A Hybrid Loss-Delay Gradient Congestion Control Algorithm for the Internet

Rasool Al-Saadi

School of Software and Electrical Engineering

Swinburne University of Technology

This thesis is submitted for the degree of

Doctor of Philosophy

Faculty of Science, Engineering and Technology

October 2019

I dedicate this thesis to my loving parents, my beloved wife Dalia and my sons Ridha and Yousif.

Declaration

I declare that this thesis submitted for the degree of Doctor of Philosophy contains no material that has been accepted for the award to the candidate of any other degree or diploma, except where due reference made a in the text of the examinable outcome. To the best of the candidate's knowledge contains no material previously published or written by another person except where due reference is made in the text of the examinable outcome. Where the work is based on joint research or publications, discloses the relative contributions of the respective creators or authors.

Rasool Al-Saadi
Digitally signed by Rasool Al-Saadi
Date: 2019.10.19 12:32:11 +03'00'

[illegible]

Rasool Al-Saadi
October 2019

Acknowledgements

I would like to express my thanks and appreciation to all the people and institutions who helped, guided, advised and supported me during my PhD journey.

I would like to thank my supervisors Prof. Grenville Armitage, Dr. Jason But and Assoc. Prof. Philip Branch for their insight, patience, friendship, encouragement, continuous support and valuable advice during my PhD candidature. I had the honour of being a student under their wise supervision. Without their support, this work would never have seen the light of day.

I am greatly indebted to my family, especially my wife Dalia Al-Zubaidy, my parents, my brother and my sister for their patience, persevering support and advice. Their encouragement allowed me to overcome the trials and tribulations during this long journey.

I would like to give special thanks to my sponsor, the Ministry of Higher Education and Scientific Research - Iraq, for giving me the opportunity to complete my PhD candidature and for supporting me during my study.

I thank all members of the I4T research group for our interesting discussions. Special thanks to Jonathan Kua for helping me in testing different implementations, for answering my questions and his interesting discussion.

Finally, I am immensely grateful to my best friend, Haider Majeed, for following up my financial and administrative matters in Iraq without hesitation or complaint.

Abstract

Congestion Control (CC) has a significant influence on the performance of Transmission Control Protocol (TCP) connections. Over the last three decades, many researchers have extensively studied and proposed a multitude of enhancements to standard TCP CC. However, this topic still inspires both academic and industry research communities due to the change in Internet application requirements and the evolution of Internet technologies. The standard TCP CC infers network congestion based on packet loss events which leads to long queuing delay when bottleneck buffer size is large in a phenomenon called Bufferbloat. Bufferbloat harms interactive and latency-sensitive applications and reduces network response speed, leading to a poor quality of experience for end users.

A potential end-to-end solution to this problem is to use the delay signal to infer congestion earlier and react to the congestion before the queuing delay reaches a high value. However, delay-based feedback is fraught with difficulties including queuing delay estimation, signal noise and coexistence with conventional TCP.

Loss-delay hybrid congestion control approaches aim to remedy the coexistence issue by operating in delay mode to provide low latency transport, and switching to loss mode to improve performance when competing with loss-based flows. Using a delay gradient enables the CC protocol to detect congestion without relying on queuing delay estimates. This motivates us to explore the previously published CAIA Delay-Gradient (CDG) CC algorithm.

In this thesis, we significantly enhance the CDG algorithm to create Hybrid Loss-Delay Gradient (HLDG), a sender-side host-to-host congestion control algorithm for the Internet. Compared to CDG, HLDG provides higher performance, lower queuing delay and better coexistence when competing with loss-based flows.

Like CDG, HLDG uses the delay gradient signal to detect early congestion, preventing a long queue from building up in the bottleneck, and providing low latency transport. It also explicitly switches between delay and loss modes depending on competing traffic characteristics to coexist with loss-based flows.

By evaluating CDG in a range of scenarios, we identify weaknesses that impact protocol performance. HLDG enhances CDG throughput by solving the unnecessary back-off issue and by maintaining the number of unacknowledged bytes close to estimated the path bandwidth-delay product (BDP). HLDG also improves coexistence by switching between loss and delay mode based on a heuristic function. This function infers the presence of loss-based flows based on queuing delay measurements. Finally, HLDG utilises an enhanced slow-start algorithm using the delay gradient signal and BDP estimate. HLDG slow-start terminates with a congestion window size close to estimated BDP, achieving high throughput without inducing high queuing delay.

HLDG can also be effectively used as a low priority CC to provide scavenger class transport for background file transfer without disturbing the foreground traffic. This can be achieved by disabling the HLDG coexistence mechanisms and reducing the back-off factor.

We experimentally evaluate HLDG under a range of scenarios and network conditions. This includes a range of link bandwidths and path RTT, with both FIFO Droptail and Active Queue Management (AQM) used at the bottleneck.

Our results show that HLDG obtains up to 400% (54% in average) throughput improvement over CDG in single flow scenarios without packet loss while maintaining queueing delay under 20ms in the worst case. Our results also show that HLDG is able to achieve up to 3000% (810% in average) better throughput than CDG when competing with TCP CUBIC. HLDG slow-start exits with a congestion window between 96% - 200% of the path BDP compared to 1% - 800% for CDG slow-start.

HLDG achieves 94%, 89% and 95% average link utilisation under PIE, FQ-CoDel and FQ-PIE respectively, compared to 89% for CDG under the same conditions. HLDG experiences only 3%, 24% and 4% of packet loss experienced by CDG when run through PIE, FQ-CoDel and FQ-PIE AQMs respectively.

Contents

List of Figures

List of Tables

List of Acronyms

AIMD	Additive Increase Multiplicative Decrease
AQM	Active Queue Management
BDP	Bandwidth Delay Product
CA	Congestion Avoidance
CC	Congestion Control
CDG	CAIA-Delay gradient
cwnd	Congestion Window
DIFTR	Dummynet Information For TCP Research
ECN	Explicit Congestion Notification
EWMA	Exponentially Weighted Moving Average
FIFO	First In First Out
FQ-CoDel	Flow-Queue Controlled Delay
FQ-PIE	Flow-Queue Proportional Integral controller Enhanced
HLDG	Hybrid Loss-Delay Gradient
HTTP	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
MSS	Maximum Segment Size

NIC	Network Interface Card
OS	Operating System
OWD	One-Way Delay
PIE	Proportional Integral controller Enhanced
RFC	Request For Comments
RTO	Retransmit Time Out
RTT	Round Trip Time
SACK	Selective ACKnowledgement
SIFTR	Statistical Information For TCP Research
SS	Slow-Start
TCP	Transmission Control Protocol
TSO	TCP Segmentation Offload
ttprobe	TEACUP TCP probe
UDP	User Datagram Protocol
WWMA	Weighted Windowed Moving Average

Chapter 1

Introduction

The Internet has changed modern communities to the extent that it has become an integral part of an individual's life. In its early days, the Internet was visualised as a static network transferring small streams of bytes representing messages and small files. Nowadays, it has grown to become a large interconnected network used by hundreds of millions of users for a variety of purposes. It forms the core of modern education, health, government and economy systems, as well as personal entertainment including video streaming, online multiplayer gaming and video conferencing.

This versatility has also led to the development of complex network protocols which are responsible for providing the best quality of experience for all types of users to fulfil their needs. Transmission Control Protocol (TCP) [1] deserves a great tribute for the success of the Internet in the last three decades due to its ability to generally perform sufficiently well in spite of the significant changes in Internet technologies. TCP is a transport protocol that provides byte-streams and reliable data transfer over the packet-based best-effort Internet Protocol (IP) layer [2]. As the preferred transport for many Internet applications, TCP has received a lot of attention from the research community keen to maximise protocol performance.

Congestion Control (CC) is a critical part of TCP that directly influences the protocol's performance. CC aims to manage network resources in an efficient manner and to provide resource sharing among competing flows while protecting the network from collapse.

Typically, TCP CC probes a path's capacity by sending data and monitoring the incoming implicit (or sometimes explicit) feedback signal. Based on the feedback signal, TCP increases or decreases the number of unacknowledged bytes in flight to minimise congestion while achieving high link utilisation. Implicit feedback can be inferred from packet loss caused by bottleneck buffer overflow (loss-based CC) or variations in packet delivery delay caused by queue building up at the bottleneck (delay-based CC). Other CC algorithms (e.g. BBR [3]) control packet emission rates based on estimated bottleneck capacity, and adjust bytes-in-flight from minimum RTT and estimated bottleneck capacity. Additionally, TCP CC can benefit from explicit feedback, such as Explicit Congestion Notification (ECN) [4], where end hosts and bottlenecks both support such a feature.

The most common, actively used TCP implementations utilise reliable and easy to implement loss-based CC algorithms. However, flows using loss-based CC become problematic when they compete with latency-sensitive flows for capacity at bottlenecks with large buffers. In such cases, loss-based TCP fills bottleneck queue until buffer overflow occurs causing long queuing delays in a phenomenon called Bufferbloat [5]. The resulting additional latency can negatively impact latency-sensitive applications (such as live video streaming and online gaming) and degrade network quality of service in general. Using packet loss as a signal also wastes network resources due to the need for retransmitting the lost packet.

Consequently, many researchers have explored the use of delay as a feedback signal (partially or fully) to remedy the shortcomings of loss as a signal.

Delay-based CC approaches provide low latency data transfer through controlling network congestion as bottleneck queues start building. Using a such strategy, they prevent packet loss which can significantly affect unreliable UDP streams such as VoIP traffic. Additionally, many delay-based CC algorithms aim to achieve high and stable throughput by reducing oscillation in data sending. Oscillation in packet sending reduces the performance of transport protocol in long-distance high-speed paths with shallow bottlenecks buffers. Delay-based CC approaches can be optimised to be used efficiently in high-speed long-distance networks to converge to full link utilisation quickly without stressing the network. Moreover, it has been shown in many industrial and academic works that delay-based CC can be efficiently used in background bulk data transfer transports and scavenger class services such as system updates [6–11]. It is also possible to utilise the delay signal to distinguish between congestion related and random losses [12, 13]. This is useful to achieve high link utilisation in lossy networks.

Unfortunately, delay-based feedback is complicated and fraught with difficulties including noise in the signal, sampling problems, coexistence and many other issues. This prevents delay-based CCs from being widely used for general purpose use. Protocol coexistence significantly impacts on the user experience in environments where loss-based and delay-based flows traverse a bottleneck. In such environments, delay-based flows typically obtain much lower bandwidth share than loss-based flows. This reduces the throughput of delay-based flows, resulting in a poor quality of experience.

A hybrid delay-loss based congestion control approach can overcome the coexistence issue. A hybrid algorithm can achieve low latency in the absence of competing loss-based flows and better performance when accompanying loss-based flows. It normally operates in delay mode where congestion is detected with the delay signal. When a competing loss-based flow is detected, hybrid algorithms switch to loss mode where congestion is detected using packet loss signal to achieve better performance.

CAIA-Delay gradient (CDG) [12] is a hybrid delay-loss based congestion control algorithm that aims to achieve high throughput and low queuing delay without relying on delay thresholds. It also aims to coexist with loss-based flow by normally operating in delay mode and switching to loss mode when it infers a competing loss-based flow. CDG has attracted researcher interest due to its ability to detect congestion without relying on queuing delay measurements while maintaining low queuing delay. Additionally, unlike many delay-based CC algorithms, it does not suffer from latecomer advantage in which recent starting flows obtain higher capacity share than older flows. However, no improvements have been proposed to produce a better hybrid congestion control.

In this thesis, we deeply evaluate and analyse CDG algorithm in different scenarios to explore its issues and weaknesses. This allows us to propose a sender-side CDG-based algorithm for the Internet called Hybrid Loss-Delay Gradient (HLDG) that provides higher performance, lower queuing delay and better coexistence with loss-based flows.

1.1 Research objectives and contributions

As mentioned above, developing high-performance, low-latency transport for the Internet is highly desirable. Such transport protocols can reduce the negative impact on latency-sensitive traffic, providing a higher quality of experience for end users. Delay-based CC approaches aim to achieve these requirements but they are fraught with difficulties preventing them from being widely used on the Internet. Using loss delay hybrid algorithms with a proper delay signal manipulation overcomes these difficulties, allowing them to achieve better performance and wider deployment. CDG is a promising hybrid congestion control algorithm that relies on delay gradient signal to infer congestion. This protocol has found interest from research communities due to its congestion detection strategy that allows it to achieve a low queuing delay. The objectives of this thesis are to 1) understand existing problems with the CDG algorithm; 2) find solutions to these problems to produce a better hybrid congestion CC algorithm; and 3) to derive new general knowledge about delay-based congestion control from the lessons learned in this research.

This thesis experimentally evaluates and analyses the CDG algorithm in scenarios emulating a typical home gateway. By exploring CDG performance in simple scenarios using a controlled testing environment, we demonstrate that CDG suffers from significant low performance in high path RTT scenarios due to unnecessary back off in different situations. We begin by evaluating CDG link utilisation when only one flow traverses a bottleneck. Even though such scenarios are uncommon in real-world networks, it provides insight to identify the potential problems that impact on protocol performance.

CDG includes coexistence mechanisms to provide better performance when competing with loss-based flows. We evaluate and analyse CDG coexistence performance when competing with CUBIC, a widely deployed loss-based protocol, in a wide range of scenarios. We demonstrate that CDG is unable to compete with loss-based flows due to limitations in its coexistence mechanisms.

In addition to congestion avoidance, we evaluate and analyse CDG slow-start as it plays an important role in protocol performance. We find that CDG slow-start terminates too early in many scenarios, preventing the protocol from achieving high throughput.

To the best of our knowledge, this type of comprehensive evaluation and microanalysis for CDG mechanisms has not performed before. This exploration of the reasons for CDG weakness enables us to propose an improved protocol that achieves high throughput, low latency and better coexistence.

We then propose HLDG, an enhanced sender-side hybrid CC algorithm built as an improvement to CDG. HLDG achieves higher throughput in single flow scenarios by eliminating the unnecessary back-offs performed by CDG. It also uses bandwidth and base RTT estimations to estimate the Bandwidth-Delay Product (BDP) to control the back-off size. Finally, HLDG uses accumulated probability in making back-off decisions, resulting in better latency control. Our experimental evaluation shows that HLDG realises up to 400% (54% in average) throughput improvement over CDG in single flow scenarios without experiencing packet loss. HLDG is able to maintain low queueing delay of no more than 20ms in the worst case.

To provide better coexistence with loss-based TCP, HLDG dynamically reduces protocol sensitivity to delay increase when a loss-based flow is present. More importantly, when it detects a competing loss-based flow, HLDG explicitly switches to CUBIC-like loss-based mode in which back-off is performed only on packet loss events. HLDG switches back to delay mode as soon as it senses no competing loss-based flow is present, maintaining low queuing delay. A heuristic function based on the estimated queuing delay is used to make this decision. HLDG obtains up to 3000% (810% in average) better throughput than CDG when competing with TCP CUBIC. Additionally, the results reveal that HLDG coexistence performs ten orders of magnitude better than CDG for large buffer sizes.

HLDG improves the CDG slow-start phase by fixing CDG signalling issues combined with the use of the passive ACK train technique to estimate bottleneck bandwidth, allowing HLDG to exit slow-start with high link utilisation. Our experiments show that HLDG slow-start terminates with a congestion window between 96% - 200% of the path BDP compared to 1% - 800% for CDG slow-start.

Moreover, we conduct a preliminary HLDG evaluation in more advanced scenarios including homogeneous and heterogeneous path RTT and AQM environments. Our results show that HLDG achieves 0.99 and 0.98 average Jain's fairness index in homogeneous and heterogeneous path RTTs respectively while keeping average queuing delay below 13ms.

We evaluate HLDG in modern AQM based bottlenecks, namely PIE (Proportional Integral controller Enhanced) [14, 15], FQ-CoDel (Flow-Queue CoDel) [16] and FreeBSD's FQ-PIE (Flow-Queue PIE) [17]. The results show that HLDG performs better than CDG and close to CUBIC performance in many scenarios. HLDG performs best when FQ-PIE [17] is used. HLDG realises 94%, 89% and 95% average link utilisation under PIE, FQ-CoDel and FQ-PIE respectively compared to 89% for CDG under the same conditions. HLDG experiences only 3%, 24% and 4% of packet loss experienced by CDG when run through PIE, FQ-CoDel and FQ-PIE AQMs respectively.

We also demonstrate that HLDG can also be used as low-priority CC providing efficient scavenger-class transport. The can be done by disabling coexistence mechanisms, resulting in using only the delay feedback signal in congestion control. The preliminary results show that HLDG in LPCC mode utilises 5 - 18% bottleneck bandwidth when competing with TCP CUBIC, producing an efficient scavenger class transport.

We provide a prototype HLDG implementation for the FreeBSD operating system, allowing us to evaluate the algorithm in a range of emulated networks. This results in more a realistic evaluation reflecting basic real-world scenarios.

More importantly, our research derives new general knowledge about delay- based congestion control. This helps inform the TCP research community to avoid making similar errors and to develop better congestion control. The new knowledge can be summarised as follows:

1. Delay-based CC algorithms should not back-off in response to small bursts caused by delayed ACKs. In doing so, delay-based CC algorithms would back-off even when no congestion is experienced.

2. A capacity estimator (e.g. using the acknowledged byte rate), combined with the delay signal improves the performance of delay-based congestion control.

3. Standing queues can be avoided by replacing a moving average with a windowed moving average with weights over the delay signal, and more importantly re-starting the calculation under certain circumstances.

This thesis also categorises and reviews popular delay-based, hybrid and delay-sensitive TCP variants. This involves presenting a novel taxonomy for these algorithms based on the congestion feedback signal usage.

1.2 Summary of research contributions

This thesis fills a number of research gaps and contributes to both academic and industrial communities. The thesis contributions are summarised as follows.

1. We explore the use of delay feedback signal in TCP congestion control by providing a taxonomy for TCP variants that use the delay signal in their operations. We categorise these techniques based on congestion feedback signal types and operation modes. Our work involves reviewing and comparing a wide range of current and historical delay-based, hybrid and delay-sensitive TCP variants. Moreover, we discuss the challenges of using the delay signal in congestion control algorithms.

2. We perform a deep experimental evaluation and microanalysis for CDG under a range of network settings. This includes comprehensive investigations for CDG low performance in simple scenarios and when competing with loss-based flows. Additionally, we explore the CDG slow start phase and provide insight into the issues causing low performance in many scenarios.

3. We propose a CDG-based algorithm called Hybrid Loss-Delay Gradient (HLDG) that addresses CDG low link utilisation, providing high throughput and low queuing delay. This involves solving unnecessary back-off issues, using a better probabilistic back-off function, as well as using the BDP estimate to reduce performance loss upon back-off. We comprehensively evaluate the proposed protocol under a range of bottleneck bandwidths and path RTTs.

4. We improve HLDG coexistence when sharing a bottleneck with loss-based flows by adapting protocol sensitivity to the delay increase and explicitly mode switching. HLDG uses a heuristic function based on queuing delay to detect competing loss-based flows. HLDG switches to CUBIC-like loss-based mode when a loss-based flow is present and switches back to delay mode when competing loss-based flows terminate. Through a range of experiments, we demonstrate improved coexistence compared with CDG.

5. We propose an improved slow-start algorithm relying on delay-gradient signal and BDP estimation. This algorithm ensures slow-start termination before a long queue builds up with high link utilisation.

6. We perform a preliminary evaluation of HLDG in more advanced scenarios and AQM environment. We explore the inter-protocol fairness in both homogeneous and heterogeneous path RTT.

7. We perform a preliminary evaluation for HLDG in LPCC mode.

8. We independently implemented CoDel, PIE and FQ-CoDel for Dummynet in FreeBSD OS based on their Internet drafts at the time [17, 18]. This project was supported by The Comcast Innovation Fund USA. We also use the Flow-Queuing scheduler with PIE AQM to produce an experimental FQ-PIE AQM. This open doors for further research to explore the benefits of using this new AQM. Our implementation was official added to FreeBSD source tree from version FreeBSD 11.0 [19] and backported to FreeBSD 10.4 [20], and is used by reputable firewalls such as pfSense [21] and OPNsense [22]. This work also contributed in fixing errors in the FQ-CoDel [23] and PIE IETF Internet drafts [24].

9. We developed ttprobe, an event-driven TCP statistics capturing tool for the Linux operating system [25]. ttprobe provides high-resolution statistics from the TCP/IP stack with lower processing overhead and better compatibility than the more commonly used Web10G [26].

10. We develop the DIFTR tool for collecting AQM and FIFO statistics from inside Dummynet in FreeBSD. This tool allows us to understand the interaction between congestion control and bottleneck queue dynamic. It provides statistics such as queue length, the queuing delay and per queue packet drop counts.

1.3 Thesis organisation

The rest of this thesis is organised as follows. Chapter 2 describes TCP background information, congestion control principles and congestion feedback signals. It introduces our novel taxonomy on delay-based and hybrid congestion control algorithms and reviews a range of congestion control algorithms that use the delay signal in their operation. It also discusses the challenges facing with the delay signal.

Chapter 3 describes our experimental methodology, tools used to conduct different experiments, our testbed and testing scenarios covered in our research.

Chapter 4 evaluates CDG algorithm and explores its issues. This includes evaluating CDG in simple scenarios, coexistence with loss-based flows and the CDG slow-start.

Chapter 5 introduces our HLDG congestion algorithm which fixes different CDG problems, providing better performance, low queuing delay and better coexistence with loss-based flows.

Chapter 6 provides preliminary evaluation for HLDG in more advanced scenarios and AQM environment. It also briefly evaluates HLDG in low priority congestion control mode in which it achieves lower throughput when sharing a bottleneck with conventional TCP.

Chapter 7 outlines future research direction and provides conclusions of this thesis.

Chapter 2

Delay-based congestion control

Congestion Control (CC) is a crucial part of TCP that controls the protocol's performance. The goals of using CC is to manage network resources in an efficient manner and to provide resource sharing among competing flows while protecting the network from collapse.

Loss-based CC is the most common and actively used CC algorithm due to its reliability and simplicity. However, loss-based flows negatively impact latency-sensitive flows when they compete for capacity at bottlenecks having large buffers. Delay-based CC responds sooner to network congestion, preventing a long queue from building at the bottleneck.

In this chapter[1], we present a literature review of congestion control techniques that utilise the delay signal as a primary or secondary indicator of control network congestion. We describe general principles of TCP CC and congestion signal types, and explore the challenges of using the delay signal and how some recently popular queuing-delay based Active Queue Management (AQM) techniques are likely to interact with delay-based CC techniques. Since there are many proposed TCP CC algorithms utilising the delay signal, this chapter covers popular techniques that have real impact on their working environments.

The rest of this chapter is structured as follow. Section 2.1 provides principles of TCP flow control and congestion control. Section 2.2 describe the interaction between TCP and the bottleneck FIFO buffer and introduces AQM functionality. Section 2.3 is devoted to TCP congestion control literature including the delay and loss congestion feedback signals and standard TCP CC algorithms. Section 2.4 is dedicated to reviewing popular delay-based, hybrid and delay-sensitive TCP variants. Section 2.6 discusses the challenges faced with using the delay signal including deploying AQM. Section 2.5 presents slow-start variations that relies on the delay measurements. Section 2.7 presents details for the promising CDG TCP CC algorithm. We conclude the chapter with Section 2.8.

[1]This chapter has been published as paper in IEEE Communications Surveys Tutorials[27]

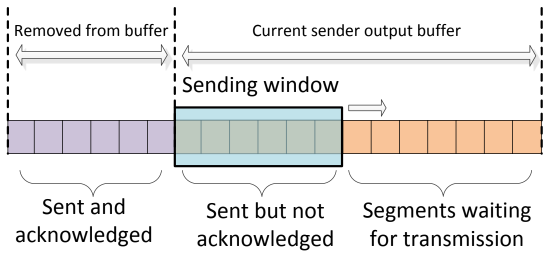

Removed from buffer
Current sender output buffer
Sending window
Sent and acknowledged
Sent but not acknowledged
Segments waiting for transmission

Figure 2.1: TCP sliding window mechanism

2.1 Transmission Control Protocol – An Overview

The TCP layer provides a reliable, connection-oriented end-to-end transport protocol that guarantees error-free, in order delivery of data to the destination [1]. TCP flow control and congestion control limit the amount of outstanding (unacknowledged) sent data. Flow control prevents fast senders from overrunning the buffers of slow receivers (which causes packet loss). Congestion control aims to prevent senders from sending too much data that can overflow buffers within the network (network congestion). In this section, we summarise the principles of TCP flow control, TCP congestion control and how the injection of packets into the network can be controlled.

2.1.1 TCP Reliability and Flow Control

The IP layer [2] provides a best-effort packet transfer service between the source to destination host. IP does not guarantee delivery, nor ensure packets are delivered in-order. TCP is responsible for both data integrity and network resource management to provide a reliable end-to-end connection.

Data transfer over TCP is initiated by an application which supplies data to the TCP stack. TCP buffers the data, allocating each byte a sequence number. TCP then partitions the buffered data into segments, assigning each segment the sequence number of the first byte in that segment. Using a sliding window mechanism (shown in Figure 2.1), TCP transmits a number of segments to the receiver over the IP protocol.

At the receiver, the destination host buffers the segments in its TCP receive buffer. If a segment arrives without error and in order (checked using the sequence number), the receiver confirms receiving the segment by generating and sending back an acknowledgement packet (ACK)[2] containing the sequence number of next byte it is expecting to receive.

The ACK packet confirms the delivery of all bytes that have sequence numbers smaller

[2]A normal TCP packet with the ACK flag in TCP header set.

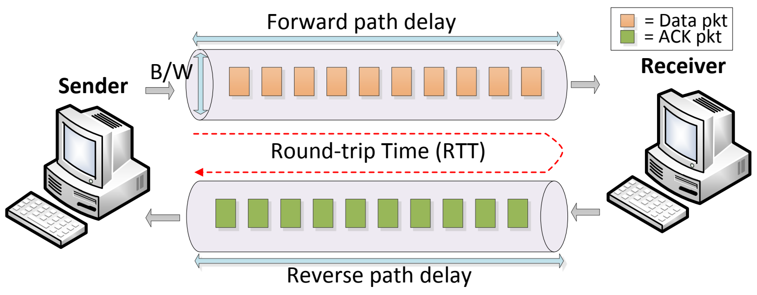


Figure 2.2: Bandwidth-Delay Product and link pipe

than the ACK number. As such, it is not required to send an ACK packet for each data packet. When an ACK packet is received by the source, the sender moves the sending window (*swnd*) by the amount of acknowledged data and sends new segments if they are ready in the sending buffer. In other words, the sender should not transmit more data than *swnd* allows until receiving acknowledgement of previously sent data i.e. $seq_{next} < seq_{una} + swnd$ where seq_{next} is the sequence number of next packet to be sent, and seq_{una} is the sequence number of the first packet sent but not yet acknowledged. As such, *swnd* effectively limits the number of unacknowledged bytes (bytes in-flight).

Using this mechanism, there is a period of time before the sender is aware if a segment arrived at the destination correctly. The earliest the sender can receive an ACK is given by the round trip time (RTT) – the combination of serialisation delays (transmission time), propagation delays, and bottleneck queuing delays along both the forward and reverse paths. In reality each ACK may be further delayed by additional factors such as the processing power of the end hosts and middleboxes along the path, and efficiency improvement mechanises (e.g. delayed ACK).

We can conceptualise the link between the source and destination as a virtual pipe between the two points (see Figure 2.2). The bandwidth (link capacity) is represented by the diameter of the pipe, while the delay (path RTT) is represented by the length. The product of these two values, the bandwidth-delay product (BDP), represents the pipe's volume – the amount of data that saturates the link between the sender and receiver.

If the sender fills the pipe completely, data transmission will be continuous because acknowledgements will be received while data is still being sent. On the other hand, if the pipe is partially filled, there will be stalls during transmission because the sender has to wait for acknowledgements to trigger sending new data as shown in Figure 2.3. In order to achieve optimum throughput, the sender should keep *swnd* greater than or equal to the link BDP.

The destination TCP receive buffer size is subject to memory availability, system/applica-

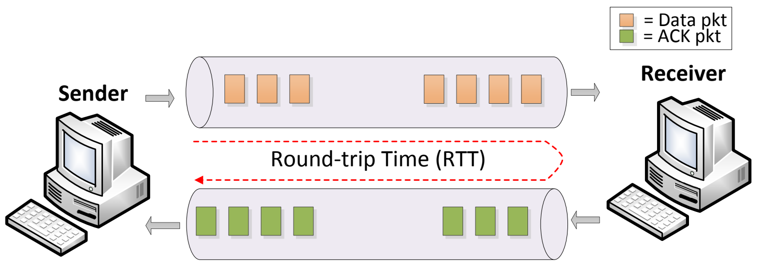


Figure 2.3: Unfilled link pipe causing low link utilisation

tion configuration and processing power. Therefore, the sender should be aware of the receiver buffer availability. As part of TCP flow control, the receiver specifies the maximum number of bytes it is willing to receive via the advertised receive window size (*rwnd*). This mechanism prevents a fast sender from exhausting the limited buffer size in a slow receiver.

2.1.2 TCP Congestion Control

The original TCP specification limits outstanding (unacknowledged) packets only by the receiver's *rwnd* [1]. However, this completely ignores network congestion.

Without awareness of the current network congestion state, TCP may send more data than a bottleneck can handle, resulting in heavy packet loss, significant reduction in network performance, increase in packet delivery delay, and could lead to a phenomenon called *congestion collapse*. This is a scenario where only a small portion of transmitted data is successfully delivered and acknowledged, leading to low goodput[3] and long queuing delays.

Congestion collapse in the Internet was first observed in the mid 1980s [28], due to TCP senders spuriously retransmitting packets that were actually not missing but waiting in long queues. The retransmissions exhausted bottleneck capacity more seriously as the number of flows increases.

An early solution to mitigate the congestion problem relied on an explicit message sent using Internet Control Message Protocol (ICMP) [29]. An ICMP Source Quench [1] message would be sent by the congested router to the sender when the bottleneck buffer becoming congested, causing the sender to throttle back. However, use of ICMP Source Quench was deprecated due to ineffectiveness and unfairness issues [30].

The fundamental solution has been the development and use of end-to-end congestion control (CC). CC algorithms aim to monitor the network's current congestion state, and to use this information to adjust the sending rate, directly or indirectly, to stabilise network usage,

[3]Goodput is the amount of data that arrives to the destination successfully

maintain high link utilisation and provide a fair share of the available bandwidth [31].

TCP CC [32] maintains a congestion window size (*cwnd*) for each TCP flow, representing the maximum number of bytes the sender may send and have outstanding (unacknowledged) based on current network congestion state. The sender selects the minimum of *rwnd* and *cwnd* as the final sending window size $swnd = min(rwnd, cwnd)$.

TCP CC attempts to fully utilise the network without causing congestion by inferring current network conditions and dynamically adjusting *cwnd* accordingly. It reacts to changing network conditions by increasing *cwnd* when no congestion is detected and reducing it when a congestion event occurs.

At the start of new TCP connection, the sender is unaware of available network bandwidth. TCP uses a phase called slow-start (see Section 2.3.3.1) to quickly probe the available bandwidth in the path. During slow-start, the sender increases *cwnd* on each ACK received. When congestion is detected, TCP CC sets *cwnd* as a portion of the achieved window size when the congestion was detected, and exits slow-start to enter another phase called congestion avoidance.

During congestion avoidance, *cwnd* increases more slowly than in slow-start, typically by one segment per RTT, to avoid causing congestion while still adapting the window size to any changes in available capacity. A detailed discussion on TCP CC algorithms is available in Section 2.3.3.

Controlling congestion efficiently is not an easy task due to the distributed nature of the TCP/IP protocol and constantly changing network conditions. Not all CC algorithms have the same goals nor are expected to function in all environments. Some algorithms may prioritise minimising delays, while others may focus on performing well under special conditions such as within a data centre. Nevertheless, listed below are some common goals shared by most CC algorithms [28].

- Preventing congestion collapse: Considered the main reason for the existence of CC.

- High bandwidth utilisation: Considered a fundamental requirement for all CC algorithms. CC should avoid path capacity underutilisation to maximise throughput.

- Fairness: CC should guarantee an acceptable equal share of the available bandwidth among all competing flows sharing the same bottleneck.

- Fast convergence to fairness: When a new TCP flow joins a shared bottleneck, the CC should react rapidly to this event by increasing the *cwnd* of the new flow and reducing it for all other flows until fairness is achieved.

- TCP-Friendly: For deployment purposes, CC algorithms intended to be used in an uncontrolled network, e.g. the Internet, should coexist with other CC algorithms by maintaining fairness.

Due to differences in CC algorithms, factors such as path RTT and flow starting sequence may affect fairness, giving advantages to some flows over others. For example, TCP Reno [32] flows transmitted over a high RTT path get lower overall throughput than flows using shorter RTT paths. This is due to the Reno algorithm increasing *cwnd* by one Maximum Segment Size (MSS) every RTT. As a result, *cwnd* increases faster for flows using shorter paths.

TCP CC protocol variations should manage *cwnd* such that it does not reduce their – or other standard CC protocols – performance significantly. A well-known example of incompatibility with standard CC is TCP Vegas [33] which realises low throughput when competing with loss-based CC in a large bottleneck buffer environment [34].

Some CC algorithms have been designed to work in a controlled environment, such as data centres, where all machines use the same algorithm. For these protocols, compatibility with other widely used algorithms is not a concern.

There are arguments about the ideology of flow fairness in TCP CC and how the fairness concerns benefits for users but not for individual flows in real world [35]. However, flow fairness is still considered important for many CC algorithms studied in academia. As such, CC algorithms typically take flow fairness into consideration, trying to distribute available bandwidth amongst competitive flows fairly.

Alternatively, Low Priority Congestion Control (LPCC) algorithms specifically aim to achieve lower capacity sharing of the available bandwidth and/or lower queuing delay when coexisting with other flows [36]. This group of algorithms (also called *scavenger class* or *less than best effort* service) are used by background applications such as bulk file transfer, peer-to-peer applications and automatic software updates. In these cases, the algorithms try to reduce the impact on higher priority foreground applications, while attempting to maintain fairness when coexisting with flows from the same class.

The original TCP specifications [1] do not specify a direct mechanism for the hosts to learn about network congestion state. As a result, TCP CC utilises indirect information to determine whether the path is congested or not, or the degree of congestion in the path. This indirect information gathered from measurements taken during packet exchange between hosts, and typically varies due to buffering effects that may occur at any point in the network path.

During the last three decades, many congestion control algorithms have been proposed to calculate an optimum *cwnd*. However, only a few have been standardised including TCP Reno [32], TCP NewReno [37] and TCP SACK[38]. We denote these CC algorithms as standard TCP in this Thesis.

2.1.3 Controlling the injection of packets into the network

Most TCP implementations utilise window-based CC strategies to limit the number of injected packets in the network. Although this mechanism is efficient, easy to implement and does not require timers, it can generate periodic packet bursts into the network. This can lead to delay fluctuations, increased packet losses, higher queuing delays, and lower throughput [39]. These bursts occur because the transmitter immediately sends new packets (as many as *swnd* allows) whenever an acknowledgement is received. If acknowledgements are delayed or compressed for any reason (e.g. congestion in the reverse path), the sender will receive multiple acknowledgements in a short period, freeing a space in the window and causing the sender to transmit multiple packets in a burst.

An alternative approach to control the number of transmitted packets in the network is to limit the actual sending rate directly. Rate-based CC can calculate the required sending rate that would fully utilise available bandwidth without causing congestion. The calculated sending rate is then used to schedule regular transmission of packets, removing the burstiness seen with the sliding window. Other types of rate-based CC can estimate the required sending rate in a similar way to window-based CC, i.e. increase the sending rate when no congestion is detected and reduce it when congestion happens to achieve acceptable fairness when competing with window-based flows.

In rate-based strategy, the required sending rate can be realised either inside TCP stack or externally (e.g. packet schedulers) using packet pacing. Packet pacing allows a chunk of packets to be spread across a time slot by adding gaps between the sending of packets. The duration of these gaps is determined by the required sending rate.

Packet pacing is also used with window-based mechanisms to eliminate packet bursts [40]. In this case, the transmission of a window's worth of packet is spread across a full RTT.

Packet pacing provides smoother traffic flows and more stable demands on the network and can improve the stability of TCP by minimising variations in queue utilisation. Additionally, it has been shown that packet pacing provides a positive impact on delay-based CC by providing smooth RTT measurements [41].

Despite their benefits relative to window-based strategies, implementation of rate-based strategies is often more complex and requires accurate timers which is considered a costly requirement for embedded and low-end devices.

2.2 Buffering and Queue Management

Network buffers are used to absorb packet bursts, reduce packet losses, and improve overall network performance. They exist in many places of the packet transmission path including

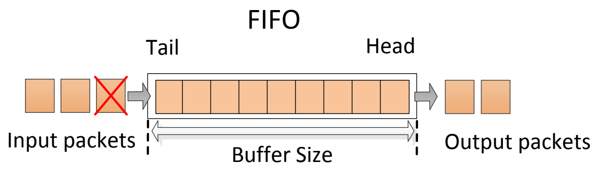


Figure 2.4: FIFO and DropTail buffer management

the host application, TCP socket, host network layer, network interface cards (NIC), network switches, routers, proxies and firewalls.

Buffers are used to temporarily queue packets when the next layer is busy or unable to process the packet as fast as they are provided. There may be a number of causes, such as devices with low processing power, network scheduling priority, temporary reductions in link layer sending rate, and transient network congestion.

2.2.1 Traditional Buffering and Queues

The most common method for implementing network buffers is First-In First-Out (FIFO) with a DropTail management mechanism. In a FIFO queue, packets are appended to the tail of the queue during the enqueue process and fetched and removed from the head of the queue during the dequeue process. When the queue size exceeds the buffer size, the DropTail mechanism drops any new packet until suitable buffer space becomes available. Figure 2.4 shows a conceptual representation of FIFO and DropTail mechanism.

When TCP was first designed, the bit error rate of transmission channel (usually wired) was very low. Therefore, packet loss was mainly caused by buffer overflow, and taken as an indication of congestion at the bottleneck. This relationship between packet loss and network congestion is exploited by loss-based TCP CC to infer congestion along the path.

The proliferation of oversized FIFO buffers in the network, coupled with the aggressiveness of loss-based TCP CC, causes high queuing delay in a phenomenon called Bufferbloat [5]. This high delay has a negative impact on latency-sensitive applications in particular, and on network performance in general.

Active Queue Management (AQM) is a mechanism used to keep the bottleneck queues of network nodes to a controlled depth, effectively creating short queues [42]. AQM is used as a replacement for the DropTail mechanism. When AQM detects congestion it reacts by dropping or marking packets with an ECN [4]. The loss event or ECN signal is then detected by the sender which reduces the transmission rate by decreasing *cwnd*.

2.2.2 Active Queue Management

In the last two decades, many AQM algorithms have been proposed to manage the queuing delay problem. However, none have yet been widely deployed due to both a reduction in network utilisation and complicated optimal configuration.

Legacy AQMs monitor queue occupancy based on bytes or packets in the queue. If the queue length becomes larger than a specific threshold, AQM infers congestion and reacts accordingly based on the congestion level. A well-known example of such statistical AQM is Random Early Detection (RED) [43]. Many queue occupancy-based AQMs have been proposed to mitigate different issues [44–46].

More recently, new AQM mechanisms have emerged that rely on queue *delay* measurement rather than queue occupancy. Queue delay is directly correlated to the network metric that AQM is intended to manage. These new AQMs are able to achieve high throughput and better delay control with low complexity. Further, these AQMs are designed to perform reasonably using their default configurations. Well-known examples of such AQMs are CoDel (Controlled Delay) [47] and PIE (Proportional Integral controller Enhanced) AQM [14, 15].

Moreover, hybrid AQM/scheduler schemes have been proposed to improve fairness between competing flows while keeping queuing delay low. They achieve these goals by diverting the flows into separately managed queues and applying an individual AQM instance for each queue. This separation protects low rate flows from aggressive flows while the individual AQM instances control the queue delay. Examples of hybrid AQM/scheduler schemes are FQ-CoDel (Flow-Queue CoDel) [16] and FreeBSD’s FQ-PIE (Flow-Queue PIE) [17]. In addition to control queuing delays and provide relatively equal sharing of the bottleneck capacity, these AQMs provides short periods of priority to lightweight flows to increase network responsiveness.

Figure 2.5 illustrates simplified FQ-CoDel and FQ-PIE algorithms where flows are hashed to separate queues which are managed by either CoDel or PIE AQM. These queues are serviced using a deficit round robin scheduler with higher priority for new flows.

2.3 TCP Congestion Control Literature -- Signals and Algorithms

In Section 2.1 we explained that CC algorithms try to estimate available bandwidth to optimally configure *cwnd* and maximise network utilisation. However, accurate bandwidth estimation is hard to achieve. Instead, CC algorithms use one or more congestion feedback signals to infer whether the path is under or over utilised. Senders react by increasing or

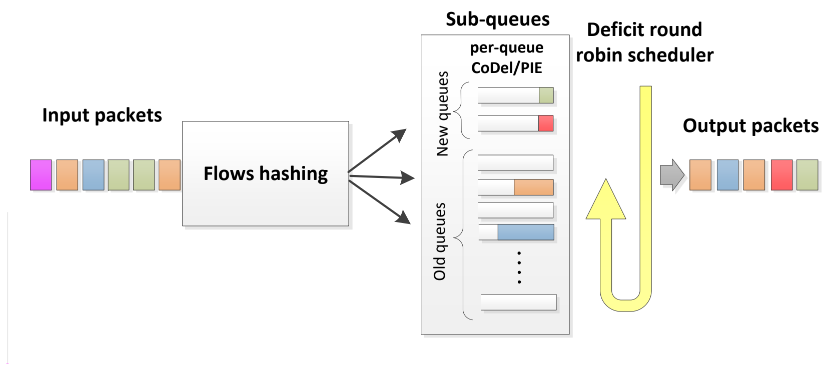


Figure 2.5: Simplified FQ-CoDel/FQ-PIE AQMs

reducing *cwnd* appropriately.

2.3.1 Implicit Congestion Feedback Signals

In many cases network infrastructure does not provide enough information to end hosts regarding the current network condition. As such, end hosts need to infer network state indirectly. One such mechanism is the congestion state of the path. Congestion state may be a simple binary signal or more advanced signal indicating the level of congestion. These signals are used to infer congestion without requiring support from middleboxes in the path between the sender and receiver. Typically, TCP uses either loss or delay signals to infer congestion.

2.3.1.1 The loss signal

Packet loss is used as a *loss signal* by many TCP CC algorithms to indicate network congestion. When a bottleneck experiences transient congestion, packets are queued until buffer space is exhausted. When this occurs, any new packets arriving at the bottleneck will be discarded until the queue drains, freeing up buffer space.

A sender typically detects packet loss using either the TCP Retransmission Time-Out (RTO) timer, or via three duplicate acknowledgement (3DUPACK) packets.

The RTO timer fires for a packet when no ACK is received for that packet, or any packet after it, within the RTO period of time. This event may occur during heavy loss situations due to severe network congestion and/or high noise in the link. RTO also commonly triggered due to loss of last packet in a window (tail drop) when the sender has nothing more to send for longer than an RTO period (application limited flows). For example, if the sender transmits three segments and the third segment lost, then there is no direct way for the receiver to inform the sender (using ACK) that it has not received last segment. Therefore, the sender will keep

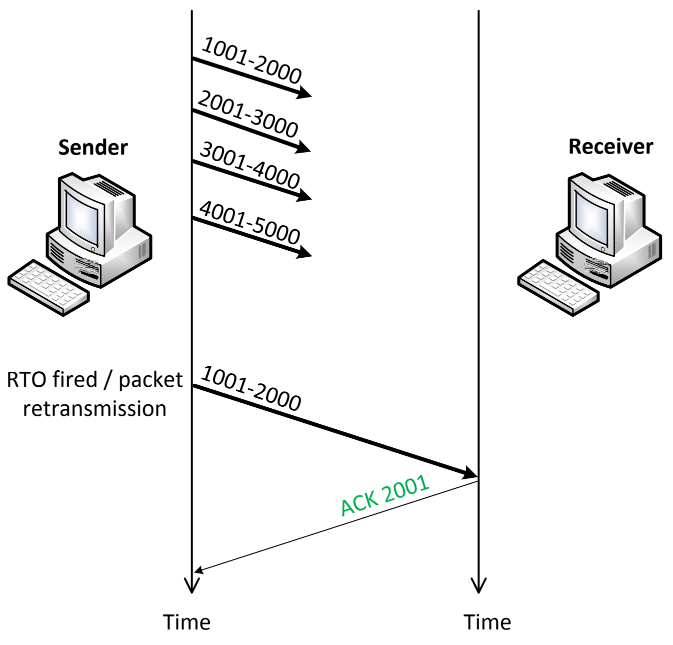


Figure 2.6: A simplified Seq./ACK numbers timeline showing a case of TCP retransmission timeout

waiting unit RTO trigged.

Figure 2.6 illustrates a simple case of the RTO mechanism. No ACKs are received and the RTO timer causes the first packet to be retransmitted. The RTO timer must be tuned correctly to maximise link utilisation. Too large a value results in longer periods with no traffic, while too low a value wastes bandwidth due to multiple transmissions of successfully received packets. TCP Tahoe introduced improved RTO time estimation [48]. The original TCP CC specification included just the RTO mechanism to detect losses.

Alternatively, lost packets result in unordered packet arrival at the receiver. In this case, an ACK is constructed containing the sequence number of the missing packet for each subsequent packet arrival. With 3DUPACK, the sender uses the arrival of the third duplicate ACK packet (four identical ACKs) to infer that the corresponding packet was lost and trigger a retransmission [32].

Detecting congestion using 3DUPACK will typically occur more rapidly than waiting for the RTO timer to fire. This allows for a quicker recovery through the fast retransmission process as shown in Figure 2.7.

The 3DUPACK mechanism was introduced by TCP Reno [32]. TCP uses a threshold of three duplicate ACKs as a balance between the speed of loss detection and false positive

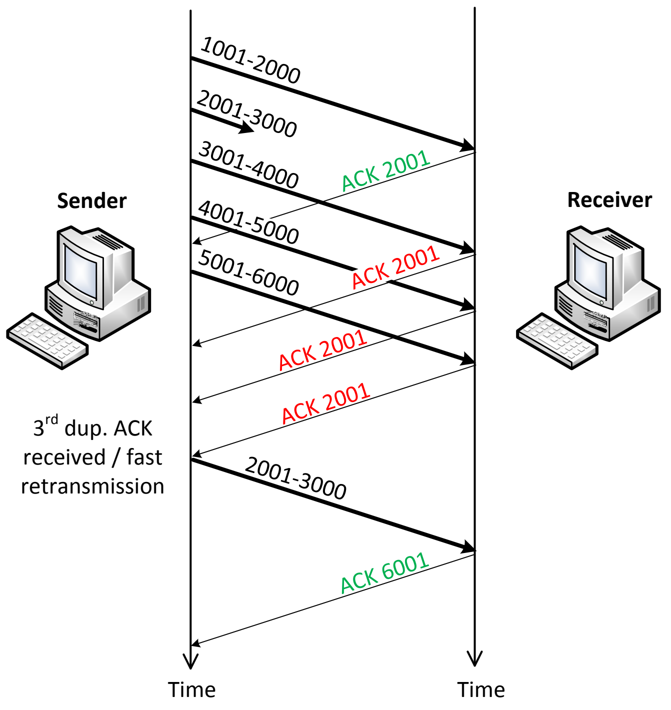


Figure 2.7: A simplified Seq./ACK numbers timeline showing 3DUPACK and fast retransmission mechanisms

detection due to the possibility of out-of-order delivery by the IP layer.

Unlike other inferred congestion signals, determining the *loss signal* does not require precise timers or time calculations, and is simple to implement with minimal code. This was important when TCP/IP was first developed as processing resources were more limited. The simplicity of the *loss signal* is why standard TCP (and key variants such as TCP CUBIC [49]) use it to infer congestion.

While using *loss* feedback is effective and easy to implement, it has some drawbacks. Firstly, using the *loss signal*, we can infer congestion only after network congestion becomes high enough to cause packet loss. The higher use of buffering capacity leads to longer queuing delays. This can lead to a poor quality of experience for latency sensitive applications sharing the bottleneck's buffer, such as VoIP and online gaming.

Secondly, packet loss is not always caused by network congestion. In some wireless networks packet loss may be caused by high bit error rate (BER) or user mobility [50]. The data link layer of modern wireless networks provides better reliability such that upper layers will rarely see packet losses on these networks. Link layer reliability instead translates into additional latency fluctuations at the IP layer, with subsequent implications for delay-based CC.

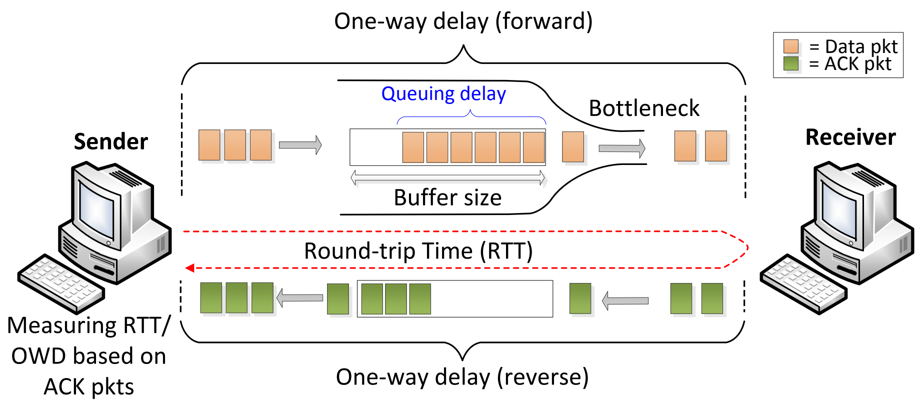


Figure 2.8: Delay congestion feedback signal

These will be further discussed in Section 2.6.2. As such, while the performance of conventional TCP on older wireless networks can be significantly affected by random packet losses, this is not true for modern wireless networks.

2.3.1.2 The delay signal

Measurement of delay can also be used to infer network congestion. The delay can be directly measured in the form of RTT or One Way Delay (OWD), or calculated as a delay gradient signal to more directly measure changes to delay. Figure 2.8 illustrates the delay signal between two hosts connected over a bottleneck.

Some studies argue that there is low correlation between measured delay and congestion in many cases [51–53]. However, McCullagh and Leith [54] studied the correlation between the delay signal and congestion and concluded that the flow's aggregate behaviour is what is important for delay-based CC, not a single observation by one flow. Also, Prasad et al. [55] investigated the reasons for the weak correlation between delays and losses caused by congestion and identify conditions under which the delay signals can fail to provide durable congestion feedback.

The *delay signal* provides more timely feedback of network congestion than the *loss signal* or even an explicit feedback signal. This is important for large BDP networks (also called Long Fat Networks LFNs) [56]. Unlike the *loss signal*, the *delay signal* gives approximate information on the degree of congestion such that the CC algorithm can proactively reduce the sending rate before packet loss occurs, or even before the queuing delay becomes high.

Moreover, the *delay signal* is also effective in lossy network environments. When packet loss occurs unrelated to congestion, packet delivery time will not increase and the *delay signal* is unaffected. As a result, a delay-based CC has the potential to be more tolerant to random packet losses and may perform better in lossy network environments.

Many delay-based CC algorithms use measured RTT as a delay signal, because this only requires TCP sender side modification. As a path's RTT combines the forward and reverse OWD, there is no way to distinguish what component of delay originates in the forward or reverse path. Including the reverse path delay in estimating queuing delay can lead to unnecessary *cwnd* backoff when the reverse path is congested. Congestion in the reverse path is not caused by the aggressiveness of the sender, and decreasing *cwnd* will not improve congestion.

Some delay-based CC algorithms attempt to use only the OWD along the forward path (direction of data flow), to avoid reacting to congestion in the reverse path. One method of OWD calculation is to timestamp packets before transmission. At the receiver, the packet timestamp is extracted and subtracted from the local time. The receiver attaches the calculated OWD to the ACK reply packet.

A problem with this approach is that the direct calculated OWD will not have any meaning without strict time synchronisation between the sender and the receiver which is difficult to achieve. However, if the CC algorithm computes the difference between the calculated OWD and OWD_{base} (OWD_{min}), then the difference will be the one way queuing delay without time synchronisation.

$$OWD_q = OWD - OWD_{base}$$

The downside of using OWD in CC is that it requires receiver side modification, making deployment of such CC algorithms more challenging.

An alternative is to use the TCP timestamp extension [57] to calculate OWD. The sender subtracts the TS Echo Reply from the TS Value fields in the ACK packet to estimate the forward OWD. However, TCP timestamp is an optional feature which not all stacks implement or enable by default. Also, OWD measurement can suffer from clock drift between the sender and the receiver, leading to an increase or decrease of OWD estimation over time. This issue can be addressed by either resetting the OWD_{min} measurement regularly, or by using methods to estimate clock drift [8, 58].

Generally, the *delay signal* is used to estimate the queuing delay D_q for the bottlenecks along the path between the source and the destination. As a bottleneck queue fills, the packet queuing time is translated to latency.

Most delay-based CC algorithms calculate D_q from the path delay by estimating the fixed base (or propagation) delay D_{base}. D_{base} is practically determined as the smallest delay D_{min} seen during a period of time and assumes that the queues along the path completely drain (no

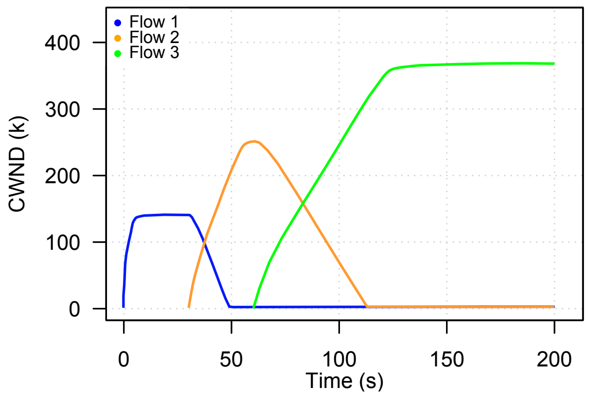


Figure 2.9: *cwnd* vs time illustrates the latecomer advantage problem

queuing delay) at some points during that period. Then, D_q is estimated as $D_q = D - D_{base}$ where D is the measured delay.

Despite the potential advantages of using the *delay signal*, it comes with a number of difficulties. Primarily, the assumption that D_{base} equals D_{min} is not always true, which can lead to over or under-estimation of D_{base}.

Over-estimation of D_{base} leads to an under-estimation D_q and can result in non-detection of a congested network state. Alternatively, under-estimation of D_{base} can result in false detection of network congestion. Errors in measuring D_{base} are caused by persistent queues in the bottleneck. The aggressive nature of loss-based CC algorithms are an obvious source of standing queues in a heterogeneous environment, however standing queues can be formed by delay-based CC algorithms as well. Most delay-based CC algorithms try to achieve optimum throughput, requiring *cwnd* to be at least BDP (see section 2.1.1). This results in queues always being partially filled. These errors can impact on CC performance where algorithms can be either over or under-aggressive, leading to to problems such as unfairness, the *latecomer advantage*, and generation of persistent large queues.

The latecomer advantage problem describes the case where a new flow gets higher bandwidth share (higher throughput) through a congested bottleneck than preexisting flows [59]. This problem is common in threshold delay-based CC algorithms that aim to maintain a constant number of packets in the queue. For example, Figure 2.9 plots *cwnd* versus time for the staggered starts of three LEDBAT flows (section 2.4.1.8) sharing a bottleneck. The plot illustrates how the new flows increase their *cwnd* (achieving higher throughput) while the existing flows decrease their *cwnd* (achieving lower throughput).

This is due to each new flow measuring a higher D_{min} due to existing standing queues in the bottleneck. This leads to overestimating D_{base}, allowing the new flow to be more aggressive in increasing *cwnd*. At the same time, existing flows decrease their own *cwnd* as they interpret the new flow's aggressiveness as network congestion.

A second challenge is the assumption that the path delay is unchanged during a connection's life time, or at least over a period of time. However, this assumption is also not always true (for example, due to path rerouting) [60, 61].

Another challenge when using *delay signal* is the noisiness of delay measurement due to variation in queue occupancy and network jitter. The noise is exacerbated under a heavy load environment and weakens the correlation between the sampled delay signal and congestion [56].

The noise can be reduced using a filter, such as the exponentially weighted moving average (EWMA) filter [58]. However, filtering the signal may reduce the responsiveness of delay-based CC.

One final issue when using *delay signal* is delayed ACKs. The TCP delayed ACK option is used to reduce the number of ACK packets in the reverse path to reduce resource wastage. Delayed ACKs can cause inaccurate D_q estimation if the ACK is incorrectly matched to the corresponding data packet.

Instead of treating the delay as a pure signal (threshold), the delay-gradient can be used to infer congestion. Use of the delay gradient was first proposed by Jain in the CARD CC algorithm which uses the normalised delay-gradient of RTT to detect congestion [62].

Hayes et al. [12] proposed a new algorithm that utilises the average smoothed delay-gradient $\bar{g_n}$ of RTT_{min} and RTT_{max} seen in the measured interval to estimate congestion level. Using the gradient of minimum and maximum RTT, and the average smoothed filter, reduces the noisiness of the RTT gradient. Depending on $\bar{g_n}$ sign and magnitude, CC algorithm can increase or decrease *cwnd*. Using this signal it is possible to distinguish between packet loss related to congestion and loss related to a noisy environment such as a wireless network.

In addition to using the *delay signal* in congestion detection, some CC algorithms use this signal in calculating *cwnd*. Moreover, it is worth noting that not only delay-based TCP CC algorithms utilise the delay signal but also other transport protocols. For example, LDA+ [63] and MLDA [64] rely on the delay measurement in additional to loss signal to control sending rate of Real-time Transport Protocol (RTP) [65] and UDP multicast respectively.

2.3.2 Explicit Congestion Feedback Signals

Explicit congestion feedback refers to explicit signals sent by the bottleneck to inform the end-host of the congested state of the bottleneck. The sender's CC algorithm should respond to the signal by reducing *cwnd*. Bottlenecks use different mechanisms to detect congestion in their buffers and mark packets when congestion is encountered. It is clear that *explicit feedback* signals needs cooperation from the bottleneck, network protocol and transport protocol in order to function.

Explicit feedback for TCP/IP is typically implemented using the Explicit Congestion Notification (ECN) extension [4]. In the forward path, ECN supports packet marking, using dedicated bits in the IP header to encode congestion state.Additionally, ECN utilises bits within the TCP header which are used to inform the sender that congestion has happened and an action has been taken place.

If the router supports ECN, it marks the packet when the congestion is detected. When processing the marked packet, the receiver sets a flag in the ACK packet. Upon receipt of an ACK with congestion flag set, the sender reduces its *cwnd*.

To support ECN, a bottleneck router needs to detect and signal congestion before complete exhaustion of available buffer space, such as using AQM in place of the traditional Droptail mechanism (as described in section 2.2). The AQM can then mark ECN-capable IP packets when congestion is experienced (rather than drop them).

As originally proposed [4], TCP was expected to react to an ECN signal in the same way as to packet loss (e.g. halving *cwnd* for TCP Reno). However, a new proposal suggests that *cwnd* should be reduced by a smaller amount for an ECN signal than for packet loss as the ECN signal is likely generated by an AQM-enabled bottleneck emulating a small queue [66]. This new proposal can improve TCP throughput without causing network collapse.

While using *explicit feedback* for CC reduces packet loss and can improve overall network performance, there are some difficulties. Middleboxes, including old, non-ECN aware firewalls, intrusion detection systems, and load balancers, may respond with an RST (reset connection) packet or drop the packet silently when processing a packet with the ECN enable flags set.

Another issue involves non-compliant hosts which pretend to support ECN during connection negotiation but do not respond to marked packets. This results in the receiver attaining higher throughput than other flows sharing the bottleneck, and increased congestion in the bottleneck.

This vulnerability can be addressed in the AQM by dropping packets instead of marking them when congestion exceeds a specific threshold. For example, PIE has a safeguard against such behaviour by dropping ECN enabled packets when queuing delay becomes high, and the sender will reduce *cwnd* as a reaction to packet loss. An alternate solution is to segregate flows into queues using scheduling techniques such as FQ-CoDel or FQ-PIE (section 2.2.2), thereby isolating well-behaved flows from unresponsive flows.

ECN-based CC approaches rely on explicit congestion feedback to estimate congestion intensity and react (by reducing *cwnd*) in different degrees based on that intensity. The purpose of this type of CC is to provide a low-latency and high-throughput protocol with low loss rate for controlled network environments such as data centres. Unlike AQM configured for conventional TCP, ECN-based CC algorithms require AQMs to mark packets much earlier (having lower target delay/occupancy and very short burst tolerance) to provide very low latency. The throughput of ECN-based CC will not be affected by such very shallow buffer emulation since *cwnd* back-off factor is adaptive based on congestion intensity. A well-known example of ECN-based CC approaches are DCTCP [67], Deadline-Aware Data Center TCP (D2TCP) [68] and L2DCT [69]. It is worth noting that a recent study proposes an architecture called L4S [70] to use ECN-based CC on the Internet by utilising a special type of AQM (e.g. DualQ Coupled AQM [71]). The proposed AQM separates classic TCP flows from ECN-based flows in two different queues and then priority packet scheduler is used to provide fair bandwidth share.

2.3.3 TCP Congestion Control Algorithms

Standard TCP deploys a combination of techniques (slow start (SS), congestion avoidance (CA), fast retransmit and fast recovery) to respond topacket losses [32]. SS is used to probe the link capacity when no previous information is available about the link. CA aims to prevent heavy network congestion while adapting to changes in network conditions. Fast retransmit and fast recovery are used to quickly resend the missing packets and recover from theses losses. In this section, we describe these algorithms in some detail as they are considered the base for most TCP variants.

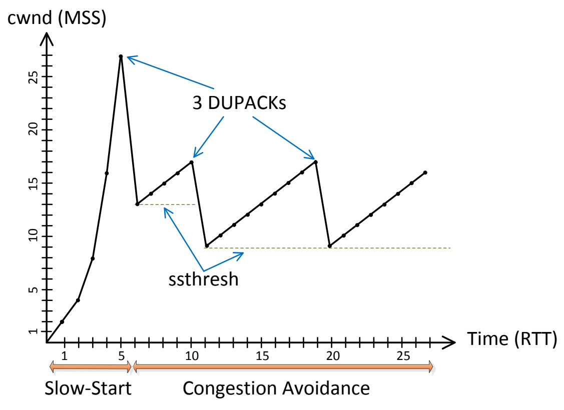
2.3.3.1 Slow Start

In the absence of specific network feedback, a TCP sender is typically unaware of path capacity when a connection is first established. TCP uses the slow start algorithm to initially probe a path's capacity.

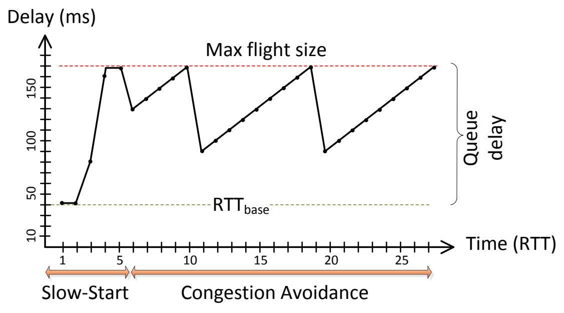
To begin a new connection in SS mode, TCP sets *cwnd* to the Initial Window (IW) and transmits this IW of bytes. For each received ACK that acknowledges new data, *cwnd* is increased by no more that one MSS, typically doubling *cwnd* every RTT. This mode is referred to as *slow* start because the sender does not begin with *cwnd* set to some large (and potentially excessive) initial value.

TCP exits SS and enters congestion avoidance mode when congestion is detected. Path capacity is estimated as a portion of *cwnd* that was realised when the congestion was detected.

To distinguish between SS and CA modes, TCP maintains a state variable called slow start threshold (*ssthresh*). The SS algorithm runs when $cwnd < ssthresh$, otherwise the CA algorithm is executed. Initially, *ssthresh* is set to a high value to allow the SS algorithm



(a) *cwnd* vs time



(b) RTT vs time

Figure 2.10: TCP Reno Slow-Start and Congestion Avoidance

to probe available bandwidth quickly. Following each congestion event, *ssthresh* is set as a multiple (usually half) of *cwnd*.

Figure 2.10 plots *cwnd* progression and RTT versus time for TCP Reno. The first six time samples cover the SS phase of the flow. The exponential growth of *cwnd* is shown in Figure 2.10a where *cwnd* increases to about 27 MSS at time 5RTT. In Figure 2.10b, we also see that RTT increases above the base RTT of 40ms after time 2RTT once *cwnd* becomes larger than the path BDP. RTT continues to increase to about 170ms when the bottleneck queue is full and packets are dropped.

When packet loss is used to detect congestion, the exponential growth of *cwnd* can lead to a large overshoot of the optimum value. This can result in high packet loss within one RTT, leading to waste in network bandwidth, long unresponsive periods and increased loads on the end-host operating systems during the loss recovery period [72]. This problem is exacerbated in networks with large BDPs network as *cwnd* grows to a large size and there is an increased time period before congestion feedback is noticed by the sender.

Consequently, improvements have been proposed to find a safe exit point from SS without resulting in high packet loss and low bandwidth utilisation. One proposed algorithm is Hybrid Start (HyStart) [72] which uses the ACK trains technique and sampled RTT to find a safe exit point to CA.

2.3.3.2 Congestion Avoidance

During congestion avoidance (CA), TCP CC tries to avoid congestion by increasing *cwnd* slowly during periods of no congestion, and reducing it significantly when congestion is detected. Some algorithms try to stabilise *cwnd* when they infer that the available bandwidth is fully utilised.

The standard technique for maintaining *cwnd* is the Additive Increase/Multiplicative Decrease (AIMD) algorithm, where *cwnd* increases linearly by α once per RTT and multiplicatively decreases it by β (where α and β are algorithm specific constants). For example, TCP Reno uses $\alpha = 1$ and $\beta = 0.5$ [32], resulting in *cwnd* increasing by no more than one MSS bytes per RTT, and being halved when congestion is detected.

In order to keep $cwnd \geq BDP$ and achieve full link utilisation (see Section 2.1.1), *cwnd* should be greater than $2 \times BDP$ upon packet loss. This can be achieved only if the bottleneck buffer size equals at least the BDP of the connection. Otherwise, after backoff *cwnd* will be dropped to less than BDP and require multiple RTTs in CA mode to regrow back to BDP. This can cause severe degradation of throughput in large BDP networks.

To simplify implementation, the additive increase can be performed using the appropriate byte counting method. The number of acknowledged bytes is accumulated until they become greater than *cwnd*, and then *cwnd* is increased by one MSS. Another formula that can be used to update *cwnd* is given in equation 2.1.

$$cwnd_{i+1} = cwnd_i + MSS * \frac{MSS}{cwnd_i} \tag{2.1}$$

When congestion is detected via RTO firing, TCP resets *cwnd* to one MSS and sets *ssthresh* to no more than half of flight size. The missing packets are then resent and TCP reenters SS.

When congestion is detected via 3DUPACK, TCP enters the fast retransmit and fast recovery phase.

2.3.3.3 Fast Retransmit and Fast Recovery

TCP Tahoe enters a *fast retransmission* phase when packet loss is detected via 3DUPACK. The missing packet is immediately retransmitted, *ssthresh* is set to half of *cwnd* and *cwnd* is reset to one MSS. TCP then enters SS.

TCP Reno augments the response to 3DUPACKs with fast retransmission and *fast recovery* [32]. The missing packet is immediately retransmitted, *ssthresh* is set to half of *cwnd* and *cwnd* is set to *ssthresh* plus $3 \times MSS$. This inflates the congestion window to reflect the three packets that departed the host after the missing packet. *cwnd* is subsequently incremented by one MSS for each additional duplicate ACK received as a reflection of the additional packets delivered to the destination. When new data is ready to be sent, TCP should send one MSS worth of bytes if *cwnd* allows.

Fast recovery finishes by receiving a new ACK that acknowledges unacknowledged data. After that, TCP Reno sets *cwnd* to *ssthresh*, and enters CA.

TCP Reno fast recovery is inefficient when multiple packets losses occur in the same transmission window since the cumulative acknowledgement doesn't reflect losses after the first missing packet. A new fast recovery algorithm called NewReno was proposed to address this issue [37].

Another solution to multiple losses within a window is to use the TCP Selective Acknowledgment (SACK) option [38]. SACK allows the receiver to inform the sender the exact sequence of bytes that have been received so the sender need only resend the missing segments without waiting for multiple RTTs. The number of SACK blocks (range of received bytes) within one packet is limited by the TCP options field. As a consequence, SACK may not be able to provide all received byte ranges if many non-contiguous losses occur within one window.

2.3.4 Congestion Control Metrics

It is important to understand congestion control evaluation metrics to be able to study and compare the performance of different CC algorithms. Various CC algorithms are designed with different aims and working environments. Some protocols focus on improving specific metrics while others focus on trading-off multiple metrics. The main metrics for evaluating congestion control mechanisms are [73]:

- Throughput: The amount of data sent per time interval. Can be measured for routers as aggregate link utilisation, for flows as connection transfer times, and for users as user wait times.
- Delay: Measures the additional queuing delay caused by the CC algorithm.
- Packet Loss Rate: Measures wastage of network resources.
- Fairness: The degree of equality in resource allocation.
- Convergence time: The time required to for flows to converge to fairness.

CC algorithms often need to consider a trade-off between metrics. For example, loss-based CC typically has higher throughput at the expense of increased delay and packet loss. Alternatively, delay-based CC has improved delays and packet loss at the expense of lower throughput when competing with loss-based flows. To achieve both high throughput and low queuing delay, CC schemes aim to maximise the *power* metric [74]. Power metric is given in Equation 2.2 where x is flow's throughput, RTT is the current round trip time and α is a constant. If $\alpha > 1$, power metric will give a preference to throughput over the response time of the flow (higher throughput, higher queuing delay). If $\alpha < 1$, this metric gives a preference for the response time (lower throughput, lower queuing delay).

$$power = \frac{x^{\alpha}}{RTT} \tag{2.2}$$

The choice of CC algorithms and preferred performance metrics are influenced by application requirements. However, in environments that include a mixture of loss and delay based CC, the ideal outcome is typically unachievable. Pure delay-based CC can suffer from unfair resource allocation as they typically defer to loss-based CC due to the high queue occupancy caused by loss-based CC.

Ensuring fair capacity sharing is not a trivial task, especially when different CC algorithms coexist over the same path, or when flows travels over different path distances. One of the most commonly used indices for measuring resource sharing fairness is Jain's fairness index [75] [73]. Jain's index is given in Equation 2.3 where n is number of flows and x_i is the throughput of the ith flow. This index ranges from 0 to 1, and is at a maximum when all flows receive the same allocation.

$$fairness = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2} \tag{2.3}$$

When a new flow joins a shared bottleneck, bandwidth should be reallocated for all competing flows. Convergence time in a high BDP environment is important as large amounts of data can be transferred over short time intervals.

One measurement of convergence time is the delta-fair convergence time [76]. This measures the time taken for two flows to go from a fully unfair share of the link capacity, to having near fair sharing of link capacity. The time is calculated as per Equation 2.4, where B is the total bandwidth, δ is the fairness wants converge to and b_0 is the initial bandwidth allocated to the new flow.

$$\delta - fair conv = (B - b_0, b_0) \rightarrow (\frac{1+\delta}{2}B, \frac{1-\delta}{2}B) \tag{2.4}$$

In addition to these metrics, robustness to noisy environments, misbehaving users, minimising *cwnd* oscillations and dependability are considered important in many cases.

2.4 CC algorithms that utilise the delay signal

We propose a taxonomy (Figure 2.11) to categorise the behaviour of TCP Congestion Control Algorithms with respect to their use of different congestion signals. Within this taxonomy we define primary categories of: 1) ECN-Based which use explicit congestion notification as described in Section 2.3.2; 2) Loss-Based which primarily rely on packet loss as a congestion signal; 3) Delay Based which primarily rely on the delay signal; 4) Hybrid which use a combination of both Loss and Delay signals; and 5) Bandwidth Estimation Delay Sensitive which use link capacity estimation and delay measurements to regulate the sending rate.

We further sub-classify Loss Based algorithms into Pure Loss Based approaches and Delay Sensitive algorithms to differentiate those that may occasionally use the loss signal to achieve their aims. We also sub-classify hybrid algorithms into Dual Signal and Dual mode algorithms.

Dual signal approaches utilise both loss and delay signals. The delay signal allows the CC algorithm to scale quickly without stressing the network. They are usually deployed in large BDP networks where traditional loss-based approaches can be slow to achieve high link utilisation.

Dual mode approaches alternate between using loss and delay signals based on internal state. They typically use the delay signal to infer early congestion and better manage queue latency, and revert to using the loss signal when competing with loss-based flows to achieve reasonable inter-flow fairness.

Pure loss based and ECN-based approaches are out of scope for the rest of this chapter.

In general, CC algorithms that utilise the delay signal use RTT or OWD based metrics to detect the degree of network congestion. Although different CC algorithms may use customised metrics, common delay metrics are queuing delay, queue occupancy or delay gradient. Algorithms that estimate queuing delay attempt to keep latency under a predefined time threshold. Algorithms that estimate queue occupancy attempt to keep bottleneck buffer utilisation to a specific threshold (bytes or packets) or to perform early detection of congestion events. Delay gradient algorithms avoid the use of thresholds to avoid issues around RTT and base RTT estimation.

2.4.1 Delay-based Algorithms

Most delay-based CC algorithms are *threshold-based* which infer early congestion in the network when the measured delay signal exceeds a pre-configured or dynamically calculated threshold or thresholds. A few use *delay-gradient* approaches to infer network congestion by monitoring congestion trends in bottleneck buffer and make decisions based on the rate of change of queuing delay.

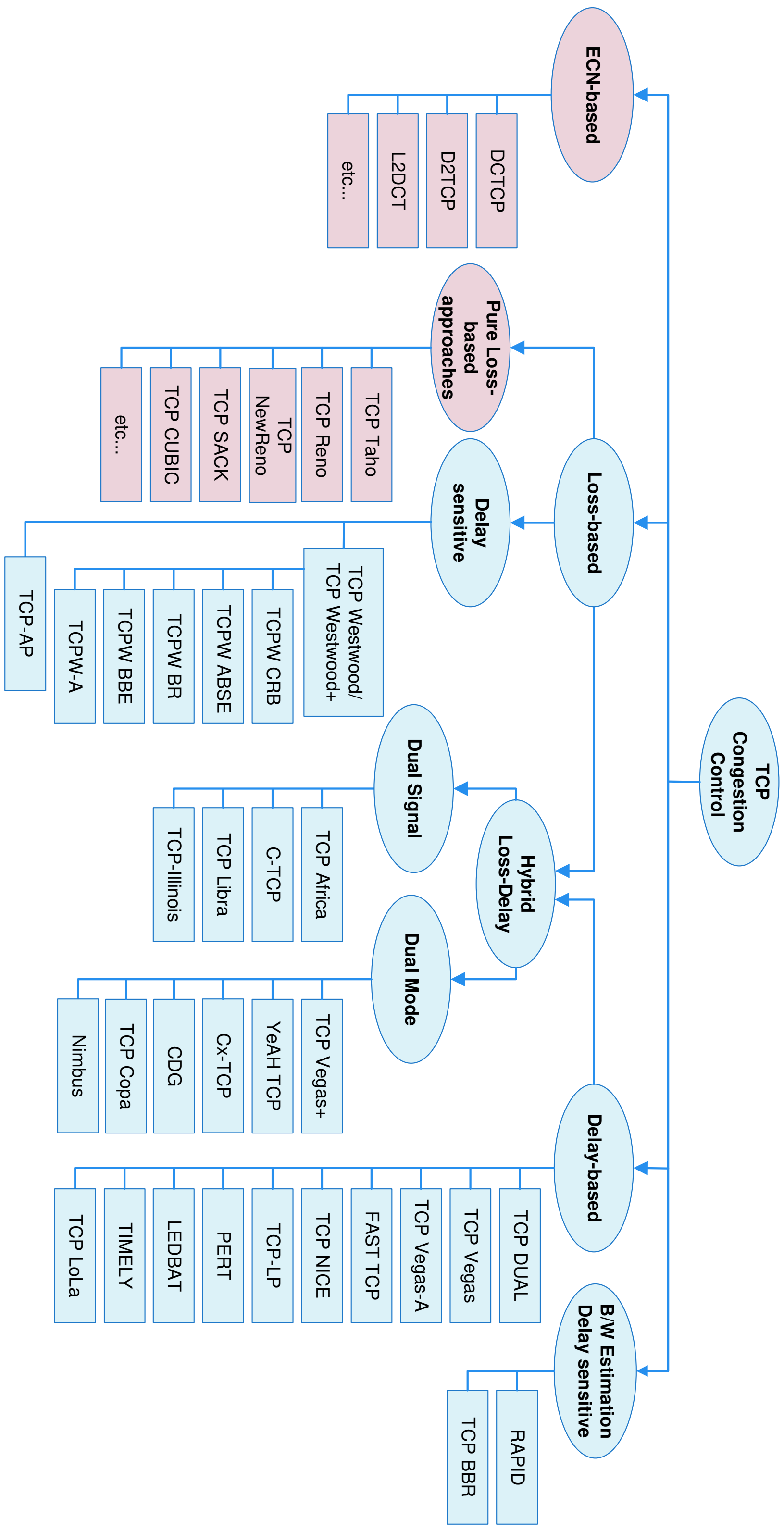


Figure 2.11: Taxonomy of TCP congestion control techniques reviewed in this survey

Most delay-based CC algorithms aim for high link utilisation with short bottleneck queues. However, others are designed for background bulk file transfer applications, and typically aim is to achieve a lower bandwidth share when competing with standard flows. Delay-based CC can achieve high throughput by reducing the oscillatory *cwnd* behaviour of standard TCP by slightly reducing the window size when the queuing delay reaches a defined threshold.

Unfortunately, CC algorithms of this category typically suffer from unfair resource allocation when sharing a bottleneck with loss-based CC and experience the latecomer advantage problem described in Section 2.3.1.2. Table 2.1 summarises the properties of the delay-based TCP variants reviewed in this section.

2.4.1.1 TCP DUAL

In 1992, Wang and Crowcroft [77] proposed an enhanced algorithm, called TCP DUAL, to minimise the oscillation of TCP Tahoe's *cwnd* dynamic in CA mode. Dampening these oscillation helps reduce fluctuations in buffer utilisation that can lead to RTT instability and periodic packets losses.

TCP DUAL estimates the queuing delay using RTT measurement to indicate the network congestion level and reduces *cwnd* before a packet loss happens. The algorithm assumes that the base RTT is RTT_{min} , and RTT_{max} represents the RTT of the highest congestion level along the path. RTT_{min} and RTT_{max} measurements are reset whenever RTO is triggered. At any time, the two-way queue delay (Q_i) can be estimated shown in equation 2.5.

$$Q_i = RTT_i - RTT_{min} \tag{2.5}$$

DUAL attempts to keep the queuing delay at a point between the minimum and maximum queue delay ($Q_i \rightarrow \delta \times Q_{max}$) where $Q_{max} = RTT_{max} - RTT_{min}$ and $0 < \delta < 1$ ($\delta = 0.5$ is used in [77]). In other words, it attempts to keep the RTT close to a threshold th where $th = (RTT_{max} + RTT_{min}) \times \delta$.

On every other received ACK, if $RTT_i > th$, TCP DUAL multiplicatively decreases *cwnd* by 7/8 to fine tune the RTT around th. If network congestion is detected using RTO mechanism, TCP DUAL behaves similarly to TCP Tahoe i.e. *ssthresh* = *cwnd*/2, *cwnd*=1 and moves to SS phase. If no congestion is detected using the delay or loss signals, TCP DUAL increases *cwnd* by one MSS every RTT.

Due to the delay component of TCP DUAL, this algorithm is affected by the latecomer advantage unfairness problem. Additionally, in common with other delay based back-off schemes, TCP DUAL flows can suffer from low bandwidth sharing when competing with loss-based flows.

Table 2.1: Delay-based TCP variants reviewed in Section 2.4.1

TCP variant	**Section**	**Algorithm aims**	**Signal type**	**Metrics**
TCP DUAL [77]	2.4.1.1	minimise *cwnd* oscillation of TCP Tahoe, better throughput	RTT	queuing delay
TCP Vegas [33]	2.4.1.2	minimise *cwnd* oscillation, low queuing delay, better throughput	RTT	queue occupancy
TCP Vegas-A [60]	2.4.1.3	remedy TCP Vegas path re-routing and fairness issues	RTT	queue occupancy
FAST TCP [78, 79]	2.4.1.4	scalability in large BDP networks, low queuing delay	RTT	queue occupancy
TCP NICE [9]	2.4.1.5	low priority, low queuing delay	RTT	queue delay
TCP-LP [58]	2.4.1.6	low priority, low queuing delay	OWD	queue delay
TCP PERT [80]	2.4.1.7	low queuing delay	RTT	queue delay
LEDBAT [8]	2.4.1.8	low priority, low queuing delay	OWD	queue delay
TIMELY [81]	2.4.1.9	low queuing delay in datacentre environments	RTT	delay gradient
TCP LoLa [82]	2.4.1.10	scalability in large BDP networks, low queuing delay	RTT	queue delay

2.4.1.2 TCP Vegas

An early, well known delay-based TCP CC, TCP Vegas [33] aims to achieve maximum throughput, low packet loss and queuing delay, with minimum *cwnd* oscillation. It uses *cwnd*, the current RTT and RTT_{base} (derived from the minimum witnessed RTT) to regularly estimate the number of in-flight bytes that reside in the bottleneck buffer, while aiming to keep this number small.

TCP Vegas deploys an Additive Increase Additive Decrease (AIAD) approach when congestion is controlled using the delay component. *cwnd* is increased by at most one MSS every RTT, ensuring the algorithm is not more aggressive than standard TCP. When packet loss is detected, TCP Vegas mimics standard TCP by halving *cwnd*.

During CA, TCP Vegas calculates the difference between the expected and actual sending rate to estimate the data currently queued at the bottleneck. The expected sending rate can be calculated as the throughput when no congestion is present in the bottleneck i.e. the RTT equals RTT_{min}:

$$expected\, rate = \frac{cwnd_i}{RTT_{min}}$$

The actual sending rate is the calculated based on the actual RTT (RTT_i) :

$$actual\, rate = \frac{cwnd_i}{RTT_i}$$

To estimate the number of queued packets at the bottleneck Δ, the difference between the rates is multiplied by RTT_{min}:

$$\Delta = (expected\, rate - actual\, rate) \times RTT_{min} \tag{2.6}$$

TCP Vegas calculates Δ upon receipt of each ACK and compares Δ with algorithm parameters α and β (default $\alpha = 1$ and $\beta = 3$). These parameters control the length and stability of the bottleneck queue. The protocol aims to maintain the queue length between α and β.

When $\Delta < \alpha$, TCP Vegas increases *cwnd* by one segment during the next RTT. When $\Delta > \beta$, network congestion is inferred and *cwnd* is decreased by one segment during the next RTT. Otherwise *cwnd* is left unchanged.

The rationale of this algorithm is that if a sender can send a *cwnd* worth of data without observing a large increase in RTT, this means the link is under-utilised and we can increase *cwnd*. Alternatively, if the increase in RTT is such that Δ exceeds threshold β, this means the link is over-utilised and we should reduce *cwnd*.

TCP Vegas also proposes an enhancement to the TCP Reno SS mechanism to detect the available bandwidth and exit SS before packet loss occurs. Specifically, it exits SS when $\Delta > \alpha$ where Δ is calculated as above upon receipt of every other ACK. TCP Vegas also introduces a more timely technique to detect loss before receiving the third duplicate ACK.

Although these modifications aim to reduce the stress on the network, experimental results [33] show a very small impact on overall network performance due to the short working time of SS and fast retransmission compared with CA.

Barkmo and Peterson [33] claim that TCP Vegas is able to achieve 37% to 71% better throughput and reduction in packet losses by 1/5 to 1/2 than TCP Reno on the Internet. Unfortunately, later studies [34, 83–86] demonstrate a number of issues with TCP Vegas including 1) low fair share of bandwidth when competing with Reno-style flows (loss-based CC); 2) low throughput following sudden increases in base RTT (eg. path rerouting); and 3) latecomer advantage (section 2.3.1.2) due to incorrect base RTT estimation.

2.4.1.3 TCP Vegas-A

Despite the limitations, TCP Vegas still displays desirable characteristics. Srijith et al. [60] claim that the re-routing and fairness problems when sharing a bottleneck with Reno flows can be remedied by dynamically adapting α and β coefficients based on the actual sending rate. They propose a modification to TCP Vegas called TCP Vegas-A.

Vegas-A uses the default α and β values (1 and 3 respectively) at the start of the connection, and keeps these values as the minimum boundaries. After that, Vegas-A dynamically changes these values based on the network conditions.

When $\alpha < \Delta < \beta$, the algorithm is in steady state. Vegas-A attempts to probe the available bandwidth to adjust α and β to maximise throughput. When Vegas-A detects an increase in the actual rate, α, β and *cwnd* are incremented.When $\alpha > 1$, $\Delta < \alpha$ and there is a decrease in the actual transmission rate, Vegas-A assumes that the coefficients have been over-estimated and decreases α, β and *cwnd*. When $\alpha > 1$, $\Delta < \alpha$ and there is an increase in the actual transmission rate, *cwnd* is increased. Otherwise, the algorithm adjusts *cwnd* using the TCP Vegas rules.

Using ns-2 simulations, Srijith et al. [60] show overall improvement with TCP Vegas-A compared to TCP Vegas for the path re-routing issue over both wired and fluctuating RTT satellite links. They also show better fairness when Vegas-A competes with TCP Reno flows.

However, Vegas-A is still unable to obtain fair capacity sharing with Reno-style flows when the number of flows is small. Further, when the number of TCP Vegas-A flows becomes relatively high, the fairness problem inverts and the Vegas-A flows get a higher capacity share than Reno flows.

2.4.1.4 FAST TCP

Inspired by the TCP Vegas idea of controlling congestion based primarily on the delay signal, Jin et al. [78, 79], proposed FAST TCP, targeting low queuing delays and high bandwidth utilisation in large BDP paths.

Similar to TCP Vegas in the steady state, FAST tries to maintain a fixed number of packets (α) in the bottleneck queue by using RTT_{base} (derived from observed RTT_{min}) and the current average RTT. Instead of adjusting *cwnd* by one MSS every RTT interval, TCP FAST updates *cwnd* every fixed interval (eg. 20ms) using the specialised equation:

$$w_{i+1} = min\left\{2.w_i,\ (1-\gamma).w_i + \gamma\left(\frac{RTT_{min}}{RTT_i}w_i + \alpha\right)\right\}$$

The window smoothing factor (γ) is a configurable parameter between 0 and 1 that affects the window update response to congestion. The target number of packets in the queue (α) is a constant that controls the protocol fairness.

Selection of α is an open challenge but the authors of FAST TCP used large values (eg. 200) in their experimental evaluation [79]. The window adjustment step is large when the the current RTT is close to the base RTT, and small as the protocol approaches the steady state (α packets buffered in the queue).

FAST TCP also uses packet pacing to control the burstiness of the congestion window mechanism in a large BDP environment and to provide accurate RTT measurement.

Emulated network and simulation based experiments show good overall throughput, scalability, stability, RTT-, inter- and intra-fairness for FAST TCP [78, 79].

However, Tan et al. [87] show unfairness problems and large variation in queue occupancy related to inaccurate propagation delay estimation, as for TCP Vegas. This inaccurate estimation happens during route change and standing queue scenarios. They also find that the FAST TCP *cwnd* update rule is more aggressive than standard TCP which can cause unfairness in specific scenarios.

Unlike most congestion control algorithms, FAST TCP is a commercial CC algorithm and is protected by patents [88, 89]. It is one of the few delay-based algorithms that is actually used in practice over the Internet.

2.4.1.5 TCP NICE

TCP Nice [9] is a scavenger class CC algorithm based on TCP Vegas with a more sensitive congestion detection mechanism. TCP Vegas by itself provides low priority CC due to its proactive reaction to congestion, but not low enough to be scavenger class CC.

During one RTT interval, TCP Nice counts the number of times that the estimated bottleneck queuing delay is greater than a fraction of the maximum queuing delay. In other words, the number of times measured RTT is larger than $RTT_{min} + (RTT_{max} - RTT_{min}) \times threshold$), where *threshold* defines the target fraction.

If the count is greater than a *fraction* of the congestion window, TCP Nice halves *cwnd*, otherwise it behaves like TCP Vegas. When loss is detected, TCP Nice halves *cwnd*.

Another mechanism used to ensure low priority is that *cwnd* is allowed to decrease to a value lower than one, this postpones packet delivery for a number of RTTs.

Although TCP NICE flows realise low throughput when competing with Reno-style flows, there is a concern about how well NICE can utilise the available capacity when just LPCC flows exist. As with TCP Vegas, NICE also experiences the latecomer advance problem due to incorrect base RTT estimation.

2.4.1.6 TCP-LP

TCP-LP [58] is an LPCC algorithm that aims to utilise available bandwidth without affecting foreground TCP flows. Unlike TCP NICE, TCP-LP uses EWMA smoothing of one-way delay measurements to infer queuing delay in the forward path, avoiding delay fluctuations caused by reverse path traffic.

TCP-LP uses the TCP timestamps option [57] to calculate a form of OWD as the difference between receiver's timestamp in the ACK packet and the sender timestamp copied to the ACK. Without synchronised clocks this is not a true OWD (section 2.3.1.2), so TCP-LP utilises the minimum and maximum measurements to calculate one-way *queuing* delay.

TCP-LP infers early congestion in the forward path when equation 2.7 is true (a similar strategy to that used by TCP Nice), where we are measuring if queuing delay is greater than a fraction of the maximum queuing delay.

$$OWD_i > OWD_{min} + (OWD_{max} - OWD_{min}) \times th \tag{2.7}$$

TCP-LP reduces *cwnd* more aggressively than standard TCP when congestion is detected. At the first sign of congestion, *cwnd* is halved. If another congestion event occurs within one RTT, TCP-LP infers that persistent congestion exists in the network and subsequently sets *cwnd* to one MSS. During periods of no congestion, *cwnd* is increased similar to TCP Reno.

Using ns-2 simulation and a real-world Linux implementation, the authors of TCP-LP [58] show that TCP-LP achieves its design goals of yielding available bandwidth to competing standard TCP flows, and high bandwidth utilisation with good fairness when no high priority flows are competing.

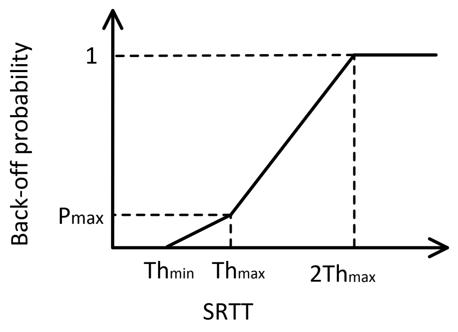


Figure 2.12: PERT back-off probability function

It is unknown how well TCP-LP will achieve its goals in an AQM or wireless environment. More evaluation is required for this and similar techniques in different scenarios.

2.4.1.7 PERT

Probabilistic Early Response TCP (PERT) [80] is a delay-based algorithm that emulates AQM and the end-host without modification to the bottleneck. PERT authors claim that any AQM can be emulated at the end-hosts but they choose RED [43] and PI [90] AQM.

PERT uses a probabilistic back-off function ($P_{backoff}$) based on delay measurements at the end host. On every receiving ACK, PERT smooths the instantaneous RTT_i sample using Exponentially Weighted Moving Average (EWMA) to produce $SRTT_i$ to reduce signal noise. Then it uses $SRTT_i$ and RTT_{min} to calculate back-off probability.

PERT defines three thresholds th_{min} (defaults to RTT_{min}+ $5ms$) , th_{max}(defaults to RTT_{min}+ $10ms$) and P_{max} (defaults to 0.05). By using RTT_{min}, PERT estimates instantaneous queuing delay similar to TCP DUAL. $P_{backoff}$ is zero if $SRTT_i$ is less than th_{min}. $P_{backoff}$ increases linearly until it reaches P_{max} when $SRTT_i$ equals th_{max}. Then, $P_{backoff}$ increases faster to reach one when $SRTT_i$ becomes larger than or equal $2.th_{max}$. Figure 2.12 shows the back-off probability function that PERT uses.

PERT uses 0.65 multiplication decrease factor if the congestion is detected using the delay component and 0.5 if packet loss occurs. Additionally, it responds to congestion once every RTT since the effect of back-off is not observed until after an RTT. This reduces the number backing off times per congestion event.

Using ns-2 network simulator[91] and fluid mathematical model, PERT authors [80] find that this algorithm achieves very low queuing delay and all most no packet loss with good fairness between competing PERT flows. They also find that PERT is able to utilise the link in similar way as using AQM on the bottleneck. However, PERT does not solve the coexisting with loss-based flows problem and PERT authors highlight some possible solutions as future work.

Kotla et al. [92] propose a modification to PERT to allow better coexistence with loss-based flow by increasing the additive increase factor if high queuing delay is observed. However, it has been shown the this modification causes high loss rate and high queuing delay in many scenarios [93]. Without a functional coexistence mechanism, PERT cannot be deployed on the Internet since loss-based algorithms are the most widely used CC.

2.4.1.8 LEDBAT

Low Extra Delay Background Transport (LEDBAT) [8] is an LPCC that is widely implemented in different bulk transfer applications such peer-to-peer file transfer [94] and software updates. LEDBAT aims to keep the forward queuing delay relativity small to reduce interference with other flows, particularly flows used by latency sensitive applications such as Voice over IP (VoIP).

Similar to most delay-based CC, LEDBAT monitors queuing delay and considers an increase in that delay as an early signal of network congestion. By responding to this signal, LEDBAT flows defer to competing standard TCP flows.

As for TCP-LP, LEDBAT uses OWD measurement instead of RTT to avoid delay fluctuations in the reverse path.

LEDBAT utilises a predefined *target* threshold (default 100ms) for queuing delay. When no packet loss is detected, LEDBAT proportionally increases or decreases *cwnd* based on the relative difference between the *target* and estimated queuing delay (equation 2.8). If loss is detected, LEDBAT behaves like standard TCP by halving *cwnd*.

$$cwnd_{i+1} = cwnd_i + \frac{G \times MSS \times \Delta \times B_{acked}}{cwnd_i} \tag{2.8}$$

The gain scale (G) determines the *cwnd* growth/decline rate and should be no greater than one to ensure LEDBAT is not more aggressive than the standard TCP. B_{acked} is the number of newly acknowledged bytes. Δ is the normalised difference between the one-way queuing delay and *target* and is defined in equation 2.9.

$$\Delta = \frac{(target + OWD_{base} - OWD_i)}{target} \tag{2.9}$$

LEDBAT requires OWD measurement to be made for every packet transmitted by the sender in order to react accurately and quickly to changes in delay.

LEDBAT maintains a history (default ten entries) of base OWD where each element represents the measured OWD_{min} in a one minute interval. OWD_{base} is the minimum value of this list. The history is used to minimise the effect of sudden changes in base OWD estimation caused by delayed ACK, clock skew and re-routing problems.

Due to the low impact of LEDBAT flows on latency sensitive applications, BitTorrent, a very popular peer-to-peer file sharing protocol, uses this algorithm in its UDP-based transport protocol [95, 96]. Additionally, Apple Inc. implemented TCP-based LEDBAT to be used for sending operating system updates to their clients [11].

A number of studies has been evaluated the performance of LEDBAT [6, 61, 97]. These studies have found that LEDBAT introduces increasing delay due to measuring its self-induced delay, and suffers from issues related to incorrect propagation delay estimation (unfairness and latecomer advantage problems). Moreover, a study found that a special care should be taken when choosing LEDBAT parameters in AQM environments since *cwnd* backs-off will be controlled using loss signal as the delay will never reach LEDBAT target delay; otherwise LEDBAT flows become too aggressive [98].

2.4.1.9 TIMELY

TIMELY [81] is a rate-based delay-gradient CC algorithm optimised to function in a datacentre environment without the need for additional support from intermediate nodes such as network switches and routers. The authors state that TIMELY can achieve high throughput while maintaining low packet latency by relying on accurate RTT measurements for congestion detection.

RTT is typically very small in a datacentre environment, and software time-stamping is too inaccurate to to measure RTT. As such, the authors suggest using hardware time-stamping provided by modern Network Interface Cards (NICs) to obtain accurate microsecond resolution RTT measurements.

TIMELY uses a delay-gradient signal similar to CDG. Due to the high accuracy of the RTT measurements, the raw RTT can be used rather than RTT_{min} and RTT_{max}.

TIMELY also uses rate-based congestion control rather than a window-based mechanism. The *Rate Computation Engine* calculates the required sending rate from the delay-gradient signal, and the packet transmission is managed by the *Rate Control Engine*.

RTT is measured once every chunk of data (16KiB - 64KiB). TIMELY uses two threshold values for RTT of T_{low} and T_{high} specifying lower and upper bounds for acceptable RTT.

TIMELY infers the network is under-utilised if $RTT < T_{low}$; or $T_{low} < RTT < T_{high}$ and the normalised RTT gradient is negative. In this case the sending rate is additively increased by δ.

TIMELY infers the network is over-utilised if $RTT > T_{high}$; or $T_{low} < RTT < T_{high}$ and the normalised RTT gradient is positive. In the first case, the sending rate is reduced using equation 2.10. In the second case, the sending rate is multiplicatively reduced in proportion to the RTT gradient and β using equation 2.11.

$$rate = rate \times \left(1 - \beta \times \left(1 - \frac{T_{high}}{RTT}\right)\right) \tag{2.10}$$

$$rate = rate \times (1 - \beta \times normalized\_gradient) \tag{2.11}$$

The authors of TIMELY claim that this is the first delay-based CC algorithm to be used in a datacenter environment and is able to achieve high throughput and low latency without ECN support. They also found a strong correlation between RTT and queue occupancy if an accurate measurement with proper sampling is used [81].

As TIMELY requires NIC hardware support, deployment is only possible where appropriate hardware is present. Further, Zhu et al. [99] found using fluid model and simulation that TIMELY converges to a stable point, but with arbitrary unfairness.

2.4.1.10 TCP LoLa

TCP LoLa [82] is another threshold delay-based congestion control algorithm that aims to achieve high link utilisation in long distance and high bandwidth networks while keeping the bottleneck queuing delay low. TCP LoLa endeavours to keep bottleneck buffer utilisation around a fixed target Q_{target} value regardless of the number of flows competing for bottleneck bandwidth. Moreover, it attempts to achieve fairness between flows having different path's RTT by using a proposed *Fair Flow Balancing* mechanism. In general, TCP LoLa is an enhanced TCP Vegas based algorithm with CUBIC *cwnd* growth function, better inter-flow RTT fairness and better RTT_{min} estimation.

More specific, TCP LoLa algorithm relies mainly on estimating the current two-way queuing delay Q_i using TCP DUAL method (Section 2.4.1.1, Equation 2.5). Similar to Vegas in SS phase, TCP LoLa uses queue occupancy based condition ($Q_i > 2.Q_{low}$) to exit SS before packets losses occur to prevent building-up long queue.

In CA phase, it uses the CUBIC TCP [49] algorithm to grow *cwnd* when Q_i is less than a predefined threshold Q_{low}. If $Q_i > Q_{low}$, LoLa enters fair flow balancing state to realise inter-RTT fairness. In this state, all flows sharing a bottleneck attempt to keep an equal number of bytes X in the bottleneck's buffer at the same time. X is calculated according to Equation

2.12 where ϕ is a constant and t is the difference between current time and time at entering the balancing state. The number of bytes in the buffer Q_{data} is estimated the same Vegas queue occupancy estimator (Equation 2.6). During the fair flow balancing if $Q_{data} < X(t)$, *cwnd* increases based on the difference between $X(t)$ and Q_{data}; otherwise *cwnd* is leaved unchanged.

$$X(t) = \left(\frac{t.1000}{\phi}\right)^3 \tag{2.12}$$

A flow exits the balancing state and enters *cwnd* holding state when $Q_i > Q_{target}$. In *cwnd* holding state, *cwnd* is kept unchanged for a certain amount of time (e.g. 250ms) to make all flows to return to normal operation state at the same time. After the holding time elapsed, *cwnd* decreases using a modified CUBIC function to realise a fully drained buffer. This allows flows to obtain a good RTT_{min} estimations.

Using a TCP LoLa Linux implementation and an emulated network testbed, TCP LoLa authors [82] state this algorithm is able to achieve high link utilisation, low queuing delay and good scalability in 100Mbps and 10Gbps links. However, the main weakness of this algorithm is it cannot coexist fairly with loss-based CC in typical FIFO queue management. It relies completely on the bottleneck (e.g. using AQM or isolating loss-based flows from low-latency flows in separate queues) to provide fair share when competing with loss-based flows. Therefore, it is practically very hard to deploy this CC protocol globally. Additionally, it is not clear how this algorithm behaves in shallow buffers and how it reacts to packet loss.

2.4.2 Dual Mode Approaches

Since delay-based CC algorithms respond to congestion feedback much earlier than loss-based CC, delay-based flows realise low link capacity sharing when competing with loss-based flows especially when bottleneck's buffer is large. Dual mode CC approaches work around that issue by switching to aggressive mode (loss mode) as soon as buffer filler flows are detected. They stay in the loss mode for an interval or until loss-based flows finish, then they return to normal delay mode. Different algorithms have different loss-based flows detection techniques but all of them use the delay signal in that matter. Table 2.2 summarises the dual mode TCP variants reviewed in this section.

2.4.2.1 TCP Vegas+

Hasegawa et al. [84] proposed TCP Vegas+ to address some of the fairness issues identified with TCP Vegas (section 2.4.1.2). This algorithm borrows the aggressive *cwnd* growth

Table 2.2: Dual mode CC approaches reviewed in Section 2.4.2

TCP variant	**Section**	**Algorithm aims**	**Signal type**	**Metrics**
TCP Vegas+ [84]	2.4.2.1	solves TCP Vegas inter-protocol fairness issue	RTT	queue occupancy
YeAH TCP [100]	2.4.2.2	scalability, low stressing on the network, low queuing delay, non-congestion related tolerance	RTT	queuing delay, queue occupancy
CDG [12]	2.4.2.3	low queuing delay, tolerance to non-congestion related losses, standard TCP compatibility	RTT	RTT_{min} and RTT_{max} gradient
Cx-TCP [101]	2.4.2.4	low queuing delay, coexistence with loss-based flows fairly	RTT	queuing delay
Copa [102]	2.4.2.5	high throughput, low queuing delay, coexistence with loss-based flows fairly	RTT	queuing delay
Nimbus [103]	2.4.2.6	high throughput, low queuing delay, coexistence with loss-based flows fairly	RTT	queuing delay, RTT_{min}

function of TCP Reno and the moderate TCP Vegas approach to produce a hybrid congestion control technique. In CA, TCP Vegas+ employs TCP Vegas algorithm when no loss-based flow is detected, and moves to TCP Reno *cwnd* increase mode if an aggressive flow is inferred to be sharing the bottleneck.

TCP Vegas+ uses the following heuristic to detect the loss-based flows based on the trend of RTT. One every received ACK, a state variable *count* is incremented by 1 if Vegas+ detects an increase in the current RTT while *cwnd* is not increased. On the contrary, it decrements *count* by 1 when RTT decreases while *cwnd* is not increased. Moreover, the algorithm halves *count* when packet loss is detected using the 3DUPACK mechanism (see Section 2.3.1.1) and resets it when the loss is detected using the RTO timer.

If *count* reached a predefined threshold (such as 8), the algorithm moves to the aggressive (loss-based) mode, and it returns to the moderate (delay-based) mode when *count* becomes zero.

The notion of the loss-based flow detection algorithm is that in a stable network, RTT should not increase when cwnd is unchanged unless there is a Reno-like flow competing for the bottleneck bandwidth. If the algorithm sees such RTT increase, it assumes another loss-based flow is sharing the bottleneck so must itself moves into the loss-based mode. On the other hand, the algorithm moves back to Vegas mode as soon as a packet loss is detected because that loss could happen due to the aggressive *cwnd* growth of TCP Vegas+ flow itself. Vegas+ uses the count threshold as an attempt to reduce the false positive detection of loss-based flows.

Although this approach attempts to solve the friendliness problem, it does not address the other issues of TCP Vegas such as rerouting problem. Additionally, in some environments that include high congestion or high RTT fluctuations (such as wireless networks), TCP Vegas+ could enter the aggressive mode and never exit from it due to wrong RTT measurements. This makes the algorithm act as loss-based most of the time, eliminating the advantages of the delay component.

Moreover, TCP Vegas+ evaluated by the authors [84] using simulated network only in which RTT measurements are not realistic since it does not reflect the noise of RTT signal in the real world or emulated environments.

2.4.2.2 YeAH TCP

Yet Another Highspeed TCP (YeAH) is a hybrid congestion control algorithm by Baiocchi et al. [100] that aims to achieve high throughput in large BDP networks but without stressing the network. YeAH originates from the observation that other high speed CC algorithms (such as HS-TCP [104] and STCP [105]) improve throughput in large BDP networks at the cost of high 'stress' on the network, causing frequent congestion events with large number of packet losses as well as high queuing delay.

Similar to TCP-Africa (Section 2.4.3.1), YeAH works on one of two modes at a time depending on the congestion level. In fast mode, the congestion window increases aggressively using STCP rules while in slow mode TCP Reno rules are applied. The decision of changing from one mode to another is also based on Vegas-like estimation of the number of packets in the bottleneck buffer and congestion level estimation (TCP DUAL-like metric). However, these estimations are redefined by YeAH in such way that RTT_{base} is the minimum RTT seen during the connection lifetime, RTT sample ($RTTmin_i$) is the minimum RTT seen during the transmission of last window (i.e. measured once per RTT) and the congestion level is calculated as a proportion to the RTT_{base} but not to the Q_{max}. Formally, it calculates the queue delay (Q_i) using equation 2.13, queue size (Δ_i) using equation 2.14 and the congestion level (L_i) using equation 2.15

$$Q_i = RTTmin_i - RTT_{base} \tag{2.13}$$

$$\Delta_i = Q_i . \left(\frac{cwnd_i}{RTTmin_i} \right) \tag{2.14}$$

$$L_i = \frac{Q_i}{RTT_{base}} \tag{2.15}$$

If $\Delta_i < \delta$ and $L < 1/\varphi$, the algorithm switches to the fast mode, otherwise the slow mode is used. δ is a tunable constant (for example, 80 packets) which governs the number of packets pushed by one flow in the bottleneck buffer. φ is another tunable constant (for example, 8) the limits the congestion level caused by all flows sharing a bottleneck.

Moreover, a precautionary de-congestion algorithm is utilised in the slow mode to control the queuing delay and buffer overflow. Whenever $\Delta_i > \delta$ and with no Reno-like greedy flows competing for the bottleneck, $cwnd$ is reduced by Δ_i and $ssthersh$ is set to $cwnd/2$ once per RTT. YeAH detects the competing greedy flows based on the mode switching behaviour of the algorithm in order to achieve inter-protocol fairness. The algorithm calculates $count_{fast}$ which is the number of RTTs the algorithm spend in the fast mode and $cwnd_{reno}$ representing an estimation for the congestion window of the greedy flows (maintained using Reno rules). If $count_{fast}$ becomes greater than a threshold, $cwnd_{reno}$ is set to $cwnd/2$ and $count_{fast}$ is reset as an indication for competing with other non-greedy flows.

The precautionary de-congestion can be applied only if the algorithm is in the slow mode and $cwnd > cwnd_{reno}$, otherwise Reno-style window growth is used.

Finally, based on TCP Westwood (Section 2.4.4.1), the algorithm exploits the queue size after packet loss to find an optimum window size when the loss is not related to network congestion. This can improve the algorithm performance in lossy environments such as wireless networks.

Experimental evaluation shows that YeAH is able to realise very good throughput and low queuing delay in fast and long distance network as well maintaining intra- and RTT-fairness and friendliness. However, this approach needs more evaluation in more complex networks and scenarios to confirm the robustness against delay signal noise and distortion.

2.4.2.3 CAIA-Delay Gradient (CDG)

CAIA-Delay Gradient (CDG) [12] is a hybrid CC algorithm that tries to maintain low queue delay and reasonable fairness by using delay-gradient CC when possible, and loss-based CC when competing with loss-based CC algorithms. CDG is also able to distinguish between congestion related and random loss, behaving differently to achieve high goodput in lossy environments such as a wireless network.

Hayes et al. [12] were inspired by an early delay-based approach proposed by Jain called CARD [62] which used a delay-gradient signal to infer early network congestion. CDG also uses delay-gradient measurements to infer bottleneck queue states (full, empty, rising and falling). The notion of estimating the queue state allows CDG to differentiate between congestion and random losses. CDG considers the loss is congestion related only if the queue state is full and never backs off *cwnd* when the losses are not congestion related.

As instantaneous RTT measurements are noisy, CDG calculates the average smoothed delay-gradient using minimum and maximum RTT seen in an RTT measured interval. When congestion is detected using the loss signal and the queue state is full, CDG halves *cwnd*. When congestion is detected using the delay gradient signal, CDG decreases *cwnd* by a back-off factor $\beta = 0.7$. To help compete with loss-based flows, CDG uses the *loss-based shadow window* technique first described in [106] and *ineffectual backoff* mechanism.

The authors of CDG [12] claim that at 1% non-congestion related packet loss, CDG achieves 65% bandwidth utilisation compared with TCP NewReno at 35% under the same network conditions. At the same time, CDG keeps bottleneck queues short (particularly compared to loss-based CC).

Despite trying to compete with loss-based CC, early back-off by CDG results in it being unable to attain fair capacity sharing with loss-based CC. For this reason, and because of its low latency, Armitage et al. [7] propose using CDG as an LPCC for home networks to reduce the impact of background traffic on latency-sensitive applications. Tangenes et al. [107] evaluated CDG and also concluded that it is a good candidate to be used as a deadline-aware LPCC as its priority can be dynamically adapted using the scaling parameter G from

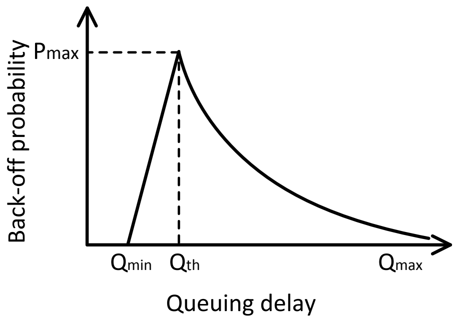


Figure 2.13: Cx-TCP back-off probability function

equation 2.27. We will describe CDG algorithm in more details in Section 2.7.

2.4.2.4 Cx-TCP

Coexistent TCP (Cx-TCP) [101] is another loss-delay hybrid congestion control that attempts to provide low latency transport while achieving better coexistence with loss-based flows. Budzisz et al. [101] were inspired by the Probabilistic Early Response TCP (PERT) [80] algorithm to use a probabilistic back-off function based on delay measurements at the end host.

The main difference between PERT and Cx-TCP is the back-off probability function behaviour. PERT back-off probability function increases when the delay exceeds a threshold (th_{min}) until it becomes one when the delay exceeds another threshold ($2.th_{max}$). On the other hand, Cx-TCP back-off probability function increases until the queuing delay exceeds a specific threshold (Q_{th}). After that point, the back-off probability decreases as the queuing delay increases. When the queuing delay reaches the maximum value (Q_{max}), the probability becomes a very small value (i.e. protocol becomes a full loss-based). This function allows Cx-TCP to coexist with loss-based flows more fairly.

The rationale of the Cx-TCP back-off probability function is that competing delay-based flows do not introduce a large queuing delay, so queuing delay should generally be lower than Q_{th}. However, competing loss-based flows will cause the queuing delay to exceed Q_{th}. Therefore, Cx-TCP should reduce the number of back-off events to better coexist with loss-based flows. When the loss-based flows leave the bottleneck, Cx-TCP reverts to its low latency mode as the queuing delay decreases below Q_{th}.

Using analytical model and simulation, Budzisz et al. [101] show that Cx-TCP is able to achieve better coexistence when competing with loss-based flows while maintaining low queuing delay in the absence of loss-based flows. However, this algorithm assumes that the sender is able to obtain accurate queuing delay measurements which is hard to achieve in many realistic scenarios. Additionally, Cx-TCP flows may obtain a low bandwidth share if the bottleneck has a shallow buffer that does not allow queuing delay above the Q_{th} threshold.

2.4.2.5 Copa

Arun et al. proposes a new loss-delay hybrid congestion control for the Internet called Copa [102]. Copa aims to achieve high throughput and low queuing delay and to coexist with loss-based flows fairly.

The authors of this algorithm state that the bottleneck bandwidth can be estimated as the inverse of the queuing delay. Therefore, they define target sending rate th_{target} to be sending rate at which the sender can transmit to achieve full bandwidth utilisation and low latency.

th_{target} is calculated as $th_{target} = 1/(\delta.Q_i)$ where δ is an adaptive parameter that controls the tread-off between throughput and queuing delay, and Q_i is the estimated queuing delay calculated similar to DUAL (2.4.1.1) but using $RTT_{standing}$ instead of current RTT. $RTT_{standing}$ is RTT_{min} measured in the previous $RTT/2$ interval to remedy ACK compression and signal noise. Copa also calculates the actual sending rate th is calculated similar to Vegas (2.4.1.2) but also using $RTT_{standing}$ instead of current RTT.

On each receiving ACK, If $th_{target} > th$, *cwnd* increases by $v/(\delta.cwnd)$ otherwise *cwnd* decreases by $v/(\delta.cwnd)$. v (defaults to 1) controls *cwnd* increase/decrease speed and its value is changed based on the direction of *cwnd* trend to make Copa flows to converge quickly to full bandwidth utilisation.

This algorithm works in another mode, called competitive mode, to be able to coexist with loss-based flows. It first detects competing loss-based flows and then adjusts δ dynamically to make Copa's flows as aggressive as loss-based flows (i.e. behaving similarly to loss-based scheme). Detecting loss-based flows is based on Copa working behaviour which involves draining the queue periodicity. Thus, if the queuing delay does not reach 10% of maximum queuing delay measured in last four RTTs intervals, buffer-filling flows are assumed to be competing with a Copa's flow. Otherwise, the algorithm works in the default mode.

Copa also changes SS exit condition to be $th_{target} < th$ which provides fast convergence with low latency. Moreover, Copa uses packet pacing to reduces traffic burntness.

Through simulation and user-space implementations, Arun et al. [102] claim that this algorithm is able to achieve similar throughput as TCP CUBIC but with much lower latency and better RTT-fairness. They also state that Copa coexists with CUBIC fairer than TCP BBR

(Section 2.4.4.5). However, it is not clear how well this algorithm performs in links with very unstable latency such as wireless networks.

2.4.2.6 Nimbus

Nimbus [103] is a rate-based loss-delay dual mode congestion control algorithm that aims to achieve low queuing delay and high throughput while fairly coexisting with loss-based flows.

Nimbus maintains a threshold-based, positive queuing delay to both ensure full link utilisation and to estimate the cross traffic rate. Nimbus calculates a sending rate equal to the bottleneck rate minus the cross traffic rate. Formally the sending rate is calculated using Equations 2.16 and 2.17.

$$D(i) = \beta \frac{C}{RTT_i}(RTT_{min} + Q_i - RTT_i) \tag{2.16}$$

$$S(i+1) = (1-\alpha)S(i) + \alpha(C - z(i)) + D(i) \tag{2.17}$$

Where α=0.8, β=0.5, C is the bottleneck capacity and z is the cross traffic rate estimation. RTT_{min} is the minimum RTT, RTT_i is the current RTT and Q_i is the current queuing delay. Bottleneck capacity C can be estimated using any bandwidth estimation technique such as in [108–111]. Due to ACK compression and other problems, Nimbus implementation uses the maximum received rate as estimation for bottleneck capacity.

Nimbus models the elasticity of cross traffic to infer the existence of competing loss-based flows, and then switches to TCP-competitive mode (CUBIC-like). This algorithm calculates the periodicity behaviour of link capacity using the Fast Fourier transform. Nimbus uses the observation of high frequency behaviour to conclude the presence of competing loss-based flows. When the frequency becomes low, Nimbus switches back to delay-based mode.

Using user-space implementation with emulated and real-world experiments, Goyal et al. [103] evaluate Nimbus and show that this algorithm is able to detect competing CUBIC and Reno like flows and achieve fair bandwidth share and low queuing delay (lower than BBR (Section 2.4.4.5)). However, the authors of this algorithm state that this algorithm is unable to detect competing BBR flows in shallow bottleneck buffers. Therefore, Nimbus flows obtain lower bandwidth share. Additionally, Nimbus considers competing delay-based flows (such as Vegas) as elastic flows and therefore it obtains much higher capacity sharing after moving to TCP-competitive mode.

Table 2.3: Dual signal TCP variants reviewed in Section 2.4.3

TCP variant	Section	Algorithm aims	Signal type	Metrics
TCP Africa [112]	2.4.3.1	scalability in large BDP networks, friendliness	RTT	queue occupancy
C-TCP [113]	2.4.3.2	scalability in large BDP networks, friendliness	RTT	queue occupancy
TCP Libra [114]	2.4.3.3	scalability and maintain the compatibility with the standard TCP, RTT fairness	RTT	queuing delay
TCP-Illinois [115]	2.4.3.4	scalability and fairness	RTT	queuing delay

2.4.3 Dual Signal Approaches

Due to the limitations of using the loss signal, some TCP CC variants introduce the delay signal in their work as a supplementary signal in addition to the loss feedback.Generally speaking, the dual signal CC approaches are designed to be scalable in fast and long-distance network environments without stressing the network by increasing *cwnd* rapidly when the queue is short and moving to slow standard *cwnd* growth after the queue becomes long. In addition to that, they attempt to maintain RTT fairness and compatibility with standard TCP. Table 2.3 summarises TCP variants that use both the delay signal (secondary signal) and loss signal reviewed in this section.

2.4.3.1 TCP Africa

King et al. [112] propose a CC algorithm called Adaptive and Fair Rapid Increase Rule for Scalable TCP (Africa) to improve TCP scalability in large BDP networks.

TCP Africa operates in one of two regimes: aggressive *cwnd* growth and conservative Reno-like *cwnd* increase. It switches between the two regimes depending on the estimated network congestion level in order to achieve fast convergence and fairness to standard TCP.

The congestion level is estimated using Vegas-like metric Δ (see Section 2.4.1.2). It then compares Δ with a threshold (α) to determine which mode should be used. If $\Delta < \alpha$, the *fast mode* is used in which *cwnd* increases aggressively uses the HS-TCP [104] CA and fast recovery rules to achieve scalability. Otherwise, TCP Africa moves to the *slow mode* in which

Reno-like *cwnd* growth style is used to achieve fairness i.e. increases the window by one MSS per RTT when no loss is detected and halves it on packet loss.

The value of α is chosen to be small constant greater than 1 (α =1.641 in [112]) and it affects the protocol performance. The authors found that no single α is optimal for all networks and conclude that more study is needed to make the value α auto-tuned.

Simulations with ns2 show that TCP Africa can scale quickly to full link utilisation, adapt quickly to network condition changes, causes low packet loss rate as well as good fairness and friendliness prosperities. However, real world experiments should be conducted to confirm these results.

2.4.3.2 Compound TCP (C-TCP)

Compound TCP (C-TCP) [113] is a compound loss-delay-based CC that aims to achieve high throughput in high speed high delay networks. Similar to TCP-Africa, it relies on the delay signal to increase *cwnd* quickly when no congestion is detected and loss signal to achieve fairness when competing with other flows.

C-TCP maintains two congestion windows, one is a standard window W_{reno} managed by Reno-style mechanism and the second is a scalable delay window W_{fast} based on TCP Vegas like algorithm. The congestion window *cwnd* that is used to control the outstanding data is the summation of W_{reno} and W_{fast}.

When Vegas queue size estimator detects small queue ($\Delta < \alpha$), W_{fast} is increased according to a modified AIMD borrowed from HS-TCP algorithm [104]. On the other hand, when Vegas estimator exceeds the threshold ($\Delta > \alpha$), W_{fast} is gradually decreased by $\zeta \times \Delta$ where ζ is a pre-defined constant ($\zeta = 30$ in [113]). This approach provides fast convergence when the queuing delay is short as well as a smooth transition from the scalable congestion control to the standard TCP style.

Real-world and simulation experiments results [113] show C-CTP exhibiting good goodput and inter/intra-fairness in a large BDP environment. C-CTP is used by default for older versions of Microsoft Windows operation system [116] and replaced by TCP CUBIC in Windows 10 Fall Creator update and Window Server 2016's 1709 update [117]. The main reason for abandoning C-TCP in Windows OS is the sensitivity of the delay component to delay fluctuations which cases low performance in many cases [117].

As the scalable component of C-TCP is basically the TCP Vegas buffer estimator, it suffers similar fairness and latecomer advantage issues when base RTT is wrongly estimated (Section 2.3.1.2).

2.4.3.3 TCP Libra

TCP Libra is a TCP CC proposed by Marfia et al. [114] to remedy the RTT-unfairness problem of NewReno when sharing a bottleneck link, improve scalability and maintain compatibility with standard TCP. TCP Libra utilises the delay signal to control *cwnd* growth/decline speed in order to become RTT independent.

Instead of increasing *cwnd* by one MSS every RTT, Libra increases the congestion window according to the equation 2.18.

$$cwnd_{i+1} = cwnd_i + \frac{\alpha_i}{cwnd_i} \frac{RTT_i^2}{RTT_i + T_0} \tag{2.18}$$

On packet loss, Libra decreases the congestion window according to equation 2.19.

$$cwnd_{i+1} = cwnd_i - \frac{T_1.cwnd_i}{2(RTT_i + T_0)} \tag{2.19}$$

where T_0 and T_1 are constant parameters (eg. $T_0 = 1$ and $T_1 = 1$). T_0 controls the algorithm's sensitivity to the RTT and T_1 is the multiplicative decrease factor. α is a control function that aims to improve the convergence speed, the scalability and the stability of the protocol, where here k_1 is a protocol constant (eg. 2) and C represents the link capacity in Mbps estimated using the CapProbe technique [118] (Equation 2.20).

$$\alpha = k_1 C p \tag{2.20}$$

p is a penalty factor that controls the the window increase step when the network is congested based on queue delay estimation borrowed from TCP DUAL (Section 2.4.1.1). p is defined in Equation 2.21 where k_2 is a constant (e.g 2) that controls link utilisation and protocol friendliness, Q and Q_{max} are the current and maximum queue delays respectively.

$$p = e^{-k_2 \frac{Q}{Q_{max}}} \tag{2.21}$$

The estimated capacity C in Equation 2.20 allows the congestion window to converge rapidly to fully utilise available bandwidth, while p reduces the window growth steps exponentially when the queue delay increases. Moreover, $RTT^2/(RTT_i + T_0)$ control RTT-fairness of the protocol in which as the RTT becomes close to T_0, the *cwnd* growth speed becomes faster.

The multiplicative decrease function (equation 2.19) shows that *cwnd* is driven not only by T_0 and T_1 but also by RTT_i. This prevents a large *cwnd* decrease after packet loss, providing better throughput in high RTT paths. However, this approach goes against the accepted logic of CC as a large RTT followed by packet loss is typically inferred as a clear sign of congestion. As such, *cwnd* decrease should be larger to control the congestion.

Using ns-2 simulator, the authors evaluate Libra and compare its performance with other TCP protocols [114], showing that Libra can achieve good fairness and high link utilisation in many scenarios. However, in some scenarios TCP Libra has lower performance comparing with TCP SACK and TCP FAST. The authors conclude the the protocol needs more evaluation using real world networks and in more complex scenarios.

2.4.3.4 TCP-Illinois

TCP-Illinois is a TCP CC proposed by Liu et al. [115] that utilises the loss signal as a primary congestion feedback and the delay signal (queuing delay) as a secondary congestion signal to improve the the scalability and fairness of TCP Reno. The underlying idea is similar to TCP Africa (Section 2.4.3.1) but rather than use HS-TCP [104] constants, the increase (α) and decrease (β) factors are set to be functions of the queue delay Q_i (similar to TCP DUAL - Section 2.4.1.1).

TCP-Illinois sets the increase factor to α_{max} if $Q_i < Q_1$ and α_{min} if $Q_i > Q_{max}$; otherwise it sets α to a concave function inversely proportional to Q_i between α_{max} and α_{min}. Moreover, it sets the decrease factor to β_{min} if $Q_i < Q_2$ and β_{max} if $Q_i > Q_3$; otherwise it sets β to a linear function directly proportional to Q_i between β_{min} and β_{max}. Q_{min} and Q_{max} are minimum and maximum queue delay seen during the connection lifetime, and Q_1, Q_2 and Q_3 are proportions (e.g. 0.01, 0.1 and 0.8 respectively) of Q_{max}. α_{min}, α_{max}, β_{min}, β_{max} are protocol constants (e.g. 0.3, 10, 0.125 and 0.5 respectively).

This algorithm updates α and β once every RTT but does not allow α to be set to α_{max} unless the queue delay stays below Q_1 for a specific amount of time (for example, 5 RTTs) to mitigate the effects of fluctuation in queue delay measurement which can be caused by noisy RTT measurement and packet bursts. Moreover, to improve the fairness with TCP NewReno, the algorithm moves to compatible mode (Reno's α and β coefficients) whenever the window size is less than a threshold (for example, 20KiB).

The authors create a mathematical model to analyse and compare TCP-Illinois with other CC algorithms, and conduct simulation-based experiments to evaluate their algorithms' performance [115]. Their results show that TCP-Illinois achieves higher link utilisation in large BDP networks compared to TCP NewReno as well as maintaining protocol intra- and inter-fairness. Moreover, the mathematical model shows that the proposed protocol causes the competing flows to backoff asynchronously which allows the congestion window of the flows to be similar in size and therefore improve overall link utilisation and fairness. However, as this approach primarily uses the loss signal to detect congestion, it cannot control the queue delay and therefore it can lead to bufferbloat problem in large bottlenecks buffers.

2.4.4 Delay-sensitive Algorithms

Some congestion control algorithms use the delay metric for specific purposes not directly related to the congestion feedback signal. These techniques use the delay signal to calculate an effective congestion window size after packet loss event and/or differentiate between congestion and non-congestion related packet losses to improve the performance in wireless networks. Others uses the delay metric to calculate reasonable bytes inflight limit to realise both high throughput and low latency. Table 2.4 summarises the delay-sensitive TCP variants reviewed in this section.

2.4.4.1 TCP Westwood/Westwood+

Mascolo et al. [119] proposed TCP Westwood (TCPW) CC algorithm to solve the low performance of NewReno in intrinsically lossy and fast networks. TCPW modifies window multiplicative decrease mechanism during fast recovery to calculate an optimum window size based on the estimated bandwidth (BWE) and RTT_{base}. The goal is for window size at any time to be approximately equal to the path's BDP, and hence achieve full link utilisation with minimal undesirable queuing.

Westwood sets the window size and *ssthresh* to $BWE \times RTT_{min}$ when packet loss is detected, which is usually a less aggressive reduction than simply halving *cwnd*. TCPW uses RTT_{min} as an estimation for RTT_{base}. RTT_{min} is measured as the smallest RTT sample seen during the connection lifetime.

Westwood's strategy for estimating the bandwidth relies on detecting the ACK receiving rate. If the receiver generates an ACK packet directly after receiving a data packet, the rate of ACK packets observed by the sender will be same as the rate of data packets received by the receiver i.e. the receiving rate equals the rate that the bottleneck supports. Then, the estimation of the utilised bandwidth in the forward path can be calculated by multiplying the ACK rate by number of bytes acknowledged by ACK packet. Even when some ACKs are lost or the receiver decides to use delayed ACK mechanism, the calculation of long-term estimation will not be significantly affected. Although the delayed ACK and ACK packet loss reduce the ACK rate, the ACK packet will carry acknowledgement for larger amount of data leading to an acceptable estimation.

TCP Westwood applies two-stage bandwidth estimation procedure to reduce the impact of fluctuation on the measurement. In the first stage, the bandwidth sampleb_k is calculated as $b_k = d_k/\Delta_k$where d_k is the amount of data that has been acknowledged and $\Delta_k$is elapsed time

Table 2.4: Delay-sensitive TCP variants reviewed in Section

TCP variant	**Section**	**Algorithm aims**	**Signal type**	**Metrics**
TCP Westwood [119]	2.4.4.1	high throughput in lossy environments	RTT	RTT_{min}
TCP Westwood+ [120]	2.4.4.1	high throughput in lossy environments, improved bandwidth estimator	RTT	RTT_{min}
TCPW CRB [13]	2.4.4.2	high throughput in lossy environments, better bandwidth estimator, friendliness	RTT	RTT_{min}
TCPW ABSE [121]	2.4.4.2	high throughput in lossy environments, better bandwidth estimator, friendliness	RTT	RTT_{min}
TCPW BR [122]	2.4.4.2	high throughput on heavy non-congestion related losses, friendliness	RTT	RTT_{min}
TCP-AP [123]	2.4.4.3	high throughput in multihop wireless networks	RTT	coefficient of RTT variation
RAPID [124]	2.4.4.4	Full bandwidth utilisation in fast network with dynamic bandwidth, low queuing delay and fairness	OWD	indirect queuing delay
TCP BBR[3]	2.4.4.5	low queuing delay, high throughput	RTT	RTT_{min}

since the receiving of the previous ACK. In the second stage, the final BWE is obtained by applying a low-pass discrete time filter:

$$BWE_k = \alpha.BWE_{k-1} + (1-\alpha)\frac{b_k + b_{k-1}}{2}$$

where BWE_k is the filtered bandwidth estimation, BWE_{k-1} is the previous estimation,α is a constant ($\alpha = 0.9$ for example) and b_k, b_{k-1} are the current and the previous bandwidth estimation samples respectively.

The evaluation results [119] show remarkable throughput improvement in the presence of random errors compared with TCP NewReno as well as very good inter- and intra-fairness. However, Grieco et al. [125] discovered that the estimator overestimates the bandwidth and causes unfriendliness in certain conditions. These conditions include ACK-compression effect that clusters ACK packet arrivals due to congestion and queue fluctuation in the reverse path [126] and employing AQM in the router [127].

A slightly modified version of this algorithm, called TCP Westwood+, was proposed to solve the ACK compression effect [120] by computing the bandwidth samples every RTT i.e. $b_k = d_k/RTT_k$ where d_k is the amount of data that has been acknowledged during the last RTT. The final BWE value is obtained by applying exponentially weighted moving average to the bandwidth samples b_k.

With respect to the RTT measurement, TCP Westwood creates a persistent queue in the bottleneck buffer if the RTT_{min} is larger than the actual RTT_{base}, such as in statistical multiplexing backoffs where many flows share the same bottleneck. Wrong RTT_{base} estimation leads to a congestion window larger than BDP after packet loss event causing unfairness between the competing flows.

2.4.4.2 TCP Westwood-Based Algorithms

In this section we briefly summarise a number of modifications proposed to remedy the vulnerabilities of TCP Westwood's bandwidth estimation technique.

TCP Westwood Combined Rate and Bandwidth estimation (TCPW CRB) [13] aims to improve the efficiency and friendliness of TCPW. It uses Rate Estimation (*RE*), which is long-term bandwidth estimation similar to what is used in Westwood+ but measured over a constant time interval (T) instead of RTT, in addition to Bandwidth Estimation (*BE*) of TCPW. The rationale of using two estimators is that RE prevent overestimation of the bandwidth when the network is suffering from congestion, but it underestimates the bandwidth when non-congestion related packet losses occur. TCPW CRB calculates the congestion window size as $BE \times RTT_{min}$ when packet loss is caused by random transmission error, and $RE \times RTT_{min}$ when the loss is caused by network congestion. TCPW CRB assumes that the loss is congestion related if $cwnd/(RE \times RTT_{min})$ upon packet loss is is smaller than a threshold (for example, 1.4); otherwise, the loss is assumed to be a random loss.

The authors of TCPW CRB claim that the dual bandwidth estimation method improves the trading-off of TCPW efficiency and friendless to NewReno [13]. However, their claim has been confirmed only using ns-2 simulation experiments. Additionally, they conclude that more study is needed to understand the impact of different conditions, such buffers sizes, the error rate and AQM, on the friendliness of the algorithm.

TCPW CRB authors later found that the time interval of RE should not be constant during connection lifetime to provide better friendless. Therefore, they proposed TCPW Adaptive Bandwidth Share Estimation (TCPW ABSE) [121]. TCPW ABSE adaptively changes the CRB sampling intervals depending on a congestion level estimation heuristic. The network congestion level is estimated based on the difference between the averaged sending rate sample ($VE = cwnd/RTT_{min}$, Vegas estimation) and throughput sample (RE, Westwood+ estimation). If the difference is large, the network is considered congested and large interval (one RTT) is used; otherwise a small interval is used. The authors state that this technique produces more precise bandwidth estimation in different congestion levels. Although ns-2 simulation shows that TCPW ABSE is able to achieve very good fairness with TCP NewReno as well as fast response to network conditions changes, real world experiments are required to validate the results.

Yang et al. [122] showed that the previous TCPW-based enhanced algorithms improve the throughput significantly in lossy environments but with random error rate $< 2\%$. Therefore, they propose TCPW with bulk repeat (TCPW BR) to improve the performance in extreme loss conditions. TCPW BR uses dual mechanisms to discriminate between congestion and non-congestion losses: the queue delay threshold method based on TCP DUAL (section 2.4.1.1) and comparing bandwidth estimation (BE) to the expected throughput (VE) similar to TCPW ABSE. If the packets losses seem to be caused by channel transfer error, TCPW BR resends all packets in flight, freezes RTO value and leaves *cwnd* unchanged; otherwise it reacts to the losses same as NewReno. Using ns-2 simulation, the author confirm the efficiency of the algorithm even at high error rate ($>5\%$) and the friendliness to TCP NewReno.

There are more TCPW variant techniques including: TCPW Bottleneck Bandwidth Estimation (BBE) [127] which aims to solve TCP Westood unfriendliness problem in different network conditions including highly varying bottleneck buffer sizes, AQM employment and heterogeneous flows RTT.

TCP Westwood with agile probing (TCPW-A) [128] aims to provide better performance in highly dynamic bandwidth networks as well as lossy environments. It achieves that by repeatedly resetting *ssthresh* based on *BE* like estimation and increasing *cwnd* exponentially (when *ssthresh* is lower than the estimation) in SS phase to quickly converge to full bandwidth utilisation, and does the same thing in the CA mode if a persistence non-congestion is detected; otherwise it uses NewReno *cwnd* increase function.

Similar to TCPW, all these modified algorithms can suffer from unfairness problems in different degrees if the RTT_{min} is an overestimation of RTT_{base}. Additionally, it is not clear how they react to the bottlenecks that utilise AQM to control the congestion in their buffers.

2.4.4.3 TCP-AP

TCP with Adaptive Pacing [123] is specialist hybrid window/rate-based CC that proposed to improve low performance of IEEE 802.11 multihop wireless networks. More specifically, this approach aims to reduces reduces link layer contention by adjusting packet sending rate based on contention estimation of the path. While window (*cwnd*) size controls the number of bytes in-flight, the pacing rate is used to provide smooth packet sending and preventing packets bursts.

This CC does not change any of standard slow-start, congestion avoidance and congestion recovery algorithms. It only controls the pacing rate of packet sending when *cwnd* allows transmitting new data.

TCP-AP mainly uses two metrics to calculate packet pacing rate. The first metric is the coefficient of RTT samples variation which is used to estimate contention degree of the path. The second metric is called out-of-interference delay which is the time between sending a packet from a node and receiving that packet at a second node existing outside the signal collision range of the first node.

The authors of this algorithm [123] claim that TCP-AP is able to achieve 10 times higher throughput than TCP NewReno in an emulated wireless network with 20 nodes. They also state that this algorithm provides good fairness with other flows and responses quickly to network condition changes. However, this algorithm relies on the RTT measurement which suffers from ACK compression and other packet delays caused by wireless link layer reliability (see Section 2.6). Moreover, this algorithm requires additional information from the operating systems, such as number of hops and data link parameters, which makes the kernel-space implementation complicated.

2.4.4.4 RAPID

RAPID congestion control [124] is rate-based approach that aims to realise fairness and high throughput in fast networks with dynamic bandwidth while maintaining inter- and intra-protocol fairness and low queuing delay. Although this algorithm does not utilise the delay signal explicitly in its operation, the theory behind RAPID is based on the queue delay growing up.

The authors of this algorithm claim that conventional TCP CC algorithms are not able to achieve full bandwidth utilisation in fast networks with variable bandwidth due to slow bandwidth probing technique of those algorithms. More specifically, conventional TCP bandwidth probing requires one RTT time interval to realise the result of a new probing cycle and the new rate probing is not much larger than the previous rate (*cwnd* typically increases by one MSS every RTT interval). Therefore, many RTT intervals are required before converging to full bandwidth utilisation.

RAPID is able to probe multiple rates by sending groups of N packets with N-1 different rates in each group i.e. increases the gap between the sending packets of a group. At the receiver host, the inter-packet arrival times (the gaps) of each group is monitored. If the gaps trend increases (i.e. $gap_i > gap_{i-1}$), this means the bottleneck queue started to build-up since every packet sent at a faster speed then its preceding will cause additional delay in packet delivery (larger gap). Then the estimated bandwidth will be the rate before seeing positive gaps trend (i.e. $rate_{i-1}$). If $gap_i \leq gap_{i-1}$ for all groups, the sender generates and transmits more groups with higher sending rate until the receiver obverses positive gap trend (i.e. $gap_i > gap_{i-1}$). The receiver then explicitly sends the value of $rate_{i-1}$ to the sender host to use it as sending rate.

Using ns-2 [91] implementation of RAPID, Konda et al. [124] claim that this algorithm can converge to full bandwidth utilisation of gigabit networks in 1-4 RTTs while keeping the queue short. They also state that RAPID provides good fairness and a small impact on conventional TCP flows. However, this algorithm requires receiver-side modification which makes the global deployment very hard. Additional, RAPID requires high timer resolution to send packets at an accurate rate which is hard to achieve for many hosts and produces additional overhead. Moreover, similar to many delay-based algorithms, RAPID can suffer and produce unpredictable behaviour in wireless networks due to collision avoidance and data link frame recovery of these networks which change packets transmission pattern.

2.4.4.5 TCP BBR

Bottleneck Bandwidth and Round-trip propagation time [3] is a recent congestion control algorithm that aims to solve the bufferbloat problem and improve TCP throughput. Rather than seek a feedback signal to detect congestion, BBR controls congestion by pacing the sending rate according to the currently estimated bottleneck bandwidth BW and RTT_{min}.

A delivery rate sample is calculated as the ratio between delivered data and time elapsed for delivering that data. BBR uses a windowed maximum filter over a 6-10 RTT period for delivery rate samples to obtain a BW estimation.

As a secondary control mechanism, BBR utilises a TCP-like *cwnd* mechanism to limit the maximum amount of inflight data. This window is set to a few multiples of BDP (with BDP calculated as $BW \times RTT_{min}$) to remedy common receiver and network issues such as delayed and aggregate ACKs.

A windowed minimum filter for RTT_i is used to estimate RTT_{min}. If RTT_{min} does not change and no RTT sample matches RTT_{min} for 10 second period, the number of packets inflight is reduced to 4 for a short period to drain the buffer and probe for a new RTT_{min}. This allows BBR to refresh its RTT_{min} estimate as well as permitting flows to converge to a fair share of the bandwidth.

BBR maintains one BDP worth of packets inflight most of the time to guarantee full link utilisation with low queuing delay. Periodically, BBR increases the sending rate and inflight size to 125% for an RTT interval to probe available bandwidth and changes the paced rate accordingly. If BW remains unchanged, the sending rate and inflight size is reduced to 75% to drain the built-up queue. BBR then returns to normal operation of using 100% of the estimated BW and inflight size.

TCP BBR has been implemented in Linux and deployed in Google services since 2015, achieving 2 to 25 times greater throughput than CUBIC [3].

BBR's authors evaluate the algorithm in large variety of scenarios and network environments. They claim that BBR improves throughput and reduces latency compared with TCP CUBIC, while being able to achieve acceptable fairness. However, the authors agree that BBR has problems in specific situations and requires more research to remedy these issues.

For example, BBR flows obtain a lower share of bandwidth when competing with loss-based flows through a bottleneck with a large (several BDPs) buffer. Additionally, BBR flows can suffer from unexpected packet loss when token-bucket traffic policers are used. More specifically, when BBR sends packets faster than the bucket fill rate, all packets are discarded by the policer.

Moreover, Hock et al. [129] conduct experimental evaluation of TCP BBR in an emulated environment and conclude that BBR is able to achieve its goals. However, they also find that BBR produces a large number of packets losses, unfairness and long queuing delay in some scenarios, specially in shallow bottleneck buffer. There is also a big concern in transport research communities regarding BBR packets losses ingratiation as well as ECN support ingratiation.

Although the algorithm attempts to remedy the wrong estimation of RTT_{base} using RTT_{min} measurement reset mechanisms, it is not clear how well this method works in environments with long standing queues causing by other CC algorithms/flows. Therefore, more evaluation is need to understand how the wrong estimation can affect its operation.

2.5 TCP slow-start variations

The purpose of slow start phase is to quickly discover the capacity of a path with no prior information of the path characteristics or support from network infrastructure. Ideally, slow start phase finishes with *cwnd* equal to path BDP to achieve both high throughput and low latency. Standard TCP considers link capacity has been reached when packet loss is detected. During this phase, *cwnd* doubles every RTT. Standard TCP SS finishes with *ssthresh* and $cwnd = (B_{rate} * RTT_{base} + bs)/2$. If the buffer size ($bs$) is too large, slow start will induce high queuing delays and heavy packet loss from the same window since *cwnd* will be much larger than path BDP.

Many slow-start alternatives have been proposed to remedy these issues. Hoe [130] proposes to set *ssthresh* to the estimated path BDP instead of using an arbitrary large value. This allows SS to finish before packet losses occur. BDP is estimated using packet-pair based bandwidth estimator [131] and RTT. It has been shown that the cross traffic can lead to bandwidth overestimation [132]. Additionally, this method requires sending back-to-back packets to probe the path.

TCP Vegas [133] uses the difference between the expected and achieved rate to infer congestion and exit slow start (see Section 2.4.1.2). This approach suffers from an early slow start termination due to bottleneck queue build-up from the bursty nature of TCP transmission [134].

Quick-start [135] is another slow start algorithm that can converge to full bandwidth utilisation very quickly. However, this approach is hard to deploy since it needs supports from middle boxes, and requires both sending and receiving host modification.

Cheng et al. [134] propose a slow start approach called Adaptive start. This approach aims to reduces the number of packet losses during slow start by slowing down *cwnd* growth when it nears to the path BDP.

Paced start [136] controls the spacing between packets of a packets train [132] to probe available path capacity. This approach requires precise timing in packet transmission and receiving which is costly and hard to implement in many systems.

CapStart [137] aims to reduce packet losses during slow start in fast networks. It infers the existence of a bottleneck in the path and moves to limited Slow Start [138] when a queue builds up. This method uses packet-pair [131] to infer queues forming.

Hystart slow start [72] aims to find a safe exit point from slow start with *cwnd* that allows flows to achieve high throughput and minimal packet losses. This exit point leaves *cwnd* between $(B_{rate} \times OWD_{min})/2$ and $B_{rate} \times OWD_{min} + bs$. Hystart exits SS whenever one of two independent algorithms finds the exit point. At the end of slow start *ssthresh* is set to *cwnd*.

The first algorithm relies on the ACK train principle to infer congestion. Instead of directly measuring the space between consecutive ACK packets, the time period between receiving the first ACK and the current ACK in a window is used. If that time period is greater than or equal to $RTT_{min}/2$, SS finishes. Hystart uses a passive ACK train by relying on TCP transmission burstiness during slow start phase.

The second algorithm monitors the $RTT_{min,n}$ gradient to see if congestion happens. $RTT_{min,n}$ is measured for the first eight RTT samples at the start of a window (the train). If $RTT_{min,n} > RTT_{min,n-1} + \eta$, Hystart exits slow start. η is the maximum acceptable amount of time the gradient can increase without exiting slow start, and is calculated based on $RTT_{min,n-1}$. The second algorithm protects slow start overestimating the bandwidth due to the challenges of estimating RTT_{min} as used by the first algorithm.

2.6 Challenges facing the delay signal

2.6.1 Inaccurate delay measurements

As we mentioned in Section 2.3.1.2, many problems facing the use of delay signal causing undesirable side effects for congestion control techniques that benefit form this signal. Usually, inaccurate delay measurement happens due to queue fluctuation, burst packet sending, delayed ACK, hardware transmission offloading, link layer buffering (especially in wireless networks) as well as inaccurate sampling. Additionally, interference between competing flows causes the delay-based techniques to wrongly estimate the congestion level. This includes wrong propagation delay estimation (which is used by many algorithm) due to route change and sequence of flow starting time (latecomer advantage problem).

Reverse path congestion is another problem that can distort the delay signal when the RTT metric is used. When congestion happens in the reverse path (ACK path), the RTT metric will not reflect the real congestion state of the network (forward path) since the ACK packets are small and do not contribute to the congestion. Therefore, some algorithms utilise the one-way delay signal instead. However, unlike using RTT, the one-way delay metric needs receiver side modification or enabling TCP timestamp option which makes using that signal not a universal solution for standard TCP. Furthermore, the one-way delay signal cannot be used as an absolute delay time without clock synchronisation between source and destination. Moreover, finding universal optimum threshold values for the threshold based algorithms is very hard or unachievable due to large varieties of network environments and conditions. More serious problem for the delay-based algorithm is the unfairness when competing against the greedy loss-based algorithms including the standard TCP congestion control.

2.6.2 Data link layer reliability and channel access method

Reliability mechanisms and channel access methods in some networks, such as wireless networks, have a significant negative impact on the delay signal usability. The data link layer of modern wireless networks, for example, attempts to supply the network layer with error-free packets by providing reliable packet recovery mechanisms. If packets arrive corrupted or do not arrive at the destination, the sending station will retransmit the packets until they arrive correctly. This behaviour makes the network layer to wait for an unknown interval of time before receiving the packets. The duration of this interval depends on the degree of the noise and interference on the channel.

Moreover, in shared transmission medium, Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) manages channel access and attempts to minimise collisions of transmitted packets. This involves sensing the carrier to see whether it is busy andwaiting for a random intervals until the channel becomes available.

CSMA/CA can also use Request to Send/Clear to Send (RTS/CTS) to reduce the problem of collisions due to hidden nodes. With RTS/CTS, the sender first sends an RTS frame to be received by the Access point (AP). The AP sends back a CTS frame to the sender to indicate that it may begin trasmission. If the channel is busy or the AP has data to send (typically AP has higher priority) the sender should further wait before reattempting the RTS/CTS protocol. If RTS/CTS frames are dropped, the node should wait for a random time period before retransmitting the RTS frame.

This introduced waiting time becomes part of the end-to-end delay measurement used by delay-based CC. This latency is not related to network congestion and it corrupts the delay feedback. Managing medium access becomes more challenging in duplex communication channels. In a point-to-point simplex communication channel, however, the issue becomes less problematic since the channel will be dedicated for one transmitter only.

Both data link reliability and CSMA/CA significantly weaken the correlation between congestion and the measured delay signal. Any congestion related decisions (e.g. *cwnd* backing-off) made by delay-based CC depending on such signal significantly reduces the throughput and produces unpredictable behaviour. In addition, these mechanisms can affect packets on both the forward path and the reverse path causing packets to be sent in bursts. Sending packets in bursts causes queue fluctuation and ACK compression which have a significant negative impact on RTT measurements.

Another problem that causes inaccurate RTT measurements is packet aggregation of some data link layers of wireless networks (e.g. IEEE802.11n). Packet aggregation aims to reduce the overhead of sending small packets by combining multiple packets and transmitting them as one frame. Karlsson et al. [139] have shown that packet aggregation can make several ACK packets to be sent back-to-back causing ACK compression effect.

It is worth noting that even rate-estimation CC techniques (e.g. TCP BBR in Section 2.4.4.5) can suffer (but in a smaller degree) from these data link mechanisms since the ACKs can arrive at the sender in burst faster than how the receiver sent. Therefore, the bandwidth estimator will overestimate the available bandwidth causing the sender hosts to transmits packets faster than the actual available bandwidth.

Therefore, it is very difficult and unreliable to use the delay signal to control congestion in noisy and shared medium networks with strong data link reliability. However, using some delay based estimations (e.g RTT_{base}) to improve protocol performance in such networks is considered useful and usable.

2.6.3 Emerging communication networks

With wide deployment of the TCP protocol in many devices and operating systems, many emerging network applications utilise this protocol to provide reliable data transfer. Different types of networks have been developed to fulfil the needs of modern network applications. These include vehicular networks (VANETs) [140], mobile ad hoc networks (MANETs) [141] and delay tolerant networks (DTNs) [142]. These types of networks typically have different characteristics to traditional wired networks.

These characteristics can include one or more of changing network paths (due to node mobility), changing numbers of connected nodes, shared multi-hop channels, varying intrinsic delays, unpredicatable medium and data link reliability, high bit error rates, lack of continuous network connectivity and low powered (electricity and/or computation) resources. These challenges can propagate up the stack where they can inpact on the performance of the transport protocols used within these applications.

For example, varying latency due to changing paths or packet loss due to unstable network topologies can result in the TCP RTO being falsely triggered. Similarly, non-congestion related packet losses can occur more frequently due to higher bit error rates in such networks [141].

Further impact may be seen where variable data transmission and unpredictable latencies can cause delay-based congestion control to misinterpret the delay signal. It is difficult to determine if any increase in measured delay is the result of network congestion or other non-congestion related effects.

Another challenging application space is the Internet of Things (IoT). Many end devices (things) utilise a TCP transport to communicate with a number of pre-existing Internet services and devices [120]. Typically, a large number of these devices would connect via wireless networks (infrastructure, Ad hoc or mesh networks).

IoT devices typically have limited processing power and/or energy resources, limiting their capabilities. CC for these devices should be lightweight and efficient to provide high performance with limited resources.

Traditional TCP algorithms may perform poorly under these circumstances. Packet loss results in wasted communication capacity, and can lead to increased processing overhead when recovering from loss. Alternatively, delay-based CC may result in improved overall performance at the increased cost of accurately predicting network state.

Each of these emerging applications have unique network and application requirements, meaning no one CC algorithm is best for all cases. This implies that selection of suitable CC mechanisms should be performed on a case-by-case basis.

2.6.4 Effects of using AQM on delay signal

While AQM and delay-based CC share the same goal of keeping queuing delay low, they achieve that goal from different places. AQM's place is the middleboxes (such as switches, routers and firewalls) and they need a reaction from the end host to control the congestion. On the other hand, delay-based CC achieves that goal using end-to-end approach without help from network appliances.

In controlled environments (data centre networks, for example), it is easy to deploy AQM in the bottlenecks since network equipment usually belongs to the one organisation. Additionally, delay-based CC algorithms can work very well in such environments since flows can be homogeneous (one CC is used) and the noise in delay signal is low. Therefore, either of the two approaches works well. On the Internet, however, there is no one organisation having control over the equipment along the path between the sender and receiver hosts. If the place of a bottleneck is known and accessible by an individual, queuing delay can be controlled by deploying AQM in that bottleneck. An example of such bottlenecks is ADSL home gateways in the upstream direction.

On the other hand,bottleneck location is unknown, inaccessible or AQM is not implemented for specific hardware, there is no way for the end-users or service providers (such as gaming servers) to use AQM. Furthermore, it is very hard to know whether AQM is deployed along the path or not. Therefore, in such common case delay-based CC (low-latency transport) is the only choice for the end-users or service providers to achieve low latency communication.

In this context, it is not uncommon to see delay-based CC flows pass across AQM-enabled bottlenecks. This raises questions about the effects of deploying AQM on delay-based flows and the coexistence of AQM and delay-based CC.

AQMs effectively create short queues (in size or time) to prevent packets from unnecessarily residing in the buffer for too long. On the other hand, delay-based CC relies on queuing delay measurement to infer congestion. Short queues reduce the signal variations that latency-sensitive CC algorithms rely on.

Most delay-based and hybrid congestion control algorithms compare the delay measurements with thresholds to infer the congestion level in a network. If these thresholds allow the bottleneck queue to include packets longer than AQM allows, the algorithm will fall back to loss signal reactive mode. This behaviour damages a substantial goal of the delay-based algorithms in which they try to reduce the number of packets losses by early reacting to the congestion.

Many delay-based CC algorithms (TCP DUAL and TCP Vegas for example) aim to reduce TCP saw-tooth *cwnd* behaviour to achieve both stability and high throughput even in shallow bottleneck buffers. However, falling back to loss mode destroys these aims and can create fluctuation in throughput because CC may excessively back off *cwnd* below path's BDP especially in high-speed long-distance links.

Additionally, working in loss-mode all the time can alter the design goal of some delay-based CC. For example, it has been shown that AQM bottlenecks raise the priority of LPCC making scavenger-class transport to compete equally with conventional TCP for bottleneck bandwidth [143]. Without taking AQM deployment into consideration, scavenger-class transports can impact negatively on conventional TCP traffic. A study on the effects of modern AQM bottlenecks on *libutp* [96], LEDBAT-based widely deployed transport, shows that LEDBAT flows become more aggressive than conventional TCP flows when AQM is used [98]. The issue is that *libutp* increases *cwnd* quickly (faster than standard TCP) when the queuing delay is small and *cwnd* growth becomes slower as the queuing delay becomes closer to LEDBAT's target delay (see Section 2.4.1.8 for details about LEDBAT CC). This strategy works well with FIFO bottleneck to converge quickly to full utilisation but not with AQM enabled bottlenecks since queuing delay will hardly reach LEDBAT's target delay.

Moreover, if the AQM allows very low queuing delay, the noise reduction techniques used by delay-based algorithms (such as moving average) can destroy the signal completely. Figure 2.14 illustrates how the use of AQM in a bottleneck can destroy the delay signal through reducing the queuing delay variation.

Despite the challenges of using a delay signal in such environments, AQM can help some algorithms to overcome the latecomer advantage problem by forcing all flows to back off. This allows resistant queues to drain giving an opportunity for flows to obtain correct propagation delay estimation. Additionally, if a bottleneck employs AQM with ECN marking support and the end hosts also support ECN, delay-based CC can work reasonably well in keeping queuing delay low without packet losses since the congestion signal is sent directly from the congested box. However, reacting to ECN signal by the delay-based CC can create fluctuations in throughput similar to packet drop. In this aspect, TCP Alternative Backoff with ECN (ABE) [66] plays a very important role in keeping unacknowledged data close to the path's BDP and reducing *cwnd* fluctuation. This proposal suggests using two different *cwnd* back-off factors; one for ECN (β_{ECN}) and another for packet drop (β_{loss}) and the proposal recommends using 0.8 for β_{ECN}.

Recently, there has been growing interest in deploying modern AQM on the Internet due to increased sensitivity of many applications to latency caused by bufferbloat. This raises questions about how AQM bottlenecks will affect CC algorithms that rely entirely or in-part on the delay signal. Therefore, research needs to be conducted to understand the behaviours of delay-signal CC with AQM in a wide range of environments. This requires more theoretical and practical studies for the delay signal and the distortion level of AQM on that signal and the CC algorithms that employ such a signal.

It is apparent that the presence of an AQM managed bottleneck may damage the delay signal with respect to delay-based CC algorithm, which could result in some implementations reverting to loss-based mode leading to unintended consequences when competing with other flows. However, the short queues enforced by the AQM will limit this impact on the performance of competing flows. Alternatively, the use of delay-based CC algorithms may help in circumstances where AQMs are not deployed.

2.7 CDG: A promising hybrid CC algorithm

As previously mentioned, large FIFO buffers coupled with conventional TCP's capacity probing strategy are a challenging problem for latency-tolerant applications (such as multimedia conferencing and multi-player games). This problem is illustrated in Figure 2.15a. We can see in this figure, a loss-based TCP flow (TCP CUBIC in this example) increases queuing delay in a bottleneck resulting in the low-rate online game traffic being significantly delayed. We can also see that the RTT experienced fluctuates due to back-off upon packet loss caused by the TCP CUBIC bandwidth probing strategy. This behaviour results in a slower game response, severely impacting on the online gaming user experience.

CAIA-Delay Gradient (CDG) [12] provides low latency transport by responding early to network congestion before long queues form at the bottlenecks. This allows latency sensitive applications to perform well in the presence of the cross traffic. Figure 2.15b illustrates how

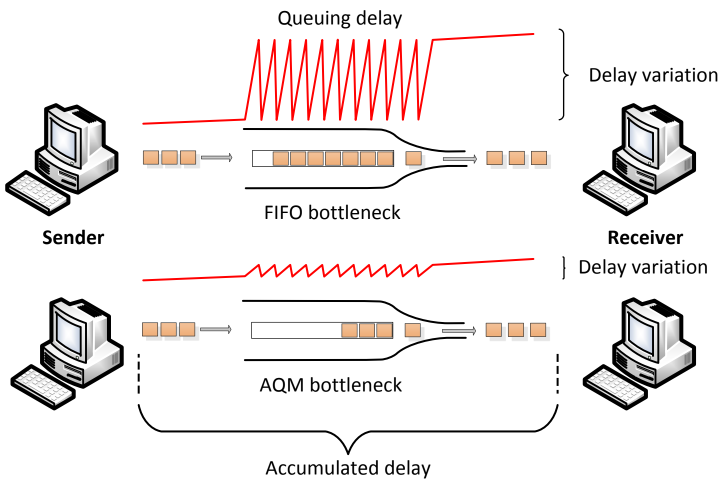


Figure 2.14: The effect of using AQM on the delay signal

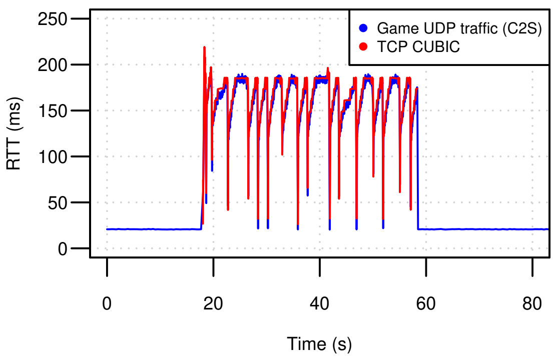
bulk data transfer using CDG does not induce high queuing delay, allowing game traffic to traverse the bottleneck without significant delay. This leads to better user experience for end users. We can see in this figure that the measured RTT remains close to the path RTT during the coexistence period. Further, no packet loss occurs in this scenario as the buffer is never full.

Unlike many delay-based and hybrid CC algorithms, CDG uses the delay gradient signal and a probabilistic back-off function to detect network congestion. This allows CDG to avoid using the problematic queuing delay estimation and thresholds [12]. Additionally, CDG ignores delay-based back-off and uses only the loss signal when a competing loss-based flow is inferred. This allows CDG flows to perform better than pure delay-based CC when coexisting with loss-based CC.

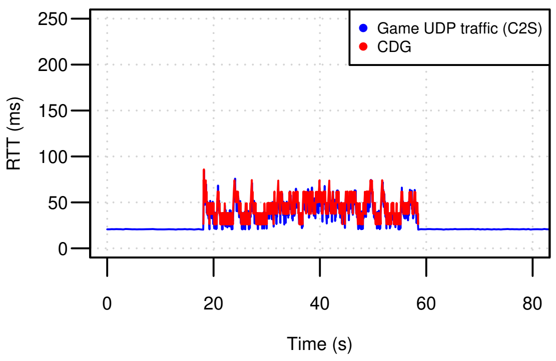
Despite the benefits of using CDG [7, 107, 144], some issues hinder wider deployment of this algorithm. The advantages of using CDG motivate us to study and address these issues to create an improved congestion control algorithm for the Internet. In this section, we describe the CDG algorithm is some more details to understand its functionality and mechanisms.

2.7.1 Delay gradient signal

To avoid issues associated with RTT_{base} estimation such as standing queue, route change and latecomer advantage, CDG uses a *delay gradient* signal, which represents queuing trend based



(a) A TCP CUBIC flow joins at t=30s, increasing the measured RTT at the end hosts



(b) A CDG flow joins at t=30s, inducing lower latency when compared with TCP CUBIC

Figure 2.15: TCP CUBIC flow causes low rate Client-to-Server game flow to experience a high RTT range while CDG has much less impact on game flow. 2Mbps bottleneck bandwidth, 20ms $RTT_{base}$and 40pkts buffer size

on delay measurements to infer congestion. The general concept is that when the bottleneck is not congested, no queue is created in the bottleneck, and no additional delays will be observed at the end host. When *cwnd* becomes larger than path BDP due to capacity probing strategy, the average sending rate becomes larger than the bottleneck bandwidth. This creates a queue in the bottleneck which results in additional delay. Therefore, end hosts will observe an increase in latency. A positive difference between old and current delay samples observed at end host indicates a queue is beginning to form.

The gradient signal does not only provide the congestion status, but also how aggressively the congestion is forming. For steady queue growth, the gradient increases proportionally to queue growth speed. Thus, a large positive gradient indicates that bottleneck inbound traffic is much higher than outbound traffic.

CDG uses RTT measurements at the sender to estimate the latency in packet delivery. Instead of using instantaneous RTT estimation, CDG uses $RTT_{min,n}$ and $RTT_{max,n}$ measurements by applying minimum and maximum filters for RTT samples obtained over a *measurement cycle* (n). A measurement cycle is one RTT interval. $RTT_{min,n}$ and $RTT_{max,n}$ are measured per Equations 2.22 and 2.23:

$$RTT_{min,n} = min(RTT_t)\forall t \in [T - RTT, T] \tag{2.22}$$

$$RTT_{max,n} = max(RTT_t)\forall t \in [T - RTT, T] \tag{2.23}$$

Using minimum and maximum filters allows CDG to minimise delay signal noise. The two measurements are used to calculate two delay gradient measurements ($g_{min,n}$ and $g_{max,n}$).

These gradient signals are then used to detect congestion.$g_{min,n}$ and $g_{max,n}$ are calculated once every measurement cycle as per Equations 2.24 and 2.25 respectively.

$$g_{min,n} = RTT_{min,n} - RTT_{min,n-1} \tag{2.24}$$

$$g_{max,n} = RTT_{max,n} - RTT_{max,n-1} \tag{2.25}$$

$g_{min,n}$ and $g_{max,n}$ give similar congestion information when the congestion is persistent and continuously increases or decreases during the measurement period. However, there is a substantial difference between $g_{min,n}$ and $g_{max,n}$ regarding congestion detection. While $g_{max,n}$ is the difference between the peaks of measured delays (spike peaks) in the current and previous RTT intervals, $g_{min,n}$ is the difference between lowest measured delays in the current and previous RTT intervals. This means that $g_{max,n}$ gives an indication of non-persistent congestion state occurring during the measurement cycles. In other words, it can detect congestion that occurs and dissipates during an RTT interval. On the other hand, $g_{min,n}$ gives an indication of the persistent congestion state that occurred during the previous RTT. That is, it tracks changes in the standing queue. Figure 2.16 illustrates the difference between the two gradient signals.

The delay gradient measurements are further smoothed using a moving average window to obtain $\bar{g}_{min,n}$ and $\bar{g}_{min,n}$. Smoothing the gradient signal allows CDG to further reduce delay signal noise. $\bar{g}_{min,n}$ and $\bar{g}_{min,n}$ are calculated as per Equation 2.26:

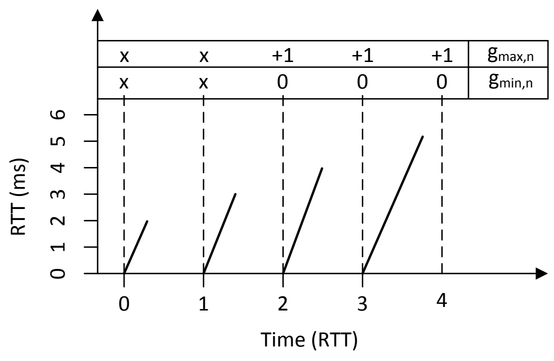
$$\bar{g}_n = \bar{g}_{n-1} + \frac{g_n - g_{n-a}}{a} \tag{2.26}$$

where $\bar{g}_n$ is either $\bar{g}_{min,n}$ or $\bar{g}_{min,n}$, n is the nth measurement cycle and $a \geq 1$ being the smoothing window size (8 by default). Without a smoothing window, there is a high probability of calculating a positive gradient due to RTT signal noise. This leads to false positive congestion detection that affects protocol performance. CDG infers congestion when $\bar{g}_{min}$> 0 or $\bar{g}_{max}$ *>0.*

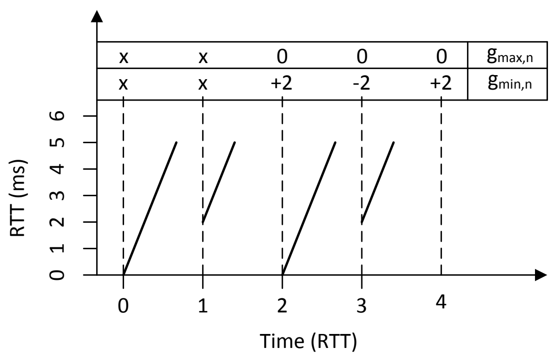
2.7.2 Probabilistic back-off function

CDG uses a probabilistic back-off function based on the gradient magnitude to decide when to back off. This allows CDG to work without requiring thresholds and provides fairness between competing flows.

For steady queue growth, the calculated gradient increases proportionally to RTT since the gradient is calculated once every RTT. Therefore, flows traversing long RTT paths will see higher gradients than flows traversing shorter paths sharing the same bottleneck. This results



(a) $g_{max,n}$ indicates peak queuing delay gradients



(b) $g_{min,n}$ indicates standing queuing delay gradients

Figure 2.16: $g_{min,n}$ and $g_{max,n}$ gives different information about queue state. RTT_{base} is zero for illustration purpose

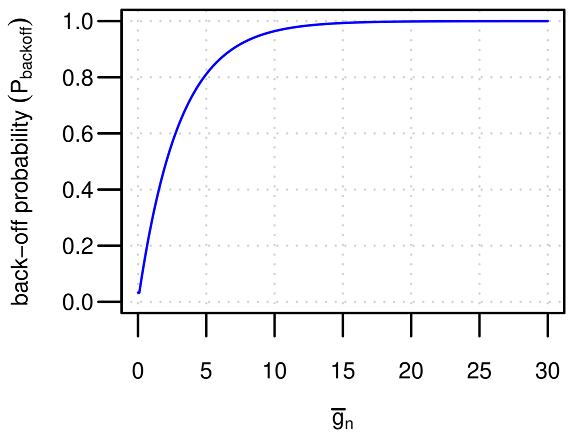


Figure 2.17: CDG back-off probability as function of $\bar{g_n}$. Default $G = 3$.

in unfairness between the competing flows. To remedy this issue, CDG uses an exponential factor in the probabilistic back-off function to achieve fairness between flows with different base RTTs. The exponential function results in flows with a smaller RTT experiencing similar back-off probability on average to flows with a larger RTT. The CDG probabilistic back-off function ($p_{backoff}$) is shown in Equation 2.27.

$$p_{backoff} = 1 - e^{-\left(\frac{\bar{g_n}}{G}\right)} \tag{2.27}$$

Figure 2.17 shows the back-off probability as function of the delay-gradient.

Table 2.5: CDG queue state inferring based on $\bar{g}_{min}$ and $\bar{g}_{max}$

$\bar{g}_{min,n}$	$\bar{g}_{max,n}$	Q_{state}
+	0/+	full
0/+	-	empty
+	+	rising
-	-	falling
otherwise		keep old state / unknown

2.7.3 Non-congestion related losses tolerance

CDG also aims to provide high goodput in lossy environments such as some wireless networks. It uses heuristic inference based on the gradient signal to guess bottleneck queue states to then distinguish between congestion and non-congestion related packets losses. By identifying the type of loss, CDG can achieve better throughput than loss-based CC by not decreasing *cwnd* when detecting non-congestion related losses.

CDG defines five queue states Q_{state} $\in$\{full, empty, rising, falling, unknown\}. As congestion loss occurs due to bottleneck buffer overflow, CDG considers packet losses to be congestion related only when Q_{state} = full at the time of loss detection. If $Q_{state} \neq$ full and packet loss is detected, CDG considers this to be a random loss caused by channel noise.

The concept behind of CDG's queue full inference heuristic is that when the bottleneck buffer is full, $RTT_{max,n}$ stops increasing before $RTT_{min,n}$. This happens because there is no more space in the buffer leads to additional delay and $RTT_{max,n}$ reaches its maximum value during the measurement period. However, $RTT_{min,n}$ continues growing for a short period before loss occurs. This happens because the bottleneck scheduler periodically serves packets giving space for newly arriving packets. As the sender keeps sending packet faster than the bottleneck capacity, the buffer space gradually becomes smaller resulting in some packets experiencing increased delay.

When Q_{state} = empty, $RTT_{min,n}$ stops decreasing before $RTT_{max,n}$. This happens because some packets see no queuing delay resulting $RTT_{min,n}$ equalling RTT_{base}, while other packets will see decreasing queuing delay due to queue draining, causing RTT_{max} to decay during a measurement cycle.

Instead of monitoring RTT_{min} and RTT_{max} progression, CDG uses $\bar{g}_{max}$ and $\bar{g}_{min}$ signs to infer the queue state as shown in Table 2.5.

Incorrect queue state inference during packet loss leads to different unwanted behaviours that reduces CDG performance. In case of CDG wrongly inferring a full queue (false positive), tolerance to non-congestion related losses will be less effective. This reduces the throughput of flows over high bit error rate (BER) links due to backing off *cwnd* when there is no congestion.

If it wrongly infers queue is not full upon packet loss, CDG will further congest the network, causing further packet loss.

Further, the queue state inference heuristic assumes that the bottleneck uses a FIFO buffer with a drop-tail mechanism. In AQM environments packets are instead dropped by AQM due to congestion well before buffer overflow occurs [45]. CDG will consider these losses as non-congestion related losses since the queue state is typically determined to be "rising" when AQM drops packets.

In this thesis, we do not explore/evaluate the accuracy of the queue state inference heuristic as random losses have become uncommon in modern networks, including wireless networks. The network and higher layers rarely see any random packet loss since loss recovery happens in the data-link layer (see Section 2.6).

2.7.4 Coexistence with loss-based flows

As mentioned before in Section 2.3.1.2, delay-based CC has problems when it comes to co-existence with loss-based flows. Early back-off of delay-based flows results in the flows not attaining fair capacity sharing with loss-based CC, and CDG is not an exception. However, CDG borrows the *shadow window mechanism* from CAIA-Hamilton Delay (CHD) CC [106], along with the *ineffectual back-off* mechanism to provide better coexistence with loss-based flows.

2.7.4.1 Ineffectual back-off and loss mode:

CDG operates in two regimes to provide better coexistence with loss-based flows. By default CDG operates in delay-mode where congestion is detected using the delay-gradient signal. When CDG infers competing loss-based flows, it switches to loss-mode for a maximum of five (default) detected delay-based congestion events. It then switches back to delay-mode. Additionally, CDG moves back to delay-mode if a negative gradient ($\bar{g}_{max}$ or $\bar{g}_{min}$) is observed.

CDG detects competing loss-based flows using *ineffectual back-off detection*. If CDG experiences a number (default 5) of consecutive delay-based back off events without observing a negative gradient ($\bar{g}_{max}$ or $\bar{g}_{min}$), CDG assumes it is competing with loss based-flows. The notion behind this assumption is that backing off *cwnd* multiple times should decrease bottleneck queue length. This would reduce queuing delay, resulting in negative gradients. In competition with a loss-based flow, even if CDG backs off multiple times, queuing delay will continue to increase as the loss-based flow keeps increasing *cwnd* until packet loss occurs.

2.7.4.2 The shadow window mechanism

CDG adopts the shadow window mechanism from CHD CC [106] to improve coexistence with loss-based flows. The shadow window s mimics the loss-based window growth/decay behaviour of NewReno while CDG is operating in the delay mode. The shadow window is only used when congestion-related losses are detected. CDG *cwnd* is typically small when operating in delay mode due to high queuing delay and multiple back-offs caused by competing flows. Therefore, using a larger *cwnd* that mimics loss-based window growth when detecting congestion related packet losses can enhance CDG's coexistence abilities.

The shadow window does not decay on delay-gradient back-off events, but it is synchronised with *cwnd* in case $cwnd > s$. When an empty queue is inferred, the shadow window is reset. The shadow window is updated according to Equation 2.28:

$$s_{i+1} = \begin{cases} cwnd_i & \text{connection initialisation} \\ max(cwnd_i, s_i) & R < P_{backoff} \wedge \bar{g}_n > 0 \\ s_i \times 0.5 & \text{packet loss } \wedge Q_{state} = Full \\ 0 & Q_{state} = empty \\ s_i + MSS & s_i > 0 \end{cases} \tag{2.28}$$

where s_i is the current shadow window and R is a random uniformly generated number. CDG sets *cwnd* to the maximum of the $\frac{s}{2}$ and $\frac{cwnd}{2}$ as soon as packet loss is detected and the queue state is not full.

2.7.5 Congestion window growth

CDG uses an additive-increase/multiplicative-decrease (AIMD) mechanism to probe network capacity. It increases *cwnd* by one MSS per RTT when no congestion occurs.

When congestion is detected using the delay gradient signal (positive gradient), equation 2.27 is used to probabilistically determine if CDG should back off *cwnd*. When probabilistic back-off occurs, CDG decreases *cwnd* by a back-off factor β_{delay} (default 0.7). The higher factor for β_{delay} allows CDG to maintain high link utilisation since it backs off as soon as a short queue is created.

If packet loss is detected and Q_{state} = full, CDG halves *cwnd* similar to TCP Reno; otherwise it keeps *cwnd* unchanged. CDG uses the congestion window growth Equation 2.29 at every RTT or packet loss where $\beta_{loss} = 0.5$ by default.

$$cwnd_{i+1} = \begin{cases} cwnd_i \beta_{delay} & R < P_{backoff} \wedge \bar{g}_n > 0 \\ max(cwnd_i \beta_{loss}, s_i) & \text{packet loss } \wedge Q_{state} = Full \\ cwnd_i & \text{packet loss } \wedge Q_{state} \neq Full \\ cwnd_i + MSS & \text{otherwise} \end{cases} \tag{2.29}$$

Since the effect of *cwnd* back-off is not observable to the sender host until the following RTT, CDG backs off *cwnd* not more than once every two RTTs.

2.7.6 CDG slow start

CDG modifies the standard loss-based slow start to prevent large spikes in queuing delay and to eliminate packet loss during this phase. CDG slow-start relies on the delay gradient measurement to find a better exit point before packet loss occurs. During this phase, CDG increase *cwnd* similar to NewReno i.e. it doubles *cwnd* every RTT.

CDG uses the same probabilistic back-off function (Equation 2.27) to exit from the slow start phase and enter congestion avoidance phase. It also exits from the slow start phase if a loss occurs before triggering a delay gradient based back-off. Similar to the CA phase, if congestion is detected using delay-gradient signal, *ssthresh* is set to $cwnd.\beta_{delay}$. On the other hand, if packet loss is detected, *ssthresh* is set to $cwnd.\beta_{loss}$.

2.7.7 CDG FreeBSD implementation consideration

CDG was originally implemented for the FreeBSD operating system [145, 146] by David Hayes, the primary author of the CDG paper [12]. This implementation is considered an official CDG implementation. In 2015, Jonassen [147] implemented CDG for Linux based on the original CDG paper and using the FreeBSD implementation as a reference. This implementation includes some modifications to the original CDG including the use of Proportional Rate Reduction [148], HyStart slow start [72] and some changes to queue state inference.

The CDG FreeBSD implementation utilises the Enhanced Round Trip Time (`h_ertt`) module [149, 150] to obtain accurate RTT estimations. This module aims to resolve three TCP mechanisms that lead to inaccurate RTT measurements, 1) Delayed Acknowledgements [151]; 2) Selective Acknowledgements (SACK) [38]; 3) and TCP Segmentation Offload (TSO)[4].

[4]TSO is a hardware accelerating technique that is used to split a large number of bytes into small TCP segments without needing intervention from CPU.

`h_errt` module utilises TCP timestamp option [152] to improve the accuracy of RTT measurements when it is enabled.

`h_errt` infers that delayed Acknowledgements are being used if receiving acknowledgements that acknowledge multiple packets. In this case, RTT is measured for the second sent packet unless TSO is enabled since the timestamp of the first packet is reflected by the receiving host. TCP Segmentation Offload causes multiple packets to have the same timestamp. `h_ertt` mitigates this issue by disabling TSO for one packet every RTT to obtain at least one accurate RTT estimation.

If the SACK option is enabled and a SACK packet is received, `h_ertt` matches only the the largest SACK with the corresponding sending packet to estimate RTT for that packet. All packets before this SACK packet are ignored [149].

`h_errt` also provides a flag (ERTT_NEW_MEASUREMENT) to inform the consuming module that a new measurement epoch (RTT interval) has elapsed. According to [149], this flag is set once per RTT to indicate a new accurate measurement is available to be used. The CDG FreeBSD implementation relies on this flag to know when one RTT has elapsed in order to perform its per RTT calculations of delay gradient calculation, back-off decision and *cwnd* growth during congestion avoidance.

We discovered that the CDG FreeBSD implementation has some minor differences to the CDG paper [12].

1. The CDG FreeBSD implementation [146] uses either smoothed or unsmoothed gradient samples to make its back-off decision during the slow start phase while the paper uses only smoothed gradients in this regard. We believe this is because the moving average reduces the gradient signal response to delay changes, which could result in CDG exiting slow start too late.

2. In the implementation, the shadow window increases only when *cwnd* increases (no increase on *cwnd* back-off) while the paper indicates that the shadow window increases even when delay-based back-off is performed.

In this thesis, we consider the FreeBSD implementation for our evaluation since it is the official unmodified implementation based on the original CDG, and was developed by the author of the paper [72].

2.8 Conclusions

During the last three decades, the Internet has become faster by many orders of magnitude and, at the same time, users have been deploying more Internet based applications with a diverse mix of bandwidth and latency requirements. To meet the demands of Internet users and applications, academic and industry research has focused on improving the performance of TCP, the Internet's dominant transport protocol.

Congestion control is a critical part of TCP, directly influencing transport performance. Consequently congestion control techniques have attracted a great deal of research attention. In this chapter, we have surveyed a range of key congestion control algorithms that utilise the delay signal to infer the existence of congestion and/or use the measured delay as part of their congestion response behaviour.

Standard TCP CC is effective in protecting the network from collapse. However, this is not the sole concern of modern CC. Efficient capacity utilisation, low queuing delay and tolerance of non-congestion loss have become key requirements imposed by many applications and services. One of the biggest issues of standard TCP is that it is unable to control the the latency caused by bottleneck queue congestion (or bufferbloat) due to using packet loss as congestion feedback. This problem has serious impact on delay-sensitive applications such as video conferencing and multiplayer online gaming.

One solution to the bufferbloat problem is to manage queues using delay-focused AQM instead of DropTail management of FIFO queues.

Delay-based congestion control interprets the delay in packet delivery (RTT or one-way delay) to infer network congestion early and control the congestion in an efficient manner without causing the buffer to fill. In general, the delay-based approaches compare the delay signal with a threshold (constant or adaptive), or monitor the delay trend or gradient to infer congestion. Such techniques are capable of reducing bottleneck queuing delay and packet loss rate, improving overall network performance.

However, using the delay signal as congestion feedback creates many challenges. Poor fair sharing with standard TCP is one of the main issues of using delay-based congestion control on the Internet. Some techniques primarily use the loss signal in their operations but utilise the delay signal as a secondary signal to improve throughput, scalability and/or compatibility of the protocol. Others are designed to switch from delay mode to loss mode based on the type of the competing flows to provide better compatibility with standard TCP. Using delay-based congestion control in the Internet can improve link utilisation and reduce the effects of bufferbloat.

CDG is a dual-mode hybrid CC algorithm that aims to provide high throughput with low queuing delay. It also aims to coexist with loss-based flows and perform well in lossy environments. Due to utilising the delay gradient signal and a probabilistic back-off function, CDG avoids using queuing delay estimation and thresholds. Moreover, switching to loss mode for a short period upon detecting competing loss based flows allows CDG to perform better than pure delay-based CC when coexisting with loss-based CC.

However, CDG is known to not fully utilise link in low multiplexing environments [12], and to have low performance when coexisting with loss-based flows (despite its coexistence measures) [153]. These issues have not been well understood and solved in the literature to date.

This inspires us to study these flaws and weaknesses in more detail and enhance CDG. In this work, we use the experimental methodology to evaluate CDG in a range of scenarios to understand its strengths and weaknesses. The experimental methodology produces more realistic results than simulations since many variables in real environments leads to different behaviours than under simulation.

Chapter 3

Experimental Methodology

3.1 Introduction

Theoretical analysis and study of congestion control protocols is fundamental and plays a key role in designing and understand the behaviour of the protocols. However, it is not enough to rely solely only on theoretical studies to confirm the functionality, usability and dependability of protocols. Protocol development also requires implementation, documentation, validation, testing and evaluation in testing environments that emulate real-world networks.

Studying CC protocol behaviours requires a fully controlled testing environment that can mimic realistic networks and testing scenarios. Additionally, different testing scenarios should be prepared to test how a CC algorithm behaves under different network conditions; otherwise, the algorithm could perform badly or unpredictably in specific untested scenarios.

CC algorithm are often evaluated using network simulators and/or network emulators. Many network simulators are designed to be flexible and easy to use with the ability to mimic real networks. However, external factors in real networks, such as channel noise, interference, jitter, OS TCP/IP stack implementations and system design and architecture, can obstruct the simulation from reflecting real networks behaviour. These factors can alter CC behaviours in many cases, especially for delay-based and delay-sensitive CC. Some network simulators, such as ns-2 [91] and ns-3 [154], support emulation mode to provide the ability to use live traffic from real OS kernel.

On the other hand, well-designed network emulators provide more realistic testing environments. They have the ability to reproduce different networks with real TCP/IP stack implementations, NIC drivers, packet processing delay, traffic generators, bandwidth limiters and many other factors. Therefore, using network emulators to evaluate CC algorithms can provide a better testing environment that reflects CC behaviours in real networks.

In this chapter, we describe the controlled testing environment and the testing scenarios that we use as well as challenges underlying the emulation quality. The rest of this chapter is organised as follows. Section 3.2 describes the tools we use to emulate links, generating traffic, TCP logging traffic statistics at the end hosts, and gathering bottleneck statistics. Section 3.3 presents the main measurements used in evaluation and comparison CC. Section 3.4 describes our testbed setup and network topology. Section 3.7 describes the scope of scenarios we cover in this thesis.

3.2 Tools

Evaluating protocols in emulated environments requires tools to emulate links, generate different types of network traffic, and collect statistics from the end hosts as well as middleboxes. In this section, we describe the tools that we use in evaluating congestion control protocols and the challenges facing some of these tools.

3.2.1 Link emulation

The TCP CC dynamic is highly influenced by the link characteristics and path conditions including bottleneck bandwidths, buffer sizes, path RTTs (RTT_{base}), random loss rates and queue management deployed at the bottlenecks. Consequently, test environments must provide the ability to set up the characteristics of the path in addition to different traffic generators to mimic TCP traffic in real-world scenarios.

Hardware link emulators can provide precise timing quality and support high data rates at the expense of cost. For example, Network emulator II from Ixia [155] (previously known as Anue Systems network emulator) can support 10 Gigabit Ethernet and accurate network condition emulation.Software network emulators can provide less precise, but acceptable in many cases, emulation quality with lower data rate speeds. However, software-based emulation has a very low or no cost at all. Popular examples of such network emulators are Dummynet [156] in FreeBSD and NetEm [157] with traffic control (tc) [158] in Linux operating system.

Topically, software network emulators utilise rate limiters (aka traffic shapers) to emulate the bandwidth of a link. Path delay is emulated by moving packets into a delay line before transmitting them to destination. To emulate lossy links, the emulator randomly drops packets based on configurable loss ratio. It is worth noting that packet loss emulation can be applied during packet enqueue or dequeue from the buffer depending on the emulator design. For example, Dummynet randomly drops packets during the enqueue process to emulate a specific loss rate.

Some software network emulators, such Dummynet, are able to emulate networks at data link level while others are able to emulate links at the IP level. Data link level emulation allows configuring testbed without packet forwarding (router) using bridging mode (similar to network switches) while IP level emulation requires the emulating machine to act as a router.

In addition, software network emulators are usually coupled with traffic classification software to specify which packets are subject to the emulation. For example, Dummynet is typically used with ipfw [159] in FreeBSD OS and NetEm is used with tc [158] in Linux. In this work, ipfw/Dummynet in FreeBSD11 is used to emulate links at IP layer level.

Dummynet stores packets in a buffer(s) and then fetches them using a packet scheduling algorithm to be sent to the destination network. In a single queue configuration, First-In First-Out (FIFO) is used where the first packet to enter the buffer will be dequeued first. The size of this buffer is configurable and is refereed to as *buffer size* (*bs*). Buffer sizes used in network middleboxes vary from brand to brand and model to model. Therefore, we use different buffer sizes in our evaluation.

Buffers are controlled by buffer/queue management schemes. The most common is Drop-tail in which packets are dropped from the end of the queue if there is no more room for new packets in a situation called *overflow*. More advanced queue management schemes (Active Queue Management) in which packets are dropped when queue lengths reach a specific threshold (in time or size). Dummynet originally implemented only Random Early Detection (RED) [43] and Gentle RED (GRED) [160] AQM. However, we implemented CoDel[47], PIE [15], FQ-CoDel [16] and FQ-PIE [17] for the FreeBSD version of Dummynet. Our implementation has been added of FreeBSD source tree since version FreeBSD 11.0 [19] and backported to FreeBSD 10.4 [20].

The Dummynet traffic shaper is used to mimic a required link speed (bandwidth). On every kernel tick, Dummynet dequeues packets from the buffer if the shaper allows. If the data rate is above the desirable bandwidth, packets rmain in the buffer.

Dummynet can be configured to add delays after packets are fetched to emulate different path RTT. Delays can be added to forward and reverse paths independently. In our experiments, we configure Dummynet to delay packets by half of the required path RTT for each of the forward and reverse paths. Dummynet implements delay emulation by temporally storing the packets in another buffer (heap) with the transmit time. On every tick, the packets are transmitted to the destination when the packets transmission time is greater than or equal to the current time. Dummynet can be configured to drop packets randomly to emulate a lossy environment where non-congestion related losses occurs.

Figure 3.1 shows a typical Dummynet link emulation.

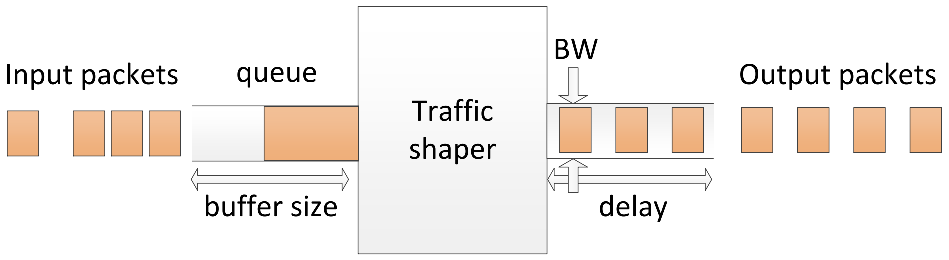


Figure 3.1: Typical Dummynet link emulation

3.2.2 Traffic generators

Traffic generators are used to mimic real network traffic that would be generated by user data transfer over the network. Traffic can be unidirectional such as audio and video multicast or bidirectional such as FTP, HTTP traffic and VoIP traffic.

Typically, traffic passes from the sending host (source) to the receiving host (sink). This involves starting the server and client software on the sending and the receiving hosts. For example, to mimic web serving traffic, an HTTP service and HTTP client should be started on the sending host and receiving hosts respectively.

In our testing environment, we use the `iperf` [161] tool to generate TCP traffic between two end points. This tool allows the sender host (client) to push data to the receiver host (server) at the maximum possible speed. TCP bulk data transfer can be easily emulated using this tool. Additionally, `iperf` can be used to generate UDP traffic at a desirable rate to mimic different UDP-based application protocol such as VoIP traffic.

As described in Section 2.1.1, TCP receiver buffer sizes should be larger than path BDP to let the sender push packets at the maximum link speed. We used a modified version of `iperf` tool [162] that accepts additional arguments to set the sizes of sending and advertised (receiving) windows. In our experiments, the receive buffer size of `iperf` is set to be larger than path BDP plus bottleneck buffer size to prevent limiting the number of bytes in flight by the receiver.

3.2.3 TCP Traffic loggers

In order to evaluate and explore TCP CC algorithms, TCP traffic statistics are collected from end hosts. Per-flow statistics such as packet size and advised window size are extracted from the packets. In this case, the `tcpdump` [163] tool is used to capture packets at all NIC's.

Other TCP/IP statistics such as congestion window size and socket buffer sizes cannot be retrieved from packet capture. Therefore, kernel TCP statistic loggers are also important.

These loggers collect TCP/IP statistics from the OS kernel TCP/IP stack and store them to a log file. This type of TCP logger is OS-specific.

In our testing environment, the Statistical Information For TCP Research (SIFTR) [164] logger is used under FreeBSD to log different TCP statistics. This logger is an event-driven logger which collects statistics on packets arriving and leaving the host. For logging CC specific statistics we patched SIFTR to collect different CDG (and our enhanced CC algorithm) statistics and internal variable values, and to output them as additional fields in the SIFTR log file.

For Linux, we developed `ttprobe` [25], a TCP event-driven[1] logger which captures more samples than the well know Web10G logger [26] with lower CPU overhead and better compatibility with newer Linux kernels.

Both SIFTR and `ttprobe` have extremely low CPU overhead, especially at low packet rates. A `ttprobe` description and performance analysis can be found in Appendix A.

3.2.4 Bottleneck queue statistic loggers

It is also useful to obtain a clear picture of what is happening inside bottleneck queue(s). Obtaining accurate measurements of the congestion level, queue length and queuing delay can help to explain the behaviours of delay-based congestion control algorithms. In our experiments, it helps in understanding how the algorithm reacts to the bottleneck queue dynamic, and the induced queuing delay that the algorithm generates.

We developed an event-driven Dummynet logger called DIFTR[2] (Dummynet Information For TCP Research) which can collect different queue statistics such as queue length, queuing delay and the number of dropped packets. It also logs AQM-specific statistics such as CoDel and PIE estimated queuing delay and FQ-CoDel and FQ-PIE per sub-queue statistics. DIFTR logs statistics on packet enqueuing, packet dequeuing and packet sending after emulating path delay.

3.3 Measurements

We use different measurement metrics to measure different performance aspects of the protocols.

[1]Similar to SIFTR

[2]DIFTR code was developed on the top SIFTR source code.

3.3.1 Throughput

Throughput is computed for flows by calculating the number bytes received per unit of time using a moving average window of size W. Interpolation is used to obtain more throughput samples when W is large to reduce noise and produce smoother plots. In our analysis unless otherwise mentioned, we use a one second window size and four interpolated points.

We also measure the flow average throughput for comparison purposes which is calculated as above using W equal to the flow running time with one interpolated point.

3.3.2 RTT

We use the Synthetic Packet-Pairs (SPP) RTT tool [165] to measure the round-trip time of a flow. SPP does not require time synchronisation between the end points and uses passive measurements. It produces more accurate RTT estimation than the RTT estimated by TCP stack.

SPP relies on packets observation times at two measurement points (called monitor and reference points). It captures packets at the two points and matches packets at one point with packets at the other point. Each packet is assigned a unique identification and coupled with capturing timestamps on both points. As a result two lists will be created, one at each measurement point. Starting from the beginning of the list at the monitor point, the closest pair of packets (one from monitor to reference point and one after it from the opposite direction) are used. An RTT sample is calculated as the timestamp difference between that pair at the reference point minus the difference between the pair at the monitor point [165].

When the two points are the sending and the receiving hosts, the RTT samples will be the actual RTT the sending host experiences. The traces captured by the `tcpdump` tool are used as input to SPP.

3.3.3 Queuing delay

Round-trip queuing delay can be measured using SPP by subtracting the base RTT from the SPP RTT estimation. A better and more accurate queuing delay can be obtained from inside the bottleneck. DIFTR computes packet sojourn time (similar to CoDel AQM [47]) for each dequeued packets. Therefore, an accurate queuing delay per queue can be obtained.

We use queuing delay from DIFTR in our experimental analysis unless otherwise mentioned.

3.3.4 Queue length

DIFTR also provides queue length statistics directly from inside bottleneck queues. Queue length samples are generated on packet enqueue and dequeue.

3.4 TEACUP - a specialised toolkit and testbed

Protocol evaluation and development requires conducting hundreds or thousands of controlled experiments that mimic different contexts and scenarios. Each experiment requires configuring many elements in the testing environment including end-hosts, routers and switches. This also involves changing TCP/IP parameters (e.g. enable/disable TCP options), operating system options (e.g. socket buffer sizes) and link emulation parameters (e.g. bandwidth and emulated delay). It also includes starting and stopping traffic generators and collecting all log files. Without a specialised and flexible automated testbed, this process becomes difficult and inefficient.

TCP Experiment Automation Controlled Using Python (TEACUP) [162] is a very powerful specialised automated TCP testbed software that provides all the required features for conducting TCP experiments. It emulates different real-world scenarios in a fully controlled environment. TEACUP can run and replicate experiments while varying experiment parameters including bandwidth, path RTT, bottleneck buffer sizes, queue management scheme and loss rate. Further, TEACUP supports a range of traffic generators and different operating systems.

TEACUP also provides a collection of basic analysis functions that simplify analysing TCP statistics such as extracting a wide range of statistics, correcting log timestamps and plotting different graphs. Moreover, it is a well-documented open source project [166] which makes extending features a straightforward process.

TEACUP separates experiment traffic from control traffic through the use of two separate networks: a control network and an experiment network. This separation prevents testbed control traffic from interfering with experiment traffic to produce a more accurate emulation.

All our experiments are conducted using the TEACUP testbed. The summarised steps of what TEACUP does when running an experiment are:

1. Boot hosts and router(s) to the desired OS (Linux, FreeBSD, Microsoft Windows or Mac OS) and test host connectivity.

2. Initialise and configure all hosts and router(s). This involves setting up TCP/IP parameter such as disabling hardware offloading, setting system receiver and sending buffer

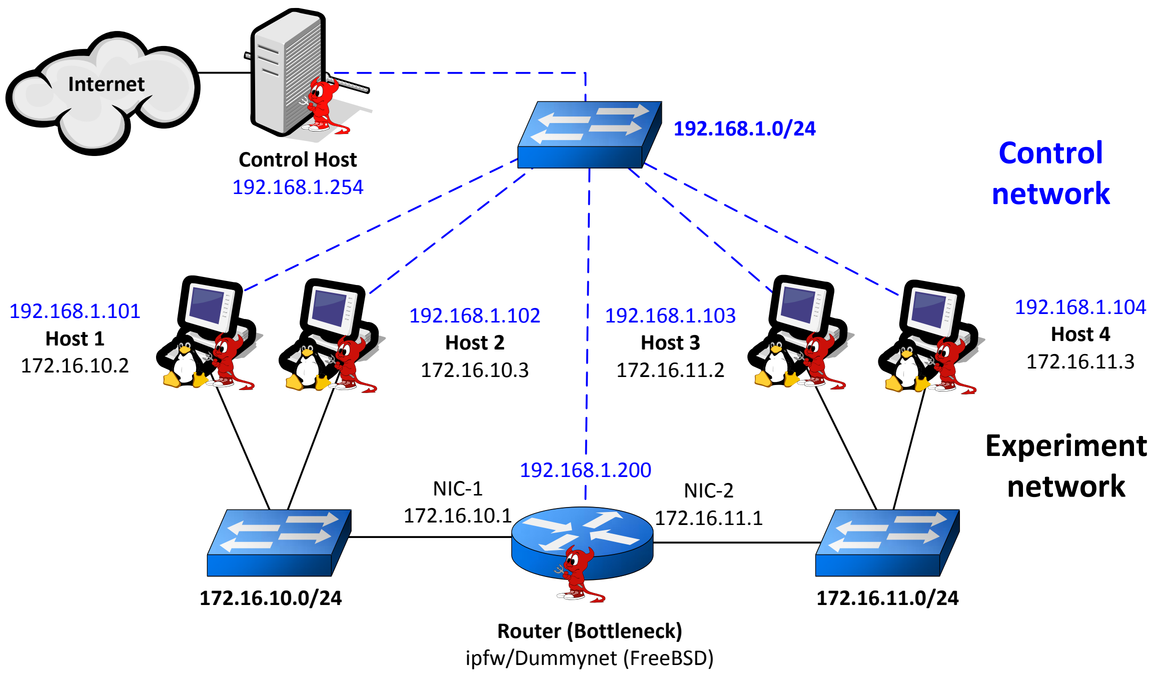


Figure 3.2: Testbed topology and network setup

sizes, setting up the bottleneck such as link's bandwidth, emulated delay, packet loss rate, buffer size, queue management.

3. Prepare traffic generators to run on the hosts at specific times.

4. Start all traffic and statistic loggers.

5. Wait for the experiment to finish.

6. Stop all loggers, collect log files and check their integrity.

7. Delete temporary files and finalise the experiment.

Our testbed includes four hosts, one software router and one control host. The control host is responsible for running the TEACUP software, storing experiment configurations, and the collected log and data files to be analysed later. Figure 3.2 shows the topology of our TEACUP based testbed.

The hosts hardware specifications are Intel Core I5 @ 3.2GHz processor, with 8GiB memory and Intel PRO/1000 Gigabit Ethernet interfaces. The router hardware specifications is Intel Core2 Duo @ 2.33GHz processor, 4GiB memory, Intel PRO/1000 Gigabit Ethernet interfaces. The testbed uses FreeBSD-11.0-RELEASE and OpenSUSE with 4.9 Linux kernel. The hosts are booted automaticity with the desired operation system at the start of the experiment.

3.5 Kernel tick rate implication

The kernel tick rate plays an important role in link emulation accuracy by providing higher time resolution and serving the queues more frequently. A high tick rate is required to better reflect the realistic operation of real links. We configured the FreeBSD router to use a 10KHz tick rate, this is large enough to allow smooth packet transmissions when emulating links up to 120Mbps.

We discovered issues with Dummynet's tick implementation which affect emulation accuracy and we provide fixes for these issues. Additionally, we find that Dummynet results in unintended TCP ACK compression in specific scenarios due to its traffic shaper. Appendix B discusses these issues in detail and their implications on link emulation, along with our solutions to realise better emulation with low system overhead.

3.6 Modern AQM implementation in Dummynet

Modern AQMs, such as CoDel[47], PIE [15] and FQ-CoDel [16], are attractive to both academic and industrial communities because of the high performance and low latency of these algorithms. Additionally, these AQMs have emerged as a solution to the bufferbloat phenomenon in home gateways. It is important to understand CC algorithm interaction with AQMs, and how AQMs can alter the CC design assumptions.

CoDel and FQ-CoDel have existed in the Linux kernel since version 3.5 [167] and PIE since version 3.14 [168]. There was no implementation for these AQM in Dummynet, we independently implemented these AQM based on IETF RFC specifications, verified their correct functionality and integrated them to FreeBSD 11 and 10.2 releases. Additionally, we applied the idea of the Flow-Queue part of FQ-CoDel to the PIE AQM to produce an experimental implementation for FQ-PIE. Our technical report [17] includes details of CoDel, PIE, FQ-CoDel and FQ-PIE implementation for ipfw/Dummynet in FreeBSD. Support for this implementation has added to TECUP [169].

3.7 Testing scenarios

In our experiments, we emulate links mimicking slow to medium speed home Internet subscriptions. We use downlink bandwidths between 1.5Mbps and 25Mbps in most experiments, and 1Mbps and 50Mbps in some experiments. We use uplink bandwidths between 0.5Mbps and 20Mbps which are enough to prevent the uplink from becoming a bottleneck [170]. Table

Table 3.1: Downlink and uplink bandwidths used in the experiments.

Down/up stream bandwidth (Mbps)	Mimic gateway of
1.5/0.5	ADSL1
4/1	low speed ADSL2+
12/1	Medium speed ADSL2+
25/5	Low speed fibre subscription
50/20	Medium speed fibre subscription

Table 3.2: Path RTTs used in the experiments.

Path RTT (ms)	Hosts locations
< 10	Local area
10	Intra-state
40	Inter-state
180	Intra-continent
240	Inter-continent
340	Far continent

3.1 shows the downlink and uplink bandwidths we use in our experiments. In our graphs and discussions, only the downlink bandwidth is mentioned for simplicity.

We use a wide range of path RTTs in our tests from 10ms (representing inter-state prorogation) to 340ms (representing far inter-continental connections) as shown in Table 3.2. In some experiments, we use smaller path RTTs to mimic local networks.

Different buffer sizes are used depending on experimental context but we never use buffer sizes less than one BDP worth of buffer. In all our experiments, we set up Host 1 and 2 to be the receivers and Host 3 and 4 (see Figure 3.2) to be the senders unless otherwise mentioned.

3.8 Conclusion

In this chapter, we described the experimental methodology we use in this thesis as well as the testbed used in protocol evaluation. Our automated testbed allows us to conduct experiments to properly evaluate performance of CC algorithms. It is able to emulate different network settings and mimic realistic scenarios. Further, the testbed software collects different TCP and traffic statistic without interfering between experiment traffic and testbed control traffic.

Chapter 4

CAIA-Delay Gradient CC Evaluation

4.1 Introduction

In this chapter, we describe the CDG algorithm and evaluate CDG in a single-flow situation, coexistence with loss-based flows and slow-start phase for different bottleneck bandwidths, buffer sizes and path RTT. Further, we identify CDG issues and the potential causes of these problems. In Chapter 5 we will use these insights to develop and improve CDG such that it can be deployed as on the Internet as both a conventional and low priority CC.

The rest of this chapter is organised as follows. Section 4.2 presents the evaluation of CDG in single flow scenarios and identifies CDG issues that reduce protocol throughput in these scenarios. In Section 4.3, we evaluate CDG coexistence with loss-based CC, namely CUBIC, and identify the potential problems leading CDG to obtain low bandwidth share. Section 4.4 presents CDG slow-start evaluation and Section 4.5 concludes this chapter.

4.2 CDG link utilisation

In this section, we investigate how much the congestion avoidance mode of CDG CC algorithm [12] utilises bottleneck capacity in single flow scenarios. In this simple scenario, one sender transmits bulk data using CDG-based TCP to a receiver without any competing background traffic. This scenario is not uncommon in real-life traffic such as downloading system updates or uploading large files overnight.

Using CDG's FreeBSD implementation [145], we conduct an experiment to measure average link utilisation for a CDG generated TCP flow as it traverses a bottleneck. The experiment is repeated for a range of bottleneck bandwidths (B_{rate}) from 5Mbps to 25Mbps, a range of emulated path RTT (RTT_{base}) from 10ms to 300ms, and a 1000 packet droptail FIFO buffer.

The B_{rate} range reflects typical Internet connection speeds in a home network context when the network is connected to the Internet through a broadband modem (see Section 3.7). The RTT_{base} range emulates connections between two points in different geographical locations from intra-state to intercontinental. We use a FIFO buffer because it is the typical mechanism utilised by most network equipment.

Although CDG does not need a large bottleneck buffer size in most cases to achieve full bandwidth utilisation, a 1000 packet buffer size is chosen to be greater than the largest BDP in our test (~ 625 packets when B_{rate}= 25, RTT_{base} = 300ms and 1500 byte packet size). The receiver is configured to use 1MiB receiving buffer size which is again greater than the largest BDP in our test (0.9 MiB for B_{rate}= 25 and RTT_{base} = 300ms) to prevent capping the flight size based on the receiver window. The `iperf` tool is used to generate a 300 seconds CDG flow.

The average link utilisation for the experiment is calculated for t=90s to 290s of each run. First 90 seconds are excluded to skip CDG's slow start and to ensure that CDG reaches a stable congestion avoidance state before making measurements.

Figure 4.1 shows that when RTT_{base} < 20ms, link utilisation is above 90%. As RTT_{base} increases link utilisation decreases. When RTT_{base} > 200ms, link utilisation drops below 50%.

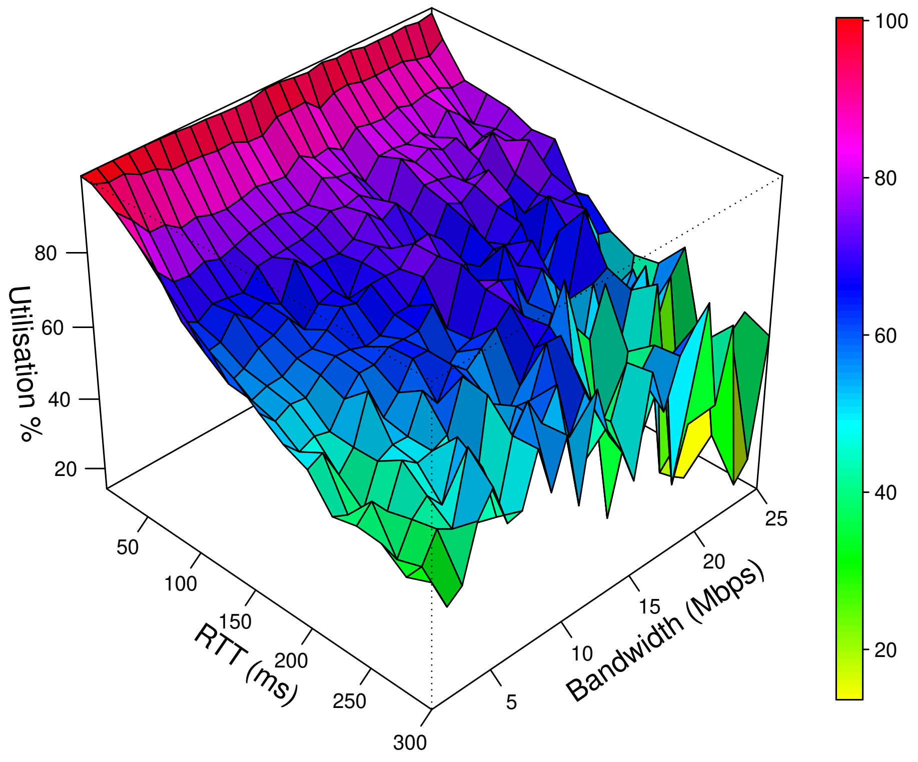


Figure 4.1: Link utilisation of single CDG flow

When RTT_{base} > 250ms and B_{rate} > 20Mbps link utilisation decreases below 40%. Figure 4.1 also shows that bottleneck bandwidth does not have a significant effect on CDG link utilisation degradation.

4.2.1 Large *cwnd* back-off and slow recovery

Achieving 100% link utilisation requires a TCP flow to fill ‘the pipe’ by keeping bytes in-flight at least equal to the “optimum” window size (i.e path’s BDP). CDG attempts to keep *cwnd* close to the path BDP to achieve high throughput while preventing build queue up in the bottleneck. In this context any back-off causes *cwnd* to fall below the path BDP.

As soon as congestion is detected using the delay gradient signal, CDG backs off *cwnd* using β_{delay} factor (default 0.7). For continuous 100% utilisation across congestion epochs we require *cwnd* to be no less than BDP after backing off *cwnd*.

Since CDG increases *cwnd* linearly by one MSS every RTT when no congestion is detected, it takes time to recover *cwnd* to reach BDP again. Flows traversing larger BDP paths require a longer time to recover *cwnd,* resulting in bytes in-flight being lower than BDP for a longer period. The degree of utilisation degradation is dependent on the time taken for *cwnd* to reach BDP after back-off. As shown in Figure 4.1, higher RTT paths result in longer recovery periods.

We take a single CDG flow experiment with 20Mbps bandwidth and 100ms emulated RTT as an example to further explore this behaviour. Figure 4.2 illustrates how CDG decays *cwnd* below the path BDP to about 180KiB, resulting in throughput (calculated over one second moving average window) being decreased to about 16Mbps. This figure also shows that it takes about 5 seconds for CDG to recover *cwnd* to BDP. This saw-tooth *cwnd* behaviour is similar to the TCP Reno algorithm. In this case the large bottleneck buffer size does not help since the congestion signal here is delay rather than loss. A similar effect can be measured in all cases where average throughput is well below 100%.

4.2.2 Unnecessary back-off

As CDG aims to maintain *cwnd* close to BDP, any *unnecessary cwnd back-off* results in *cwnd* decreasing well below the path BDP, leading to a reduction in link utilisation. We define unnecessary back-off as when *cwnd* decreases while it is smaller than the path BDP (i.e. average sending rate is lower than the bottleneck bandwidth). This issue becomes more problematic when the number of flows sharing the link is very small (lightly multiplexed environment). In this case, link capacity will not be fully utilised.

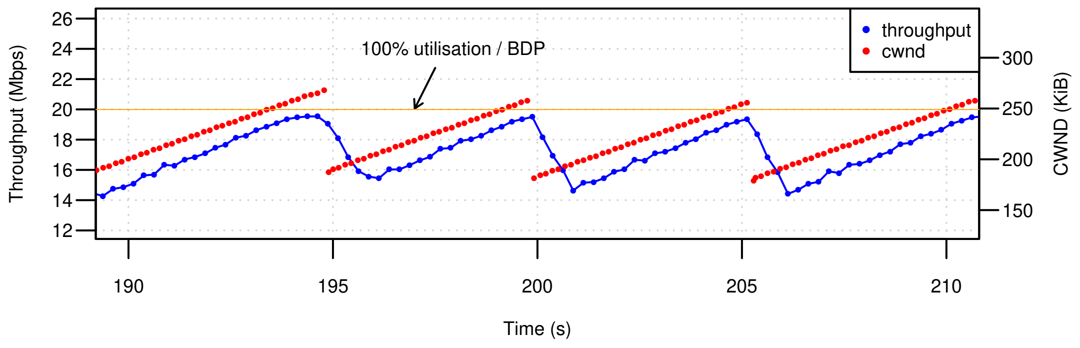


Figure 4.2: The impact of *cwnd* back-off on flow throughput

To understand why CDG sometimes backs-off unnecessarily, a microanalysis is performed for one instance of Figure 4.1. We select an experiment with B_{rate}=10Mbps and RTT_{base}=100ms (the path BDP = 122KiB) for our case study. In this experiment, the link utilisation was approximately 73%. We can clearly see in Figure 4.3 CDG occasionally backs off *cwnd* when *cwnd* is below BDP.

We count the total number of *cwnd* backs-off and number of back-off events while *cwnd* is below the path BDP (i.e. unnecessary back-off). We exclude the slow-start interval from the calculations.

The results show that CDG backed-off *cwnd* 140 times in total. 66% (92 times) of those back-offs occurred when *cwnd* was below BDP. Additionally, we found that CDG performed a double back-off (two successive backs-off with a two RTT interval gap between them) while *cwnd* was below BDP 11 times. This multiple back-off behaviour causes *cwnd* to drop significantly below BDP, leading to a large reduction in average link utilisation. As the window size drops to $(cwnd \times 0.7 + MSS) \times 0.7$ after a double back-off.

To evaluate why this occurs, we calculate the probability that CDG chooses to back-off in the N^{th} RTT after first back-off when *cwnd* remains below BDP. Figure 4.4 illustrates that

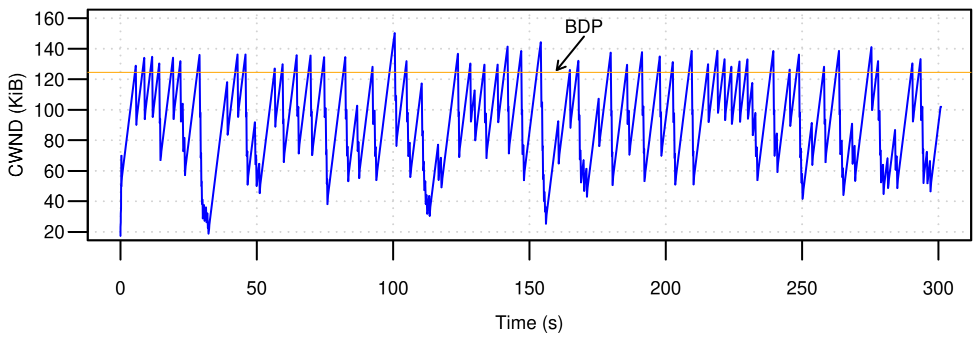


Figure 4.3: CDG occasionally backs off *cwnd* while *cwnd* below BDP

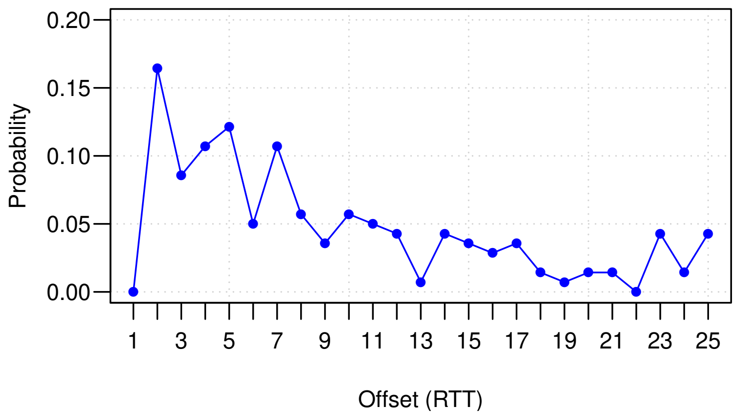


Figure 4.4: Probability of CDG backing-off again after N^{th} RTT

there is a relatively high probability that CDG backs-off close to (within four RTT intervals) the first back off. These statistics clearly reveal that CDG performs unnecessary back-off on a large scale for this scenario.

Both consecutive and early *cwnd* back-off reduce CDG performance in a single flow scenario. Additionally, we can see in Figure 4.1 that link utilisation is less than 50% when RTT_{base} > 200ms and B_{rate} >15Mbps scenarios. This large utilisation reduction is related to both the CDG β_{delay} multiplicative factor (default 0.7) and the relatively large BDP in these scenarios. In such conditions, *cwnd* drops well below the path BDP whenever CDG instigates a delay-based back-off with *cwnd* close to the path BDP. When RTT is large, CDG needs a longer time for *cwnd* to recover to the path BDP, causing the link to be underutilised for a longer period.

4.2.2.1 Underlying problem for consecutive back-off

To understand why CDG sometimes backs off unnecessarily, we instrument different CDG state variables during an interval when an unnecessary back-off occurs. We ignore $ertt_{max}$ gradient in this analysis because all back-offs occur based on $ertt_{min}$ gradient measurements in the selected interval.

The *cwnd* vs. time plot shown in Figure 4.5a shows that CDG backs off three times unnecessarily after first back-off at t=154.83s. The lower area between the path BDP and *cwnd* reflects the amount of unused bandwidth. As this area increases, the degree of link underutilisation will be larger. We can see in this figure that after *cwnd* exceeds the path BDP for an interval of time, CDG decreases *cwnd* using 0.7 beta factor. This back-off is enough to decrease *cwnd* below the BDP and reduce the queuing delay. The progression of bottleneck queue occupancy shown in Figure 4.5b clearly shows that the first back-off is effective and drains the queue to about three packets. This is inline with ERTT and $ERTT_{min,n}$ plots shown in Figures 4.5b and 4.5d respectively. In these figures we can see the RTT drops from 122ms to about 102ms after the back-off. We note that minimum RTT is higher than emulated the path RTT (100ms) due to ipfw/dummynet's internal architecture and operating system interrupt frequency as described in 3.5.

The first *cwnd* back-off event occurs at t=154.83s due to queue building up result in $ertt_{min}$ reaching 117ms. At the same point, $\bar{g}_{min,n}$ becomes 1.25 due to non-negative $g_{min,n}$ samples as shown in Figures 4.6c and 4.6a. This results in $P_{backoff}$ (Figure 4.6d) being 0.34, resulting in CDG backing off *cwnd* (based on a random generated number).

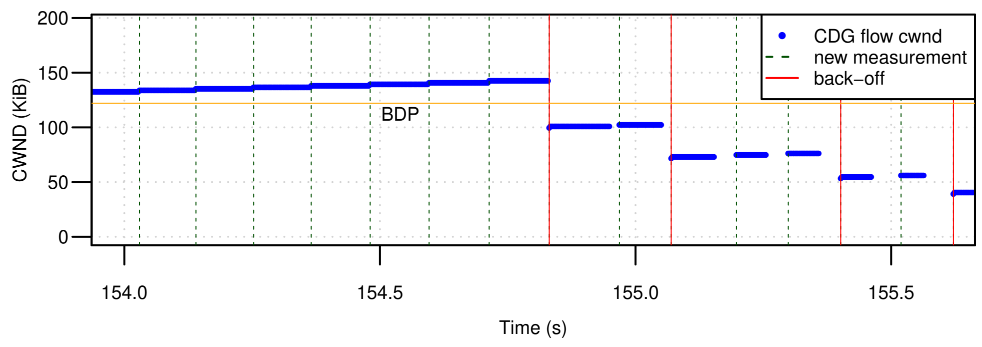
This is a desired back-off due to the queue build-up. However, at t=155.07s CDG performs another (unnecessary back-off) because $P_{backoff}$ is still high at 0.25 (since $\bar{g}_{min,n}$ =0.875) even through $ertt_{min}$ drops to 102ms. $g_{min,n}$ and $\bar{g}_{min,n}$ values for the interval from t=154.03s to t=155.62s are shown in table 4.1.

Two causes combine to result in a high probability of a consecutive *cwnd* back-off. The first is the effect of the smoothing window used by CDG, and the second is skipping a delay gradient sample after the back-off without skipping RTT measurements. These factors keep $P_{backoff}$ high after the first *cwnd* back-off event.

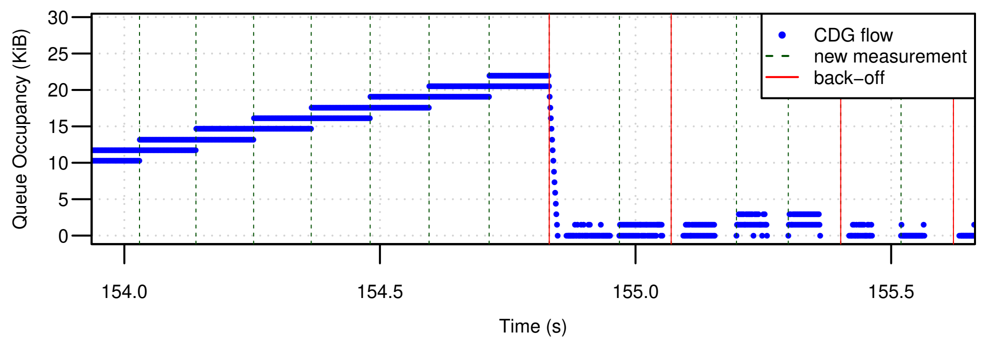
Firstly, CDG gradient smoothing window (default eight $g_{min,n}$ measurements) causes $\bar{g}_{min,n}$ to slowly reflect RTT reduction after back-off. As mentioned in Section 2.7, CDG uses the moving average window to address the fluctuation and noise of the RTT signal and measurement. However, this mechanism reduces algorithm responsiveness to delay signal changes as mentioned in Section 2.3.1.2. Therefore, CDG requires multiple RTT cycles after back-off to calculate a small $\bar{g}_{min,n}$ and as a result obtain a small $P_{backoff}$. This results in a higher chance for CDG to back off multiple times unnecessarily. Table 4.1 shows slow $\bar{g}_{min,n}$ decrease after the first back-off at t=154.83s regardless of the small g_n values.

Secondly, CDG misses an important $RTT_{min,n}$ gradient sample ($g_{min,n}$) due to skipping a delay gradient sample after back-off without ignoring RTT measurements during that cycle. This sample can reduce $\bar{g}_{min,n}$ value and change CDG behaviour significantly.

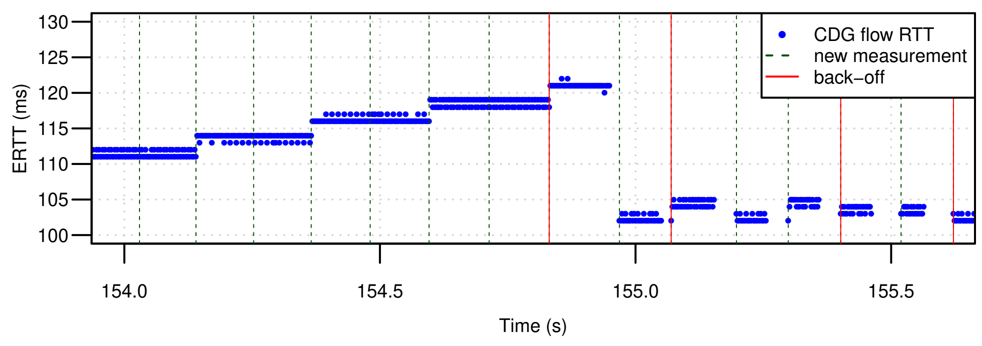
CDG wisely skips gradient measurements for an RTT interval (one measurement cycle) after backing-off because the sender cannot measure the effect of *cwnd* reduction before one RTT from that back-off. However, CDG only skips $g_{min,n}$ and $g_{max,n}$ without skipping RTT measurements ($RTT_{min,n}$, $RTT_{max,n}$). Before explaining the implication of not skipping RTT measurements of next measurement cycle, let us discuss the characteristic of RTT samples during a measurement interval after back-off when actual congestion happens in the network.



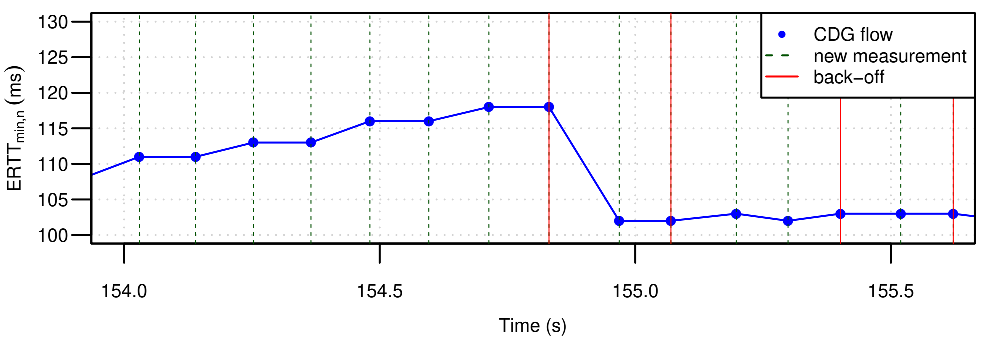
(a) multiple *cwnd* backing-off



(b) Queue occupancy

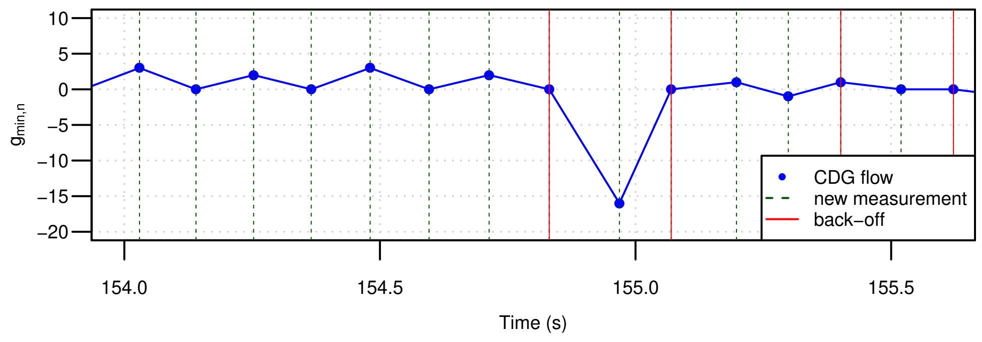


(c) ERTT

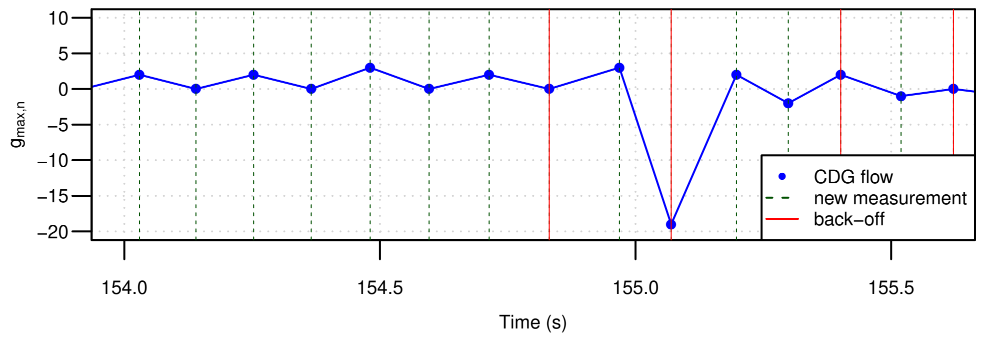


(d) $ERTT_{min}$ versus time sampled every RTT

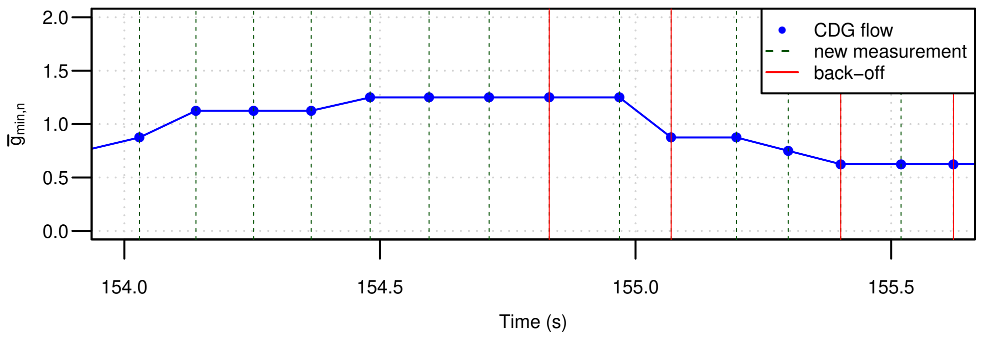
Figure 4.5: Single CDG flow performing multiple back-off



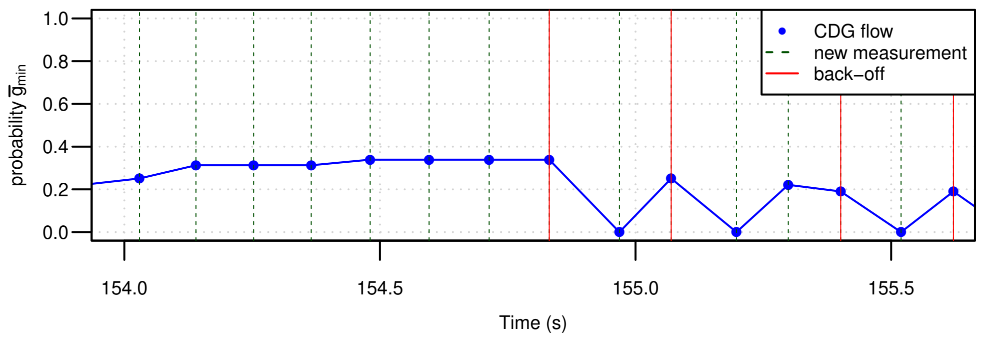
(a) $g_{min,n}$ versus time (RTT_{min} unsmoothed gradient)



(b) $g_{max,n}$ versus time (RTT_{max} unsmoothed gradient)



(c) $\bar{g}_{min,n}$ versus time



(d) back-off probability

Figure 4.6: Zoomed-in delay gradient stats when CDG flow performing multiple back-off

During an RTT interval after *cwnd* back-off, $RTT_{max,n}$ will stay high because $RTT_{max,n}$ is measured over an RTT interval. Since a maximum filter is used, any large RTT sample will keep $RTT_{max,n}$ large. Therefore, $g_{max,n}$ most probably will be positive or zero in next measurement round after back-off in a single CDG flow scenario.

On the other hand, $RTT_{min,n}$ decreases because at exactly one RTT interval the effect of the back-off will be measurable at the sender. Due to the effect of the minimum filter used to measure RTT_{min}, $RTT_{min,n}$ will include at least one RTT sample that reflects the *cwnd* reduction. That is, the ACK packet for the first sent packet in the sending window (last ACK packet received) will carry a delay measurement sample that reflects the reduction in *cwnd*. Therefore, $g_{min,n}$ most probably will be negative in next measurement round after back-off.

For this reason, skipping $g_{max,n}$ is useful to prevent additional back-off while skipping $g_{min,n}$ is bad because it prevents $\bar{g}_{min,n}$ from decreasing quickly after *cwnd* reduction.

Now assume CDG backs off *cwnd* at cycle b. CDG calculates $g_{min,b+2}$ according to equation 4.1. Since $g_{min,b+1}$ is skipped, it is clear that no $RTT_{min,n}$ gradient will include the difference between RTT_{min} before and after seeing the effect of *cwnd* back-off.

$$g_{min,b+2} = RTT_{min,b+1} - RTT_{min,b+2} \tag{4.1}$$

We use the same experiment described above to analyse this behaviour. We can see in Figure 4.5b that the bottleneck queue starts to drain immediately after back-off (t=154.83s) but the sender does not see RTT reduction until one RTT later at t=154.97s (Figure 4.5c). These figures illustrate why it is important to skip RTT measurements for one RTT interval after back-off.

In Figure 4.5d, however, we can see at t=154.97s RTT_{min} decreases to around 102ms causing $g_{min,n}$ to dip to -16 (Figure 4.6a) which happens directly after the back-off. As shown in Table 4.1, this value is excluded from $\bar{g}_{min,n}$ calculation and thus does not have significant impact to reduce the probability of further back-off.

On the other hand, in Figure 4.6b we can see that at t=154.97s $g_{max,n}$ is still positive at 3 and after another cycle at t=155.07s it dips to -19. This clearly shows that RTT_{max} does not reflect *cwnd* back-off after one RTT interval but after two RTTs.

It is worth noting that at t=155.30s $P_{backoff}$ is around 0.2 and before t=154.83s $P_{backoff}$ is around 0.3 but CDG does not back-off. The reason behind that behaviour is that making a probabilistic backing-off decision is based on a random generated number ($random < P_{backoff}$) and fortuitously the random generated numbers were greater than $P_{backoff}$.

Table 4.1: $\bar{g}_{min,n}$ calculation for interval t=108.73s to t=110.05s using 8 samples smoothing window where g_n is $g_{min,n}$

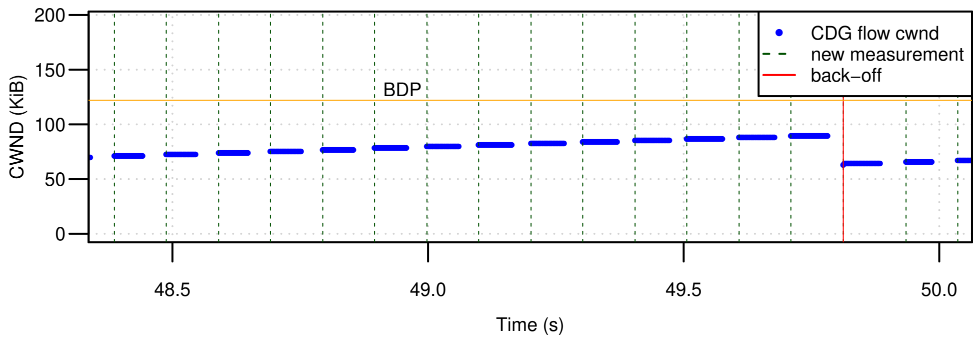
t(Sec)	g_{n-7}	g_{n-6}	g_{n-5}	g_{n-4}	g_{n-3}	g_{n-2}	g_{n-1}	g_n	$\bar{g}_{min,n}$
154.03	-2	2	0	2	0	2	0	3	0.875
154.14	2	0	2	0	2	0	3	0	1.125
154.25	0	2	0	2	0	3	0	2	1.125
154.37	2	0	2	0	3	0	2	0	1.125
154.48	0	2	0	3	0	2	0	3	1.25
154.60	2	0	3	0	2	0	3	0	1.25
154.71	0	3	0	2	0	3	0	2	1.25
154.83	3	0	2	0	3	0	2	0	1.25
154.97	0	2	0	3	0	2	0	(-16)	**(skipped)**
155.07	0	2	0	3	0	2	0	0	**0.875**
155.20	2	0	3	0	2	0	0	(1)	**(skipped)**
155.30	2	0	3	0	2	0	0	-1	**0.75**
155.40	0	3	0	2	0	0	-1	1	**0.625**
155.52	3	0	2	0	0	-1	1	(0)	**(skipped)**
155.62	3	0	2	0	0	-1	1	0	**0.625**

4.2.2.2 Underlying problem for spurious back-off

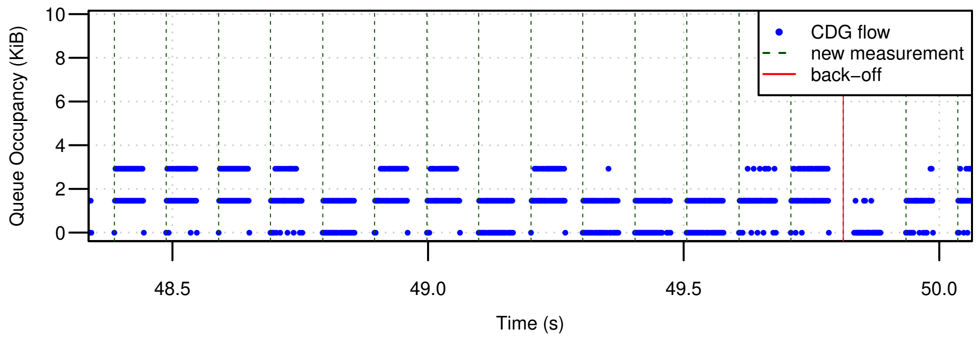
CDG randomly backs off *cwnd* when the window is smaller than the path BDP. This spurious back-off causes loss in flow throughput and has no useful outcome on reducing queuing delay. Back-off happens in this case study occurs only due to $ERTT_{min}$ measurements which is why we do not explore the $ERTT_{max}$ measurements in this investigation.

We can see in Figure 4.7a that at t=49.812s CDG backs off *cwnd* while the window has not reached BDP. In Figure 4.7b, we also can see that there is no actual queue build up at that instance in time. This figure also shows that the queue length fluctuates between zero and two packets which is considered normal behaviour. This queue occupancy fluctuation happens due to the TCP bursty sending caused by the TCP delayed acknowledgement mechanism [151] as well as the asynchronous timing between the sent packets and the scheduler at the bottleneck. The estimated RTT by the sender also reflects that behaviour as shown in Figure 4.7c.

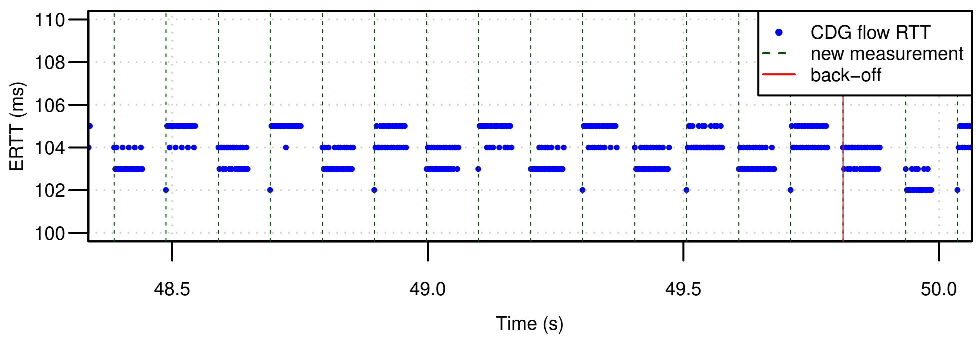
This instability causes the delay gradient signal (g_n) as well as the smoothed gradient $(\bar{g}_n)$ to oscillate. Although the smoothing window used by CDG can minimise this noise, it cannot be eliminated completely. In Figures 4.8a we can see g_n is vulnerable to queue oscillation significantly since delay gradient fluctuates between -2 and 2 almost every RTT. In Figure 4.8b, however, we can see the smoothed gradient signal is semi-stable with some oscillations. These oscillations lead to a high back-off probability value which causes the sender to back-off when there is no actual congestion.



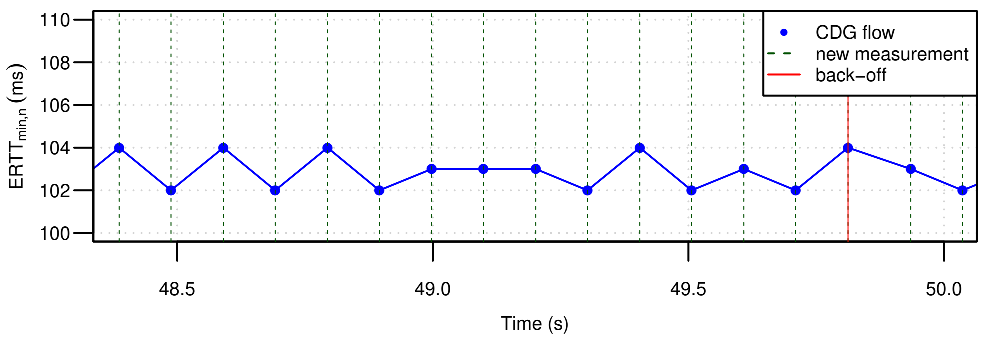
(a) Spurious *cwnd* backing-off



(b) Queue occupancy

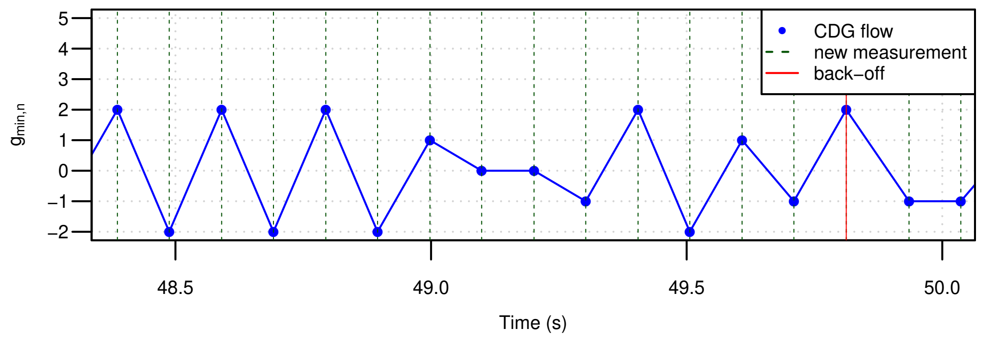


(c) ERTT

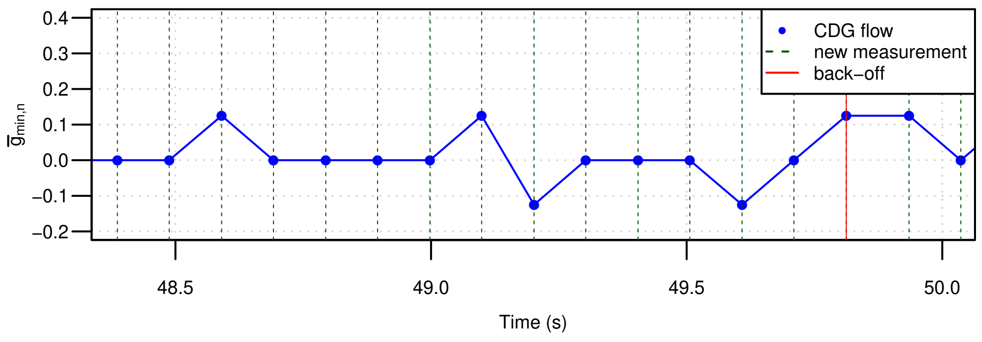


(d) $ERTT_{min}$ versus time sampled every RTT

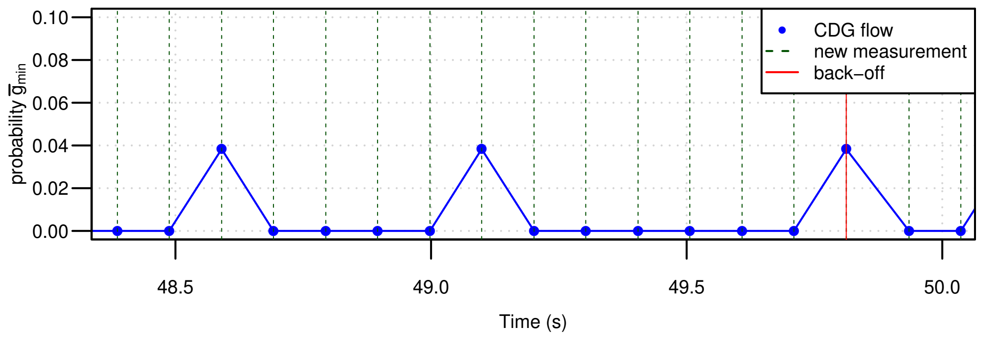
Figure 4.7: Single CDG flow performing spurious back-off



(a) $g_{min,n}$ versus time (unsmoothed gradient)



(b) $\bar{g}_{min,n}$ versus time



(c) back-off probability

Figure 4.8: Zoomed-in delay gradient stats when CDG flow performing spurious back-off

Figure 4.8c shows that the back-off probability becomes 0.038 three times during the shown interval (t=48.5s to t=50s). Although 0.038 is a low $P_{backoff}$, CDG decides to back off *cwnd* at the third instance at t=49.81s. As mentioned in Section 4.2.2.1, that occurs as the decision is made based on a random generated number ($random < P_{backoff}$).

Preventing spurious backing-off without heuristic is hard to achieve. Using thresholds to prevent CDG from backing off can work if accurate queue delay measurement can be obtained. However, estimating queue delay is fraught with difficulties including RTT_{base} estimation.

4.2.3 Late *cwnd* back-off behaviour

Occasionally CDG makes a late decision to back-off *cwnd* which causes RTT spikes or undesirable queuing delay. The uneven RTT leads to undesirable queue fluctuation which affects protocol stability without performance improvement. Figure 4.9 shows this behaviour for a 60 second CDG flow traversing a bottleneck that emulates 10Mbps and 100ms base RTT link. We can see in this figure that RTT occasionally increases much higher than the average RTT of this experiment. At t={12, 18, 38}s for example, RTT increases above 115ms, which is 15ms higher than the base RTT. The average queuing delay for this experiment is 2.7ms and average throughput is 7.76Mbps.

To understand why these spikes appear, we examine the period around the RTT spike at t=38s for a close-up analysis. In Figure 4.10a, we can see that *cwnd* increases over the path BDP at t=37.5s and keeps growing for 1.5 second until reaches 148KiB at t=39.1s. This causes a queue up to 20KiB in size to build up in the bottleneck before backing off *cwnd* as shown in Figure 4.10. As a consequence, the measured $ERTT_{min,n}$ at the sender increases up to 118ms as shown in Figure 4.11a.

The reason for the late back-off decision is the nature of the probabilistic function which depends on randomly generated numbers, occasionally resulting in CDG not backing off. In addition, the noise of the delay signal causes the smoothed gradient to increase slowly. In Figure 4.12a we can see $g_{min,n}$ values fluctuate between 0 and 2 resulting in $\bar{g}_{min,n}$ rising slowly as shown in Figure 4.12b. The reason for non-increasing RTT (which causes zero g_n) is the asynchronous timing between measurement cycles of the sender and bottleneck packet scheduler.

Most importantly, in single CDG flow scenarios, the $\bar{g}_{min,n}$ value is capped by the serialisation delay of MSS times the smoothing window size. In this experiment for example, $\bar{g}_{min,n}$ cannot increase higher than 9.6 (1.2×8) since the serialisation delay of 1500 bytes packet at 10Mbps is 1.2ms. We can see how $\bar{g}_{min,n}$ becomes almost constant between t=38.6s and

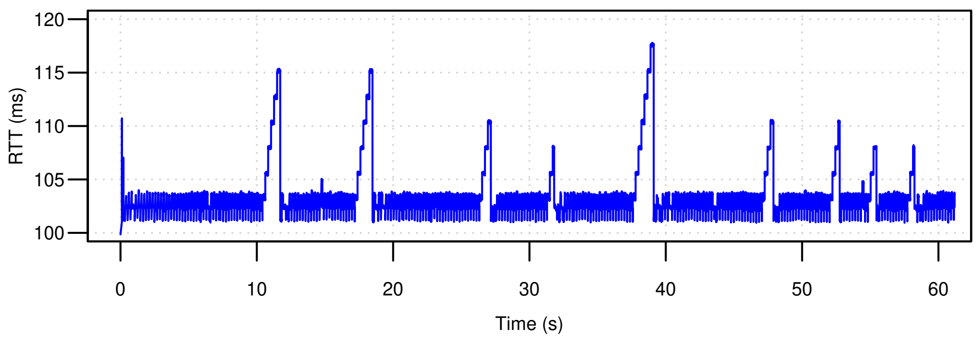
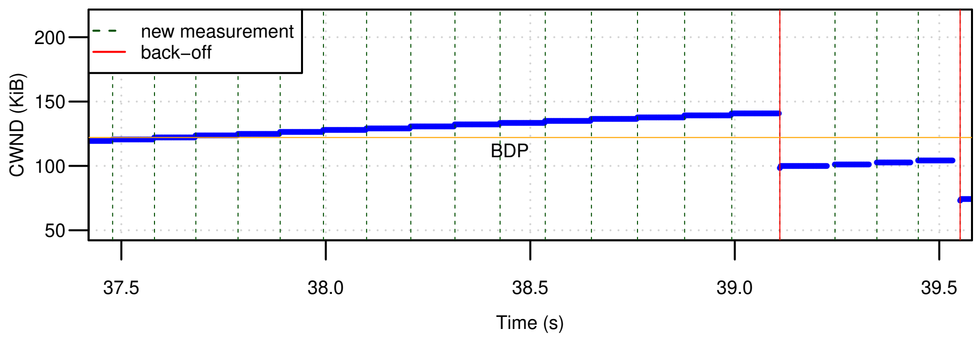


Figure 4.9: CDG occasionally does not back off *cwnd* on time resulting in RTT spikes



(a) CDG increases *cwnd* higher than BDP

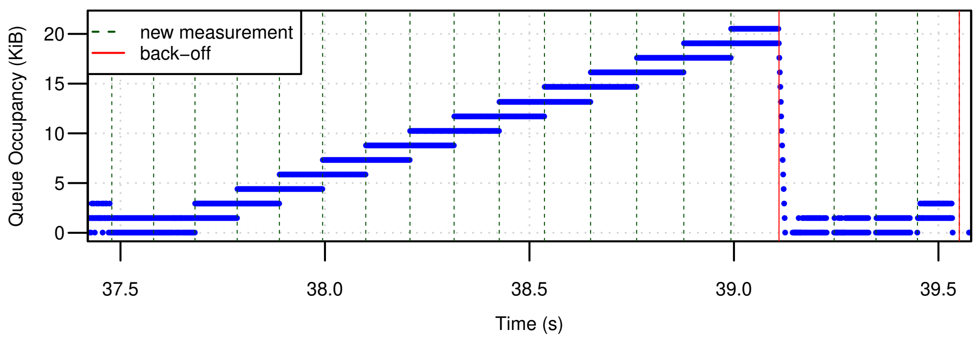
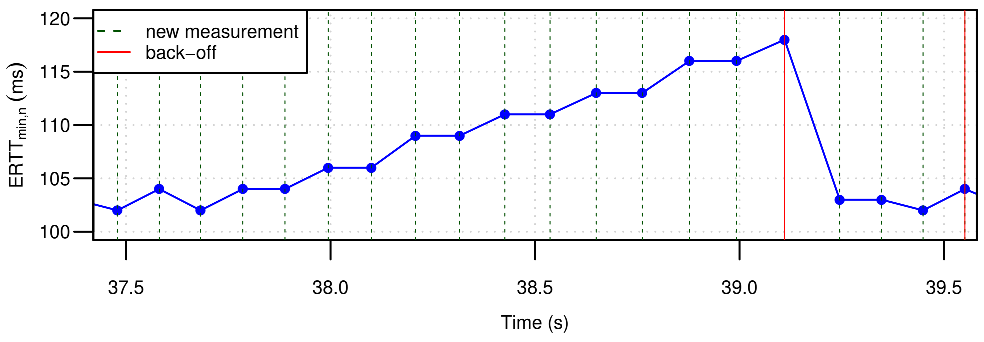


Figure 4.10: Queue occupancy



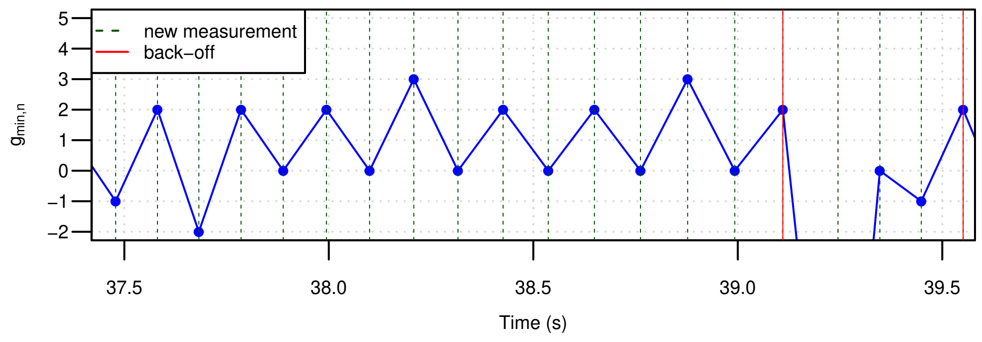
(a) $ERTT_{min}$ versus time sampled every RTT

Figure 4.11: CDG increases *cwnd* too much causing an RTT spike before backing off

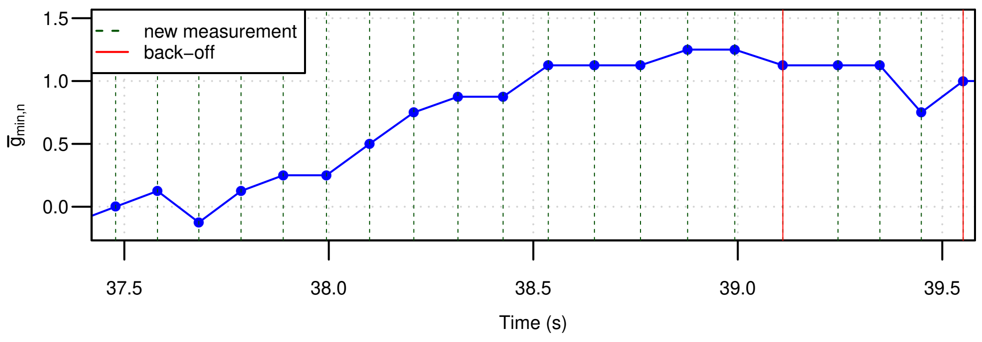
t=39.1s due to the smoothing window.

As a result, the back-off probability slowly increases and then becomes flat as shown in Figure 4.12c.

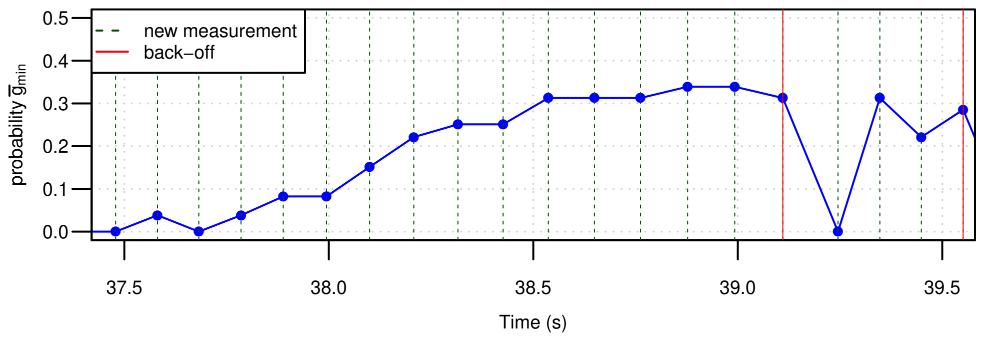
Another method in making the back-off decision is essential to solve this problem since making a decision based on $P_{backoff}$ is not sufficient. In Section 5.2.7 we propose a method to solve this issue and provide a lower queuing delay.



(a) $g_{min,n}$ versus time (unsmoothed gradient)



(b) $\bar{g}_{min,n}$ versus time



(c) back-off probability

Figure 4.12: Delay gradient measurement indicates congestion but CDG does not back off on time

4.2.4 Link utilisation in multi-flow environment

In multiplexed environment, the effect of low link utilisation of a single CDG flow becomes less harmful. The multiplexing back-off of each individual flow allows *cwnd* of other flows to grow further. Therefore, the overall link utilisation becomes close to 100% as the number of concurrent flows increase.

4.3 Coexistence with loss-based flows

As previously described, the main obstacle preventing wide deployment of delay-based TCP on the Internet is its poor ability to coexist with loss-based flows. More specifically, when delay-based TCP competes with loss-based TCP for bottleneck bandwidth, delay-based TCP obtains a very low share, particularly in large FIFO buffer sizes.

In a typical home network for example, user devices connect to the home broadband gateway via fast links (wired or wireless). Home gateways are bottlenecks for upstream traffic because they have lower uplink speed slower than home network speed. FIFO with droptail is the common (and usually default) mechanism to manage buffers in typical home gateways[1].

Let us consider the following scenario to understand the impact of flow coexistence on user experience. Suppose user A is uploading a very large personal video file to a video-sharing service while user B is sending an email with a large attachment. Assume user A's laptop is configured to use loss-based TCP (e.g. CUBIC for example) while user B's PC is configured to use low latency delay-based TCP transport (e.g. TCP Vegas).

In this context, user B will notice that his email takes a long time to be sent so he/she has to wait much longer than when he/she is alone at home. Similarly, if user B is busy with a video conferencing call and then user A starts uploading his file. User B will notice that his video call quality reduces and the video starts becoming choppy and unstable.

It is clear that in these contexts, the unfairness between loss-based and delay-based flows impacts badly on Quality of Experience (QoE) for user B.

A similar situation occurs when two users are downloading large files where the bottleneck is at the service provider side (Digital Subscriber Line Access Multiplexer (DSLAM) in Digital Subscriber Line (DSL) connection for example).

As previously described, conventional loss-based TCP probes path capacity by increasing the number of packets in-flight until packet loss occurs. As soon as *cwnd* reaches the path BDP (i.e. average sending rate reaches bottleneck bandwidth capacity), a queue starts forming in the bottleneck buffer. This queue delay packet delivery for all flows sharing the same bottleneck. Delay-based TCP measures that delay and interprets it as a congestion signal, reducing *cwnd* in an attempt to reduce queuing delay. At the same time, the competing loss-based flows keep increasing *cwnd* to use the newly available bandwidth. *cwnd* continues to increase until packet loss is observed when the bottleneck buffer exhausts.

The problem worsens when buffer size is very large and bottleneck bandwidth is low. A large buffer size allows loss-based flows to push a large number of packets to the buffer, creating long queuing delays. On the other hand, low bandwidth makes queuing delay longer

[1]Some companies have already started producing high-end home gateways/ WiFi routers with modern AQM implemented such pfSense [21], Netgear, ubnt [171])

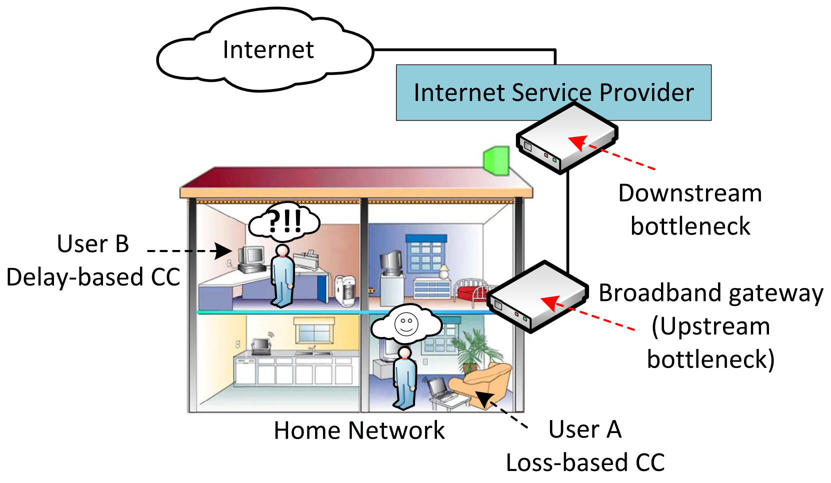


Figure 4.13: Typical home network with two users sharing a bottleneck. User A uses loss-based transport and User B uses delay-based transport

due to increased serialisation delay. As a consequence, the *cwnd* dynamic slows, resulting in loss-based flows backing off less frequently without giving delay-based flows an opportunity to recover. Moreover, delay-based flows will back off more frequently as the queuing delay increases quickly.

As an example, Figure 4.14 shows a throughput versus time plot for a TCP Vegas and a TCP CUBIC flow sharing a 10Mbps bottleneck bandwidth. The bottleneck emulates a 20*ms* RTT path with 200 packet buffer. We can see in this figure that when the CUBIC flow starts at t=10s, Vegas throughput drops significantly. *cwnd* decreases due to observing high delay (through RTT measurements of the sending packets) caused by the aggressiveness of the CUBIC flow. CUBIC keeps filling the buffer while Vegas gets no opportunity to increase its *cwnd*. The high buffer occupancy of the CUBIC flow causes the scheduler to serve more packets from the CUBIC flow than from the Vegas flow. As a consequence, the Vegas flow realises only 0.24Mbps bandwidth share while CUBIC achieves 9.75Mbps during the competition period between t=10s to t=30s.

Delay-based TCP cannot be deployed widely on the Internet without mechanisms allowing acceptable coexistence with loss-based flows. Some threshold delay based algorithms use higher delay thresholds (e.g. 100ms queuing delay in LEDBAT [8]) which allow delay-based flows to compete better with loss-based TCP. Although this can prevent the flows from starvation in some cases, it does not solve the problem, especially for large bottleneck buffer sizes. It can also create standing queues and introduce additional issues as described in Section 2.3.1.2.

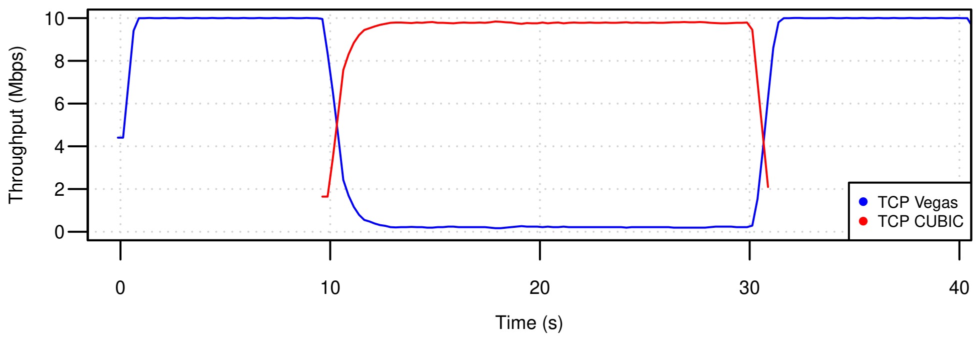


Figure 4.14: A TCP Vegas and a TCP CUBIC flows competing for 10Mbps bottleneck capacity

Dual mode algorithms (see Section 2.4.2) attempt to infer competing loss-based flows and explicitly switch between delay and loss based mode based on the detection. CDG falls into this category as it switches between two modes seamlessly when the presence of a loss-based flow is inferred. In addition, CDG uses a shadow window mechanism to further improve coexistence further.

In this section, we use experimental analysis and evaluation methodologies to understand the CDG coexistence mechanisms and discuss their issues.

4.3.1 Evaluating CDG coexistence with loss-based flows

We evaluate the performance of CDG coexistence with loss-based TCP by conducting experiments that consist of one CDG flow competing with one TCP CUBIC flow[2]. The bottleneck emulates a link with bandwidth B_{rate}=$\{1.5, 4, 12, 25\}$Mbps and RTT_{base}=$\{10, 40, 180, 240, 340\}$ms (see Section 3.7).

Bottleneck buffer size plays an important role in loss-based TCP performance. A BDP worth of buffer is required for standard TCP to achieve full bandwidth utilisation (see Section 2.3.3.2 for more details). Selecting bottleneck FIFO buffer size in evaluating CC algorithms is hard because different buffer sizes can change algorithm behaviour especially when it comes to coexistence between delay and loss- based flows.

Practically, different vendors configure their equipment with different buffer sizes based on Internet connection link speed. Typically, Internet Service Providers (ISPs) supply their customers with home gateways that provide highest posible TCP throughput. This allows users to upload and download files at maximum speed, and obtain high scores on Internet speed test services. It is not uncommon for home gateways to be shipped with buffer sizes equal to a

[2]Linux TCP CUBIC is used since it is the most widely deployed loss-based CC.

Table 4.2: Buffer size equivalent to 340ms delay and calculated based on bottleneck bandwidth

Bandwidth (Mbps)	BDP(KiB)	Buffer size (packet[3])
1.5	62.26	43
4	166	114
12	498	340
25	1037.6	709

BDP calculated based on RTT_{base} = 340ms or more [5]. This base RTT is selected because the average round trip time for distant international hosts is around 340ms. For example, the typical RTT between a server in Europe and Australia is 340ms [170].

For this reason, 340ms worth of buffer size is used to reflect this network configuration, calculated as $0.34 \times B_{rate}$. Note that some of the scenarios in this experiment are not realistic but are used for completeness. For instance, it is unlikely to have a fast connection (e.g 25Mbps) connecting to a far distance point. Table 4.2 shows the calculated buffer sizes used.

Since we are comparing throughput during the congestion avoidance phase, we patched CDG to use a loss-based slow-start, allowing CDG to quickly converge to full capacity usage. In this experiment, the CDG flow starts first with CUBIC starting after 10 seconds, giving the CDG flow enough time to stabilise before entering the competition period. The CUBIC flow finishes after 100 seconds. We measure throughput for the period between t=20s and t=100s to avoid any instability caused by CUBIC start-up and termination.

Figure 4.15 shows bottleneck capacity sharing between a CDG and CUBIC flows. Average throughput is calculated as the average throughput for each individual flow (CDG or CUBIC). This figure demonstrates that CDG cannot coexist with loss-based flows reasonably when bottleneck buffer sizes are large.

At 1.5Mbps CDG achieved between 10 - 21% of bottleneck capacity while CUBIC used 78% to 90% of the bandwidth. CDG performance decreased at 4Mbps where it achieved 5 - 9% bandwidth sharing with the lowest throughput at 320ms RTT_{base}. This is primarily due to large buffer sizes as calculated based on bottleneck bandwidth. Additionally, CUBIC performs better at larger path RTT due to its larger multiplicative decrease factor (0.7) and faster cubic *cwnd* growth.

As bandwidth increases, CDG cedes more bandwidth to CUBIC. The results show than CDG utilised between 3 - 4% of bottleneck bandwidth at 12Mbps and 1 - 2% at 25Mbps.

This experiment reveals that CDG does not starve completely when buffer size is larger than 60 packets and link speed is 1.5Mbps. However, at larger buffer sizes and higher bandwidth, CDG flows starve without any ability to compete with CUBIC flows.

[3]Supposing a packet is 500 bytes long

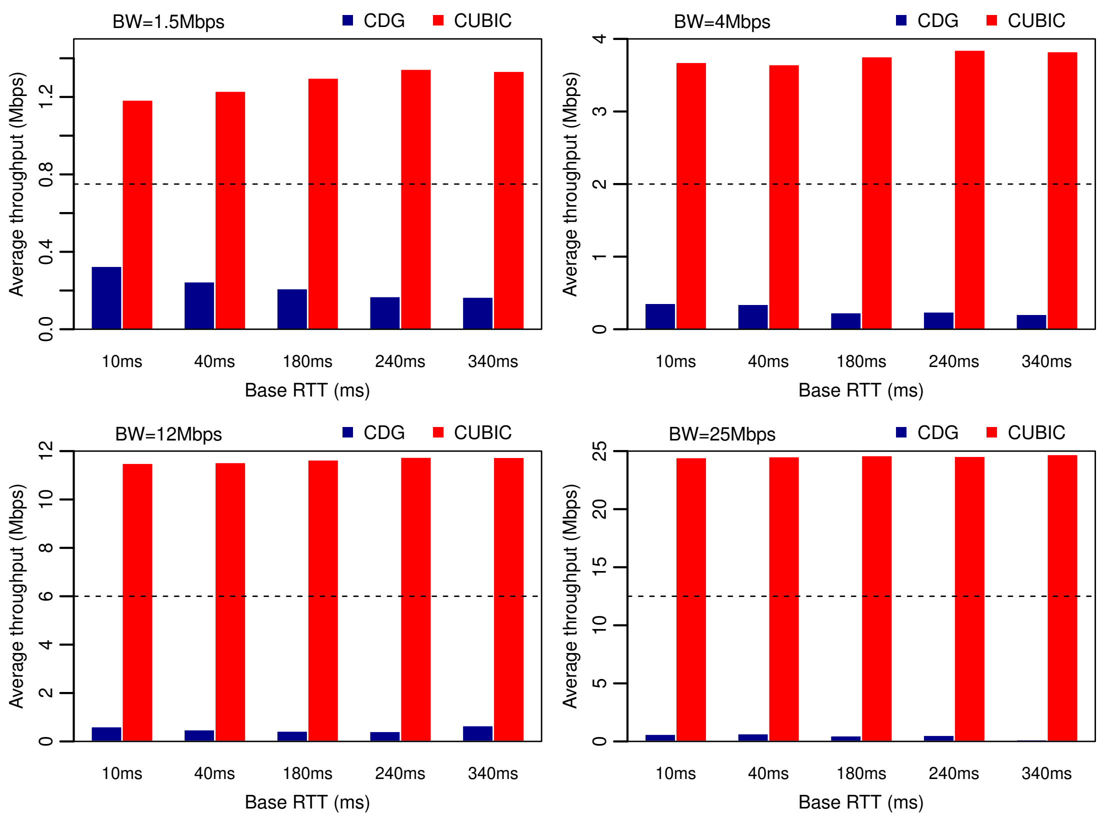


Figure 4.15: Bandwidth sharing between CDG and CUBIC flows competing for bottleneck capacity. Bottleneck has 340ms delay equivalent worth of buffer.

4.3.2 The impact of bottleneck buffer size on CDG coexistence

Since different buffer sizes lead to different capacity sharing between delay-based and loss-based flows, we conduct a simple experiment to understand the effect using different buffer sizes on CDG flow when competing with CUBIC.

In these experiments, we use a setup similar to the previous experiment which consists of one CDG flow competing with a CUBIC flow for bottleneck bandwidth. The bottleneck emulates links with B_{rate}={1.5, 4, 12, 25}Mbps and RTT_{base}={10, 40, 240}ms, and FIFO buffer sizes bs ={1, 2, 4, 6, 8}BDP. The CDG flow starts first with CUBIC starting after 10 seconds, giving CDG flow time to stabilise before entering the competition period. The CUBIC flow lasts for 100 seconds. Figure 4.16 shows the average throughput for each individual flow for this experiment.

We can see in this figure that when RTT_{base}=10ms and B_{rate}=1.5Mbps, CDG achieved a maximum of ~35% bandwidth utilisation when buffer size is 4 BDP. For buffer sizes of 2 and 8 BDP, CDG acheived ~20% throughput. CDG performed similarly for RTT_{base} of 40ms with a slightly better throughput with buffer sizes equal to 1 and 2 BDP. CDG coexistence performance decreases dramatically when RTT_{base} is 240ms due to larger buffer size allowing CUBIC to push more packets to the buffer. Longer queues make CDG back off more often as the delay gradient increases.

At 4Mbps and 10ms RTT, CDG performs better and was able to coexist fairly when *bs* is set to 2 and 4 BDPs, achieved 35% at 8 BDPs and 15% when bs is set to 1 and 12 BDPs. At 40ms RTT, CDG also realised fair share with CUBIC but only when *bs* is 1 BDP. Bandwidth utilisation then decreases as *bs* increases. Bandwidth utilisation dropped to 13% when buffer size is 8 BDP (~ 107 packets). At 240ms RTT, CDG coexistence performance collapsed for all buffer sizes. It achieved almost zero bandwidth utilisation at 8 BDP (~640 packets).

CDG performance at 12Mbps and 25Mbps is similar. For both bandwidths, CDG is able to compete when path RTT is 10ms with buffer sizes of 1,2 and 4. However, CUBIC outperforms CDG when *bs* is larger than 4 BDP. When RTT_{base} is 40ms, CDG starts collapsing gradually. CDG is completely ineffective when RTT_{base} is 240ms due to large buffer sizes.

This experiment shows that CDG can coexist reasonably with loss-based flows when buffer size is between 6 and 60 packets in many scenarios. However, it is unable to achieve acceptable capacity sharing otherwise. The experiment also reveals that CDG in general collapses when buffer size increases above 160 packets. Table 4.3 summarises the CDG bandwidth utilisation fraction for each bandwidth setting against the buffer size in packets.

4.3.3 Underlying cause for CDG poor coexistence

In this section we explore the CDG coexistence mechanisms to identify the issues that prevent CDG from coexisting well with loss-based CC. We take case studies from the experiments in Section 4.3.2 to investigate the problem. First we start with the ineffectual back-off mechanisms since it is the main mechanism that allows CDG to coexist with loss-based CC. Then, we explore issues with the shadow window mechanism.

4.3.3.1 Ineffectual back-off mechanism

Switching to loss mode (no delay gradient probabilistic back-off) when detecting of competition with loss-based flows is an efficient technique to remedy the delay-based coexistence problem. However, detection of competing loss-based flows is a challenging problem. A harder challenge is to detect when loss-based flows have terminated.

We take a single experiment instance from the experiment in Section 4.3.2 to understand the issues related to the ineffectual back-off mechanism. The selected instance consists of a CDG flow competing with a CUBIC flow over a 12Mbps bottleneck bandwidth with 40ms

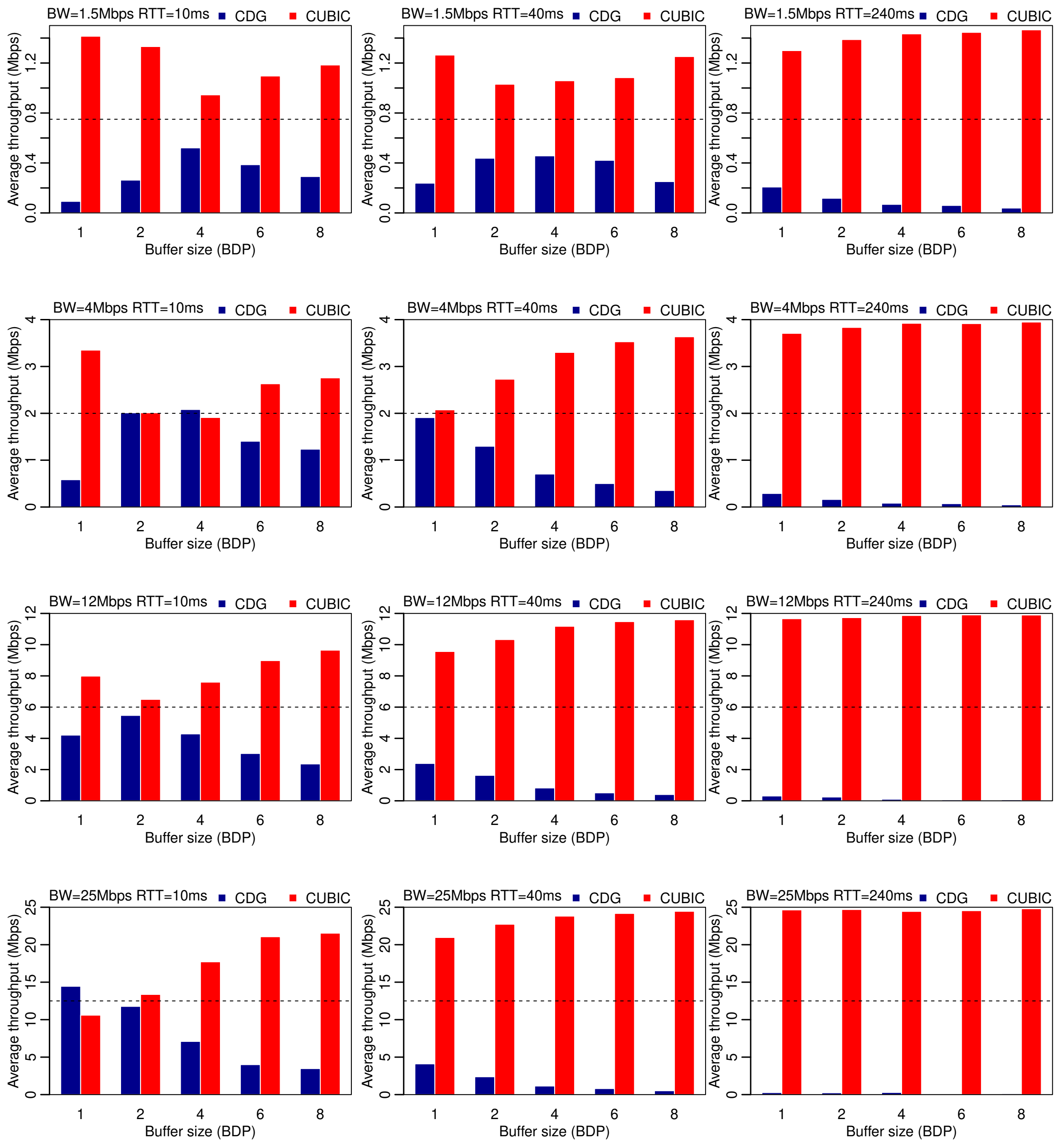


Figure 4.16: CDG and CUBIC throughput achieved when competing over bottlenecks of different bandwidths, buffer sizes, and RTT.

Table 4.3: Bandwidth fraction for a CDG flow competing with CUBIC flow for bottleneck bandwidth with different buffer sizes in packets.

1.5Mbps		
RTT_{base}(ms)	bs (pkt)	B/W fraction
5	2	0.062
5	3	0.175
5	6	0.347
5	8	0.258
5	10	0.194
20	6	0.159
20	10	0.292
20	20	0.305
20	30	0.280
20	40	0.167
120	30	0.138
120	60	0.083
120	120	0.046
120	180	0.04
120	240	0.026

4Mbps		
RTT_{base}(ms)	bs (pkt)	B/W fraction
5	4	0.146
5	7	0.503
5	14	0.520
5	20	0.351
5	27	0.307
20	14	0.477
20	27	0.324
20	54	0.176
20	80	0.125
20	107	0.088
120	80	0.072
120	160	0.040
120	320	0.021
120	680	0.018
120	640	0.012

12Mbps		
RTT_{base}(ms)	bs (pkt)	B/W fraction
5	10	0.351
5	20	0.456
5	40	0.357
5	60	0.253
5	80	0.197
20	40	0.199
20	80	0.136
20	160	0.069
20	240	0.042
20	320	0.034
120	240	0.026
120	480	0.017
120	960	0.007
120	1440	0.005
120	1920	0.006

25Mbps		
RTT_{base}(ms)	bs (pkt)	B/W fraction
5	21	0.578
5	42	0.471
5	84	0.284
5	126	0.160
5	167	0.139
20	84	0.164
20	167	0.095
20	334	0.046
20	500	0.032
20	667	0.021
120	500	0.011
120	1000	0.01
120	2000	0.008
120	3000	0.004
120	4000	0.005

RTT_{base} and 2 BDP (80 packets) buffer size. The test duration is 150 second and the competition period is from t=10s to t=110s. CDG achieved only 13.6% of the bottleneck capacity during that period, and operated in delay mode for 93.2 seconds and in loss mode for only 3.6 seconds. CDG operated in loss mode for only 3.6% of the competition period.

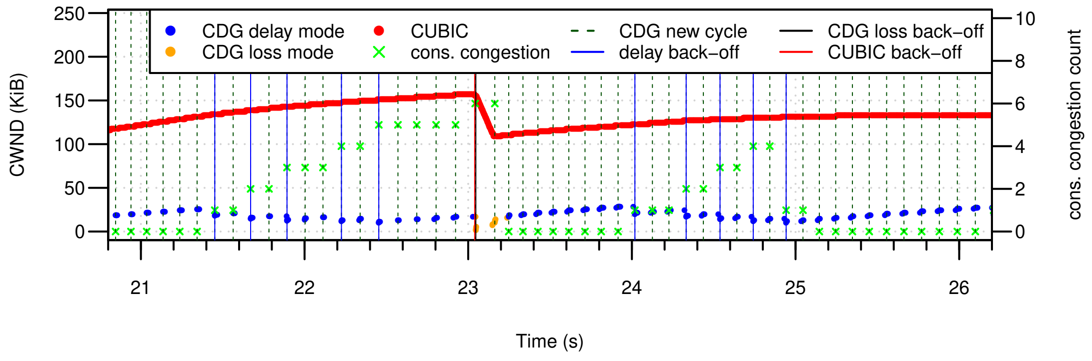
As previously mentioned, CDG detects competing loss-based flows when it experiences a number of consecutive congestion events (default 5) while not observing a decrease in latency (see section 2.7.4.1 for more details). It then moves to loss mode for a few (default 5) measurement cycles where the delay congestion feedback signal is ignored. CDG exits loss mode whenever a negative smoothed gradient is seen or the period elapses. There are some problems regarding this strategy that leads to poor coexistence performance.

First, when CDG switches back to delay mode, it cedes a large amount of bandwidth to competing flows. After the loss mode period elapses, CDG observes large positive delay gradients as a result of the aggressiveness of loss based flows and CDG loss mode capacity probing strategies. Additionally, the moving average window will include many positive gradient samples that are collected during loss mode. Therefore, CDG will observe a large $\bar{g_n}$, increasing the back-off probability, resulting in an immediate back-off. Since the loss-based flow keeps pushing packets to the bottleneck, there is an increased chance of consecutive back-off events, further decreasing *cwnd*.

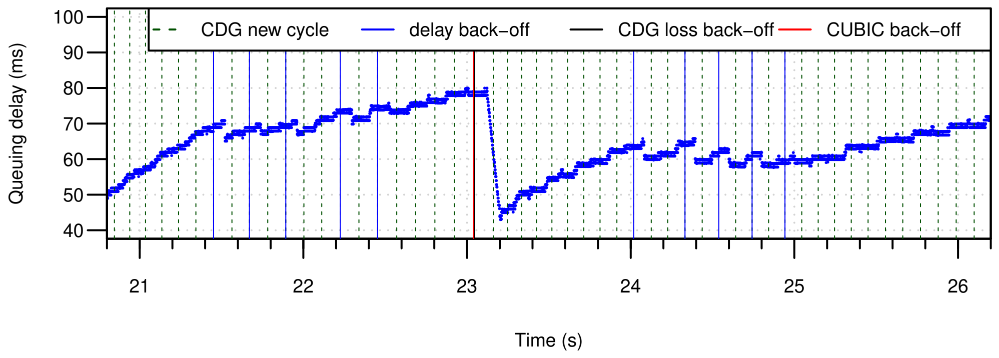
The second issue with the ineffectual back-off mechanism is that CDG rarely enters loss mode due to observing negative gradients during competition. CDG back-off events reduce bottleneck queue length even when competing with loss-based flows. Whether it is probabilistic delay or packet loss driven back off, the number of packets arriving at the bottleneck will reduce. This leads to a reduction in queuing delay, causing sending hosts to observe a decrease in the measured RTT, resulting in negative delay gradients. In addition, if a competing flow backs off its *cwnd*, queuing delay will also reduce and produce negative gradients. In either case, CDG will reset its consecutive congestion counter, preventing CDG from entering loss mode.

The final issue identified is that in many cases, CDG exits loss mode shortly after entering it due to buffer overflow caused by CDG or competing flows. There is a higher probability of packet loss when CDG switches to loss mode due to an increase in the number packets arriving the bottleneck. Whether CDG or a competing flow lose packets, the queuing delay will reduce due to *cwnd* decrease. CDG will then reset its consecutive congestion counter and exit loss mode upon discovering negative delay gradients.

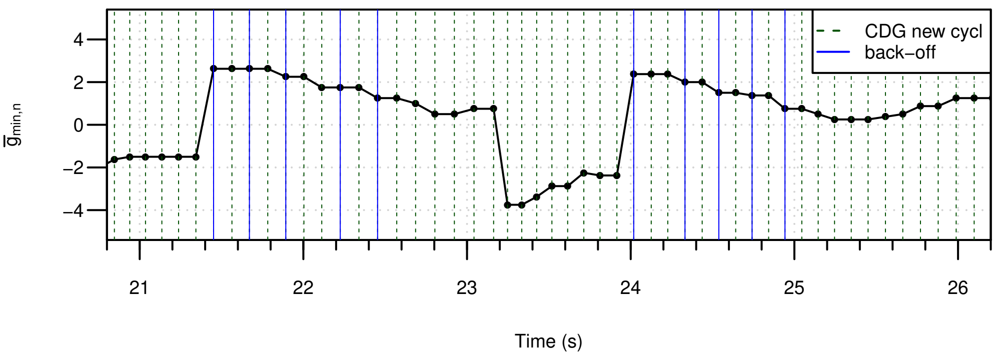
Delay signal noise, a well-know phenomenon that the delay signal suffers from [56], could also result in RTT fluctuation resulting in negative gradients. As a result, CDG may reset the consecutive congestion counter, causing CDG to exit or not enter loss mode.



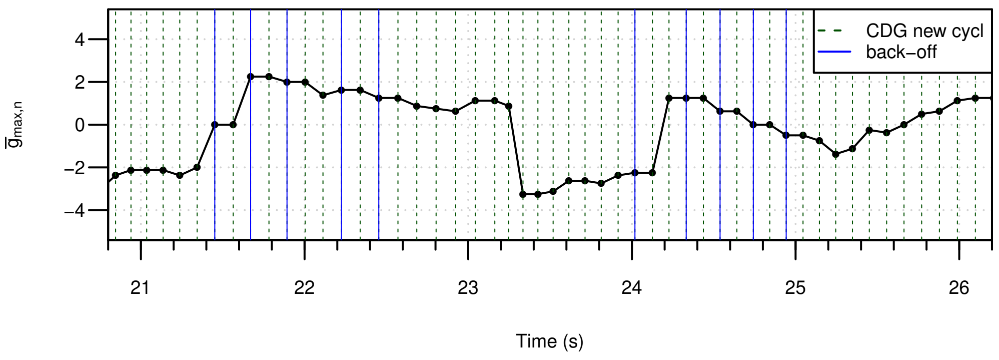
(a) *cwnd* and the consecutive congestion counter trajectory



(b) Bottleneck queue occupancy trajectory



(c) $\bar{g}_{min,n}$ trajectory



(d) $\bar{g}_{max,n}$ trajectory

Figure 4.17: CDG exits the loss mode quickly due to experiencing a packet loss

To see the ineffectual back-off mechanism effectiveness in the selected test, we plot a zoomed-in *cwnd* trajectory for both CDG and CUBIC flows along with the consecutive congestion counter, $\bar{g}_{min,n}$ and $\bar{g}_{max,n}$ for the period between t=21s and t=26s of the test. Additionally, we plot the queuing delay trajectory for the same period measured at the bottleneck[4]. The plots are shown in Figure 4.17. We can see in Sub-figure 4.17a CDG backs off *cwnd* t=21.45s due to an increase in the queuing delay (shown in Figure 4.17b). Also, the consecutive congestion counter increases by one. This congestion event reduces the queuing delay (shown in figure 4.17b) due to *cwnd* decay after this back-off. However, the smooth gradients (shown in figure 4.17c and 4.17d) does not become negative due to the small queuing delay reduction and the property of the smoothing window.

The consecutive congestion counter keep increasing and CDG enters loss mode at t=23.07s. Unfortunately, CDG switches back to delay mode after just 200ms due to packet loss for both the CDG and CUBIC flows. This event significantly reduces the queuing delay, resulting in the gradient to drop below zero, resetting the consecutive congestion counter. The reason for the packet drops is that CDG and CUBIC experience packet loss shortly after CDG switching to loss mode. CDG and CUBIC backing off reduces the queuing delay as shown in Figure 4.17b. Since all flows sharing the same bottleneck experience similar RTT decrease, CDG observes negative gradients and exits loss mode.

At t=24.03 CDG backed off again due to an increase in the queuing delay and the consecutive congestion counter starts increasing until it reaches four at t=24.75s. Then, CDG delay-based back-off drives $\bar{g}_{max,n}$ to a negative value so the consecutive congestion counter is reset as a reflection of observing negative delay gradients. This situation occurs many times during the experiment (not shown in the figure).

This is due to the CUBIC *cwnd* growth function reduces the *cwnd* increasing step when it approaches the maximum *cwnd* captured when packet loss occurs [49]. The flat plateau results in trivial queuing delay change, causing RTT variation to be driven entirely by CDG *cwnd* growth. As a result, CDG assumes that there are no competing loss-based flows and backs off to reduce latency. We also note that high RTT (due to both path RTT and queuing delays) slows the *cwnd* dynamic, contributing to this behaviour.

Figure 4.18 shows CDF plots for the CDG consecutive congestion counter when competing with CUBIC for bottleneck rate B_{rate}=$\{1.5, 4, 12, 25\}$Mbps over emulated path RTT_{base}=$\{10, 40, 240\}$ms and bottleneck FIFO buffer size bs =$\{1, 2, 4, 8, 16\}$ BDP. This figure shows that CDG rarely enters loss mode at 10ms path RTTs. At 25Mbps and 16 BDP buffer size, CDG enters loss mode the most but switches back to delay mode quickly after one or two RTTs.

The ineffectual back-off mechanism performs better at 40ms path RTT. When B_{rate} $\geq$12Mbps

[4]A queuing delay sample is measured when a packet leaves the bottleneck similar to CoDel AQM [47].

and buffer size > 4 BDPs, CDG is able to switch to loss mode more frequently and stay in this mode for longer.

At 240ms path RTT, the ineffectual back-off mechanism performs the best compared with the other test with lower path RTTs. However, CDG sill exits from loss mode quickly due to observing negative gradients before reaching the exit threshold.

In a nutshell, the ineffectual back-off mechanism does improve the coexistence with loss-based flows. However, the improvement is not significant and has limited impact on the coexistence especially in small bottleneck buffer sizes.

4.3.3.2 The shadow window mechanism

The shadow window mechanism, described in Section 2.7.4.2, aims to improve CDiG coexistence by potentially recovering *cwnd* after multiple delay-based back-offs when competing with loss-based flows. Since the shadow window s increases regardless of delay-based back-offs, its size can be larger than the current congestion window size. CDG sets *ssthresh* to the maximum of current *cwnd* and s when detecting congestion related losses. If s is larger than *cwnd*, *cwnd* will increase exponentially similar to the slow-start phase because *cwnd* is smaller than *ssthresh*. Therefore, CDG flow will obtain a higher bandwidth due to using a larger *cwnd*.

There are some potential issues with the shadow window mechanism. 1) if CDG wrongly infers the buffer is full when observing packet losses, the shadow window will not be used. In this case s will continuously increase until a packet loss with full queue are detected or inferring an empty queue. 2) if the buffer wrongly is inferred empty, CDG will reset the shadow window making s smaller than the mimic (loss-based) window. 3) When the shadow window is used, there is a high probability for packet losses to occur since the exponential *cwnd* increase causes a burst of packets to arrive at a saturated bottleneck and activating loss-based back-off. In addition, a fast *cwnd* increase increases the queuing delay activating multiple back-offs until the latency stabilises. This means CDG will continue observing large positive gradients and backs off quickly, resulting in *cwnd* decreasing to a small value again.

We use the same case study in Section 4.3.3.1 to understand the issues involved with the shadow window mechanism. We calculate the number of loss events and the number of times that CDG used the shadow window during the test. Unfortunately, we found that CDG used the shadow window only 4 times out of 40 loss events. We also found that 35 events out of the 40 loss events, CDG wrongly inferred queue state was not full considering the losses to be non-congestion related. On 29 events out of 35 loss events the shadow window was larger than *cwnd*. We also found that only 7 events out of the 40 loss events, the shadow window was smaller than *cwnd*. These results reveal that CDG wrongly ignored the shadow window

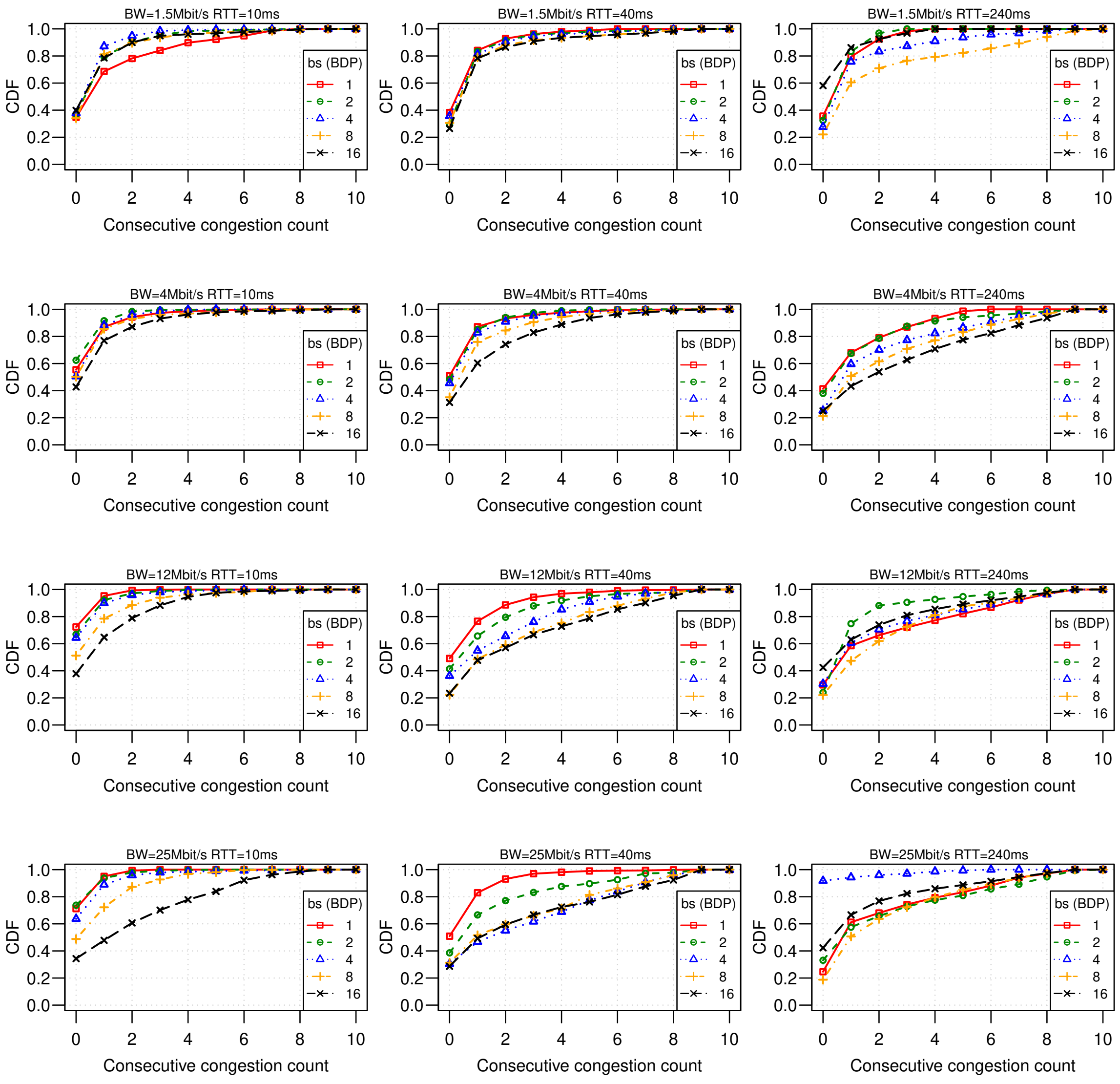


Figure 4.18: CDF for CDG consecutive congestion counter when coexisting with CUBIC flow over different bandwidth, path RTT and bottleneck buffer sizes

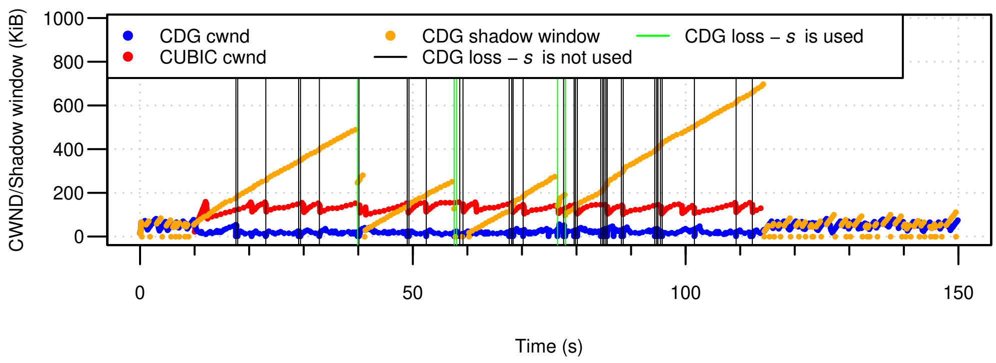


Figure 4.19: CDG shadow window trajectory with *cwnd* trajectory for CDG and CUBIC flows

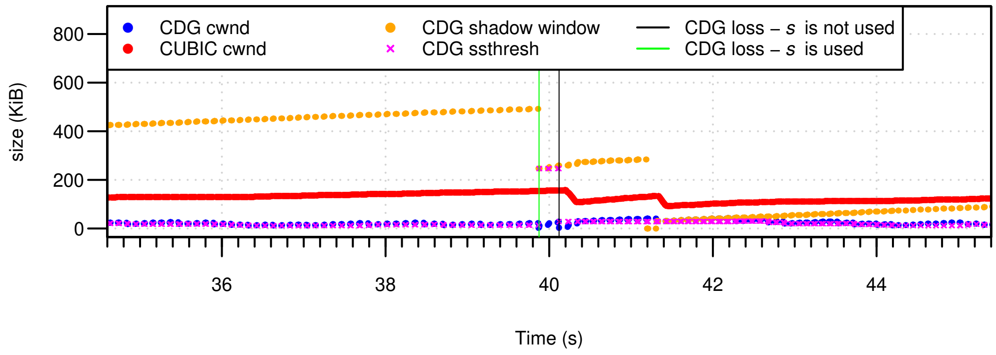


Figure 4.20: Zoomed-in CDG shadow window and *ssthresh* trajectories with *cwnd* trajectory for CDG and CUBIC flows

in 72% of the loss events while it should use it. That happened because queue state inference was wrong 87% of the time. We did not emulate random packet losses in the bottleneck and, therefore, all the losses were congestion related losses.

Additionally, CDG inferred an empty queue for 4 times during the competition period which is not correct. This resetting makes the shadow window much smaller than *cwnd* of the competing loss-based flow.

Figure 4.19 shows the shadow window trajectory along with congestion window of CDG and CUBIC flows. We can see in this figure the shadow window is oscillating between zero and *cwnd* when only CDG flow is utilising the bottleneck bandwidth. That happens because the queue builds up and drains periodically due to CDG *cwnd* increase and decrease as well as resetting the shadow window when an empty queue is detected.

As soon as CUBIC joins the competition, the shadow window starts increasing continuously without reaction to packet losses (due to incorrect queue state inference) until t=40s. Figure 4.20 shows a zoomed-in *cwnd*, *ssthresh* and shadow window trajectories for the period between t=35s and t=45s. We can see in this figure, packet drop occurs at t=39.84s and CDG infers a full queue. CDG sets *ssthresh* to the shadow window since the shadow window is larger than *cwnd*. However, another packet loss quickly occurred after two RTTs while CDG inferred the queue was not full. Since CDG does not reduce *cwnd* when loss is non-congestion related, it sets *ssthresh* to the current *cwnd* to preserve congestion window size after loss recovery [5]. The second loss event causes *ssthresh* to be set to the current *cwnd* because CDG was in congestion recovery[6]. This led CDG to leave congestion recovery and enter CA, preventing *cwnd* from reaching the shadow window that was previously set to *ssthresh*.

At t=41.2s, CDG incorrectly inferred an empty queue and reset the shadow window. After that, the shadow window increased as normal but remains lower than the competing CUBIC flow.

In this experiment, CDG did not get a significant benefit from the shadow window mechanism due to the difficulties faced during the experiment. The main issue that negatively impacts the performance of the shadow window mechanism is CDG queue state inference. Our experiment shows that queue state inference does not work well in many scenarios. However, we do not explore the reason for the false negative full queue and false positive empty queue states in this thesis. This needs further study to understand how delay signal sampling, the moving average window and signal noise can affect queue state detection. We leave this investigation for future work.

4.4 CDG Slow-Start Evaluation

Optimally, the slow-start phase should finish as soon as the congestion window reaches the path BDP. This allows flows to converge quickly to utilise available bandwidth without introducing high latency and packet drops. It also permits CA phase to start with high throughput from the beginning. As *cwnd* growth is typically slow in CA phase, exiting slow-start too early costs the throughput, especially in high path RTTs. On the other hand, if the slow-start finishes too late, it creates a long queue in the bottleneck leading to packet loss. Standard TCP relies on the loss signal to exit the slow-start phase which causes high queuing delay and heavy packet loss in large buffer sizes.

It is clear that the side-effects of Standard TCP SS bandwidth probing are not in-line with CDG goals. Therefore, CDG uses the delay gradient signal to detect congestion in the slow-start as an attempt to find a proper exit point from slow-start phase before a long queue builds up. This allows maintaining low latency and preventing packet loss.

[5]CDG FreeBSD implementation uses this method to keep the *cwnd* value unchanged since FreeBSD TCP stack set *cwnd* to one MSS and uses slow-start like *cwnd* growth during loss recovery.

[6]We found that there is a bug in CDG implementation causing *ssthresh* to always set to the current *cwnd* when packet loss occurs while CDG is in congestion recovery.

Table 4.4: Ratio of *ssthresh* to the path BDP when CDG exiting slow-start phase

		RTT_{base}(ms)				
		10	**40**	**180**	**240**	**340**
B_{rate}(Mbps)	**1.5**	813%	203%	45%	34%	24%
	4	305%	76%	17%	13%	9%
	12	333%	25%	13%	22%	3%
	25	238%	60%	9%	16%	8%
	50	119%	9%	2%	4%	1%

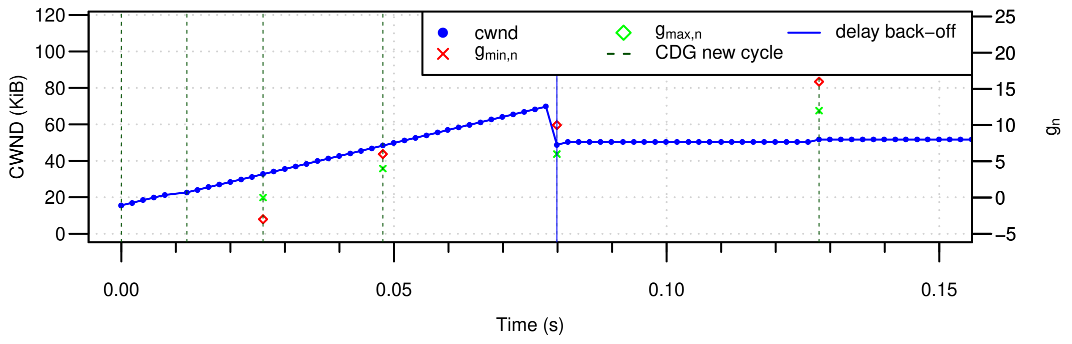
As mentioned in Section 2.7.6, CDG uses the probabilistic back-off function (Equation 2.27) to make a decision to exit from slow-start. It leverages NewReno slow-start *cwnd* growth (i.e. increases *cwnd* by double the acknowledged bytes for each received ACK). CDG *ssthresh* is set to $cwnd.\beta_{delay}$ if congestion is detected using the delay gradient signal, and $cwnd.\beta_{loss}$ if congestion is detected using loss signals.

To explore CDG SS performance in simple scenarios, we conduct a simple experiment consisting of a single CDG flow traversing a bottleneck that emulates links with B_{rate}={1.5, 4, 12, 25, 50}Mbps and RTT_{base}={10, 40, 180, 240, 340}ms. Bottleneck FIFO buffer size is set to 2000 packets and the Droptail mechanism is used. This large buffer size is selected to prevent packet loss since we explore the performance of the delay gradient congestion signal during slow-start.

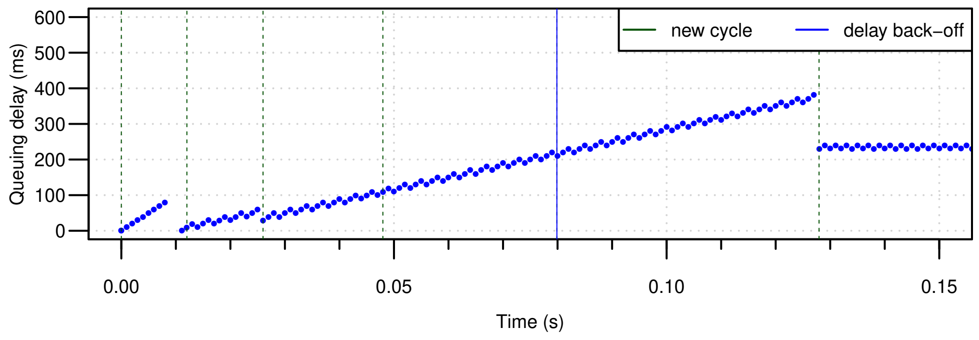
Table 4.4 shows the percentage of *ssthresh* to the path BDP $(ssthresh/BDP) \times 100$ for each test. We extract *ssthresh* value just after CDG switches from SS to CA phase. This table reveals that CDG leaves SS phase late when the path BDP is small, while exiting SS too early in large BDP cases. For example, when B_{rate}=1.5Mbps and RTT_{base}=10ms (BDP ~1.83KiB), *ssthresh* is about 8 times larger than BDP. On the other hand, *ssthresh* is set to only 1% of the path BDP when B_{rate}=50Mbps and RTT_{base}=340ms (BDP ~2MiB).

In small BDP scenarios, *cwnd* reaches BDP in a very small number of RTTs. Therefore, the number of coin tosses to back off and exit SS is small before reaching BDP since CDG makes back-off decisions once per RTT. As a result, *cwnd* grows large causing a high queuing delay before moving to CA phase. We take B_{rate}=12Mbps and RTT_{base}=10ms (BDP ~14.6KiB) scenario as an example to see this behaviour in action. Figure 4.21 shows *cwnd*, $g_{min,n}$ and $g_{max,n}$ trajectories, and queuing delay progression for the interval between t=0 and t=0.15s. The unsmoothed gradient is shown because CDG FreeBSD implementation relies on this measurement in addition to smooth gradient during SS phase (see Section 2.7.7).

In Figure 4.21a, we can see at t=0s *cwnd* is initiated with 10 packet (IW) which is already



(a) *cwnd*, $g_{min,n}$ and $g_{max,n}$ trajectories



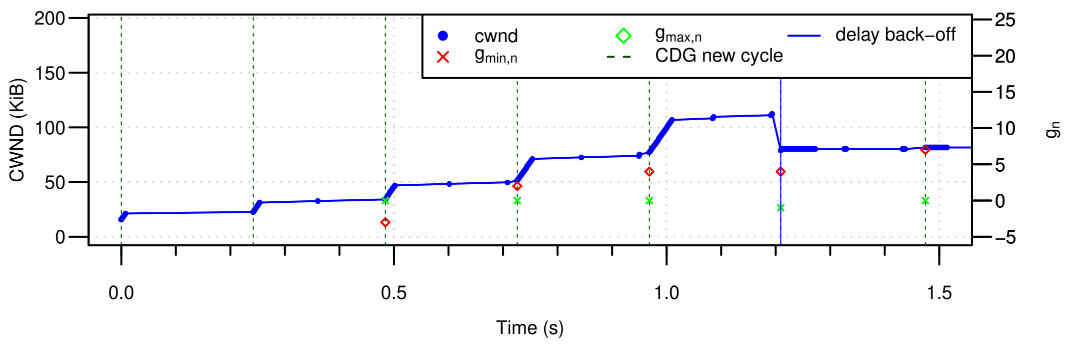
(b) Bottleneck queue occupancy trajectory

Figure 4.21: CDG leaves SS phase late when BDP is small. B_{rate} = 12Mbps and RTT_{base} = 10ms.

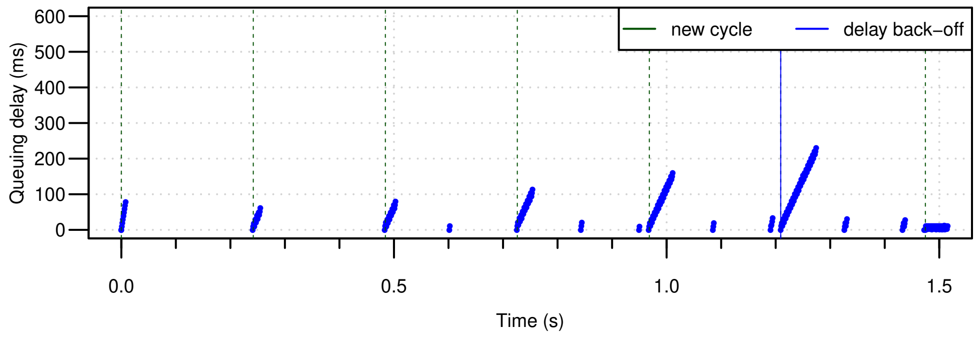
larger than BDP. This results in TCP sending 10 back-to-back packets to the router. This packet burst increases the queuing delay as shown in Figure 4.21b. After one RTT at t=0.012s, the queue drains since one RTT is needed to send all packets in the queue.

We can see in the second RTT, the queuing delay increases slower than the previous RTT as ACK packets arrive at the sender spaced in time due to bottleneck traffic shaping, resulting in *cwnd* growth being less aggressive. The lower queuing delay causes $g_{max,n}$ to become negative. $g_{min,n}$ indicates an unchanged gradient (zero) due to queue draining causing $RTT_{min,n}$ to be the same as in previous RTT. Then, $g_{min,n}$ and $g_{max,n}$ increase as the queue grows. After four RTTs, the probabilistic back-off triggers back-off due to observing high delay gradients ($g_{min,n}$ and $g_{max,n}$) and as the randomly generated number (see equation 2.28). The slow start phase finishes with *ssthersh* set to 48.8KiB (~333% of BDP).

On the other hand, CDG needs many RTTs for *cwnd* to reach BDP in a large BDP scenarios. Therefore, the number of CDG coin tosses to back off is large before exceeding BDP. That means there is a higher chance for CDG to exit SS before *cwnd* reaches the path BDP. Additionally, since TCP SS involves transmitting many packet bursts due to exponential *cwnd* growth, $g_{max,n}$ will show large positive delay gradients which leads to back-off. We take B_{rate}=12Mbps and RTT_{base}=240ms (BDP ~351.6KiB) scenario as an example to discuss this



(a) *cwnd* and the consecutive congestion counter trajectory



(b) Bottleneck queue occupancy trajectory

Figure 4.22: CDG leaves SS phase early when BDP is large. B_{rate} = 12Mbps and RTT_{base} = 180ms.

behaviour. Figure 4.22 shows *cwnd*, $g_{min,n}$ and $g_{max,n}$ trajectories, and queuing delay progression for the interval between t=0 and t=1.5s.

In Figure 4.22a, we can see at t=0s *cwnd* is initiated with ten packets which allows TCP to send a burst of ten packets to the bottleneck. This packet burst rises the queuing delay as shown in Figure 4.22b. After a few milliseconds, the queue drains and there is a stall until next RTT at t=0.24s where ACK packets return to the sender. We can see at that time, the queuing delay does not increase significantly compared with the previous burst even though *cwnd* is doubled. This occurs because the ACK packets are spaced due to bottleneck traffic shaping. As a consequence, $g_{max,n}$ is negative and $g_{min,n}$ is zero at t=0.5s. As *cwnd* increases, $g_{max,n}$ increases due to higher delay spikes. However, $g_{min,n}$ stays close to zero since the queue drains at least once ever RTT. High $g_{max,n}$ values actuates back-off which happens at t=1.42s while *cwnd* is much smaller than the path BDP. The slow start phase finishes with *ssthresh* set to 80.65KiB (~22% of BDP).

CDG SS is able to achieve low queuing delay but underestimates the available bandwidth when the path BDP is large (> 40KiB), while it overestimates the available capacity when path BDP is small.

4.5 Conclusions

In this chapter, we evaluated the CAIA-Delay Gradient (CDG) congestion control algorithm. We used experimental methodology to explore CDG flow behaviours in many simple scenarios that mimic typical home network internet connections.

Firstly, we explored CDG in a single flow scenario in which CDG operates without competing flows. We found that CDG performs unnecessary back-offs in many situations leading to low link utilisation especially in high RTT links. These unnecessary back-offs can be due to consecutive back-off events or spurious back-off. Consecutive back-off happens due to both using average moving window to smooth the delay gradient signals, and due to skipping next RTT measurement after backing off *cwnd*. Spurious back-off occurs due to bursty TCP sending and delayed ACK mechanism that leads to bottleneck queue fluctuation. This fluctuation causes CDG to observe many positive delay gradient samples leading CDG to back off without actual congestion occurrence.

On the other hand, we showed that CDG occasionally delays the back-off decision, leading to RTT spikes. The reason for the late back-off decision is the nature of the random probabilistic function that CDG uses. Additionally, the delay signal noise and asynchronous timing between measurement cycles and the bottleneck packet scheduler cause the smoothed gradient to increase slowly. Therefore, CDG does not back off until the gradient becomes high.

Secondly, we explore CDG coexistence with loss-based flows, namely TCP CUBIC. We found that CDG is able to obtain acceptable bandwidth sharing only when the bottleneck has a small buffer. As buffer size increases, CDG realises very low capacity sharing. We investigated the issues with CDG coexistence mechanisms and we found that the ineffectual back-off mechanism has limited impact on coexistence performance. On the other hand, CDG did not get a significant benefit from the shadow window mechanism due to difficulties facing this mechanism. These difficulties include false negative queue-full state and false positive queue-empty inference. This leads CDG to reset the shadow window periodically, and to ignore using the shadow window when packet loss is detected.

Finally, we evaluated the CDG slow-start algorithm and we found that CDG SS underestimates the available bandwidth when the path BDP is large (> 40KiB), while it overestimates the available capacity when the path BDP is small. Underestimating the available bandwidth reduces protocol throughput while overestimating the bandwidth results in high queuing delay.

In the next chapter, we will propose a hybrid congestion algorithm that extends and improves CDG. The new algorithm aims to solve the identified CDG issues to provide high throughput, lower queuing delay and better coexistence with loss based flows.

Chapter 5

Hybrid Loss-Delay Gradient Congestion Control

5.1 Introduction

In chapter 4, we evaluated CDG under a wide range of scenarios focussing on different aspects of the algorithm, and we identified a number of failings with CDG. In this chapter, we propose an enhanced congestion control algorithm, called Hybrid Loss-Delay Gradient (HLDG), which overcomes various CDG issues to provide better performance. HLDG can achieve high throughput, low queuing delay, good fairness and good coexistence with loss-based CC flows.

Similar to CDG, HLDG utilises the delay gradient signal to infer early network congestion. It solves the CDG unnecessary and spurious back-off problems (see Section 4.2.2) in which *cwnd* decreases while it is below the path BDP. Additionally, HLDG decreases *cwnd* to a fraction of estimated path BDP to remedy the early and large back-off effect, improving flow throughput.

HLDG uses explicit operation mode switching to improve coexistence with loss-based flows. As soon as it detects the presence of a competing loss-based flow, HLDG switches to loss mode where congestion is inferred using packet loss events to provide better coexistence with competing loss-based flows. When HLDG infers there are no competing loss-based flows, it switches back to delay-based mode to maintain low queuing delay.

HLDG uses estimated BDP and delay gradient signal during the slow-start phase to move to the congestion avoidance with both high throughput and low queuing delay. Further, HLDG improves protocol throughput in lossy environments by relying on estimated BDP.

Additionally, HLDG can be used as a low priority congestion control to provide scavenger class services. In LPCC mode, HLDG never switches to loss mode, operating in delay mode all the time. In this way, it obtains lower bandwidth share when competing with other flows. Our HLDG implementation in FreeBSD allows users to select the mode of operation (LPCC or conventional CC) through `sysctl` variables.

This rest of chapter is organised as follows. Section 5.2 describes HLDG improvements and evaluation in single flow scenarios. Section 5.3 presents HLDG coexistence enhancements and evaluation when competing with CUBIC flows. Section 5.4 presents an improved slow-start algorithm used by HLDG. In Section 5.5, we apply TCP-Westwood like loss tolerance technique to improve HLDG in a loosy environment, and evaluate HLDG in different scenarios. We conclude this chapter in Section 5.6.

5.2 HLDG in single-flow scenarios

As presented in Section 4.2, CDG does not fully utilise the available link bandwidth in large BDP path, and when the number of competing flows are small. This is due to large *cwnd* back-off and slow recovery, consecutive back-offs and spurious back-offs. In this section we propose solutions to address these problems, and validate the functionality of these solutions.

5.2.1 Preventing consecutive delay-based back-offs

In Section 4.2.2.1 we identified that the main causes for consecutive back-offs was the application of the smoothing window to gradient samples and skipping a delay gradient sample after *cwnd* reduces upon back-off.

Although reducing the smoothing window size can improve protocol response to congestion as well as prevent some unnecessary back-offs, it results in the gradient filter (averaging) becoming less effective. This makes the delay signal more vulnerable to signal noise, leading CDG to back off more frequently in real world situations.

This can be demonstrated by conducting the same experiment from Section 4.2.2 using a smaller CDG smoothing window size of 4 samples. The experiment consists of a single 300 second CDG flow generated using `iperf` tool. The bottleneck is configured to emulate B_{rate}=10Mbps, RTT_{base}=100ms (path BDP = 122KiB) with a 1000 packet buffer size. In this experiment, link utilisation was approximately 70%.

Figure 5.1 shows the probability of CDG backing-off again in the N^{th} RTT following the first back-off while *cwnd* is below BDP. This illustrates that by reducing the smoothing window to 4 samples (the blue curve), consecutive back-off slightly reduces comparing with Figure 5.1 (the red curve, smoothing window of 8 samples). However, reducing the smoothing window causes CDG to back off slightly more frequently (145 times comparing to 140 with an 8 sample window) causing the flows link utilisation to further reduce to 70% (compared to 73% using an 8 sample window). Similar results were seen with different bandwidths and RTT_{base}.

Skipping a delay gradient sample after *cwnd* backs off without skipping the corresponding RTT measurements has a more significant impact on the number of consecutive back-offs. This results in CDG missing an important $g_{min,n}$ sample that can prevent a further unnecessary back-off. As mentioned in Section 2.7, CDG obtains $RTT_{min,n}$ and $RTT_{max,n}$ measurements over an RTT interval using minimum and maximum filters. The CC algorithm can measure the effect of *a cwnd* reduction after exactly one RTT interval. Typically, the last sample in the measurement window will include a delay sample that reflects the reduction in *cwnd*. The smaller RTT sample does not affect the $RTT_{max,n}$ value for this interval, but could change $RTT_{min,n}$.

Assume that CDG backs off *cwnd* at cycle b when actual network congestion occurs. Based on the above hypothesis, $g_{max,b+1}$ will most likely be non-negative and $g_{min,b+1}$will be negative. Also, $g_{max,b+2}$ most likely will be negative and $g_{min,b+2}$will be non-negative. This behaviour has been shown experimentally in Section 4.2.2.1. To better reflect the effect of *cwnd* back-off, $g_{min,n}$ and $g_{max,n}$ should be calculated in a different manner following a back-off.

CDG calculates $g_{min,n}$ and $g_{max,n}$ as per equations 2.24 and 2.25, regardless of whether previous measurements have been skipped or not [1].

HLDG modification

HLDG differentiates between gradient calculation during normal operation and after back-off to prevent consecutive delay-based back-offs.

We propose to differentiate between gradient calculation during normal operation and after back-off to prevent consecutive delay-based back-offs. Specifically, we propose to calculate $g_{min,n}$ and $g_{max,n}$ according to equations 5.1 and 5.2 respectively.

$$g_{min,n} = \begin{cases} RTT_{min,n} - RTT_{min,n-2} & \text{backoff at interval (n-2)} \\ RTT_{min,n} - RTT_{min,n-1} & \text{otherwise} \end{cases} \tag{5.1}$$

$$g_{max,n} = \begin{cases} RTT_{max,n} - RTT_{max,n-2} & \text{backoff at interval (n-2)} \\ RTT_{max,n} - RTT_{max,n-1} & \text{otherwise} \end{cases} \tag{5.2}$$

[1]Based on the CDG FreeBSD implementation [145]

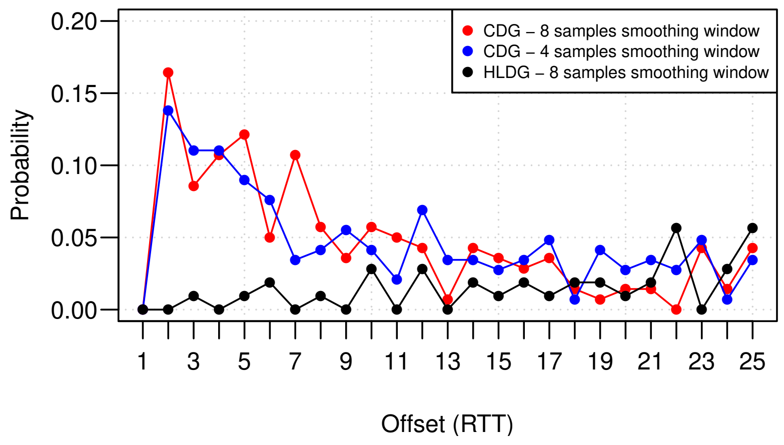


Figure 5.1: Probability of CDG and HLDG backing-off again after N^{th} RTT. CDG with 4 and 8 sample CDG smoothing window size

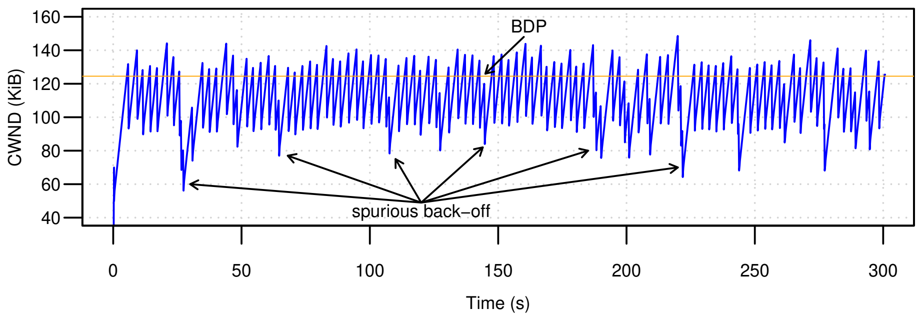
Using these equations, both $g_{min,n}$ and $g_{max,n}$ will be negative during the same sampling epoch after *cwnd* backs off as RTT measurements will be ignored. Similar to CDG, HLDG does not use $g_{min,n}$ and $g_{max,n}$ samples in $\overline{g}_{min,n}$ and $\overline{g}_{max,n}$ calculations.

To verify the effectiveness of this improvement, we run an experiment containing a single HLDG flow generated using `iperf` over a bottleneck with B_{rate}=10Mbps, RTT_{base}=100ms and 1000 packets buffer size. In this experiment, HLDG functions similar to CDG except for changes to the $g_{min,n}$ and $g_{max,n}$ calculations as per Equations 5.1 and 5.2.

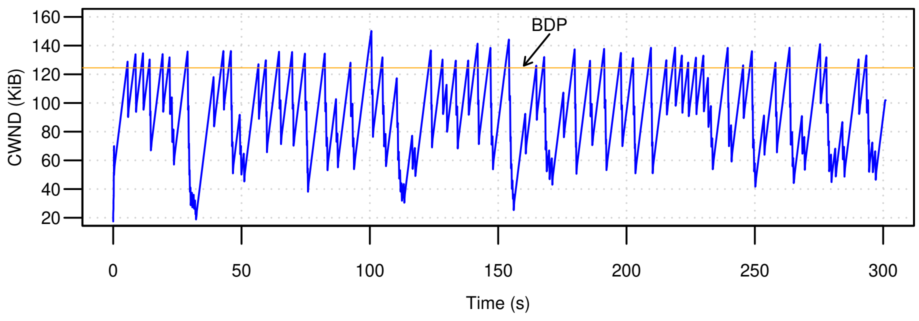
The black curve in Figure 5.1 shows the probability of HLDG backing-off again in the N^{th} RTT following the first back-off while *cwnd* is below BDP. Whe compared with CDG, we can see a significant drop on the probability of a second back-off in the immediate period following the first. In this experiment, HLDG backs off 106 times (compared to 140 times for CDG) with only 20 unnecessary back-offs and no consecutive back-offs. This results in the HLDG flow achieving 88.6% of link capacity utilisation, an increase of about %14 when compared with CDG.

In Figure 5.2a, we can see HLDG backs off *cwnd* unnecessarily less often than CDG (Figure 5.2b) after modifying $g_{min,n}$ and $g_{max,n}$ calculations. This results in *cwnd* being close to the path BDP, improving protocol throughput. However, we still see some unnecessary spurious back-offs. We will discuss how we address this issue in Section 5.2.2.

We examine one back-off case in detail in Figures 5.3 and 5.4. We can see in Figure 5.3a that HLDG backs off after *cwnd* exceeds BDP at t=154.66s. Figure 5.3b shows that the sender does not observe the reduction in RTT until one RTT later at t=154.79s. However, at



(a) HLDG



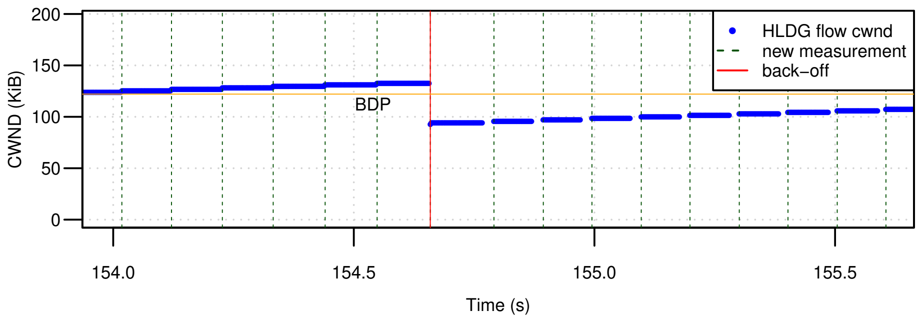
(b) CDG

Figure 5.2: Fixing $g_{min,n}$ and $g_{max,n}$ calculations in HLDG significantly reduces unnecessary *cwnd* back-off compared with CDG.

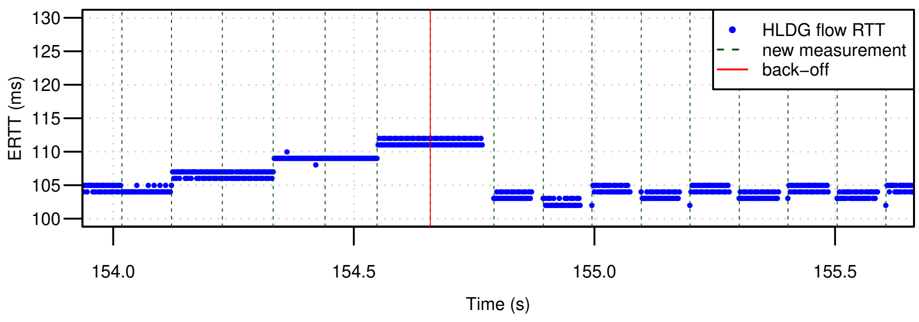
t=154.79s, we can see RTT_{min}, shown in Figure 5.3c, decreases to 103ms while RTT_{max} shown in Figure 5.3d, stays at 112ms at the same point of time. This demonstrates that RTT_{min} and RTT_{max} decrease asynchronously after delay back-off.

Due to equations 5.1 and 5.2, both $g_{min,n}$ and $g_{max,n}$ decrease synchronously after two RTTs at t=154.89s as shown in Figures 5.4a and 5.4b respectively. The asynchronous gradient decrease one RTT after back-off is not important since it is ignored.

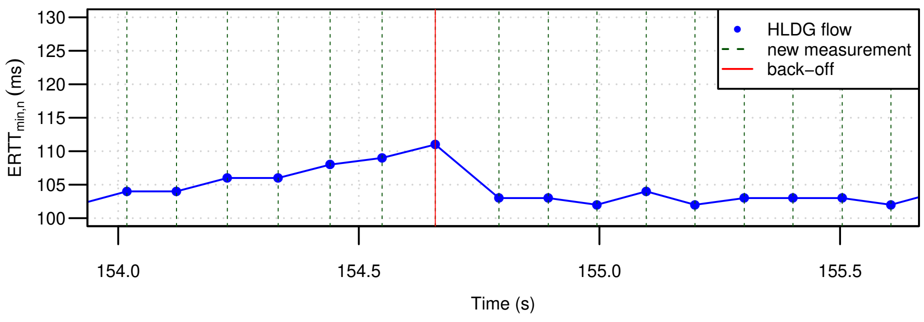
The smoothed gradients ($\bar{g}_{min,n}$ *and* $\bar{g}_{max,n}$) shown in Figures 5.4c and 5.4d show that gradients decrease immediately after two RTTs to reflect the impact of the *cwnd* back-off. As a result, the back-off probability decreases following *cwnd* reduction. This subsequently reduces the number of consecutive back-offs, resulting in significant improved throughput.



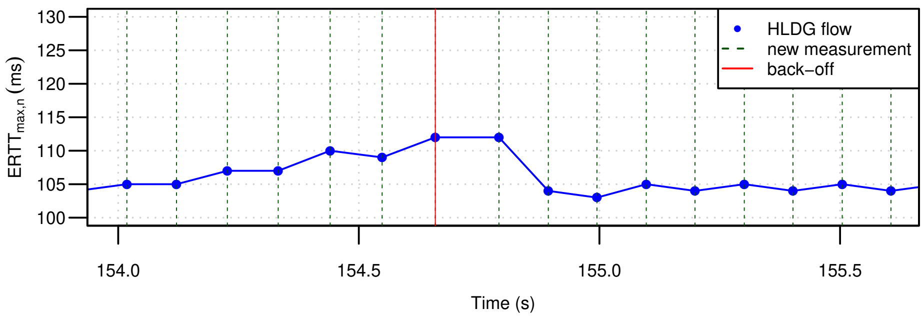
(a) HLDG *cwnd* backing-off



(b) ERTT progression

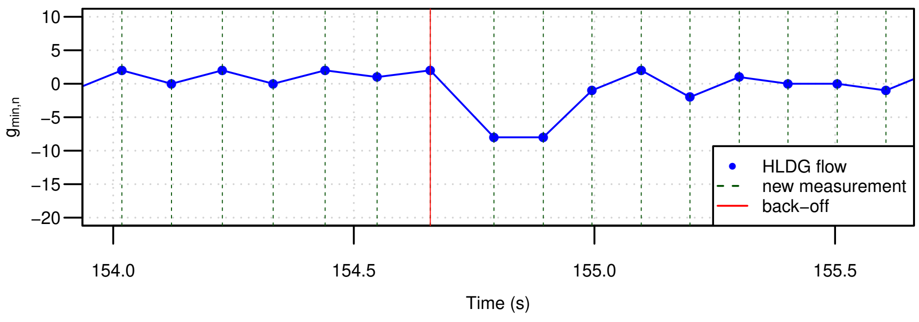


(c) $ERTT_{min,n}$ versus time sampled every RTT

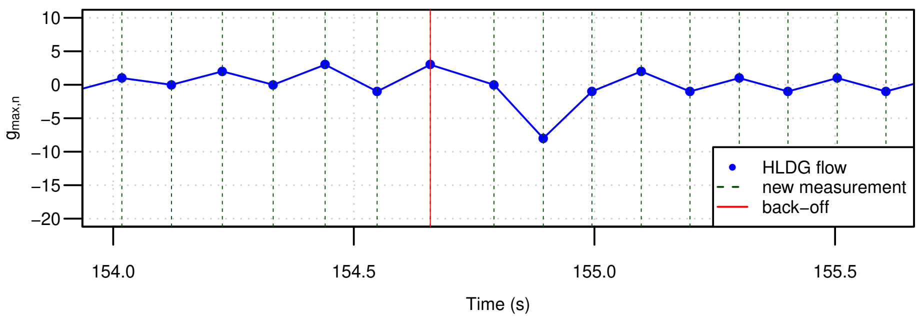


(d) $ERTT_{max,n}$ versus time sampled every RTT

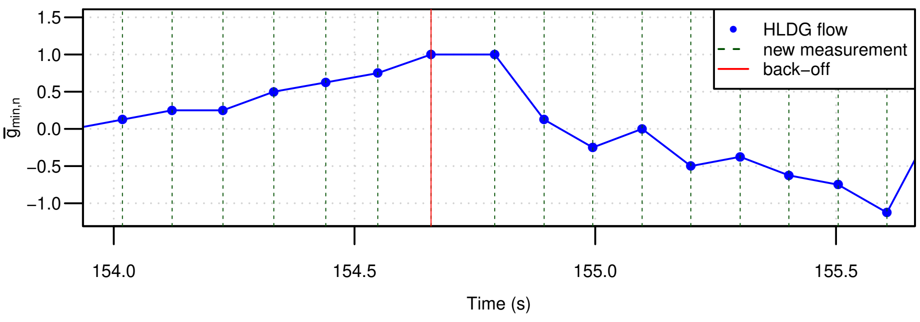
Figure 5.3: Single HLDG flow performing back-off



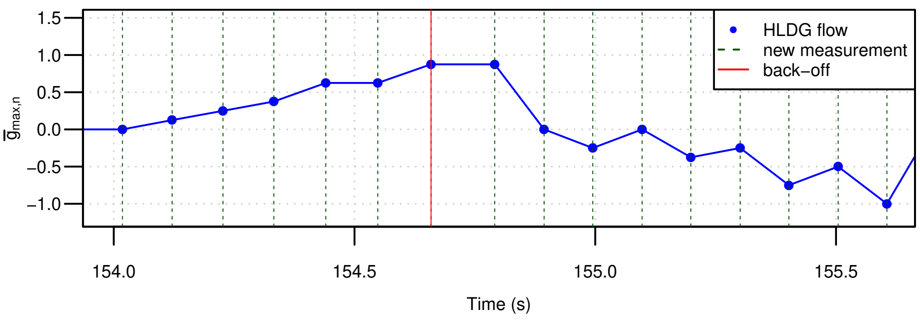
(a) $g_{min,n}$ versus time (unsmoothed gradient)



(b) $g_{max,n}$ versus time (unsmoothed gradient)



(c) $\bar{g}_{min,n}$ versus time



(d) $\bar{g}_{max,n}$ versus time

Figure 5.4: Zoomed-in delay gradient stats when HLDG flow performing back-off

5.2.2 Preventing spurious delay-based back-offs

In Section 4.2.2.2 we demonstrated how the RTT signal noise caused by the TCP delayed acknowledgement mechanism, coupled with the asynchronous timing between sent packets and the scheduler at the bottleneck, can result in CDG backing off randomly when no actual network congestion is present. More precisely, the TCP delayed acknowledgement mechanism and asynchronous timing allow short queues (a few packets) to build and drain, leading to queuing delay oscillation. The sender measures this oscillation through its RTT measurements (small positive gradient), leading to small positive $\overline{g}_{min,n}$ and $\overline{g}_{max,n}$. Since CDG uses a probability function to back off, there is always the chance that back-off can occur even with small RTT oscillations.

The decrease in the flow throughput caused by spurious back-offs becomes more noticeable for large RTT paths where the dynamic *cwnd* response of (growth and back-off) becomes slower due to the longer time taken for *cwnd* to reach the path BDP after back-off. On the other hand, a flow traversing a short RTT path can more quickly recover *cwnd* to the path BDP.

> ***HLDG modification***
>
> HLDG allows a small number of packets to be queued without backing off even if the probabilistic function indicates congestion.

We propose to solve the spurious back-off problem by allowing a small number of packets to be queued without backing off even if the probabilistic function indicates congestion. One method to achieve this is by estimating queue length using the TCP Vegas estimation (Section 2.4.1.2). However, this method requires an accurate RTT_{base} estimation which has its own challenges for short-term flows (Section 2.3.1.2). If RTT_{base} is overestimated, a long queue will build up as more packets will be allowed to queue. This can also create a standing queue which is an undesirable state for many applications. Similarly, if the estimator underestimates RTT_{base}, the spurious back-off preventer will be ineffective.

Our proposed solution is to ignore the probabilistic back-off decision if the smoothed gradient ($\overline{g}_{min,n}$ and $\overline{g}_{max,n}$) is less than the equivalent delay of queuing a few packets. In a single flow scenario, a flow introduces a positive gradient sample every RTT when *cwnd* is greater than BDP since *cwnd* increases by one MSS every RTT. In this scenario, the magnitude of an unsmoothed gradient sample is equivalent to the bottleneck serialisation delay of one packet. We can easily estimate smoothed delay gradient for γ packets by dividing γ packets serialisation delay by the smoothing window size. Given that the queue keeps growing under congestion, but both increases and decreases when experiencing signal noise, we can safely allow a few packets to queue without destroying the probabilistic back-off. Similar to other signal filtering methods this solution could reduce the congestion control responsiveness but by not more than γ RTTs.

More formally, HLDG back-off probability is calculated according to Equation 5.3 where γ is the number of packets we allow to queue, W_{size} is the smoothing window size and B is bottleneck bandwidth at the network layer in bps, n is the nth measurement cycle (RTT) and $\overline{g}_n$ is either $\overline{g}_{min,n}$ or $\overline{g}_{max,n}$. In this equation, the back-off probability is zero when the smoothed gradient is smaller than the queuing delay of γ packets divided by smoothing window size. Otherwise, CDG's normal back-off probability function is used.

$$P(\overline{g}_n) = \begin{cases} 0 & \bar{g}_n < \frac{\gamma \times 8 \times MTU}{B_n \times W_{size}} \\ 1 - e^{-(\overline{g}_n/G)} & \text{otherwise} \end{cases} \tag{5.3}$$

γ can be set to a small constant value (e.g. 2) or can be calculated dynamically based on bottleneck bandwidth. In general, γ should be small for low bandwidth connections and increase as connection speed increases to provide low queuing delay. We propose using Equation 5.4 to prevent spurious back-offs at different bottleneck speeds. Note that by using this equation, we allow not more than 1.2ms additional queuing delay when bandwidth is larger than 20Mbps, and 12ms if bandwidth is between 2Mbps and 20Mbps. We disable this mechanism for slower connections to prevent long queue delay. We also prevent queue lengths larger than $W_{size} - 1$ packets because HLDG can only use the measured smoothed gradient of W_{size} samples to make a back-off decision. As such, this solution alone does not completely prevent spurious back-off on fast links (B > 70Mbps when W_{size}=8).

$$\gamma = \begin{cases} min(W_{size} - 1, max(2, \frac{B}{10^7})) & B \geq 2Mbits/s \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

This proposal requires HLDG to have a bandwidth (B_n) estimation mechanism. In end-to-end CC, we do not have explicit information about bottleneck bandwidth, the sender must estimate bandwidth using available measurements. One method to estimate B_n is by using the TCP Vegas actual throughput estimator ($cwnd_i/RTT_i$) [133] where $cwnd_i$ is the current congestion window size and RTT_i is the current estimated RTT. This method is based on the fact that the sender can transmit *cwnd* worth of bytes per RTT. At the same time, the sender cannot transmit more data than bottleneck capacity per RTT without introducing queuing delay. This queuing delay is combined with path RTT to produce current estimated RTT. Increasing *cwnd* (bytes) above BDP will also increase RTT. Therefore, this method produces an approximation of bottleneck bandwidth. However, RTT signal noise affects this estimator, resulting in

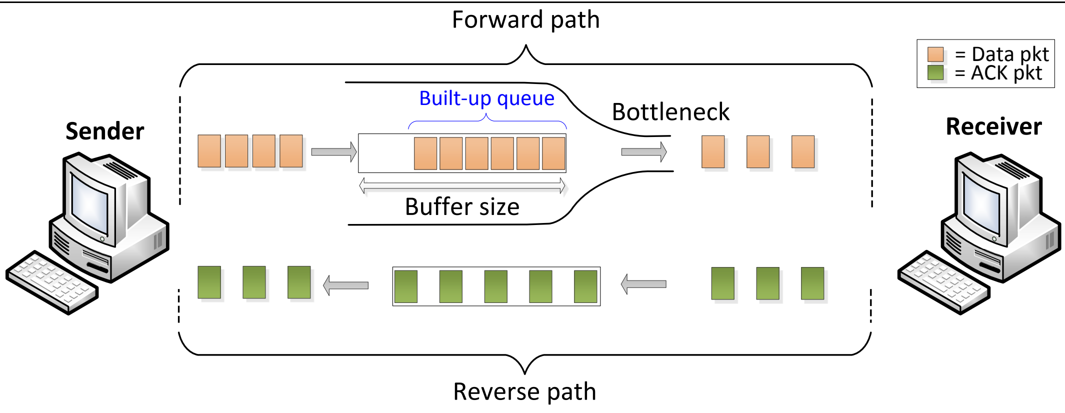


Figure 5.5: ACK rate reflects bottleneck bandwidth

a noisy bandwidth estimation. Moreover, this technique requires an accurate RTT estimation otherwise the estimator will overestimate or underestimate link bandwidth.

HLDG uses a better (discussed later) bandwidth estimator that calculates acknowledged bytes over an interval of time (ACKed byte rate). The rationale behind this estimator is that the receiver receives packets/bytes as the same rate as the bottleneck. The receiver then sends acknowledgements at the same rate to the sender. Therefore, the acknowledged byte rate received at the sender side will reflect bottleneck bandwidth in the forward path. Figure 5.5 illustrates how the acknowledged byte rate reflects bottleneck bandwidth.

We select the measurement interval to be one RTT long, obtaining a new bandwidth estimate whenever HLDG makes its back-off decision. Formally, bandwidth estimation B_n for the measurement cycle n is calculated using Equation 5.5 where $acked_i$ is number of bytes acknowledged by packet i, m is number of packets during the measurement cycle, and Δt is the difference between the start time (t_s) and end time (t_e) of the measurement cycle.

$$B_n = \frac{\sum_{i=1}^{m} acked_i}{\Delta t} \tag{5.5}$$

We assume that the flow's throughput is an estimation for bandwidth but that is not true when the sending rate is limited by the application or *cwnd* is below BDP. In both cases throughput will be less than link bandwidth leading the estimator to underestimate bandwidth. However, that does not affect Equation 5.3 because there is no network congestion if the sender emits data slower than bottleneck bandwidth. Therefore, preventing *cwnd* back-off in this case has no impact on network congestion.

We conduct an experiment to compare the ACK rate bandwidth estimator with the Vegas-like bandwidth estimator to see which technique results in a better estimation. This experiment

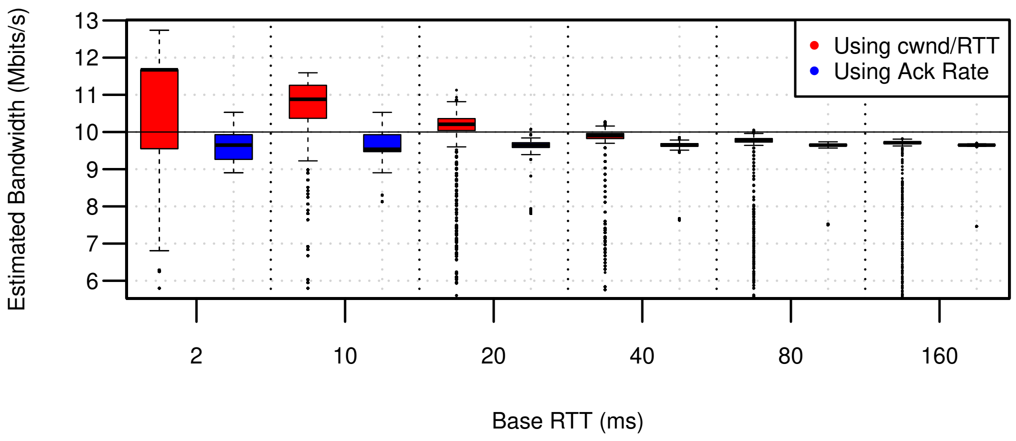


Figure 5.6: A comparison of bandwidth estimate using TCP Vegas-like estimator and Ack rate estimator for a symmetric 10Mbps emulated link with different path's RTT

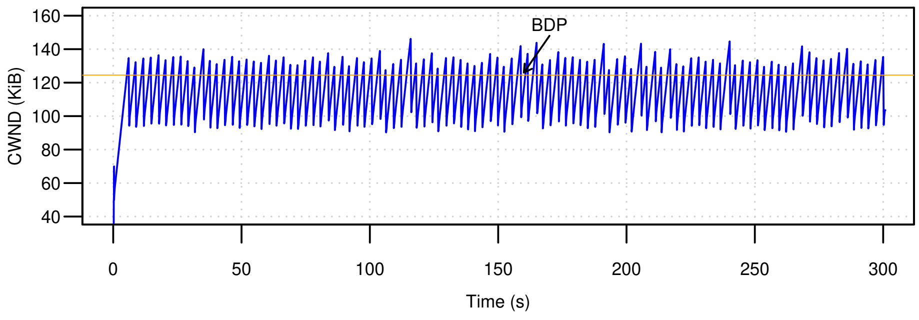
involves a single Reno-based flow session generated using `iperf`. The bottleneck is configured to emulate a symmetric 10Mbps link with 2ms, 10ms, 20ms ,40ms, 80ms and 160ms path RTT and the buffer size is set to double the path BDP. Each run lasts for 5 minutes and both estimator measurements are collected simultaneously.

The box plot for the experiments is shown in Figure 5.6. We can see that when RTT_{base} is less than 40ms, the TCP Vegas technique both overestimates link bandwidth and that the estimation has a large variance. When RTT_{base} is small, *cwnd* changes frequently, resulting in RTT measurements varying as well. Also, since the current RTT carries the delay measurement of the previous window (one RTT before), *cwnd* and RTT values are not fully synchronised. Additionally, TCP uses an exponential moving average to calculate a smoothed RTT to minimise signal noise. This results in previous RTT samples impacting on the current estimate.

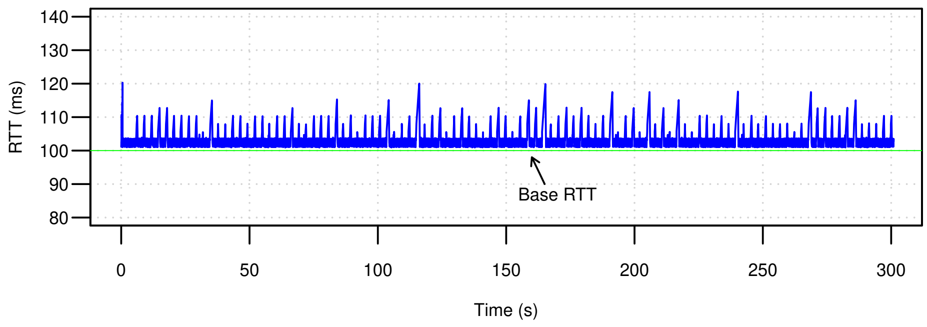
On the other hand, (see Figure 5.6) bandwidth estimation using the ACK rate does not overestimate bottleneck bandwidth and produces a better and more stable estimation. The is due to the estimator not relying on other measurements such as RTT, but instead directly measuring the number acknowledged bytes over an interval of time. This estimator however, can be affected by the packet aggregation mechanism[2] and ACK compression phenomenon. We expect that by calculating the ACK rate over a long enough interval that would filter out such packet bursts. This requires further investigation to measure the effect of ACK compression on the estimator.

We repeat the same experiment from Section 5.2.1 to validate the functionality of our proposed solution to the spurious delay-based back-off problem. The experiment consists

[2]Used by some wireless networks to improve performance



(a) HLDG does not back off *cwnd* unnecessarily



(b) HLDG does not create standing queue when RTT_{base} is 100ms

Figure 5.7: HLDG provides better *cwnd* control and low queuing delay (B_{rate}=10Mbps, RTT_{base}=100ms)

of a single 300 second HLDG flow with both consecutive and back-off fixes applied. The bottleneck is configured to emulate B_{rate}=10Mbps, RTT_{base}=100ms (the path BDP = 122KiB) with 1000 packets buffer size.

In this experiment, HLDG backs off *cwnd* 103 times without any unnecessary backs-off. The HLDG flow achieves 90% link utilisation (a 23% improvement over CDG). In Figure 5.7a we can see HLDG does not perform any unnecessary back-off in this experiment after applying the proposed solution. Furthermore, we can see in Figure 5.7b that HLDG does not create a standing queue in this experiment with the queuing delay being mostly stable with some RTT spikes causing by the probabilistic nature of the back-off function.

These findings allow us to draw a general conclusion that delay-based CC should filter out signal noise caused by the delayed acknowledgement mechanism to prevent losses in protocol throughput. The delayed acknowledgement mechanism leads to short RTT bursts which will otherwise be interpreted as a congestion signal, leading to unnecessary back-off and reduced throughput.

5.2.3 Standing queue issue

Unlike CDG, HLDG backs off *cwnd* only when an actual queue builds up. Although this improves throughput, it can create a standing queue for specific scenarios when RTT_{base} is small. However, when RTT_{base} is large HLDG does not create a standing queue as shown in Figure 5.7b. A standing queue has a negative impact on low latency applications since all packets will experience extra delay which is contrary to HLDG goals.

Figure 5.8 shows the cumulative distribution function for queuing delay of HLDG flows traversing bottlenecks emulating B_{rate}=$\{1, 10, 25, 50\}$Mbps links and RTT_{base}=$\{0, 2, 4, 8, 12, 16, 20, 40\}$ms. We can see in those graphs that as path RTT increases, the queuing delay decreases. Additionally, the standing queue becomes shorter when bottleneck bandwidth increases.

Figure 5.8a (B_{rate}=1Mbps) reveals that HLDG creates a standing queue for all RTT_{base} settings. The shortest standing queue is when $RTT_{base}$is 40ms. In Figure 5.8b (B_{rate}=10Mbps) we can see that when RTT_{base} is 20ms or less, a standing queue is created. The graph also shows no standing queue when RTT_{base} = 40ms.

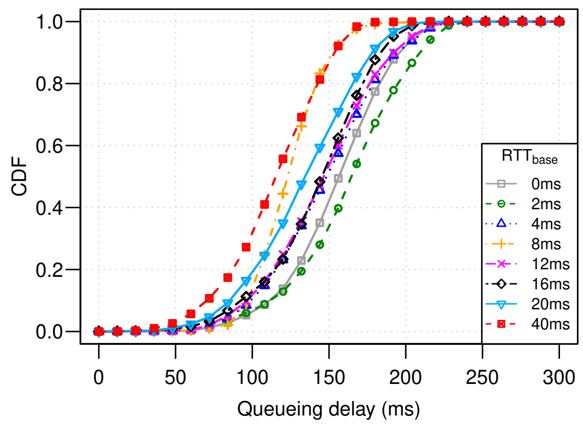
In Figure 5.8c (B_{rate}=25Mbps) we can see a standing queue when RTT_{base} is 12ms or less. The graph shows no standing queue when $RTT_{base} \geq$ 16ms. Figure 5.8d (B_{rate}=50Mbps) shows that when RTT_{base} is around 8ms or less, HLDG creates a shorter standing queue. The graph shows no standing queue when $RTT_{base} \geq$ 8ms.

These results show that a standing queue is created when the path BDP is lower than approximately 50KiB (~34pkts).

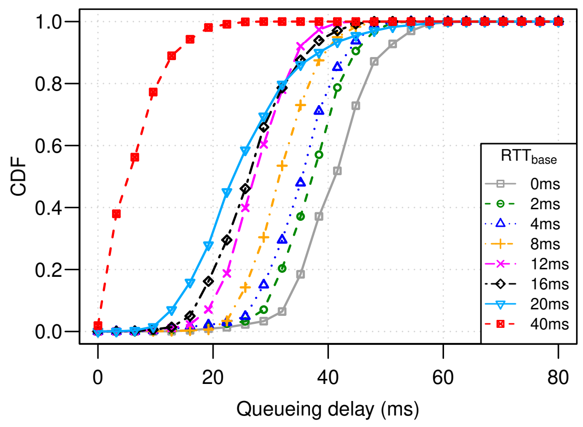
To further investigate this issue, we conduct an experiment consisting of a single 300 second HLDG flow. The bottleneck is configured to emulate a B_{rate}=10Mbps, RTT_{base}=2ms with a 1000 packet buffer size. We can see in Figure 5.9a that HLDG maintains a *cwnd* value much higher than the path BDP (BDP = 2.4KiB). Due to high *cwnd* values (greater than 40KiB) a standing queuing delay over 35ms in average is created as shown in Figure 5.9b.

This behaviour only happens with small RTT_{base}. When the path BDP is small, backing off *cwnd* using a 0.7 factor does not reduce *cwnd* by a large value. As a consequence, *cwnd* more rapidly recovers to BDP and starts forming a queue. In addition, HLDG needs many RTTs (up to W_{size} RTTs) to properly measure this RTT increase due to the smoothing window to overcome the negative gradient added to the smoothed gradient after a back-off. In other words, HLDG does not back off *cwnd* quickly enough and does not reduce *cwnd* enough when backing off. This behaviour results in the bottleneck queue growing until it reaches a stable state where the queue length oscillates around a specific length and never decreases to zero. The average RTT for this experiment is 41.29ms.

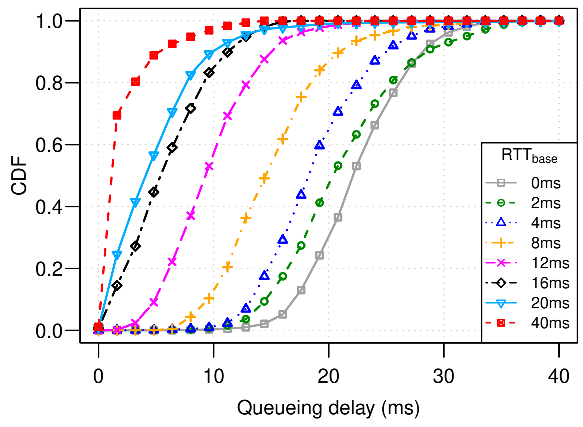
CDG also creates this standing queue behaviour but has a smaller effect. For the sample



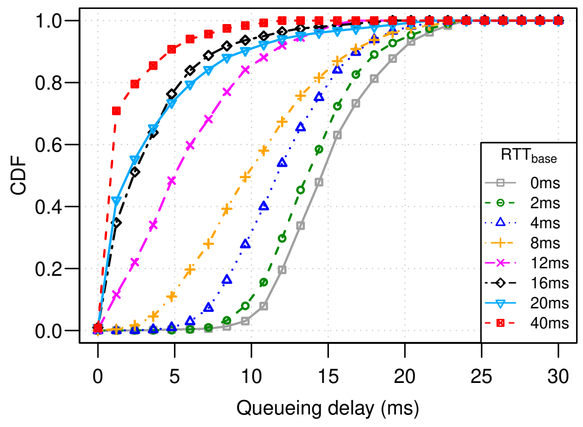
(a) B_{rate}=1Mbps



(b) B_{rate}=10Mbps



(c) B_{rate}=25Mbit/s

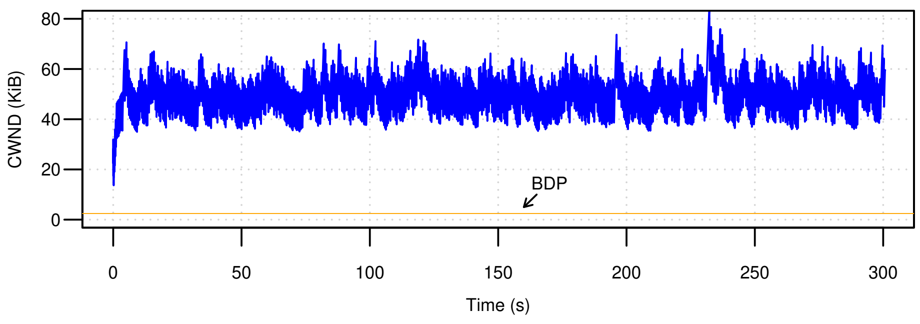


(d) B_{rate}=50Mbit/s

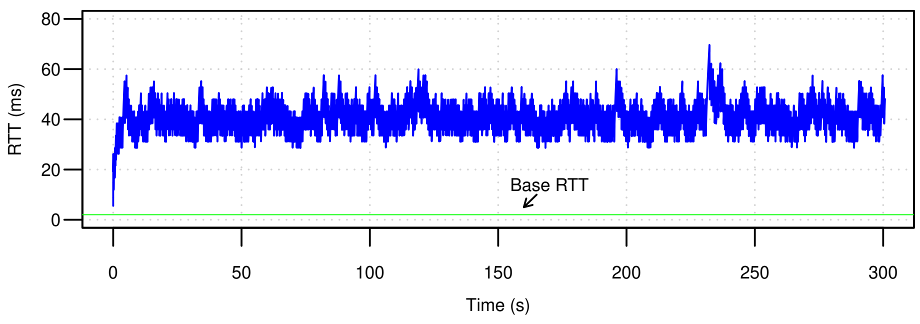
Figure 5.8: Queuing delay CDF plots for a single HLDG flow showing the effect of bottleneck bandwidth and path's RTT on standing queue build-up

experiment parameters above with a CDG flow, the average RTT is 19.28ms.

We perform a micro-analysis of a small period of the flow to better understand the cause. Figure 5.10 plots $ERTT_{min,n}$, $g_{min,n}$ and $\bar{g}_{min,n}$ for the experiment for 154s < t < 155.5s. We can see in Figure 5.10a that when HLDG backs off, $RTT_{min,n}$ decreases causing $g_{min,n}$ to dip to around -14 as shown in Figure 5.10b. The very small $g_{min,n}$ causes $\bar{g}_{min,n}$ to decay to around -0.75 as shown in Figure 5.10c. Although $g_{min,n}$ fluctuates between 0 and 3, $\bar{g}_{min,n}$ remains at about -0.75 for eight cycles due to the dominant negative $g_{min,n}$ value immediately after back-off. After eight cycles $\bar{g}_{min,n}$ jumps to 1.2 as the large negative $g_{min,n}$ value moves outside the window. This behaviour reduces HLDG back off frequency causing queue build up.



(a) HLDG keeps *cwnd* higher than BDP when RTT_{base} is 2ms



(b) A standing queue is created causing undesirable queuing delay when RTT_{base} is 2ms

Figure 5.9: HLDG causes standing queue in small RTT_{base} paths (B_{rate}=10Mbps, RTT_{base}=2ms)

5.2.4 Standing queue solutions

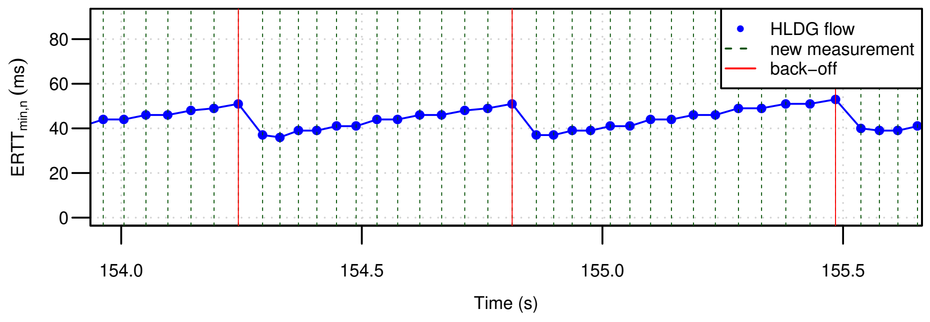
Many methods can be used to remedy the standing queue problem described in Section 5.2.3. In this section, we validate, compare and contrast proposals to reduce the standing queue created by HLDG flows.

5.2.4.1 Method 1: Reducing smoothing window size

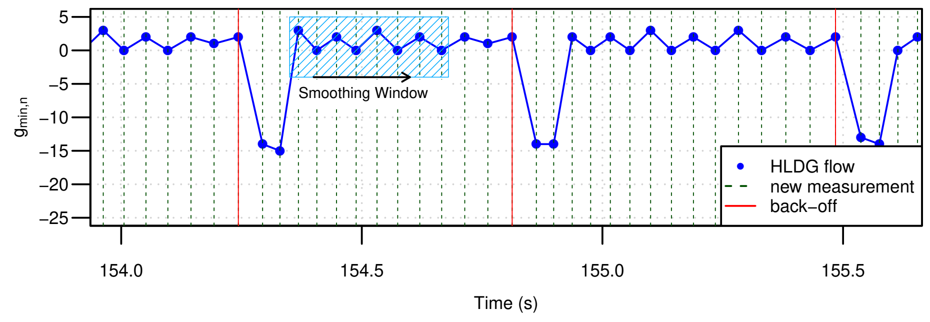
A simple solution is to reduce W_{size}. This allows the large negative g_n sample after the back-off to be more quickly excluded from the smoothing window calculation. HLDG will back off more often, preventing a long standing queue from building up. Although this solution is simple to apply, it can weaken the smoothing filter leading to nosier measurements.

5.2.4.2 Method 2: Weighed Windowed Moving Average Filter

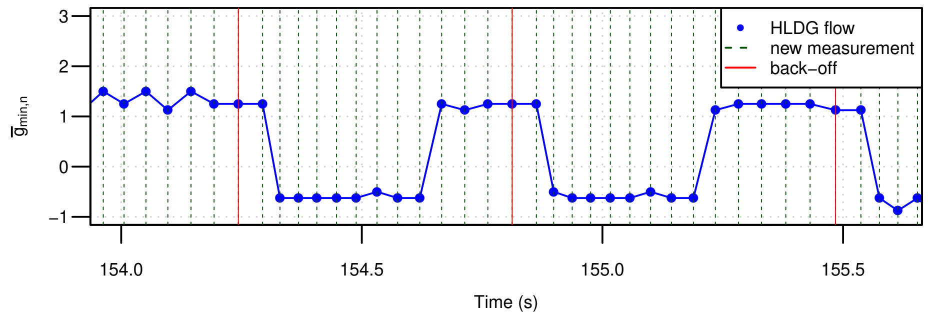
The second method to remedy the standing queue problem is to use different gradient filter.



(a) $ERTT_{min,n}$ versus time sampled every RTT



(b) $g_{min,n}$ versus time (unsmoothed gradient)



(c) $\bar{g}_{min,n}$ versus time

Figure 5.10: Zoomed-in HLDG delay measurements shows the effect of smoothing window and large negative gradient on back-off frequency

HLDG modification

HLDG utilises a weighted, windowed moving average (WWMA) to compute $\overline{g}_n$.

An exponential weighted moving average (EWMA) is a simple mechanism that can be deployed to calculate a moving average that gives more weight towards more recent samples. A similar technique is used to estimate RTT by many TCP CC algorithms. One of the drawbacks of an EWMA us that all previous samples impact on the current smoothed delay gradient, even though the impact decays in time. As such, abrupt changes in underlying network conditions will take some time to filter out of the EWMA calculation.

We propose to use a weighted windowed moving average (WWMA) instead of a normal moving average to compute $\overline{g}_n$. Our WWMA is designed to emulate the basic properties of an EMWA by giving stronger weight to more recent g_n samples. This provides the advantage that only the last (n=8) samples are used, ensuring currency and removing extreme transients more quickly. Given that HLDG always maintains the last n samples, this comes at minimal storage cost but at a slight computational cost. We propose to use WWMA weights of $(\frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{8}, \frac{1}{8}, \frac{1}{4}, \frac{1}{4})$ [3].

Using such an approach would ensure that the impact of a large negative g_n value on $\overline{g}_n$ will rapidly decrease following backoff. This increases the probability of HLDG backing off again if the queue builds up, leading to better control of the queue delay.

We conduct an experiment consisting of a single 300 second HLDG flow while applying WWMA in calculating $\overline{g}_n$. The bottleneck is configured to emulate B_{rate}=10Mbps, RTT_{base}=2ms with a 1000 packet buffer size (the same experiment setup above).

We perform a micro-analysis of a small period of the flow to better understand the effect. Figure 5.11 plots $ERTT_{min,n}$, $g_{min,n}$ and $\bar{g}_{min,n}$ for the experiment. We can see in Figure 5.11a that when HLDG backs off, $RTT_{min,n}$ decreases causing $g_{min,n}$ to drop to about -10 (shown in Figure 5.11b). The small $g_{min,n}$ causes $\bar{g}_{min,n}$ to decrease to about -2 (shown in Figure 5.11c). Although $g_{min,n}$ in this experiment is a smaller negative than in Figure 5.10b, we see a larger negative $\bar{g}_{min,n}$ compared with 5.10c. This is due to newer $g_{min,n}$ samples having higher weight than older samples, resulting in the large negative $\bar{g}_{min,n}$ having a higher but more transient impact than when linear smoothing window is applied. Unlike Figure 5.10, $\bar{g}_{min,n}$ gradually increases as the queue grows due to the WWMA filter.

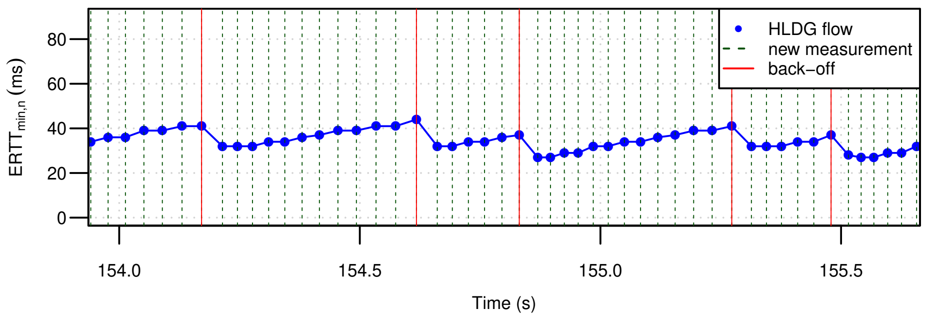
The WWMA filter allows HLDG to more quickly respond to queuing delay increase, leading to more frequent back-off. We can see in Figure 5.11 at t=154.6s that HLDG backs off and after six cycles backs off again. The average RTT of this experiment is 33.09ms which is better than with a linear moving average (41.29ms).

Although this technique can reduce the standing queue, it reduces the effect of the smoothing filter, making the delay measurements more prone to signal noise. This WWMA can also produce additional processing overhead compared with a linear moving average.

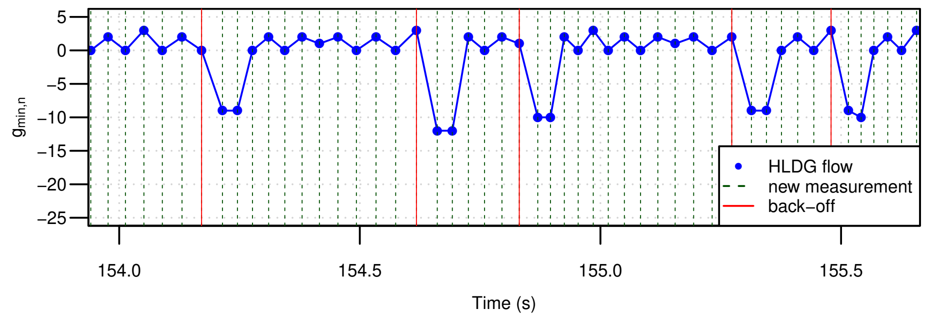
5.2.4.3 Method 3: Reset the smoothed gradient window after back-off

In specific situations, samples in the smooth gradient window become invalid, resulting in either unnecessary or late back-offs.

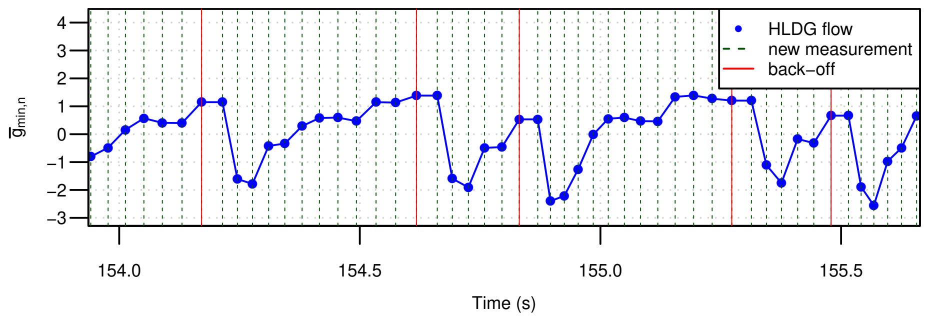
[3] Shift operations can be used to implement this filter to reduce division and multiplication operation overhead



(a) $ERTT_{min,n}$ versus time sampled every RTT



(b) $g_{min,n}$ versus time (unsmoothed gradient)



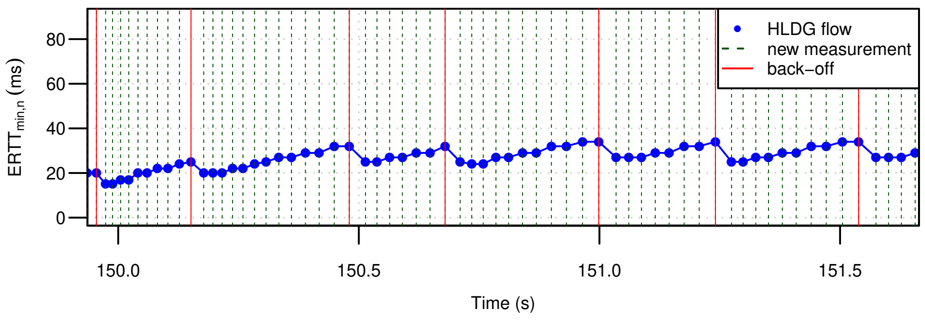
(c) $\bar{g}_{min,n}$ versus time

Figure 5.11: Zoomed-in HLDG with WWMA delay measurements shows the effect of WWMA on the smoothed gradient and back-off frequency

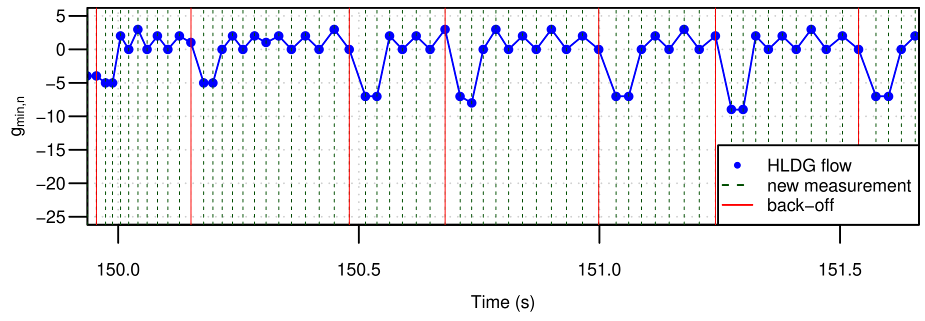
HLDG modification

HLDG resets the smoothed gradient window when a negative gradient sample is observed after back-off.

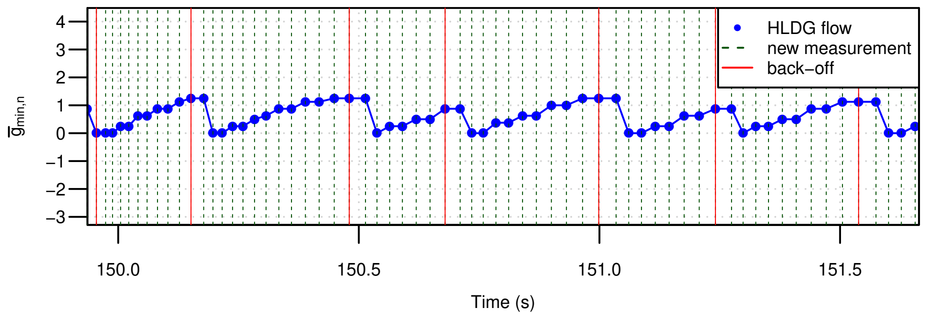
We propose to reset the smoothed gradient window (i.e. zeroing all samples in the window) if a negative gradient sample (g_n) is observed after back-off. The decision to reset the window is made based on the sign of g_n two RTTs from back-off to ensure the back-off is effective (i.e.



(a) $ERTT_{min,n}$ versus time sampled every RTT



(b) $g_{min,n}$ versus time (unsmoothed gradient)



(c) $\bar{g}_{min,n}$ versus time

Figure 5.12: Zoomed-in delay measurements of HLDG with gradient window reset shows the effect of this method on the smoothed gradient and back-off frequency

decrease queuing delay) and due to of the measurement asynchronisation problem mentioned in Section 5.2.1. Zeroing old g_n samples allows $\overline{g}_n$ to be quickly influenced by new g_n samples, allowing the back-off probability to increase as soon as a queue starts to form again. We call this proposal the zeroing smoothed gradient window (ZSGW).

The concept behind restarting the smoothed gradient calculation is that after backing off *cwnd*, the $\overline{g}_n$ value will be invalid due to the sudden change in queuing delay. Even if the back-off is effective, the queue may not be completely drained since *cwnd* could be either above or below BDP. As such, it is better to ignore all previous measurements and begin again. If *cwnd* is below BDP, the queue will not form until *cwnd* reaches BDP. Alternatively, if *cwnd* is above BDP, the queue will grow causing the gradient to increase quickly. HLDG will then back off based on the increasing probability.

After implementing this method, we repeat the same experiment. A micro-analysis of $ERTT_{min,n}$, $g_{min,n}$ and $\bar{g}_{min,n}$ for the experiment is shown in Figure 5.12. We can see that HLDG backs off more frequently than with the WWMA method, resulting in a lower queuing delay. The average RTT for this experiment is 29.3ms which is better than 41.29ms (CDG) and 33.09ms (WWMA).

We can see in Figure 5.12a that when HLDG backs off, $RTT_{min,n}$ decreases causing $g_{min,n}$ to drop to around -10 (shown in Figure 5.12b). The large negative $g_{min,n}$ does not cause $\bar{g}_{min,n}$ to become negative (shown in Figure 5.12c) since all gradient window samples are zeroed following the *cwnd* decrease. Somewhat similar to WWMA , $\bar{g}_{min,n}$ gradually increases as the queue grows. Interestingly, $\bar{g}_{min,n}$ behaviour is similar to $ERTT_{min,n}$ behaviour but independent of the RTT absolute level.

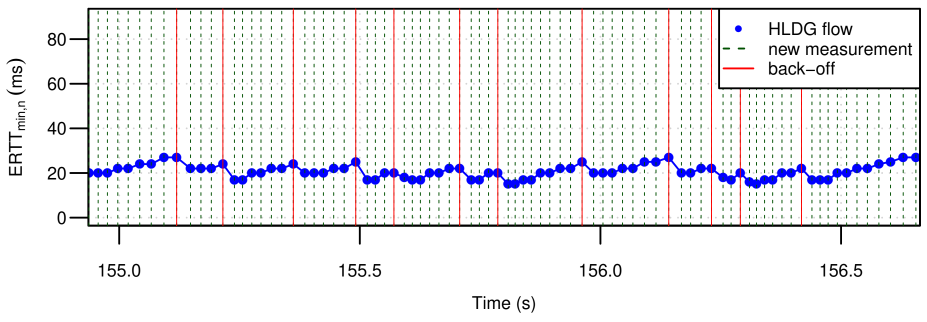
5.2.4.4 Method 4: Using WWMA and reset the smoothed gradient window after back-off

Combining WWMA with resetting the smoothed gradient window after back-off (methods 2 and 3 above) can produce lower queuing delay. WWMA results the smoothed gradient to respond more quickly to delay changes while restarting values in the gradient window allows HLDG to better infer network congestion after backing off *cwnd*.

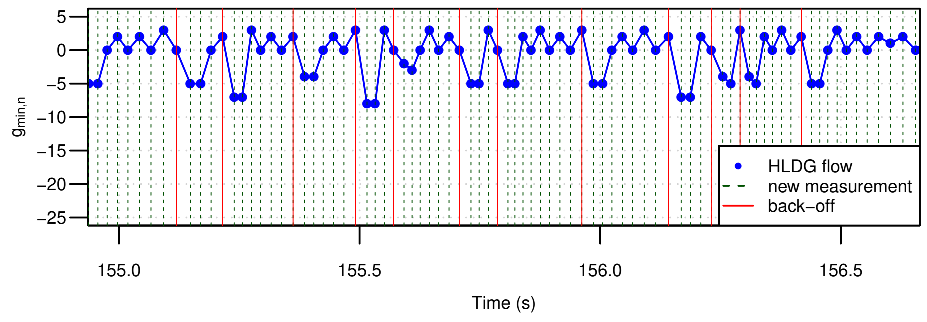
After applying the two methods we again repeat the experiment with results plotted in Figure 5.13. Similar to previous observations, we can see in Figure 5.13a that when HLDG backs off, $RTT_{min,n}$ decreases causing $g_{min,n}$ to drop to about -8 (shown in Figure 5.13b). The negative $g_{min,n}$ after two RTT from back-off, $\bar{g}_{min,n}$ and all eight samples in the smoothing window are zeroed. After that $\bar{g}_{min,n}$ quickly increases (shown in Figure 5.13c) because of positive $g_{min,n}$ samples and the higher weighting of WWMA for new samples. We still see $\bar{g}_{min,n}$ gradually increase as the queue grows. The average RTT for HLDG in this experiment is 23.3ms which is very close to the original CDG. However, HLDG is able to prevent unnecessary back-off in a higher and more realistic RTT_{base} for Internet scenarios.

These results show that WWMA is an effective filter that both ensures a quick response to changes in the delay signal while eliminating sharp transitions in the delay signal. More generally, this filter can be deployed with any delay-based CC to improve response times to delay signal changes.

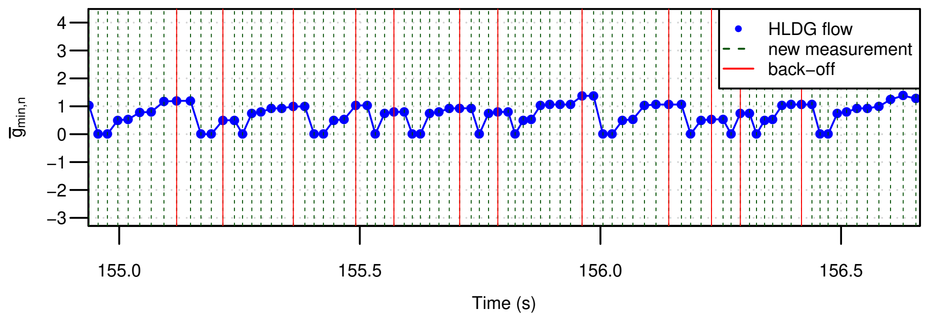
In the next section, we provide a solution that can further enhance HLDG standing queue behaviour and improve HLDG throughput in large RTT_{base} paths.



(a) $ERTT_{min,n}$ versus time sampled every RTT



(b) $g_{min,n}$ versus time (unsmoothed gradient)



(c) $\bar{g}_{min,n}$ versus time

Figure 5.13: Zoomed-in delay measurements of HLDG with WWMA and ZSGW shows the effect of this method on the smoothed gradient and back-off frequency

5.2.5 Backing off *cwnd* to estimated BDP

The negative impacts of the CDG constant β_{delay} factor on link utilisation was demonstrated in Section 4.2.1. CDG keeps *cwnd* close to the path BDP to achieve low queuing delay. However, if congestion is detected using delay, CDG will decrease *cwnd* by 30%, which will now be below BDP, leading to the link being underutilised. We can see in Figure 5.7a that *cwnd* drops to about 95KiB (~ 27KiB below BDP) after *cwnd* back-off.

In this case, where the base RTT is larger, recovery of *cwnd* to BDP is slow and HLDG suffers from low link utilisation, resulting in 10% of the bandwidth not being used. Further, as discussed in Section 5.2.3, there is the problem of a standing queue created by HLDG in small BDP paths. These two problems can be minimised if HLDG backs off *cwnd* to a value close to the path BDP instead of using β_{delay} factor.

HLDG modification

HLDG sets *cwnd* to a fixed fraction of estimated path BDP when congested is detected using delay.

We instead propose to set *cwnd* to a fixed fraction of the estimated path BDP when congestion is detected using the delay component. This method can prevent *cwnd* from dropping too far below BDP and avoid a standing queue from forming at the bottleneck. It also ensures that *cwnd* will not drop too far below BDP, thus preserving high link utilisation. HLDG will not require too many RTT cycles for *cwnd* to recover which is critical in larger RTT paths. Further, setting *cwnd* below BDP allows the building queue to drain, preventing a standing queues from forming.

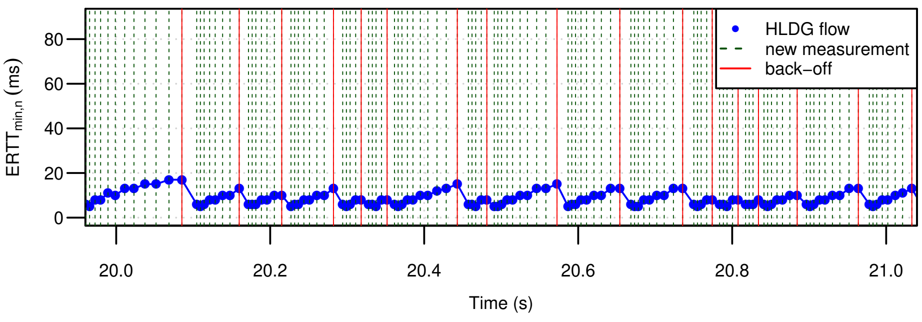
HLDG sets *cwnd* every RTT according to equation 5.6 where λ (< 1) is the back-off factor (default 0.95) and determines how far below the estimated BDP *cwnd* should be set to following a delay-based back-off. X is a uniformly distributed random number between 0 and 1, n is the nth measurement cycle (RTT), and $P(\overline{g}_n)$ is the back-off probability for $\overline{g}_{min,n}$ or $\overline{g}_{max,n}$ (Equation 5.3).

$$cwnd_{i+1} = \begin{cases} BDP.\lambda & X < P(\overline{g}_n) \\ cwnd_i + MSS & otherwise \end{cases} \tag{5.6}$$

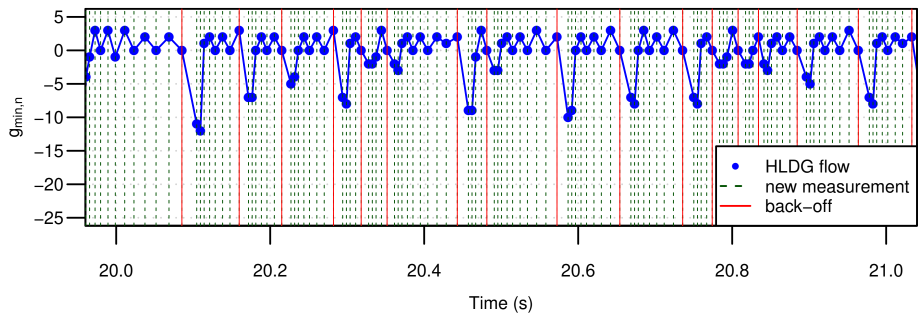
In order to use this technique, the sender requires a reasonable BDP estimation without relying on intermediate nodes. HLDG estimates the BDP using a conventional BDP calculation as per Equation 5.7.

$$BDP = B_n \times RTT_{ebase} \tag{5.7}$$

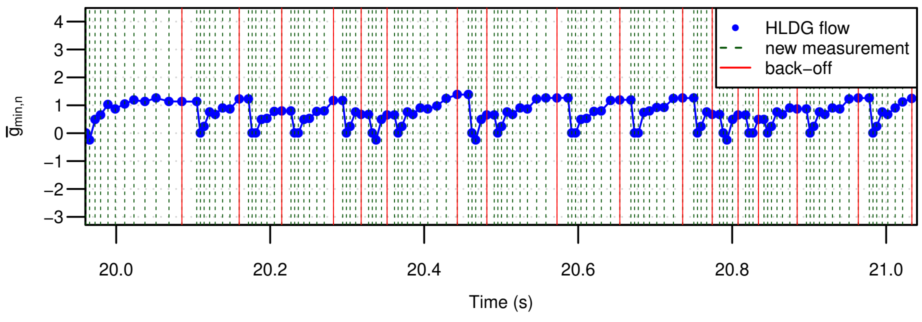
where B_n is the estimated bottleneck bandwidth during the nth RTT, and RTT_{ebase} is the estimated base RTT. B_n can be estimated using ACK rate method (Equation 5.5). RTT_{ebase} estimation is described in Section 5.2.6. BDP should never be greater than *cwnd* in our calculations as the B_n estimation is based on the sender transmission rate and bottleneck bandwidth. As such, *cwnd* will always decreases when congestion is detected. We set *cwnd* to $cwnd_i.\beta_{delay}$ when congestion is detected and $cwnd_i < BDP$, since BDP estimation is inaccurate in that case.



(a) $ERTT_{min,n}$ versus time sampled every RTT



(b) $g_{min,n}$ versus time (unsmoothed gradient)



(c) $\bar{g}_{min,n}$ versus time

Figure 5.14: Zoomed-in delay measurements of HLDG with backing off to $BDP.\lambda$ shows the effect of this method on the smoothed gradient and back-off frequency

We conducted an experiment consisting of a single 300 second HLDG flow (with WWMA, resetting the smoothed gradient window samples after back-off, and using estimated BDP proposals implemented). The bottleneck is configured to emulate B_{rate}= 10Mbps, RTT_{base}= 2ms with a 1000 packet buffer size (the same experiment as in Section 5.2.4). In this experiment, HLDG introduces an average RTT of 9.81ms with a median RTT of 9.52ms and standard deviation of 4.32ms. This RTT is lower than using the default β_{delay} factor of 0.7 (average RTT=23ms). This is due to HLDG backing off *cwnd* close to the estimated BDP which is lower than $0.7 \times cwnd_i$ (*cwnd* is larger than BDP/0.7). HLDG flows do not induce long queuing delay.

A micro view of $ERTT_{min,n}$, $g_{min,n}$ and $\bar{g}_{min,n}$ for the experiment for 20s < t < 21s are shown in Figure 5.14. We can see in Figure 5.14a that HLDG controls the latency very well by keeping queuing delay as small as possible. WWMA and zeroing gradient samples of smoothing window allow HLDG to back off shortly after the queuing delay increases.

Figure 5.14b shows that some back-off events result in large decreases in queuing delay, resulting in $g_{min,n}$ dropping below -6 while other back-off events have smaller impact on queuing delay. This variation is due to HLDG decreasing *cwnd* by an adaptive ratio of bottleneck capacity rather than constant β_{delay}.

When a longer queue is created (due to the probabilistic back-off function or the effect of the smoothing window), HLDG decreases *cwnd* by a large amount. As a result, the sender reduces packets in-flight allowing the queue to drain. Then, $RTT_{min,n}$ becomes much smaller causing $g_{min,n}$ to drop deeply. Alternatively, in the case of a short queue, HLDG backs off *cwnd* by a smaller amount to preserve high link utilisation. As the queue is short, $RTT_{min,n}$ decreases by a small amount since it is already close to RTT_{base}. As a consequence $g_{min,n}$ does not drop as far. In Figure 5.14c we can still see $\bar{g}_{min,n}$ quickly increase after back-off due to WWMA and zeroing gradient samples of smoothing window.

Despite the the benefits, there is potential for HLDG to drop *cwnd* too far if the path BDP is underestimated. This occurs when the bandwidth estimator is unable to obtain an accurate measurement due to the ACK compression phenomenon or a congested reverse path. BDP can also be underestimated if RTT_{base} is underestimated.

Further, backing off *cwnd* to $BDP.\lambda$ may result in HLDG obtaining a lower bandwidth share when competing with loss-based flows. If the sender host incorrectly infers congestion (in the presence of competing loss-based flows) and reduces *cwnd,* the estimator will underestimate the available bandwidth. As a result, the BDP estimate will be smaller than the path BDP, causing the HLDG flow to realise very low throughput.

To address this, HLDG provides an option to modify Equation 5.6 by backing off *cwnd* to the maximum of $BDP.\lambda$ and $cwnd_i \times \beta_{delay}$ as per Equation 5.8. When using this option, HLDG will not decrease *cwnd* too far when the path BDP is underestimated. However, it does create a standing queue when the path BDP is small (less than 50KiB).

We recommend enabling this option if HLDG is used on small BDP paths or when competing with loss-based flows is expected to in order obtain higher throughput. Alternatively, if HLDG is used to provide scavenger class services and low latency transport, then this option should not be enabled.

$$cwnd_{i+1} = \begin{cases} max(BDP.\lambda, cwnd_i \times \beta_{delay}) & X < P(\overline{g}_n) \\ cwnd_i + MSS & otherwise \end{cases} \tag{5.8}$$

Using an accurate BDP estimation allows HLDG to achieve higher throughput while maintaining low queuing delay. This result can be generalised to any delay-based CC. Using a fraction of estimated BDP to set the congestion window size following back-off can help ensure that bytes in flight remains close to the path BDP.

5.2.6 RTT_{base} estimation

RTT_{base} estimation (RTT_{ebase}) is considered a challenging problem for many delay-based CC algorithms that use this estimation to infer congestion [56]. An inaccurate RTT_{ebase} estimation can lead to many issues including fairness issues, link underutilisation and undesirable latency depending on how the CC algorithm uses this estimation. As a result, accurate RTT_{ebase} is important.

For HLDG, a highly accurate RTT_{ebase} is not critical since BDP estimation is used only to improve link utilisation while operating in delay mode. However, different RTT_{ebase} estimates by multiple HLDG flows can lead to some degree of unfairness. In many realistic scenarios, HLDG flows can obtain a reasonable RTT_{ebase} because HLDG keeps queuing delay low by backing off *cwnd* below the estimated BDP whenever congestion is inferred. Typically, RTT_{ebase} can be estimated as the minimum RTT measured over the connection life-time.

In network environments experiencing frequent route changes (such as mobile networks), HLDG can underutilise the link if the route moves to a large RTT_{base} path. When this occurs, RTT_{base} is underestimated, causing the path BDP to be underestimated, resulting in HLDG backing off *cwnd* too far when congestion is detected. On the other hand, no problem occurs when the route changes from a longer to a shorter path since the estimator will observe the smaller RTT_{min} as soon as the route changes.

HLDG modification

HLDG estimates RTT_{base} over a window size of at least W_{rtt} seconds in addition to using a periodic RTT_{ebase} probing cycle.

To provide a better RTT_{base} estimation in such network environments, HLDG estimates RTT_{base} over a window size of at least W_{rtt} seconds in addition to using a periodic RTT_{ebase}

probing cycle. A simplified RTT_{ebase} estimation is shown in Equation 5.9.

$$RTT_{ebase} = min(RTT_t)\,\forall t \in [T_{begin}, T_{begin} + W_{rtt}] \tag{5.9}$$

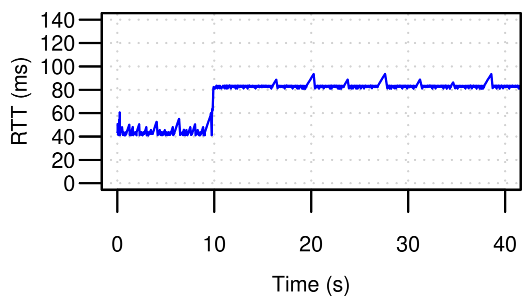
Where T_{begin} signifies the start of the RTT_{base} measurement period which is set to the event time of observing a lower RTT sample, and W_{rtt} is the measurement window length in second.

At the end of RTT_{base} measurement cycle (i.e. at $T_{begin} + W_{rtt}$), HLDG starts a new cycle by probing for a new minimum RTT value. The RTT_{base} probing phase begins by setting *ssthresh* to *cwnd* and reducing *cwnd* (default $0.8 \times cwnd$) [4]. This allows any existing queue to drain, allowing HLDG flows to obtain RTT samples close to RTT_{base}. Additionally, this prevents self induced queuing delay (similar to the problem in LEDBAT [6]) which leads to an increased standing queue. The 0.8 factor is a trade off between link utilisation loss and draining a very long queues (is unlikely due to HLDG keeping queuing delay small).

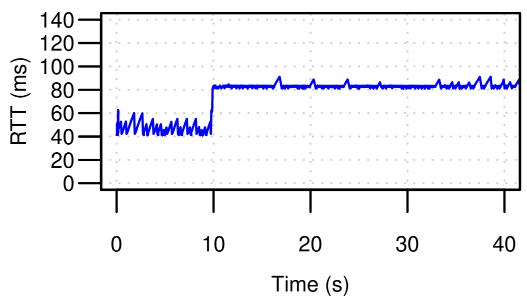
cwnd is restored using an exponential increase similar to the slow-start phase (i.e. increase *cwnd* by the number of acknowledged bytes for every ACK). Restoring *cwnd* using packet pacing can provide a smoother traffic pattern, preventing RTT spikes during the restoration phase. However, packet pacing adds complexity and overhead to the implementation and requires a high resolution timer which is not available in all operating systems. We suggest exploring packet pacing to restore *cwnd* as a future work.

Setting T_{begin} dynamically based on new minimum RTT observations allows HLDG flows to synchronise resetting RTT_{base} estimation. This allows many flows to obtain similar RTT_{base} estimates at the same time, leading to better fairness between competing flows.

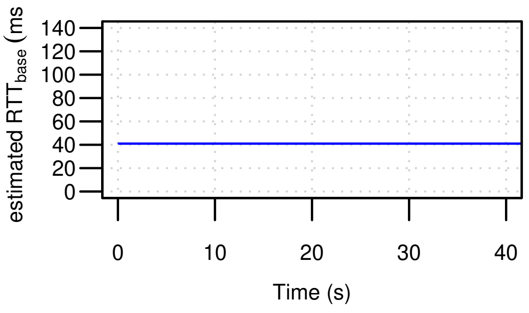
[4]if *cwnd* is smaller than one MSS, HLDG does not reduce it further since *cwnd* is small already



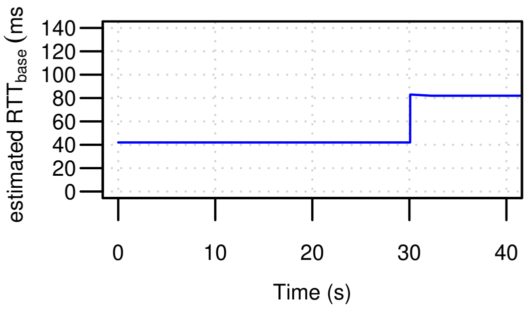
(a) SPP RTT versus time shows route change at t=10s



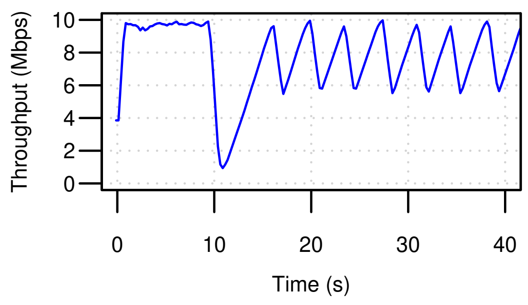
(b) SPP RTT versus time shows route change at t=10s



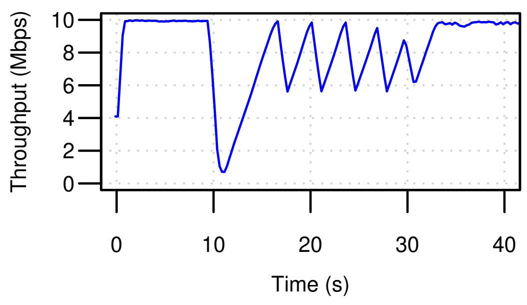
(c) Incorrect RTT_{base} estimation after route change



(d) RTT_{base} estimator produces correct estimation at t=30s after starting new RTT_{base} measurement cycle



(e) Flow throughput decreases after route change due to an incorrect BDP estimate



(f) Flow throughput increases after obtaining a correct RTT_{base} estimate

Figure 5.15: Route change results in HLDG flow to underutilise the bottleneck, but better bandwidth utilisation is achieved when using RTT_{base} measurement window method.

To measure the effectiveness of this RTT_{base} estimator, we conduct an experiment consisting of a single HLDG flow traversing a 10Mbps bottleneck. RTT_{base} is set to 40ms path RTT from t=0s to t=10s, changing to 80ms from t=10s to t=40s. Figure 5.15e shows the effect of the path change on HLDG throughput when no RTT_{base} measurement window is used. In Figure 5.15a, we see the change in RTT from 40ms to 80ms at t=10s. However, the RTT_{base} estimator underestimates RTT_{base} after the change as shown in Figure 5.15c. This results in

BDP being underestimated when backing of *cwnd*. As a result, HLDG backs off *cwnd* too far after the change, resulting in an underutilised link.

We can see that at t=10s, HLDG gets confused and backs off *cwnd* multiple times as the delay gradient increases significantly after the RTT change. Additionally, HLDG considers the back-off ineffective and does not zero the gradient window since no negative gradient is observed. As a consequence, the back-off probability remains high, causing HLDG to back off *cwnd* multiple times until the smoothing window excludes the large unsmoothed gradient. Also, due to the underestimated BDP, HLDG sets *cwnd* to a small value causing throughput to significantly decrease.

After applying an RTT_{base} measurement window method with a 30 second window size (W_{rtt}), we repeat the experiment. Figure 5.15f shows the effect of the RTT change on HLDG throughput when RTT_{base} measurement window is used. Figure 5.15b, shows the RTT change from 40ms to 80ms at t=10s. The RTT_{base} estimator produces an underestimated RTT_{base} of 40ms until time t=30s when a new RTT_{base} measurement cycle begins as shown in Figure 5.15d. This results in HLDG using a correct BDP estimate when backing off *cwnd*, utilising available bandwidth more efficiently.

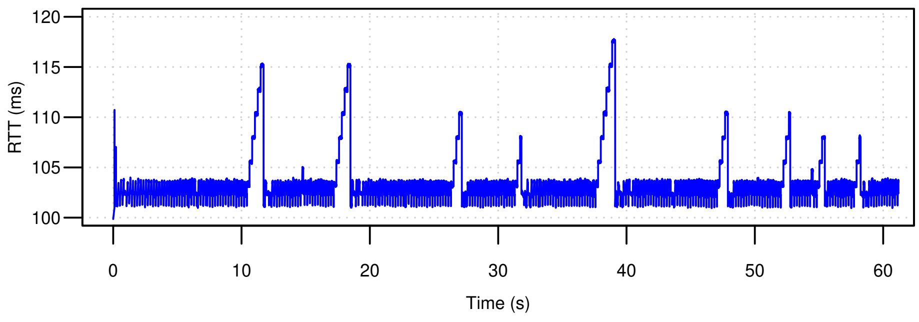
5.2.7 Eliminating queuing delay spikes

As mentioned in Section 4.2.3, the probabilistic nature of the back-off function and the gradient smoothing window mechanism occasionally result in CDG making late back-off decisions, leading to small RTT spikes. These issues cause the back-off probability to increase slowly or stop increasing while queuing delay increases.

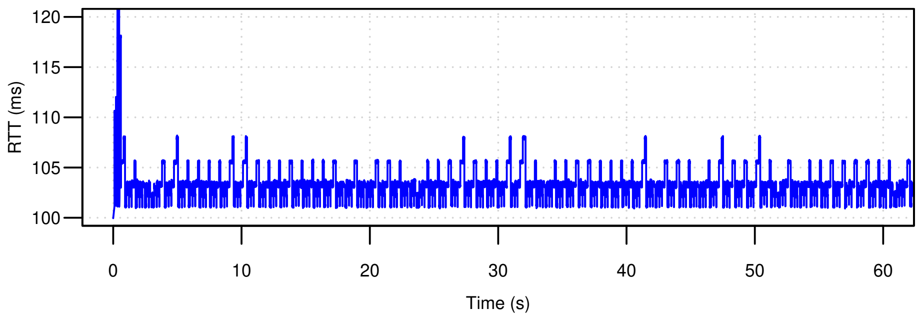
HLDG modification

HLDG accumulates back-off probabilities to allow it to be more responsive to increasing delay.

To address the RTT spike issue, HLDG modifies the CDG back-off decision function to make HLDG more responsive to increasing delay. Instead of using the calculated back-off probability directly, HLDG accumulates (sums) the probabilities and uses the sum to make its back-off decision. More precisely, the sum of probabilities $\bar{P}(n)$ is calculated as per Equation 5.10 where $\bar{P}_n$ is either $\bar{P}(\bar{g}_{min,n})$ or $\bar{P}(\bar{g}_{max,n})$, and m is the number of probability samples since the last back-off event. $P(\bar{g}_i)$ is either $P(\bar{g}_{min,i})$ $P(\bar{g}_{max,i})$ which are calculated as per Equation



(a) CDG



(b) HLDG

Figure 5.16: RTT spikes are eliminated when accumulated probability is applied for HLDG

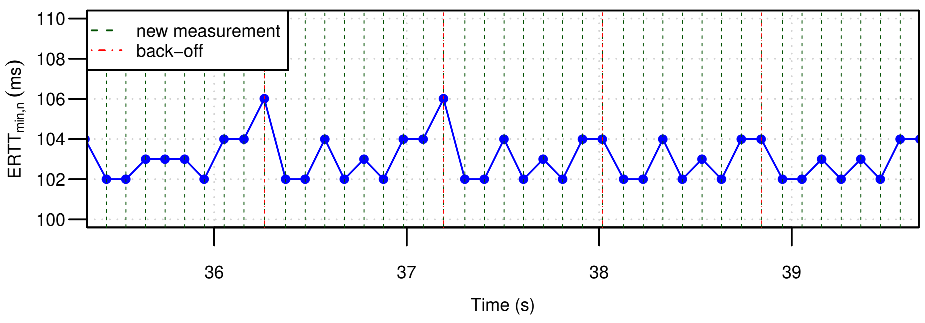
5.3. HLDG resets $\bar{P}(n)$ when $P(\overline{g}_i)$ becomes zero to prevent an unnecessary back-off.

$$\bar{P}_n = \sum_{i=1}^{m} P(\overline{g}_i) \tag{5.10}$$

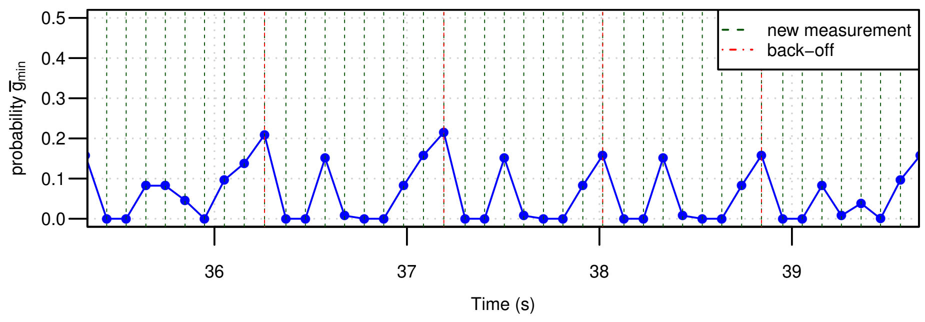
Similar to the CDG $P(\overline{g}_i)$ probabilistic back-off, if $\bar{P}_n$ is larger than an uniformed random number, HLDG backs off *cwnd*; otherwise it does not. Accumulating the probabilities reduces the chance of overshooting *cwnd* and minimises undesirable delay spikes and deviations in queuing delay.

We conduct an experiment where a 60 second HLDG flow traverses a bottleneck emulating 10Mbps and 100ms base RTT link to observe the effectiveness of the proposal. Figure 5.16b shows the RTT versus time plot for the experiment. We can see that the RTT spikes are both less frequent and with a lower magnitude than for CDG (Figure 5.16a). The average queuing delay for this experiment is similar at 2.3ms (compared with 2.7ms in CDG case), but the average throughput has increased to 9.78Mbps (from 7.76Mbps with CDG).

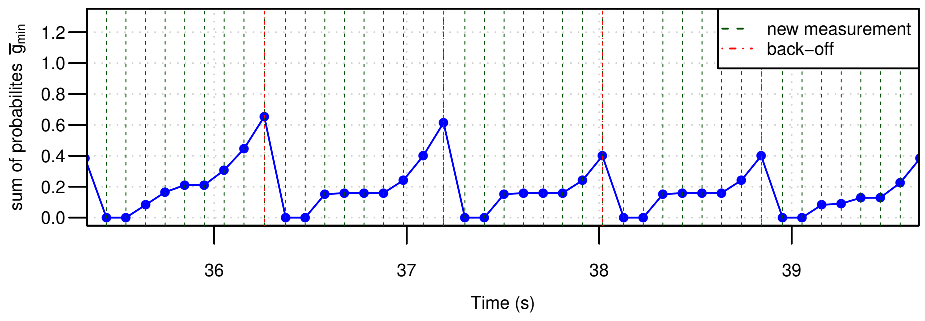
Figure 5.17 shows a close view of $ERTT_{min}$, back-off probability and accumulated back-off probability versus time for this experiment. The figures demonstrate that the accumulated probability increases smoothly and quickly as soon as a queue starts to form, allowing HLDG



(a) $ERTT_{min}$ versus time (unsmoothed gradient)



(b) Back-off probability



(c) Accumulated back-off probability

Figure 5.17: Using accumulated probability allows HLDG to make back off decisions in timely bases

to back off well before a long queue is created. In Figure 5.17c we can see that the accumulated probability feedback signal is less noisy than the new back-off probability shown in Figure 4.12c.

5.2.8 HLDG performance evaluation in single flow scenarios

In this section we evaluate and compare the performance of HLDG when the different standing queue solutions described in Section 5.2.4 are used. Additionally, we compare the performance of the HLDG+BDP method described in Section 5.2.5 with CDG.

We note that for small RTT_{base} paths, CDG does not suffer from low link utilisation. However, CDG flows are unable to fully utilise available bandwidth at higher path RTTs. HLDG can better utilise the bandwidth by addressing the unnecessary back-off problem. However, HLDG with a linear moving average smoothing mechanism produces a standing queue in small BDP paths. Using WWMA, ZSGW and backing off close to BDP can obtain lower queuing delays while preserving high link utilisation.

We start by comparing the solutions to the standing queue issue on small BDP paths. The experiment consists of a single 300s flow traversing a bottleneck emulating a 10Mbps link with 2ms path RTT. The experiment is repeated for CDG, HLDG with moving average, HLDG with WWMA, HLDG with ZSGW, HLDG with WWMA+ZSGW and HLDG with WWMA+ZSGW+backing off to BDP.

Table 5.1 shows the average, median and standard deviation of the queuing delay for all experiments. The results show that by using WWMA instead of linear moving average, HLDG reduces the standing queue by about 8ms. ZSGW method reduces queuing delay by a further 4ms. HLDG with WWMA+ZSGW provides high link utilisation and low queuing delay without relying on BDP measurement.

Backing off *cwnd* to $\lambda.BDP$ instead of $cwnd\beta_{delay}$ results in HLDG achieving a lower queuing delay (10.22ms) and very high link utilisation as bytes in-flight are kept close to the estimated BDP. When using the estimated BDP with WWMA and ZSGW, HLDG achieves very low queuing delay (6.9ms) whilst maintaining 99.9% bandwidth utilisation. WWMA allows HLDG to quickly respond to queuing delay growth, allowing HLDG to maintain a lower latency.

HLDG achieves extremely low latency and high stability with only 4.4ms average queuing delay while utilising 99.9% of the link bandwidth when accumulated probability, the estimated BDP, WWMA and ZSGW are used together. The accumulated probability prevents HLDG *cwnd* from overshooting BDP by too much, leading to low latency. At the same time, backing off *cwnd* to $\lambda.BDP$ prevents HLDG from reducing bytes in-flight too low than path BDP, realising higher throughput.

Figure 5.18 shows a queuing delay CDF plot for the experiment. The graph reinforces the results in Table 5.1 and illustrates that HLDG with WWMA+ZSGW+BDP produces very low queuing delay compared to the other algorithms.

For the rest of this thesis, we refer to **HLDG** as HLDG with accumulated probability+WWMA+ZSGW+BDP as this mode is the default operational mode of HLDG.

Next, we compare HLDG improvement over CDG in single flow scenarios. We conduct two experiments, one using CDG and one with HLDG using the testing scenarios from Section

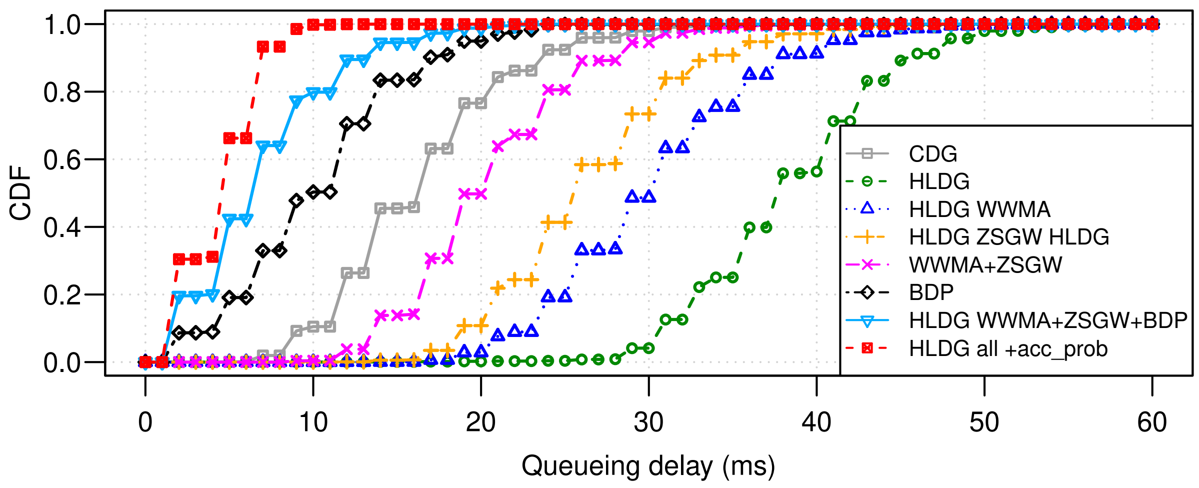


Figure 5.18: CDF for queuing delay for a single flow traverse a bottleneck BW=10Mbps and RTT_{base}=2ms

3.7. Each experiment consists of a 90 second `iperf` flow traversing a bottleneck emulating B_{rate}={1.5, 4, 12, 25, 50}Mbps links and RTT_{base}={0, 10, 40, 180, 240, 340}ms paths with $max(150 \times MSS, BDP)$ buffer size to prevent packet loss during the CA phase. We modified both CDG and HLDG to use loss-based slow start (instead of delay-based slow start) to allow the flow to converge to full bandwidth. The major metrics used in this comparison are throughput and queuing delay, all statistics are obtained from time t=30s to t=90s, allowing the flow to reach a stable state in CA mode.

HLDG throughput improvement percentage is shown in Figure 5.19. The figure shows that HLDG improves throughput by up to 400% over CDG in specific scenarios. For RTT_{base}<40ms,

Table 5.1: RTT mean, median and standard deviation and link utilisation for a single 300s flow traverses a bottleneck with BW=10Mbps and RTT_{base}=2ms

Algorithm	QD mean	QD median	QD SD	Utili. %
CDG	16.28ms	16.1ms	5.43ms	99.9%
HLDG moving Avg.	38.29ms	37.78ms	5.82ms	99.9%
HLDG w/ WWMA	30.09ms	30.47ms	6.26ms	99.9%
HLDG w/ ZSGW	26.3ms	25.74ms	5.79ms	99.9%
HLDG w/ WWMA+ZSGW	20.34ms	20.78ms	5.19ms	99.9%
HLDG w/ backoff to BDP	10.22ms	9.14ms	5.322ms	99.9%
HLDG w/ WWMA+ZSGW+BDP	6.9ms	6.51ms	4.4ms	99.9%
HLDG w/ accumulated probability & WWMA+ZSGW+BDP	4.4ms	4.15ms	2.26ms	99.9%

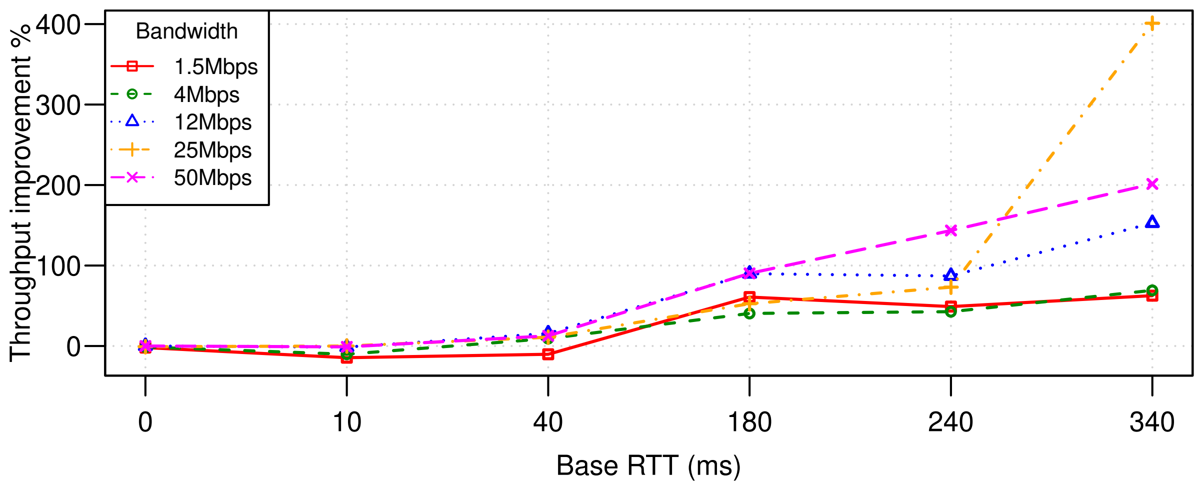


Figure 5.19: HLDG throughput improvement over CDG

HLDG and CDG obtain similar throughput for all bandwidths with slightly lower throughput for HLDG in specific scenarios. As RTT_{base} increases, HLDG provides better throughput improvement.

Queuing delay and throughput boxplots for the experiment are shown in Figures 5.20 and 5.21. We can see that HLDG achieves lower or similar queuing delay in general for almost all tests. On small path RTTs (< 40ms), HLDG introduces lower queuing delay especially at small bottleneck bandwidths. For example, at 4Mbps B_{rate} and RTT_{base} is 10ms, HLDG introduces around 10ms queuing delay while CDG keeps queuing delay over 15ms. As bandwidth increases, the queuing delay decreases as serialisation delay becomes smaller. Smaller transmission delays allow the gradient signal increase steps to be finer. Therefore, both HLDG and CDG can make back-off decisions before long queues build up.

Figure 5.21 shows that HLDG achieves higher throughput than CDG especially at higher RTT_{base}. At B_{rate} = 1.5Mbps, HLDG realises about 87% bandwidth utilisation when RTT_{base} is 10ms and 40ms, while CDG realises 90 - 100% bandwidth utilisation. At higher RTT_{base}, HLDG outperforms CDG significantly. At B_{rate} = 4Mbps, HLDG achieves 92% bandwidth utilisation when RTT_{base} is 10ms, while CDG realise 98% bandwidth utilisation. At higher RTT_{base}, HLDG outperforms CDG by up to 1.8Mbps. At B_{rate} = 12Mbps, HLDG outperforms CDG by up to 6Mbps when RTT_{base} is larger than 40ms. Similar results are observed when B_{rate} = 25Mbps. At 50Mbps B_{rate}, HLDG still performs better than CDG, especially when RTT_{base} is larger than 10ms.

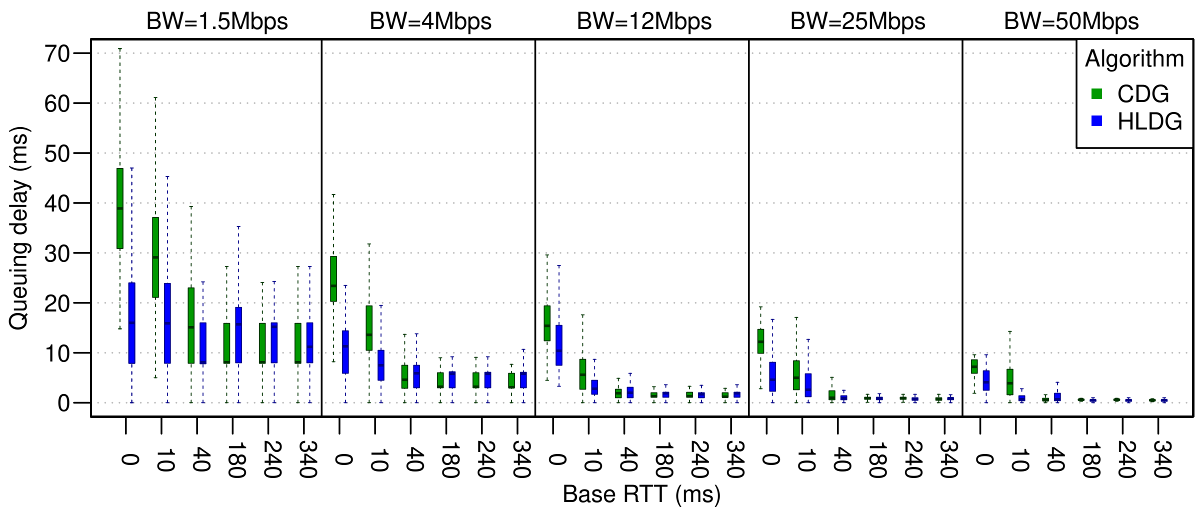


Figure 5.20: Queuing delay for HLDG and CDG

5.2.9 Summary of HLDG throughput improvements

Hybrid Loss-Delay Gradient (HLDG) is an improved hybrid TCP congestion control algorithm that achieves high link utilisation with low queuing delay. The algorithm is based on CDG CC algorithm but with enhancements to provide higher throughput in single flow scenarios and lower latency.

HLDG prevents unnecessary back-off which includes consecutive delay-based and spurious back-off. HLDG also prevents consecutive delay-based back-off by synchronising the smoothed gradient signals ($\bar{g}_{min,n}$ and $\bar{g}_{max,n}$) after back-off. This causes the smoothed gradient to include a negative gradient after back-off. This negative gradient sample reflects the *cwnd* decrease, indicating that the action is effective in reducing queuing delay. Smaller smoothed gradients decreases the probabilistic back-off, which inhibits multiple back-offs.

Spurious back-off occurs due to the noisy delay signal, delayed acknowledgement mechanism, and asynchronism between packet transmission at the sender side and bottleneck packet scheduler. HLDG mitigates this problem by allowing 1.2ms additional queuing delay without backing off *cwnd* to filter out noise.

HLDG uses a windowed moving average with weights (WWMA) instead of a linear moving average to quickly reflect the changes in delay gradient signal. Additionally, it zeros the smoothing gradient window (ZSGW) samples after backing off *cwnd* and negative gradient is observed. This restarts the smoothed gradient measurement, allowing valid delay gradient samples to be used in a new smoothed gradient. Using WWMA and ZSGW together reduces standing queues significantly.

Moreover, HLDG backs off *cwnd* to a portion of the estimated BDP instead of using the

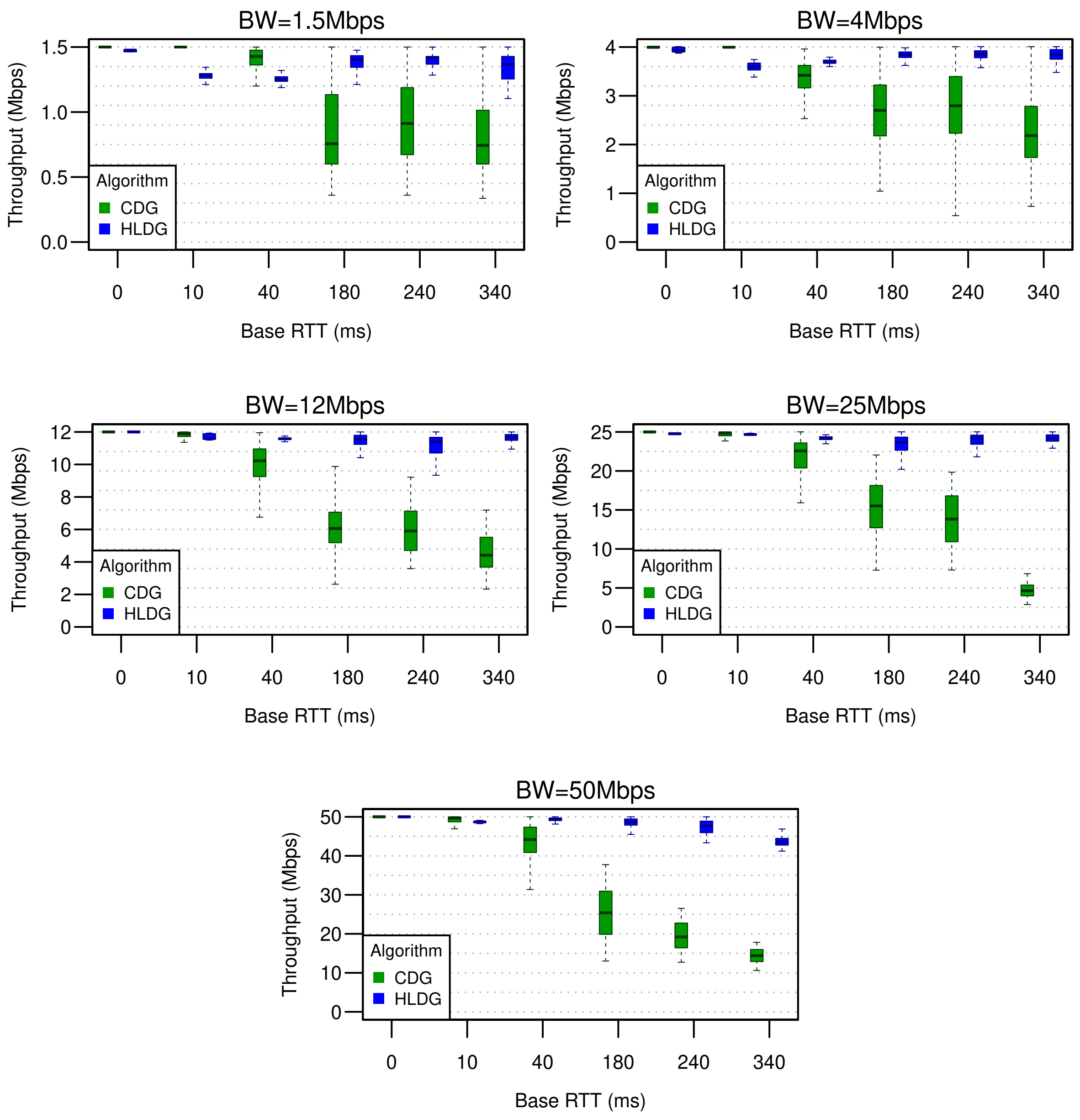


Figure 5.21: Throughput for HLDG and CDG flows

β_{delay} factor. This allows HLDG to realise high throughput in large RTT paths. The BDP is estimated by measuring bottleneck bandwidth using the acknowledged byte rate over an RTT interval and the estimated RTT_{base}.

In addition, HLDG controls queuing delay better than CDG, preventing *cwnd* from increasing too far above BDP. HLDG accumulates back-off probabilities and makes back-off decisions on the accumulated values to more quickly respond to any latency increase before long queues form. It also produces a smoother congestion feedback signal, leading to better congestion control.

The evaluation results show that HLDG achieves up to 400% (54% in average) throughput improvement over CDG in single flow scenarios without packet loss while maintaining queuing delay at no more than 20ms in the worst case. Therefore, HLDG congestion control can be used instead of CDG in many application that require high throughput without negatively impacting delay-sensitive applications such as VoIP and multiplayer online gaming.

These experimental results allow us to draw some general conclusions with regards to delay-based CC that can be deployed to improve all such algorithms:

1) The delayed acknowledgement mechanism causes a noticeable increase in the delay metric which can lead to incorrect decision making by the CC algorithm. As such, all delay-based CC should filter out these increases to prevent unnecessary back-off.

2) Using a WWMA filter instead of a simple moving average to smooth delay samples. This allows delay-based CC to more quickly respond to changes in delay while also eliminating sharp delay transitions (eg. after back-off)

3) Use of a good BDP estimation (eg. using acknowledged by rate with path RTT) to reset the congestion window following backoff. This allows delay-based flows to achieve higher throughput while maintaining low queueing delay.

5.3 Enhancing HLDG coexistence with loss-based flows

HLDG provides low latency transport with high throughput and almost no packet loss. It achieves this by keeping the number of packets in the queue very low. Despite these desirable properties, HLDG suffers when competing with loss-based TCP similar to many delay-based techniques such as TCP Vegas [34].

Some solutions have been proposed to provide better coexistence between delay-based and loss-based TCP [84, 86, 101–103, 172–175]. However, these solutions are either designed based on algorithm specific characteristics, or too complicated to be implemented in many systems.

Deploying AQM in the bottleneck can remedy the issue to some degree [173]. AQM sends congestion signals to the sender explicitly (mark packets) or implicitly (drop packets) when a standing queue starts building. Therefore, loss-based flows back off early giving an opportunity for delay-based flows to increase their *cwnd* and obtain a better bandwidth share. Additionally, bottleneck buffer occupancy will be shared more fairly since loss-based flows cannot overfill in the buffer due to AQM control polices. Using a scheduler/AQM hybrid scheme such us FQ-CoDel [16] and FQ-PIE [17] can provide better coexistence between loss-based and delay-based flows due to the flow separation mechanism.

As mentioned in 4.3.1, CDG coexistence mechanisms (the ineffectual back-off and shadow

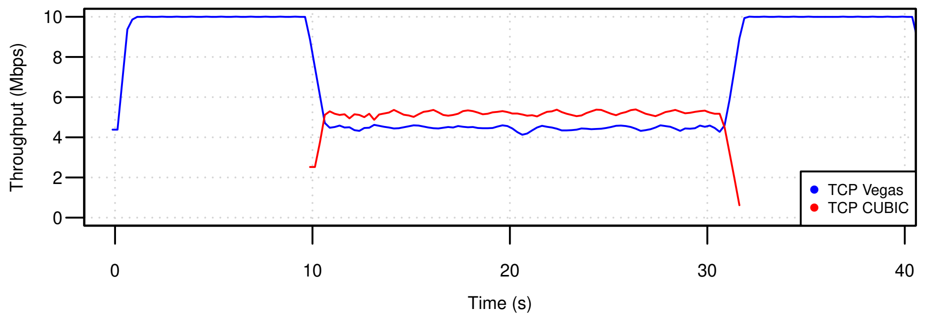


Figure 5.22: A TCP Vegas flow and a TCP CUBIC flow competing for 10Mbps bottleneck capacity

window mechanisms) are not able to allow CDG to obtain acceptable capacity sharing. Therefore, we propose two additional mechanisms to improve HLDG coexistence with loss-based TCP. These improvements allow a HLDG flow to obtain reasonable bandwidth sharing without adding much complexity to the implementation. Additionally, we disabled the non-congestion loss tolerance mechanism to allow HLDG to use the shadow window whenever packet loss occurs to improving coexistence.

5.3.1 G parameter adaptation

Similar to CDG, HLDG uses a scaling factor G to control the sensitivity of the back-off function to delay changes. The algorithm aggressiveness can be controlled through changing G parameter. By default, CDG uses G=3 factor to reduce false positive back-off due to delay signal noise.

A large G reduces the number of back-offs as the probabilistic back off function will be less responsive to the gradient increase. Tangenes et al. [107] confirm this behaviour and found that increasing G makes CDG more aggressive and achieve a better bandwidth share when competing with loss-based flows. However, the impact of the G parameter on the HLDG coexistence capability has not been studied.

Since HLDG uses probability accumulation to make its back-off decision, it impacts on HLDG coexistence differently. We evaluate HLDG in a range of scenarios to understand the effect of G on the coexistence between HLDG and loss-based flows, namely CUBIC.

Our experiment consists of one HLDG and one CUBIC flow competing for bottleneck bandwidth. The bottleneck emulates a link with B_{rate} ={1.5, 4, 12}Mbps, RTT_{base} = {10, 40, 180, 240}ms and buffer size = {1, 2, 4, 8, 16}BDP. Bottleneck bandwidth and the path RTT are chosen based on the testing scenarios in Section 3.7. We use G = {1, 3, 6, 12, 24, 48, 96,

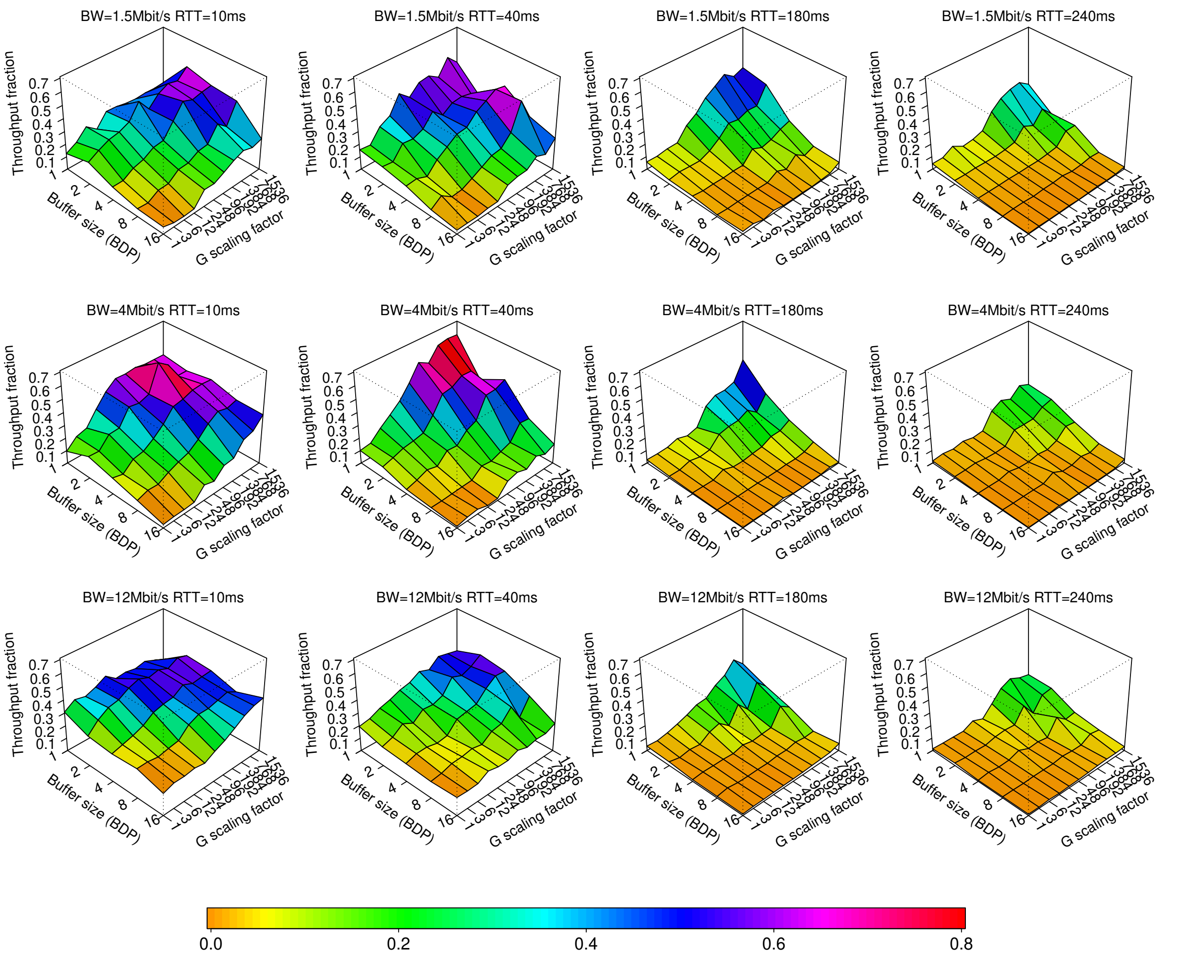


Figure 5.23: The effect of HLDG G factor on bandwidth sharing with CUBIC

192, 384, 768, 1536} in this experiment. The HLDG flow starts first, the CUBIC flow joins at t=10s and runs for 50 seconds. Traffic is generated using the `iperf` tool.

Figure 5.23 shows the average throughput fraction for the HLDG flow during the competition period. The fraction is calculated as per Equation 5.11 where T_{start} and T_{end} are the start and end times. We use T_{start}=20s and T_{end}=60s to avoid any transition phase caused by CUBIC flow start-up.

$$avg\_throughput\_fraction = \frac{\sum_{t=T_{start}}^{T_{end}} Throughput_{HLDG,t}}{B_{rate}} \tag{5.11}$$

The results reveal that higher G values allow HLDG to compete better with CUBIC. They also show that the bottleneck buffer size plays a significant role in HLDG coexistence with loss-based flows. As buffer size increases, HLDG obtains a lower bandwidth share. This observation is in-line with the CDG coexistence results from Section 4.3.2. This is due to buffer size increasing proportionally with RTT when set to a multiple of BDP. Large buffer sizes allow CUBIC to push more packets into buffer than HLDG as HLDG not only backs off due to packet loss, but also based on the delay gradient signal. As a result, more CUBIC packets will be forwarded in the bottleneck.

Another cause of this behaviour is the larger multiplicative decrease factor and faster *cwnd* growth function of CUBIC. CUBIC backs off *cwnd* using a 0.7 beta factor and uses a cubic *cwnd* increase function while HLDG uses 0.5 beta factor and increases *cwnd* once every RTT (similar to NewReno). Therefore, HLDG needs more time than CUBIC to recover *cwnd* above BDP. This gives an opportunity for CUBIC to push more packets and achieve higher throughput.

We can see in Figure 5.23, when $G > 12$, RTT_{base} is 10ms and buffer size is smaller than 8 BDPs (80 packets), HLDG flow coexists well with CUBIC. When $G < 192$ and buffer size > 8 BDPs, HLDG obtains a low bandwidth share (< 20%). The CUBIC flow gains bandwidth as RTT_{base} increases. At 240ms RTT_{base}, HLDG is unable to obtain more than 20% of bandwidth, even at one BDP buffer size. 25Mbps bandwidth results reveal similar behaviour to 12Mbps bandwidth with slightly better HLDG performance.

Despite the improved performance that can be obtained with larger a G factors, they cause a significant negative impact on queuing delay. To understand the impact of G factor on the queuing delay, we measure the average queuing delay for the first 10 seconds of each run of the same experiments during this period. The average queuing delay only relates to the HLDG flow.

Figure 5.24 plots the average queuing delay and shows that a large G factor prevents HLDG from maintaining a low queuing delay, especially with short paths. The results also show that as RTT_{base} increases, the average queuing delay decreases. This is caused by HLDG needing a longer time to recreate a high queuing delay after backing off as *cwnd* only grows by one MSS per RTT. Therefore, the queue is kept short for a longer period of time resulting in lower queuing delays.

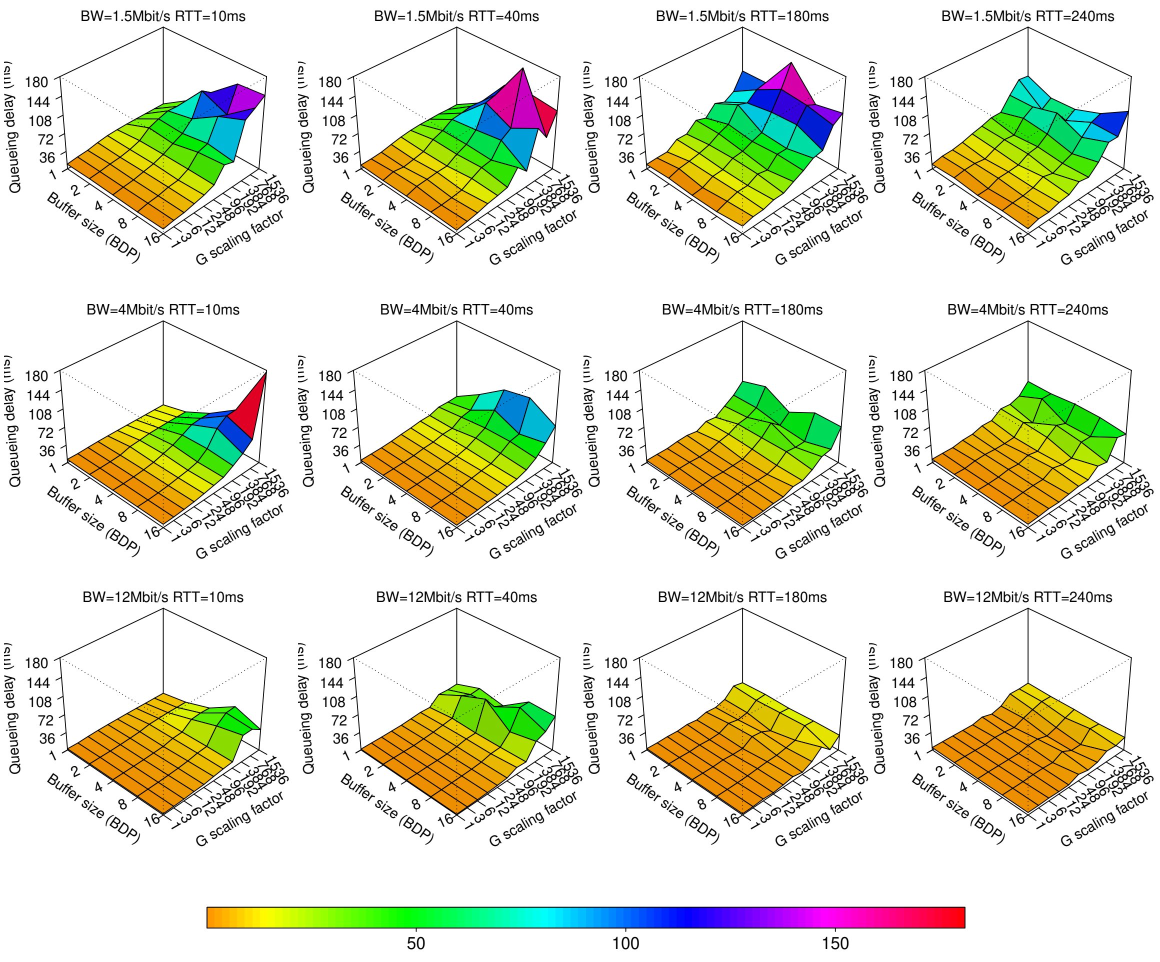


Figure 5.24: The effect of HLDG *G* factor on queuing delay

From this experiment we can conclude that it is impossible to select a constant *G* value that results in HLDG providing both low latency and acceptable coexistence with loss-based flows. However, if G adaptively increases when a competing loss-based flow is detected, HLDG can achieve better coexistence with loss-based flows.

Since HLDG does not cause high queuing delays, competing loss-based flows can be inferred if queuing delay exceeds a specific threshold (e.g. 30ms).

HLDG modification

HLDG calculates the *G* parameter dynamically based on estimate queueing delay when competing loss based flows are detected.

When a competing loss-based flow is detected, the *G* parameter is calculated dynamically based on the estimated queuing delay. Otherwise the default *(G* = 3) is used to provide low

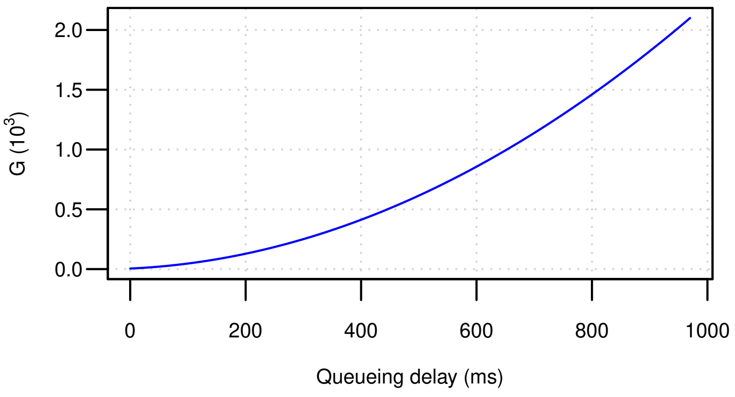


Figure 5.25: HLDG adaptive G function

latency transport. The rationale for this adaptation is to keep queuing delay low for normal HLDG operation. However, HLDG becomes more aggressive when we infer the presence of competing loss-based flows through monitoring the estimated queuing delay. Queuing delay is estimated using Equation 5.12 where RTT_{ebase} is the estimated base RTT.

$$Q_i = RTT_i - RTT_{ebase} \tag{5.12}$$

Using the data from Figure 5.24, coupled with the maximum queuing delay for each buffer size, we applied a curve fitting algorithm to generate the exponential function shown in Figure 5.25. We scaled down the function coefficients to reduce the impact on queuing delay. This is a preliminary approach, further work is required to tune and improve this function. This allows HLDG to better compete with the loss-based flows without impacting queuing delay when no loss based flows are present.

Although this approach can improve HLDG coexistence, it does not realise an equal bandwidth share. There is always a high probability for HLDG flows to back off before loss-based flows as HLDG backs off using the delay signal as well as packet loss. Therefore, CUBIC flows keep increasing their *cwnd* and backing off well after HLDG flows. As a consequence we also propose an explicit mode switching for HLDG to remedy this problem in environments with large buffer sizes.

5.3.2 Three operation modes (delay, loss and probing)

In the previous section we explored the impact of the HLDG G parameter on coexistence with loss-based flows. Our proposal was to use an adaptive G to improve performance. This technique can improve HLDG coexistence in low RTT paths with small buffer sizes but not for large buffer sizes or large path RTT.

Dual-mode hybrid loss-delay congestion control algorithms have been proposed to remedy the poor coexistence performance of delay-based CC. These algorithms explicitly switch to loss-based regime when loss-based competing flows are detected. We reviewed well-known algorithms of this category in Section 2.4.2. The switching mechanisms used by those proposals either do not fit with HLDG goals or the design, or too complicated to be implemented in many systems.

HLDG modification

HLDG employs a new mode mode switching technique.

Therefore, we propose an new explicit mode switching technique that allows HLDG to switch between delay-based and loss-based mode. Switching to loss-based mode allows HLDG to achieve average throughput similar to loss-based flows. The decision to switch between delay and loss mode is made based on queuing delay measurements.

In delay mode, HLDG backs off when congestion is detected using the delay gradient signal and packet loss signal. Additionally, HLDG keeps using the ineffective back off and shadow window mechanisms in this mode. In loss mode however, HLDG becomes a fully loss-based algorithm and backs off *cwnd* only upon packet loss events. In this mode HLDG behaves like TCP CUBIC [49] congestion avoidance mode. We select CUBIC *cwnd* growth function because of its fast convergence[49], high throughput, and wide deployment across the Internet.

This improves the initial performance of HLDG which might otherwise exit slow start early due to observing queuing delay spikes created during slow start. Exiting early results in HLDG starting with too-low *cwnd*, taking time to recover to a fair share.

To avoid synchronised slow-start of HLDG and loss based flows causing heavy packet losses, HLDG delays switching to loss mode by **one** second. This give a reasonable time for competing loss-based flows to exit their slow-start phase before HLDG enters its slow-start phase.

When packet loss is detected, HLDG enters what we call probing mode for a short period of time to check if loss based flows are still present. If no competing loss-based flows are detected, HLDG returns to normal delay-based mode; otherwise it re-enters loss mode.To reduce the number of mode-switching events, HLDG stays in loss mode for at least **5** seconds. During these 5 seconds, HLDG does not enter probe mode or switch back to delay mode.

The mode switching mechanism involves:

1. Detecting competing loss-based flows.

2. Switching to loss mode.

3. Probing and detecting leaving loss-based flows.

4. Switching back to the delay mode or moving back to the loss mode.

5.3.2.1 Detecting competing loss-based flows

As previously described, HLDG maintains low queuing delay during congestion avoidance phase. However, loss-based flows push queuing delay much higher, especially with large bottleneck buffer sizes. We exploit this to infer the presence of competing loss based flows.

HLDG monitors the estimated queuing delay for a short period of time (a few RTTs). If the queuing delay remains high, HLDG infers that a competing loss based flow is present.

More precisely, it compares the estimated minimum queuing delay $Q_{min,n}$ measured during thw previous RTT with a threshold (Qth_{max_delay}) once every RTT interval[5]. $Q_{min,n}$ is the difference between the estimated path base RTT (RTT_{ebase}) and $RTT_{min,n}$, as per Equation 5.13.

$$Q_{min,n} = RTT_{min,n} - RTT_{ebase} \tag{5.13}$$

If all $RTT_{min,n}$ samples are larger than Qth_{max_delay} for the previous m RTTs, we consider competing loss-based flows to be present.

The rationale of this approach is that HLDG flows should not introduce queuing delay larger than Qth_{max_delay}. However, loss based flows introduce higher delays. To remedy the impact of queuing delay fluctuation, HLDG does not make an immediate decision based on the current $RTT_{min,n}$, instead, monitoring queuing delay for m RTT intervals to smooth the measurement.

The value of m impacts detection speed as well as sensitivity to delay signal noise. If a large m is selected, HLDG can take longer to detect competing loss based flows, especially with high RTT_{base} paths. Alternatively, if a small m is used, HLDG may enter loss mode when no loss-based flows are present. In our HLDG implementation, we use a default m=4 which appears to provide reasonably fast detection with a small number of false positives.

The value of Qth_{max_delay} controls the protocol sensitivity to queuing delay increase. If Qth_{max_delay} chosen is too small, it results in false positives and enters loss mode incorrectly. If the value chosen is too large, HLDG will not enter loss mode when the buffer size is not large enough to produce high queuing delays. Our default value of 100ms Qth_{max_delay} appears to work well to prevent false positive detection. HLDG can obtain better coexistence if a smaller

[5]In our implementation, we use ERTT new measurement flag to determine an RTT has elapsed.

Qth_{max_delay} is used. However, it causes HLDG to switch to loss mode more often, even when no competing loss-based flow is present.

It is clear that this technique works only if the buffer size can produce a queue delay higher than Qth_{max_delay} equivalent. Otherwise, HLDG will remain in delay mode and never switch to loss mode. Since HLDG provides acceptable performance for small buffer sizes, this method can be used to provide enhanced coexistence for HLDG in large buffer sizes.

5.3.2.2 Detect leaving competing loss-based flows and exiting loss mode

After entering loss mode, HLDG remains in this mode until all loss-based flows leave the bottleneck. When packet loss occurs while in loss mode, HLDG reattempts to detect competing loss-based flows. When no loss-based flows are detected, HLDG returns to delay mode.

HLDG counts the number of times that $Q_{min,n}$ is less than Qth_{max_delay} for the previous m RTTs. If at least 75% of the samples satisfy that condition, we assume loss-based flows are no longer present, and HLDG switches back to delay mode. Using a 75% threshold allows HLDG to exit loss mode even when a few $Q_{min,n}$ samples reveal a high queuing delay.

If m RTTs elapse and $Q_{min,n}$ is still high, HLDG returns to loss mode. Returning to loss mode starts by setting *cwnd* to the old *cwnd* value after backing off (i.e. *cwnd* * 0.7) plus the *cwnd* increase that would have occurred between packet loss and the restoration time. Instead of using a shadow window for this period to mimic the *cwnd* growth function, we use the CUBIC *cwnd* growth function directly. The CUBIC function provides the exact *cwnd* value at time t where t is the time since the last back off event[49].

Restoring the *cwnd* value directly can result in a large packet burst, causing multiple packet losses and a spike in queuing delay. Packet pacing can help in this situation to smoothly increase the sending rate. However, implementing packet pacing requires precise timers and modification to the TCP stack. Therefore, HLDG restores *cwnd* gradually over two RTT intervals in an attempt to reduce packet bursts. During the *cwnd* restoring phase, *cwnd* increases by one MSS every $T_{restore}$ using Equation 5.14.

The state machine governing HLDG mode switching is shown in Figure 5.26.

$$T_{restore} = \frac{2 \times RTT_i \times MSS}{(cwnd_{cubic(t)} - cwnd_i)} \tag{5.14}$$

There are two main issues with this technique.

1. When an HLDG flow switches to loss mode, it behaves similarly to normal loss based flows. Therefore, the estimated queuing delay will be always high even when competing loss based flows leave. As a result, $Q_{min,n}$ will not become less than Qth_{max_delay} since the delay-based back off does not occur.

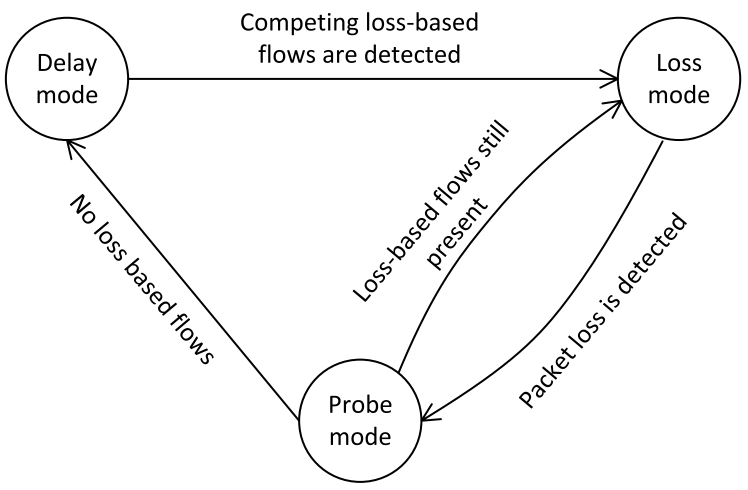


Figure 5.26: HLDG mode switching state machine.

2. If multiple delay-based flows join a bottleneck and one flow decides to switch to loss mode, all other flows will switch to loss mode as well. They will remain in loss mode even if some flows leave the bottleneck. This occurs as each flow assumes the other competing flows are loss-based and attempts to compete with them.

We propose to remedy the first issue by entering an intermediate mode (probe mode) in which the bottleneck is probed to see whether loss-based flows are present. In probe mode, HLDG sets *cwnd* to two MSS[6] and uses delay mode congestion control rules. This allows any built up queue to drain so the sender can see a lower queuing delay when no competing loss-based flows are present. HLDG remains in probe mode for an $m \times RTT$ interval, allowing HLDG to collect enough $Q_{min,n}$ samples to make a decision on whether to leave loss mode. We found that the packet loss events are good points to enter probe mode since these events already reduce queuing delay due to backing off *cwnd*.

To remedy the second issue, all HLDG flows should enter probe mode at the same time. This synchronisation provides an opportunity for any queue in the bottleneck to drain.

We exploit the fact the there is a high probability for some competing flows (working in loss mode and having a similar sending rate) to experience packet loss at the same time due to buffer overflow [176]. When many flows back off and enter the probe mode, there is a higher chance that the queuing delay will become low, allowing flows to observe a low queuing delay and exit loss mode.

[6]if *cwnd* is larger than that 2 MSS, *cwnd* is not changed.

In specific scenarios, the probe phase becomes longer than loss mode. This occurs for large buffer scenarios where CUBIC dynamic is slow. This situation negatively impacts HLDG throughput during the coexistence phase. We solve this problem by entering the probe mode only when the HLDG flow spends a longer time in loss mode than in probe mode.

5.3.2.3 HLDG mode switching algorithm

The pseudo code for the HLDG mode switching algorithm is shown in Algorithms 5.1 and 5.2. HLDG initially works in delay mode and a *detection_q* circular queue of size m is created.

$RTT_{min,n}$ is obtained by comparing old $RTT_{min,n}$ with current RTT (RTT_i). RTT_{ebase} is also obtained once every ACK (see Section 5.2.6).

During each RTT interval, $Q_{min,n}$ is calculated as the difference between $RTT_{min,n}$ and the estimated RTT_{ebase}. If $Q_{min,n} > Qth_{max\_delay}$, 100[7] is added to *detection_q;* otherwise 0 is added. If we have m samples in *detection_q*, HLDG tests the state of the competing flows.

Algorithm 5.1 Pseudo code for HLDG mode switching algorithm - packet loss event

```
1: On packet loss:
2: $loss\_epoch \leftarrow NOW - t\_in\_loss\_mode$
3: $inswitching\_epoch \leftarrow NOW - t\_in\_mode\_switching$
4: if $mode == MODE\_LOSS$ and $inswitching\_epoch > 10sec$ and loss_epoch > probe_epoch then
5:     $cwnd \leftarrow 2 * MSS$
6:     $in\_probe\_mode \leftarrow 0$
7:     $old\_cwnd \leftarrow cwnd$
8:     $mode \leftarrow MODE\_PROBE$
9:     $t\_in\_probe\_mode \leftarrow NOW$
10: end if
```

[7] We use integer arithmetic to allow us to implement the algorithm in OS kernel space.

Algorithm 5.2 Pseudo code for HLDG mode switching algorithm - main part

```
1: Initialisation :
2: $mode \leftarrow MODE\_DELAY$
3: $RTT_{min,n} \leftarrow RTT_{ebase} \leftarrow \infty$
4: $detection\_q \leftarrow circular\_queue(size = m)$

5: On each ACK:
6: if $RTT_i < RTT_{min,n}$ then
7:     $RTT_{min,n} \leftarrow RTT_i$
8: end if
9: if $RTT_i < RTT_{ebase}$ then
10:     $RTT_{ebase} \leftarrow RTT_i$
11: end if

12: On each RTT:
13: $Q_{min,n} \leftarrow RTT_{min,n} - RTT_{ebase}$
14: if $Q_{min,n} > Qth_{max}$ then
15:     $insert$ 100 $into\ detection\_q$
16: else
17:     $insert$ 0 $into\ detection\_q$
18: end if
19: if $mode == MODE\_PROBE$ then
20:     $in\_probe\_mode \leftarrow in\_probe\_mode + 1$
21: end if
22: if $length(detection\_q) == m$ then
23:     if $mode == MODE\_DELAY$ and $sum(detection\_q) == m * 100$ then
24:         After one second do:
25:             $mode \leftarrow MODE\_LOSS; ssthresh \leftarrow \infty$
26:             $t\_in\_mode\_switching \leftarrow t\_in\_loss\_mode \leftarrow NOW$
27:     else if $sum(detection\_q) \leqslant m * 25$ then
28:         $mode \leftarrow MODE\_DELAY$
29:     end if
30:     if $mode == MODE\_PROBE$ and $in\_probe\_mode \geqslant m$ then
31:         $mode \leftarrow MODE\_LOSS$
32:         $probe\_epoch \leftarrow NOW - t\_in\_probe\_mode$
33:         $t\_in\_loss\_mode \leftarrow NOW$
34:         RESTORE_CWND(old_cwnd)
35:     end if
36: end if
37: $RTT_{min,n} \leftarrow \infty$
```

If HLDG is operating in delay mode and all elements in *detection_q* indicate high queuing delay, it switches to the loss mode starting by slow-start phase (setting *ssthresh* to a large integer value). If 25% or fewer samples in *detection_q* indicate high queuing delay, HLDG switches to delay mode. On the other hand, if HLDG is in probe mode for at least *m* RTTs, HLDG returns to loss mode and restores *cwnd* according to CUBIC *cwnd* growth function. $RTT_{min,n}$ is set to a large integer number to start a new measurement during the next RTT cycle.

When a packet loss is detected, loss mode epoch and time since starting the switching mode are calculated. If HLDG is in loss mode and it has been more than 5 seconds since switching to loss mode and the HDLG flow has spent more time in loss mode than in probe mode, then HLDG sets *cwnd* two MSS and enters probe mode.

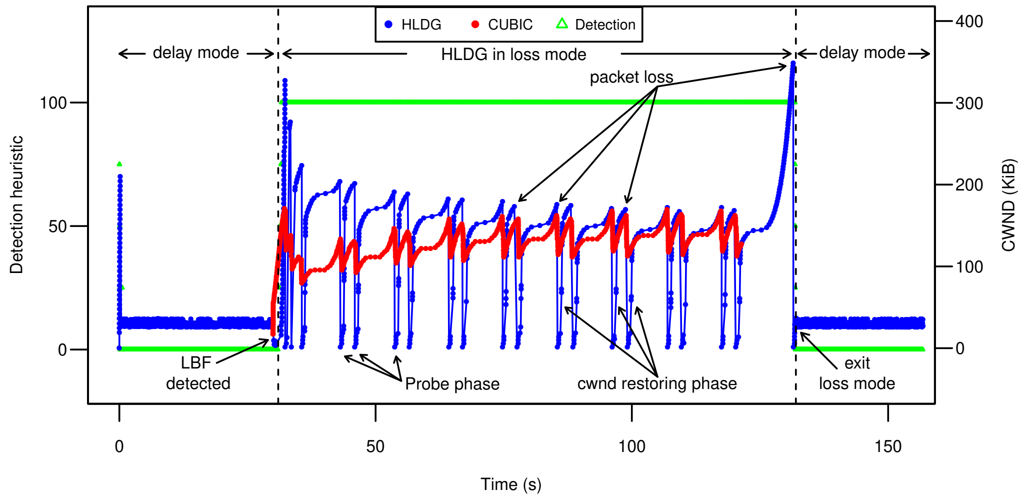
Figure 5.27 illustrates the algorithm in action. The figure shows *cwnd*, the detection heuristic, queuing delay and throughput versus time plots for a HLDG flow competing with a CUBIC flow over a 12Mbps bottleneck bandwidth with 10ms path RTT. The bottleneck buffer size is set to 250pkts (250ms queuing delay equivalent). The HLDG flow starts first with CUBIC starting at t=30s. The CUBIC flow finishes at t=120s and the experiment terminates at t=160s. *m* is set to 4, and Qth_{max_delay}=50ms (for illustrational purpose).

We can see in Figure 5.27 that HLDG operates in delay mode for the first 30 seconds before the CUBIC flow begins, achieving low queuing delay and full bandwidth utilisation. As soon as the CUBIC flow starts at t=30s, the detection heuristic becomes 100 as all $Q_{min,n}$ measurements in last the four RTTs are larger than Qth_{max_delay}. HLDG enters loss mode starting with slow-start phase after one second.

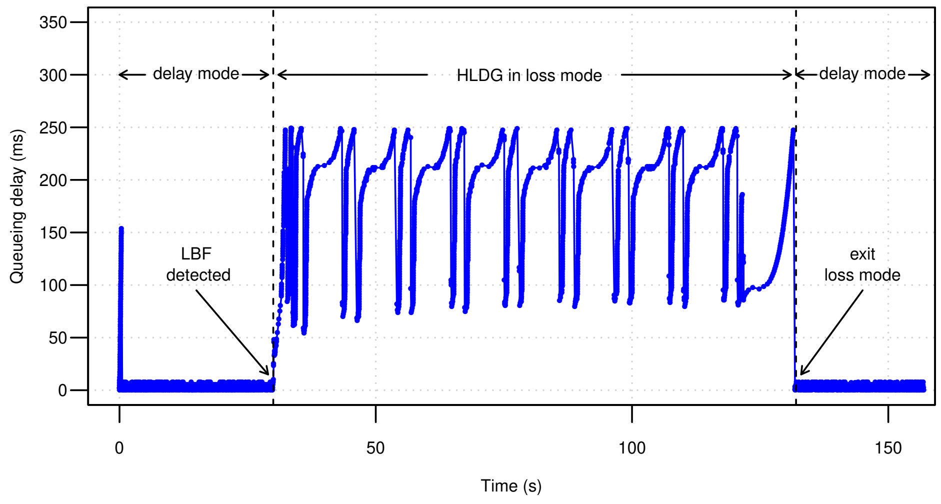
After 5 seconds and when packet loss is detected, HLDG enters probe mode to monitor if the loss-based flow has finished. As the detections heuristic remains high, HLDG switches back to loss mode. Both HLDG and CUBIC flows continue to back off and increase *cwnd* until they achieve a fair-share of the bandwidth. When the CUBIC flow terminates, the HLDG flow keeps operating in loss mode as queuing delay remains high. We can see in Figure 5.27b that queuing delay reaches up to 250ms during the competition period because both HLDG and CUBIC flows aggressively push packets to the buffer until packet loss occurs.

Since more bandwidth becomes available after the CUBIC flow finishes, HLDG exponentially increases *cwnd* due to the cubic function being used. This behaviour quickly results in packet loss, causing HLDG to enter probe mode. In probe mode, the detection heuristic reveals a low queuing delay and HLDG exits loss mode, returning to delay mode. HLDG will then continue in this mode, maintaining low queuing delay.

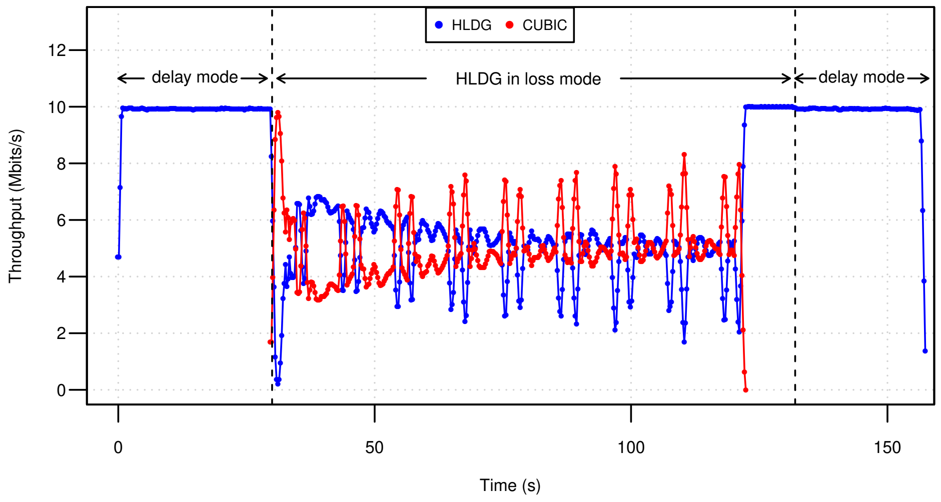
Figure 5.27c shows that both flows achieve very good bandwidth sharing in average. It also shows that when HLDG flows enters probe mode, it loses some bandwidth which is quickly



(a) Loss-based flows detection heuristic and *cwnd* versus time



(b) queuing delay versus time



(c) Throughput versus time

Figure 5.27: HLDG mode switching when competing with CUBIC

Table 5.2: HLDG bandwidth share percentage

		RTT_{base}**(ms)**				
		10	**40**	**180**	**240**	**340**
B_{rate}**(Mbps)**	**1.5**	47%	45%	40%	39%	36%
	4	57%	49%	47%	48%	32%
	12	52%	46%	44%	34%	40%
	25	43%	40%	25%	35%	21%

taken by CUBIC. However, HLDG recovers as soon as *cwnd* is restored.

5.3.3 HLDG coexistence Evaluation

In the previous section we proposed an adaptive scaling parameter coupled with a new operational mode selection state machine in order to improve HLDG performance when coexisting with loss-based flows. We also ran a simple experiment to verify its functionality. In this section we perform a complete evaluation of our algorithm.

We conduct experiments consisting of one HLDG flow competing with one CUBIC flow for bottleneck capacity. The bottleneck emulates {1.5, 4, 12, 25}Mbps bandwidth, RTT_{base} = {10, 40, 240}ms, with buffer size = {1, 2, 4, 6, 8}BDP. The HLDG flow starts first and runs for 150 seconds. CUBIC starts at t=10s and runs for 100 seconds. Traffic is generated using the `iperf` tool.

We measure average throughput for each flow individually for the period between t=20s and t=110s (i.e. 10 seconds after the CUBIC flow begins) to give time for flows to stabilise. Figure 5.28 plots the results for this experiment.

In general, the performance of HLDG is significantly better than that for CDG (see Figure 4.16) at large buffer sizes that would be typically found in the home environment.

At 1.5Mbps bandwidth the HLDG flow achieves approximately equal or higher average throughput than CDG for all scenarios.

Generally speaking, HLDG flow achieves higher utilisation than CDG at 4 Mbps. HLDG performance decreases with small buffer sizes at 10ms RTT (< 6 BDP buffers at 10ms RTT and < 4 BDP buffer at 40ms RTT). At higher RTT and buffer sizes, HLDG achieves fair bandwidth share except at 240ms RTT and 8 BDP (640 packets) buffer size. HLDG is unable to achieve parity due to the very high queuing delay. In this case, the queuing delay can reach 1920ms, causing HLDG to enter probe mode for 7680ms on packet loss. This gives an opportunity for the CUBIC flow to utilise more bandwidth, decreasing HLDG throughput

Table 5.3: CDG bandwidth share percentage

		RTT_{base}(ms)				
		10	**40**	**180**	**240**	**340**
B_{rate}(Mbps)	**1.5**	21%	16%	13%	11%	11%
	4	8%	8%	5%	6%	5%
	12	5%	4%	4%	3%	5%
	25	2%	3%	2%	2%	0.6%

significantly.

At 12 and 25 Mbit/s, HLDG performance is largely better than CDG except for low RTT coupled with small buffers.

Figure 5.29 plots the throughput improvement of HLDG over CDG when competing with CUBIC for bottleneck bandwidth. When RTT_{base} = 10ms, HLDG provides improvement up to 120% over CDG at 1.5Mbps. However, HLDG performs worse than CDG when bandwidth is {4, 12, 25}Mbps and buffer size is {1, 2, 4}BDP.

When $RTT_{base} = 40ms$, HLDG achieves throughput up to 22 times better than CDG under same conditions. HLDG provides much better coexistence when buffer size > 2 BDPs. For small buffer size scenarios, HLDG and CDG perform similarly.

At $RTT_{base} = 240ms$, HLDG clearly improves performance at all buffer sizes and bandwidths due to its mode switching technique. HLDG is able to achieve up to 4500% more throughput than CDG when bandwidth is 25Mbps and buffer size is 2 and 6 BDP.

Although HLDG provides more achievable throughput than CDG, it still cannot achieve ~50% when coexisting with loss-based flows under many scenarios (see Figure 5.28). More work is needed to further improve HLDG coexistence mechanisms. Even so, HLDG is still able to perform reasonably in many realistic scenarios and does not starve.

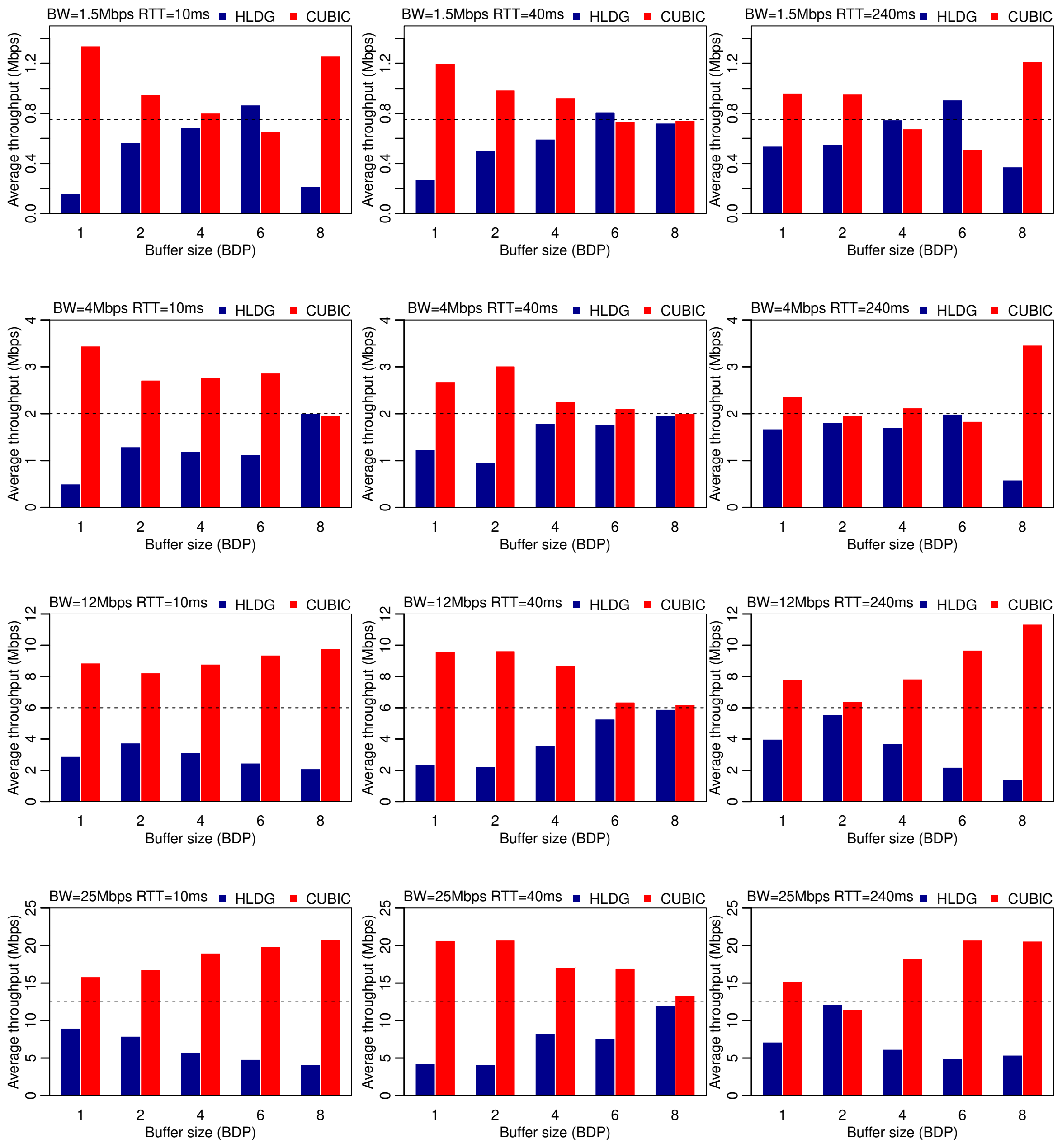


Figure 5.28: HLDG does not starve with compete with CUBIC

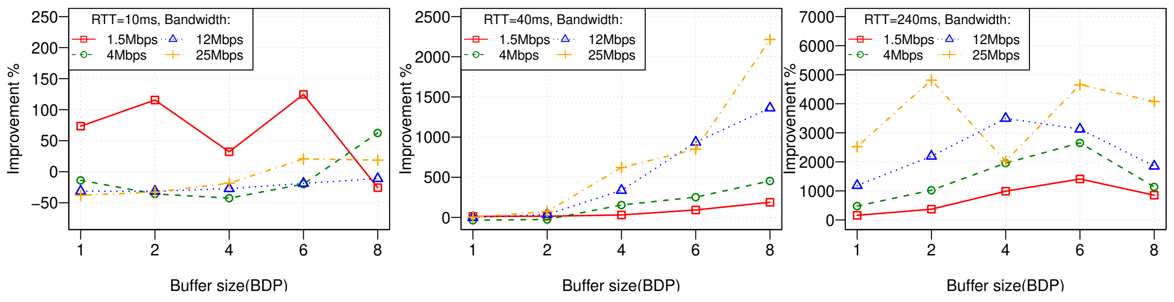


Figure 5.29: HLDG throughput improvement over CDG when coexisting with a CUBIC flow

We note that HLDG can achieve better throughput if HLDG backs off *cwnd* to the maximum of $cwnd_i \times \beta_{delay}$ and $BDP \times \gamma$ (as mentioned in section 5.2.5). However, we did not enable this option in our experiment to ensure that HLDG introduces as low a queuing delay as possible.

As we evaluated CDG using a buffer size equivalent to 340ms in Section 4.3.1, we rerun this experiment with HLDG to observe the performance of HLDG under these network condition. This experiment consists of one HLDG flow competing with one CUBIC flow over B_{rate}={1.5, 4, 12, 25}Mbps bottleneck bandwidth. The bottleneck emulates paths with RTT_{base}={10, 40, 180, 240, 340}ms and has 340ms worth of bottleneck buffer size. The bottleneck bandwidth and RTT_{base} are selected based on the experimental methodology in Section 3.7. The HLDG flow starts first and the CUBIC flow starts at t=10s. The CUBIC flow runs for 100 seconds and the total experiment lasts for 150 seconds. Average throughput is measured for each flow individually for the period between t=20s and t=110s.

Figure 5.30 depicts HLDG throughput improvement over CDG when competing with CUBIC TCP. This figure reveals significant improvement for most testing scenarios except at 1.5Mbps. In these scenarios HLDG does not result in much improvement in throughput as

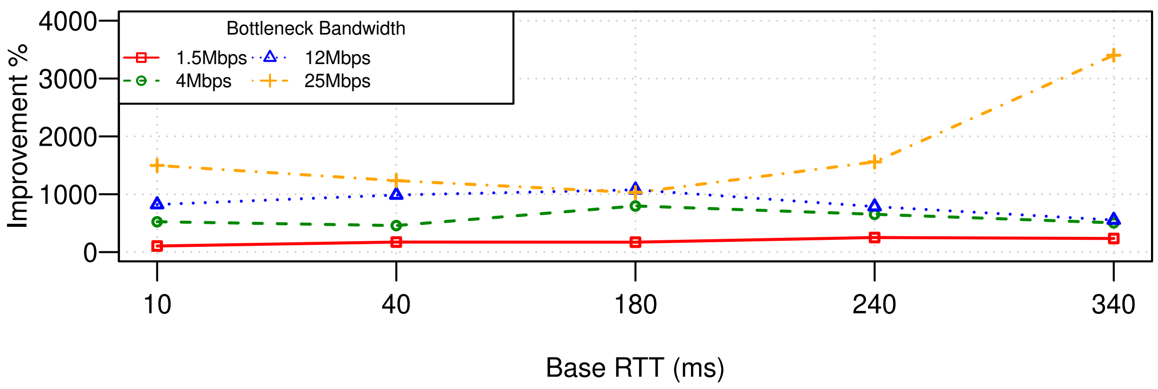


Figure 5.30: HLDG throughput improvement over CDG when coexisting with a CUBIC flow. Bottleneck buffer size is set 340ms × bandwidth.

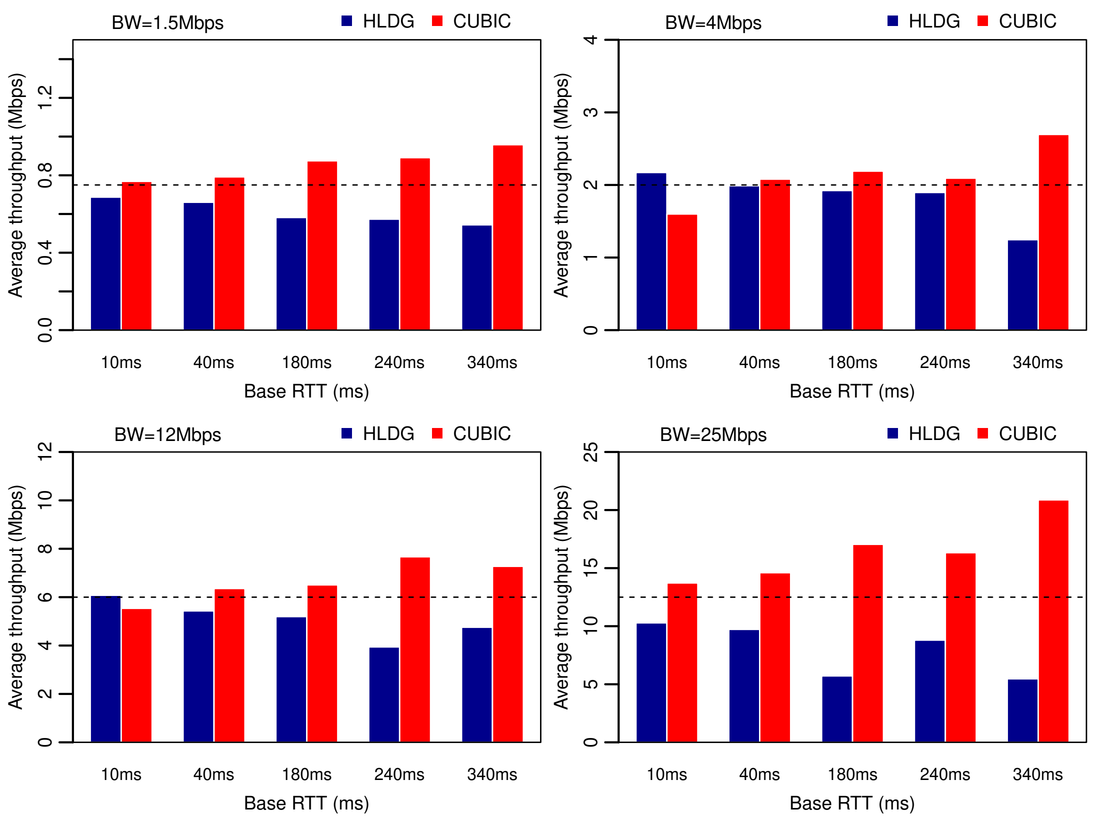


Figure 5.31: Bandwidth sharing between a HLDG and CUBIC flows competing for bottleneck capacity. Bottleneck buffer size is set 340ms × bandwidth.

CDG already achieves acceptable throughput in these conditions. On faster network links, however, HLDG outperforms CDG for all base RTTs.

Figure 5.31 compares the average throughput of HLDG and CUBIC flows for each scenario. In general, the HLDG flow realises between 25% to 52% bandwidth share in all scenarios. Compared to earlier CDG experimental results (shown in Figure 4.15), HLDG clearly achieves better performance. Large buffer sizes allow HLDG loss-mode to better utilise the buffer during the competition phase and achieve near-parity with CUBIC. Table 5.2 shows the bandwidth share percentage for the experiment calculated as HLDG throughput divided by the total utilised bandwidth. The results show that HLDG reduces its coexistence capability as the path BDP increases. However, HLDG achieves significantly higher performance for all scenarios compared with CDG results as shown in Table 5.3.

5.4 HLDG slow start

It is clear that Standard TCP SS side-effects are not in-line with CDG goals. Therefore, CDG uses the delay gradient signal to find a proper slow-start exit point before long queues build up in the bottleneck. As we previously discussed the issues of this mechanism in Section 4.4, CDG SS underestimates bottleneck capacity in large BDP paths, while it overestimates available capacity when path BDP is small. This leads to high queuing delay or low throughput respectively.

5.4.1 HLDG slow start algorithm

HLDG modification

HLDG avoids premature slow start termination by only using $g_{min,n}$ to exit slow start.

A remedy to CDG premature slow start termination (see Section 4.4) is to use only $g_{min,n}$ gradient measurement to exit SS while ignoring $g_{max,n}$ measurement. Using only $g_{min,n}$ protects against early exit due to temporary packet transmission bursts.

$g_{max,n}$ is ignored as this measurement is calculated based on $RTT_{max,n}$. $RTT_{max,n}$ represents the highest delay seen in an RTT and includes gradients of temporary delay spikes. Since the slow start phase involves packet bursts that increase every RTT. $g_{max,n}$ will be a large positive after a few RTTs, even when *cwnd* is well below BDP.

$g_{min,n}$ is calculated based on $RTT_{min,n}$ and it indicates the state of the queue regardless of temporary RTT spikes caused by TCP sending bursts. $g_{min,n}$ increases only when the queue grows and never drains during an RTT round. This occurs only when the bottleneck is actually congested. Moreover, we ignore the first $g_{min,n}$ sample since it is based on the bursty initial window transmission.

After patching the CDG SS code, we conduct the same experiment in Section 4.4. This experiment consists of a single modified CDG flow traversing a bottleneck that emulates links with B_{rate}={1.5, 4, 12, 25, 50}Mbps and RTT_{base}={10, 40, 180, 240, 340}ms. Bottleneck FIFO buffer size is set to 2000 packets and the Droptail mechanism is used. This large buffer size is selected to prevent packet loss since we explore the performance of the delay gradient congestion signal during slow start.

Table 5.4 shows the percentage of *ssthresh* to the path BDP $(ssthresh/BDP) \times 100$ for each test. We extract *ssthresh* just after the modified CDG switches from SS to CA phase. This table reveals that the modified CDG SS produces a better bandwidth estimation with less premature SS termination. However, slow start terminates late in many situations leading to large *cwnd*

Table 5.4: Ratio of *ssthersh* to path BDP for CDG SS after using only $g_{min,n}$ for exiting slow-start phase

		RTT_{base}**(ms)**				
		10	**40**	**180**	**240**	**340**
B_{rate}**(Mbps)**	**1.5**	2556%	641%	142%	165%	178%
	4	961%	240%	185%	139%	100%
	12	476%	265%	314%	71%	175%
	25	509%	286%	340%	264%	187%
	50	572%	323%	170%	304%	314%

and *ssthresh* values. This bottleneck bandwidth overestimation leads to unnecessary high latency and packet loss for shallow buffers. The cause is that modified CDG checks for SS exit points once every RTT. As *cwnd* doubles each RTT during SS, congestion detection occurs late, causing *cwnd* to be large. Additionally, some congestion events are randomly ignored due to probabilistic back off.

Our solution for late slow start termination is inspired by [130] and that HLDG already uses BDP estimation. The idea is to set *ssthresh* to the estimated BDP instead of an arbitrary large value during SS. This allows capping *ssthresh* to the upper limit to prevent high queuing delay.

As soon as a BDP estimation sample is ready, HLDG sets *ssthresh* to that value. BDP is calculated as $B_{erate} \times RTT_{ebase}$ where B_{erate} is the estimated bottleneck bandwidth and RTT_{ebase} is the estimated base RTT (see Section 5.2.6). During SS, *cwnd* increases according to standard TCP SS i.e. doubles every RTT.

HLDG SS relies on the passive ACK train technique similar to [72] instead of using a packet-pair technique [131] to estimate available bandwidth B_{erate} during SS. The packet-pair technique suffers from overestimation when there is cross traffic in the reverse path [132] and requires the sender to actively transmit a back-to-back train of packets to probe the path. The ACK train technique is similar to the bandwidth estimator from Equation 5.5 but using measurements over a shorter period than one RTT. A full RTT period is not usedfor the measurement is because TCP typically does not start transmitting with large *cwnd.* Many TCP implementations use an initial window of 10 MSS. If the measurement is performed over an RTT period, then the bandwidth will be $10 \times MSS/RTT$ which is less than the actual bottleneck bandwidth if BDP > $10 \times MSS$.

Since slow start already contains transmission bursts, we can exploit these bursts to measure the bottleneck bandwidth. We found that using 6 to 9 consecutive ACK samples within

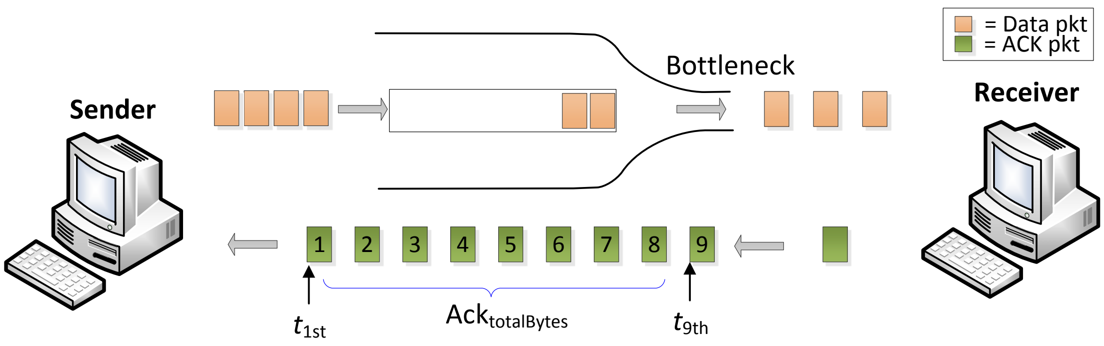


Figure 5.32: ACK train bottleneck bandwidth estimation

one transmission window can produce acceptable bandwidth estimation. By using an IW of ten MSS with delayed acknowledgement mechanism enabled, the consecutive ACKs are received within three RTT from the beginning of the connection. As such, *ssthresh* is set to BDP at the beginning of the third RTT. We use 8 ACK samples in our implementation.

Assume that t_{1st}, t_{9th} are the times at receiving first and ninth ACK in a window respectively. Further, assume $Ack_{totalBytes}$ is the total bytes acknowledged by the first eight ACKs. The estimated available bandwidth B_{erate} is calculated as per Equation 5.15. Figure 5.32 illustrates ACK train bandwidth estimator.

$$B_{erate} = \frac{Ack_{totalBytes}}{t_{9th} - t_{1st}} \tag{5.15}$$

HLDG modification

HLDG uses both an ssthresh limiting mechanism and delay gradient signal to terminate slow start.

HLDG SS uses both *ssthresh* limiting mechanism (using ACK train) and delay gradient signal to terminate slow start. The *ssthresh* limiting mechanism in SS only estimate predicts total bottleneck bandwidth when the bottleneck has a short or no queue.

Since HLDG maintains low queuing delay, bottleneck queues will include very few packets. Therefore, when a new flow joins a bottleneck and injects a train of packets, the queue will include mostly packets from the new flow. As a consequence, the ACK train arrives at the sender at a rate similar to if no other flows are sharing the bottleneck. This leads to overestimating *ssthresh* and a spike in queuing delay. Using the delay gradient signal to terminate SS can somewhat mitigate this issue since the sending rate increases gradually giving an opportunity for other flows to share the bandwidth.

Table 5.5: Ratio of *ssthresh* to path BDP for CDG SS after using only $g_{min,n}$ for exiting slow-start phase

		RTT_{base}(ms)				
		10	**40**	**180**	**240**	**340**
B_{rate}(Mbps)	**1.5**	202%	121%	101%	101%	99%
	4	135%	106%	99%	98%	97%
	12	116%	102%	98%	97%	98%
	25	105%	102%	99%	98%	97%
	50	102%	84%	98%	96%	96%

The pseudo code for the full HLDG SS algorithm is shown in Algorithm 5.3. The algorithm includes both *ssthresh* limiting (using ACK train) and delay gradient signal to terminate slow start.

5.4.2 Simple HLDG SS evaluation

We conduct the same experiment from Section 5.4.1 using HLDG slow start. Table 5.5 shows the percentage of *ssthresh* to the path BDP $(ssthresh/BDP) \times 100$ for each test. We extract *ssthresh* value just after HLDG switches from SS to CA phase. This table shows that HLDG SS terminates setting *ssthresh* to better BDP estimation in all scenarios with 0.096 average error ratio.

It is possible that HLDG SS might experience early termination from the slow start phase. One example would be when multiple flows join a bottleneck and each flow starts immediacy after the previous flow finishes emitting its IW burst. In this case, each flow will observe positive delay gradients, resulting in them terminating SS early. Such extreme circumstances require further research and investigation to improve the performance of the HLDG SS algorithm.

5.5 HLDG in lossy environments

CDG uses queue state inference on packet loss events to distinguish between congestion-related and random losses. If the queue state is full upon packet loss, CDG considers the loss is due to network congestion and halves *cwnd*; otherwise it preserves *cwnd* value during loss recovery. We described this mechanism in Section 2.7.3. According to our discussion in Section 4.3.3.2, the CDG queue state heuristic is not robust and produces a high false negative

Algorithm 5.3 Pseudo code for HLDG slow-start

```
1: Initialisation :
2: $AckCount \leftarrow Agg\_AckBytes \leftarrow 0$
3: $RTT_{min,n} \leftarrow \infty$
4: $RTT_{min,n-1} \leftarrow 0$

5: On each ACK:
6: $RTT_{min,n} \leftarrow min(RTT_{min,n}, RTT)$
7: if $ssthreh < cwnd$ then
8:     $\#ssthresh\ cappingSS$
9:     if $is\_new\_measurement\_cycle()$ then
10:        $AckCount \leftarrow Agg\_AckBytes \leftarrow 0$
11:        $stime = NOW$
12:    else if $AckCount == 8$ then
13:        $ssthresh \leftarrow agg\_ack\_bytes/(NOW - stime)$
14:        if $cwnd > ssthresh$ then
15:            $cwnd \leftarrow ssthresh$
16:        end if
17:    end if
18:    $Agg\_AckBytes \leftarrow Agg\_AckBytes + AckBytes$
19:    $AckCount \leftarrow AckCount + 1$

20:    $\#delay\ gradientSS$
21:    if $is\_new\_measurement\_cycle()$ then
22:        if $RTT_{min,n-1} > 0$ then
23:            $g_{min,n} \leftarrow RTT_{min,n} - RTT_{min,n-1}$
24:            if $probabilistic\_backoff(g_{min,n})$ then
25:                $ssthresh \leftarrow cwnd$
26:            end if
27:        end if
28:        $RTT_{min,n-1} \leftarrow RTT_{min,n}$
29:        $RTT_{min,n} \leftarrow \infty$
30:    end if
31: end if

32: On each packet loss:
33: if $ssthreh < cwnd$ then
34:     $ssthresh \leftarrow cwnd * \beta_{loss}$
35: end if
```

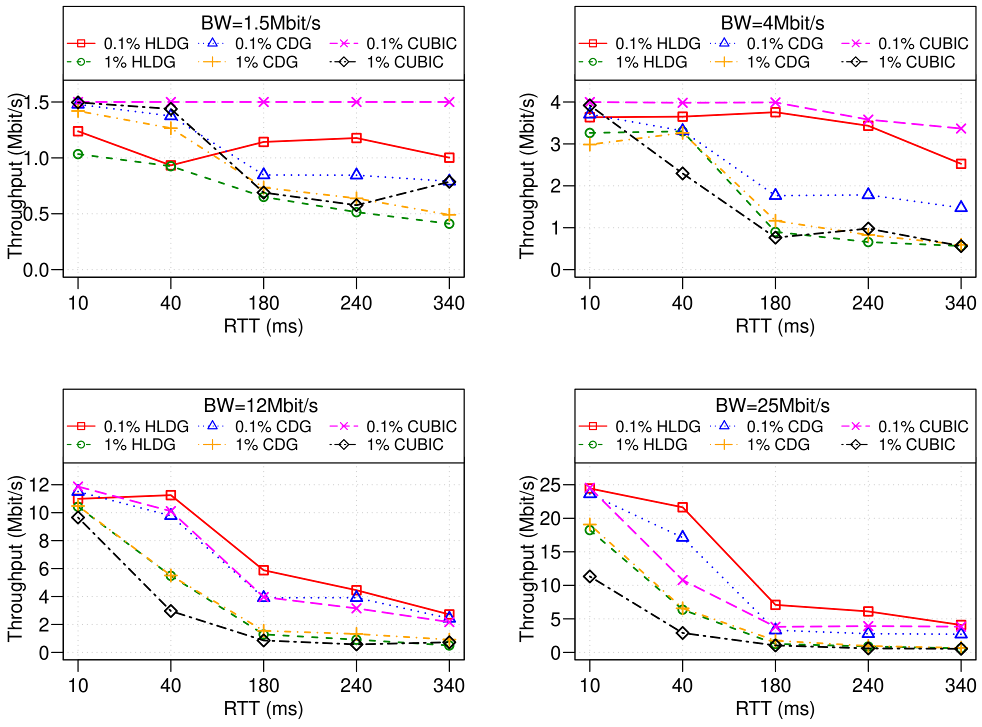


Figure 5.33: HLDG performs similar or better than CDG on lossy link.

error ratio for queue full state inference. That means that CDG will not react to congestion related losses when the queue state is wrongly inferred, causing a bad impact on network stability.

Additionally, queue state heuristic design does not take AQM bottlenecks into consideration. AQM bottlenecks drop packets when queue lengths (in size or time) exceed a specific threshold. AQMs, such as RED [43], PIE [14, 15] and FQ-PIE [17], drop packets randomly when congestion is detected. These packet drops are inferred by the CDG queue state heuristic as non-congestion related losses. Ignoring such losses leads to high queuing delay and packet losses that affect all flows sharing the bottleneck.

HLDG disables the loss tolerance mechanism to improve the shadow window mechanism. We used a TCP Westwood [119] like mechanism to provide better throughput in the existence of random losses. Instead of ignoring *cwnd* decay when packet losses are detected, the *cwnd* is set to the estimated BDP. Since HLDG already uses BDP estimation in congestion control, implementing this mechanism is simple. To prevent *cwnd* from being larger than current window size or smaller than $\beta_{delay} \times cwnd_i$ when packet loss is detected, HLDG sets *cwnd* to the estimated BDP upon packet loss only when $BDP < cwnd \wedge BDP > \beta_{delay} \times cwnd$.

We explore the effectiveness of this solution through conducting an experiment consisting of a single HLDG flow traverses a bottleneck emulating a link with B_{rate}={1.5, 4, 12, 25}Mbps, random loss ratio={0.1%, 1%} and RTT_{base}={10, 40, 180, 240, 340}ms. Bottleneck FIFO buffer Droptail mechanism is used with one BDP buffer size. We also repeat the same experiment but for CDG and CUBIC for comparison purposes with HLDG.

Figure 5.33 plots average throughput for the flows calculated between t=10s and t=60s for HLDG, CDG and CUBIC. Generally speaking, HLDG realises very good throughput at small path RTTs.

At B_{rate}=1.5Mbps and 0.1% random loss, HLDG achieves around 80% bandwidth utilisation when $RTT_{base} = 40ms$. At higher path RTT, it utilises around 65% of the bandwidth. CDG achieves higher throughput than HLDG when $RTT_{base} \leq 40ms$. However, it realises lower throughput at higher path RTT. On the other hand, CUBIC performs the best in this scenario due to the larger CUBIC decrease factor and the faster *cwnd* growth function. Additionally, path BDP in this scenario is small which makes *cwnd* recovery fast.

At B_{rate}=1.5Mbps and 1% loss, CDG and CUBIC perform similarly. They achieve around 85% utilisation when $RTT_{base} \leq 40ms$ while HLDG realises around 65% utilisation. At higher base RTTs, the three algorithms perform similarly with around 45% bandwidth utilisation. CDG performs a bit better in these scenarios.

HLDG performs better at 4Mbps than in 1.5Mbps scenarios. When the loss ratio is 0.1%, HLDG and CUBIC obtain similar bandwidth with slightly advantage for CUBIC flow. When $RTT_{base} = 10ms$, HLDG, CDG and CUBIC achieve between 90% to 100% link utilisation. When $RTT_{base} = 40ms$, they realise between 80% to 95% link utilisation. When $RTT_{base} = 180ms$, HLDG and CUBIC obtain about 95% of the bandwidth while CDG uses only 45%. At higher RTTs, HLDG and CUBIC obtain between 60% to 80% of the bandwidth while CDG achieves around 45%.

At B_{rate}=4Mbps and 1% loss, HLDG and CDG perform almost identically. when $RTT_{base} \leq 40ms$, they obtain around 80% of the bandwidth. CUBIC, however, realise around 95% bandwidth at when $RTT_{base} = 10ms$ and 55% when $RTT_{base} = 40ms$. At higher RTTs, the three algorithms perform similarly with around 25% link utilisation.

At higher bandwidths, the three algorithms perform similarly with a slight advantage to HLDG in almost every test. In B_{rate}=12Mbps and random loss ratio is 0.1% scenarios, they achieve around 95% utilisation when $RTT_{base} = 10ms$ and about 85% when $RTT_{base} = 40ms$. At higher base RTTs, the bandwidth usage drops as path BDP increases. When $RTT_{base} \geq 180ms$, HLDG realises between 25% and 50% while CDG and CUBIC obtain between 25% and 30% bandwidth. When random loss ratio is 0.1%, throughput decreases significantly as path RTT increases. HLDG and CDG perform better than CUBIC but all of them utilise 10% or less of the bandwidth.

In B_{rate}=25Mbps scenarios, HLDG can still perform better than CDG and CUBIC. However, all of the algorithms utilise less than 5% when $RTT_{base} \geq 180ms$ and random loss ratio is 1%.

Low throughput at large path BDP is due to the bandwidth estimator being unable to obtain an accurate estimation and *cwnd* requiring a longer time to recover. Packet losses lead to smaller *cwnd* and never reaching BDP. This results in sending stalling until the next RTT. As bandwidth is calculated based on ACKs received in the previous window, the estimator underestimates the bandwidth due to sending stall.

All algorithms suffer from low throughput at large BDP path due to spending most of the time recovering from losses which takes longer as RTT increases. They also suffer from transmission time-out due to many packet losses during loss recovery.

In summary, HLDG can perform well in a lossy environment due to the bandwidth estimator. Compared to CDG and TCP CUBIC, HLDG obtains better or similar throughput for most scenarios on the same testing conditions, especially at higher data rates. Although loss type differentiation can provide better throughput in lossy environments, it can impact negatively if the heuristic fails to infer congestion-related losses. This aspect requires further study and research to produce a robust heuristic to produce accurate differentiation. Therefore, we leave this aspect to future work study.

5.6 Conclusions

Coexistence with loss-based TCP flows is one of the major issues preventing delay-based TCP from being deployed globally. CDG uses the ineffective back-off and shadow window mechanisms to improve coexistence with conventional TCP flows. These mechanisms do not work well in many scenarios. Since HLDG maintains a lower queuing delay than CDG, it is exposed to starvation in many scenarios due to the aggressiveness of competing loss-based algorithms. To better cope, HLDG adds further coexistence mechanisms to enhance performance.

Firstly, HLDG exploits the *G* parameter of probabilistic back-off function. It has been shown that the *G* parameter has an impact on the aggressiveness of the protocol. Instead of using a constant value for *G*, HLDG adaptively calculates G based on the current estimated queuing delay. As queuing delay increases, G increases exponentially. This mechanism allows HLDG to achieve better coexistence in small bottleneck buffer size scenarios. However, it is less effective when the bottleneck buffer size is large.

To improve coexistence in large buffers, HLDG deploys an explicit mode switching mechanism. HLDG detects competing loss-based flows based on queuing delay measurement and transitions to CUBIC-like loss based mode. It keeps operating in this mode until it no longer detects the presence of loss based flows. The detection technique is enforced by a probe mode in which HDLG enters a delay-based like temporary mode whenever packet loss occurs. This mode ensures that any built-up queue in the bottleneck is drained when no loss based flows are competing. If low queuing delay is observed, HLDG returns to delay mode, otherwise it returns to the loss mode.

Our experimental results show reasonable bandwidth share of HLDG when competing with loss-based TCP, namely CUBIC. HLDG is able to achieve up to 3000% (810% in average) better throughput than CDG when competing with TCP CUBIC. However, HLDG is still not perfect with respect to coexistence and need further work to improve coexistence mechanisms. Additionally, this approach need to be evaluated for more scenarios including very low rate (<1.5 Mbps) and high rate (>25Mbps), multiple HLDG and CUBIC flows, and a mix of mice and elephant background flows.

CDG slow start suffers from premature termination, resulting in flow with low throughput on large BDP paths. HLDG slow start relies on minimum delay-gradient measurements to exit slow start and ACK train to prevent SS from overshooting. The ACK train technique is used to estimate bottleneck bandwidth which is used to calculate path BDP. *ssthresh* is set to BDP to protect SS from overshooting in case the delay-gradient SS mechanism does not terminate in time. HLDG SS does not require receiver side modification or explicitly sending back-to-back packets. The result shows that HLDG slow-start exits with a congestion window between 96% - 200% of the path BDP compared to 1% - 800% for CDG slow-start. It is easy to implement, and produces high throughput and low queuing delay with average 0.096 *cwnd* to the path BDP error ratio when SS terminates.

Additionally, we showed that HLDG is tolerant to random losses better than CDG in most testing scenarios. Instead of loss type differentiation, HLDG relays on TCP Westwood like technique in which cwnd is set to the estimated BDP at packet loss events.

Chapter 6

Further HLDG evaluation

6.1 Introduction

In previous chapters, we described the HLDG algorithm and provided an evaluation for a wide range of simple scenarios. However, we did not consider more complex scenarios and specific types of network environments.

In many real-world situations, traffic does not involve one or two flows sharing a bottleneck. Multiple users and applications open concurrent connections with remote hosts. Even the simple task of browsing a website on the Internet involves opening concurrent TCP connections.

Additionally, FIFO with DropTail is no longer the only deployed mechanism on home gateways due to renewed interest in deploying modern AQMs on home broadband services. Modern Active Queue Management (AQM) has become a desirable feature for many routers and home gateways due to global awareness of the Bufferbloat phenomenon [5].

In this chapter, we explore and provide preliminary results for HLDG algorithm performance in more complex situations than we have considered so far. In particular we consider multiple HLDG flows with homogeneous and heterogeneous path RTTs.

Additionally, we explore the performance of HLDG with bottlenecks that use modern AQM, namely CoDel (Controlled Delay) [47], PIE (Proportional Integral controller Enhanced) AQM [14, 15], FQ-CoDel (Flow-Queue CoDel) [16] and FreeBSD's FQ-PIE (Flow-Queue PIE) [17].

Finally, we explore the possibility of using HLDG to serve as a LPCC where HLDG acquires lower throughput in the existence of competing conventional TCP flows.

The rest of this chapter is organised as follows. Section 6.2 explores HLDG inter-protocol fairness for both homogeneous and heterogeneous path RTTs. Section 6.3 provides a preliminary evaluation for HLDG under different modern AQM bottlenecks. Section 6.4 explores

Table 6.1: Jain fairness index and average queuing delay for five HLDG flows sharing a bottleneck emulating 12Mbit/s link and {10, 40, 180, 240, 340}ms path RTT

RTT_{base}	10ms	40ms	180ms	240ms	340ms
Jain fairness index	0.999	0.999	0.998	0.993	0.976
max-min fairness index	0.981	0.915	0.88	0.80	0.63
Avg. queuing delay	12ms	9ms	6ms	5ms	5ms

using HLDG as low priority CC for time-insensitive background TCP bulk data transfer. We conclude this chapter in Section 6.5.

6.2 Intra-protocol fairness

We conducted experiments consisting of five staggered-start HLDG flows traversing a 1000 packet FIFO bottleneck emulating a 12Mbps link and {10, 40, 180, 240, 340}ms path RTT. Each flow starts 10 seconds after the previous one begins and runs for 5 minutes. Jain's fairness index (Equation 2.3) and max-min fairness index (Equation 6.1) are calculated for each individual scenario when all five flows are actively competing (i.e. period between t=50s and t=300). In Equation 6.1, x_i represents throughput for the ith flow.

$$fairness_{max\_min} = \frac{Min(x_i)}{Max(x_i)} \forall 1 \leq i \leq n \tag{6.1}$$

6.2.1 Homogeneous path RTT

First, we evaluate the protocol fairness when all flows traverse the same path RTT. Table 6.1 shows Jain's fairness index, max-min fairness index and average queuing delay for the experiment. This table reveals that the HLDG flows achieve similar throughput with low queuing delay for all testing scenarios. However, the fairness between flows decreases as RTT_{base} increases. This is due to *cwnd* dynamic (including back-off) being slower for higher path RTTs, than in smaller RTT paths.

The max-min fairness index shows that a few flows achieve higher or lower throughput than the other flows in higher RTT paths. This occurs because if one flow realises higher throughput (randomly), other flows will obtain lower throughput until that flow backs off multiple times, releasing some bandwidth for other flows.

Figure 6.1 shows throughput versus time for the 12Mbit/s and 40ms RTT_{base} experiment calculated using a moving window of 5 seconds with 10-sample interpolation. We can see

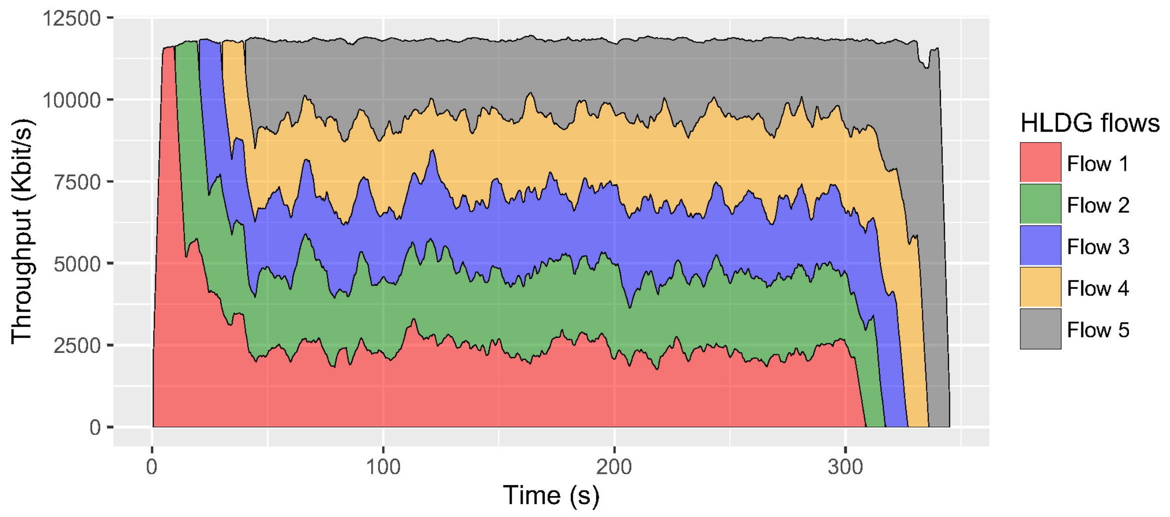


Figure 6.1: Five HLDG flows sharing a bottleneck emulating 12Mbit/s and 40ms RTT_{base}. In this stacked area graph, each area represents individual flow throughput stacked above the area of the previous flow. The entire graph represents link utilisation. Flow throughput is calculated over a 5-second window with 10-sample interpolation.

that the link is fully utilised and each flow achieves similar throughput during the experiment life-time.

6.2.2 Heterogeneous path RTT

RTT-unfairness is a well know issue for most end-to-end congestion control algorithms [177]. This results in flows traversing shorter RTT paths obtaining higher bandwidth than flows travelling over longer RTT paths when they share a bottleneck. This happens mainly due to two causes; 1) slower *cwnd* increases, and 2) backing off by a larger amount. Standard TCP [32] increases *cwnd* by one MSS every RTT, causing *cwnd* to reach BDP faster for flows traversing shorter RTT paths. As the BDP for longer paths is higher than for shorter paths, flows with higher RTT require more RTT rounds to recover *cwnd* above BDP (or pushing a similar number of packets to the bottleneck buffer) after back off.

The HLDG protocol also exhibits a similar issue but in a lower scale and in the opposite manner where flows with higher RTT achieve higher throughput than flows traversing shorter paths. This is due to the HLDG probabilistic back-off function being performed on every RTT cycle. As a result, flows with smaller RTT have higher back off probability than flows with larger RTT.

CDG remedies this issue by using the exponential factor in the probabilistic back off function (see section 2.7.2). However, this solution is not very effective, especially in HLDG

Table 6.2: Jain fairness index and average queuing delay for five HLDG flows sharing a bottleneck emulating 12Mbit/s link. Two flows traverse {90, 120, 260, 320, 420}ms RTT paths, and three flows traverse {10, 40, 180, 240, 340}ms RTT paths

Flow 1,2 RTT_{base}(ms)	90	120	260	320	420
Flow 3,4, 5 RTT_{base}(ms)	10	40	180	240	340
Jain fairness index	0.993	0.97	0.99	0.992	0.959
max-min fairness index	0.787	0.682	0.77	0.784	0.532
Avg. queuing delay	12ms	12ms	8ms	7ms	5ms

where *cwnd* is set to a fraction of estimated BDP. Nevertheless, HLDG flows can reach better fair-share in long-lived connections.

We repeat the same experiment but with two flows traversing a longer path (80ms higher path RTT than the others) to explore HLDG performance in heterogeneous RTT scenarios. This emulates real-life scenarios when multiple users/applications share a bottleneck and some of them are connected to further servers than the others.

Table 6.2 shows Jain's fairness index, max-min fairness index and average queuing delay for the experiment. The results reveal that the flows maintain low average queuing delay similar to the homogeneous path RTT experiment. Jain's fairness index shows that the HLDG flows achieve similar throughput with lower fairness for the 420ms/340ms scenario. However, the max-min fairness index shows low fairness between some flows, especially for the 420ms/340ms scenario. This mean that some flows achieve higher or lower throughput than the other flows.

Figure 6.2 shows the throughput trajectory for 12Mbit/s with 40ms/120ms RTT_{base} experiment, calculated using a 5-second moving average window with 10 interpolated samples. We can see in this figure, the link is fully utilised but flows 1 and 2 achieve slightly higher throughput than the others due to fewer back-offs compared to the other flows.

6.3 HLDG and modern AQM bottlenecks

Active Queue Management (AQM) is a mechanism used to keep the bottleneck queues of network nodes to a controlled depth, effectively creating short queues [42]. AQM replaces the traditional DropTail mechanism. AQM targets loss-based TCP flows by dropping or marking (with an ECN [4]) packets when bottleneck queuing delay reaches specified levels, causing TCP to back-off earlier than would occur with FIFO queues. The potential for significant reduction in overall round trip time (RTT) is motivating deployment of these new AQMs at

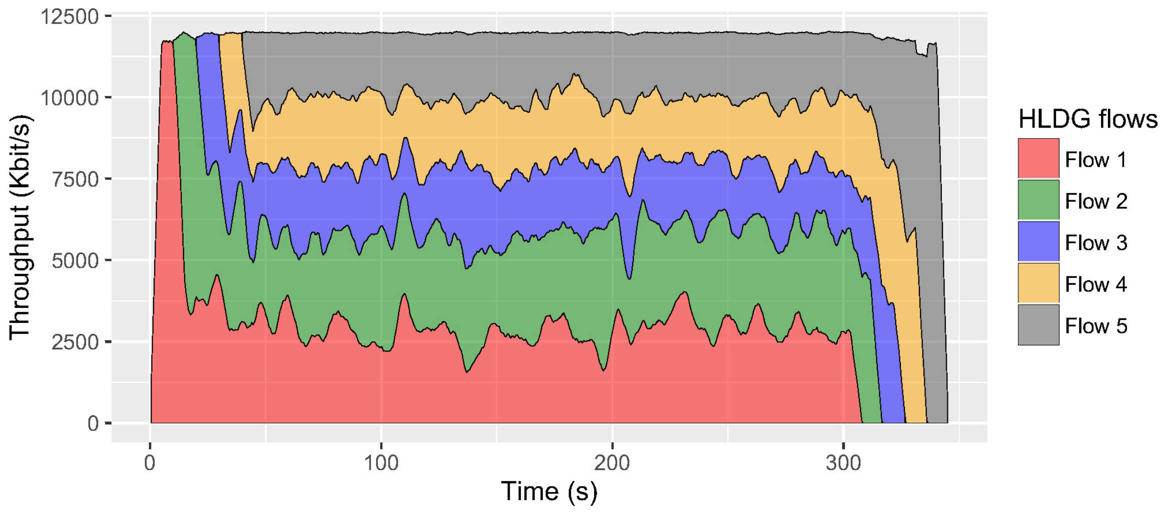


Figure 6.2: Five HLDG flows sharing a bottleneck emulating 12Mbit/s link. Flow 1, 2 traverse a 120ms RTT path and flow 3,4, 5 traverse 40ms RTT path

either end of shared, home broadband last-mile services. In Section 2.2.2, we briefly described how modern AQM use queuing delay measurements to detect congestion inside the bottleneck buffer.

In our work [98] we find that the interaction between AQM and delay-based flows can lead to undesirable side-effects, impacting on protocol stability. If CC does not take AQM existence into a consideration, different types of anomalies can affect the performance of all flows sharing a bottleneck. In this section we briefly explore the performance of HLDG through PIE (Proportional Integral controller Enhanced) [14, 15], FQ-CoDel (Flow-Queue CoDel) [16] and FreeBSD's FQ-PIE (Flow-Queue PIE) [17] AQMs.

We conduct an experiment consisting of five HLDG flows traversing a bottleneck emulating 12Mbit/s link and $\{10, 40, 180, 240, 340\}$ms path RTT. PIE, FQ-CoDel and Flow-Queue PIE are used with 1000 packet buffer size. We repeat the same experiment with CDG and CUBIC for comparison purposes.

Tables 6.3, 6.4 and 6.5 show Jain's fairness index, link utilisation and packet count dropped by AQM respectively for HLDG, CDG and CUBIC when PIE, FQ-CoDel and FQ-PIE are used by the bottleneck. Jain's fairness index shows that all algorithms perform similarly with the best results achieved by CUBIC. In terms of link utilisation, CUBIC still achieves the highest performance due to the cubic *cwnd* growth function and higher multiplicative β factor.

In PIE AQM scenarios, HLDG performs better than CDG over high RTT paths ($\geq$ 180ms) and is close to CUBIC performance. However, the number of dropped packets is much larger with CUBIC due to its aggressiveness and using only the loss signal. When RTT_{base} =$\{$10ms, 40ms$\}$, HLDG experiences no packet loss while CDG has a large number of packet drops, even larger than CUBIC. CDG experiences large number of packet drops due to false negative detection for full queue state which prevents *cwnd* reducing.

At higher RTT_{base}, HLDG experiences a small number of packet loss compared to CUBIC. CDG experiences fewer losses since it does not achieve high link utilisation.

With FQ-CoDel, HLDG and CDG perform similarly with a smaller number of losses than CUBIC. At RTT_{base} ={10ms, 40ms}, HLDG experiences fewer losses than CDG while CDG has fewer losses at higher RTTs.

With FQ-PIE, HLDG performs very similar to CUBIC but with much fewer packet drops. CDG, on the other hand, suffers a higher number of packet losses at small RTTs and lower performance at higher RTTs.

These results indicate that HLDG performs reasonability in AQM environments while preserving a low number of packet drops. HLDG exhibits the best performance with low packet loss when FQ-PIE is deployed in the bottleneck. It realises 90-98% link utilisation compared to 77-100% for CDG and experiences 96% less packet loss than CDG. This is due to the queuing delay induced by HLDG being similar to what the PIE AQM allows, as well as the flow-queue flow separation mechanism.

Table 6.3: Jain fairness index for five flows traversing 12Mbps bottleneck with PIE, FQ-CoDel and FQ-PIE AQM.

	PIE			FQ-CoDel			FQ-PIE		
RTT_{base} **(ms)**	**HLDG**	**CDG**	**CUBIC**	**HLDG**	**CDG**	**CUBIC**	**HLDG**	**CDG**	**CUBIC**
10	0.997	0.979	0.999	0.997	0.999	0.999	0.997	0.999	0.999
40	0.998	0.995	0.998	0.997	0.999	0.999	0.997	0.999	0.999
180	0.996	0.996	0.996	0.998	0.999	0.999	0.998	0.999	0.999
240	0.994	0.992	0.993	0.996	0.997	0.999	0.999	0.998	0.999
340	0.97	0.995	0.995	0.995	0.995	0.994	0.999	0.973	0.999

Table 6.4: Link utilisation for five flows traversing 12Mbps bottleneck with PIE, FQ-CoDel and FQ-PIE AQM.

	PIE			FQ-CoDel			FQ-PIE		
RTT_{base} **(ms)**	**HLDG** %	**CDG** %	**CUBIC** %	**HLDG** %	**CDG** %	**CUBIC** %	**HLDG** %	**CDG** %	**CUBIC** %
10	99	100	100	98	100	100	98	100	100
40	98	99	100	97	98	99	98	99	99
180	94	86	96	89	88	96	97	87	98
240	91	82	91	84	82	94	94	84	96
340	86	77	88	76	77	92	90	77	92

Table 6.5: Dropped packets to total packets ratio for five flows traversing 12Mbps bottleneck with PIE, FQ-CoDel and FQ-PIE AQM.

	PIE			FQ-CoDel			FQ-PIE		
RTT_{base} **(ms)**	**HLDG**	**CDG**	**CUBIC**	**HLDG**	**CDG**	**CUBIC**	**HLDG**	**CDG**	**CUBIC**
10	0	2.979	2.496	0.008	3.146	0.207	0	3.16	2.222
40	0	1.315	0.787	0.592	1.241	0.923	0	0.01	0.753
180	0.043	0	0.237	0.259	0.165	0.307	0.0055	0.071	0.863
240	0.059	0	0.457	0.199	0.044	0.317	0.054	0.006	1.098
340	0.049	0.005	0.634	0.145	0.011	0.334	0.087	0.009	1.188

6.4 HLDG as LPCC protocol

It was shown in [7] that CDG can efficiently be used to provide low priority transport for time-insensitive background TCP bulk data transfer in home networks. The similarity between HLDG and CDG in terms of congestion detection technique inspires us to introduce a LPCC mode for HLDG.

HLDG modification

HLDG can optionally operate in low priority mode by disabling its coexistence mechanisms.

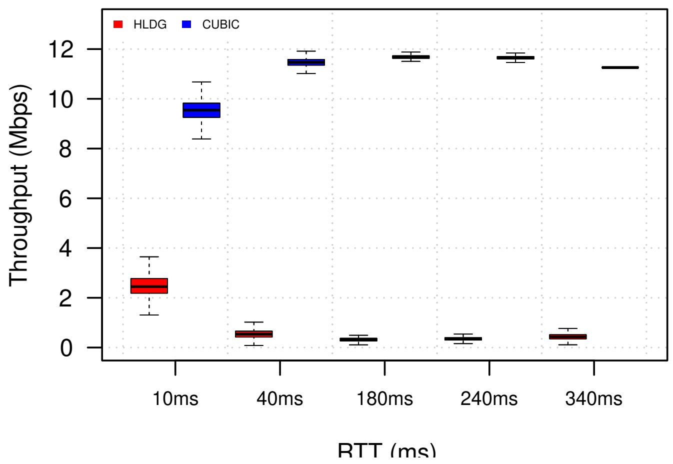


Figure 6.3: HLDG in LPCC mode achieves lower throughput when sharing a CUBIC for a 12Mbps bottleneck. Buffer size is 2 BDP.

Since HLDG reacts to network congestion earlier than loss-based flows, it can be used as a low priority congestion control protocol by disabling the coexistence mechanisms. This includes disabling the ineffectual back-off, shadow window, adaptive G and mode switching mechanisms. To further reduce the impact of HLDG on other flows, the BDP back-off factor (γ) can be set to 75%, allowing *cwnd* to decrease well below BDP. Further, a smaller G parameter can be used which increases HLDG sensitivity to RTT increase. However, this can lead to lower throughput, even in the absence of competing conventional TCP flows.

After enabling the LPCC mode, we conduct an experiment consisting of an HLDG flow and a CUBIC flow traversing a bottleneck emulating a 10Mbps link bandwidth and {10, 40, 180, 240, 340}ms RTT_{base} with a two BDP FIFO buffer. The HLDG flow starts first, with the CUBIC flow starting at t=10s. The experiment lasts for 300 seconds.

Figure 6.3 shows the throughput boxplot of the HLDG flow and CUBIC flow for the experiment. The figure reveals that HLDG in LPCC mode achieves much lower throughput than CUBIC in large RTT paths. However, HLDG obtains about 18% of available bandwidth when RTT_{base} is 10ms. This occurs due to the small buffer size (20 packets) that results in a smaller number of CUBIC to HLDG packets ratio. At larger path RTT, HLDG utilises less than 5% of bandwidth.

This experiment demonstrates that HLDG running in LPCC mode is a promising low priority protocol that can effectively provide scavenger class transport without impacting on conventional TCP flows. However, further research is needed to evaluate the protocol under more scenarios and network settings.

6.5 Conclusions

In this chapter, we performed some preliminary evaluations for HLDG in more complex network scenarios. On HLDG achieves very good fairness between flows sharing a bottleneck in both homogeneous and heterogeneous path RTTs. However, fairness between flows reduces as path RTT increases due to slow *cwnd* dynamic. The results show that HLDG achieves 0.99 and 0.98 average Jain's fairness index in homogeneous and heterogeneous path RTTs respectively while keeping average queuing delay below 13ms.

We then evaluated HLDG when AQM, namely PIE, FQ-CoDel and FQ-PIE, are used at the bottleneck instead of a FIFO buffer. We have shown that HLDG performs well in most network settings and leads to a small number of packet drops by AQM. The results show that HLDG achieves 94%, 89% and 95% average link utilisation under PIE, FQ-CoDel and FQ-PIE respectively compared to 89% for CDG under the same conditions. HLDG experiences only 3%, 24% and 4% of packet loss experienced by CDG when run through PIE, FQ-CoDel and FQ-PIE AQM respectively. HLDG achieves the best performance when FQ-PIE is used due to its flow isolation mechanism and the similar queuing delay allowed by PIE AQM and HLDG.

We also performed a simple evaluation for HLDG in low priority mode. HLDG operates in this mode by disabling the coexistence mechanisms and using a smaller γ factor. In this mode, HLDG acts as a scavenger class transport, allowing the conventional TCP flows sharing a bottleneck to obtain higher bandwidth than HLDG flows. The results show that HLDG in LPCC mode obtains 5 - 18% bottleneck bandwidth when sharing a link with CUBIC.

Chapter 7

Future work and conclusions

7.1 Future work

Since its first deployment in 1987, TCP congestion control has earned significant attention from both research and industrial communities due to its direct impact on transport protocol performance. Work in this area has not stopped since then due to rapid advances in network technologies, increasing demands from end users, the high complexity of the signal and network setting variation that influence congestion. Further, the delay signal is highly affected by network conditions, configurations, background traffic and application behaviours. Similar to other congestion control algorithms, there are always limitations to address, gaps to fill and improvements to apply. Potential future work arising from this thesis is summarised as follows.

Similar to other TCP CC algorithms, HLDG should be comprehensibly evaluated under a wider range of scenarios before being deployed on the Internet. This involves further testing HLDG with slower and faster links, multi-bottleneck configurations, different flow starting sequences and with a mixture of background traffic. Additionally, evaluating HLDG over the Internet between different points around the world can produce more realistic results that reflect protocol performance. This also allows comparing performance between HLDG and other protocols over realistic scenarios.

Our observations show that the CDG queue state heuristic produces inaccurate inferences for many scenarios. This causes CDG to ignore back-off when congestion related losses occur while the heuristic incorrectly infers a non-full queue. For this reason, HLDG disables this loss-tolerance mechanism. An investigation into the causes for CDG's low accuracy in queue state detection is required to provide a CC protocol that provides high performance in lossy environments without ignoring congestion-related packet loss.

HLDG uses the adaptive G function to reduce protocol sensitivity to delay gradient increase when competing with loss-based flows. The current function uses the queuing delay measurement to infer the presence of competing loss-based flows, and change the G value as required.

Packet pacing reduces TCP traffic burstiness, resulting in smooth traffic with stable delays. HLDG can benefit from packet pacing in two aspects. Firstly, packet pacing leads HLDG to detect permanent congestion rather than temporary congestion caused by a short packet bursts. Secondly, packet pacing can improve the RTT_{base} estimation mechanism by eliminating the *cwnd* restoration impact at the beginning of the estimation cycle (5.2.6).

HLDG uses some parameters to control different parts of the algorithm. We set the default parameters based on experimental observations and trading off multiple algorithm aspects. However, a comprehensive analysis is required to tune these parameters for different network configurations. Tuning these parameters requires conducting experiments over different scenarios. Therefore, a comprehensive experimental study should be conducted to find optimum parameters for different network settings.

In Section 6.3, we performed a preliminary evaluation for HLDG under AQM environments. However, that evaluation included only simple scenarios under limited network configurations. The results show that HLDG performs better under specific AQM (e.g. FQ-PIE) than other AQMs (e.g. FQ-CoDel). A comprehensive theoretical and experimental analysis is required to characterise the interaction between HLDG and different AQM schemes.

Finally, HLDG can be used to provide scavenger class transport in which HLDG flows obtain lower bandwidth sharing then conventional TCP flows. We demonstrated how HLDG performs in LPCC mode under simplistic scenarios (Section 6.4). However, comprehensive evaluation and analysis are required to quantify HLDG in LPCC mode under different network conditions.

7.2 Conclusions

The primary contribution of this thesis is to enhance the previously published CAIA Delay-Gradient (CDG) CC algorithm to create the Hybrid Loss Delay Gradient (HLDG) CC algorithm. HLDG is a sender-side dual-mode hybrid TCP congestion control algorithm for the Internet that provides low latency, high throughput and coexists well when sharing a bottleneck with loss-base flows. HLDG fixes various CDG shortcomings and flaws that we identified. Our experimental results indicate that HLDG responds appropriately to network congestion, does not cause heavy packet loss, and therefore, is safe to use on the Internet.

The development of HLDG has evolved into the following key contributions.

1. We experimentally evaluated and analysed CDG performance in different network set-

tings to identify possible issues and the causes of these problems. We studied CDG link utilisation in simple scenarios where only a single CDG flow traversing a bottleneck. This helped us understand the relation between network congestion and the delay-gradient signal, and identified protocol issues that lead to low link utilising in high path RTT scenarios.

2. We evaluated CDG coexistence performance when competing with loss-based flows, namely CUBIC. We found that CDG suffers from low throughput and starves in most scenarios, especially in large buffer sizes. In addition to the congestion avoidance phase, we evaluated and analysed the CDG slow-start phase and identified issues limiting slow-start performance.

3. We proposed HLDG, a sender-side hybrid congestion control algorithm for the Internet. HLDG fixes CDG low link utilisation issues by preventing unnecessary back-off in which *cwnd* is smaller than the path BDP and the probabilistic back-off function triggers congestion event. HLDG backs off *cwnd* to a portion of estimated BDP when congestion is detected using the delay gradient signal, allowing HLDG to maintain high throughput and low queuing delay. We evaluated HLDG performance under different network settings and the results show significant improvement over CDG. Our experimental evaluation shows that HLDG realises up to 400% (54% in average) throughput improvement over CDG in single flow scenarios without experiencing packet loss. HLDG is able to maintain low queueing delay of no more than 20ms in the worst case.

4. We proposed a mode switching algorithm for HLDG which dynamically switches the algorithm operation between delay and loss mode. In delay mode, HLDG reacts to the delay-gradient signal while in loss mode it backs off only on packet loss. This allows HLDG to provide improved performance when competing with loss based flows in large buffer environments. Additionally, HLDG dynamically reduces protocol sensitivity to delay increase based on the congestion level to improve protocol coexistence. Our experimental evaluation shows that the improved coexistence allows HLDG to achieve up to 3000% (810% in average) better throughput than CDG when competing with TCP CUBIC. Additionally, it shows that HLDG coexistence performs ten orders of magnitude better than CDG in large buffer sizes.

5. HLDG overcomes CDG slow start low performance by using RTT_{min} gradient signal combined with BDP estimate. The new algorithm allows HLDG to leave the slow start phase with *cwnd* close to the path BDP, improving protocol performance in large BDP paths. The results show that HLDG slow-start terminates with a congestion window

between 96% - 200% of the path BDP compared to 1% - 800% for CDG slow-start.

6. We provide a preliminary evaluation for HLDG intra-protocol fairness under FIFO. The results show that HLDG achieves 0.99 and 0.98 average Jain's fairness index in homogeneous and heterogeneous path RTTs respectively while keeping average queuing delay below 13ms. In AQM environments, HLDG realises 94%, 89% and 95% average link utilisation under PIE, FQ-CoDel and FQ-PIE respectively compared to 89% for CDG under the same conditions. HLDG experiences only 3%, 24% and 4% of packet loss experienced by CDG when run through PIE, FQ-CoDel and FQ-PIE AQMs respectively.

7. Finally, we explore the possibility of using HLDG to serve as a LPCC where HLDG flows acquire lower throughput when competing with conventional TCP flows. Our preliminary results indicate that HLDG in LPCC mode utilises between 5 - 18% link utilisation when sharing a link with the CUBIC flow.

Our research also derives new general knowledge about delay-based congestion control which can be used to improve delay-based CC algorithms in general.

1. Any delay-based CC algorithm should not back-off in response to small bursts caused by the delayed ACK mechanism. Delayed ACK bursts increase the queuing delay for a short period, which can then be wrongly interpreted by the CC algorithm as a sign of congestion. Therefore, such a burst should be filtered out from the delay congestion feedback calculations, otherwise delay-based CC algorithm would back-off even when no congestion is experienced. Many delay-based CC algorithms do not ignore the delayed ACK burst in congestion feedback calculations, this then results in unnecessary back-off, reducing protocol throughput.

2. A capacity estimator, using ACKs in Westwood/BBR style for example, combined with the delay signal improves the performance of delay-based congestion control. Instead of using a constant multiplicative back-off factor, a capacity estimation can be utilised to limit the amount of congestion window reduction after congestion events. This helps to maintain the average sending rate close to available bandwidth while also maintaining low queueing delay.

3. Standing queues can be avoided by replacing a moving average calculation with a Weighted Windowed Moving Average (WWMA) over the delay signal, and by re-starting the calculation under certain circumstances. A smoothed average is used to reduce noise in the delay signal, however a common moving average approach can result in transient delay measurements impacting decision making over a larger a period

of time. A weighted average can help to maintain currency by allowing a delay-based CC to more quickly respond to changes in delay while also minimising this impact more quickly after responding. Re- starting calculations allow the CC algorithm to discard old samples following an initial response to a delay signal, allowing future responses to be made based on new network conditions.

References

[1] J. Postel, “Transmission Control Protocol,” Internet Engineering Task Force, RFC 793, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793.txt

[2] J. Postel, “Internet Protocol,” Internet Engineering Task Force, RFC 791, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc791.txt

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control,” *Commun. ACM*, vol. 60, no. 2, pp. 58–66, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3009824

[4] K. Ramakrishnan, “The Addition of Explicit Congestion Notification (ECN) to IP,” Internet Engineering Task Force, RFC 3168, September 2001. [Online]. Available: https://tools.ietf.org/html/rfc3168

[5] J. Gettys and K. Nichols, “Bufferbloat: Dark buffers in the internet,” *Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063166.2071893

[6] D. Ros and M. Welzl, “Assessing LEDBAT’s Delay Impact,” *IEEE Communications Letters*, vol. 17, no. 5, pp. 1044–1047, May 2013.

[7] G. Armitage and N. Khademi, “Using delay-gradient TCP for multimedia-friendly ’background’ transport in home networks,” in *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, Oct 2013, pp. 509–515.

[8] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, “Low extra delay background transport (LEDBAT),” Internet Engineering Task Force, RFC 6817, December 2012. [Online]. Available: https://tools.ietf.org/html/rfc6817

[9] A. Venkataramani, R. Kokku, and M. Dahlin, “Tcp nice: A mechanism for background transfers,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 329–343, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844159

[10] C. Huitema, D. Havey, M. Olson, O. Ertugay, and P. Balasubramanian, "New Transport Advancements in the Anniversary Update for Windows 10 and Windows Server 2016," Microsoft, Jul 2016. [Online]. Available: https://goo.gl/ZfI8cG

[11] "TCP LEDBAT Implementation," Apple Inc. [Online]. Available: http://opensource.apple.com//source/xnu/xnu-3248.60.10/bsd/netinet/tcp_ledbat.c

[12] D. A. Hayes and G. Armitage, "Revisiting TCP congestion control using delay gradients," in *NETWORKING 2011*. Springer, 2011, pp. 328–341.

[13] R. Wang, M. Valla, M. Y. Sanadidi, B. K. F. Ng, and M. Gerla, "Efficiency/friendliness tradeoffs in TCP Westwood," in *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, 2002, pp. 304–311.

[14] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg, "PIE: A lightweight control scheme to address the bufferbloat problem," in *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*, July 2013, pp. 148–155.

[15] R. Pan, P. Natarajan, F. Baker, and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem," Internet Engineering Task Force, RFC 8033, February 2017. [Online]. Available: https://tools.ietf.org/html/rfc8033

[16] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, "The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm," Internet Engineering Task Force, RFC 8290, January 2018. [Online]. Available: https://tools.ietf.org/html/rfc8290

[17] R. Al-Saadi and G. Armitage, "Dummynet AQM v0.2 – CoDel, FQ-CoDel, PIE and FQ-PIE for FreeBSD's ipfw/dummynet framework," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160418A, 18 April 2016. [Online]. Available: http://caia.swin.edu.au/reports/160418A/CAIA-TR-160418A.pdf

[18] R. Al-Saadi and G. Armitage, "Dummynet AQM v0.1 - CoDel and FQ-CoDel for FreeBSD's ipfw/dummynet framework," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160226A, 26 February 2016. [Online]. Available: http://caia.swin.edu.au/reports/160226A/CAIA-TR-160226A.pdf

[19] "Revision 300779: Import Dummynet AQM version 0.2.1 (CoDel, FQ-CoDel, PIE and FQ-PIE) to FreeBSD 11.0," May 2016. [Online]. Available: https://svnweb.freebsd.org/base?view=revision&revision=300779

[20] "Revision 301772: Import Dummynet AQM version 0.2.1 (CoDel, FQ-CoDel, PIE and FQ-PIE) to FreeBSD 10.4," May 2016. [Online]. Available: https://svnweb.freebsd.org/base?view=revision&revision=301772

[21] "pfSense 2.4," Netgate Blog. [Online]. Available: https://www.netgate.com/blog/pfsense-2-4-4-release-now-available.html

[22] "OPNsense 16.7 release," jul 2016. [Online]. Available: https://opnsense.org/opnsense-16-7-released/

[23] "Spoting a critical error in IETF FlowQueue-Codel draft-ietf-aqm-fq-codel-03," January 2016. [Online]. Available: https://mailarchive.ietf.org/arch/msg/aqm/qN5ujEj3B9D8_EkpdJL8q5FbYF8

[24] "Spoting a error in IETF PIE draft-ietf-aqm-pie-05," March 2016. [Online]. Available: https://mailarchive.ietf.org/arch/msg/aqm/-2HYq6WjfsQF5A396pwo-CeAP2M

[25] R. Al-Saadi, G. Armitage, and J. But, "ttprobe v0.1: Packet-Driven TCP Stack Statistics Gathering for TEACUP," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 150911A, 11 September 2015. [Online]. Available: http://caia.swin.edu.au/reports/150911A/CAIA-TR-150911A.pdf

[26] "The Web10G Project." [Online]. Available: http://web10g.org/

[27] R. Al-Saadi, G. Armitage, J. But, and P. Branch, "A Survey of Delay-Based and Hybrid TCP Congestion Control Algorithms," *IEEE Communications Surveys Tutorials*, pp. 1–1, 2019.

[28] S. Floyd, "Congestion Control Principles," Internet Engineering Task Force, RFC 2914, September 2000. [Online]. Available: https://tools.ietf.org/html/rfc2914

[29] J. Postel, "Internet Control Message Protocol," Internet Engineering Task Force, RFC 792, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc791.txt

[30] F. Gont, "Deprecation of ICMP Source Quench Messages," Internet Engineering Task Force, RFC 6633, May 2012. [Online]. Available: https://tools.ietf.org/html/rfc6633

[31] S. Floyd and K. Fall, “Promoting the use of end-to-end congestion control in the internet,” *IEEE/ACM Trans. Netw.*, vol. 7, no. 4, pp. 458–472, Aug. 1999. [Online]. Available: http://dx.doi.org/10.1109/90.793002

[32] M. Allman, V. Paxson, and E. Blanton, “TCP congestion control,” Internet Engineering Task Force, RFC 5681, September 2009. [Online]. Available: https://tools.ietf.org/html/rfc5681

[33] L. Brakmo and L. Peterson, “Tcp vegas: end to end congestion avoidance on a global internet,” *Selected Areas in Communications, IEEE Journal on*, vol. 13, no. 8, pp. 1465–1480, Oct 1995.

[34] R. J. La, J. Walrand, and V. Anantharam, *Issues in TCP vegas*. Electronics Research Laboratory, College of Engineering, University of California, 1999.

[35] B. Briscoe, “Flow rate fairness: Dismantling a religion,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 63–74, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1232919.1232926

[36] D. Ros and M. Welzl, “Less-than-best-effort service: A survey of end-to-end approaches,” *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 898–908, Second 2013.

[37] S. Floyd, A. Gurtov, and T. Henderson, “The NewReno modification to TCP’s fast recovery algorithm,” Internet Engineering Task Force, RFC 6582, April 2004. [Online]. Available: https://tools.ietf.org/html/rfc6582

[38] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP Selective Acknowledgment Option,” Internet Engineering Task Force, RFC 2018, October 1996. [Online]. Available: https://tools.ietf.org/html/rfc2018.txt

[39] A. Aggarwal, S. Savage, and T. Anderson, “Understanding the performance of TCP pacing,” in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, Mar 2000, pp. 1157–1165 vol.3.

[40] D. Wei, P. Cao, S. Low, and C. EAS, “TCP pacing revisited,” 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.2658&rep=rep1&type=pdf

[41] J. Sing and B. Soh, “TCP New Vegas: Improving the Performance of TCP Vegas Over High Latency Links,” in *Fourth IEEE International Symposium on Network Computing and Applications*, July 2005, pp. 73–82.

[42] F. Baker and G. Fairhurst, "IETF Recommendations Regarding Active Queue Management," Internet Engineering Task Force, RFC 7567, July 2015. [Online]. Available: https://tools.ietf.org/html/rfc7567

[43] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *Networking, IEEE/ACM Transactions on*, vol. 1, no. 4, pp. 397–413, Aug 1993.

[44] S. Ryu, C. Rump, and C. Qiao, "Advances in internet congestion control," *IEEE Communications Surveys Tutorials*, vol. 5, no. 1, pp. 28–39, Third 2003.

[45] G. Thiruchelvi and J. Raja, "A Survey On Active Queue Management Mechanisms," *International journal of computer science and network security IJCSNS.*, vol. 8, no. 12, December 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.509.460&rep=rep1&type=pdf

[46] R. Adams, "Active queue management: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 3, pp. 1425–1476, Third 2013.

[47] D. K. M. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," RFC 8289, Jan. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8289.txt

[48] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.

[49] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[50] Y. Tian, K. Xu, and N. Ansari, "Tcp in wireless environments: problems and solutions," *IEEE Communications Magazine*, vol. 43, no. 3, pp. S27–S32, March 2005.

[51] J. Andren, M. Hilding, and D. Veitch, "Understanding end-to-end Internet traffic dynamics," in *Global Telecommunications Conference, 1998. GLOBECOM 1998. The Bridge to Global Integration. IEEE*, vol. 2, 1998, pp. 1118–1122 vol.2.

[52] J. Martin, A. Nilsson, and I. Rhee, "Delay-based congestion avoidance for tcp," *IEEE/ACM Trans. Netw.*, vol. 11, no. 3, pp. 356–369, Jun. 2003. [Online]. Available: http://dx.doi.org/10.1109/TNET.2003.813038

[53] S. Biaz and N. H. Vaidya, “Is the Round-trip Time Correlated with the Number of Packets in Flight?” in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’03. New York, NY, USA: ACM, 2003, pp. 273–278. [Online]. Available: http://doi.acm.org/10.1145/948205.948240

[54] G. McCullagh and D. Leith, “Delay-based congestion control: Sampling and correlation issues revisited,” *Hamilton Institute, National University of Ireland Maynooth, Tech. Rep*, 2008.

[55] R. S. Prasad, M. Jain, and C. Dovrolis, “On the effectiveness of delay-based congestion avoidance,” in *Proc. PFLDNet*, vol. 4, 2004.

[56] D. A. Hayes and D. Ros, “Delay-based congestion control for low latency,” in *ISOC Workshop on Reducing Internet Latency, Sep*, 2013. [Online]. Available: http://www.internetsociety.org/sites/default/files/pdf/accepted/17_delay_cc_pos-v2.pdf

[57] V. Jacobson, R. Braden, and D. Borman, “TCP extensions for high performance,” Internet Engineering Task Force, RFC 1323, September 1992. [Online]. Available: https://tools.ietf.org/html/rfc1323

[58] A. Kuzmanovic and E. W. Knightly, “Tcp-lp: Low-priority service via end-point congestion control,” *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 739–752, Aug. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.879702

[59] G. Carofiglio, L. Muscariello, D. Rossi, and S. Valenti, “The Quest for LEDBAT Fairness,” in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, Dec 2010, pp. 1–6.

[60] K. Srijith, L. Jacob, and A. Ananda, “TCP Vegas-A: Improving the Performance of TCP Vegas,” *Computer Communications*, vol. 28, no. 4, pp. 429 – 440, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366404003214

[61] A. J. Abu and S. Gordon, “Impact of Delay Variability on LEDBAT Performance,” in *2011 IEEE International Conference on Advanced Information Networking and Applications*, March 2011, pp. 708–715.

[62] R. Jain, “A Delay-based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 5, pp. 56–71, Oct. 1989. [Online]. Available: http://doi.acm.org/10.1145/74681.74686

[63] D. Sisalem and A. Wolisz, “LDA+ TCP-friendly adaptation: A measurement and comparison study,” in *Proc. NOSSDAV*, 2000, pp. 362–368. [Online]. Available: http://nossdav.org/2000/papers/23.pdf

[64] D. Sisalem and A. Wolisz, “MLDA: a TCP-friendly congestion control framework for heterogeneous multicast environments,” in *2000 Eighth International Workshop on Quality of Service. IWQoS 2000 (Cat. No.00EX400)*, 2000, pp. 65–74.

[65] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” Internet Engineering Task Force, RFC 3550, jul 2003. [Online]. Available: https://tools.ietf.org/html/rfc3550.txt

[66] N. Khademi, M. Welzl, G. Armitage, and G. Fairhurst, “TCP Alternative Backoff with ECN (ABE),” Internet Engineering Task Force, RFC 8511, December 2018. [Online]. Available: https://tools.ietf.org/html/rfc8511

[67] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data Center TCP (DCTCP),” in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[68] B. Vamanan, J. Hasan, and T. Vijaykumar, “Deadline-aware Datacenter TCP (D2TCP),” in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’12. New York, NY, USA: ACM, 2012, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342388

[69] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, “Minimizing flow completion times in data centers,” in *2013 Proceedings IEEE INFOCOM*, April 2013, pp. 2157–2165.

[70] B. Briscoe, K. Schepper, and M. B. Braun, “Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture,” Internet Engineering Task Force, Internet Draft, March 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-codel-07

[71] K. De Schepper, B. Briscoe, O. Bondarenko, and I. Tsang, “DualQ Coupled AQMs for Low Latency, Low Loss and Scalable Throughput (L4S),” Internet Engineering Task Force, Internet Draft, July 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-aqm-codel-07

[72] S. Ha and I. Rhee, “Taming the elephants: New TCP slow start,” *Computer Networks*, vol. 55, no. 9, pp. 2092 – 2110, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128611000363

[73] S. Floyd, “Metrics for the Evaluation of Congestion Control Mechanisms,” Internet Engineering Task Force, RFC 5166, March 2008. [Online]. Available: https://tools.ietf.org/html/rfc5166

[74] A. Giessler, J. Haenle, A. KÃ¶nig, and E. Pade, “Free buffer allocation - an investigation by simulation,” *Computer Networks (1976)*, vol. 2, no. 3, pp. 191 – 208, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0376507578900284

[75] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system.* Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984, vol. 38.

[76] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker, “Dynamic behavior of slowly-responsive congestion control algorithms,” *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 263–274, Aug. 2001. [Online]. Available: http://doi.acm.org/10.1145/964723.383080

[77] Z. Wang and J. Crowcroft, “Eliminating periodic packet losses in the 4.3-tahoe bsd tcp congestion control algorithm,” *SIGCOMM Comput. Commun. Rev.*, vol. 22, no. 2, pp. 9–16, Apr. 1992. [Online]. Available: http://doi.acm.org/10.1145/141800.141801

[78] C. Jin, D. Wei, S. H. Low, J. Bunn, H. D. Choe, J. C. Doylle, H. Newman, S. Ravot, S. Singh, F. Paganini, G. Buhrmaster, L. Cottrell, O. Martin, and W. chun Feng, “Fast tcp: from theory to experiments,” *IEEE Network*, vol. 19, no. 1, pp. 4–11, Jan 2005.

[79] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, “Fast tcp: Motivation, architecture, algorithms, performance,” *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.886335

[80] S. Bhandarkar, A. L. N. Reddy, Y. Zhang, and D. Loguinov, “Emulating aqm from end hosts,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 349–360, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1282427.1282420

[81] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, “Timely: Rtt-based congestion control for the datacenter,” *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 537–550, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2829988.2787510

[82] M. Hock, F. Neumeister, M. Zitterbart, and R. Bless, “TCP LoLa: Congestion Control for Low Latencies and High Throughput,” in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, Oct 2017, pp. 215–218.

[83] J. Mo, R. J. La, V. Anantharam, and J. Walrand, “Analysis and comparison of tcp reno and vegas,” in *IEEE INFOCOM*, vol. 3. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 1999, pp. 1556–1563.

[84] G. Hasegawa, K. Kurata, and M. Murata, “Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet,” in *Network Protocols, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 177–186.

[85] U. Hengartner, J. Bolliger, and T. Gross, “TCP Vegas revisited,” in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, Mar 2000, pp. 1546–1555 vol.3.

[86] K. Srijith, L. Jacob, and A. Ananda, “TCP Vegas-A: solving the fairness and rerouting issues of TCP Vegas,” in *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, April 2003, pp. 309–316.

[87] L. Tan, C. Yuan, and M. Zukerman, “Fast tcp: fairness and queuing issues,” *IEEE Communications Letters*, vol. 9, no. 8, pp. 762–764, Aug 2005.

[88] C. Jin, S. H. Low, and X. Wei, “Method and apparatus for network congestion control,” Jul. 5 2011, uS Patent 7,974,195.

[89] C. Jin, S. Low, D. Wei, B. Wydrowski, A. Tang, and H. Choe, “Method and apparatus for network congestion control using queue control and one-way delay measurements,” Apr. 6 2010, uS Patent 7,693,052. [Online]. Available: https://www.google.com/patents/US7693052

[90] C. V. Hollot, V. Misra, D. Towsley, and W.-B. Gong, “On designing improved controllers for aqm routers supporting tcp flows,” in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, April 2001, pp. 1726–1734 vol.3.

[91] “The Network Simulator - ns-2,” ns-2. [Online]. Available: https://www.isi.edu/nsnam/ns/

[92] K. Kotla and A. L. N. Reddy, "Making a delay-based protocol adaptive to heterogeneous environments," in *2008 16th Interntional Workshop on Quality of Service*, June 2008, pp. 100–109.

[93] L. Budzisz, R. Stanojevic, A. Schlote, R. Shorten, and F. Baker, "On the fair coexistence of loss- and delay-based TCP," in *17th International Workshop on Quality of Service, IWQoS 2009, Charleston, South Carolina, USA, 13-15 July 2009.*, 2009, pp. 1–9. [Online]. Available: https://doi.org/10.1109/IWQoS.2009.5201387

[94] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "LEDBAT: The New BitTorrent Congestion Control Protocol," in *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, Aug 2010, pp. 1–6.

[95] "uTorrent Transport Protocol," Bittorrent.org. [Online]. Available: http://www.bittorrent.org/beps/bep_0029.html

[96] "libutp - uTorrent Transport Protocol implementation," BitTorrent Inc., 2010. [Online]. Available: https://github.com/bittorrent/libutp

[97] J. Schneider, J. Wagner, R. Winter, and H. J. Kolbe, "Out of my way - evaluating Low Extra Delay Background Transport in an ADSL access network," in *Teletraffic Congress (ITC), 2010 22nd International*, Sept 2010, pp. 1–8.

[98] R. Al-Saadi, G. Armitage, and J. But, "Characterising LEDBAT Performance Through Bottlenecks Using PIE, FQ-CoDel and FQ-PIE Active Queue Management," in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, Oct 2017, pp. 278–285.

[99] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye, "Ecn or delay: Lessons learnt from analysis of dcqcn and timely," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 313–327.

[100] A. Baiocchi, A. P. Castellani, and F. Vacirca, "YeAH-TCP: yet another highspeed TCP," in *Proc. PFLDnet*, vol. 7, 2007, pp. 37–42. [Online]. Available: http://www.gdt.id.au/~gdt/presentations/2010-07-06-questnet-tcp/reference-materials/papers/baiocchi+c

[101] . Budzisz, R. Stanojevic, A. Schlote, F. Baker, and R. Shorten, "On the fair coexistence of loss- and delay-based tcp," *IEEE/ACM Transactions on Networking*, vol. 19, no. 6, pp. 1811–1824, Dec 2011.

[102] V. Arun and H. Balakrishnan, "Copa: Practical Delay-Based Congestion Control for the Internet," in *15th USENIX Symposium on Networked Systems Design and*

Implementation (NSDI '18). USENIX Association, apr 2018. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi18/nsdi18-arun.pdf

[103] P. Goyal, A. Narayan, F. Cangialosi, D. Raghavan, S. Narayana, M. Alizadeh, and H. Balakrishnan, "Elasticity detection: A building block for delay-sensitive congestion control," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '18. New York, NY, USA: ACM, 2018, pp. 75–75. [Online]. Available: http://doi.acm.org/10.1145/3232755.3232772

[104] S. Floyd, "HighSpeed TCP for large congestion windows," Internet Engineering Task Force, RFC 3649, December 2003. [Online]. Available: https://tools.ietf.org/html/rfc3649

[105] T. Kelly, "Scalable tcp: Improving performance in highspeed wide area networks," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 83–91, Apr. 2003. [Online]. Available: http://doi.acm.org/10.1145/956981.956989

[106] A. H. David and G. Armitage, "Improved coexistence and loss tolerance for delay based TCP congestion control," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, Oct 2010, pp. 24–31.

[107] T. C. Tangenes, D. A. Hayes, A. Petlund, and D. Ros, "Evaluating CAIA Delay Gradient as a Candidate for Deadline-Aware Less-than-Best-Effort Transport," *Workshop on Future of Internet Transport (FIT 2017), Stockholm*, june 2017. [Online]. Available: http://dl.ifip.org./db/conf/networking/networking2017/1570350870.pdf

[108] N. Hu and P. Steenkiste, "Estimating available bandwidth using packet pair probing," CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, Tech. Rep., 2002.

[109] K. Lai and M. Baker, "Nettimer: A tool for measuring bottleneck link bandwidth." in *USITS*, vol. 1, 2001, pp. 11–11.

[110] V. Jacobson, "Pathchar: A tool to infer characteristics of internet paths," 1997.

[111] K. Lai and M. Baker, "Measuring link bandwidths using a deterministic model of packet delay," in *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4. ACM, 2000, pp. 283–294.

[112] R. King, R. Baraniuk, and R. Riedi, "TCP-Africa: an adaptive and fair rapid increase rule for scalable TCP," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3, March 2005, pp. 1838–1848 vol. 3.

[113] K. Tan, J. Song, Q. Zhang, and M. Sridharan, “A Compound TCP Approach for High-Speed and Long Distance Networks,” in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, April 2006, pp. 1–12.

[114] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Y. Sanadidi, and M. Roccetti, “Tcp libra: Exploring rtt-fairness for tcp,” in *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, I. F. Akyildiz, R. Sivakumar, E. Ekici, J. C. d. Oliveira, and J. McNair, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1005–1013.

[115] S. Liu, T. Başar, and R. Srikant, “TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks,” *Performance Evaluation*, vol. 65, no. 6, pp. 417–440, 2008.

[116] “Description of a hotfix that adds Compound TCP (CTCP) support to computers that are running Windows Server 2003 or a 64-bit version of Windows XP,” Microsoft Corporation. [Online]. Available: https://web.archive.org/web/20120413235134/http://support.microsoft.com/kb/949316

[117] P. Balasubramanian, “Updates on Windows TCP,” Microsoft, nov 2017. [Online]. Available: https://datatracker.ietf.org/meeting/100/materials/slides-100-tcpm-updates-on-windows-tcp

[118] R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi, “CapProbe: A Simple and Accurate Capacity Estimation Technique,” in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM ’04. New York, NY, USA: ACM, 2004, pp. 67–78. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015476

[119] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, “TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links,” in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom ’01. New York, NY, USA: ACM, 2001, pp. 287–297. [Online]. Available: http://doi.acm.org/10.1145/381677.381704

[120] L. A. Grieco and S. Mascolo, *TCP Westwood and Easy RED to Improve Fairness in High-Speed Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 130–146. [Online]. Available: http://dx.doi.org/10.1007/3-540-47828-0_9

[121] R. Wang, M. Valla, M. Y. Sanadidi, and M. Gerla, “Adaptive bandwidth share estimation in TCP Westwood,” in *Global Telecommunications Conference, 2002. GLOBECOM ’02. IEEE*, vol. 3, Nov 2002, pp. 2604–2608 vol.3.

[122] G. Yang, R. Wang, M. Y. Sanadidi, and M. Gerla, “TCPW with bulk repeat in next generation wireless networks,” in *Communications, 2003. ICC ’03. IEEE International Conference on*, vol. 1, May 2003, pp. 674–678 vol.1.

[123] S. M. ElRakabawy and C. Lindemann, “A Practical Adaptive Pacing Scheme for TCP in Multihop Wireless Networks,” *IEEE/ACM Transactions on Networking*, vol. 19, no. 4, pp. 975–988, Aug 2011.

[124] V. Konda and J. Kaur, “RAPID: Shrinking the Congestion-Control Timescale,” in *IEEE INFOCOM 2009*, April 2009, pp. 1–9.

[125] L. A. Grieco and S. Mascolo, “Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control,” *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 25–38, Apr. 2004. [Online]. Available: http://doi.acm.org/10.1145/997150.997155

[126] L. Zhang, S. Shenker, and D. D. Clark, “Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic,” *SIGCOMM Comput. Commun. Rev.*, vol. 21, no. 4, pp. 133–147, Aug. 1991. [Online]. Available: http://doi.acm.org/10.1145/115994.116006

[127] H. Shimonishi, M. Y. Sanadidi, and M. Gerla, “Improving efficiency-friendliness tradeoffs of TCP in wired-wireless combined networks,” in *IEEE International Conference on Communications, 2005. ICC 2005. 2005*, vol. 5, May 2005, pp. 3548–3552 Vol. 5.

[128] R. Wang, K. Yamada, M. Y. Sanadidi, and M. Gerla, “Tcp with sender-side intelligence to handle dynamic, large, leaky pipes,” *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 2, pp. 235–248, Feb 2005.

[129] M. Hock, R. Bless, and M. Zitterbart, “Experimental evaluation of BBR congestion control,” in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct 2017, pp. 1–10.

[130] J. C. Hoe, “Improving the start-up behavior of a congestion control scheme for tcp,” in *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM ’96. New York, NY, USA: ACM, 1996, pp. 270–280. [Online]. Available: http://doi.acm.org/10.1145/248156.248180

[131] S. Keshav, “A control-theoretic approach to flow control,” *SIGCOMM Comput. Commun. Rev.*, vol. 21, no. 4, pp. 3–15, Aug. 1991. [Online]. Available: http://doi.acm.org/10.1145/115994.115995

[132] C. Dovrolis, P. Ramanathan, and D. Moore, “Packet-dispersion techniques and a capacity-estimation methodology,” *IEEE/ACM Trans. Netw.*, vol. 12, no. 6, pp. 963–977, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1109/TNET.2004.838606

[133] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson, *TCP Vegas: New techniques for congestion detection and avoidance*. ACM, 1994, vol. 24, no. 4.

[134] R.-S. Cheng and H.-T. Lin, “Modified tcp startup procedure for large bandwidth-delay networks,” *J. High Speed Netw.*, vol. 17, no. 1, pp. 37–50, Jan. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404538.1404541

[135] F. S, A. M, J. A, and S. P, “Quick-Start for TCP and IP,” Internet Engineering Task Force, RFC 4782, January 2007. [Online]. Available: https://tools.ietf.org/html/rfc4782

[136] and P. Steenkiste, “Improving tcp startup performance using active measurements: algorithm and evaluation,” in *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, Nov 2003, pp. 107–118.

[137] D. Cavendish, K. Kumazoe, M. Tsuru, Y. Oie, and M. Gerla, “Capstart: An adaptive tcp slow start for high speed networks,” in *2009 First International Conference on Evolving Internet*, Aug 2009, pp. 15–20.

[138] F. S, “Limited Slow-Start for TCP with Large Congestion Windows,” Internet Engineering Task Force, RFC 3742, March 2004. [Online]. Available: https://tools.ietf.org/html/rfc3742

[139] J. Karlsson, A. Kassler, and A. Brunstrom, “Impact of packet aggregation on TCP performance in Wireless Mesh Networks,” in *2009 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks Workshops*, June 2009, pp. 1–7.

[140] F. Hui and P. Mohapatra, “Experimental characterization of multi-hop communications in vehicular ad hoc network,” in *Proceedings of the 2Nd ACM International Workshop on Vehicular Ad Hoc Networks*, ser. VANET ’05. New York, NY, USA: ACM, 2005, pp. 85–86. [Online]. Available: http://doi.acm.org/10.1145/1080754.1080770

[141] C. Lochert, B. Scheuermann, and M. Mauve, “A survey on congestion control for mobile ad hoc networks,” *Wireless Communications and Mobile Computing*, vol. 7, no. 5, pp. 655–676, 2007. [Online]. Available: http://dx.doi.org/10.1002/wcm.524

[142] A. P. Silva, S. Burleigh, C. M. Hirata, and K. Obraczka, “A survey on congestion control for delay and disruption tolerant networks,” *Ad Hoc Networks*, vol. 25, pp. 480 – 494, 2015, new Research Challenges in Mobile, Opportunistic and Delay-Tolerant Networks Energy-Aware Data Centers: Architecture, Infrastructure, and Communication. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870514001668

[143] Y. Gong, D. Rossi, C. Testa, S. Valenti, and M. Taht, “Fighting the bufferbloat: On the coexistence of AQM and low priority congestion control,” *Computer Networks*, vol. 65, pp. 255 – 267, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128614000188

[144] N. Iya, N. Kuhn, F. Verdicchio, and G. Fairhurst, “Analyzing the impact of bufferbloat on latency-sensitive applications,” in *Communications (ICC), 2015 IEEE International Conference on*, June 2015, pp. 6098–6103.

[145] “ CC_CDG(4) ,” FreeBSD Kernel Interfaces Manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=cc_cdg

[146] “ CDG FreeBSD implementation ,” FreeBSD official repository. [Online]. Available: https://svnweb.freebsd.org/base/release/12.0.0/sys/netinet/cc/cc_cdg.c?view=markup

[147] K. K. Jonassen, “Implementing caia delay-gradient in linux,” Master’s thesis, University of Oslo, Faculty of Mathematics and Natural Sciences, Department of Informatics, Oslo, Norway, 2015.

[148] Y. C. M. Mathis, N. Dukkipati, “Proportional Rate Reduction for TCP,” Internet Engineering Task Force, RFC 6937, September 2013. [Online]. Available: https://tools.ietf.org/html/rfc6937

[149] D. Hayes, “Timing enhancements to the FreeBSD kernel to support delay and rate based TCP mechanisms,” Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 100219A, 19 February 2010. [Online]. Available: http://caia.swin.edu.au/reports/100219A/CAIA-TR-100219A.pdf

[150] “H_ERTT(4),” FreeBSD Kernel Interfaces Manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=h_ertt

[151] R. Braden, “Requirements for Internet Hosts – Communication Layers,” Internet Engineering Task Force, RFC 1122, oct 1989. [Online]. Available: https://tools.ietf.org/html/rfc896.txt

[152] D. Borman, B. Braden, and V. Jacobson, “TCP extensions for high performance,” Internet Engineering Task Force, RFC 7323, September 2014. [Online]. Available: https://tools.ietf.org/html/rfc7323

[153] T. C. Tangenes, “Evaluating CAIA delay gradient as a Less-than-best-effort congestion control in Linux,” Master’s thesis, University of Oslo, Faculty of Mathematics and Natural Sciences, Department of Informatics, Oslo, Norway, 2016.

[154] “The Network Simulator - ns-3,” ns-3. [Online]. Available: https://www.nsnam.org/

[155] “Network Emulator II,” ixia. [Online]. Available: https://www.ixiacom.com/products/network-emulator-ii

[156] M. Carbone and L. Rizzo, “Dummynet revisited,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 12–20, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1764873.1764876

[157] S. Hemminger *et al.*, “Network emulation with NetEm,” in *Linux conf au*, apr 2005, pp. 18–23.

[158] “tc(8),” Linux man page, FreeBSD System Manager’s Manual. [Online]. Available: https://linux.die.net/man/8/tc

[159] “IPFW(8),” FreeBSD System Manager’s Manual, FreeBSD System Manager’s Manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?ipfw(8)

[160] S. FLOYD, “Recommendations on using the gentle variant of red,” *http://www.aciri.org/floyd/red/gentle.html*, 2000. [Online]. Available: https://ci.nii.ac.jp/naid/10015409941/en/

[161] “iperf Web Page.” [Online]. Available: http://iperf.fr/

[162] S. Zander and G. Armitage, “TEACUP v1.0 - A System for Automated TCP Testbed Experiments,” Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 150529A, 2015. [Online]. Available: http://caia.swin.edu.au/reports/150529A/CAIA-TR-150529A.pdf

[163] “tcpdump Web Page.” [Online]. Available: https://www.tcpdump.org/

[164] L. Stewart, “SIFTR - Statistical Information For TCP Research.” [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools.html

[165] S. Zander and G. Armitage, "Minimally-intrusive frequent round trip time measurements using synthetic packet-pairs," in *38th Annual IEEE Conference on Local Computer Networks*, Oct 2013, pp. 264–267.

[166] S. Zander and G. Armitage, "TEACUP project on Sourceforge," 2019. [Online]. Available: https://sourceforge.net/projects/teacup/

[167] "Codel and fq-codel in linux kernel," Bufferbloat.net, 2014. [Online]. Available: https://www.bufferbloat.net/projects/codel/wiki/

[168] V. Subramanian, "Pie aqm scheme," Kernel commit log, Jan 2014. [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=d4b36210c2e6ecef0ce52fb6c18c51144f5c2d88

[169] J. Kua, R. Al-Saadi, and G. Armitage, "Using Dummynet AQM - FreeBSD's CoDel, PIE, FQ-CoDel and FQ-PIE with TEACUP v1.0 testbed," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160708A, 08 July 2016. [Online]. Available: http://caia.swin.edu.au/reports/160708A/CAIA-TR-160708A.pdf

[170] G. Armitage, J. Kennedy, S. Nguyen, J. Thomas, and S. Ewing, "Household internet and the 'need for speed': evaluating the impact of increasingly online lifestyles and the internet of things," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 170113A, 13 January 2017. [Online]. Available: http://caia.swin.edu.au/reports/170113A/CAIA-TR-170113A.pdf

[171] "CoDel and FQ-CoDel Deployments," Bufferbloat. [Online]. Available: https://www.bufferbloat.net/projects/codel/wiki/

[172] W. Feng and S. Vanichpun, "Enabling compatibility between TCP Reno and TCP Vegas," in *Applications and the Internet, 2003. Proceedings. 2003 Symposium on*, Jan 2003, pp. 301–308.

[173] K. Kurata, G. Hasegawa, and M. Murata, "Fairness comparisons between tcp reno and tcp vegas for future deployment of tcp vegas," in *Proceedings of INET*, vol. 2000, no. 2.2, 2000, p. 2.

[174] A. Tang, J. Wang, S. Hegde, and S. H. Low, "Equilibrium and fairness of networks shared by tcp reno and vegas/fast," *Telecommunication Systems*, vol. 30, no. 4, pp. 417–439, 2005. [Online]. Available: http://dx.doi.org/10.1007/s11235-005-5499-1

[175] L. Budzisz, R. Stanojevic, R. Shorten, and F. Baker, "A strategy for fair coexistence of loss and delay-based congestion control algorithms," *IEEE Communications Letters*, vol. 13, no. 7, pp. 555–557, July 2009.

[176] S. R. L. D.J., "Impact of Drop Synchronisation on TCP Fairness in High Bandwidth-Delay Product Networks. ," in *Proc. Protocols for Fast Long Distance Networks, Nara, Japan.*, 2006.

[177] E. Gavaletz and J. Kaur, "Decomposing rtt-unfairness in transport protocols," in *2010 17th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, May 2010, pp. 1–6.

[178] "TCP Probe Kernel Module." [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/net/ipv4/tcp_probe.c?id=refs/tags/v4.1.6

[179] "KProbes." [Online]. Available: https://www.kernel.org/doc/Documentation/kprobes.txt

[180] Linux Foundation, "tcpprobe," 2009. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe

[181] "Sysstat Utilities." [Online]. Available: http://sebastien.godard.pagesperso-orange.fr/

[182] S. Zander and G. Armitage, "CAIA Testbed for TEACUP Experiments Version 2," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 150210C, May 2015. [Online]. Available: http://caia.swin.edu.au/reports/150210C/CAIA-TR-150210C.pdf

[183] R. Al-Saadi, "Dummynet patch to fix tick rate and ack compression problems," 2019. [Online]. Available: http://i4t.swin.edu.au/people/ralsaadi/uploads/Dummynet_tick_rate_and_ack_fix.patch

[184] "microuptime(9)," FreeBSD Kernel Developer's Manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=getmicrouptime&sektion=9&apropos=0&manpath=FreeBSD+11.2-RELEASE+and+Ports

Appendix A

ttprobe v0.1: Packet-Driven TCP Stack Statistics Gathering for TEACUP

Rasool Al-Saadi, Grenville Armitage, Jason But

CAIA Technical Report 150911A, September 2015

Abstract

TEACUP is an automated software tool that is used to run TCP/UDP experiments in a testbed with different operating systems including FreeBSD, Linux and Microsoft Windows. Internally, TEACUP uses many tools to emulate different networks conditions, generate network traffics, log different system and traffic statistics, analyse data, plot graphs and many other functions. To collect internal TCP information such Congestion Window Size (CWND) and smoothed TCP Round Trip Time (RTT), TEACUP uses operating system specific tools such as SIFTR in FreeBSD and Web10g in Linux. We developed ttprobe (TEACUP TCP probe), an alternative packet-driven TCP state logger for TEACUP experiments under Linux. At high packet rates ttprobe provides significantly more samples than web10g, with lower CPU overhead in many scenarios. ttprobe code is based on a loadable kernel module called TCP Probe.

A.1 Introduction

Analysis and understanding network protocols are very important to improve network performance. Researchers are looking forward for tools that simplify these processes and to get more accurate and realistic statistics. TEACUP is one of these tools that makes studying and enhancing network protocols easier and required less time than before [162]. TEACUP provides a controlled environment for doing TCP experiments on a testbed. Providing testbed with a controlled environment is very important for doing TCP experiments, such as analysis congestion control algorithms, to get accurate and comparable results. TEACUP uses TCP loggers to capture internal TCP statistics and information from inside TCP/IP stack such as congestion window size (CWND) and smoothed TCP Round Trip Time (RTT). In FreeBSD, TEACUP uses SIFTR logger which can collect per packet statistics and it has been included in FreeBSD kernel since version 8.2-RELEASE [164]. On the other hand, in Linux environment, TEACUP uses Web10g [26] which is a kernel patch and user space tools that allows users to capture TCP statistics per time interval of 1 millisecond or more.

As an improvement to TEACUP and Linux TCP logger, we developed ttprobe logger (TEACUP TCP probe) which is a per-packet TCP statistics logger that can capture detailed and accurate TCP statistics. Moreover, we adapted TEACUP code to be compatible with our logger. ttprobe brings many benefits to TEACUP when it is used with Linux hosts including detailed statistics capturing and reasonable CPU overhead. ttprobe code is based on a loadable kernel module called TCP Probe [178]. The ttprobe source code is available as part of the official TEACUP source distribution [166].

A.2 TCP Probe

TCP probe is a Linux loadable kernel module that able to capture different information of TCP connections on each incoming packet. TCP probe hooks `tcp_rcv_established` kernel function by using kprobes framework [179], so the internal TCP/IP stack structures that contain TCP/IP variables, such as congestion window, slow start threshold (ssthresh) and acknowledge number (ACK), can be accessed and captured [180]. TCP probe also provides a basic filter that can be set up to capture only data of a specific TCP port number and/or when CWND is changed. When the module is loaded, it creates a virtual file, named tcpprobe, in `/proc/net` directory which is used to transfer TCP probe output to user space processes. Basically, TCP probe can be started using the following shell commands:

```
> modprobe tcp_probe port=0 full=1
> cat /proc/net/tcpprobe >/tmp/out.log
```

These commands will log TCP statistics for all TCP flows on each TCP packet arrive.

An indication of the data format as output by TCP probe can be seen in Table A.1. Figure A.1 is a sample of the TCP probe output for a small subset of data from a real experiment.

Table A.1: TCP Probe output headings

Column	**Contents**
1	Kernel Timestamp
2	Source_IP:port
3	Destination_IP:port
4	Packet Length
5	Send Next
6	Send Unacknowledged
7	Send window
8	Receive window
9	Congestion windows
10	ssthresh
11	Smoothed RTT

```
1.622631348 192.168.1.101:22 192.168.1.100:52680 84 0x3e4fafe 0x3e4fafe 42 2147483647 65024 25300 34688
1.822848521 192.168.1.101:22 192.168.1.100:52680 20 0x3e4fb3e 0x3e4fafe 10 2147483647 65024 25300 34688
1.950667701 192.168.1.101:22 192.168.1.100:52680 84 0x3e4fb3e 0x3e4fb3e 10 2147483647 65024 47144 34688
1.957206390 192.168.1.101:22 192.168.1.100:52680 20 0x3e52d1a 0x3e4fb3e 10 2147483647 65024 47144 34688
```

Figure A.1: A sample of TCP Probe output

A.3 Overview of ttprobe v0.1

ttprobe (TEACUP TCP Probe) is a packet-driven Linux TCP instrumentation that can collect per-packet TCP statistics on each incoming and/or outgoing packet. ttprobe code is based on TCP probe source-code with many improvements to meet TEACUP requirements.

Getting accurate and detailed statistics are very important for protocols analysis. As TEACUP uses Web10g framework which is a time interval based method of a minimum of 1 millisecond, this causes TEACUP to miss many useful samples especially when link speed is high.

Another issue with Web10g is that it requires a patched kernel to be functional, and patching a kernel is highly depending on its version. The latest (at the report writing time) stable Web10g kernel patch is for Linux kernel 3.17. This means, currently[1], there is no applicable Web10g kernel patch for newer stable Linux kernels versions such as 4.0.9 and 4.1.3. Moreover, Web10g puts a high impact on CPU usage due to the time interval even when there is a very low traffic flow in the network.

On the other hand, ttprobe's per-packet data collection ensures virtually[2] no TCP state changes are missed. Furthermore, ttprobe consumes lower processing power than Web10g

[1]This report was written on 2nd of September 2015

[2]When ttprobe's buffer size is too small and there is a very high traffic, ttprobe silently drops some samples.

and it does not require patching the core kernel to work. However, some parts of ttprobe source code should be modified if there are substantial changes in the kernel socket structure and the hooked functions prototypes.

As ttprobe v0.1 is built on top of kprobes framework, ttprobe requires Linux kernel compiled with kprobes support to be functional. New versions of many Linux distributions, such as Ubuntu, Debian, CentOS and openSUSE come with kprobes enabled kernels.

A.4 ttprobe Development

The main features that exist in ttprobe but not in TCP probe are:

1. The output timestamps are date/time timestamps (`timeval`) while in TCP probe are relative kernel timestamps (`k_time`). This change is very important to make TEACUP works properly with the output.

2. Hooking `tcp_v4_do_rcv` and `tcp_v6_do_rcv` (for incoming packets) and `tcp_transmit_skb` (for outgoing packets) instead of just `tcp_rcv_established` (for incoming packets). This change makes ttprobe able to log TCP information in all TCP connection states and on every incoming and/or outgoing packet.

3. Collecting additional TCP states such as maximum segment size (MSS) and TCP connection state.

4. Providing three output formats which are ttprobe, binary and web10g format.

5. Changing the module name and virtual file name to make ttprobe works without any interference with the original TCP probe module.

6. Implementing a buffer flushing function to flush TCP probe kernel buffer to user space buffer. This is very important to make sure that all the data inside ttprobe buffer is logged to user space buffer at the end of the experiment without any lost.

ttprobe provides many parameters that can be set up before starting the logger. A list of available ttprobe parameters is shown in Table A.2. The parameters have a format of *parameter=value* and must be written in the same command that is used to load the module as follow:

```
> modprobe ttprobe parameter1=value1 parameter2=value2 ....
```

Table A.2: ttprobe parameters

Option	**Description**
`omode`	Output mode ‘0’ for ttprobe format (default). ‘1’ for binary output. ‘2’ for web10g format.
`direction`	Capturing direction ‘0’ on every outgoing packet. ‘1’ on every incoming packet (default). ‘2’ on every incoming and outgoing packet
`port`	Source or destination port number to match default: ‘0’ (capture all port)
`bufsize`	ttprobe buffer size in packet default: 8192
`full`	Full log or just when CWND changed ‘0’ ttprobe will capture samples just when CWND value is changed. ‘1’ ttprobe will log on every packet (default).

Table A.3: ttprobe output headings

Column	Contents
1	Direction (i or o)
2	Timestamp
3	Source IP addr.
4	Source port no.
5	Destination IP addr.
6	Destination port no.
7	Packet counter
8	MSS
9	Smoothed RTT
10	Congestion windows
11	ssthresh
12	Send window
13	Receive window
14	Socket state
15	Send unacknowledged
16	Send next
17	Packet size

Table A.3 shows columns descriptions of ttprobe output format. Binary format can be used to reduce log file size and CPU load. Our TEACUP patch all TEACUP to read binary format and ttprobe format natively. Moreover, we developed a small tool, called `ttpb2ascii`, that can decode binary format log file and produce ttprobe ASCII format. The disadvantage of using binary mode is it increases the execution time of data analysis functions.

After loading ttprobe module, it will create a virtual file with path-name `/proc/net/ttprobe`. This file is used to read the log data from the kernel to a user space process, and to send commands to ttprobe module. ttprobe commands can be sent to the module from a shell using `echo` command.

```
> echo "command" > /proc/net/ttprobe
```

Currently, there are two commands that can be used with ttprobe which are `flush` command that used to flush ttprobe kernel buffer, and `finish` command that is used to send end of file signal to the reader process.

ttprobe was tested on Linux kernel 3.18 and 4.1, but it should work fine on any kernel version higher than 3.0.

Example of using ttprobe from Linux shell:

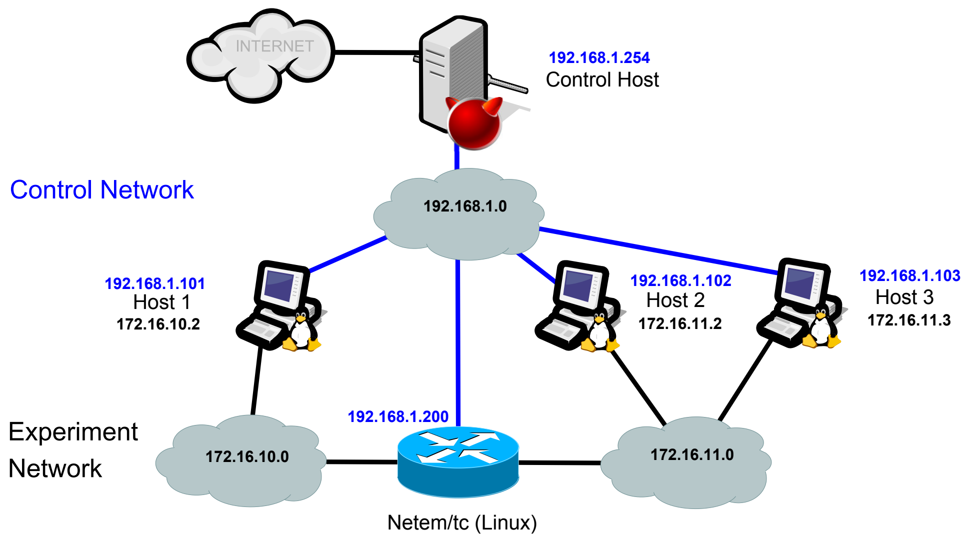


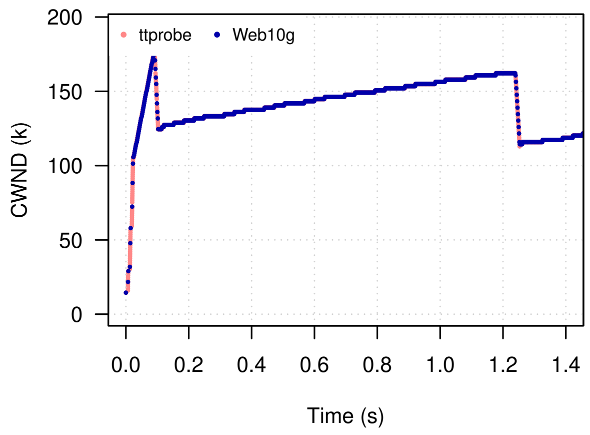
Figure A.2: TEACUP Testbed topology

```
> modprobe ttprobe direction=2 omode=0 port=5000 bufsize=16384
   full=1
> cat /proc/net/ttprobe > /tmp/ttprobe.log&
> # waiting for an experiment to complete
> echo "flush" > /proc/net/ttprobe
> sleep 0.5
> echo "finish" > /proc/net/ttprobe
> rmmod ttprobe
```

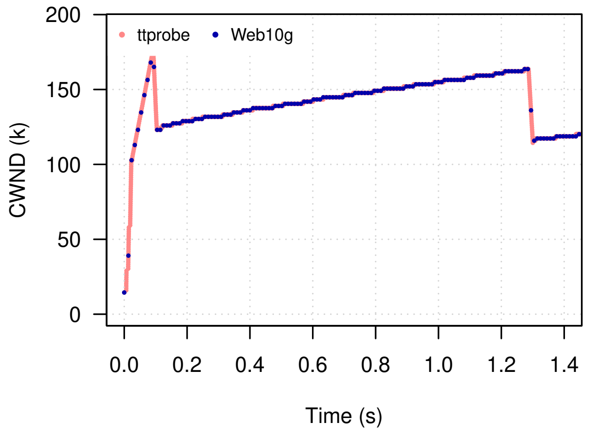
A.5 Experimental comparisons of Web10g and ttprobe v0.1

We did practical experiments to compare the details of captured data and the CPU load when Web10g or ttprobe is used. TEACUP tool was used in all our experiments.

The testbed that we used in the experiments includes three hosts, one bottleneck and a control host. The hosts and the bottleneck are normal PCs with Intel Core 2 Due @ 3GHz processes, 4GiB RAM and Gigabit and Fast Ethernet cards, and the control host is a Virtualbox virtual machine. Host 1 is connected directly to the bottleneck while host 2 and host 3 are connected through a Gigabit Ethernet switch, and the switch is connected to the bottleneck. All machines having additional Ethernet cards to be connected to the control network. The host machines and bottleneck run Linux 3.17.4 while the control host runs FreeBSD 10.1. Figure A.2 shows the network topology of the testbed that was used in our experiments.



(a) Using ttprobe and Web10g loggers at the same time (Web10g poll interval is 1 ms)



(b) Using ttprobe and Web10g loggers at the same time (Web10g poll interval is 10 ms)

Figure A.3: The first 1.4 seconds of CWND versus time plot captured using ttprobe and Web10g

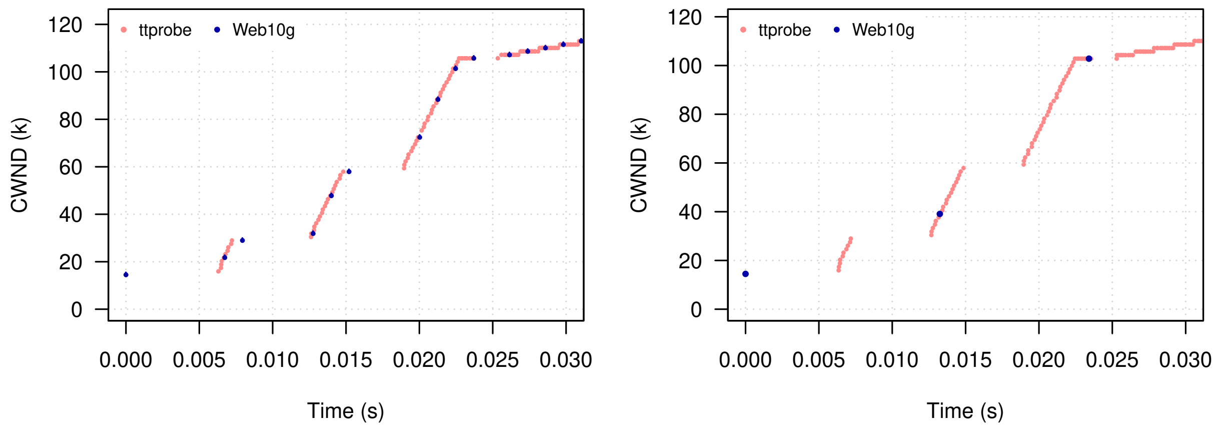
A.5.1 Comparing the Details of the Captured TCP Data Samples

Firstly, we did two identical experiments (two individual runs) except that one has 1 ms Web10g poll interval and the other has 10 ms Web10g poll interval. In these experiments, the bottleneck shapes the traffic to 100 Mbps and emulates 6 ms RTT. Both Web10g and ttprobe ran together, iperf was used as a traffic generator and the congestion control algorithm is TCP CUBIC.

Tacking CWND values as a sample, we notice that web10g misses many samples in TCP rapid actions such as slow start stage and TCP congestion back-off event. These samples losses happen because TCP stack changes CWND size many times in one web10g poll time interval. As Web10g poll time interval is set to a high value, Web10g samples losses will become even bigger.

Figure A.3a and Figure A.3b show the first 1.4 second of CWND graphs for the first experiment (1 ms Web10g poll interval) and second experiment (1 ms Web10g poll interval) respectively. These figures illustrate that web10g missed many CWND values during the experiment even when the time interval was very short as opposite as ttprobe which captured all CWND samples. As a note, Web10g data points are plotted on top of ttprobe data points in all the graphs in this report.

Figure A.4a and Figure A.4b show zoomed in CWND plot of the first 30 ms of these experiments. These figures show more clearly the details of ttprobe CWND plot and how Web10g missed many data points especially during the slow start stage.



(a) Using ttprobe and Web10g loggers (Web10g poll interval is 1 ms)

(b) Using ttprobe and Web10g loggers (Web10g poll interval is 10 ms)

Figure A.4: A zoomed in of CWND versus time plot captured using ttprobe and Web10g

A.5.2 Comparing CPU Overhead of Web10g and ttprobe

In this section, our goal is to compare CPU overhead of Web10g and ttprobe. We utilised TEACUP tool with additional function that logs CPU utilization to get CPU usage in different scenarios. Additionally, `tcpdump` was disabled in TEACUP during the experiment to get CPU utilisation that related to the loggers as much as possible.

For both loggers, there is no way to get CPU overhead specifically for the logger. This because a big part of Web10g code is injected inside TCP/IP stack which is run as a part of the kernel, and the largest part of ttprobe is a kernel module that creates hooks to kernel functions (similar to a callback).

Obtaining comparative CPU usage from specific sections within the kernel is more than we require to do a simple comparison of how the choice of Web10g or ttprobe impacts on aggregate end-system load during TEACUP experiments.

Using averaged CPU load every one second is sufficient for comparison purpose as there are no big changes in the CPU load during the actual experiment load. System Activity Reporter utility (`sar`) , which it is part of Sysstat Utilities [181], was used to log CPU usage during the experiments.

First, we did a TEACUP experiment to understand CPU load behavior during TEACUP experiment. This experiment ran for 60 second and it included one TCP flow transmitted between two computer (host 2 and host 3) connected through 1Gbps link.

Figure A.5 shows the percentage of CPU usage relative to maximum CPU capacity of the sender machine when Web10g logger is used. CPU usage is including traffic generator and other default system processes. In this figure, we can see that there are three regions, TEACUP initialisation, experiment load and TEACUP finalisation regions.

During the initialisation stage, TEACUP configures the host, collects different information about the host, starts loggers and then starts traffic generators. During the finalisation stage, TEACUP stops traffic generators, stops loggers and collect log files. The important stage for comparing TCP loggers is the experiment load stage.

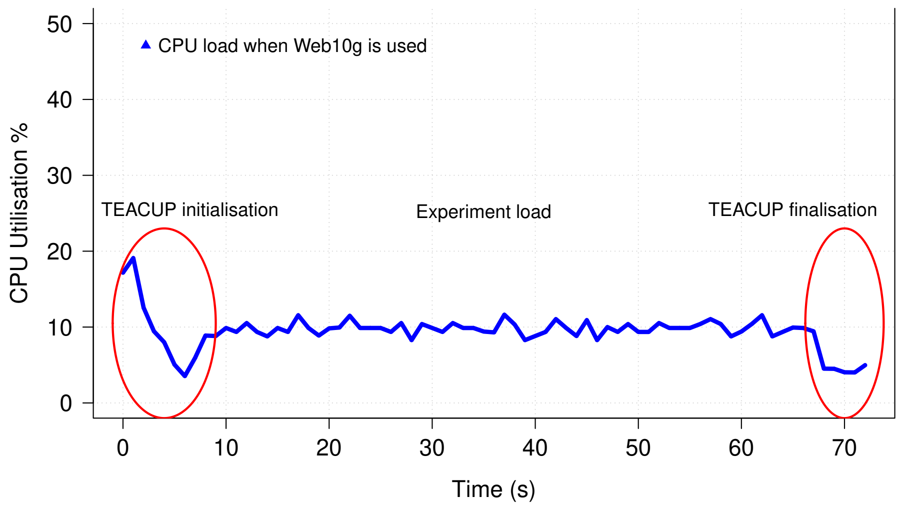


Figure A.5: CPU usage % relative to maximum CPU capacity of the sender machine including traffic generator and other default system processes CPU utilisation when Web10g is used

To compare CPU overhead of using Web10g and ttprobe, we did five TEACUP experiments of 60 seconds duration and 10 runs each. The first experiment was run without TCP logger, the second one with ttprobe logs just on receiving packets, the third one with ttprobe logs on sending/receiving packets, the fourth one with Web10g logger and 1 ms poll interval and the last one with Web10g logger and 10 ms poll interval.

In these experiments, there were two PCs (host 2 and host 3) which having 1Gbps Ethernet cards and connected through 1Gbps network switch. This is considered the maximum throughput that can be achieved in our testbed. For each experiment, we extracted CPU utilisation for the experiment load period (t=15 s to t=65 s) of each run.

We then calculated cumulative distribution function (CDF) for each experiment using the extracted CPU utilisation data of the ten runs.

Figure A.6 shows CDF of CPU utilisation for the four experiments. This figure illustrates that ttprobe (logging on sending/receiving packets) consumes less processing power than Web10g when Web10g poll interval is 1 ms, and slightly more than Web10g when Web10g poll interval is 10 ms at the same testbed condition. The figure also shows that ttprobe (logging on receiving packets) consumes less processing power than Web10g in all cases.

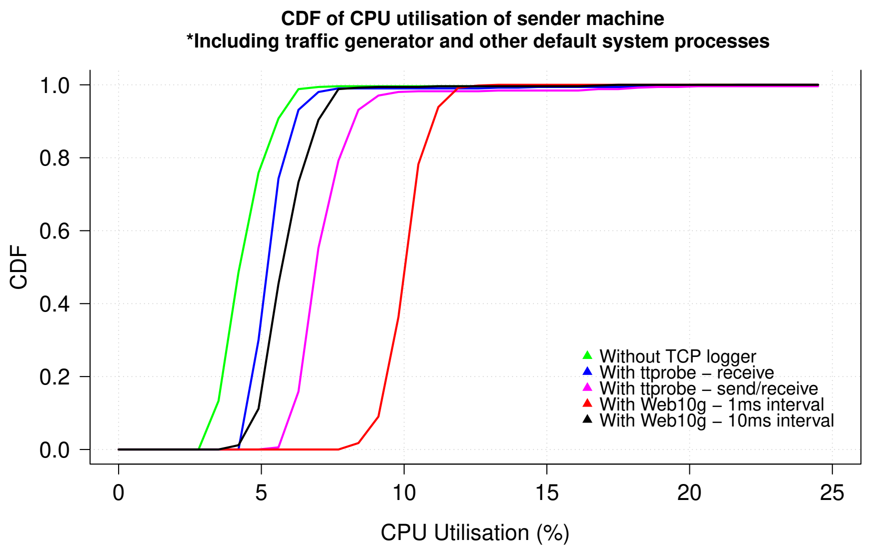


Figure A.6: CDF plot of CPU utilisation for five experiments with different TCP logger in each experiment (link speed is 1Gbps)

To understand how link speed affects CPU overhead, we replicated the five experiments but with traffic shaping of 100 Mbps and emulated base RTT of 6 ms. Figure A.7 shows CDF of the five experiments with different loggers when the link speed is 100 Mbps. This figure illustrates that in this scenario, ttprobe overhead is highly depends on traffic speed.

A.6 Installation Procedures

To make TEACUP working properly with ttprobe logger, ttprobe module should be install in all TEACUP hosts and ttprobe TEACUP patch must be applied to TEACUP code in the control host.

A.6.1 ttprobe Installation Procedure

Before starting the compilation and installation process, the system must be updated and all software dependencies must be installed. The required packages to be installed are:

1. make

2. gcc

3. kernel-devel and/or linux-headers

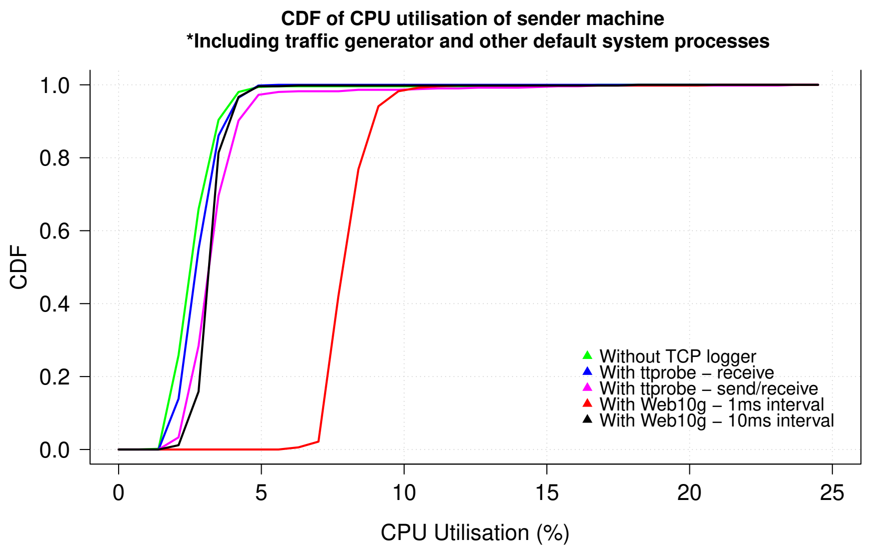


Figure A.7: CDF plot of CPU utilisation for five experiments with different TCP logger in each experiment (link speed is 100 Mbps)

The procedure of updating the system and installing the dependencies are depending on Linux distribution. Table A.4 shows dependencies installation commands for some Linux distributions. All commands must be executed by super user (root) account.

ttprobe module can be installed by using the following procedure:

1. Extract `ttprobe-0.1.tar.gz` archive file.

```
> tar xzvf ttprobe-0.1.tar.gz
```

2. Compile the module.

```
> cd ttprobe-0.1
> make
```

3. If all the dependencies are installed correctly, ttprobe.ko file should be created in the current directory.

```
> ls ttprobe.ko
```

4. Copy ttprobe kernel module to Linux kernel modules directory.

```
> mkdir -p /lib/modules/$(uname -r)/extra
> cp -r ttprobe.ko /lib/modules/$(uname -r)/extra/
```

5. Update modules dependency descriptions.

Table A.4: ttprobe Linux dependencies installation commands

Linux Distribution	Tested on	Commands

Ubuntu	14.04.2-desktop-amd64[a]	`>apt-get update`
		`>apt-get install make gcc linux-headers-$(uname -r)`

Debian	8.2.0-amd64	`>apt-get update`
		`>apt-get make gcc install linux-headers-$(uname -r)`

CentOS	7-x86_64	`>yum update`
		`>yum install make gcc kernel-devel kernel-headers`

openSUSE	13.2-x86_64	`>zypper update`
		`>zypper install make gcc kernel-devel`

[a]This version comes with all required dependencies installed by default.

Table A.5: TEACUP `TPCONF` variables for ttprobe logger

Option	**Description**
`TPCONF_linux_tcp_logger`	TCP logger to be used in Linux hosts ‘`web10g`’ web10g only (default). ‘`ttprobe`’ ttprobe only. ‘`both`’ to use ttprobe and web10g together.
`TPCONF_ttprobe_direction`	Capturing direction ‘o’ on every outgoing packet. ‘i’ on every incoming packet. ‘io’ on every incoming and outgoing packet (default).
`TPCONF_ttprobe_output_mode`	ttprobe output mode ‘0’ for ttprobe format (default). ‘1’ for binary output.

```
> depmod $(uname -r)
```

If all TEACUP hosts have same Linux kernel version, it is easier to compile ttprobe kernel module on one machine and copy it to all hosts machines (use steps 4 and 5 of ttprobe installation procedure).

A.6.2 Applying ttprobe’s TEACUP patch

We created a patch for TEACUP v1.0 to support ttprobe logger natively. This patch allows TEACUP to start/stop ttprobe logger, and makes `analys_*/extract_*` functions working properly with ttprobe output file (file name ends with `_ttprobe.log.gz`). The procedure of applying the patch to TEACUP code is as follow:

1. Install TEACUP v1.0, if it is not already installed, by following the instructions in CAIA technical reports [162][182].

2. Extract `ttprobe-0.1.tar.gz` archive inside the TEACUP directory <teacup_directory>[3] using the following commands:

```
> cd <teacup_directory>
> tar xzvf ttprobe-0.1.tar.gz
```

3. Apply the patch to TEACUP.

[3] <teacup_directory> is the full path to your TEACUP installation.

```
> patch -p1 <  ttprobe-0.1/teacup-ttprobe-0.1.patch
```

A.7 TEACUP Configuration

TEACUP configuration file (`config.py`) should include some additional `TPCONF` variables to setup ttprobe options. Table A.5 lists TEACUP `TPCONF` variables that are used with our TEACUP patch.

If both loggers are chosen to be used in an experiment, `LINUX_TCP_LOGGER` environment variable must be set to either '`ttprobe`' or '`web10g`' in order to select which logger output will be used in TEACUP `analyse_*/extract_*` functions.

A.8 Conclusions and Future Work

ttprobe module has many benefits over Web10g with respect to the details of the captured information, kernel patching and CPU overhead load. Moreover, the installation process of ttprobe is much easier than Web10g. ttprobe can easily integrate with virtually any Linux kernel with version higher than 3.0 compiled with kprobe support. However, Web10g collects more TCP statistics than ttprobe. For this reason, our update to TEACUP gives the choice to the users to select the desired TCP logger depending on their needs, as well as the option to use Web10g and ttprobe together. As a future work, ttprobe requires more development to add more TCP statistics to its output and improve its filters.

Appendix B

Dummynet network emulation challenges

This appendix discusses challenges and solutions for Dummynet network emulation. Section B.1 discusses the unintended TCP ACK compression issue and our solution. Section B.2 explores the problem and solutions for Dummynet excess delay emulation. Our Dummynet patch that solves these issues is available online on [183].

In Section B.3, we discuss the implication of the kernel tick rate on the emulation quality, and presents our Dummynet timing enhancement. We conclude this Appendix in Section B.4

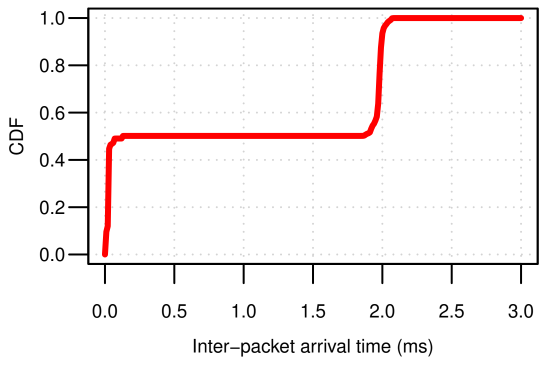
B.1 Unintended TCP ACK compression

Due to Dummynet's architecture and the relatively low operating system tick rate, Dummynet compresses the gaps between ACK packets (unintended ACK compression) in specific situations. Dummynet's traffic shaper regulates packet transmission rate using the leaky bucket technique. To emulate the desired link bandwidth, Dummynet uses the `credit` variable which contains the calculated number of bytes that can be transmitted at an instance of time. Periodically, packets are extracted from the queue until `credit` becomes negative or no more packets available in the queue. Negative `credit` means the last packet has been dequeued too early. Therefore, Dummynet calculates the serialisation delay of the bytes that dequeued too early and adds an extra delay to the packet in the delay line.

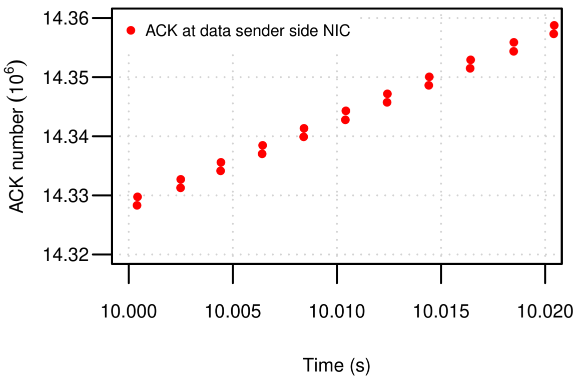
Since Dummynet relies on kernel's ticks as a unit of timer, the time resolution highly depends on the kernel tick rate. For example, FreeBSD11 has 1000 ticks per second by default[1] which provides 1ms resolution. Therefore, the minimum delay adjustment for a packet is 1ms.

Whenever `credit` becomes negative, Dummynet adds one tick (1ms) to the transmitting time of the last packet. Additionally, if the queue becomes empty and a new packet arrives,

[1]Executing `sysctl kern.hz` shows 1000



(a) CDF for inter-packet arrival time



(b) ACK progression shows Dummynet ACK compression effect

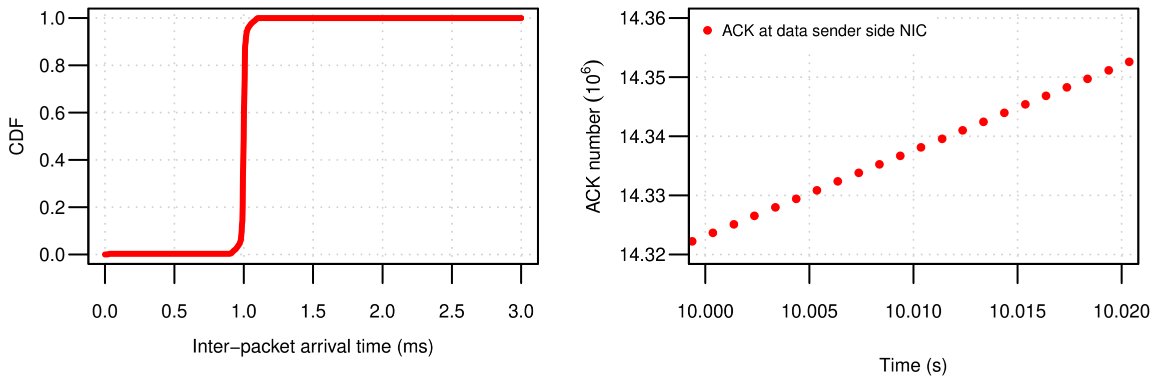
Figure B.1: Dummynet ACK compression: ACK captured at the sender side for a single 60 second NewReno TCP flow, 12Mbps, 40ms RTT and 100pkt bottleneck buffer size

`credit` is set to zero. Although this method prevents packets from being sent faster than the configured bandwidth, it leads to unintended ACK compression.

To explore this issue, assume bulk data is transmitted using TCP protocol over a Dummynet bottleneck configured to shape the traffic at 12Mbps for both directions. Dummynet will send one 1500 byte packet every 1ms to emulate the required bandwidth. If the receiver does not use the TCP delayed acknowledgement algorithm [151], it will send one ACK packet every 1ms as a reflection to the data packets. However, that does not happen in our experiment due to the transmitting time adjustment performed by the traffic shaper. The CDF plot for ACK inter-packet arrival time logged at the sender host is shown in Figure B.1a. This figure illustrates that around 50% of ACK packets are sent in burst and the other 50% are sent after 2ms. Moreover, we can see in Figure B.1b ACK packets arrive to the sender in bursts of two packets every 2ms.

In this experiment, Dummynet resets `credit` on every other tick for the reverse path pipe because the bitrate of the reverse (ACK) path is smaller than the configured bandwidth (12Mbps). When `credit` is zero, a dequeued packet is considered too early to send and packet transmission is delayed by one tick. Since this packet leads to a busy scheduler instance, `credit` will not be reset. As a result, next ACK packet (on next tick) will not be delayed. This results in two ACK packets to have the same transmission time leading to the ACK compression behaviour.

To remedy this problem, packet transmission time adjustment should be greater than or equal to zero (instead of one or more ticks). Although this solution could result in periodically sending at most one packet slightly faster, the average throughput will no be more than the



(a) CDF for inter-packet arrival time

(b) ACK progression shows ACK compression effect disappeared after patching Dummynet

Figure B.2: Fixing Dummynet ACK compression: ACK captured at the sender side - A single 60 second NewReno TCP flow, 12Mbps, 40ms RTT and 100pkt bottleneck buffer size

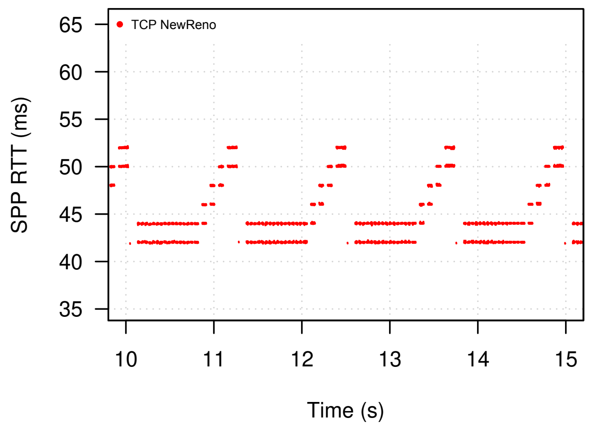
configured bandwidth.

After applying this solution to Dummynet and repeating the same experiment, the ACK compression behaviour reduces significantly as shown in Figure B.2. The CDF graph for ACK inter-packet arrival time (shown in Figure B.2a) illustrates that most packets are sent regularity every millisecond. Furthermore, Figure B.2b shows ACK packets are sent smoothly with burst.

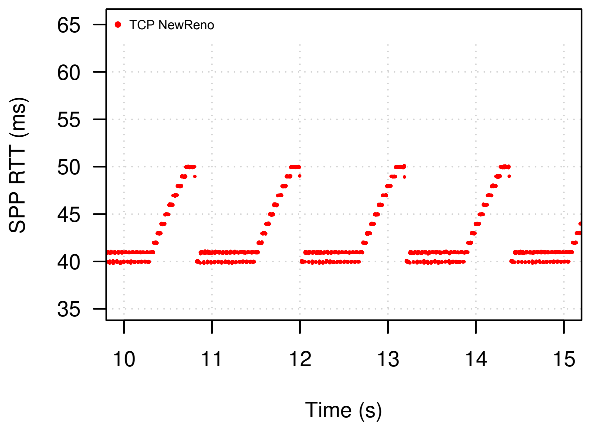
Another useful solution that can be used is to increase kernel's tick rate to make the adjustment much finer. However, the ACK compression problem will appear again when the length of one tick equals the serialisation delay of one packet for the configured bandwidth. For example, if the tick rate is set to 10KHz, the ACK compression will appear when the bandwidth is configured to 120Mbps. Combining the two solutions produces better and more accurate network emulation.

B.2 Excess emulated delay

The out time adjustment for packets that causes ACK compression (see Section B.1) also leads to additional latency for packets having adjusted transmitting time. In fact, almost no packet will be sent without an extra delay whether due to queuing delay or packet transmitting time adjustment delay. The only exception is the last packet in the queue that does not causes `credit` to be negative (i.e. causing the scheduler to go to idle state). Additionally, due to kernel's 1000 tick rate (default), 1ms error can be added to the emulated delay for each direction [156]. Therefore, the measured RTT at the end hosts will include additional 2ms above the



(a) SPP RTT plot for a flow passed through unmodified Dummynet bottleneck



(b) SPP RTT plot for a flow passed through patched Dummynet bottleneck

Figure B.3: Dummynet excess emulated delay: single NewReno TCP flow, 12Mbps, 40ms RTT and 10pkt bottleneck buffer size

sum of configured delays to the forward and reverse paths.

Figure B.3a shows SPP RTT versus time for 5 seconds of a single TCP NewReno flow traversing a Dummynet bottleneck that emulates 12Mbps link speed, 40ms path RTT and has 10pkt[2] buffer size. We can see in this figure that RTT_{min} is around 42ms because of transmitting time adjustment of Dummynet's traffic shaper.

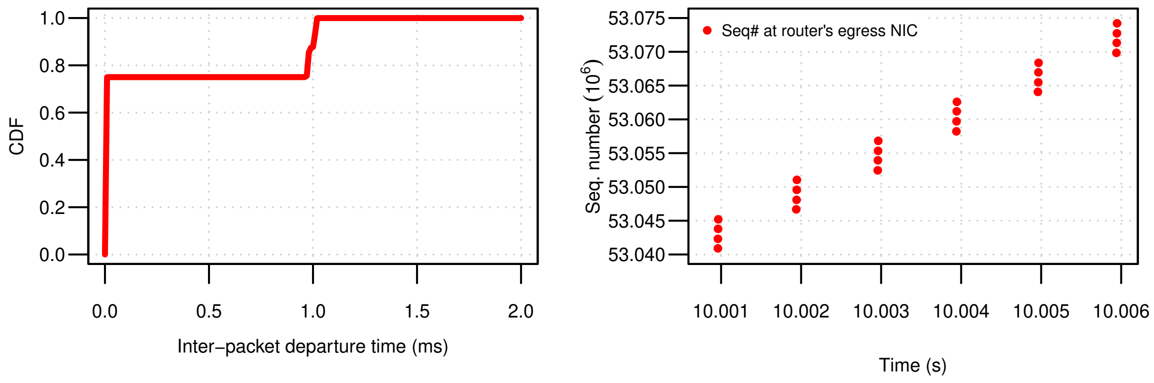
Implementing the solutions described in Section B.1 reduces the extra delay significantly as shown in Figure B.3b. We can see in this figure RTT_{min} is about 40ms which is the same as the configured delays of this experiment without excess emulated delay. We can also see that RTT steps is around 1ms which equals to the serialisation delay of the emulated link. Again, higher kernel tick rate can improve time resolution and reduce the delay emulation errors.

B.3 The kernel tick rate implication on Dummynet emulation accuracy

Dummynet uses timer driven events to serve the scheduler(s), shape the traffic, and transmit packets from the delay line(s). Dummynet schedules these events on every tick. As a consequence, delay emulation error in Dummynet is $\frac{1}{tickrate}$ second.

This timer generates bursty traffic when the packet serialisation delay of the configured traffic shaper is larger than $1/tickrate$ second i.e. when the kernel's tick rate is less than

[2]Small buffer size is used to allow queue draining after cwnd backing-off



(a) CDF for inter-packet departure time (b) SEQ number progression shows packets bursts

Figure B.4: Dummynet packet burst behaviour due to low kernel tick rate: data packets captured at the router egress NIC - single 60 second NewReno TCP flow, 48Mbps, 40ms RTT, and 200pkt bottleneck buffer size at 1KHz tick rate

packet sending rate.

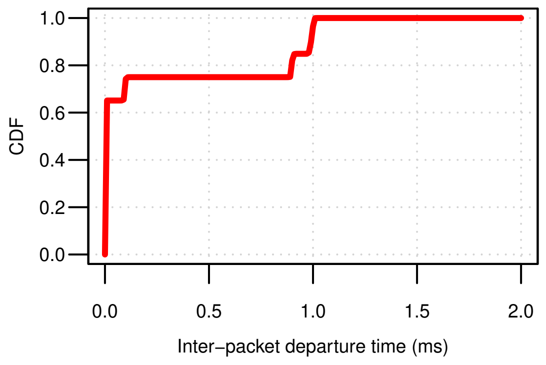
For example, consider if Dummynet is configured to emulate a 48Mbps link at 1000Hz kernel tick rate and the sender host is transmitting data at the same speed. Dummynet will aggregate four packets and send them in bursts on every tick. The reason is that Dummynet will serve the scheduler(s) and delay line(s) once every 1ms at that tick rate. However, the events should occur every 250µs since the serialisation delay of one packet[3] is 250µs at that sending speed. Thus, the only way to achieve the required rate is to aggregate four send events into one event for 75% of the packets so the averaged sending rate will be 48Mbps $(0.75 \times 0ms + 0.25 \times 1000ms = 250\mu s)$.

To practically demonstrate this behaviour, we conduct an experiment using Dummynet bottleneck with the kernel tick rate set at 1000Hz (default). The experiment consists of running a single 60 second NewReno TCP flow over a symmetric 48Mbps emulated link and 40ms emulated path RTT with 200pkt bottleneck buffer size. As expected, bursts of four packets are transmitted every one millisecond as shown in Figure B.4b.

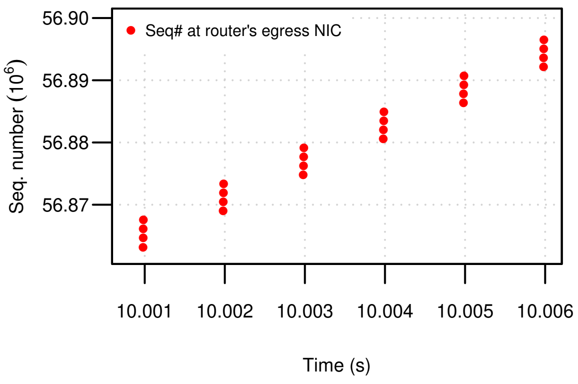
The CDF graph of inter-packet departure time of the data packets [4] shown in Figure B.4a illustrates that about 75% of data packets are sent in bursts while 25% of them are sent after one millisecond of the previous packets. This does not reflect how real networks function and produces unrealistic network emulation.

It is well known that increasing the kernel tick rate improves timer resolution, leading to fewer emulation errors and more precise link emulation [156]. However, Dummynet was de-

[3]Assume one packet = 1500 bytes
[4]captured at the router egress NIC



(a) CDF for inter-packet departure time



(b) TCP sequence number progression shows packets bursts

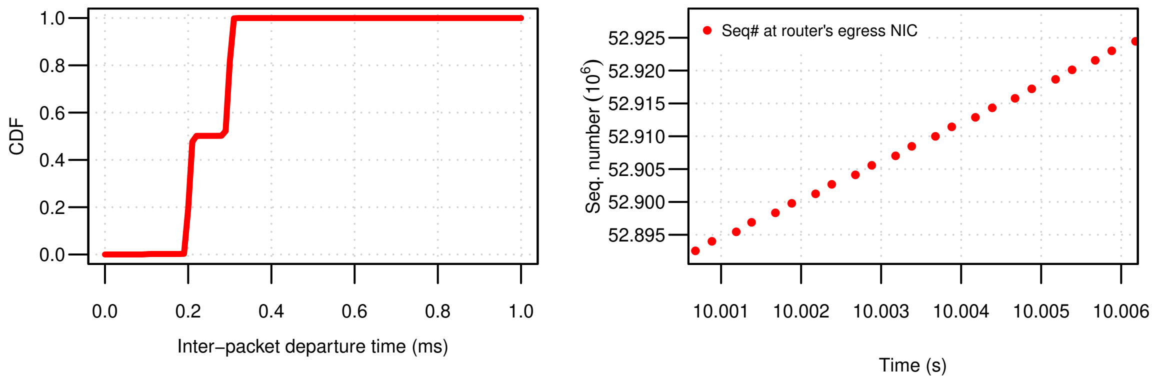
Figure B.5: Dummynet sends packets in bursts due to Dummynet inaccurate tick counter implementation. Data packets are captured at the router egress NIC - A single 60 second NewReno TCP flow, 48Mbps, 40ms RTT, and 200pkt bottleneck buffer size at 10KHz kernel tick rate

signed to work efficiently without adding large overhead on the operating system and hardware resources, at the cost of timing resolution.

The Dummynet FreeBSD implementation uses `getmicrouptime()` function [184] to implement a tick counter. This function is executed quickly but returns a less precise time than `microuptime()` function. In our testbed, we found that precision of `getmicrouptime()` function is around one millisecond regardless of kernel's tick rate. Therefore, in current Dummynet implementation, increasing the tick rate more than 1000 Hz (default) does not improve the timer precision. Thus, delay emulation error is not improved and packets burst5s at fast emulated links cannot be remedied by only increasing the tick rate.

We conduct an experiment to explore this issue. The experiment consists of a single 60 second NewReno TCP flow traversing a symmetric 48Mbps emulated link with 40ms emulated path's RTT and 200pkt bottleneck buffer size.

Figure B.5a shows CDF graph for inter-packet departure time of the data packets captured at the router egress NIC. We can see in this figure, about 65% of packets have inter-packet departure time ~12µs and 10% have inter-packet departure time ~900µs. This means 65% of packets has sent in bursts as the observed inter-packet departure time (~12µs) is smaller than the serialisation delay for a 1500 byte packet at 48Mbps (~250µs). Moreover, about 10% of the packets have ~100µs inter-packet departure time and 15% of the packets have ~1ms. Dummynet sends around 75% of packets too early and, therefore, around 25% of packets should be delayed to emulate the desired bandwidth. Figure B.5b illustrates how Dummynet transmits four packets in bursts due to inaccurate tick counter implementation.



(a) CDF for inter-packet departure time (b) SEQ number progression shows packets bursts

Figure B.6: Dummynet packets sending behaviour when `ticks` variable is used at 10KHz kernel's tick rate: data packets captured at router egress NIC - A single 60 second NewReno TCP flow, 48Mbps, 40ms RTT, and 200pkt bottleneck buffer size

The obvious solution to this issue is to use `microuptime()` function instead of `getmicrouptime()`. However, `microuptime()` causes system higher overhead by reading the time from hardware.

A better solution is to use the tick counter (`ticks`) provided by FreeBSD's kernel clock system. FreeBSD increments `ticks` variable automatically on every tick with no additional overhead. When system `ticks` counter is used, Dummynet does not require to calculate ticks.

We repeat the same experiment but using patched Dummynet (using `ticks` instead of `getmicrouptime`() function) and kernel's tick rate is set to 10KHz[5]. Figure B.6a shows CDF graph for inter-packet departure time of the data packets captured at the router egress NIC. We can see in this figure, around 50% of packets experience 200μs inter-packet departure time while the other 50% experience 300μs. This means that the patched Dummynet is able to provide more accurate link emulation. However, it is still unable to send a packet every 250μs. This is due to the timer resolution for 10KHz tick rate is 100μs, and therefore Dummynet is unable to emulate exact serialisation delay for one packet. To achieve 50μm time resolution, the tick rate should be set to 20KHz which produces better emulation but at the expense of additional interrupt overhead. In Figure B.6b, we can see packets are sent regularly without bursts which confirms the functionality of the proposed fix.

B.4 Conclusions

In this appendix, we presented Dummynet link emulation challenges and enhancements. Dummynet leads to unintended ACK compression due to its traffic shaper and relativity low kernel

[5]By setting kernel tunable kern.hz=10000 in loader.conf file.

tick rate. We also showed that this problem could lead to excess delay emulation, resulting in delaying packets by two ticks over the configured delay. We solved these issues by allowing some packets to transmit slightly earlier than the traffic shaper allows.

Moreover, we discussed the implication of the kernel tick rate on Dummynet emulation quality. We showed that Dummynet sends packets in bursts when the serialisation delay for the desired bandwidth is smaller than one tick. We also explored Dummynet tick calculation issue at kernel tick rates higher than 1KHz, and we presented our Dummynet timing enhancement to solve this issue.

Our Dummynet improvements allow us to conduct experiments under emulated networks that mimic real network behaviours with more accurate link emulation.

List of Publications

A number of technical reports and peer-reviewed papers have been published during PhD candidature.

Peer-reviewed papers:

- R. Al-Saadi, G. Armitage, J. But, and P. Branch, "A Survey of Delay-Based and Hybrid TCP Congestion Control Algorithms," IEEE Communications Surveys Tutorials, pp. 1–1, 2019.
- R. Al-Saadi, G. Armitage, and J. But, "Characterising LEDBAT Performance Through Bottlenecks Using PIE, FQ-CoDel and FQ-PIE Active Queue Management," in 2017 IEEE 42nd Conference on Local Computer Networks (LCN), Oct 2017, pp. 278–285.

Technical reports:

- R. Al-Saadi, G. Armitage, and J. But, "ttprobe v0.1: Packet-Driven TCP Stack Statistics Gathering for TEACUP," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 150911A, 11 September 2015.
  Available: http://caia.swin.edu.au/reports/150911A/ CAIA-TR-150911A.pdf
- R. Al-Saadi and G. Armitage, "Dummynet AQM v0.1 - CoDel and FQ-CoDel for FreeBSD's ipfw/dummynet framework," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160226A, 26 February 2016.
  Available: http://caia.swin.edu.au/reports/160226A/CAIA-TR-160226A.pdf
- R. Al-Saadi and G. Armitage, "Dummynet AQM v0.2 – CoDel, FQ-CoDel, PIE and FQ-PIE for FreeBSD's ipfw/dummynet framework," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160418A, 18 April 2016.
  Available: http://caia.swin.edu.au/reports/ 160418A/CAIA-TR-160418A.pdf
- J. Kua, R. Al-Saadi, and G. Armitage, "Using Dummynet AQM - FreeBSD's CoDel, PIE, FQ-CoDel and FQ-PIE with TEACUP v1.0 testbed," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160708A, 08 July 2016.
  Available: http: //caia.swin.edu.au/reports/160708A/CAIA-TR-160708A.pdf