum 1 server use very precise time measuring techniques, like atomic clocks or GPS. We use a nearby server as reference instead of a remote one to avoid imprecise replies due to asymmetric delays.

4.5 Metrics

4.5.1 Throughput

In this thesis our focus will be to look at how much throughput a LBE flow utilize in the presence of BE flows as-well as other LBE flows. We will also investigate to which extent LBE flows utilize spare bandwidth when there are no BE flows traversing a common path. We will consider cases where LBE flows are initially present, and when BE flows initially occupies the bottleneck queue. In the former case LBE flows must adapt to the onset of BE traffic increasing the delay, while in the latter case delay is already

---

\(^1\)The SI definition of a seconds is: ‘the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium 133 atom’ [29]. This is the basic property used in atomic clocks.
high. We calculate throughput at the sender for a time interval $i$ using the formula given in Equation 4.2

$$T_i = \frac{B_i}{1000 \times 8 \Delta t_{i-1,i}}$$

where $B_i$ is the nr of bytes sent in interval $i$, and $\Delta t_{i-1,i}$ is the elapsed time between the last measure sample and the current.

4.5.2 Fairness

We use Jain’s Fairness Index as a metric for fairness. We are interested in fairness among LBE flows, when CDG LBE flows operate in absence of BE, to see if they scavenge unused capacity fairly among themselves.

4.6 Traffic generation and analyzes tools

4.6.1 iperf

We use the tool ‘iperf’ to generate greedy traffic. Iperf is a network testing tool used to measure available bandwidth between two hosts. When using tcp, iperf operates on the application layer by injecting as much data as it can into the transport layer. It reports average throughput in time intervals and as-well average throughput of the complete run. It uses an server/client architecture where the server and client instance is running on the receiver and sender respectively. Although it can generate bi-directional traffic, we will only generate uni-directional traffic from sender to receiver.

4.6.2 D-ITG

For generating background traffic we use the ‘Distributed Internet Traffic Generator’ platform. It works in a distributed manner with managing, logging, decoding, sending and receiving components spread out among hosts in the system under test. We will only use its basic components ‘ITGSend’ and ‘ITGRecv’ to create flows between two sender and receiver hosts. It uses stochastic processes to vary ITT and packet size. We will generate packets with ITT following an exponential distribution, while we will keep packet sizes constant.

4.6.3 tcpdump

We use a packet sniffer tool called tcpdump to capture traffic on both sender and receiver end. Tcpdump stores packets received at a specified interface in a ‘pcap’ format, which can later be read by applications using the libpcap library. Tcpdump has the ability the employ filters to selectively record traffic of interest. Although our testbed uses dedicated channels for experimental traffic, we use filters to remove background traffic from the traces.
4.6.4 flow_stat

We wrote a libpcap-based tool we called flow_stat to extract statistics from our capture traces. Given a trace file, an averaging interval, a flow filter, and a test duration, it outputs a table with throughput averages for specified flows using Equation 4.2. We use these tables to generate time series plot in MATLAB.
Chapter 5

Experimental evaluation

This chapter presents the results of our evaluation. When we refer to CDG LBE-ness, we implicitly mean the LBE-ness exhibited by CDG towards CUBIC. The goals of the evaluation are to answer the following questions.

1. How is the degree of CDG LBE-ness affected by varying its G parameter.

2. How does changes in base RTT and BDP affect CDG LBE-ness.

3. Does the start-up order of CDG and CUBIC affect CDG LBE-ness.

4. How fair does CDG remain towards itself when operating in LBE-mode

We have conducted three sets of experiments. All experiments have been repeated 10 times. Following are a description of each set. The sections 5.2, 5.3 and 5.4 present each of the experiments. Each section contains a general description of the experiment, followed by the listings of the relevant experimental parameters of the emulation test-bed. The first list ‘Network parameters’, lists parameter of the network emulation, followed by lists describing parameters for traffic generation at the sender machines. Next comes the plotted data along with statistics listed in tables. We summarize our findings in the subsections called ‘Key observations’. All plots are generated with the MATLAB framework 2016a. We use the plots that we find illustrative of our arguments and discussion of CDG’s LBE-ness. The error bars of the plots are based on the maximum and minimum observations. The rest of the plots are included in the appendix.

Experiment set 1: This experiment is designed to see how the relative throughput of CDG traffic responds to competing CUBIC cross traffic as a function of the G scaling parameter. Both flow types are subjected to the same network conditions, where RTT and bottleneck queue size is varied. CDG and CUBIC traffic is generated from separate hosts. The CUBIC flows are started first, followed later by the CDG flows, and then run to the end of the experiment.
5.1 Varying the G scaling parameter in Linux

The Linux implementation of CDG uses the term ‘backoff factor’ instead for the G scaling parameter. The backoff factor is a parameter in the form of an integer. This allows us to specify fractional values of G (e.g. G=2.5) [23] The backoff factor is calculated using Equation 5.1.

\[
\text{backoff factor} = \frac{1000}{\text{Window} \times G} \quad (5.1)
\]

Where Window is the size of the \( \tau_{\text{min,max}} \) smoothing window. A downside of this conversion method is that as the backoff factor is decreased, the step between G increases multiplicatively, (e.g. you cannot specify a G value between 30 and 60, or 60 and 120). The tables 5.1 and 5.2 shows the relevant G factors we use in our experiments and the equivalent backoff factor value. The probability of CDG backing off is determined by equation 3.8. When the delay gradient \( \bar{g}_n \) increase the expression \( e^{-\bar{g}_n/G} \) decreases and consequently the back off probability P[Back-off]. When we increase G, the expression \( \bar{g}_n/G \) grows slower resulting in a lower probability of back-off.

5.2 Experiment set 1

In this experiment we analyze the competition between two greedy CUBIC flows with two greedy CDG flows using relative throughput. We vary
emulation delay at both bridge machines (see figure 4.1, so that delay is symmetric in both directions (e.g. upstream and downstream traffic is delayed equally). For each delay setting we size the buffer to hold a BDP worth of packets according to Equation 4.1. All CDG’s coexistence heuristics are disabled CUBIC runs with its default Linux settings.

We study how varying the G scaling parameter affects how often CDG backs off, and consequently yields to CUBIC when CUBIC is present and how RTT affects CDG backs off. Each experiment can be described by the combination of one of the six emulation delays together with one of the 20 G parameters used by the two CDG flows. Thus there are a total of 120 experiments in experiment set 1, each running for 5 minutes. Each experiment is started with events taking place in the following order.

1. At the beginning of the experiment a single CUBIC flow is started along with the UDP background traffic
2. The experiment waits for 30 seconds, plus a pseudo-random value between 0-5 seconds
3. After have waited, a second CUBIC flow is initiated
4. The experiments wait 30 seconds
5. A single CDG flow is started with a given G parameter
6. The experiment waits for another 30 seconds plus a random delay between 0-5 seconds
7. Another CDG flow is started with the same G parameter as the first CDG flow
8. The experiment runs until the end (5 minutes after the start of the first CUBIC flow)

5.2.1 Parameter space

Following is a description of the parameters of the experiment.

Network parameters

- Bottleneck rate: 10Mbit/s
- Background traffic: 1Mbit/s
- RTT:{20, 50, 100, 150, 200, 300}ms
- BDP:{0.2, 0.5, 1, 1.5, 2, 3}

sender parameters

- Congestion control: CDG, with no coexistence heuristics
- Traffic type: Greedy LBE

43
• Number of flows: 2
• G scaling parameters: {0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 6.0, 7.5, 10.0, 15.0, 20.0, 25.0, 30.0, 60.0, 120.0}
• Flow startup offset: 60s, 90s

xsender parameters
• Congestion control: CUBIC, default settings
• Traffic type: Greedy BE
• Number of flows: 2
• Flow startup offset: 0s, 30s

5.2.2 Key observations
Figure 5.1 shows a time series plot of the first experiment where G = 0.1 and RTT = 20ms. After both CUBIC flows have started they quickly converge taking up about 90% capacity. When the CDG flows begin they lower the CUBIC flows capacity by 20% until the end of the experiment. For RTT’s equal to 20ms we saw larger back-off responses for smaller than with higher G values as shown in Figure 5.8a. As seen in Figure 5.8b,
5.9a and 5.9b increasing RTT leads to tougher competition for CDG against CUBIC. Note that the scales in Figure ?? are different and get tighter when RTT increases. For 20ms RTT, CUBIC throughput sinks by 33% when G exceeds 5. For higher RTTs CDG throughput generally stays low and needs to increase to higher values of G in order to affect CUBIC throughput. We can see from figures 5.6a to 5.6d that increasing G when RTT is low, results in CUBIC and CDG nearly converges already at G = 3. On the other hand when the RTT is much higher as seen in figures 5.6e to 5.6h CDG will usually not take a larger share than 10%, even for values of G. Figure 5.5 is a 3d plot of CUBIC aggregate throughput against G values and RTT’s. Contrary to what we would except we can that as G gets higher CUBIC uses less throughput for RTT=50ms than for RTT=20ms. Since CDG is acts more like NewReno when G is high we compared CUBIC and NewReno in similar setting with the same RTTs and BDP values. We also tried run single instances of CDG together with one more CUBIC flow(s) to see if any interaction with CUBIC itself would be detrimental its aggregate throughput. Neither gave any indications to being the cause. When we turned off the UDP background traffic however, CUBIC would be monotonically larger for increasing RTTs for larger values of G, including G = 120. This led us to conclude that the background traffic visibly affect CUBIC BE flows when RTT is near 50ms, however we do not know why this happens.

5.2.3 Why RTT affects CDG’s fraction of throughput

Although CDG uses a larger AIMD $\beta$ than NewReno, it calculates the new cwnd the same way when backing off due to packet loss. Since higher G values give a lower probability of back-off from rising delay gradients as expressed in Equation 5.2, we except CDG to act more like NewReno when G increases.

$$\lim_{G \to \infty} P[\text{Back-off}] = 0 \quad (5.2)$$

If the shadow window is disabled CDG effectively uses an AIMD $\beta$ of 0.7. Because CDG is similar to NewReno in this way, it also suffer from some of the same weaknesses as NewReno. Since NewReno increases cwnd by only one segment for each ACK it receives during CA, higher RTTs leads to poorer compete-ability against high speed TCP variants such as CUBIC which is RTT independent, relying instead on time between congestion events to control cwnd increase.
(a) Aggregate CUBIC share of capacity for RTT = 20ms when CUBIC starts first

(b) Aggregate CUBIC share of capacity for RTT = 50ms when CUBIC starts first
(a) Aggregate CUBIC share of capacity for RTT = 100ms when CUBIC starts first

(b) Aggregate CUBIC share of capacity for RTT = 150ms when CUBIC starts first
(a) Aggregate CUBIC share of capacity for RTT = 200ms when CUBIC starts first

(b) Aggregate CUBIC share of capacity for RTT = 300ms when CUBIC starts first
5.3 Experiment set 2

This experiment is very similar to Experiment 1. The only difference is the startup order of flows. LBE flows are started initially, followed by BE flows. All other parameters are the same. In the previous experiment, BE traffic was already occupying the bottleneck queue when LBE flows entered the stage. In this experiment set we wish to see how established and stable LBE back-off in response to BE traffic entering later.

The sequence of events is as follows:

1. At the beginning of the experiment a single CDG flow with a given G parameter is started along with the UDP background traffic.
2. The experiment waits for 30 seconds, plus a pseudo-random value between 0-5 seconds.
3. After waiting a second CDG flow is initiated, with the same G parameter.
4. The experiments wait 30 seconds.
5. A single CUBIC flow is started.
6. The experiment waits for another 30 seconds plus a random delay between 0-5 seconds.
7. Another CUBIC flow is started.
8. The experiment runs until the end (5 minutes after the start of the first CDG flow).
5.3.1 Parameter space

Following is a description of the parameters of the experiment.

Network parameters

- Bottleneck rate: 10Mbit/s
- Background traffic: 1Mbit/s
- RTT: {20, 50, 100, 150, 200, 300} ms
- BDP: {0.2, 0.5, 1, 1.5, 2}

Sender parameters

- Congestion control: CDG, with no coexistence heuristics
- Traffic type: Greedy LBE
- Number of flows: 2
- G scaling parameters: {0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 6.0, 7.5, 10.0, 15.0, 20.0, 25.0, 30.0, 60.0, 120.0}
- Flow startup offset: 0s, 30s

Xsender parameters

- Congestion control: CUBIC, default settings
- Traffic type: Greedy BE
- Number of flows: 2
- Flow startup offset: 60s, 90s
5.3.2 Key observations

By comparing with the figures of the previous section, we can see that startup order does not affect CDG LBE-ness. However, the time it takes for CDG flows to stabilize after BE traffic appear is a function of G and base RTT, for which higher values mean slower convergence to the mean throughput after back-off. We can also see that background traffic still affects CUBIC visibly at 50ms RTT.

5.4 Experiment set 3

To now we have only varied the G scaling parameter to evaluate CDG LBE behavior towards BE CUBIC flows. While this is the main point, it is also important to evaluate intra-protocol fairness among CDG LBE flows in the absence of BE traffic. We also wish to see how changing the AIMD $\beta$ affects fairness and flow stability. In this experiment events take place in the following order. Initially we show some plots to illustrate how throughput share between flows vary with $\beta$. Lastly we present plots of the last 150 seconds of experiments showing Jain’s Fairness Index (JFI) of the flows against time.

1. At the beginning of the experiment a single CDG flow with a given G parameter and $\beta$ AIMD parameter is started along with the UDP background traffic.
2. The experiment waits for 30 seconds.
3. After have waiting a second CDG flow is initiated, with same parameters as flow 1.
4. The experiments wait another 30 seconds.
5. A third CDG flow is started with the same parameters as flow 1.
6. The experiment runs until the end (5 minutes after the start of the first CUBIC flow).

5.4.1 Parameter space

Following is a description of the parameters of the experiment.

Network parameters

- Bottleneck rate: 10Mbit/s
- Background traffic: 500Kbit/s
- RTT: 150ms
- Queue length: 125 packets
(a) Aggregate CUBIC share of capacity for RTT = 20ms when CDG starts first

(b) Aggregate CUBIC share of capacity for RTT = 50ms when CDG starts first
(a) Aggregate CUBIC share of capacity for RTT = 100ms when CDG starts first
(b) Aggregate CUBIC share of capacity for RTT = 150ms when CDG starts first
(a) Aggregate CUBIC share of capacity for RTT = 200ms when CDG starts first

(b) Aggregate CUBIC share of capacity for RTT = 300ms when CDG starts first
sender parameters

- Congestion control: CDG, with no coexistence heuristics
- Traffic type: Greedy LBE
- Number of flows: 3
- G scaling parameters: \{0.1, 3.0, 7.5, 20.0, 120.0\}
- AIMD $\beta$:\{0.2, 0.4, 0.6, 0.8, 1\}
- Flow startup offset: 0s, 30s, 60s

5.4.2 Key observations

Figures 5.13 to 5.17 are plots of three CDG flow, with G set to the default value 3, running in parallel. The plots use 1 seconds sampling interval of throughput to show the fluctuation. As we can be seen see the figures, low $\beta$ values tends to make CDG flows more unstable when running together. This may be due to more aggressive back-off from one flow creates a higher capacity vacuum that is sensed by the other flows when the delay gradients decrease rapidly for the other flows, decreasing back-off probability for at least an RTT. If the flow backing-off, is pushed below ssthresh SS it may try to compensate for its loss, but this is partly mitigated by Hybrid Slow Start implemented in Linux CDG. This may explain the
CUBIC / CDG Bottleneck = 10Mbps, G = 0.1, base RTT: 20ms

(a)

CUBIC / CDG Bottleneck = 10Mbps, G = 1.0, base RTT: 20ms

(b)

CUBIC / CDG Bottleneck = 10Mbps, G = 0.1, base RTT: 300ms

(c)

CUBIC / CDG Bottleneck = 10Mbps, G = 1.0, base RTT: 300ms

(d)

CUBIC / CDG Bottleneck = 10Mbps, G = 7.5, base RTT: 300ms

(e)

CUBIC / CDG Bottleneck = 10Mbps, G = 3.0, base RTT: 300ms

(f)
Homogeneous Capacity Sharing, Bottleneck = 10Mbps, G = 3.0, beta backoff = 0.2, base RTT: 150ms

Figure 5.13

Homogeneous Capacity Sharing, Bottleneck = 10Mbps, G = 3.0, beta backoff = 0.4, base RTT: 150ms

Figure 5.14
Homogeneous Capacity Sharing, Bottleneck = 10Mbps, G = 3.0, beta backoff = 0.6, base RTT: 150ms

Figure 5.15

Homogeneous Capacity Sharing, Bottleneck = 10Mbps, G = 3.0, beta backoff = 0.8, base RTT: 150ms

Figure 5.16
highly fluctuating pattern seen in the previous figures having smaller $\beta$ values. Consequently fairness between flows may be affected as seen in Figures ?? to ?? . The G parameter however does not seem to influence fairness to a significant degree, unless $\beta$ is low. As long as CDG flows backs-off with the same probability, fairness should not be compromised. The CDG’s default back-off $\beta$ of 0.7 ensures that the available capacity will be fairly stable. The following are time series plots of JFI between three CDG flows. A noticeable scenario is when $G=0.1$ and $\beta = 1$. Here JFI is near 1 most of the time. After all flows converge they and they use nearly a third of the capacity each. Because there is less spare capacity (about 5%), a flow with a low G value will not increase much before backing-off. However, since all back-offs are small it is maintained in equilibrium among with the other CDG flows. Figures 5.25 and 5.26 show the difference between an increase in $\beta$ from 0.8 to 1. We the flows with parameters $G = 0.1$ and $\beta = 1$ scavenges spare bandwidth much better than the flows with $G = 0.1$ and $\beta = 0.8$. Generally we saw for all G values that higher $\beta$ gives better utilization of capacity. However as G gets higher the difference, with respect to varying $\beta$ diminishes.
(a) CDG Homogeneous Capacity Sharing

(b) CDG Homogeneous Capacity Sharing
(a) CDG Homogeneous Capacity Sharing

(b) CDG Homogeneous Capacity Sharing

CDG HCS Fairness Index $G=0.1$ backoff $\beta=0.6$

CDG HCS Fairness Index $G=0.1$ backoff $\beta=0.8$
(a) CDG Homogeneous Capacity Sharing

(b) CDG Homogeneous Capacity Sharing
CDG HCS Fairness Index $G=7.5$ backoff beta $=0.4$

(a) CDG Homogeneous Capacity Sharing

CDG HCS Fairness Index $G=7.5$ backoff beta $=0.6$

(b) CDG Homogeneous Capacity Sharing
(a) CDG Homogeneous Capacity Sharing

(b) CDG Homogeneous Capacity Sharing
(a) CDG Homogeneous Capacity Sharing
CDG HCS Fairness Index $G = 120.0$ backoff beta $= 0.4$

(b) CDG Homogeneous Capacity Sharing
CDG HCS Fairness Index $G = 120.0$ backoff beta $= 0.6$
(a) CDG Homogeneous Capacity Sharing

(b) CDG Homogeneous Capacity Sharing
CDG HCS, Bottleneck = 10Mbps, G = 0.1, beta backoff = 0.8, base RTT: 150ms

Figure 5.25: Three CDG flows with G = 0.1 and Beta = 0.8
Figure 5.26: Three CDG flows with G = 0.1 and Beta = 1
Chapter 6

Conclusion and further work

We stated in the introduction that we would evaluate CDG’s performance as a LBE CC against the default High Speed TCP variant of CUBIC. We have tuned CDG’s LBE-ness by disabling its coexistence heuristic and vary the G parameter. We also looked at intra-protocol fairness by varying CDG’s AIMD β parameter. The results indicate that CDG is indeed able to exhibit a significant amount of LBE-ness against CUBIC in several settings. We reduced the RTT to as much as 20ms, where we saw that the fraction of throughput a single CDG flow used could be tuned to down to 10%

This thesis has established CDG as a suitable LBE candidate in a Linux environment with CUBIC CC as default. It can be set to a G value to guarantee that it will on average stay within certain bit rate fraction (see Figures 5.5 and 5.11. This means CDG does not need dynamic tuning as opposed to LEDBAT. Also, it does not suffer from latecomer advantage like LEDBAT does. There are however outstanding questions which we did not include in our scope, or did not have sufficient time to investigate. We have identified the following are candidates for future work on CDG LBE-ness.

CDG LBE against New Reno BE: We did not investigate the effect CDG LBE has on New Reno BE. Some preliminary results showed that increases of G did not always yield to New Reno monotonically. A further work on this may vary a more broader range of parameters to if it is possible to achieve a more predictable function of G back-off against New Reno throughput.

CDG LBE back-off response time: CDG backs-off multiplicatively by a factor of 0.7 every time it backs. Every time it backs-off it waits an RTT before measuring the gradients again. Thus CDG backs off at most one time pr. RTT. In addition CDG, even for small values G, does not always back-off. We believe these are the main factors influencing the time it takes for CDG to reach LBE steady-state (the average bit rate when CDG has backed-off sufficiently for the variation to be within a certain threshold). Given the time at which BE appears \( t_1 \) and the time it takes CDG to reach steady-state LBE mode \( t_2 \) response time be define according to the following equation.
\[ RT = \frac{t_2 - t_1}{BRTT} \]  (6.1)

Where BRTT is the base RTT of the CDG flow and \( t_2 \) is the point at which CDG has reached steady-state. In other words, how RTT it takes for CDG to completely back off.

**Deadline aware LBE CC:** A possibility of a LBE tunable CC is to make a deadline aware functionality which act like LBE as long as it stays within a given deadline (for instance a FTP file transfer to be completed with half an hour). It calculates the bit rate it must maintain in order to complete the job within the deadline. A challenge facing this idea is that there exists several types of CC algorithms reacting to and measuring congestion differently. One possible approach is to evaluate CDG LBE against the most common CCs used and try to find an intersection.

**Delay impact on web traffic:** We only considered greedy BE flows in our evaluation. We don’t know whether CDG LBE backs-off against shorter flows in the manner. This may depend on the volume of traffic, its temporal distribution and how much queuing it introduces to a common bottleneck.
Bibliography


[40] A. Turner. TCP replay. URL: http://tcpreplay.synfin.net/.

Appendices
Appendix A

Plots of the results

A.1 CUBIC / CDG Time series (CUBIC starts first)
CUBIC / CDG, Bottleneck = 10Mbps, G =0.1, base RTT =20ms

CDG 1 stat, mean:0.09, std:0.00 min:0.09 max:0.10
CDG 2 stat, mean:0.10, std:0.02 min:0.08 max:0.18
CUBIC 1 stat, mean:0.34, std:0.01 min:0.33 max:0.36
CUBIC 2 stat, mean:0.35, std:0.01 min:0.32 max:0.39

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =0.5, base RTT =20ms

CDG 1 stat, mean:0.12, std:0.01 min:0.11 max:0.13
CDG 2 stat, mean:0.12, std:0.01 min:0.10 max:0.13
CUBIC 1 stat, mean:0.32, std:0.01 min:0.30 max:0.34
CUBIC 2 stat, mean:0.33, std:0.03 min:0.31 max:0.45

(b)
CUBIC / CDG, Bottleneck = 10Mbps, $G = 1.0$, base RTT = 20ms

**Figure (a):**
- CDG 1 stat, mean: 0.15, std: 0.01 min: 0.13 max: 0.16
- CDG 2 stat, mean: 0.15, std: 0.01 min: 0.14 max: 0.17
- CUBIC 1 stat, mean: 0.29, std: 0.01 min: 0.28 max: 0.32
- CUBIC 2 stat, mean: 0.29, std: 0.01 min: 0.27 max: 0.34

CUBIC / CDG, Bottleneck = 10Mbps, $G = 1.5$, base RTT = 20ms

**Figure (b):**
- CDG 1 stat, mean: 0.17, std: 0.01 min: 0.14 max: 0.19
- CDG 2 stat, mean: 0.17, std: 0.03 min: 0.15 max: 0.29
- CUBIC 1 stat, mean: 0.28, std: 0.01 min: 0.26 max: 0.30
- CUBIC 2 stat, mean: 0.28, std: 0.01 min: 0.25 max: 0.30

81
CUBIC / CDG, Bottleneck = 10Mbps, G =2.0, base RTT =20ms

CDG 1 stat, mean:0.18, std:0.01 min:0.17 max:0.19
CDG 2 stat, mean:0.18, std:0.01 min:0.16 max:0.20
CUBIC 1 stat, mean:0.26, std:0.01 min:0.25 max:0.29
CUBIC 2 stat, mean:0.27, std:0.01 min:0.25 max:0.32

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =2.5, base RTT =20ms

CDG 1 stat, mean:0.18, std:0.01 min:0.17 max:0.20
CDG 2 stat, mean:0.19, std:0.03 min:0.18 max:0.31
CUBIC 1 stat, mean:0.26, std:0.01 min:0.23 max:0.27
CUBIC 2 stat, mean:0.27, std:0.01 min:0.24 max:0.28

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 20ms

CDG 1 stat, mean: 0.19, std: 0.01 min: 0.17 max: 0.21
CDG 2 stat, mean: 0.19, std: 0.01 min: 0.18 max: 0.22
CUBIC 1 stat, mean: 0.25, std: 0.01 min: 0.23 max: 0.27
CUBIC 2 stat, mean: 0.25, std: 0.01 min: 0.24 max: 0.30

CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 20ms

CDG 1 stat, mean: 0.20, std: 0.01 min: 0.18 max: 0.21
CDG 2 stat, mean: 0.19, std: 0.01 min: 0.18 max: 0.21
CUBIC 1 stat, mean: 0.25, std: 0.01 min: 0.24 max: 0.27
CUBIC 2 stat, mean: 0.24, std: 0.01 min: 0.23 max: 0.27

83
CUBIC / CDG, Bottleneck = 10Mbps, G =4.0, base RTT =20ms

CDG 1 stat, mean:0.20, std:0.01 min:0.19 max:0.22
CDG 2 stat, mean:0.20, std:0.01 min:0.19 max:0.24
CUBIC 1 stat, mean:0.24, std:0.01 min:0.24 max:0.26
CUBIC 2 stat, mean:0.24, std:0.01 min:0.23 max:0.26

CUBIC / CDG, Bottleneck = 10Mbps, G =4.5, base RTT =20ms

CDG 1 stat, mean:0.20, std:0.01 min:0.19 max:0.22
CDG 2 stat, mean:0.20, std:0.01 min:0.18 max:0.22
CUBIC 1 stat, mean:0.24, std:0.01 min:0.22 max:0.25
CUBIC 2 stat, mean:0.25, std:0.04 min:0.23 max:0.41
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 20ms

CDG 1 stat, mean:0.20, std:0.01 min:0.18 max:0.21
CDG 2 stat, mean:0.20, std:0.01 min:0.18 max:0.24
CUBIC 1 stat, mean:0.24, std:0.01 min:0.22 max:0.26
CUBIC 2 stat, mean:0.24, std:0.01 min:0.23 max:0.26

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 20ms

CDG 1 stat, mean:0.21, std:0.01 min:0.19 max:0.22
CDG 2 stat, mean:0.20, std:0.01 min:0.19 max:0.24
CUBIC 1 stat, mean:0.24, std:0.01 min:0.23 max:0.26
CUBIC 2 stat, mean:0.24, std:0.04 min:0.22 max:0.42
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 20ms

CDG 1 stat, mean: 0.21, std: 0.01 min: 0.20 max: 0.23
CDG 2 stat, mean: 0.21, std: 0.01 min: 0.20 max: 0.24
CUBIC 1 stat, mean: 0.23, std: 0.01 min: 0.22 max: 0.24
CUBIC 2 stat, mean: 0.24, std: 0.02 min: 0.22 max: 0.32

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 20ms

CDG 1 stat, mean: 0.22, std: 0.01 min: 0.20 max: 0.24
CDG 2 stat, mean: 0.22, std: 0.01 min: 0.20 max: 0.26
CUBIC 1 stat, mean: 0.23, std: 0.01 min: 0.22 max: 0.25
CUBIC 2 stat, mean: 0.23, std: 0.01 min: 0.22 max: 0.25

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 20ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 20ms

(b)

87
CUBIC / CDG, Bottleneck = 10Mbps, G = 25.0, base RTT = 20ms

- CDG 1 stat, mean: 0.22, std: 0.01 min: 0.21 max: 0.23
- CDG 2 stat, mean: 0.22, std: 0.03 min: 0.21 max: 0.36
- CUBIC 1 stat, mean: 0.23, std: 0.01 min: 0.21 max: 0.24
- CUBIC 2 stat, mean: 0.23, std: 0.01 min: 0.22 max: 0.25

CUBIC / CDG, Bottleneck = 10Mbps, G = 30.0, base RTT = 20ms

- CDG 1 stat, mean: 0.22, std: 0.01 min: 0.20 max: 0.23
- CDG 2 stat, mean: 0.22, std: 0.01 min: 0.20 max: 0.25
- CUBIC 1 stat, mean: 0.22, std: 0.01 min: 0.21 max: 0.24
- CUBIC 2 stat, mean: 0.23, std: 0.01 min: 0.21 max: 0.25
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 20ms

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 20ms

(a)

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 0.1, base RTT = 50ms

CDG 1 stat, mean: 0.05, std: 0.00 min: 0.05 max: 0.06
CDG 2 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.10
CUBIC 1 stat, mean: 0.37, std: 0.01 min: 0.33 max: 0.40
CUBIC 2 stat, mean: 0.39, std: 0.02 min: 0.35 max: 0.43

CUBIC / CDG, Bottleneck = 10Mbps, G = 0.5, base RTT = 50ms

CDG 1 stat, mean: 0.07, std: 0.00 min: 0.06 max: 0.08
CDG 2 stat, mean: 0.07, std: 0.00 min: 0.06 max: 0.07
CUBIC 1 stat, mean: 0.37, std: 0.01 min: 0.34 max: 0.39
CUBIC 2 stat, mean: 0.37, std: 0.04 min: 0.34 max: 0.53

(a)
(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 50ms

CDG 1 stat, mean:0.08, std:0.00 min:0.07 max:0.09
CDG 2 stat, mean:0.08, std:0.01 min:0.07 max:0.09
CUBIC 1 stat, mean:0.35, std:0.01 min:0.32 max:0.36
CUBIC 2 stat, mean:0.35, std:0.02 min:0.32 max:0.44

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 50ms

CDG 1 stat, mean:0.10, std:0.01 min:0.08 max:0.12
CDG 2 stat, mean:0.10, std:0.01 min:0.09 max:0.12
CUBIC 1 stat, mean:0.33, std:0.01 min:0.31 max:0.36
CUBIC 2 stat, mean:0.34, std:0.03 min:0.31 max:0.48
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 50ms

(a) CDG 1 stat, mean:0.12, std:0.01 min:0.10 max:0.13
CDG 2 stat, mean:0.13, std:0.04 min:0.10 max:0.29
CUBIC 1 stat, mean:0.31, std:0.01 min:0.29 max:0.34
CUBIC 2 stat, mean:0.32, std:0.02 min:0.29 max:0.35

(b) CDG 1 stat, mean:0.13, std:0.01 min:0.12 max:0.15
CDG 2 stat, mean:0.13, std:0.01 min:0.11 max:0.16
CUBIC 1 stat, mean:0.30, std:0.01 min:0.27 max:0.33
CUBIC 2 stat, mean:0.30, std:0.01 min:0.28 max:0.34
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 50ms

(a) CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 50ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 50ms

- **CDG 1 stat**: mean:0.16, std:0.01 min:0.14 max:0.18
- **CDG 2 stat**: mean:0.16, std:0.01 min:0.15 max:0.18
- **CUBIC 1 stat**: mean:0.27, std:0.01 min:0.25 max:0.29
- **CUBIC 2 stat**: mean:0.27, std:0.01 min:0.26 max:0.30

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 50ms

- **CDG 1 stat**: mean:0.17, std:0.01 min:0.15 max:0.18
- **CDG 2 stat**: mean:0.17, std:0.01 min:0.15 max:0.19
- **CUBIC 1 stat**: mean:0.26, std:0.01 min:0.23 max:0.28
- **CUBIC 2 stat**: mean:0.27, std:0.01 min:0.25 max:0.31

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 50ms

CDG 1 stat, mean:0.17, std:0.01 min:0.15 max:0.18
CDG 2 stat, mean:0.17, std:0.01 min:0.16 max:0.19
CUBIC 1 stat, mean:0.26, std:0.01 min:0.25 max:0.30
CUBIC 2 stat, mean:0.26, std:0.01 min:0.25 max:0.28

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 50ms

CDG 1 stat, mean:0.18, std:0.01 min:0.16 max:0.20
CDG 2 stat, mean:0.18, std:0.02 min:0.15 max:0.25
CUBIC 1 stat, mean:0.25, std:0.01 min:0.24 max:0.27
CUBIC 2 stat, mean:0.25, std:0.01 min:0.24 max:0.28

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 50ms

CDG 1 stat, mean:0.19, std:0.01 min:0.17 max:0.21
CDG 2 stat, mean:0.19, std:0.01 min:0.16 max:0.20
CUBIC 1 stat, mean:0.24, std:0.01 min:0.23 max:0.26
CUBIC 2 stat, mean:0.25, std:0.02 min:0.23 max:0.31

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 50ms

CDG 1 stat, mean:0.20, std:0.01 min:0.19 max:0.22
CDG 2 stat, mean:0.21, std:0.05 min:0.19 max:0.42
CUBIC 1 stat, mean:0.24, std:0.01 min:0.22 max:0.26
CUBIC 2 stat, mean:0.23, std:0.02 min:0.21 max:0.29
CUBIC / CDG, Bottleneck = 10Mbps, G =15.0, base RTT =50ms

CDG 1 stat, mean:0.21, std:0.01 min:0.19 max:0.23
CDG 2 stat, mean:0.21, std:0.03 min:0.19 max:0.31
CUBIC 1 stat, mean:0.23, std:0.01 min:0.21 max:0.27
CUBIC 2 stat, mean:0.22, std:0.01 min:0.21 max:0.26

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =20.0, base RTT =50ms

CDG 1 stat, mean:0.21, std:0.01 min:0.19 max:0.24
CDG 2 stat, mean:0.22, std:0.04 min:0.20 max:0.39
CUBIC 1 stat, mean:0.22, std:0.01 min:0.21 max:0.26
CUBIC 2 stat, mean:0.22, std:0.01 min:0.20 max:0.25

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =25.0, base RTT =50ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =30.0, base RTT =50ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 50ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 50ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =0.1, base RTT =100ms

CDG 1 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.03, std:0.01 min:0.03 max:0.05
CUBIC 1 stat, mean:0.40, std:0.02 min:0.37 max:0.44
CUBIC 2 stat, mean:0.41, std:0.02 min:0.37 max:0.44

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =0.5, base RTT =100ms

CDG 1 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CUBIC 1 stat, mean:0.40, std:0.01 min:0.38 max:0.44
CUBIC 2 stat, mean:0.40, std:0.02 min:0.36 max:0.45

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 100ms

CDG 1 stat, mean: 0.03, std: 0.00 min: 0.03 max: 0.04
CDG 2 stat, mean: 0.03, std: 0.00 min: 0.03 max: 0.04
CUBIC 1 stat, mean: 0.39, std: 0.02 min: 0.36 max: 0.42
CUBIC 2 stat, mean: 0.41, std: 0.03 min: 0.38 max: 0.52

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 100ms

CDG 1 stat, mean: 0.04, std: 0.00 min: 0.03 max: 0.04
CDG 2 stat, mean: 0.04, std: 0.00 min: 0.03 max: 0.05
CUBIC 1 stat, mean: 0.40, std: 0.01 min: 0.38 max: 0.42
CUBIC 2 stat, mean: 0.38, std: 0.01 min: 0.36 max: 0.40
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 100ms

(a) CDG 1 stat, mean:0.04, std:0.00 min:0.04 max:0.05
CDG 2 stat, mean:0.04, std:0.00 min:0.04 max:0.05
CUBIC 1 stat, mean:0.38, std:0.02 min:0.34 max:0.42
CUBIC 2 stat, mean:0.40, std:0.02 min:0.37 max:0.45

(b) CDG 1 stat, mean:0.05, std:0.01 min:0.04 max:0.06
CDG 2 stat, mean:0.05, std:0.01 min:0.04 max:0.09
CUBIC 1 stat, mean:0.37, std:0.02 min:0.34 max:0.43
CUBIC 2 stat, mean:0.39, std:0.03 min:0.34 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 100ms

CDG 1 stat, mean:0.07, std:0.01 min:0.06 max:0.08
CDG 2 stat, mean:0.07, std:0.01 min:0.05 max:0.08
CUBIC 1 stat, mean:0.37, std:0.02 min:0.33 max:0.40
CUBIC 2 stat, mean:0.37, std:0.02 min:0.33 max:0.43

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 100ms

CDG 1 stat, mean:0.07, std:0.01 min:0.06 max:0.08
CDG 2 stat, mean:0.07, std:0.01 min:0.06 max:0.08
CUBIC 1 stat, mean:0.36, std:0.02 min:0.32 max:0.39
CUBIC 2 stat, mean:0.36, std:0.02 min:0.33 max:0.39

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 100ms

CDG 1 stat, mean:0.07, std:0.01 min:0.06 max:0.09
CDG 2 stat, mean:0.08, std:0.01 min:0.06 max:0.11
CUBIC 1 stat, mean:0.36, std:0.02 min:0.31 max:0.39
CUBIC 2 stat, mean:0.37, std:0.01 min:0.33 max:0.39

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 100ms

CDG 1 stat, mean:0.08, std:0.01 min:0.07 max:0.10
CDG 2 stat, mean:0.09, std:0.01 min:0.08 max:0.11
CUBIC 1 stat, mean:0.34, std:0.01 min:0.32 max:0.37
CUBIC 2 stat, mean:0.35, std:0.01 min:0.33 max:0.38

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 100ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 100ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 100ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 100ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, $G = 25.0$, base RTT = 100ms

![Graph](image)

CUBIC / CDG, Bottleneck = 10Mbps, $G = 30.0$, base RTT = 100ms

![Graph](image)
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 100ms

(a) 

CDG 1 stat, mean:0.16, std:0.01 min:0.14 max:0.19
CDG 2 stat, mean:0.16, std:0.02 min:0.12 max:0.19
CUBIC 1 stat, mean:0.27, std:0.02 min:0.24 max:0.33
CUBIC 2 stat, mean:0.27, std:0.02 min:0.24 max:0.33

(b) 

CDG 1 stat, mean:0.17, std:0.01 min:0.15 max:0.20
CDG 2 stat, mean:0.16, std:0.01 min:0.14 max:0.18
CUBIC 1 stat, mean:0.27, std:0.02 min:0.23 max:0.30
CUBIC 2 stat, mean:0.27, std:0.02 min:0.24 max:0.34
CUBIC / CDG, Bottleneck = 10Mbps, G = 0.1, base RTT = 150ms

CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.02
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.02
CUBIC 1 stat, mean: 0.40, std: 0.01, min: 0.39, max: 0.43
CUBIC 2 stat, mean: 0.43, std: 0.02, min: 0.41, max: 0.52

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 0.5, base RTT = 150ms

CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.02
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.02
CUBIC 1 stat, mean: 0.40, std: 0.02, min: 0.36, max: 0.43
CUBIC 2 stat, mean: 0.42, std: 0.02, min: 0.39, max: 0.46

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 150ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 150ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, $G = 2.0$, base RTT = 150ms

(a) CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CUBIC 1 stat, mean: 0.41, std: 0.02, min: 0.36, max: 0.44
CUBIC 2 stat, mean: 0.40, std: 0.02, min: 0.36, max: 0.45

(b) CDG 1 stat, mean: 0.03, std: 0.00, min: 0.02, max: 0.03
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CUBIC 1 stat, mean: 0.41, std: 0.02, min: 0.37, max: 0.44
CUBIC 2 stat, mean: 0.40, std: 0.02, min: 0.37, max: 0.44
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 150ms

(a) CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 150ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 150ms

CDG 1 stat, mean:0.03, std:0.00 min:0.03 max:0.04
CDG 2 stat, mean:0.03, std:0.00 min:0.03 max:0.04
CUBIC 1 stat, mean:0.40, std:0.02 min:0.36 max:0.43
CUBIC 2 stat, mean:0.40, std:0.04 min:0.36 max:0.52

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 150ms

CDG 1 stat, mean:0.03, std:0.00 min:0.03 max:0.04
CDG 2 stat, mean:0.04, std:0.01 min:0.03 max:0.05
CUBIC 1 stat, mean:0.40, std:0.02 min:0.37 max:0.42
CUBIC 2 stat, mean:0.40, std:0.02 min:0.37 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 150ms

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 150ms
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 150ms

CDG 1 stat, mean: 0.05, std: 0.01 min: 0.04 max: 0.06
CDG 2 stat, mean: 0.05, std: 0.01 min: 0.04 max: 0.06
CUBIC 1 stat, mean: 0.38, std: 0.02 min: 0.34 max: 0.41
CUBIC 2 stat, mean: 0.39, std: 0.02 min: 0.36 max: 0.47

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 150ms

CDG 1 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.08
CDG 2 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.09
CUBIC 1 stat, mean: 0.37, std: 0.01 min: 0.34 max: 0.39
CUBIC 2 stat, mean: 0.37, std: 0.02 min: 0.36 max: 0.43

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 150ms

CDG 1 stat, mean:0.07, std:0.01 min:0.06 max:0.09
CDG 2 stat, mean:0.08, std:0.01 min:0.07 max:0.09
CUBIC 1 stat, mean:0.36, std:0.02 min:0.33 max:0.39
CUBIC 2 stat, mean:0.36, std:0.02 min:0.33 max:0.39

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 150ms

CDG 1 stat, mean:0.09, std:0.01 min:0.07 max:0.11
CDG 2 stat, mean:0.08, std:0.01 min:0.07 max:0.10
CUBIC 1 stat, mean:0.35, std:0.01 min:0.32 max:0.38
CUBIC 2 stat, mean:0.34, std:0.02 min:0.31 max:0.37
CUBIC / CDG, Bottleneck = 10Mbps, G =25.0, base RTT = 150ms

CDG 1 stat, mean:0.09, std:0.01 min:0.07 max:0.10
CDG 2 stat, mean:0.09, std:0.01 min:0.08 max:0.12
CUBIC 1 stat, mean:0.33, std:0.01 min:0.31 max:0.35
CUBIC 2 stat, mean:0.35, std:0.02 min:0.33 max:0.38

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =30.0, base RTT =150ms

CDG 1 stat, mean:0.10, std:0.01 min:0.09 max:0.12
CDG 2 stat, mean:0.10, std:0.01 min:0.08 max:0.11
CUBIC 1 stat, mean:0.33, std:0.01 min:0.32 max:0.35
CUBIC 2 stat, mean:0.34, std:0.01 min:0.32 max:0.36

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =60.0, base RTT =150ms

CDG 1 stat, mean:0.11, std:0.01 min:0.09 max:0.13
CDG 2 stat, mean:0.11, std:0.01 min:0.09 max:0.13
CUBIC 1 stat, mean:0.32, std:0.02 min:0.27 max:0.35
CUBIC 2 stat, mean:0.32, std:0.02 min:0.29 max:0.36

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =120.0, base RTT =150ms

CDG 1 stat, mean:0.12, std:0.01 min:0.11 max:0.14
CDG 2 stat, mean:0.12, std:0.01 min:0.09 max:0.15
CUBIC 1 stat, mean:0.31, std:0.02 min:0.28 max:0.34
CUBIC 2 stat, mean:0.31, std:0.02 min:0.27 max:0.35

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 0.1, base RTT = 200ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.03
CUBIC 1 stat, mean:0.41, std:0.01 min:0.39 max:0.43
CUBIC 2 stat, mean:0.42, std:0.02 min:0.40 max:0.45

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 0.5, base RTT = 200ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.03
CUBIC 1 stat, mean:0.42, std:0.01 min:0.41 max:0.45
CUBIC 2 stat, mean:0.41, std:0.02 min:0.40 max:0.42

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 200ms

CDG 1 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.02
CDG 2 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.02
CUBIC 1 stat, mean: 0.41, std: 0.01 min: 0.40 max: 0.44
CUBIC 2 stat, mean: 0.42, std: 0.01 min: 0.40 max: 0.44

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 200ms

CDG 1 stat, mean: 0.02, std: 0.00 min: 0.01 max: 0.02
CDG 2 stat, mean: 0.02, std: 0.00 min: 0.01 max: 0.02
CUBIC 1 stat, mean: 0.42, std: 0.01 min: 0.41 max: 0.44
CUBIC 2 stat, mean: 0.41, std: 0.02 min: 0.39 max: 0.46

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =2.0, base RTT =200ms

(a) CDG 1 stat, mean:0.02, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.02, std:0.00 min:0.01 max:0.02
CUBIC 1 stat, mean:0.41, std:0.01 min:0.40 max:0.44
CUBIC 2 stat, mean:0.42, std:0.01 min:0.38 max:0.43

CUBIC / CDG, Bottleneck = 10Mbps, G =2.5, base RTT =200ms

(b) CDG 1 stat, mean:0.02, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.02, std:0.00 min:0.01 max:0.03
CUBIC 1 stat, mean:0.42, std:0.01 min:0.39 max:0.53
CUBIC 2 stat, mean:0.42, std:0.01 min:0.38 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 200ms

CDG 1 stat, mean:0.02, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.02, std:0.00 min:0.01 max:0.03
CUBIC 1 stat, mean:0.42, std:0.00 min:0.40 max:0.44
CUBIC 2 stat, mean:0.40, std:0.00 min:0.38 max:0.45

CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 200ms

CDG 1 stat, mean:0.02, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.02, std:0.00 min:0.02 max:0.04
CUBIC 1 stat, mean:0.41, std:0.00 min:0.39 max:0.44
CUBIC 2 stat, mean:0.41, std:0.00 min:0.39 max:0.44
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 200ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 200ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 200ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 200ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 200ms

<table>
<thead>
<tr>
<th></th>
<th>Stat</th>
<th>Mean</th>
<th>Std</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDG 1</td>
<td>mean:0.04, std:0.01</td>
<td>0.03</td>
<td>0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDG 2</td>
<td>mean:0.05, std:0.01</td>
<td>0.04</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUBIC 1</td>
<td>mean:0.39, std:0.01</td>
<td>0.37</td>
<td>0.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUBIC 2</td>
<td>mean:0.38, std:0.02</td>
<td>0.35</td>
<td>0.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 200ms

<table>
<thead>
<tr>
<th></th>
<th>Stat</th>
<th>Mean</th>
<th>Std</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDG 1</td>
<td>mean:0.05, std:0.00</td>
<td>0.05</td>
<td>0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDG 2</td>
<td>mean:0.05, std:0.01</td>
<td>0.04</td>
<td>0.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUBIC 1</td>
<td>mean:0.37, std:0.01</td>
<td>0.35</td>
<td>0.38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUBIC 2</td>
<td>mean:0.39, std:0.02</td>
<td>0.35</td>
<td>0.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 200ms
CDG 1 stat, mean: 0.08, std: 0.01 min: 0.06 max: 0.10
CDG 2 stat, mean: 0.08, std: 0.01 min: 0.05 max: 0.09
CUBIC 1 stat, mean: 0.35, std: 0.02 min: 0.34 max: 0.40
CUBIC 2 stat, mean: 0.36, std: 0.02 min: 0.33 max: 0.43

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 200ms
CDG 1 stat, mean: 0.09, std: 0.01 min: 0.07 max: 0.11
CDG 2 stat, mean: 0.09, std: 0.01 min: 0.06 max: 0.13
CUBIC 1 stat, mean: 0.34, std: 0.02 min: 0.32 max: 0.39
CUBIC 2 stat, mean: 0.35, std: 0.02 min: 0.32 max: 0.41
CUBIC / CDG, Bottleneck = 10Mbps, G =0.1, base RTT =300ms

CUBIC / CDG, Bottleneck = 10Mbps, G =0.5, base RTT =300ms
CUBIC / CDG, Bottleneck = 10Mbps, G =1.0, base RTT =300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =1.5, base RTT =300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 300ms

(a)

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CUBIC 1 stat, mean:0.42, std:0.01 min:0.39 max:0.45
CUBIC 2 stat, mean:0.42, std:0.02 min:0.40 max:0.47

CUBIC / CDG, Bottleneck = 10Mbps, G = 2.5, base RTT = 300ms

(b)

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CUBIC 1 stat, mean:0.43, std:0.01 min:0.41 max:0.46
CUBIC 2 stat, mean:0.42, std:0.01 min:0.39 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, $G = 3.0$, base RTT = 300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, $G = 3.5$, base RTT = 300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 300ms

(b)
(a) CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 300ms

(b) CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 300ms
CUBIC / CDG, Bottleneck = 10Mbps, G =15.0, base RTT =300ms

CDG 1 stat, mean:0.02, std:0.00 min:0.02 max:0.02
CDG 2 stat, mean:0.02, std:0.00 min:0.02 max:0.03
CUBIC 1 stat, mean:0.41, std:0.01 min:0.39 max:0.42
CUBIC 2 stat, mean:0.41, std:0.01 min:0.39 max:0.43

CUBIC / CDG, Bottleneck = 10Mbps, G =20.0, base RTT =300ms

CDG 1 stat, mean:0.02, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.02, std:0.00 min:0.02 max:0.03
CUBIC 1 stat, mean:0.40, std:0.01 min:0.38 max:0.42
CUBIC 2 stat, mean:0.42, std:0.01 min:0.39 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, G =25.0, base RTT =300ms

CDG 1 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.03, std:0.00 min:0.02 max:0.04
CUBIC 1 stat, mean:0.39, std:0.01 min:0.38 max:0.42
CUBIC 2 stat, mean:0.42, std:0.01 min:0.39 max:0.43

CUBIC / CDG, Bottleneck = 10Mbps, G =30.0, base RTT =300ms

CDG 1 stat, mean:0.03, std:0.00 min:0.03 max:0.04
CDG 2 stat, mean:0.03, std:0.00 min:0.02 max:0.04
CUBIC 1 stat, mean:0.40, std:0.01 min:0.38 max:0.42
CUBIC 2 stat, mean:0.40, std:0.01 min:0.39 max:0.41
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 300ms

CDG 1 stat, mean:0.05, std:0.01 min:0.04 max:0.06
CDG 2 stat, mean:0.04, std:0.01 min:0.03 max:0.06
CUBIC 1 stat, mean:0.39, std:0.01 min:0.37 max:0.40
CUBIC 2 stat, mean:0.39, std:0.01 min:0.37 max:0.42

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 300ms

CDG 1 stat, mean:0.05, std:0.00 min:0.04 max:0.06
CDG 2 stat, mean:0.05, std:0.01 min:0.04 max:0.07
CUBIC 1 stat, mean:0.38, std:0.01 min:0.35 max:0.39
CUBIC 2 stat, mean:0.38, std:0.01 min:0.37 max:0.40

(b)
A.2  CUBIC / CDG Time series (CDG starts first)
CUBIC / CDG, Bottleneck = 10Mbps, G =0.1, base RTT =20ms

(a) CDG 1 stat, mean:0.09, std:0.00 min:0.09 max:0.10
CDG 2 stat, mean:0.10, std:0.02 min:0.09 max:0.17
CUBIC 1 stat, mean:0.35, std:0.02 min:0.33 max:0.41
CUBIC 2 stat, mean:0.35, std:0.01 min:0.33 max:0.38

(b) CDG 1 stat, mean:0.12, std:0.01 min:0.11 max:0.13
CDG 2 stat, mean:0.12, std:0.01 min:0.11 max:0.13
CUBIC 1 stat, mean:0.32, std:0.02 min:0.31 max:0.40
CUBIC 2 stat, mean:0.33, std:0.04 min:0.29 max:0.47
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 20ms

CDG 1 stat, mean:0.15, std:0.01 min:0.14 max:0.16
CDG 2 stat, mean:0.15, std:0.01 min:0.14 max:0.18
CUBIC 1 stat, mean:0.29, std:0.01 min:0.27 max:0.33
CUBIC 2 stat, mean:0.30, std:0.01 min:0.28 max:0.33

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 20ms

CDG 1 stat, mean:0.17, std:0.01 min:0.14 max:0.19
CDG 2 stat, mean:0.17, std:0.03 min:0.15 max:0.30
CUBIC 1 stat, mean:0.28, std:0.02 min:0.25 max:0.31
CUBIC 2 stat, mean:0.27, std:0.01 min:0.25 max:0.29

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 20ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 2.5, base RTT = 20ms

(b)

CDG 1 stat, mean:0.19, std:0.01 min:0.18 max:0.20
CDG 2 stat, mean:0.19, std:0.03 min:0.17 max:0.31
CUBIC 1 stat, mean:0.25, std:0.01 min:0.23 max:0.27
CUBIC 2 stat, mean:0.27, std:0.01 min:0.25 max:0.29
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 20ms

(a) CDG 1 stat, mean: 0.19, std: 0.01 min: 0.18 max: 0.20
CDG 2 stat, mean: 0.20, std: 0.01 min: 0.18 max: 0.23
CUBIC 1 stat, mean: 0.25, std: 0.01 min: 0.24 max: 0.29
CUBIC 2 stat, mean: 0.25, std: 0.01 min: 0.23 max: 0.30

(b) CDG 1 stat, mean: 0.20, std: 0.01 min: 0.18 max: 0.21
CDG 2 stat, mean: 0.20, std: 0.01 min: 0.19 max: 0.21
CUBIC 1 stat, mean: 0.25, std: 0.01 min: 0.23 max: 0.27
CUBIC 2 stat, mean: 0.25, std: 0.01 min: 0.23 max: 0.27
CUBIC / CDG, Bottleneck = 10Mbps, $G = 4.0$, base RTT = 20ms

- CDG 1 stat, mean:0.20, std:0.01 min:0.19 max:0.21
- CDG 2 stat, mean:0.20, std:0.01 min:0.17 max:0.25
- CUBIC 1 stat, mean:0.25, std:0.01 min:0.24 max:0.26
- CUBIC 2 stat, mean:0.24, std:0.01 min:0.23 max:0.26

CUBIC / CDG, Bottleneck = 10Mbps, $G = 4.5$, base RTT = 20ms

- CDG 1 stat, mean:0.20, std:0.01 min:0.18 max:0.23
- CDG 2 stat, mean:0.21, std:0.01 min:0.19 max:0.23
- CUBIC 1 stat, mean:0.25, std:0.01 min:0.23 max:0.29
- CUBIC 2 stat, mean:0.24, std:0.04 min:0.21 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 20ms

CDG 1 stat, mean: 0.20, std: 0.01, min: 0.19, max: 0.22
CDG 2 stat, mean: 0.20, std: 0.01, min: 0.16, max: 0.24
CUBIC 1 stat, mean: 0.24, std: 0.01, min: 0.23, max: 0.25
CUBIC 2 stat, mean: 0.24, std: 0.01, min: 0.23, max: 0.25

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 20ms

CDG 1 stat, mean: 0.21, std: 0.01, min: 0.20, max: 0.23
CDG 2 stat, mean: 0.21, std: 0.01, min: 0.20, max: 0.23
CUBIC 1 stat, mean: 0.24, std: 0.01, min: 0.21, max: 0.29
CUBIC 2 stat, mean: 0.25, std: 0.05, min: 0.22, max: 0.44

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =7.5, base RTT =20ms

(a) CUBIC / CDG, Bottleneck = 10Mbps, G =10.0, base RTT =20ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 20ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 20ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 25.0, base RTT = 20ms

- CDG 1 stat: mean=0.22, std=0.01, min=0.21, max=0.23
- CDG 2 stat: mean=0.23, std=0.03, min=0.21, max=0.37
- CUBIC 1 stat: mean=0.23, std=0.01, min=0.22, max=0.24
- CUBIC 2 stat: mean=0.22, std=0.01, min=0.20, max=0.24

CUBIC / CDG, Bottleneck = 10Mbps, G = 30.0, base RTT = 20ms

- CDG 1 stat: mean=0.22, std=0.01, min=0.21, max=0.23
- CDG 2 stat: mean=0.22, std=0.01, min=0.21, max=0.28
- CUBIC 1 stat: mean=0.23, std=0.00, min=0.22, max=0.23
- CUBIC 2 stat: mean=0.23, std=0.00, min=0.20, max=0.24

(a)

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 20ms

CDG 1 stat, mean: 0.22, std: 0.01 min: 0.21 max: 0.24
CDG 2 stat, mean: 0.22, std: 0.01 min: 0.20 max: 0.23
CUBIC 1 stat, mean: 0.22, std: 0.01 min: 0.21 max: 0.24
CUBIC 2 stat, mean: 0.22, std: 0.01 min: 0.18 max: 0.25

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 20ms

CDG 1 stat, mean: 0.23, std: 0.01 min: 0.21 max: 0.24
CDG 2 stat, mean: 0.22, std: 0.01 min: 0.20 max: 0.23
CUBIC 1 stat, mean: 0.22, std: 0.01 min: 0.21 max: 0.23
CUBIC 2 stat, mean: 0.23, std: 0.01 min: 0.21 max: 0.24
CUBIC / CDG, Bottleneck = 10Mbps, G =0.1, base RTT =50ms

CDG 1 stat, mean:0.05, std:0.00 min:0.05 max:0.06
CDG 2 stat, mean:0.06, std:0.01 min:0.05 max:0.11
CUBIC 1 stat, mean:0.39, std:0.03 min:0.35 max:0.52
CUBIC 2 stat, mean:0.37, std:0.03 min:0.26 max:0.41

CUBIC / CDG, Bottleneck = 10Mbps, G =0.5, base RTT =50ms

CDG 1 stat, mean:0.06, std:0.00 min:0.06 max:0.07
CDG 2 stat, mean:0.07, std:0.00 min:0.06 max:0.07
CUBIC 1 stat, mean:0.38, std:0.05 min:0.34 max:0.60
CUBIC 2 stat, mean:0.36, std:0.05 min:0.26 max:0.52
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 50ms

CDG 1 stat, mean: 0.14, std: 0.01 min: 0.12 max: 0.15
CDG 2 stat, mean: 0.14, std: 0.01 min: 0.13 max: 0.17
CUBIC 1 stat, mean: 0.29, std: 0.02 min: 0.26 max: 0.37
CUBIC 2 stat, mean: 0.29, std: 0.02 min: 0.22 max: 0.34

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 50ms

CDG 1 stat, mean: 0.15, std: 0.01 min: 0.14 max: 0.17
CDG 2 stat, mean: 0.15, std: 0.01 min: 0.14 max: 0.17
CUBIC 1 stat, mean: 0.28, std: 0.03 min: 0.25 max: 0.40
CUBIC 2 stat, mean: 0.29, std: 0.04 min: 0.26 max: 0.46

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 50ms

CDG 1 stat, mean:0.16, std:0.01 min:0.14 max:0.17
CDG 2 stat, mean:0.16, std:0.01 min:0.14 max:0.18
CUBIC 1 stat, mean:0.27, std:0.02 min:0.25 max:0.32
CUBIC 2 stat, mean:0.28, std:0.02 min:0.22 max:0.31

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 50ms

CDG 1 stat, mean:0.16, std:0.01 min:0.13 max:0.18
CDG 2 stat, mean:0.17, std:0.01 min:0.14 max:0.19
CUBIC 1 stat, mean:0.27, std:0.03 min:0.24 max:0.37
CUBIC 2 stat, mean:0.27, std:0.02 min:0.24 max:0.33
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 50ms

CDG 1 stat, mean:0.17, std:0.01 min:0.15 max:0.18
CDG 2 stat, mean:0.17, std:0.01 min:0.15 max:0.19
CUBIC 1 stat, mean:0.26, std:0.02 min:0.25 max:0.32
CUBIC 2 stat, mean:0.26, std:0.01 min:0.22 max:0.29

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 50ms

CDG 1 stat, mean:0.18, std:0.01 min:0.16 max:0.20
CDG 2 stat, mean:0.19, std:0.02 min:0.16 max:0.26
CUBIC 1 stat, mean:0.26, std:0.01 min:0.24 max:0.28
CUBIC 2 stat, mean:0.25, std:0.01 min:0.23 max:0.27

(b)
CUBIC / CDG, Bottleneck = 10Mbps, $G = 7.5$, base RTT = 50ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, $G = 10.0$, base RTT = 50ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 50ms

CDG 1 stat, mean:0.20, std:0.01 min:0.19 max:0.22
CDG 2 stat, mean:0.21, std:0.03 min:0.19 max:0.32
CUBIC 1 stat, mean:0.23, std:0.01 min:0.21 max:0.29
CUBIC 2 stat, mean:0.23, std:0.01 min:0.18 max:0.25

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 50ms

CDG 1 stat, mean:0.22, std:0.01 min:0.20 max:0.23
CDG 2 stat, mean:0.22, std:0.05 min:0.19 max:0.41
CUBIC 1 stat, mean:0.22, std:0.01 min:0.21 max:0.26
CUBIC 2 stat, mean:0.22, std:0.01 min:0.20 max:0.24
CUBIC / CDG, Bottleneck = 10Mbps, G = 25.0, base RTT = 50ms

CDG 1 stat, mean:0.21, std:0.01 min:0.20 max:0.23
CDG 2 stat, mean:0.22, std:0.02 min:0.20 max:0.32
CUBIC 1 stat, mean:0.22, std:0.01 min:0.21 max:0.24
CUBIC 2 stat, mean:0.22, std:0.01 min:0.20 max:0.24

CUBIC / CDG, Bottleneck = 10Mbps, G = 30.0, base RTT = 50ms

CDG 1 stat, mean:0.22, std:0.01 min:0.20 max:0.24
CDG 2 stat, mean:0.22, std:0.02 min:0.20 max:0.32
CUBIC 1 stat, mean:0.22, std:0.01 min:0.21 max:0.24
CUBIC 2 stat, mean:0.22, std:0.01 min:0.20 max:0.24
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 50ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 50ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =0.1, base RTT =100ms

CDG 1 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.03, std:0.00 min:0.02 max:0.04
CUBIC 1 stat, mean:0.40, std:0.06 min:0.36 max:0.65
CUBIC 2 stat, mean:0.41, std:0.05 min:0.22 max:0.46

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =0.5, base RTT =100ms

CDG 1 stat, mean:0.03, std:0.00 min:0.03 max:0.04
CDG 2 stat, mean:0.03, std:0.00 min:0.03 max:0.04
CUBIC 1 stat, mean:0.41, std:0.06 min:0.36 max:0.66
CUBIC 2 stat, mean:0.40, std:0.05 min:0.22 max:0.44

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 100ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 100ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 100ms

(a) CDG 1 stat, mean: 0.04, std: 0.00, min: 0.04, max: 0.05
CDG 2 stat, mean: 0.04, std: 0.00, min: 0.03, max: 0.05
CUBIC 1 stat, mean: 0.39, std: 0.06, min: 0.35, max: 0.61
CUBIC 2 stat, mean: 0.38, std: 0.05, min: 0.20, max: 0.43

CUBIC / CDG, Bottleneck = 10Mbps, G = 2.5, base RTT = 100ms

(b) CDG 1 stat, mean: 0.05, std: 0.00, min: 0.04, max: 0.06
CDG 2 stat, mean: 0.05, std: 0.01, min: 0.04, max: 0.11
CUBIC 1 stat, mean: 0.40, std: 0.04, min: 0.36, max: 0.56
CUBIC 2 stat, mean: 0.37, std: 0.03, min: 0.25, max: 0.40
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 100ms

CDG 1 stat, mean: 0.06, std: 0.01 min: 0.04 max: 0.07
CDG 2 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.09
CUBIC 1 stat, mean: 0.39, std: 0.05 min: 0.34 max: 0.59
CUBIC 2 stat, mean: 0.36, std: 0.04 min: 0.21 max: 0.41

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 100ms

CDG 1 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.08
CDG 2 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.08
CUBIC 1 stat, mean: 0.38, std: 0.04 min: 0.32 max: 0.54
CUBIC 2 stat, mean: 0.36, std: 0.05 min: 0.19 max: 0.42

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 100ms

(a)

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 100ms

CDG 1 stat, mean: 0.07, std: 0.01 min: 0.06 max: 0.09
CDG 2 stat, mean: 0.07, std: 0.01 min: 0.07 max: 0.10
CUBIC 1 stat, mean: 0.37, std: 0.05 min: 0.33 max: 0.56
CUBIC 2 stat, mean: 0.35, std: 0.03 min: 0.24 max: 0.39

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 100ms

CDG 1 stat, mean: 0.08, std: 0.01 min: 0.07 max: 0.10
CDG 2 stat, mean: 0.09, std: 0.01 min: 0.07 max: 0.11
CUBIC 1 stat, mean: 0.34, std: 0.03 min: 0.30 max: 0.46
CUBIC 2 stat, mean: 0.35, std: 0.03 min: 0.25 max: 0.39

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 100ms

CDG 1 stat, mean: 0.09, std: 0.01 min: 0.07 max: 0.11
CDG 2 stat, mean: 0.10, std: 0.01 min: 0.08 max: 0.11
CUBIC 1 stat, mean: 0.36, std: 0.05 min: 0.29 max: 0.53
CUBIC 2 stat, mean: 0.33, std: 0.04 min: 0.22 max: 0.43

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 100ms

CDG 1 stat, mean: 0.11, std: 0.01 min: 0.08 max: 0.12
CDG 2 stat, mean: 0.11, std: 0.01 min: 0.09 max: 0.13
CUBIC 1 stat, mean: 0.33, std: 0.06 min: 0.28 max: 0.55
CUBIC 2 stat, mean: 0.32, std: 0.04 min: 0.20 max: 0.42

(a)

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =15.0, base RTT =100ms

CDG 1 stat, mean:0.13, std:0.01 min:0.10 max:0.15
CDG 2 stat, mean:0.13, std:0.01 min:0.12 max:0.15
CUBIC 1 stat, mean:0.32, std:0.04 min:0.29 max:0.49
CUBIC 2 stat, mean:0.29, std:0.04 min:0.21 max:0.42

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =20.0, base RTT =100ms

CDG 1 stat, mean:0.14, std:0.01 min:0.12 max:0.16
CDG 2 stat, mean:0.14, std:0.01 min:0.11 max:0.17
CUBIC 1 stat, mean:0.30, std:0.04 min:0.27 max:0.44
CUBIC 2 stat, mean:0.28, std:0.03 min:0.22 max:0.34

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 25.0, base RTT = 100ms

(a) CUBIC / CDG, Bottleneck = 10Mbps, G = 30.0, base RTT = 100ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 100ms

CDG 1 stat, mean:0.16, std:0.01 min:0.14 max:0.18
CDG 2 stat, mean:0.16, std:0.01 min:0.13 max:0.18
CUBIC 1 stat, mean:0.28, std:0.03 min:0.23 max:0.38
CUBIC 2 stat, mean:0.27, std:0.03 min:0.20 max:0.35

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 100ms

CDG 1 stat, mean:0.16, std:0.02 min:0.14 max:0.21
CDG 2 stat, mean:0.17, std:0.01 min:0.15 max:0.20
CUBIC 1 stat, mean:0.27, std:0.03 min:0.24 max:0.35
CUBIC 2 stat, mean:0.26, std:0.02 min:0.19 max:0.29

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 0.1, base RTT = 150ms

CUBIC / CDG, Bottleneck = 10Mbps, G = 0.5, base RTT = 150ms
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 150ms

CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.04
CUBIC 1 stat, mean: 0.43, std: 0.07, min: 0.38, max: 0.71
CUBIC 2 stat, mean: 0.39, std: 0.06, min: 0.16, max: 0.44

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 150ms

CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.04
CUBIC 1 stat, mean: 0.43, std: 0.07, min: 0.38, max: 0.69
CUBIC 2 stat, mean: 0.39, std: 0.06, min: 0.14, max: 0.44

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =2.0, base RTT =150ms

CDG 1 stat, mean:0.02, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.02, std:0.00 min:0.02 max:0.03
CUBIC 1 stat, mean:0.44, std:0.08 min:0.38 max:0.73
CUBIC 2 stat, mean:0.38, std:0.07 min:0.11 max:0.43

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =2.5, base RTT =150ms

CDG 1 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CDG 2 stat, mean:0.03, std:0.00 min:0.02 max:0.03
CUBIC 1 stat, mean:0.43, std:0.07 min:0.39 max:0.69
CUBIC 2 stat, mean:0.38, std:0.06 min:0.13 max:0.42

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 3.0, base RTT = 150ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 3.5, base RTT = 150ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 150ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 150ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 150ms

- CDG 1 stat, mean: 0.03, std: 0.00 min: 0.03 max: 0.04
- CDG 2 stat, mean: 0.04, std: 0.00 min: 0.03 max: 0.04
- CUBIC 1 stat, mean: 0.41, std: 0.08 min: 0.35 max: 0.71
- CUBIC 2 stat, mean: 0.39, std: 0.07 min: 0.13 max: 0.44

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 150ms

- CDG 1 stat, mean: 0.04, std: 0.01 min: 0.03 max: 0.05
- CDG 2 stat, mean: 0.04, std: 0.01 min: 0.03 max: 0.05
- CUBIC 1 stat, mean: 0.41, std: 0.08 min: 0.36 max: 0.71
- CUBIC 2 stat, mean: 0.38, std: 0.06 min: 0.14 max: 0.45
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 150ms

CDG 1 stat, mean:0.05, std:0.01 min:0.04 max:0.06
CDG 2 stat, mean:0.05, std:0.01 min:0.04 max:0.07
CUBIC 1 stat, mean:0.41, std:0.08 min:0.37 max:0.70
CUBIC 2 stat, mean:0.36, std:0.06 min:0.15 max:0.48

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 150ms

CDG 1 stat, mean:0.06, std:0.01 min:0.05 max:0.08
CDG 2 stat, mean:0.06, std:0.01 min:0.04 max:0.10
CUBIC 1 stat, mean:0.39, std:0.07 min:0.32 max:0.64
CUBIC 2 stat, mean:0.36, std:0.05 min:0.17 max:0.41

(b)
(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =15.0, base RTT =150ms

CDG 1 stat, mean:0.08, std:0.01 min:0.05 max:0.09
CDG 2 stat, mean:0.07, std:0.01 min:0.05 max:0.09
CUBIC 1 stat, mean:0.38, std:0.07 min:0.33 max:0.64
CUBIC 2 stat, mean:0.34, std:0.06 min:0.12 max:0.38

(b)

CUBIC / CDG, Bottleneck = 10Mbps, G =20.0, base RTT =150ms

CDG 1 stat, mean:0.09, std:0.01 min:0.07 max:0.11
CDG 2 stat, mean:0.08, std:0.01 min:0.07 max:0.10
CUBIC 1 stat, mean:0.36, std:0.06 min:0.31 max:0.58
CUBIC 2 stat, mean:0.34, std:0.05 min:0.16 max:0.38
CUBIC / CDG, Bottleneck = 10Mbps, G =25.0, base RTT =150ms

CDG 1 stat, mean:0.09, std:0.01 min:0.08 max:0.12
CDG 2 stat, mean:0.09, std:0.01 min:0.08 max:0.12
CUBIC 1 stat, mean:0.36, std:0.05 min:0.32 max:0.55
CUBIC 2 stat, mean:0.32, std:0.04 min:0.16 max:0.35

CUBIC / CDG, Bottleneck = 10Mbps, G =30.0, base RTT =150ms

CDG 1 stat, mean:0.10, std:0.01 min:0.08 max:0.11
CDG 2 stat, mean:0.10, std:0.01 min:0.08 max:0.12
CUBIC 1 stat, mean:0.35, std:0.05 min:0.31 max:0.54
CUBIC 2 stat, mean:0.32, std:0.05 min:0.13 max:0.35
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 150ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 150ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 0.1, base RTT = 200ms

CDG 1 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.02
CDG 2 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.03
CUBIC 1 stat, mean: 0.45, std: 0.07 min: 0.40 max: 0.69
CUBIC 2 stat, mean: 0.39, std: 0.07 min: 0.17 max: 0.44

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 0.5, base RTT = 200ms

CDG 1 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.02
CDG 2 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.03
CUBIC 1 stat, mean: 0.44, std: 0.08 min: 0.39 max: 0.71
CUBIC 2 stat, mean: 0.39, std: 0.08 min: 0.13 max: 0.45

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 1.0, base RTT = 200ms

CDG 1 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.02
CDG 2 stat, mean: 0.01, std: 0.00 min: 0.01 max: 0.02
CUBIC 1 stat, mean: 0.45, std: 0.07 min: 0.40 max: 0.70
CUBIC 2 stat, mean: 0.38, std: 0.06 min: 0.14 max: 0.44

CUBIC / CDG, Bottleneck = 10Mbps, G = 1.5, base RTT = 200ms

CDG 1 stat, mean: 0.02, std: 0.00 min: 0.01 max: 0.02
CDG 2 stat, mean: 0.02, std: 0.00 min: 0.01 max: 0.02
CUBIC 1 stat, mean: 0.45, std: 0.08 min: 0.38 max: 0.74
CUBIC 2 stat, mean: 0.39, std: 0.07 min: 0.14 max: 0.48

(a)

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 200ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 2.5, base RTT = 200ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, $G = 3.0$, base RTT = 200ms

CDG 1 stat, mean:0.02, std:0.00 min:0.02 max:0.02
CDG 2 stat, mean:0.02, std:0.00 min:0.01 max:0.03
CUBIC 1 stat, mean:0.44, std:0.07 min:0.39 max:0.71
CUBIC 2 stat, mean:0.39, std:0.06 min:0.16 max:0.43

CUBIC / CDG, Bottleneck = 10Mbps, $G = 3.5$, base RTT = 200ms

CDG 1 stat, mean:0.02, std:0.01 min:0.02 max:0.04
CDG 2 stat, mean:0.02, std:0.01 min:0.01 max:0.04
CUBIC 1 stat, mean:0.44, std:0.07 min:0.39 max:0.70
CUBIC 2 stat, mean:0.39, std:0.06 min:0.16 max:0.45
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 200ms

CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.03
CUBIC 1 stat, mean: 0.44, std: 0.07, min: 0.39, max: 0.70
CUBIC 2 stat, mean: 0.38, std: 0.07, min: 0.15, max: 0.44

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 200ms

CDG 1 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.02
CDG 2 stat, mean: 0.02, std: 0.00, min: 0.02, max: 0.02
CUBIC 1 stat, mean: 0.43, std: 0.07, min: 0.39, max: 0.69
CUBIC 2 stat, mean: 0.39, std: 0.06, min: 0.16, max: 0.43

(a)

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =5.0, base RTT =200ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =6.0, base RTT =200ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 200ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 200ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =15.0, base RTT =200ms

CDG 1 stat, mean:0.04, std:0.01 min:0.03 max:0.06
CDG 2 stat, mean:0.05, std:0.01 min:0.03 max:0.06
CUBIC 1 stat, mean:0.40, std:0.07 min:0.34 max:0.65
CUBIC 2 stat, mean:0.38, std:0.06 min:0.16 max:0.42

CUBIC / CDG, Bottleneck = 10Mbps, G =20.0, base RTT =200ms

CDG 1 stat, mean:0.05, std:0.01 min:0.04 max:0.07
CDG 2 stat, mean:0.05, std:0.01 min:0.04 max:0.08
CUBIC 1 stat, mean:0.42, std:0.08 min:0.35 max:0.68
CUBIC 2 stat, mean:0.35, std:0.07 min:0.12 max:0.42
CUBIC / CDG, Bottleneck = 10Mbps, G = 25.0, base RTT = 200ms

CDG 1 stat, mean:0.06, std:0.01 min:0.04 max:0.07
CDG 2 stat, mean:0.06, std:0.01 min:0.05 max:0.07
CUBIC 1 stat, mean:0.41, std:0.07 min:0.37 max:0.66
CUBIC 2 stat, mean:0.34, std:0.06 min:0.11 max:0.38

CUBIC / CDG, Bottleneck = 10Mbps, G = 30.0, base RTT = 200ms

CDG 1 stat, mean:0.06, std:0.01 min:0.05 max:0.07
CDG 2 stat, mean:0.06, std:0.01 min:0.05 max:0.07
CUBIC 1 stat, mean:0.41, std:0.07 min:0.36 max:0.67
CUBIC 2 stat, mean:0.34, std:0.06 min:0.11 max:0.39
CUBIC / CDG, Bottleneck = 10Mbps, G =60.0, base RTT =200ms

CDG 1 stat, mean:0.08, std:0.01 min:0.06 max:0.10
CDG 2 stat, mean:0.08, std:0.01 min:0.06 max:0.09
CUBIC 1 stat, mean:0.38, std:0.06 min:0.34 max:0.60
CUBIC 2 stat, mean:0.34, std:0.06 min:0.13 max:0.42

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =120.0, base RTT =200ms

CDG 1 stat, mean:0.09, std:0.01 min:0.08 max:0.11
CDG 2 stat, mean:0.09, std:0.01 min:0.07 max:0.11
CUBIC 1 stat, mean:0.37, std:0.07 min:0.32 max:0.60
CUBIC 2 stat, mean:0.32, std:0.06 min:0.12 max:0.43

(b)

190
CUBIC / CDG, Bottleneck = 10Mbps, $G=0.1$, base RTT = 300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, $G=0.5$, base RTT = 300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =1.0, base RTT =300ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CUBIC 1 stat, mean:0.44, std:0.07 min:0.41 max:0.73
CUBIC 2 stat, mean:0.41, std:0.05 min:0.21 max:0.46

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =1.5, base RTT =300ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CUBIC 1 stat, mean:0.44, std:0.06 min:0.41 max:0.69
CUBIC 2 stat, mean:0.40, std:0.05 min:0.20 max:0.43

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 2.0, base RTT = 300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 2.5, base RTT = 300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G =3.0, base RTT =300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G =3.5, base RTT =300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 4.0, base RTT = 300ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CUBIC 1 stat, mean:0.43, std:0.05 min:0.39 max:0.65
CUBIC 2 stat, mean:0.41, std:0.05 min:0.21 max:0.46

CUBIC / CDG, Bottleneck = 10Mbps, G = 4.5, base RTT = 300ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CUBIC 1 stat, mean:0.44, std:0.06 min:0.41 max:0.69
CUBIC 2 stat, mean:0.40, std:0.05 min:0.21 max:0.43
CUBIC / CDG, Bottleneck = 10Mbps, G = 5.0, base RTT = 300ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.01
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CUBIC 1 stat, mean:0.42, std:0.05 min:0.39 max:0.64
CUBIC 2 stat, mean:0.42, std:0.05 min:0.21 max:0.45

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 6.0, base RTT = 300ms

CDG 1 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CDG 2 stat, mean:0.01, std:0.00 min:0.01 max:0.02
CUBIC 1 stat, mean:0.43, std:0.08 min:0.39 max:0.76
CUBIC 2 stat, mean:0.42, std:0.06 min:0.21 max:0.54

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 7.5, base RTT = 300ms

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 10.0, base RTT = 300ms

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 15.0, base RTT = 300ms

CDG 1 stat, mean:0.02, std:0.00 min:0.02 max:0.04
CDG 2 stat, mean:0.02, std:0.00 min:0.02 max:0.03
CUBIC 1 stat, mean:0.43, std:0.05 min:0.41 max:0.66
CUBIC 2 stat, mean:0.39, std:0.05 min:0.18 max:0.42

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 20.0, base RTT = 300ms

CDG 1 stat, mean:0.03, std:0.01 min:0.02 max:0.07
CDG 2 stat, mean:0.02, std:0.00 min:0.02 max:0.04
CUBIC 1 stat, mean:0.42, std:0.04 min:0.37 max:0.60
CUBIC 2 stat, mean:0.40, std:0.05 min:0.18 max:0.44

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 25.0, base RTT = 300ms

CDG 1 stat, mean: 0.03, std: 0.01 min: 0.02 max: 0.08
CDG 2 stat, mean: 0.03, std: 0.01 min: 0.02 max: 0.06
CUBIC 1 stat, mean: 0.42, std: 0.05 min: 0.38 max: 0.60
CUBIC 2 stat, mean: 0.39, std: 0.06 min: 0.16 max: 0.43

(a)

CUBIC / CDG, Bottleneck = 10Mbps, G = 30.0, base RTT = 300ms

CDG 1 stat, mean: 0.03, std: 0.01 min: 0.02 max: 0.07
CDG 2 stat, mean: 0.03, std: 0.01 min: 0.02 max: 0.07
CUBIC 1 stat, mean: 0.42, std: 0.04 min: 0.38 max: 0.60
CUBIC 2 stat, mean: 0.38, std: 0.05 min: 0.16 max: 0.40

(b)
CUBIC / CDG, Bottleneck = 10Mbps, G = 60.0, base RTT = 300ms

(a)

CDG 1 stat, mean: 0.04, std: 0.01 min: 0.03 max: 0.08
CDG 2 stat, mean: 0.04, std: 0.01 min: 0.04 max: 0.06
CUBIC 1 stat, mean: 0.40, std: 0.05 min: 0.37 max: 0.60
CUBIC 2 stat, mean: 0.38, std: 0.06 min: 0.14 max: 0.41

CUBIC / CDG, Bottleneck = 10Mbps, G = 120.0, base RTT = 300ms

(b)

CDG 1 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.10
CDG 2 stat, mean: 0.06, std: 0.01 min: 0.05 max: 0.08
CUBIC 1 stat, mean: 0.40, std: 0.05 min: 0.37 max: 0.60
CUBIC 2 stat, mean: 0.35, std: 0.06 min: 0.12 max: 0.39
A.3 CDG LBE Homogeneous Capacity Sharing
CDG HCS, Bottleneck = 10Mbps, $G = 0.1$, beta backoff = 0.2, base RTT: 150ms

CDG 1 stat, mean:0.04, std:0.01 min:0.03 max:0.06
CDG 2 stat, mean:0.04, std:0.01 min:0.03 max:0.07
CDG 3 stat, mean:0.04, std:0.01 min:0.03 max:0.06

CDG HCS, Bottleneck = 10Mbps, $G = 0.1$, beta backoff = 0.4, base RTT: 150ms

CDG 1 stat, mean:0.05, std:0.01 min:0.03 max:0.07
CDG 2 stat, mean:0.05, std:0.01 min:0.03 max:0.07
CDG 3 stat, mean:0.05, std:0.01 min:0.03 max:0.06
CDG HCS, Bottleneck = 10Mbps, G = 0.1, beta backoff = 0.6, base RTT: 150ms

(a) CDG 1 stat, mean: 0.05, std: 0.01 min: 0.04 max: 0.08
CDG 2 stat, mean: 0.06, std: 0.01 min: 0.03 max: 0.10
CDG 3 stat, mean: 0.05, std: 0.01 min: 0.03 max: 0.07

(b) CDG 1 stat, mean: 0.09, std: 0.01 min: 0.05 max: 0.13
CDG 2 stat, mean: 0.08, std: 0.01 min: 0.06 max: 0.14
CDG 3 stat, mean: 0.09, std: 0.01 min: 0.06 max: 0.13
CDG HCS, Bottleneck = 10Mbps, G = 0.1, beta backoff = 1, base RTT: 150ms

CDG 1 stat, mean: 0.30, std: 0.01 min: 0.28 max: 0.32
CDG 2 stat, mean: 0.30, std: 0.01 min: 0.28 max: 0.32
CDG 3 stat, mean: 0.30, std: 0.01 min: 0.28 max: 0.33

CDG HCS, Bottleneck = 10Mbps, G = 3.0, beta backoff = 0.2, base RTT: 150ms

CDG 1 stat, mean: 0.21, std: 0.04 min: 0.12 max: 0.32
CDG 2 stat, mean: 0.20, std: 0.04 min: 0.11 max: 0.30
CDG 3 stat, mean: 0.22, std: 0.05 min: 0.13 max: 0.39
CDG HCS, Bottleneck = 10Mbps, G = 3.0, beta backoff = 0.4, base RTT: 150ms

CDG 1 stat, mean: 0.23, std: 0.04 min: 0.13 max: 0.33
CDG 2 stat, mean: 0.22, std: 0.04 min: 0.12 max: 0.34
CDG 3 stat, mean: 0.23, std: 0.04 min: 0.14 max: 0.35

(b)
CDG HCS, Bottleneck = 10Mbps, G =3.0, beta backoff =0.8, base RTT: 150ms

(a)

CDG HCS, Bottleneck = 10Mbps, G =3.0, beta backoff =−1, base RTT: 150ms

(b)
CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = 0.2, base RTT: 150ms

CDG 1 stat, mean: 0.22, std: 0.05 min: 0.12 max: 0.33
CDG 2 stat, mean: 0.23, std: 0.05 min: 0.11 max: 0.38
CDG 3 stat, mean: 0.25, std: 0.06 min: 0.13 max: 0.44

CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = 0.4, base RTT: 150ms

CDG 1 stat, mean: 0.28, std: 0.04 min: 0.18 max: 0.39
CDG 2 stat, mean: 0.24, std: 0.04 min: 0.16 max: 0.36
CDG 3 stat, mean: 0.24, std: 0.04 min: 0.17 max: 0.33
CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = 0.6, base RTT: 150ms

(a)

CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = 0.8, base RTT: 150ms

(b)
CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = -1, base RTT: 150ms

CDG 1 stat, mean: 0.29, std: 0.02 min: 0.24 max: 0.34
CDG 2 stat, mean: 0.31, std: 0.03 min: 0.26 max: 0.36
CDG 3 stat, mean: 0.31, std: 0.02 min: 0.27 max: 0.36

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.2, base RTT: 150ms

CDG 1 stat, mean: 0.26, std: 0.04 min: 0.18 max: 0.36
CDG 2 stat, mean: 0.26, std: 0.04 min: 0.17 max: 0.38
CDG 3 stat, mean: 0.25, std: 0.04 min: 0.15 max: 0.36
CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.4, base RTT: 150ms

(a) CDG 1 stat, mean: 0.28, std: 0.04 min: 0.19 max: 0.39
CDG 2 stat, mean: 0.27, std: 0.04 min: 0.19 max: 0.36
CDG 3 stat, mean: 0.29, std: 0.04 min: 0.17 max: 0.38

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.6, base RTT: 150ms

(b) CDG 1 stat, mean: 0.30, std: 0.03 min: 0.23 max: 0.39
CDG 2 stat, mean: 0.28, std: 0.03 min: 0.22 max: 0.36
CDG 3 stat, mean: 0.31, std: 0.04 min: 0.21 max: 0.40
CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.8, base RTT: 150ms

CDG 1 stat, mean: 0.32, std: 0.03 min: 0.27 max: 0.43
CDG 2 stat, mean: 0.30, std: 0.03 min: 0.23 max: 0.35
CDG 3 stat, mean: 0.29, std: 0.02 min: 0.23 max: 0.34

(a)

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = -1, base RTT: 150ms

CDG 1 stat, mean: 0.33, std: 0.03 min: 0.27 max: 0.41
CDG 2 stat, mean: 0.28, std: 0.03 min: 0.23 max: 0.35
CDG 3 stat, mean: 0.30, std: 0.03 min: 0.22 max: 0.36

(b)
CDG HCS, Bottleneck = 10Mbps, G =120.0, beta backoff =0.2, base RTT: 150ms

CDG stat, mean:0.27, std:0.05 min:0.16 max:0.39
CDG 2 stat, mean:0.31, std:0.06 min:0.18 max:0.47
CDG 3 stat, mean:0.31, std:0.05 min:0.21 max:0.43

CDG HCS, Bottleneck = 10Mbps, G =120.0, beta backoff =0.4, base RTT: 150ms

CDG stat, mean:0.30, std:0.06 min:0.18 max:0.43
CDG 2 stat, mean:0.33, std:0.05 min:0.24 max:0.44
CDG 3 stat, mean:0.28, std:0.03 min:0.21 max:0.34
CDG HCS, Bottleneck = 10Mbps, G =120.0, beta backoff =0.6, base RTT: 150ms

(a)

CDG HCS, Bottleneck = 10Mbps, G =120.0, beta backoff =0.8, base RTT: 150ms

(b)
CDG HCS, Bottleneck = 10Mbps, G = 120.0, beta backoff =~1, base RTT: 150ms

CDG 1 stat, mean: 0.32, std: 0.02, min: 0.29, max: 0.39
CDG 2 stat, mean: 0.29, std: 0.02, min: 0.25, max: 0.33
CDG 3 stat, mean: 0.30, std: 0.02, min: 0.25, max: 0.35

Figure A.133
CDG HCS Fairness Index $G = 120.0$

Caption

CDG HCS, Bottleneck = 10Mbps, $G = 3.0$, beta backoff = 0.2, base RTT: 150ms

CDG 1 stat, mean: 0.21, std: 0.04 min: 0.12 max: 0.32
CDG 2 stat, mean: 0.20, std: 0.04 min: 0.11 max: 0.30
CDG 3 stat, mean: 0.22, std: 0.05 min: 0.13 max: 0.39

Figure A.136
Figure A.137

Figure A.138
CDG HCS, Bottleneck = 10Mbps, $G = 3.0$, beta backoff $= 0.8$, base RTT: 150ms

CDG 1 stat, mean:0.27, std:0.03 min:0.19 max:0.33
CDG 2 stat, mean:0.27, std:0.03 min:0.19 max:0.34
CDG 3 stat, mean:0.31, std:0.03 min:0.24 max:0.40

Figure A.139

CDG HCS, Bottleneck = 10Mbps, $G = 3.0$, beta backoff = $-1$, base RTT: 150ms

CDG 1 stat, mean:0.29, std:0.03 min:0.24 max:0.35
CDG 2 stat, mean:0.30, std:0.02 min:0.27 max:0.35
CDG 3 stat, mean:0.31, std:0.02 min:0.27 max:0.36

Figure A.140
CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = 0.2, base RTT: 150ms

CDG 1 stat, mean: 0.22, std: 0.05 min: 0.12 max: 0.33
CDG 2 stat, mean: 0.23, std: 0.05 min: 0.11 max: 0.38
CDG 3 stat, mean: 0.25, std: 0.06 min: 0.13 max: 0.44

Figure A.141

CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = 0.4, base RTT: 150ms

CDG 1 stat, mean: 0.28, std: 0.04 min: 0.18 max: 0.39
CDG 2 stat, mean: 0.24, std: 0.04 min: 0.16 max: 0.36
CDG 3 stat, mean: 0.24, std: 0.04 min: 0.17 max: 0.33

Figure A.142
CDG HCS, Bottleneck = 10Mbps, G =7.5, beta backoff =0.6, base RTT: 150ms

![Graph](image)

Figure A.143

CDG HCS, Bottleneck = 10Mbps, G =7.5, beta backoff =0.8, base RTT: 150ms

![Graph](image)

Figure A.144
CDG HCS, Bottleneck = 10Mbps, G = 7.5, beta backoff = -1, base RTT: 150ms

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.2, base RTT: 150ms

Figure A.145

Figure A.146
Figure A.147

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.4, base RTT: 150ms

Figure A.148

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.6, base RTT: 150ms
Figure A.149

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = 0.8, base RTT: 150ms

CDG 1 stat, mean:0.32, std:0.03 min:0.27 max:0.43
CDG 2 stat, mean:0.30, std:0.03 min:0.23 max:0.35
CDG 3 stat, mean:0.29, std:0.02 min:0.23 max:0.34

Figure A.150

CDG HCS, Bottleneck = 10Mbps, G = 20.0, beta backoff = -1, base RTT: 150ms

CDG 1 stat, mean:0.33, std:0.03 min:0.27 max:0.41
CDG 2 stat, mean:0.28, std:0.03 min:0.23 max:0.35
CDG 3 stat, mean:0.30, std:0.03 min:0.22 max:0.36
CDG HCS, Bottleneck = 10Mbps, G = 120.0, beta backoff = 0.2, base RTT: 150ms

Figure A.151

CDG HCS, Bottleneck = 10Mbps, G = 120.0, beta backoff = 0.4, base RTT: 150ms

Figure A.152

225
Figure A.153

CDG HCS, Bottleneck = 10Mbps, G = 120.0, beta backoff = 0.6, base RTT: 150ms

Figure A.154

CDG HCS, Bottleneck = 10Mbps, G = 120.0, beta backoff = 0.8, base RTT: 150ms
CDG HCS, Bottleneck = 10Mbps, G = 120.0, beta backoff = 1, base RTT: 150ms

CDG 1 stat, mean: 0.32, std: 0.02 min: 0.29 max: 0.39
CDG 2 stat, mean: 0.29, std: 0.02 min: 0.25 max: 0.33
CDG 3 stat, mean: 0.30, std: 0.02 min: 0.25 max: 0.35

Figure A.155